

# Utilizar los objetos .NET con PowerShell

Desde el comienzo de este libro, salvo algunas excepciones, hemos manipulado numerosos objetos que nos eran directamente accesibles sin realmente preocuparnos de sus orígenes. Pero lo que hay que saber, es que la utilización de PowerShell no se limita únicamente a la utilización de estos tipos. En efecto, PowerShell ofrece también la posibilidad de manipular otros tipos de objetos definidos en la biblioteca de clase del Framework, y esto es lo que vamos a ver en este apartado.

Antes de todo, lo que hay que saber es que con el entorno Framework .NET, todo tiene un tipo. Hasta ahora, sin prestar realmente atención, hemos manipulado numerosos objetos que poseen cada uno un tipo particular definido en la biblioteca del Framework. Por ejemplo el caso del objeto devuelto por el comando `Get-Date`.

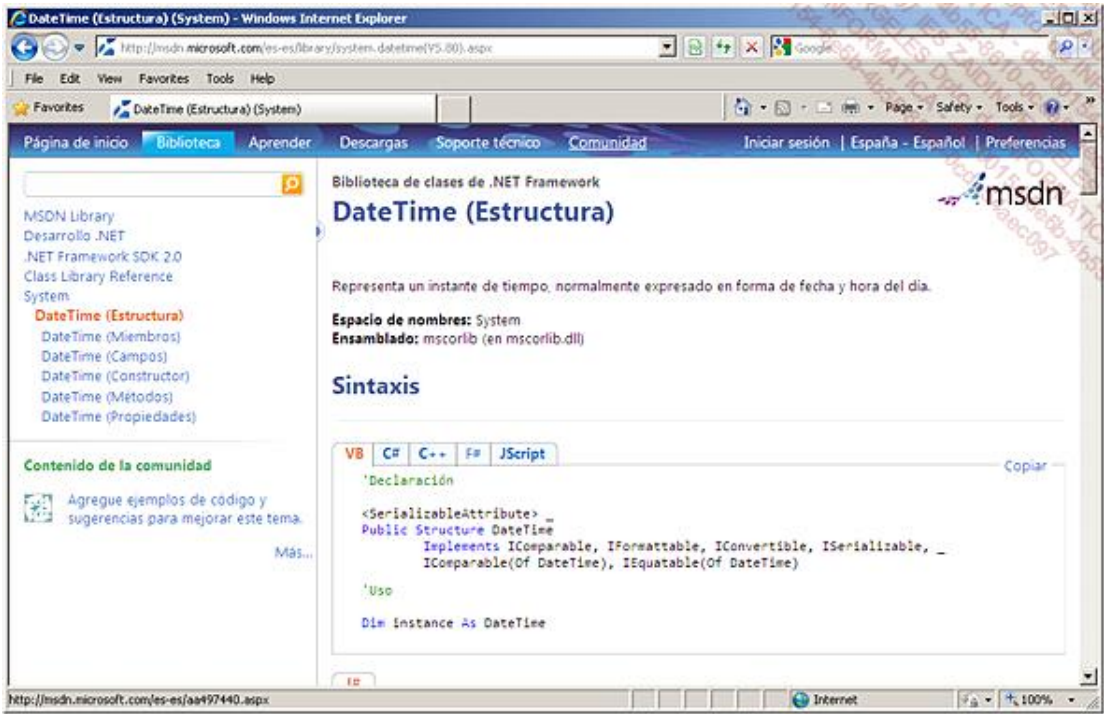
```
PS > $Date=Get-Date
PS > $Date.GetType()

IsPublic IsSerial Name                                     BaseType
-----
True     True     DateTime                                                  System.ValueType
```

Aplicando el método `GetType` a este objeto, podemos observar que el tipo utilizado es `DateTime` y que su espacio de nombres es «System». Siendo el nombre completo: `System.DateTime`.

➤ Se denomina espacio de nombres (o Namespace en inglés), lo que precede al nombre de una o varias clases .NET o WMI. Se utilizan con el propósito de organizar los objetos y de este modo evitar confundir las clases que podrían eventualmente tener el mismo nombre.

Para obtener más información, sepa que todos los tipos (tanto si se trata de clases, estructuras, enumeraciones, delegados o de interfaces) definidos en la biblioteca de clase Framework, están detallados en el web de Microsoft MSDN.



🔍 Utilizando el método personalizado `GetMsdnHelp` creado en el capítulo Control del Shell, podrá, para cada objeto, acceder directamente a la página del sitio web MSDN relacionada con el tipo de su objeto.

Al igual que los tipos vistos hasta el momento, cada tipo .NET que encuentre tendrá uno o más miembros que podrá

obtener con el commandlet **Get-Member**. Para listar los métodos y las propiedades de tipo `DateTime` teclee simplemente:  
`$date | Get-Member`.

```
PS > Get-date | Get-Member

TypeName: System.DateTime

Name            MemberType Definition
-----
Add             Method      System.DateTime Add(TimeSpanvalue)
AddDays        Method      System.DateTimeAddDaysDoublevalue)
ToFileTimeUtc  Method      System.Int64 ToFileTimeUtc()
ToLocalTime    Method      System.DateTime ToLocalTime()
ToLongDateString Method      System.String ToLongDateString()
ToLongTimeString Method      System.String ToLongTimeString()
ToOADate       Method      System.Double ToOADate()
```


Entre todos los miembros que se pueden encontrar en un tipo .NET, existen los denominados miembros estáticos. Estos miembros, aunque similares a los demás, difieren por el hecho de que deben ser llamados sin inicializar previamente la clase a la que se refieran. Por ejemplo, el tipo `DateTime` dispone según la información proporcionada por el sitio MSDN de ciertos miembros estáticos como *Now*, *Today*, *UtcNow*, etc.

Y para utilizar un método, o una propiedad «estática» de una clase del Framework, bastará con utilizar la sintaxis siguiente: `[<Espacio de nombres>.<Tipo .NET>]::<Miembro-estático>`

#### Ejemplo:

```
PS > [System.DateTime]::Now


domingo 4 octubre 2009 17:32:27
```

 Para hacer referencia a un tipo, se debe especificar su nombre entre corchetes. Ejemplo: `[System.DateTime]`


Observamos de paso, que se trata del mismo resultado devuelto por el commandlet **Get-Date**:

```
PS > Get-Date

domingo 4 octubre 2009 17:32:35
```

 Existen clases que contienen únicamente miembros estáticos. Estas clases se denominan «estáticas». Es decir que no pueden ser inicializadas con la ayuda del commandlet **New-Object**.

Para conocer los miembros estáticos contenidos en un tipo, existen dos soluciones:

- Ir a la página del sitio MSDN correspondiente al tipo deseado, y ver todos los miembros definidos con el logo  que quiere decir «static».
- Utilizar el parámetro `-static` con el commandlet **Get-Member**.

#### Ejemplo:

```
PS > [System.DateTime] | Get-Member -static

TypeName: System.DateTime
```

Name	MemberType	Definition
----	-----	-----
Compare	Method	static System.Int32 Compare(DateTime t1, DateTime t2)
DaysInMonth	Method	static System.Int32 DaysInMonth(Int32 year, Int32 month)
Equals	Method	static System.Boolean Equals (DateTime t1, DateTime t2),
FromBinary	Method	static System.DateTime FromBinary(Int64 dateData)...

## 1. Crear una instancia de tipo (Object)

Como en todos los lenguajes orientados a objetos, la creación de un objeto no es más que una instanciación del tipo. Y con PowerShell, la instanciación del tipo, que pueden ser .NET o COM (como verá en el siguiente apartado) se realiza con el commandlet `New-Object`.



Para instanciar objetos de tipo COM, el commandlet `New-Object` necesita el parámetro `-ComObject`.

Cada vez que se instancia un tipo por medio del commandlet `New-Object`, es necesario llamar a un constructor. Un constructor es un método que tiene el mismo nombre que el tipo en cuestión, y que permite generalmente la inicialización de variables. Cada tipo posee al menos un constructor. Y se habla de sobrecarga de constructor cuando existen varios constructores que utilizan diferentes argumentos. Para conocer la lista de sobrecargas de un constructor, la solución más rápida es entrar en el sitio MSDN, para ver las características del tipo en cuestión.



Como veremos un poco más adelante en este apartado, se habla de constructor por defecto cuando éste no requiere ningún parámetro para instanciar el tipo. Pero hay que tener cuidado ya que, pese a su nombre, todos los tipos no poseen un constructor así.

En la tabla de una utilización .NET, el commandlet `New-Object` posee dos parámetros (fuera de los parámetros comunes), veámoslos al detalle:

Parámetro	Descripción
<code>-typeName</code>	Especifica el nombre completo de la clase .NET, es decir, el espacio de nombres más la clase.
<code>-argumentList</code>	Especifica una lista de argumentos a pasar al constructor de la clase .NET.



Se puede omitir «System» en la descripción de los tipos que se encuentran en el espacio de nombres «System», puesto que «System» es el espacio de nombres por defecto.

### Ejemplo:

```
New-Object -typeName System.DateTime
```

es equivalente a:

```
New-Object -typeName DateTime
```

### Ejemplo:

Creación de una fecha con el objeto *.NET DateTime*.

En este ejemplo, crearemos un objeto de tipo *DateTime*. Es decir una instancia de tipo *System.DateTime*. Después verificamos su valor y su tipo (con el método *GetType*).

```
PS > $var = New-Object -typeName DateTime
PS > $var

lunes 1 enero 0001 00:00:00

PS > $var.GetType()

IsPublic IsSerial Name                                     BaseType
-----
True     True     DateTime                                     System.ValueType
```

Observe que hemos llamado al constructor por defecto ya que no hemos informado ningún parámetro. Por tanto, nuestro objeto tendrá como valor la fecha 1º enero del año 01 a las 0h00.

Probemos ahora de crear este objeto con la ayuda del constructor sobrecargado siguiente:

*DateTime (Int32, Int32, Int32)*: este constructor inicializa una nueva instancia de la estructura *DateTime* con el año, el mes y el día especificados.

Utilizando este constructor, y informando correctamente los valores como se muestra a continuación, obtendremos un objeto fecha que tendrá como valor la fecha 13 de febrero del año 2008 a las 0h00:

```
PS > $var = New-Object -typeName DateTime -argumentList 2008, 2, 13
PS > $var

miercoles 13 febrero 2008 00:00:00
```



Para todos los tipos definidos por defecto en PowerShell, no será útil la utilización del commandlet *New-Object*. Bastará con especificar el tipo entre corchetes para hacer la llamada al constructor.

### Ejemplo:

```
PS > [System.DateTime]'2/12/2008'

martes 12 febrero 2008 00:00:00
```

O

```
PS > [System.DateTime]$Date = '2/12/2008'
PS > $Date

martes 12 febrero 2008 00:00:00
```

### Ejemplo:

Creación de una cadena con el objeto *.NET String*.

He aquí un ejemplo que pone de relieve la importancia de los constructores. Vamos ahora tratar de crear un objeto de

tipo `String`. Para ello, procederemos a instanciar la clase `System.String` con la ayuda del comando `New-Object`, pero sin precisar argumentos.

He aquí el resultado:

```
PS > $Cadena = New-Object -typeName System.String

New-Object: No se encontró el constructor.
No se encuentra ningún constructor adecuado para el tipo System.String.
```

PowerShell indica que no encuentra el constructor apropiado, y que por consiguiente, no puede crear el objeto. Para instanciar esta clase, es necesario informar uno o varios argumentos al constructor.

#### Ejemplo:

```
PS > $Cadena = New-Object -typeName System.string -argumentList 'Hola'
PS > $Cadena
Hola
```

Observe que también se puede prescindir de `-typeName` y de `-argumentList` y escribir:

```
PS > $cadena = New-Object System.String 'Hola'
```



Para los que estén acostumbrados a la programación con objetos, recuerde que es posible especificar la lista de los argumentos de un constructor entre paréntesis.

#### Ejemplo:

```
PS > $Cadena = New-Object -typeName System.String -argumentList 'Hola'
```

Es equivalente a:

```
PS > $Cadena = New-Object System.String('Hola')
```

#### Ejemplo:

*Intento de creación de un Windows Form (formulario gráfico).*

Probemos ahora de instanciar el tipo `System.Windows.Forms.Form` para crear un *Windows Form*. Para ello, utilizamos una vez más el commandlet `New-Object`:

```
PS > $var = New-Object -typeName System.Windows.Forms.Form

New-Object: No se encuentra el tipo [System.Windows.Forms.Form].
Asegúrese de que está cargado el ensamblado que lo contiene.
```

Un mensaje de error de este tipo se muestra cada vez que intentamos instanciar un tipo del Framework que no está cargado vía un «assembly» en PowerShell.

## 2. Los ensamblados

Elemento esencial del Framework .NET, un ensamblado (también denominado *assembly*) puede considerarse como un conjunto de tipos .NET que constituyan una unidad lógica ejecutable por el CLR (*Common Language Runtime*) del Framework. Sin embargo, para la mayoría de usuarios PowerShell, el término de biblioteca de tipo .NET nos bastará.

Aunque puede asimilarse, en el concepto, un ensamblado no es lo mismo que una DLL (*Dynamic Link Library*), puesto que un ensamblado está compuesto por un ejecutable y varias DLLs indisociables unas de otras, así como un manifiesto garantizando las versiones correctas de las DLLs cargadas.

Elemento indispensable para el buen funcionamiento de PowerShell, algunos ensamblados del Framework se cargan desde el inicio de PowerShell, a fin de garantizar la utilización de un cierto número de tipos de objetos.

- Para conocer los ensamblados cargados por PowerShell, escriba el comando siguiente:

```
PS > [System.AppDomain]::CurrentDomain.GetAssemblies()

GAC    Version      Location
---    -
True   v2.0.50727   C:\Windows\Microsoft.NET\Framework\v2.0.50727\mscorlib.
True   v2.0.50727   C:\Windows\assembly\GAC_MSIL\Microsoft.PowerShell.Conso...
True   v2.0.50727   C:\Windows\assembly\GAC_MSIL\System.Management.Automati...
True   v2.0.50727   C:\Windows\assembly\GAC_MSIL\System\2.0.0.0__b77a5c5619...
True   v2.0.50727   C:\Windows\assembly\GAC_MSIL\System.Xml\2.0.0.0__b77a5c...
True   v2.0.50727   C:\Windows\assembly\GAC_MSIL\System.Configuration.Insta...
True   v2.0.50727   C:\Windows\assembly\GAC_MSIL\Microsoft.PowerShell.Comma...
True   v2.0.50727   C:\Windows\assembly\GAC_MSIL\System.ServiceProcess\2.0.
True   v2.0.50727   C:\Windows\assembly\GAC_MSIL\Microsoft.PowerShell.Secur...
True   v2.0.50727   C:\Windows\assembly\GAC_MSIL\Microsoft.PowerShell.Comma...
True   v2.0         C:\Windows\assembly\GAC_MSIL\Microsoft.PowerShell.Conso...
True   v2.0.50727   C:\Windows\assembly\GAC_MSIL\System.Management\2.0.0.0_
True   v2.0.50727   C:\Windows\assembly\GAC_MSIL\System.DirectoryServices\2...
True   v2.0.50727   C:\Windows\assembly\GAC_32\System.Data\2.0.0.0__b77a5c5...
True   v2.0         C:\Windows\assembly\GAC_MSIL\System.Management.Automati...
True   v2.0.50727   C:\Windows\Microsoft.NET\Framework\v2.0.50727\mscorlib.
True   v2.0         C:\Windows\assembly\GAC_MSIL\Microsoft.PowerShell.Secur...
True   v2.0         C:\Windows\assembly\GAC_MSIL\Microsoft.PowerShell.Comma...
```



Para conocer el número de ensamblados cargados, escriba el comando siguiente:

```
[System.AppDomain]::CurrentDomain.GetAssemblies().Count
```

Cada ensamblado devuelto por el comando `[System.AppDomain]::CurrentDomain.GetAssemblies()` es de tipo `System.Reflection.Assembly` y posee un conjunto de métodos, entre los cuales `GetExportedTypes` que nos permite listar todos los tipos contenidos en un ensamblado.

#### Ejemplo:

```
PS > $ensamblado = [System.AppDomain]::CurrentDomain.GetAssemblies()
PS > $ensamblado[0]

GAC    Version      Location
---    -
True   v2.0.50727   C:\Windows\Microsoft.NET\Framework\v2.0.50727\
                    mscorlib.dll

PS > $ensamblado[0].GetExportedTypes()

IsPublic IsSerial Name                                     BaseType
-----
-----
```

True	True	Boolean	System.ValueType
True	False	Buffer	System.Object
True	True	Byte	System.ValueType
True	True	CannotUnloadAppDomainException	System.SystemException
True	True	Char	System.ValueType
True	True	CharEnumerator	System.Object
True	False	Console	System.Object
True	True	ConsoleColor	System.Enum
...			



Para conocer los detalles de todos los ensamblados cargados, use el comando siguiente:

```
[System.AppDomain]::CurrentDomain.GetAssemblies() | format-list *
```

He aquí el detalle del primer ensamblado cargado:

```
CodeBase          : file:///C:/Windows/Microsoft.NET/Framework/
v2.0.50727/mscorlib.dll
EscapedCodeBase    : file:///C:/Windows/Microsoft.NET/Framework/
v2.0.50727/mscorlib.dll
FullName           : mscorlib, Version=2.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089
EntryPoint         :
Evidence           : {<System.Security.Policy.Zone version="1">
                     <Zone>MyComputer</Zone>
                     </System.Security.Policy.Zone>
                     , <System.Security.Policy.Url version="1">
                     <Url>file:///C:/Windows/assembly/GAC_32/mscorlib/
2.0.0.0__b77a5c561934e089/mscorlib.dll</Url>
                     </System.Security.Policy.Url>
                     , <System.Security.Policy.GacInstalled version="1"/>
                     , mscorlib...}
ManifestModule     : CommonLanguageRuntimeLibrary
ReflectionOnly     : False
Location           : C:\Windows\Microsoft.NET\Framework\
v2.0.50727\mscorlib.dll
ImageRuntimeVersion : v2.0.50727
GlobalAssemblyCache : True
HostContext        : 0
```

### 3. Cargar un ensamblado

Desde el inicio de este libro (con la excepción de la creación de una Windows Form), hemos utilizado tipos «integrados» gracias a los ensamblados cargados al inicio de PowerShell. Aunque tenemos otros tipos disponibles, y para su importación, es necesario cargar los ensamblados correspondientes. El ejemplo más concreto hace referencia a la utilización de Windows Forms. La utilización de objetos gráficos no es posible con PowerShell sin antes cargar el ensamblado *System.Windows.Forms*.

PowerShell no dispone de un comando capaz de cargar los ensamblados, por tanto recurriremos una vez más a llamar a una clase .NET para conseguirlo. Utilizamos el método `LoadWithPartialName` de la clase `System.Reflection.Assembly` para cargar el assembly *System.Windows.Forms*:

```
PS > [System.Reflection.Assembly]::LoadWithPartialName('System.Windows.
Forms')


GAC      Version      Location
---      -
True     v2.0.50727 C:\Windows\assembly\GAC_MSIL\System.Windows.Forms\2.0.0...
```

El método `LoadWithPartialName` no es el único que permite cargar una assembly: con los métodos `Load` o `LoadFrom` obtenemos exactamente el mismo resultado. La diferencia reside en el hecho de que en un caso utilizamos el nombre corto del assembly y en el otro el nombre largo. El nombre largo está constituido por cuatro partes: el nombre del assembly, la versión, la cultura y el token de clave pública (versión comprimida de la clave pública del assembly).

#### Ejemplo:

```
PS > [System.Reflection.Assembly]::Load('System.Windows.Forms,
version=2.0.0.0, culture=neutral, '+ 'PublicKeyToken=b77a5c561934e089')

GAC      Version      Location
---      -
True     v2.0.50727 C:\Windows\assembly\GAC_MSIL\System.Windows.Forms\2.0.0...
```

 Aunque el uso del método `LoadWithPartialName` sea más sencillo de utilizar, no permite verificar que se está cargando la versión correcta de un assembly. Por ello este método es susceptible de desaparecer en las versiones futuras del Framework.

## 4. Listar los tipos contenidos en los ensamblados

Cuando se preve utilizar un tipo particular de Framework, puede ser interesante saber si este tipo está ya cargado o no, a fin de evitar una carga del assembly que podría ser innecesaria. Para ello, existe una técnica que consiste en crear un comando capaz de recuperar todos los tipos disponibles con los ensamblados cargados en la memoria y luego hacer una selección en su espacio de nombres.

Por ejemplo, para obtener el contenido del ensamblado `System.Windows.Forms`, bastará con las dos líneas de comandos siguientes:

```
PS > $FormAssembly = [System.AppDomain]::CurrentDomain.GetAssemblies() |
where {$_.Fullname -match 'System.windows.forms'}

PS > $FormAssembly.GetExportedTypes() | foreach{$_.Fullname}

...

System.Windows.Forms.CaptionButton
System.Windows.Forms.CharacterCasing
System.Windows.Forms.CheckBox
System.Windows.Forms.CheckBox+CheckBoxAccessibleObject
System.Windows.Forms.CheckBoxRenderer
System.Windows.Forms.ListControl
System.Windows.Forms.ListBox
System.Windows.Forms.ListBox+ObjectCollection
System.Windows.Forms.ListBox+IntegerCollection
```



```
System.Windows.Forms.ListBox+SelectedIndexCollection
System.Windows.Forms.ListBox+SelectedObjectCollection
System.Windows.Forms.CheckedListBox
System.Windows.Forms.CheckedListBox+ObjectCollection
System.Windows.Forms.CheckedListBox+CheckedIndexCollection
System.Windows.Forms.CheckedListBox+CheckedItemCollection
System.Windows.Forms.CheckState
System.Windows.Forms.Clipboard
...
```

Son más de mil tipos, entre ellos existen algunos como: ListBox, TextBox, Form, viejos conocidos de quienes algún día han tenido que desarrollar una pequeña interfaz gráfica.

Aunque la gestión de búsqueda de un tipo no sea ya tan laboriosa, siempre es interesante automatizar este tipo de tareas. Para ello, puede crear una función de búsqueda en los tipos disponibles en los ensamblados cargados por PowerShell.

```
PS > function Get-TypeName ($nombre = '.') {
    [System.AppDomain]::CurrentDomain.GetAssemblies() |
    Foreach-Object{$_ .GetExportedTypes() } |
    Where-Object { $_.Fullname -match $nombre } | %{ $_.Fullname }}
```

En esta función, compuesta por tres tuberías, recuperamos en primer lugar, utilizando el método `CurrentDomain.GetAssemblies`, todos los ensamblados cargados por PowerShell.

El resultado pasa entonces por la primera tubería para finalmente aplicarle el método `GetExportedTypes`, que permitirá listar el contenido de cada ensamblado.

El resto de la función permite, por intermediación de un `Where-Object`, hacer una selección de los nombres de los tipos pasados por la segunda tubería y mostrar el nombre completo.

Así, utilizando esta función, podemos, con un simple comando, recuperar todos los tipos que incluyan la palabra clave que usted ha definido.

Ejemplo con la palabra clave `TextBox`:

```
PS > Get-TypeName TextBox

System.Windows.Forms.TextBoxBase
System.Windows.Forms.TextBox
System.Windows.Forms.DataGridTextBox
System.Windows.Forms.DataGridTextBoxColumn
```

La función nos devuelve todos los tipos que incluyen en su nombre completo el término `TextBox`.



A fin de poder utilizar esta función posteriormente, le recomendamos que la incluya en su perfil.