

Las variables y las constantes

1. Creación y asignación

La creación de una variable en PowerShell es realmente fácil. Al igual que los lenguajes con objetos, PowerShell no dispone de un lenguaje tipo, es decir que las variables no necesitan ser definidas antes de ser utilizadas. De este modo, será suficiente asignar mediante el operador "=", un valor a su variable y PowerShell se encarga del resto, es decir, la crea y determina su tipo. La sintaxis utilizada es la siguiente:

```
$variable = valor de un tipo cualquiera
```

Del mismo modo, para leer una variable, bastará teclear simplemente el nombre de la variable en la consola.

- Tecleando `$var_1 = 12` en la consola PowerShell crearemos una variable denominada «var_1», de tipo «int» (entero) y le asignaremos el valor 12.
- Tecleando `$var_2 = 'A'` realizaremos la misma operación con la excepción que esta vez la variable es de tipo «string» (cadena) incluso conteniendo un único carácter y que el valor asociado es la letra A.



Puede recuperar el tipo de su variable aplicando el método `GetType`.

Ejemplo:

```
PS > $var_1 = 12
PS > $var_1.GetType()
```

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



Si utiliza nombres de variables con caracteres especiales (& @ % - £ \$. , etc.) es indispensable utilizar los caracteres «{» y «}».

Ejemplo:

```
${www.powershell-scripting.com} = 1
```

Aquí asignará el valor 1 a la variable entre paréntesis.

a. Conversión de variables

Sin embargo, puede ser interesante por diversas razones seguir controlando la tipificación de variables. Los incondicionales pueden tranquilizarse porque existe una alternativa a la tipificación automática. Para hacerlo, debemos definir el tipo deseado entre corchetes antes de la creación de la variable, como por ejemplo:

```
PS > [int]$var=12
```

Escribiendo la línea anterior se asegurará de que la variable `$var` es de tipo entero. Como objeción podríamos decir que no hay ninguna diferencia entre `$var = 12` y `[int]$var = 12`. Eso es cierto, ya que por el momento el interés es mínimo, pero cuando sea un gran « powersheller » y sus scripts empiecen a tomar cierta magnitud, el hecho de declarar sus variables con un tipo asociado hará que su script sea mucho más comprensible para los demás, y permitirá

sobre todo evitar que le sea asignado un valor de un tipo diferente.

Por ejemplo:

```
PS > [int]$numero = read-host 'Introduzca un número'
Introduzca un número: cien
Imposible convertir el valor «cien» a tipo «System.Int32».
Error: «El formato de la cadena de entrada es incorrecto.»
```

En el ejemplo superior, el texto introducido («cien») no ha sido reconocido como un número entero y por tanto, el intérprete de comandos lo rechaza. Si no hubiéramos precisado el tipo [int] delante de la variable \$numero, el texto hubiera sido aceptado y su tratamiento futuro habría podido plantear problemas.

Si ahora probamos de asignar un valor entero en uno tipo «char»:

```
PS > [char]$var=65
```

¿Que sucederá? PowerShell sencillamente convertirá el valor entero en un carácter, y no en cualquier carácter, sino en el carácter al que corresponda ese valor entero en código ASCII. En nuestro ejemplo \$var contendrá «A» ya que el carácter «A» corresponde a 65 en código ASCII.

Finalmente, probaremos de realizar la operación inversa, es decir pasar del tipo «string» al tipo «int». Lamentablemente no es posible hacerlo directamente:

```
PS > [int]$var = 'A'
Imposible convertir el valor «A» a tipo «System.Int32».
Error: «El formato de la cadena de entrada es incorrecto.»
A nivel de línea: 1 Character: 8+ [int]$a <= 'A'
```

No obstante es posible convertir una variable de tipo «char» a tipo «int»:

```
PS > [int][char]$var = 'A'
PS > $var
65
```



El hecho de poder convertir únicamente variables de tipo «char» proviene del hecho de sólo se puede hacer corresponder un carácter a un código ASCII, y no toda una cadena.

Veamos ahora que pasa si asignamos un valor decimal de tipo «doble» a una variable de tipo «int»:

```
PS> $var1=10.5
PS> $var1
10,5
PS> $var2=[int]$var1
PS> $var2
10
```

Lógicamente, la variable \$var2 se redondea a la parte entera más próxima, ya que una variable de tipo entero acepta únicamente los enteros en un rango comprendido entre -2 147 483 648 y 2 147 483 647 inclusive.

Intentaremos convertir un valor mucho mayor que el rango cubierto por los enteros, veamos qué pasa:

```
PS> $var1=1e27
PS >1E+27
PS > $var2=[int]$var1
Imposible convertir el valor «1E+27» a tipo «System.Int32».
Error: «El valor es demasiado grande o demasiado pequeño para un Int32.»
```

PowerShell especificará un error para decirnos que no ha tenido éxito al convertir un valor tan grande a una variable de tipo entero.

Naturalmente la asignación de variables no se limita al sistema decimal, también podemos convertir los valores decimales en hexadecimales y almacenarlos en una variable. Para realizar este tipo de operación, PowerShell se basa en los formatos de visualización de las cadenas de caracteres (operador -f) de Framework .NET. Como no hemos visto todavía ni las cadenas de caracteres, ni los métodos del Framework .NET, vamos a ver únicamente los comandos que nos permitirán la conversión.

Ejemplo:

Conversión de un número decimal en hexadecimal:

```
PS > $dec = 1234
PS > $hex = "{0:X}" -f $dec
PS > $hex
4D2
```

Atención ya que la utilización del formato de visualización de cadenas cambia el tipo de la variable \$hex a tipo «cadena de caracteres» (string). Para verificarlo, teclee \$hex.GetType()

Siguiendo el mismo principio, podemos convertir cualquier número decimal de nuestra elección en un número en la base deseada. Para esto basta con utilizar el comando siguiente: [System.Convert]::ToString(<valor_1>,<valor_2>) donde valor_1 corresponde al número (en base 10) a convertir y valor_2 la nueva base del número.

Ejemplo:

Conversión de un número decimal en base 8.

```
PS > $Nb = [System.Convert]::ToString(1234,8)
PS > $Nb
2322
```


Ejemplo:

Conversión de un número decimal en base 2.

```
PS > $Nb = [System.Convert]::ToString(1234,2)
PS > $Nb
10011010010
```

2. Las variables predefinidas

PowerShell dispone de un cierto número de variables automáticas que es bueno conocer. Veamos una lista no exhaustiva:

 Las variables con un asterisco sólo están disponibles en la versión 2 de PowerShell.

Variable	Descripción
\$\$	Variable que contiene el contenido del último miembro de la última línea recibida por el entorno (es decir, la última palabra del último comando)

	tecleado en la consola).
<code>\$?</code>	Variable que contiene <i>true</i> si la última operación es correcta o <i>false</i> en caso contrario.
<code>\$^</code>	Variable que contiene el primer miembro de la última línea recibida por el entorno (es decir, la primera palabra del último comando tecleado en la consola).
<code>\$_</code>	Variable que contiene el objeto actual transmitido por el pipe « », el pipe se tratará posteriormente en este capítulo.
<code>\$Args</code>	Variable que contiene una tabla de argumentos pasados a una función o a un script.
<code>\$ConfirmPreference</code>	Variable que permite determinar qué commandlets pedirán automáticamente la confirmación del usuario antes de su ejecución. Cuando el valor de <code>\$ConfirmPreference</code> (High, Medium, Low, None [Alto, Medio, Bajo, Ninguno]) es superior o igual al riesgo de la acción del applet del comando (High, Medium, Low, None), Windows PowerShell pedirá automáticamente la confirmación del usuario antes de ejecutar la acción.
<code>\$ConsoleFileName</code>	Variable que contiene la ruta de acceso al archivo consola (.pscl) que se ha utilizado en último lugar en la sesión.
<code>\$DebugPreference</code>	Variable que contiene un valor específico correspondiente a una acción preferente a establecer. Se utiliza con el comando Write-Debug (véase el Capítulo Gestión de los errores y depuración - La depuración - Visualización de mensajes en modo debug).
<code>\$Error</code>	Variable en forma de tabla que contiene el registro de los errores mostrados durante la sesión (véase el Capítulo Gestión de los errores y depuración - Los errores no críticos - El tipo ErrorRecord).
<code>\$ErrorActionPreference</code>	Variable que contiene un valor específico correspondiente a una acción preferente a establecer en caso de error. Se utiliza con el comando Write-Error (véase el Capítulo Gestión de los errores y depuración).
<code>\$ErrorView</code>	Variable que determina el formato de visualización de los mensajes de error en Windows PowerShell (véase el Capítulo Gestión de los errores y depuración).
<code>\$ExecutionContext</code>	Variable que contiene un objeto EngineIntrinsics que representa el contexto de ejecución del servidor Windows PowerShell.
<code>\$False</code>	Variable que contiene el valor <i>false</i> . Esta variable es una constante, y en consecuencia no se puede modificar.
<code>\$Foreach</code>	Variable que hace referencia al enumerador de un bucle Foreach.
<code>\$FormatEnumerationLimit</code>	Variable que determina el número de elementos enumerados incluidos en una vista.
<code>\$Home</code>	Variable que contiene la ruta (<i>path</i>) del directorio de inicio del usuario.
<code>\$Host</code>	Variable que contiene la información del servidor.

\$Input	Variable que enumera los objetos transmitidos por la tubería.
\$LastExitCode	Variable que contiene el código de salida de la última ejecución de un archivo ejecutable Win32.
\$MaximumAliasCount	Variable que contiene el número máximo de alias posibles en la sesión.
\$MaximumDriveCount	Variable que contiene el número máximo de lectores posibles en la sesión (los proporcionados por el sistema no se tendrán en cuenta).
\$MaximumErrorCount	Variable que contiene el número máximo de errores registrados en el histórico de errores de la sesión.
\$MaximumFunctionCount	Variable que contiene el número máximo de funciones posibles en la sesión.
\$MaximumHistoryCount	Variable que contiene el número máximo de comandos que pueden registrarse en el histórico.
\$MaximumVariableCount	Variable que contiene el número máximo de variables posibles en la sesión.
\$MyInvocation	Variable que contiene un objeto relativo a la información sobre el comando en curso.
\$NestedPromptLevel	Variable que indica el nivel del prompt actual. El valor 0 indica el nivel de origen. El valor se incrementa cuando se accede a un nivel interrelacionado y disminuye cuando se sale.
\$Null	Variable vacía.
\$OFS	Variable que contiene el separador de campo en la conversión de una tabla en cadena.
\$OutputEncoding	Variable que contiene el método de codificación de caracteres utilizado por Windows PowerShell cuando se envía el texto a otras aplicaciones.
\$PID	Variable que contiene el número ID del proceso PowerShell.
\$Profile	Variable que contiene la ruta (<i>path</i>) del perfil Windows PowerShell.
*\$ProgressReference	Variable que determina la forma en que Windows PowerShell responde a las actualizaciones de progresión generadas por un script, un commandlet o un proveedor.
*\$PSBoundParameters	Variable que contiene un diccionario actualizado de parámetros y valores en curso.
*\$PSCulture	Variable que contiene el nombre de la cultura utilizada actualmente en el sistema operativo (es-ES para el idioma español).
*\$PSEmailServer	Variable que contiene el servidor de mensajería a utilizar por defecto con el commandlet Send-MailMessage.
\$PsHome	Variable que contiene la ruta (<i>path</i>) donde está instalado PowerShell.
*\$PSSessionApplicationName	Variable que contiene el nombre de la aplicación utilizada para utilizar comandos remotos. La aplicación del sistema por defecto es WSMAN.

*\$PSSessionConfigurationName	Variable que contiene la URL de la configuración de la sesión utilizada por defecto.
*\$PSSessionOption	Variable que contiene los valores por defecto en una sesión a distancia.
*\$PSUICulture	Variable que contiene el nombre de la cultura de la interface de usuario (IU) que se utiliza actualmente.
*\$PSVersionTable	Variable que contiene una tabla en modo lectura que muestra los detalles relativos a la versión de Windows PowerShell.
\$PWD	Variable que indica la ruta completa del directorio activo.
\$ReportErrorShowExceptionClass	Variable que muestra los nombres de las clases de las excepciones visualizadas.
\$ReportErrorShowInnerException	Variable que muestra (cuando su valor es <i>true</i>) la cadena de excepciones internas.
\$ReportErrorShowSource	Variable que muestra (cuando su valor es <i>true</i>) los assembly names (véase el Capítulo .NET - Utilizar objetos .NET con PowerShell - Los Ensamblados) de las excepciones visualizadas.
\$ReportErrorShowStackTrace	Variable que emite (cuando su valor es <i>true</i>) las estructuras de las llamadas de procedimientos de excepciones.
\$ShellID	Variable que indica el identificador del Shell.
\$ShouldProcessPreference	Especifica la acción a realizar cuando ShouldProcess se utiliza en un commandlet.
\$ShouldProcessReturnPreference	Variable que contiene el valor devuelto por ShouldPolicy.
\$StackTrace	Variable que contiene la información de la estructura de las llamadas a procedimientos detallados relativas al último error.
\$True	Variable que contiene el valor <i>true</i> .
\$VerbosePreference	Variable que contiene un valor específico que corresponde a una acción preferente a establecer. Se utiliza con el comando Write-Verbose (véase el Capítulo Gestión de los errores y depuración - La depuración - Visualización de mensajes en modo verbose).
\$WarningPreference	Variable que contiene un valor específico correspondiente a una acción preferente a establecer. Se utiliza con el comando Write-Warning (véase el Capítulo Gestión de los errores y depuración - La depuración - Visualización de mensajes en modo warning).
\$WhatIfPreference	Variable que determina si el parámetro WhatIf está activado automáticamente para cada comando que lo soporta.

3. Los diferentes operadores

Existen numerosos tipos de operadores, tanto si son de tipo aritmético, binario, lógico u otros, le permitirán actuar sobre

las variables. Tenga en cuenta que conocer y controlar las distintas operaciones es esencial para la elaboración de un buen script.

a. Los operadores aritméticos

Respecto a los operadores aritméticos, no hay gran complicación. PowerShell trata las expresiones de izquierda a derecha respetando las reglas de las propiedades matemáticas así como los paréntesis.

Ejemplo:

```
PS > 2+4*3
14
PS > (2+4)*3
18
```

La lista de los operadores aritméticos disponibles es la siguiente:

Signo	Significado
+	Adición
-	Sustracción
*	Multiplicación
/	División
%	Módulo

Los primeros cuatro operadores deben lógicamente parecerle familiares, en cuanto al último, el operador módulo, permite reenviar el resto de una división entera de a por b.

Por ejemplo: tecleando `5%3` en la consola, obtendremos 2. Simplemente porque hay 1 vez 3 en 5 y el resto de la división vale 2.

Podrá observar que los operadores aritméticos se aplican igualmente a variables. De este modo, `$var_1 + $var_2` será la suma de las dos variables si contienen valores numéricos.

Ejemplo del operador "+" sobre dos enteros:

```
PS > $int1 = 10
PS > $int2 = 13
PS > $int2 + $int1
23
```

El operador adición se emplea de igual forma con cadenas (variables de tipo string). En este caso, el operador sirve para concatenar las dos cadenas:

```
PS > $cadena1 = 'A'
PS > $cadena2 = 'B'
PS > $cadena1 + $cadena2
AB
```

Siempre con las cadenas de caracteres, tenga en cuenta que es posible repetir el contenido de una cadena gracias al operador multiplicación:

```
PS > $cadenal = 10 * 'A'
PS > $cadenal
AAAAAAAAAA
```



Veremos otros operadores matemáticos como el cálculo de un seno, coseno, raíz cuadrada, etc. vía la clase `System.Math` disponible en el Framework .NET (véase el capítulo .NET a cerca de la utilización de Framework y de los tipos .NET).

b. Los operadores de comparación

Con un nombre tan evocador, no es necesario precisar que los operadores de comparación nos van a permitir hacer comparaciones de variables. En efecto, además de la utilización de las estructuras condicionales que veremos más tarde en este capítulo, utilizamos estos famosos operadores para obtener un resultado de tipo booleano, es decir Verdadero (True) o Falso (False), en una comparación determinada. Para conocer las diferentes comparaciones, echemos un vistazo a todas las operaciones de comparación.

Operador	Significado
-eq	Igual
-ne	No igual (diferente)
-gt	Estrictamente superior
-ge	Superior o igual
-lt	Estrictamente inferior
-le	Inferior o igual

A señalar que los operadores de comparación no respetan las mayúsculas y minúsculas en una comparación de cadena. Para remediar esto simplemente hay que hacer preceder el nombre del operador con la letra «c», como por ejemplo -cle.



Para que el operador no respete las mayúsculas y minúsculas hay que hacer preceder el nombre del operador con la letra «i» como por ejemplo -ile. Pero esto no será necesario ya que los operadores de comparación no respetan las mayúsculas y minúsculas por defecto.

c. Los operadores de comparación genéricos

Una expresión genérica es una expresión que contiene un carácter denominado «genérico». Por ejemplo «*» para indicar cualquier conjunto de caracteres, o un «?» para un único carácter. Existen dos operadores de comparación que le permitirán comparar una cadena con una expresión genérica.

Operador	Significado
-like	Comparación de igualdad de expresión genérica.
-notlike	Comparación de desigualdad de expresión genérica.

Para comprender mejor la utilización de estos operadores, veamos algunos ejemplos de su aplicación:


```

PS > 'Powershell' -like '*shell'
True

PS > 'powershell' -like 'power*'
True

PS > 'powershell' -like '*wer*'
True

PS > 'powershell' -like '*war*'
False

PS > 'powershell' -like 'po?er*'
True

PS > 'power' -like 'po?er*'
True

PS > 'potter' -like 'po?er*'
False

```



El Operador de comparación genérico puede (como los operadores de comparación) o no respetar las mayúsculas o minúsculas. Si desea que el operador respete las mayúsculas y minúsculas, debe hacer preceder el nombre del operador con la letra «c». Para hacer lo contrario, debe hacer preceder el nombre con la letra «i».

d. Los operadores de comparación de las expresiones regulares

Una expresión regular llamada igualmente «RegEx» es una expresión compuesta por los denominados «metacaracteres», que corresponden a valores específicos de caracteres.

Si nunca ha oído hablar de las expresiones regulares, le aconsejamos dar una ojeada a los numerosos libros que tratan de este tema o bien consultar la ayuda online ([Help about_Regular_Expression](#)) que está muy bien explicada.

PowerShell dispone de dos operadores de comparación de expresiones regulares, que nos retornarán un booleano en función del resultado obtenido en la comparación.

Operador	Significado
-match	Comparación de igualdad entre una expresión y una expresión regular.
-notmatch	Comparación de desigualdad entre una expresión y una expresión regular.

Para comprender mejor la utilización de estos operadores, veamos algunos ejemplos de su aplicación:

```

PS > 'Powershell' -match 'power[sol]hell'
True

PS > 'powershell' -match 'powershel[a-k]'
False

PS > 'powershell' -match 'powershel[a-z]'
True

```



El Operador de comparación de expresión regular puede, como los operadores de comparación, respetar o no



las mayúsculas y minúsculas. Para que el operador respete las mayúsculas y minúsculas, debe hacer preceder el nombre del operador con la letra «c», para hacer lo contrario, debe hacer preceder el nombre con la letra «i».

e. Los operadores de intervalo

El operador de intervalo se indica: «..» (pronunciado, dos puntos). Permite, como su nombre indica, cubrir un intervalo de valores sin necesidad de introducirlos. Supongamos que deseamos cubrir un intervalo de valores que van de 1 a 10 (para realizar un bucle por ejemplo); será suficiente con teclear la línea siguiente:

```
PS > 1..10
```

Se puede, del mismo modo, definir un intervalo dinámico utilizando variables. Nada le impide definir un intervalo de \$var1 a \$var2 si estos valores son enteros.

Ejemplo:

```
PS > $var1 = 5
PS > $var2 = 10
PS > $var1 .. $var2
5
6
7
8
9
10
```

f. El operador de sustitución

El operador de sustitución permite reemplazar todo o parte de un valor por otro. Supongamos que nuestra variable es de tipo cadena, que su contenido es «PowerShell», y que queremos reemplazar «Shell» por «Guy».

Será necesario utilizar el operador de sustitución (-replace) seguido de la parte a reemplazar y del nuevo valor.

Ejemplo:

```
PS > 'PowerShell' -replace 'Shell', 'Guy'
PowerGuy
```



El operador de sustitución puede (como los operadores de comparación) o no respetar las mayúsculas y minúsculas. Para que el operador respete las mayúsculas y minúsculas, debe hacer preceder el nombre del operador con la letra «c», para hacer lo contrario, debe hacer preceder el nombre con la letra «i».



Las cadenas de caracteres (string) poseen un método llamado Replace con la misma funcionalidad.

Ejemplo:

```
PS > $MiCadena = 'PowerShell'
PS > $MiCadena.Replace('Shell', 'Guy')
PowerGuy
```

g. Los operadores de tipo

Hasta ahora, le hemos mostrado cómo asociar un tipo a su valor y cómo recuperar el tipo con el método `GetType`. Pero lo que estamos ahora por descubrir, es cómo evaluar el tipo de una variable. Por ejemplo, podríamos estar interesados en saber si una variable es de tipo "int" para poder atribuirle un valor entero. Todo esto se efectúa con los dos operadores de tipo siguientes:

Operador	Significado
-is	Verifica si el objeto es del mismo tipo
-isnot	Verifica si el objeto no es del mismo tipo

Para comprender mejor la utilización de estos operadores, veamos algunos ejemplos de su aplicación:

```
PS > 'Buenos días' -is [string]
True

PS > 20 -is [int]
True

PS > 'B' -is [int]
False
```

h. Los operadores lógicos

Los operadores lógicos permiten verificar hasta múltiples comparaciones en una misma expresión. Por ejemplo: `($var1 -eq $var2) -and ($var3 -eq $var4)`, le devolverá un booleano *true* si `$var1` es igual a `$var2` y `$var3` es igual a `$var4`, en caso contrario el valor *false* se reenviaría. Veamos la lista de operadores lógicos disponibles:

Operador	Significado
-and	Y lógico
-or	O lógico
-not	No lógico
!	No lógico
-xor	O exclusivo

Para entender mejor la utilización de estos operadores, veamos algunos ejemplos de su aplicación:

```
PS > (5 -eq 5) -and (8 -eq 9)
False
```

Falso, ya que 5 es igual a 5, pero 8 no es igual a 9.

```
PS > (5 -eq 5) -or (8 -eq 9)
True
```

Verdadero, debido a que una de las expresiones es verdadera, 5 es igual a 5.

```
PS > -not (8 -eq 9)
True
```

```
PS > !(8 -eq 9)
True
```

Verdadero, ya que 8 no es igual a 9.

i. Los operadores binarios

Los operadores binarios se utilizan para efectuar operaciones entre números binarios. Como recordatorio, el sistema binario es un sistema en base 2, a diferencia del sistema decimal que es en base 10. Es decir que únicamente consta de combinaciones de «0» y «1».

Ejemplo de conversión de números decimales en base binaria:

Decimal	Binaria
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101

Cuando hacemos una llamada a uno de los operadores binarios siguientes, los bits de los valores se comparan uno tras otro, y luego según apliquemos un Y o un O obtendremos un resultado diferente.

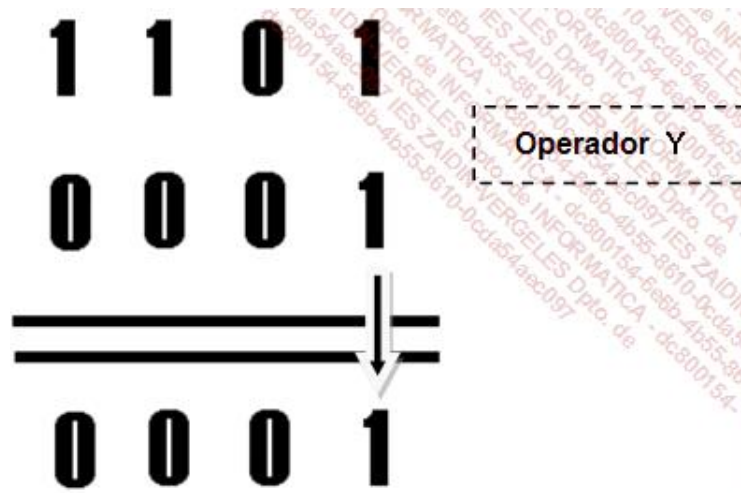
Operador	Significado
-band	Operador Y
-bor	Operador O
-bnot	Operador NO
-bxor	Operador O Exclusivo



El resultado devuelto después de una comparación binaria se convierte automáticamente en sistema decimal y no en sistema binario.

Supongamos que para una aplicación cualquiera queremos saber si el bit menos significativo (LSB) de una variable es igual a 1. Tomemos como ejemplo el valor decimal 13, es decir, 1101 en formato binario. Evidentemente se ve entonces claramente que el bit menos significativo es 1, pero para verificar esta afirmación vía PowerShell, utilizamos nuestro operador binario *-band*.

Utilizando este operador, vamos a realizar de hecho lo que se denomina una máscara sobre el bit menos significativo. Si el resultado es igual a la máscara aplicada entonces el bit menos significativo tiene el valor 1. Aquí tiene lo que daría gráficamente la comparación:



Máscara sobre el bit menos significativo

Resultado:

```
PS > $var = 13
PS > $var -band 1
1
```

➤ PowerShell 1.0 utiliza operadores de bits que trabajan sobre los enteros de 32 bits (valores comprendidos entre -2.147.483.648 y 2.147.483.647). La versión 2.0 por su parte, permite trabajar sobre 64 bits (que abarca los valores comprendidos entre -9223372036854775807 y 9223372036854775807).

j. Los operadores de asignación

Usted sabe ahora cómo asignar un valor a una variable y realizar una operación sobre esta última. Ahora vamos a mostrarle cómo hacer ambas al mismo tiempo en una única operación.

Las operaciones que se describen en este cuadro dan exactamente el mismo resultado.

Notación clásica	Notación abreviada
$\$i = \$i + 8$	$\$i += 8$
$\$i = \$i - 8$	$\$i -= 8$
$\$i = \$i * 8$	$\$i *= 8$
$\$i = \$i / 8$	$\$i /= 8$
$\$i = \$i \% 8$	$\$i \% = 8$
$\$i = \$i + 1$	$\$i ++$
$\$i = \$i - 1$	$\$i --$

➤ La notación $\$i ++$ o $\$i --$ es muy utilizada en las condiciones de bucle para incrementar $\$i$ en 1 en cada paso.

De este modo, por ejemplo, veamos como añadir un valor con el operador de afectación «+=».


```
PS > $i = 0
PS > $i += 15
PS > $i
15
```

Si se muestra el valor de la variable `$i` se obtiene 15, ya que esta instrucción es equivalente a: `$i = $i + 15`.

Continuamos con el cálculo de los factoriales de las cifras de 1 a 10.

Para ello, vamos a crear un bucle y para cada valor `$i`, lo multiplicaremos por el valor de `$var` antes de reasignárselo:

```
PS > $var = 1
PS > foreach($i in 1..10){$var *= $i ; $var}
1
2
6
24
120
720
5040
40320
362880
3628800
```

 Como todavía no hemos abordado la noción de bucle `foreach`, no es necesario prestar mayor atención a este ejemplo. Lo esencial es que haya comprendido que se trata de un bucle, comprendido entre 1 y 10, en el que para cada valor de `i`, se multiplica el valor de `$i` por el valor de `$var` y se registra el total en `$var`.

k. Los operadores de redirección

Lo que hay que saber es que los intérpretes de comandos tratan la información según una entrada, una salida y un error estándar, cada elemento está identificado por un descriptor de archivo. A la entrada estándar, se le asigna el descriptor 0, a la salida estándar el 1 y al error estándar el 2. Por defecto, se utiliza el teclado como entrada estándar, y la visualización en la consola es para la salida. Pero para redirigir estos flujos de información con más flexibilidad, PowerShell dispone de un conjunto de operadores, idénticos a los utilizados en la interfaz de línea de comandos de Unix:

Operador	Significado
>	Redirige el flujo hacia un archivo, si el archivo ya está creado, el contenido del archivo anterior se reemplaza.
>>	Redirige el flujo hacia un archivo, si el archivo ya está creado, el flujo se añade al final del archivo.
2>&1	Redirige los mensajes de error hacia la salida estándar.
2>	Redirige el error estándar hacia un archivo, si el archivo ya está creado, el contenido del archivo anterior se reemplaza.
2>>	Redirige el error estándar hacia un archivo, si el archivo ya está creado, el flujo se añade al final del archivo.

Supongamos que queremos enviar el resultado de un comando hacia un archivo de texto en lugar de a la consola, para ello utilizamos el operador `>>`:

```
PS > Get-Process > c:\temp\process.txt
```

Obtenemos un archivo de texto en c:\temp nombre del proceso.txt que contiene el resultado del comando.

```
PS > Get-Content c:\temp\process.txt
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
66	4	1768	3520	57	0,14	4200	acrotray
62	2	940	2924	31		1424	audiodg
297	76	4832	452	72	0,70	2096	ccApp
851	14	13008	7532	105		548	CcmExec
497	11	9528	5320	87		1800	ccSvcHst
34	2	796	3464	36	0,05	5152	conime
582	5	1712	3124	87		768	csrss
301	10	2732	12728	135		4784	csrss
202	6	2256	5720	70		540	DefWatch
82	3	1388	4436	45	0,09	2636	dwm
678	27	27960	41488	189	20,42	368	explorer
0	0	0	16	0		0	Idle

Ahora, si escribimos de nuevo un comando cuya salida estuviese dirigida en el mismo archivo vía el operador «>», los datos se machacarían. El contenido del archivo se borrará y se sustituirá por la nueva salida. Para evitar esto, es necesario utilizar el operador «>>» que indica a PowerShell que tiene que añadir la salida de la orden al final del fichero especificado.



Para redirigir un flujo hacia un archivo, también puede utilizar el commandlet `Out-File` en lugar de los operadores de redirección, sobre todo si desea utilizar parámetros tales como la codificación, el número de caracteres en cada línea de salida, etc. Pero todo esto se explicará en detalle en el Capítulo Control del Shell.

Último ejemplo, la redirección del error estándar hacia un archivo. Para ello utilizamos simplemente un comando susceptible de devolver un mensaje de error, como `Get-ChildItem` en un directorio inexistente. Después enviamos la totalidad a un archivo vía el operador «2>».

```
PS > Get-ChildItem c:\temp\DirInexistente 2> c:\err.txt
```

No se muestra ningún mensaje en la consola. Pero al recuperar el contenido del archivo err.txt, se comprueba que contiene el mensaje de error relativo al comando introducido.

```
PS > Get-Content c:\err.txt
```

```
Get-ChildItem: Imposible encontrar la ruta de acceso  
«C:\temp\DirInexistente», ya que no existe.
```

I. Operadores de fraccionamiento y de concatenación

Los operadores de fraccionamiento y concatenación están disponibles únicamente en la versión 2.0 de PowerShell. Permiten combinar o dividir a voluntad las cadenas de caracteres.

Operador	Significado
-split	Fracciona una cadena en sub-cadenas.
-join	Concatena varias cadenas en una sola.

Por ejemplo, veamos como fraccionar una cadena colocando el operador `-split` al inicio de línea.

```
PS > -split 'PowerShell es fácil'
PowerShell
es
fácil
```

Por defecto, en el fraccionamiento se utiliza como delimitador el espacio en blanco. Para cambiar este delimitador, conviene situar el operador al final de línea seguido del carácter delimitador deseado.

Ejemplo:

```
PS > 'Nombre:Apellido:Dirección:Fecha' -split ':'
Nombre
Apellido
Dirección
Fecha
```

El operador `-join` permite realizar la concatenación de diferentes cadenas de caracteres de una misma tabla.

Ejemplo:

```
PS > $tabla = 'Lunes', 'Martes', 'Miércoles', 'Jueves', 'Viernes',
'Sábado', 'Domingo'
PS > -join $tabla
LunesMartesMiércolesJuevesViernesSábadoDomingo
PS > $tabla -join ', luego '
Lunes, luego Martes, luego Miércoles, luego Jueves, luego Viernes,
luego Sábado, luego Domingo
```

m. Recapitulación de los operadores

En esta lista encontrará todos los operadores ya enunciados a lo largo de este Capítulo (los operadores marcados con un asterisco sólo están disponibles con PowerShell v2).

Operador	Significado
<code>-eq</code>	Igual.
<code>-lt</code>	Inferior a.
<code>-gt</code>	Superior a.
<code>-le</code>	Inferior o igual a.
<code>-ge</code>	Superior o igual a.
<code>-ne</code>	Diferente de.
<code>-not</code>	No lógico.
<code>!</code>	No lógico.
<code>-match</code>	Comparación de igualdad entre una expresión y una expresión regular.

-notmatch	Comparación de desigualdad entre una expresión y una expresión regular.
-like	Comparación de igualdad de expresión genérica.
-notlike	Comparación de desigualdad de expresión genérica.
-replace	Operador de reemplazo.
-and	Y lógico.
-or	O lógico.
-bor	Operador de bits O.
-band	Operador de bits Y.
-bxor	Operador de bits O EXCLUSIVO.
-xor	O EXCLUSIVO.
-is	Operador de igualdad de tipo.
-isnot	Operador de desigualdad de tipo.
-ceq	Igual (respecto a las mayúsculas y minúsculas).
-clt	Inferior a (respecto a las mayúsculas y minúsculas).
-cgt	Superior a (respecto a las mayúsculas y minúsculas).
-cle	Inferior o igual a (respecto a las mayúsculas y minúsculas).
-cge	Superior o igual a (respecto a las mayúsculas y minúsculas).
-cne	Diferente de (respecto a las mayúsculas y minúsculas).
-cmatch	Comparación de igualdad entre una expresión y una expresión regular (respecto a las mayúsculas y minúsculas).
-cnotmatch	Comparación de desigualdad entre una expresión y una expresión regular (respecto a las mayúsculas y minúsculas).
-clike	Comparación de igualdad de expresión genérica (respecto a las mayúsculas y minúsculas).
-cnotlike	Comparación de desigualdad de expresión genérica (respecto a las mayúsculas y minúsculas).
-creplace	Operador de reemplazo (respecto a las mayúsculas y minúsculas).
>	Dirige el flujo hacia un archivo, si el archivo ya está creado, el contenido del archivo anterior se reemplaza.
>>	Dirige el flujo hacia un archivo, si el archivo ya está creado, el flujo se añade al final del archivo.

2>&1	Dirige los mensajes de error hacia la salida estándar.
2>	Dirige los mensajes de error hacia un archivo, si el archivo ya está creado, el contenido del archivo anterior se reemplaza.
2>>	Dirige el error estándar hacia un archivo, si el archivo está ya creado, el flujo se añade al final del archivo.
-split (*)	Fracciona una cadena en sub-cadenas.
-join (*)	Concatena varias cadenas en una sola.