Lab: Exception Handling

The piece from this lecture is **not going to add** any **additional functionality** to what the final user can see, **only handle** some **possible errors** that may appear for some corner cases. These cases are not so much, because

1. We haven't got so much code, in order to have many error prone places.
2. We are taking safety precautions and check much of the input information, so that such unexpected events can't happen.

So let's start get started with filling some holes in our application.

Problem 1. Cover Possible Unexpected Behavior in Traversal Method in the IOManager

The first thing you might want to think about is whether your user can access all the folders and files in the file system and if there are some that you may not, what happens. Well, let's try.

Try traversing the **windows directory** on your PC, but before that you should go to that directory using the absolute change of directory.

```
static void Main()
{
    IOManager.ChangeCurrentDirectoryAbsolute(@"C:\Windows");
    IOManager.TraverseDirectory(20);
}
```

The result should be something like the following lines:

```
-C:\Windows\L2Schemas
--------------------\LAN_policy_v1.xsd
--------------------\LAN_profile_v1.xsd
--------------------\OneX_v1.xsd
--------------------\WFD_profile_v1.xsd
--------------------\WLANAP_profile_v1.xsd
--------------------\WLAN_policy_v1.xsd
--------------------\WLAN_profile_v1.xsd
--------------------\WLAN_profile_v2.xsd
--------------------\WWAN_profile_v1.xsd
--------------------\WWAN_profile_v2.xsd
--------------------\WWAN_profile_v3.xsd
-C:\Windows\LiveKernelReports

Unhandled Exception: System.UnauthorizedAccessException: Access to the path 'C:\Windows\LiveKernelReports' is denied.
   at System.IO.__Error.WinIOError(Int32 errorCode, String maybeFullPath)
   at System.IO.FileSystemEnumerableIterator`1.CommonInit()
   at System.IO.FileSystemEnumerableIterator`1..ctor(String path, String originalUserPath, String searchPattern, SearchOption searchOption,
hResultHandler`1 resultHandler, Boolean checkHost)
   at System.IO.Directory.GetFiles(String path)
   at Executor.IOManager.TraverseDirectory(String path, Int32 depth) in E:\bojo\Labs\StoryMode\Executor\IOManager.cs:line 34
   at Executor.Program.Main(String[] args) in E:\bojo\Labs\StoryMode\Executor\Program.cs:line 13
Press any key to continue . . .
```

As you've probably noticed, trying to access folders for which we do not have rights, **throws** an UnauthorizedAccessException, and it **ruins** the **user experience and breaks** the **functionality of** our **application**.

In order for our program to **catch** such an **exceptional event**, **handle it and continue working**, we have to **add** the **try-catch block** to **put** the **reading of** the **data in** the **try block** and **if** an **exception** is **catch** we **display** a **message** suitable **for** the **current event**, but **in** a more **user friendly way**.

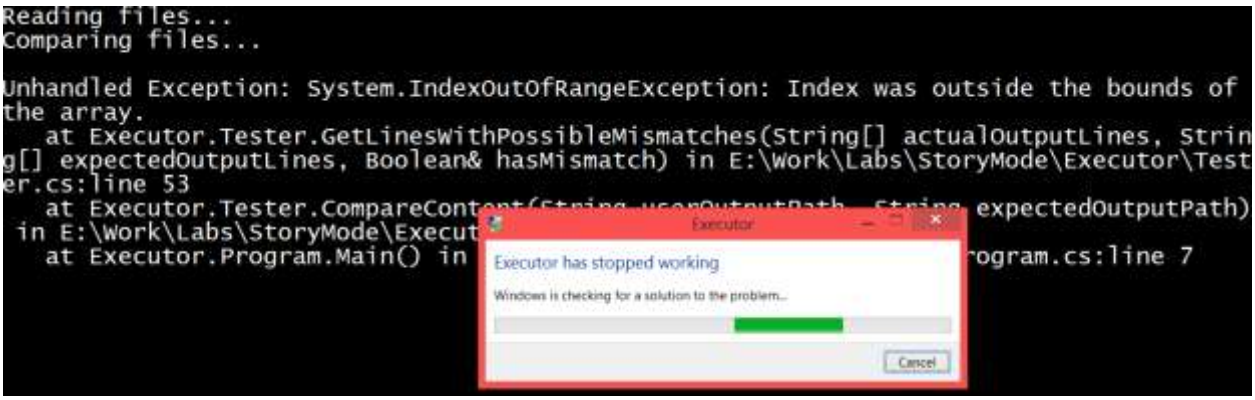
```
try
{
    foreach (var file in Directory.GetFiles(currentPath))
    {
        int indexOfLastSlash = file.LastIndexOf("\\");
        string fileName = file.Substring(indexOfLastSlash);
        OutputWriter.WriteMessageOnNewLine(new string('-', indexOfLastSlash) + fileName);
    }

    foreach (string directoryPath in Directory.GetDirectories(currentPath))
    {
        subFolders.Enqueue(directoryPath);
    }

}
catch (UnauthorizedAccessException)
{
    OutputWriter.DisplayException(ExceptionMessages.UnauthorizedAccessExceptionMessage);
}
```

This type of exception message is not yet in the **ExceptionMessages**, so you should **add it and put** the **following message**: "`The folder/file you are trying to get access needs a higher level of rights than you currently have.`".

Now the possible problems with the traversal are solved. And we can proceed with the next thing.

Problem 2. Reading Two Files for Comparison in the Tester Class

We need to take care of one more thing before we finally leave our main logic and move onto printing the results. What **if one of** the **files** is **smaller than** the **other one**? **Try comparing** the **two files** given to you, called **expected and actual** from the current piece and you may **see** the **result**. It should be something like this:



The **outputs** are definitely **not identical**, but we still would like a match/mismatch report. There are many ways to achieve this but maybe to catch the exception here would not be the best choice. For that reason, we are going to **add one variable**, the **minimal number of lines of** the **two files**. We **check if** the **arrays** that hold all the lines from the files, **are with** the **same length** and **if** they **are not**, **set** the **minimal number of lines to** the **shorter length**, **set** the **hasMismatch** variable to **true** and finally **display** an **error.** However, we first need to **add it to** the **Exception messages class**, named **ComparisonOfFilesWithDifferentSizes** and with the following message "`Files not of`

`equal size, certain mismatch.`" All what we've just talked about is displayed below in the piece of code that you should insert before the for loop that compares line by line.

```
int minOutputLines = actualOutputLines.
if (actualOutputLines.  != expectedOutputLine  )
{
     = true;
    minOutputLines = Math.  (actualOutputLines.  , expectedOutputLines.  ;
                          n(ExceptionMessages.ComparisonOfFilesWithDifferentSizes);
}
```

After that we should only **replace** the **variable in** the **for loop** for the **upped boundary** of the index.

```
for (int index = 0; index < minOutputLines; index++)
```

Finally, we should also **move** the **initialization** of the **mismatch array**, **under** the **if statement** and also **change** the **capacity of** the **array to** the **value of minOutputLines**.

Now that we've fixed the situation here, we should **proceed to** the **next step**.

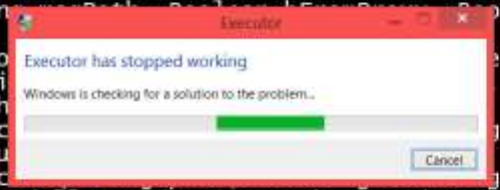
Problem 3. Reading two files for comparison from invalid path

We took safety precautions about the number of rows in each file, but what we didn't think of, **what could happen if** the **path given** to the file is **not** a **valid** path. Let's try it:

```
static void Main()
{
    Tester.CompareContent(@          Labs\actdual.txt", @"          abs\expecdted.txt");
}
```

Results in the following:

```
Reading files...

Unhandled Exception: System.IO.FileNotFoundException: Could not find file 'E:\Work\Lab
s\actdual.txt'.
   at System.IO.__Error.WinIOError(Int32 errorCode, String maybeFullPath)
   at System.IO.FileStream.Init(String path, FileMode mode, FileAccess access, Int32 r
ights, Boolean useRights, FileShare share, Int32 bufferSize, FileOptions options, SECU
RITY_ATTRIBUTES secAttrs, Strin                                              olean useLongPath, Boo
lean checkHost)
   at System.IO.FileStream..cto                                              Access access, FileSh
are share, Int32 bufferSize, Fi                                               Boolean bFromProxy,
Boolean useLongPath, Boolean ch
   at System.IO.StreamReader..c                                              , Boolean detectEncod
ingFromByteOrderMarks, Int32 bu
   at System.IO.StreamReader..c                                              )
   at System.IO.File.InternalReadAllLines(String path, Encoding encoding)
   at System.IO.File.ReadAllLines(String path)
   at Executor.Tester.CompareContent(String userOutputPath, String expectedOutputPath)
 in E:\Work\Labs\StoryMode\Executor\Tester.cs:line 14
   at Executor.Program.Main() in E:\Work\Labs\StoryMode\Executor\Program.cs:line 7
```



If we are **making** any kind of **user interface**, the **application should** always **presume** that the **user is** a **two**-year-old and **can probably do or enter** just about **everything** you can imagine **and** even **more**.

So the thing we are going to **change** in the **Tester class** is to **put** the **reading from** the **files in** a **try block** and **catch** the **file not found exception** and **display** a **related** to the **error message**. Now you code should be looking like this:

```
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!");
```

This should change to:

```
try
{
    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!");
}
catch (FileNotFoundException)
{
    OutputWriter.DisplayException(ExceptionMessages.InvalidPath);
}
```

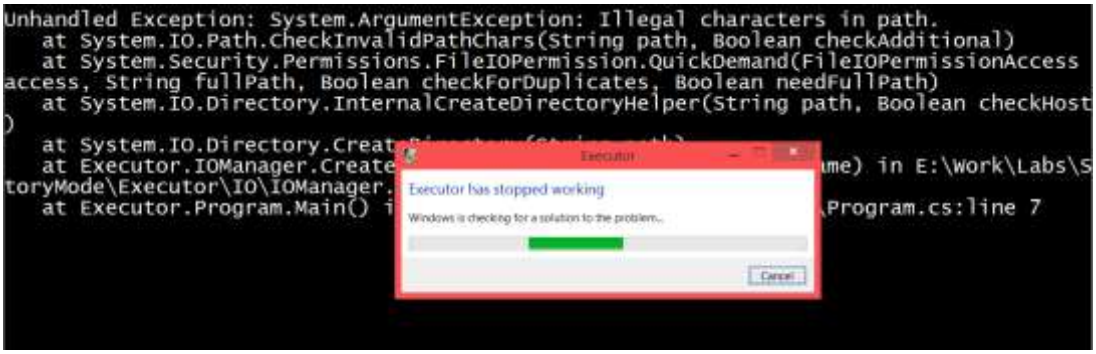
We are reusing the message for the invalid path in the current action, so we do not need to make a new one. Alright, now that we are done, let's proceed to what is considered forbidden and what is consider allowed when talking about creating names of files and folders.

Problem 4. Making a Directory with Illegal Symbols

I don't know if you've noticed but not every symbol is permitted to be used when giving a name to a folder or a file. This is why we must **consider listening for exceptions when** the **user creates** a **new folder using** the **public method** **`CreateDirectoryInCurrentFolder`**, because the **user can always make** some **mistakes** and **enter** an **invalid folder/file name**… Let's see what happens now if we **try to create** a **new folder called *2**.

```
static void Main()
{
    IOManager.CreateDirectoryInCurrentFolder("*2");
}
```

And the result of the current operation will give us the following horrible screen:



Our task now is to **catch** that **argument exception** and **display** an **understandable user message on** the **output writer**

The **operation** that **throws** the **exception in** the **creation** of **directory method** is clearly the **Directory.CreateDirectory(path)** and since we know that fact, we can easily **put it in** a **try block**, to **catch** the **raised exception**.

The modified implementation of the method should look pretty similar to the following piece of code:

```
string path = GetCurrentDirectoryPath() + "\\" + name;
try
{
    Directory.CreateDirectory(path);
}
catch (ArgumentException)
{
    OutputWriter.DisplayException(ExceptionMessages.ForbiddenSymbolsContainedInName);
}
```

As you can see we are **displaying on** the **output** a **message called ForbiddenSymbolsContainedInName**, however it is **no yet added in** the **ExceptionMessages class**, **so** it is your job to **do it** now. The **message** it has **is** "**The given name contains symbols that are not allowed to be used in names of files and folders.**".

Now you can **try starting** the **program again and** the **output should** be the **user styled message**.

Problem 5. Printing to a Non-Existing Path

Since we generate the path for the mismatches from the expected output path, if it is wrong, the program shouldn't even arrive to the point in the PrintOutput in the Tester class, however we can never be sure whether some event might trigger such an exception, so that's why we'll double check and put the File.WriteAllLines in a try block with a DirectoryNotFoundException catch block watching whether such an exception is raised. After this change the print output should look like this:

```
if (hasMismatch)
{
    foreach (var line in mismatches)
    {
        OutputWriter.WriteMessageOnNewLine(line);
    }

    try
    {
        File.WriteAllLines(mismatchesPath, mismatches);
    }
    catch (DirectoryNotFoundException)
    {
        OutputWriter.DisplayException(ExceptionMessages.InvalidPath);
    }

    return;
}

OutputWriter.WriteMessageOnNewLine("Files are identical. There are no mismatches.");
```

Problem 6. Going One Folder up the Hierarchy

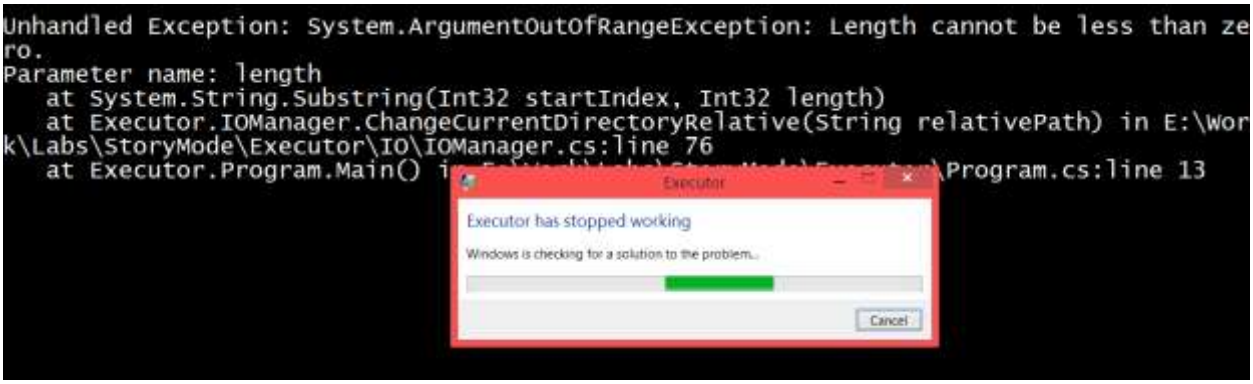
As we know, the logic for the changing of the folders works correctly, but have you tried to go one folder up when you are in the root folder of the partition.

Let's **call** the **ChangeCurrentDirectoryRelative enough times with** the parameter **"..", so** that we are **sure to go up until** the **root folder of** the **current partition and** then **one folder above**.

In my case that's 7 calls of the following line of code:

```
static void Main()
{
    IOManager.ChangeCurrentDirectoryRelative("..");
    IOManager.ChangeCurrentDirectoryRelative("..");
    IOManager.ChangeCurrentDirectoryRelative("..");
    IOManager.ChangeCurrentDirectoryRelative("..");
    IOManager.ChangeCurrentDirectoryRelative("..");
    IOManager.ChangeCurrentDirectoryRelative("..");
    IOManager.ChangeCurrentDirectoryRelative("..");
}
```

And that results in the following situation:



If we **put all** the **operations** that are in the **body of** the **if** that checks for the two dots, **in** a **try block**, we'll be able **to catch** the **raised exception** in the exact time **and print** the **corresponding message** for such a situation.

```
try
{
    string currentPath = SessionData.currentPath;
    int indexOfLastSlash = currentPath.LastIndexOf("\\");
    string newPath = currentPath.Substring(0, indexOfLastSlash);
    SessionData.currentPath = newPath;
}
catch (ArgumentOutOfRangeException)
{
        OutputWriter.DisplayException(ExceptionMessages.UnableToGoHigherInPartitionHierarchy);
}
```

Now try running the same code you did and see the result.

These are surely not all the exceptional cases in our program, but these are some of them. You may use the techniques that we used in order to find these holes in the functionality and try to find some other errors that might occur.

Congratulations! You've completed the lab exercises for Exception Handling.