

# Cmdlets en Windows Power Shell

<b>1. Introducción a PowerShell</b>	<b>2</b>
<b>2. Cmdlets</b>	<b>2</b>
Estructura de un cmdlet	2
Ayuda sobre cmdlets	3
<b>3. Módulos</b>	<b>3</b>
Alias	4
<b>4. Gestión de archivos y carpetas</b>	<b>4</b>
<b>5. Tuberías y redireccionamiento</b>	<b>4</b>
<b>6. Algunos comandos básicos</b>	<b>5</b>
Where-Object	5
Select-Object	5
Sort-Object	6
Group-Object	6
<b>7. Formato de salida</b>	<b>7</b>
Format-Table	7
Format-List	8
Format-Wide	8
<b>8. Cómo procesar ficheros e importar/exportar datos</b>	<b>8</b>
Import-CSV y Export-CSV para ficheros .csv	8
Archivos de texto y Out-File	10
XML	10

## 1. Introducción a PowerShell

PowerShell es el **sustituto de la línea de comandos de Windows tradicional**, que estaba basada en el sistema operativo MS-DOS. Está **escrito con el framework .NET de Microsoft**, lo que significa que es **compatible con todos los programas y librerías escritas en .NET**, en cualquiera de sus lenguajes (C#, Visual Basic, F#, etc.). Ser compatible significa, por una parte, que **puede utilizar los objetos (clases, métodos, paquetes, etc.)** generados con .NET. Por otra parte, significa que **PS también genera objetos .NET** que se pueden **intercambiar con cualquier otro programa .NET**, lo cual tiene muchísimas implicaciones. La cantidad de software disponible para utilizar en PS es ingente y aumenta cada día con aportaciones de Microsoft y de la comunidad.

Debemos realizar una distinción entre PowerShell y PowerShell ISE. El sustituto de la línea de comandos es PS, mientras que **PS-ISE es un entorno de desarrollo** de scripts para PS. Es decir, **programamos scripts en PS-ISE, y utilizamos PS** de forma genérica para **administrar nuestro sistema**. También está bien conocer que existe una versión multiplataforma de PS, llamada **PowerShell Core**, que es capaz de ejecutarse en GNU/Linux y MacOS, con gran parte de la funcionalidad del PS completo.

Se puede iniciar PS de muchos modos, algunos de los cuales son:

- Click derecho en el logo de Windows y seleccionar la opción PowerShell.
- Tecla Win+R y escribir “PowerShell”.
- Desde el explorador de archivos, escribir “PowerShell” en el campo de dirección. Este atajo resulta muy útil porque inicia PS en la carpeta en la que nos encontramos con el explorador de archivos.

Hay que tener en cuenta que **para realizar ciertas operaciones, será necesario ejecutar PowerShell en modo administrador**. El atajo más rápido para iniciar PS como administrador es WIN+R para abrir el menú de sistema y después pulsar “A”.

## 2. Cmdlets

Una gran diferencia entre PS y otras consolas de comandos es que **la entrada y salida de datos no está basada en texto**. La comunicación con PS siempre se produce mediante **objetos**, de hecho, PS sigue el **paradigma de programación orientado a objetos**. Los “Cmdlets”, **comandos ligeros o comándulos**, son la forma de enviar este tipo de objetos a PS para interactuar con ella.

Como resultado de un cmdlet, **PS nos devolverá otro objeto**. Estos objetos pueden ser **canalizados hacia otros cmdlets** o dejar que PS nos los **muestre por pantalla**. La diferencia puede parecer sutil, pero es bastante profunda, un **cmdlet no se encarga de formatear los datos para presentarlos por pantalla**, es PS (o otro cmdlet específico) el que recibe el objeto resultado del cmdlet y lo formatea para mostrarlo por pantalla.

### Estructura de un cmdlet

Todos los cmdlet **se componen de un verbo y un sujeto separados por un guión**. De este modo, le indicamos a PS **qué queremos hacer (verbo) y con qué tipo de objeto (sujeto)**.

- ✓ Ejemplos de **verbos** son: Get, Set, Remove, New.
- ✓ Ejemplos de **sujetos**: LocalUser, Location, ChildItem, LocalGroup, NetAdapter.
- ✓ Ejemplos de **cmdlets**: Get-NetAdapter, New-LocalUser, Remove-LocalGroup.

Como se puede apreciar, la **estructura de verbo-sujeto** es muy clara. **Get-LocalUser** obtiene una lista de los usuarios locales de la máquina (o los detalles de un solo usuario si se especifica uno); **New-LocalUser** sirve para crear un nuevo usuario local y **Remove-LocalUser** sirve para borrar un usuario que ya existe.

## Ayuda sobre cmdlets

La ayuda de PS es muy detallada y resulta tremendamente útil. La primera vez que trabajemos con PS, es recomendable ejecutar el **cmdlet Update-Help** para actualizar la información de ayuda de todos los comandos disponibles. Es necesario realizar esta operación con permisos de administrador y es normal que no se pueda actualizar la información de algunos comandos.

Existe un conjunto de cmdlets que sirven específicamente para dar información sobre el resto de cmdlets. Los siguientes son especialmente útiles:

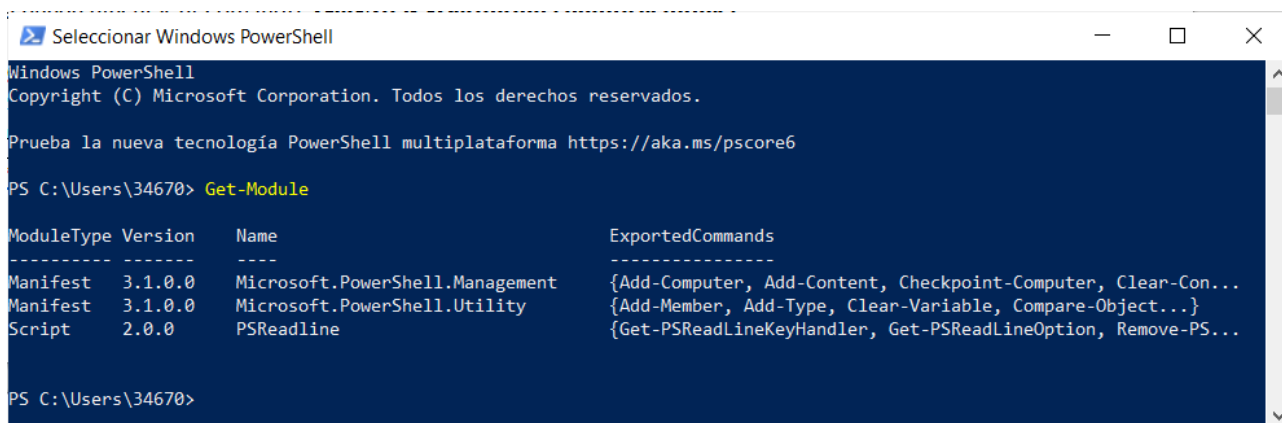
- **Get-Command -verb New**: Muestra todos los comandos para el verbo “New”.
- **Get-Command -Noun LocalGroup**: Muestra todos los comandos para el sujeto “LocalGroup”.
- **Get-Command \*LocalGroup\***: Muestra todos los comandos de cualquier verbo para cualquier sujeto que contenga “LocalGroup”.
- **Get-Help {nombreCmdLet}**: Obtiene ayuda de un comando específico. Si añadimos la opción “-examples”, nos mostrará ejemplos de uso.

Además, PS tiene una **función de autocompletado** que se activa con el **tabulador**, que nos puede ayudar mucho a escribir correctamente los cmdlets y las rutas del sistema.

## 3. Módulos

Los **cmdlets están contenidos en módulos que los agrupan en función de su naturaleza**. Para hacer la ejecución de PS más rápida, **sólo se cargan en memoria los cmdlets de algunos módulos**. En general, no será necesario cargar más módulos en memoria, pero a veces resulta imprescindible. Estos son los comandos para gestionar los módulos:

- **Get-Module**: muestra todos los módulos cargados en memoria.
- **Get-Command -module {nombre\_módulo}**: lista todos los cmdlets del módulo indicado.
- **Get-Module -ListAvailable**: muestra los módulos disponibles para cargar en memoria.
- **Import-Module {nombre\_módulo}**: carga en memoria el módulo indicado.
- **Remove-Module {nombre\_módulo}**: descarga de memoria el módulo indicado.



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

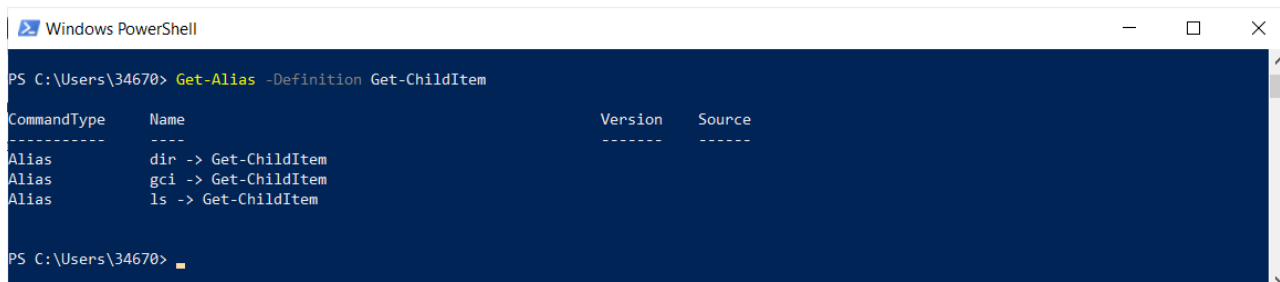
Prueba la nueva tecnología PowerShell multiplataforma https://aka.ms/pscore6

PS C:\Users\34670> Get-Module

ModuleType Version      Name                                ExportedCommands
-----
Manifest 3.1.0.0 Microsoft.PowerShell.Management {Add-Computer, Add-Content, Checkpoint-Computer, Clear-Con...
Manifest 3.1.0.0 Microsoft.PowerShell.Utility {Add-Member, Add-Type, Clear-Variable, Compare-Object...}
Script 2.0.0 PSReadline {Get-PSReadLineKeyHandler, Get-PSReadLineOption, Remove-PS...
```

## Alias

Para agilizar el uso de PS y **facilitar la adaptación de los administradores que vienen de otras consolas de comandos, existen los alias**. Los alias son **nombres alternativos de los cmdlets** y pueden usarse en lugar del nombre completo del cmdlet al que están asociados en cualquier lugar. Para listar los alias de un cmdlet se puede ejecutar el comando: **Get-Alias -Definition {nombrecmdlet}**.



```

PS C:\Users\34670> Get-Alias -Definition Get-ChildItem

CommandType      Name                                Version Source
-----
Alias             dir -> Get-ChildItem               Version Source
Alias             gci -> Get-ChildItem               Version Source
Alias             ls -> Get-ChildItem                Version Source
  
```

**Get-ChildItem** obtiene los **elementos** (archivos y directorios) **dentro del directorio actual** (o de otro si se especifica). Además, **tiene tres alias: dir** (como en MS-DOS), **ls** (como en sistemas Unix y GNU) y **gci** (acrónimo de Get-ChildItem). Sin embargo, esto no debe llevar a confusión, **que podamos invocar a Get-ChildItem utilizando “ls” no significa que podamos utilizar “ls” tal y como se utiliza en GNU**. Los parámetros y modificadores **NO** tienen alias, por lo que hay que aprender cómo funciona Get-ChildItem si queremos obtener una información distinta a la que nos da sin más parámetros (ver Get-Help Get-ChildItem -examples).

## 4. Gestión de archivos y carpetas

La siguiente tabla es un resumen de **cmdlets más comunes para realizar una gestión básica de archivos y carpetas** (en negrita el alias equivalente a GNU):

Cmdlet	Alias	Descripción	Ejemplos
<b>Get-Location</b>	gl, pwd	Muestra la ruta en la que nos encontramos	Get-Location
<b>Get-ChildItem</b>	dir, gci, ls	Lista ficheros y directorios	ls C:\ -Attributes hidden
<b>Set-Location</b>	cd, sl	Cambia de directorio	sl ..
<b>New-Item</b>	ni, md, mkdir	Crea archivos y directorios	New-Item fotos -ItemType Directory
<b>Remove-Item</b>	del, erase, rd, ri, rm, rmdir	Borra archivos y directorios	Remove-Item midirectorio -Recurse
<b>Move-Item</b>	mi, move, mv	Mueve archivos y directorios	mv -Path *.*.txt -Destination C:\Logs
<b>Rename-Item</b>	rni, ren	Renombra y mueve archivos y directorios	Rename-Item -Path "c:\logfiles\daily_file.txt" -NewName "monday_file.txt"
<b>Copy-Item</b>	copy, cp, cpi	Copia ficheros y directorios	cp hola.txt C:\ficheros
<b>Get-Content</b>	cat, gc, type	Muestra el contenido de un fichero	cat fichero.txt

## 5. Tuberías y redireccionamiento

Las tuberías y el redireccionamiento funcionan, a efectos prácticos, igual que en GNU/Linux o en Batch. A bajo nivel, la diferencia está en que los comandos de GNU/Linux pasan texto al siguiente nivel (al siguiente comando o a la siguiente salida), mientras que PowerShell siempre mueve objetos de un nivel al siguiente.

Algunos ejemplos de tuberías y de redireccionamiento:

- **Get-Command | Measure-Object**: cuenta las líneas de la respuesta de Get-Command.
- **ls -Recurse | Where-Object {\$\_.Length -gt 100 Mb} | Sort-Object -Descending -Property Length**: parece bastante complejo, pero resulta muy lógico al entender que **ls pasa un objeto de tipo lista con sus propiedades al cmdlet Where-Object** para que lo filtre y este le pasa otra lista de objetos a **Sort-Object** para que lo ordene:
  - **ls -Recurse**: lista todos los contenidos de un directorio de forma recursiva.
  - **Where-Object {\$\_.Length -gt 100 Mb}**: filtra todos los objetos de la lista cuyo atributo “Length” sea mayor que 100 Mb.
  - **Sort-Object -Descending -Property Length**: ordena todos los objetos de la lista por el campo “Length” de forma descendiente.
- **Get-NetTCPConnection | Where-Object { \$\_.State -eq ‘Established’}**: lista aquellas conexiones en estado “establecido”.
- **Get-Command -CommandType Cmdlet | Where-Object Verb -like new**: lista aquellos cmdlets que contienen “new” en su verbo.
- **Get-Command -verb new > ComandosConNew.txt**: crea un fichero de texto llamado todosLosComandosConNew.txt con la lista de todos los cmdlets que utilizan el verbo “new”.
- **“hola” >> saludos.txt**: añade una línea con el texto “hola” al fichero de texto “saludos.txt”.

## 6. Algunos comandos básicos

### Where-Object

Es el **principal comando que utilizamos para filtrar los resultados** obtenidos después de haber utilizado otro cmdlet. Básicamente, lo que se hace es **comprobar alguna de las propiedades del objeto de entrada y redirigirlo** a la salida **sólo si cumple la condición** indicada. Hasta la versión 3.0 de PowerShell, cuando se utiliza el Where-Object y tenemos como entrada **varios objetos**, podríamos decir que **los evalúa uno a uno**, y en el momento en el que se procesa ese objeto **se asigna a la variable \$\_**.

A continuación vamos a ver algunos ejemplos del uso de este comando:

- Si queremos obtener los **servicios cuyo nombre empieza por “s”** podemos hacer lo siguiente:  

```
Get-Service -Name s*
Get-Service | Where-Object { $_.Name -like "s*" }
```
- O con la **nueva nomenclatura de PowerShell**:  

```
Get-Service | Where-Object Name -like "s*"
```
- Y podemos **canalizar distintos filtros en serie**:  

```
Get-Service | Where-Object Name -like "s*" | Where-Object Status -eq "running"
```

### Select-Object

Con este cmdlet lo que podemos es **seleccionar los campos del objeto u objetos del resultado final que queremos obtener**, de forma que no tengamos el objeto con todas sus propiedades sino solo las que nosotros necesitamos. También nos permite **obtener un número concreto de objetos**, por ejemplo, los diez primeros de un resultado.

- Vemos que nos muestra una **cantidad de objetos y unas propiedades concretas**:  
`Get-Process -Name s*`
- Podemos **cambiar las propiedades que se muestran** de la siguiente manera:  
`Get-Process -Name s* | Select-Object id, Processname, Starttime -First 5`
- Podemos **eliminar resultados repetidos** con el parámetro **-Unique**:  
`Get-Process -Name s* | Select-Object ProcessName`  
`Get-Process -Name s* | Select-Object ProcessName -Unique`
- Podemos **crear propiedades personalizadas** para mostrar información a partir de otras propiedades:  
`Get-Process S* | Select-Object Processname, StartTime, @{Name="Horas en ejecucion";Expression=((Get-Date) - $_.StartTime).Hours}}`
- Si una de las **propiedades** que mostramos está **compuesta** por un grupo de objetos, **mostrar toda la información con el parámetro** y no vemos todos los servicios dependientes:  
`Get-Service Winmgmt | Select Name, DependentServices`
- Vemos que la propiedad **DependentServices** **no se muestra completa**, al contener varios objetos, pero con el parámetro **-ExpandProperty** podemos ver todo su contenido:  
`Get-Service Winmgmt | Select Name, DependentServices | Select-Object -ExpandProperty DependentServices`

## Sort-Object

Nos permite **ordenar un conjunto de objetos según la propiedad indicada**. Por defecto, los ordena de forma ascendente tanto alfabética como numéricamente.

- Podemos **indicar varias propiedades** a partir de las cuales ordenar los objetos:  
`Get-Service | Select-Object -First 20`
- Ahora los podemos **ordenar según otra propiedad**. Ordenamos los objetos por el estado:  
`Get-Service | Select-Object -First 20 | Sort-Object Status`
- Tenemos que tener cuidado con las canalizaciones ya que aquí **el orden sí afecta al resultado**:  
`Get-Service | Sort-Object Status | Select-Object -First 20`

## Group-Object

Nos permite **organizar y agrupar los objetos** basándonos en una propiedad.

- El resultado es un nuevo tipo de objeto de tipo **Microsoft.PowerShell.Commands.GroupInfo**:  
`Get-Service s*`
- Comprobamos que los objetos del resultado son de tipo **System.ServiceProcess.ServiceController**:  
`Get-Service s* | Get-Member`
- Ahora vamos a **agrupar el resultado por el estado**:  
`Get-Service s* | Group-Object Status`
- Y comprobamos que el **resultado es un tipo de objeto distinto**:  
`Get-Service s* | Group-Object Status | Get-Member`
- También podemos obtener un objeto de tipo **HashTable PowerShell**:  
`Get-Service s* | Group-Object Status -AsHashTable`

## 7. Formato de salida

En PowerShell hay varias formas de mostrar el resultado de uno o varios comandos. Cuando ejecutamos un proceso, **por defecto, la salida se muestra con una canalización al cmdlet Out-Default**. PowerShell decide cómo mostrar el resultado dependiendo del tipo de objeto resultante. Se basa en la configuración que se indica en los archivos de configuración que se encuentran en la carpeta **\$pshome**. Dependiendo de esas reglas ya definidas, el resultado se muestra de una forma u otra.

Ejemplo de salida en PowerShell donde vemos que muestra el resultado en una tabla:

```
Get-Service s* | Select-object -First 5
```

Salida en formato lista, donde el mismo conjunto de objetos se muestran de una forma diferente

```
Get-Service s* | Select-object Name, DisplayName, Status, CanStop, CanPauseAndContinue  
-First 5
```

Normalmente, para mostrar el resultado de una forma u otra, se sigue el siguiente proceso:

- 1) ¿El tipo de objeto tiene una **vista predefinida**?
  - **List**
  - **Table**
  - **Wide**
  - **Custom**
- 2) ¿Hay un **conjunto de propiedades** predefinidas?
  - **Sí**: se muestran las predefinidas.
  - **No**: se muestran todas.
- 3) ¿**Cuántas propiedades** se muestran?
  - **Cuatro o menos**: se muestra en formato de tabla.
  - **Cinco o más**: se muestra en formato de lista.

También tenemos la opción de decidir nosotros cómo queremos que sea la salida, siempre que la **indiquemos al final de la canalización de comandos**.

### Format-Table

**Muestra el resultado en formato de tabla.** Dependiendo de la cantidad de información a mostrar puede que oculte parte de ella. Permite personalizar la vista de distintas formas.

Seleccionando las propiedades a mostrar, por ejemplo

```
Get-Service s* | Format-Table Name, DisplayName, Status, CanStop, CanPauseAndContinue
```

Creando propiedades personalizadas

```
Get-Service s* | Sort-Object Status | Format-Table Name, DisplayName,  
@{Name="Estado";Expression={if ($_.status -eq "running") {"Ejecutando"} else {"Parado"}}}
```

Agrupando el resultado por una propiedad

```
Get-Service s* | Sort-Object Status | Format-table Name, DisplayName, Status -GroupBy  
Status
```

Ajustar el contenido al tamaño de la vista (Wrap y Autosize) para intentar mostrar todo el contenido

```
Get-Service s* | Sort-Object Status | Format-Table Name, DisplayName,  
@{Name="Estado";Expression={if ($_.status -eq "running") {"Ejecutando"} else {"Parado"}}}  
-Wrap
```

```
Get-Service s* | Sort-Object Status | Format-Table Name, DisplayName,  
@{Name="Estado";Expression={if ($_.status -eq "running") {"Ejecutando"} else {"Parado"}}}  
-Autosize
```

## Format-List

En este caso **muestra la información en formato lista**.

Seleccionar las propiedades a mostrar

```
Get-Service s* | Sort-Object -First 5 | Format-List Name, DisplayName, Status, CanStop,  
CanPauseAndContinue
```

Crear propiedades personalizadas

```
Get-Service s* | Sort-Object Status | Sort-Object Status | Format-List Name, DisplayName,  
@{Name="Estado";Expression={if ($_.status -eq "running") {"Ejecutando"} else  
{"Parado"}}} -Autosize
```

Agrupar los datos

```
Get-Service s* | Sort-Object Status | Format-Table Name, DisplayName,  
@{Name="Estado";Expression={if ($_.status -eq "running") {"Ejecutando"} else  
{"Parado"}}} -Autosize
```

## Format-Wide

**Permite mostrar la información en una o varias columnas** pero únicamente de una de las propiedades de los objetos. En este caso podemos:

Seleccionar la propiedad a mostrar

```
Get-Service s* | Sort-Object Status | Format-Wide DisplayName
```

Seleccionar el número de columnas

```
Get-Service s* | Sort-Object Status | Format-Wide DisplayName -Column 3
```

Agrupar los datos por una propiedad

```
Get-Service s* | Sort-Object Status | Format-Wide DisplayName -GroupBy Status
```

## 8. Cómo procesar ficheros e importar/exportar datos

### Import-CSV y Export-CSV para ficheros .csv

Los **ficheros .csv son ficheros de texto con una estructura muy sencilla** además de ser uno de los formatos más extendidos y sencillos a la hora de trabajar con archivos de texto. Son similares a una hoja de cálculo (de hecho se pueden abrir con Excel o LibreOffice Calc) en los que **cada línea del fichero representa un dato** (un usuario, un grupo, una unidad organizativa, etc.).

A su vez, **cada línea tiene varios campos separados por uno o más caracteres llamados “separadores”**, habitualmente una coma. Por ejemplo, para cada usuario, tendremos campos con el usuario, el teléfono, el e-mail, etc. Generalmente, la primera línea del fichero se utiliza a modo de “cabecera” y, en lugar de contener datos, contiene los nombres de los campos.

Disponemos del cmdlet **Import-CSV**, que permite importar cualquier archivo CSV. **Por defecto, el delimitador es la coma**, pero se puede indicar otro. **La cabecera del archivo pasan a ser los nombres de las propiedades pero también se pueden indicar otros**. Todos los valores son importados como cadenas de texto salvo, que PowerShell identifique el contenido como uno de los objetos ya definidos.



Por ejemplo, disponemos de un csv que almacena una agenda de teléfonos:

```
1. nombre;apellido;telefono
2. raúl;marín;965377541
3. rosa;aravid;965390633
4. pedro;ximenez;965388596
```

Lo que hace este cmdlet es **devolver una lista de objetos**. A cada objeto, le asocia las propiedades que aparecen en la cabecera (en la primera línea). Así, Import-Csv devolvería la siguiente lista:

nombre	apellido	telefono
raúl	marín	965377541
rosa	aravid	965390633
pedro	ximenez	965388596

Desde un script, podríamos **procesar los datos del siguiente modo**:

```
$contactos = Import-Csv -Path agenda.csv
foreach($contacto in $contactos) {
    Write-Host "Contacto en agenda con nombre: $contacto.nombre
    apellido: $contacto.apellido y teléfono $contacto.telefono."
}
```

Si el fichero **no tuviese cabecera**, es decir, que la **primera línea** del fichero .csv ya fuesen **datos válidos**, **habría que especificar la cabecera al cmdlet Import-Csv**:

```
Import-Csv -Path agenda.csv -Header nombre, apellido, telefono
```

**Export-CSV** permite **crear archivos CSV a partir de un objeto o conjunto de objetos**. Hay que tener en cuenta que **las propiedades del objeto se convierten en la cabecera del archivo CSV** y que los arrays y objetos no se exportan, por lo que puede que no obtengamos la salida deseada.

También indicar que **como primera línea del archivo CSV se añade el tipo de objeto que se ha exportado como comentario**. Se puede **eliminar con el parámetro NoTypeInfo**.

Los cmdlets **ConvertTo-CSV** y **ConvertFrom-CSV** permiten trabajar con formato CSV pero sin la necesidad de utilizar archivos, como sí necesitamos con Import-CSV y Export-CSV.

## Archivos de texto y Out-File

La forma más sencilla de importar y exportar datos, utilizando archivos de texto planos. Podemos crear un archivo de texto con el resultado con varias opciones:

- Utilizando el cmdlet **Out-File**.
- **Redirigiendo la salida con >** (o >> si no queremos que sobrescriba la información).
- **Canalizando** la salida a un archivo.

Para obtener la información de archivos de texto, utilizamos el **cmdlet Get-Content**. Como opciones, podemos **leer el principio o el final** del archivo (**-Head** y **-Tail**).

```
Get-Content .\servicios.txt -Head 5
```

También podemos procesar el contenido de forma que si le indicamos un carácter delimitador, se separará cada campo en distintas líneas con la opción **-Delimiter**.

```
Get-Content .\servicios.csv -Delimiter ", "
```

## XML

XML es mejor elección cuando se necesita trabajar con **objetos más complejos**. PowerShell soporta la librería XML de .NET. **Es más útil si queremos compartir la información con terceras aplicaciones.**

Los comandos disponibles son similares a CSV:

- **Export-Clixml:** permite exportar los objetos a un archivo XML. A diferencia de CSV, cuando exportamos a XML se encarga de procesar y mostrar toda la información de los objetos
- **Import-Clixml:** permite exportar los objetos a un archivo XML
- **ConvertTo-Clixml:** permite trabajar con el formato XML sin la necesidad de utilizar archivos.