# Unidad 21

Descargar estos apunte en pdf o html

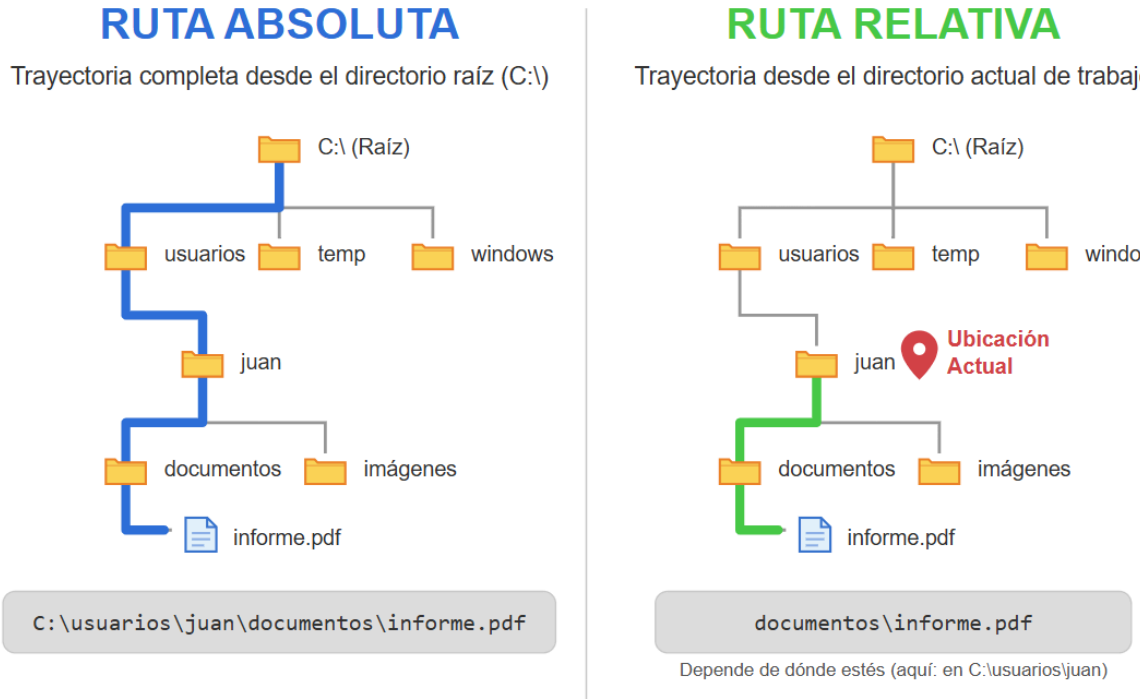
# Índice

# Ficheros

## 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 las siguientes propiedades, que pueden tomar valores diferentes dependiendo del sistema operativo donde estemos ejecutando.

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

**Ejemplo:** Si en lugar de asignar el siguiente literal 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 caracter `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**. 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**:

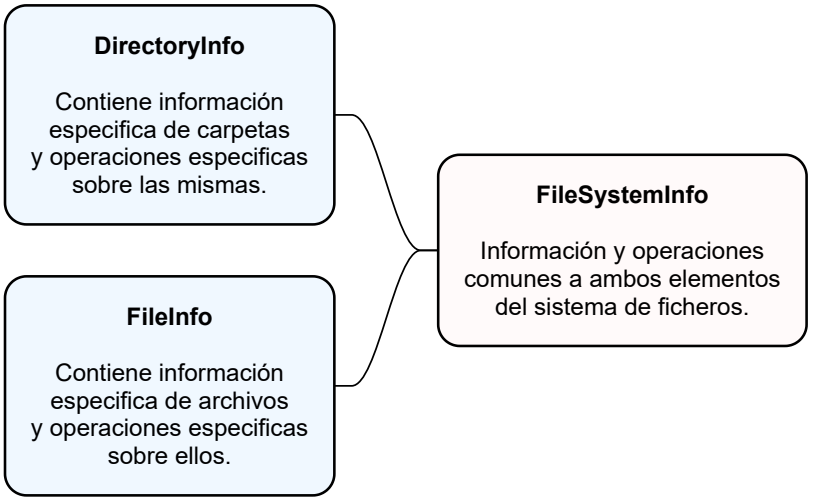
```
static void Main()
{
    char s = Path.DirectorySeparatorChar;
    string ruta = $".{s}datos{s}datos.txt";

    // 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) ?? $"{s}");
    Console.Write("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 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 del sistema de ficheros se almacena en 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. Échale un vistazo a los bits de la máscara en el enlace del tipo.

Aunque es "imposible" tratar todas los casos de uso de estos objetos. Vamos a ver un par ejemplos y un caso de uso.

**Ejemplo 1:** (Puedes descargar el código completo desde **este enlace**)

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

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

        StringBuilder informacion = new ();

        // Añadimos información del fichero.
        // Fíjate que f.Attributes muestra todos
        // los valores dele enum añadidos a la máscara.
        if (f.Exists)
            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")
                       .Append($"Atributos: {f.Attributes}\n");
        else
            informacion.Append("Archivo no encontrado");

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

**Ejemplo 2:** (Puedes descargar el código completo desde **este enlace**)

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()
{
    string 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.
        DirectoryInfo infoCarpeta = new DirectoryInfo(home);

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

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

        // Recorro el array.
        foreach (FileSystemInfo infoEnFS in infosEnFS)
        {
            // Compruebo si en la máscara el item que estoy recorriendo
            // me indica que es una carpeta.
            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:** (Puedes descargar el código completo desde **este enlace**)

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 caracter 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}'";
    try
    {
        FileSystemInfo d = new DirectoryInfo(ruta);
        if (!d.Exists)
            throw new FileNotFoundException(
                    log, new FileNotFoundException(
                            $"El directorio '{ruta}' no existe"));
        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);
    }
}
```

Genero la excepción indicando lo que estoy haciendo y además le añado como `innerException` otra instancia donde indico realmente el error.

Añado el atributo `Hidden` **(Oculto)** a la máscara de atributos del `FileSystemInfo` del directorio. Enumerado no excluyente, por lo que uso un **OR de bits** para añadirlo.

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 `OcultaDirectorio`.
   Además, capturo cualquier excepción que se pueda producir, tanto creando el directorio prueba, como llamándo al método para ocultarlo,

```
static void Main()
{
    try
    {
        DirectoryInfo d = Directory.CreateDirectory("oculto");
        OcultaDirectorio(d.FullName);
    }
    catch (Exception? e)
    {
        while (e != null) {
            Console.WriteLine(e.Message);
            e = e.InnerException;
        }
    }
}
```

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 `OcultaDirectorio(...)`

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

1. Ejecutaló 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 somo se quitan los permisos al Administrador en Windows. **Intenta hacer lo mismo pero solo para tu usuario**.

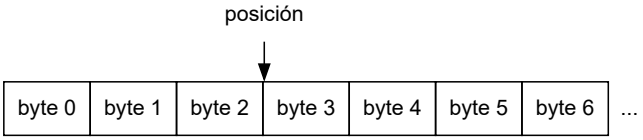
# Streams

## Conceptos generales

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 ...

Podemos considerar pues 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 ...

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`.

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.

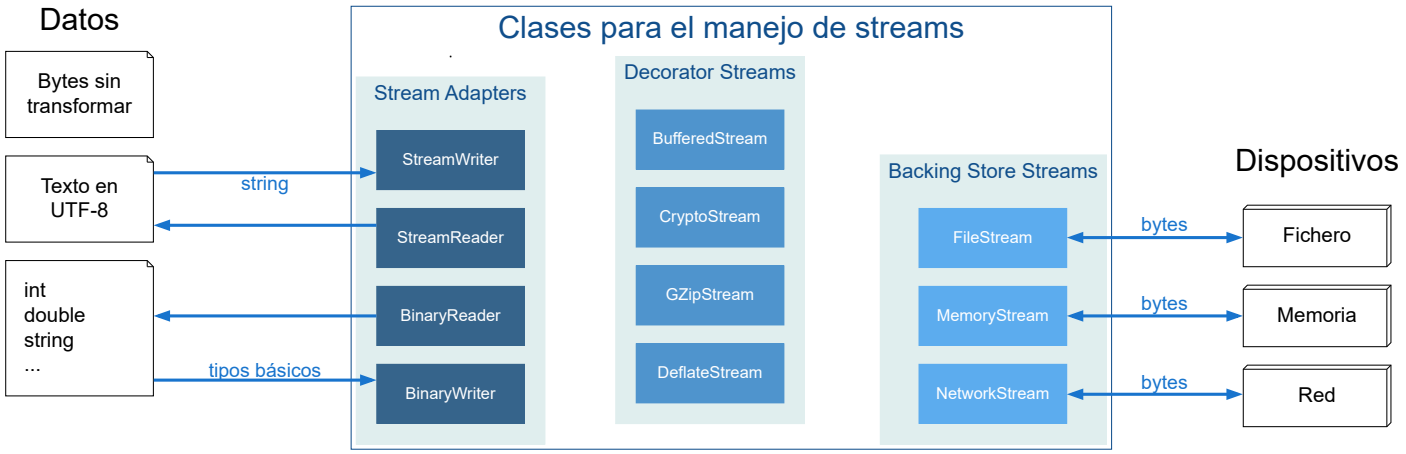
Por tanto, la operación de **escritura** en ficheros se realiza sobre un buffer de memoria RAM el cual se **'*volcará*'** en el soporte de almacenamiento por bloques para optimizar, ya sea de forma transparente o forzada.

# Streams en CSharp

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. 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 del 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.
- 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.

> **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 (
                                    path: "ejemplo.txt",      // Nombre del fichero
                                    mode: FileMode.Create,    // Lo creo o sobrescribo si existe
                                    access: FileAccess.Write);  // Solo puedo escribir en él.
            fichero.Close();
        }
    }
}
```

### Escritura de un fichero con FileStream

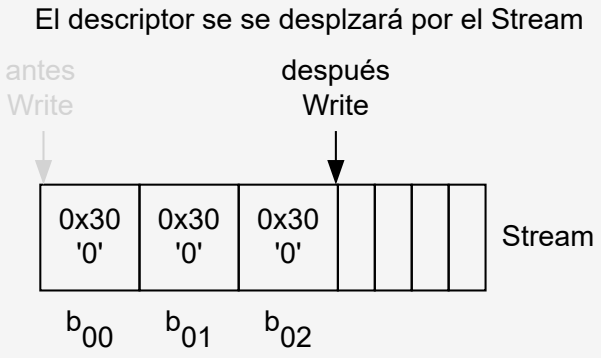
La forma escribir más común, herencia del lenguaje C es el método...

**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:**

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

// Escribo 3 bytes con el valor en hexadecimal del
// dígito '0' en UTF-8 al principio del stream.
byte[] datos = new byte[] { 0x30, 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.
- `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**.

> **Aviso**
>
> El **array** debe estar predimensionado con el espacio suficiente. Además, esta dimensión puede ser mayor al número de bytes a leer.

**Ejemplo básico:**

```
FileStream fichero = new FileStream(
                            path: "ejemplo.txt",
                            mode: FileMode.Open,      // Abro un fichero existente (sino error)
                            access: 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
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.Write($"{datos[i]:X} ");

fichero.Close();
```

**public override int fileStream.ReadByte()**

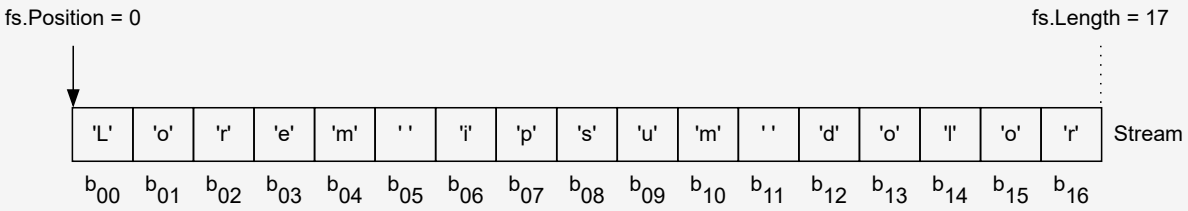
Lee un byte del archivo y avanza la posición de lectura un byte. Devuelve, el byte, convertido en un int, o **-1 si se ha alcanzado el final de la secuencia**.

### Longitud y Posición en el Stream

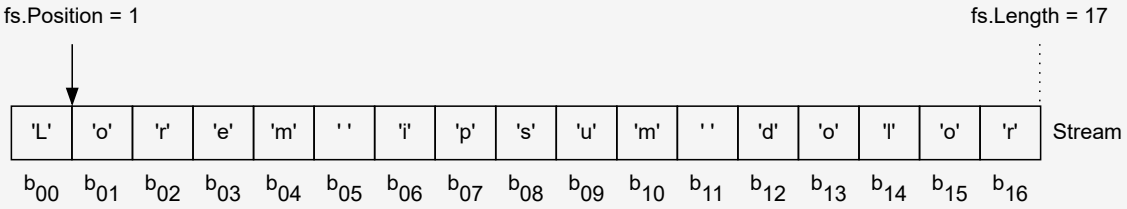
- **long Length**: Número de bytes almacenados en el flujo (tamaño del flujo).
  En ficheros, el tamaño en bytes el fichero.
- **long Position**: Número del **byte actual** en el flujo **empezando en 0**.
  En ficheros, la posición actual **donde se encuentra el descriptor** de lectura o escritura del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

**Ejemplo básico:** (Puedes descargar el código completo desde **este enlace**)

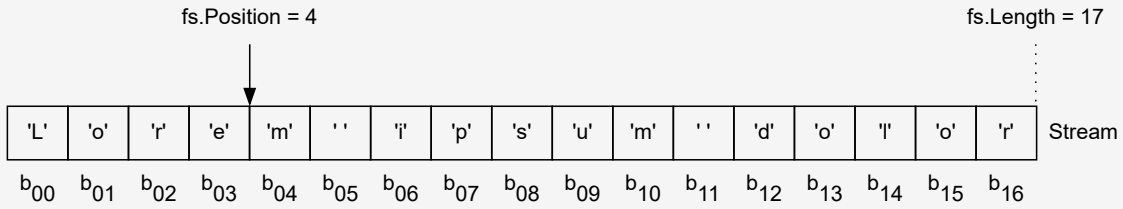
```
FileStream fs = File.Create("ejemplo.txt");
// Paso la cadena a un array de 17 bytes para poder
// escribirla con un FileStream.
byte[] buffer = Encoding.UTF8.GetBytes("Lorem ipsum dolor");
fs.Write(buffer, 0, buffer.Length);
fs.Close();

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




> **Consejo**
>
> 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 origin 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 origin, mediante el uso de un valor del `Enum SeekOrigin`.
  Estos valores pueden ser: `Begin`, `Current` y `End`

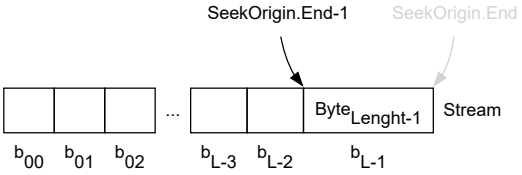
> **Aviso**
>
> No en todos los streams podremos desplazarnos con `Seek` como en los `FileStream`. Por esa razón la clase base `Stream` dispone de una propiedad `CanSeek` que me dirá si puedo desplazarme por él o no. Fíjate en el siguiente ejemplo.

**Ejemplo básico:**

```
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**:

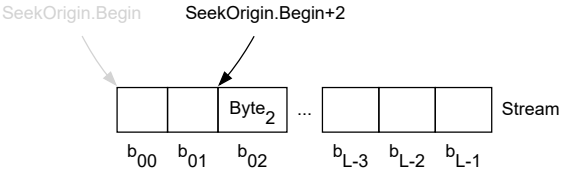
`fichero.Seek(-1, SeekOrigin.End);`




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**.

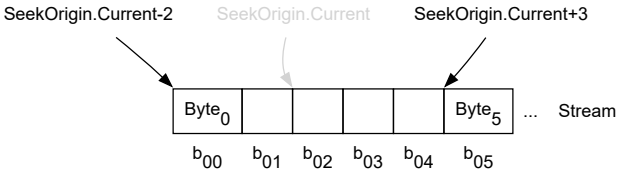
**Caso de desplazamiento 2**:

`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.

> **Peligro**
>
> 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.Write("\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.Write("\n");
    int _byte;
    while ((_byte = fs.ReadByte()) != -1)
        Console.Write($"{(char)_byte}");
    fs.Close();
}
```

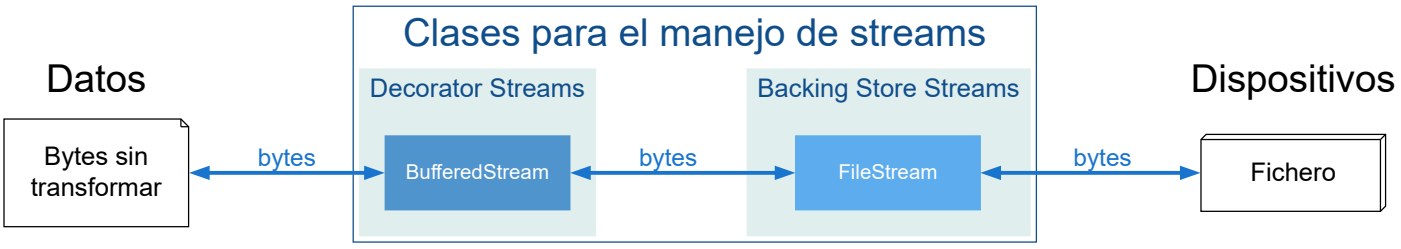
Voy **leyendo byte a byte** mientras el valor de byte leído sea distinto de -1 o positivo. Esta opción es mucho más ineficiente en términos temporales que leer bloques de bytes con Read().

## 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 heréncia.

### 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*'). Aunque `FileStream` ya tiene un buffer de escritura intermedio por defecto, nosotros **podremos ampliarlo o reducirlo** mediante esta capa de abstracción.



🎓 **Caso de estudio:**

1. Descarga el código del caso de estudio desde **este enlace** y ejecútalo para ver los tiempos de escritura con y sin `BufferedStream`.
2. Prueba a ver que sucede si aumentamos la capacidad de almacenaje del `BufferedStream` de **100** a **1000000**.
3. Prueba a ver que sucede si hacemos un `Flush()` después de cada escritura.

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

    cronometro.Start();
    FileStream fichero = new ("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 extremádamente costosa.
    BufferedStream ficheroBuff = new (fichero, 100);

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

## 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.

### ¿Por qué usar Stream Adapters?

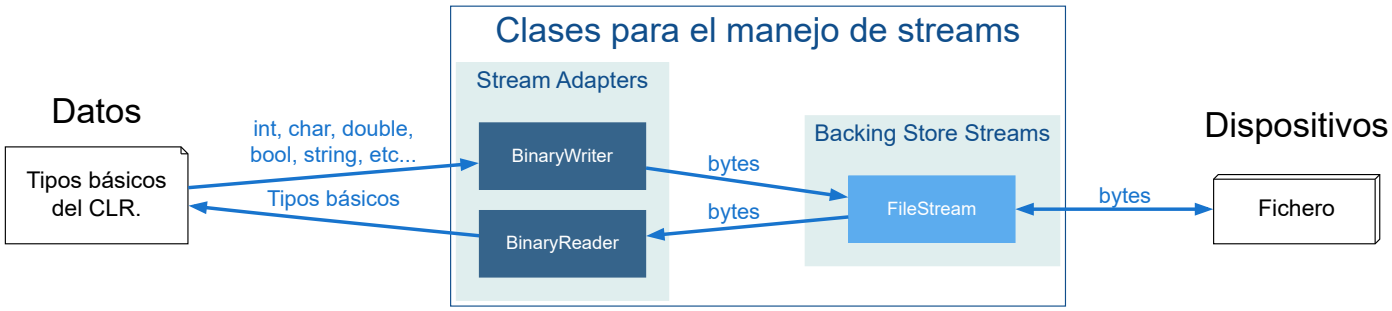
La forma de acceder a ficheros a nivel de Byte no es muy conveniente, en tanto que generalmente, en las aplicaciones no suele trabajarse directamente con bytes sino con objetos más complejos formados por múltiples bytes y/o cadenas de texto. Por esto, en `System.IO` se han incluido tipos que encapsulan `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.

### Ideas principales:

- 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 escriben o leen cadenas en una **codificación específica**.




Sus método 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`

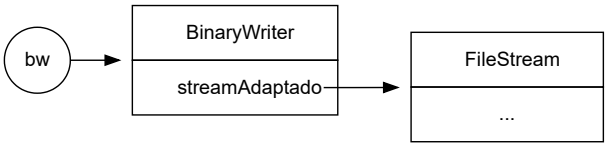
El constructor básico será **`BinaryWriter(Stream streamAdaptado, Encoding codificacionCadenas)`**

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

```
FileStream fs = new FileStream("ejemplo.bin", FileMode.Create, FileAccess.Write);
// Lo pasamos a un objeto adaptador como una agregación, indicándo 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 ("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 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`

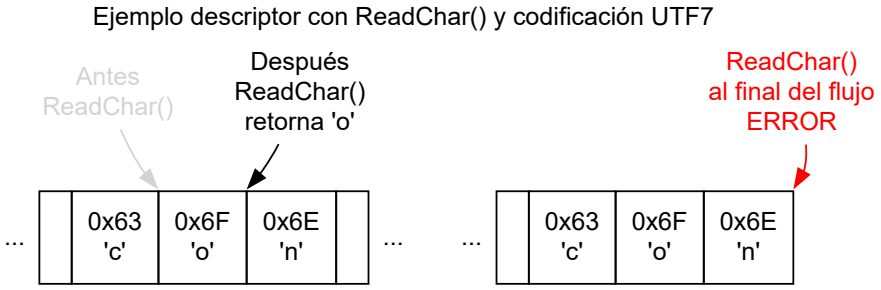
El constructor básico será `BinaryReader(Stream streamAdaptado, Encoding codificacionCadenas)`

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

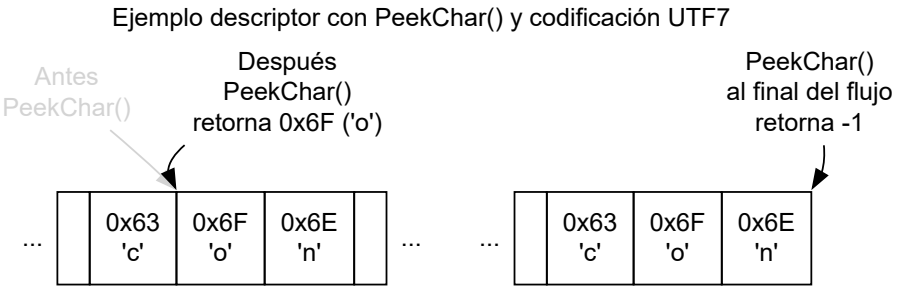
```
FileStream fs = new FileStream("ejemplo.bin", FileMode.Open, FileAccess.Read);
BinaryReader br = new BinaryReader(fs, Encoding.UTF8);

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

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

Si nos fijamos el 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 caracter del flujo y **avanzará el descriptor**. Si el descriptor se encuentra al final del mismo, se producirá una excepción. Además, 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**.




`int PeekChar()` : Mirará el siguiente caracter 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).




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 caracter de dicho byte.
**Utilizamos `PeekChar()` para detectar el final del fichero**. Si fuera un **fichero en binario** esta lectura no tendría mucho sentido pues obtendríamos caracteres no representables por consola.

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

**Ejemplo:** (Puedes descargar el código completo desde **este enlace**)

Aunque es poco común en la programación "*moderna*" el **uso de ficheros de datos en binario sin transformar**, veamos un ejemplo didáctico donde usaremos `BinaryWriter` y `BinaryReader` para crear una pequeña base de datos a modo de inventario de productos con registros de tamaño fijo. Esto es una aproximación muy básica a cómo funcionan los **SGBD** actuales y que su extendida adopción a partir de mediados de los 90 hizo que el manejo de archivos planos de datos en binario cayera en desuso.

Imagina que tienes un archivo de datos donde cada "`Producto`" ocupa exactamente el mismo espacio (por ejemplo, 76 Bytes). Gracias a `Seek`, **no necesitas leer todo el archivo para encontrar el producto**; puedes **saltar directamente a su posición física** en el disco instantáneamente a partir de una índice con una fórmula sencilla....

```
long posicion = indice * Producto.RecordSizeBytes;
```

**Paso 1**: Primero definimos el registro "`Producto`" como un **record class** con las propiedades que tendrá cada producto en nuestro inventario. Además, definimos una constante `RecordSizeBytes` que nos indicará el **tamaño en bytes fijo** que ocupará cada registro en el fichero.

```
record class Producto(int Id, string Nombre, double Precio)
{
    // Tamaño fijo en bytes (4 bytes ID + 64 bytes Nombre + 8 bytes Precio = 76 bytes)
    public const int RecordSizeBytes = 4 + 64 + 8;
    public override string ToString() => $"ID: {Id,04:D} {Nombre,-32} {Precio,8:F2} EUR";
}
```

**Paso 2**: Implementamos el método estático `EscribirProducto` que recibirá un índice y un objeto `Producto` y lo escribirá en la posición correspondiente del fichero de datos.

```
class Program
{
    const string FILE_PATH = "inventario.dat";

    static void EscribirProducto(int indice, Producto producto)
    {
        using FileStream fs = new (FILE_PATH, FileMode.OpenOrCreate, FileAccess.Write);
        using BinaryWriter writer = new (fs, Encoding.UTF8);

        // SEEK: Movemos el puntero al lugar exacto del índice
11      fs.Seek(indice * Producto.RecordSizeBytes, SeekOrigin.Begin);

        // Aseguramos que el nombre ocupa exactamente 32 caracteres
        // como cada caracter en UTF-8 ocupa 2 bytes, serán 64 bytes en total.
15      string nombreFijo = producto.Nombre.PadRight(32)[..32];

        writer.Write(producto.Id); // 4 bytes
        writer.Write(nombreFijo); // 64 bytes
        writer.Write(producto.Precio); // 8 bytes
    }
}
```

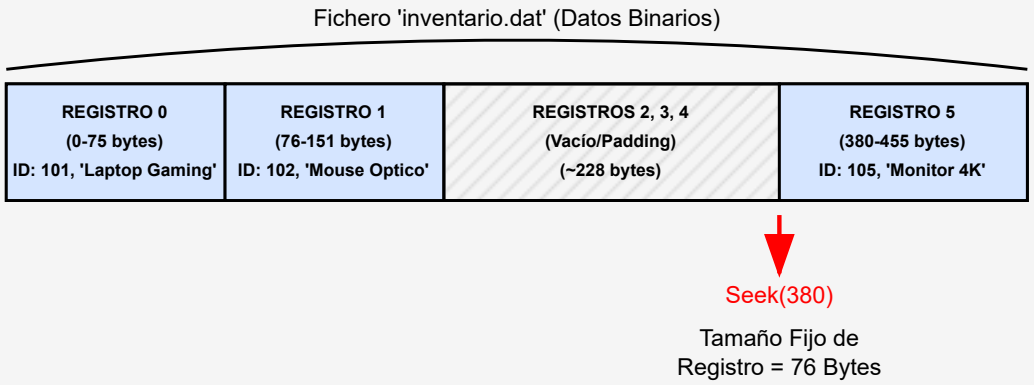
**Paso 3**: Implementamos el método estático `LeerProducto` que recibirá un índice y devolverá el objeto `Producto` leído en la posición correspondiente del fichero de datos o `null` si no existe el fichero o el registro.

```
class Program
{
    const string FILE_PATH = "inventario.dat";

    // ... código anterior ...

    static Producto? LeerProducto(int indice)
    {
        if (!File.Exists(FILE_PATH)) return null;

        using FileStream fs = new (FILE_PATH, FileMode.Open, FileAccess.Read);
        using BinaryReader reader = new (fs, Encoding.UTF8);

        long posicion = indice * Producto.RecordSizeBytes;
        if (posicion < fs.Length)
        {
            // SEEK: Saltamos directamente al registro deseado sin leer los anteriores
            fs.Seek(posicion, SeekOrigin.Begin);

            int id = reader.ReadInt32();
            string nombre = reader.ReadString().Trim();
            double precio = reader.ReadDouble();

            return new Producto(id, nombre, precio);
        }
        else
        {
            Console.WriteLine($"El registro {indice} no existe.");
            return null;
        }
    }
}
```

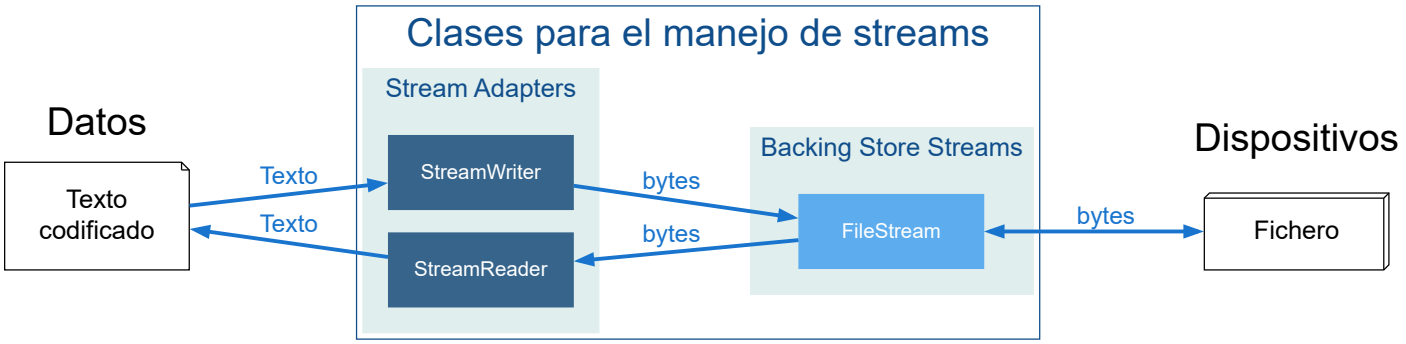
**Paso 4**: Finalmente, en el método `Main` escribimos algunos productos en el fichero y luego los leemos para mostrarlos por consola.

```
    static void Main()
    {
        EscribirProducto(0, new Producto(101, "Laptop Gaming", 1200.50));
        EscribirProducto(1, new Producto(102, "Mouse Optico", 25.99));
        // Saltamos posiciones a propósito
        EscribirProducto(5, new Producto(105, "Monitor 4K", 450.00));

        Console.WriteLine("--- Datos guardados ---");

        var producto1 = LeerProducto(5); // Leer el monitor
        if (producto1 != null)
            Console.WriteLine($"[Registro 5] {producto1}");

        var producto2 = LeerProducto(1); // Leer el mouse
        if (producto2 != null)
            Console.WriteLine($"[Registro 1] {producto2}");
    }
```

Fíjate que al guardar el producto con índice **5** hemos dejado un hueco entre los registros **1** y **5**. Los primeros SGBD en la década de los 90 usaban un fichero con los índices de los registros **para evitar leer estos huecos** y localizar rápidamente los registros en el fichero de datos.



### Stream Adapters StreamWriter y StreamReader




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.

Tendremos pues **dos adaptadores principales**:

1. `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**).
2. `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`.

> **Cuidado**
>
> 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.

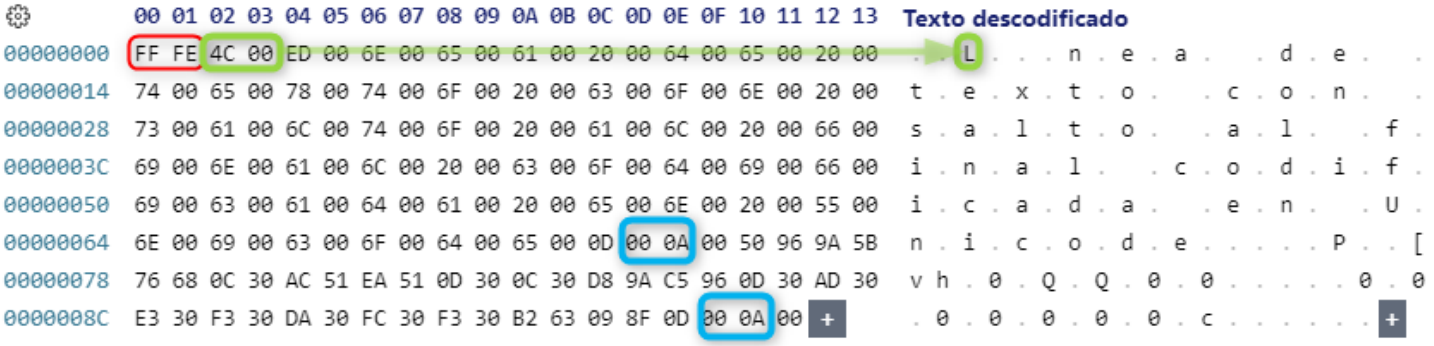
#### Adaptador `StreamWriter`

```
FileStream fs = new ("ejemplo.txt", FileMode.Create, FileAccess.Write);
StreamWriter sw = new (fs, Encoding.Unicode);
// 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
// serian bien interpretados los siguiente caracteres.
sw.WriteLine("限定桶「冬凪」「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);  // UTF-16
sw.WriteLine("Línea de texto con salto al final codificada en Unicode");
sw.WriteLine("限定桶「冬凪」「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 un editor hexadecimal** como el plugin de VSCode Hex Editor de Microsoft para ver que secuencia de bytes ha escrito nos encontraremos que cada carácter ocupa **2 bytes** (16 bits).




Donde `FF FE` serán los `bytes de BOM` indicando que los caracteres a continuación están codificados en Unicode (UTF-16), 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`.

> **Nota**
>
> `FF FE` es el BOM por defecto de **UTF-16** en **Little Endian** (UTF-16 LE). Si fuera **UTF-16** en **Big Endian** (UTF-16 BE) sería `FE FF` y el caracter `'L'` del ejemplo anterior se codificaría `00 4C`. Si fuera **UTF8** el BOM sería `EF BB BF` y aunque parezca que no es necesario porque UTF-8 solo ocupa un byte esto no es así porque algunos símbolos pueden ocupar más de un byte.
>
> Este tema del BOM es más complejo de lo que en este tema se ha intentado resumir y simplificar, por lo que puede ser un poco confuso ya es el resultado de diferentes convenios a lo largo del tiempo. Puedes ahondar más en el tema en los siguientes enlaces:
>
> - **Marca de orden de bytes en Wikipedia**.
> - **RFC-3629 de la IETF**

Además, si nos fijamos en el correspondiente hexadecimal, los caracteres que 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).

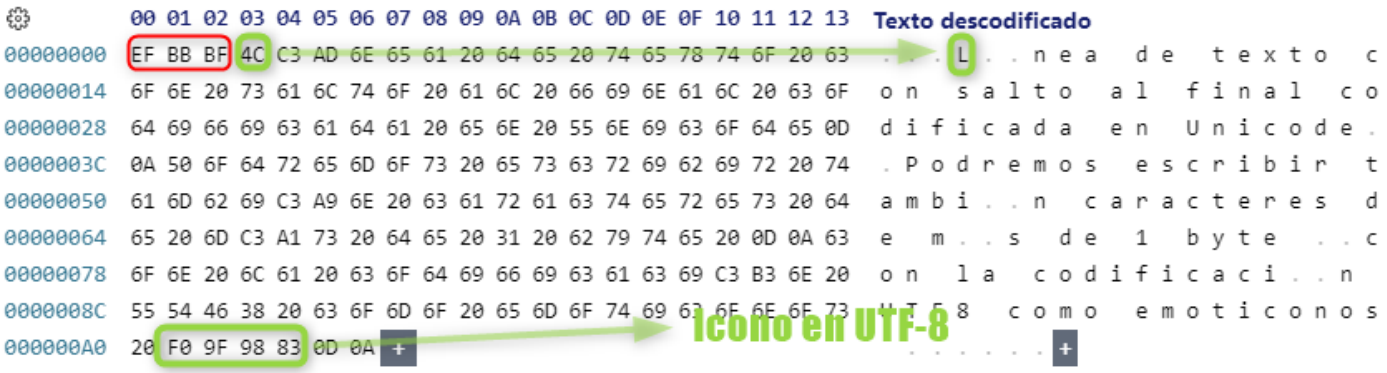
Dependiendo del SO este **salto de línea** o `\n` se codificará de diferente manera.

| SO | Caracter/es | Hexadecimal |
|---|---|---|
| Windows | CR+LF | `0D 0A` |
| Linux, Unix, Android, iOS, MacOS X | LF | `0A` |
| Hasta Mac OS 9 (Ya no se usa) | CR | `0D` |

Si usamos otra codificación como **UTF-8**, **La mayoría** de los caracteres ocuparían un solo byte, pero **algunos caracteres especiales** ocuparían más de un byte. De esta manera el fichero ocupará menos espacio en disco si es un texto en inglés o español, donde la mayoría de los caracteres ocupan un solo byte y el segundo se rellenaría a ceros como en el ejemplo de antes donde `00 4C = L` en UTF-16 y `4C = L` en UTF-8.

Por **ejemplo**, si guardamos en **UTF-8** ...

```
const bool AÑADIR_AL_FINAL = false;
using StreamWriter sw = new StreamWriter("ejemplo.txt", AÑADIR_AL_FINAL, Encoding.UTF8); // UTF-8
sw.WriteLine("Línea de texto con salto al final codificada en Unicode");
sw.WriteLine("Podremos escribir también caracteres de más de 1 byte ");
sw.WriteLine("con la codificación UTF8 como emoticonos 😀");
```




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

Fíjate que los bytes de BOM para representar UTF8 ahora son los que se comentaban en la nota anterior `EF BB BF` y los saltos de línea si has ejecutado en un SO Windows son `0D 0A`. Sin embargo, el carater que representaba el emoticono 😀 ocupará **4 bytes** en UTF-8 `F0 9F 98 83` y en la lectura C# sabrá por el BOM que tiene que leer los 4.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 **trasparente** 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 ("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 por nosotros.
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. 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. 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 ("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).

> ⚠ **Cuidado**
>
> Si el **fichero es muy grande**, podemos obtener un error de **memoria insuficiente**.
> 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 ("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 y streams

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 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
        EscribeFichero();
    }
    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 `EscribeFichero()` 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 cundo busquemos el uso de Streams en C# por Internet.

```
static void EscribeFichero()
{
    const bool AÑADIR_AL_FINAL = true;
    using (StreamWriter sw = new(@".\\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, a partir de C# 8 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 `EscribeFichero()` de la siguiente forma...

```
static void EscribeFichero()
{
    const bool AÑADIR_AL_FINAL = true;
    using StreamWriter sw = new(@".\\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.
}
```

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

# Serilización y deserialización a CSV

Los archivos CSV (del inglés **Comma-Separated Values**) son un tipo de documento en formato abierto sencillo para representar datos en forma de tabla, en las que las columnas se separan por comas (o punto y coma en donde la coma es el separador decimal: Chile, Perú, Argentina, España, Brasil...) y las filas por saltos de línea.

El formato CSV es muy sencillo y no indica un juego de caracteres concreto, ni cómo van situados los bytes, ni el formato para el salto de línea. **Estos puntos deben indicarse muchas veces al abrir el archivo, por ejemplo, con una hoja de cálculo**.

El **formato CSV no está estandarizado**. La idea básica de separar los campos con una coma es muy clara, pero se vuelve complicada cuando el valor del campo también contienen comillas dobles o saltos de línea. Las implementaciones de CSV pueden no manejar esos datos, o usar comillas de otra clase para envolver el campo. Pero esto no resuelve el problema: algunos campos también necesitan embeber estas comillas, así que las implementaciones de CSV pueden incluir caracteres o secuencias de escape.

Además, se suele crear un primer registro de cabeceras que indica el nombre de los campos de cada columna:

```
idpro,descrip,precio
1,productoA,50.0
2,productoB,90.0
3,productoC,40.0
```

## Separadores de campos alternativos

Además, el término "CSV" también denota otros formatos de valores separados por delimitadores que usan delimitadores diferentes a la coma (como los valores separados por tabuladores). Un delimitador que no está presente en los valores de los campos (como un tabulador) mantiene el formato simple. Estos archivos separados por delimitadores alternativos reciben en algunas ocasiones la extensión aunque este uso sea incorrecto.

Esto puede causar problemas en el intercambio de datos, por ello muchas aplicaciones que usan archivos CSV tienen opciones para cambiar el carácter delimitador.
Por ejemplo, un separador muy utilizado es el carácter '`|`'

```
idpro|descrip|precio
1|productoA|50.0
2|productoB|90.0
3|productoC|40.0
```

Este tipo de archivos puede abrirse y crearse desde hojas de cálculo. Crea un archivo denominado coches.csv con el siguiente contenido y ábrelo con la hoja de cálculo Calc.

```
Año,Marca,Modelo,Descripción,Precio
1997,Ford,E350,"ac, abs, moon",3000.00
1999,Chevy,Venture,Extended Edition,4900.00
1999,Chevy,Venture,"Extended Edition, Very Large",5000.00
1996,Jeep,Grand Cherokee,"MUST SELL! air, moon roof, loaded",4799.00
```

Verás que lo representa en una tabla con las columnas correspondientes.

| Año | Marca | Modelo | Descripción | Precio |
|---|---|---|---|---|
| 1997 | Ford | E350 | ac, abs, moon | 3000.00 |
| 1999 | Chevy | Venture | Extended Edition | 4900.00 |
| 1999 | Chevy | Venture | Extended Edition, Very Large | 5000.00 |
| 1996 | Jeep | Grand Cherokee | MUST SELL! air, moon roof, loaded | 4799.00 |

# CSV desde CSharp

Como sucede en otros lenguajes de programación, en C# no existe una clase específica para gestionar ficheros CSV. No obstante, usando las clases de `StreamWriter` y `StreamReader` podemos crear nuestras propias clases para gestionar este tipo de ficheros. Además, en cualquier lenhuage existen librerías de terceros que nos facilitan la gestión de este tipo de ficheros.

Veamos pues, a través de una ejemplo esando `StreamWriter`, `StreamReader` y excepciones, cómo podríamos gestionar un fichero CSV con datos de alumnos. Puedes descargar el código completo desde **este enlace**

Vamos a escribir un programa que gestione datos de **Alumnos** y los almacene en un **fichero CSV** con la siguiente estructura:

```
Nia,Apellidos,Nombre
1,"Sánchez López","María Jesús"
2,"Pérez Reverte","Juan José"
```

**Veamos pues los pasos necesarios para implementar la gestión de este fichero CSV:**

**Paso 1**: Definiremos la clase `AlumnoEntity` cuya **única responsabilidad** será gestionar los datos de la entidad **Alumno**.

```
public class AlumnoEntity
{
    public long Nia { get; }
    public string Apellidos { get; }
    public string Nombre { get; }

    public AlumnoEntity(long nia, string apellidos, string nombre)
    {
        Nia = nia;
        Apellidos = apellidos;
        Nombre = nombre;
    }

    public override string ToString() => $"{Nia,-8}{Apellidos,-32}{Nombre,-32}";
}
```

**Paso 2**: Definiremos la clase estática `AlumnoCSV` que contendrá los métodos necesarios para gestionar el fichero CSV.

Definimos primero el método privado `Lee(StreamReader sr)` que leerá un registro del fichero CSV y lo devolverá como objeto de `AlumnoEntity`. Esta función procesará una línea del fichero CSV, dividiéndola con un `Split` por el separador definido. Para ello, supone que en los campos de texto que pueden contener el separados. Además, como los campos de texto pueden contener comillas dobles, hacemos un `Trim` para eliminarlas.

```
public static class AlumnoCSV
{
    const string SEP = ",";

    private static AlumnoEntity Lee(StreamReader sr)
    {
        string linea = sr.ReadLine()
        ?? throw new NullReferenceException("No se ha podido leer Alumno en el CSV");
        string[] campos = linea.Split(SEP);

        if (campos.Length < 3)
            throw new FormatException("El formato del CSV no es válido");
        for (int i = 1; i < campos.Length; i++)
            campos[i] = campos[i].Trim('"');
        return new AlumnoEntity(
                        nia: long.Parse(campos[0]),
                        apellidos: campos[1],
                        nombre: campos[2]);
    }
}
```

A continuación, implementamos el método `Lee(string ruta)` que abrirá el fichero CSV y devolverá una **secuencia perezosa (`IEnumerable<AlumnoEntity>`)** con todos los registros del mismo como objetos de

tipo `AlumnoEntity`. Para ello, usaremos el método privado anterior. Fíjate además que, al principio, **saltamos la primera línea del fichero que contiene la cabecera con los nombres de las columnas**.

```
public static class AlumnoCSV
{
    // ... código omitido por brevedad
    public static IEnumerable<AlumnoEntity> Lee(string ruta)
    {
        using FileStream stream = new(ruta, FileMode.Open, FileAccess.Read);
        using StreamReader sr = new(stream, Encoding.UTF8);
        sr.ReadLine(); // Saltamos las columnas de la cabecera
        while (!sr.EndOfStream) yield return Lee(sr);
    }
}
```

Implementamos ahora un método de extensión `Guarda(this AlumnoEntity alumno, string ruta)` que añadirá un nuevo registro al final del fichero CSV. Fíjate que si estmos añadiendo el primer registro, tendremos que escribir antes la **cabecera con los nombres de las columnas**. Además, usamos el mismo separador definido en la lectura y añadimos las comillas dobles a los campos de texto.

```
public static class AlumnoCSV
{
    // ... código omitido por brevedad
    public static void Guarda(this AlumnoEntity alumno, string ruta)
    {
        using FileStream stream = new(ruta, FileMode.Append, FileAccess.Write);
        using StreamWriter sw = new(stream, Encoding.UTF8);

        if (stream.Length == 0)
            sw.WriteLine($"Nia{SEP}Apellidos{SEP}Nombre");

        sw.WriteLine($"{alumno.Nia}{SEP}\"{alumno.Apellidos}\"{SEP}\"{alumno.Nombre}\"");
        sw.Flush();
    }
}
```

Por último, implementamos el método `Busca(long nia, string ruta)` que buscará un alumno por su NIA en el fichero CSV y devolverá el objeto `AlumnoEntity` correspondiente o `null` si no lo encuentra. Para ello, usaremos el método privado de lectura de registros.

```
public static class AlumnoCSV
{
    // ... código omitido por brevedad
    public static AlumnoEntity? Busca(long nia, string ruta)
    {
        AlumnoEntity? alumno = default;
        using FileStream stream = new(ruta, FileMode.Open, FileAccess.Read);
        using StreamReader sr = new(stream, Encoding.UTF8);

        sr.ReadLine();
        while (!sr.EndOfStream && alumno == null)
        {
            AlumnoEntity a = Lee(sr);
            if (nia == a.Nia)
                alumno = a;
        }
        return alumno;
    }
}
```

**Paso 3**: Vamos a implementar el programa principal basándonos en el código anterior.

Esta implementación, ya es más trivial y solo requiere de unos pocos métodos para gestionar el menú, la lectura de opciones y las diferentes funcionalidades.

Primero, implementamos el método `Menu()` que mostrará las opciones del menú por consola.

```
public class Program
{
    public static void Menu()
    {
        Console.Clear();
        Console.WriteLine("""
                1 - Introduce Alumno.
                2 - Mostrar Alumnos.
                3 - Buscar Alumno.
                4 - Salir.
                """);
    }
}
```

En segundo lugar, implementamos el método `Lee()` que leerá la opción del menú seleccionada por el usuario y validará que es correcta.

```
public class Program
{
    // ... código omitido por brevedad
    public static int Lee()
    {
        bool válida;
        int o = 4;
        do
        {
            Console.Write("Introduce una opción: ");
            válida = int.TryParse(Console.ReadLine(), out int valor);
            if (válida)
            {
                válida = valor >= 1 && valor <= 4;
            }
            if (válida)
                o = valor;
            else
                Console.WriteLine("Opción incorrecta!!");
        }
        while (!válida);
        return o;
    }
}
```

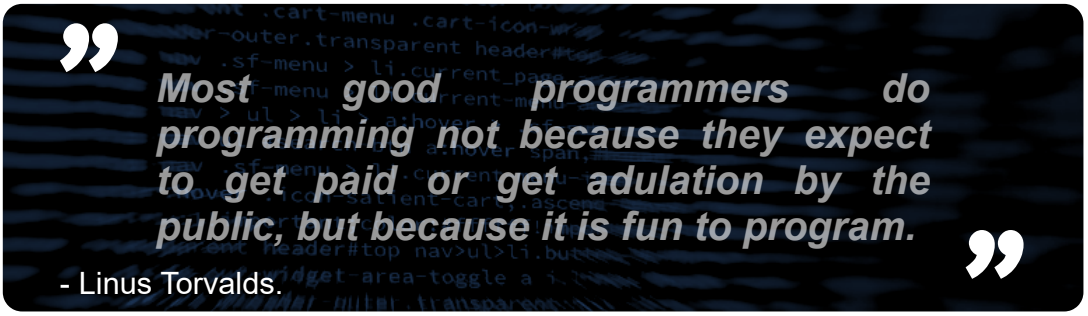
Implementamos ahora el método **`IntroduceAlumno(string fichero)`** que pedirá los datos del alumno por consola y lo guardará en el fichero CSV usando el método de extensión definido anteriormente.

```
public class Program
{
    // ... código omitido por brevedad
    public static void IntroduceAlumno(string fichero)
    {
        Console.Write("NIA: ");
        long nia = long.Parse(Console.ReadLine() ?? "0");
        Console.Write("Nombre: ");
        string nombre = Console.ReadLine() ?? "";
        Console.Write("Apellido: ");
        string apellido = Console.ReadLine() ?? "";

        AlumnoEntity alumno = new(nia, apellido,  nombre);
        alumno.Guarda(fichero);
        Console.WriteLine("Datos guardados.");
    }
}
```

Implementamos el método **`BuscaAlumno(string fichero)`** que pedirá el NIA del alumno a buscar y mostrará sus datos si lo encuentra o un mensaje indicándolo si no existe.

```
public class Program
{
    // ... código omitido por brevedad
    public static void BuscaAlumno(string fichero)
    {
        string salida;
        if (File.Exists(fichero))
        {
            Console.Write("NIA a buscar: ");
            long nia = long.Parse(Console.ReadLine() ?? "0");

            AlumnoEntity? a = AlumnoCSV.Busca(nia, fichero);
            if (a != null)
                salida = $"Los datos del alumno con nia {nia} son:\n{a}";
            else
                salida = $"No existe ningún alumno de nia {nia}.";
        }
        else
            salida = $"El fichero de datos {fichero} aún no se ha creado.";

        Console.WriteLine(salida);
    }
}
```

El programa principal `Main()` implementa el típico bucle del menú y llama a los métodos anteriores según la opción seleccionada. Además, gestiona las excepciones que puedan producirse durante la ejecución.



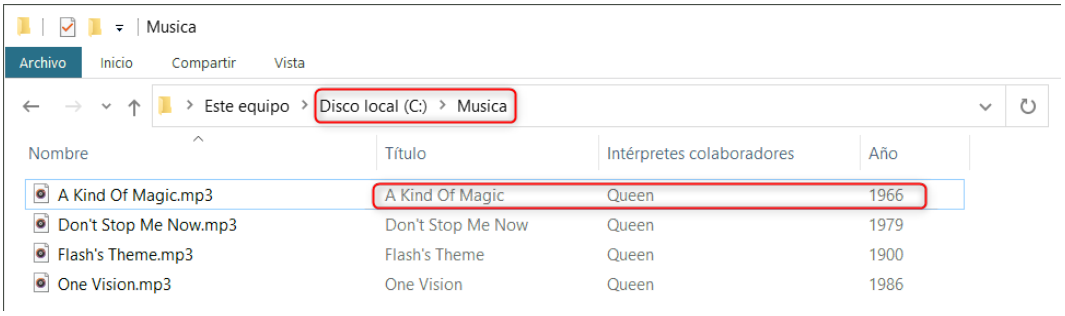
```
public class Program
{
    // ... código omitido por brevedad
    public static void Main()
    {
        const string FICHERO = "alumnos.csv";
        int opcionMenu = 4;

        do
        {
            bool fin = true;
            try
            {
                Menu();
                opcionMenu = Lee();
                switch (opcionMenu)
                {
                    case 1:
                        IntroduceAlumno(FICHERO);
                        break;
                    case 2:
                        string salida;
                        salida = File.Exists(FICHERO)
                                 ? string.Join("\n", AlumnoCSV.Lee(FICHERO))
                                 : $"El fichero de datos {FICHERO} aún no se ha creado.";
                        Console.WriteLine(salida);
                        break;
                    case 3:
                        BuscaAlumno(FICHERO);
                        break;
                    case 4:
                        fin = false;
                        break;
                }
            }
            catch (Exception e)
            {
                Console.WriteLine($"Se ha producido un error: {e.Message}");
            }
            if (fin)
            {
                Console.Write("Pulsa una tecla...");
                Console.ReadKey();
            }
        } while (opcionMenu != 4);
    }
}
```

# Anexo I: Ejemplo de lectura binaria opcional

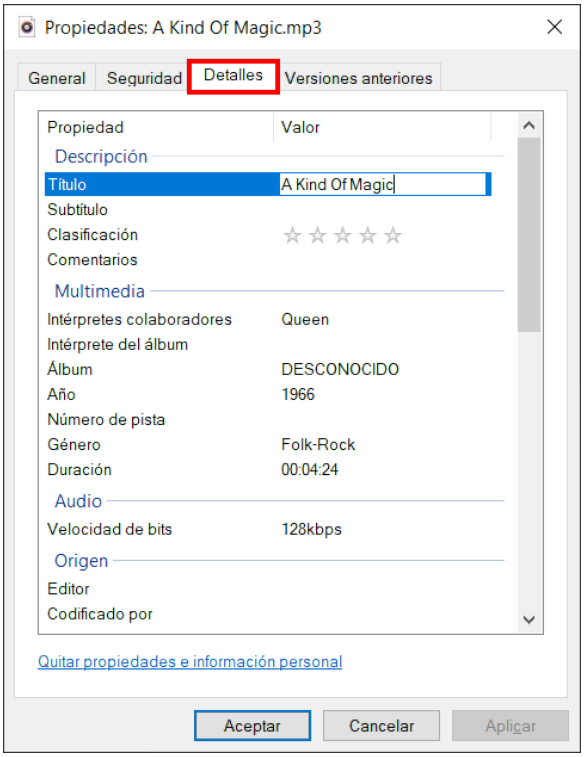
En este anexo vamos a ver un ejemplo de uso de lectura binaria adicional para extraer las etiquetas de información que pueden tener los archivos de audio en formato **mp3** antiguos. **Este ejemplo es complementario al tema y no es obligatorio su estudio. Simplemente es por si te interesa profundizar en el manejo de ficheros binarios**.

Puedes descrargar el código completo del ejemplo y los recursos necesarios para seguirlo en el **siguiente enlace**.

Imaginemos que tenemos una serie de archivos de audio en formato **mp3** en la carpeta `C:\Musica`. Si los examinamos con el explorador de archivos de Windows (**Mostrando las columnas adecuadas**). Podemos ver que nos muestra cierta información de Título, Álbum, etc.




Es más **podemos editar y modificar esta información** a través del propio Windows 10 a través de la pestaña **Detalles**, haciendo click con el botón derecho sobre el fichero y seleccionando la opción del menú contextual **Propiedades**.




Hay varias formas de guardar estas etiquetas de información en un archivo mp3 pero la usada para estos es la más simple y se denomina **TAGID3V1**.

### ¿Cómo se guarda la información del TAGID3V1?

De estar incluida esta información, será al final del archivo:
Se añaden 128 bytes fijos:

- `TagID`: 3 caracteres **fijos** 'T', 'A' y 'G'
  Estos bytes me indicarán si en esos 128 bytes del final del mp3 realmente hay información del TAGID3V1 o es simple codificación de audio.

- `Título` : 30 caracteres
- `Artista` : 30 caracteres
- `Álbum` : 30 caracteres
- `Año` : 4 caracteres
- `Comentario` : 30 caracteres
- `Género` : 1 byte
  Será un índice natural entre 0 y 79 codificado en 1byte de tal manera que:ç...
  0 → BLUES, 1 → CLASSIC ROCK, … , 79 → HARD ROCK y si hubiera un valor superior a 79 se calificaría como OTHERS.

> **Importante**
>
> Si te fijas 3 + 30 + 30 + 30 + 4 + 30 + 1 = **128**.

Si examinamos el código en binario del final del fichero editado en el diálogo anterior tendremos:

```
...
0040b020: FF FF FF 54 41 47 41 20 4B 69 6E 64 20 4F 66 20    ...TAGA.Kind.Of.
0040b030: 4D 61 67 69 63 20 20 20 20 20 20 20 20 20 20 20    Magic..........
0040b040: 20 20 20 20 51 75 65 65 6E 20 20 20 20 20 20 20    ....Queen.......
0040b050: 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20    ................
0040b060: 20 20 44 45 53 43 4F 4E 4F 43 49 44 4F 20 20 20    ..DESCONOCIDO...
0040b070: 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20    ................
0040b080: 31 39 36 36 20 20 20 20 20 20 20 20 20 20 20 20    1966............
0040b090: 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20    ................
0040b0a0: 20 20 0C
```

**Veamos los pasos a seguir para implementar la lectura de esta información en C#:**

**Paso 1**: Vamos a definir una clase en C# denominada `TagId3v1`, para leer esta información en los ficheros. En ella en primer lugar **definiremos constantes** para los tamaños reservados para cada dato según la especificación anterior y un array '`generosTagID3v1`' de solo lectura con los nombres de los géneros conforme a la especificación de TAGID3V1.

```
public record class TagId3V1
{
    const int BYTES_TAG_ID = 3;
    const int BYTES_TITULO = 30;
    const int BYTES_ARTISTA = 30;
    const int BYTES_ALBUM = 30;
    const int BYTES_AÑO = 4;
    const int BYTES_COMENTARIO = 30;
    const int BYTES_GENERO = 1;
    const int TOTAL_BYTES = BYTES_TAG_ID + BYTES_TITULO + BYTES_ARTISTA +
                            BYTES_ALBUM + BYTES_AÑO + BYTES_COMENTARIO + BYTES_GENERO;
```

```
    private static readonly string[] generosTagID3v1 =
    [
        "BLUES", "CLASSIC ROCK", "COUNTRY", "DANCE", "DISCO", "FUNK", "GRUNGE", "HIP-HOP", "JAZZ",
        "METAL", "NEW AGE", "OLDIES", "OTHER", "POP", "R&B", "RAP", "REGGAE", "ROCK", "TECHNO",
        "INDUSTRIAL", "ALTERNATIVE", "SKA", "DEATH METAL", "PRANKS", "SOUNDTRACK",
        "EURO-TECHNO", "AMBIENT", "TRIP-HOP", "VOCAL", "JAZZ+FUNK", "FUSION", "TRANCE",
        "CLASSICAL", "INSTRUMENTAL", "ACID", "HOUSE", "GAME", "SOUND CLIP", "GOSPEL", "NOISE",
        "ALTERN ROCK", "BASS", "SOUL", "PUNK", "SPACE", "MEDITATIVE", "INSTRUMENTAL POP",
        "INSTRUMENTAL ROCK", "ETHNIC", "GOTHIC", "DARKWAVE", "TECHNO-INDUSTRIAL", "ELECTRONIC",
        "POP-FOLK", "EURODANCE", "DREAM", "SOUTHERN ROCK", "COMEDY", "CULT", "GANGSTA", "TOP 40",
        "CHRISTIAN POP", "POP/FUNK", "JUNGLE", "NATIVE AMERICAN", "CABARET", "NEW WAVE",
        "PSYCHADELIC", "RAVE", "SHOWTUNES", "TRAILER", "LO-FI", "TRIBAL", "ACID PUNK", "ACID JAZZ",
        "POLKA", "RETRO", "MUSICAL", "ROCK & ROLL", "HARD ROCK"
    ];
}
```

**Paso 2**: A continuacíon vamos a añadir a la clase `TagId3v1` las **propiedades públicas** para acceder a los campos del TAGID3V1. Puesto

```
public record class TagId3V1
{
    // ... código omitido por abreviar

    public string Titulo { get; }
    public string Artista { get; }
    public string Album { get; }
    public ushort Año { get; }
    public byte Genero { get; }
    public string Comentario { get; }
    public string EtiquetaGenero => Genero >= 80 ? "OTHER" : generosTagID3v1[Genero];
}
```

**Paso 3**: **Constructor privado** que reciba todos los campos (excepto, claro está, el array de géneros). El hecho de que sea privado es porque no quiero que nadie pueda crear instáncias de la clase directamente. Más adelante implementaremos un método factoría estático que se encargue de leer el TAGID3V1 de un fichero y crear la instáncia correspondiente.

Además, vamos a invalidar el método `ToString` para que muestre los datos del objeto.

```
public record class TagId3V1
{
   // ... código omitido por abreviar
   private TagId3V1(
               string titulo, string artista,
               string album, ushort año,
               byte genero, string comentario)
   {
       Titulo = titulo;
       Artista = artista;
       Album = album;
       Año = año;
       Genero = genero;
       Comentario = comentario;
   }
```

```
    public override string ToString() => $"""
                {"Titulo:",11} {Titulo}
                {"Artista:",11} {Artista}
                {"Album:",11} {Album}
                {"Año:",11} {Año}
                {"Genero:",11} {EtiquetaGenero}
                {"Comentario:",11} {Comentario}
                """;
}
```

**Paso 4**: Sería interesante implementar un método de clase público `HayTAG` que **reciba un fichero** y me devuelva un boolenao indicándome si el fichero contiene un TAGIDID3 válido. Una posible implementación sería...

```
public record class TagId3V1
{
    public static bool HayTAG(string rutaFichero)
    {
        using FileStream stream = new(rutaFichero, FileMode.Open, FileAccess.Read, FileShare.Read);
        using BinaryReader streamRB = new(stream);

        bool hayTAG = false;
        // Habrá TAG si como mínimo el fichero mide más de los 128 bytes que ocupa el mismo.
        hayTAG = stream.Length > TOTAL_BYTES;
        if (hayTAG)
        {
            // Se que ocupa esos bytes y me desplazo donde en teoría está el principio del TAG
            stream.Seek(-TOTAL_BYTES, SeekOrigin.End);
            // Leo los 3 caracteres del TAG y los transformo a cadena.
            string tag = new(streamRB.ReadChars(BYTES_TAG_ID));
            hayTAG = tag == "TAG";
        }
        return hayTAG;
    }
}
```

**Paso 5**: Vamos a crear un método de método de clase `LeeTAG` y que reciba la **ubicación de una fichero fichero mp3** y que genere una excepción `TagId3v1Exception` si no hay tag y si lo hay, lo lea devolviendo una instancia de la clase `TagId3v1` con los datos del mismo.

Si durante la lectura se genera una excepción. Esta se relanzará '*envuelta*' en un **`TagId3v1Exception`** que indique de que fichero se estaba intentando leer el TAG.

```
// Definiremos la clase con la excepción personalizada fuera de la clase TagId3v1
public class TagId3v1Exception : Exception
{
    public TagId3v1Exception(string message) : base(message) { ; }
    public TagId3v1Exception(string message, Exception innerException)
    : base(message, innerException) { ; }
}
```

```
public record class TagId3V1
{
    // ... código omitido por abreviar

    public static TagId3V1 LeeTAG(string rutaFichero)
    {
        string error = $"El fichero {rutaFichero} no tiene información válida de TagId3v1";
        // Si llamo a este método es porque estoy seguro de que hay 'tag' para leer
        // en caso contrario generaré una excepción.
        if (!HayTAG(rutaFichero))
            throw new TagId3v1Exception(error);

        using FileStream stream = new(rutaFichero, FileMode.Open, FileAccess.Read);
        using BinaryReader streamBR = new(stream);
        try
        {
            // Me sitúo al principio del título justo después de los bytes ['T']['A']['G']
            stream.Seek(BYTES_TAG_ID - TOTAL_BYTES, SeekOrigin.End);

            // Como pude haber espácios hasta rellenar lo que ocupa cada valor
            // los recorto con el método Trim
            string titulo = new string(streamBR.ReadChars(BYTES_TITULO)).Trim();
            string artista = new string(streamBR.ReadChars(BYTES_ARTISTA)).Trim();
            string album = new string(streamBR.ReadChars(BYTES_ALBUM)).Trim();
            ushort año = ushort.Parse(new string(streamBR.ReadChars(BYTES_AÑO)).Trim());
            string comentario = new string(streamBR.ReadChars(BYTES_COMENTARIO)).Trim();
            byte genero = streamBR.ReadByte();

            return new TagId3V1(titulo, artista, album, año, genero, comentario);
        }
        catch (Exception e)
        {
            throw new TagId3v1Exception(error, e);
        }
    }
}
```

**Paso 6**: Vamos ahora a realizar un pequeño programa principal, usando métodos de la clase TagId3v ...

Leeremos todos archivos mp3 que encuentre en el directorio `C:\Musica` y mostrará la información del TAGID3V1 si la tuviera y "*No contiene información*" en caso contrario.

Además, controlaremos cualquier `TagId3v1Exception` que se produzca al leer un fichero, nos recuperaremos y continuaremos intentado leer el siguiente.

```
class Program
{
    static void Main()
    {
        string[] ficheros = Directory.GetFiles(@"C:\Musica", "*.mp3");
        foreach (var fichero in ficheros)
        {
            try
            {
                Console.WriteLine($"Información del fichero {fichero}...");
                string informacion = (TagId3V1.HayTAG(fichero)
                                    ? TagId3V1.LeeTAG(fichero).ToString()
                                    : "\tNo contiene información."
                                        ) + "\n";
                Console.WriteLine(informacion);
            }
            catch (TagId3v1Exception e)
            {
                Console.WriteLine(e.Message);
            }
        }
    }
}
```