Problem 1. DIY Judge System

Idea overview

Our first task is to **implement** a simple "**judge**" system which we will later **use to test** our **solutions**. Why not use the good old judge? Well he's taken the week off and we still need a way to test our code. The idea is simple – **create a program which will read a text file** (your output for a given problem) and **compare** its **contents** to the contents of another text file (expected output for that problem), **if** the contents are **identical** then the files are identical and your **output is correct** and everything's smooth. **If** the files **differ** in any way then an extra file called "**Mismatches.txt**" is **created** which **holds detailed information about the lines that do not match**. Let's start off.

Set up our Tester Class

Create a new Visual Project Solution and a new Console Application called "SimpleJudge". In the SimpleJudge project **add** a **new class** called "**Tester**". Mark it as **public static** class and **declare** a **new public static void** method called **CompareContent(string userOutputPath, string expectedOutputPath)**:

```
public static class Tester
{
    public static void CompareContent(string userOutputPath, string expectedOutputPath)
    {

    }
```

The idea here is that using **userOutputPath** and **expectedOutputPath** we can find the **files holding** the **user output** and **expected output respectively**, **read** the **user output** and **the expected output** and **compare** them **line by line** to see if they are identical.

As we mentioned above, however, we will also need a path to **create** the **Mismatches.txt** text file which will **hold** the **mismatches** (if any). In order to do that efficiently we can **use** the **expectedOutputPath** and simply **create** the **Mismatches.txt in the same folder**. How can we go about this? First we need to **extract the path** to the directory **of** the **expected output file**. We achieve this by creating a helper method:

```
private static string GetMismatchPath(string expectedOutputPath)
{
    int indexOf = expectedOutputPath.LastIndexOf('\\');
    string directoryPath = expectedOutputPath.Substring(0, indexOf);
    string finalPath = directoryPath + @"\Mismatches.txt";
    return finalPath;
}
```

What this method does is simply **get the path to** the directory of the **expected output file by finding the index of the last '\'** in the path of the expected output file. For example if the path is *C:\OutputFiles\OddLinesExpectedOutput.txt* we **find** the **index** of the second **'\'** (14 in our case) then we simply **get a substring of that path up until that index** and we end up with *C:\OutputFiles* which is the path to the directory of the output file. Then we finally **append** the **name of** our **file and** a **slash *"\Mismatches.txt"*** and we finally end up with a path looking something like this "*C:\OutputFiles\Mismatches.txt*". You might wonder how come we use a path to a file that does not currently exist. We'll get to that in a moment, but first let's call out helper method up in our main method.

```
public static void CompareContent(string userOutputPath, string expectedOutputPath)
{
    OutputWriter.WriteMessageOnNewLine("Reading files...");

    string mismatchPath = GetMismatchPath(expectedOutputPath);

}
```

Read from and create files

Next up we need to **read** the **two files** – the user output and the expected output. This is done again in just one line of code. We call the **File** class and invoke the **.ReadAllLines(string path)** method. However this time around we need a variable in which we can actually store the contents of the files we read from. The **File.ReadAllLines(string path)** function **returns a string array** so our code will look something like this:

```
public static void CompareContent(string userOutputPath, string expectedOutputPath)
{
    OutputWriter.WriteMessageOnNewLine("Reading files...");

    string mismatchPath = GetMismatchPath(expectedOutputPath);

    string[] actualOutputLines = File.ReadAllLines(userOutputPath);
    string[] expectedOutputLines = File.ReadAllLines(expectedOutputPath);
```

We end up with a variable input which holds all the **user output**, read from the user output text file line by line, and a variable called **expectedOutput** which holds the… expected output, again read from the expected output text file line by line. We are ready to start the **comparison of the two files**. The information we will need while comparing the files is whether there are any mismatches and also the result of the comparison of two corresponding lines. So we can **make** one **Boolean** for the **mismatches and one string array** called **mismatches** which **gets** it's **value from** the **method GetLineWithPossibleMismatches** with it's three parameters shown in the picture below:

```
bool hasMismatch;
string[] mismatches = GetLinesWithPossibleMismatches(actualOutputLines, expectedOutputLines, out hasMismatch);
```

We'll get to the implementation of this method in a moment. First we need to finish the **CompareContent** method so that we can focus our attention on the other functionality waiting to be written.
The last thing we can do **after all** the **checks for mismatches** is to **write** them **on** the **set output writer and in** the **mismatches.txt file** which is in the same folder as the first file given for comparison and that is done by the **PrintOutput** method. And finally print on the output writer that the files are read:

```
PrintOutput(mismatches, hasMismatch, mismatchPath);
OutputWriter.WriteMessageOnNewLine("Files read!");
```

As you can see the **method for printing** the output of the comparison **takes 3 parameters**, which are **related to** the

possible mismatches We will discuss the implementation of this method after the previous one, so it is last on the queue now.

Finally the **CompareContent** method should look like something pretty similar to this:

```
public static void CompareContent(string userOutputPath, string expectedOutputPath)
{
    OutputWriter.WriteMessageOnNewLine("Reading files...");

    string mismatchPath = GetMismatchPath(expectedOutputPath);

    string[] actualOutputLines = File.ReadAllLines(userOutputPath);
    string[] expectedOutputLines = File.ReadAllLines(expectedOutputPath);

    bool hasMismatch;
    string[] mismatches = GetLinesWithPossibleMismatches(actualOutputLines, expectedOutputLines, out hasMismatch);

    PrintOutput(mismatches, hasMismatch, mismatchPath);
    OutputWriter.WriteMessageOnNewLine("Files read!");
}
```

Find mismatches of two files line by line

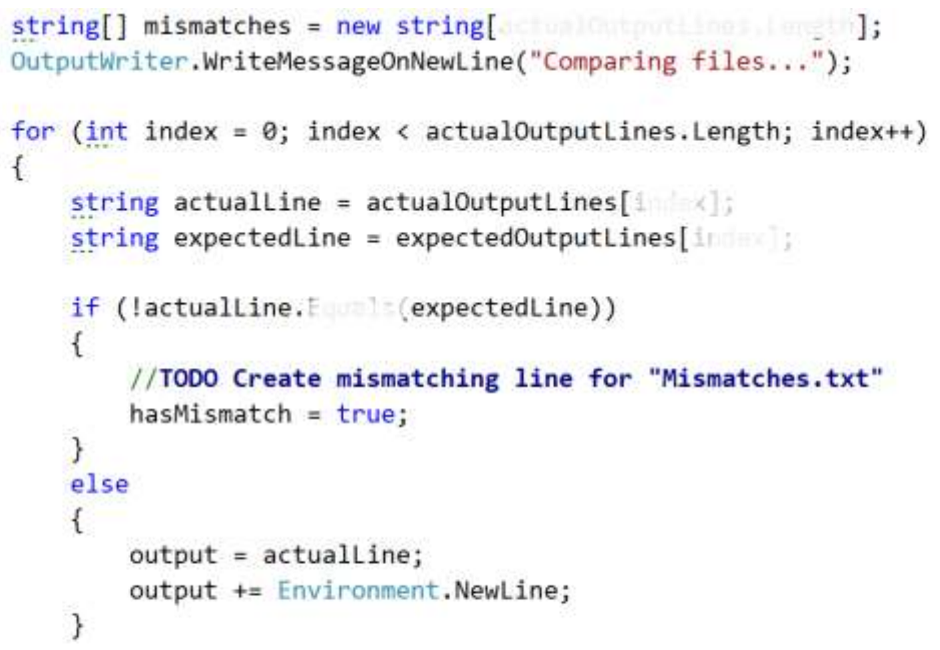
Since we are going to **compare two files**, and that is a separate task, we will use a separate **method** to do so. It's **called `GetLinesWithPossibleMismatches`** and **takes three parameters** which are the **strings array from** the **first file**, the **string array from** the **second file and** an **out parameter for** whether there are any **mismatches**, **so** that the following **method can change a variable outside of it's scope**. The method **returns** a **new string array** which **represents** the **result after** the **comparison** of each line.

```
private static string[] GetLinesWithPossibleMismatches(
    string[] actualOutputString, string[] expectedOutputString, out bool hasMismatch)
```

Before we start the actual comparison and matching it'd be a good idea to **declare one helper variable** which will come into play a bit later. A **string** that has an **initial value** of an **empty string** and is later **used for** the line by line **comparison of** the **two output files** that are given for comparison. Another thing we might want to **set** is the **hasMismatch variable** to **false** and **only if** on some place **two lines** are found **with** a **difference** between them, the **hasMismatch** variable is **set to true and** the one that is **outside** of the method **will also be set to true**.

```
hasMismatch = false;
string output = string.Empty;
```

Now that we have that sorted out we can safely get to the actual comparison. How do we go about that? Well we will need a few things in order to do effective comparison and write down our mismatches. In order to **compare** the **lines** we can **simply run** a **single for loop** which iterates **through** both **user generated** output **and** the **expected output comparing** each **line at every iteration** and writes the result of each comparison in a new string array called mismatches which we create in after creating the two variable above.



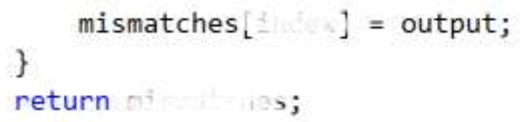
```
string[] mismatches = new string[actualOutputLines.Length];
OutputWriter.WriteMessageOnNewLine("Comparing files...");

for (int index = 0; index < actualOutputLines.Length; index++)
{
    string actualLine = actualOutputLines[index];
    string expectedLine = expectedOutputLines[index];

    if (!actualLine.Equals(expectedLine))
    {
        //TODO Create mismatching line for "Mismatches.txt"
        hasMismatch = true;
    }
    else
    {
        output = actualLine;
        output += Environment.NewLine;
    }
```

What's going on here is pretty straightforward. We simply **iterate over all** the **lines** from both the files by **assigning** the **current line to** the **actual line** variable **and** the **expected line to** the **expectedItem and comparing them**. **If** they are **not matching** we **mark mismatch** as **true**, and we will set the output to the following message:

```
string.Format("Mismatch at line {0} -- expected: \"{1}\", actual: \"{2}\"", index, expectedLine, actualLine)
```

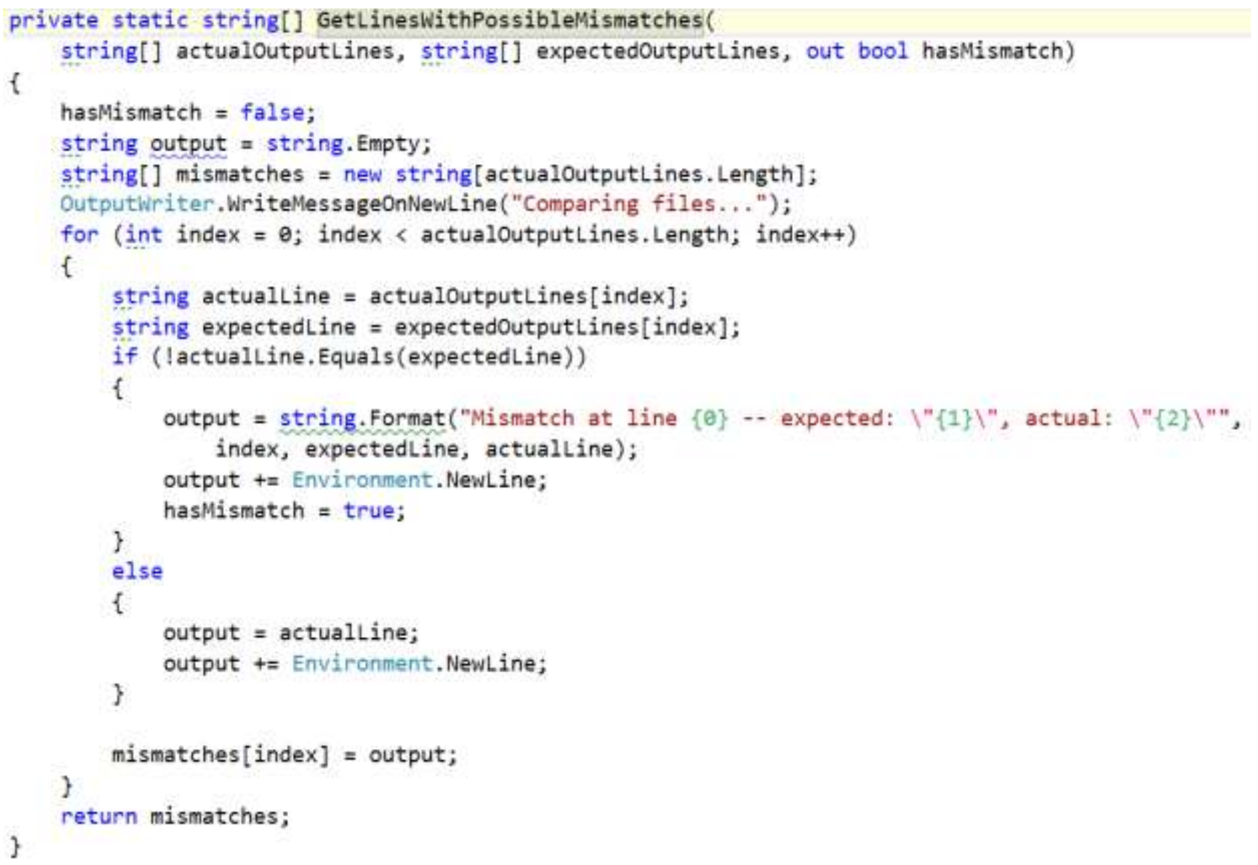
And after that append a new line like shown in the else clause in the code above

. **If** however we **don't get** a **mismatch**, **if** the **lines** are **identical** then we simply **write down** the **line** in **Mismatches.txt**. Why? Well because **if** we get an eventual **mismatch** down the line or if we've gotten one already, it'll be **easier to see where it occurred if** we also **have** the **rest of** the **text** written down. Finally, on **each iteration** you **put** the **output in** the **corresponding cell in** the **mismatches array and after** the **for loop** we should **return** the **mismatches array** and now we are sure to have the mismatches and also the **hasMismatch variable to be changed to** the **corresponding value**, because it's an out parameter.



```
        mismatches[index] = output;
    }
    return mismatches;
```

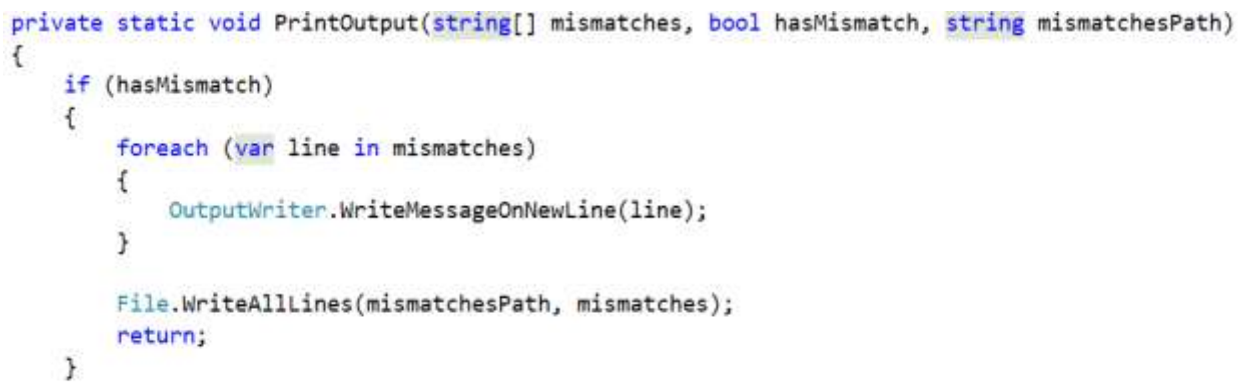
Here is a final version of the `GetLinesWithPossibleMismatches method:`



```
private static string[] GetLinesWithPossibleMismatches(
    string[] actualOutputLines, string[] expectedOutputLines, out bool hasMismatch)
{
    hasMismatch = false;
    string output = string.Empty;
    string[] mismatches = new string[actualOutputLines.Length];
    OutputWriter.WriteMessageOnNewLine("Comparing files...");
    for (int index = 0; index < actualOutputLines.Length; index++)
    {
        string actualLine = actualOutputLines[index];
        string expectedLine = expectedOutputLines[index];
        if (!actualLine.Equals(expectedLine))
        {
            output = string.Format("Mismatch at line {0} -- expected: \"{1}\", actual: \"{2}\"",
                index, expectedLine, actualLine);
            output += Environment.NewLine;
            hasMismatch = true;
        }
        else
        {
            output = actualLine;
            output += Environment.NewLine;
        }

        mismatches[index] = output;
    }
    return mismatches;
}
```

Printing the data from the comparison to the output writer and to a mismatch file created.

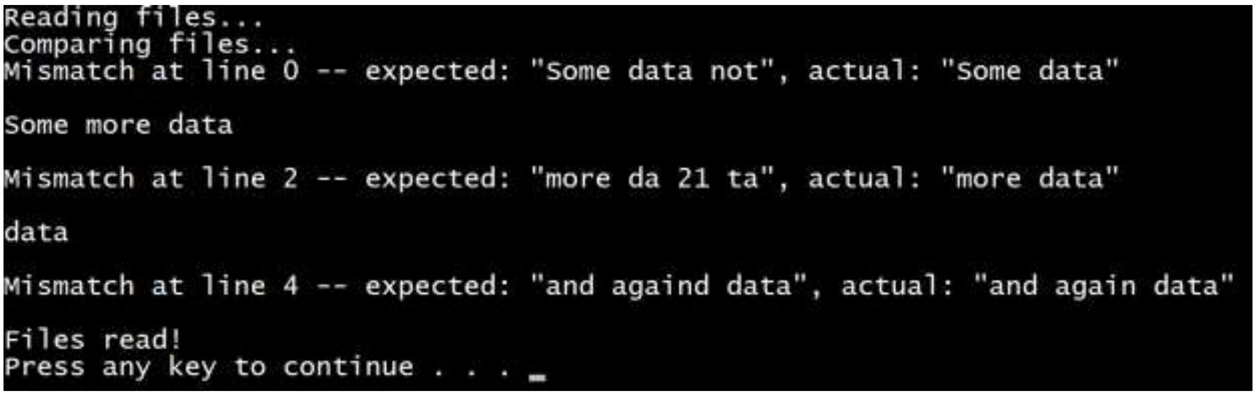
We've gotten to the place where we need to **implement** the **PrintOutput method**. It **has 3 parameters in** it's **declaration**. The **first one** is the **array** that we just generated **with** the **mismatches** from the previous method. The **second parameter** is whether there are **any mismatches and** the **third one** is the **path to** the **mismatches file**. All we have to do is **write all the lines from** the **mismatches on** the **output writer if** there **has** a **mismatch**, **append** all the **lines to** the **mismatch file** using the given path **and return** so that we exit the method. **If** the **hasMismatch** is **not true**, we do not enter in the body of the if and all we do is **write** a **message on** a **new line** which is the following: `“Files are identical. There are no mismatches.”`
Here is a how the implementation of what we just described above, should look:



```
private static void PrintOutput(string[] mismatches, bool hasMismatch, string mismatchesPath)
{
    if (hasMismatch)
    {
        foreach (var line in mismatches)
        {
            OutputWriter.WriteMessageOnNewLine(line);
        }

        File.WriteAllLines(mismatchesPath, mismatches);
        return;
    }
```

Now we should be ready for testing. You are given three files with the current story piece called **test1.txt, test2.txt, test3.txt**. **First compare** the **content of test1.txt**, **test2.txt**, see what log is written in the mismatches file (mismatch file should not be existing, because there are no mismatches) and **then** compare **test2.txt and test3.txt** and again see the mismatches file to see what has changed.

```
static void Main()
{
    Tester.CompareContent(@"E:\Work\Labs\test2.txt", @"E:\Work\Labs\test3.txt");
}
```



```
Reading files...
Comparing files...
Mismatch at line 0 -- expected: "Some data not", actual: "Some data"

Some more data

Mismatch at line 2 -- expected: "more da 21 ta", actual: "more data"

data

Mismatch at line 4 -- expected: "and againd data", actual: "and again data"

Files read!
Press any key to continue . . . _
```

Problem 2. Saving some data for our current session

The story doesn't end here. We have to make some modifications to some existing classes and also add some new.

The first new class we are going to write will hold the data for the current session. For now our only purpose is to have a place where we can save out current location and then move using only relative paths.

So we make our public static class called **SessionData** and our only variable in it will be the **currentPath**, which starts with a value of, the application's directory in the file system.

```
public static class SessionData
{
    public static string currentPath = Directory.GetCurrentDirectory();

}
```

This variable can be very useful in the **IOManager**, because we can use it for different operations like **traversing the current folder**, **creating files** in the current folder, **moving up and down in the folder tree** and also as a starting point in order to navigate to the "resources" folder and **read the Database** from a **file** and not from the **console**…

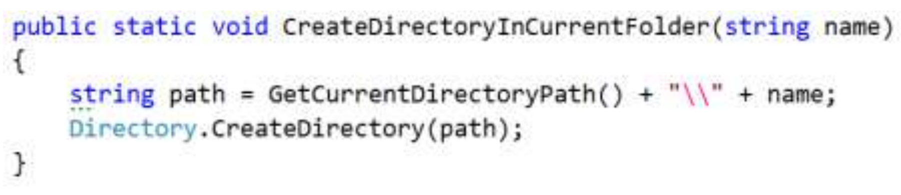
We are going to go through each of these steps in big details so you would be able to understand how each component works.

So enough chit chat, let's start extending the current classes we have.

Problem 3. Making directories

First we are going to stop in the **IOManager** and make a method for making a directory. Since we have our **currentPath** in the **SessionData** class al we need is the name of the folder we are going to create.

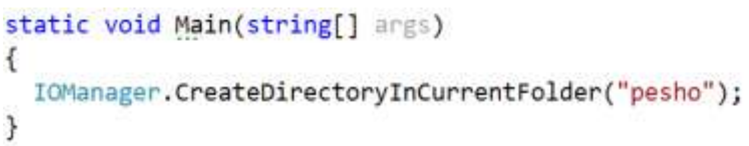
Our method can be called **CreateDirectoryInCurrentFolder** (string <the name of the folder>) and it's implementation should look like this.



```
public static void CreateDirectoryInCurrentFolder(string name)
{
    string path = GetCurrentDirectoryPath() + "\\" + name;
    Directory.CreateDirectory(path);
}
```

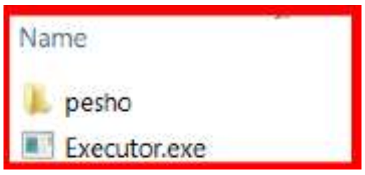
We use the method given fro the directory class, which takes an absolute an creates.

So now if we call it from the main method like this



```
static void Main(string[] args)
{
  IOManager.CreateDirectoryInCurrentFolder("pesho");
}
```

And since the folder that our application is currently running in the bin\Debug debug folder of the application, there a folder with a name "pesho" should be added.

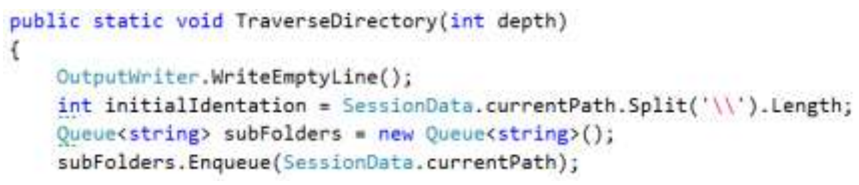


```
Name

pesho
Executor.exe
```

Problem 4. Modifying the traversal

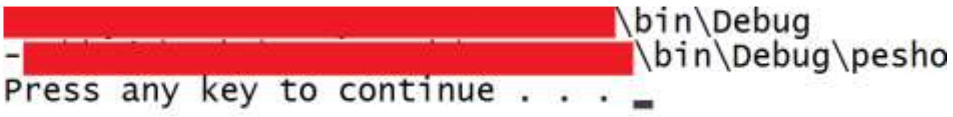
Now that we are done with that and since we now have some space where we can save the current folder, we are going to start our traversal method, using the current folder. All we have to do is **remove** the **string path argument** and also change it with **Session.currentPath**

Your traverse method should now start like this:



```
public static void TraverseDirectory(int depth)
{
    OutputWriter.WriteEmptyLine();
    int initialIdentation = SessionData.currentPath.Split('\\').Length;
    Queue<string> subFolders = new Queue<string>();
    subFolders.Enqueue(SessionData.currentPath);
```

Try testing the functionality now!



```
[redacted]\bin\Debug
-[redacted]\bin\Debug\pesho
Press any key to continue . . . _
```

If this is your result you've done your job well.

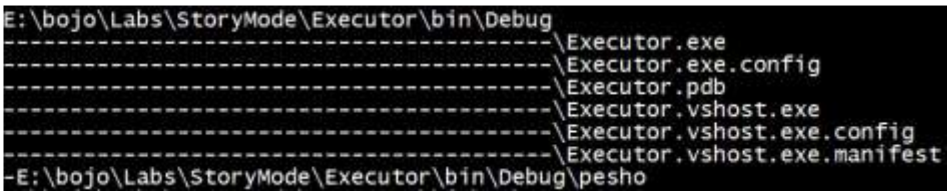
Another thing we might want to add to the current implementation of the traversal, the display of the files in the current folder. It is pretty similar to the adding of the subfolders. All we need is a **foreach loop** and to **use** the **Directory**.**GetFiles**(**path**) to get all the files and display them. The display of the files should be **between** the **display of** the **current path and** the **adding of** the **subfolders**.

```
foreach (var file in Directory.GetFiles(currentPath))
{
        //Display the current file
}
```

In order **to display** the **file**, we will **change** the **path to** the **given file** with **dashes**, because we can see the folder we are in on the line before the display of the files and this way we can focus on the file names.
To get the whole path, we will get the index of the last '\'(backslash) and print a string with such a length of dashes, followed by the file name like shown below:

```
foreach (var file in Directory.GetFiles(currentPath))
{
    int indexOfLastSlash = file.LastIndexOf("\\");
    string fileName = file.Substring(indexOfLastSlash);
    OutputWriter.WriteMessageOnNewLine(new string('-', indexOfLastSlash) + fileName);
}
```

and the output of traversal with depth 0 should now be similar to the given:



```
E:\bojo\Labs\StoryMode\Executor\bin\Debug
-----------------------------------------\Executor.exe
-----------------------------------------\Executor.exe.config
-----------------------------------------\Executor.pdb
-----------------------------------------\Executor.vshost.exe
-----------------------------------------\Executor.vshost.exe.config
-----------------------------------------\Executor.vshost.exe.manifest
-E:\bojo\Labs\StoryMode\Executor\bin\Debug\pesho
```

There is just one final piece of code we need to add to this method and it should be tip top. Bearing in mind that we added a parameter for the depth of the traversal, maybe we should **include** it as some kind of a **condition** in our code **so** that **it would be easier to know when to stop traversing** if we've gone deep enough and in order to check how deep we've gone, we can use the indentation variable that gives us exactly this. So after the assigning of a value to this variable, you can add the following line of code:

```
if (depth - identation < 0)
{
    break;
}
```

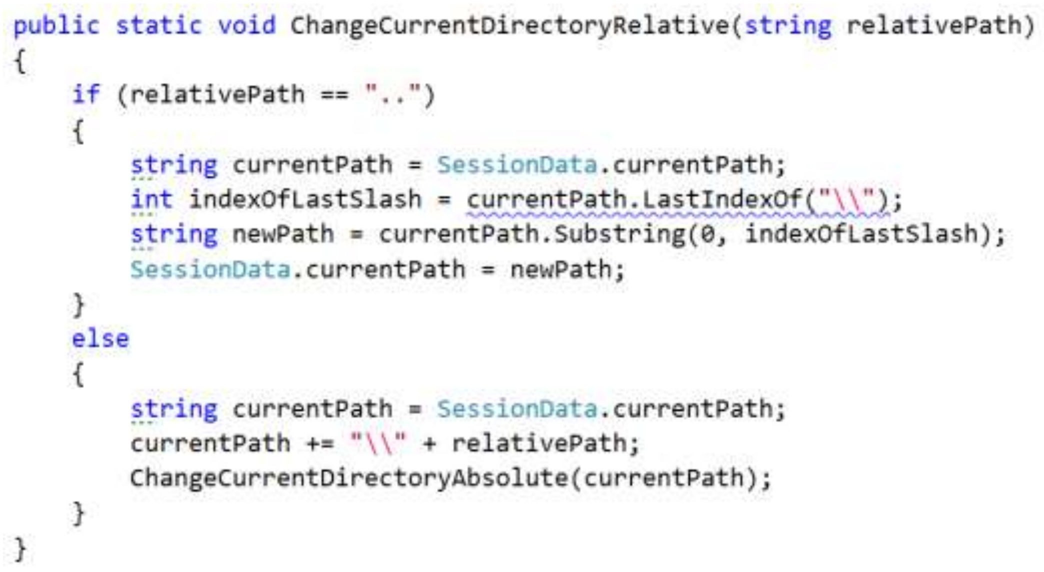
This way we are sure to **stop** the **traversal before** we **print** the **current folder**, **if** we've gone **deep enough**.

Now the question remains, how do we change the starting folder and can we do it with relative and absolute paths. Well we should be able to implement it and the only thing we should probably keep in mind is whether the given path exists.

Problem 5. Changing directories

So again using the **IOManager**, we make two functions. One that moves forwards and backwards in the folders and one that gets an absolute path and goes directly there(***Note***: there are many machine specific things in the path if it is absolute).

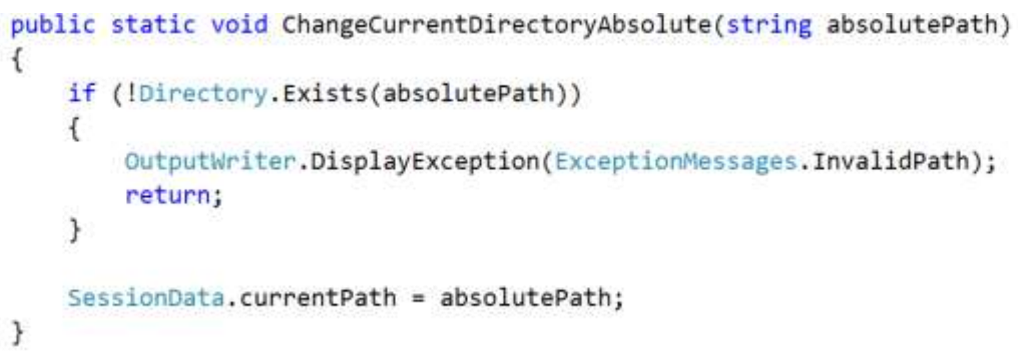
In the **relative change** of **folder method**, we may won't to check **if** the **user** would like to **go one folder back**, and the **command** for this **is** "**cdRel ..**" in the command prompt, so we will use **".."** for a string that **indicates that the user wants to go one folder up the file tree**. **If** he **wants to go into** one **folder**, the **string** for **path** should be the **current session path + '\' + the name of the folder** we want to enter. **Using** the **relative path** and the current path of the traverser, we can easily **create** an **absolute path** an by **using** the **method change for absolute path**, we can **reuse** the **check** whether the given path exists or not.



```
public static void ChangeCurrentDirectoryRelative(string relativePath)
{
    if (relativePath == "..")
    {
        string currentPath = SessionData.currentPath;
        int indexOfLastSlash = currentPath.LastIndexOf("\\");
        string newPath = currentPath.Substring(0, indexOfLastSlash);
        SessionData.currentPath = newPath;
    }
    else
    {
        string currentPath = SessionData.currentPath;
        currentPath += "\\" + relativePath;
        ChangeCurrentDirectoryAbsolute(currentPath);
    }
}
```

Note that for going to the previous path, we **take** the **last index of** the **backslash** which is right after the previous folder and after that we **take** a **substring from 0** with the given **index** representing the **number of elements** before the backslash, so if we take that substring, we have the absolute path to the parent folder of the current one, so we take it as a current folder.
However **if** the **command is not** "**..**", but a path, we add **make** a **new absolute path** and **reuse some code** by calling the other method. This way we have less code duplicates in the two methods.

The change directory with absolute path method is actually not very complicated. All we do is **using** the **API** from the **Directory class**, **check whether** such a **path exists** in the operating system. **If** it **does not**, we **display** an **error message** for **invalid Path** which we should first add in the ExceptiionMessages class, called **`InvalidPath`** **and** a **message** containing the following text: "`The folder/file you are trying to access at the current address, does not exist.`" and after that **return** so that it can exit the method. **If** the **device has** a **folder with such** a **path**, it is **set to** the **currentPath** at the end of the method.



```
public static void ChangeCurrentDirectoryAbsolute(string absolutePath)
{
    if (!Directory.Exists(absolutePath))
    {
        OutputWriter.DisplayException(ExceptionMessages.InvalidPath);
        return;
    }

    SessionData.currentPath = absolutePath;
}
```

By now we should be ready with everything in the **IOManager class**, so we can test the whole functionality. Now you can **test** the **functionality** of everything we've written today and more specific the part with the **IOManager** and if there is something wrong with the whole picture, you may want to fix it, so that everything it according to the documents, for the next exercise.