

# Unidad 22

Descargar estos apunte en [pdf](#) o [html](#)

## Índice

- Índice
- ▼ Librerías de clases
  - ▼ Definiciones
    - Paquete de clases
    - Artefacto
  - Empaquetando clases en CSharp
  - Accediendo las clases definidas en un paquete
- ▼ Separando en '*librerías*'
  - Archivos generados
  - Gestores de paquetes de librerías o artefactos
  - Concepto de dependencia de ensamblados
- ▼ Usando librerías de terceros
  - Nomenclatura de versiones de artefactos a subir a un repositorio
  - ▼ Ejemplo de uso de nuget como consumidor o cliente
    - Otros comandos de utilidad del CLI dotnet relacionados con librerías
  - Creando nuestras propias librerías
  - Encapsulación en paquetes de librerías
- Anexo I - Creando un ensamblado para publicar
- ▼ Anexo II - Gestionando dependencias entre librerías
  - Evitar dependencias circulares
  - Tip de diseño para modularizar en paquetes y librerías

# Librerías de clases

Antes de ver cómo **organizar en librerías** nuestras clases en C#, vamos a ver una serie de **definiciones relacionadas con la organización de Software en UML** y vamos a ver su correspondencia en .NET.

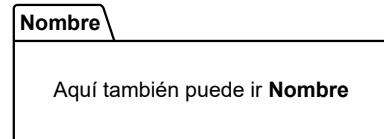
## Definiciones

### Paquete de clases

La primera definición sería de la **paquete de clases** la cual podemos decir que es un **espacio de nombres** que se utiliza para agrupar definiciones de tipos (clases, interfaces, estructuras, etc.) que están relacionados entre sí por una raíz semántica común o pueden cambiar juntos.

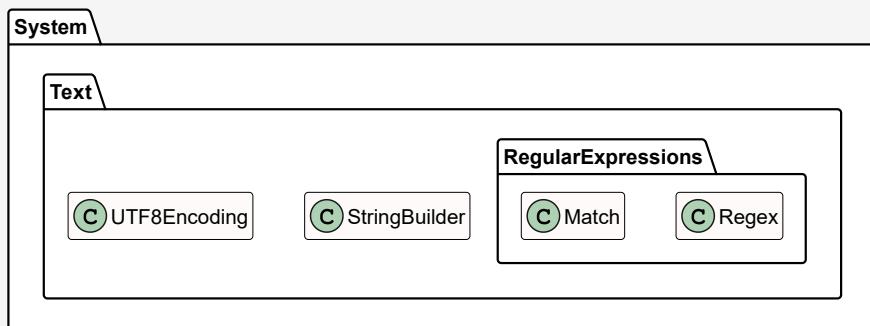
Según el lenguaje tendremos diferentes formas de definirlos, pero en C# los haremos a través de la palabra reservada **namespace** como vimos de forma más superficial a principio del curso y seguramente ya habrás intuido por la definición.

Nosotros en estos apuntes los vamos a representar mediante la siguiente figura en cuya pestaña vamos a poner el nombre del paquete.



### Ejemplo:

En C# las clases relacionadas con texto **StringBuilder** y **UTF8Encoding** C# las agrupa en el paquete de clases de nombre **System.Text**. Pero además, tendremos otro sub-paquete de clases dentro de **System.Text** denominado **System.Text.RegularExpressions** donde podremos encontrar definiciones de clases relacionadas con el manejo de texto y más específicamente con su gestión a través de expresiones regulares como son **Regex** o **Match**.



## Artefacto

Un **artefacto** es un producto tangible o entidad física resultante del proceso de desarrollo de software. En esta definición tan amplia, podríamos incluir cosas como un **documento de texto**, un archivo **fuente con un script**, un archivo **ejecutable**, una **librería**, etc.

En el caso de C# y estos apuntes, serán ensamblados o ensambles tales como ficheros **ejecutables .exe** o **librerías** de clases con extensión **.dll** los vamos a representar mediante la siguiente figura.



### Importante

En estos apuntes describiremos un **tipo de especial de artefacto de .NET** que serán los **paquetes de ensamblados de NuGet** que son ficheros comprimidos con extensión **.nupkg** y que pueden llevarnos a confusión con los **paquetes de clases** por llamarse también paquetes.

## Empaquetando clases en CSharp

Ya hemos visto que clases relacionadas se pueden agrupar bajo un espacio de nombres (namespace) al que podremos denominar también "*Paquete de clases*".

```
① namespace IdPaquete
{
    public class ClaseA { ... }
    public class ClaseB { ... }
}
```

También podemos hacer *subagrupaciones* anidando definiciones de espacios de nombres.

```
① namespace IdPaquete
{
    public class ClaseA { ... }
    public class ClaseB { ... }
    namespace IdSubPaquete
    {
        ⑥     {
            public class ClaseC { ... }
            public class ClaseD { ... }
        }
    }
}
```

Una **sintaxis más conveniente** y equivalente a la anterior podría ser:

```
① namespace IdPaquete
{
    public class ClaseA { ... }
    public class ClaseB { ... }
}
namespace IdPaquete.IdSubPaquete
⑦ {
    public class ClaseC { ... }
    public class ClaseD { ... }
}
```

Además, una de las principales ventajas de los namespaces, es que **podremos repetir el identificador o nombre de una clase** en paquetes diferentes.

```
① namespace IdPaquete1
{
    public class ClaseA { ... }
}

⑥ namespace IdPaquete2
{
    public class ClaseA { ... }
}
```

## Accediendo las clases definidas en un paquete

Para acceder a las clases '*públicas*' de un paquete, desde otro paquete podremos usar el **nombre completamente cualificado ( ccn )**, esto es, indicando la ruta separada por puntos.

```
① // Supongamos que queremos usar alguna de las clases definidas anteriormente y
② // que estarían en otro fichero fuente e incluso en otro ensamblado.

namespace IdPaquete
{
    public class ClaseA { ... }
    public class ClaseB { ... }
}

namespace IdPaquete.IdSubPaquete
{
    public class ClaseC { ... }
    public class ClaseD { ... }
}
```

```
// Clase definida en MiClase.cs -----
namespace MiPaquete
{
    public class MiClase
    {
        public void MetodoDeMiClase()
        {
            IdPaquete.ClaseA a = new IdPaquete.ClaseA();
            IdPaquete.SubPaquete.ClaseD d = new IdPaquete.SubPaquete.ClaseD();
            ...
        }
        ...
    }
}
```

O podremos usar la cláusula `using <NombreDelPaquete>;` normalmente al principio del fuente y donde indicaremos aquellos espacios de nombres o paquetes de los que queramos utilizar sus clases '*públicas*'.

```
1   using IdPaquete;
2   using IdPaquete.SubPaquete;

namespace MiPaquete
{
    public class MiClase
    {
        public void MetodoDeMiClase()
        {
            10    ClaseA a = new ClaseA();
            11    ClaseD d = new ClaseD();
            ...
        }
        ...
    }
}
```

Otra característica del `using`, es que **me permite crear alias** que nos facilita el acceso a espacios de nombres o a tipos sobre todo **cuando hay un nombre de clase repetido**. Es uso de alias es menos frecuente.

```

1 // Supongamos que queremos usar alguna de las dos ClassA definidas anteriormente y
2 // que estarían en otro ficheros fuentes o ensamblados diferentes
namespace IdPaquete1
{
    public class ClaseA { ... }
}

namespace IdPaquete2
{
    public class ClaseA { ... }
}

// Clase definida en MiClase.cs -----
// Accedo directamente al tipo del paquete y con el using le doy
// otro nombre directamente.
16 using ClaseA1 = IdPaquete1.ClaseA;
17 using ClaseA2 = IdPaquete2.ClaseA;

namespace MiPaquete
{
    public class MiClase
    {
        public void MetodoDeMiClase()
        {
            25 ClaseA1 a1 = new ClaseA1();
            26 ClaseA2 a2 = new ClaseA2();
            ...
        }
        ...
    }
}

```

**“Once *gotos* are introduced, they spread through the code like termites through a rotting house.”**

- Steve McConnell.

# Separando en 'librerías'

Además de por espacios de nombres, también podremos separar nuestras clases en unidades mayores denominadas **librerías**, en el mundo java se les llama directamente **artefactos**.

En el caso de C# son ficheros que contendrán **IL** de clases organizadas en uno o más espacios de nombres o paquetes.

## Archivos generados

En lenguajes compilados a **bytecode**. Estás librerías podrán ser archivos aunque normalmente se publicarán e instalarán a través de algún **CLI**:

- **Ensamblando**s con extensión **.dll** en para **C#** y F# en Windows.
- **Artefactos** con extensión **.jar** en el caso de **Java**, Kotlin, Scala o Groovy.

En lenguajes interpretados normalmente, los generaremos e instalaremos directamente desde un CLI:

- **Librerías** con extensión **.tar.gz** o **.whl** en el caso de **Python** en el caso de querer instalarlas **offline**.

## Gestores de paquetes de librerías o artefactos

Podré **publicar mis librerías opensource** y usar **librerías de terceros** a través de gestores de paquetes como:

- **NuGet** para ensamblados de **C#** y F#.
- **Maven** para artefactos de **Java**, Kotlin, Scala o Groovy.
- **PyPI** para librerías/artefactos de **Python**.
- **npm** para librerías de **JavaScript** y TypeScript.

Aunque vamos a hablar de ellos más adelante para el caso concreto de **NuGet**, podemos resumir que son **repositorios** de artefactos públicos o privados con **control de versiones**.

### Nota

En la práctica todos los lenguajes dispondrán de un **gestor de paquetes** que se encargará de publicar y descargar los paquetes online. Esto es así porque normalmente existan **dependencias** con otros paquetes y el gestor se encargará también de descargarlas.

# Concepto de dependencia de ensamblados

En general, siempre que usemos una clase, tendremos una dependencia de uso sobre la misma. Esto es, necesitaremos su definición. Pero esta puede estar definida:



Hasta ahora en un programa sencillo en C# como el del ejemplo **hemos usados librerías de clases definidas ya en las BCL**.

```
using System;
using System.IO;

namespace Program
{
    public class Program
    {
        private static void Main()
        {
            try
            {
                Console.WriteLine("Borrando carpeta datos...");
                Directory.Delete("datos", true);
                Console.WriteLine("Carpeta borrada");
            }
            catch (FileNotFoundException e)
            {
                Console.WriteLine(e.Message);
            }
        }
    }
}
```

Del ejemplo anterior, se nos habrá generado una dependencia entre clases ya que nuestra clase **Program** usa otras clases como **Console**, **Directory** y **FileNotFoundException** estas clases si estamos usando el **Framework** o (Marco de trabajo) de **Microsoft.NETCore.App (X.0.0)** donde (**X.0.0**) sería la versión de .NET que tenemos instalada. Estarán definidas donde se haya instalado el mismo y se encuentren sus ensamblados para usar. En el caso de Windows sería la ruta:

Si vemos la **definición** de cada una de las clases anteriores tendremos que ...

Para el caso de `Console.cs` podemos observar forma parte del paquete `System`. Este espacio de nombres o 'paquete' está **parcialmente** definido en el ensamblado `System.Console.dll`.

```
1 namespace System
{
    public static class Console
    ...
}
```

Para el caso de `Directory.cs` podemos observar forma parte del paquete `System.IO`. Este espacio de nombres o 'paquete' está **parcialmente** definido en el ensamblado `System.IO.FileSystem.dll`.

```
1 namespace System.IO
{
    public static class Directory
}
```

Para el caso de `FileNotFoundException.cs` podemos observar forma parte del paquete `System.IO`. Este espacio de nombres o 'paquete' está **parcialmente** definido en el ensamblado `System.Runtime.dll`.

```
1 namespace System.IO
{
    public class FileNotFoundException : IOException
}
```



### Importante

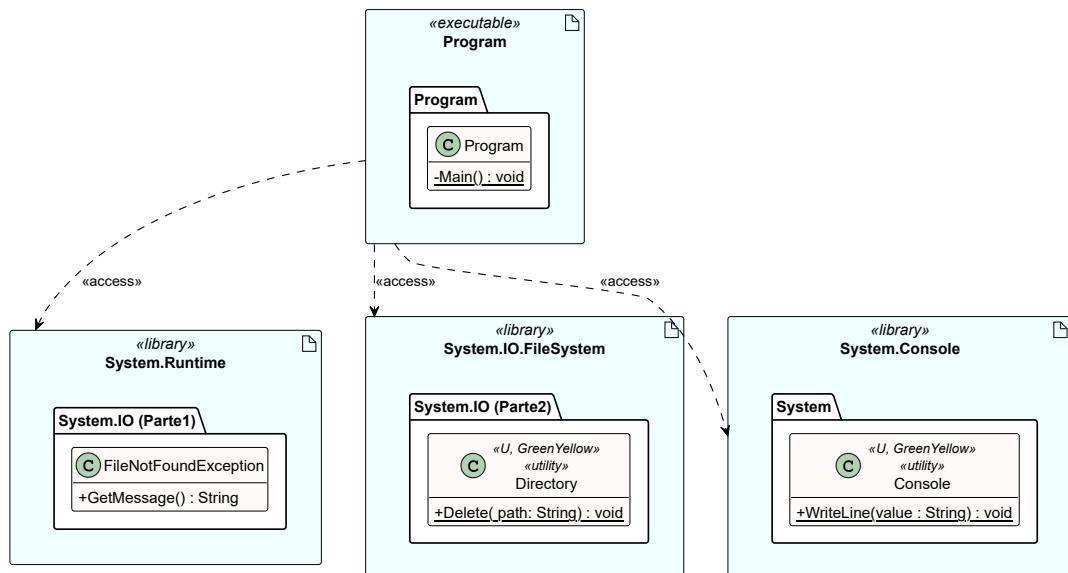
De lo anterior deducimos que en .NET, un mismo paquete de clases puede estar definido en varios ensamblados o dll's diferentes. Por ejemplo, en el caso anterior el paquete `System.IO` está parcialmente definido en los ensamblados `System.IO.FileSystem.dll` y `System.Runtime.dll`.

De lo anterior, vemos que **nuestro programa depende de tres ensamblados o 'librerías' del Framework que forman parte de las BCL**.

En el siguiente diagrama podemos observar las dependencias de ejecución **en un primer nivel**, extraídas de los fuentes anteriores, para nuestro programa.

### Nota

Estas dependencias son en un **primer nivel** puesto que si nos fijamos a su vez hay otros usings de otros paquetes de las BCL en dichas clases lo que a su vez generaría otras dependencias. Esto significa que no bastaría copiar solo esas 3 dll's del Framework para que nuestro programa funcione, ya que asu vez ellas necesitan de otras para ejecutarse.



**Spend your time on the 20 percent of the refactorings that provide 80 percent of the benefit.**

- Steve McConnell.

# Usando librerías de terceros

Según Microsoft NuGet es un repositorio de librerías/artefactos de .NET.

Proporciona su propia herramienta de gestión de paquetes CLI denominada también NuGet.

Básicamente, como indica la ilustración, nos permitirá:

**Paso 1:** Cómo desarrolladores crear un artefacto formado por uno o más ensamblados y publicarlo en un repositorio público como [nuget.org](https://nuget.org) o bien en uno privado como por ejemplo Azure DevOps. Según la documentación de microsoft...

## „ Cita

*"Con independencia de su naturaleza, un host actúa como un punto de conexión entre los creadores y los consumidores de paquetes. Los creadores compilan paquetes NuGet útiles y los publican en un host. Después, los consumidores buscan paquetes útiles y compatibles en hosts accesibles, los descargan y los incluyen en sus proyectos. Una vez instalados en un proyecto, las API de los paquetes están disponibles para el resto del código del proyecto."*

**Paso 2:** Posteriormente usar el mismo en otros desarrollos o proyectos de .NET de forma 'automatizada' y con **control de versiones**.



Es importante que los paquetes que descargamos sean **compatibles** con el FrameWork que estamos usando para realizar nuestra aplicación. Según la documentación de microsoft...

### ” Cita

*"Un paquete "compatible" implica que contiene ensamblados compilados para al menos una plataforma .NET de destino que es compatible con la plataforma de destino del proyecto de consumo."*

## Nomenclatura de versiones de artefactos a subir a un repositorio

Usaremos una sintaxis similar a la siguiente:

`<VersiónPrincipal>.<VersiónSecundaria>.<Revisión>[-<Sufijo>[.<Revisión>]]` donde ...

- **VersiónPrincipal**: Cambios importantes. (Puede no haber compatibilidad hacia atrás)
- **VersiónSecundaria**: Nuevas características, **compatibles** con versiones anteriores
- **Revisión**: Solo correcciones de errores **compatibles** con versiones anteriores
- **Sufijo**(opcional): Un guión seguido de una cadena que denota una versión preliminar (según la convención de [Versionamiento Semántico](#)).

### Ejemplo de orden de versiones para una misma revisión:

1. **1.0.0-alpha** Versión de pruebas completa para verificadores.
2. **1.0.0-alpha.1**
3. **1.0.0-beta.2** Versión de pruebas completa pública revisión 2.
4. **1.0.0-beta.11**
5. **1.0.0-rc.1** Versión completa pública candidata a ser liberada revisión 1.
6. **1.0.0** Versión completa pública liberada.

Si tienes curiosidad de cómo generar versiones para usar en NuGet puedes consultar [este enlace](#).

# Ejemplo de uso de nuget como consumidor o cliente

## Notación

A partir de ahora, vamos a usar el acrónimo **PW** (Project Workspace) para denotar la **carpeta**  donde está ubicado el **espacio de trabajo de nuestro proyecto**.

Vamos a **verlo a través de un ejemplo**. En nuestro caso queremos comprobar ortográficamente una palabra introducida por teclado y vamos a **buscar si hay algún tipo de librería de terceros** que nos ofrezca esta funcionalidad.

**Paso 1:** iremos a [nuget.org](https://nuget.org) y buscaremos por ejemplo por la palabra '**spell**'



**Paso 2:** Hemos marcado librerías disponibles para la versión actual LTS del Framework que es **net8.0** y nos filtrará por orden de más usado y la primera entrada es **WeCantSpell.Hunspell** y aunque nos ponga en la descripción una versión de .NET que no es la nuestra, no hay problema, ya que **al marcar los check de filtrado** nos aseguramos que es compatible y podemos asegurarnos al acceder a la información del paquete.



**Paso 3:** La mejor opción es **descargar el paquete desde la línea de comandos con el CLI** de .NET **dotnet** para ello como vemos en **2** seleccionaremos la pestaña **.NET CLI** y copiaremos el comando para instalarlo **3**.



## Importante

Fíjate que en **1** nos indica con qué versión de .NET es compatible el artefacto o ensamblado que nos estamos descargando como habíamos indicado en el filtro. Resaltando las versiones **LTS** (Long Term Support).

Otra opción es descargar el paquete para instalarlo de forma offline con el enlace de **4**. Esto nos descargará un fichero con la extensión **.nupkg** por si queremos tener una copia para hacer nuestro propio repositorio. Aunque **esta opción no la vamos a ver aquí**.



**Paso 4:** Desde nuestro **PW** ejecutaremos el siguiente comando indicado en el paso 3.

```
C:\MiProyectoNET> dotnet add package WeCantSpell.Hunspell --version 7.0.1
Determinando los proyectos que se van a restaurar
...
```

Tras instalar el paquete se abrirá añadido la dependencia en el fichero de configuración de nuestro proyecto, que en el ejemplo se llamaba **MiProyectoNET.csproj**

```
<Project Sdk="Microsoft.NET.Sdk">
  ...
  <ItemGroup>
    <4>      <PackageReference Include="WeCantSpell.Hunspell" Version="7.0.1" />
  </ItemGroup>
</Project>
```

Además, habrá descargado en ensamblado **WeCantSpell.Hunspell.dll** junto al del proyecto ya que lo necesitaremos para ejecutar.

Bueno, **ya podemos usar las clases definidas en la librería que hemos instalado**. Para ello, vamos a necesitar bajarnos el diccionario en castellano a usar por la Librería siguiendo los siguientes pasos:

1. Bájate el zip [diccionario\\_castellano\\_recurso.zip](#).
2. Extrae los archivos **Spanish.dic** y **Spanish.aff** que forman el diccionario en el área de trabajo del proyecto (junto al archivo **.csproj**).
3. Copia en el **Program.cs** el siguiente código de prueba del que puedes leer los comentarios para entender lo que hace. (Puedes descargar el código completo desde el [siguiente enlace](#)).

```
using System;
using System.Linq;
using WeCantSpell.Hunspell;

public class Program
{
    static void Main()
    {
        Console.WriteLine("Cargando diccionario...\n");
        string rutaEjecucion = Regex.Match(
            input: Directory.GetCurrentDirectory(),
            pattern: @"^(<ruta>.*?)(?=\\bin)",
            options: RegexOptions.IgnoreCase
        ).Groups["ruta"].Value;
        WordList diccionario = WordList.CreateFromFiles(Path.Combine(rutaEjecucion, "Spanish.dic"));
    }
}
```

```

string palabra;
do
{
    Console.WriteLine("Introduce una palabra: ");
    palabra = Console.ReadLine()!;

    string mensaje;
    // Si la palabra no está en el diccionario en español.
    if (!diccionario.Check(palabra))
    {
        mensaje = $"{palabra} no es correcta.\n";
        // Le pido al objeto diccionario en castellano que me devuelva
        // un array de sugerencias posibles a partir de la palabra introducida.
        string[] sugerencias = [... diccionario.Suggest(palabra)];
        if (sugerencias.Length > 0)
            mensaje += $"{Quisiste decir {string.Join(", ", sugerencias)}}?\n";
    }
    else
        mensaje = $"{palabra} es correcta.\n";
    Console.WriteLine(mensaje);

} while (palabra != "adios");
}
}

```

4. Si ejecuto el programa obtendremos por consola:

```

Cargando diccionario...
Introduce una palabra: prueb
prueb no es correcta.
¿Quisiste decir prueba, pruebe, pruebo?

```

## Otros comandos de utilidad del CLI dotnet relacionados con librerías

Si quisiéramos dejar de usar la librería en nuestro proyecto ejecutaríamos...

```
C:\PW> dotnet remove package WeCantSpell.Hunspell  
info : Quitando PackageReference para el paquete "WeCantSpell.Hunspell" del ...
```

Si quisiéramos, actualizar o bajarnos las dependencias de un proyecto ejecutaremos...

```
C:\PW> dotnet restore
```

Si quisiéramos saber las dependencias de nuestro proyecto...

```
C:\PW> dotnet list package  
El proyecto "ejemplo1" tiene las referencias de paquete siguientes  
[]:  
Paquete de nivel superior      solicitado    Resuelto  
> WeCantSpell.Hunspell        7.0.1          7.0.1
```

Si quisiéramos saber los repositorios de los que nos estamos bajando los paquetes... Podríamos cambiar los repositorios mediante otros comandos del CLI.

```
C:\PW> dotnet nuget list source  
origenes registrados:  
1. nuget.org [Habilitado]  
   https://api.nuget.org/v3/index.json  
2. Microsoft Visual Studio Offline Packages [Habilitado]  
   C:\Program Files (x86)\Microsoft SDKs\NuGetPackages\
```

# Creando nuestras propias librerías

## Notación

- **PW** (Project Workspace) para denotar la carpeta  donde está ubicado el **espacio de trabajo de nuestro proyecto**.
- **SW** (Solution Workspace) para denotar la carpeta  donde está ubicado el **espacio de trabajo de nuestra solución** (Debería ser una carpeta antecesora en la jerarquía la **PW** de cada proyecto).

Vemos los pasos para hacerlo con el CLI `dotnet`:

Imaginemos que queremos crear una solución en la carpeta **SW** y dentro tenemos un proyecto de Consola en `SW\MiProyecto\MiProyecto.csproj` (siendo `SW\MiProyecto\` el **PW**). Puedes crear el proyecto y añadirlo a la solución haciendo:

```
C:\SW> dotnet new console -n MiProyecto  
C:\SW> dotnet new sln -n Sw  
C:\SW> dotnet sln add .\MiProyecto\MiProyecto.csproj  
Se ha agregado el proyecto "MiProyecto\MiProyecto.csproj" a la solución.
```

Ahora queremos crear una librería llamada `MiLibreria.dll` donde definiremos diferentes clases o tipos a usar en nuestro programa y que además podrán ser reusables por otros programas.

**Paso 1:** Vamos a la carpeta donde esté ubicada nuestra solución y creamos un proyecto de librería de clases con ...

```
C:\SW> dotnet new classlib -n MiLibreria
```

**Paso 2:** Con la misma sintaxis que vimos en temas anteriores, agregaremos un nuevo proyecto a nuestra solución con...

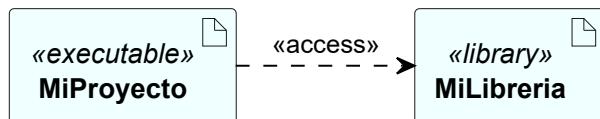
```
C:\SW> dotnet sln add .\MiLibreria\MiLibreria.csproj  
Se ha agregado el proyecto "MiLibreria\MiLibreria.csproj" a la solución.
```

**Paso 3:** Añadiremos una '**referencia**' al mi proyecto de consola a la nueva librería de clases creada. Con esto, esto estaremos generando una dependencia con el ensamblado en `MiLibreria` para poder usarlo en `MiProyecto`.

```
C:\SW> dotnet add .\MiProyecto\MiProyecto.csproj reference ..\MiLibreria\MiLibreria.csproj  
Se ha agregado la referencia "..\MiLibreria\MiLibreria.csproj" al proyecto.
```

Si ahora examinamos el contenido del archivo `MiProyecto.csproj` se habrá añadido la línea:

```
<Project Sdk="Microsoft.NET.Sdk">  
    ...  
    <ItemGroup>  
        ④     <ProjectReference Include="..\MiLibreria\MiLibreria.csproj" />  
    </ItemGroup>  
</Project>
```



### Importante

Esto indicará a la solución y al proyecto, que antes de compilarse `MiProyecto.csproj` debería compilarse `MiLibreria.csproj`

**Paso 4:** Dentro del proyecto de la librería de clases, el espacio de nombres por defecto para empaquetar nuestras clases será el mismo nombre de la librería en este caso `MiLibreria`.

- **No deberemos tener ningún método Main()** pues no se trata de un ejecutable.
- Podremos definir otros **subespacios de nombres** para agrupar nuestras clases en paquetes más específicos. En un momento dado, alguno de estos subpaquetes podría ir a una nueva librería.

**Paso 5:** Vamos aadir una clase de utilidad en la librería para ampliar las funcionalidades desde la entrada por la consola. Para ello, renombra remos el archivo `Class1.cs` generado por defecto a `ConsolaAmpliada.cs` y copiaremos el siguiente código en su interior.

```

static public class ConsolaAmpliada
{
    private static string LeeLinea(bool hidden = true)
    {
        string text = "";
        do
        {
            ConsoleKeyInfo key = Console.ReadKey(true);
            if (key.Key != ConsoleKey.Backspace && key.Key != ConsoleKey.Enter)
            {
                text += key.KeyChar;
                Console.WriteLine(hidden ? "*" : key.KeyChar.ToString());
            }
            else
            {
                if (key.Key == ConsoleKey.Backspace && text.Length > 0)
                {
                    text = text.Substring(0, (text.Length - 1));
                    Console.Write("\b \b");
                }
                else if (key.Key == ConsoleKey.Enter)
                {
                    Console.WriteLine("\n");
                    break;
                }
            }
        } while (true);
        return text;
    }

    public static string LeePassword(string label, bool hidden = true)
    {
        string password;
        bool valid;
        do
        {
            Console.WriteLine($"{label}: ");
            password = LeeLinea(hidden);
            valid = password.Length > 0;
            if (!valid)
                Console.WriteLine("El password debe tener una al menos un carácter.");
        } while (!valid);
        return password;
    }
}

```

**Paso 6:** Ahora ya podremos usar la librería en `MiProyecto` siempre y cuando hayamos hecho la referencia que se indicó en el paso 3.

Para ello, añadiremos la cláusula `using` para indicar el paquete de clases de la librería referenciada que vamos a usar en nuestro programa (**Línea 3**) y ya podremos usar las clases y tipos definidos en ella (**Línea 9**)

```
namespace MiProyecto;

③ using MiLibreria;

public static class Program
{
    public static void Main()
    {
        ⑨     string clave = ConsolaAmpliada.LeePassword("Clave", true);
        Console.WriteLine($"La clave introducida es {clave}");
    }
}
```

## Encapsulación en paquetes de librerías

Posiblemente ya te hayas fijado pero en todos nuestros ejemplos hemos antepuesto el modificador de acceso `public` a la definición de nuestras clases. Esto sucede porque, al igual que en nuestras clases, podemos definir cierto grado de '**encapsulación**' para la definición de los tipos y clases en los paquetes.



### Importante

De esta manera evitaremos el uso inapropiado por parte de terceros de nuestras clases auxiliares o tipos internos y así generar **dependencias innecesarias**.

Para ellos usaremos los siguientes modificadores de acceso antepuestos a la definición de nuestros tipos:

1. **private**: El tipo o la clase solo puede ser usada **dentro del paquete** o namespace donde se define.
2. **internal**: El tipo o la clase solo puede ser usada **dentro del ensamblado** o librería donde se define. Pero sí se puede usar en cualquier paquete dentro ensamblado.
3. **public**: El tipo o la clase se podrá usar en cualquier ámbito.

```
namespace MiLibreria
{
    ③    public class ClasePublica
    {
        // Se podrá usar en cualquier sitio que hagamos un using del namespace.
        // tanto fuera como dentro del ensamblado.
    }

    namespace MisTipos
    {
        ⑪    internal struct ClaseValorInmutableDelEnsamblado
        {
            // Se podrá usar SOLO dentro de este ensamblado siempre que hagamos un
            // using MiLibreria.MisTipos si estamos fuera del paquete MiLibreria.
        }

        ⑯    private enum EnumDelPaquete
        {
            // Solo se puede usar dentro de MisTipos
        }
    }
}
```

# Anexo I - Creando un ensamblado para publicar

En la siguiente URL puedes encontrar la información oficial acerca de la publicación o [despliegue de aplicaciones](#).

Generando un único ejecutable para una plataforma específica. Deberíamos ejecutar la siguiente instrucción del CLI de .NET en el 'Workspace' donde se encuentre nuestra solución ...

```
dotnet publish --configuration Release  
    --runtime win-x64  
    --output ./release nombre_solucion.sln  
    --self-contained=false -p:PublishSingleFile=true
```

Donde:

- `--configuration Release` : Indica que queremos hacer una compilación para cliente sin información de depuración.
- `--runtime win-x64` : Indica que queremos que la solución sea compilada para plataformas Windows de 64 bits. Donde `win-x64` es el `<RID>` (Runtime Identifier) y puedes ver las sus diferentes valores aquí <https://learn.microsoft.com/es-es/dotnet/core/rid-catalog>. De esta forma se creará un 'cargador' para ejecución específico de la plataforma.
- `--output ./release` : Indica donde queremos que nos genere el ejecutable final final.
- `nombre_solucion.sln` : Será el nombre de nuestra solución.
- `--self-contained=false` : Indicará si queremos incluir el CLR necesario para ejecutar en el ejecutable. Al poner false el CLR de .NET deberá estar instalado en la máquina donde queramos ejecutar.
- `-p:PublishSingleFile=true` : Indicamos que queremos generar un único fichero `.exe` sin librerías que lo acompañes.

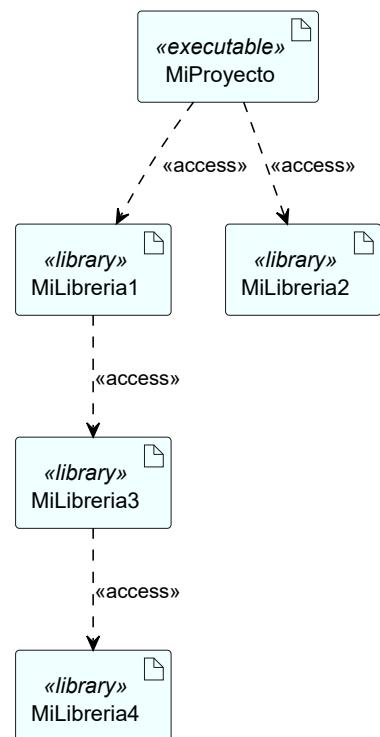
# Anexo II - Gestionando dependencias entre librerías

## Evitar dependencias circulares

Veamos el concepto de **dependencia circular** a través de un ejemplo. Para ellos supongamos que tenemos las siguiente dependencias entre librerías....

Si nos fijamos dichas dependencias nos darán lugar a un **grafo dirigido sin ciclos**.

Además podemos ver que **MiLibreria1.dll** **depende de forma indirecta de MiLibreria4.dll**. Esto significa que al compilar la solución **MiLibreria4.dll** **se deberá compilar antes** que **MiLibreria1.dll**.

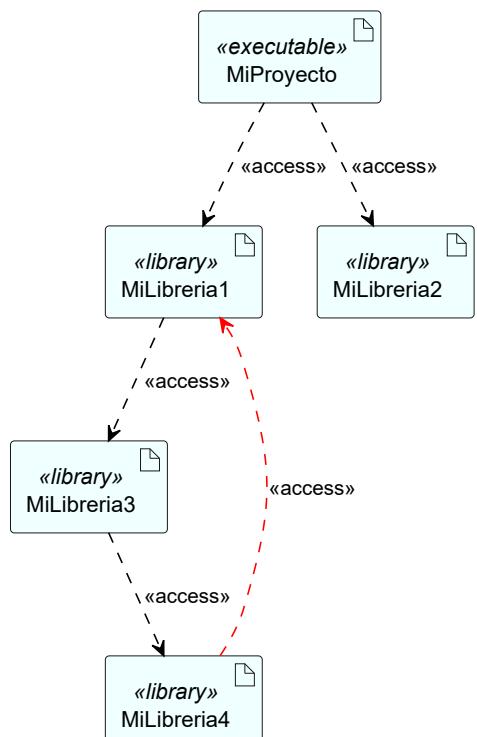


Imaginemos ahora que queremos usar una clase en **MiLibreria4.dll** que está definida **MiLibreria1.dll**. Deberemos añadir una referencia en **MiLibreria4.dll** a **MiLibreria1.dll** y por tanto una dependencia sobre la misma.

Si nos fijamos eso generará un ciclo en el grafo y por tanto un **dependencia circular**. Esto implicará que dotnet no sepa que librería tiene que compilar primero.

### Peligro

Deberemos de evitar este tipo de **dependencias circulares** 🤦 siempre, tanto entre nuestras librerías, **como entre nuestras clases y paquetes**.



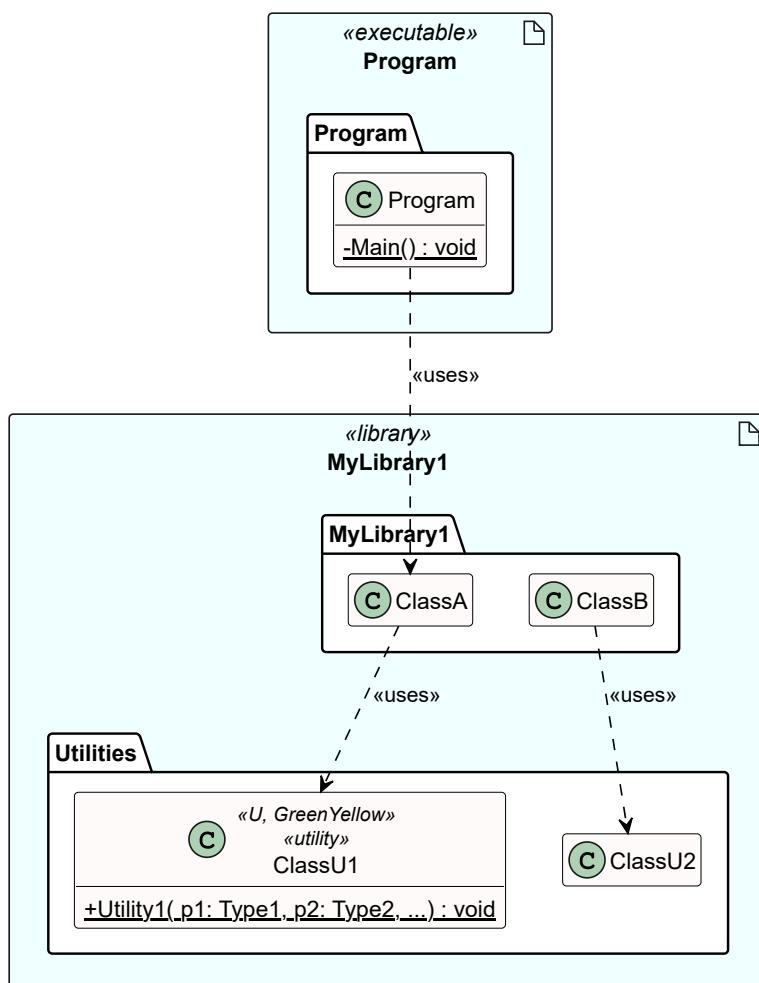
# Tip de diseño para modularizar en paquetes y librerías

Vamos a verlo a través de un ejemplo. Para ello, supongamos una librería `MyLibrary1.dll` con el paquete de clases (namespace) `MyLibrary1`.

El **paquete de clases** `MyLibrary1` está definido dentro de una librería en un ensamblado aparte denominado `MyLibrary1.dll`. Además, este **paquete de clases** contiene la definición de la clase `ClassB` y la clase `ClassA` que usa un método estático de utilidad definido en `MyLibrary1.Utilities.ClassU1` y por tanto tiene una dependencia con el tipo `ClassU1`. Estando `ClassU1` definido también dentro de `MyLibrary1.dll`.

Además tenemos un proyecto de consola `Program.exe` que usa el tipo `MyLibrary1.ClassA` definido en la librería anterior y por tanto tendremos una dependencia con la misma.

Vamos a intentar representar el caso descrito arriba para intentar visualizar las dependencias de forma gráfica. (Vuelve a releerlo viendo la representación)

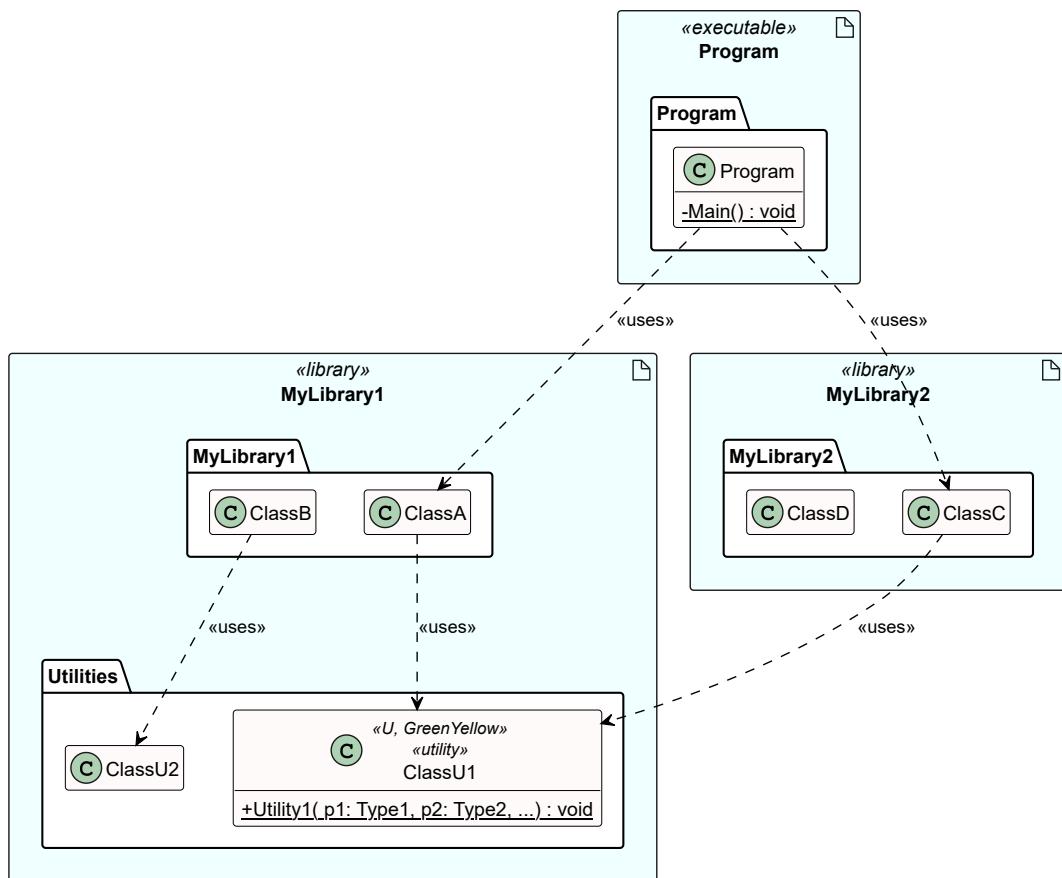


Nuestro proyecto va creciendo y ahora desde `Program.exe` usamos una clase `ClassC`, que por circunstancias, hemos definido en el paquete de clases (namespace) `MyLibrary2` incluido en una biblioteca de clases del mismo nombre llamada `MyLibrary2.dll`.

Además, esta clase `ClassC` usa también un método de utilidad definido en `MyLibrary1.Utilities.ClassU1`. Por tanto ...

1. Se generará una dependencia de `MyLibrary2.dll` sobre `MyLibrary1.dll` (Porque `ClassC` usa `ClassU1`).
2. Se generará una dependencia de `Program.dll` sobre `MyLibrary2.dll` (Porque `Program` usa `ClassC`).

Vamos a intentar representar la nueva situación tras evolucionar nuestro proyecto y sus dependencias mediante el correspondiente diagrama. (Vuelve a releer las nueva situación viendo la representación)



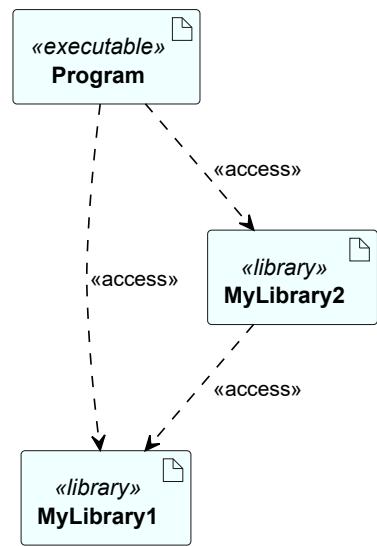
### Importante

Se producirá una situación de **doble ámbito** de uso de la clase de utilidad `ClassU1`. Pues es usada **dentro de la propia** `MyLibrary1.dll` por parte de `ClassA` y **ahora también fuera**, por parte de una librería diferente `MyLibrary2.dll` a través de su clase `ClassC`.

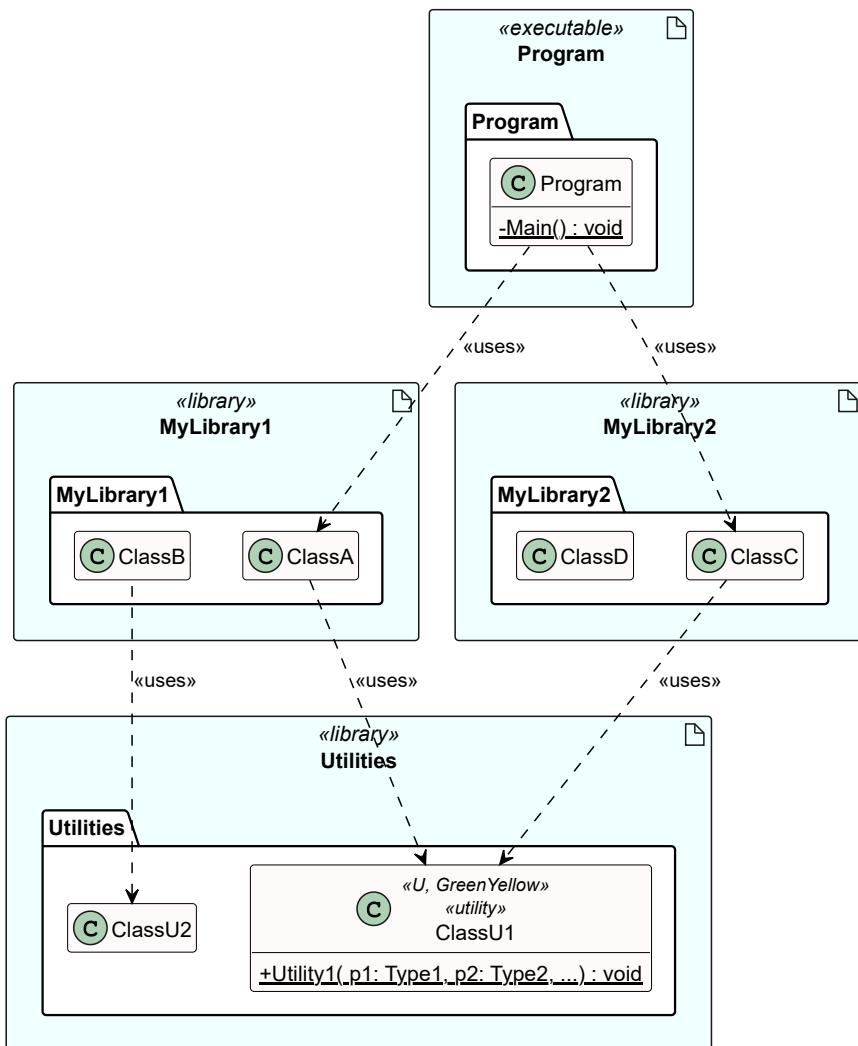
Si simplificamos el diagrama de dependencia de librerías anterior a sus artefactos tendremos el esquema de la derecha...

Además, la situación de **doble ámbito** descrita implicará que:

- Existe una dependencia de `MyLibrary1.dll` por parte de `MyLibrary2.dll`, cuando realmente solo hay una clase que queremos reutilizar y estaba en `MyLibrary1.dll` porque hasta ahora solo se utilizaba ahí.
- Ahora siempre que queramos usar `MyLibrary2.dll` vamos a necesitar junto a ella `MyLibrary1.dll`, cuando realmente hay un montón de clases que no vamos a utilizar ni nos interesan de la misma. Esto derivará en una situación de **acoplamiento** entre nuestras librerías.



Una posible solución será **sacar la clase que produce el 'doble ámbito de uso' a una tercera librería** como por ejemplo **Utilities.dll** y así separar lo que estaba produciendo la dependencia no adecuada.



Fíjemonos que ahora el **doble ámbito de uso** ha desaparecido, pues **en ambos casos** estamos accediendo a una clase **fuerá** de nuestra librería. Ahora en **Utilities.dll**.