FICHEROS ENTRADA/SALIDA DE DATOS PERSISTENCIA

MANEJO DE ARCHIVOS Y DIRECTORIOS

Rutas: Clase Path - I

- Nos ayuda a configurar rutas a archivos de forma independiente al SO en que nos encontremos.

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 /

Por ejemplo para la ruta "\datos\datos.dat" escribiremos:

```
char s = Path.DirectorySeparatorChar;

string ruta = $"{s}datos{s}datos.dat";
```

MANEJO DE ARCHIVOS Y DIRECTORIOS

Rutas: Clase Path – II

```
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**

Obtiene el archivo o directorio del final de la ruta. (vació 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 **""**

MANEJO DE ARCHIVOS Y DIRECTORIOS

Rutas: Clase Path – II

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

MANEJO DE ARCHIVOS Y DIRECTORIOS

Clases File Y Directory

- 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.

```
static void Main()
{
    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");
}
```

MANEJO DE ARCHIVOS Y DIRECTORIOS

Clases DirectoryInfo Y FileInfo

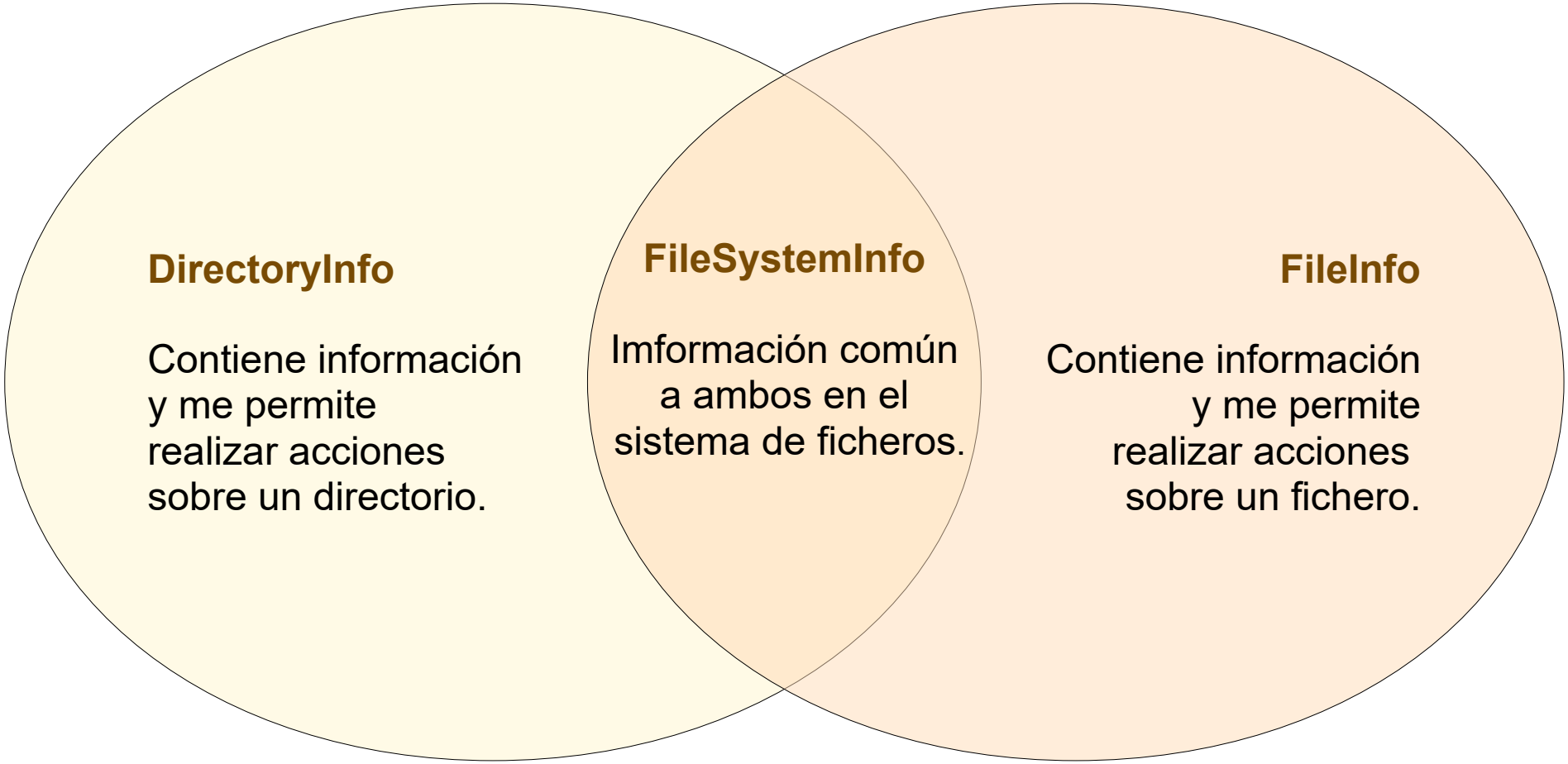
- Clases no estáticas para manejo (deberán ser instanciadas con new).
- Me permitirán navegar y realizar acciones con más precisas y con más detalle sobre el sistema de ficheros.

Clase FileSystemInfo

- No podrá se instanciada sino que accederé a través de DirectoryInfo y FileInfo pues contine información común a ambos.
- 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 directoros.

MANEJO DE ARCHIVOS Y DIRECTORIOS

Clases DirectoryInfo Y FileInfo



MANEJO DE ARCHIVOS Y DIRECTORIOS

Ejemplo Programa Que Lista El Contenido De Nuestro Home

```
static 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.");
}
```

FLUJOS DE DATOS SERIE O STREAMS

GENERALIDADES

Concepto De Archivo O Fichero

- Los programas se comunican con el entorno leyendo y escribiendo ficheros.
- Un fichero se considera como flujo de datos (Stream) que se procesan de forma secuencial.
- El concepto de fichero es amplio:
  - Fichero de disco: Conjunto de datos que se puede leer y escribir repetidamente.
  - Tubería (pipe), canal: Flujo de bytes generados por un programa.
  - Dispositivos: Flujo de "bytes" recibidos desde o enviados hacia un dispositivo periférico (teclado, monitor, ...).
- Todas estas clases de ficheros se manejan casi de la misma forma.
- En C# los archivos, directorios y flujos con ficheros, se manejan con clases del las BCL definidas en el namespace: System.IO

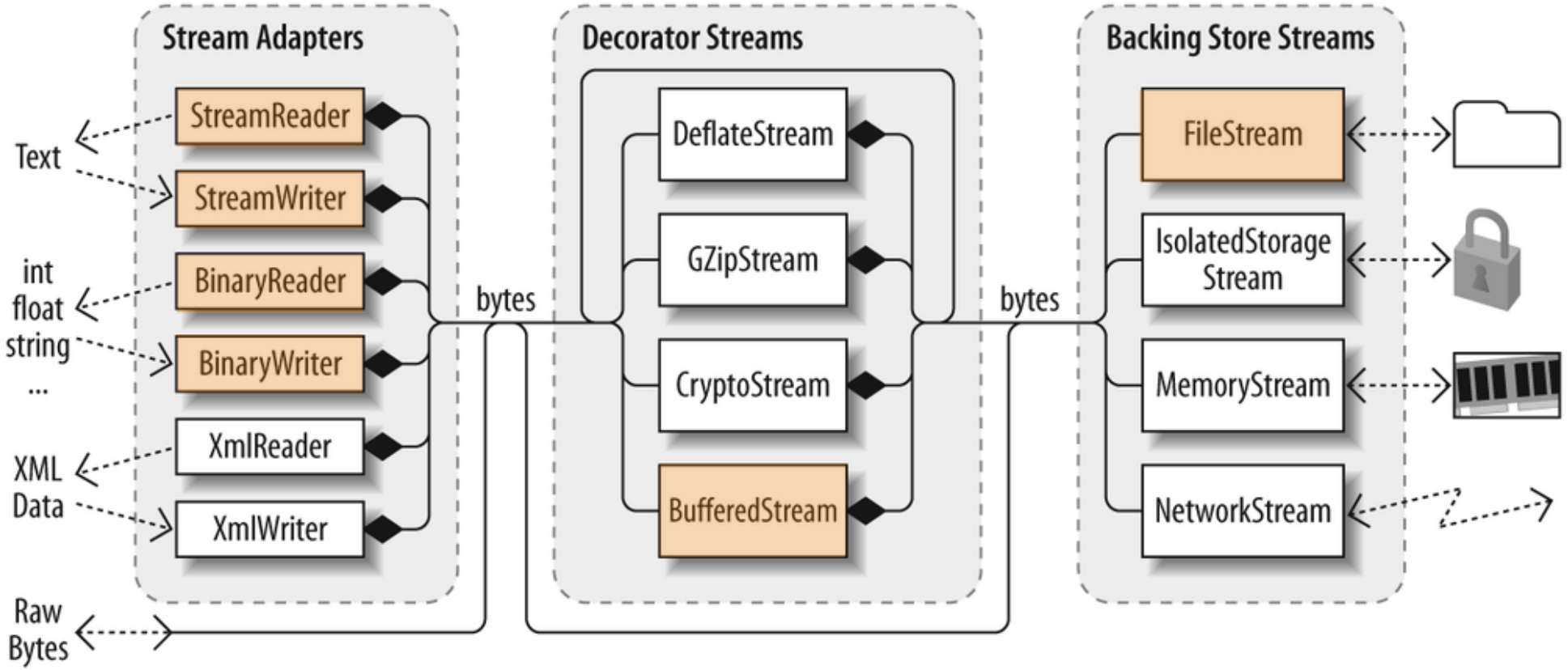
LECTURA Y ESCRITURA DE FICHEROS EN C# - I

Flujos De Datos De Entrada O Salida (Streams)

- Como hemos comentado antes estos flujos no solo se utilizan para escribir y leer en un archivo de disco duro en serie. Sino también para otras operaciones como...
  - R/W en un área de memoria.
  - La consola.
  - La impresora.
  - Red.
- Todos los flujos de datos en C# heredan de la clase Stream.
- En concreto para ficheros usaremos la clase FileStream.
- Tras crear un flujo de datos diremos que lo estamos abriendo.
- Tras finalizar el flujo de datos deberemos cerrarlo con el método Close().
- Normalmente las operaciones de R/W se realizan sobre un buffer de memoria que se volcará (Flush) de forma transparente o forzada.
- Escribiremos y leeremos flujos de bytes (Como unidad básica).

LECTURA Y ESCRITURA DE FICHEROS EN C# - I

Diagrama De Clases Para Uso De Streams En C#



LECTURA Y ESCRITURA DE FICHEROS EN C# - II

Stream Para Ficheros FileStream - Creación

La forma de crearlo más común es con:

```
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.

LECTURA Y ESCRITURA DE FICHEROS EN C# - II

Stream Para Ficheros FileStream - Creación

- FileAccess: 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.

```
using System.IO;
namespace Ejemplo
{
    class Program
    {
        static void Main()
        {
            FileStream fichero = new FileStream(
                        "ejemplo.txt",      // Nombre del fichero
                        FileMode.Create,    // Lo creo o sobreescribo
                        FileAccess.Write);  // Solo puedo escribir.
                    fichero.Close();
        }
    }
}
```

LECTURA Y ESCRITURA DE FICHEROS EN C# - II

Stream Para Ficheros FileStream – Escritura

```
public override void 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.

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

byte[] datos = new byte[] { 0x31, 0x30, 0x30 };

// Escribo 3 bytes al principio.
fichero.Write(datos, 0, datos.Length);

// Volcamos a disco el buffer intermedio.
// De lo contrario,  se volcará automáticamente
// en llenarse el buffer o al cerrar el stream.
fichero.Flush();
fichero.Close();
```

LECTURA Y ESCRITURA DE FICHEROS EN C# - II

Stream Para Ficheros FileStream - Lectura

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

- byte[] array: Array a rellenar con los bytes leídos.

  Debe estar dimensionado 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.

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

byte[] datos = new byte[3];
int bytesLeidos = fichero.Read(datos, 0, datos.Length);
for (int i = 0; i < bytesLeidos; i++)
    Console.Write($"{datos[i]:X} ");
fichero.Close();
```

Stream Para Ficheros FileStream – 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.

  En ficheros, la posición actual donde se encuentra el descripto de lectura o escritura del fichero.

```
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();
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.

Stream Para Ficheros FileStream – Desplazamiento 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**, **End**.

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

if (fichero.CanSeek) // Para streams que no admiten desplazamientos
{
    // Me situo al final y me desplzado 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();
```

EJEMPLO DE STREAM DECORATOR

Stream Decorator BufferedStream

- 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, nosotros podremos ampliarlo mediante esta capa de abstracción.

Stream Decorator BufferedStream - Ejemplo

```
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 ?

STREAM ADAPTERS DE UTILIDAD EN FICHEROS – I

Razón De Ser

- 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<Tipo> 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 Read**String**

STREAM ADAPTERS DE UTILIDAD EN FICHEROS – I

Ejemplo Adaptador BinaryWriter

```
FileStream stream = new FileStream(
                    "ejemplo.txt", FileMode.Create, FileAccess.Write);
BinaryWriter adaptadorDeStream =
                    new BinaryWriter(stream, Encoding.UTF8);
int valor = 10;
adaptadorDeStream.Write(valor);
adaptadorDeStream.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.Write("Valor guardado en binario con fotmato(Little-Indian):");
foreach (byte _byte in bytesInt) Console.Write($"{_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 DE UTILIDAD EN FICHEROS - II

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.

STREAM ADAPTERS DE UTILIDAD EN FICHEROS – I

Ejemplo Adaptador StreamWriter

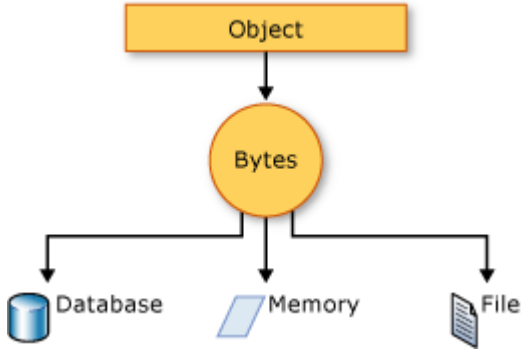
```
FileStream stream = new FileStream(
                    "ejemplo.txt", FileMode.Create, FileAccess.Write);
string text = "限定桶「冬凪」「高雅」キャンペーン掲載";
StreamWriter adaptadorDeStream =
                    new StreamWriter(stream, Encoding.Unicode);
adaptadorDeStream.WriteLine(text);
adaptadorDeStream.Close();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

Ejemplo Adaptador StreamReader

Si está a true, tomará la codificación a partir de los bytes del BOM, si están definidos en el archivo.

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

SERIALIZACIÓN



SERIALIZACIÓN CONCEPTOS GENERALES

Persistencia

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

SERIALIZACIÓN CONCEPTOS GENERALES

¿A Qué Serializaremos Una Clase?

- Lo más común es:
  - Hacerlo a un simple flujo de bytes en binario ya sea a disco o a memoria.
  - A la definición de algún lenguaje XML.
  - A alguna notación de objetos estándar como JSON.
- Aunque la mayoría de lenguajes ya la traen implementada. Si tuviéramos implementar nosotros las operaciones para serializar objetos.

  Podría ser algo parecido a esto...

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

A la hora de serializar una clase llamaríamos a su método Serializa y este a su vez a los Serializa de los objetos y tipos que contenga, así sucesivamente.

SERIALIZACIÓN EN C#

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

Atributos En .NET

- Un Atributo en .NET es una etiqueta de la sintaxis [nombre] que podremos aplicar a un ensamblado, clase, constructor, enumeración, campo, interfaz, método, etc… y que genera información en el ensamblado en forma de metadatos heredando de la clase Attribute.
- Por ejemplo si queremos realizar una simple serialización binaria etiquetaremos la clase a serializar y todas las que contenga con el atributo ya definido [Serializable], sobre la definición de la clase.

SERIALIZACIÓN EN C#

Serialización A Binario - I

- Supongamos la siguiente clase Alumno con el atributo 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;
    }
}
```

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

```
IFormatter f = new BinaryFormatter();

f.Serialize(<medioalmacenamiento>, <objetoaserializar>);
```

SERIALIZACIÓN EN C#

Serialización A Binario – II

```
public static void Main() {
    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();
}
```

- De forma análoga realizaremos la deserialización.

```
public static void Main() {
    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();
}
```

SERIALIZACIÓN EN C#

Serialización A Binario – III

- ¿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;
```

SERIALIZACIÓN EN C#

Serialización A Binario – IV

- Una clase a menudo contiene campos que no se deben serializar. Por ejemplo campos específicos que almacenen datos confidenciales. Podremos excluir dichos campos del proceso aplicándoles el atributo [NonSerialized].

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

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

SERIALIZACIÓN EN C#

Serialización A Binario – V

- Si quisieramos refactorizar el código para pasar la responsabilidad de la serialización a la clase alumno.
- Para ello vamos a implementar las operaciones que definimos al principio.

```
[Serializable]
public class Alumno {
    …
    public void SerializaBin(Stream s) {
        new BinaryFormatter().Serialize(s, this);
    }
    public static Alumno DeserializaBin(Stream s) {
        return new BinaryFormatter().Deserialize(s) as Alumno;
    }
}
…
public static void Main() {
    Stream s = new FileStream(…, FileAccess.ReadWrite);
    new Alumno("Pepa", "Pérez", 25).SerializaBin(s);
    // O podremos hacer
    Alumno a = Alumno.DeserializaBin(s);
    s.Close();
}
```