# Lab: Manual String Processing

Probably the key moment for the application we are building is just now, because our app is a stack of different functionality, that is coupled to the class with the Main method and to be more specific to the commands we have written there. However, our application has no predefined order of the commands and the main idea is that it should be able to interpret such at runtime. So now our job is to **build** an **interpreter** **that** **calls** the **functionality** that we have.

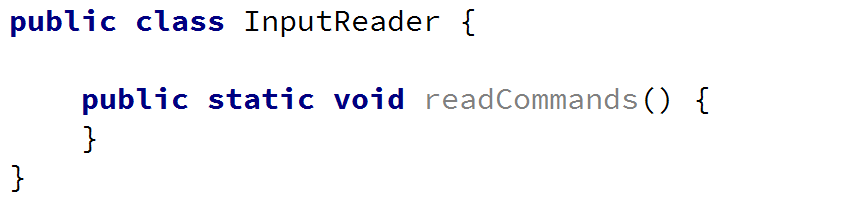
We are going to need **two** **public** **static** **classes** that **handle** the **input** and the **commands**. The **first** **one** is called **InputReader** and the second one is called **CommandInterpreter.**

Now that you have these classes we are going to have to write some code in order for them to do their jobs.

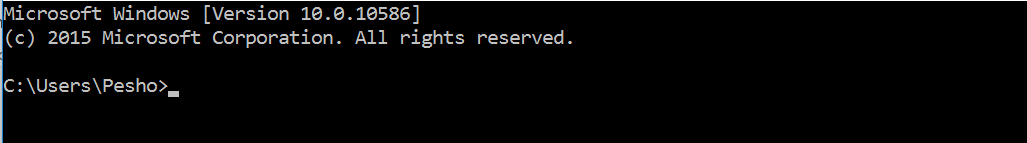
## Implement InputReader Class

First we are going to start with the InputReader, because it uses the command interpreter to do some of its job.

The only method for now will be a method that will be called from the main that starts to listen for commands and executes them if the syntax is correct. We will name this method readCommands() and it’s return type will be void.

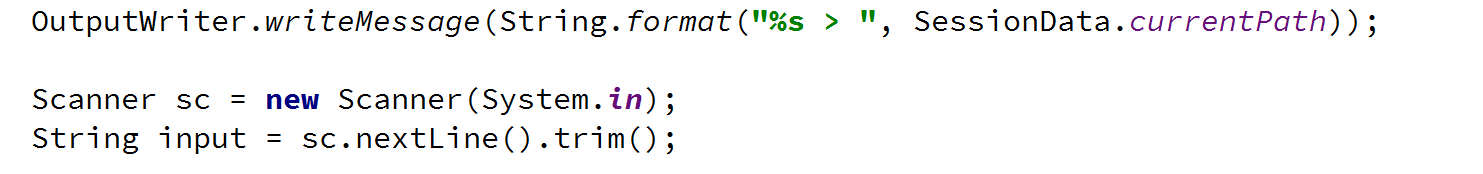


You’ve probably opened the Command Prompt and you’ve saw that you do not write you commands on empty lines, instead the folder that you are currently in is in the beginning of the line.



So in order to implement this functionality and our bash to be able to **look** **like** the **command** **prompt**, we will **write** a **message** **on** the **OutputWriter** and our message will be the **current** **path** **from** the **SessionData** class **followed** **by** **‘>**’.

Now it’s time to read an input and trim it from all white spaces.



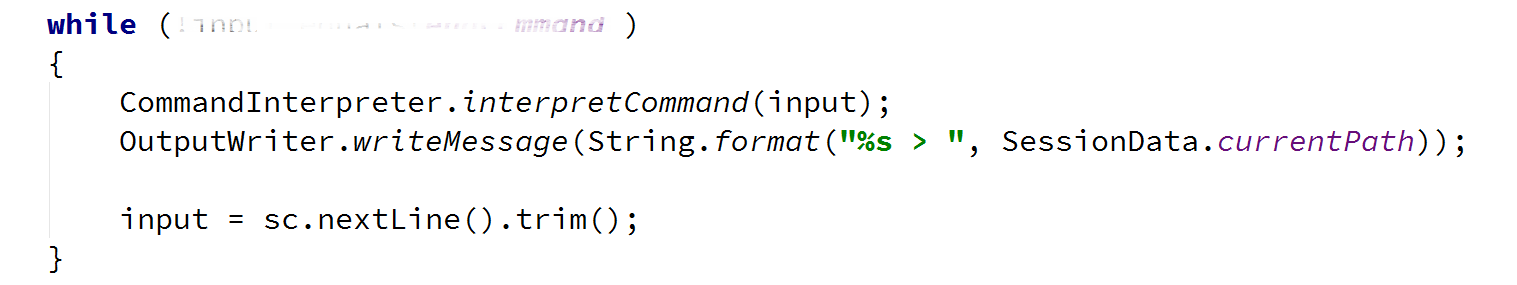
However, we want to continue reading once we’ve interpreted the current command, so maybe here will be a good time to add a while loop and read a new input at the end of the loop. Note that we repeat the code above in our while loop. But we need to do the first read out of the loop, because even the first command can be the command for terminating the BashSoft.

Now we have two things to do. First, we should set some condition for which the while loop has to be true. A good way of doing this is to make a **constant** for the command for termination (which is “quit”) and **then** **check** if the input is different from the termination command.

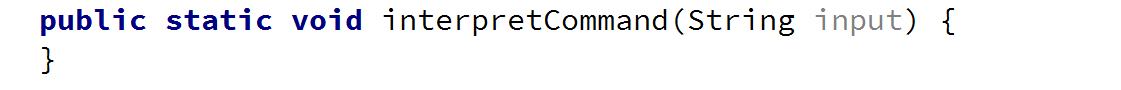
The declaration of a constant looks like this:



It is private, because we do not want other classes to be able to see it and use it. Your task now it to make the check between the end command and the input and then build the while loop:



## Interpreting Commands

It is time to move on to the interpreting of the command, but in order to change the comment with code, we have to write the functionality for interpreting a command. This functionality is somewhat a different task from reading input and for this reason we will use another class you’ve already made and write the method that interprets a command. It can be called exactly by its purpose and its declaration should be similar to this: 

In order to write an implementation for this method we need to know all the commands that our interpreter can do. The declaration of a command will be given in the following format:

**Description of the command – actual command and possible parameters**

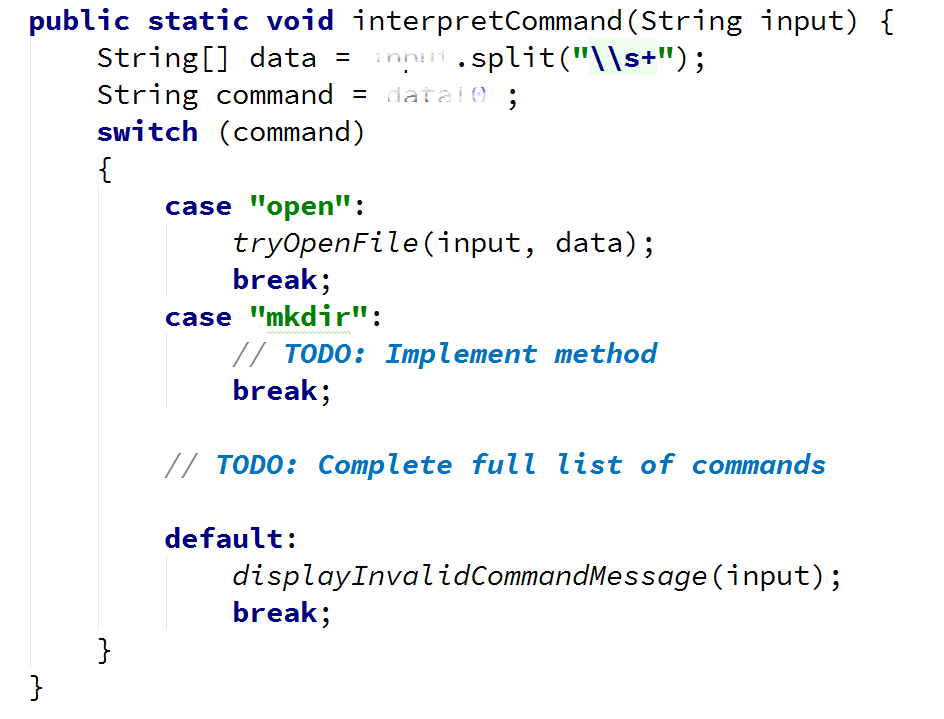
Here is a list of all of them:

Commands list:

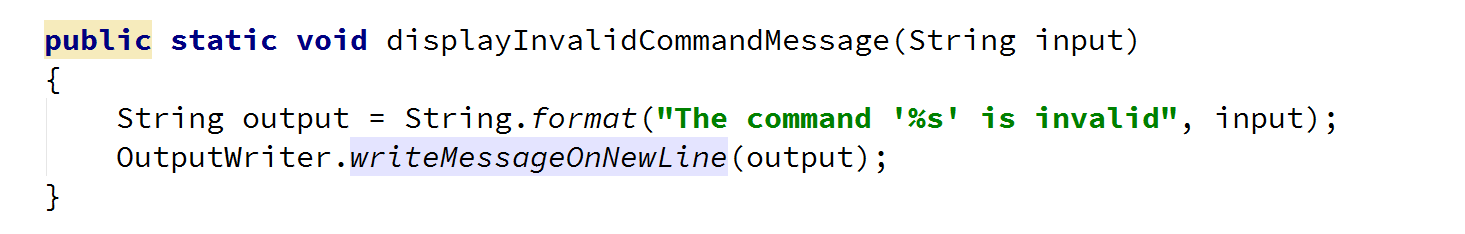
* mkdir directoryName – make directory in current directory
* ls (depth) – traverse current directory in given depth
* cmp absolutePath1 absolutePath2 – comparing two files by given two absolute paths
* changeDirRel relativePath – change current directory by a relative path
* changeDirAbs absolutePath – change current directory by an absolute path
* readDb dataBaseFileName – read students data base by given the name of the data base file which is searched in the current folder
* filter courseName poor/average/excellent take 2/10/42/all – filter students from some course by given filter and add quantity for the number of students to take, or all, if you want to take all the students matching the current filter
* order courseName ascending/descending take 3/26/52/all – order student from given course by ascending or descending order and then taking some quantity of the filter, or all that match it
* download (path of file) – download file
* downloadAsynch: (path of file) – download file asinchronously
* help – get help
* open – open file

An easy approach is to check if the input command corresponds to the ones given in the commands set. And if the given command exists, to check for the input parameters. The main check you may have to make for the input parameters in each command is whether the number of commands corresponds to the given number of parameters. So you’ll probably need this piece of code in each method for calling a given operation (data is all the commands given on the current line, split by a space):

An approach to checking whether the command is one of the possible can be achieved if we split the input by a space and check the element with index 0 in a switch-case and if it is one of the cases, we call the corresponding method, that executes the requested command. If no command matches the input, then the default is a method that displays a message for an invalid command. interpretCommand() method should look something like this:

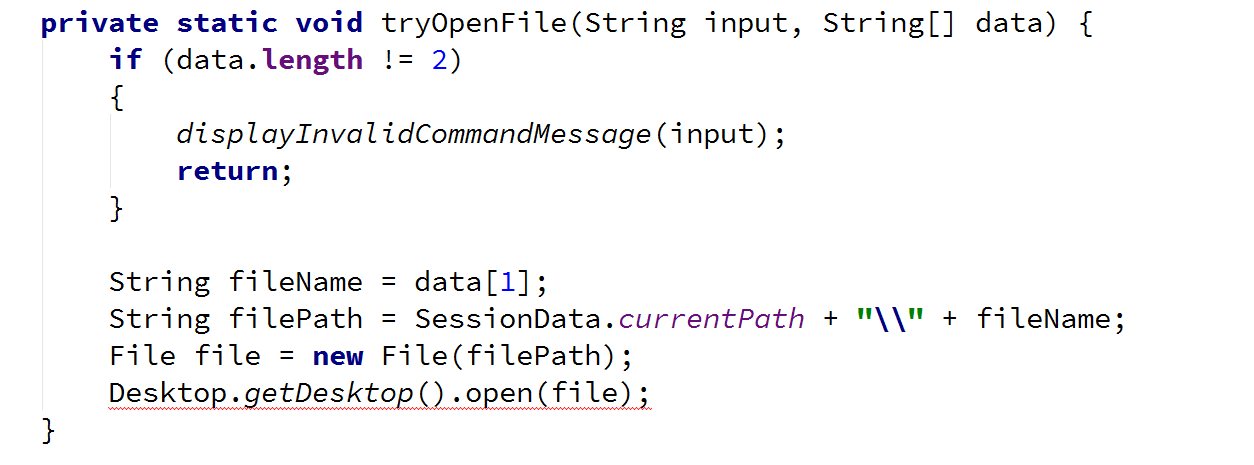


Try completing the list of commands by yourself. Don’t make methods but just add a TODO so you know that this is a task that should be done. We have a lot of methods which are not yet known and we haven’t talked about them, however almost every single one of them contains the check for the number of parameters. First we are going to look at the implementation of the method that displays an invalid command message. Actually the only thing that we do in this function is to display an exception with the OutputWriter in the following format: "The command '{input}' is invalid". If something with the commands or parameters is not ok this is the method that will be called, because it is a good way of notifying the user that something is wrong.

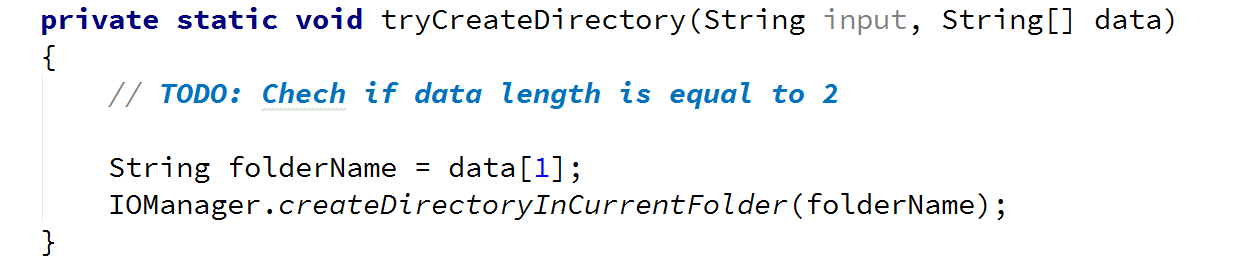


Now we have to look at the implementations of the other methods and complete them in the order they were given above.

1. **Open file** – all we need here is to know the name of the file that we have to open, and then we use the current path from the **Session Data** to generate the absolute path of the file. The length of the data must be 2 elements. Finally, we need to know how to open files with their default program, using Java this is done using the following code:

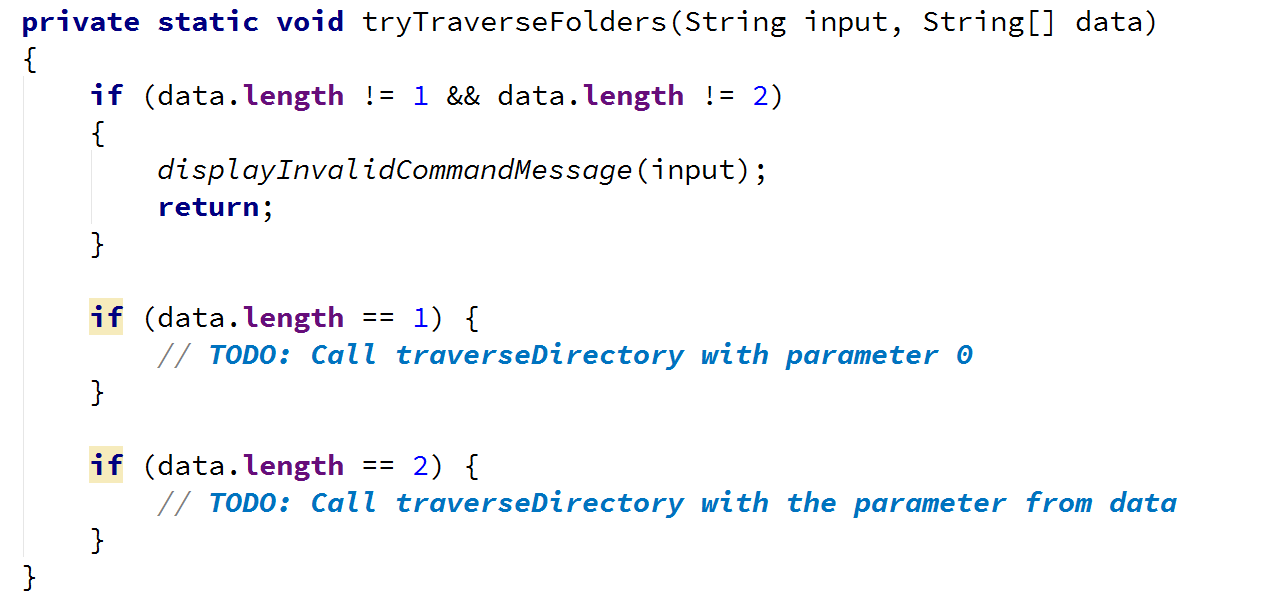


You can see that this throws an exception. Well you already know how to handle exceptions, so handle them.

1. **Make directory** – for creating a directory again we need to check if the length of the data array is 2 and then take the folder name and create such a folder using the functionality in the **IOManager**:  
   

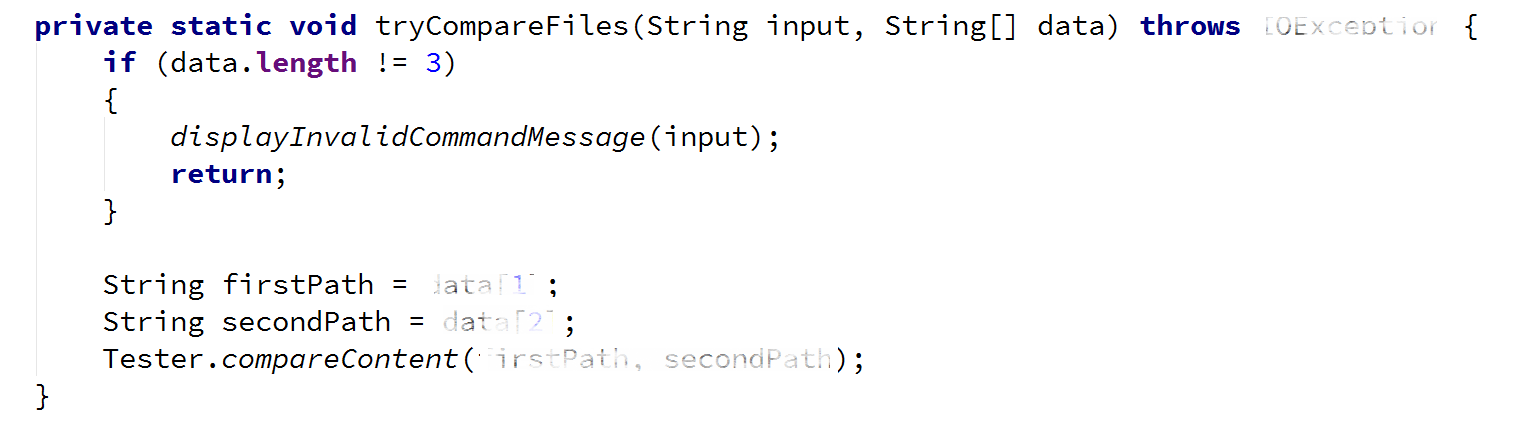
You can complete the TODO by yourself.

1. **Traverse current folder** – You can get either no parameters for this method (only ls, which displays only the files and subfolders in the current folder) or one parameter (the depth to go in, for example: ls 4). If the number of elements in the data array is only 1, we call the **traverseDirectory()** from the **IOManager** with depth 0 and if the elements are 2, then the second element should be the depth and we use it to pass it to the method for traversal. The code inside the check for whether the elements are two looks something like this:

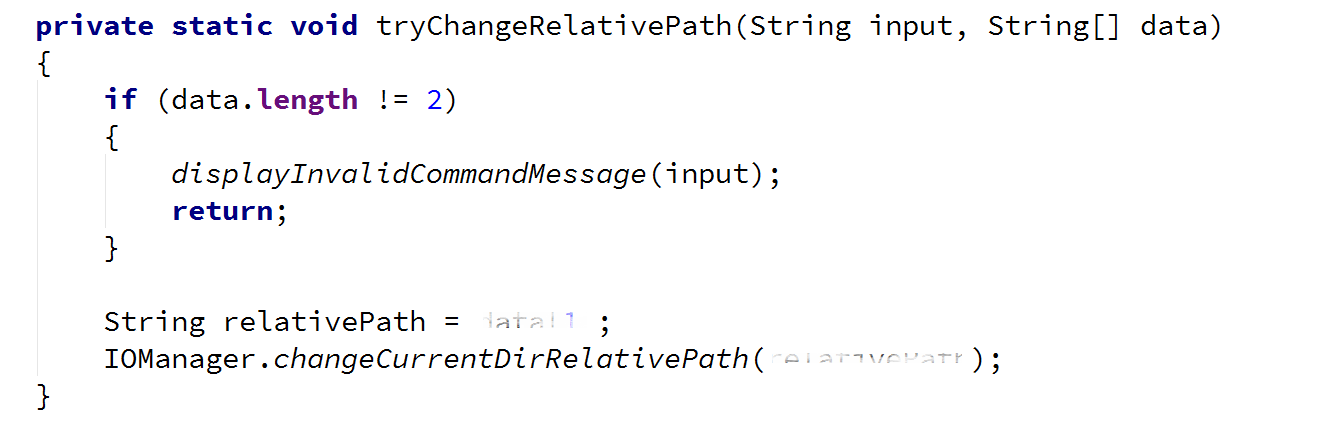


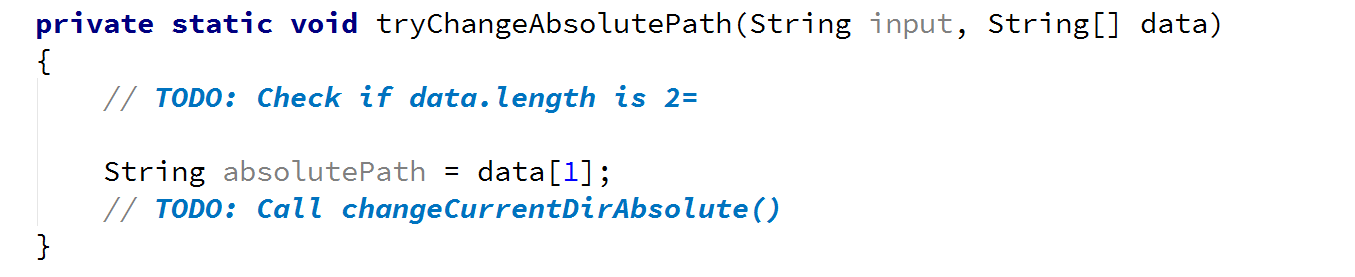
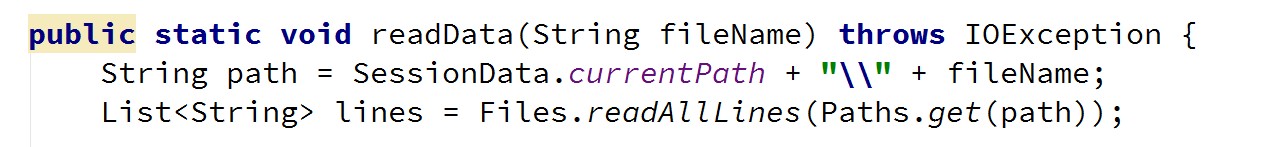
Again you should be able to complete the TODOs by yourself.

1. **Compare content of two files –** if the input corresponds to this command, two parameters are expected, which are the absolute path of the first and the absolute path of the second file. If there are any mismatches, a new log file is created in the same folder as the second file path. The way we compare two files is already implemented in the **Tester** class, so we just need to call it if all conditions are true…

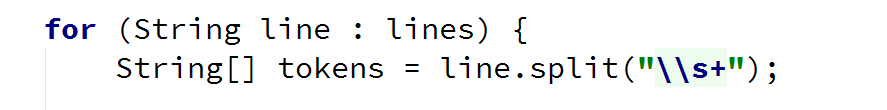
****

1. **Change directory relative –** here the path given should be appended to the current path in the **SessionData** and then it is passed to the **change directory absolute**, because an actual absolute path is generated, but we have all of this implemented in the **IOManager** so we are going to use it to change the current directory by a relative path…

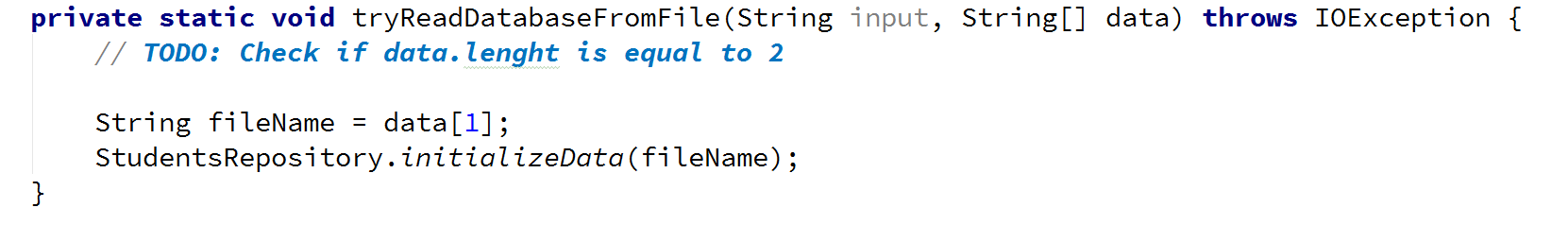
****

1. **Change directory absolute** – the approach now is pretty much the same as in the previous method. Try completing the TODOs by yourself.   
   ****
2. **Read database** – the parameter needed here for the initialization of the database is a file name from which to read the database of SoftUni. Note that only the name is needed, which means that the file will be searched in the current folder. So maybe we can use the StudentRepository and make a few changes so that our new input comes from a file and not from the console. First you want to add a parameter for the public method initializeData() from the student repo so it should look something like this :  
     
     
     
   However initializeData() is just a façade for the method that does the actual reading of the data, so we need to add the same parameter in this method and then pass the filename to the readData() call:  
     
     
     
   Now just a little change in the readData() method. First we need to get the full path to the file. After that we read all lines in the file using a helper utility class from the Java library.   
     
   

And then we just modify the while loop into a foreach loop:



If the path exists we are going to do all the processing of the input, so the body of the loop stays almost the same. If it does not, we just make the method to throw an exception which we can handle somewhere else. Now, let’s finish the readDatabase() method:



1. **Get help** – does not need any parameters. Displays some information about all of the commands, so that we can use them easily. We’ve given the whole code for the get help method in the file appended with this lecture. Use it to copy all the printing and not lose time in doing such things. The file is called getHelp.txt.
2. **For the rest of the commands –** you may leave the body unimplemented, because we do not have the functionality implemented yet.

Now we should be done with the functionality for interpreting commands and we will only extend it further on in future pieces in order to implement the full functionality of our BashSoft.

Congratulations! You’ve successfully completed the lab for Manual String Processing.