

Tema 8

[Descargar estos apuntes](#)

Índice

1. Definiciones básicas
2. Gestión de rutas en .NET
 1. Clase Path
 1. Operaciones con rutas
3. Gestión de archivos y directorios
 1. Clases de utilidad File y Directory
 1. Ejemplo de uso de File y Directory
 2. Clases FileInfo y DirectoryInfo
 1. Ejemplos de uso de FileInfo y DirectoryInfo
4. Flujos de datos en serie o streams
 1. Conceptos generales sobre Streams
 1. Operaciones y nomenclatura relacionada con Streams
 1. Apertura (Open)
 2. Cierre (Close)
 3. Lectura (Read) y Escritura (Write)
 4. Volcado (Flush)
 2. Streams en C#
 1. Lectura y escritura de ficheros sin transformar
 1. Apertura y Cierre de un fichero con FileStream
 2. Escritura de un fichero con FileStream
 3. Lectura de un fichero con FileStream
 4. Longitud y Posición en el Stream
 5. Desplazándonos por el Stream
 6. Recorriendo un FileStream hasta el final
 2. Pasando el flujo por un '*Decorator Stream*'
 1. Ejemplo de '*Decorator Stream*' usando `BufferedStream`
 3. Transformando el flujo con un '*Stream Adapter*'

Definiciones básicas

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

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

Gestión de rutas en .NET

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

Clase Path

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

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

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

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

escribimos...

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

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

Operaciones con rutas

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

En su lugar hay tres alternativas:

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

Esta opción es la más simple, si sabemos cual es el SO donde vamos a ejecutar.

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

string Path.GetDirectoryName(string path)

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

string Path.GetFileName(string path)

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

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

string Combine(string path1, string path2);

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

Gestión de archivos y directorios

Clases de utilidad File y Directory

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

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

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

Ejemplo de uso de File y Directory

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

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

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

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

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

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

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

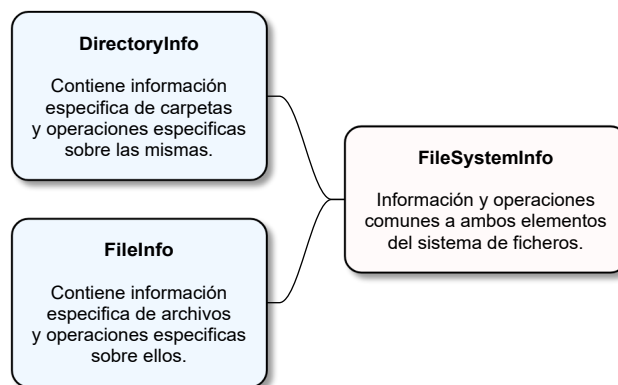
Clases FileInfo y DirectoryInfo

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

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

DirectoryInfo y **FileInfo** pues ...

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



- Una ejemplo de esta información común detallada. Es el campo **`Attributes`** del sistema de ficheros. El campo **`Attributes`** es una máscara creada a partir del **Enum `FileAttributes`**. Con información común a ficheros y directorios.

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

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

Ejemplos de uso de FileInfo y DirectoryInfo

Ejemplo 1:

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

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

        StringBuilder informacion = new StringBuilder();

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

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

Ejemplo 2:

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

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

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

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

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


Caso de estudio:

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

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

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

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

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

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

continuación ...

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

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

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

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

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

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

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

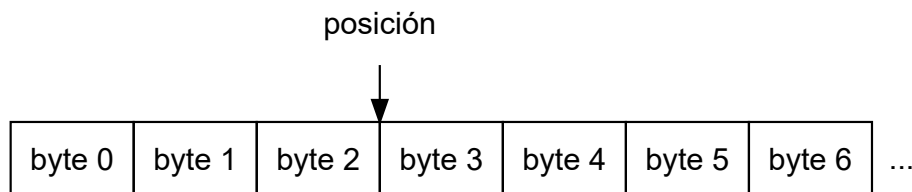
Flujos de datos en serie o streams

Conceptos generales sobre Streams

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

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

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




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

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

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

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

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


Operaciones y nomenclatura relacionada con Streams

Apertura (Open)

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

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

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

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

Cierre (Close)

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

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

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

Lectura (Read) y Escritura (Write)


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

Volcado (Flush)

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

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

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

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

Streams en C#

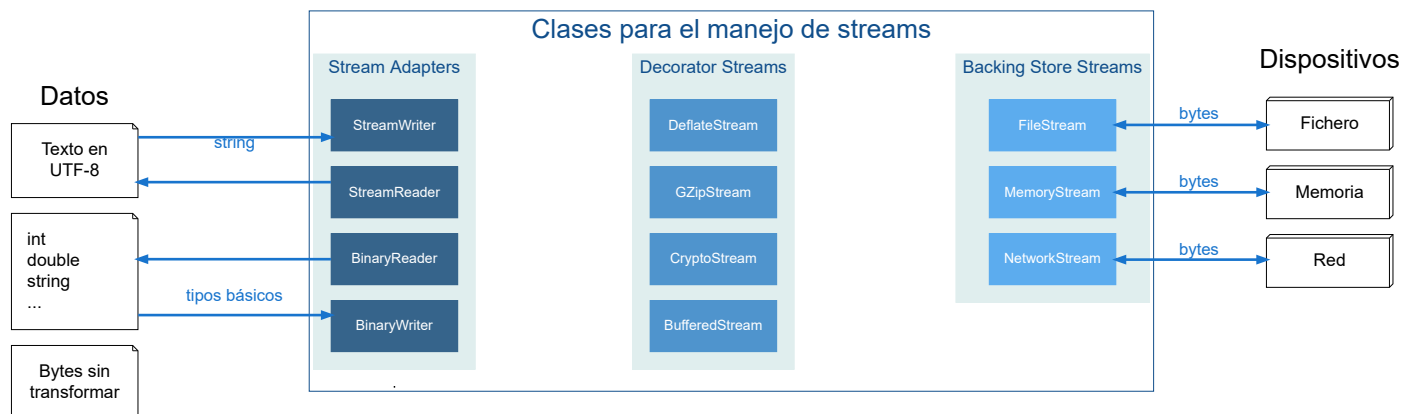
En C# los archivos, directorios y flujos con ficheros, se manejan con clases de las BCL definidas en el namespace:

`System.IO`

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

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

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



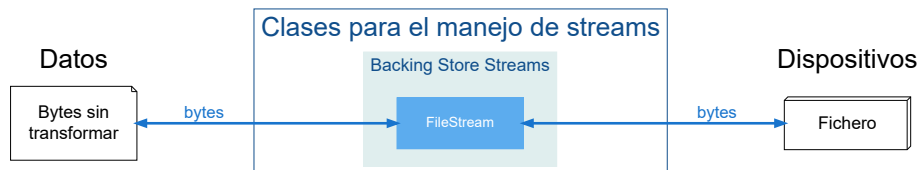
Si nos fijamos, tendremos de derecha a izquierda ...

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

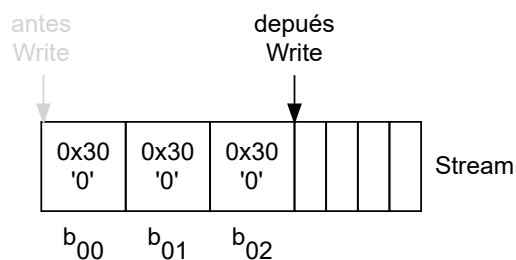
Ejemplo básico de uso:

```
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();
```

El descriptor se se desplazará por el Stream




public override void FileStream.WriteByte(byte byte)

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

Lectura de un fichero con FileStream

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

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

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

Ejemplo básico de uso:

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

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

// Leo desde el principio datos.Length = 3 bytes y los añado del array datos
11 int bytesLeídos = 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 < bytesLeídos; i++)
    Console.WriteLine($"{datos[i]:X} ");
fichero.Close();
```

public override int FileStream.ReadByte()

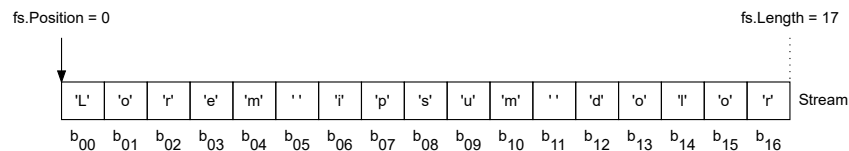
Lee un byte del archivo y avanza la posición de lectura un byte. Devuelve, el byte, convertido en un int, o **-1 si se ha alcanzado el final de la secuencia**.

Longitud y Posición en el Stream

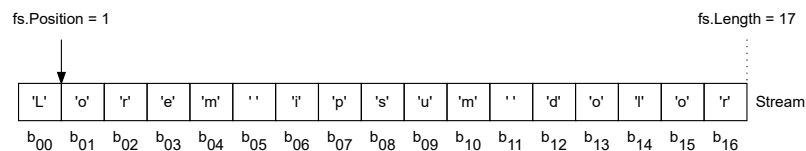
- **long Length:** Número de bytes almacenados en el flujo (tamaño del flujo).
En ficheros, el tamaño en bytes el fichero.
- **long Position:** Número del **byte actual** en el flujo **empezando en 0**.
En ficheros, la posición actual **donde se encuentra el descriptor** de lectura o escritura del fichero.

Ejemplo básico:

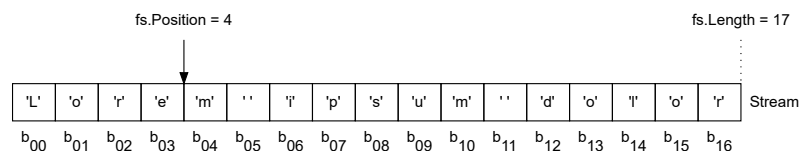
```
FileStream fs = File.Create("ejemplo.txt");  
// Paso la cadena a un array de 17 bytes para poder  
// escribirla con un FileStream.  
4 byte[] buffer = Encoding.UTF8.GetBytes("Lorem ipsum dolor");  
fs.Write(buffer, 0, buffer.Length);  
fs.Close();  
  
fs = new FileStream("ejemplo.txt", FileMode.Open,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



```
char c = (char)fs.ReadByte(); // c = 'L'  
2 Console.WriteLine("Posicion 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("Posicion descriptor lectura: " + fs.Position); // Devolverá un 4  
fs.Close();
```



💡 **Tip:** estos valores nos pueden ser útiles para controlar si hemos llegado o no al final de las secuencia en **algunos casos**.


Por ejemplo con la condición `while (fichero.Position < fichero.Length)` puedo saber si estoy al final de un stream.

Desplazándonos por el Stream

public override long Seek(long offset, SeekOrigin origin)

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

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

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

Ejemplo básico de uso:

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

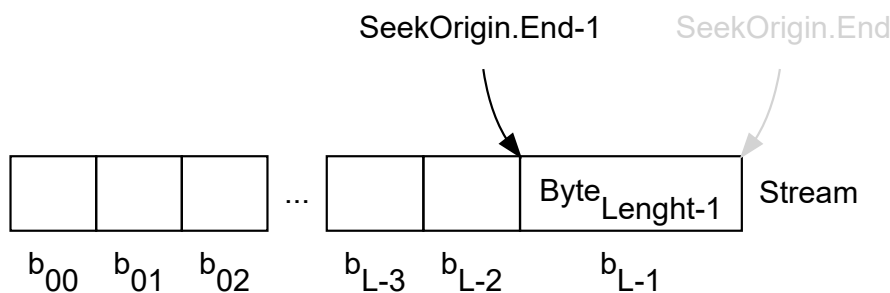
if (fichero.CanSeek) // Para streams que no admiten desplazamientos
{
    // Me sitúo al final y me desplazo 1 a la izquierda.
    fichero.Seek(-1, SeekOrigin.End);
    // Leo el último byte.
    int ultimoByte = fichero.ReadByte();
    Console.WriteLine($"El valor del último byte es {ultimoByte:X}");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

fichero.Close();
```

Caso de desplazamiento 1:

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

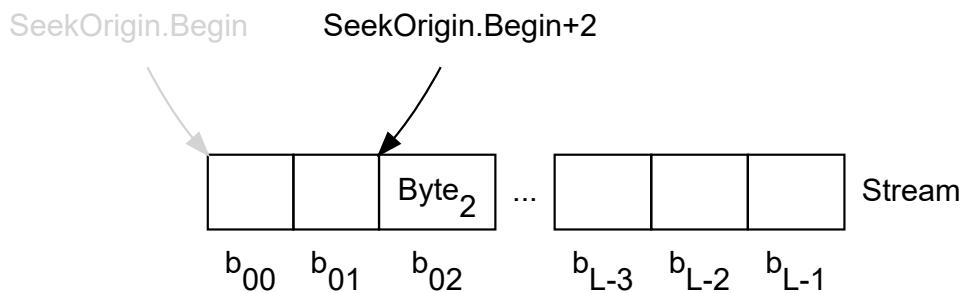
Me sitúo al **final** y me desplazo **1 byte a la izquierda**, de tal manera que me quedo para escribir o leer sobre el último byte el **byte n**.



Caso de desplazamiento 2:

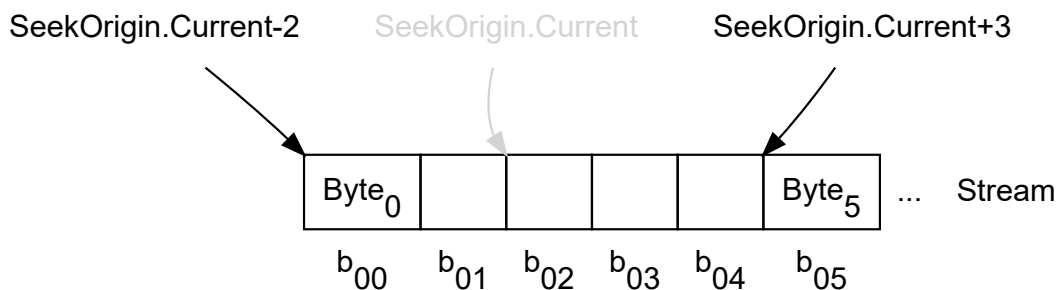
```
fichero.Seek(2, SeekOrigin.Begin);
```

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



Caso de desplazamiento 3:

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



Recorriendo un FileStream hasta el final

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


• Caso 1:

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

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

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

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

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

• Caso 2:

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


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

• Caso 3:

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

fs.Close();
}
```

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

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

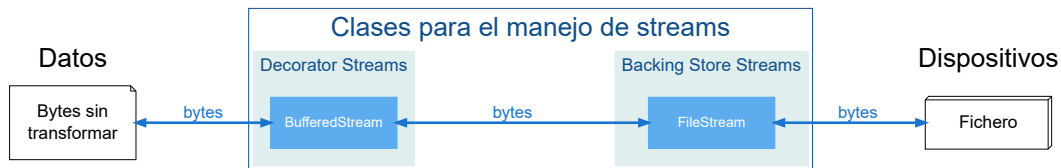
Pasando el flujo por un 'Decorator Stream'

Aunque este tipo de streams no los vamos a ver en profundidad en este curso por falta de tiempo. Básicamente, un **decorator** en POO es un patrón de diseño que añade funcionalidad a un objetos sin necesidad de utilizar el mecanismo de 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.pru",
        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.pru",
        FileMode.Create,
        FileAccess.Write);

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

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

34

Casos de estudio:

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

Transformando el flujo con un '*Stream Adapter*'

Ver apuntes de XUSA

// [https://es.wikipedia.org/wiki/Adaptador_\(patrón_de_diseño\)](https://es.wikipedia.org/wiki/Adaptador_(patrón_de_diseño))

Razón de

- Trabajar con arrays de bytes es poco útil además de engorroso.
- Se añade pues una capa de abstracción para manejar de forma más cómoda los tipos datos que usamos en nuestros programas.
- Debo crearlos a partir de otros streams de almacenamiento abiertos como FileStream.

Stream Adapters BinaryWriter y BinaryReader

- Escribe o lee tipos valor primitivos en binario en una secuencia y admite escribir o leer cadenas en una codificación específica.
- Sus método Write y Read permite escribir y leer los tipos primitivos, int, short, string etc. 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

Ejemplo Adaptador BinaryWriter

```
FileStream fs = new FileStream("ejemplo.txt", FileMode.Create, FileAccess.Write);
BinaryWriter bw = new BinaryWriter(stream, Encoding.UTF8);
int valor = 10;
bw.Write(valor);
bw.Close();
```

Ejemplo Adaptador BinaryReader

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

byte[] bytesInt = new byte[4];
stream.Read(bytesInt, 0, bytesInt.Length);
Console.WriteLine("Valor guardado en binario con fotmato(Little-Indian):");
foreach (byte _byte in bytesInt) Console.WriteLine($"{_byte:X2}");
stream.Seek(0, SeekOrigin.Begin);
BinaryReader adaptadorDeStream
    = new BinaryReader(stream, Encoding.UTF8);

ulong valor = adaptadorDeStream.ReadInt32();
Console.WriteLine("\nEl valor guardado como ulong es: " + valor);
adaptadorDeStream.Close();
```

Stream Adapters StreamWriter y StreamReader

- Están diseñado para escribir y leer caracteres como salida en una codificación determinada.
- Por tanto están pensados para trabajar solo con archivos de texto y no archivos con datos en binario.

- StreamWriter utiliza de forma predeterminada una instancia de UTF8 Encoding, a menos que se especifique lo contrario.
- Además, se añadirá a la cabecera del archivo de texto 2 bytes con la codificación de caracteres utilizada. Estos bytes se denominarán BOM (Byte Order Mark).
- Para StreamReader la forma de controlar que se ha llegado al final de fichero. Es cuando una lectura devuelve null en lugar de un string o a través de la propiedad EndOfStream.
- No podemos en ningún caso basarnos en el atributo Position de su FileStream base pues su valor no lo controlaremos. Ni podremos hacer Seek.

Ejemplo Adaptador StreamWriter

```
FileStream stream = new FileStream(
    "ejemplo.txt", FileMode.Create, FileAccess.Write);
string text = "限定桶「冬風」「高雅」キャンペーン掲載";
StreamWriter adaptadorDeStream =
    new StreamWriter(stream, Encoding.Unicode);

adaptadorDeStream.WriteLine(text);
adaptadorDeStream.Close();
```

Ejemplo Adaptador StreamReader

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

// Si está a true, tomará la codificación a partir de los bytes del BOM,
// si están definidos en el archivo.
StreamReader adaptadorDeStream =
    new StreamReader(stream, Encoding.Unicode, true);

StringBuilder textBuilder =
    new StringBuilder(adaptadorDeStream.ReadToEnd());

textBuilder.Replace("\r\n", "<br>");
string text = textBuilder.ToString();
adaptadorDeStream.Close();
```