GOBIERNO DE ESPAÑA
MINISTERIO DE INDUSTRIA, COMERCIO Y TURISMO
Escuela de organización industrial

CURSO DE PROGRAMACIÓN.NET

M.374.001.003

ACCESO A FICHEROS

CONTROL DE ERRORES

Unión Europea
Fondo Social Europeo
Iniciativa de Empleo Juvenil
El FSE invierte en tu futuro

Ayuntamiento de Málaga

polo de contenidos digitales

Index

Acceso a ficheros

Antes de empezar a leer y escribir ficheros con C#, vamos a ver una serie de consejos útiles.

Para empezar, si queremos acceder a un archivo solo por su nombre,en función del IDE que utilicemos, tendremos que crearlo en una carpeta o en otra.

- ✔ Si utilizamos Visual Studio Code lo guardaremos en la misma carpeta donde se encuentra el archivo **program.cs.** Para acceder a ella directamente desde el VS Code, abriremos el menú contextual (botón derecho) del archivo **program.cs** (situado en la pestaña del explorador situado en la parte izquierda) y seleccionaremos “Mostrar en el explorador de archivos”.

- ✔ Si utilizamos Visual Studio 2019 debemos crearlo en la misma carpeta donde se genera el **fichero ejecutable (.exe)** del proyecto que se encuentra en **\bin\Debug\netcoreapp3.1**. Para acceder a la carpeta **bin** directamente desde el Visual Studio 2019 abriremos el menú contextual (botón derecho) del archivo **program.cs** (situado en el “Explorador de soluciones”) y seleccionaremos “Abrir carpeta contenedora”). Una vez allí accederemos a la ruta especificada **\bin\Debug\netcoreapp3.1**.

Si lo tenemos en otro sitio debemos indicar la ruta hasta el archivo:

- **Ruta relativa**: Ruta al directorio donde esta el archivo desde el directorio principal del proyecto. Por ejemplo, si tenemos el archivo dentro de un directorio llamado **files** en nuestro proyecto → **files\archivo.txt**. Si está en un directorio llamado prueba dentro de files → **files\prueba\archivo.txt**. Para ir al directorio anterior se utilizan 2 puntos seguidos → **..\archivo.txt**.

- **Ruta absoluta**: Ruta completa desde la raíz de la unidad donde está el archivo. Solo se recomienda este tipo de rutas para acceder a archivos que no están dentro de la carpeta del proyecto → **c:\Users\Pepito\Documents\archivo.txt**.

Para escribir una ruta, lo mejor es utilizar el carácter @ delante de la cadena. En C# esto implica que la cadena se va a interpretar de forma literal y no se tendrán en cuenta caracteres especiales como \n o \t. Esto implica que la barra de separación de directorios (en Windows) tampoco se tiene que ‘escapar’: “c:\\dir\\dir2” → @”c:\dir\dir2”.

Si estamos creando aplicaciones multiplataforma, ya que .NET Core lo permite, lo ideal es no utilizar un separador de directorios prefijado en la cadena. Aunque en Windows los directorios se separan con ‘\’, tanto en Linux como en Mac, se separan con la barra contraria ‘/’. Es por ello que podemos utilizar, o bien una función que nos devuelva la ruta correcta a partir de un listado de directorios en función del sistema operativo, o una constante que representa al separador (diferente según el sistema).

```
string ruta = System.IO.Path.Join("dir1", "dir2", "a.txt");
Console.WriteLine(ruta); // dir1\dir2\a.txt (Windows) - dir1/dir2/a.txt (Linux)

char sep = System.IO.Path.DirectorySeparatorChar;
string ruta = $"dir1{sep}dir2{sep}a.txt";
Console.WriteLine(ruta); // dir1\dir2\a.txt (Windows) - dir1/dir2/a.txt (Linux)
```

Lectura de ficheros

Para leer de un fichero línea a línea hasta el final, necesitamos saber la ruta al archivo y utilizar la clase StreamReader de C#. Lo que haremos será abrir el archivo en modo lectura e ir indicando que nos devuelva la siguiente línea usando el método **ReadLine()**. Este método nos devuelve un string con la siguiente línea del archivo o bien null cuando hemos llegado al final (es lo que debemos controlar).

Es importante cerrar el archivo cuando hayamos terminado con el método **Close**. Ya que si lo dejamos abierto, no podremos modificarlo, borrarlo, etc. mientras esté el programa abierto.

```
string line;
System.IO.StreamReader file = new System.IO.StreamReader("archivo.txt");
while ((line = file.ReadLine()) != null)
{
    System.Console.WriteLine(line);
}

file.Close();
```

Cuando el fichero no es muy grande (ya que el rendimiento podría verse afectado) se puede leer todo el contenido del archivo con una sola instrucción. El contenido se puede guardar entero en una cadena.

```
string texto = System.IO.File.ReadAllText("archivo.txt");
Console.WriteLine(texto);
```

O bien en un array de cadenas que podemos luego recorrer, transformar, etc.

```
string[] lines = System.IO.File.ReadAllLines("archivo.txt");
foreach (string line in lines)
{
    System.Console.WriteLine(line);
}
```

Escribir en ficheros

Escribir contenido en un fichero es muy similar a leerlo, y se puede hacer de varias formas. Si se quiere crear o sobrescribir un archivo existente cuyo contenido esté almacenado en una sola cadena, tenemos el método **WriteAllText**.

```
string texto = "Contenido del archivo\ncon varias líneas.";
System.IO.File.WriteAllText("prueba.txt", texto);

prueba.txt
Contenido del archivo
con varias líneas.
```

También existe el método **WriteAllLines**, si tenemos en un array de cadenas las líneas que queremos escribir.

```
string[] lineas = {
    "Línea 1",
    "Línea 2",
    "Línea 3"
};
System.IO.File.WriteAllLinesAsync("prueba.txt", lineas);
```

O podemos escribir en el fichero línea a línea:

```
string[] lineas = {
    "Línea 1",
    "Línea 2",
    "Línea 3"
};
System.IO.StreamWriter writer = new System.IO.StreamWriter("prueba.txt");
foreach(string line in lineas)
{
    writer.WriteLine(line);
}
writer.Close();
```

Si en lugar de sobrescribir todo el contenido del fichero queremos concatenar algo a lo ya existente, se puede enviar como segundo parámetro a **StreamWriter** un booleano con valor **true**.

```
System.IO.StreamWriter writer = new System.IO.StreamWriter("prueba.txt", true);
```

Cerrar automáticamente los recursos abiertos

Existen varios recursos, como los archivos, que deben cerrarse una vez se termina de trabajar con ellos para que otras aplicaciones del sistema (o la nuestra más adelante) puedan acceder más tarde.

Al trabajar con archivos pueden producirse errores que generan excepciones (de las cuales hablaremos más adelante). Si eso ocurre, la instrucción de cerrar el acceso al archivo podría no ejecutarse, o sería más complicado de controlar.

En C# existe la instrucción **using**, dentro de la cual inicializamos los recursos (StreamReader, StreamWriter, ...) que queramos que se cierren automáticamente una vez hayamos acabado. Esta instrucción creará un bloque que, una vez terminado, cerrará los recursos abiertos automáticamente (aunque se produzca un error).

```
string line;
using (System.IO.StreamReader file = new System.IO.StreamReader("archivo.txt"))
{
    while ((line = file.ReadLine()) != null)
    {
        System.Console.WriteLine(line);
    }
} // StreamReader se cierra al acabar este bloque
```

Control de excepciones

Existen instrucciones que pueden generar errores ante ciertas situaciones, por ejemplo una división por cero o intentar leer un archivo que no existe. Cuando se produce un error de este tipo, el lenguaje genera una excepción, que si no se procesa cerrará inmediatamente el programa mostrando el error por consola.

Veamos por ejemplo el siguiente código:

```
int a = 0;
int b = 2 / a;
Console.WriteLine("Instrucción después del error");
```

Al ejecutar este programa, se produce una división entre cero. Esto genera un error, y como no lo estamos controlando, se mostraría el siguiente mensaje por la consola (además de no ejecutar la última instrucción).

```
    Unhandled Exception: System.DivideByZeroException: Attempted to divide
by zero.
    at Mi_Proyecto.Program.Main(String[] args) in /home/arturo/Documentos/
Mi Proyecto/Program.cs:line 10
```

El error nos informa de que hay una excepción no controlada y el tipo de excepción, en este caso **DivideByZeroException**.

Para controlar los posibles errores debemos utilizar una estructura **try/catch**, donde la instrucciones que pueden causar errores se pondrán dentro del bloque **try**, y el tratamiento de errores se gestionará en el bloque **catch**. Opcionalmente, tenemos un bloque **finally** cuyas instrucciones se ejecutarán al final independientemente de si ha ocurrido un error o no (por ejemplo, para cerrar un lector o escritor de fichero).

```
try
{
    // Código que puede generar excepciones

}
catch(Exception e)
{
    // Tratamiento de la excepción

}
finally
{
    // Acciones al finalizar el bloque (con error o sin él)

}
```

El error anterior lo podíamos haber controlado así:

```
try
{
    int a = 0;
```

```
    int b = 2 / a;
}
catch (DivideByZeroException e)
{
    Console.WriteLine($"Se ha producido un error: {e.Message}");
}

Console.WriteLine("Siguiente instrucción");

Se ha producido un error: Attempted to divide by zero.
Siguiente instrucción
```

En el ejemplo anterior, si se hubiera generado una excepción de otro tipo diferente al de la división por cero, el bloque catch no hubiera controlado dicho error. Usando el tipo genérico **Exception**, capturaríamos cualquier tipo de error, aunque perderíamos información específica acerca del mismo.

```
try
{
    int a = 0;
    int b = 2 / a;
}
catch (Exception e)
{
    Console.WriteLine($"Se ha producido un error: {e.Message}");
}
```

Puede haber varios bloques catch correspondientes a un bloque try. En cada uno pondríamos un tipo diferente de excepción y dependiendo del error saltaría a uno u otro bloque. Si queremos usar un bloque que capture cualquier tipo de excepción usaríamos el tipo Exception, pero solo tendría sentido usarlo solo o ponerlo el último si queremos varios bloques (por ejemplo para capturar errores no esperados).

```
int i = 0;
int[] array = { 12, 23, 4, 6 };
try
{
    for (; i < 8; i++)
    {
        array[i] = array[i] / array[i + 1];
    }
}
catch (DivideByZeroException e)
{
    Console.WriteLine($"Error: División por cero en posición {i + 1}");
}
catch (IndexOutOfRangeException e)
{
    Console.WriteLine($"Error: Índice fuera del array {i + 1}");
}
```

Lanzar excepciones

No solo algunas instrucciones del lenguaje generan excepciones. También podemos generarlas manualmente si creemos que se ha producido un error grave

que debe ser controlado o el programa corre el riesgo de no funcionar correctamente.

```
class Program
{
    public static void Saludar(string nombre)
    {
       if (nombre == null)
       {
         throw new ArgumentNullException(nombre, "El nombre no puede ser nulo");
       }
    }

    static void Main(string[] args)
    {
        try
        {
            Saludar(null);
        }
        catch (Exception e)
        {
            Console.WriteLine($"ERROR: {e.Message}");
        }
    }
}
```

```
ERROR: El nombre no puede ser nulo
```

Depuración de código en Visual Studio Code

Muchas veces se producen errores en el código que aunque no generen excepciones, afectan a la correcta ejecución del programa. Esto produce comportamientos impredecibles que a veces son difíciles de detectar.

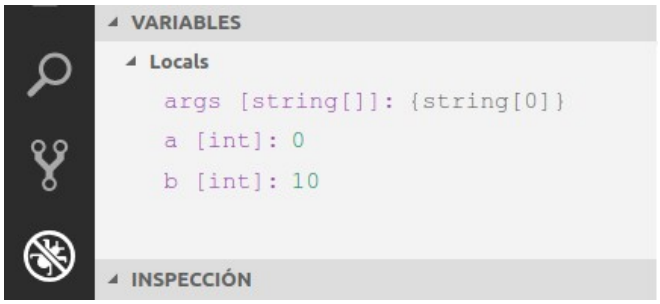
Para ayudar a detectar este tipo de errores podemos analizar el código mientras se ejecuta el programa. Esto se puede hacer desde el modo depuración de Visual Studio Code, añadiendo puntos de parada al programa.

Para añadir un punto de parada en una instrucción, haremos clic a la izquierda del número de línea, aparecerá un punto rojo y cuando se ejecute el programa en modo depuración (F5), se detendrá justo en ese sitio para que analicemos la situación.

```
 7        static void Main(string[] args)
 8        {
 9            int a = 0;
10            int b = 10;
11            for (; a < b; a++, b--)
12            {
13                System.Console.WriteLine($"a vale {a} y b vale {b}");
14            }
15        }
```

Punto de parada

Lo ideal también, es ejecutar el programa (usando F5) desde la pestaña de depuración de VSCode. Cuando se pare el código en el punto de parada podremos analizar las variables, llamadas a métodos, etc.



Cuando el programa esté parado, podemos continuar con la ejecución (hasta el siguiente punto de parada o final del programa), avanzar a la siguiente instrucción, salir de la depuración (y continuar con la ejecución normal), reiniciar el programa, o detener el programa.