



UNIVERSIDAD NACIONAL DE LA MATANZA

Departamento de Ingeniería e Investigaciones Tecnológicas

Sistemas Operativos (Plan 2009)

Jefe de Cátedra: Fabio E. Rivalta

Equipo de Docentes: Boettner F., Catalano L., de Lizarralde R, Villamayor A.

Auxiliares docentes: Loiacono F., Hirschfeldt D., Piubel F., Rodriguez A., Segura L., Fernandez Piñeiro, Radice A.

SCRIPTING -> POWERSHELL

SISTEMAS OPERATIVOS UNLAM

OBJETIVOS

- Poder definir PS
- Saber cuando usarlo
- Poder construir un script



ROADMAP

- Definimos PS y .NET
- Vemos características de .NET y PS
- Estudiamos algunos componentes de PS
- Sacamos conclusiones



INTRODUCCIÓN

QUÉ ES POWERSHELL?

Lenguaje de Scripting (Dinámico) caracterizado por ser
Orientado a Objetos

EN DONDE APRENDO?

APUNTE OFICIAL DE LA CATEDRA

<http://www.sisop.com.ar/files/catedra/apuntes/ps/PowerShell.pdf>

Microsoft | PowerShell Introducción Docs Galería Comunidad Todo Microsoft 

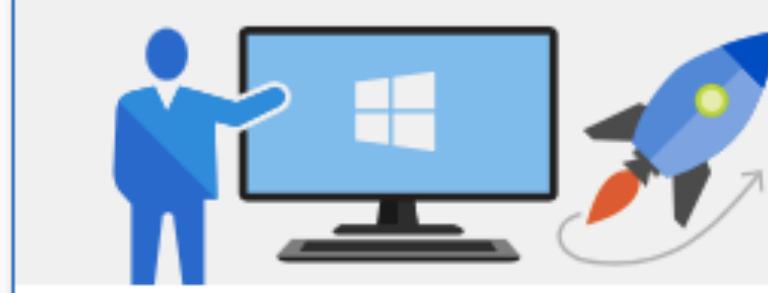
PowerShell Documentation

Get Started

Features

Reference

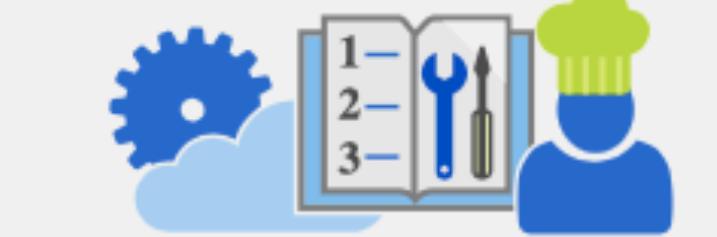
Community



Get started with PowerShell
Learn how to use PowerShell.



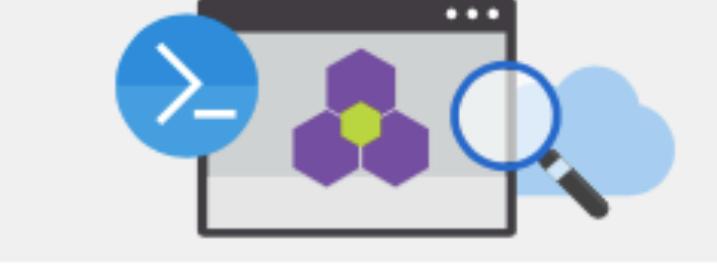
Setup and installation
Get PowerShell installed in your environment.



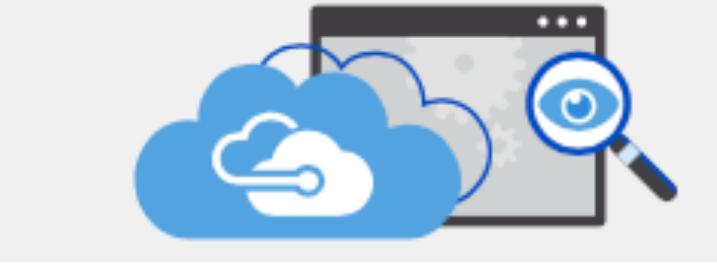
Tutorials
A cookbook of common scripting tasks.



PowerShell on GitHub
PowerShell is an open-source project and available for Windows, Linux, and macOS.



PowerShell Module Browser
Search for PowerShell modules and cmdlets.



PowerShell in Azure Cloud Shell
PowerShell in Azure Cloud Shell is now available in public preview. Learn more!

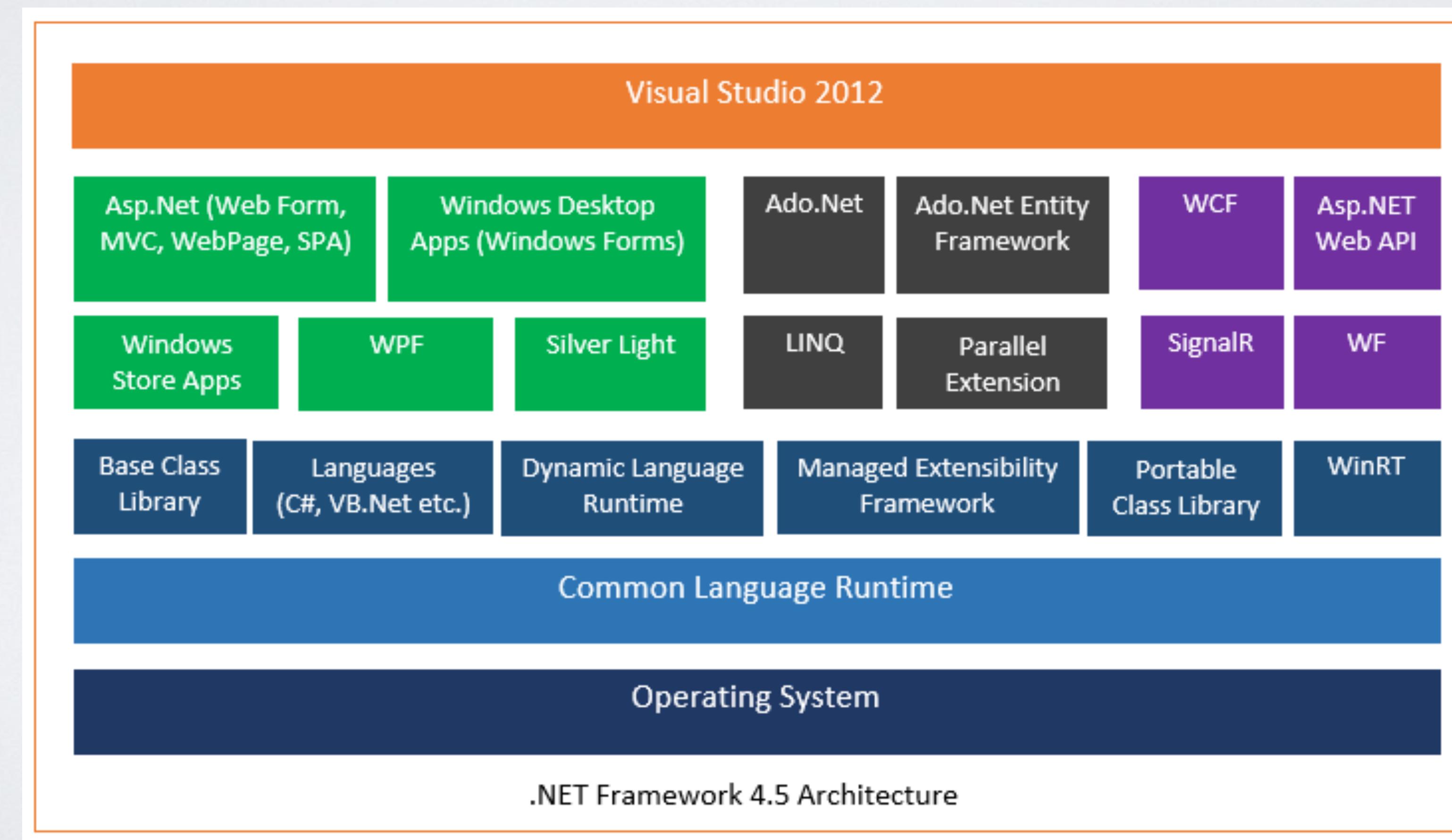
<https://docs.microsoft.com/es-es/>

EN QUE SE BASA POWERSHELL?

.NET

QUE ES .NET?

.NET ES UN ESTANDAR DONDE POR EJEMPLO SE DEFINE .NET FRAMEWORK



QUE NECESITO PARA EJECUTAR UN
PRODUCTO DE .NET ?

RUNTIME

QUE NECESITO PARA DESARROLLAR UN
PRODUCTO DE .NET ?

SDK

QUE ES .NET CORE?

UN “REBOOT” DE .NET

CONOCEN ALGUNA DIFERENCIA?

.NET ES MONOLITICO
.NET CORE ES MODULAR

Y ESTO QUE VENTAJA METRAE?

EJEMPLO

Tenemos un Servicio que usa Entity Framework

.NET ademas de tener Entity Framework deberíamos tener LINQ TO SQL

.NET core solo tendríamos Entity Framework

OTRO EJEMPLO INVESTIGUEN REDIS COMO SE ACOPLA

EN UN MISMO AMBIENTE
PUEDO TENER .NET Y .NET
CORE?

SI PERFECTAMENTE!

En un servidor se puede desear tener una herramienta gráfica que muestre gráficamente el
estado de un servicio

La herramienta gráfica podría desarrollarse con WPF (.NET)
El servicio podría desarrollarse sobre ASP (.NET CORE)

LO QUE HAGO EN .NET
FUNCIONA EN .NET CORE?

DEPENDE
POR EJEMPLO DEPENDE DE LAS DEPENDENCIAS QUE
GENEREMOS EN EL STACK

EN QUE VERSIÓN DE
POWERSHELL TENEMOS QUE
ENTREGAR LOS TP?

POWERSHELL 6

COMO SE MI VERSIÓN DE
POWERSHELL?

ABRO UNA TERMINAL, INICIO UNA SESIÓN DE POWERSHELL Y EJECUTO

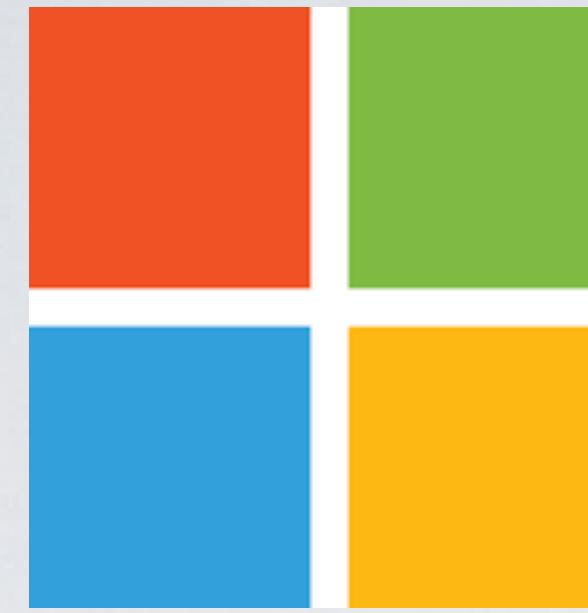
GET-HOST

```
adrianradice:~ ➜ pwsh -Command get-host

Name : ConsoleHost diapositiva
Version : 6.2.0
InstanceId : dff6fcfa6-747b-4bb2-9cb5-6bf35c12a886
UI : System.Management.Automation.Internal.Host.InternalHostUserInterface
CurrentCulture :
CurrentUICulture :
PrivateData : Microsoft.PowerShell.ConsoleHost+ConsoleColorProxy
DebuggerEnabled : True
IsRunspacePushed : False
Runspace : System.Management.Automation.Runspaces.LocalRunspace

22
adrianradice:~ ➜
```

COMO ABRO POWERSHELL?



EN WINDOWS

- DESDE EL MENU INICIO BUSCAMOS POWERSHELL
- BOTON DERECHO DEL MOUSE SOBRE EL BOTON INICIO Y APRETAMOS POWERSHELL
- PRESIONAMOS WIN + R y EJECUTAMOS PowerShell

DONDE ESCRIBO MIS SCRIPT?

DONDE TE SEA COMODO

POWERSHELL ISE

The screenshot shows the Windows PowerShell ISE interface. The top menu bar includes File, Edit, View, Tools, Debug, Add-ons, and Help. The toolbar contains various icons for file operations. The main window has tabs for PowerShell 1, PowerShell 2, and PowerShell 3, with PowerShell 1 selected. The code editor pane displays a script named TestISE_1.ps1:

```
1 Get-Service | gm
2
3
4
5 Get-Service *sql* |
6 where DependentServices |
7 select Name, DependentServices |
8 Format-List
9
10
```

The output pane shows the results of running the script:

```
ServiceHandle      Property   System.Runtime.InteropServices.SafeHandle ServiceHandle {get;}
ServiceName        Property   string ServiceName {get; set;}
ServicesDependedOn Property   System.ServiceProcess.ServiceController[] ServicesDependedOn {get;}
ServiceType        Property   System.ServiceProcess.ServiceType ServiceType {get;}
Site              Property   System.ComponentModel.ISite Site {get; set;}
Status            Property   System.ServiceProcess.ServiceControllerStatus Status {get;}
ToString          ScriptMethod System.Object ToString();

PS C:\> Get-Service *sql* |
where DependentServices |
select Name, DependentServices |
Format-List

Name          : MSSQL$SQLSRV2008R2
DependentServices : {SQLAgent$SQLSRV2008R2}

Name          : MSSQL$SQLSRV2012
DependentServices : {SQLAgent$SQLSRV2012}
```

To the right of the main workspace is a Commands palette titled "Commands X". It lists various PowerShell cmdlets under categories A, B, and C. The "A" category includes cmdlets like Add-BitsFile, Add-Computer, Add-Content, etc. The "B" category includes cmdlets like Backup-ASDatabase, Backup-SqlDatabase, Backup-WebConfiguration, etc. The "C" category includes cmdlets like cd., Checkpoint-Computer, Clear-Content, etc.

OTRA OPCIÓN VISUAL CODE CON LA EXTENSIÓN DE POWERSHELL (OPCIONAL)

The screenshot shows the Visual Studio Code interface with the following components:

- Explorador (Explorer) Panel:** Shows a tree view of files and folders. The current path is `Desktop/Sisop/Practicos/PowerShell`. A file named `EJ1.ps1` is selected.
- Editor Panel:** Displays the PowerShell script `EJ1.ps1` with the following content:

```
1 Param($pathsalida)
2 $existe = Test-Path $pathsalida
3 if ($existe -eq $true) {
4     $lista = Get-ChildItem -File
5     foreach ($item in $lista) {
6         Write-Host "$($item.Name) $($item.Length)"
7     }
8 } else {
9     Write-Error "El path no existe"
10 }
```
- Terminal Panel:** Shows a PowerShell session running in the background. The prompt is `PS /Users/adrianradice/Desktop/Sisop/Desktop/Sisop/Practicos/PowerShell>`.
- Bottom Status Bar:** Includes icons for file status (0 changes), Go Live, Líne 3, Col 1, Espacios: 4, UTF-8, LF, PowerShell, 6.2, and a notification bell with 2 notifications.

EMPEZEMOS

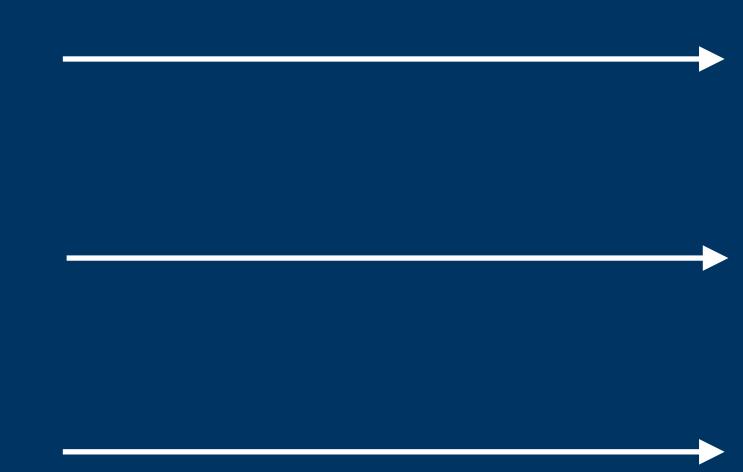
Qué podemos hacer?

Qué podemos hacer?

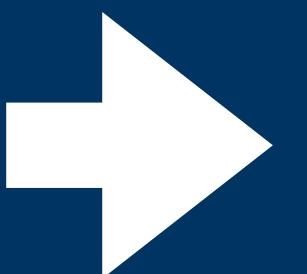
Comandos

- Podemos ejecutar

- cmdlets
- Scripts
- Binarios



```
Get-Content ps1.ps1  
./ps1.ps1  
explorer.exe
```



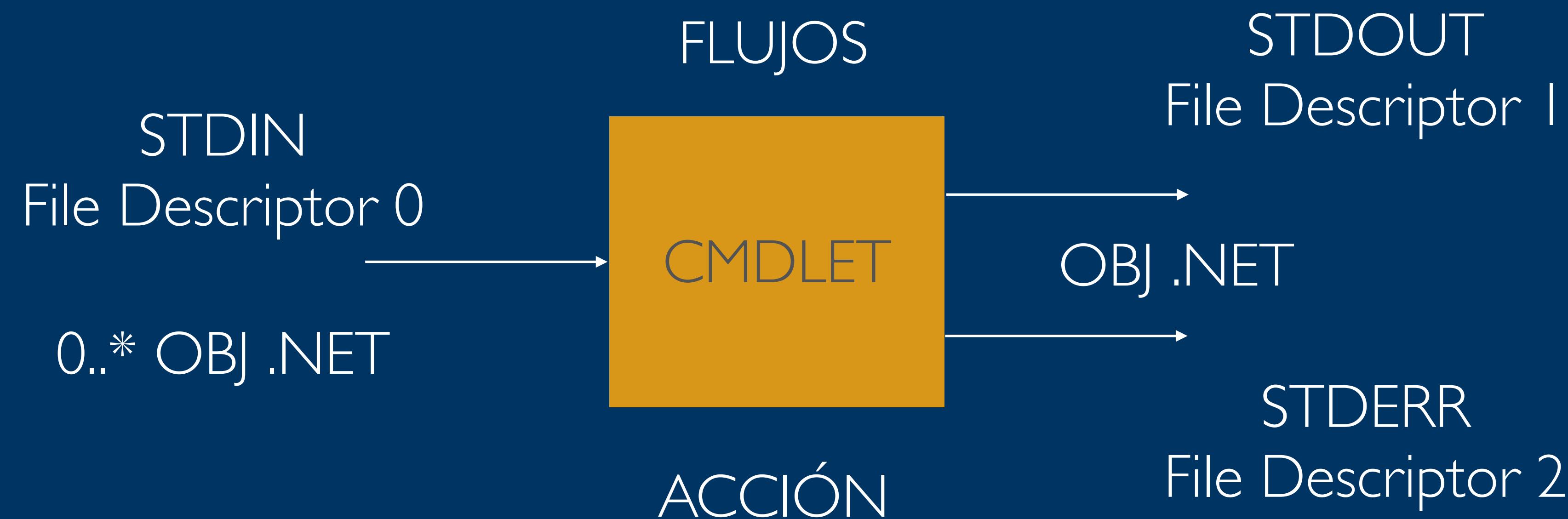
```
Write-Host "hola mundo"  
Write-Host "Esto es un script"  
hola mundo  
Esto es un script
```

GUI WIN

CMDLET

CMDLET

Qué son?



CMDLET

Características

- Son instancias de clases de .NET
- Tanto su entrada como salida son tratados como objetos
- La salida de un cmdlet es gestionada por PS
- El formato de su identificación es Verbo-Sustantivos
- A todo CMDLET se le puede asociar un ALIAS
- Se pueden ejecutar en equipos remotos, si se lo indica por parámetro por ejemplo: <cmdlet> -ComputerName <nameAD>

CMDLET

Ejemplos

- Get-Command => Listado de cmdlet
- Get-ChildItem => Obtiene el contenido de una entrada, por defecto seria el contenido de un dir.
- Get-Counter => Muestra contadores de performance
- Get-Process => Muestra los procesos del sistema
- Exit => Termina la sesión
- Get-Host => Obtiene la información del equipo en que se ejecuta
- Write-Output => Escribe una salida
- Read-Host => Obtiene una entrada
- Get-Member => Muestra las propiedades, métodos y eventos de una clase

DOCUMENTACIÓN

COMENTARIOS

Documentación

- Los comentarios son textos utilizados para documentar el script.
- Los comentarios son ignorados por el interprete de comando.
- Existen dos tipos de comentarios en PS
 - Single Line
 - Multi Line

COMENTARIO SINGLE LINE #...

Documentación

- Se indica al comienzo del comentario el carácter ‘#’, de esta manera el interprete sabrá que deberá ignorar el contenido que le sigue hasta el próximo salto de linea. Es equivalente al // de muchos otros lenguajes
- Ejemplo

```
$pathDirTrabajo = "" #Completar
$script = "$pathDirTrabajo\Saludo.ps1" #Completar
#Creamos un directorio de trabajo
New-Item -Path $pathDirTrabajo -ItemType "directory"
#Creamos nuestro primer script
New-Item -Path $script -ItemType "file"
#Escribimos nuestro script
"Write-Output """Hola Mundo""" " | Out-File $script
#Ahora Vamos a Crear un alias para invocar a nuestro Script
New-Item -Path $profile.CurrentUserAllHosts -ItemType file -Force
"New-Alias Hola $script" | Out-File $profile.CurrentUserAllHosts
#Finalmente ejecutamos mediante el alias en una nueva sesión
Hola
```

COMENTARIO MULTI LINE

Documentación

- Se indica al comienzo del comentario con ‘<#’ y indicamos el fin con ‘#>’. Es equivalente al /* */ de muchos otros lenguajes
- Ejemplo

```
<#
Comentando en
multiples
lineas
#>
```

AYUDA BASADA EN COMENTARIOS

Documentación

- Se puede definir ayuda tanto para nuestro script como para nuestras funciones.
- Get-Help identificara los bloques de ayuda siempre que se los defina como comentario y en la siguiente ubicación:
 - SCRIPT:
 - Se define al comienzo de este y seguido por dos saltos de línea si a continuación del mismo se encuentra la definición de una función.
 - También la ayuda podrá definirse al final del script únicamente si él mismo no está firmado.
 - FUNCIÓN:
 - Al final de una función antes de la llave de cierre
 - Antes de la firma de la función sin incluir ningún espacio entre medio del comentario y la firma.
 - Como primera linea de la definición de la función.
 - Las palabras claves son opcionales, pero infieren en como se consultara y mostrara la ayuda

```
<#
 .SYNOPSIS
<Breve descripción del script.>

.DESCRIPTION
<Descripción detallada del script.>

.PARAMETER <nombre parametro 1>
<Descripción del parametro 1>

.PARAMETER <nombre parametro 2>
<Descripción del parametro 2>

.PARAMETER <nombre parametro N>
<Descripción del parametro N>

.EXAMPLE
<ejemplo del uso del script>
.EXAMPLE
<ejemplo del uso del script>
#>
```

PUEDE UTILIZARSE COMENTARIO DE LINEA EN LUGAR DE BLOQUE

AYUDA BASADA EN COMENTARIOS

Documentación

- Antes de definir la firma, con el cuidado de no dejar ningún salto de línea de por medio

```
<#
.< palabra clave>
< texto de ayuda>
#>
function Get-Function
{
    <cuerpo de la funcion>
}
```

- Al final de la función antes de la llave de cierre

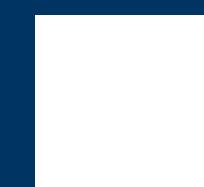
```
function Get-Function
{
    <cuerpo de la funcion>

    <#
    .< palabra clave>
    < texto de ayuda>
    #>
}
```

- Seguido de la definición de la firma de la misma

```
function Get-Function
{
    <#
    .< palabra clave>
    < texto de ayuda>
    #>

    <cuerpo de la funcion>
}
```



Para más información sobre como generar ayuda para nuestro script
`Get-Help about_Comment_Based_Help`

Espacios de memoria

Espacios de memoria

Características

- Ambito local
- Siempre precede a su nombre el signo \$
- Recordar: PS no es case sensitive, por lo tanto \$var y \$VaR no hacen ref al mismo espacio de memoria
- Puede ser dinámicas, o de tipo estático

Ejemplo variable tipo dinámico

Declaremos una variable y le asignamos un valor.

```
PS > $var = 123
```

Ahora obtenemos su tipo de datos

```
PS > $var.GetType()
IsPublic IsSerial Name          BaseType
----- -----
True      True     Int32       System.ValueType
```

Ahora le asignamos un nuevo valor y obtenemos nuevamente el tipo de datos

```
PS > $var /= 2
PS > $var.GetType()
PS > $var.GetType()
IsPublic IsSerial Name          BaseType
----- -----
True      True     Double      System.ValueType
```

Notemos como cambio de Int32 a Double

Ejemplo variable tipo estático

Declaremos una variable y le asignamos un valor.

```
PS > [Int32] $varInt = 123
```

Ahora obtenemos su tipo de datos

```
PS > $var.GetType()  
IsPublic IsSerial Name  
----- -----  
True True Int32  
  
                                     BaseType  
-----  
                                     System.ValueType
```

Ahora le asignamos un nuevo valor y obtenemos nuevamente el tipo de datos

```
PS > $varInt /= 2  
PS > $varInt.GetType()  
PS > $varInt.GetType()  
IsPublic IsSerial Name  
----- -----  
True True Int32  
  
                                     BaseType  
-----  
                                     System.ValueType
```

Notar que a diferencia de la dinámica no cambio de tipo de datos y solo guardo la parte entera de la nueva asignación.

Arreglos

Operaciones

Indices

\$Vec = @()	a	, b	, c	, d)
Index +	0	1	2	3	
Index -	-4	-3	-2	-1	

Usando los índices

- Obtener último elemento `$a[-4]`
- Obtener de átras para adelante `$a[-4..-1]`
- Extraer `b = $a[+0..1+3]`
- Obtener a b d c `$a[+0..1+-4..3]`
- Obtener b, d `$a[1,4]`
- filtrar c `$fil = $vec -ne c`

```
#Crear un arreglo vacío
$array = @()

#Crear un arreglo inicializado
$array = 1, 32, 2, "Hola"

#Crear un arreglo inicializado con secuencias
$array = @(@(a..z), @(1..4), @(2..9) )

#Borrar arreglo
$array = $null

#Vaciar arreglo
$array = @()

#Añadir elemento
$array += "Test"

#Modificar una posición
$array[2] = 2

#Contar elementos
$array.count

#Casteo fuerte
[<T>[]]$array
```

Arreglos asociativos

Arrays indexados por claves. Operaciones

Instanciar vacío

```
$ht = @{}
```

Instanciar Inicializado

```
$ht = @{38693065="Pepe" ; 56621="Tito"}
```

Acceder a un elemento

```
echo "El dni 56621 corresponde a $ht[56621]"
```

Agregar elemento

```
$ht.add(50015,"Oscar")
```

Quitar elemento

```
$ht.Remove(50015)
```

Arreglos asociativos

Ejemplo. Uso de array asociativo como contador

Contemos las letras de nuestro nombre y apellido

```
[string[]]$apyn.Replace(' ','').ToCharArray() |  
% { $contador = @{} } |  
{ $contador[$_]++ } |  
{  
    $contador.GetEnumerator() |  
    Sort-Object Value -Descending |  
    ft @{Label="Lera";Expression={$_.Key}} , @{Label="Cant Apariciones";Expression={$_.Value}}  
}
```

- Necesitamos almacenar nuestro apellido y nombre => \$apyn
- Quitamos los espacios para no contarlos => Trim()
- Obtenemos un array de char de nuestra cadena \$apyn => ToCharArray()
- Utilizamos Foreach (%) para recorrer \$apyn
- En el begin creamos el array asociativo para contar
- En el process accedemos al array por el caracter (clave) y sumamos uno. Notar que si no existe la clave (caracter) lo agrega
- En el end convertimos al ht en una enumeración, lo ordenamos en forma descendente por valor y formateamos su salida

Matriz estática

Operaciones

- Instanciar

```
$mat = New-Object '<Type>[,,,...]' <d1>,<d2>,...,<dn>
```

- Acceder a un elemento

```
$mat[$fil][$col] = 20
```

OPERADORES

OPERADORES

COMPARACIÓN

Operador	Nombre	Ejemplo	Equivalente C	Uso
-eq	Equality	2 -eq 2	==	Compara si dos objetos son iguales
-ne	Not Equal	4 -ne 2	!=	Compara si dos objetos son distintos
-lt	Less Than	1 -lt 2	<	Compara si un objeto es menor a otro
-le	Greater Than or Equal To	1 -le 5	<=	Compara si un objeto es menor igual a otro
-gt	Greater Than	2 -gt 1	>	Compara si un objeto es mayor a otro
-ge	Less Than or Equal To	2 -ge 1	>=	Compara si un objeto es mayor igual a otro

Por defecto PS al comparar no es CaseSensitive (no distingue entre mayúsculas y minúsculas), podemos modificar su comportamiento si anteponemos:

- c si queremos que sea casesensitive ej (-ceq)
- i si queremos ser explícitos que no se comporte como casesensitive (ieq)

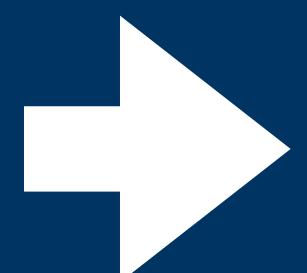
OPERADORES

Lógicos

Operador	Equivalente C	Evaluacion
-and	&&	evalúa que las dos expresiones sean verdaderas
-or		evalúa que al menos una de las dos expr sea verdadera
-xor	^	evalúa que solo una de las dos expr sea verdadera
-not	!	niega el resultado de una expresion
!	!	equivalente al -not

Recordar que estos operadores se aplican de izquierda a derecha por ejemplo

```
$a = 1  
(1 -eq 2) -and ($($a+=1) -eq 2)  
echo "a= $a"
```



Resultado de la ejecución: PS> a= 1

Notar que el operador -and como devuelve \$True solamente si las dos expresiones son verdaderas, al ser \$False la primera no continúo evaluando y se perdió el incremento de \$a.

OPERADORES

Aritmético

Operador	Nombre
+	Suma
-	Resta
*	Multiplicación
/	División
%	Resto

OPERADORES

Asignación

Operador	Uso	Ejemplo	Forma Larga
=	Asignación	\$a = 1	
+=	Sumar y Asignar resultado	\$a+=2	\$a = \$a + 2
-=	Restar y Asignar resultado	\$a-=2	\$a = \$a - 2
=	Multiplicar y Asignar resultado	\$a=2	\$a = \$a * 2
/=	Dividir y asignar resultado	\$a/=2	\$a = \$a / 2
%=	Dividir y asignar resto	\$a%=2	\$a = \$a % 2

Post-Incremento Post-Decremento

Incrementa la variable después de evaluar la expresión.

```
$a = 1  
$b = ($a++ -eq 1) -and ($a -eq 2)  
echo "b= $b a= $a"
```

PS > b= True a= 2 El anterior script se ejecutó de esta forma:

```
$a = 1  
$b = $a -eq 1  
$a = $a + 1  
$b = $a -eq 2  
echo "b= $b a= $a"
```

Pre-Incremento Pre-Decremento

Incrementa la variable la variable antes de evaluar la expresión.

```
$a = 1  
$b = (++$a -eq 1) -and ($a -eq 2)  
echo "b= $b a= $a"
```

PS > b= False a= 2 El anterior script se ejecutó de esta forma:

```
$a = 1  
$a = $a + 1  
$b = $a -eq 1 -and $a -eq 2  
echo "b= $b a= $a"
```

OPERADORES

De tipo

Operador	Nombre	Ejemplo
-Is	Comprueba si un objeto es de un cierto tipo de objeto	\$a -is [int]
-IsNot	Comprueba si un objeto no es de un cierto tipo de objeto	\$a -IsNot b.getType()
-As	Fuerza a tratar un objeto como si fuese un tipo concreto de objeto	\$a -as [float]

OPERADORES

De Rango

Permite generar un arreglo entero que almacena una secuencia con limite inf y sup

<lim_inf>..<lim_sup>

```
# Generando directorio para entregar los tp
$grupo = 1
6..1 | % { mkdir "Sisop\PS\G$Grupo\$_\Source"; mkdir "Sisop\PS\G$Grupo\$_\Lote"; }
# % -> alias de Foreach-Object
```

IF

IF SINTAXIS

```
if (condition 1) {command}
elseif (condition 2) {command}
elseif (condition 3) {command}
else {command}
```

Los else son opcionales

SWITCH

SWITCH

Sintaxis

```
switch [-regex|-wildcard|-exact] [-casesensitive] (<value>)
{
    "string|number|variable|{ expression } { statementlist }
    default { statementlist }
}
0
switch [-regex|-wildcard|-exact] [-casesensitive] -file filename
{
    "string|number|variable|{ expression } { statementlist }
    default { statementlist }
}
```

SWITCH

Ejemplo, menu operaciones

```
enum Operacion {
    PUSH
    PULL
    COMMIT
}

$acc = @([Operacion]::PULL,[Operacion]::COMMIT,'', $null, 2)

$op = switch ( $acc ) {
    ([Operacion]::COMMIT) { 'Operaciones COMMIT' ;continue; }
    ([Operacion]::PUSH)   { 'Operaciones PUSH'  }
    ([Operacion]::PULL)  { 'Operaciones PULL'  }
    ([Operacion]::COMMITE) { 'NO' }
    $null { 'No admitido'; break; }
    default {'invalido'}
}
```

SWITCH

Ejemplo, filtrar contenido por edad

```
switch ( $edad )
{
    ($PSItem -le 18)
    {
        'Contenido No Apto'
    }
    ($PSItem -gt 18)
    {
        'Contenido Apto'
    }
}
```

SWITCH

Ejemplo, menu operaciones

```
switch -Wildcard -File $path
{
    '*Error*'
    {
        Write-Error -Message $PSItem
        continue
    }
    '*Warning*'
    {
        Write-Warning -Message $PSItem
        continue
    }
    default
    {
        Write-Output $PSItem
    }
}
```

CICLOS

CICLOS

while - do while - do until

WHILE

```
while(<condicion>)
{
    <codigo>
}
```

Itera mientras la condición sea verdadera.

Cant Iteraciones 0 a N

DO WHILE

```
do
{
    <codigo>
}
while(<condicion>)
```

Itera mientras la condición sea verdadera.

Cant Iteraciones 1 a N

DO UNTIL

```
do
{
    <codigo>
}
until(<condicion>)
```

Itera mientras la condición sea falsa.

Cant Iteraciones 1 a N

CICLOS

For

```
for(<inicio>; <condicion>; <repeticion>)
{
    <codigo>
}
```

Itera mientras la condición sea verdadera. Cantidad de Iteraciones 0 a N

CICLOS

Foreach

```
foreach(item In collection)
{
    <codigo>
}
```

Recorre todos los objetos de un contenedor

CICLOS

Foreach Ejemplo

```
$alumnos = @(new [Alumno]::("X",2,7),new [Alumno]::("Y",4,7),new [Alumno]::("Z",8,7))
Write-Host "Estado de los alumnos"
foreach($alumno In $alumnos)
{
    Write-Host "$alumno.ToString() -> $alumno.condicion()"
}
Write-Host "Firma:"
```

Recorre el listado de alumnos en \$alumnos

CICLOS

Foreach-Object

```
$alumnos = @(new [Alumno]::("X",2,7),new [Alumno]::("Y",4,7),new [Alumno]::("Z",8,7))
Write-Host "Estado de los alumnos"
$alumnos | Foreach-Object
{
    #BEGIN
    Write-Host "Estado de los alumnos"
}
{
    #PROCESS
    Write-Host "$_.ToString() -> $_.condicion()"
}
{
    #END
    Write-Host "Firma:"
}
```

Recorre el listado de alumnos en \$alumnos, como foreach, pero esta vez se ejecuta como cmdlet

Control de errores

Control de errores

TRY CATCH FINALLY

Utilice el bloque Try para definir un bloque de comandos en el que desea que PS controle los errores.

Cuando se produce un error dentro del bloque Try, PS:

- Guarda el error en la variable \$Error.
- Luego busca un bloque de Catch para controlar el error.
 - Si la instrucción Try no tiene un bloque Catch coincidente, Windows PowerShell continúa buscando un bloque Catch apropiado o una declaración Trap en los ámbitos principales.
 - Finalmente se ejecuta el bloque Finally.

Si no se puede manejar el error, el error se escribe en la secuencia de error.

Un bloque catch puede incluir comandos para rastrear la falla o para recuperar el flujo esperado de la secuencia de comandos. Un bloque de catch puede especificar qué tipos de error captura.

Control de errores

Analicemos un ejemplo!

- Las dos primeras iteraciones del for (`$a= 2` y `$ a=1`) no producen ningún error así que solo se pasa por el bloque try y el finally.
- En la tercera iteración (`$a= 0`) se intenta dividir por 0 en el bloque try, por lo tanto, se produce una trap, powershell buscara el bloque catch que mejor aplique, en este caso el `DivideByZeroException`
- En la próxima iteración del while `$a` comienza valiendo 3 por lo que en el bloque try, se entra al if y aparece `throw` que burbujea una excepción, por lo tanto powershell buscara el mejor bloque catch, en este caso el que no especifica tipo, y se encuentra con un nuevo `throw` que genera la próxima excepción a capturar por el bloque catch del try exterior. Notar que no se continuó luego de la instrucción `throw` (no se ejecutó `$a=500`), pero si el finally.
- Finalmente el último bloque catch produce una nueva excepción pero esta es burbujeada y al no estar contenido el bloque de código en otro try, la misma es capturada por la trap

```
trap {"Error :(" ; continue}
$sup = 2
try
{
    #CODIGO CON PROBABILIDAD DE PRODUCIR UNA EXCEPCION
    while ( $True )
    {
        foreach ( $a in $sup..0 )
        {
            try
            {
                #CODIGO CON PROBABILIDAD DE PRODUCIR UNA EXCEPCION
                $b = 2 / $a
                if ( $a -eq 3 )
                {
                    throw "Fin"
                }
            }
            catch [ DivideByZeroException ] #Captura Error especifico
            {
                echo $_.Exception
                $sup = 3
            }
            catch #Culturara error no especificado
            {
                throw "Abortar Ciclo"
                $a = 500
            }
            finally #accion a ejecutar si o si se produzca error o no
            {
                #generalmente se coloca la liberacion de recursos
                echo $a
            }
        }
    }
    catch
    {
        echo $_.Exception
        throw "Fin del ejemplo"
    }
}
$a = 222
echo $a
```

Control de errores

CMDLET'S Para imprimir errores

CMDLET	Uso
Write-Error	Mostrar errores
Write-Warning	Mostrar advertencias
Write-Debug	Mostrar mensajes solo durante el debug

Redirección

REDIRECCIÓN

De manera predeterminada, Powershell dirige la salida de comandos a la consola de Powershell. Puede utilizar los métodos siguientes para redirigir la salida:

- Usar el cmdlet Out-*
- Usar el cmdlet Tee-Object
- Usar los operadores de redirección de Windows PowerShell.

Operadores

- > : Si el destino no es \$null y no existe lo crea y anexa la entrada, de lo contrario lo pisa con el nuevo contenido.
- >> : Si el destino no es \$null y existe agrega la entrada al final de la misma.

Tipos de salidas a redireccionar

- ○ All output
- 1 Success output
- 2 Errors
- 3 Warning messages
- 4 Verbose output
- 5 Debug messages

Pipeline (|)

En PS consiste en que los objetos devueltos como salida de la ejecución de un comando sea la entrada del siguiente de PS

Sintaxis:

```
Command-1 | Command-2
```

La salida de Command-1 es la entrada de Command-2

Ejemplo:

```
get-process paint | stop-process
```

Nota: stop-process espera como primer parámetro el proceso a finalizar, el mismo es entregado por get-process

Un uso común que se le da al pipe es al momento de leer salidas grandes usando el cmdlet `more`

```
get-command | more
```

Pipeline (|)

Condiciones a cumplir para recibir la entrada por pipe

- El parámetro debe aceptar la entrada de una canalización (no todos lo hacen)
- El parámetro debe aceptar el tipo de objeto que se envía o un tipo en el que el objeto se pueda convertir.
- El parámetro no puede estar ya en uso en el comando

El ejemplo fue posible porque stop-process admite recibir el proceso por canalización

```
get-help stop-process -Parameter InputObject
-InputObject <Process[]>
    Specifies the process objects to stop. Enter a variable that contains the objects, or type a command or expression
    that gets the objects.

    Requerido?      true
    Posición?       0
    Valor predeterminado   None
    Aceptar canalización?   True (ByValue)
    Aceptar caracteres comodín? false
```

Pipeline (|)

EJEMPLOS

Ejemplos

```
Get-Process > salida  
Get-Process $process 2>> log.txt
```

Cmdlet Tee-Object

Almacena la salida en un archivo o variable y también la envía por pipe. Ejemplo:

```
Get-Process notepad | Tee-Object -Variable proc | Select-Object processname,handles
```

Cadenas

Cadenas

Literales Comillas dobles (Débiles)

Cualquier expresión encerrada entre comillas dobles será resuelta para obtener su valor.

```
# Ejemplo del uso de comillas dobles
$var = 2
Write-Host "el valor de var ($var) multiplicada por dos es: $($var * 2)"
```

- El resultado de ejecutar el script anterior sería:
ps> el valor de var (2) + 1 es: 4

Cadenas

Literales Comillas dobles (Débiles)

No se realizan ningún tipo de sustitución:

```
# Ejemplo del uso de comillas dobles
$var = 2
Write-Host "el valor de var ($var) multiplicada por dos es: $($var * 2)"
```

- El resultado de ejecutar el script anterior sería:
ps> el valor de var (\$var) multiplicada por dos es: \$(\$var * 2)

Cadenas

Evitando algunas sustituciones en cadenas con comillas Débiles

La precedencia del carácter escape () a un token anula su interpretación

```
# Uso del character escape en una cadena entre ""  
$var = 2  
Write-Host "`$var * 2 = $($var * 2)"
```

- El resultado de ejecutar el script anterior sería:
ps> \$var * 2 = 4

Cadenas

Tipo HERE-STRING

Este tipo de cadenas se usa para escribir textos multilínea o en los que aparecen “

Su definición es encerrando la cadena entre @" y "@

```
$cadena=@"  
Hola  
"UNLAM"  
"@
```

Cadenas & Command In String

El operador & permite ejecutar el cmdlet contenido en una cadena

```
$a = "ls"  
&a
```

Cadenas

Caracteres especiales

```
`\0 => Null  
`\a => Alert bell/beep  
`\b => Backspace  
`\f => Form feed  
`\n => New line  
`\r => Carriage return  
`\r`\n => Carriage return + New line  
`\t => Horizontal tab  
`\v => Vertical tab
```

Funciones

Funciones

Introducción

- Los objetivos de las funciones en powershell son los mismos que los de cualquier lenguaje. Las funciones pueden recibir o no parámetros, y pueden no retornar nada o tener múltiples retornos

Funciones

Definición

Hay muchas formas de definir una función en powershell.
Veremos una de ellas.

- Powershell identifica una función al encontrar la palabra clave function. Luego sigue el nombre donde Generalmente sigue la reglas de verbo-sustantivo utilizado en los cmdlet de powershell, y continuación el bloque de código de la misma encerrada entre llaves.

Funciones

Definición simple

```
function <nombre de la funcion> {  
    <codigo>  
}
```

Funciones

Definición con parámetros explícita

- Opción 1

```
function <nombre de la funcion> ([ [type]$parameter1[, [type]$parameter2] ] )  
{  
}  
}
```

- Opción 2

```
function <nombre de la funcion>  
{  
    param ([type]$parameter1[, [type]$parameter2])  
}
```

Funciones

Definición con parámetros implicita

- Se recuperan los parametros de \$args

Ejemplo función con retorno múltiple y parámetros recuperados por \$args

```
Function Get-SumProd
{
    $args | % { $sum = 0; $prod = 1 }{ $sum += $_ ; $prod *= $_} { $sum , $prod }
}
```

Nota: esta función recibe n parámetros, y tiene dos retornos Esta función podemos invocarla de la siguiente manera: \$a, \$b = sumaToria 2 3 4 6

Funciones

Atributos para los parámetros

Todos los atributos son opcionales. Sin embargo, si se omite el atributo CmdletBinding, para que la función se reconozca como función avanzada, debe incluir el atributo Parameter.

Funciones

Parámetros obligatorios

Por defecto pasar un valor a un parámetro, no es obligatorio, salvo que el mismo se lo declare como mandatorio

```
Function Set-NombrePC {
    Param
    (
        [parameter(Mandatory=$true)]
        [String[]]
        $ComputerName
    )
}
```

Funciones

Parametros con posición

Si no se especifica el argumento Position, el nombre del parámetro (o una abreviatura o alias de nombre de parámetro) debe preceder al valor del parámetro cada vez que el parámetro se use en un comando. De forma predeterminada, todos los parámetros de función son posicionales. Windows PowerShell asigna números de posición a los parámetros en el orden en que los parámetros se declaran en la función.

```
Function Set-NombrePC {
    Param
    (
        [parameter(Mandatory=$true)]
        [parameter(Position=0)]
        [String[]]
        $ComputerName,
        [parameter(Mandatory=$true)]
        [parameter(Position=1)]
        [bool]
        $validar
    )
}
```

Funciones

Parametros Grupos

Podemos formar grupos de parámetros, para definir distintos comportamientos según al grupo al cual pertenezcan

```
Function Set-NombrePC {
    Param
        (
            [parameter(Mandatory=$true,
                       ParameterSetName="Computer")]
            [String[]]
            $ComputerName,

            [parameter(Mandatory=$true,
                       ParameterSetName="User")]
            [String[]]
            $UserName,

            [parameter(Mandatory=$false, ParameterSetName="Computer")]
            [parameter(Mandatory=$true, ParameterSetName="User")]
            [Switch]
            $Validar
        )
}
```

Funciones

Algunas validaciones para parámetros. CADENAS VACIAS

El atributo AllowEmptyString permite que el valor de un parámetro obligatorio sea una cadena vacía (""). En el ejemplo siguiente se declara un parámetro ComputerName que puede tener un valor de cadena vacía.

```
Param
(
    [parameter(Mandatory=$true)]
    [AllowEmptyString()]
    [String]
    $ComputerName
)
```

Funciones

Algunas validaciones para parámetros. CADENAS VACIAS

El atributo `ValidateNotNullOrEmpty` especifica que el valor de parámetro no puede ser null (`$null`) y no puede ser una cadena vacía (""). Windows PowerShell genera un error si el parámetro se usa en una llamada de función, pero su valor es null, una cadena vacía o una matriz vacía.

```
Param  
(  
    [parameter(Mandatory=$true)]  
    [ValidateNotNullOrEmpty()]  
    [String[]]  
    $ComputerName  
)
```

Funciones

Algunas validaciones para parámetros. Regex

El atributo ValidatePattern especifica una expresión regular que se compara con el parámetro o el valor de la variable. Windows PowerShell genera un error si el valor no coincide con el patrón de la expresión regular.

```
Param  
(  
    [parameter(Mandatory=$true)]  
    [ValidatePattern("[0-9][0-9][0-9][0-9]")]  
    [String[]]  
    $ComputerName  
)
```

Funciones

Algunas validaciones para parámetros. RANGO

```
Param
(
    [parameter(Mandatory=$true)]
    [ValidateRange(0,10)]
    [Int]
    $Attempts,
    [parameter(Mandatory=$true)]
    [ValidateSet("rojo", "azul", "blanco")]
    [String[]]
    $colores
)
```

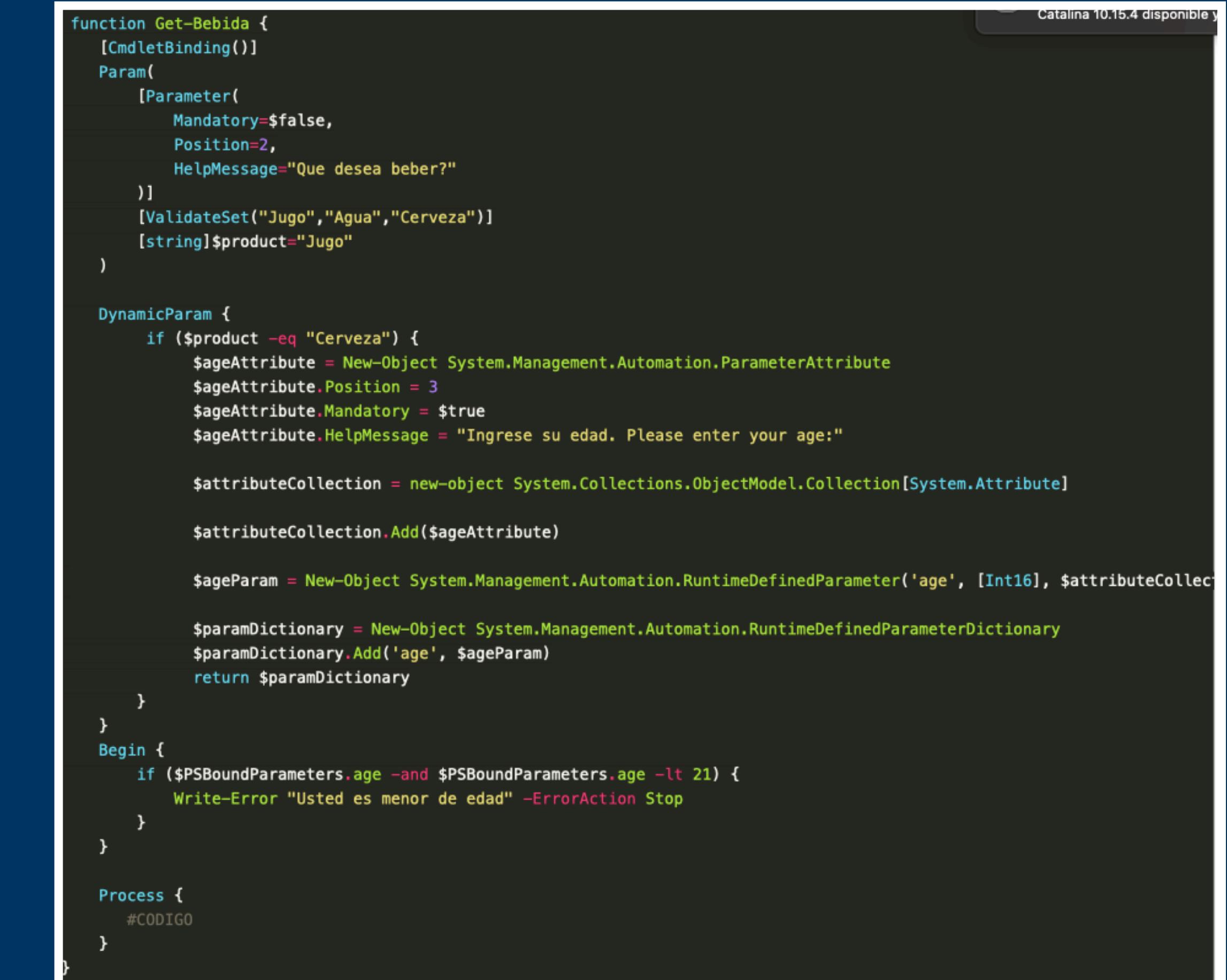
Funciones

Parámetros dinámicos

Los parámetros dinámicos son parámetros de un cmdlet, función o script que solo están disponibles en ciertas condiciones.

Por ejemplo, varios cmdlets de proveedor tienen parámetros que solo están disponibles cuando el cmdlet se usa en la unidad del proveedor, o en una ruta de acceso concreta de la unidad del proveedor. Por ejemplo, el parámetro Encoding está disponible en los cmdlets Add-Content, Get-Content y Set-Content únicamente cuando se usa en una unidad del sistema de archivos.

También puede crear un parámetro que solo aparezca cuando se usa otro parámetro en el comando de función o cuando otro parámetro tiene un valor determinado.



A screenshot of a PowerShell window titled "Catalina 10.15.4 disponible". The window contains the following PowerShell script:

```
function Get-Bebida {
    [CmdletBinding()]
    Param(
        [Parameter(
            Mandatory=$false,
            Position=2,
            HelpMessage="Que desea beber?")
    )
    [ValidateSet("Jugo","Agua","Cerveza")]
    [string]$product="Jugo"
)

DynamicParam {
    if ($product -eq "Cerveza") {
        $ageAttribute = New-Object System.Management.Automation.ParameterAttribute
        $ageAttribute.Position = 3
        $ageAttribute.Mandatory = $true
        $ageAttribute.HelpMessage = "Ingrese su edad. Please enter your age:"}

        $attributeCollection = new-object System.Collections.ObjectModel.Collection[System.Attribute]
        $attributeCollection.Add($ageAttribute)

        $ageParam = New-Object System.Management.Automation.RuntimeDefinedParameter('age', [Int16], $attributeCollection)
        $paramDictionary = New-Object System.Management.Automation.RuntimeDefinedParameterDictionary
        $paramDictionary.Add('age', $ageParam)
        return $paramDictionary
    }
}
Begin {
    if ($PSBoundParameters.age -and $PSBoundParameters.age -lt 21) {
        Write-Error "Usted es menor de edad" -ErrorAction Stop
    }
}
Process {
    #CODIGO
}
```

Funciones

Estructura requerida para una función que permite ser canalizada

Una vez al principio

Una vez al final

```
function <nombre Funcion>{
    [CmdletBinding()]
    param
    (
        [Parameter(Mandatory=$True,
        ValueFromPipeline=$True,
        ValueFromPipelineByPropertyName=$True)]
        .....
    )
    begin {
    }
    process {
    }
    end{
    }
}
```

Dependerá de la
entrada

Funciones

Bloques

Una vez al principio

```
Function sumaToria
{
begin
{
    $suma = 0
    $productoria=1
}
process
{
    $suma += $_
    $productoria *= $_
}
end
{
    $suma, $productoria
}
```

Una vez al final

Se ejecuta una vez si
la entrada no es por
PIPE

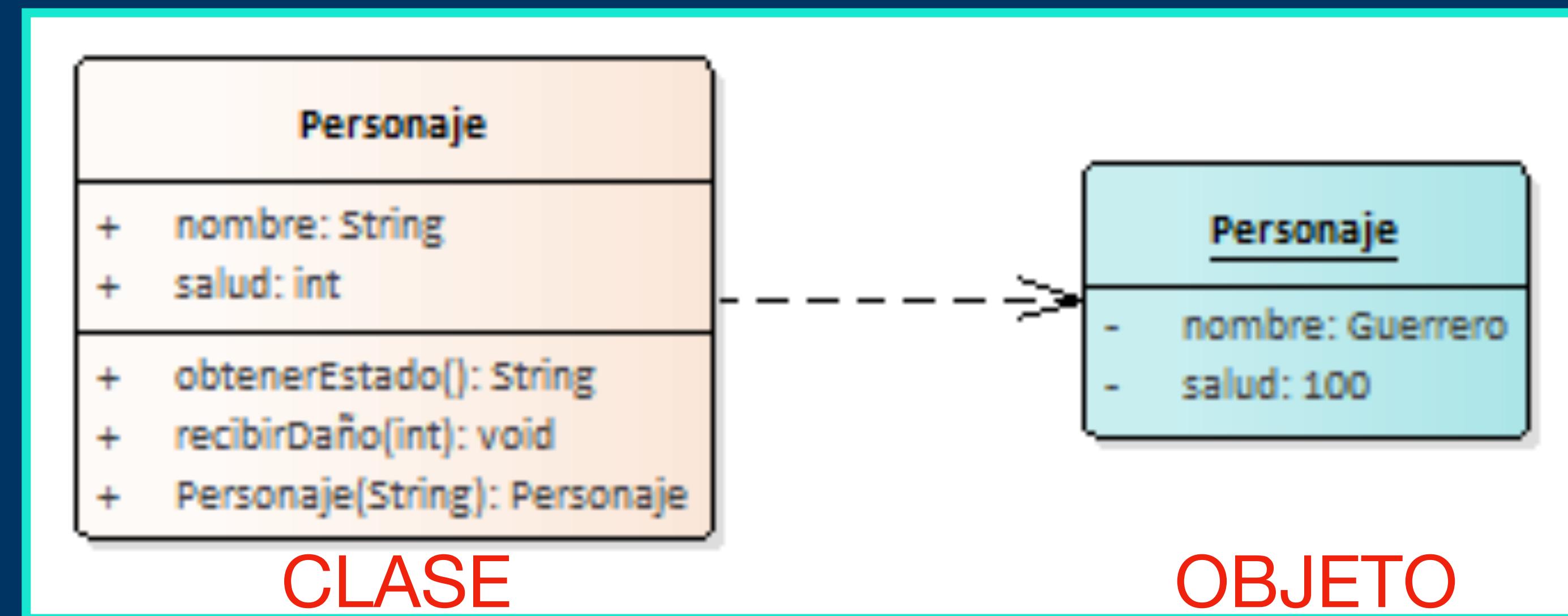
PS Y POO

PS Y POO

Repasso rápido de POO

- Clase: agrupa atributos y métodos
- Objeto: instancia de una clase

Para entender mejor, pensemos en un juego



PS Y POO

Clase

- El interprete de PS identificara una clase el encontrar el token **class**
- Salvo las clases anónimas, las clases necesitan un nombre, el mismo siempre inicia con mayúscula.
- Ejemplo: `class <Nombre_clase> {`
 `}`

PS Y POO

Propiedades de una clase

- En PS no se diferencian los conceptos de atributos a propiedades.
- Ejemplo

```
class Personaje
{
    [String] nombre
    [int] salud
}
```



El nombre y la salud serán las características que diferenciarán a los objetos de la clases

PS Y POO

Métodos

- Los métodos son las acciones que pueden realizar las instancias de nuestra clase.
- Un método posee un cuerpo y una firma
 - El cuerpo contiene las acciones
 - La firma el identificador del método (nombre más la definición de parámetros) y el tipo de retorno.

```
class Personaje
{
    [String] nombre
    [int] salud
    [String] obtenerEstado()
    {
        if ( $this.salud > 0 )
        {
            return "Vivo"
        }
        return "Muerto"
    }
    [void] recibirDanio([int] $danio)
    {
        $this.salud -= $danio
    }
}
```

Nota de color: fijarse que definimos recibir daño y no infringir, porque en realidad cada objeto sabrá cómo le impactara el daño, pensar que pasaría si cada personaje podría portar una armadura.

PS Y POO

Instanciar una clase

- Al instanciar una clase obtendremos un objeto.
- Para crear un objeto necesitamos de un constructor. Por defecto PS ofrece el constructor sin parámetros que inicia los valores de los atributos de acuerdo a lo indicado en la clase o el tipo de dato primitivo.
- \$<nombre_objeto> = [<nombre_clase>]::new(<parametros>)

Creemos un personaje (objeto)

```
$profe = [Personaje]::new()  
$profe.nombre = "Adrian"  
$profe.salud = 100
```

Ahora hagamos daño al personaje

```
$profe.recibirDanio(50)
```

Si ahora ejecutamos

```
$profe.obtenerEstado()
```

Obtendríamos como salida ps>Vivo

PS Y POO

Constructores

- Por defecto powershell define el constructor sin parámetros. En el caso de que definamos uno este será anulado, y si queremos usarlo también deberemos redefinirlo.

```
class Personaje
{
    [String] nombre
    [int] salud

    Personaje(String $nombre){
        $this.nombre = $nombre
        $this.salud = 100
    }

    [String] obtenerEstado()
    {
        if ( $this.salud -gt 0 )
        {
            return "Vivo"
        }
        return "Muerto"
    }

    [void] recibirDanio([int] $danio)
    {
        $this.salud -= $danio
    }
}
```

No sería correcto definir la salud de un personaje por lo tanto deberíamos definir un constructor que reciba únicamente el nombre de nuestro personaje y cree un personaje con una salud inicial del 100%.

El operador this nos permitió distinguir entre el atributo de la clase y el parámetro

VARIABLES ESPACIALES

Variables especiales

Listado de más usuales

Variable/Cosntante	Contenido
\$\$	Ultimo Token durante la sesión
\$^	Primer Token durante la sesión
\$_	Objeto que proviene de un Pipeline
\$HOME	Variable de entorno que contiene Directorio del usuario actual
\$Args	Parámetros que se especificaron como entrada
\$?	Estado del ultimo Comando => TRUE o FALSE
\$PWD	Path actual
\$Errors	Contenedor de mensajes de errores donde el [0] es el mas reciente
\$True	Constante que representa al True
\$False	Constante que representa al False
\$Null	Constante que representa al NULL
\$PSVersionTable	Versión de PowerShell
\$OFS	(output field separator) Equivalente al ifs de linux, es el delimitador de campos

Ejemplos variables especiales

\$\$ Obtener el Último token del cmdlet previamente ejecutado en la sesión actual

```
ls | select name  
echo $$  
PS c:\ > name
```

Ejemplos variables especiales

\$^ Obtener el Primer token del cmdlet previamente ejecutado en la sesión actual

```
ls | select name  
echo $^  
PS c:\ > ls
```

Ejemplos variables especiales

\$_ Obtener el objeto que viene desde un pipeline

```
PS C:\ > 1..2 | % { mkdir "\$_\Source"; }

    Directorio: C:\1

Mode          LastWriteTime         Length Name
----          -----              -----
d-----      31/03/2018     20:08          Source

    Directorio: C:\2

Mode          LastWriteTime         Length Name
----          -----              -----
d-----      31/03/2018     20:08          Source
```

Ejemplos variables especiales

\$HOME Obtener directorio actual del usuario mediante VAR de entorno

```
echo $HOME  
PS c:\ > C:\Users\sisop
```

Ejemplos variables especiales

\$PWD obtener el path actual en la sesión

```
PS C:\> $PWD  
  
Path  
----  
C:\
```

Ejemplos variables especiales

\$? Estado del último cmdlet ejecutado => TRUE or FALSE

```
PS C:\> TEST-PATH c:\  
True  
PS C:\> $?  
True
```

Ejemplos variables especiales

\$Error, Acceder al stack de errores y obtener los más reciente

```
mkdir sisop
$Error.Clear()
mkdir sisop 2>$null # redirecciono el error para que no se imprima (2>$null)
$(2/0) 2>$null # redirecciono el error para que no se imprima (2>$null)
echo "Ultimo error $Error[0] "
echo "Ante ultimo error $Error[1] "

PS c:\ >
Ultimo error Intento de dividir por cero.
En línea: 10 Carácter: 1
+ 2/0
+ ~~~
+ CategoryInfo          : NotSpecified: () [], RuntimeException
+ FullyQualifiedErrorId : RuntimeException

Ante ultimo error mkdir : Ya existe un elemento con el nombre especificado: C:\sisop.
En línea: 6 Carácter: 1
+ mkdir sisop
+ ~~~~~
+ CategoryInfo          : ResourceExists: (C:\sisop:String) [New-Item], IOException
+ FullyQualifiedErrorId : DirectoryExist,Microsoft.PowerShell.Commands.NewItemCommand
```

Ejemplos variables especiales

\$True / \$False

```
IF ($(Test-Path Sisop) -eq $True) {  
    echo "El directorio Sisop existe"  
}
```

Ejemplos variables especiales

`$OFS => Listar números del 1 al 9 separados por guiones`

```
$antOFS = $OFS  
$OFS = "_"  
$array = 1..9  
echo "$array"  
$OFS = $antOFS
```

CSV

CSV

Ejemplo lectura CSV

Minimamente se requiere:

- un objeto en el cual recuperar el contenido del mismo
- la ubicación del csv
- el delimitador de campos

```
$csv = Import-Csv $pathCSV -Delimiter $delimCampos
```

CSV

Ejemplo guardado en CSV

```
$csv | Export-Csv $pathCSV -Delimiter $delimCampos
```

CSV

Ejemplo contar filas CSV

```
$csv = Import-Csv $pathCSV -Delimiter $delimCampos  
$csv.Count
```

CSV

Ejemplo modificar CSV

Solo implica importarlo trabajarla en memoria con la variable en la que lo recuperamos y luego exportarlo. Ejemplo

```
$csv = Import-Csv $pathCSV -Delimiter $delimCampos
if(( ($csv[$nroReg] -ne $null) ) ){
    $csv[$nroReg].Sueldo += 1000
}
$csv = Export-Csv $pathCSV -Delimiter $delimCampos
```

JOBS

Jobs

Sintaxis

Iniciar un trabajo

```
Start-Job -Name $nombre -FilePath $pathScript -ArgumentList $param1,$param2,...$paramN
```

Finalizar un trabajo

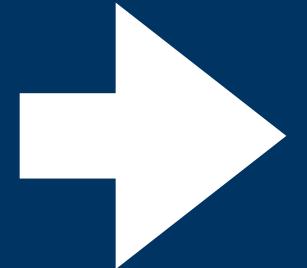
```
Stop-Job -Name $nombre
```

TIMER

TIMER

Ejemplo

```
$accion = {  
    #code  
}  
$timer = New-Object "System.Timers.Timer"  
$timer.Enabled = $true  
$timer.Interval =5000  
Register-ObjectEvent $timer Elapsed -Action $accion  
$timer.Start()
```



```
$timer.stop()  
Unregister-Event thetimer
```

Creamos un timer con un tick de 5 segundos, en el que se ejecuta el contenido de \$accion

Para detener el timer

SAQUEMOS CONCLUSIONES

GRACIAS

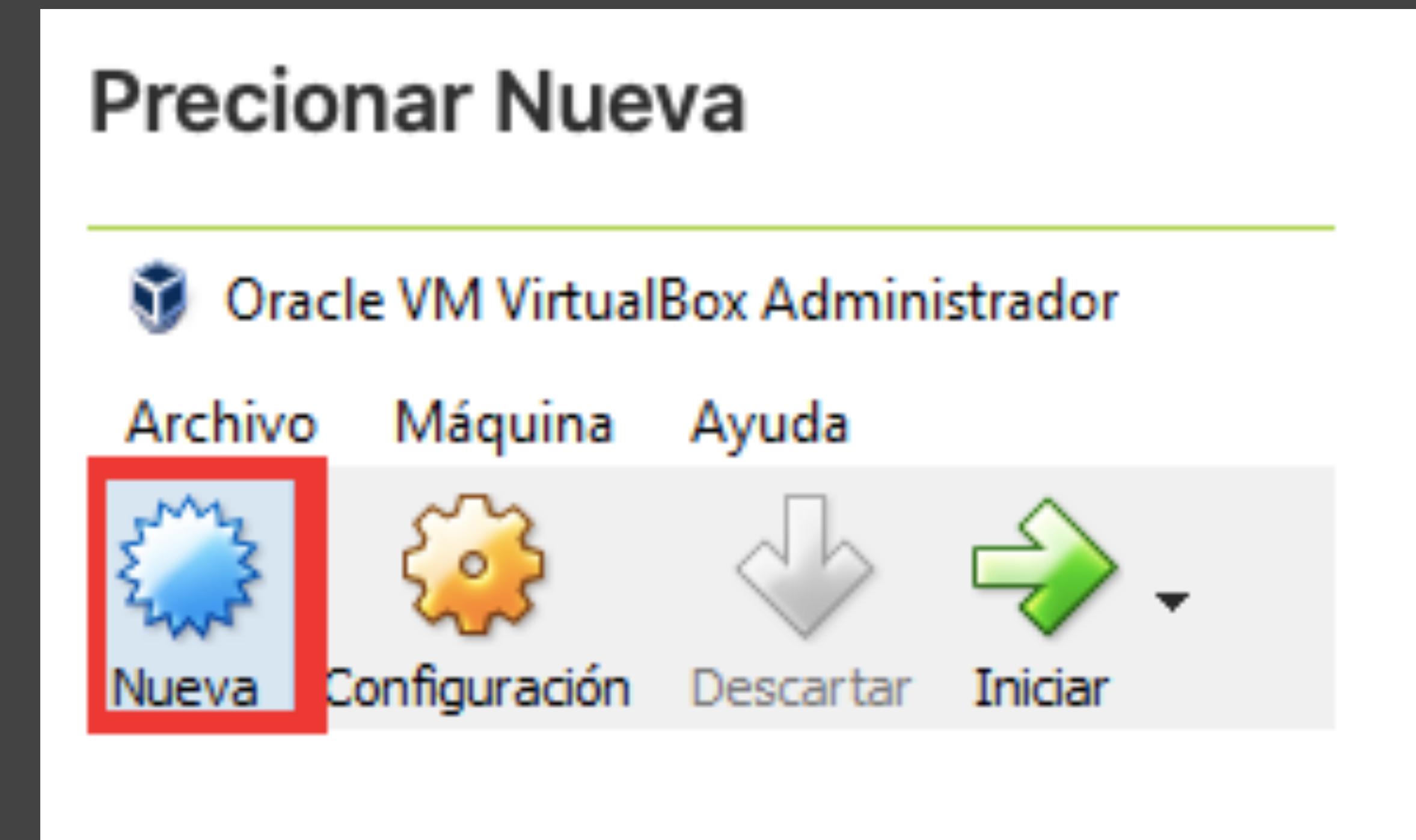
ANEXO

PREPARAR UN ENTORNO CON VIRTUAL BOX

Virtual Box (HY TIP2)

ENTORNO DESARROLLO PARA TP

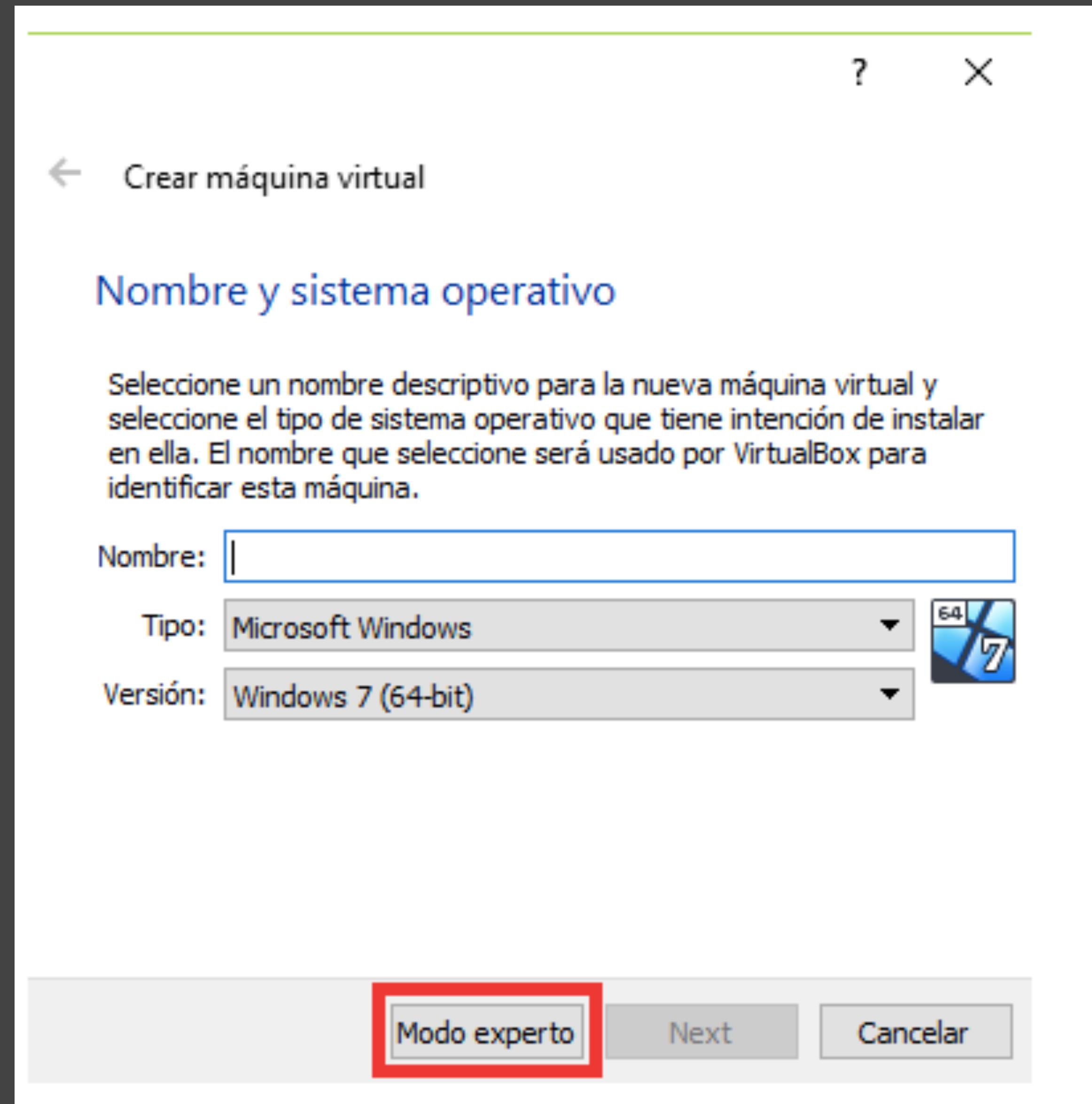
- Descargar Virtual Box
 - <https://www.virtualbox.org>
- Instalamos Virtual Box
- Abrimos Virtual Box para crear nuestra VM



Virtual Box (HY TIP2)

ENTORNO DESARROLLO PARA TP

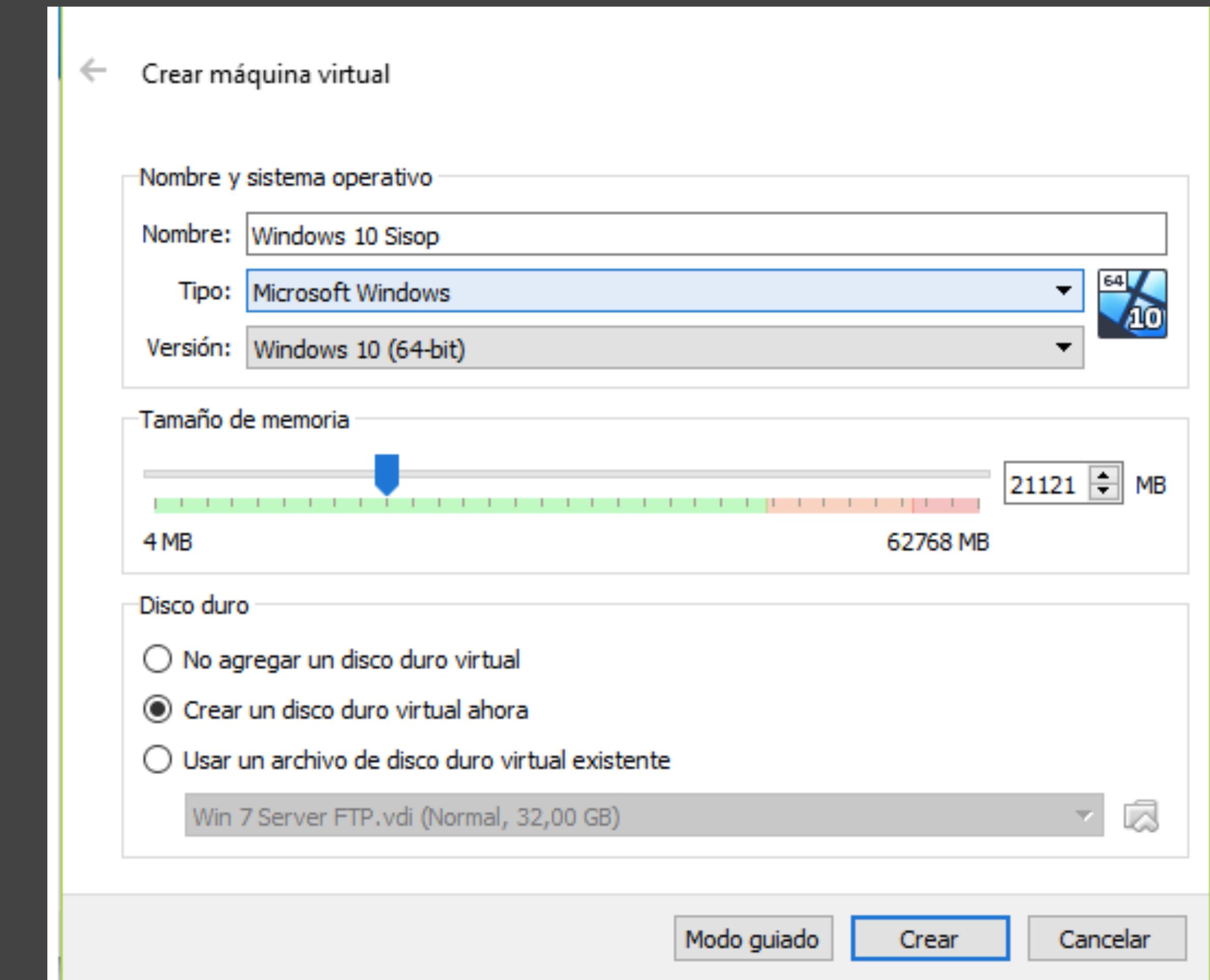
- Presionamos modo experto



Virtual Box (HY TIP2)

ENTORNO DESARROLLO PARA TP

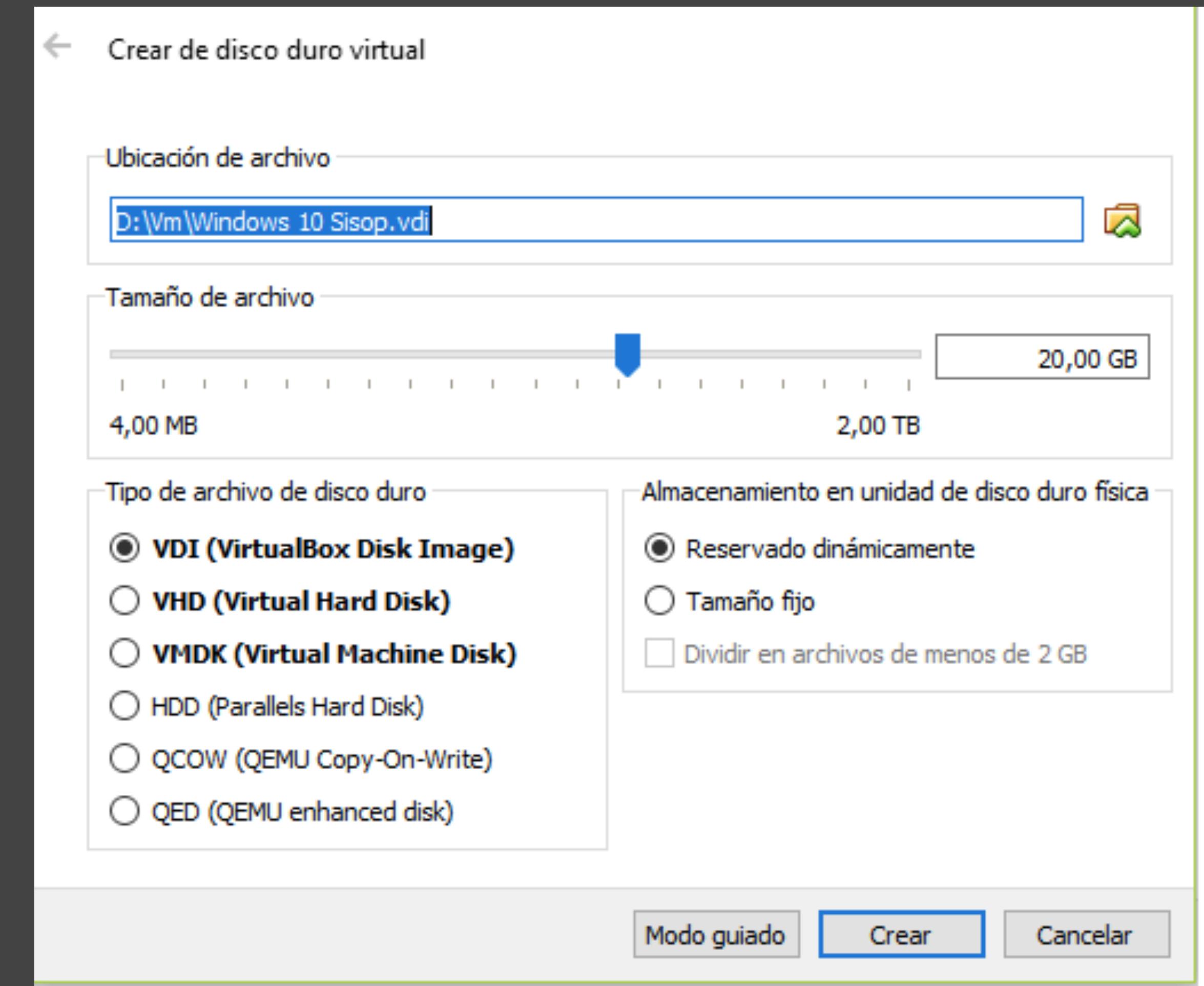
- Ingresamos un nombre para identificar la maquina virtual
- Seleccionamos el tipo (Windows) y versión del sistema operativo que instalaremos en la VM.
- Determinamos que tamaño de memoria destinaremos a nuestra VM.
- Seleccionamos Crear un disco duro virtual ahora
- Presionamos Crear



Virtual Box (HY TIP2)

ENTORNO DESARROLLO PARA TP

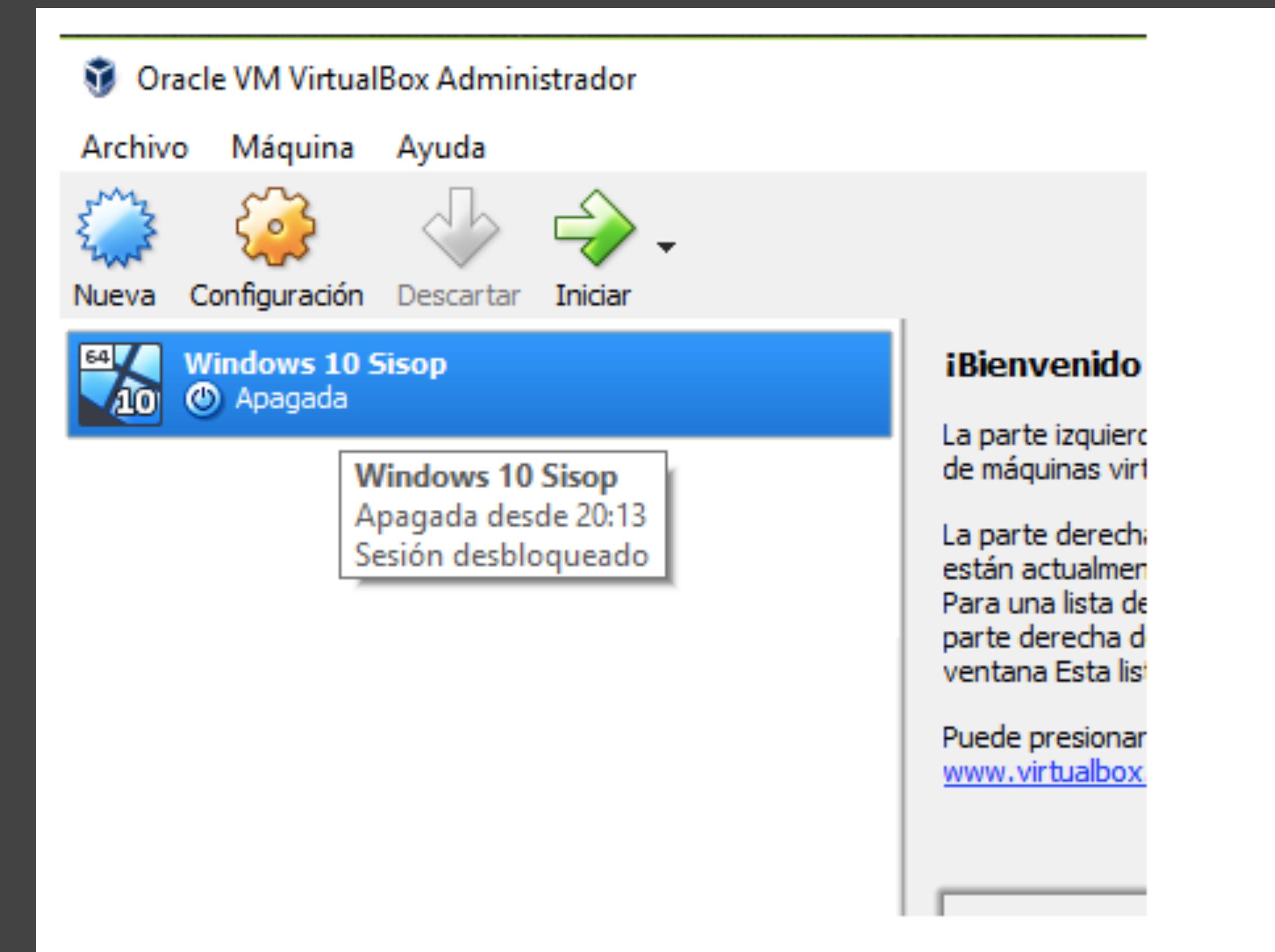
- Indicamos la ubicación para nuestro VDI.
- Indicamos el tamaño de disco que vera el sistema operativo virtualizado.
- Seleccionamos VDI como tipo de archivo.
- Seleccionamos Reservado dinámicamente.
- Apretamos crear



Virtual Box (HY TIP2)

ENTORNO DESARROLLO PARA TP

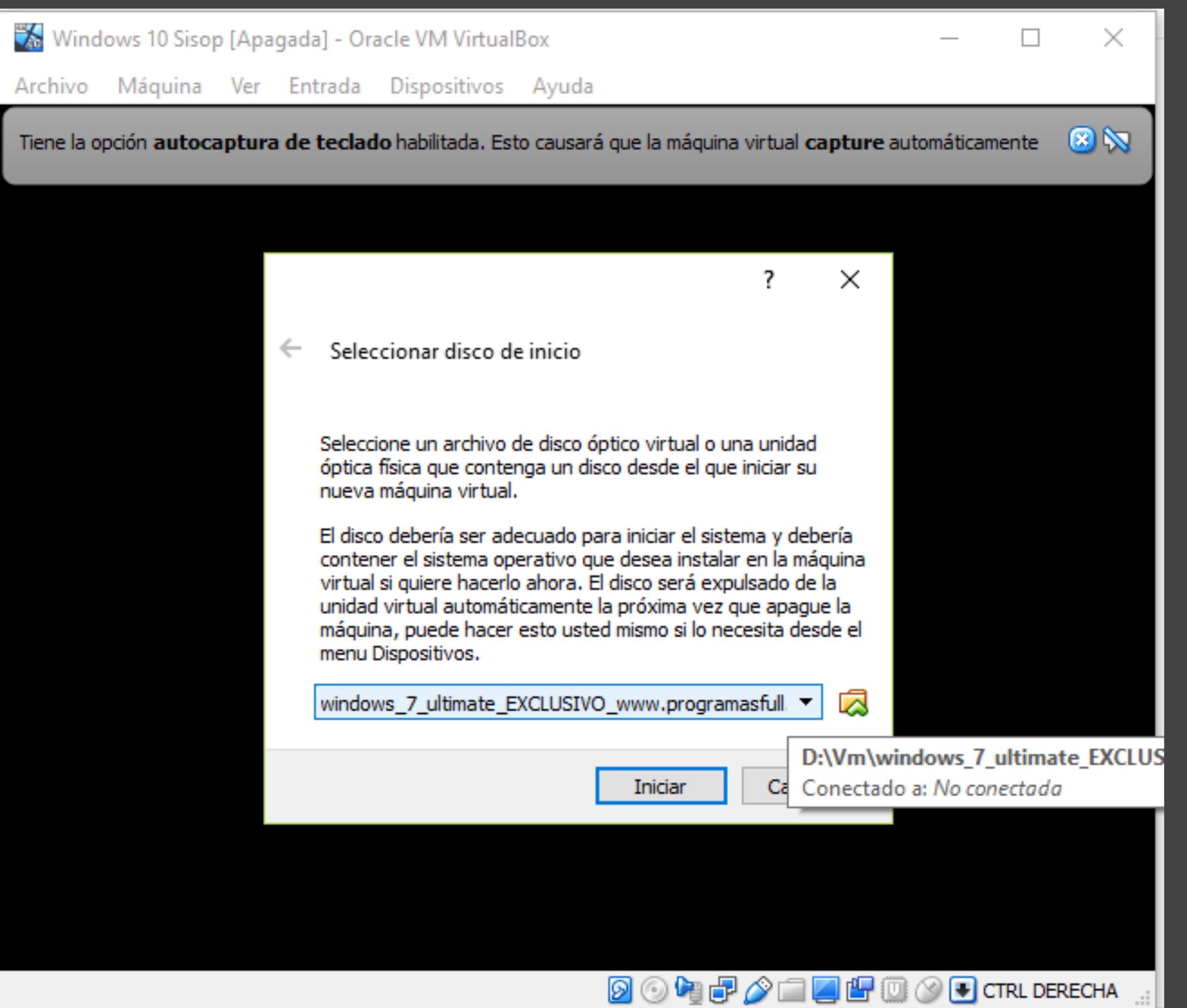
- Iniciamos nuestra máquina virtual.



Virtual Box (HY TIP2)

ENTORNO DESARROLLO PARA TP

- Seleccionamos la imagen del sistema operativo a instalar, y finalizada la instalación de la misma ya tenemos el entorno para hacer el tp.



PREPARAR UN ENTORNO CON VIRTUAL BOX + VAGRANT

Vagrant + Virtual Box

ENTORNO DESARROLLO PARA TP

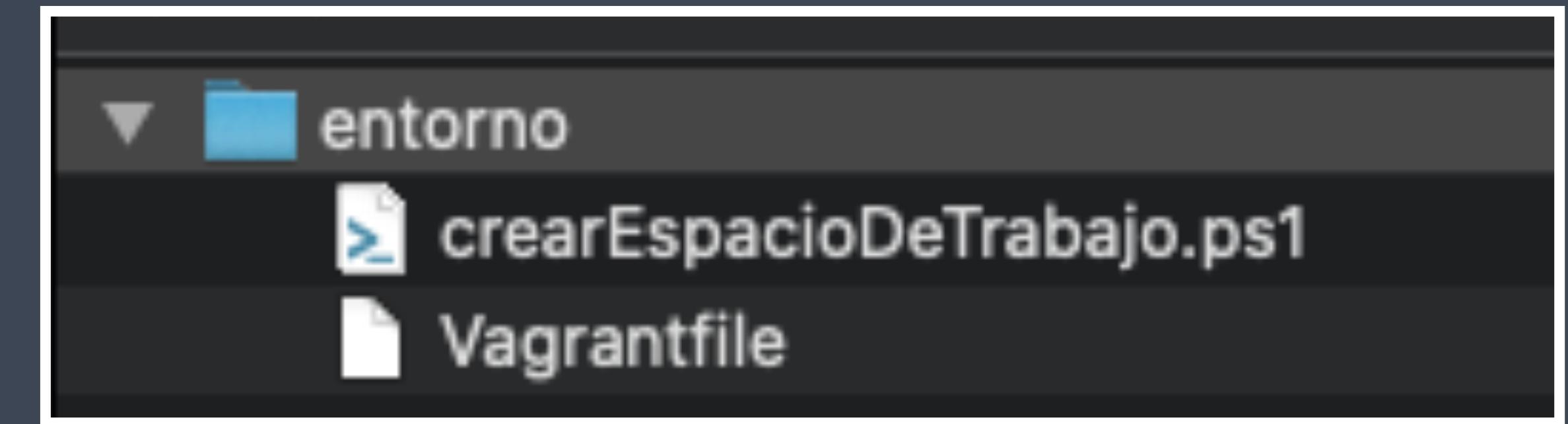
- Descargar Virtual Box
 - <https://www.virtualbox.org>
- Descargar
 - <https://www.vagrantup.com>
- Instalamos Vagrant y Virtual Box



Virtual Box + Vagrant

ENTORNO DESARROLLO PARA TP

- Creamos un directorio con dos archivos en su interior:
 - Vagrantfile
 - scripts/crearEspacioDeTrabajo.ps1



scripts/crearEspacioDeTrabajo.ps1

```
Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy Unrestricted

$grupo = 1
1..6 | % { mkdir "c:\sisop\Powershell\$grupo\EJ $_\Source"; mkdir "c:\sisop\Powershell\$grupo\EJ $_\Test"; }
```

Virtual Box + Vagrant

ENTORNO DESARROLLO PARA TP

- Mediante un script en ruby especificamos la configuración de la VM que queremos que Vagrant entregue.
- Como parte de la especificación vamos a indicar la ejecución de nuestro script luego de instalarse el os en nuestra vm

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

# Remplazar por versión de vagrant a utilizar
VAGRANTFILE_API_VERSION = "2"

# Indicamos el hypervisor
ENV['VAGRANT_DEFAULT_PROVIDER'] = 'virtualbox'

# Especificamos que queremos que realice vagrant
Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.guest = :windows
  config.vm.communicator = "winrm"
  config.vm.box = "Microsoft/EdgeOnWindows10"
  #config.vm.network "private_network", ip: "192.168.50.4"
  #config.vm.network :forwarded_port, guest: 3389, host: 3389
  config.vm.provision :shell, path: "scripts/crearEspacioDeTrabajo.ps1"
  config.vm.provider "virtualbox" do |vb|
    vb.name = "Win10 Sisop"
    vb.memory = 2096
    vb.cpus = 1
  end
end
```

Virtual Box + Vagrant

ENTORNO DESARROLLO PARA TP

- Abrimos una terminal en el directorio creado y ejecutamos vagrant up



POWERSHELL EN LINUX

INSTALAMOS PS CORE EN LINX

PS CORE + LINUX

ENTORNO DESARROLLO PARA TP

```
curl https://packages.microsoft.com/keys/microsoft.asc | sudo apt-key add -
```

```
curl https://packages.microsoft.com/config/ubuntu/16.04/prod.list | sudo tee /etc/apt/sources.list.d/microsoft.list
```

```
sudo apt-get update
```

```
sudo apt-get install -y powershell
```

#Finalizada la instalación ejecutar:

```
powershell
```