# Unidad 22

Descargar estos apunte en pdf o html

# Índice

# 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.

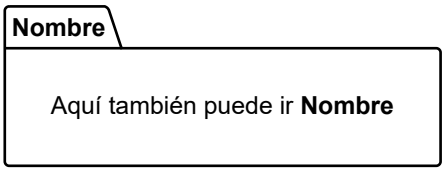
# Definiciones

## 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.

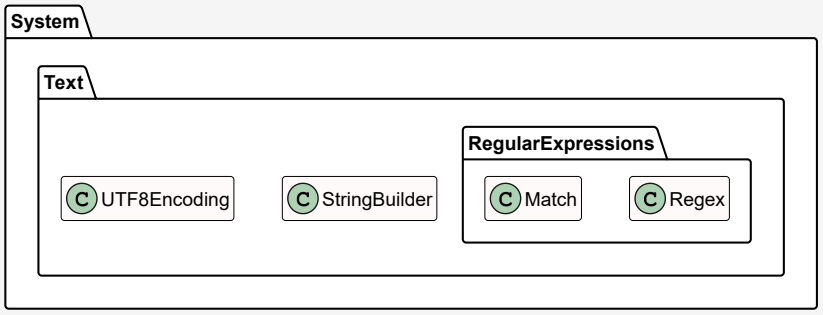
Según el lenguaje tendremos diferentes formas de definirlos, pero en C# los haremos a través de ña palabra reservada `namespace` como vimos de forma más superficial a principio del curso y seguramente ya habríamos intuido por la definición.

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




**Ejemplo:**

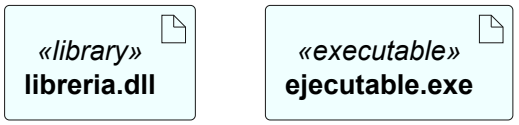
En C# las clases relacionadas con texto `StringBuilder` y `UTF8Encoding` C# las agrupa en el paquete de clases de nombre **System.Text**. Pero además, tendremos otro sub-paquete de clases dentro de `System.Text` denominado **System.Text.RegularExpressions** donde podremos encontrar definiciones de clases 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 **.ngpkg** 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***".

```
namespace IdPaquete
{
    public class ClaseA { … }
    public class ClaseB { … }
}
```

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

```
namespace IdPaquete
{
    public class ClaseA { … }
    public class ClaseB { … }
    namespace IdSubPaquete
    {
        public class ClaseC { … }
        public class ClaseD { … }
    }
}
```

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

```
namespace IdPaquete
{
    public class ClaseA { … }
    public class ClaseB { … }
}
namespace IdPaquete.IdSubPaquete
{
    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.

```
namespace IdPaquete1
{
    public class ClaseA { … }
}

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.

```
// Supongamos que queremos usar alguna de las clases definidas anteriormente y
// 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*'.

```
using IdPaquete;
using IdPaquete.SubPaquete;

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

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

> 📌 **Nota:** Es uso de alias es menos frecuente.

```
// Supongamos que queremos usar alguna de las dos ClassA definidas anteriormente y
// 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.
using ClaseA1 = IdPaquete1.ClaseA;
using ClaseA2 = IdPaquete2.ClaseA;

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

# 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**:

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

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

- **Librerías** con extensión **.tar.ar** 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 ha 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 practica 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.