# Lab: Files and Directories

This document defines the lab overview for the ["Java Advanced" course @ Software University](https://softuni.bg/trainings/1377/advanced-java-may-2016). Please submit your solutions (source code) of all below described problems at the end of the course at [softuni.bg](https://softuni.bg/trainings/1377/advanced-java-may-2016).

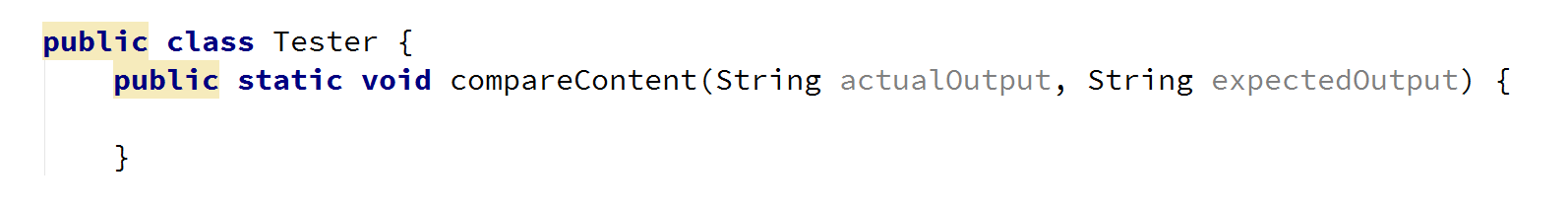
# Introduction

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.

# Part I: Tester Class

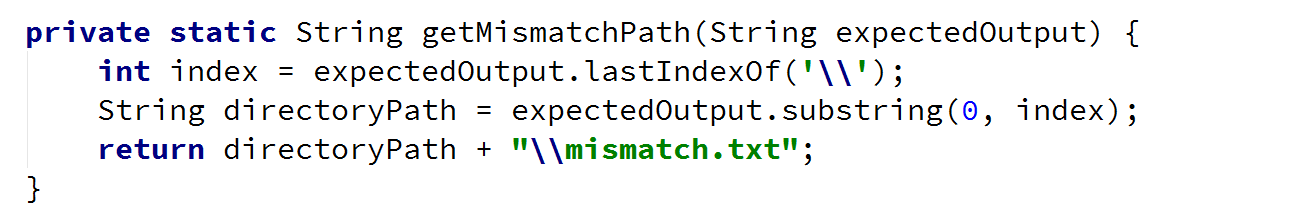
## Set up our Tester Class

Add a new class called "Tester". Mark, it as public and declare a new public static void method called compareContent(), which will compare content:

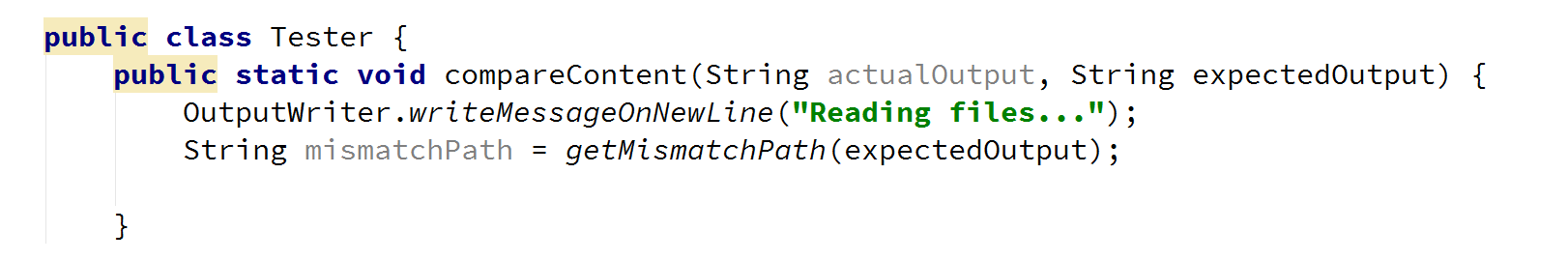


The idea here is that using paths to the **actual output** and to the **expected output**, we can find the files holding the user output and expected output respectively, read and compare them line by line and see if they are identical.

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 we can **use the output path** 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 can create a method:

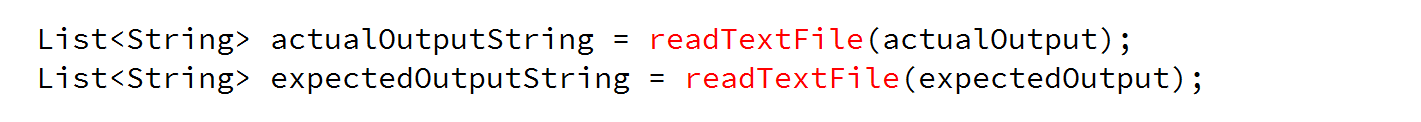


This method simply gets the path to the directory of the expected output file by finding the index of the last slash. It then appends the name of our file and a slash “\mismatch.txt” and returns a string with the new path. Now let’s call the helper method:

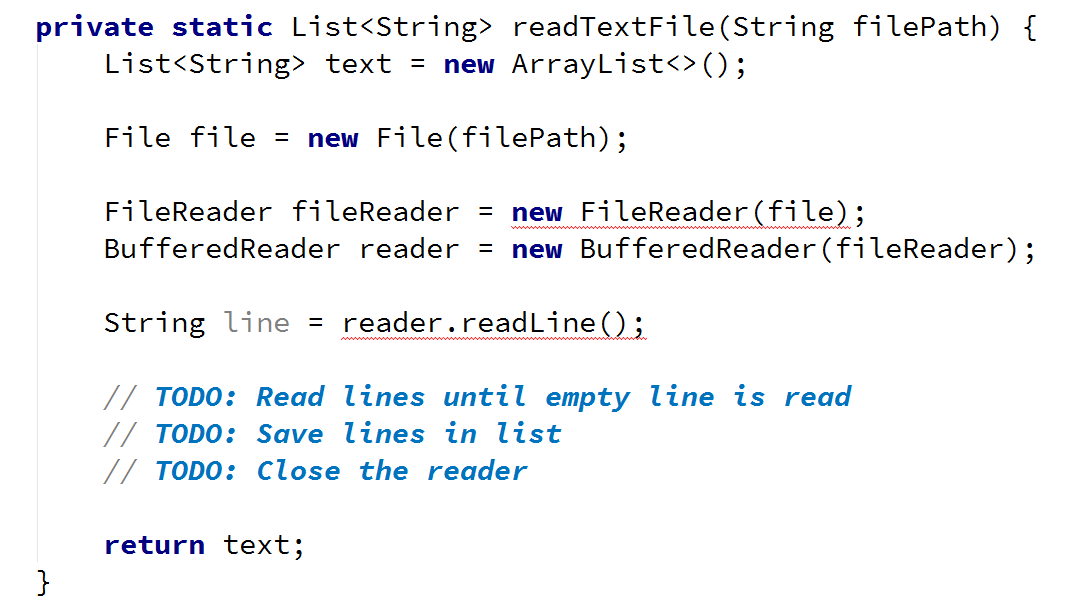


## Reading Text from a File

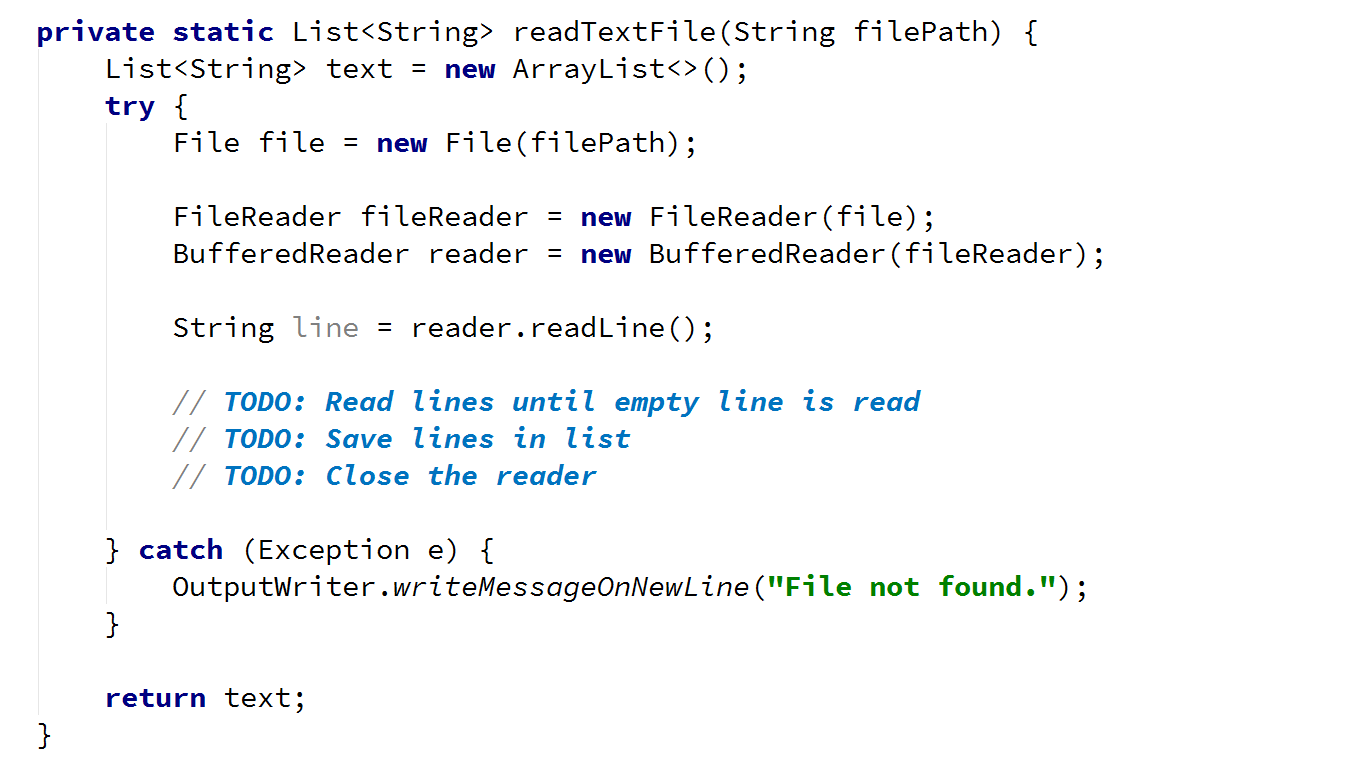
Next up we need to read the two files that will be compared and create a file with mismatches. We will make our own method for reading from files:



The method needs a path to a file, and returns a list with lines. It first creates a new file and then creates a buffered reader which uses a file reader to read the file.



You can see that the code above wont compile. This is because the java compiler is smart enough to warn us that an Input Output exception can be thrown. This means that we must wrap our code in a try catch block:



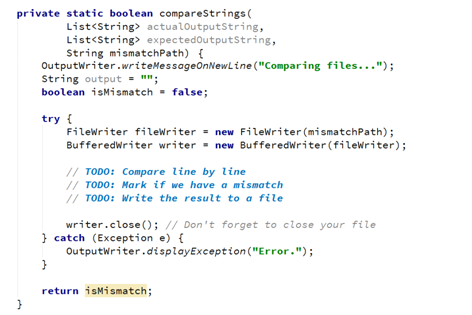
Replacing the TODOs with a simple while loop is your job. **And don’t forget to close your file.** Think about what is the problem with the above code. Consider using **try-with-resources**.

## Writing Text to a File

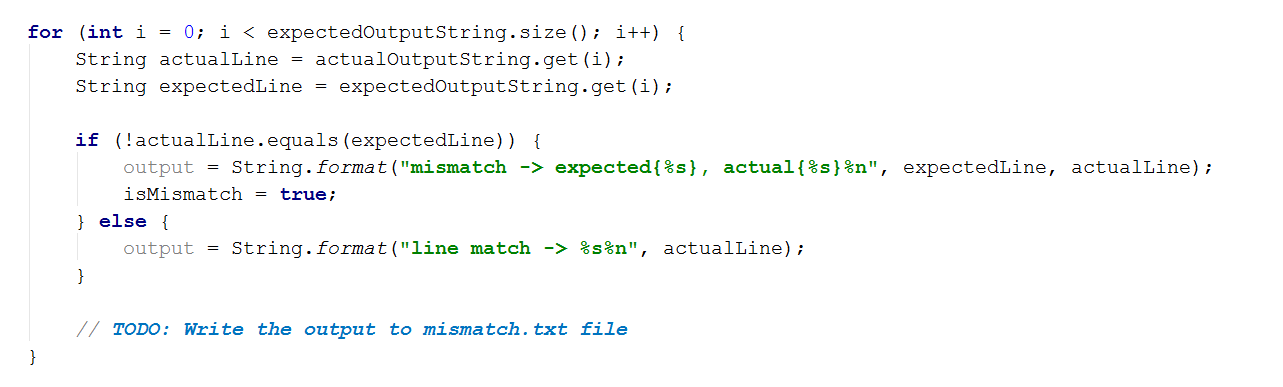
Next, we will create another method in which the actual comparing will take place. It will take as parameters three different things, two lists with the actual and expected output and one with the mismatch path that we created earlier:



It will return a Boolean variable, which we can use when choosing the output. Also, it will use two different streams that will depend on each other, a File Writer wrapped in a Buffered Writer.



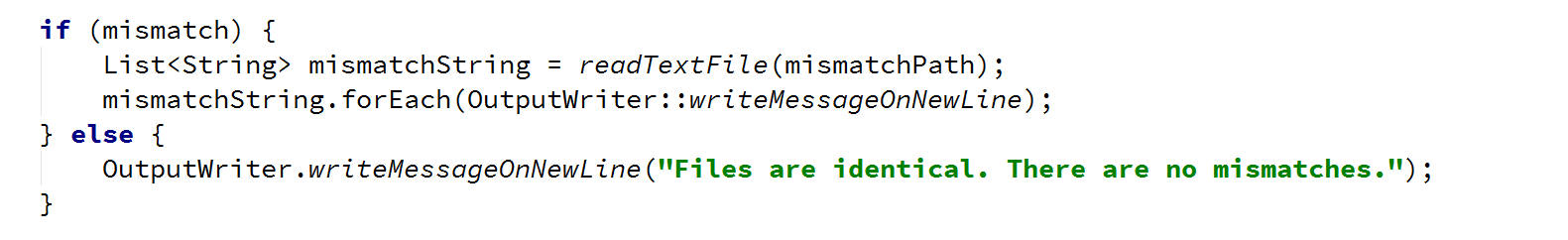
Again, think about what is the problem with the above code. Consider using **try-with-resources**.

What we are doing next is pretty straightforward. We simply iterate over all the lines from both files by assigning the actualLine variable and the expected line to the expectedLine and comparing them. If they are not matching, we mark mismatch as true, increase the mismatchCount by one, and we will write down the mismatch in the output variable. If, however we don’t get a mismatch, if the lines are identical then we simply write down the line in output. In both cases, we then save the output to the 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. With all that out of the way let’s get to writing things down!

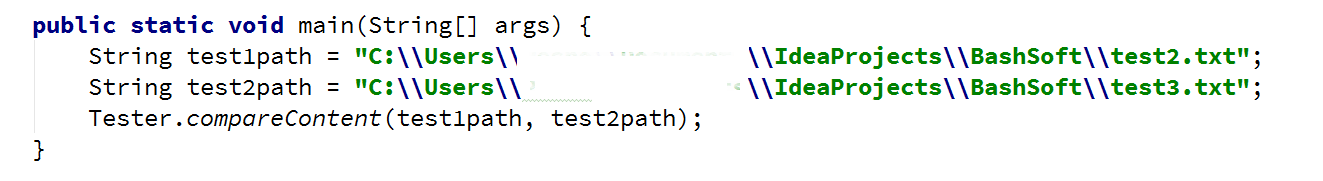
Complete the TODO yourself.

Also, we need to take care of one more thing. What if one of the files is **smaller** than the other one? For now, we are sure that this exception will be handled within the catch block. How exactly, is a different matter, that we will leave for now and come back to at a later stage.

So now, we need to print out the details of the “test”. If there were mismatches we read the **mismatches.txt** and print it on the console and if the files are identical, we print just that. Put the code below at the end of compareContent() method.



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.



That’s about it in terms of reading/writing from/to a file. If you’ve followed the steps thoroughly then you will have your own simple judge which you can use to compare text files. Superb!

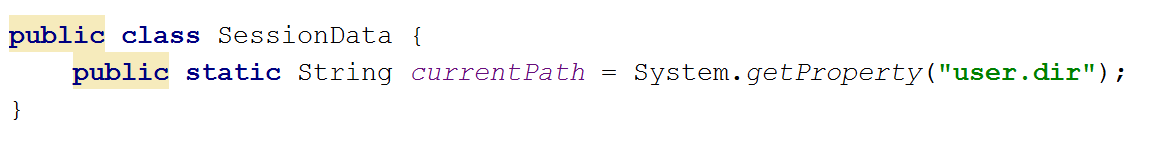
# Part II: Adding Functionality

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

## Saving Data for Our Current Session

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 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.



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 "src" folder and **read the** **Database** from a **file** and not from the **console**…

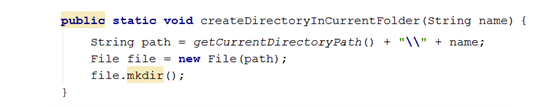
We are going to go through each of these steps in big details so you would be able to understand how each component works.

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

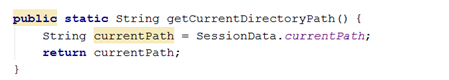
## Creating Directories

First, we are going to make a method for creating a directory in the IOManager. Since we have our currentPath in the SessionData class, all we need to know is the name of the folder we are going to create.

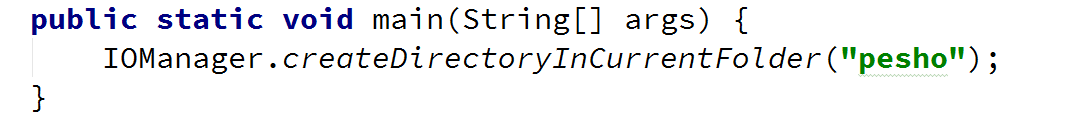
Our method can be called createDirectoryInCurrentFolder() and its implementation should look like this.



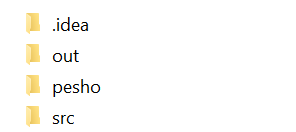
And a helper method for getting the current directory path.



Now we can call it from the main method and we can create a folder named "pesho" there in the folder that our application is currently running from.



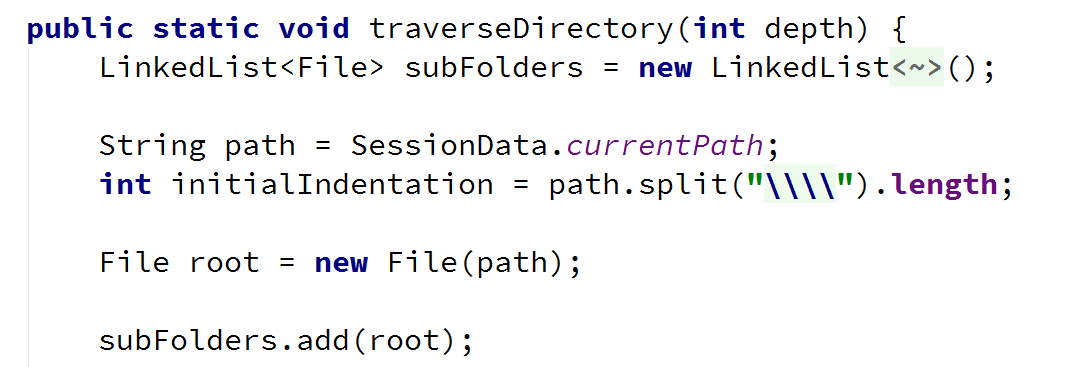
And you can see the result in your file explorer.



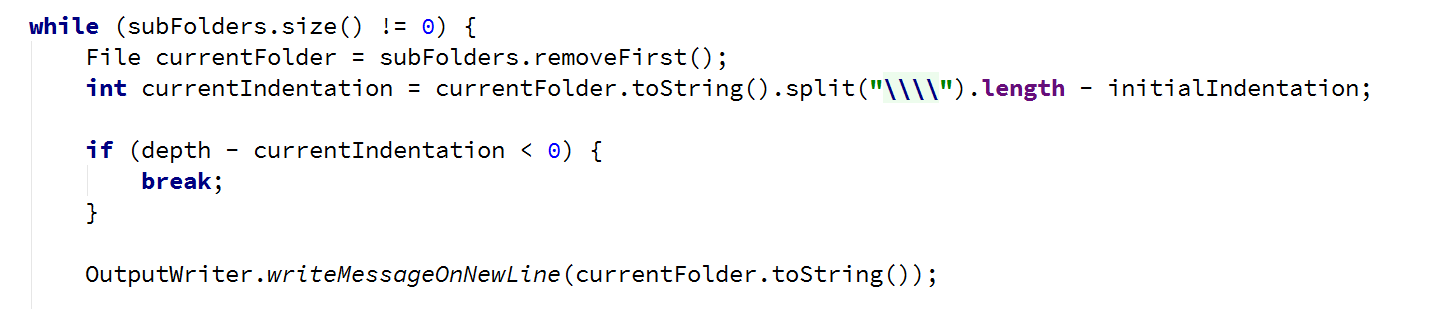
## Modifying the traversal

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 replace the string path argument with an integer depth one. We will always start the traversal from SessionData.currentPath**.**

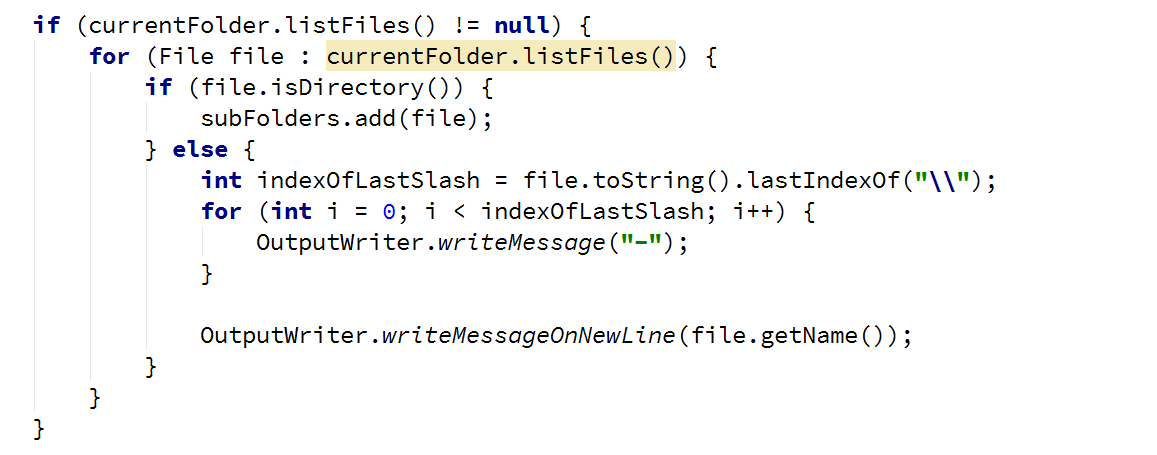
Your traverse method should now look like this:



Another thing we might want to add to the current implementation of the traversal, is stopping the traversal at a given depth. We passed the desired depth as parameter so now we need to break the loop if we reach it. We are going to do this using the folder path. This is why we extract the initial indentation above and we can use it to know how deep we are currently traversing the folders.



Also, we need to display the files in the current folder. It is pretty similar to the adding of the subfolders but if it is not a folder we print the file.



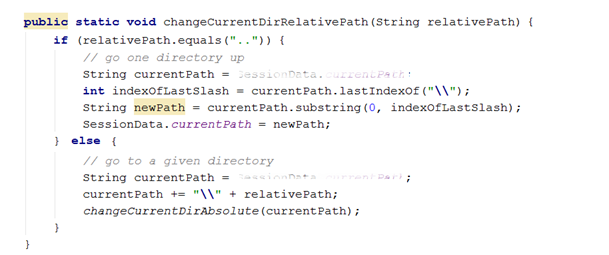
Now one question remains, how do we change the starting folder and can we do it with relative and absolute paths. Well we should be able to implement it and the only thing we should probably keep in mind is whether the given path exists.

## Changing Directories

So again, we are going to extend a bit further the IOManager. We are going to add two methods, one for changing the current path by relative path and one by absolute one.

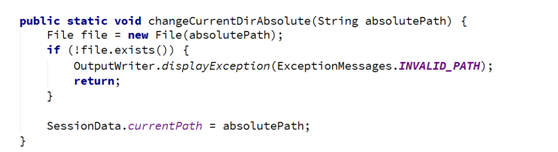
In the relative change of folder method, we may want to check if the user would like to go one folder back, and the command for this is "cdRel .." in the command prompt, so we will use “..” for a string that indicates that the user wants to go one folder up the file tree. If he wants to go into one folder, 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 and by using the method change for absolute path, we can reuse the check whether the given path exists or not.

The method should look like this:



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 File 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 ExceptionMessages class, called INVALID\_PATH and a message containing the following text: “File does not exist.”.



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.

Congratulations! You’ve successfully completed the lab exercises!