



*Universidad Nacional de La Matanza*  
Florencio Varela 1903 - San Justo - Buenos Aires - Argentina

## **Departamento de Ingeniería e Investigaciones Tecnológicas**

# **Cátedra de Sistemas Operativos**

**Jefe de Cátedra:**

**Fabio Rivalta**

**Autor:**

**Fernando Piubel**

**Año:**

**2015**

**Revisión:**

**2016**

## **Guía sobre Power Shell**

## Contenido

Introducción .....	3
Instalación .....	4
Consola.....	5
Windows PowerShell ISE .....	6
Sintaxis de scripts .....	7
Variables y parámetros.....	7
Uso de comillas .....	7
Comentarios .....	8
Condiciones y ciclos .....	8
Funciones .....	9
Parámetros .....	9
Opciones de parámetros .....	10
Estructura de las funciones .....	10
Uso de la Ayuda .....	11
Manejo de Errores .....	14
Cmdlets más usados .....	16
Bibliografía .....	17

## Introducción

"Windows PowerShell es una interfaz de consola (CLI) con posibilidad de escritura y unión de comandos por medio de instrucciones (scripts en inglés). Es mucho más rica e interactiva que sus predecesores, desde DOS hasta Windows 7. Esta interfaz de consola está diseñada para su uso por parte de administradores de sistemas, con el propósito de automatizar tareas o realizarlas de forma más controlada. Originalmente denominada como MONAD en 2003, su nombre oficial cambió al actual cuando fue lanzada al público el 25 de abril del 2006.

PowerShell no sólo permite interactuar con el sistema operativo, sino también con programas de Microsoft como SQL Server, Exchange o IIS. La principal utilidad de PowerShell es permitir automatizar tareas administrativas al usuario.

El lenguaje de la consola incluye declaración de variables, variables especiales predefinidas, operadores matemáticos, incluyendo igualdades y desigualdades numéricas, manejo de vectores, comparación entre estos, operadores de asignación, vectores asociativos (hashtables), valores booleanos, ciclos y ruptura de los mismos, operadores de expansión para simplificación de ejecuciones complejas (creación de vectores por medio de llamados a procedimientos, creación dinámica de vectores, etc.); comentarios, operadores de comparación binaria, caracteres de escape, orden de ejecución, ciclos del tipo "foreach", creación de procedimientos y funciones, creación de filtros, estructuras condicionales complejas (if/then/else/elseif/switch), operador de invocación dinámica del contenido de variables (\$p = "MiProceso" --> &\$p ejecuta MiProceso), llamado a métodos de tipo "\$p.ToUpper()", acceso a propiedades de instancias de objetos, redirección de salida normal de consola a archivos, retorno de valores, manejo de cadenas de caracteres por medio de operadores, manejo de excepciones y conversión explícita de tipos.

Una de las principales funciones de PowerShell es tratar de igualar al famoso lenguaje Perl de UNIX. El cual está considerado versátil, potente y con facilidad para interactuar con el sistema operativo. Exchange server 2007 utiliza PowerShell internamente. La tendencia es que todas las aplicaciones tengan su sección visual y una opción para ver el código generado en PowerShell.

La característica distintiva de PowerShell, es que es un intérprete de comandos orientado a objetos. La información de entrada y de salida en cada etapa del proceso es un conjunto de instancias de objeto, a diferencia de lo que ocurre con los intérpretes de comandos tradicionales, que sólo devuelven y reciben texto."

Fuente: Windows PowerShell - <https://es.wikipedia.org>

## Instalación

Para la creación y prueba de los scripts utilizaremos la versión 4.0 del Windows Management Framework 4.0.

Para instalar la versión 4.0 es necesario ingresar al siguiente link y descargar la versión correspondiente al sistema operativo que se utilice:

<http://www.microsoft.com/es-ES/download/details.aspx?id=40855>

Una vez instalado el framework abrir la consola de PowerShell y ejecutar el siguiente comando: Get-Host y verificar que la versión que se muestra sea la 4.0

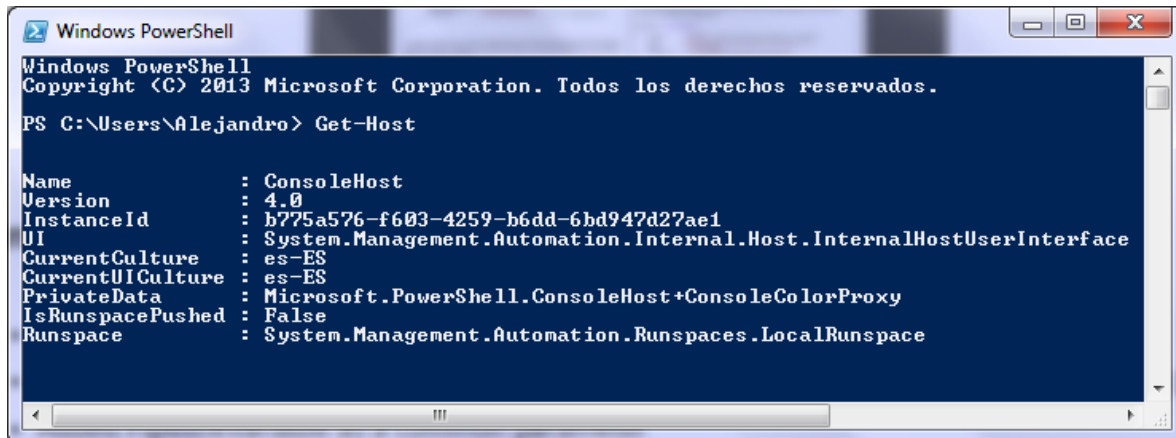


Imagen 1 – Versión del framework instalado

## Consola

Desde PowerShell se pueden ejecutar tres tipos de comandos: cmdlets, scripts y programas ejecutables (binarios).

Los cmdlets son comandos de PowerShell que realizan una acción y como resultado devuelven un objeto basado en el framework .NET de Microsoft. Sus nombres están formados por un verbo y un sustantivo, separados por un guion, como por ejemplo, *Get-Help*, *Select-String*, *Format-Table*, *Set-Variable*, *Start-Process*, *Stop-Process*, *Update-Help*, *Write-Output*.

A diferencia de los comandos de otros shells, los cmdlets tienen las siguientes características:

- Son instancias de clases de .NET.
- Generalmente no manejan su propia salida de datos, ya sea la salida por pantalla o la salida de error. Estas tareas las maneja el mismo PowerShell.
- Procesan objetos como entrada en vez de texto, y normalmente también devuelven objetos. Normalmente, la salida de un cmdlet, es la entrada de otro.

Para pasar la salida de un cmdlet como entrada de otro se usa el símbolo "|".

A continuación se mencionan y explican algunos de los cmdlets más comunes.

- *Get-Command*: Muestra un listado de todos los cmdlets que se encuentran disponibles para usar.
- *Get-Help*: Muestra la ayuda de un cmdlet. Ejemplo:

```
Get-Help Get-Command      # Muestra la ayuda de Get-Command
Get-Help Get-ChildItem -detailed # Muestra la ayuda detallada de Get-ChildItem
Get-Help Select-String -online  # Muestra la ayuda de Select-String en Internet
```

- *Update-Help*: Actualiza la ayuda.
- *Get-Member*: Muestra las propiedades, métodos y eventos de un tipo de objeto. Ejemplo de uso:

```
Get-ChildItem | Get-Member
# En este caso mostrará información acerca de las clases System.IO.DirectoryInfo y
System.IO.FileInfo, que son los tipos de objetos que devuelve Get-ChildItem.
```

- *Get-ChildItem*: Muestra el contenido de un directorio.
- *Get-Process*: Muestra los procesos que están corriendo actualmente en el sistema.
- *Get-Counter*: Muestra contadores de performance (uso de la red, uso del procesador, uso de memoria, etc.) del equipo local o de un equipo remoto.
- *Start-Sleep -Seconds 10*: Hace una pausa de 10 segundos.
- *Write-Output*: Muestra un mensaje por pantalla.
- *Read-Host*: Lee una línea de entrada de la consola.
- *Exit*: Termina el script actual o la sesión de PowerShell.
- *Get-host*: obtiene un objeto con la información del host. Con este objeto se puede verificar la versión de PowerShell.

Teniendo en cuenta que PowerShell está pensado como una herramienta de administración de sistemas, muchos de los comandos pueden ser ejecutados de manera remota en otra computadora, normalmente con el parámetro *-ComputerName*.

## Windows PowerShell ISE

Windows PowerShell ISE (Integrated Scripting Engine), es el IDE que nos provee Windows donde podemos ejecutar comandos o cmdlets, escribir, probar, crear, guardar o depurar scripts en un entorno con interfaz gráfica, con edición multimedia, autocompletado con el tabulador, sintaxis coloreada, ejecución selectiva, ayuda contextual y compatibilidad con idiomas que se escriben de derecha a izquierda, con ella podemos tener una experiencia de usuario más enriquecida a la hora de automatizar nuestras tareas de administración con la creación de scripts y la fácil incorporación cmdlets, gracias al panel de módulos que nos provee ayuda en la utilización y búsqueda de los mismos.

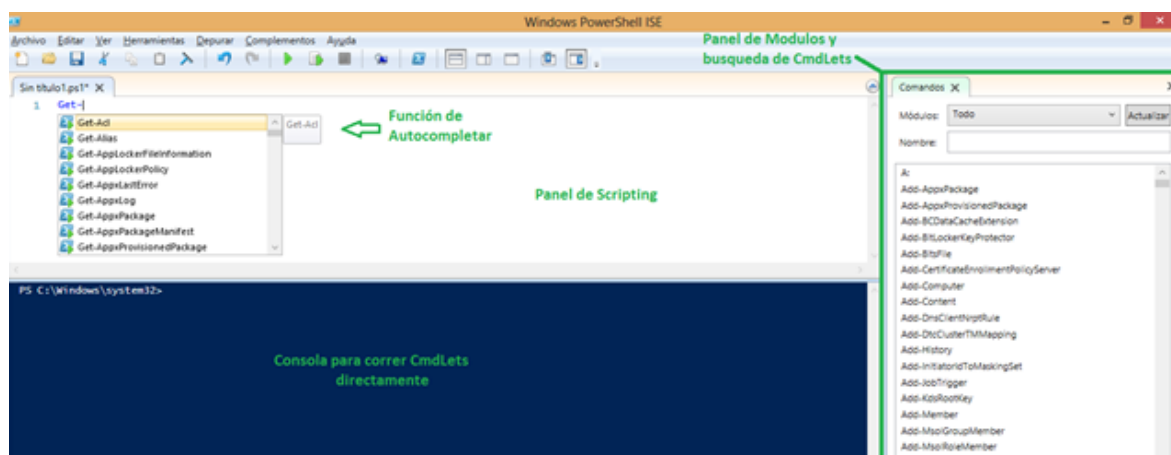


Imagen 2 - Windows PowerShell ISE

## Sintaxis de scripts

### Variables y parámetros

PowerShell soporta el uso de variables en scripts. Todos los nombres de variables empiezan con el signo \$, pueden ser dinámicas o de un tipo específico y no son case sensitive (al igual que todos los comandos).

Las variables pertenecen al ámbito en donde fueron declaradas. Esto quiere decir que si se declara una variable dentro de una función, sólo será accesible dentro de ella.

Las siguientes son variables propias de PowerShell:

Variable	Contiene
\$_	Objeto proveniente del pipeline
\$Args	Array que contiene cada uno de los parámetros pasados
\$?	Estado del último comando ejecutado. True si fue exitoso, False en caso de error
\$False	Constante False
\$True	Constante True
\$Null	Constante Null
\$PSVersionTable	Versión de PowerShell
\$Pwd	Path actual
\$Errors	Array con los últimos errores, siendo \$Error[0] el más reciente

Ejemplo:

```
write-Output ('Primer parámetro: ' + $Args[0])  
write-Output ('Último parámetro: ' + $Args[-1])  
write-Output ('Cantidad de argumentos: ' + $Args.Count)
```

Salida:

```
Primer parámetro: param1  
Último parámetro: param3  
Cantidad de argumentos: 3
```

### Uso de comillas

Es importante señalar que al momento de trabajar con texto, el uso de las comillas varía dependiendo del comportamiento que se espera. Existen 3 tipos de comillas a utilizar: las comillas dobles ("), comillas simples (') y acento grave (`).

- Las comillas dobles, también denominadas comillas débiles, permiten utilizar texto e interpretar variables al mismo tiempo.
- Las comillas simples, también denominadas comillas fuertes, generan que el texto delimitado entre ellas se utilice de forma literal, lo que evita que se interpreten las variables.
- El acento grave, es el carácter de escape para evitar la interpretación de los caracteres especiales, como por ejemplo el símbolo \$.

Ejemplo:

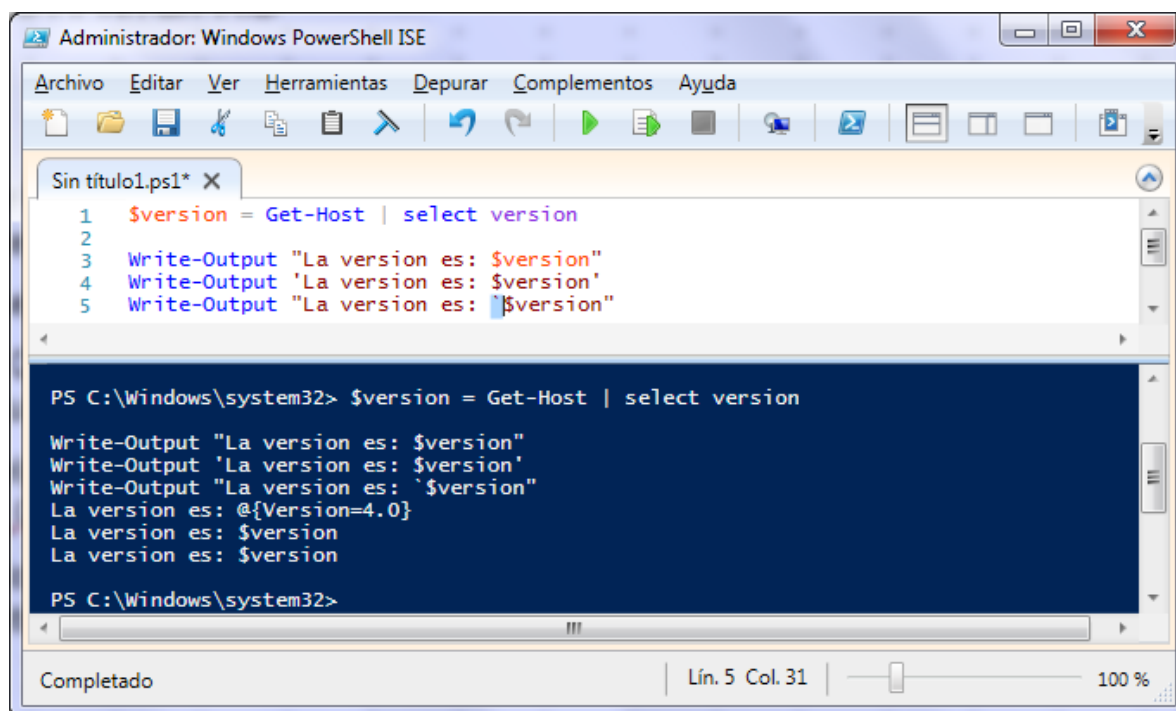


Imagen 3 - Utilización de comillas

## Comentarios

Para la utilización de comentarios dentro del código tenemos 2 opciones, los comentarios de línea o los de múltiples líneas:

Ejemplo:

```
# Comentario Simple
<# Comentario
  Multi
  Línea
#>
```

## Condiciones y ciclos

El siguiente ejemplo muestra la sintaxis de las condiciones y ciclos de PowerShell:

```
# Condiciones
if ($variable -eq 'valor')
{ ... }
elseif ($variable -lt 'valor')
{ ... }
else
{ ... }

# Ciclo for
for ($i = 0; $i -lt 10; $i++)
{ ... }

# Ciclo foreach
foreach ($item in $array)
{ ... }

# Ciclo while
while ($variable -lt 5)
{ ... }

# Ciclo do while
do
{ ... } while ($variable -lt 10)
```



## Funciones

En los scripts de PowerShell se pueden definir funciones para facilitar el armado de scripts.

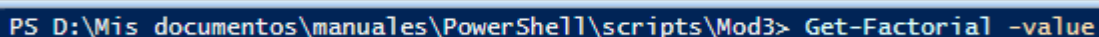
El siguiente es un script que utiliza una función de manera recursiva para calcular el factorial de un número.

```
function Get-Factorial()
{
    Param (
        [int]$value
    )
    if($value -lt 0)
    {
        $result = 0
    }
    elseif($value -le 1)
    {
        $result = 1
    }
    else
    {
        $result = $value * (Get-Factorial($value - 1))
    }
    return $result
}

$value = Read-Host 'Ingrese un número'
$result = Get-Factorial $value # $result = Get-Factorial -value $value
Write-Output "$value! = $result"
```

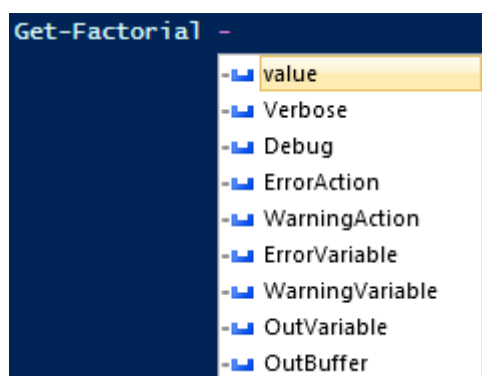
El uso de funciones, nos permite crear funcionalidades similares a los cmdlets y por tal motivo, se brinda la directiva **CmdLetBinding**, que transforma nuestras funciones en una especie de CmdLets proporcionando los parámetros comunes como -Verbose -Debug.

De la función anterior obtenemos el siguiente comportamiento:



```
PS D:\Mis documentos\manuales\PowerShell\scripts\Mod3> Get-Factorial -value
```

Pero al aplicar la directiva [CmdLetBinding()] el comportamiento cambia a:



## Parámetros

A pesar de que se puede acceder a los parámetros del script con la variable \$Args, también es posible accederlos al igual que hacemos en Shell scripting \$1, \$2, etc. y adicionalmente, podemos declarar parámetros con nombre y asignarle un tipo de datos y validaciones.

En el siguiente ejemplo se declaran dos parámetros con nombre. El primero, llamado *name*, es obligatorio (Mandatory=\$True), de tipo String y si no se lo pasa por nombre, será el primer

parámetro sin nombre (Position=1). El segundo parámetro, *firstName*, no es obligatorio y al no tener especificada una posición, si no se lo pasa por nombre no tendrá valor.

Código:

```
Param(
    [Parameter(Mandatory=$True, Position=1)] [string]$name,
    [Parameter(Mandatory=$False)] $firstName
)

write-Output "Hola $name $firstName"
```

Se ejecuta de la siguiente manera:

```
# Los dos parámetros se pasan por nombre.
.\script.ps1 -name 'Nombre' -firstName 'Apellido'

# Un parámetro es pasado por nombre y el otro por posición.
.\script.ps1 -firstName 'Apellido' 'Nombre'

# No se puede ejecutar de esta manera, ya que dará error porque no se especificó el
parámetro "name" que era obligatorio.
.\script.ps1 -firstName 'Apellido'
```

Salida:

```
Hola Nombre Apellido
```

## Opciones de parámetros

La definición de parámetros en PowerShell, nos permite especificar una gran variedad de opciones y validaciones, que el Engine se encargará de resolver automáticamente al recibir los valores en los parámetros.

Además de las opciones ya vistas, se pueden añadir validaciones a los parámetros, ya sea de tipo, rango, largo, vacío o null, patrones (con expresiones regulares), entre otras.

```
Param
(
    [parameter(Mandatory=$true)]
    [ValidateNotNullOrEmpty()]
    [ValidateLength(5,8)] # De 5 a 8 caracteres de largo
    [ValidatePattern('^.*@.*\.[a-z]{3}$')]
    [string]$cadena,

    [parameter(Mandatory=$true)]
    [ValidateNotNullOrEmpty()]
    [ValidateRange(1,10)]
    [int]$entero
)
```

## Estructura de las funciones

Ya hemos visto la definición de los parámetros y ahora veremos las secciones que se pueden definir dentro de las funciones:

```
Begin{<#Code#>}
Process{<#Code#>}
End{<#Code#>}
```

Esta estructura es similar a la de AWK, donde tenemos el bloque Begin, que se ejecutará por única vez al comienzo de la invocación y el bloque End que se ejecutará por única vez, al finalizar la ejecución. En cambio, el bloque Process veremos que se puede ejecutar más de una vez si utilizamos la función en un pipeline de ejecución.

En el siguiente ejemplo veremos la iteración del bloque Process:

```
function Get-Multiplicacion()
{
    [CmdletBinding()]
    param(
        [Parameter(ValueFromPipeline=$true,
                    ValueFromPipelineByPropertyName=$true)]
        [int]$value
    )
    Begin{ $result = 0 }
    Process {
        Write-Output "Resultado parcial: $result"

        if($value -lt 0)
        {
            $result = 0
        }
        elseif($value -le 1)
        {
            $result = 1
        }
        else
        {
            $result = $value * $result
        }
    }
    End{ Write-Output "Resultado Final: $result" }
}
```

Resultado de la ejecución:

```
PS D:\Mis documentos\manuales\PowerShell\scripts\mod4> 1..8 | Get-Multiplicacion
Resultado parcial: 0
Resultado parcial: 1
Resultado parcial: 2
Resultado parcial: 6
Resultado parcial: 24
Resultado parcial: 120
Resultado parcial: 720
Resultado parcial: 5040
Resultado Final: 40320
```

## Uso de la Ayuda

Para obtener ayuda sobre los cmdlet existe el Get-Help, que nos brindará la información funcional, sobre los parámetros, el modo de uso, entre otros temas.

PowerShell genera de forma automática, una hoja de ayuda con las funciones que creamos, con la información básica. Por ejemplo, de la función anterior Get-Factorial, obtenemos lo siguiente:

```
PS D:\Mis documentos\manuales\PowerShell\scripts\mod4> get-help Get-Factorial

NAME
    Get-Factorial

SYNTAX
    Get-Factorial [[-value] <int>] [<CommonParameters>]

ALIASES
    None

REMARKS
    None
```

Esto nos brinda una ayuda estándar para todos los cmdlet y funciones de usuario que se desarrollan. A continuación, veremos cómo explotar un poco más la utilización de la ayuda.

La recopilación de la información de ayuda de la función se realiza mediante metadata que se debe especificar inmediatamente antes de la palabra reservada function, sin dejar espacios ni saltos de línea, ya que de lo contrario no se identificará como parte de la ayuda sino como un comentario del script.

Dentro de la especificación existen varias secciones que son las que luego se mostraran con el Get-Help. Las mismas son:

- .Synopsis
- .Description
- .Example (puede ser más de 1)
- .Inputs
- .Outputs

A continuación se muestra un ejemplo básico de estas secciones:

```
<#
.Synopsis
    Short description
.DESCRIPTION
    Long description
.EXAMPLE
    Example of how to use this cmdlet
.EXAMPLE
    Another example of how to use this cmdlet
.INPUTS
    Inputs to this cmdlet (if any)
.OUTPUTS
    Output from this cmdlet (if any)
.NOTES
    General notes
.COMPONENT
    The component this cmdlet belongs to
.ROLE
    The role this cmdlet belongs to
.FUNCTIONALITY
    The functionality that best describes this cmdlet
#>
function Verb-Noun {
<# CODE #>
}
```

Agregando la sección de ayuda a la función Get-Factorial obtenemos el siguiente resultado:

```
<#
.Synopsis
Esta función calcula el factorial de un número de manera recursiva
.Description
Esta función calcula el factorial de un número de manera recursiva
.Parameter value
```

```
Este parámetro indicará cuál el número al cuál calcular el factorial
.Example
Get-Factorial -value 5
120
#>
function Get-Factorial()
{
```

```
PS D:\Mis documentos\manuales\PowerShell\scripts\mod4> Get-Help Get-Factorial

NAME
    Get-Factorial

SYNOPSIS
    Esta función calcula el factorial de un número de manera recursiva

SYNTAX
    Get-Factorial [[-value] <Int32>] [<CommonParameters>]

DESCRIPTION
    Esta función calcula el factorial de un número de manera recursiva

RELATED LINKS

REMARKS
    To see the examples, type: "get-help Get-Factorial -examples".
    For more information, type: "get-help Get-Factorial -detailed".
    For technical information, type: "get-help Get-Factorial -full".
```

Note que para obtener diferentes grados de información podemos utilizar los parámetros – Example, –Detailed o –Full, los cuales irán mostrando un mayor detalle de la ayuda informada en la metadata de la función.

## Manejo de Errores

Para el manejo de errores, tendremos la opción de capturarlos y darles un tratamiento adecuado, al igual que podemos hacer en lenguajes como .NET o Java, entre otros, y adicionalmente, definir un comportamiento por defecto ante la aparición de alguna excepción.

La sintaxis es la siguiente:

```
try {  
    <#Código con posibilidad de error#>  
}  
catch {  
    <#Acciones a tomar ante la excepción#>  
}  
finally {  
    <#Acciones que siempre se ejecutaran#>  
}
```

Al incorporar la directiva `CmdLetBinding`, que ya hemos mencionado anteriormente, se nos habilitan los siguientes argumentos a las funciones: *ErrorAction* y *ErrorVariable*.

**ErrorAction (EA)**, es una enumeración que nos permite setear el comportamiento ante una excepción en la ejecución del `CmdLet` o la función. Si queremos que el error encontrado sea capturado, entonces se debe setear el argumento con "Stop".

**ErrorVariable (EV)**, nos permite crear una variable donde se almacenaran los errores obtenidos para inspeccionarlos y poder analizar las causas y determinar las acciones a tomar.

Adicionalmente, a estos argumentos, es importante resaltar la existencia y diferencia en el uso de los `cmdLets` para la escritura de información:

- **Write-Error:** Muestra el mensaje especificado y finaliza la ejecución de la función.
- **Write-Warning:** Muestra el mensaje en un color diferente al normal a la consola, con la leyenda "WARNING" o "ADVERTENCIA", dependiendo del lenguaje.
- **Write-Output:** Muestra el mensaje especificado en el estándar output.
- **Write-Debug:** Muestra el mensaje en un color diferente al normal con la leyenda "Debug", solamente si la variable `-Debug` ha sido seteada en la ejecución del `CmdLet`.

```
function Detener-Proceso()  
{  
    [CmdLetBinding()]  
    Param(  
        [Parameter(Mandatory=$True,  
                    ValueFromPipeline=$true,  
                    ValueFromPipelineByPropertyName=$true)]  
        [Int[]] $Id  
    )  
    Process{  
        foreach ($proc in $Id)  
        {  
            try{  
                kill -Id $proc  
                write-Output "El proceso $proc fue detenido exitosamente"  
            }  
            catch{  
                write-warning "El proceso $proc no existe en el sistema"  
            }  
        }  
    }  
}
```

Ejemplos de uso:

- 1- Ejecutar la función con Ids inexistentes:

```
PS> Detener-Proceso -Id 3324, 3325 -ErrorVariable errorkill -ErrorAction Stop
```

El resultado será el siguiente:

```
ADVERTENCIA: El proceso 3324 no existe en el sistema
ADVERTENCIA: El proceso 3325 no existe en el sistema
```

## 2- Uso del Pipeline:

Verificamos que existan procesos:

```
PS> get-process -Name notepad*
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
61	3	1212	4904	64	0,06	5344	notepad
121	9	10592	16760	100	0,42	4124	notepad++

Ejecución de la instrucción:

```
PS> get-process -Name notepad* | Detener-Proceso -EA Stop -EV errorkill
```

```
El proceso 5344 fue detenido exitosamente
El proceso 4124 fue detenido exitosamente
```

## 3- Ejecución de la función para Ids existentes e inexistentes:

```
PS> Detener-Proceso -Id 4364, 3984 -EV errorkill -EA Stop
```

```
ADVERTENCIA: El proceso 4364 no existe en el sistema
El proceso 3984 fue detenido exitosamente
```

```
PS> $errorkill
```

Se detuvo el comando en ejecución porque la variable de preferencia "ErrorActionPreference" o un parámetro común se han establecido en Stop: No se encuentra ningún proceso con el identificador 4364.

kill : No se encuentra ningún proceso con el identificador 4364.

En C:\Users\Public\Documents\Manuales\PowerShell\Scripts\pruebas.ps1: 15 Carácter: 17

```
+ kill -Id $proc
```

```
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (4364:Int32) [Stop-Process],
ProcessCommandException
+ FullyQualifiedErrorId :
NoProcessFoundForGivenId,Microsoft.PowerShell.Commands.StopProcessCommand
```

## Cmdlets más usados

A continuación una lista de los cmdlets más comunes y equivalentes de Linux:

- Archivos:
  - Get-ChildItem
  - Copy-Item
  - Move-Item
  - Get-Content
  - Set-Content
  - Out-File
  - Out-Null
  - Export-Csv
- Cadenas:
  - Select-String
- Manejo de colecciones de objetos:
  - Select-Object
  - Where-Object
  - ForEach-Object
  - Sort-Object
- Procesos y datos del sistema:
  - Get-Process
  - Get-WmiObject
  - Get-Counter
- Test-Connection (ping)
- Add-Type (agrega bibliotecas de .Net)



## Bibliografía

- [http://en.wikiversity.org/wiki/Windows PowerShell](http://en.wikiversity.org/wiki/Windows_PowerShell)
- <http://powershell.org/wp/>