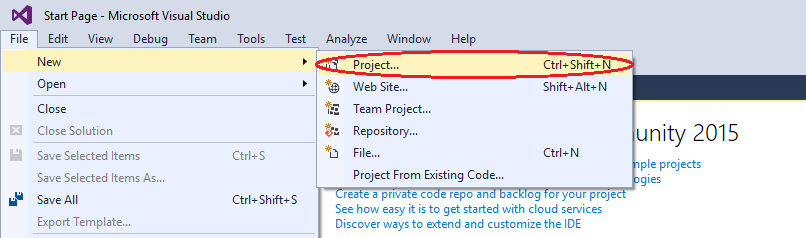
# BashSoft piece: Stacks and Queues

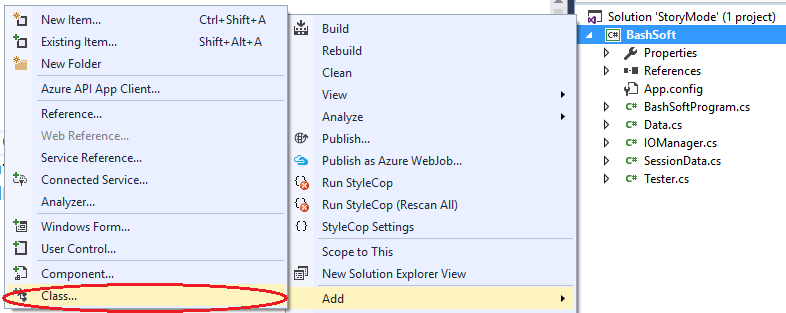
# Part I: Creating the base functionality needed

## Create a Visual Studio Project

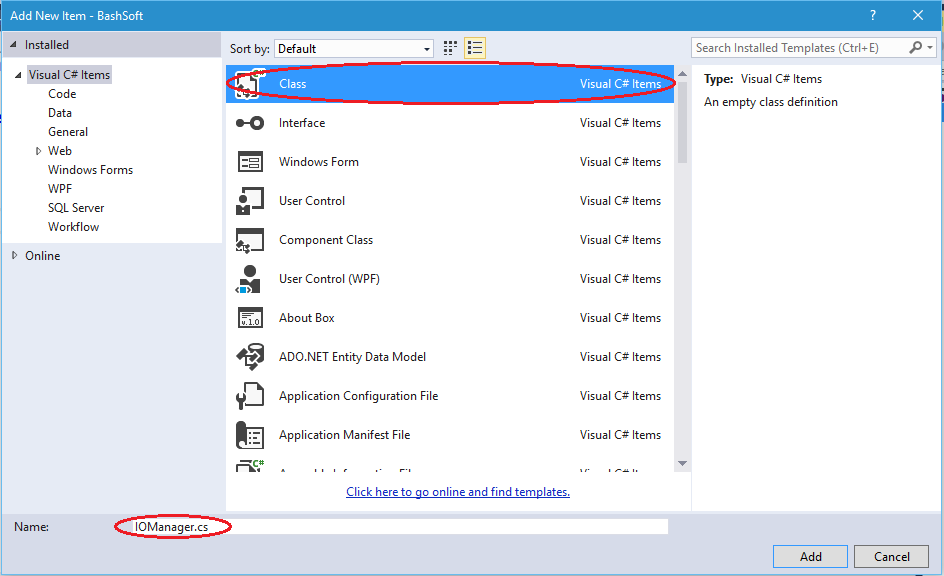
Our first task is to **create** a **project** called **BashSoft**, which we will extend until the end of the course so you might want to **save** **it** somewhere, where you can **easily** **find** it and where you can be **sure** you **won’t delete** it. You can call the class with the **Main()** method, **Launcher**, because from it we will only call the specific functions we want to execute, but our execution logic will be in other classes.



Once you have created the project, you have to add a class that we will call **IOManager** and it will give us the functionality for traversing the folders and other behaviors.

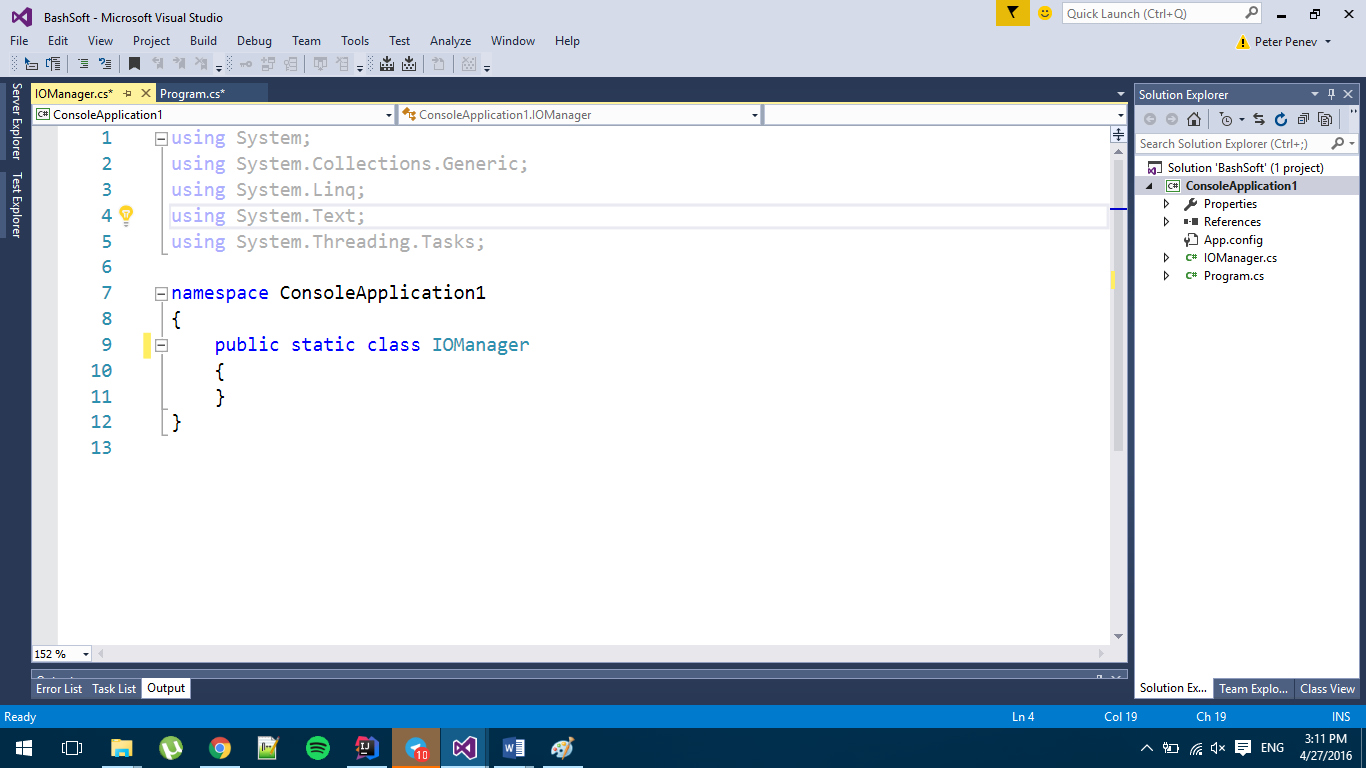


In the next menu you have to choose to create a new class with the name “**IOManager**”



Next the only things we have to **change** over the **generated class** is **to** add it “**public static**” before the keyword **class**. The keyword “**public**” means we can **use** our class **everywhere** in our project. Sometimes we will leave some methods **private**, because we may want to **hide** some of the **functionality** of our class, in front of the other world. The other keyword “**static**” means that we can do “**general/global**” stuff with it. Example: “**Math**”, “**Console**”.

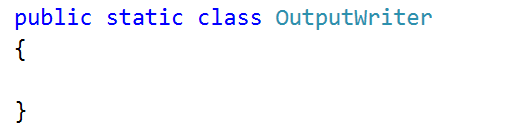
The **opposite** **of** **static** we can say are classes like “**Stack**, **List**, **StringBuilder**” for which we have to say “**new List<T>”** in order **to** **create** a **new** list. The **static** classes **do not need to be created** like we don’t say “new Math”, instead we just use **Math.Sqrt().**

So now your class should look something like this:

## Create a flexible interface for output to the user

So now that we have our first class we are going to have to implement some functionality that this class should have. But before that, first we have to decide **how** are we going **to** **communicate with the user efficiently** and if this is something that we have to use in many places, how can we change it or replace it easily using doing only a few changes in one place. The solution behind such a problem give us one of the [**Design Patterns**](https://en.wikipedia.org/wiki/Software_design_pattern) which are a topic of the **next course**, but the main idea of this one is that we can **hide** some **functionality** (The writing to the console, which can easily be changed for writing in a file), **by using** a **class** that only gives us **base functionality** for communication with a user.

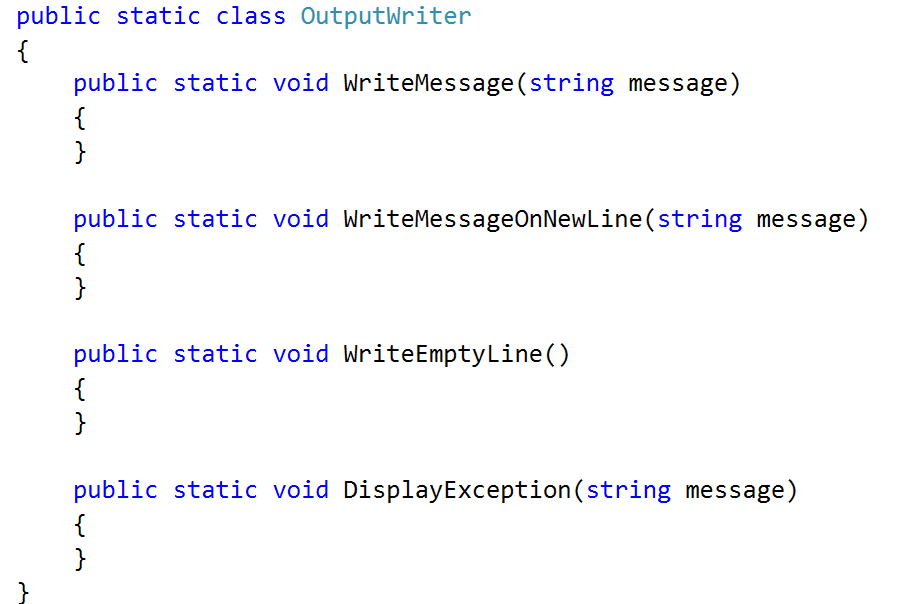
Our new class can be called **OutputWriter** and you should make it following the steps above as described for the **IOManager**. The **new class** again has to be **public** and **static** and after you’ve created it, it should look something like this:



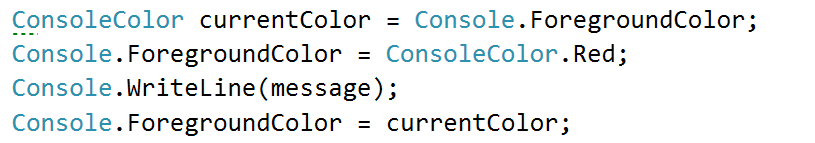
So now we can add a few methods that we will use throughout our whole app that write to the currently set output.

* The first method gives us the ability to **write a message.**
* The second method to implement is a method for **writing a message on a new line**.
* The third method is to **write a new empty line**.
* The fourth method is to **write a different kind of message** which is an **error/exception**.

The class with the three methods inside it should look something like this:



The implementation of the first three methods is pretty common. The **first** one only **writes the message on the console**, and the **second** one **writes the message** and goes to the **next line after that**. The **third** only **writes an empty line on the console**. The **fourth** method however has some small specifics. The specifics are that we need to **get** the **current** **foreground color**(font color), **save it**, **change the foreground color to red**, **write the given message** and finally **change** the **foreground color** **to** the **one** **before**. Here is how this has to look in code:



Now that we are ready with the user output. It’s time to implement the traversal of the folders and in the future, **if** we want to **change** the **output destination**, we **only** need to **change** it here in the **class** we **just made**, and not everywhere where we’ve written **Console.WriteLine()**.

# Part II: Implementing the traversal algorithm

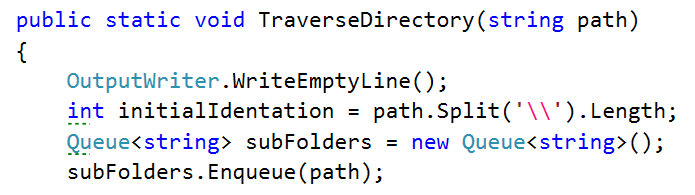
## Traversing the folder of the project

Our next task is to learn how to **traverse folders** in order to be able to do all kinds of operations with files that are stored on the hard drive. This is our first small step into the big picture.

We will **traverse the folder** of the project **using** **queue** with a technique called [**BFS**](https://en.wikipedia.org/wiki/Breadth-first_search). [Here](https://upload.wikimedia.org/wikipedia/commons/5/5d/Breadth-First-Search-Algorithm.gif) is a animation that can probably help you understand how **BFS** works, however this is not the main point, so you may just use it, without going into too much depth about how it works.

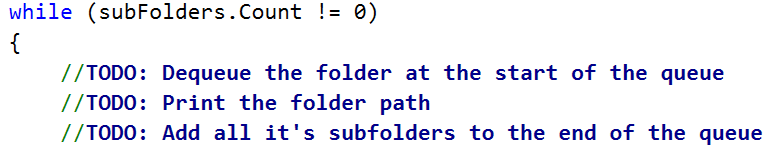
Shortly we will create a method **TraverseFolder (string path)**. How does it traverse a folder? **First** it **enqueues** the **folder** that we **pass as parameter** in the method signature. After that it **dequeues** **every** **folder** in the queue one at a time **until** the **queue** becomes **empty**, **while** **at the sam**e time **enqueues** **all** of its **subfolders** at the end of the queue.

For our purposes we will **use** the static class **DirectoryInfo**, which **will** **give** us all the **information** we need **for** the **directories** we work with, don’t worry you’ll get familiar with it in a few lectures. Here is the initialization of the method with the queue. We **enqueue** the **root** **folder** we wanted to traverse first and also **create** a **variable** **for** the **indentation** of the first path, so it can be later **used** **for** **displaying** the **levels** **of depth** we’ve entered while traversing.

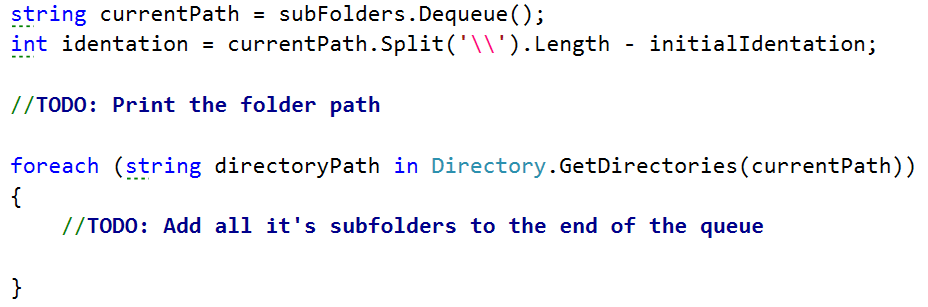


Next we need to make sure we will **traverse** **all** of the **subfolders** that we have **in** the **queue** so we will traverse **while** the **queue** is **not empty** (that is why we **push** the **initial** **element** in the queue).

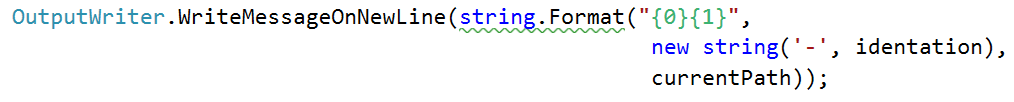
**For each iteration** **of** the **while** **loop** we want to **dequeue** a **folder** that we are going to traverse and to **print** its **path**, but in order to know how many level in depth we have entered, we are going to **use** **another** **indentation** **variable** and **take** **the delta between the two**.

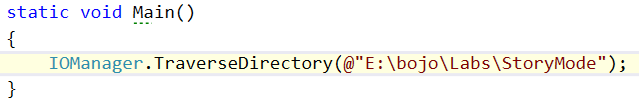


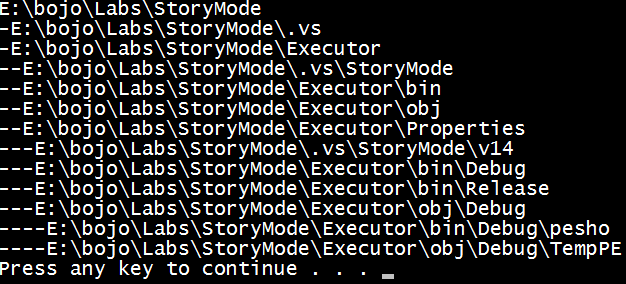
Also **for each folder** we need to **iterate** **all** its **subfolders** and **add** **them** **to** the end of the **queue**. We can do this with a simple foreach loop:



You can **print** the **full** **name** **of** the **directory** with the following line of code:



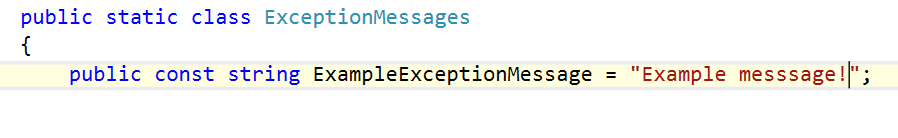
Now if you filled your TODOs properly when you run your code you should get some output like this if we call the method through the **Main()**   
  




You are now ready with your first tool for the wanted bash. Soon you will be able to easily change your position in the file system and do different operations with other files.

# Part III: Exceptions handling

## Creating a set of error messages to display if a function produces such an error:

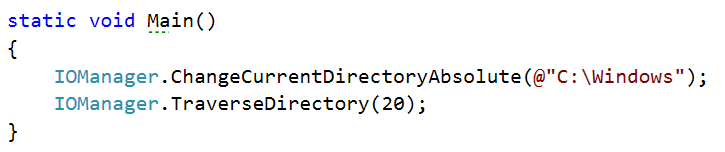
Since we are making a **fairly** **big** **project**, we will have **different** **constant** **messages** to display **in** the whole **project** to the user, so a **good** **idea** would be to **extract** **all** these **messages** **in** **one** **place** and be able to **change** what you want **from** **1** **place** only. So now we are going to **create** such a **class**, where to **save** such **messages** that are **used** **often**.  
The **class** should be **named** **ExceptionMessages** and is **public** and **static**. The only things we are going to **put** in this class are **public** **const** **strings** **with** a given **name** **and** it’s corresponding **message**:  
  


So from now on, every time we have to add a message you should follow the format described above.

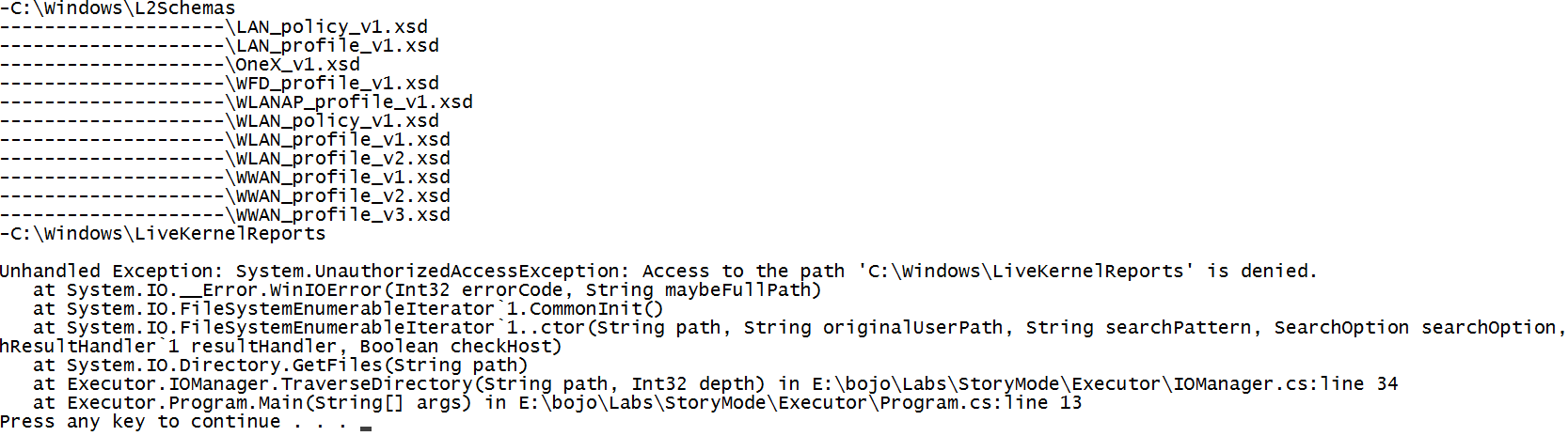
## Cover Possible Unexpected Behavior in Traversal Method in the IOManager

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.

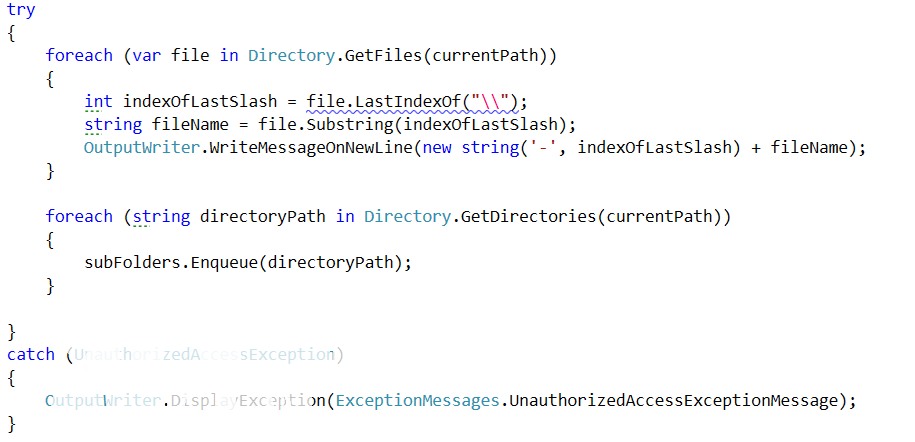
**Try** **traversing** the **windows** **directory** on your PC, but before that you should go to that directory using the absolute change of directory.



The result should be something like the following lines:



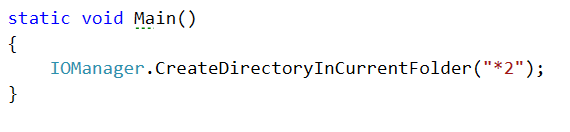
As you’ve probably noticed, trying to access folders for which we do not have rights, **throws** an **UnauthorizedAccessException**, and it **ruins** the **user** **experience** **and** **breaks** the **functionality** **of** our **application**.

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**.  
  


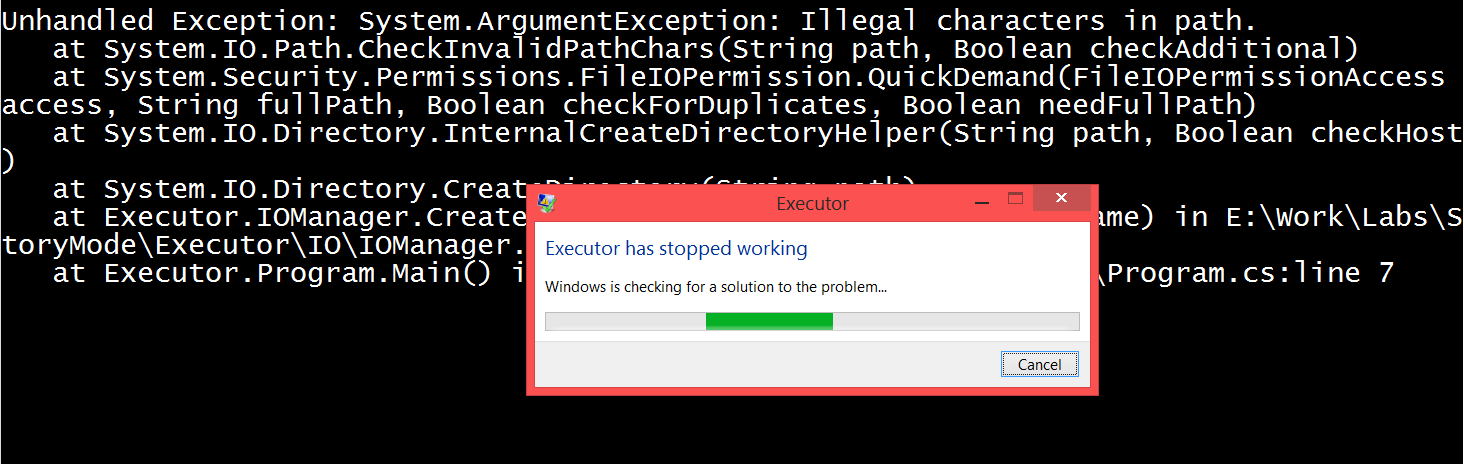
This type of exception message is not yet in the **ExceptionMessages**, so you should **add** **it** **and** **put** the **following** **message**: “The folder/file you are trying to get access needs a higher level of rights than you currently have.”  
  
Now the possible problems with the traversal are solved. And we can proceed with the next thing.

## 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**.



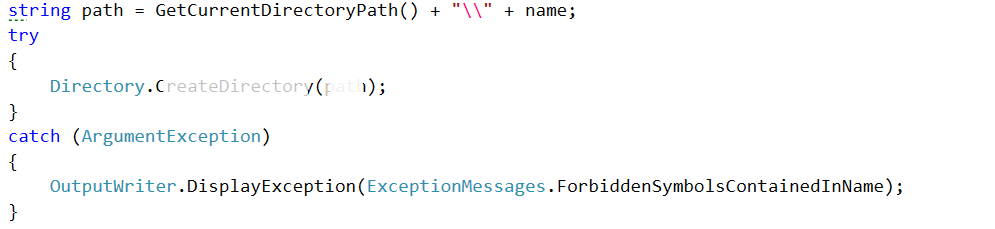
And the result of the current operation will give us the following horrible screen:



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:



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**.

## 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:

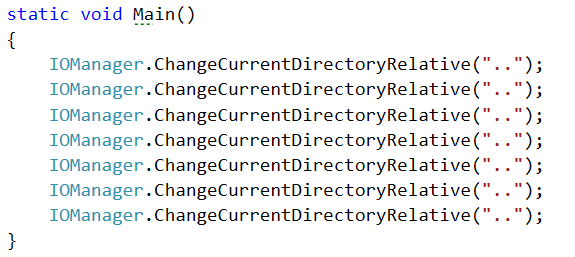


## Going One Folder up the Hierarchy

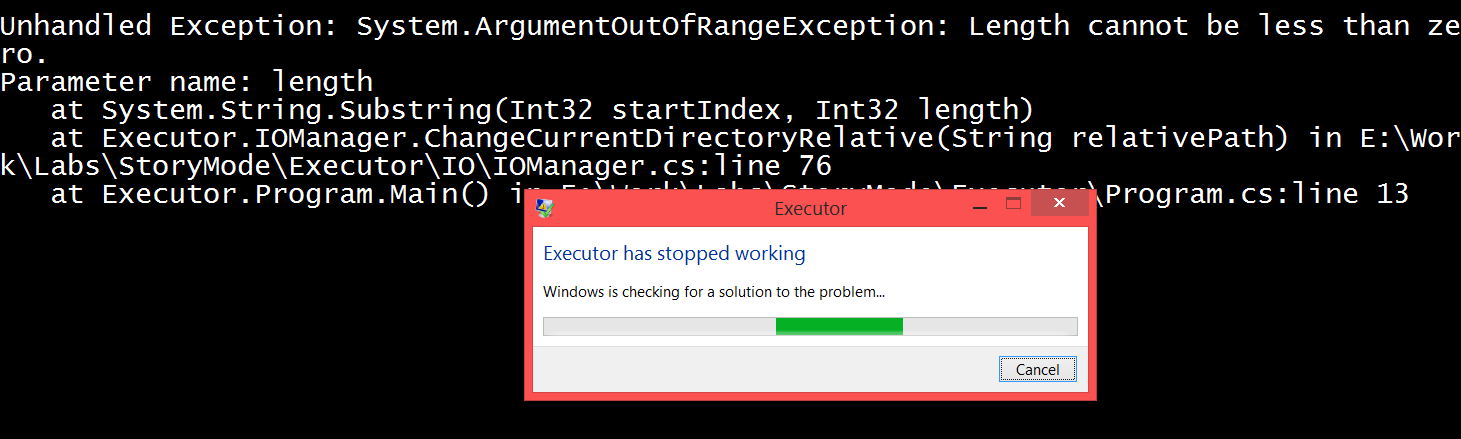
As we know, the logic for the changing of the folders works correctly, but have you tried to go one folder up when you are in the root folder of the partition.

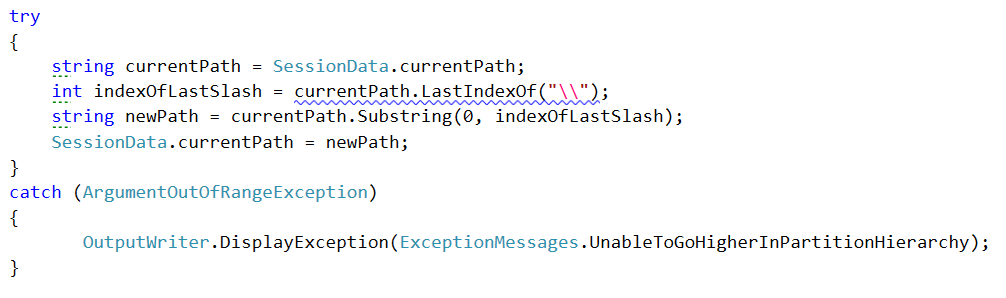
Let’s **call** the **ChangeCurrentDirectoryRelative** **enough** **times** **with** the parameter **“..”,** **so** that we are **sure** **to** **go** **up** **until** the **root** **folder** **of** the **current** **partition** **and** then **one** **folder** **above**.

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



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.

  
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.