# *File and Directory Access*

## Introduction to Files and File Access

File and directory access will covers the processes of creating and deleting files and directories, opening and reading files, and reading and writing to files. By default, file access operations assume the file is in the same directory as the application. To access a file that is in another directory, the absolute or relative address of the file needs to be provided. An example of an absolute addresses would be:

D:\CLASS\CSApplicationDevelopment\CSNotes\07-FileAccess

Since C# string formatting recognizes "\" as preceding an escape sequence command, we need to modify this the address by using the escape sequence for the backslash which is "\\". Therefore, when providing an address we need to type it as

D:\\CLASS\\CSApplicationDevelopment\\CSNotes\\07-FileAccess

Another way of accomplishing the same result is by using "@" which makes the string recognized as a literal. In this case we would type the address as

@"D:\CLASS\CSApplicationDevelopment\CSNotes\07-FileAccess"

This is particularly useful since one can cut and paste directory paths without needing to change all the "\" to "\\".

The directory and file access methods are provided under the namespace:

`System.IO`

To get started create a new project with Visual Studio and drag and drop a button onto Form1. View the code and at the top add the namespace System.IO by incerting "using System.IO;" into the using block. It should now look like

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.IO;
```

## Directory manipulation

Creating and deleting directories are done though the "Directory" class in System.IO. The methods of the Directory class that we will be looking at are all static and can therefore

be used directly without instantiating an instance of the class. To create a directory named "myDirectory" in the current directory you can add the following command to the Click method of your button:

```
Directory.CreateDirectory("myDirectory");
```

If you run your application and press the button, you should see the directory created in the directory that the application is. You can create a directory at any location on the hard drive (given you have access to do so) through providing the full address.

The current directory from which the application was launched can be obtained through "`Application.StartupPath`". This can be used to develop paths relative to the current startup directory. For example, the following command lines first get the application startup directory and then create a directory in the directory two levels up in the directory tree:

```
string AppPath = Application.StartupPath;
Directory.CreateDirectory(AppPath+@"\..\..\myDirectory");
```

The Directory class has several useful methods among which are:

| Method | Return value | Function |
|---|---|---|
| `CreateDirectory(string path)` | void | Creates the directory and all other needed directories in the path indicated |
| `Delete(string path)` | void | Deletes specified directory |
| `Exists(string path)` | bool | Returns true if directory exists |
| `GetCurrentDirectory()` | string | Gets current directory |
| `GetDirectoryRoot(string path)` | string | Gets root of path |
| `GetFiles(string path)` | string[] | Gets array of strings containing files in directory indicated by path |
| `Move(string sourceDirName, string destDirName)` | void | Moves directory at path address `sourceDirName` to address `destDirName` |
| `SetCurrentDirectory(string path)` | void | Sets current directory to the one designated by the path |

## File manipulation

File manipulation refers to creating, deleting and moving files. Reading and writing to a file will be shown in the next section. File manipulations are done through static methods of the class "File" in System.IO. For example, to create a file with the name "myFile.txt"

in the currently nonexistent directory “myDirectory” located in the current directory we can use the command lines:

```
Directory.CreateDirectory("myDirectory");
File.Create(@"myDirectory\myFile.txt");
```

If the directory exists, there is no need to create it, even though CreateDirectory will do nothing if the directory exists. Other useful methods of the File class are:

| Method | Return value | Function |
|---|---|---|
| `Create(string path)` | void | Create the file with given address path (The directory needs to exist) |
| `File.Copy(string sourceFileName, string destFileName)` | void | Copy file `sourceFileName` to `destFileName` |
| `Delete(string path)` | void | Delete indicated file |
| `Exists(string path)` | bool | Returns true if indicated file exists |
| `Move(string sourceFileName, string destFileName)` | void | Moves file `sourceFileName` to `destFileName` |

## Writing to a file

File access is done through string or character interfaces. We will focus on string methods. Writing to files is done through the “StreamWriter” class. To create the file and create and connect a StreamWriter to a file “myDirectory\myFile.txt” we can use the following command lines that first create the directory, then create and connect the StreamWriter “sw” to the file `"myDirectory\myFile.txt")`, then writes `"This is a test of writing to a file."` as a line into this files, and finally closes the file :

```
Directory.CreateDirectory("myDirectory");
StreamWriter sw = File.CreateText(@"myDirectory\myFile.txt");
sw.WriteLine("This is a test of writing to a file.");
sw.Close();
```

In this example one line was written. Repeating the WriteLine method would add lines to the end of the file. An entire file can be written using the Write method of StreamWriter. For example, the text containing ten lines will be written using the following code:

```
StreamWriter sw = File.CreateText(@"myDirectory\myFile.txt");
string textOfFile = "";
for(int i=0; i<10;i++)
{
      textOfFile +="This is line "+i.ToString()+".\n";
}
sw.Write(textOfFile);
sw.Close();
```

This same effect can be obtained using the WriteLine method:

```
StreamWriter sw = File.CreateText(@"myDirectory\myFile.txt");
for(int i=0; i<10;i++)
{
      sw.WriteLine( "This is line "+i.ToString()+".");
}
sw.Close();
```

To append to the end of an existing file without overwriting it, the StreamWriter can be instantiated with the AppendText method:

```
StreamWriter sw = File.AppendText(@"myDirectory\myFile.txt);
```

Such a StreamWriter will write to the end of the existing file.

## Reading from a file

Reading of files is done using the StreamReader class. In the following code a StreamReader “sr” is declared and instantiated using File.OpenText method, then the content of the file is read line by line using the StreamReader.ReadLine method, and finally the instance of the StreamReader is closed to release the file.

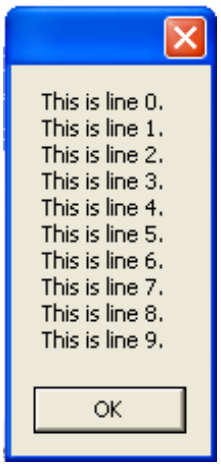
```
StreamReader sr = File.OpenText(@"myDirectory\myFile.txt");
for(int i=0; i<10;i++)
{
      string textOfLine = sr.ReadLine();
      MessageBox.Show(textOfLine);
}
sr.Close();
```

This code would read one line at a time and display it in the MessageBox.

The entire content of the file can be read into one string through the ReadToEnd method of the StreamReader. The following code shows an example of this.

```
StreamReader sr = File.OpenText(@"myDirectory\myFile.txt");
string textOfFile = sr.ReadToEnd();
sr.Close();
MessageBox.Show(textOfFile);
```

The output would be:




The methods Read and ReadBlock of SteamReader can be used to read arrays of characters.

To read an entire file without knowing how many lines are in the file, one can use the following code:

```
StreamReader sr = File.OpenText(@"myDirectory\myFile.txt");
string textOfFile = "";
sr.BaseStream.Seek(0, SeekOrigin.Begin);
while (sr.Peek() > -1)
{
      textOfFile += sr.ReadLine()+"\n";
}
sr.Close();
MessageBox.Show(textOfFile);
```

In this code, the BaseStream.Seek method is used to find the start of the file and the Peek method is used to find the end of the file, and the ReadLine method is used to read each line until it gets to the end of the file.

## Writing and Reading comma formatted data

Structured data that is comma formatted can be written to and read from a file. The two dimensional array “data” is written to a file with comma seperators by the following code:

```
double[,] data = {{1,2,3,4},{2,3,4,5},{3,4,5,6}};
StreamWriter sw = File.CreateText(@"myDirectory\myFile.txt");
for(int i=0; i<3; i++)
{
      string line ="";
      for(int j=0; j<4; j++)
      {
            line += data[i,j].ToString()+", ";
```

```
        }
        sw.WriteLine(line);
}
sw.Close();
```

The following code will read the data in the file back to an array:

```
StreamReader sr = File.OpenText(@"myDirectory\myFile.txt");
sr.BaseStream.Seek(0, SeekOrigin.Begin);
int i=0;
while (sr.Peek() > -1)
{
        string line = sr.ReadLine();
        for(int j=0; j<4; j++)
        {
                int flag = line.IndexOf(",");
                data[i,j] = Convert.ToDouble(line.Substring(0,flag));
                MessageBox.Show("data = "+data[i,j].ToString());
                line = line.Remove(0,flag+1);
        }
        i++;
}
```

This can be modified to read different data types such as float or int.