Tema 8.2

Descargar estos apuntes en pdf o html

Índice

- Flujos de datos en serie o streams
  - Conceptos generales sobre Streams
    - Operaciones y nomenclatura relacionada con Streams
      - Apertura (Open)
      - Cierre (Close)
      - Lectura (Read) y Escritura (Write)
      - Volcado (Flush)
  - Streams en CSharp
    - Lectura y escritura de ficheros sin transformar
      - Apertura y Cierre de un fichero con FileStream
      - Escritura de un fichero con FileStream
      - Lectura de un fichero con FileStream
      - Longitud y Posición en el Stream
      - Desplazándonos por el Stream
      - Recorriendo un FileStream hasta el final
    - Pasando el flujo por un '*Decorator Stream*'
      - Ejemplo de '*Decorator Stream*' usando BufferedStream
    - Transformando el flujo con un '*Stream Adapter*'
      - Stream Adapters BinaryWriter y BinaryReader
      - Stream Adapters StreamWriter y StreamReader
- Manejo de excepciones con ficheros

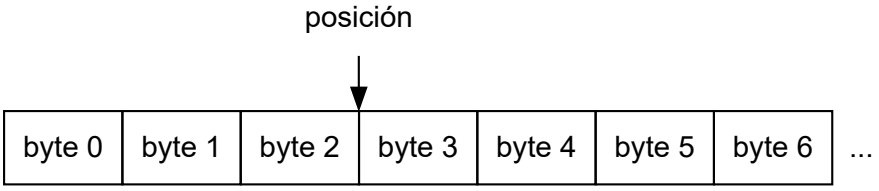
Flujos de datos en serie o streams

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.

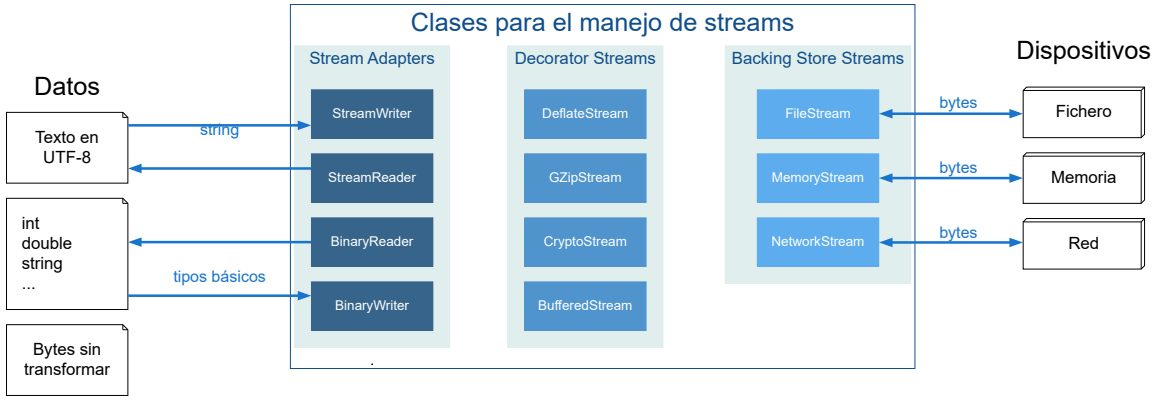
Streams en CSharp

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.

> 📌 **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 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.
- ✋ **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

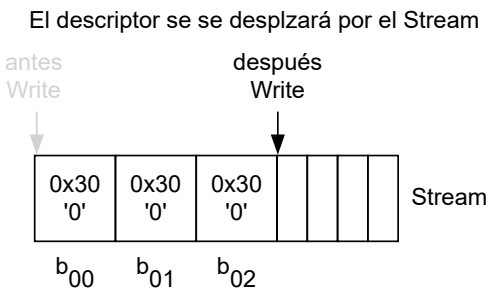
La forma escribir más común, herencia del lenguaje C es el método...

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[] { 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.
  > ✋ **Importante**: 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",
        FileMode.Open,      // Abro un fichero existente (sino error)
        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()

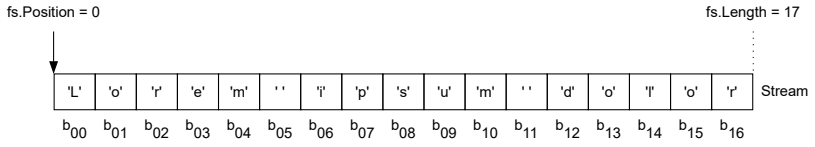
Lee un byte del archivo y avanza la posición de lectura un byte. Devuelve, el byte, convertido en un int, o **-1 si se ha alcanzado el final de la secuencia**.

Longitud y Posición en el Stream

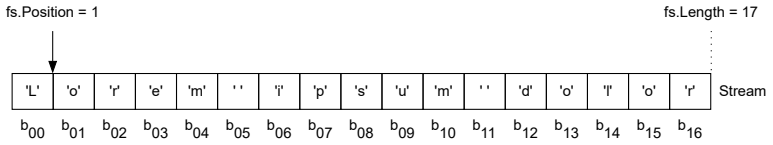
- **long Length**: Número de bytes almacenados en el flujo (tamaño del flujo).
  En ficheros, el tamaño en bytes el fichero.
- **long Position**: Número del **byte actual** en el flujo **empezando en 0**.
  En ficheros, la posición actual **donde se encuentra el descriptor** de lectura o escritura del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

Ejemplo básico:

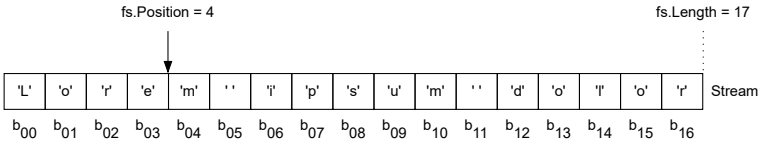
```
FileStream fs = File.Create("ejemplo.txt");
// Paso la cadena a un array de 17 bytes para poder
// escribirla con un FileStream.
byte[] buffer = Encoding.UTF8.GetBytes("Lorem ipsum dolor");
fs.Write(buffer, 0, buffer.Length);
fs.Close();

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();
```



> 💡 **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 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`

  > ✋ **Importante:** 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.

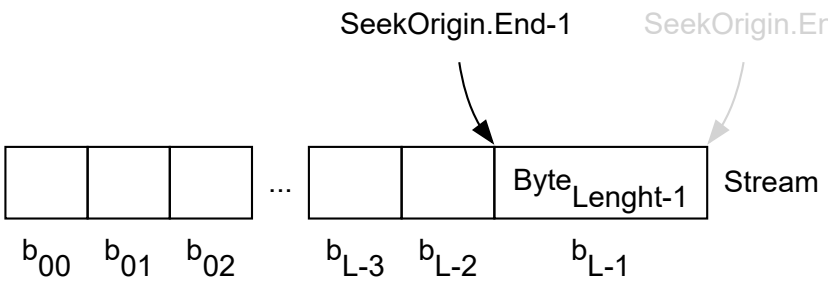
Ejemplo básico de uso:

```
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);`

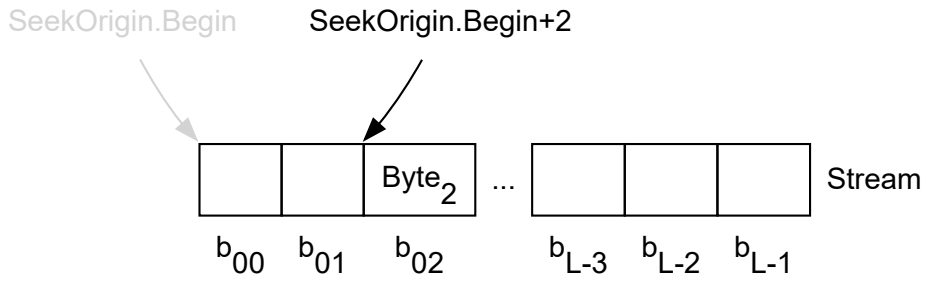
Me sitúo al **final** y me desplazo **1 byte a la izquierda**, de tal manera que me quedo para escribir o leer sobre el último byte el **byte n**.



Caso de desplazamiento 2:

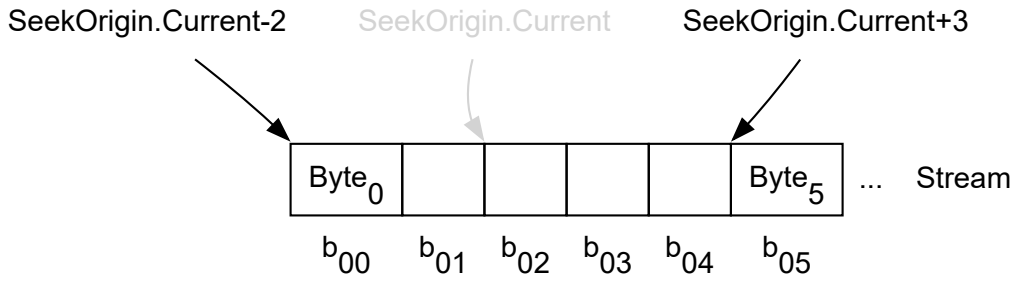
```
fichero.Seek(2, SeekOrigin.Begin);
```

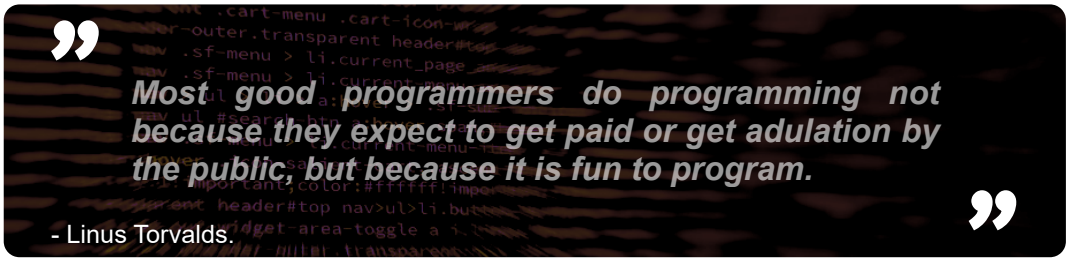
Me sitúo al **principio** y me desplazo **2 bytes** a la derecha, de tal manera que me quedo para escribir o leer sobre el **byte 2**.



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.

> 💀 **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.

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

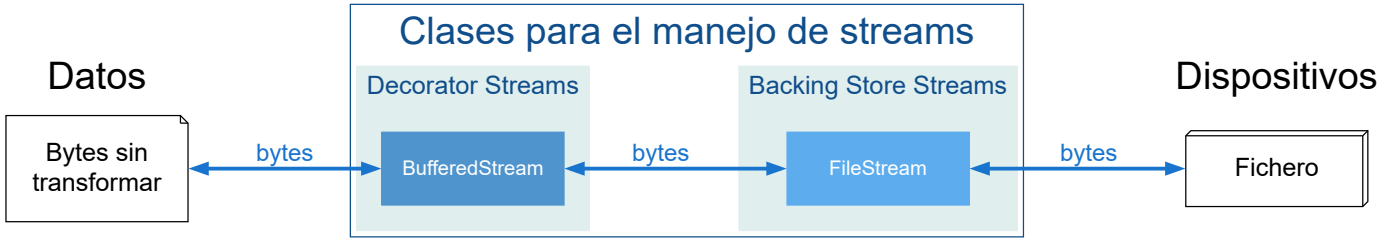
Pasando el flujo por un '*Decorator Stream*'

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 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*').

> **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 extremádamente 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");
}
```

🎓 **Caso 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*'

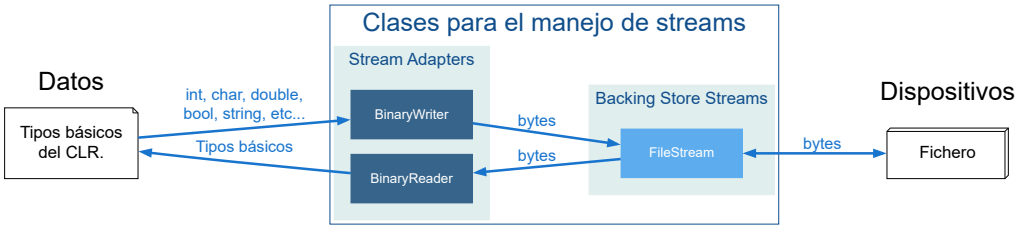
Un **adaptador** en POO es un patrón de diseño que básicamente se utiliza para ampliar y/o transformar el interfaz de las operaciones sobre una determinada clase.

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



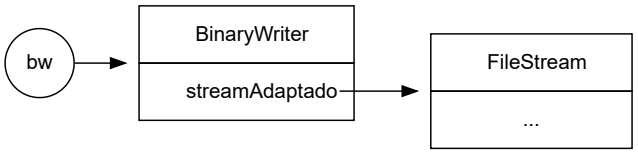
- Sus método de escritura `Write` y lectura `Read<Tipo>` '*adaptarán*' los Write y Read de FileStream permitiendo escribir y leer respectivamente los tipos básicos definidos en el lenguaje, `int`, `short`, `string` etc.
- Estos tipos básicos se transformarán en una secuencia de bytes **tal y como los 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.

```
// Creamos el FileStream
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 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 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

Propiedad	Formato que representa el objeto devuelto
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`
Adaptador `BinaryReader`

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

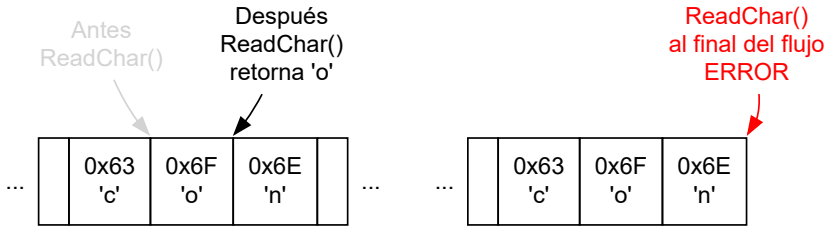
Realizará el proceso inverso al **BinaryWriter** y tendrémos un método de lectura específico para la adaptar la lectura de cada tipo básico.

```
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();
```

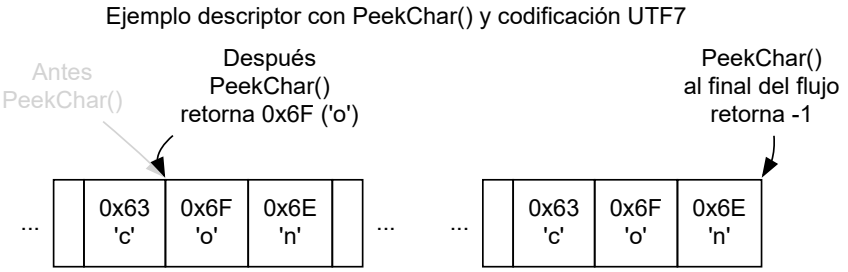
Si nos fijamos el los **métodos de lectura definidos por la clase**, todos leerán los tipos definidos en el lenguaje y al leer cualquier tipo, el descriptor del flujo avanzará en el mismo. Sin embargo habrá un caso especial de dos métodos que se complementan y comentaremos a continuación.

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

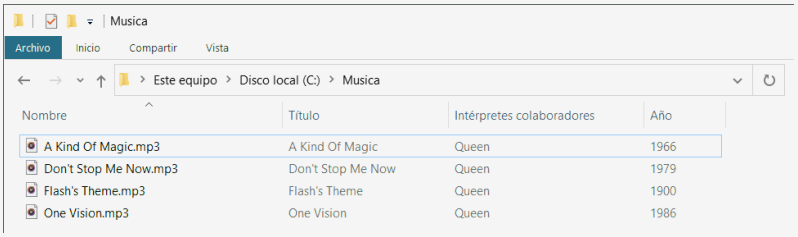
> 📌 **Nota:** Si fuera un **fichero en binario** esta lectura no tendría mucho sentido pues obtendríamos caracteres no representables por consola.

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

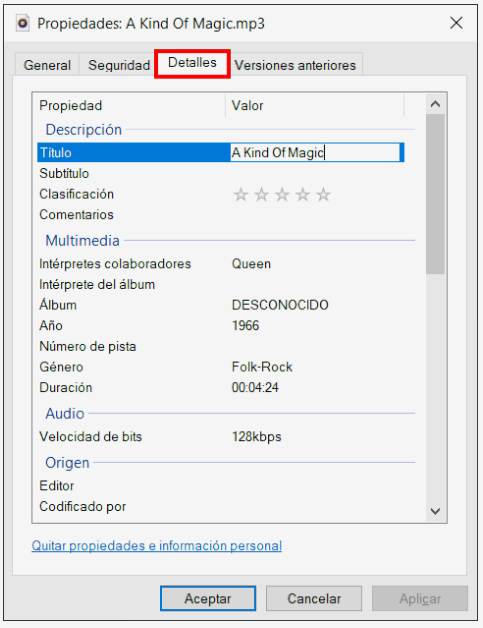
🎓 **Caso de estudio:**

Lectura en binario con BinaryReader:

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 10 (**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.

> 📌 **Nota:** 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
```

Pasos para resolver el caso de estudio

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.

```
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 = new string[]
    {
        "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"
    };
}
```

2. A continuacíon vamos a añadir a la clase `TagId3v1` los campos privados **titulo**, **artista**, **álbum** y **comentario** de tipo `string` , **año** de tipo `ushort` y **género** de tipo `byte` . (Se accederán siempre a través de sus getters).
   Además, vamos a definir los accesores públicos para título, artista, álbum, comentario y año. Sin embargo, para género definiremos **dos** accesores públicos uno que devuelva el `byte` y otro el `string` con el texto del género.
   Una posible implementación sería la siguiente ...

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

    private string titulo;
    private string artista;
    private string album;
    private ushort año;
    private byte genero;
    private string comentario;

    public string GetTitulo()
    {
        return titulo;
    }

    public string GetArtista()
    {
        return artista;
    }
    public ushort GetAño()
    {
        return año;
    }

    public string GetComentario()
    {
        return comentario;
    }
    public string GetAlbum()
    {
        return album;
    }
    ...
```

```
        ...
        public string GetGenero()
        {
            if (genero >= 80)
                return "OTHER";
            else
                return generosTagID3v1[genero];
        }

        public byte GetByteGenero()
        {
            return genero;
        }
    }
```

3. **Constructor privado** que reciba todos los campos (excepto, claro está, el array de géneros). Además, vamos a invalidar el método `ToString` para que muestre los datos del objeto.

```
class TagId3V1
{
   // ... código omitido por abreviar
   private TagId3V1(
       string titulo, string artista, string album,
       in ushort año, in byte genero, string comentario)
   {
       this.titulo = titulo ?? throw new ArgumentNullException(nameof(titulo));
       this.artista = artista ?? throw new ArgumentNullException(nameof(artista));
       this.album = album ?? throw new ArgumentNullException(nameof(album));
       this.año = año;
       this.genero = genero;
       this.comentario = comentario ?? throw new ArgumentNullException(nameof(comentario));
   }


    public override string ToString()
    {
        return $"{"Titulo:",11} {GetTitulo()}\n"
               + $"{"Artista:",11} {GetArtista()}\n"
               + $"{"Album:",11} {GetAlbum()}\n"
               + $"{"Año:",11} {GetAño()}\n"
               + $"{"Genero:",11} {GetGenero()}\n"
               + $"{"Comentario:",11} {GetComentario()}";
    }
}
```

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...

> ✋ **Importante:** Con el fín de mostrate un código de mayor calidad, se han controlado las excepciones de apertura usado la clausula `using` . Esta se explicará más adelante en el punto **Manejo de excepciones con ficheros**. Así pues, si aunque ahora no la entiendas, te recomiendo que vuelvas a revisar este código cuando la veas.

```
class TagId3V1
{
    // ... código omitido por abreviar
    public static bool HayTAG(string rutaFichero)
    {
        bool hayTAG = false;
        using (FileStream stream = new FileStream(
            rutaFichero,
            FileMode.Open,
            FileAccess.Read,
            FileShare.Read))

        using (BinaryReader streamRB = new BinaryReader(stream))
        {
            // 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 string(streamRB.ReadChars(BYTES_TAG_ID));
                hayTAG = tag == "TAG";
            }
        }
        return hayTAG;
    }
}
```

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.

   > 📌 **Nota:** A este tipo de métodos que instáncian un objeto de la clase a la que pertenecen sin ser un constructor, se les denomina `factoría` .

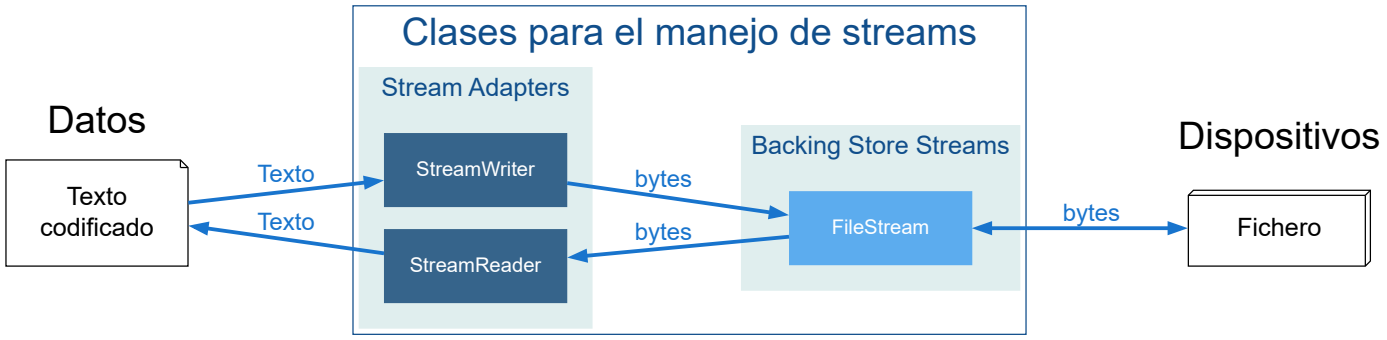
```
// Definiremos la clase con la excepción personalizada fuera de la clase TagId3v1
public class TagId3v1Exception : Exception
{
    public TagId3v1Exception(string message) : base(message) { ; }
    public TagId3v1Exception(string message, Exception innerException)
    : base(message, innerException)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

```
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 FileStream(
                rutaFichero, FileMode.Open,
                FileAccess.Read))
        using (BinaryReader streamBR = new BinaryReader(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);
            }
        }
    }
}
```

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);
            }
        }
    }
}
```

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.
- `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**).
  // Añadirá los bytes de BOM con la codificación al principio del Stream.
  // Ya no podremos usar fs porque no nos pertenecerá ya.
- 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`.

  > ✋ **Importante**: 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.

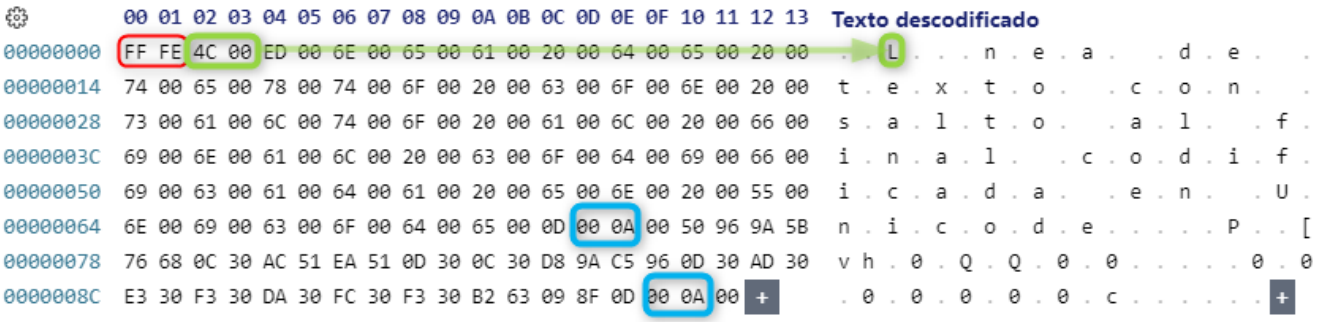
Adaptador `StreamWriter`

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

// 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 al principio de la `línea 8` están 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).

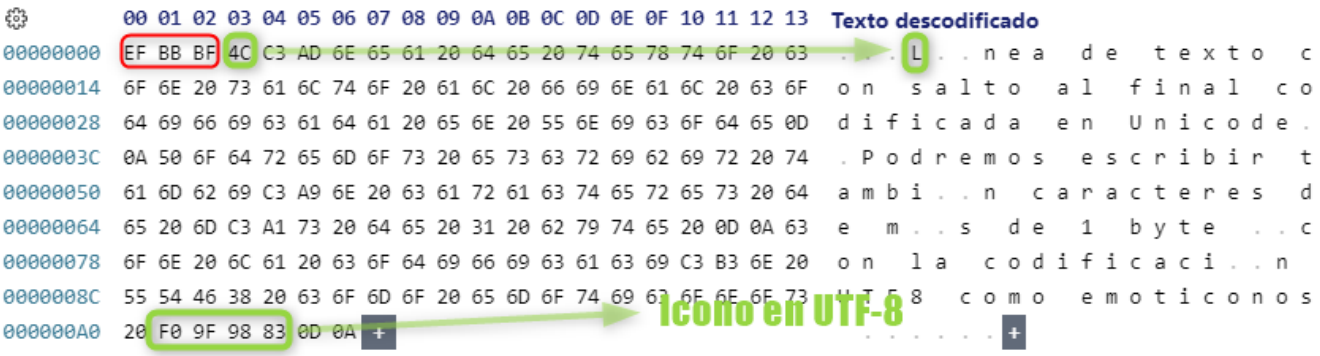
> 📌 **Nota:** Dependiendo del SO este **salto de línea** o `\n` se codificará de diferente manera.

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

   > 📌 **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).

   > ⚠️ **Aviso:**
   >
   > - 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 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 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);
    }
}
```

🎓 **Caso de estudio:**

Usando StreamWriter, StreamReader y excepciones:

Vamos a escribir un programa que gestione datos de Alumnos y los almacene en un **fichero CSV (Comma-Separated Values)**.

1. Para ello vamos a definir una clase `Alumno` de la siguiente forma:
   - Guardaremos los siguientes campos: `nia`, `nombre`, `apellido`.
   - La clase tendrá un constructor que recibe todos los campos.
   - **Accesores** o *getters* para cada uno de los campos.
   - Invalidación de `ToString()` para que muestre los datos del alumno **en una línea**.
2. Definiremos un menú con las siguientes opciones:
   - **Introduce Alumno:** El cual pedirá los datos de un alumno y lo almacenará en un fichero de texto con codificación **UTF8** y separado por comas CSV denominado `alumnos.csv` en la misma carpeta del ejecutable. Si el archivo existe, se permitirá añadir al final más información sin eliminar el contenido.
   - **Mostrar Alumnos:** La cual mostrará en pantalla el contenido del fichero **alumnos.csv**.
   - **Buscar Alumno:** Buscará en el fichero mostrará la información de un determinado alumno a partir de su **NIA** introducido.
3. Para realizar la gestión del punto 2 vamos a definir una concreción de la clase `Alumno` denominada `AlumnoCSV`. Dicha clase tendrá como '*responsabilidad*' las operaciones '*básicas*' sobre el fichero CSV que **NO impliquen una lectura y escritura por consola**:
   - Método de **instancia** `void Guarda(string ruta)` que añada los datos del alumno al final del fichero CSV indicado en la ruta.
     La '*coma*' de separación de los campos será el carácter **';'** (punto y coma).
     Al añadir un alumno, deberemos tener en cuenta que **la primera línea del fichero csv guarda el nombre de los campos**. Por lo que si estamos añadiendo el primer alumno, antes deberemos añadir la fila con el nombre de los campos.
     > 📌 **Nota:** Podrá haber NIA repetidos, puesto que no controlaremos tal hecho.

     Ejemplo: Tras guardar 2 alumnos el aspecto del fichero podría ser el siguiente ...

     ```
     NIA;Apellido;Nombre
     1;Garcia;Xusa
     2;Guarinos;Juanjo
     ```

   - Método de **clase** `Alumno[] Lee(string ruta)` que leerá los alumnos guardados en el fichero especificado en la ruta o null si está vacío.
   - Método de **clase** `public static bool Busca(in ulong nia, string ruta, out Alumno alumno)` que me dirá si existe un alumno un alumno o no en el fichero especificado y en caso afirmativo me devolverá un objeto con la información del mismo.
4. Por último, añadiremos la gestión de **excepciones** para controlar posibles problemas de acceso a los ficheros.

Propuesta de solución explicada (**lee los comentarios del código**):

1. Definiremos la clase `Alumno` donde su **única responsabilidad** será gestionar los campos, sus accesores y como se pasa a cadena sus datos.

```
public class Alumno
{
    readonly private ulong nia;
    readonly private string nombre;
    readonly private string apellido;

    public ulong GetNIA()
    {
        return nia;
    }
    public string GetNombre()
    {
        return nombre;
    }
    public string GetApellido()
    {
        return apellido;
    }
    public Alumno(in ulong nia, string nombre, string apellido)
    {
        this.nia = nia;
        this.nombre = nombre;
        this.apellido = apellido;
    }
    public override string ToString()
    {
        return $"{GetNIA(),-8}{GetApellido(),-12}{GetNombre(),-12}";
    }
}
```

2. Ahora definimos la concreción de `Alumno` con para gestionar la escritura y lectura de sus datos sobre el fichero CSV.

```
public class AlumnoCSV : Alumno
{
    // Una de las posibilidades de gestionar excepciones particulares de una clase
    // es definir un tipo anidado para ela excepción. De tal manera que cuando
    // la usemos fuera de la misma el tipo será AlumnoCSV.Excepción lo cual
    // me indicará claramente que serán excepciones derivadas de la escritura
    // y lectura del CSV por parte de AlumnoCSV
    public class Excepción : Exception
    {
        public Excepción(string message, Exception inner)
        : base(message, inner) { }
    }
    public AlumnoCSV(in ulong nia, string nombre, string apellido)
    : base(nia, nombre, apellido)
    {
        // Constructor
    }
```

```
        // Método de clase privado que se encargará de leer la próxima línea del
        // CSV a partir de donde se encuentre situado el descriptor de
        // StreamReader sr (debería estar al principio de la misma).
        // EVITA QUE SE REPITAN ESTAS LÍNEAS EN EL LEE Y EN EL BUSCA.
        // Al ser privada cualquier excepción producida por la lectura o no correspondencia
        // de de lo leído en la línea con lo que se espera para poder crear un objeto Alumno
        // será gestionada por los métodos que la usen.
        // Por ejemplo: Una excepción posible podría ser un OutOfBoundsException
        // resultado que que no hay tres elementos en el array tras el Split.
        private static Alumno Lee(StreamReader sr)
        {
            string[] campos = sr.ReadLine()?.Split(new char[] { ';' })
                            ??
                            throw new NullReferenceException("No se ha podido leer Alumno en el CSV");
            return new Alumno(ulong.Parse(campos[0]), campos[1], campos[2]);
        }

        // Recibe una ruta a un fichero CSV de alumnos y me devuelve un array con los
        // alumnos guardados en el mismo o null si no hay ninguno.
        // EL método no puede mostrar nada porque generaríamos una dependencia con
        // la consola que excedería la responsabilidad de la clase y la haría menos
        // reutilizable. Por la misma razón, tampoco devolvemos un string con todos
        // los alumnos.
        // Quien use el método decidirá como se visualizan.
        public static Alumno[] Lee(string ruta)
        {
            // Inicializamos el array a null.
            Alumno[] alumnos = new Alumno[0];
            try
            {
                // Al usar el using, nos aseguramos que al salir del ámbito de
                // declaración se ha realizado un finally para cerrar los streams
                using FileStream stream = new(ruta, FileMode.Open, FileAccess.Read);
                using StreamReader sr = new(stream, Encoding.UTF8);

                 // Leemos y saltamos cabecera de las columnas
                sr.ReadLine();
                while (!sr.EndOfStream)
                {
                    Array.Resize(ref alumnos, alumnos?.Length + 1 ?? 1);
                    // Añadimos el array leído al hueco que hemos hecho para el mismo.
                    alumnos[^1] = Lee(sr);
                }
            }
            catch (Exception e)
            {
                // Relanzo cualquier excepción envolviéndola en un tipo
                // AlumnoCSV.Excepción e indicando el proceso que estaba haciendo
                // para poder saber mejor de donde ha venido la misma.
                // En este momento se que los streams están cerrados porque ya se
                // ha ejecutado el finally implícito por los using.
                throw new Excepción($"Leyendo alumnos de {ruta}", e);
            }
            return alumnos;
        }
```

```
// Método de instancia encargado de guardar el Alumno que en el fichero CSV
// especificado en la ruta.
public void Guarda(string ruta)
{
    try
    {
        // Abrimos con Append que añade al final del fichero o lo crea si no existe.
        using FileStream stream = new(ruta, FileMode.Append, FileAccess.Write);
        using StreamWriter sw = new(stream, Encoding.UTF8);
        // Si el stream se acaba de crear añadiremos la fila con los nombre de
        // las columnas antes de escribir el primer registro.
        if (stream.Length == 0)
            sw.WriteLine($"NIA;Apellido;Nombre");

        // Escribimos la línea y hacemos un flush para asegurarnos de que
        // se modifique el fichero.
        sw.WriteLine($"{GetNIA()};{GetApellido()};{GetNombre()}");
        sw.Flush();
    }
    catch (Exception e)
    {
        // Relanzo cualquier excepción envolviéndola en un tipo
        // AlumnoCSV.Excepción e indicando el proceso que estaba haciendo
        // para poder saber mejor de donde ha venido la misma.
        // En este momento se que los streams están cerrados porque ya se
        // ha ejecutado el finally implícito por los using.
        throw new Excepción($"Guardando Alumno {this} en {ruta}", e);
    }
}

// Método de clase que recibe un NIA y una ruta a un fichero CSV y busca
// el primer alumno con el NIA especificado.
// Si no lo encuentra devuelve falso y si lo encuentra devuelve cierto y
// una instáncia a un objeto Alumno con los datos del mismo.
public static bool Busca(in ulong nia, string ruta, out Alumno? alumno)
{
    alumno = default;
    try
    {
        using FileStream stream = new(ruta, FileMode.Open, FileAccess.Read);
        using StreamReader sr = new(stream, Encoding.UTF8);
        // Leemos y saltamos cabecera de las columnas
        sr.ReadLine();
        while (!sr.EndOfStream && alumno == null)
        {
            // Llamo al método de privado de clase Lee al que le paso
            // el stream apuntado a la lectura de la siguiente línea
            // con los datos de una alumno.
            Alumno a = Lee(sr);
            // Si encuentro el nía establezco el parámetro de salida
            // encontrado y finalizo el bucle.
            if (nia == a.GetNIA())
                alumno = a;
        }
    }
```

```
            catch (Exception e)
            {
                // Relanzo cualquier excepción envolviéndola en un tipo
                // AlumnoCSV.Excepción e indicando el proceso que estaba haciendo
                // para poder saber mejor de donde ha venido la misma.
                // En este momento se que los streams están cerrados porque ya se
                // ha ejecutado el finally implícito por los using.
                throw new Excepción($"Buscando alumno con NIA {nia}", e);
            }

            return alumno != null;
        }
    }
```

3. Vamos a implementar el programa principal basándonos en las clases definidas anteriormente.

```
    class Program
    {
        // Enumeración con las opciones del programa para autodocumentar y evitar números mágicos.
        private enum OpciónPrograma {IntroduceAlumno = 1, MostrarAlumnos = 2, BuscarAlumno = 3, Salir = 4 }

        // Modularización de mostrar menú para autodocumentar y simplificar el Main.
        static void Menu()
        {
            Console.Clear();
            Console.WriteLine($"{(int)OpciónPrograma.IntroduceAlumno} - Introduce Alumno.");
            Console.WriteLine($"{(int)OpciónPrograma.MostrarAlumnos} - Mostrar Alumnos.");
            Console.WriteLine($"{(int)OpciónPrograma.BuscarAlumno} - Buscar Alumno.");
            Console.WriteLine($"{(int)OpciónPrograma.Salir} - Salir.");
        }
        // Lee una opción válida de programa.
        static OpciónPrograma Lee()
        {
            bool válida;
            OpciónPrograma o = OpciónPrograma.Salir;
            do
            {
                Console.Write("Introduce una opción: ");
                válida = int.TryParse(Console.ReadLine(), out int valor);
                if (válida)
                {
                    int[] valores = (int[])Enum.GetValues(typeof(OpciónPrograma));
                    válida = Array.IndexOf(valores, valor) >= 0;
                }
                if (válida)
                    o = (OpciónPrograma)valor;
                else
                    Console.WriteLine("Opción incorrecta!!");
            }
            while (!válida);
            return o;
        }
```

```
// Modularización del proceso de introducción de un alumno para
// autodocumentar y simplificar el Main.
// Cualquier excepción producida se gestionará en el Main
static void IntroduceAlumno(string fichero)
{
    Console.Write("NIA: ");
    ulong nia = ulong.Parse(Console.ReadLine() ?? "0");
    Console.Write("Nombre: ");
    string nombre = Console.ReadLine() ?? "";
    Console.Write("Apellido: ");
    string apellido = Console.ReadLine() ?? "";

    // Creo una instáncia de AlumnoCSV y guardo sus datos en el
    // fichero que me indican por parámetro.
    new AlumnoCSV(nia, nombre, apellido).Guarda(fichero);
    Console.WriteLine("Datos guardados.");
}

// Modularización del proceso de buscar y mostrar los datos de un alumno
// para autodocumentar y simplificar el Main.
// Cualquier excepción producida se gestionará en el Main
static void BuscaAlumno(string fichero)
{
    // Hago un control preliminar para evitar excepciones obvias que se
    // pueden controlar fácilmente con la lógica del programa.
    if (File.Exists(fichero))
    {
        Console.Write("NIA a buscar: ");
        ulong nia = ulong.Parse(Console.ReadLine() ?? "0");

        // Una ves leído el NIA uso el método de clase definido en AlumnoCSV
        if (AlumnoCSV.Busca(nia, fichero, out Alumno? a))
            Console.WriteLine($"Los datos del alumno con nia {nia} son:\n{a}");
        else
            Console.WriteLine($"No existe ningún alumno de nia {nia}.");
    }
    else
        Console.WriteLine($"El fichero de datos {fichero} aún no se ha creado.");
}
```

Continuamos con la implementación del Main que además de gestionar y recuperarse de las posibles excepciones. Controlará la lógica de ejecución de las opciones posibles.
Aunque ha quedado bastante legible, su número de líneas podría sugerirnos una posible modularización más para simplificar que incluya el Switch....

```
        static void Main()
        {
            const string FICHERO = "alumnos.csv";
            OpciónPrograma opción = OpciónPrograma.Salir;

            do
            {
                bool pausaTrasOpción = true;
                try
                {
                    Menu();
                    opción = Lee();
                    switch (opción)
                    {
                        case OpciónPrograma.IntroduceAlumno:
                            IntroduceAlumno(FICHERO);
                            break;
                        case OpciónPrograma.MostrarAlumnos:
                            if (File.Exists(FICHERO))
                            {
                                var alumnos = AlumnoCSV.Lee(FICHERO);
                                Console.WriteLine(alumnos != null
                                ? string.Join<Alumno>("\n", alumnos)
                                : $"El fichero {FICHERO} está vacío.");
                            }
                            else
                                Console.WriteLine($"El fichero de datos {FICHERO} aún no se ha creado.");
                            break;
                        case OpciónPrograma.BuscarAlumno:
                            BuscaAlumno(FICHERO);
                            break;
                        case OpciónPrograma.Salir:
                            pausaTrasOpción = false;
                            break;
                    }
                }
                catch (Exception? e)
                {
                    while (e != null)
                    {
                        Console.WriteLine(e?.Message);
                        e = e?.InnerException;
                    }
                }
                if (pausaTrasOpción)
                {
                    Console.Write("Pulsa una tecla...");
                    Console.ReadKey();
                }
            } while (opción != OpciónPrograma.Salir);
        }
    } // Cierre de la definición de la clase Program
```