COLEXIO VIVAS S.L.	RAMA:	Computing	CICLO:	MAP	
	MÓDULO	Service and processes programming		CURSO:	2º
	PROTOCOLO:	Reading	AVAL:	DATA:	
	Autor	Francisco Bellas Aláez (Curro)			

Acceso a Archivos en C#

Introducción

Los archivos son flujos o secuencias de bytes de E/S.

En .Net, las clases que nos permiten manejar la E/S de datos se encuentra en el espacio de nombres denominado ***System.IO*** (de Input/Output).

Para el manejo de archivos conviene especificar la importación de este espacio de nombres si no se quiere escribir el IO siempre:

```
using System.IO;
```

Las clases de manejo de archivos heredan de las clases abstractas ***Stream, TextReader*** o ***TextWriter***. Un *Stream* es una secuencia o flujo de bytes de salida o de entrada (archivos binarios) y *TextReader* y *TextWriter* permiten el manejo de archivos de texto.

Un ejemplo claro de esto último lo tenéis con la Consola donde se le envía un flujo de caracteres mediante *Write* o *WriteLine* y la consola se encarga de mostrarlos. La consola implementa objetos *TextReader* y *TextWriter* para poder hacer estas tareas.

En este tema vamos a ver clases que heredan de las anteriormente comentadas y que nos van a permitir el manejo de archivos y directorios.

Se debe tener presente siempre que aunque se trabajen con distintos “tipos de archivos” (binario/texto), en el fondo esto no es más que un convenio cómodo para los programadores. Un archivo son siempre datos binarios (ceros y unos) que tienen distinto significado según nos interese. Algo similar a los tipos de datos al declarar una variable.

Por último hay que tener en cuenta que **las operaciones sobre archivos se realizan sobre un buffer intermedio**, no directamente sobre el archivo físico. Así al estar modificando un archivo **puede no reflejarse en el disco duro** hasta que se haga el volcado del buffer al disco duro mediante la instrucción correspondiente o c**errando el archivo.**

COLEXIO VIVAS S.L.	RAMA:	Computing	CICLO:	MAP	
	MÓDULO	Service and processes programming		CURSO:	2º
	PROTOCOLO:	Reading	AVAL:	DATA:	
	Autor	Francisco Bellas Aláez (Curro)			

Archivos de Texto

La clase StreamWriter (Text Files)

Clase que permite realizar operaciones de escritura sobre un **archivo de texto**. Su utilización habitual consta de 3 fases:

- Instanciar el objeto mediante un constructor: Aquí se le indica tanto el nombre del archivo con el que se va a trabajar como si se va a reescribir el archivo o si se va a añadir texto al final del mismo (append).
- Escritura del texto mediante *Write* o *WriteLine*
- Cierre del archivo mediante *Close*. Esto además vuelca el contenido del buffer al archivo físico.

Es necesario recordar que el acceso a archivos de texto es **secuencial**.

Ejemplo (El System.IO está en el using):

```
string directory;
string separator;
string fullName;
StreamWriter s;

// If only use Windows, you don't need to do this
if (Environment.OSVersion.Platform == PlatformID.Unix)
{
    directory = Environment.GetEnvironmentVariable("HOME");
    separator = "/";
} else {
    directory = Environment.GetEnvironmentVariable("homepath");
    separator = "\\";
}
fullName = directory + separator + "testfile.txt";

// Open File
s = new StreamWriter(fullName);

// Write File
for (int i = 1; i <= 10; i++)
{
     s.Write("{0,-2}", i);
}
s.WriteLine();

for (int i = 1; i <= 10; i++)
{
    s.WriteLine($"Line :{i,3}");
}

// Close File. Actually needs try-catch.
s.Close();
```

Mediante *GetEnvironmentVariables* obtenemos el contenido de la **variable de entorno** deseada. No importa si se escribe en mayúsculas o minúsculas y no se ponen los % que la rodean habitualmente en Windows ni el $() de sistemas Unix.

También como curiosidad se puede ver una forma de detectar plataforma por si estamos realizando una clase que sea multiplataforma mediante *Environment.OSVersion.Platform.*

Si quisiéramos agregar texto a un archivo (***append***), el constructor sobrecargado usado debería ser:

```
s = new StreamWriter(fullName, true);
s.WriteLine("Appending text to file");
s.Close();
```

Si se pone como segundo parámetro *true* en el constructor, la acción es la de añadir texto al final (equivalente al *append* clásico). Si se pone a *false* o no se pone dicho parámetro, sobreescribe el archivo o lo crea si no existía.

Si en algún momento se desea volcar la información del buffer al archivo sin cerrar el *StreamWriter*, se puede hacer mediante el método **Flush().**

La clase StreamReader

Esta clase permite realizar lectura de datos de texto desde un archivo. Presenta al igual que en el caso anterior distintos métodos para realizar la lectura.

Al igual que en el caso anterior, se inicializa con el constructor pasándole como parámetro el nombre del archivo a leer.

En el caso de que un archivo no exista saltará una excepción (*FileNotFoundException*)

Métodos:
ReadLine(): Devuelve una cadena que contiene una línea completa.
ReadToEnd(): Devuelve el texto desde la posición actual hasta el final del archivo.

Ejemplos:
Lectura de un archivo línea a línea mediante un bucle:

```
string line;
StreamReader sr;
sr = new StreamReader(fullName);
line = sr.ReadLine();
while (line != null)
{
    Console.WriteLine(line);
    line = sr.ReadLine();
}
sr.Close();
```

Lectura de un archivo de texto y mostrarlo de golpe.

```
sr = new StreamReader(fullName);
Console.WriteLine(sr.ReadToEnd());
sr.Close();
```

Excepciones y cierre automático (using)

La lectura y escritura de datos en archivos evidentemente genera excepciones. La **excepción** más habitual en este caso es que el directorio/archivo donde se quiere trabajar no exista o no haya permisos en el mismo. Para ello se puede utilizar el clásico *try-catch-finally*.

Para una revisión sobre excepciones se puede ver el enlace:
https://msdn.microsoft.com/es-es/library/system.io.ioexception(v=vs.110).aspx

Si no se desea hacer nada en caso de que salte una excepción pero se quiere realizar un cierre automático se puede reducir la sentencia a un *try-finally*. Es decir, evitando el bloque catch y cerrando el archivo en el finally si es distinto de null. Sin embargo existe en c# una forma de escribir este bloque más cómoda mediante la sentencia using.

Tomando el último ejemplo, quedaría de la siguiente forma:

```
using (sr = new StreamReader(fullName))
{
    Console.WriteLine(sr.ReadToEnd());
}
```

Si se desea hacer el control de excepciones simplemente se mete en un sentencia try el bloque anterior. Por ejemplo así:

```
try
{
    using (StreamReader sr = new StreamReader("MyFile.txt"))
    {
        Console.WriteLine(sr.ReadToEnd());
    }
}
catch (FileNotFoundException e)
{
    Console.WriteLine($"The file was not found: '{e}'");
}
catch (DirectoryNotFoundException e)
{
    Console.WriteLine($"The directory was not found: '{e}'");
}
catch (IOException e)
{
    Console.WriteLine($"The file could not be opened: '{e}'");
}
```

Manejo e información de archivos y directorios

Archivos: Clases File y FileInfo

Ambas clases sirven para el manejo de archivos según una serie de métodos.
File no hace falta instanciarla, se pueden usar sus métodos directamente (Clase estática).

FileInfo sí es necesario instanciarla pero nos va a ofrecer una serie de métodos añadidos de información sobre el archivo.

File

Los métodos más comunes de esta clase son:

Exist(filename): Devuelve *true* si existe cierto fichero en el directorio de trabajo.

Copy(), Move, Delete(): Copia, mueve y borra archivos respectivamente. Ver MSDN.

Ejemplo:

```
StreamReader sr;
StreamWriter sw;
string fullName;

Console.WriteLine("Please, introduce file name with full path:");
fullName = Console.ReadLine();
if (File.Exists(fullName))
{
    using (sr = new StreamReader(fullName))
    {
        Console.WriteLine(sr.ReadToEnd());
    }
}

else
{
    using (sw = new StreamWriter(fullName))
    {
        sw.WriteLine("This is");
        sw.WriteLine("a new file");
    }
}
```

FileInfo

Esta clase nos va a permitir obtener una información adicional de cierto archivo. Hay que instanciarlo, no se puede manejar como clase al igual que *File*. La información principal que ofrecerá sobre un archivo es su Nombre, su Extensión y el directorio en el que se encuentra.

Ejemplo :

```
FileInfo f;

f = new FileInfo("C:\\windows\\win.ini");
Console.WriteLine("Name: {0}\n"
            + "Extension: {1}\n"
            + "Directory: {2}\n"
            + "Length: {3} bytes\n\n"
            + "And the full name: {4}",
            f.Name, f.Extension, f.DirectoryName, f.Length, f.FullName);
```

Directorios: Las clases Directory y DirectoryInfo

Nuevamente la clase *Directory* nos da una serie de métodos genéricos y no es necesaria instanciarla (*Shared* o estática), sin embargo de *DirectoryInfo* sí hace falta crear un objeto para poder usarlo.

Veamos cuales son las principales posibilidades que nos da cada una:

Directory

GetCurrentDirectory: función que devuelve el directorio actual de trabajo. Inicialmente es en el que se encuentra el ejecutable de la aplicación.

SetCurrentDirectory(nombre): Establece un nuevo directorio de trabajo. Se puede usar indicadores relativos:

```
Directory.SetCurrentDirectory("..\")
```

DirectoryInfo

Instanciamos el objeto para tener más información sobre cierto directorio.

GetDirectories: Devuelve un ***array*** de *DirectoryInfo* que contiene todos los subdirectorios del directorio actual. Opcionalmente se le puede pasar un parámetro para filtrar los elementos a presentar.

Por ejemplo *GetDirectories("s*")* presenta sólo los directorios que comiecen por S.
GetDirectories("DATOS??B") Presenta los directorios que empiezan por datos, tiene luego dos caracteres cualesquiera y acaban por B.

GetFiles: Devuelve un ***array*** de *FileInfo* que contiene todos los archivos de ese subdirectorio. Se puede usar parámetros igual que en el caso anterior:
GetFiles("*.bmp") obtiene todos los ficheros de extensión bmp

Tanto *GetFiles* como *GetDirectories* devuelve una matriz, por lo tanto para conocer la cantidad de directorios o ficheros se puede usar la propiedad *Lenght* de *array*.

Veamos el ejemplo siguiente para comprender mejor su funcionamiento.

Ejemplo:

```
DirectoryInfo d;
string systemRoot = Environment.GetEnvironmentVariable("SystemRoot");
Console.WriteLine($"{"Current directory:",-20}{Directory.GetCurrentDirectory()}");
Console.WriteLine($"{"Change to:",-20}{systemRoot}");

Directory.SetCurrentDirectory(systemRoot);

d = new DirectoryInfo(Directory.GetCurrentDirectory());

Console.WriteLine($"Subdirectories of {d.Name}");
foreach (DirectoryInfo dirs in d.GetDirectories())
{
    Console.WriteLine($"\t{dirs.Name}");
}

Console.WriteLine("Files:");
foreach (FileInfo f in d.GetFiles())
{
    Console.WriteLine($"\t{f.Name}");
}
```

El código es sencillo de entender. Se observa de nuevo el método para acceder a distintos directorios mediante variables de entorno.

***Ejercicios*:**

1. *Realizar un programa de manejo de directorios y archivos. Se planteará un formulario con las siguientes posibilidades:*
   - *Cambiar de directorio: Mediante un TextBox se indica el directorio de trabajo. Tendrá un botón asociado que al pulsarlo hará que se cambie a dicho directorio. Al pulsar Enter también se accederá al directorio. Además se permitirá escribir una variable de entorno. Si la palabra empieza y acaba por % entenderá que es una variable de entorno y si se trata de un directorio cambiará a él.*
   - *Ver subdirectorios del directorio actual en un ListBox. La primera entrada será .. que permitirá ir al directorio por encima del actual en la jerarquía. Si se selecciona otro directorio se accederá al mismo (actualiza también el textbox y el listbox de archivos).*
   - *Ver ficheros del directorio actual en otro ListBox distinto. Si se selecciona un archivo se mostrará en una label el tamaño (bytes, KB, MB,… según se considere). Si además es de extensión txt, se muestra en un formulario modal (en un textbox que ocupe todo y de título el nombre del archivo). Si al cerrar el modal el archivo ha sido modificado preguntará al usuario si desea guardarlo.*

   ***No** se pueden usar cuadros de diálogo estándar.*

2. *Realiza una aplicación a la cual se le pueda introducir un directorio en un textbox y una palabra en otro textbox. Al pulsar un botón de búsqueda el programa buscará todas las apariciones de la palabra en los archivos de texto indicando cuantas veces aparece la palabra en cada archivo en un tercer textbox (este multilínea). La búsqueda de cada archivo será en un hilo distinto. Mediante un checkbox se permitirá que se distinga o no entre mayúsculas.*

   Las extensiones consideradas como de archivos de texto estarán a su vez almacenadas en un archivo de texto cada una separada de la siguiente por una coma. Si no existe el archivo por defecto coge extensión TXT.

   Mediante una textbox más se permite modificar dichas extensiones separadas por comas. Al salir de la aplicación se guardará el archivo de extensiones y se cargará al iniciarla. Dicho archivo se encontrará en la carpeta del usuario actual (cada usuario la suya, claro).

Nota 1: Desde un hilo no se puede acceder a un componente que no haya sido creado en dicho hilo. Por ello debes crear una función que cumpla un delegado que haga la acción que deseas, por ejemplo añadir texto a un TextBox:

```
delegate void Delega(string texto, TextBox t);
private void cambiaTexto(string texto, TextBox t)
{
    t.AppendText(texto+Environment.NewLine);
}
```

luego dentro de la función de ejecución del hilo creas el delegado y lo llamas pero en lugar de hacerlo directamente lo haces a través del formulario con el método Invoke para que sea este el hilo que llama a su propio componente:

```
Delega d = new Delega(cambiaTexto);
this.Invoke(d,valor.ToString(),textBox1);
```

En Invoke el primer parámetro es la función a ejecutar y los siguientes parámetros son los propios de dicha función delegado.

Nota 2: Si se hace un Join de un hilo en el Closing o Closed de un formulario, hay que tener cuidado porque dicho Join para el hilo principal (del formulario) y si desde el hilo se está haciendo un Invoke del formulario no se va a ejecutar porque está bloqueado, por lo que entrará en deadlock. Una estructura en el closing puede ser:

acabar=true; //booleana que hace que los hilos acaben
while(hilos corriendo) //buscar la forma de comprobarlo
BUCLE hilo[i].Join(20); //se le mete tiempo para que si no sale se termine el Join y haga otras cosas.

De todas formas si se puede solucionar con IsBackground mucho mejor.

3. *Ampliar el programa gestor de Personas (Ejercicio 3 del Tema 3) de forma que guarde la información de la base de datos que se está creando. Para ello debe crearse nuevas clases lectoras y escritoras que hereden de BinaryWriter y BinaryReader (Ver Apéndice I) de forma que en la primera haya un write que sobrecargue los de la clase padre y permita grabar una estructura completa. La segunda debe tener nuevos métodos (p.ej. denominado ReadEmplead, ReadDirectivo) que permita obtener una estructura/objeto completo de un archivo del disco duro.*

   La colección de Personas debe cargarse al iniciar el programa y guardarse en el disco duro al finalizar el mismo de forma transparente al usuario (el usuario no participa). Esos son los únicos cambios en el programa, al inicio y fin del mismo, no cambies nada más.

   Si no hiciste el ejercicio completo hazlo al menos con la posibilidad de disponer de una colección donde se puedan añadir y eliminar Empleados y directivos para poder hacer el apartado de archivos.

4. ***(Opcional)****Ampliar el programa de comidas rápidas para que, de forma similar al ejercicio 3, guarde los datos en el disco duro al finalizar el programa y los lea al iniciar el mismo. En este caso se guardará toda la información en un archivo de texto con estructura XML o JSON. Busca qué clases se pueden usar en .Net para trabajar con este tipo de archivos.*

Apéndice I: Archivos Binarios en C#

La clase FileStream

Esta clase hereda de la clase abstracta *Stream* que como ya se comentó sirve para el manejo de secuencias de bytes. Por tanto las clases que hereden de *Stream* nos servirán para manejar los archivos a nivel de byte.

En concreto *FileStream* nos permite realizar acciones de escritura y lectura byte a byte o en grupos de bytes. Por supuesto esto puede requerir futuras conversiones a tipos de datos buscados.

Además permite el **acceso aleatorio** a archivos.

Constructor: Nos permite indicarle el nombre del archivo a manejar y la forma de abrirlo (lectura, escritura, añadir,…). Hay varios (sobrecarga) pero el que quizá más nos interese es el que admite dos parámetros: Nombre del archivo y modo de apertura. El segundo parámetro es un **enumerado *FileMode*** que puede tomar los valores:

Nombre	Descripción
Append	Abre el archivo si ya existe y se prepara para escribir al final del mismo, o crea un nuevo archivo si no existe. Cualquier intento de leer provoca un error e inicia una excepción ArgumentException.
CreateNew	Especifica que el sistema operativo debe crear un archivo nuevo. Si el archivo ya existe, se inicia una excepción IOException.
Open	Especifica que el sistema operativo debe abrir un archivo existente. Para lectura o escritura System.IO.FileNotFoundException se inicia si el archivo no existe.
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. Por tanto vacía un archivo ya existente. Si el archivo no existe salta la excepción System.IO.FileNotFoundException
Create	Especifica que el sistema operativo debe crear un archivo nuevo. Si el archivo ya existe, se sobrescribirá. **System.IO.FileMode.Create** es equivalente a solicitar que se utilice **CreateNew** si no existe el archivo; si el archivo existe, es equivalente **Truncate**.

Algunos métodos:
WriteByte(valor): Escribe un byte en el archivo.

Ejemplo:

```
string directory = Environment.GetEnvironmentVariable("homepath");
FileStream fs;
fs = new FileStream(directory + "\\testfile.bin", FileMode.Create);
fs.WriteByte(15);
fs.WriteByte(20);
fs.Close();

fs = new FileStream(directory + "\\testfile.bin", FileMode.Append);
fs.WriteByte(200);
fs.WriteByte(255);
```

```
fs.Close();
```

ReadByte(): Función que devuelve un byte (convertido a integer) del archivo indicado. El archivo debe estar abierto en alguna configuración de lectura. Devuelve -1 si se llega al final del archivo.

Ejemplos: Tres formas de leer todos los bytes de un archivo y mostrarlos en pantalla.

1.

```
string directory = Environment.GetEnvironmentVariable("homepath");
FileStream fs;
using (fs = new FileStream(directory + "\\testfile.bin", FileMode.Open))
{
    for (long i = 0; i < fs.Length; i++)
        Console.Write("{0} ", fs.ReadByte());
}
```

2.

```
using (fs = new FileStream(directory + "\\testfile.bin", FileMode.Open))
{
    int valor = fs.ReadByte();
    while (valor != -1)
    {
        Console.Write("{0} ", valor.ToString());
        valor = fs.ReadByte();
    }
}
```

3.

```
byte by;
using (fs = new FileStream(directory + "\\testfile.bin", FileMode.Open))
{
    try
    {
        while (true)
        {
            checked
            {
                by = (byte)fs.ReadByte();
            }
            Console.Write("{0} ", by);
        }
    }
    catch (OverflowException) { }
}
```

Seek(desplazamiento, origen): Sitúa el cursor de lectura del archivo en la posición deseada. Nos permite el acceso aleatorio al archivo.

Enumeración ***seekOrigin***:

Begin	Especifica el comienzo de una secuencia.
Current	Especifica lo posición actual dentro de la secuencia.
End	Especifica el final de una secuencia.

En el caso de usar como origen *Current* o *End*, se pueden usar valores negativos en el desplazamiento.

Las clases BinaryReader y BinaryWriter

Son clases similares a *FileStream* pero que permite el acceso al archivo de un tipo fijo de dato.

Por ejemplo *BinaryWriter* puede hacer *Write* de caracteres, enteros, reales etc…

Y *BinaryReader* dispone de los métodos *ReadChar*, *ReadInteger*, *ReadBoolean*, etc.

A diferencia de las anteriores clases vistas, estas no se pueden inicializar a partir del nombre de un archivo, si no que hay que hacerlo a través de un *FileStream*. Por ejemplo:

```
BinaryWriter bw;
bw= new BinaryWriter(new FileStream(directory + "\\testfile.bin",
FileMode.CreateNew));


BinaryReader br;
br = new BinaryReader(new FileStream(directory + "\\testfile.bin",
FileMode.Open));
```

Una diferencia importante entre el *BinaryWriter* y el *BinaryReader* es que el primero dispone directamente del método *seek* tal y como lo vimos en el *FileStream* mientras el segundo no. Pero esto no es problema ya que disponemos del seek del *FileStream* asociado.

```
fs = New FileStream(name, FileMode.Open)
br = New BinaryReader(fs);
…
fs.Seek(pos, SeekOrigin.Begin);
o
br.BaseStream.Seek(pos, SeekOrigin.Begin);

x = br.ReadInt32()
```

El problema que nos podemos encontrar con este método es cuando queremos hacer la lectura o escritura sobre algún tipo de dato que hemos creado nosotros como puede ser una estructura. Para estos casos tendremos que escribir una clase nueva y aplicar conceptos de POO para explotar dicha posibilidad.

Vemos un ejemplo. En un archivo aparte, dentro del espacio de nombres de la aplicación se construye el siguiente código:

```
    public struct Person
    {
        public string name;
        public int age;
    }

    class BinWriterPerson : BinaryWriter
    {
        public BinWriterPerson(Stream str) : base(str)
        {
        }

        public void Write(Person p) //Overload
        {
            base.Write(p.name);
            base.Write(p.age);
        }
    }

    class BinReaderPerson : BinaryReader
    {
        public BinReaderPerson(Stream str) : base(str)
        {
        }

        public Person ReadPerson()
        {
            Person p;
            p.name = base.ReadString();
            p.age = base.ReadInt32();
            return p;
        }
    }
```

*Posteriormente en el programa principal o en un método deseado de cualquier clase se pueden escribir las operaciones de lectura y escritura de datos. En la variable **directorio** lee la variable de entorno deseada.*

Escritura:

```
            Person person;
            BinWriterPerson bwp;
            string directory = Environment.GetEnvironmentVariable("homepath");

            person.name = "Mr. Potato";
            person.age = 33;

            bwp = new BinWriterPerson(new FileStream(directory + "\\people.dat",
FileMode.CreateNew));
            bwp.Write(person);
            bwp.Dispose(); //Dipose implies close
```

Lectura (y muestra de los datos):

```
            BinReaderPerson brp;
            brp = new BinReaderPerson(new FileStream(directory + "\\people.dat",
FileMode.Open));
            person=brp.ReadPerson();
            brp.Dispose();

            Console.WriteLine(person.name + " "+ person.age);
```