

Tema 8

[Descargar estos apuntes](#)

Índice

1. [Definiciones básicas](#)
2. [Gestión de rutas en .NET](#)
 1. [Clase Path](#)
 1. [Operaciones con rutas](#)
3. [Gestión de archivos y directorios](#)
 1. [Clases de utilidad File y Directory](#)
 1. [Ejemplo de uso de File y Directory](#)
 2. [Clases FileInfo y DirectoryInfo](#)
 1. [Ejemplos de uso de FileInfo y DirectoryInfo](#)
4. [Flujos de datos en serie o streams](#)
 1. [Conceptos generales sobre Streams](#)
 1. [Operaciones y nomenclatura relacionada con Streams](#)
 1. [Apertura \(Open\)](#)
 2. [Cierre \(Close\)](#)
 3. [Lectura \(Read\) y Escritura \(Write\)](#)
 4. [Volcado \(Flush\)](#)
 2. [Streams en C#](#)
 1. [Lectura y escritura de ficheros sin transformar](#)
 1. [Apertura y Cierre de un fichero con FileStream](#)
 2. [Escritura de un fichero con FileStream](#)
 3. [Lectura de un fichero con FileStream](#)
 4. [Longitud y Posición en el Stream](#)
 5. [Desplazándonos por el Stream](#)
 6. [Recorriendo un FileStream hasta el final](#)
 2. [Pasando el flujo por un '*Decorator Stream*'](#)
 1. [Ejemplo de '*Decorator Stream*' usando BufferedStream](#)
 3. [Transformando el flujo con un '*Stream Adapter*'](#)
 1. [Stream Adapters BinaryWriter y BinaryReader](#)
 2. [Stream Adapters StreamWriter y StreamReader](#)
5. [Manejo de excepciones con ficheros](#)
6. [Serialización](#)
 1. [Conceptos Generales](#)
 1. [Persistencia](#)
 2. [Atributos en .NET](#)
 2. [Serialización a binario](#)
 1. [Saber si una clase es serializable](#)
 2. [Excluir campos o propiedades de la serialización](#)
 3. [Definiendo la Serialización como operaciones de la clase](#)
 4. [Versiones de un objeto serializado](#)

Definiciones básicas

El siguiente *'pre-conocimiento'* será necesario tenerlo presente para abordar el siguiente tema. Por pertenecer los siguiente conceptos al currículo del módulo de **'Sistemas Informáticos'**, únicamente vamos a enumerar los conceptos que vamos a necesitar.

- Un **archivo o fichero** informático es un conjunto de bytes que son almacenados en un dispositivo.
(Fuente Wikipedia)
- En informática, **una ruta (path, en inglés)** es la forma de referenciar un archivo informático o directorio en un sistema de archivos de un sistema operativo determinado.
(Fuente Wikipedia)
- Las **rutas absolutas** señalan la ubicación de un archivo o directorio desde el **directorio raíz** del sistema de archivos.
(Fuente Wikipedia)
- Las **rutas relativas** señalan la ubicación de un archivo o directorio a partir de nuestra **posición actual** en el sistema de archivos.
(Fuente Wikipedia)

Gestión de rutas en .NET

Al tratarse .NET de un Framework **multiplataforma**, dispondrá de una serie de **clases de utilidad para el manejo de rutas** de forma *'transparente'* al **Sistema Operativo (SO)** en el que estemos ejecutando.

Clase Path

Esta clase nos proporcionará de forma abstracta los siguientes propiedades que pueden tomar valores diferentes dependiendo del SO donde estemos ejecutando.

Campo	Parte de la ruta que su valor representa
<code>DirectorySeparatorChar</code>	Separador de directorios. En Windows es <code>\</code> , en Unix es <code>/</code> y Macintosh es <code>:</code>
<code>AltDirectorySeparatorChar</code>	Carácter alternativo usable como separador de directorios. En Windows y Macintosh es <code>/</code> , - mientras que en Unix es <code>\</code>
<code>PathSeparator</code>	Separador entre rutas. Aunque en los sistemas operativos más comunes es <code>;</code> podría variar en otros.
<code>VolumeSeparatorChar</code>	Separador de unidades lógicas. En Windows y Macintosh es <code>:</code> (por ejemplo <code>C:\datos</code>) y en Unix <code>/</code>

Ejemplo: Si en lugar de asignar el siguiente literal en en el código ...

```
string ruta = @"\"datos\"fichero.txt";
```

escribimos...

```
char s = Path.DirectorySeparatorChar;  
string ruta = $"{s}datos{s}fichero.txt";
```

Conseguiremos que la variable `ruta` almacene el formato de la misma según corresponda al sistema operativo sobre el que se ejecute el código anterior. Es decir, mientras que en **Windows** contendría `\datos\fichero.txt`, en **Linux o Mac OSX** contendría `/datos/fichero.txt`

Operaciones con rutas

Si te fijas en el ejemplo anterior, ocurre que en Windows el carácter usado como separador de directorios `\` coincide con el que C# usa como indicador de secuencias de escape. Por eso es incorrecto indicar literales de cadena para rutas como `"C:\datos"` ya que C# entendería que estamos intentando escapar el carácter `d`.

En su lugar hay tres alternativas:

1. Usar el campo independiente del sistema operativo `$"C:{Path.DirectorySeparatorChar}datos"`
2. Duplicar los caracteres de los literales para que dejen de considerarse secuencias de escape. Así, la ruta de ejemplo anterior quedaría... `"C:\\datos"`
3. Lo que hemos hecho en nuestro ejemplo que consiste en especificar la ruta mediante un literal de cadena plano, pues en ellos no se tienen en cuenta las secuencias de escape. Así, ahora la ruta del ejemplo quedaría ... como `@\"C:\datos"`.
Esta opción es la más simple, si sabemos cual es el SO donde vamos a ejecutar.

- Obtener la ruta con el **directorio 'padre'** o **null** si es raíz.

string Path.GetDirectoryName(string path)

- `GetDirectoryName(@"C:\MyDir\MySubDir\myfile.ext")` devuelve `"C:\MyDir\MySubDir"`
 - `GetDirectoryName(@"C:\MyDir\MySubDir")` devuelve `"C:\MyDir"`
 - `GetDirectoryName(@"C:\MyDir\")` devuelve `"C:\MyDir"`
 - `GetDirectoryName(@"C:\MyDir")` devuelve `"C:\"`
 - `GetDirectoryName(@"C:\")` devuelve `null`
- Obtener el archivo o directorio del final de la ruta. (vacío si acaba en separador)

string Path.GetFileName(string path)

- `GetFileName(@"C:\MyDir\MySubDir\myfile.ext")` devuelve `"myfile.ext"`
 - `GetFileName(@"C:\MyDir\MySubDir")` devuelve `"MySubDir"`
 - `GetFileName(@"C:\MyDir\")` devuelve `""` (Cadena vacía)
- Combinar un **DirectoryName path1** y un **FileName path2** para formar **una nueva ruta**.

⚠ **Nota:** Siempre que **path2** no empiece por el carácter separador de directorio o de volumen.

string Combine(string path1, string path2);

- `Combine(@"C:\MyDir", @"myfile.ext")` devuelve `"C:\MyDir\myfile.ext"`
- `Combine(@"C:\MyDir\", @"myfile.ext")` devuelve `"C:\MyDir\myfile.ext"`
- `Combine(@"C:\MyDir", @"MySubDir\myfile.ext")` devuelve `"C:\MyDir\MySubDir\myfile.ext"`
- `Combine(@"C:\MyDir\", @"MySubDir\myfile.ext")` devuelve `"C:\MyDir\MySubDir\myfile.ext"`
- ✗ `Combine(@"C:\MyDir\", @"\MySubDir\myfile.ext")`
- ✗ `Combine(@"C:\MyDir\", @"C:\MySubDir\myfile.ext")`

Gestión de archivos y directorios

Clases de utilidad File y Directory

Ambas clases contienen gran cantidad de **métodos estáticos de utilidad** para hacer **operaciones** con ficheros/archivos y directorios/carpetas del estilo de las que se hacen desde la **línea de comandos**.

Lógicamente los métodos definidos en **Directory** realizan operaciones sobre **directorios** y los definidos en **File** sobre **archivos**.

Nota: Cómo no es idea de estos temas copiar la documentación de Microsoft en castellano. Es recomendable que le eches un vistazo a los enlaces del anterior párrafo para hacerte una idea de las operaciones definidas en estas clases.

Ejemplo de uso de File y Directory

Veamos alguno de estos métodos a través del siguiente **ejemplo comentado**:

```
static void Main()
{
    char s = Path.DirectorySeparatorChar;
    string ruta = $"{s}datos{s}datos.txt";

    // Si no existe el directorio datos en la ruta relativa actual lo crearé.
    if (Directory.Exists(Path.GetDirectoryName(ruta)) != false)
        Directory.CreateDirectory("datos");

    // Creo el fichero datos.txt vacío. Más adelante en el tema
    // veremos que llamar al Close() es importante para que no
    // se quede abierto.
    File.Create(ruta).Close();

    // Me sitúo en el directorio datos.
    Directory.SetCurrentDirectory(Path.GetDirectoryName(ruta));

    Console.WriteLine("El fichero " + ruta + " ");

    // Compruebo si se ha creado el fichero correctamente viendo si
    // existe o no en el directorio datos (donde me acabo de situar).
    // Mostraré si existe o no por pantalla.
    Console.WriteLine(File.Exists(Path.GetFileName(ruta)) ? "existe" : "no existe");
}
```

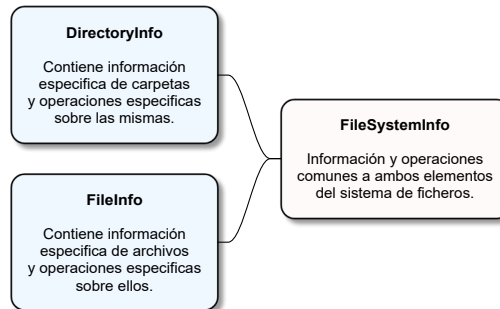
Clases FileInfo y DirectoryInfo

Además de las clases con métodos de operaciones sobre directorios y ficheros. También dispondremos de objetos que contendrán información sobre los mismos además de permitirnos también hacer ciertas operaciones.

Dichos objetos se instanciarán en memoria a través de las clases...

DirectoryInfo y **FileInfo** pues ...

- Me devolverán información sobre **carpetas** y **archivos** respectivamente.
- Además de permitirme *'navegar'* por el sistema de ficheros, moviéndome entre las diferentes carpetas. Me permitirán **realizar acciones con más precisas y con más detalle** sobre el sistema de ficheros.
- **Ambas clases heredarán de la clases abstracta `FileSystemInfo`** que contendrá la información común a archivos y carpetas. Un ejemplo sería la fecha de creación y otras propiedades comunes.
- La abstracción de elemento el sistema de ficheros almacenada la clase **`FileSystemInfo`** :
 - Se accederá a través de la sustitución de Liskov de objetos de tipo **`DirectoryInfo`** o **`FileInfo`** .



- Una ejemplo de esta información común detallada. Es el campo **`Attributes`** del sistema de ficheros.

El campo **`Attributes`** es una máscara creada a partir del Enum **`FileAttributes`**. Con información común a ficheros y directorios.

Nota: Échale un vistazo a los bits de la máscara al enlace del tipo.

Aunque sería inabordable tratar todas los casos de uso de estos objetos. Vamos a ver un par ejemplos y un caso de uso.

Ejemplos de uso de FileInfo y DirectoryInfo

Ejemplo 1:

Supongamos una función que recibe la ruta al fichero del sistema Windows 10 `C:\Windows\write.exe` y nos muestra la información común almacenada en la clase abstracta `FileSystemInfo` sobre él.

```
class EjemploFileInfo
{
    static string ObtenInformacion(string rutaAFichero)
    {
        // Obtenemos la información del fichero y
        // hacemos una sustitución a la superclase.
        7        FileSystemInfo f = new FileInfo(rutaAFichero);

        StringBuilder informacion = new StringBuilder();

        // Añadimos información del fichero.
        // Fíjate que f.Attributes muestra todos
        // los valores de enum añadidos a la máscara.
        if (f.Exists)
        15        informacion.Append($"Nombre completo: {f.FullName}\n")
            .Append($"Nombre : {f.Name}\n")
            .Append($"Extensión : {f.Extension}\n")
            .Append($"Fecha creación: {f.CreationTime}\n")
            .Append($"Fecha último acceso: {f.LastAccessTime}\n")
            .Append($"Fecha última modificación: {f.LastWriteTime}\n")
        21        .Append($"Atributos: {f.Attributes}\n");
    }
    else
        informacion.Append("Archivo no encontrado");

    return informacion.ToString();
}
static void Main()
{
    Console.WriteLine(ObtenInformacion(@"C:\Windows\write.exe"));
}
}
```

Ejemplo 2:

Supongamos un programa que me liste lo que contiene el directorio '*HOME*' de un usuario para diferentes SO, indicándome si lo encontrado es un archivo o una carpeta.

```
static void Main()
{
    //La siguiente expresión está fuera del alcance de este curso pero por resumir
    // lo que hace diremod que ...
    // Si me estoy ejecutando en Unix, Linux y MacOS X tomo el valor de la
    // carpeta de usuario de la variable HOME, en caso contrario de la
    // ubicación que indique Windows en su variable de ambiente.
    string home = Environment.OSVersion.Platform == PlatformID.Unix
        ? Environment.GetEnvironmentVariable("HOME")
        : Environment.ExpandEnvironmentVariables("%HOMEDRIVE%%HOMEPATH%");
    // Compruebo si existe la ruta devuelta por el entorno.
    if (Directory.Exists(home))
    {
        // Me sitúo en el directorio home.
        Directory.SetCurrentDirectory(home);
        // Instancio el objeto de tipo DirectoryInfo con la información de la carpeta.
        17 DirectoryInfo infoCarpeta = new DirectoryInfo(home);

        // Obtengo información de todos los objetos que hay en dicha carpeta ya sean
        // otras carpetas o archivos. Para eso llamo a GetFileSystemInfos() que me
        // devuelve un array de FileSystemInfo con dicha información.
        22 FileSystemInfo[] infoEnFS = infoCarpeta.GetFileSystemInfos();

        // Si hubiera querido ver si hay otras carpetas hubiera hecho...
        25 // -> DirectoryInfo[] infoCarpetas = infoCarpeta.GetDirectories();
        // De forma análoga si hubiera querido coger solo información de archivos...
        27 // -> FileInfo[] infoArchivos = infoCarpeta.GetFiles();

        // Recorro el array.
        foreach (FileSystemInfo infoEnFS in infoEnFS)
        {
            // Compruebo si en la máscara el item que estoy recorriendo
            // me indica que es una carpeta.
            34 bool esCarpeta = (infoEnFS.Attributes & FileAttributes.Directory)
                == FileAttributes.Directory;
            // Muestro el nombre completo indicando si es un archivo o una carpeta.
            string info = $"{(esCarpeta ? "Carpeta":"Archivo")}->{infoEnFS.FullName}";
            Console.WriteLine(info);
        }
    }
    else
        Console.WriteLine("No se ha podido encontrar la carpeta home.");
}
```

Caso de estudio:

En el siguiente caso de estudio, vamos a crear un método llamado **void OcultaDirectorio(string ruta)** que reciba una ruta a una carpeta y la marque como oculta.

Además, vamos a controlar las posibles excepciones que se generen y las vamos a relanzar al **Main**. Para ello, si hacemos **Ctrl + Click** sobre el constructor de **DirectoryInfo** podemos ver que genera las excepciones:

- **UnauthorizedAccessException** : Si no tenemos permiso de acceso a la carpeta.
- **ArgumentException** : Si la ruta contiene algún carácter inválido.
- **PathTooLongException** : Si la ruta es demasiado larga para el SO.

Además de las anteriores, generaré yo la excepción **FileNotFoundException** si no existe la ruta que recibe el método por parámetro.

Una **propuesta de solución** podría ser la siguiente ...

```
static void OcultaDirectorio(string ruta)
{
    string log = $"Ocultando el directorio '{ruta}'";
    // Mensaje para la consola de depuración.
    Debug.WriteLine(log);
    try
    {
        FileSystemInfo d = new DirectoryInfo(ruta);
        // Genero la excepción indicándo lo que estoy haciendo y además le añado
        // como innerException otra instancia donde indico realmente el error.
        if (!d.Exists)
            throw new FileNotFoundException(log,
                new FileNotFoundException($"El directorio '{ruta}' no existe"));
        // Añado con un OR de bit el atributo Hidden (Oculto) a la máscara de
        // atributos del FileSystemInfo del directorio.
        d.Attributes |= FileAttributes.Hidden;
    }
    catch (UnauthorizedAccessException e)
    {
        throw new UnauthorizedAccessException(log, e);
    }
    catch (ArgumentException e)
    {
        throw new ArgumentException(log, e);
    }
    catch (PathTooLongException e)
    {
        throw new PathTooLongException(log, e);
    }
}
```


continuación ...

Ahora defino un **Main** donde voy a usar el método definido de tal manera que:

1. Primero creo un directorio llamado '**oculto**' donde estoy ejecutando la aplicación.
2. Posteriormente lo ocultaré llamando al método **OcultarDirectorio**.

Además, capturo cualquier excepción que se pueda producir, tanto creando el directorio prueba, como llamando al método para ocultarlo,

```
static void Main()
{
    try
    {
        DirectoryInfo d = Directory.CreateDirectory("oculto");
        // Fíjate que Directory.CreateDirectory(..) devuelve un DirectoryInfo con la
        // información del directorio que acabo de crear y que aprovecho para
        // pasar la información de la ruta completa a OcultarDirectorio(..)
        OcultarDirectorio(d.FullName);
    }
    catch (Exception e)
    {
        while (e != null) {
            Console.WriteLine(e.Message);
            e = e.InnerException;
        }
    }
}
```

En este caso de estudio, se propone hacer las siguientes modificaciones...

1. Ejecutalo y comprueba si se ha creado un directorio oculto llamado '*oculto*'.
2. Intenta ocultar un directorio inexistente.
3. Revoca todos los permisos a tu usuario para esa carpeta y prueba a ocultarla.

Nota: En el [siguiente enlace](#) puede ver como se quitan los permisos al Administrador en Windows 10. **Haz lo mismo pero solo para tu usuario.**

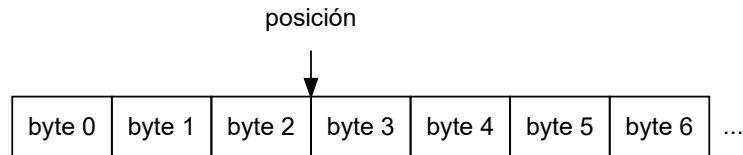
Flujos de datos en serie o streams

Conceptos generales sobre Streams

La lectura y escritura de un archivo son hechas usando un concepto genérico llamado **stream**.

Los **stream** son **flujos de datos secuenciales** que se utilizan para la **transferencia de información de un punto a otro**.

Los datos del stream se agrupan de forma básica en una **secuencia de bytes**....




Los stream pueden ser transferidos en **dos posibles direcciones**:

1. Si los datos son transferidos desde una fuente externa al programa, entonces se habla de **'leer desde el stream'**.
2. Si los datos son transferidos desde el programa a alguna fuente externa, entonces se habla de **'escribir al stream'**.

Frecuentemente, la fuente externa será un **archivo**, pero no es absolutamente necesario. Las fuentes de información externas pueden ser de diversos tipos. Algunas posibilidades incluyen:

- Leer o escribir datos a una red utilizando algún protocolo de red, donde la intención es que estos datos sean recibidos o enviados por otro computador.
- Lectura o escritura a un área de memoria.
- La Consola.
- La Impresora.
- Otros ...

 **Resumen:** Por tanto, podemos considerar un **flujo** o **stream** como una **secuencia bytes** sobre la que podemos leer o escribir. Un fichero es un tipo específico de **stream**.


Operaciones y nomenclatura relacionada con Streams

Apertura (Open)

Además de memoria, muchos flujos necesitan recursos extra. Por ejemplo, las conexiones de red crean sockets y los ficheros crean descriptores de ficheros en el sistema operativo.

En ocasiones, este proceso de apertura puede devolver errores o excepciones normalmente generadas por el SO como podrían ser:


- Falta de permisos de acceso.
- Bloqueo o uso por parte de otro programa.
- El SO no puede crear más descriptores o '*manejadores*' de ficheros.
- Otros ...

 **Resumen:** Por tanto, al proceso de reservar, adquirir o bloquear estos recursos se denomina '**apertura**' y tras crear un flujo de datos diremos que lo estamos '**abriendo**'.

Cierre (Close)

Cómo hemos comentado, si en el proceso de apertura necesitamos reservar, adquirir o bloquear recursos. Necesitaremos realizar el proceso opuesto de '**liberación**' de los mismos.

Deberemos llevar especial cuidado que este proceso de cierre se haga también si se ha producido algún error. Por tanto, si usamos excepciones, el '**cierre**' de un flujo debería ir en un bloque **finally**.

 **Resumen:** Por tanto, tras finalizar el trabajo con el flujo de datos deberemos '**cerrarlo**'.

Lectura (Read) y Escritura (Write)


Son las operaciones básicas sobre flujos y por tanto en su forma más básica transferirán bytes. Sin embargo en ocasiones estos bytes se pueden '*agrupar*' en la lectura para obtener tipos más manejables y por tanto también puede suceder el proceso inverso en el proceso de escritura.

Volcado (Flush)

Muchos flujos, en especial los que manejan ficheros, trabajan internamente con buffers donde se almacena temporalmente los bytes que se solicita **escribir**, hasta que su número alcance una cierta cantidad, momento en que son verdaderamente escritos todos a la vez en el flujo.

Esto se hace porque las **escrituras** en flujos suelen ser **operaciones lentas**, e interesa que se hagan el menor número de veces posible.

Sin embargo, hay ocasiones en que puede interesar asegurarse de que en un cierto instante se haya realizado el '**volcado**' real de los bytes en un flujo. En esos casos puede forzarse el volcado mediante la operación **Flush**, que vacía por completo su buffer interno en el flujo.

 **Resumen:** Por tanto, la operación de **escritura** en ficheros se realiza sobre un buffer de memoria RAM el cual se '**volcará**' en el soporte de almacenamiento por bloques para optimizar, ya sea de forma transparente o forzada.

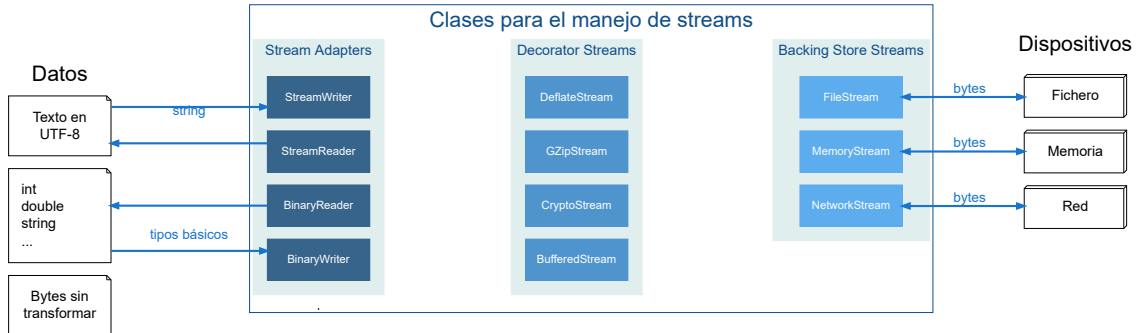
Streams en C#

En C# los archivos, directorios y flujos con ficheros, se manejan con clases del las BCL definidas en el namespace: **System.IO**

Todos los flujos de datos en C# heredan de la clase **Stream** que implementa las operaciones básicas antes descritas.

En el diagrama siguiente podemos ver de forma **resumida** cómo ha diseñado .NET sus clases para manejo de flujos.

Nota: Estos patrones de diseño y clases son análogos en otros lenguajes OO.



Si nos fijamos, tendremos de derecha a izquierda ...

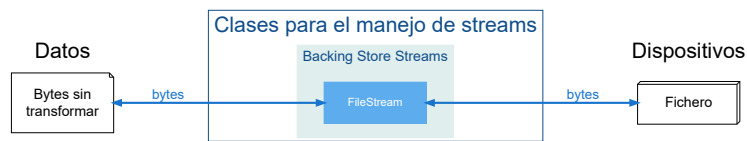
- Una serie de **dispositivos** donde vamos a realizar las operaciones de lectura y escritura.
- Una serie de clases denominadas **Backing Store Streams** que serán las que hereden de **Stream** y hagan las operaciones de lectura y escritura en los dispositivos. En concreto, **para ficheros usaremos** la subclase de stream **FileStream**.
- Una serie de clases denominadas **Decorator Streams** que transformarán una secuencia de **bytes** en otra secuencia de **bytes** y cuyo uso será opcional si queremos hacer operaciones como compresión, cifrado, etc. En este, tema veremos un case de uso **BufferedStream** pero no profundizaremos en su uso.
- **Importante:** Una serie de clases denominadas **Stream Adapters** adapter que adaptarán, en ambos sentidos, **tipos de datos básicos** manejables por los programas a secuencias de **bytes** manejables por los streams.

Nota: Si los datos que manejamos en el programa son directamente **bytes sin transformar** nos saltaremos estas clases para manejar directamente '*Decorator Streams*' o '*Backing Store Streams*'

Podremos realizar correspondencias, entre los diferentes bloques de clases, dependiendo de flujo de datos que queramos manejar y lo que queramos hacer teniendo en cuenta los datos.

Lectura y escritura de ficheros sin transformar

Escribiremos y leeremos bloques de **bytes**. Por tanto, solo necesitaremos usar la clase **FileStream** y por tanto estaremos usando la siguiente combinación...



Apertura y Cierre de un fichero con FileStream

La forma de abrirlo o crearlo más común de encontrar en la mayoría de lenguajes es usando el constructor...

public FileStream(string path, FileMode mode, FileAccess access)

- **string path** : Ubicación del fichero sobre el que queremos abrir un flujo.
- **FileMode mode** : Enum con las posibilidades de apertura del fichero.
 - **Append** : Abre el archivo si existe y realiza una búsqueda hasta el final del mismo, o crea un archivo nuevo.
 - **Create** : Especifica que el sistema operativo debe crear un archivo nuevo. Si el archivo ya existe, se sobrescribirá.
 - **CreateNew** : Especifica que el sistema operativo debe crear un archivo nuevo. Si el archivo ya existe hay una excepción.
 - **Open** : Especifica que el sistema operativo debe abrir un archivo existente.
 - **OpenOrCreate** : Especifica que el sistema operativo debe abrir un archivo si ya existe; en caso contrario, debe crearse uno nuevo.
 - **Truncate** : Especifica que el sistema operativo debe abrir un archivo existente. Una vez abierto, debe truncarse el archivo para que su tamaño sea de cero bytes.
- **FileAccess access** : Enum con el tipo de operación que vamos a realizar.
 - **Read** : Acceso de lectura al archivo.
 - **ReadWrite** : Acceso de lectura y escritura al archivo.
 - **Write** : Acceso de escritura al archivo.

El cierre del fichero lo haremos a través de...

public void Stream.Close()

- Libera el descriptor o manejador del fichero creado por el SO.
- Hace un **Flush()** del buffer del **Stream** en el dispositivo si lo hemos abierto para escritura y hay escrituras pendientes de guardar.
- En el lenguaje C#, esta tarea la hace el método **Dispose()** con el que todo flujo cuenta como **estándar recomendado** para liberar recursos de manera determinista. Sin embargo, por analogía con otros lenguajes a la clase **Stream** también dispone de un método **close()** que hace lo mismo.

Ejemplo básico de uso:

```
using System.IO;
namespace Ejemplo
{
    class Program
    {
        // El siguiente programa crea o sobrescribe el archivo ejemplo.txt en el
        // directorio donde me estoy ejecutando.
        static void Main()
        {
            FileStream fichero = new FileStream(
                "ejemplo.txt",    // Nombre del fichero
                FileMode.Create,  // Lo creo o sobrescribo si existe
                FileAccess.Write); // Solo puedo escribir en él.

            fichero.Close();
        }
    }
}
```

Escritura de un fichero con FileStream

La forma escribir más común, herencia del lenguaje C es el método...

public override void FileStream.Write(byte[] array, int offset, int count)

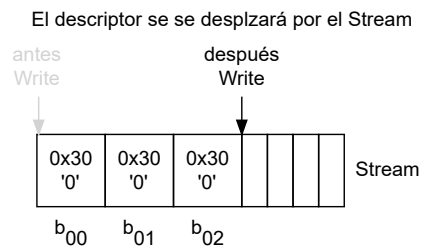
- **byte[] array** : Buffer que contiene los datos a escribir.
- **int offset** : Desplazamiento en bytes de base cero de array donde se comienzan a copiar los datos en la secuencia actual.
- **int count**: Número máximo de bytes que se deben escribir.

Ejemplo básico de uso:

```
FileStream fs = new FileStream("ejemplo.txt", FileMode.Create, FileAccess.Write);

// Escribo 3 bytes con el valor en hexadecimal del
// dígito '0' en UTF-8 al principio del stream.
byte[] datos = new byte[] { 0x31, 0x30, 0x30 };
fs.Write(datos, 0, datos.Length);

// Volcamos a disco el buffer de manera forzada.
fs.Flush();
// Cerramos el stream.
fs.Close();
```



public override void fileStream.[WriteByte](#)(byte byte)

Esté método será análogo al anterior pero escribirá solo 1 byte en el flujo desplazando el descriptor una posición.

Lectura de un fichero con FileStream

La forma escribir más común, herencia del lenguaje C es el método...

public override int [Read](#)(byte[] array, int offset, int count)

- **byte[] array** : Array a rellenar con los bytes leídos.
👉 **Debe estar predimensionado con el espacio suficiente.** Además, esta dimensión puede ser mayor al número de bytes a leer.
- **int offset** : Desplazamiento de bytes en el parámetro array donde debe comenzar la lectura.
- **int count** : Número máximo de bytes que se pueden leer.
- **Devuelve**: un entero con **el número de bytes leídos**.

Ejemplo básico de uso:

```
FileStream fichero = new FileStream(
    "ejemplo.txt",
3    FileMode.Open,      // Abro un fichero existente (sino error)
4    FileAccess.Read);   // Lo abro específicamente para lectura.

// Como voy a leer los 3 bytes que escribí en el ejemplo anterior
// creo un array con espacio de 3 como mínimo.
byte[] datos = new byte[3];

// Leo desde el principio datos.Length = 3 bytes y los añado del array datos
11 int bytesLeidos = fichero.Read(datos, 0, datos.Length);

// Cómo no se lo que ha leído realmente porque en el fichero a lo
// mejor solo había 2 bytes. En lugar de recorrer con un foreach
// recorro hasta el número de bytes leídos que me ha devuelto el método.
for (int i = 0; i < bytesLeidos; i++)
    Console.WriteLine($"{datos[i]:X} ");
fichero.Close();
```

public override int fileStream.[ReadByte](#)()

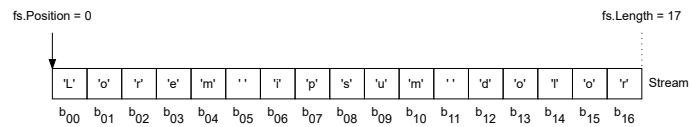
Lee un byte del archivo y avanza la posición de lectura un byte. Devuelve, el byte, convertido en un int, o **-1 si se ha alcanzado el final de la secuencia**.

Longitud y Posición en el Stream

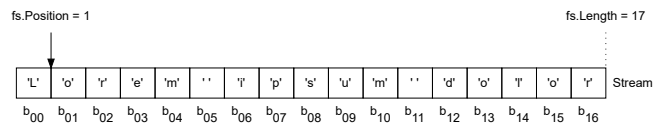
- **long Length**: Número de bytes almacenados en el flujo (tamaño del flujo).
En ficheros, el tamaño en bytes el fichero.
- **long Position**: Número del **byte actual** en el flujo **empezando en 0**.
En ficheros, la posición actual **dónde se encuentra el descriptor** de lectura o escritura del fichero.

Ejemplo básico:

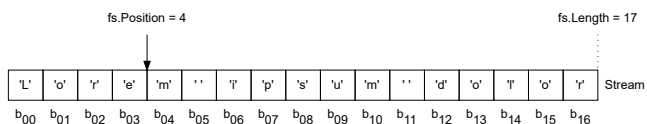
```
FileStream fs = File.Create("ejemplo.txt");  
// Paso la cadena a un array de 17 bytes para poder  
// escribirla con un FileStream.  
4 byte[] buffer = Encoding.UTF8.GetBytes("Lorem ipsum dolor");  
fs.Write(buffer, 0, buffer.Length);  
fs.Close();  
  
fs = new FileStream("ejemplo.txt", FileMode.Open, FileAccess.Read);  
9 Console.WriteLine("Longitud Fichero: " + fs.Length); // Mostrará 17  
Console.WriteLine("Posicion descriptor lectura: " + fs.Position); // Devolverá un 0
```



```
char c = (char)fs.ReadByte(); // c = 'L'  
2 Console.WriteLine("Posición descriptor lectura: " + fs.Position); // Devolverá un 1
```



```
c = (char)fs.ReadByte(); // c = 'o'  
c = (char)fs.ReadByte(); // c = 'r'  
c = (char)fs.ReadByte(); // c = 'e'  
Console.WriteLine("Posición descriptor lectura: " + fs.Position); // Devolverá un 4  
fs.Close();
```




💡 **Tip:** estos valores nos pueden ser útiles para controlar si hemos llegado o no al final de las secuencia en **algunos casos**.
Por ejemplo con la condición `while (fichero.Position < fichero.Length)` puedo saber si estoy al final de un stream.

Desplazándonos por el Stream

`public override long Seek(long offset, SeekOrigin origin)`

- `long offset` : El punto relativo a origen desde el que comienza la operación Seek. Pude ser un valor negativo si me desplazo hacia la “izquierda”.
- `SeekOrigin origin` : Especifica el comienzo, el final o la posición actual como un punto de referencia para origen, mediante el uso de un valor del Enum `SeekOrigin` .

Estos valores pueden ser: `Begin` , `Current` y `End`

-  **Nota:** No en todos los streams podremos desplazarnos con `Seek` como en los `FileStream` . Por esa razón la clase base `Stream` dispone de una propiedad `CanSeek` que me dirá si puedo desplazarme por él o no. Fíjate en el siguiente ejemplo.

Ejemplo básico de uso:

```
FileStream fichero = new FileStream("ejemplo.txt", FileMode.Open, FileAccess.Read);

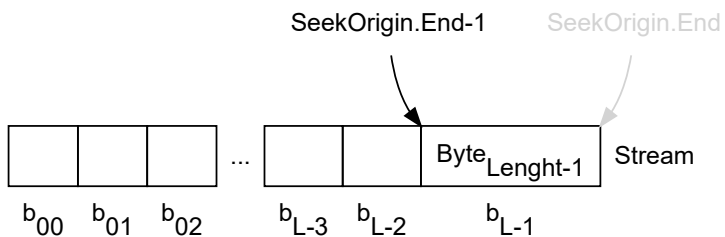
if (fichero.CanSeek) // Para streams que no admiten desplazamientos
{
    // Me sitúo al final y me desplazo 1 a la izquierda.
    fichero.Seek(-1, SeekOrigin.End);
    // Leo el último byte.
    int ultimoByte = fichero.ReadByte();
    Console.WriteLine($"El valor del último byte es {ultimoByte:X}");
}
else
    Console.WriteLine("El stream no admite desplazamientos.");

fichero.Close();
```

Caso de desplazamiento 1:

```
fichero.Seek(-1, SeekOrigin.End);
```

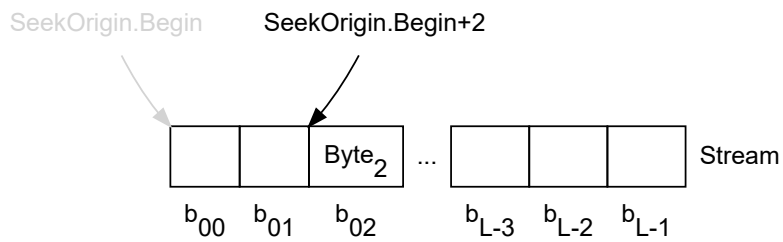
Me sitúo al **final** y me desplazo **1 byte a la izquierda**, de tal manera que me quedo para escribir o leer sobre el último byte el **byte n**.



Caso de desplazamiento 2:

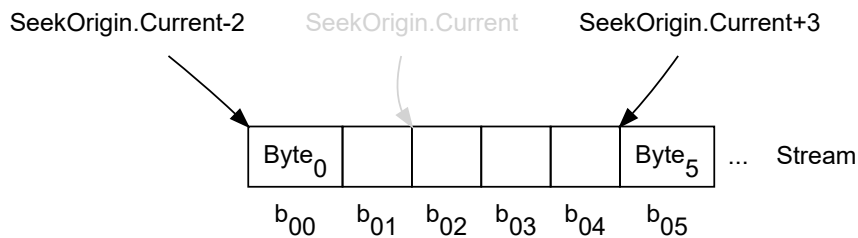
```
fichero.Seek(2, SeekOrigin.Begin);
```

Me sitúo al **principio** y me desplazo **2 bytes** a la derecha, de tal manera que me quedo para escribir o leer sobre el **byte 2**.



Caso de desplazamiento 3:

1. `fichero.Seek(-2, SeekOrigin.Current);` Desde **donde me encuentre**, me desplazo **2 bytes** a la **izquierda**, de tal manera que me quedo para escribir o leer sobre el primer byte el **byte 0**.
2. `fichero.Seek(3, SeekOrigin.Current);` Desde **donde me encuentre**, me desplazo **3 bytes** a la derecha, de tal manera que me quedo para escribir o leer sobre el **byte 5**.



Recorriendo un FileStream hasta el final

Veamos a través de un ejemplo simple, varias formas de recorrer leyendo un `FileStream` hasta el final de la secuencia.


- **Caso 1:**

```
static void Main()
{
    FileStream fs = File.Create("ejemplo.txt");
    byte[] buffer = Encoding.UTF8.GetBytes("Lorem ipsum dolor");
    fs.Write(buffer, 0, buffer.Length);
    fs.Close();

    fs = new FileStream("ejemplo.txt", FileMode.Open, FileAccess.Read);
    // Definimos un buffer de 3 bytes para las lecturas
    // esto es, podremos leer de tres en tres bytes con fs.read()
    byte[] datos = new byte[3];

    int bytesLeidos;
    do
    {
        bytesLeidos = fs.Read(datos, 0, datos.Length);
        for (int i = 0; i < bytesLeidos; i++)
            Console.Write($"{(char)datos[i]}");
    } while (bytesLeidos >= datos.Length);
}
```

En el código anterior, leeré el stream mientras se llene el buffer de 3 bytes en cada lectura. En el momento que lea menos de 3 es que he llegado al final de la secuencia.

 **Aviso:** No deberíamos recorrer en ningún caso el buffer con un `foreach` ya que siempre recorrerá 3 que es la longitud del buffer y podríamos haber leído solo 2 y por tanto podría mostrarme bytes que realmente no he leído y que están en el buffer de lecturas anteriores.

- **Caso 2:**

```
fs.Seek(0, SeekOrigin.Begin); // Vuelvo al principio para volver a recorrerla.
Console.WriteLine("\n");
while (fs.Position < fs.Length)
{
    bytesLeidos = fs.Read(datos, 0, datos.Length);
    for (int i = 0; i < bytesLeidos; i++)
        Console.Write($"{(char)datos[i]}");
}
```


Recorreremos la secuencia, mientras la **posición** en la que nos encontramos sea menor que la **longitud** de la misma.

- **Caso 3:**

```
fs.Seek(0, SeekOrigin.Begin); // Vuelvo al principio para volver a recorrerla.
Console.WriteLine("\n");
int _byte;
while ((_byte = fs.ReadByte()) != -1)
    Console.Write($"{(char)_byte}");

fs.Close();
}
```

Voy **leyendo byte a byte** mientras el valor de byte leído sea distinto de -1 o positivo.

 **Nota:** Esta opción es mucho más ineficiente en términos temporales que leer bloques de bytes con `Read()`.

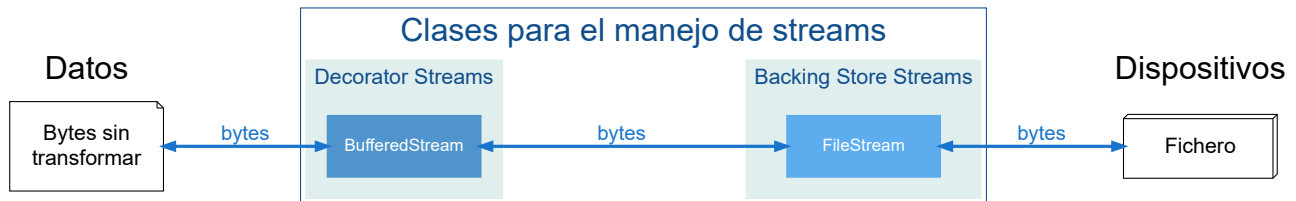
Pasando el flujo por un 'Decorator Stream'

Aunque este tipo de streams no los vamos a ver en profundidad en este curso por falta de tiempo. Básicamente, un **decorator** en POO es un patrón de diseño que añade funcionalidad a un objetos sin necesidad de utilizar el mecanismo de herencia.

Ejemplo de 'Decorator Stream' usando BufferedStream

La funcionalidad que agrega una capa de almacenamiento en buffer a las operaciones de lectura y escritura para otra secuencia (la cual 'envuelve').

Nota: Aunque **FileStream** ya tiene un buffer de escritura intermedio por defecto, nosotros **podremos ampliarlo o reducirlo** mediante esta capa de abstracción.



```
static void Main()
{
    Stopwatch cronometro = new Stopwatch();

    cronometro.Start();
    FileStream fichero = new FileStream("prueba.txt", FileMode.Create, FileAccess.Write);
    // Voy escribiendo bytes y cuando se llene el buffer del FileStream que yo no controlo
    // se realizará un volcado (flush) del mismo en el disco.
    for (int i = 0; i < 100000000; i++)
        fichero.WriteByte(33);
    fichero.Close();
    cronometro.Stop();
    Console.WriteLine($"Sin BufferedStream milisegundos = {cronometro.ElapsedMilliseconds}ms");

    cronometro.Reset();
    cronometro.Start();
    fichero = new FileStream("prueba.txt", FileMode.Create, FileAccess.Write);
    // Añado un decorador que me añade la posibilidad de gestionar un buffer de forma
    // 'transparente' antes del mandar los bytes al FileStream.
    // Le añado una capacidad de almacenaje en memoria antes de volcado de 100 bytes
    // que es bastante menor que la que tiene el FileStream por defecto. Lo cual
    // ralentizará muchísimo la escritura porque se realizarán más volcados o (flush).
    // en el disco que es una operación extremadamente costosa.
    BufferedStream ficheroBuff = new BufferedStream(fichero, 100);

    for (int i = 0; i < 100000000; i++)
        ficheroBuff.WriteByte(33);
    ficheroBuff.Close();
    cronometro.Stop();
    Console.WriteLine($"Sin BufferedStream milisegundos = {cronometro.ElapsedMilliseconds}ms");
}
```

Casos de estudio:

- Prueba a ver que sucede si aumentamos la capacidad de > almacenaje del **BufferedStream** de **100** a **1000000**.
- Prueba a ver que sucede si hacemos un **Flush()** después > de cada escritura.

Transformando el flujo con un 'Stream Adapter'

Un **adaptador** en POO es un patrón de diseño que básicamente se utiliza para ampliar y/o transformar el interfaz de las operaciones sobre una determinada clase.

- **Razón de ser:**

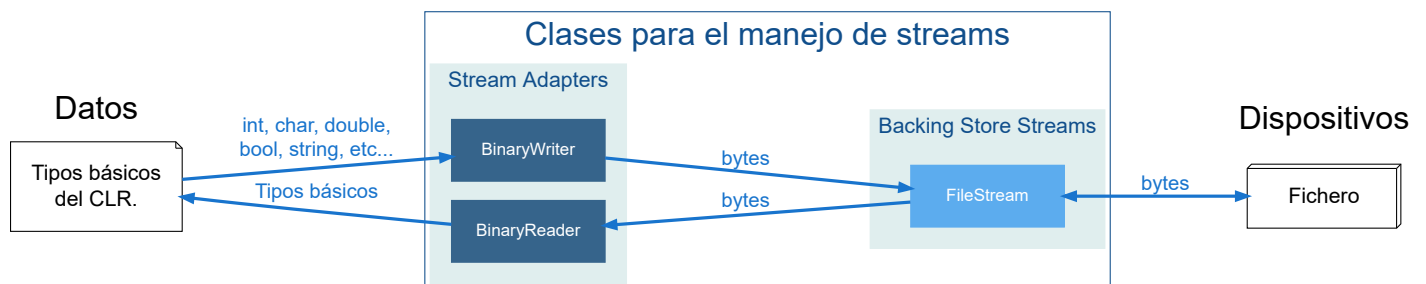
La forma de acceder a ficheros a nivel de Byte no es muy conveniente, en tanto que generalmente, en las aplicaciones no suele trabajarse directamente con bytes sino con objetos más complejos formados por múltiples bytes y/o cadenas de texto. Por esto, en **System.IO** se han incluido tipos que encapsulan **FileStream** a través de una agregación y les proporcionan una serie de métodos mejorados con los que se simplifica la lectura y escritura de datos de cualquier tipo en ficheros.

Resumen:

- Trabajar con **arrays de bytes** (buffers) es poco útil además de engorroso.
- Son una capa de abstracción para manejar de forma más cómoda los tipos datos que usamos en nuestros programas.
- 'Envuelven' otros streams de almacenamiento abiertos como por ejemplo **FileStream**.

Stream Adapters BinaryWriter y BinaryReader

Básicamente escriben o leen tipos básicos del lenguaje en una secuencia y también escribir o leer cadenas en una **codificación específica**.



- Sus métodos de escritura `Write` y lectura `Read<Tipo>` 'adaptarán' los `Write` y `Read` de `FileStream` permitiendo escribir y leer respectivamente los tipos básicos definidos en el lenguaje, `int`, `short`, `string` etc.
- Estos tipos básicos se transformarán en una secuencia de bytes **tal y como los guarda internamente en memoria .NET**. Por tanto, si hacemos un `Write` de una **cadena** la escribirá en disco guardando la marca de fin de cadena para saber hasta donde tiene que leer en un posterior `ReadString` del adaptador de lectura.

Adaptador BinaryWriter

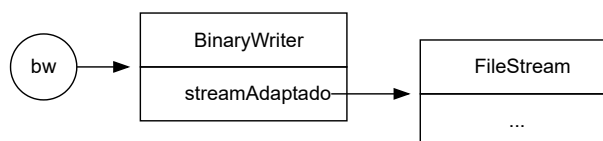
El constructor básico será `BinaryWriter(Stream streamAdaptado, Encoding codificacionCadenas)`

Veamos un **ejemplo básico de uso** escribiendo un tipo `int` (entero) en la secuencia.

```
// Creamos el FileStream
FileStream fs = new FileStream("ejemplo.bin", FileMode.Create, FileAccess.Write);
// Lo pasamos a un objeto adaptador como una agregación, indicando que las cadenas las
// codifique cómo UTF-8.
// Aunque es posible hacerlo, ya no deberíamos usar el fs porque ahora es responsabilidad del BinaryWriter.
BinaryWriter bw = new BinaryWriter(fs, Encoding.UTF8);

int valor = 10;
// El Write de bw espera todos los tipos básicos del lenguaje y los adaptará
// a la secuencia de bytes correspondiente para pasárselos al fs que adapta.
bw.Write(valor);

// El Close de BinaryWriter cerrará el stream subyacente al que está adaptando.
bw.Close();
```



Una implementación simplificada de lo que haría el `Write` del `BinaryWriter` en la `línea 11` sería la siguiente...

```
class BinaryWriter
{
    ...
    private Stream streamAdaptado;
    private Encoding codificacionCadenas;
    ...
    public void Write(int valor)
    {
        // Adapto el entero a un array de bits.
        byte[] bytes = BitConverter.GetBytes(valor);
        // Lo escribo en el flujo que estoy adaptando.
        streamAdaptado.Write(bytes, 0, bytes.Length);
    }
}
```

De hecho, si ahora leyésemos la secuencia de bytes en '*bruto*' de lo que se ha grabado en en fichero con el siguiente código...

```
FileStream fs = new FileStream("ejemplo.bin", FileMode.Open, FileAccess.Read);

// Reservo espacio para leer los 4 bytes (32 bits) resultado de guardar el entero.
byte[] bytesInt = new byte[4];
// Los leo con un FileStream normal.
fs.Read(bytesInt, 0, bytesInt.Length);


// Algunos Sistemas Operativos guardan los bytes a la inversa en memoria,
// por tanto tengo que preguntar si lo están haciendo así para invertirlos.
// Si tines curiosidad en saber el porqué puedes leer esta entrada de la
// Wikipedia https://es.wikipedia.org/wiki/Endianness aunque no es objetivo de este curso.
if (BitConverter.IsLittleEndian)
    Array.Reverse(bytesInt);

// Muestro cada uno de los bytes leídos.
// Nos mostrará 0000000A que es el 10 en hexadecimal guardado con 32 bits.
Console.Write("Valor guardado en hexadecimal: ");
foreach (byte _byte in bytesInt)
    Console.Write($"{_byte:X2}");

fs.Close();
```

Algo similar hará para el resto de tipos básicos y en el caso concreto de las cade las cadenas realizaremos un proceso parecido utilizando la codificación indicada (**Si no la indicamos en el constructor por defecto será UTF-8**). Las codificaciones pueden ser ...

Propiedad	Formato que representa el objeto devuelto
ASCII	ASCII (7 bits por carácter)
Unicode	Unicode (16 bits por carácter) usando notación little-endian
BigEndianUnicode	Unicode (16 bits por carácter) usando notación big-endian
UTF8	UTF8 (16 bits por carácter en grupos de 8 bits)
UTF7	UTF7(16 bits por carácter en grupos de 7 bits)
Default	Juego de caracteres usado por defecto en el sistema.

 **Importante:**

Es importante tener en cuenta que la codificación a usar al leer los caracteres de un fichero de texto debe ser la misma que la que se usó para escribirlos, pues si no podrían obtenerse resultados extraños.

Además, aunque binary reader permita guardar y leer cadenas. Al guardarse tal y como se almacenan en memoria, su uso no estará indicado para leer y escribir ficheros de texto `.txt` para ser abiertos posteriormente por un editor como el **notepad**.

Adaptador `BinaryReader`

El constructor básico será `BinaryReader(Stream streamAdaptado, Encoding codificacionCadenas)`

Realizará el proceso inverso al **BinaryWriter** y tendremos un método de lectura específico para la adaptación la lectura de cada tipo básico.

```
FileStream fs = new FileStream("ejemplo.bin", FileMode.Open, FileAccess.Read);
BinaryReader br = new BinaryReader(fs, Encoding.UTF8);

// Leemos un entero de 4 bytes (32 bits) a partir de la posición donde se encuentre
// el descriptor del fichero actualmente.
6 int valor = br.ReadInt32();

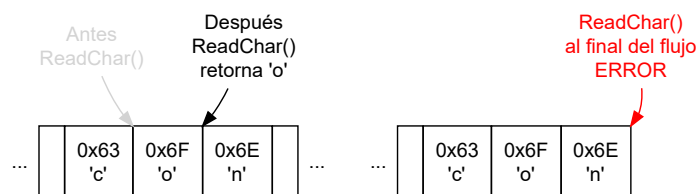
Console.WriteLine($"El valor guardado como ulong es: {valor}");
br.Close();
```

Si nos fijamos en los **métodos de lectura definidos por la clase**, todos leerán los tipos definidos en el lenguaje y al leer cualquier tipo, el descriptor del flujo avanzará en el mismo. Sin embargo habrá un caso especial de dos métodos que se complementan y comentaremos a continuación.

- **`char ReadChar()`** : Leerá un carácter del flujo y **avanzará el descriptor**. Si el descriptor se encuentra al final del mismo, se producirá una excepción.

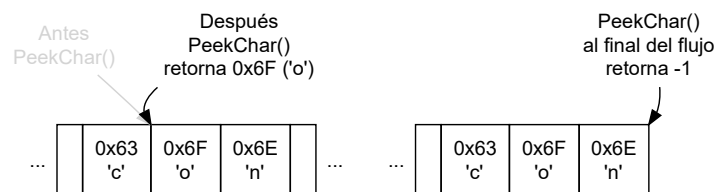
Nota: El número de bytes que avanza el descriptor, dependerá de la codificación de caracteres indicada. Por ejemplo **1 byte** para **UTF7** y **2 bytes** para **Unicode**.

Ejemplo descriptor con `ReadChar()` y codificación UTF7



- **`int PeekChar()`** : Mirará el siguiente carácter del flujo **sin avanzar el descriptor**. Si el descriptor se encuentra al final de la secuencia, devolverá **-1** (Por esa razón retorna un **int** en lugar de un **char** como el otro método).

Ejemplo descriptor con `PeekChar()` y codificación UTF7

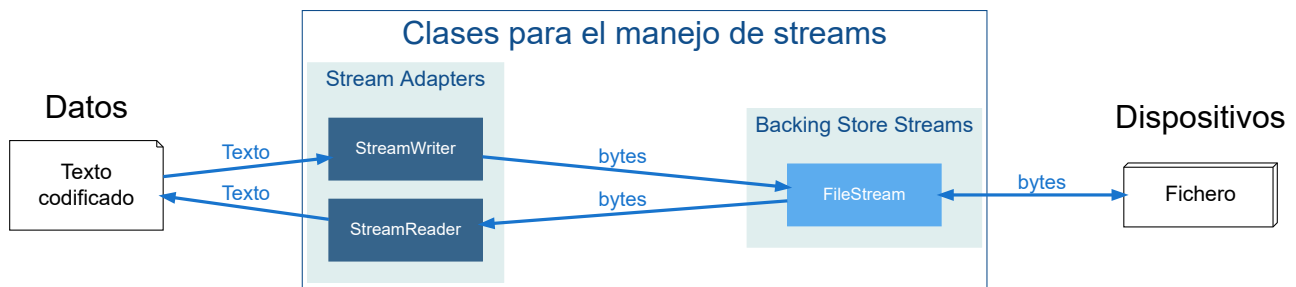


Por ejemplo, en el siguiente código leemos byte a byte de un **fichero de texto** que se guardó como UTF7 y mostramos la representación en carácter de dicho byte.

Utilizamos `PeekChar()` para detectar el final del fichero.

Nota: Si fuera un **fichero en binario** esta lectura no tendría mucho sentido pues obtendríamos caracteres no representables por consola.

```
FileStream fs = new FileStream("ejemplo.txt", FileMode.Open, FileAccess.Read);
BinaryReader br = new BinaryReader(fs, Encoding.UTF8);
3 while(br.PeekChar() >= 0)
    Console.Write(br.ReadChar());
br.Close();
```

- Están diseñados para escribir y leer texto como salida en una codificación determinada. Por tanto **están pensados para trabajar solo con archivos de texto** y no archivos con datos en binario.
- StreamWriter** utiliza de forma **predeterminada** una instancia de **UTF8** Encoding, a menos que se especifique lo contrario. Además, **se añadirá a la cabecera del archivo** de texto **2 bytes** con la codificación de caracteres utilizada. Estos bytes se denominarán **BOM (Byte Order Mark)**.
- Para **StreamReader** la forma de controlar que se ha llegado al final de fichero. Es cuando una lectura **devuelve null en lugar de un string** o a través de la propiedad **EndOfStream**.

⚠️ No podemos en ningún caso basarnos en el atributo **Position** de su **FileStream** **base** pues su valor no lo controlaremos. **Tampoco podremos desplazarnos por el FileStream al que adapta con un Seek.**

Por tanto, una vez adaptemos un **FileStream** con un **StreamWriter** o **StreamReader** **ya no podremos usarlo**, pues deja de estar bajo nuestro control.

Adaptador StreamWriter

```

FileStream fs = new FileStream("ejemplo.txt", FileMode.Create, FileAccess.Write);
StreamWriter sw = new StreamWriter(fs, Encoding.Unicode);
// Añadirá los bytes de BOM con la codificación al principio del Stream.
// Ya no podremos usar fs porque no nos pertenecerá ya.

// Escribimos dos líneas de texto con la codificación de salto de línea específica
// del Sistema Operativo donde estemos ejecutando.
sw.WriteLine("Línea de texto con salto al final codificada en Unicode");
// Podemos escribir cualquier caracter Unicode. Fuera de esta codificación ya no
// serían bien interpretados los siguientes caracteres.
sw.WriteLine("限定桶「冬風」「高雅」キャンペーン掲載");

// El Close de StreamWriter cerrará el stream subyacente al que está adaptando.
sw.Close();

```

Cómo una vez adaptado **StreamWriter** ya no tiene sentido usar el **FileStream**, nos ofrece un constructor para hacer la adaptación de forma completamente transparente, abriendo el fichero directamente en modo escritura para creación o añadir al final. Por ejemplo, el siguiente código sería equivalente al anterior.

```

const bool AÑADIR_AL_FINAL = false;
StreamWriter sw = new StreamWriter("ejemplo.txt", AÑADIR_AL_FINAL, Encoding.Unicode);
sw.WriteLine("Línea de texto con salto al final codificada en Unicode");
sw.WriteLine("限定桶「冬風」「高雅」キャンペーン掲載");
sw.Close();

```

Si abrimos el fichero con un editor de texto, lo podremos leer claramente con los saltos de línea correspondiente. Pero si lo **abrimos con editor hexadecimal** para ver que secuencia de bytes ha escrito nos encontraremos lo siguiente...

```

00000000: FF FE 4C 00 ED 00 6E 00 65 00 61 00 20 00 64 00  .~L.m.n.e.a...d.
00000010: 65 00 20 00 74 00 65 00 78 00 74 00 6F 00 20 00  e...t.e.x.t.o...
00000020: 63 00 6F 00 6E 00 20 00 73 00 61 00 6C 00 74 00  c.o.n...s.a.l.t.
00000030: 6F 00 20 00 61 00 6C 00 20 00 66 00 69 00 6E 00  o...a.l...f.i.n.
00000040: 61 00 6C 00 20 00 63 00 6F 00 64 00 69 00 66 00  a.l...c.o.d.i.f.
00000050: 69 00 63 00 61 00 64 00 61 00 20 00 65 00 6E 00  i.c.a.d.a...e.n.
00000060: 20 00 55 00 6E 00 69 00 63 00 6F 00 64 00 65 00  .U.n.i.c.o.d.e.
00000070: 0D 00 0A 00 50 96 9A 5B 76 68 0C 30 AC 51 EA 51  ....P..[vh.0,QjQ
00000080: 0D 30 0C 30 D8 9A C5 96 0D 30 AD 30 E3 30 F3 30  .0.0X.E..0-0c0s0
00000090: DA 30 FC 30 F3 30 B2 63 09 8F 0D 00 0A 00       Z0|0s02c.....

```

Donde `FF FE` serán los `bytes de BOM` indicando que los caracteres a continuación están codificados en Unicode, esto es, cada caracter está codificado con 16 bits (2 bytes). Así pues, el caracter `'L'` se ha adaptado a la secuencia de bytes `4C 00`.

Además, si nos fijamos al principio de la `línea 8` están los caracteres que definen el salto de línea en los Sistemas Windows `0D 00 0A 00`. Donde, `0D 00` esl *Carriage Return (CR)* y `0A 00` es el *Line Feed (LF)* codificados en Unicode. (También aparecen al final del fichero porque hemos escrito 2 líneas).

👁 **Nota:** Dependiendo del SO este `salto de línea` o `\n` se codificará de diferente manera.

SO	Caracter/es	Hexadecimal
Windows	CR+LF	0D 0A
Linux, Unix, Android, iOS, MacOS X	LF	0A
Hasta Mac OS 9	CR	0D

Si usamos otra codificación como `UTF8`, ya no podremos usar caracteres asiáticos porque no son soportados pero sin embargo cada caracter ocuparía 1 byte (8 bits) y el fichero ocuparía menos a pesar de tener más caracteres. Por **ejemplo**, si hacemos ...

```
const bool AÑADIR_AL_FINAL = false;
StreamWriter sw = new StreamWriter("ejemplo.txt", AÑADIR_AL_FINAL, Encoding.UTF8);
sw.WriteLine("Línea de texto con salto al final codificada en Unicode");
sw.WriteLine("Ya no podríamos escribir caracteres asiáticos en esta codificación");
sw.Close();
```

Ahora tendremos la siguiente secuencia de bytes en el fichero, en la cual ya no tenemos tantos bytes a `00`.

```
00000000: EF BB BF 4C C3 AD 6E 65 61 20 64 65 20 74 65 78      o;?LC-nea.de.tex
00000010: 74 6F 20 63 6F 6E 20 73 61 6C 74 6F 20 61 6C 20      to.con.salto.al.
00000020: 66 69 6E 61 6C 20 63 6F 64 69 66 69 63 61 64 61      final.codificada
00000030: 20 65 6E 20 55 6E 69 63 6F 64 65 0D 0A 59 61 20      .en.Unicode..Ya.
00000040: 6E 6F 20 70 6F 64 72 C3 AD 61 6D 6F 73 20 65 73      no.podríC-amos.es
00000050: 63 72 69 62 69 72 20 63 61 72 61 63 74 65 72 65      cribir.caractere
00000060: 73 20 61 73 69 C3 A1 74 69 63 6F 73 20 65 6E 20      s.asiC!ticos.en.
00000070: 65 73 74 61 20 63 6F 64 69 66 69 63 61 63 69 C3      esta.codificaciC
00000080: B3 6E 0D 0A                                           3n..
```

Fíjate que los bytes de BOM para representar UTF8 ahora son 3 `EF BB BF` y los saltos de línea si has ejecutado en un SO Windows son `0D 0A`. Para nosotros todo esto es **'transparente'** si usamos `StreamWriter`.

Aunque en los ejemplos hemos usado el método `WriteLine(string texto)` que escribe con un salto de línea al final, también podemos usar `Write(string texto)` para escribir un texto **sin** salto de línea al final.

Adaptador `StreamReader`

Hará el proceso inverso al `StreamWriter` y me ayudará a leer flujos de texto de forma **transparente** a la la codificación usada para los caracteres y los saltos de línea.

Tendremos varias posibilidades de lectura de caracteres con este adaptador:

1. **De uno en uno:** El método `int Read()` devuelve el próximo carácter del flujo como **entero**, o un `-1` si se ha llegado a su final. Por si sólo quisiésemos consultar el carácter actual pero no pasar al siguiente también se ha incluido un método `int Peek()`.

```
// Abrimos la secuencia.
FileStream fs = new FileStream("ejemplo.txt", FileMode.Open, FileAccess.Read);
const bool AUTODETECTAR_CODIFICACION_BOM = true;
// Adaptamos la secuencia para leer texto codificado.
// Si el fichero contiene BOM lo leerá con la codificación indicada en el mismo.
// sino contiene BOM lo leerá con la codificación indicada
StreamReader sr = new StreamReader(fs, Encoding.Unicode, AUTODETECTAR_CODIFICACION_BOM);

// Leo carácter a carácter y lo muestro.
int c;
while((c = sr.Read()) >= 0)
    Console.Write((char)c);
sr.Close();
```

2. **Por grupos:** El método `int Read(out char[] caracteres, int inicio, int nCaracteres)` lee un grupo de `n` caracteres y los almacena a partir de la posición `inicio` en la tabla que se le indica. El valor que devuelve es el número de caracteres que se hayan leído, que puede ser inferior a `n` si el flujo tenía menos caracteres de los indicados o un `-1` si se ha llegado al final del flujo.

Nota: Podemos decir que es un método análogo al `Read` del `FileStream`.

3. **Por líneas:** El método `string ReadLine()` devuelve la cadena de texto correspondiente a la **siguiente línea** del flujo o null si se ha llegado a su final. La cadena devuelta no incluirá al final el caracter de salto de línea correspondiente.

Nota: Puede ser una opción interesante si se trata de un **fichero de texto muy grande o desconocemos su tamaño**.

Ejemplo:

```
const bool AUTODETECTAR_CODIFICACION_BOM = true;
StreamReader sr = new StreamReader("ejemplo.txt", Encoding.Unicode, AUTODETECTAR_CODIFICACION_BOM);
// Leo línea a línea hasta final de fichero y la muestro por consola.
while(!sr.EndOfStream)
    Console.WriteLine(sr.ReadLine());
sr.Close();
```

4. **Por completo:** Un método muy útil es `string ReadToEnd()`, que nos devuelve una cadena con todo el texto que hubiese desde la posición actual del flujo sobre el que se aplica hasta el final del mismo (o null si ya estábamos en su final).

Notas:

1. Si el **fichero es muy grande**, podemos obtener un error de **memoria insuficiente**.
2. En la cadena se incluirán los saltos de línea con la codificación dependiente de la plataforma.

Ejemplo:

```
const bool AUTODETECTAR_CODIFICACION_BOM = true;
StreamReader sr = new StreamReader("ejemplo.txt", Encoding.Unicode, AUTODETECTAR_CODIFICACION_BOM);
// Muestro todo el texto pero antes, cambio cualquier posible salto de línea como CRLF o CR por LF.
Console.Write(new StringBuilder(sr.ReadToEnd()).Replace("\r\n", "\n").Replace("\r", "\n"));
sr.Close();
```

Manejo de excepciones con ficheros

Como ya se vió al hablar de la cláusula `finally` al manejar excepciones. Uno de los casos donde el uso de excepciones se hace casi imprescindible, es la manejo de ficheros. Esto es, porque se pueden producir multitud de errores como:

- Rutas de acceso (path) no válidas.
- Carencia de premisos de acceso.
- Estar el acceso bloqueado por otro usuario.
- Problemas de disco devueltos por el Sistema Operativo.
- etc.

Además, si no hacemos un `Close()` tras una excepción, puede que dejemos el recurso abierto y bloqueado para su acceso por otros usuarios o borrado.

Veamos **a través de un ejemplo** cual es la **mejor forma de abordarlo** en C#. Para ello, imaginemos una función que encapsula una escritura en disco de las que hemos realizado en los puntos anteriores y que va a realizar una gestión de excepciones tradicional de las que vimos en el tema anterior.

```
static void EscribeFichero()
{
    // Debo declarar el id aquí para que esté accesible desde el bloque finally.
    StreamWriter sw = default;
    try
    {
        const bool AÑADIR_AL_FINAL = true;
        sw = new StreamWriter(@"..\NOEXISTE\ejemplo.txt", AÑADIR_AL_FINAL, Encoding.UTF8);
        sw.WriteLine("Línea de texto con salto al final codificada en Unicode");
        // Ya no hago el Close() aquí pues se hace en el finally
    }
    // En este ejemplo solo captura las excepciones de entrada y salida y las muestro.
    catch (IOException e)
    {
        Console.WriteLine($"Creando ejemplo.txt {e.Message}");
    }
    finally
    {
        // El cierre solo lo hago aquí y solo si logré abrir el Stream
        // y he finalizado correctamente el proceso o el error se produjo
        // una vez abierto, durante el proceso de escritura.
        if (sw != null)
            sw.Close();
    }
}
```

Como podemos apreciar de la implementación anterior hemos tenido que añadir muchos bloques de control. No obstante, lo normal sería hacer solo el bloque `finally` y el control de errores en un módulo superior como.

```
static void EscribeFichero()
{
    StreamWriter sw = default;
    try
    {
        const bool AÑADIR_AL_FINAL = true;
        sw = new StreamWriter(@"..\NOEXISTE\ejemplo.txt", AÑADIR_AL_FINAL, Encoding.UTF8);
        sw.WriteLine("Línea de texto con salto al final codificada en Unicode");
    }
    finally
    {
        if (sw != null)
            sw.Close();
    }
}
```

```

static void Main()
{
    try
    {
        // Aquí podemos llamar a otros módulos que puede generar también IOException
        EscribirFichero();
    }
    catch (Exception e)
    {
        while (e != null)
        {
            Console.WriteLine(e.Message);
            e = e.InnerException;
        }
    }
}

```

Cómo vemos el **finally** lo tenemos que hacer en el módulo donde se hace la apertura del fichero y aunque hemos quitado el **catch** el código sigue siendo algo engorroso. Por esa razón, el lenguaje C# añade una cláusula que nos permitirá hacer el código anterior de forma simplificada.

La cláusula es **using** y **aunque la explicaremos en más profundidad en temas posteriores**. Lo que hará básicamente, es definir un bloque de uso de un **Stream** para el cual, si ocurre cualquier excepción **nos asegurará el cierre del mismo**.

El código del anterior de **EscribirFichero()** se puede reescribir de la siguiente manera y sería equivalente al anterior pero más simplificado y que será la forma de uso más común en los ejemplos de uso de Streams en la documentación oficial de Microsoft o cuando busquemos el uso de Streams en C# por Internet.

```

static void EscribirFichero()
{
    const bool AÑADIR_AL_FINAL = true;
    4 using (StreamWriter sw = new StreamWriter(@"..\NOEXISTE\ejemplo.txt", AÑADIR_AL_FINAL, Encoding.UTF8))
    {
        // sw estará accesible solo en este bloque y nos asegurará su cierre ante un error.
        sw.WriteLine("Línea de texto con salto al final codificada en Unicode");
        // No tenemos que cerrar el Stream porque ya lo hace el using. (NOS ASEGURA SU CIERRE)
    }
}

```

Sin embargo, **C#8** (dotnet 3.1) ha simplificado aún más el **uso de la cláusula using** de tal manera que ahora podremos hacer un código equivalente los dos anteriores del método **EscribirFichero()** de la siguiente forma...

```

static void EscribirFichero()
{
    const bool AÑADIR_AL_FINAL = true;
    4 using var sw = new StreamWriter(@"..\NOEXISTE\ejemplo.txt", AÑADIR_AL_FINAL, Encoding.UTF8);
    sw.WriteLine("Línea de texto con salto al final codificada en Unicode");

    // sw será accesible en todo el método y además nos asegura su cierre
    // al salir del ámbito del mismo ya sea porque finaliza correctamente o porque
    // se ha producido algún error.
}

```

Esta sintaxis además nos permitirá relanzar la excepción de forma sencilla sin tener que añadir el bloque **finally** y asegurándonos el cierre cuando relancemos la excepción.

```

static void EscribirFichero()
{
    try
    {
        const bool AÑADIR_AL_FINAL = true;
        6 using var sw = new StreamWriter(@"..\NOEXISTE\ejemplo.txt", AÑADIR_AL_FINAL, Encoding.UTF8);
        sw.WriteLine("Línea de texto con salto al final codificada en Unicode");
    }
    catch (IOException e)
    {
        11 throw new IOException($"Creando ejemplo.txt", e);
    }
}

```

Serialización

Conceptos Generales

Persistencia

Se define por persistencia en el mundo de la POO, como la **capacidad que tienen los objetos de sobrevivir al proceso padre que los creo**. Esto decir, que su ciclo de vida excede de la del programa que lo instanció.

La persistencia permite al programador pues almacenar, transferir y recuperar fácilmente el estado de los objetos.

¿Cómo podemos conseguir la persistencia?

La forma más común de conseguirlo es mediante la **serialización**.

La serialización es el proceso de convertir el estado de un objeto a un formato que se pueda **almacenar o transportar**. Normalmente el proceso producirá una **secuencia** de bytes o texto de marcado.

El complemento de la serialización es la **deserialización**, que convierte una secuencia de bytes o texto de marcado a un objeto. Ambos procesos pues permiten almacenar y transferir fácilmente datos.

¿A qué serializaremos una clase?

.NET ofrece dos tecnologías de serialización:

- La **serialización binaria** conserva la fidelidad de tipos, lo que resulta útil para conservar el estado de un objeto entre distintas llamadas a una aplicación. Puedes serializar un objeto en una secuencia de bytes que cómo hemos visto podemos pasar a disco, la memoria, a través de la red, etc.
 - Ejemplos:**
 - Compartir un objeto entre distintas aplicaciones '*serializándolo*' en el Portapapeles.
 - Hacer **streaming** de vídeo serializando objetos con información de vídeo comprimido a través de la red.
- La **serialización a lenguajes de marcado** como:
 - La **serialización a XML** sólo serializa las propiedades públicas y los campos, y no conserva la fidelidad de tipos. Esto resulta útil cuando desea proporcionar o consumir datos sin restringir la aplicación que utiliza los datos.
 - La **serialización a alguna notación de objetos** estándar cada vez más comunes y cuya función sería la misma que el XML pero menos '*verbosas*' y más fáciles de leer y/o modificar para humanos. Los más comunes son:
 - **JSON**: Definir configuraciones o consumo de datos a través de microservicios web o bases de datos NoSQL.
 - **YAML**: Muy usado para definir archivos de configuración de sistemas.

¿Qué interfaz podemos usar para realizar el proceso de serialización?

Aunque **la mayoría de lenguajes** ya la traen implementada. Si tuviéramos implementar nosotros las operaciones para serializar objetos. Podría ser algo parecido a esto...

```
public class <NuestraClase>
{
    public void Serializa(Stream flujo);
    public static <NuestraClase>? Deserializa(Stream flujo);
}
```

A la hora de serializar un objeto llamaríamos a su método `Serializa(Stream flujo)` y este a su vez a los Serializa de los objetos y tipos que contenga, así sucesivamente.

Muchos lenguajes como Java o C# solucionan la serialización de forma sencilla, ya que al serializar un objeto contenedor, este a su vez serializa mediante un mecanismo de '*reflexión*' y de forma transparente aquellas referencias a objetos que contiene. Lo mismo sucede al cargar o deserializar un objeto.

Durante este proceso, los campos público y privado del objeto y el nombre de la clase, incluido el ensamblado que contiene la clase, se convierten en una secuencia de bytes que, a continuación, se escribe en una secuencia de datos. Cuando, después, el objeto se deserializa, se crea una copia exacta del objeto original.

Con este fin C# me ofrece el marcar mis clases como serializables a través de un '*atributo*'.

Atributos en .NET

Un **Atributo** en .NET es una etiqueta de la sintaxis `[nombre]` que podremos aplicar a un ensamblado, clase, constructor, enumeración, campo, interfaz, método, etc... y que **genera información en el ensamblado** en forma de metadatos heredando de la clase `Attribute`.

Por ejemplo, si queremos realizar una simple serialización binaria etiquetaremos **la clase a serializar y todas las que contenga** con el atributo ya definido `[Serializable]`, sobre la definición de la clase.

Serialización a binario

Supongamos la siguiente clase `Alumno` con el atributo `[Serializable]`

```
1 [Serializable]
  public class Alumno
  {
      private string nombre;
      private string apellido;
      private int edad;

      public Alumno(string nombre, string apellido, int edad)
      {
          this.nombre = nombre;
          this.apellido = apellido;
          this.edad = edad;
      }
  }
```

Para posteriormente serializar el tipo deberemos utilizar un formateador, el más común es el `IFormatter`, que se utiliza de la siguiente manera:

```
IFormatter f = new BinaryFormatter();
f.Serialize(<medioalmacenamiento>, <objetoaserializar>);
```

Ejemplo a partir de la clase `Alumno` anterior:

```
public static void Main()
{
    var a = new Alumno("Pepa", "Pérez", 25);
    using var s = new FileStream("Dato.bin", FileMode.Create, FileAccess.Write);
5    IFormatter f = new BinaryFormatter();
6    f.Serialize(s, a);
}
```

Si examinamos en Hexadecimal el fichero serializado nos habrá generado algo similar a esto...

00000000: 00 01 00 00 00 FF FF FF FF 01 00 00 00 00 00 00
00000010: 00 0C 02 00 00 00 3E 45 6A 65 6D 70 6C 6F 2C 20>Ejemplo,.
00000020: 56 65 72 73 69 6F 6E 3D 31 2E 30 2E 30 2E 30 2C	Version=1.0.0.0,
00000030: 20 43 75 6C 74 75 72 65 3D 6E 65 75 74 72 61 6C	.Culture=neutral
00000040: 2C 20 50 75 62 6C 69 63 4B 65 79 54 6F 68 65 6E	,.PublicKeyToken
00000050: 3D 6E 75 6C 6C 05 01 00 00 00 06 41 6C 75 6D 6E	=null.....Alumn
00000060: 6F 03 00 00 00 06 6E 6F 6D 62 72 65 08 61 70 65	o.....nombre.ape
00000070: 6C 6C 69 64 6F 04 65 64 61 64 01 01 00 08 02 00	llido.edad.....
00000080: 00 00 06 03 00 00 00 04 50 65 70 61 06 04 00 00Pepa....
00000090: 00 06 50 C3 A9 72 65 7A 19 00 00 00 0B	..PC)rez.....

De forma análoga realizaremos la **deserialización**.

```
public static void Main()
{
    using var s = new FileStream("Dato.bin", FileMode.Open, FileAccess.Read);
4    IFormatter f = new BinaryFormatter();
    Alumno a = f.Deserialize(s) as Alumno ?? new Alumno("Desconocido", "", 0);
6    Console.WriteLine(a);
}
```

Saber si una clase es serializable

Podemos utilizar el API de Reflexión para consultar los atributos de un tipo. Una posible forma sería:

```
bool esSerializable = typeof(Alumno).Attributes.ToString().IndexOf("Serializable") > 0;
```

Excluir campos o propiedades de la serialización

Una clase a menudo contiene **campos que no se quieran serializar**. Por ejemplo, **campos calculados** o campos con información que no queremos guardar. Para que un campo no se serialice, deberemos aplicarle el atributo `[NonSerialized]`.

```
[Serializable]
public class Alumno
{
    private string nombre;
    private string apellido;
    private int edad;
    [NonSerialized] // Excluirá el campo nombreCompleto de la serialización y deserialización.
    private string nombreCompleto;

    public Alumno(string nombre, string apellido, int edad)
    {
        this.nombre = nombre;
        this.apellido = apellido;
        this.edad = edad;
        nombreCompleto = $"{nombre} {apellido}";
    }
}
```

Definiendo la Serialización como operaciones de la clase

Si quisiéramos refactorizar el código para pasar la '**responsabilidad**' de la serialización a la clase `Alumno`. Tendríamos que implementar las operaciones que definimos al principio dentro de la propia clase.

Ejemplo más completo de serialización para ver esto último:

```
[Serializable]
class Alumno
{
    private string nombre;
    private string apellido;
    private int edad;

    public Alumno(string nombre, string apellido, int edad)
    {
        this.nombre = nombre;
        this.apellido = apellido;
        this.edad = edad;
    }

    15 public void SerializaABinario(Stream s)
    {
        18     new BinaryFormatter().Serialize(s, this);
    }

    20 public static Alumno? DeserializaDeBinario(Stream s)
    {
        23     return new BinaryFormatter().Deserialize(s) as Alumno;
    }

    public override string ToString()
    {
        return $"Nombre: {nombre}\nApellido: {apellido}\nEdad: {edad}";
    }
}
```



```

static class Programa
{
    static void MuestraAlumnos()
    {
        using var s = new FileStream("alumnos.bin", FileMode.Open, FileAccess.Read);
        while (s.Position < s.Length)
        {
            var a = Alumno.DeserializaDeBinario(s);
            if (a == null)
            {
                Console.WriteLine("Error leyendo alumnos/as.");
                break;
            }
            else
                Console.WriteLine(a);
        }
    }

    static void Main()
    {
        // Serializamos datos de tres alumnos.
        using (var s = new FileStream("alumnos.bin", FileMode.Append, FileAccess.Write))
        {
            new Alumno("Pepa", "Pérez", 25).SerializaABinario(s);
            new Alumno("Maria", "Peláez", 22).SerializaABinario(s);
            new Alumno("Rosa", "López", 26).SerializaABinario(s);
        }
        MuestraAlumnos();
    }
}

```

Versiones de un objeto serializado

En muchas ocasiones el programa evoluciona y **nuevos requerimientos** hacen que añadamos nuevos campos. Esto hace necesaria la posibilidad de mantener diferentes versiones de un objeto y **asegurar la compatibilidad hacia atrás** del mismo.

Esta posibilidad está contemplada en C# a través de marcar ciertos campos con **atributos** específicos.

Por ejemplo, supongamos que queremos guardar un nuevo objeto **Direccion** junto con los datos del **Alumno**.

Podemos crear una clase dirección **[Serializable]** de forma análoga a como hicimos la de alumno...

```

1 // Puesto que también se va a serializar por ser una agregación de Alumno también deberemos
// marcarla como serializable aunque no añadamos opciones de serialización por separado.
3 [Serializable]
class Direccion
{
    private string calle;
    private int numero;
    private string ciudad;
    private string codigoPostal;

    public Direccion()
    {
        calle = "Desconocida";
        numero = 0;
        ciudad = "Desconocida";
        codigoPostal = "0000";
    }

    public Direccion(string calle, int numero, string ciudad, string codigoPostal)
    {
        this.calle = calle;
        this.numero = numero;
        this.ciudad = ciudad;
        this.codigoPostal = codigoPostal;
    }

    public override string ToString()
    {
        return $"{calle} {numero} {ciudad} {codigoPostal}";
    }
}

```

A continuación, añadimos el nuevo campo dirección a la clase `Alumno`. Indicando con el atributo `[OptionalField(VersionAdded = 2)]` que se tratará de un campo opcional que solo estará a partir de la **versión 2** de la misma.

Además, podemos marcar con el atributo `[OnDeserializing]` un método que se ejecutará durante el proceso de 'deserialización' y inicializará el campo a una **instancia por defecto** por si acaso el mismo no existe en la versión del objeto que estamos deserializando.

```
[OnDeserializing]
private void SetDirección(StreamingContext sc)
{
    direccion = new Direccion();
}
```

Por tanto, la clase alumno modificada podría quedar así ...

```
[Serializable]
class Alumno
{
    private string nombre;
    private string apellido;
    private int edad;

    8 [OptionalField(VersionAdded = 2)]
    9 private Direccion direccion;

    public Alumno(string nombre, string apellido, int edad, Direccion direccion)
    {
        this.nombre = nombre;
        this.apellido = apellido;
        this.edad = edad;
        this.direccion = direccion;
    }

    19 [OnDeserializing]
    private void SetDirección(StreamingContext sc)
    {
        23 direccion = new Direccion();
    }

    // Métodos de seriralización omitidos para abreviar ...

    public override string ToString()
    {
    29 return $"Nombre: {nombre}\nApellido: {apellido}\nEdad: {edad}\nDirección: {direccion}";
    }
}
```

Imaginemos que ahora modificamos el `Main` de tal manera que añadimos objetos `Alumno` con el nuevo dato al fichero `alumnos.bin` y posteriormente 'deserializamos' todos los objetos `Alumno`, mostrándolos con la mismo método `MuestraAlumnos()`

```
static void Main()
{
    // Serializamos datos de tres alumnos.
    using (var s = new FileStream("alumnos.bin", FileMode.Append, FileAccess.Write))
    {
        new Alumno("Juani", "Ortiz", 21, new Direccion("Marqués de Molins", 58, "Alicante", "03004")).SerializaABinario(s);
        new Alumno("Marcos", "Jiménez", 20, new Direccion("Cerámica", 24, "Alicante", "03010")).SerializaABinario(s);
    }
    MuestraAlumnos();
}
```

EL programa sabrá recuperar versiones anteriores del objeto `Alumno` mostrando instanciándolo a la información por defecto que hemos definido.

```
Nombre: Rosa
Apellido: López
Edad: 26
4 Dirección: Desconocida 0 Desconocida 0000
Nombre: Juani
Apellido: Ortiz
Edad: 21
8 Dirección: Marqués de Molins 58 Alicante 03004
```