**Lab: Manual String Processing**

**Implementing the command interpreter of our Bash**

Now is probably the key moment in the application we are building. Currently our app is a stack of different functionalities that are 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 aim is to provide interpretation of these commands at runtime. So now our job is to **build** an **interpreter** **that** **calls** the **functionalities** we already have.

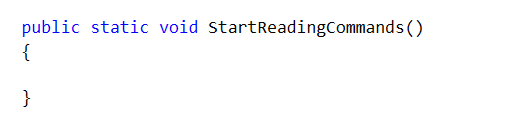
We are going to need **two** **public** **static** **classes** that **handle** the **input** and the **commands**. The **first** **one** is called **InputReader** and the **second** **one** is called **CommandInterpreter.**

Now that you have created these classes we are going to write some code so that they could get their jobs done.

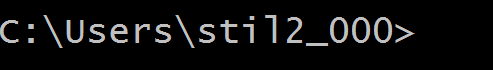
## Implement InputReader class

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

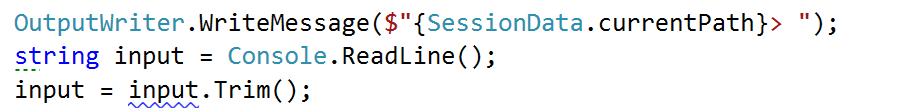


You’ve probably opened the Command Prompt before and you’ve seen that you do not write your commands on empty lines. Instead the folder that you are currently in is the beginning of the line.



In order to add this functionality so that our bash **looks** **like** the **command** **prompt**, we will **write** a **message** **on** the **OutputWriter** which 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, **once** we’ve **interpreted** the **current** **command** we want to **continue reading** the next commands 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 **do** the **first** **read** **out** **of** the **loop**, because even the first command can be the command for terminating the BashSoft.



Now we have only two things left to do in the while loop, to finish with its implementation. Firstly, 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** **in** the **condition** **of** the **loop** whether the input is different from the termination command.

The declaration of a constant looks like this:



and it is **private** because **we** **do** **not** **want** **other** **classes** **to** be able to **see** **it** **or** **use** **it**. Your task now is to **implement** the **check** **between** the **end** **command** **and** the **input**.

## Interpreting commands

Once you’ve done that it is time to move on to the **interpreting** of a **command**. Before substituting the comment with some 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 as its purpose and its declaration should be similar to this: 

However, in order **to** **write** an **implementation** **for** this **method** we need to **know** **all** the **commands** that our **interpreter** **is able to understand**.

The declaration of a command will be given in the following format:

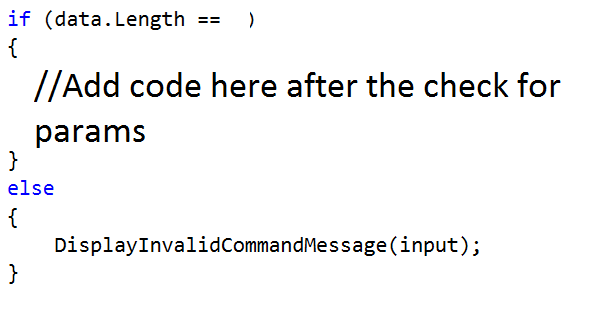
**Description of the command – actual command and possible parameters**

Here is a list of all of them:

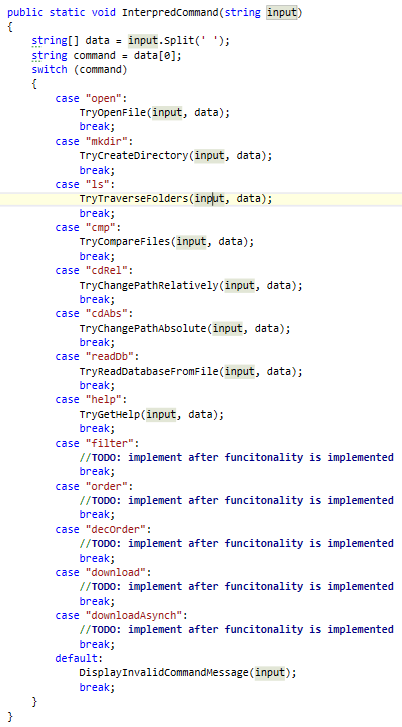
Commands list:

* **mkdir** **directoryName** – create a directory in the current directory
* **ls (depth)** – traverse the current directory to the given depth
* **cmp** **absolutePath1** **absolutePath2** – comparing two files by given two absolute paths
* **changeDirRel** **relativePath** – change the current directory by a relative path
* **changeDirAbs** **absolutePath** – change the current directory by an absolute path
* **readDb** **dataBaseFileName** – read students database by a given name of the database file which is placed in the current folder
* **filter** **courseName** **poor/average/excellent take 2/10/42/all** – filter students from а given course by a given filter option and add quantity for the number of students to take or all if you want to take all the students matching the current filter option
* **order** **courseName** **ascending/descending take 3/26/52/all** – order student from a 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 a file
* **downloadAsynch: (path of file)** – download file asinchronously
* **help** – get help
* **open** – opens a 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 primary check you may need to perform over the input parameters in each command could be whether the **number** **of** **parameters** **corresponds** **to** the **number** **of** **parameters required by the respective command**. So you’ll probably need this piece of code in each method for calling the given command (data is all the parameters given on the current line, split by a space):



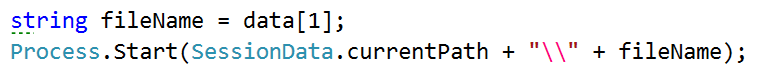
An approach to **check** **whether** the **command** **is** one of the **possible** can be achieved if we **split** the **input** by a **space** and **check** the **element** with index **0** in a **switch**-**case.** **If** it **enters** one of the cases, we **call** the **corresponding** **method**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

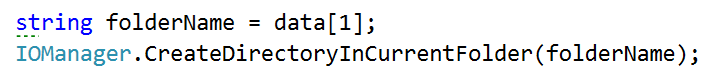
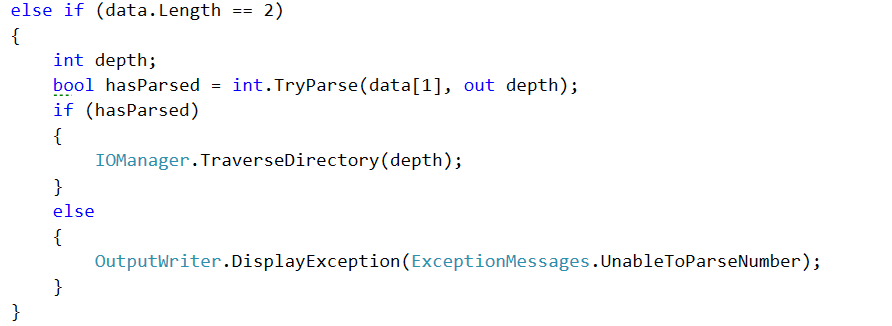
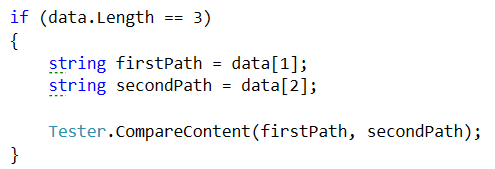


In all the cases we have a lot of methods that we call which are not yet known and we haven’t talked about. However almost **every** **single** **one** of them **contains** the **check** **for** the **number** of **parameters**. First we are going to **look** **at** the **implementation** **of** the **method** that **displays** an **invalid** **command** **message**. Actually the only thing that we do **in** this **function** is to **display** an **exception** in the **following** **format**: $"**The command '{input}' is invalid**” (***Display******exception******using*** *the* ***OutputWriter***). We are going to call this method every time when something with the commands or parameters is not ok and notify the user that something went wrong.

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

1. **Open file** – all we need to know here is 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 C# and this is done using the following code:



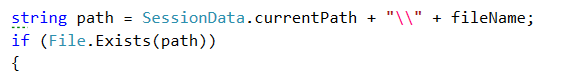
1. **Make directory** – when making a directory, again we need to check if the length of the data array equals 2 and then take the folder name and create such a folder using the functionality in the **IOManager**:  
   
2. **Traverse current folder** – here it is not necessary to have any parameters (only **ls** will display **the files and subfolders in the current folder**) or you can have just one parameter (the depth to go in [ls 4]). If the number of elements in the data array is 1, we call the **TraverseDirectory** from the **IOManager** with depth of 0 and if the elements are 2, then the second element should be the depth and we try to parse it. In case of success pass it to the method for traversal. **If the parameter can’t be parsed**, we print an exception message on the output writer using its method Display exception. We should first add the exception we talked about to the ExceptionsMessages class with the name UnableToParseNumber and a message: “The sequence you've written is not a valid number.” The code inside the check for whether the elements are two looks something like this:   
     
   
3. **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** and 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:  
   ****
4. **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…  
   ****
5. **Change directory absolute** – the approach now is pretty much the same as in the previous command.   
   ****
6. **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 wanted, which should mean 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 thing you might want to add is a parameter for the public method **InitializeData()** from the student repo so it should look something like this :  
  
public static void InitializeData(string fileName)

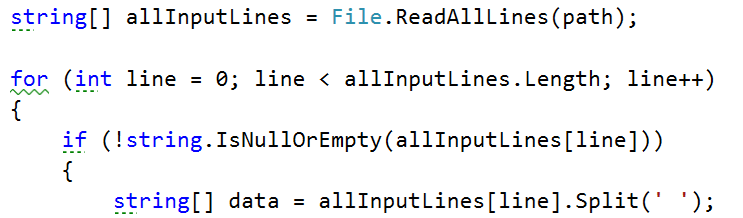
However **InitializeData** is just a **façade** **for** the **method** **that** **does** the **actual** **reading** of the data, **so** we need to **add** the **same** **parameter** **in** this **method** **and** then **pass** the **filename** **to** the **ReadData** call:

private static void ReadData(string fileName)

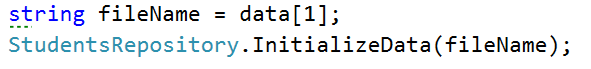
Now it’s time for only a little change in the read data method. First we need to remove the while loop and all the places where we read from the console and finally the input variable. After that you can make a new variable to generate the absolute path and instead checking the Direcotry we will check if the file exists.



If the path exists we are going to do all the processing of the input, so you may **copy all the code that was in the while loop and paste it in the body of the if statement**. Now that you know that there is such a file, you may read it. And after that **wrap everything that was in the while loop in a for loop**, **iterating through all the lines of the file and processing them one by one**. Your code in the if should begin with something like this:



If the path does not exist however, an exception with the name InvalidPath from the ExceptionsMessages is displayed on the OutputWriter. Now that we’ve done all these changes, we can easily call the method from the command interpreter like this.

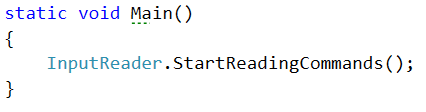


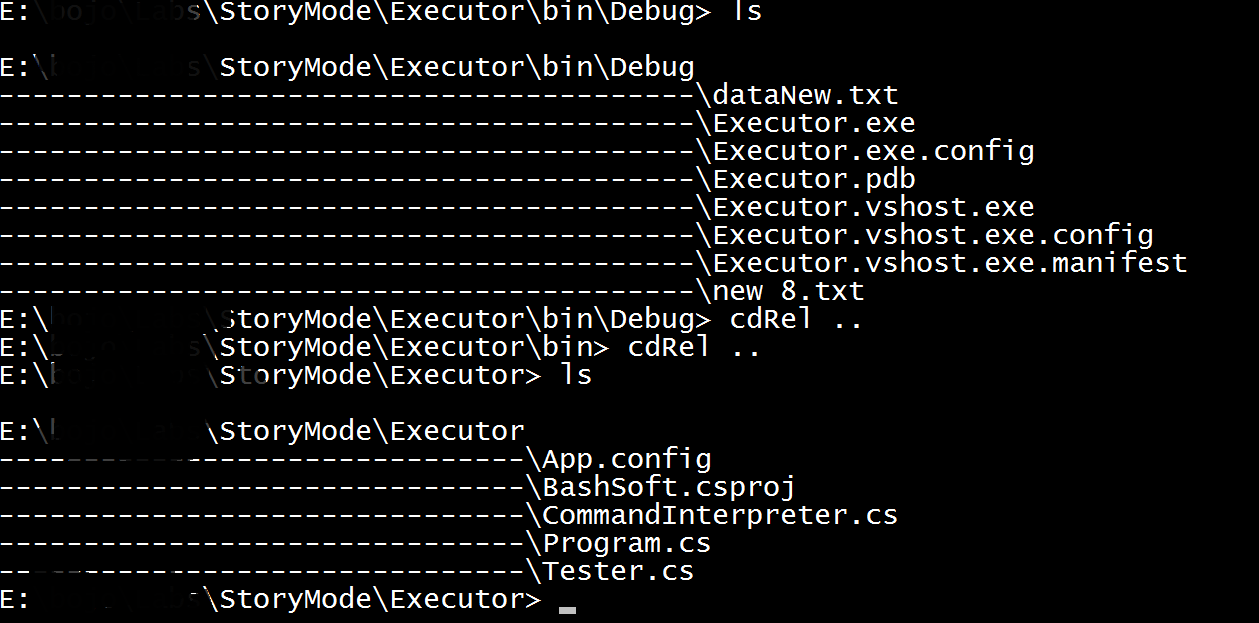
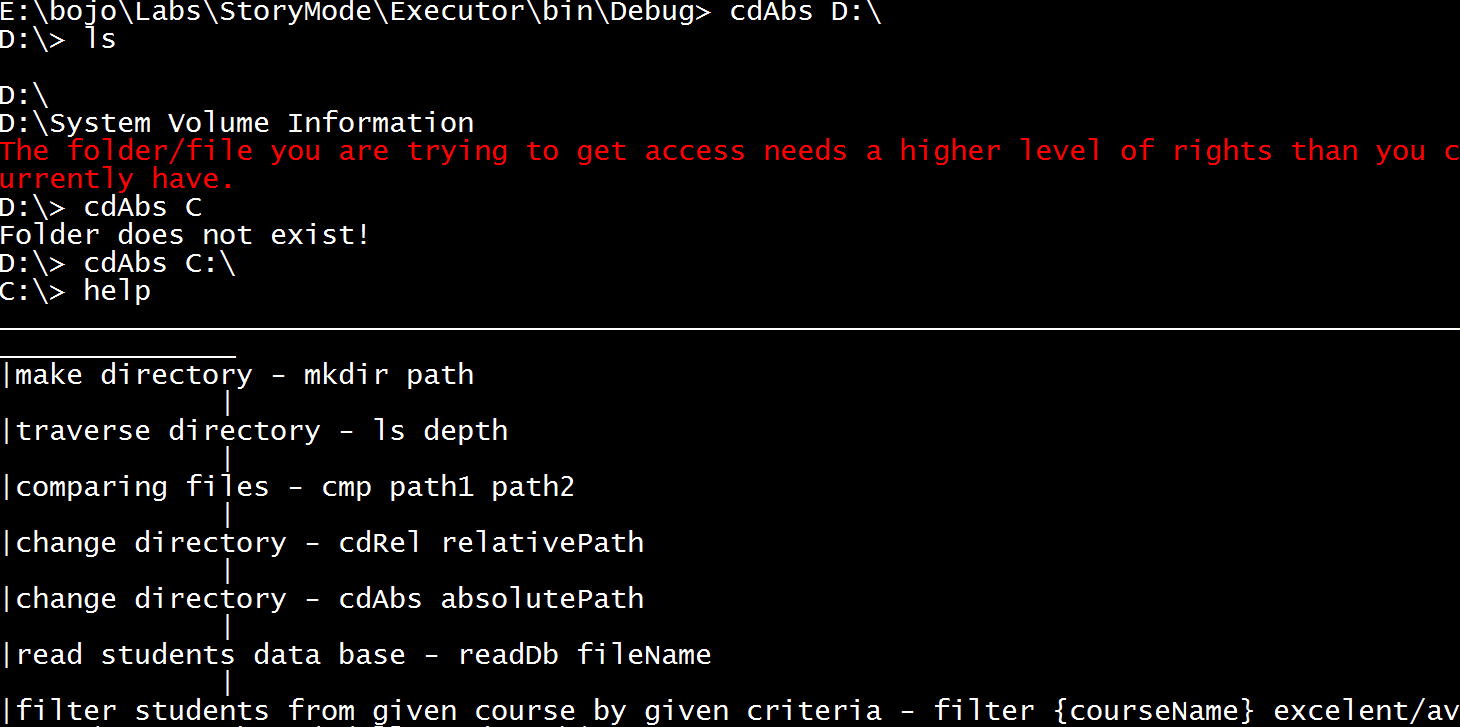
1. **Get help** – does not need any parameters. Displays some information about all of the commands, so that we can use them easily. We’ve given the whole code for the get help method in the file appended with this lecture. Use it to copy all the printing and not lose time in doing such things. The file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

So now that we’ve written the functionality for the command interpreter, we can link it to the **InputReader** and we should be finally done. All we have to do is to go back to the input reader and **change** the **comment** **for** **interpreting** the command **with** the **method** that interprets a command from the command interpreter.

CommandInterpreter.InterpredCommand(input);

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. And we should also be ready with the whole piece. The only thing left is to call the StartReadingCommands from the main method, and test all the functionality that we have by now. We’ll leave the part with the testing to you, but we’ll show a few pictures of the current state of the program:



In the next piece we are going to learn how to make more restricted, pattern following data and

filter it easily.

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