File Processing: Streams

Source	http://www.codeproject.com/csharp/IntroReflection.asp

To support file processing, the .NET Framework provides the System.IO namespace that contains many different classes to handle almost any type of file operation you may need to perform. The parent class of file processing is Stream. With Stream, you can store data to a stream or you can retrieve data from a stream. Stream is an abstract class, which means that you cannot use it to declare a variable in your application. As an abstract class, Stream is used as the parent of the classes that actually implement the necessary operations. You will usually use a combination of classes to perform a typical stream operation. For example, some classes are used to create a stream object while some others are used to write data to the created stream.

The Name of the File

Before performing file processing, one of your early decisions will consist of specifying the type of operation you want the user to perform. For example, you may want to create a brand new file. You may want to open an existing file. Or you may want to perform a routine operation on a file. In all or most cases, whether you are creating a new file or manipulating an existing one, you must specify the name of the file. You can do this by declaring a string variable but most classes used to create a stream as we will learn later can take a string that represents the name of the file.

If you are creating a new file, there are certainly some rules you must observe. The name of a file follows the directives of the operating system. On MS DOS and Windows 3.X (that is, prior to Microsoft Windows 9X), the file had to use the 8.3 format. The actual name had to have a maximum of 8 characters with restrictions on the characters that could be used. The user also had to specify three characters after a period. The three characters, known as the file extension, were used by the operating system to classify the file. That was all necessary for those 8-bit and 16-bit operating systems. Various rules have changed. For example, the names of folders and files on Microsoft Windows >= 95 can have up to 255 characters. The extension of the file is mostly left to the judgment of the programmer but the files are still using extensions. Applications can also be configured to save different types of files; that is, files with different extensions.

At the time of this writing, the rules for file names for Microsoft Windows were on the MSDN web site at Windows Development\Windows Base Services\Files and I/O\SDK Documentation\Storage\Storage Overview\File Management\Creating, Deleting, and Maintaining Files\Naming a File (because it is a web site and not a book, its pages can change anytime).

Based on this, if you declare a string variable to hold the name of the file, you can simply initialize the variable with the necessary name and its extension. Here is an example:

```
using System;

class Exercise
{
```

```
        static int Main()
        {
                string NameOfFile = "Employees.spr";

                return 0;
        }
}
```

The Path of the File

If you declare a string as above, the file will be created in the folder as the application. Otherwise, you can create your new file anywhere in the hard drive. To do that, you must provide a complete path where the file will reside. A path is a string that specifies the drive (such as A:, C:, or D:). The sections of a complete path string are separated by a backslash. For example, a path can the made of a folder followed by the name of the file. An example would be

```
C:\Palermo.tde
```

A path can also consist of a drive followed by the name of the folder in which the file will be created. Here is an example:

```
C:\Program Files\Palermo.tde
```

A path can also indicate that the file will be created in a folder that itself is inside of another folder. In this case, remember that the names of folder must be separated by backslashes.

In Lesson 1, we saw that the backslash character is used to create or manage escape sequences and it can be included in a string value to make up an escape sequence. Because of this, every time you include a backslash in a string, the compiler thinks that you are trying to provide an escape sequence. In this case, if the combination of the backslash and the character that follows the backslash is not recognized as an escape sequence, you would get an error. To solve this problem, you have two alternatives. To indicate that the backslash must be considered as a character in its own right, you can double it. Here are examples:

```
using System;

class Exercise
{
        static int Main()
        {
                string NameOfFile = "C:\\Documents and Settings\\Business
Records\\Employees.spr";

                return 0;
        }
}
```

Alternative, you can keep one backslash in each placeholder but precede the value of the string with the @ symbol. Here is an example:

```
using System;

class Exercise
{
        static int Main()
```

```
        {
                string NameOfFile = @"C:\Documents and Settings\Business
Records\Employees.spr";

                return 0;
        }
}
```

In the same way, you can declare a string variable to represent the name of an existing file that you plan to use in your program. You can also represent its path.

When providing a path to the file, if the drive you specify doesn't exist or cannot be read, the compiler would consider that the file doesn't exist. If you provide folders that don't exist in the drive, the compiler would consider that the file doesn't exist. This also means that the compiler will not create the folder(s) (the .NET Framework provides all means to create a folder but you must ask the compiler to create it; simply specifying a folder that doesn't exist will not automatically create it, even if you are creating a new file). Therefore, it is your responsibility to make sure that either the file or the path to the file is valid. As we will see in the next section, the compiler can check the existence of a file or path.

File Existence

While Stream is used as the parent of all file processing classes, the .NET Framework provides the File class equipped with methods to create, save, open, copy, move, delete, or provide detailed information about, files. Based on its functionality, the File class is typically used to assist the other classes with their processing operations. To effectively provide this support, all File's methods are static; which means that you will usually not need to declare a File variable to access them.

One of the valuable operations that the File class can perform is to check the existence of the file you want to use. For example, if you are creating a new file, you may want to make sure it doesn't exist already because if you try to create a file that exists already, the compiler may first delete the old file before creating the new one. This could lead to unpredictable result, especially because such a file is not sent to the Recycle Bin. On the other hand, if you are trying to open a file, you should first make sure the file exists, otherwise the compiler will not be able to open a file it cannot find.

To check the existence of a file, the File class provides the Exists method. Its syntax is:

```
public static bool Exists(string path);
```

If you provide only the name of the file, the compiler would check it in the folder of the application. If you provide the path to the file, the compiler would check its drive, its folder(s) and the file itself. In both cases, if the file exists, the method returns true. If the compiler cannot find the file, the method returns false. It's important to know that if you provided a complete path to the file, any slight mistake would produce a false result.

Access to a File

In order to perform an operation on a file, you must specify to the operating system how to proceed. One of the options you have is to indicate the type of access that will be granted on the file. This access is specified using the FileAccess enumerator. The members of the FileAccess enumerator are:

- **FileAccess.Write: New data can be written to the file**

- **FileAccess.Read: Existing data can be read from the file**
- **FileAccess.ReadWrite: Existing data can be read from the file and new data be written to the file**

File Sharing

In standalone workstations, one person is usually able to access and open a file then perform the necessary operations on it. In networked computers, you may create a file that different people can access at the same time or you may make one file access another file to retrieve information. For example, suppose you create an application for a fast food restaurant that has two or more connected workstations and all workstations save their customers orders to a common file. In this case, you must make sure that any of the computers can access the file to save an order. An employee from one of these workstations must also be able to open the file to retrieve a customer order for any necessary reason. You can also create a situation where one file holds an inventory of the items of a store and another file holds the customers orders. Obviously one file would depend on another. Based on this, when an operation must be performed on a file, you may have to specify how a file can be shared. This is done through the FileShare enumerator.

The values of the FileShare enumerator are:

- **FileShare.Inheritable: Allows other file handles to inherit from this file**
- **FileShare.None: The file cannot be shared**
- **FileShare.Read: The file can be opened and read from**
- **FileShare.Write: The file can be opened and written to**
- **FileShare.ReadWrite: The file can be opened to write to it or read from it**

The Mode of a File

Besides the access to the file, another option you will most likely specify to the operating system is referred to as the mode of a file. It is specified through the FileMode enumerator. The members of the FileMode Enumerator are:

- **FileMode.Append: If the file already exists, the new data will be added to its end. If the file doesn't exist, it will be created and the new data will be added to it**
- **FileMode.Create: If the file already exists, it will be deleted and a new file with the same name will be created. If the file doesn't exist, then it will be created**
- **FileMode.CreateNew: If the new already exists, the compiler will throw an error. If the file doesn't exist, it will be created**
- **FileMode.Open: If the file exists, it will be opened. If the file doesn't exist, an error would be thrown**
- **FileMode.OpenOrCreate: If the file already exists, it will be opened. If the file doesn't exist, it will be created**
- **FileMode.Truncate: If the file already exists, its contents will be deleted completely but the file will be kept, allowing you to write new data to it. If the file doesn't exist, an error would be thrown**

~~~ End of Article ~~~