

La depuración

PowerShell, en la parte de la depuración, está dotado de características suficientemente ricas en relación con su primo VBScript. Y es todavía más cierto con PowerShell v2 que integra el concepto de puntos de interrupción y de ejecución paso a paso de modo gráfico.

Hay muchas y diversas formas de depurar un script. La más básica es la de intercalar en un script pantallas de variables o mensajes aquí y allá para tratar de encontrar el o los errores. Dicho esto, con PowerShell veremos que podemos hacer bastante más que colocar `write-host` de depuración en todo nuestro script y así «contaminar» nuestro código.

Otro método básico de depuración consiste también en forzar la declaración de las variables; lo veremos más adelante.

1. Visualización de mensajes en modo verbose

Existe en PowerShell un modo verbose. Éste permite a los scripts o a los commandlets (y son numerosos) mostrarnos la información complementaria que estos poseen.

Para escribir una información visible sólo en «modo verbose», es necesario utilizar el commandlet `write-verbose`. Este sería el primer paso, pero no es suficiente, ya que para permitir que su información aparezca en la consola, será necesario ajustar el valor de la variable `$VerbosePreference` que por defecto está en modo `silentlyContinue`. Esto significa que, por defecto, la información adicional no se mostrará. Los otros valores posibles son los mismos que para la variable `$ErrorActionPreference`, es decir `Continue`, `Stop`, e `Inquire`.

Ejemplo:


Probemos de escribir una información en el modo por defecto.

```
PS > $VerbosePreference
SilentlyContinue
PS > Write-Verbose '¡Esto es una prueba!'
PS >
```

Como estaba previsto, no ocurre nada. Ahora vemos lo que ocurre poniendo el valor `continue` a la variable `$VerbosePreference`.

```
PS > $VerbosePreference = 'continue'
PS > Write-Verbose '¡Esto es una prueba!'
DETALLADO: ¡Esto es una prueba!
PS >
```

Estamos obligados a señalar que nuestra cadena de caracteres comienza por "DETALLADO:" y sobre todo que se muestra en amarillo, lo que la hace perfectamente visible en medio de una visualización estándar.

 Tenemos que decir que el amarillo no se ve muy bien en una impresión en blanco y negro. Es por esta razón por la que hemos puesto en negrita lo que usted vería normalmente en amarillo en pantalla.

En modo `stop`, el mensaje se visualiza pero la ejecución se detiene porque se ha producido una excepción; mientras que en el modo `Inquire` se nos propone el menú de opciones habituales.

2. Visualización del mensaje en modo debug

El modo debug para la visualización de mensajes funciona exactamente igual que el commandlet `write-verbose`, con las diferencias siguientes:

- La escritura de un mensaje de depuración se efectúa con `Write-Debug`.
- La variable de preferencia a ajustar es `$DebugPreference`.
- El mensaje visualizado comenzará por «DEPURACIÓN:».

Ejemplo:

```
PS > $DebugPreference = 'continue'
PS > Write-Debug 'Esto es una información de depuración.'
$DEPURACIÓN: Esto es una información de depuración.
PS >
PS > $DebugPreference = 'stop'
PS > Write-Debug 'Esto es una información de depuración.'
$DEPURACIÓN: Esto es una información de depuración.
Write-Debug: Se detuvo la ejecución del comando porque la variable de
preferencia "DebugPreference" o un parámetro común está establecido en
Stop.
En línea: 1 Carácter: 12
+ Write-Debug <<<< 'Esto es una información de depuración.'
+ CategoryInfo          : OperationStopped: (:) [Write-Debug],
ParentContainsErrorRecordException
+ FullyQualifiedErrorId : ActionPreferenceStop,Microsoft.PowerShell.Commands
.WriteDebugCommand
PS >
```

3. Visualización de mensajes en modo warning

El último modo de visualización posible para mensajes de estado es el **modo advertencia**. Funciona de igual forma que los dos modos anteriores (**verbose** y **debug**), con las diferencias:

- La escritura de un mensaje de advertencia se efectuará con `Write-Warning`.
- La variable de preferencia a ajustar es `$WarningPreference`.
- El mensaje visualizado comenzará por «ADVERTENCIA:».



En lugar de manipular las variables de preferencia que acabamos de ver, es posible, para cada commandlet, utilizar los parámetros comunes siguientes:

- **Verbose**: para ver la información adicional si hay.
- **Debug**: para mostrar un menú de opciones de tipo «Inquire» cuando se dé uno de los siguientes casos: visualización de un mensaje de depuración, información o advertencia, error no crítico.
- **Confirm**: pide al usuario una confirmación antes de efectuar un commandlet que modifica el estado del sistema.

4. Forzar la definición de las variables

Con PowerShell no está obligado a definir sus variables. Una simple asignación del valor es suficiente para declarar una

variable; PowerShell se encarga del resto.

¿Ha observado que una variable no definida es una **cadena vacía** para el caso de una cadena de caracteres, y **cero** para un entero? En principio PowerShell asigna el valor `$null` a las variables que no están definidas.

Veamos aquí un caso de error clásico donde cometemos un error en el nombre de una variable:

```
PS > $MiVariable = 25
PS > $Total = $miVariable * 12
```

Esto va a producir, sin duda, iun resultado imprevisible pero sobre todo imprevisto!

Para evitar esto, y forzar la declaración de todas las variables, PowerShell pone a nuestra disposición el commandlet `Set-PSDebug` seguido del parámetro `-strict`.



`Set-PSDebug -strict` corresponde en esencia a «**Option Explicit**» en VBScript pero menos restrictivo.

Retomemos nuestro ejemplo para ver el cambio:

```
PS > Set-PSDebug -Strict
PS > $MiVariable = 25
PS > $Total = $miVariable * 12
```

```
La variable 'miVariable' no se puede recuperar porque no se ha establecido.
En línea: 1 Carácter: 21
+ $Total = $miVariable <<<< * 12
+ CategoryInfo          : InvalidOperation: (miVariable:Token) [],
RuntimeException
+ FullyQualifiedErrorId : VariableIsUndefined
```

El mensaje es muy claro: no hemos definido `$miVariable`; lo que es normal dado que nos podemos haber confundido al teclear. Con una indicación tan clara, es difícil no localizar su error.

Con la versión 2, PowerShell va todavía más lejos en la definición de variables gracias al commandlet `set-strictmode`. Muy similar al funcionamiento de `set-PSDebug` seguido del parámetro `-strict`, `set-strictmode` no permite únicamente determinar los errores relativos a las variables no inicializadas, sino también los provocados por las propiedades que no existen. De hecho, `set-strictmode` puede utilizar los diferentes niveles de versión definidos a continuación:

Versión	Definiciones asociadas
Versión 1.0	<ul style="list-style-type: none">Prohíbe las referencias a las variables no inicializadas, con excepción de las presentes en las cadenas.
Versión 2.0	<ul style="list-style-type: none">Prohíbe las referencias a las variables no inicializadas (especialmente las variables no inicializadas presentes en las cadenas).Prohíbe las referencias a propiedades inexistentes de un objeto.Prohíbe las llamadas de funciones que utilizan la sintaxis para la llamada de métodos.Prohíbe una variable sin nombre (<code>\${}</code>).
Versión Latest	<ul style="list-style-type: none">Elige la versión más reciente (la más estricta) disponible. Utilizable en versiones futuras de PowerShell.

Por ejemplo, tomemos el caso típico de una propiedad que no existe. Incluso activando el modo `set-PSDebug -strict`, no se mostrará ningún error.

```
PS > Set-PSDebug -Strict
PS > $MiVariable = 25
PS > $MiVariable.essolounapropiedadquenoexiste
```

Ahora, si utilizamos el commandlet **set-strictmode** en su versión 2.0 (que prohíbe las referencias a propiedades que no existen), un error se mostrará esta vez.

```
PS > Set-StrictMode -Version 2.0
PS > $MiVariable = 25
PS > $MiVariable.essolounapropiedadquenoexiste
No se encuentra la propiedad 'essolounapropiedadquenoexiste' en este objeto.
Asegúrese de que existe.
En línea: 1 Carácter: 13
+ $MiVariable. <<<< essolounapropiedadquenoexiste
               ~~~~~
+ CategoryInfo          : InvalidOperation: (.:OperatorToken) [],
RuntimeException
+ FullyQualifiedErrorId : PropertyNotFoundStrict
```

El mensaje es nuevamente muy claro: no tenemos la propiedad citada, por lo tanto, el script se parará.

¡He aquí una buena costumbre a adoptar para ganar tiempo en el desarrollo de los scripts!

5. Ejecutar un script paso a paso

Ejecutar un script paso a paso, introducir puntos de interrupción, inspeccionar las variables durante la ejecución de un script; todas estas cosas forman parte del sueño de cualquier desarrollador de scripts que ya haya disfrutado de un lenguaje de desarrollo de alto nivel, como Visual Basic o C++. Bien sabe que todo esto no es un sueño, sino una realidad con PowerShell.

Para entrar en el modo de ejecución paso a paso, hará falta utilizar el comando siguiente:

```
Set-PSDebug -Step
```

La ejecución paso a paso va a permitir la ejecución de un script línea a línea, y para cada línea le va a tener que indicar al intérprete de comandos lo que debe hacer. Tendrá las opciones siguientes:

- **Sí** (tecla «S» o «Enter»): ejecuta el comando.
- **Sí a todo** (tecla «O»): sale del modo paso a paso y ejecuta el script hasta el final.
- **No** (tecla «N»): deniega la ejecución del comando actual.
- **No a todo** (tecla «T»): deniega la ejecución de todos los comandos hasta el final del script.
- **Suspender** (tecla «U»): suspende la ejecución del script en curso y entra en un intérprete de comandos anidados.

Veamos el ejemplo siguiente:

```
PS > Set-PSDebug -Step
PS > For ($i=1 ; $i -le 5; $i++) {Write-Host "Buenos días $i"}
```

Resultado:

```
¿Desea continuar con la operación?
1+ For ($i=1 ; $i -le 5; $i++) {Write-Host "Buenos días $i"}
```

Responderemos «Sí», tres veces seguidas y observaremos lo que ocurre.

```
PS > Set-PSDebug -Step
PS > For ($i=1 ; $i -le 5; $i++) {Write-Host "Buenos días $i"}

¿Desea continuar con la operación?
    1+ For ($i=1 ; $i -le 5; $i++) {Write-Host "Buenos días $i"}
[S] Sí      [O] Sí a todo    [N] No      [T] No a todo
[U] Suspendir    [?] Ayuda
(el valor predeterminado es "S"):
DEPURACIÓN:    1+ For ($i=1 ; $i -le 5; $i++) {Write-Host "Buenos días $i"}

¿Desea continuar con la operación?
    1+ For ($i=1 ; $i -le 5; $i++) {Write-Host "Buenos días $i"}
[S] Sí      [O] Sí a todo    [N] No      [T] No a todo
[U] Suspendir    [?] Ayuda
(el valor predeterminado es "S"):
DEPURACIÓN:    1+ For ($i=1 ; $i -le 5; $i++) {Write-Host "Buenos días $i"}
Buenos días 1

¿Desea continuar con la operación?
    1+ For ($i=1 ; $i -le 5; $i++) {Write-Host "Buenos días $i"}
[S] Sí      [O] Sí a todo    [N] No      [T] No a todo
[U] Suspendir    [?] Ayuda
(el valor predeterminado es "S"):
DEPURACIÓN:    1+ For ($i=1 ; $i -le 5; $i++) {Write-Host "Buenos días $i"}
Buenos días 2

¿Desea continuar con la operación?
    1+ For ($i=1 ; $i -le 5; $i++) {Write-Host "Buenos días $i"}
[S] Sí      [O] Sí a todo    [N] No      [T] No a todo
[U] Suspendir    [?] Ayuda
(el valor predeterminado es "S"):
```

Se válida una primera vez para confirmar la ejecución del comando, y las veces siguientes se válida para ejecutar cada Iteración. Observe que a cada iteración tenemos derecho a la presentación del resultado, exactamente como si ejecutásemos nuestro script normalmente.

Entramos ahora en el modo depuración escogiendo suspender la ejecución del script pulsando la tecla «U». Al hacer esto entramos en un sub-Shell o Shell anidado. A partir de ese momento, estaremos en una nueva instancia PowerShell y podremos examinar el contenido de las variables en curso de ejecución, e incluso modificarlas.

```
¿Desea continuar con la operación?
    1+ For ($i=1 ; $i -le 5; $i++) {Write-Host "Buenos días $i"}
[S] Sí      [O] Sí a todo    [N] No      [T] No a todo
[U] Suspendir    [?] Ayuda
(el valor predeterminado es "S"):U
PS >>>
PS >>> $i
3
PS >>> $i=-2
PS >>> exit

¿Desea continuar con la operación?
```

```

1+ For ($i=1 ; $i -le 5; $i++) {Write-Host "Buenos días $i"}
[S] Sí      [O] Sí a todo      [N] No      [T] No a todo
[U] Suspende      [?] Ayuda
(el valor predeterminado es "S"):
DEPURACIÓN: 1+ For ($i=1 ; $i -le 5; $i++) {Write-Host "Buenos días $i"}
Buenos días -2


¿Desea continuar con la operación?
1+ For ($i=1 ; $i -le 5; $i++) {Write-Host "Buenos días $i"}
[S] Sí      [O] Sí a todo      [N] No      [T] No a todo
[U] Suspende      [?] Ayuda
(el valor predeterminado es "S"):


```


Entrando en un shell anidado, constatará que un prompt ligeramente diferente al que estamos habituados aparece (observaremos un carácter doble de mayor que «>>»).

Hemos pedido el valor `$i` (que vale 3), después lo hemos modificado al valor -2. También podríamos haber hecho otras muchas cosas, como lanzar commandlets o scripts. Por último hemos abandonado el sub-Shell gracias el comando `exit`, y el modo paso a paso reanudó su curso, como si nada hubiera sucedido al mismo tiempo que hemos modificado `$i`.

¡He aquí la potencia de PowerShell! Poder depurar un script PowerShell sobre él mismo. Esta maravilla no se da siempre, ¿no?

 Hay que saber que la entrada en un Shell anidado puede hacerse en todo momento siempre que utilice el comando siguiente: `$host.EnterNestedPrompt()` Para saber si se encuentra en el Shell principal o en un Shell anidado, consulte el valor de la variable `$NestedPromptLevel`. Si ésta es diferente de **cero**, es que usted está en un Shell anidado.

 El hecho de que el prompt PowerShell se transforme añadiendo dos símbolos «>>» adicionales es debido a la definición de la función `Prompt`. Ésta se define así en la instalación de PowerShell (véase el capítulo Control del Shell). Si modifica la función `Prompt`, tenga en cuenta que es posible que tenga una visualización diferente.

 Para volver en un modo de ejecución normal y desactivar el modo paso a paso, el comando a introducir es `Set-PSDebug -Off`.

6. Los puntos de interrupción (break points) con PowerShell v1

Incluso antes de comenzar a presentar la utilización de los puntos de interrupción, es indispensable disociar la utilización de PowerShell V1 y V2. La utilización de los puntos de interrupción con PowerShell v1, descrita a continuación, no tiene comparación con la de PowerShell v2. Es por este motivo, que le animamos a utilizar la v2 en caso de tener que utilizar con asiduidad los puntos de interrupción. Sin embargo, si por razones especiales, desea colocar los puntos de interrupción, le indicamos a continuación cómo hacerlo.

Como acabamos de ver anteriormente, podemos utilizar el método `EnterNestedPrompt()` del objeto `$host` con el fin de suspender la ejecución de un script y entrar en un shell anidado. Esto corresponde de hecho a crear lo que se denomina comúnmente un «**punto de interrupción**». Podemos entonces en todo momento, utilizar en un script el comando `$host.EnterNestedPrompt()` si nos apetece.

Ahora bien, en el blog (<http://blogs.msdn.com/powershell/>) de los todopoderosos creadores de PowerShell, se puede encontrar una función interesante para crear puntos de interrupción; y la utilización de ésta, digamos que será más elegante que diseminar `$host.EnterNestedPrompt()`.

Es la siguiente:

```
function Start-Debug
{
    $scriptName = $MyInvocation.ScriptName
    function Prompt
    {
        "Debugging [{0}]> " -F $(if ([String]::IsNullOrEmpty
            $scriptName)) { 'globalscope' } else { $scriptName } )
    }
    $host.EnterNestedPrompt()
}

Set-Alias bp Start-Debug
```

Esta función va a modificar el prompt con el fin de mejorar un poco el estándar «>>» indicándole en qué ámbito se encuentra (globalscope o el del script). Esta información se obtendrá por `$MyInvocation.ScriptName`. Después se creará el alias «**bp**», para «**break point**», con el fin de facilitar la utilización de la función.

Ejemplo:

Veremos el resultado si escribe simplemente «**bp**» en el interprete de comando.

```
PS > bp
Debugging [globalscope]> $NestedPromptLevel
1
Debugging [globalscope]> exit
PS >
```

Práctico y elegante, ¿no cree?

Esta función podría ubicarse perfectamente en su perfil para utilizarse con asiduidad y así evitar estar buscándola cuando se necesite.

7. Los puntos de interrupción (break points) con PowerShell v2

La gestión de puntos de interrupción ha mejorado notablemente y se ha enriquecido en la versión 2 PowerShell. Mientras que en la versión 1.0 de PowerShell nos vimos obligados a crear nuestras propias funciones para depurar nuestros scripts (véase más arriba), v2 aporta su lote de nuevos comandos descritos a continuación:

Comando	Descripción
Set-PsBreakpoint	Permite definir un punto de interrupción.
Get-PsBreakpoint	Permite listar los puntos de interrupción.
Disable-PsBreakpoint	Permite desactivar los puntos de interrupción.
Enable-PsBreakpoint	Permite activar los puntos de interrupción.
Remove-PsBreakpoint	Permite suprimir los puntos de interrupción.
Get-PsCallStack	Permite visualizar la pila de las llamadas.

Ejemplo de utilización:

Tomemos como ejemplo la siguiente función que nos devolverá el tamaño libre en giga-bytes (GB) del disco C: así como el espacio total de todos nuestros discos.

```
Function Get-FreeSpace {

# Creación de la instancia del objeto WMI
$elementos = Get-WmiObject Win32_LogicalDisk

$tamaño_total = 0 # inicialización de la variable

# Bucle para recorrer todos los discos
foreach ( $disco in $elementos ) {
    if ($disco.Name -Like 'C:') {
        # Cálculo del tamaño en Gigabytes
        $tamaño = $disco.freespace / 1GB
        $tamaño = [math]::round($tamaño, 1) #Redondear el tamaño a 1 decimal
        Write-Host "El disco $($disco.Name) tiene $tamaño Gb disponibles"
    }
    $tamaño_total = $tamaño_total + $tamaño
}
Write-Host "Tamaño disponible acumulado = $tamaño_total Gb"
}
```

Colocaremos ahora un punto de interrupción en la entrada de la función:

```
PS > Set-PsBreakpoint -Command Get-FreeSpace

ID Script      Line Command      Variable      Action
--
0           Get-FreeSpace
```

Al ejecutar la función, el modo de depuración se activa:

```
PS > Get-FreeSpace
Entrando en modo de depuración. Use h o ? para obtener ayuda.
Alcanzar Punto de interrupción de comando en 'get-freespace'
Get-FreeSpace
[DBG]: PS >>>
```

Cuando el prompt PowerShell muestra [DBG], significa que usted se encuentra en el entorno de depuración de PowerShell. Para navegar por el depurador PowerShell, he aquí los comandos a utilizar:

Comando depurador	Descripción
s «Step-Into»	Ejecuta la instrucción siguiente, después se detiene.
v «Step-Over»	Ejecuta la instrucción siguiente, pero ignora las funciones y las llamadas.
o «Step-Out»	Efectuará un paso a paso fuera de la función actual, remontando un nivel si está anidado. Si se encuentra en el cuerpo principal, la ejecución continúa hasta el final o hasta el punto de interrupción siguiente.
c «Continue»	Continúa ejecutándose hasta que el script se termina o hasta que se alcance el punto de interrupción siguiente.
l «List»	Muestra la parte del script que se ejecuta. Por defecto el comando muestra la línea en curso, las cinco líneas anteriores y las 10 líneas siguientes. Para seguir listando

	el script, pulse [Intro].
L <x> «List»	Muestra 16 líneas del comienzo de script con el número de línea especificado por el valor <x>.
L <x> <n> «List»	Muestra <n> líneas del script empezando por el número de línea especificado por <x>.
G «Stop»	Para la ejecución del script y abandona el depurador.
K «Get-PsCallStack»	Muestra la pila de las llamadas.
<Intro>	Repite el último comando si se trata de Step (s), Step-over (v) o List (l). En los demás casos, representa una acción de envío.
?, h	Muestra la ayuda sobre los comandos del depurador.

Ejemplo :

```
PS > Get-FreeSpace
Entrando en modo de depuración.
[DBG]: PS >>> S
$elementos = Get-WmiObject Win32_LogicalDisk
[DBG]: PS >>> S
$tamaño_total = 0 # inicialización de la variable
[DBG]: PS >>> S
foreach ( $disco in $elementos ) {
[DBG]: PS >>> K

Command                Arguments                Location
-----
Get-FreeSpace           {}                        prompt
prompt                  {}                        prompt

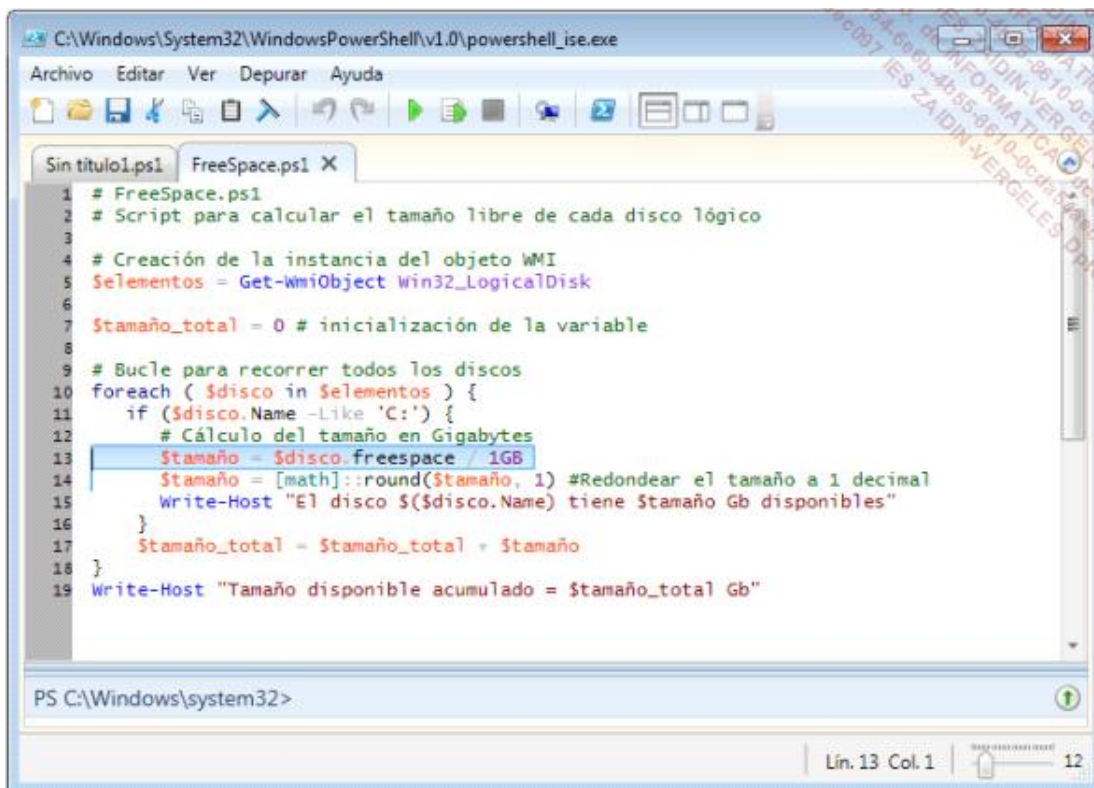
[DBG]: PS >>> Q
PS >
```

Para eliminar los puntos de interrupción, basta con utilizar el comando `Remove-PSbreakpoint` pasándole como argumento el nombre o el ID del punto de interrupción. Ejemplo, con el punto de interrupción con ID 0:

```
PS > Remove-PSbreakpoint -ID 0
```

Es de esta forma que usted podrá navegar con el depurador en modo consola. Sin embargo, con PowerShell ISE, la depuración también puede realizarse gráficamente.

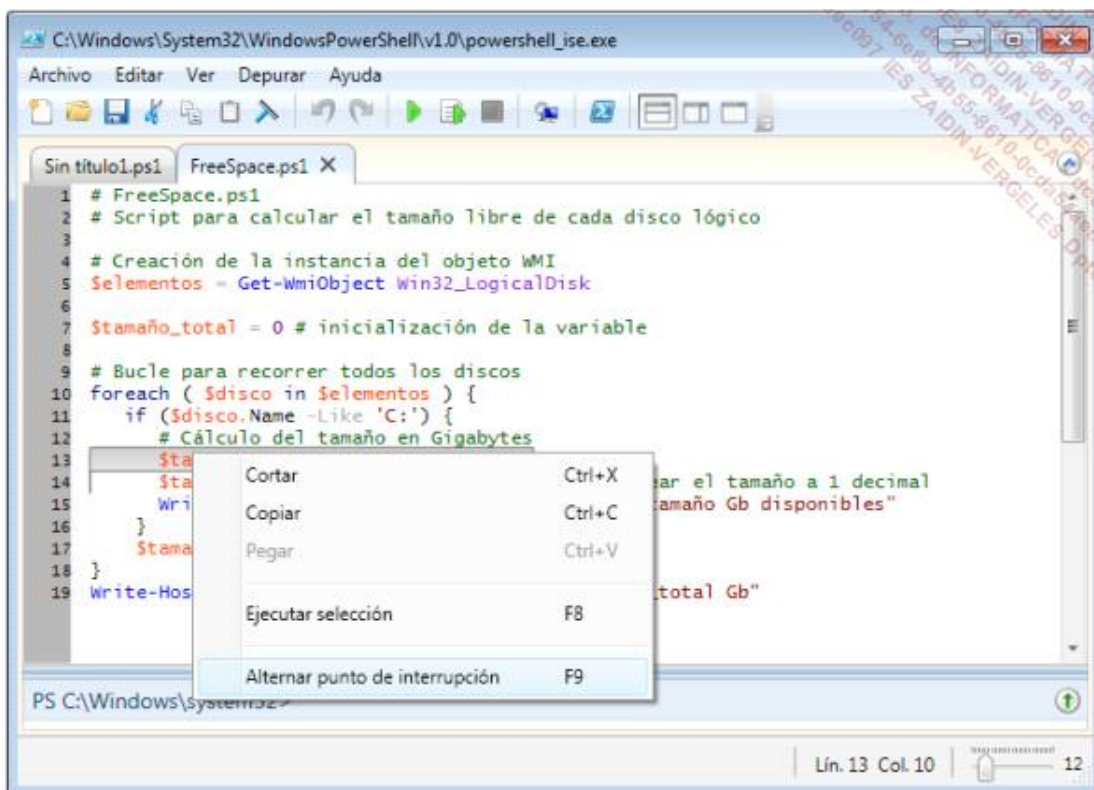
En el recuadro de edición (arriba), es posible seleccionar una línea y colocar un punto de interrupción.



Puntos de interrupción vía PowerShell ISE-1

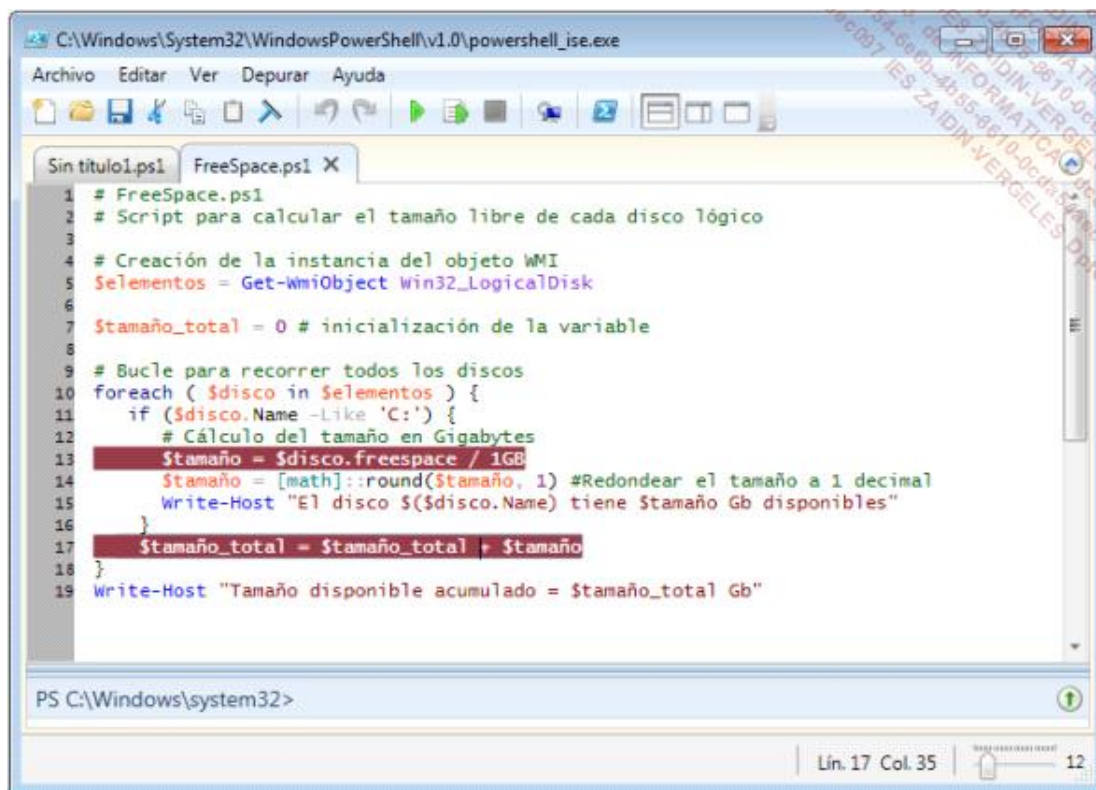
- Para poder situar los puntos de interrupción, PowerShell requiere que el script en curso de edición esté registrado.

El punto de interrupción se indica seleccionando la línea y haciendo posteriormente clic en el botón secundario del ratón pulsando la opción **Alternar punto de interrupción**. O bien pulsando la tecla [F9].



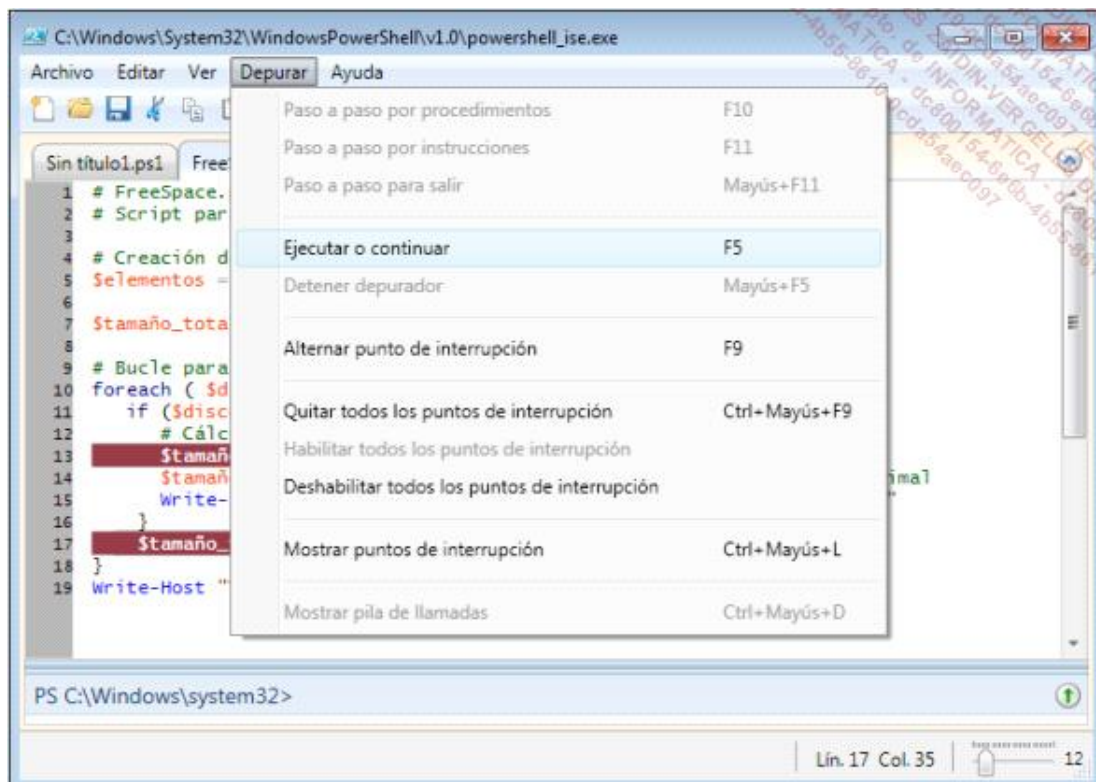
Puntos de interrupción vía PowerShell ISE-2

Varios puntos de interrupción pueden colocarse dentro de un mismo script.



Puntos de interrupción vía PowerShell ISE-3

Por último, la ejecución se realiza presionando la tecla [F5], o bien seleccionando **Ejecutar o continuar** en el menú **Depurar**.



Puntos de interrupción vía PowerShell ISE-4



En la ejecución, el estado de cada variable es visible posicionando el puntero del ratón encima.

8. Modo trace de Set-PSDebug

El modo «trace» nos permitirá comprender cómo PowerShell interpreta un script; veremos así el resultado de la ejecución de cada tratamiento. Esto nos permitirá, por ejemplo, descubrir más rápidamente la fuente de un bug.

La activación del modo «trace» se hace de la siguiente forma:

```
Set-PSDebug -Trace [1 | 2]
```

Existen dos modos de trace, el primero «-trace 1», es el modo básico que únicamente nos muestra los tratamientos. El segundo modo «-trace 2» es el modo detallado que muestra además de los tratamientos, todas las llamadas de scripts o funciones. Se encuentran también los términos «**nivel de seguimiento**» para designar a estos modos.

Utilicemos por ejemplo el script siguiente que nos devolverá el tamaño libre en giga-bytes del disco C: así como el espacio total de todos nuestros discos.

```
# FreeSpace.ps1
# Script para calcular el tamaño libre de cada disco lógico

# Creación de la instancia del objeto WMI
$elementos = Get-WmiObject Win32_LogicalDisk

$tamaño_total = 0 # inicialización de la variable

# Bucle para recorrer todos los discos
foreach ( $disco in $elementos ) {
    if ($disco.Name -Like 'C:') {
        # Cálculo del tamaño en Gigabytes
        $tamaño = $disco.freespace / 1GB
        $tamaño = [math]::round($tamaño, 1) #Redondear el tamaño a 1 decimal
        Write-Host "El disco $($disco.Name) tiene $tamaño Gb disponibles"
    }
    $tamaño_total = $tamaño_total + $tamaño
}
Write-Host "Tamaño disponible acumulado = $tamaño_total Gb"
```

Veamos el resultado empleando el primer modo de trace:

```
PS > Set-PSDebug -Trace 1
DEPURACIÓN: 2+ $foundSuggestion = $false
DEPURACIÓN: 4+ if ($lastError -and
DEPURACIÓN: 15+ $foundSuggestion
PS > ./FreeSpace.ps1
DEPURACIÓN: 1+ .\FreeSpace.ps1
DEPURACIÓN: 5+ $elementos = Get-WmiObject Win32_LogicalDisk
DEPURACIÓN: 7+ $tamaño_total = 0 # inicialización de la variable
DEPURACIÓN: 10+ foreach ( $disco in $elementos ) {
DEPURACIÓN: 11+ if ($disco.Name -Like 'C:') {
DEPURACIÓN: 13+ $tamaño = $disco.freespace / 1GB
DEPURACIÓN: 14+ $tamaño = [math]::round($tamaño, 1)
DEPURACIÓN: 15+ #Redondear el tamaño a 1 decimal
DEPURACIÓN: 15+ Write-Host "El disco $($disco.Name)
```

```

                tiene $tamaño Gb disponibles"
DEPURACIÓN: 1+ $disco.Name
El disco C: tiene 57.1 Gb disponibles
DEPURACIÓN: 17+ $tamaño_total = $tamaño_total + $tamaño
DEPURACIÓN: 11+ if ($disco.Name -Like 'C:') {
DEPURACIÓN: 17+ $tamaño_total = $tamaño_total + $tamaño
DEPURACIÓN: 19+ Write-Host "Tamaño disponible acumulado =
$tamaño_total Gb"
Tamaño disponible acumulado = 114.2 Gb
DEPURACIÓN: 2+ $foundSuggestion = $false
DEPURACIÓN: 4+ if ($lastError -and
DEPURACIÓN: 15+ $foundSuggestion

```

En la consola, observamos que todos los tratamientos se muestran en amarillo y como mensaje de depuración. Además, un número seguido del signo «+» se muestra ante cada tratamiento. Este número corresponde al número de línea del script en curso de ejecución. Se observa también que varios números de líneas reaparecen como el 11 y el 17. Esto es normal ya que nuestro script ejecuta un bucle empleando la instrucción `foreach`.

Veamos ahora lo que pasa al definir el nivel de seguimiento a 2:

```

PS > Set-PSDebug -Trace 2
DEPURACIÓN: 1+ Set-PSDebug -Trace 2
DEPURACIÓN: 2+ $foundSuggestion = $false
DEPURACIÓN: ! SET $foundSuggestion = 'False'.
DEPURACIÓN: 4+ if ($lastError -and
DEPURACIÓN: 15+ $foundSuggestion
PS > ./FreeSpace.ps1
DEPURACIÓN: 1+ .\FreeSpace.ps1
DEPURACIÓN: ! CALL function 'FreeSpace.ps1' (defined in file
'C:\FreeSpace.ps1')
DEPURACIÓN: 5+ $elementos = Get-WmiObject Win32_LogicalDisk
DEPURACIÓN: ! SET $elementos = '\\WRKOSCAR\root\cimv2:
Win32_LogicalDisk.DeviceID="C...'.
DEPURACIÓN: 7+ $tamaño_total = 0 # inicialización de la variable
DEPURACIÓN: ! SET $tamaño_total = '0'.
DEPURACIÓN: 10+ foreach ( $disco in $elementos ) {
DEPURACIÓN: 11+ if ($disco.Name -Like 'C:') {
DEPURACIÓN: 13+ $tamaño = $disco.freespace / 1GB
DEPURACIÓN: ! SET $tamaño = '57.0652503967285'.
DEPURACIÓN: 14+ $tamaño = [math]::round($tamaño, 1)
#Redondear el tamaño a 1 decimal
DEPURACIÓN: ! SET $tamaño = '57.1'.
DEPURACIÓN: 15+ Write-Host "El disco $($disco.Name)
tiene $tamaño Gb disponibles"
DEPURACIÓN: 15+ $disco.Name
El disco C: tiene 57.1 Gb disponibles
DEPURACIÓN: 17+ $tamaño_total = $tamaño_total + $tamaño
DEPURACIÓN: ! SET $tamaño_total = '57.1'.
DEPURACIÓN: 11+ if ($disco.Name -Like 'C:') {
DEPURACIÓN: 17+ $tamaño_total = $tamaño_total + $tamaño
DEPURACIÓN: ! SET $tamaño_total = '114.2'.
DEPURACIÓN: 19+ Write-Host "Tamaño disponible acumulado =
$tamaño_total Gb"
Tamaño disponible acumulado = 114.2 Gb
DEPURACIÓN: 2+ $foundSuggestion = $false
DEPURACIÓN: ! SET $foundSuggestion = 'False'.
DEPURACIÓN: 4+ if ($lastError -and
DEPURACIÓN: 15+ $foundSuggestion

```

En este modo vemos, además, aparecer la llamada a nuestro script, las distintas asignaciones de variables y sus valores

asociados, así como la llamada de los métodos estáticos del Framework .NET.

9. Trace-Command

Este commandlet permite obtener trazas de muy bajo nivel. Ha sido inicialmente concebida para (y por) los empleados de Microsoft encargados del desarrollo de PowerShell pero también para los que se ocupan de la asistencia a los usuarios. Su uso y su compleja interpretación la reservan más a los desarrolladores experimentados que a los usuarios finales de PowerShell que, en la versión 1.0 se contentaron por su parte con `set-PSDebug`. Existe muy poca documentación sobre `Trace-Command`.

Para el conjunto de operaciones, puede ser útil saber que la mecánica de trazas de este comando es la del Framework .NET.

Veamos cuales son los parámetros de `Trace-Command`:

Parámetro	Descripción
Name	Nombre del origen de seguimiento o traza.
Expression	Bloque de script a realizar el seguimiento.
Option	Tipo de eventos del que se va a realizar el seguimiento, All es el valor por defecto.
FilePath	Envío del resultado del seguimiento a un archivo.
Debugger	Envío del resultado del seguimiento al depurador.
PSHost	Envío del resultado del seguimiento a la pantalla.
ListenerOption	Nivel de detalle de cada línea del seguimiento.

Los parámetros más usuales son los siguientes:

- **-Name:** indica aquí el nombre del origen de la traza; es decir la información de la que nos interesa realizar el seguimiento. Por ejemplo podemos centrar nuestra atención en las conversiones de tipo que realiza PowerShell en una asignación de variable, o bien en la asignación de los parámetros en la llamada de un script o de un commandlet. Los orígenes de seguimiento son numerosos: hay cerca de ciento ochenta. Para conocerlos todos, use el comando: `Get-TraceSource`.
- **-Expression:** se especifica en este parámetro un bloque de scripts entre llaves. Ejemplo: `{./miScript.ps1}`.
- **-PSHost:** visualiza la traza en la pantalla.
- **-FilePath:** cuando la información es considerable es preferible dirigirla a un fichero. Indicar que esta opción puede utilizarse conjuntamente con `-PSHost`.

Los orígenes de seguimiento son increíblemente numerosos, para obtener una lista utilice el comando `Get-TraceSource`. Encontrará una lista completa en el anexo Lista de los orígenes de seguimientos.

Veamos una descripción de algunos orígenes de seguimiento:

Origen	Descripción
TypeConversion	Traza la mecánica interna de conversión de tipo. Por ejemplo, en la asignación de variables.
CommandDiscovery	Permite observar cómo el interprete de comandos hace para encontrar un

	comando o un script.
ParameterBinding	Traza la asociación de parámetros entre la llamada de un script o una función y el interprete de comandos.
FormatViewBinding	Permite saber si una vista predefinida existe o no.

Ejemplo:

Origen de seguimiento *TypeConversion*.

Tomemos un ejemplo sencillo donde definimos una variable forzando su tipo:

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

Asignamos a una variable de tipo `char` el valor «65», con el fin de obtener su carácter ASCII correspondiente, es decir «A».

Gracias a *Trace-Command*, vamos a entender mejor lo que acontece dentro de las entrañas de nuestro interprete de comandos preferido.

Probemos la línea de comandos siguiente:

```
PS > Trace-Command -Name TypeConversion -Expression {[char]$var=65} -Pshost
```

He aquí el resultado obtenido:

```
PS > Trace-Command -Name TypeConversion -Expression {[char]$var=65} -Pshost


DEPURACIÓN : TypeConversion Information: 0 : Converting "char" to
           "System.Type".
DEPURACIÓN : TypeConversion Information: 0 : Original type before getting
           BaseObject: "System.String".
DEPURACIÓN : TypeConversion Information: 0 : Original type after getting
           BaseObject: "System.String".
DEPURACIÓN : TypeConversion Information: 0 : Standard type conversion.
DEPURACIÓN : TypeConversion Information: 0 : Converting integer
           to System.Enum.
DEPURACIÓN : TypeConversion Information: 0 : Type conversion from string.
DEPURACIÓN : TypeConversion Information: 0 : Conversion
to System.Type
DEPURACIÓN : TypeConversion Information: 0 : The conversion is a
standard conversion. No custom type conversion will
be attempted.
DEPURACIÓN : TypeConversion Information: 0 : Converting "65" to
           "System.Char".
DEPURACIÓN : TypeConversion Information: 0 : Original type before
getting BaseObject: "System.Int32".
DEPURACIÓN : TypeConversion Information: 0 : Original type after
getting BaseObject: "System.Int32".
DEPURACIÓN : TypeConversion Information: 0 : Standard type conversion.
DEPURACIÓN : TypeConversion Information: 0 : Converting integer to
System.Enum.
DEPURACIÓN : TypeConversion Information: 0 : Type conversion from
string.
DEPURACIÓN : TypeConversion Information: 0 : Custom type conversion.
DEPURACIÓN : TypeConversion Information: 0 : Parse type conversion.
DEPURACIÓN : TypeConversion Information: 0 : Constructor type
conversion.
```



```

DEPURACIÓN : TypeConversion Information: 0 :      Cast operators type
conversion.
DEPURACIÓN : TypeConversion Information: 0 :      Looking for
"op_Implicit" cast operator.
DEPURACIÓN : TypeConversion Information: 0 :      Cast operator for
"op_Implicit" not found.
DEPURACIÓN : TypeConversion Information: 0 :      Looking for
"op_Explicit" cast operator.
DEPURACIÓN : TypeConversion Information: 0 :      Cast operator
for "op_Explicit" not found.
DEPURACIÓN : TypeConversion Information: 0 :      Could not find cast operator.
DEPURACIÓN : TypeConversion Information: 0 :      Cast operators
type conversion.
DEPURACIÓN : TypeConversion Information: 0 :      Looking
for "op_Implicit" cast operator.
DEPURACIÓN : TypeConversion Information: 0 :      Cast operator
for "op_Implicit" not found.
DEPURACIÓN : TypeConversion Information: 0 :      Looking
for "op_Explicit" cast operator.
DEPURACIÓN : TypeConversion Information: 0 :      Cast operator
for "op_Explicit" not found.
DEPURACIÓN : TypeConversion Information: 0 :      Could not find cast operator.
DEPURACIÓN : TypeConversion Information: 0 :      Conversion
using IConvertible succeeded.
DEPURACIÓN : TypeConversion Information: 0 :      Converting
"A" to "System.Char".
DEPURACIÓN : TypeConversion Information: 0 :      Result
type is assignable from value to convert's type

```

 El resultado obtenido puede cambiar en función de si se está utilizando PowerShell 1.0 ó 2.0. Aquí, poco importa la versión, lo esencial es mostrar la funcionalidad del commandlet `trace-command`.

Ejemplo:

Origen de seguimiento CommandDiscovery.

En este ejemplo, intentaremos ejecutar un script que no existe y observar el comportamiento del interprete de comandos.

Probemos la línea de comandos siguiente:

```

PS > Trace-Command -Name CommandDiscovery -Expression {c:\miScript.ps1}
-Pshost

```

He aquí el resultado obtenido:

```

PS > Trace-Command -Name CommandDiscovery -Expression {c:\miScript.ps1}
-PShost
DEPURACIÓN : CommandDiscovery Information: 0 : Looking up command:
c:\miScript.ps1
DEPURACIÓN : CommandDiscovery Information: 0 : Attempting to resolve
function or filter: c:\miScript.ps1
DEPURACIÓN : CommandDiscovery Information: 0 : The name appears to be
a qualified path: c:\miScript.ps1
DEPURACIÓN : CommandDiscovery Information: 0 : Trying to resolve the path
as an PSPath
DEPURACIÓN : CommandDiscovery Information: 0 : ERROR: The path could
not be found: c:\miScript.ps1
DEPURACIÓN : CommandDiscovery Information: 0 : The path is rooted,

```



```

so only doing the lookup in the specified directory:
c:\
DEPURACIÓN : CommandDiscovery Information: 0 : Looking for miScript.ps1
in c:\
DEPURACIÓN : CommandDiscovery Information: 0 : Looking for
miScript.ps1.ps1 in c:\
DEPURACIÓN : CommandDiscovery Information: 0 : Looking for
miScript.ps1.COM in c:\
DEPURACIÓN : CommandDiscovery Information: 0 : Looking for
miScript.ps1.EXE in c:\
DEPURACIÓN : CommandDiscovery Information: 0 : Looking for
miScript.ps1.BAT in c:\
DEPURACIÓN : CommandDiscovery Information: 0 : Looking for
miScript.ps1.CMD in c:\
DEPURACIÓN : CommandDiscovery Information: 0 : Looking for
miScript.ps1.VBS in c:\
DEPURACIÓN : CommandDiscovery Information: 0 : Looking for
miScript.ps1.VBE in c:\
DEPURACIÓN : CommandDiscovery Information: 0 : Looking for
miScript.ps1.JS in c:\
DEPURACIÓN : CommandDiscovery Information: 0 : Looking for
miScript.ps1.JSE in c:\
DEPURACIÓN : CommandDiscovery Information: 0 : Looking for
miScript.ps1.WSF in c:\
DEPURACIÓN : CommandDiscovery Information: 0 : Looking for
miScript.ps1.WSH in c:\
DEPURACIÓN : CommandDiscovery Information: 0 : Looking for
miScript.ps1.MSC in c:\
DEPURACIÓN : CommandDiscovery Information: 0 : The command
[c:\miScript.ps1] was not found, trying again with get-prepended
DEPURACIÓN : CommandDiscovery Information: 0 : Looking up command:
get-c:\miScript.ps1
DEPURACIÓN : CommandDiscovery Information: 0 : Attempting to resolve
function or filter: get-c:\miScript.ps1
DEPURACIÓN : CommandDiscovery Information: 0 : The name appears
to be a qualified path: get-c:\miScript.ps1
DEPURACIÓN : CommandDiscovery Information: 0 : Trying to resolve
the path as an PSPATH
DEPURACIÓN : CommandDiscovery Information: 0 : ERROR: A drive could
not be found for the path: get-c:\miScript.ps1
DEPURACIÓN : CommandDiscovery Information: 0 : ERROR: The drive does
not exist: get-c
DEPURACIÓN : CommandDiscovery Information: 0 : The path is relative,
so only doing the lookup in the specified directory:

DEPURACIÓN : CommandDiscovery Information: 0 : ERROR: 'get-c:\
miScript.ps1' is not recognized as a cmdlet, function, operable
program or script file.
El término 'c:\miScript.ps1' no se reconoce como nombre de un cmdlet,
función, archivo de script o programa ejecutable.
Compruebe si escribió correctamente el nombre o, si incluyó una ruta de
acceso, compruebe que dicha ruta es correcta e inténtelo de nuevo.
En línea: 1 Carácter: 66
+ Trace-Command -Name CommandDiscovery -Expression
{c:\miScript.ps1 <<<< -Pshost
+ CategoryInfo          : ObjectNotFound: (c:\miScript.ps1:String) [],
CommandNotFoundException
+ FullyQualifiedErrorId : CommandNotFoundException

```

Constatamos que PowerShell comienza a buscar una función o un filtro que contenga el nombre indicado

«c:\miscript.ps1»). Después, como no encuentra ninguno, determina que se trata de una ruta a un archivo. Busca entonces el archivo «miscript.ps1» en el directorio «c:\». Este archivo es imposible de encontrar, pasa entonces a revisar todas las extensiones contenidas en la variable de entorno `PATHEXT` con el fin de encontrar un archivo a ejecutar. Para finalizar, como la búsqueda ha resultado hasta el momento infructuosa, el interprete busca un commandlet de tipo «get» añadiendo el prefijo «get-» a «c:\miScript.ps1», es decir «get-c:\miscript.ps1». Finalmente, como se han agotado todas las soluciones PowerShell genera un error.

Interesante, ¿no? Es difícil imaginar todo lo que pasa detrás de una simple operación de ejecución de script.

Ejemplo:

Origen de seguimiento FormatViewBinding.

Este origen de seguimiento nos permite saber si el resultado de un comando mostrado en pantalla presenta un formateo por parte de PowerShell. En efecto, hemos podido constatar en el capítulo anterior que un gran número de tipos de objetos presentan un formateo por defecto, que se describe en los archivos **.ps1xml** contenidos en el directorio de instalación de PowerShell (en la variable `$PSHOME`, encontrándose por lo general en **C:\Windows\System32\WindowsPowerShell\v1.0**).

Probemos la línea de comandos siguiente:

```
PS > Trace-Command -Name FormatViewBinding -Expression {Get-Process  
notepad | Out-Host} -PSHost
```

He aquí el resultado obtenido:

```
PS > Notepad.exe  
PS > Trace-Command -Name FormatViewBinding -Expression {Get-Process  
notepad | Out-Host} -PSHost  
  
DEPURACIÓN : FormatViewBindin Information: 0 : FINDING VIEW TYPE:  
System.Diagnostics.Process  
DEPURACIÓN : FormatViewBindin Information: 0 : NOT MATCH Table NAME:  
ThumbprintTable TYPE:  
System.Security.Cryptography.X509Certificates.X509Certificate2  
DEPURACIÓN : FormatViewBindin Information: 0 : NOT MATCH List NAME:  
ThumbprintList GROUP: CertificateProviderTypes  
DEPURACIÓN : FormatViewBindin Information: 0 : NOT MATCH Wide NAME:  
ThumbprintWide GROUP: CertificateProviderTypes  
DEPURACIÓN : FormatViewBindin Information: 0 : NOT MATCH Table NAME:  
PSThumbprintTable TYPE: System.Management.Automation.Signature  
DEPURACIÓN : FormatViewBindin Information: 0 : NOT MATCH Wide NAME:  
PSThumbprintWide TYPE: System.Management.Automation.Signature  
DEPURACIÓN : FormatViewBindin Information: 0 : NOT MATCH List NAME:  
PathOnly GROUP: CertificateProviderTypes  
DEPURACIÓN : FormatViewBindin Information: 0 : NOT MATCH Table NAME:  
System.Security.Cryptography.X509Certificates.X509CertificateEx TYPE:  
System.Security.Cryptography.X509Certificates.X509CertificateEx  
DEPURACIÓN : FormatViewBindin Information: 0 : NOT MATCH Table NAME:  
System.Reflection.Assembly TYPE: System.Reflection.Assembly  
DEPURACIÓN : FormatViewBindin Information: 0 : NOT MATCH Table NAME:  
System.Reflection.AssemblyName TYPE: System.Reflection.AssemblyName  
DEPURACIÓN : FormatViewBindin Information: 0 : NOT MATCH Table NAME:  
System.Globalization.CultureInfo TYPE: System.Globalization.CultureInfo  
DEPURACIÓN : FormatViewBindin Information: 0 : NOT MATCH Table NAME: System.  
Diagnostics.FileVersion Info TYPE: System.Diagnostics.FileVersionInfo  
DEPURACIÓN : FormatViewBindin Information: 0 : NOT MATCH Table NAME:  
System.Diagnostics.EventLogEntry TYPE: System.Diagnostics.EventLogEntry  
DEPURACIÓN : FormatViewBindin Information: 0 : NOT MATCH Table NAME:  
System.Diagnostics.EventLog TYPE: System.Diagnostics.EventLog
```

```

DEPURACIÓN : FormatViewBindin Information: 0 :      NOT MATCH Table NAME:
System.Version  TYPE: System.Version
DEPURACIÓN : FormatViewBindin Information: 0 : NOT MATCH Table NAME:
System.Drawing.Printing.PrintDocument TYPE: System.Drawing.Printing.
PrintDocument
DEPURACIÓN : FormatViewBindin Information: 0 :      NOT MATCH Table NAME:
Dictionary TYPE: System.Collections.DictionaryEntry
DEPURACIÓN : FormatViewBindin Information: 0 :      NOT MATCH Table NAME:
ProcessModule  TYPE: System.Diagnostics.ProcessModule
DEPURACIÓN : FormatViewBindin Information: 0 :      MATCH FOUND Table NAME:
process  TYPE: System.Diagnostics.Process
DEPURACIÓN : FormatViewBindin Information: 0 :      MATCH FOUND Table NAME:
process  TYPE: Deserialized.System.Diagnostics.Process
DEPURACIÓN : FormatViewBindin Information: 0 : An applicable
view has been found

```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
-----	-----	-----	-----	-----	-----	--	-----
53	3	1444	5564	53	0,41	2888	notepad
53	2	1484	5436	53	1,17	5476	notepad

Podemos observar en todas las últimas líneas la siguiente información «DEPURACIÓN: FormatViewBindin Information: 0 : An applicable view has been found». Esto significa que una vista ha sido encontrada.

En cuanto a la primera línea «DEPURACIÓN: FormatViewBindin Information: 0 : FINDING VIEW TYPE: System.Diagnostics.Process», esta sí es interesante ya que nos indica precisamente el nombre del tipo de la vista.

Si no se ha encontrado ninguna vista para el tipo en cuestión, hubiésemos obtenido el mensaje siguiente en la última línea: «DEPURACIÓN: FormatViewBindin Information: 0 : No applicable view has been found».