UNIDAD 8. FLUJOS ENTRADA/SALIDA DE DATOS. PERSISTENCIA

1. Conceptos básicos

Un **fichero** puede verse como una porción de un dispositivo de almacenamiento no volátil (disco duro, disquete, etc.) a la que se le asocia un determinado nombre. Por no volátil se entiende que a diferencia de lo que ocurre con otros almacenes de datos como la memoria RAM, la información en ellos almacenadas no se pierde al apagarse el ordenador.

Dada la importancia de los ficheros como almacenes no volátiles de la información, la BCL incluye todo un espacio de nombres llamado **System.IO** especialmente orientado al trabajo con ellos.

1.1 Rutas independientes del sistema operativo

Como ya sabemos, cada sistema operativo utiliza una sintaxis propia para representar las rutas de acceso a los archivos. Esto puede llevarnos a crear código no valido para cualquier S.O. Para que esto no ocurra la BCL incorpora mecanismos que permiten escribir las aplicaciones de manera independiente del formato usado por el sistema operativa de la máquina, sobre la que se ejecute el programa. Pues bien, esos mecanismos consisten en escribir dichas rutas usando los siguientes campos de la clase Path:

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

Tabla 1: Campos de Path independizadores del formato de rutas de los sistemas operativos

Si en lugar de escribir directamente la ruta “\datos\datos.dat” escribimos:

```
String ruta = Path.DirectorySeparatorChar +”datos”+
Path.DirectorySeparatorChar + ”datos.dat”;
```

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\datos.dat”, en Unix contendría “/datos/datos.dat”

1.2 Operaciones con rutas

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 representar rutas como c:\datos con literales como “c:\datos”. En su lugar hay tres alternativas:

- Usar el campo independizador del sistema operativo, aunque ello tiene el problema de que da lugar a códigos poco compacto. Por ejemplo, para la ruta anterior habría que representarla con “c:”+Path.DirectorySeparatorChar+”\datos”.

- Duplicar los caracteres **** de los literales para que dejen de considerarse secuencias de escape. Así, la ruta de ejemplo anterior quedaría como “c:\\datos”

- 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 muchos más compacta y fácil de leer al no incluir tantos **** duplicados.

```
string Path.GetDirectoryName(string path)
```

Obtiene la ruta con el directorio padre o null si es raíz.
• 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

```
string Path.GetFileName(string path)
```

Obtiene el archivo o directorio del final de la ruta. (vació si acaba en separador)
• GetFileName(@"C:\MyDir\MySubDir\myfile.ext") devuelve "myfile.ext"
• GetFileName (@"C:\MyDir\MySubDir") devuelve "MySubDir"
• GetFileName (@"C:\MyDir\") devuelve ""

```
string Combine (string path1, string path2);
```

Combina un DirectoryName path1 y un FileName path2 para formar una nueva ruta. Siempre que path2 no empiece por el carácter separador de directorio o de volumen.
• 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") error
• Combine(@"C:\MyDir\", @"C:\MySubDir\myfile.ext") error

2. Representación de ficheros y directorios en .Net

2.1 Clase FileInfo y DirectoryInfo

En la plataforma .NET el trabajo con ficheros y directorios se hace, como no, siguiendo un enfoque orientado a objetos, de manera que los ficheros se encapsulan en objetos FileInfo y los directorios en objetos DirectoryInfo basados en la clase abstracta FileSystemInfo. Algunos de los elementos comunes a ambos serían:

- Determinación de existencia → Para ello se mira el valor de la propiedad **bool Exists**.
- Crear → Se utiliza el método **Create().**
- Borrado → Se proporciona el método **Delete()** que posibilita el borrado del objeto. Sin embargo, al usarlo hay que tener en cuenta que si el elemento a borrar es un directorio, éste ha de estar vacío o utilizar la sobrecarga con el parámetro a true.
- Mover → El método **MoveTo**(ruta) te mueve el objeto a la ruta que le indiques.
- Acceso a información → Permite obtener información del objeto utilizando las propiedades de solo lectura **FullName, Name** o **Extension**.
- Acceso al contenido de los ficheros → Para acceder al contenido de los ficheros se ofrece un método sobrecargado **Open() .**
- Actualización de los atributos → Para ello se utiliza el método **Refresh()** que nos devuelve el objeto con todos sus atributos actualizados.
- Acceso a los atributos de los ficheros y directorios → Con la propiedad **Attributes** podemos acceder a ellos tanto para leerlos como para modificarlos. Esta propiedad es de un tipo enumerado llamado **FileAttributes** que admite los siguientes literales, que pueden ser propios de uno u otro S.O.:

Literal	**Significa que el fichero o directorio...**
Normal	Es fichero normal y corriente, sin ningún otro atributo
Directory	Es un directorio, no un fichero
Hidden	Ha de considerarse como oculto y no mostrarse en los listados normales del contenido de su directorio padre
System	Forma parte del sistema operativo
ReadOnly	Es de sólo lectura

Encrypted	Está cifrado. Si es un directorio entones los ficheros que se le añadan se cifrarán automáticamente
Compressed	Está comprimido
Temporary	Es temporal y será borrado por el sistema operativo cuando finalice la ejecución de la aplicación que lo creó
Archive	Está archivado. Este atributo suelen activarlo las aplicaciones de backup para marcar los archivos o directorios de los que se dispone de copias de seguridad
Offline	No está disponible debido a que está almacenado en una máquina remota con la que no se puede conectar
NotContentIndexed	No ha de ser indexado por los servicios de indexado de contenidos de los que disponga el sistema operativo
SparseFile	Es un fichero esparcido, lo que significa que es un fichero grande cuyo contenido son casi todo ceros.
ReparsePoint	Contiene un bloque de datos asociado con información configurable por el usuario

Tabla 2: Literales indicadores de atributos de ficheros de FileAttributes

Hay que tener en cuenta que la propiedad **Attributes** puede almacenar combinaciones OR de los literales de **FileAttributes.** Un ejemplo para convertir ficheros o directorios en ocultos y de sólo lectura puede escribirse así:

```
public void ConvierteEnOcultoDeSoloLectura(FileSystemInfo f)
{
      f.Attributes = FileAttributes.Hidden | FileAttributes.ReadOnly;
}
```

Ejemplo:

```
static void MuestraInfo(FileSystemInfo f)
{
   if (f.Exists) {
        Console.WriteLine("Nombre completo: {0}", f.FullName);
        Console.WriteLine("Nombre : {0}", f.Name);
        Console.WriteLine("Extensión : {0}", f.Extension);
        Console.WriteLine("Fecha creación: {0}", f.CreationTime);
        Console.WriteLine("Fecha último acceso: {0}", f.LastAccessTime);
        Console.WriteLine("Fecha última modificación: {0}", f.LastWriteTime);
        Console.WriteLine("Atributos: {0}", f.Attributes.ToString());}
   else Console.WriteLine("Archivo no encontrado");
}
static void Main(string[] args)
{
    FileInfo f = new FileInfo(Console.ReadLine());
    MuestraInfo(f);
}
```

2.2 Clase File y Directory

Son dos clases estáticas que complementan a las anteriores, contiene gran cantidad de métodos estáticos de utilidad para hacer operaciones con ficheros del estilo de las que se hacen desde la línea de comandos.

```
char s = Path.DirectorySeparatorChar;
string ruta = $".{s}datos{s}datos.txt";
if (Directory.Exists(Path.GetDirectoryName(ruta)) == false)
Directory.CreateDirectory("datos");
File.Create(ruta).Close();
Directory.SetCurrentDirectory(Path.GetDirectoryName(ruta));
Console.Write("El fichero " + ruta + " ");
Console.WriteLine(File.Exists(Path.GetFileName(ruta))? "existe": "no exite");
```

Ejemplo en el que se aplica la funcionalidad de las clases de los dos puntos anteriores. Aplicación que lista el contenido del home:

```
publicstatic void Main()
{
  string home =(Environment.OSVersion.Platform == PlatformID.Unix ||
                Environment.OSVersion.Platform == PlatformID.MacOSX)
                ? Environment.GetEnvironmentVariable("HOME")
                : Environment.ExpandEnvironmentVariables("%HOMEDRIVE%%HOMEPATH%");
  if (Directory.Exists(home)) // Compruebo si existe la ruta
  {
     // Me situo en el directorio home.
     Directory.SetCurrentDirectory(home);
     DirectoryInfo infoCarpeta = new DirectoryInfo(home);
     // Obtengo información de todo lo que hay
     // en la carpeta archivos y directorios.
     FileSystemInfo[] infosEnFS = infoCarpeta.GetFileSystemInfos();
     foreach (FileSystemInfo infoEnFS in infosEnFS)
     {
        // Compruebo si en la máscara me indica que es una carpeta.
        bool esCarpeta =
       ((infoEnFS.Attributes & FileAttributes.Directory) ==
                       FileAttributes.Directory);
       // Compruebo si en la máscara me indica que es un archivo.
       bool esArchivo =
       ((infoEnFS.Attributes & FileAttributes.Archive) ==
                     FileAttributes.Archive);
       // Muestro el nombre completo indicando si es un archivo o una carpeta.
       Console.WriteLine((esCarpeta ? "Carpeta" : "Archivo") +
                     " -> " + infoEnFS.FullName);
     }
   }
   else Console.WriteLine("No se ha podido encontrar la carpeta home.");
}
```

3. Flujos de datos en c#

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 stream pueden ser transferidos en dos posibles direcciones:

Si los datos son transferidos desde una fuente externa al programa, entonces se habla de “leer desde el stream”.
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 ...

3.1 Volcado de datos en flujos

Muchos flujos 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 físico de los bytes en un flujo. En esos casos puede forzarse el volcado llamando al método **Flush()** del flujo, que vacía por completo su buffer interno.

3.2 Cierre de flujos

Además de memoria, muchos flujos acaparan recursos extra que interesa liberar una vez dejen de ser útiles. Por ejemplo, las conexiones de red acaparan sockets y los ficheros acaparan manejadores de ficheros del sistema operativo, que son recursos limitados y si no liberarlos podría impedir la apertura de nuevos ficheros o conexiones de red. En principio es tarea del método **Dispose()** con el que todo flujo cuenta como estándar recomendado para liberar recursos de manera determinista. Sin embargo, por similitud con otros lenguajes a la clase Stream también dispone de un método **Close()** que hace lo mismo.

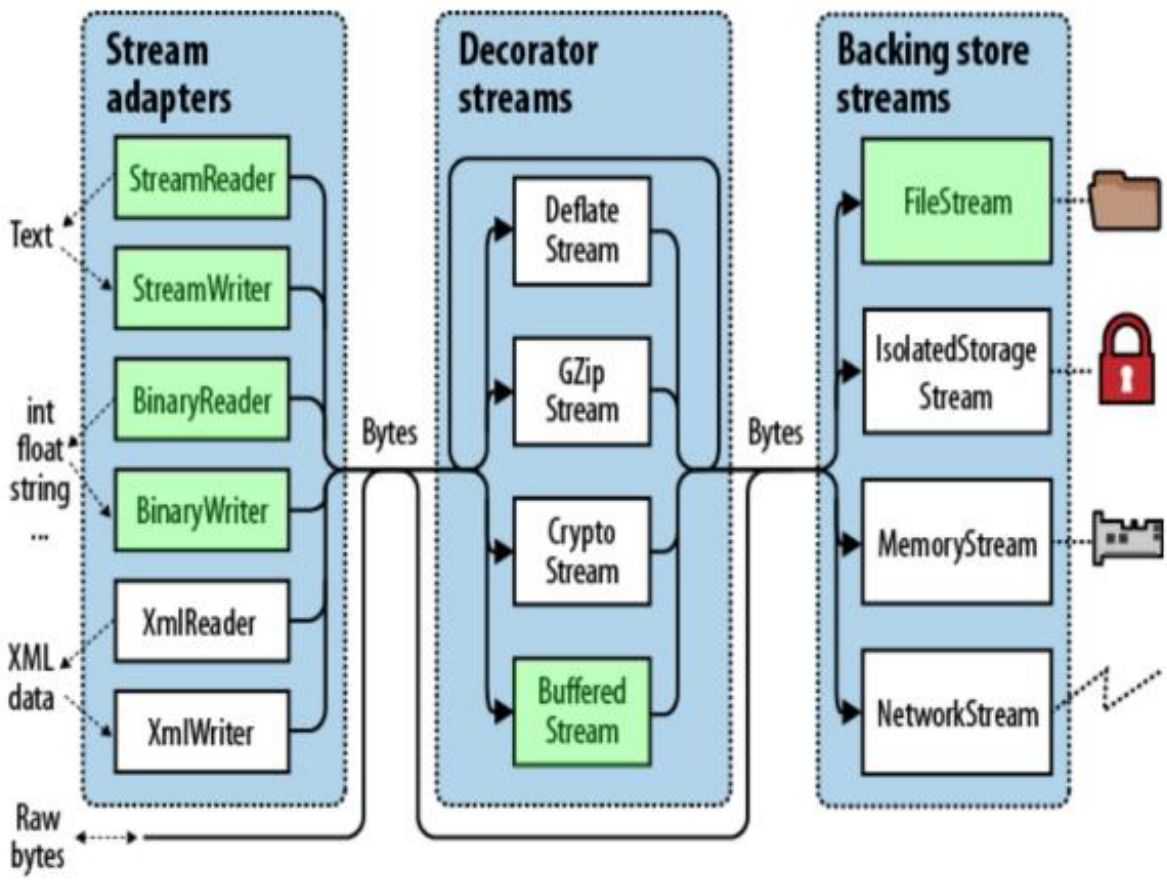
3.3 Lectura y escritura de datos con Flujos

La clase **BufferedStream** se utiliza para leer y para escribir a otro stream, debido a que el uso de stream para la lectura y escritura de archivos es lento se optimiza utilizando **BufferedStream**. Aunque para operaciones de archivo es posible utilizar **FileStream**, donde el buffering está ya incluido.

Las clases más relacionadas con la escritura y lectura de archivos son:

FileStream, cuyo propósito es lectura y escritura de datos binarios (no de texto legible), aunque se puede utilizar para acceder a cualquier tipo de archivo, inclusive los de texto.

StreamReader y StreamWriter, las cuales están diseñadas para lectura y escritura de archivos de texto. Estas clases se asumen como de un nivel más alto que **FileStream.**



4. Lectura y escritura de archivos binarios

4.1 Crear objetos FileStream

La clase **FileStream** proporciona una familia de constructores, que crearan objetos que permitirán trabajar con archivos binarios, la sobrecarga más completa del constructor es la siguiente:

```
FileStream(string rutaFichero, FileMode modoApertura, FileAccess modoAcceso, FileShare modoCompartición)
```

Donde los literales correspondientes a cada parámetro los podemos ver en las siguientes tablas:

Literal de FileMode	Modo de apertura
Open	Abre el fichero presuponiendo existe. Si no fuese así se lanza una **FileNotFoundException**.
OpenOrCreate	Si el fichero no existe, lo crea
CreateNew	Abre el fichero presuponiendo que no existe y creándolo. Si existiese se lanzaría una **IOException**
Create	Crea el fichero, y si ya existía lo sobreescribe
Truncate	Abre el fichero presuponiendo que existe y borrando todo su contenido. Si no existiese se lanzaría una **FileNotFoundException**
Append	Abre el fichero en modo de concatenación, lo que significa que sólo podrá escribirse a su final. Si el fichero no existe, lo crea.

Tabla 3: Modos de apertura de ficheros (enumeración FileMode)

Literal de FileAccess	Modo de acceso solicitado
Read	Sólo lectura del contenido
Write	Sólo escritura de contenido
ReadWrite	Tanto lectura como escritura de contenido

Tabla 4: Modos de acceso a ficheros (enumeración FileAccess)

Literal de FileShare	Modo de compartición
None	No se puede compartir
Read	Puede compartirse con hilos que sólo quieran leerlo
Write	Puede compartirse con hilos que sólo quieran escribir en él
ReadWrite	Puede compartirse con cualquier otro hilo

Tabla 5: Modos de compartición de ficheros (enumeración FileShare)

Por ejemplo, para leer el contenido del fichero c:\datos.dat podemos generar un **FileStream** así:

```
FileStream contF = new FileStream(@"c:\datos.dat", FileMode.Open,
FileAccess.Read);
```

4.2 Lectura y escritura con FileStream

Como es lógico, todo flujo dispondrá de mecanismos mediante los que se le puedan extraer y añadir bytes. Para la lectura se ofrecen los métodos **ReadByte()** y **Read()** que, respectivamente, permiten extraerles uno o varios bytes (se almacenarían en una tabla byte[]); y para la escritura se hace lo mismo con **WriteByte()** y **Write().**
Tras cada lectura, la siguiente llamada a estos métodos devolvería los bytes del flujo siguientes a los últimos leídos, y para detectar cuando se alcance el final del flujo basta mirar su valor de retorno, pues en ese caso **ReadByte()** devolverá un –1 y **Read()** un 0. Respecto a la escritura, cada vez que se escriba se escribirá a continuación de los últimos bytes escritos en el flujo.

Propiedades interesantes de la clase **FileStream**:
- **long Length.** Número de bytes almacenados en el flujo (tamaño del flujo). En ficheros, el tamaño en bytes del fichero.
- **long Position**: Número del byte actual en el flujo. En ficheros, la posición actual donde se encuentra el descripto de lectura o escritura del fichero. Dándole valores podemos movernos por el flujo.

.

```
Console.SetCursorPosition(0, 17);
FileStream fichero = new FileStream("ejemplo.txt", FileMode.Open, FileAccess.Read);
Console.WriteLine("Longitud Fichero: " + fichero.Length);

// Devolverá un 0
Console.WriteLine("Posicion descriptor lectura: " + fichero.Position);
fichero.ReadByte();
// Devolverá un 1
Console.WriteLine("Posicion descriptor lectura: " + fichero.Position);
fichero.ReadByte();
fichero.ReadByte();
// Devolverá un 3
Console.WriteLine("Posicion descriptor lectura: " + fichero.Position);
fichero.Close();
```

Ej: con while(f.Position < f.Lenght) puedo saber si estoy al final de un stream.

4.3 Movimiento por el flujo

Por defecto, la primera operación que se realice sobre un flujo se aplica a su comienzo y las siguientes se aplican tras la posición resultante de la anterior. Es decir, en cada lectura o escritura se lee o escribe a continuación de la última posición accedida, por lo que antes de acceder a una determinada posición habrá que pasar antes por las previas. A esto se le conoce como acceso secuencial.
Sin embargo, hay situaciones en las que puede resultar interesante poderse escribir o leer de cualquier posición del flujo sin necesariamente tener que pasar antes por sus anteriores. A esto se le conoce como acceso aleatorio.
Si un flujo admite acceso aleatorio le serán aplicables los miembros de Stream relativos a dicho tipo de acceso que a continuación se muestran:

- **SetLength(long tamaño)**. Cambia el número de bytes almacenados en el flujo por el indicado. Si este nuevo tamaño es inferior al que tenía se truncarán los bytes que no sobren, mientras que si es superior se rellenarán los nuevos bytes con valores inespecificados.

  Ejemplo:

```
static void MuestraByte(FileStream flujo, long númeroByte)
{
      long posiciónAnterior = flujo.Position;
       if (númeroByte > flujo.Length)     Console.WriteLine("Fin archivo");
       else  {
                flujo.Position = númeroByte;
                Console.WriteLine("Byte {0} = {1}", númeroByte, flujo.ReadByte());
                flujo.Position = posiciónAnterior;
             }
}
```

- **long Seek(long posición, SeekOrigin inicio)**: Permite colocarnos en un byte del flujo determinado. La posición de éste se indica de manera relativa respecto a la posición de inicio señalada por su segundo parámetro, que es de un tipo de enumeración cuyos posibles literales son **Current** (posición actual en el flujo), **Begin** (inicio del flujo) y **End** (final del flujo)

Ejemplo:

```
// Coloca justo antes del comienzo del flujo
flujo.Seek(0, SeekOrigin.Begin);
// Coloca dos bytes a continuación de la posición actual en el flujo
flujo.Seek(2, SeekOrigin.Current);
// Coloca en el penúltimo byte del flujo
flujo.Seek(-2, SeekOrigin.End);
```

4.4 Stream Decorator BufferedStream

Esta clase agrega una capa de almacenamiento en buffer a las operaciones de lectura y escritura en otra secuencia. Aunque FileStream ya tiene un buffer de escritura intermedio, se puede ampliar mediante esta capa de abstracción.

```
static void Main(string[] argumentos)
{
    DateTime t1 = DateTime.Now;
    FileStream fichero = new FileStream(
                         "prueba.pru", FileMode.Create, FileAccess.Write);
    for (int i = 0; i < 100000000; i++)   fichero.WriteByte(33);
    fichero.Close();
    DateTime t2 = DateTime.Now;
    TimeSpan dif = t2 - t1;
    Console.WriteLine($"Sin BufferedStream Seg = {dif.Seconds}s {dif.Milliseconds}ms");
    fichero = new FileStream("prueba.pru", FileMode.Create, FileAccess.Write);
    BufferedStream ficheroBuff = new BufferedStream(fichero, 1000);
    for (int i = 0; i< 100000000; i++) ficheroBuff.WriteByte(33);
    ficheroBuff.Close();
    DateTime t3 = DateTime.Now;
    dif = t3 - t2;
    Console.WriteLine($"Con BufferedStream Seg = {dif.Seconds}s {dif.Milliseconds}ms");
}
```

¿Qué pasa si aumentamos el BufferedStream a 1000000?
¿Qué pasa si hacemos un Flush() después de cada escritura ?

4.5 BinaryReader y BinaryWriter

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. Por esto, en System.IO se han incluido un par de tipos llamados **BinaryReader** y **BinaryWriter** que encapsulan **FileStreams** y les proporcionan una serie de métodos con los que se simplifica la lectura y escritura de objetos de cualquier tipo en ficheros.

Para crear objetos de estos tipos usaremos los constructores:

`BinaryReader(Stream flujo, Encoding codificación)`
`BinaryWriter(Stream flujo, Encoding codificación)`

Por defecto, se considera que la codificación de las cadenas de texto que se lean o escriban del flujo es UTF-8, pero con el parámetro Encoding se puede especificar cualquier otra de la siguiente tabla:

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

Tabla 6: Propiedades indicadores de tipo de codificación incluidas en Encoding

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.

Los **BinaryWriters** se caracterizan por disponer de un método **Write()** con múltiples sobrecargas que toman parámetros de cualquiera de los tipos básicos excepto **object** y los escriben directamente en el flujo que encapsulan.

Análogamente, los **BinaryReader**s disponen de una familia de métodos **ReadXXX()** (**int ReadInt()**, **string ReadString()**, etc.) que permiten leer del flujo cualquier tipo básico. Además, adicionalmente se les ha añadido un método **int PeekChar()** que permite consultar el siguiente carácter del flujo en forma de **int** (o un –1 si no quedasen más) pero sin extraerlo del mismo, de forma que la siguiente lectura se realizaría como si la consulta nunca hubiese sido realizada.

Por ejemplo, para escribir en un flujo que represente a un fichero llamado datos.dat usando UTF8 al codificar los caracteres puede hacerse lo siguiente:

```
FileStream fichero = new FileStream("datos.dat", FileMode.Create);
 BinaryWriter ficheroBinario = new BinaryWriter(fichero, Encoding.UTF8);
 ficheroBinario.Write("Este mensaje se escribirá en UTF8 dentro de datos.dat");
```

Del mismo modo, para leer correctamente dicho mensaje habría que hacer algo como:

```
FileStream fichero = new FileStream("datos.dat", FileMode.Open);
BinaryReader ficheroBinario = new BinaryReader(fichero, Encoding.UTF8);
Console.WriteLine("Leido de datos.dat: {0}", ficheroBinario.ReadString());
```

5. Lectura y escritura de archivos de texto

Para la lectura y escritura de texto en ficheros se utilizan las clases **StreamReader** y **StreamWriter,** que son dos subclases de las clases abstractas **TextReader** y **TextWriter** que definen características para trabajar con fuentes genéricas.

5.1 Constructores de StreamReader y StreamWriter

Especializada en la extracción de texto de flujos. Sus constructores básicos son:

```
StreamReader(string rutaFichero, Encoding codificacion)
StreamWriter(string rutaFichero, bool concatenar, Encoding codificacion)
```

La codificación tomada por defecto es UTF-8, aunque puede cambiarse dándole el valor apropiado al parámetro codificación. Como se ve, el constructor StreamWriter toma un parámetro adicional llamado concatenar. Este parámetro permite indicar qué ha de hacerse si el fichero donde se desea escribir ya existe. Si vale false será sobreescrito, pero si vale true se concatenarán a su final los nuevos datos que se escriban en él. Por defecto vale true.

5.2 Métodos de Lectura

La familia de métodos que permiten leer caracteres de diferentes formas son:

- **De uno en uno:** El método **int Read()** devuelve el próximo carácter del flujo como, 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()** que funciona como **Read()** pero no avanza la posición en el flujo tras la lectura.
- **Por grupos:** El método **int Read(out char[] caracteres, int inicio, int nCaracteres)** lee un grupo de nCaracteres 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 nCaracteres si el flujo tenía menos caracteres de los indicados o un –1 si se ha llegado al final del flujo.
- **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. Por compatibilidad con los sistemas operativos más frecuentemente usados, considera que una línea de texto es cualquier secuencia de caracteres terminada en **'\n'**, **'\r'** ó **"\r\n"**, aunque la cadena que devuelve no incluye dichos caracteres.
- **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)

Ejemplo:

```
static void Main(string[] args)
{
  if (args.Length < 1)
  {
    Console.WriteLine("Error: Llamada incorrecta. Formato correcto de uso:");
    Console.WriteLine("\n\t NombrePrograma <rutaFichero>");
  }
```

```
  else
  {
    String nombreFichero = args[0];
    StreamReader contenido = new StreamReader(nombreFichero);
    Console.WriteLine("Contenido {0}:\n{1}",nombreFichero, contenido.ReadToEnd());
  }
}
```

5.3 Métodos de Escritura

La familia de métodos que permiten escribir caracteres son:

- **Escribir cadenas de texto:** El método **Write()** permite escribir cualquier cadena de texto en el destino que tengan asociado. Esta cadena se les pasa como primer parámetro, pero no tiene porqué ser un **string** sino que puede ser un objeto de cualquier tipo y lo que se haría sería aplicarle su método **ToString()** para obtener su representación en forma de cadena de texto. Este método se puede usar de forma equivalente al **Write()** de **Console.**
- **Escribir líneas de texto: WriteLine()** funciona igual que el anterior pero añade un indicador de fin de línea.

Ejemplo:

```
  static void Main(string[] args)
  {
     String nombreFichero = @"c:\datos\datos.dat";
     StreamWriter contenido=new StreamWriter(nombreFichero,true,Encoding.UTF8);
     contenido.WriteLine(Console.ReadLine());
     contenido.Close();
  }
```

En el archivo datos.dat se introducirá la cadena que recojamos de teclado, si el archivo no existe se crea y en caso de que exista se añadirá la otra cadena al final.

6. Persistencia y Serialización

Se define por persistencia en el mundo de la POO, como la capacidad que tienen los objetos de sobrevivir al proceso padre que los creo. La persistencia permite al programador almacenar, transferir y recuperar fácilmente el estado de los objetos.

¿Cómo Podemos Conseguir La Persistencia?

La forma más común de conseguirlo es mediante la serialización.

La serialización es el proceso de convertir el estado de un objeto a un formato que se pueda almacenar o transportar. El complemento de la serialización es la deserialización, que convierte una secuencia a un objeto. Juntos, estos procesos permiten almacenar y transferir fácilmente datos.

.NET Framework ofrece dos tecnologías de serialización:

- La serialización binaria conserva la fidelidad de tipos, lo que resulta útil para conservar el estado de un objeto entre distintas llamadas a una aplicación. Puedes serializar un objeto en una secuencia, un disco, la memoria, a través de la red, etc. Por ejemplo, se puede compartir un objeto entre distintas aplicaciones si lo serializa en el Portapapeles.
- La serialización XML sólo serializa las propiedades públicas y los campos, y no conserva la fidelidad de tipos. Esto resulta útil cuando desea proporcionar o consumir datos sin restringir la aplicación que utiliza los datos. Como XML es un estándar abierto, es una opción atractiva para compartir datos a través del Web.

- La serialización a alguna notación de objetos estándar como JSON.

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

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

A la hora de serializar una clase llamaríamos a su método Serializa y este a su vez a los Serializa de los objetos y tipos que contenga, así sucesivamente. Muchos lenguajes como Java o C# solucionan la serialización de forma sencilla, ya que al serializar un objeto contenedor, este a su vez serializa mediante un mecanismo de reflexión y de forma trasparente aquellas referencias a objetos que contiene. Lo mismo sucede al cargar o deserializar un objeto.

Durante este proceso, los campos público y privado del objeto y el nombre de la clase, incluido el ensamblado que contiene la clase, se convierten en una secuencia de

bytes que, a continuación, se escribe en una secuencia de datos. Cuando, después, el objeto se deserializa, se crea una copia exacta del objeto original.

Para ello C# me ofrece marcar mis clases como serializables a través de **un atributo**.

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

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

[serializable]

```
[Serializable]
public class Alumno
{
    private string nombre;
    private string apellido;
    private int edad;
    public Alumno(string nombre, string apellido, int
edad)
    {
        this.nombre = nombre;
        this.apellido = apellido;
        this.edad = edad;
    }
    public override string ToString()
    {
        return nombre + " " + apellido;
    }
}
```

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

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

En el siguiente ejemplo de código se muestra cómo serializar una instancia de esta clase en un archivo.

```
    static void Main(string[] argumentos)
    {
        Alumno a = new Alumno("Pepa", "Pérez", 25);
        Stream s = new FileStream("Dato.bin", FileMode.Create, FileAccess.Write);
        IFormatter f = new BinaryFormatter();
        f.Serialize(s, a);
        s.Close();
     }
```

Este ejemplo utiliza un formateador binario para realizar la serialización. Todo lo que necesita es crear una instancia de la secuencia y el formateador que desee utilizar y, a continuación, llamar al método **Serialize** en el formateador.

Igual de sencillo resulta restaurar el objeto a su estado anterior. En primer lugar, cree una secuencia para leer y un formateador a continuación, indique el formateador que deserialice el objeto. Su generalización sería:

```
IFormatter <nombreobjeto> = new BinaryFormatter();
Objeto=<nombreobjeto>.Deserialize(<medioalmacenamiento>);
```

En el siguiente ejemplo de código se muestra cómo hacerlo.

```
static void Main(string[] argumentos)
{

    Stream s = new FileStream("Dato.bin", FileMode.Open, FileAccess.Read);
    IFormatter f = new BinaryFormatter();
    Alumno a = f.Deserialize(s) as Alumno;
    Console.WriteLine(a ?? new Alumno("Desconocido", "", 0));
    s.Close();
 }
```

Es importante tener en cuenta que el atributo **Serializable** no se puede heredar. Si deriva una nueva clase de MiClase, la nueva clase debe marcarse también con el atributo; de lo contrario, no se podrá serializar.

Una clase a menudo contiene campos que no se deben serializar. Por ejemplo campos específicos que almacenen datos confidenciales. Si no se excluye estos campos de la serialización, los datos que almacenan dichos campos estarán expuestos a cualquier código que tenga permiso de serialización. Para que un campo no se serialize, deberemos aplicarle el atributo [**NonSerialized].**

```
[Serializable]
public class MiClase
{
  public int n1;
  [NonSerialized] public int n2;
  public String str;
}
```

¿Cómo podemos saber si una clase es serializable o no? Podemos utilizar el API de Reflexión para consultar los atributos de un tipo. Una posible forma sería:

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

Si quisieramos refactorizar el código para pasar la responsabilidad de la serialización a la clase serializable, tendríamos que implementar las operaciones que definimos al principio dentro de la propia clase. Ejemplo más completo de serialización para ver esto último:

```
[Serializable]
class Persona
{
    string nombre;
    int edad;
    string nif;
    public Persona() { }
    public Persona(string nombre, int edad, string nif)
    {
        this.nombre = nombre;
        this.edad = edad;
        this.nif = nif;
    }
    void Cumpleaños()
    {
        edad++;
    }
    public void Serializar(Stream s)
    {
        IFormatter formatter = new BinaryFormatter();
        formatter.Serialize(s, this);
    }
    public static Persona Deserializar(Stream s)
    {
         return (Persona)new BinaryFormatter().Deserialize(s);
    }
    public override string ToString()
    {
        return nombre + " " + edad;
    }
}
class Program
{
    static void Main()
    {
        FileStream s = new FileStream(@"C:\datos.txt", FileMode.Append,
FileAccess.Write);
        Persona p = new Persona("lulu", 10, "22112211L");
        p.Serializar(s);
        s.Close();
        s = new FileStream(@"C:\datos.txt", FileMode.Open, FileAccess.Read);
        Console.WriteLine( Persona.Deserializar(s));
    }
}
```