**BashSoft**

**Creating the Basic Functionality**

**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 from the rest 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” which require us to use “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 Output Interface**

We have our first class and we are going to implement some functionality that it needs. But first we have to decide how we are 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 doing only a few changes in one place. The solution behind this problem can 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 output to the console, which can easily be swapped for writing to 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 create it in a similar fashion to the IOManager. The new class again has to be public and static and after you’ve created it, it should look something like this:



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 four 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 back. 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().

**Traverse 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 a 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 an 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 same 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. 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.

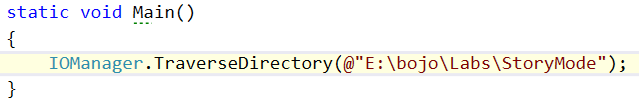
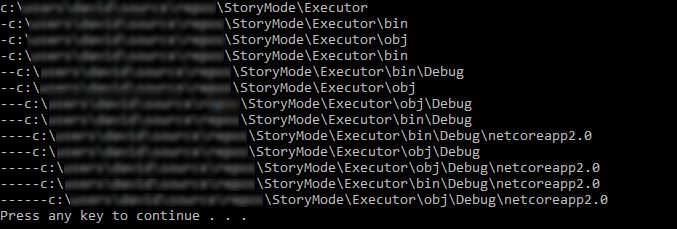


Also, for each folder we need to iterate all its subfolders and add them to the end of the queue. We can do this with a simple foreach loop:



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



Now, 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.

**Create a Set of Error Messages**

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 these often-used messages.  
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 its corresponding message:  
  


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

**Creating the Data Structure**

**Create a Data Structure**

Our next task is to create a fast and efficient data structure that we can use in out command interpreter to store data, easily make changes, find wanted information or generate some statistics from the data.

First thing you have to do is to set up a class in which you will store your data. You have to create a new class, following the steps from the previous piece of the story. This class will be called “StudentsRepository” and has to be static and public. By now you should be somewhere around here:



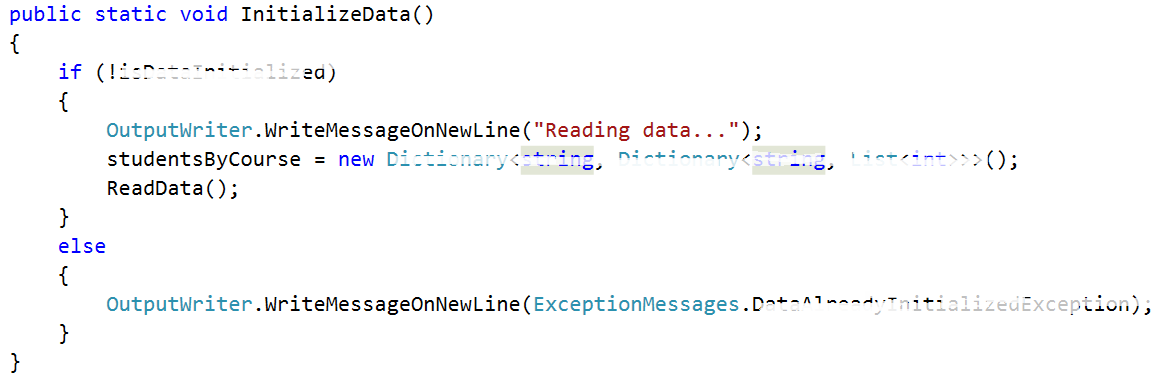
Now it is time to decide what data structure to define for our application in order to be able to make fast operations and have easy access to your data. Since we have to save different courses, the students in those courses have unique usernames and a list of grades, we can save them in two nested dictionaries with one additional list. See below:


We will also add a public boolean flag for whether the data structure we want to have has been initialized. You may have noticed but we’ve put private in front of our data structure and that is because we do not want everybody outside of this class to see our data structure and change it, so by making it private we can only see it in the current class and we will allow some of the data searching and filtration throughout public methods that give to the other classes the basic operations needed over the SoftUni system’s data.

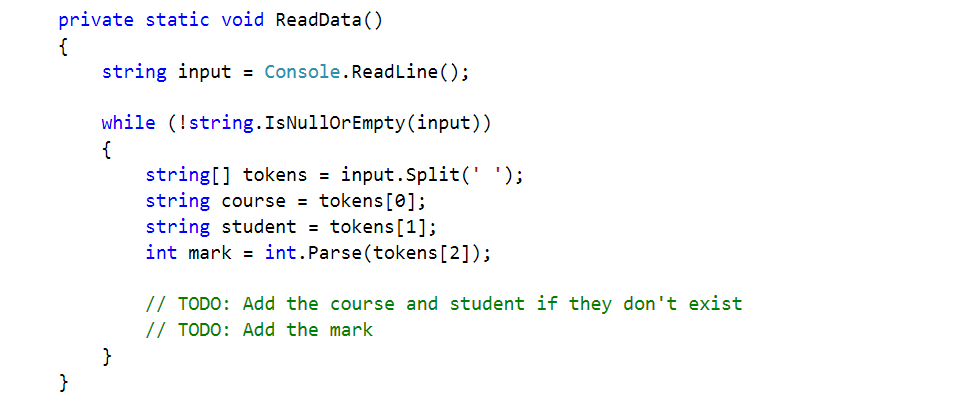
**Initializing and Saving Data**

In order to complete our task, we need to initialize our data structure and fill it, so we will make a new method that initializes the data structure. If it is not initialized yet, reads the data, if it is, we display a new message called DataAlreadyInitializedException that we need to add first in the ExceptionMessages class. Its message should be: “Data is already initialized!”  
The implementation of the method for the initialization should look like this:

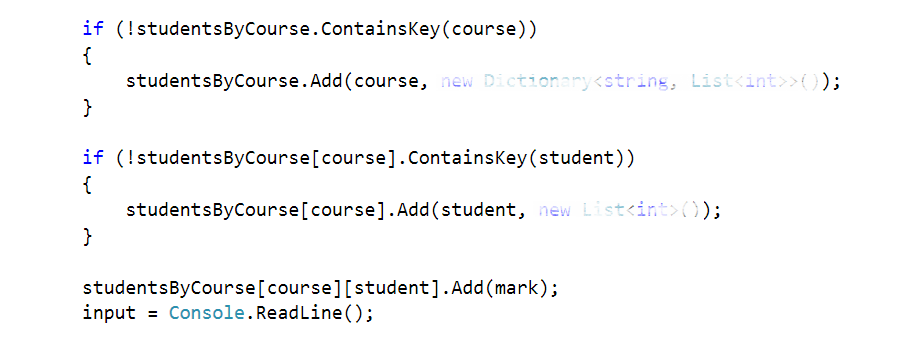


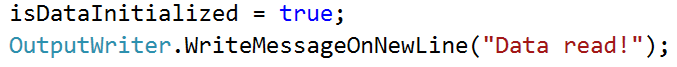
Now it’s time to fill the private ReadData method (the data will always be valid). It is private because we do not want it to be reachable out of our class.

All we are going to do, is to read from the console until an empty line is read. The data you need to read is in the data.txt file given with the current document. We also need to extract the information we need from the input and save it in our data structure.



Now we need to check if our course and student exists in our data. If we don’t do this, we are sure to get an exception. So, if the course doesn’t exist we must initialize the inner dictionary holding the students for the given course. Also, if the student doesn’t exist we have to initialize the inner list with grades. Finally, we add the mark.

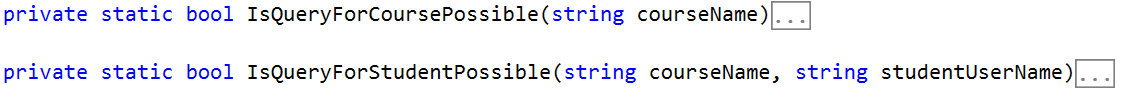


Finally, after the while loop we need to set the isDataInitialized to true and print “Data read!” on a new line!  


**Security Checks**

Since we are going to make queries to the data structure, it would be a good idea to make a method for the security checks in order to retrieve some data for a given course or a given student. This way we will save ourselves the writing of checks each time and invoke the methods where such a check is needed.

So, the first method will be called IsQueryForCoursePossible and the second will be called IsQueryForStudentPossiblе. Both should be private and static and as you might guess their return type is bool. The first one take one parameter (the course name) and the second one takes two parameters (the course name and the username of the student). Their definition should look like the following:



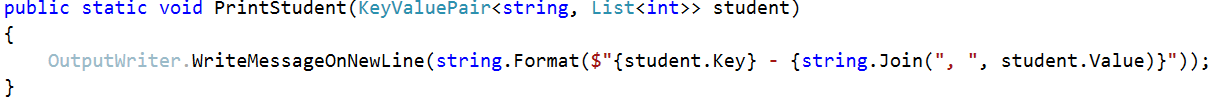
Since the second method will have to do half of the checks for the course that are done in the first method we will reuse the first one and for this reason we are starting with its implementation.

First thing we need to check in order to search for the given course name, is whether the data structure is actually initialized. If it hasn’t been initialized we create a new message in the ExceptionsMessages that is called DataNotInitializedExceptionMessage and its message is:  
"The data structure must be initialized first in order to make any operations with it.":

  
We are now returning true if the data structure has been initialized, but we haven’t checked whether the given courseName exists as a key in the data structure.   
So now we have to add this check in the body of the if and if the data structure contains the key, we return true while in the other case we display an exception that we’ll need to add in the ExceptionsMessages called InexistingCourseInDataBase with the following message:  
"The course you are trying to get does not exist in the data base!":  
  
  
Now that we’ve implemented the first method for the checks, it’s time for its sidekick. As we’ve said we will reuse the check from the first method and also add a check for whether the given student user name exists in the data structure of the university. If it is present, we return true, if it is not we display an exception that we’ll need to add in the ExceptionsMessages called InexistingStudentInDataBase with the following message:  
"The user name for the student you are trying to get does not exist!"  
and finally we return false:  
Now that we are ready with the security checks we can proceed with the next step.

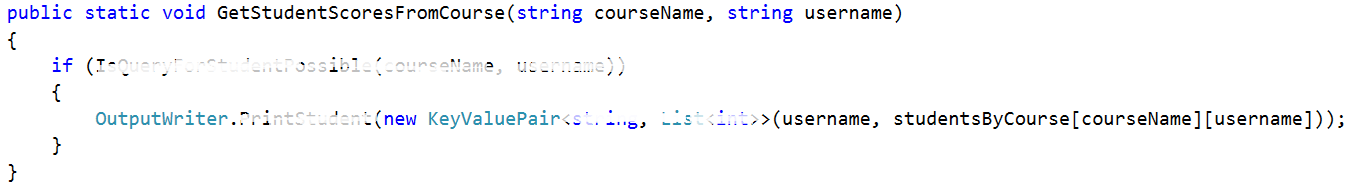
**Display a Student Entry**

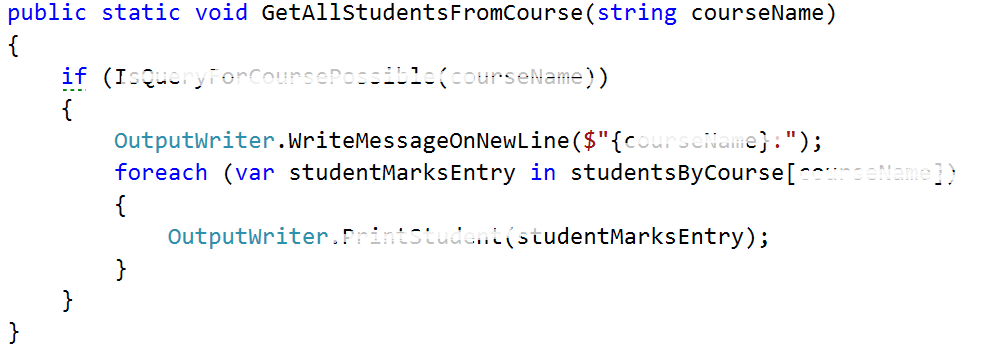
Before we continue with the reading of the data, there is just one last thing we might add in order to make our life easier. Since now we have two methods that are going to display a student somehow and we might have more things that need to display a student after a filter or a sorting for example, by implementing such a method we do not need to write formatting strings in every method that displays students on the output writer. The given method will be called DisplayStudent receiving a KeyValuePair of a string (user name) and a value: List<int> (scores on tasks). A good place to put the print student method may be the Student repository, but maybe an even better place is in the output writer since it implements the logic for how thing are displayed on the standard output. The implementation of the method should be as follows:

  
Now that we are ready with the display of a student we can proceed with the actual reading of data from the data structure.

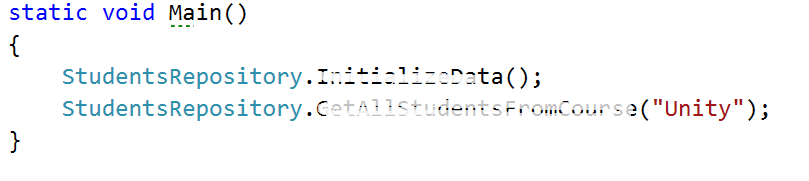
**Reading Information From our Data**

The most basic operations for extracting information will be to get all students from a given course and get all the scores on the tasks. We need define two methods. Let’s start with the first one. It should be public static with return type void. It’s parameters are the course name and the user name of the student. So if the query for the given student is possible, we need to print him on the output and so we give a new student to the Output writer in order to be printed:



The other method is analogical. It gets all students from a given course if the query for course is possible.  
First we write the course name followed by two dots and after that we foreach the collection with students from the given course and print all of the students.  
  


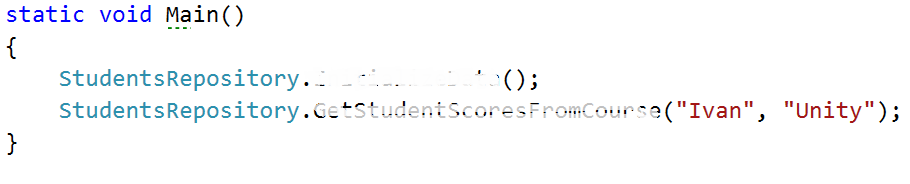
**Test your code**

If you put the given input and get all the students from the Unity course(query should look like this): `

And the result should look like this:



Now we want to test the functionality for getting student’s grades from a given course. The request should look something like this:



And the result, something like this:



Now we are ready with the current piece and now we can easily keep track of the courses and students inside them and if needed, view some data that we might want. Soon we will learn how to make filters and sort our data so that it is in a more accurate format and moreover we will go into depth about the constraints for the possible course names, user names and scores on a given task.

**DIY Judge System**

**Idea overview**

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.

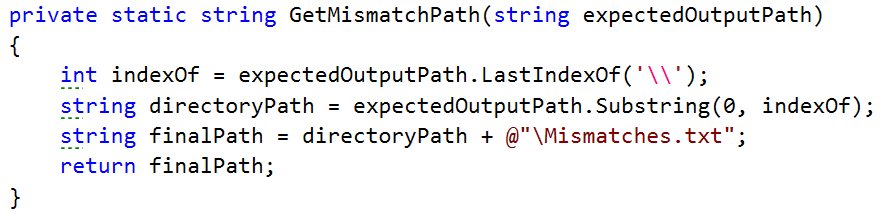
**Set up our Tester Class**

Create a new Visual Project Solution and a new Console Application called “SimpleJudge”. In the SimpleJudge project add a new class called “Tester”. Mark it as public static class and declare a new public static void method called CompareContent(string userOutputPath, string expectedOutputPath):

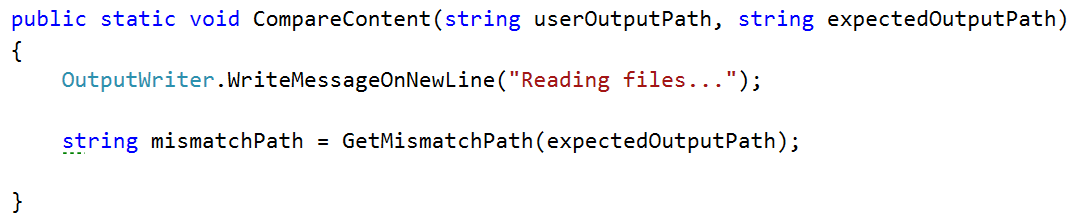


The idea here is that using userOutputPath and expectedOutputPath we can find the files holding the user output and the expected output respectively, read the user output and the expected output and compare them line by line to see if they are identical.

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 efficiently we can use the expectedOutputPath 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 achieve this by creating a helper method:

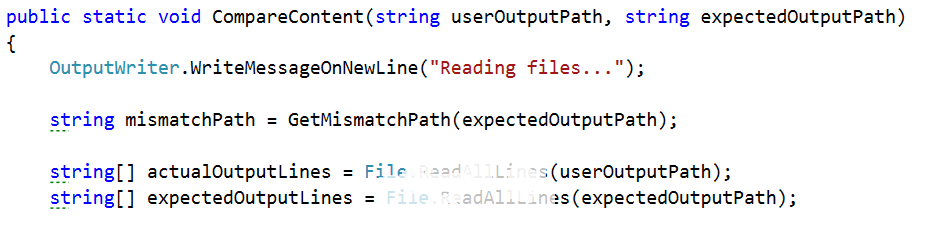


What this method does is simply get the path to the directory of the expected output file by finding the index of the last ‘\’ in the path of the expected output file. For example if the path is C:\OutputFiles\OddLinesExpectedOutput.txt we find the index of the second ‘\’ (14 in our case) then we simply get a substring of that path up until that index and we end up with “C:\OutputFiles” which is the path to the directory of the output file. Then we finally append the name of our file and a slash “\Mismatches.txt” and we finally end up with a path looking something like this “C:\OutputFiles\Mismatches.txt”. You might wonder how come we use a path to a file that does not currently exist. We’ll get to that in a moment, but first let’s call out helper method up in our main method.



**Read and create files**

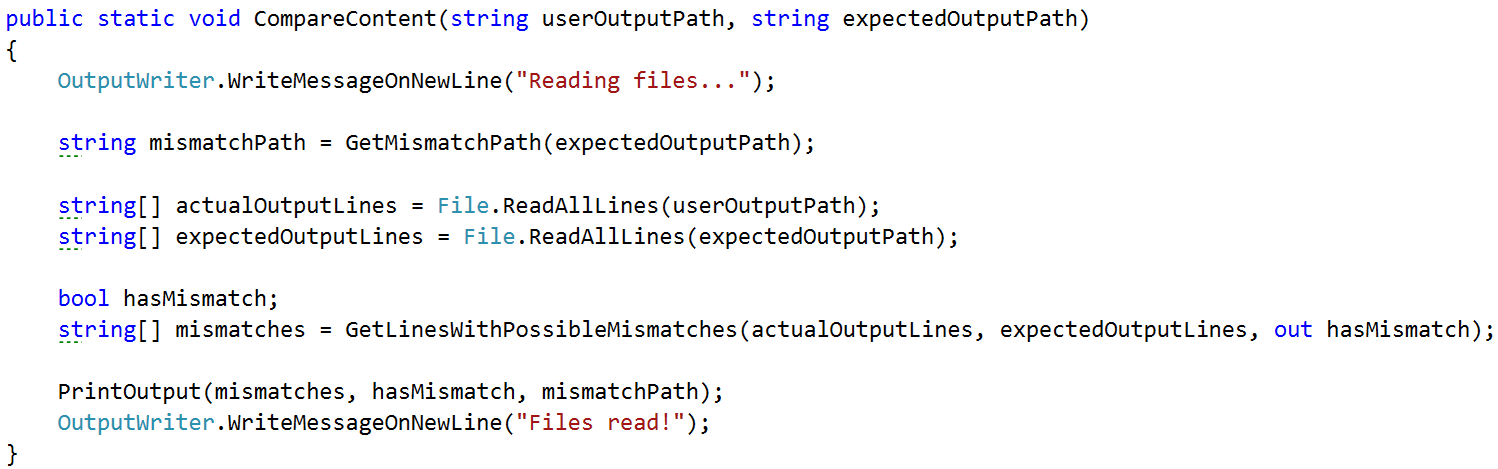
Next up we need to read the two files – the user output and the expected output. This is done again in just one line of code. We call the File class and invoke the .ReadAllLines(string path) method. However, this time around we need a variable in which we can actually store the contents of the files we read from. The File.ReadAllLines(string path) function returns a string array so our code will look something like this:



We end up with a variable input which holds all the user output, read from the user output text file line by line, and a variable called expectedOutput which holds the… expected output, again read from the expected output text file line by line. We are ready to start the comparison of the two files. The information we will need while comparing the files is whether there are any mismatches and also the result of the comparison of two corresponding lines. So we can make one Boolean for the mismatches and one string array called mismatches which gets it’s value from the method GetLineWithPossibleMismatches with the three parameters shown in the picture below:  
  
  
  
 We’ll get to the implementation of this method in a moment. First we need to finish the CompareContent method so that we can focus our attention on the other functionality waiting to be written.   
The last thing we can do after all the checks for mismatches is to write them on the set output writer and in the mismatches.txt file which is in the same folder as the first file given for comparison and that is done by the PrintOutput method. And finally print on the output writer that the files are read:

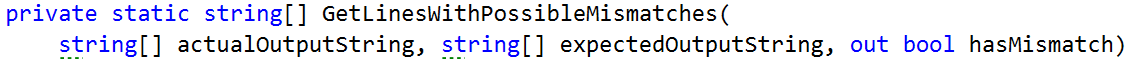
  
As you can see the method for printing the output of the comparison takes 3 parameters, which are related to the possible mismatches. We will discuss the implementation of this method after the previous one, so it is last on the queue now.

Finally the CompareContent method should look like something pretty similar to this:



**Find mismatches of two files line by line**

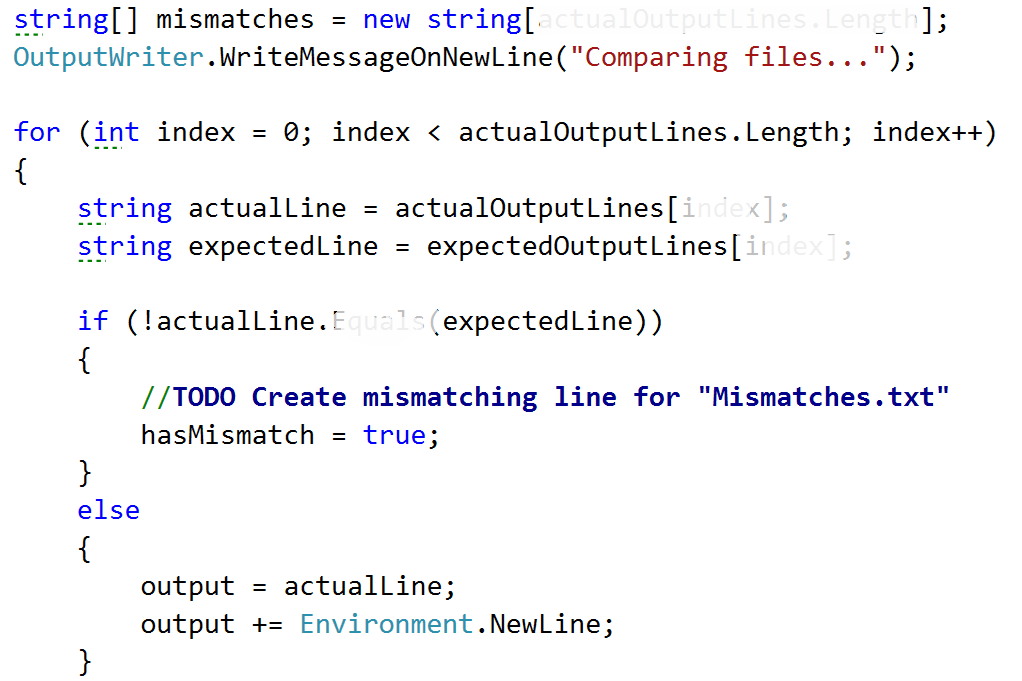
Since we are going to compare two files, and that is a separate task, we will use a separate method to do so. It’s called GetLinesWithPossibleMismatches and takes three parameters which are the strings array from the first file, the string array from the second file and an out parameter for whether there are any mismatches, so that the following method can change a variable outside of it’s scope. The method returns a new string array which represents the result after the comparison of each line.



Before we start the actual comparison and matching it’d be a good idea to declare one helper variable which will come into play a bit later. A string that has an initial value of an empty string and is later used for the line by line comparison of the two output files that are given for comparison. Another thing we might want to set is the hasMismatch variable to false and only if on some place two lines are found with a difference between them, the hasMismatch variable is set to true and the one that is outside of the method will also be set to true.



Now that we have that sorted out we can safely get to the actual comparison. How do we go about that? Well we will need a few things in order to do effective comparison and write down our mismatches. In order to compare the lines we can simply run a single for loop which iterates through both user generated output and the expected output comparing each line at every iteration and writes the result of each comparison in a new string array called mismatches which we create in after creating the two variable above.

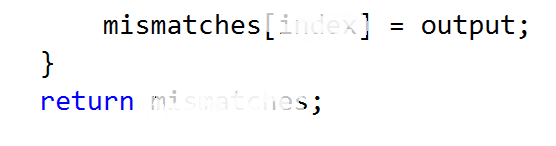
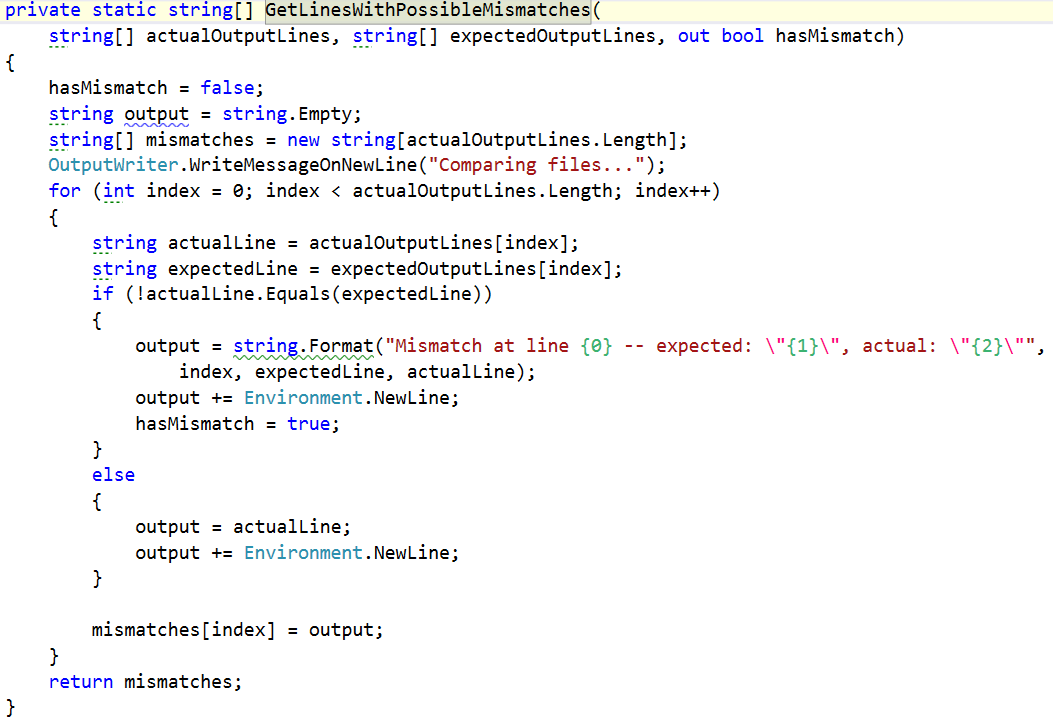


What’s going on here is pretty straightforward. We simply iterate over all the lines from both the files by assigning the current line to the actual line variable and the expected line to the expectedItem and comparing them. If they are not matching we mark mismatch as true, and we will set the output to the following message:

C:\Users\david\Desktop\stringFormat.png

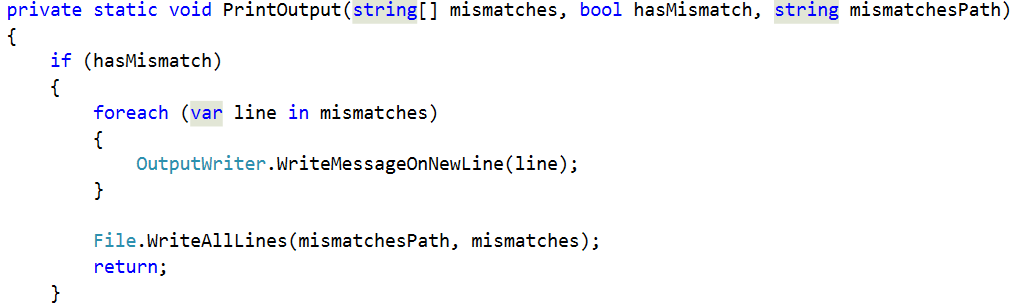
And after that append a new line like shown in the else clause in the code above.

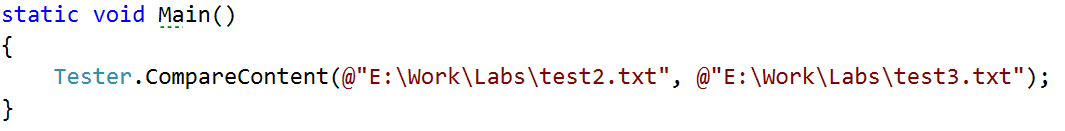
If however we don’t get a mismatch, if the lines are identical then we simply write down the line in Mismatches.txt. Why? Well because if we get an eventual mismatch down the line or if we’ve gotten one already, it’ll be easier to see where it occurred if we also have the rest of the text written down. Finally, on each iteration you put the output in the corresponding cell in the mismatches array and after the for loop we should return the mismatches array and now we are sure to have the mismatches and also the hasMismatch variable to be changed to the corresponding value, because it’s an out parameter.

  
Here is a final version of the GetLinesWithPossibleMismatches method:  
  


**Printing the data from the comparison and created mismatch file**

We’ve gotten to the point where we need to implement the PrintOutput method. It has 3 parameters in it’s signature. The first one is the array that we just generated with the mismatches from the previous method. The second parameter is whether there are any mismatches and the third one is the path to the mismatches file. All we have to do is write all the lines from the mismatches on the output writer if there has a mismatch, append all the lines to the mismatch file using the given path and return so that we exit the method. If the hasMismatch is not true, we do not enter in the body of the if statement and all we do is write a message on a new line which is the following: “Files are identical. There are no mismatches.”   
Here is a how the implementation of what we just described above, should look:



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.  
  


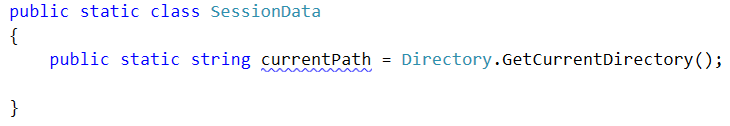


**Saving some data for our current session**

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

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 out 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 “resources” folder and read the Database from a file and not from the console…

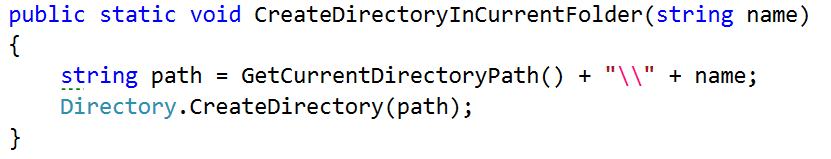
We are going to go through each of these steps in big details so you would be able to understand how each component works.

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

Making directories

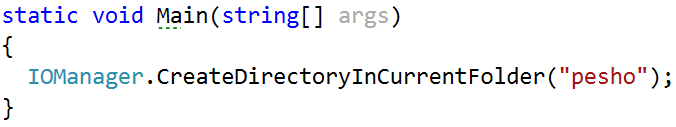
First we are going to stop in the IOManager and make a method for making a directory. Since we have our currentPath in the SessionData class all we need is the name of the folder we are going to create.

Our method can be called CreateDirectoryInCurrentFolder (string <the name of the folder>) and it’s implementation should look like this.

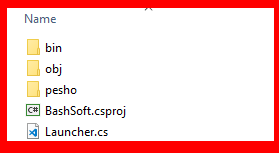


We use the method given from the Directory class, which takes an absolute route and creates new directory.

So now if we call it from the main method like this



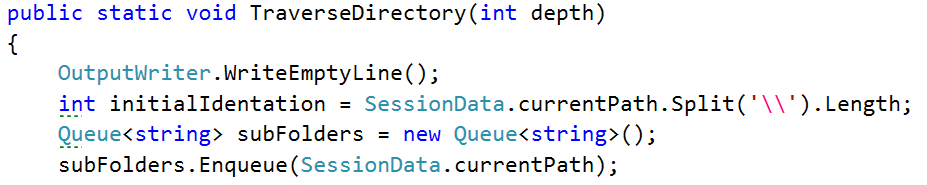
And since the folder that our application is currently running in the main directory of the project, there a folder with a name “pesho” should be added.



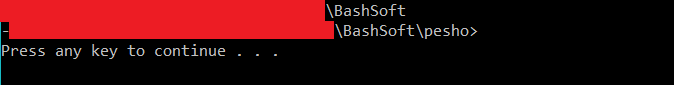
**Modifying the traversal**

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 remove the string path argument and also change it with Session.currentPath

Your traverse method should now start like this:

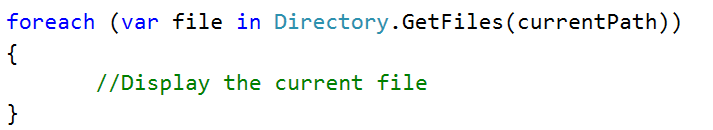


Try testing the functionality now!

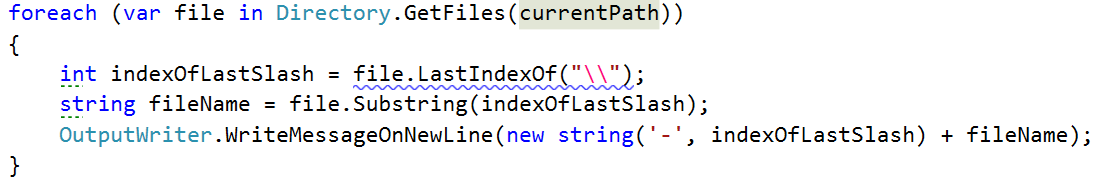


If this is your result you’ve done your job well.

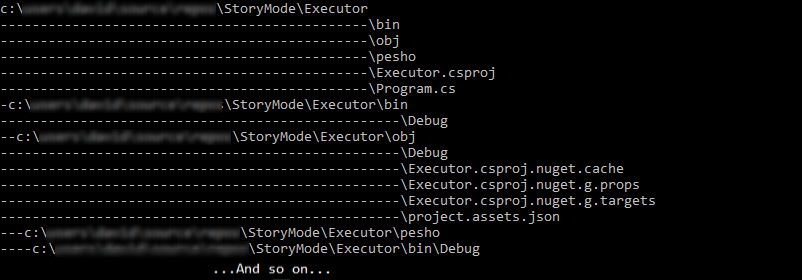
Another thing we might want to add to the current implementation of the traversal, the display of the files in the current folder. It is pretty similar to the adding of the subfolders. All we need is a foreach loop and to use the Directory.GetFiles(path) to get all the files and display them. The display of the files should be between the display of the current path and the adding of the subfolders.



In order to display the file, we will change the path to the given file with dashes, because we can see the folder we are in on the line before the display of the files and this way we can focus on the file names.  
To get the whole path, we will get the index of the last ‘\’ (backslash) and print a string with such a length of dashes, followed by the file name like shown below:



And the output of the traversal with depth 0 should now be similar to the given:



There is just one final piece of code we need to add to this method and it should be tip top. Bearing in mind that we added a parameter for the depth of the traversal, maybe we should include it as some kind of a condition in our code so that it would be easier to know when to stop traversing if we’ve gone deep enough and in order to check how deep we’ve gone, we can use the indentation variable that gives us exactly this. So after the assigning of a value to this variable, you can add the following line of code:  
  
 

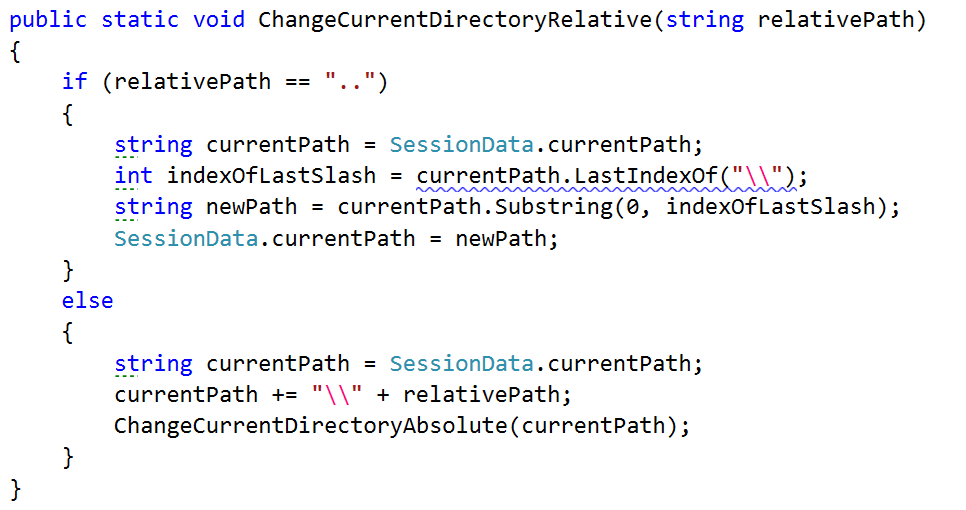
This way we are sure to stop the traversal before we print the current folder, if we’ve gone deep enough.

Now the 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**

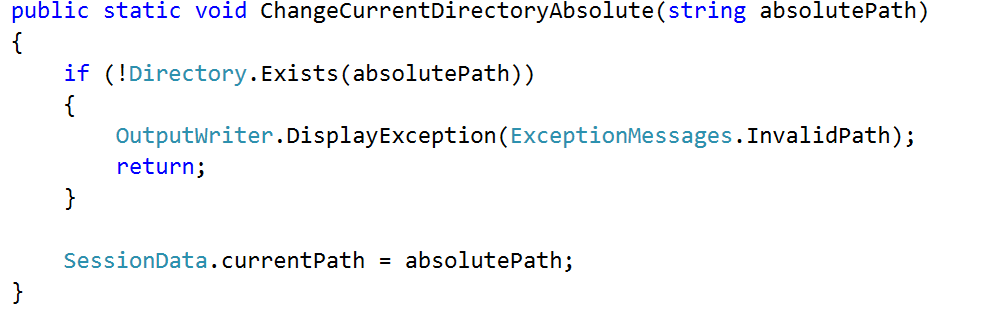
So again using the IOManager, we make two functions. One that moves forwards and backwards in the folders and one that gets an absolute path and goes directly there(Note: there are many machine specific things in the path if it is absolute).

In the relative change of folder method, we may won’t 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 by using the method change for absolute path, we can reuse the check whether the given path exists or not.



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 Directory 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 InvalidPath containing the following text: “The folder/file you are trying to access at the current address, does not exist.” and after that return so that it can exit the method. If the device has a folder with such a path, it is set to the currentPath at the end of the method.



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.

**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:

We haven’t got so much code, in order to have many error prone places.

We are taking safety precautions and check much of the input information, so that such unexpected events can’t happen.

So let’s get started with filling out some holes in our application.

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.  
  

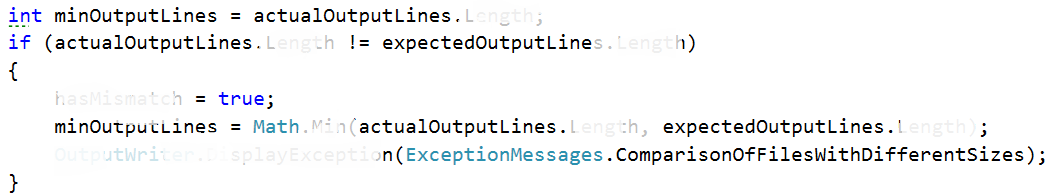

This type of exception message is not yet in the ExceptionMessages, so you should add it and put the following message: “The folder/file you are trying to get access needs a higher level of rights than you currently have.”  
  
Now the possible problems with the traversal are solved. And we can proceed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

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.



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

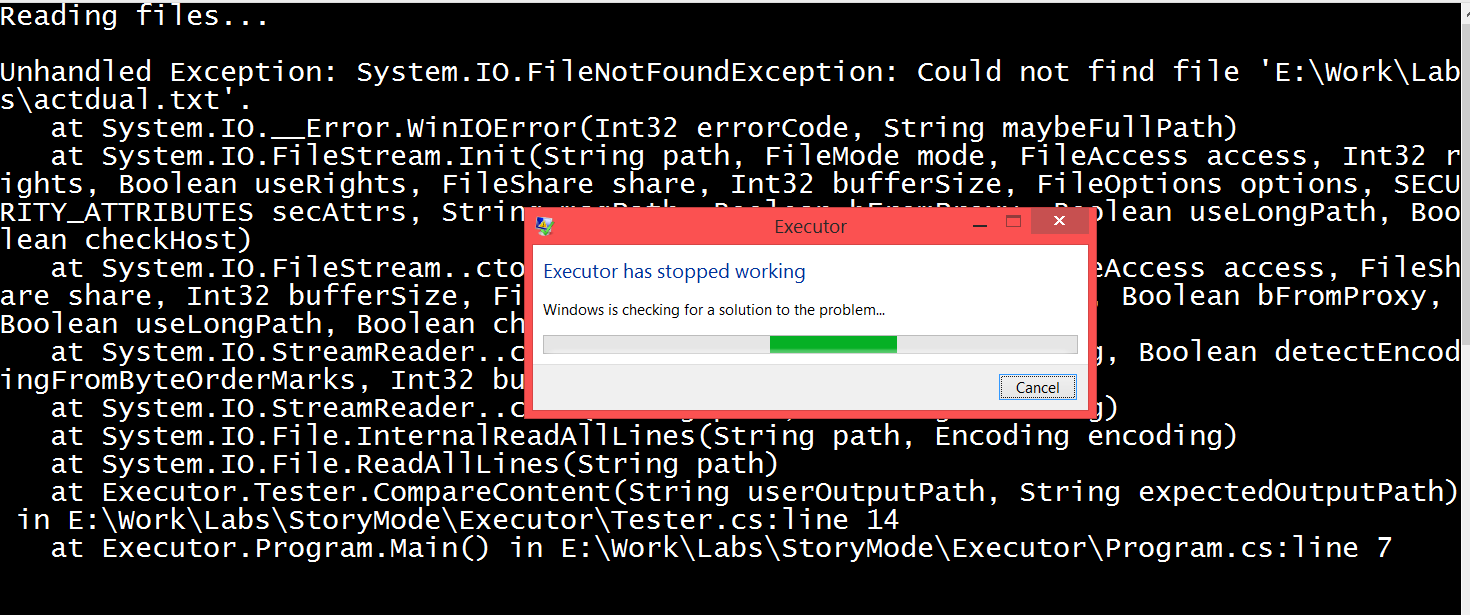
Now that we’ve fixed the situation here, we should proceed to the next step.

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:

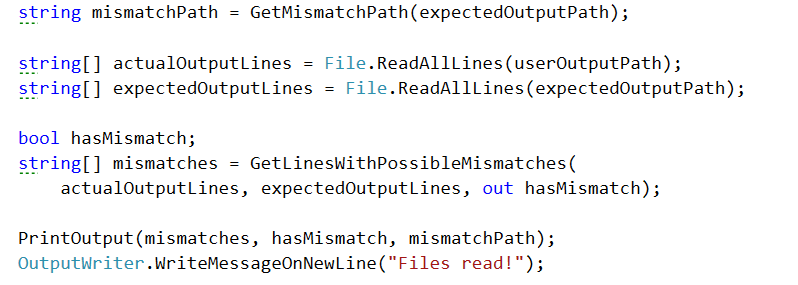


Results in the following:

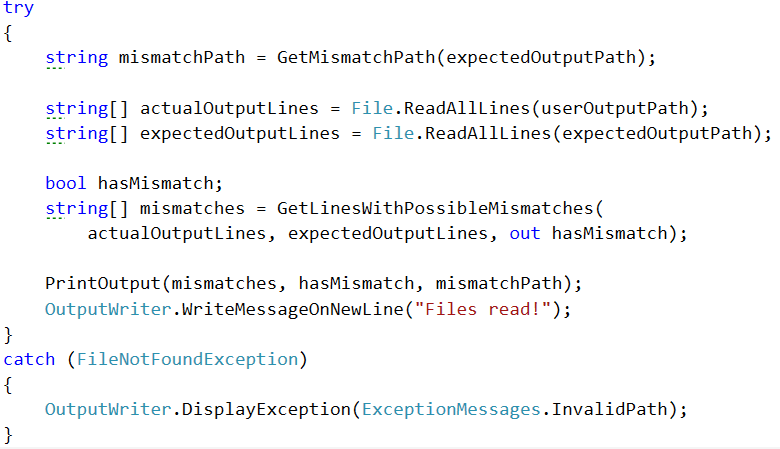


If we are making any kind of user interface, the application should always presume that the user is a two-year-old and can probably do or enter just about everything you can imagine and even more.

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 your code should be looking like this:



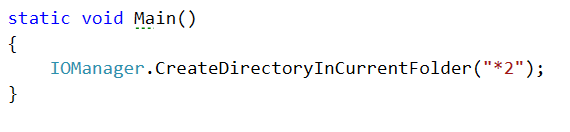
This should change to:



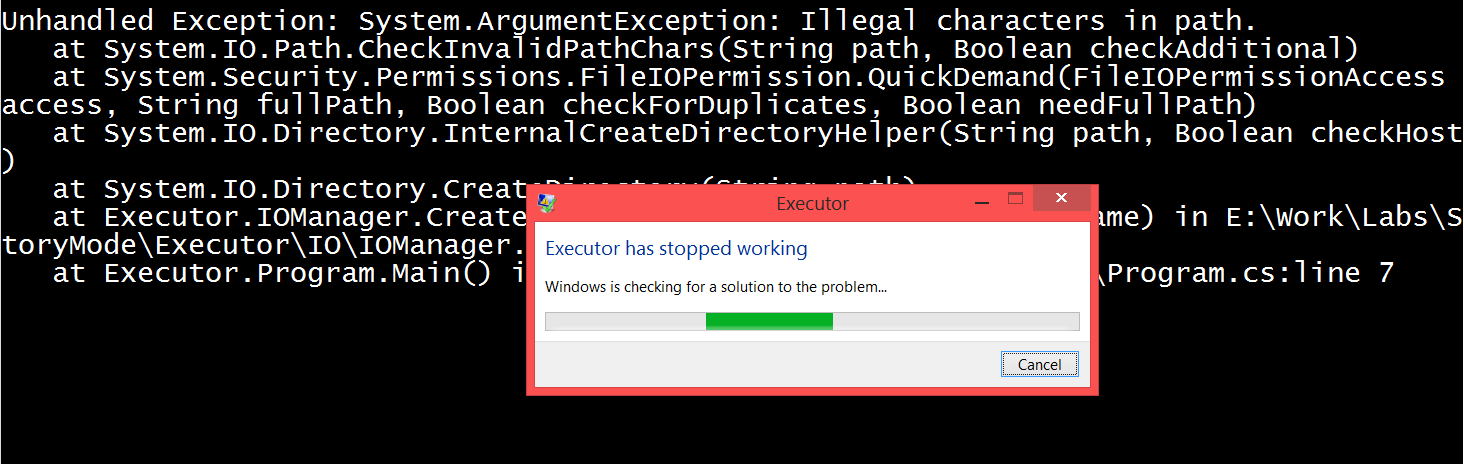
We are reusing the message for the invalid path in the current action, so we do not need to make a new one.   
Alright, now that we are done, let’s proceed to what is considered forbidden and what is consider allowed when talking about creating names of files and folders.

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:

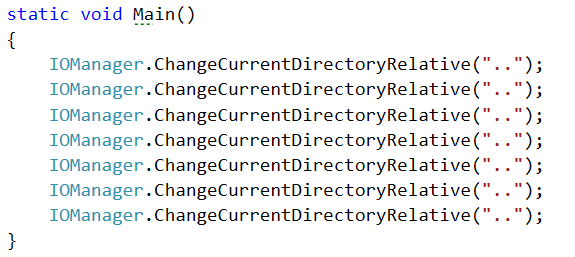


Going One Folder up the Hierarchy

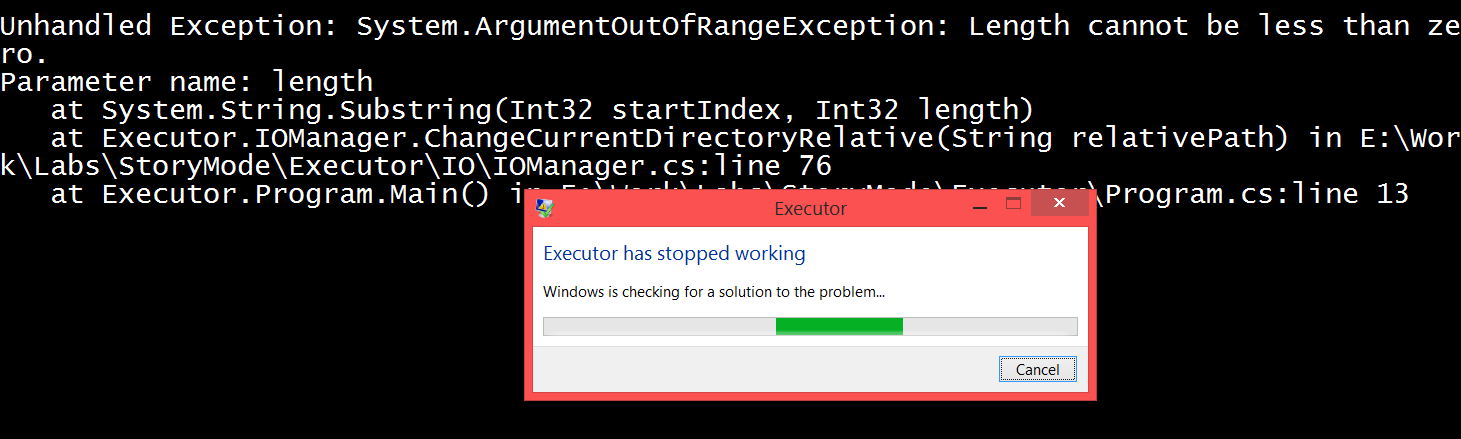
As we know, the logic for the changing of the folders works correctly, but have you tried to go one folder up when you are in the root folder of the partition.

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.

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

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.

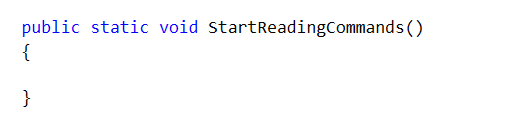
We are going to need two public static classes that handle the input and the commands. The first one is called InputReader and the second one is called CommandInterpreter.

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.

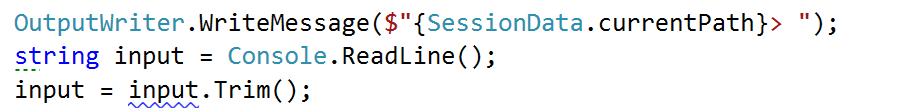


You’ve probably opened the Command Prompt before and you’ve seen that you do not write your commands on empty lines. Instead the folder that you are currently in is the beginning of the line.

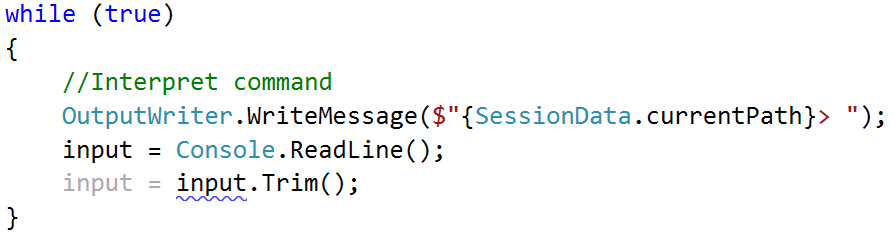


In order to add this functionality so that our bash looks like the command prompt, we will write a message on the OutputWriter which will be the current path from the SessionData class followed by ‘>’.

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:



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 asynchronously

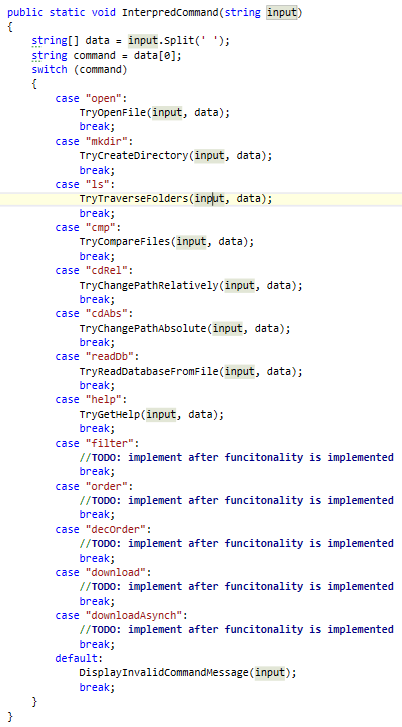
help – get help

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.

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:



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:  


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:   
  


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:  


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…  


Change directory absolute – the approach now is pretty much the same as in the previous command.   


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 facade 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.



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.

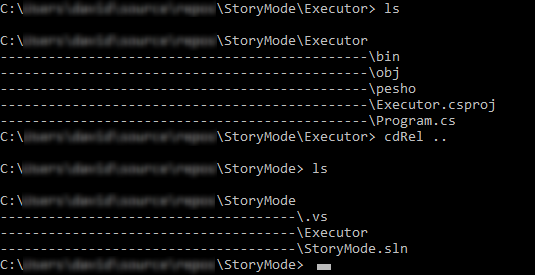
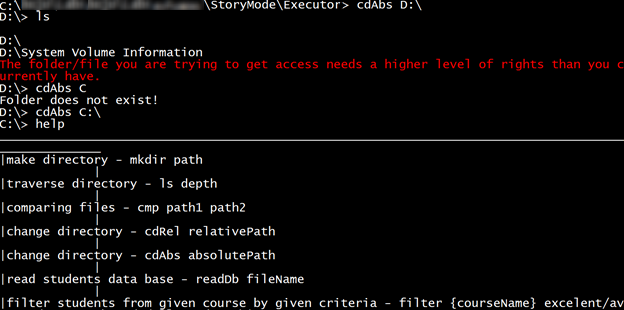
For the rest of the commands – you may leave the body empty, because we do not have the functionality implemented yet.

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 processed you string commands.

Regular Expressions

In the current lab we are going to introduce some restrictions in the data for our database. Below you can see the constraints to validate the input entry before adding it to the data structure.

Input format – the format for the input should be the following:

{Course Name}\_{Course Instance}{One or more white spaces}{Username}{One or more white spaces}{Score}

Our task now is to write a regular expression that matches only valid entries, so we can add them to our data structure safely. Here is some example input data that may be given:

C#\_Feb\_2015 Kiko23\_4144 69

JSApps\_Dec\_2014 Ivo42\_230 17

C#\_Jul\_2016 Kiko23\_4144 94

JSApps\_Dec\_2014 Sten16\_96 41

C#\_Feb\_2015 Desi12\_2001 77

WebFundamentals\_Oct\_2015 Ivo42\_230 238

DataStructures\_Apr\_2016 Ivan23\_923 94

C#\_July\_2016 Rdsauja16\_23 71

JSApps\_Dec\_2014 NiK68\_0192 1

Unity\_Jan\_2016 Sten16\_96 56

unity\_Jan\_2016 Sten16\_96 53

JSApps\_Dec\_2014 Stan21\_23 46

C#\_Feb\_2015 NiK68\_0192 53

DataStructures\_Apr\_2016 Stan21\_23 93

WebFundamentals\_Oct\_2015 Desi12\_2001 81

Java\_May\_2015 Ivo12\_2341 77

C#\_Feb\_15 Sten16\_96 12

C#\_Feb\_2015 Desi12\_2001 93

WebFundamentals\_Oct\_2015 Kiko23\_4144 87

Course name – starts with a capital letter and may contain only small and capital English letters, plus ‘+’ or hashtag ‘#’.

Course instance – should be in the format ‘Feb\_2015’, e.g. containing the first three letters of the month, starting with a capital letter, followed by an underscore and the year. The year should be between 2014 and the current year.

Username – starts with a capital letter and should be followed by at most 3 small letters after that. Then it should have 2 digits, followed by an underscore, followed by two to four digits. Correct: Ivan23\_234, Nas12\_4215, Re14\_203. Incorrect: Ivana33\_123, Stan\_12, Мари31\_421

Score – should be in the range from 0 to 100.

We are going to write a regular expression for validating the input and implement it in the method for reading data from a file for the database of the university.

Start Using a Regex Editor

First we want to open some regex editor that will help us to complete our task. You can use whatever editor you like but you should be already familiar with <https://regex101.com/> so we give you its link here. Next you may want to paste the sample data given above in the TEST STRING box:



Next you need to include the global modifier by simply adding a ‘g’ in the upper right corner:



Ok, that was pretty easy. Let’s proceed with the next task.

Using Groups

We are going to have three groups, which are as follows:

1: Course name and instance

2: Student user name

3: Student score on task

First we will start with the first one and to be more specific with the course name. It should start with a capital letter so this is the first thing to add and you will be able to see as we go, how some data does not meet the conditions of the regex. So our regular expression so far should look like this:



And the matches are still quite unclear:



As you can see even from the first condition we don’t catch the unity course written with small letters.

The next thing our regular expression should include is that the course name may contain small and capital letters as well as the symbols ‘+’ and ‘#’:

  
We have put an asterisk after the range, because the name of the course may be only of one capital letter. The result of the current modification looks like this:



Now it’s time to add an underscore and the condition for the month name that follows, followed by an underscore:   


Here the condition after the range should be exactly two letters, because the total number of letters for the name of the month is 3 and the first one should be capital. The result after this addition clearly shows where we are headed:



As you can see, now the C# course in July is no longer valid because the month is written with four letters.

Finally, it is time to add the year to the matching regular expression:

  
Now we should be ready with the group for the course and the result that we match by now should be as the one below:



As you can see, now we don’t catch the C#\_Feb\_15, because the year is not in the valid format.

Now it is time to write the next group for the user name which is really similar to the one we just wrote.  
We have to put a separator for one or more spaces followed by the group starting with a first capital letter:



The result after this filter is pretty much the same for the input we’ve chosen, but there could have been a person whose name starts with lower letter or an entry where there is no space between the course and the user name:  


Now we should finish the regex for the rest of the name before the two numbers that follow. We should keep in mind that we can have at most 3 letters after the first capital letter. This means that we may have 3 letters but there may be no letters after the first one. So this may be expressed as follows in the regular expression:





After this addition you can note that even if C#\_July\_2016 was written following the conditions, the user name would still be incorrect and of course the whole entry would be invalid.

So the only thing left for the username is the two digits that follow after the letters, followed by an underscore, which is also followed by two to four digits:



You may see below that 2 more matches are now invalid, because they don’t match the required format for the user name and more specifically for the numbers that are in the user name:



The final group we need to catch is the group for the task‘s score for the given person and the given course:



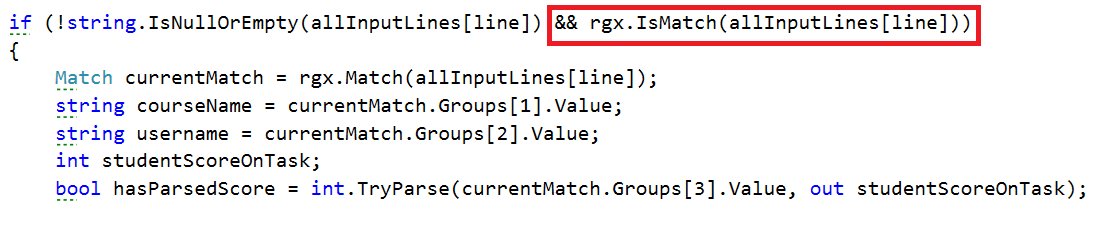
The result with the given matches from the example should now look like this:



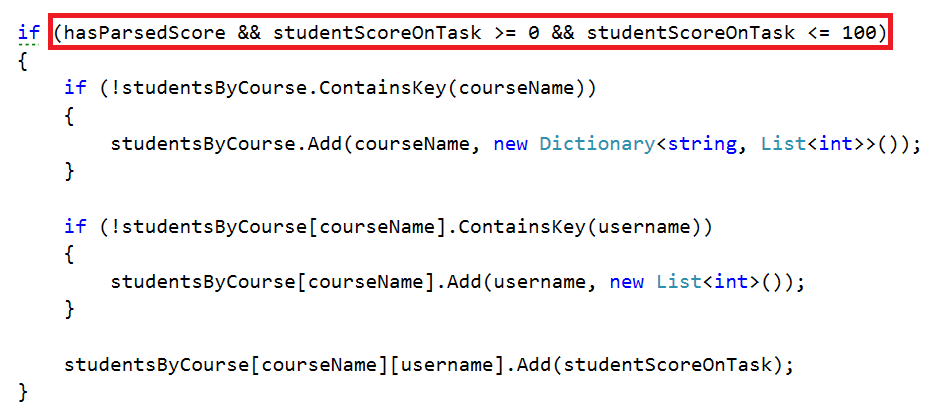
We’ve finally written the whole regular expression and it is time to implement it in C# in the method that reads all the students from the “database” file, but now you will be given a new file that will contain entries that do not match the given format.

We are going to refactor some code in the ReadData method in the StudentRepository class because we will now have to get the data from the groups of the current match.

The first thing is to copy the pattern of the match for the entries and also create a new Regex with the given pattern:   
  


Now that we have this instance of the Regex class, we can use it to check if there is a match with the current input line and if there is such, to get the data that we need from it in order to insert it in the data structure:  
  


So the only thing left is to check if the score has been parsed and whether it is in the range between 0 and 100 and if all the three conditions are true, we can insert the data that we have extracted.



By now we should be ready with the implementation of the regular expressions.

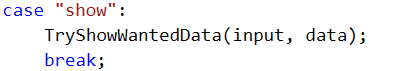
Adding Features to the Command Interpreter

Just before testing our new functionality there is one little thing we can add to the Command Interpreter, because obviously we forgot adding it in the previous piece. We are going to add a new command which has the following format:

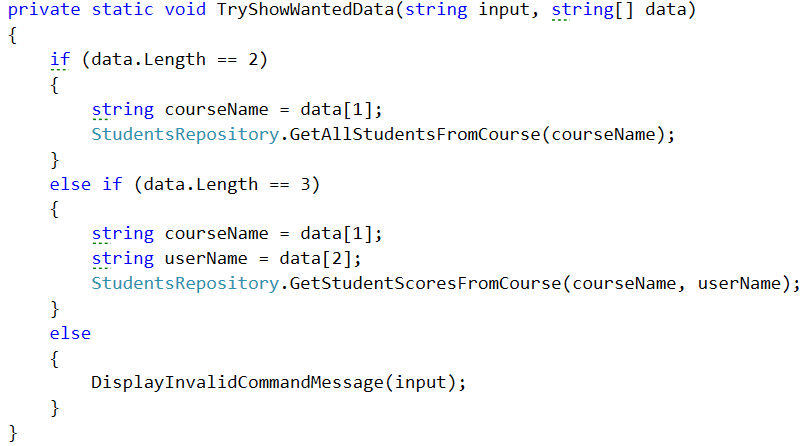
show courseName (username) – user name may be omitted

It’s purpose is to show information for the given course or the given username for a course from the database.

In the switch of the Command Interpreter the case looks like as follows:



Now it’s time to add the TryShowWantedData(string input, string[] data) method and implement its functionality. We should only check for the number of parameters and depending on this, call the corresponding method.

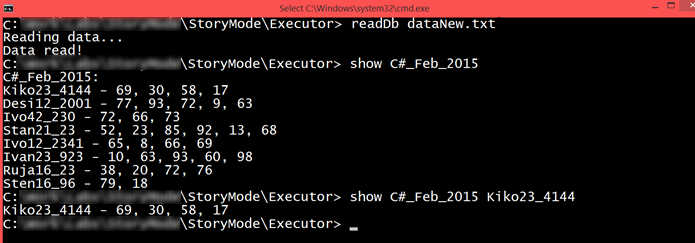


Testing Your Code

Finally, we should be done with the corrections and now it’s time for testing. Тhe only thing we should call in the main method is InputReader.StartReadingCommands();



Since I’ve put the dataNew.txt in the Main directory folder, I only read it, but you’ve put it elsewhere, you’ll have to navigate to the folder first.



You should be ready with the testing! Next time expect to filter and order the data we have just read by some criteria using functional programming.

Congratulations! You’re done with the validation of the entries.

Functional Programming

In the current Bashsoft piece we are going to add some filters and implement some sorting algorithms so that we may see how functional programming could be helpful here. The filters and sort types are described in the String’s part but let’s do a short revision. We said that we are going to add a filter for a given course in order to extract some/all poor/average/excellent students and print them on the current output in the OutputWriter. After that we are going to sort the filtered data by a given criteria (ascending/descending) and again take some or all the students from the query.

Let’s first stat by creating two new public static classes called RepositoryFilters and RepositorySorters.

Filtered Students Query

Implement Filters

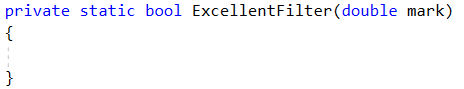
The first method we need in the filters repository class is the public API we are going to give to the world to use. It’s going to be a public static void method called FilterAndTake. Since we are going to filter students from a given course, we need to receive the dictionary that corresponds to the students with their scores from the seeked course. Another thing the method has to receive is which filter to use. Since we are reading strings from the InputReader we can pass them to this method as a string and here in the RepositoryFilters class we can decide which filter to apply to the data. The final parameter that the method needs to receive is the number of students to take. Since we parse it in the command interpreter when we check the input data, the type of the variable will be an integer. By now the method signature of FilterAndTake should look like this:

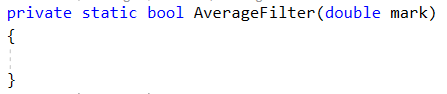
C:\Users\david\Desktop\FilterAndTake.jpg

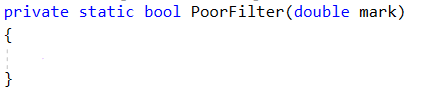
Since the public method receives the wanted filter as a string, it’s his job to decide which method for filtering to use. The method which will actually do the filtration can be named FilterAndTake again, however it’s going to be private static void and with a change in the parameters. The new FilterAndTake is going to receive the same wantedData, and the same variable studentsToTake, but the wantedFilter is now a Predicate (method that returns a bool) that receives a double. The description above should look like this:

C:\Users\david\Desktop\FilterAndTake.jpg

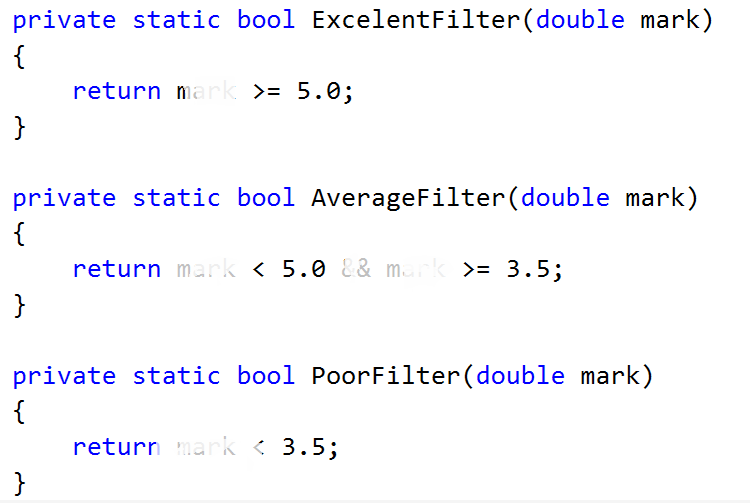
As you can see things are a bit coupled but in the same time quite detached because we can easily extend the methods. Now we need to implement the methods that we are going to be passed as predicates which are actually the real filters. We are going to have three of them since we have three types of students (excellent/poor/average). This is how the initialization of the methods should look like:







The parameter representing the mark should be in the range from 2 to 6, so it’s up to us to decide which mark corresponds to excellent, average and poor. We suggest that you return true for an excellent mark if it is higher or equal to 5.00, return true for an average mark if it is higher or equal to 3.50 and less than 5.00 and finally return true for a poor mark if it is less than 3.50. If you’ve followed the instructions, by now you should have something like this:



Implement Average Mark

There is one more helper method we need to implement in order to do the job. It’s called Average and receives a list of scores. It should be private and static and since it’s going to return the average mark, we leave it up to you to decide what’s the best return type of this method.

Let’s create this method. First we’ll need a variable to hold the total score of all the tasks. Then we should iterate through the list and add each value to the total score. Finally after the foreach we should take the percentage of the total success rate which can be obtained by dividing the average score by the number of tasks multiplied by 100 to get percents. Now we have the percentage of total success and we can easily calculate our mark by the formula mark = percentageOfAll \* 4 + 2. If you’ve done everything correct, by now your implementation of the method for calculating the average mark should be something pretty close to this piece of code:   
  


Implement private FilterAndTake method

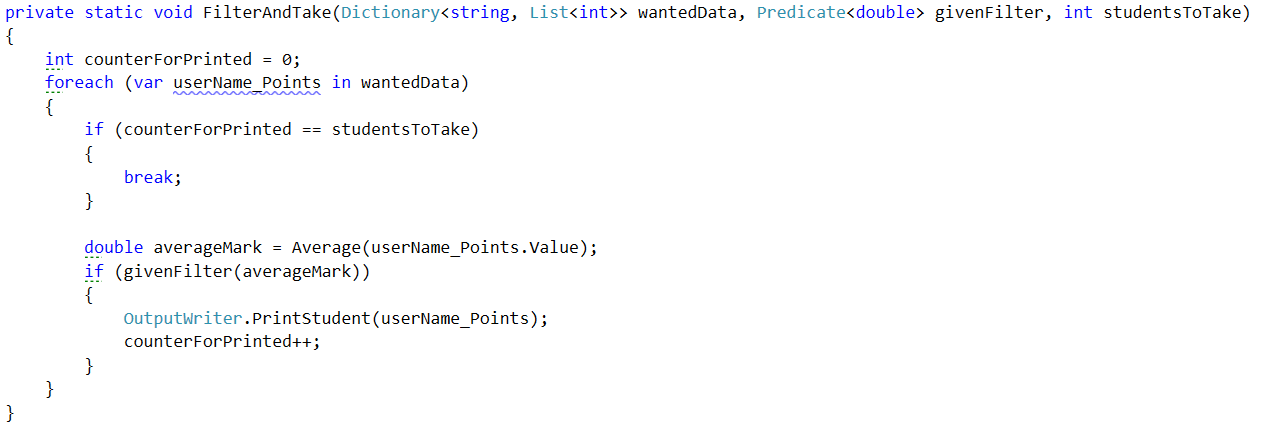
Now that we are done with the helper methods it’s time to move to the actual place where the filtering is done - the private FilterAndTake method.

First thing we are going to need in the method is a variable to hold the number of printed students that match the given filter in order not to exceed the limited number of students we are asked to return.

Next we’ll iterate through all the entries in the dictionary called wantedData and for each student, we calculate it’s average mark using the method we implemented just before.

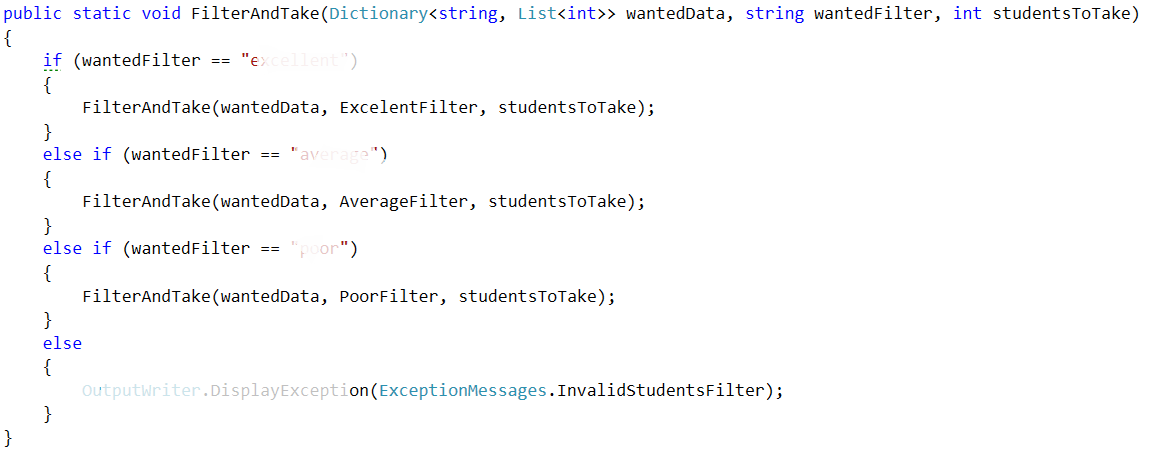
Finally we check if the average mark, passed to the given filter, returns true. And if so we print the student on the OutputWriter using PrintStudent method and increment the counter for printed students.

There is just one more little problem. We have no condition to stop printing students when the limit is reached. So we have to add a block of code that breaks the loop if we’ve printed enough students and it should be first in the foreach loop. By doing this, our foreach loop now look like the following:



Implement Public FilterAndTake Method

Now we are only left with the public FilterAndTake method which is actually going to be the method that the outer world is going to use in order to filter the given data. It’s implementation is very straightforward. All we do is to check if the wanted filter corresponds to one of the possible categories (excellent/average/poor) and if so, we call the private FilterAndTake method, with an input parameter for the Predicate, the function that corresponds to the category. If the given word does not match any of the categories, we display an exception called InvalidStudentFilter, which we first need to add to the ExceptionMessages with a message of: “The given filter is not one of the following: excellent/average/poor”. So our implementation of the public method should look likes this:



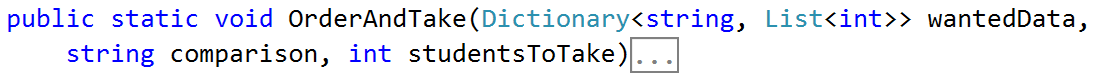
We should be ready with the filtering repositories class and it’s time to move on to the sorting repos’ class.

Part II: Sorted Students Query

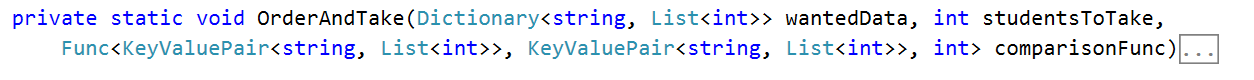
Implement Sorters

The first method we need in the sorter repository class is the public API we are going to give to the world to use. It’s going to be a public static void method called OrderAndTake. Since we are going to sort students from a given course, we need to receive the dictionary that corresponds to the students with their scores from the wanted course. Another thing the method has to receive is which sorter to use. Since we are reading strings, from the InputReader , we can pass it to this method as a string and here in the RepositorySorters class we can now decide which sorter to apply to the data. The final parameter that the method needs to receive is the number of students to take. Since we can parse it in the checking of the data, that we do in the command interpreter, the data type of the variable can be an integer.

By now the method signature of OrderAndTake should look like this:

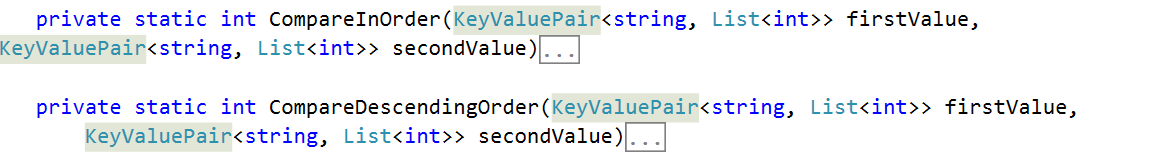


Since the public method receives the wanted sorter as a string here we’ll decide which sorting method to use. Again we will put the real sorting in another method. Similarly it can be called OrderAndTake, however it’s going to be again a private static void and with a change in the parameters. The new OrderAndTake is going to receive the same wantedData, and the same variable studentsToTake, but the comparison type (sorter) is now a Func that receives a two key value pairs (students with marks) and returns an int which is the result of the comparison. The description above should look like this:

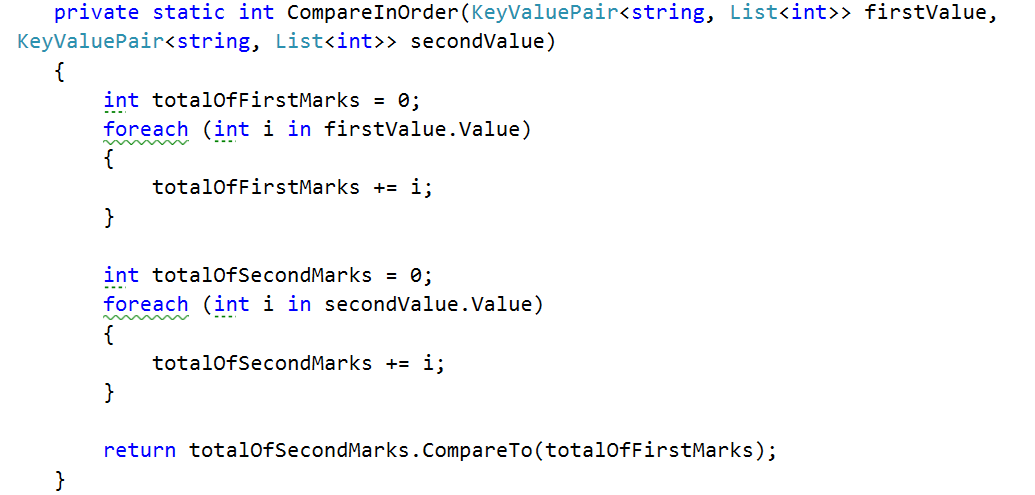


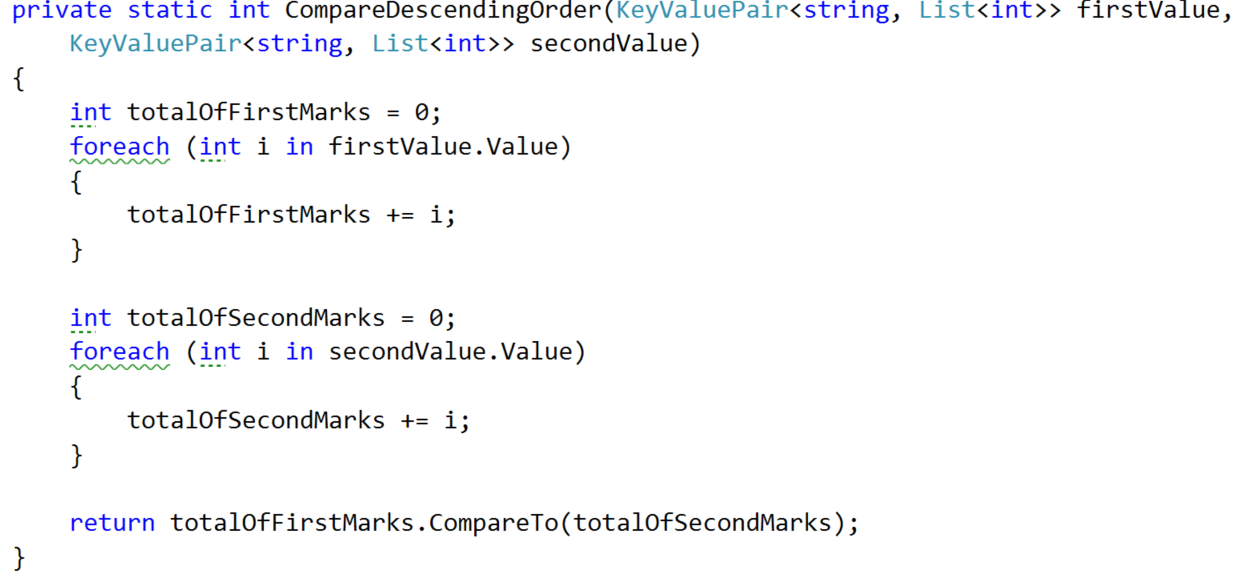
Now we need to implement the functions that are actually going to be our comparison types, in order to figure out how the OrderAndTake method is going to work.

There are going to be two methods of such type since we have two types of comparisons (ascending/descending). This is how the initialization of the methods should look like:



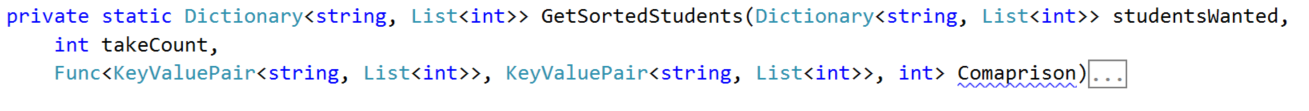
We have to compare the two students by a given way and return 1, 0 or -1 depending on which one is greater/smaller. To compare them in order, we compare the sum of the scores of all tasks and return the result of the second compared to the first. For the other one we do the same thing, but we compare them in the opposite way. The implementation should look like the following:





Implement Private OrderAndTake Method

Now that we are done with the helper methods it’s time to move to the actual place where the sorting is printed and that is the private OrderAndTake method. We simply make a new dictionary which should contains a string and a list of integers called studentsSorted that is equal to the GetSortedStudents method. We haven’t talked about it yet but it’s signature should look like this:

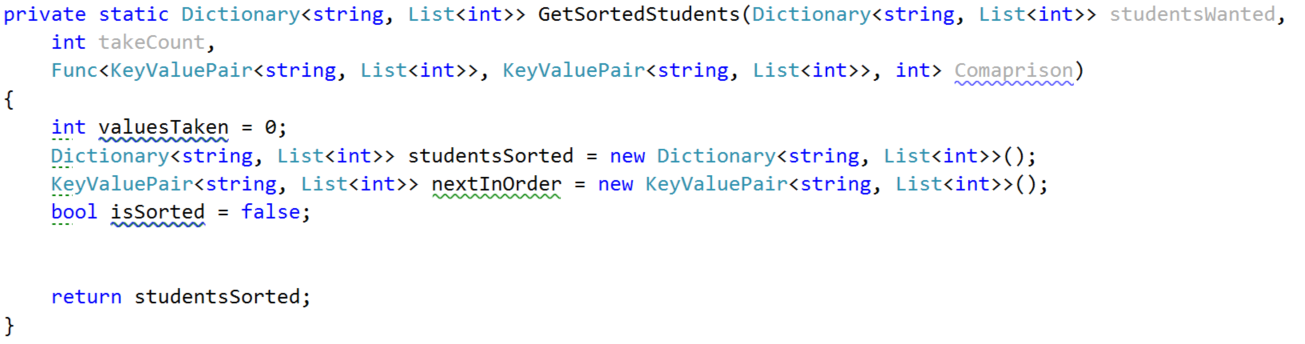


After we’ve gotten the sorted student in a dictionary, we simply print it on the output writer using the PrintStudent method.

Implement Private GetSortedStudents

The first thing we do in this method is to set up a variable for the number of values taken and set it to zero to help us return only the requested amount of students. Next make a new dictionary for the sorted students. Finally, we should make one more helper variable to hold the next value that is in the requested order.

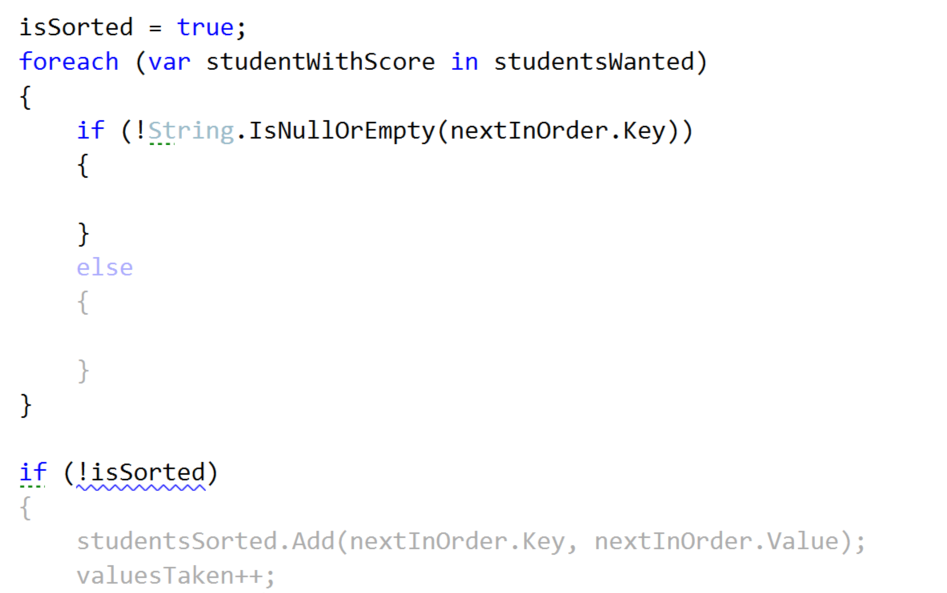
Now it’s time to implement an easily understandable sorting algorithm and for that reason we’ve chosen the bubble sort algorithm. For the job you need to add one final helper variable of Boolean type that is called isSorted, because the bubble sort needs such a variable for the condition of the loop. By now your method should look like this:



From now on we place the while loop of the bubble sort and on each iteration we first set the is sorted to true. At the end of the loop we check if the isSorted bool is not true and if so, add the data from the nextInOrder to the studentsSorted. After that increment the valuesTaken and finally set the nextInOrder to a new KeyValuePair:

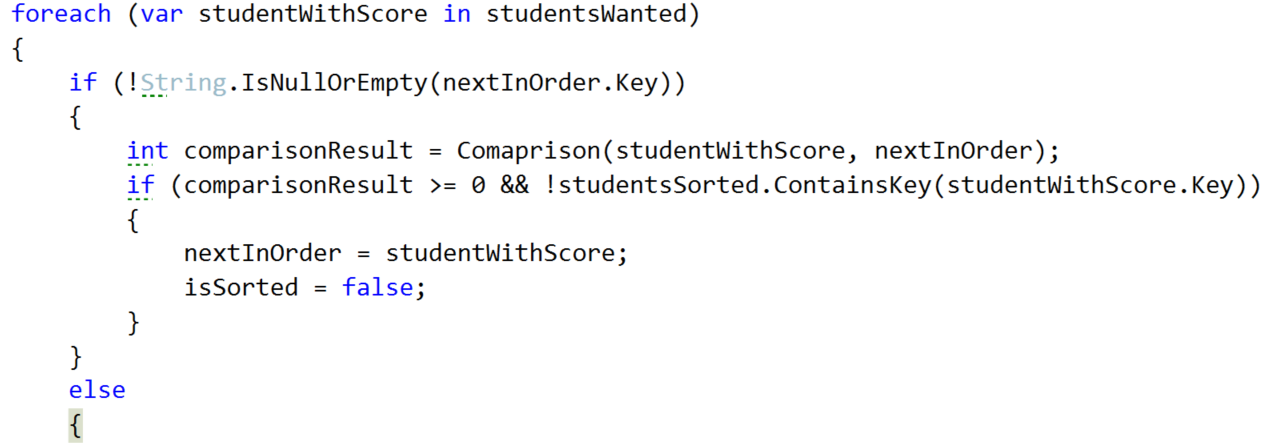


Next thing in the queue for implementation is the inner loop that finds the current min/max element. For that reason we make a new foreach over the studentsWanted. The keyvalue pair nextInOrder could be set but it could also null so we may have a null key and a null value. So we can check if the nextInOrder’s key is not null or empty and do something and if not do another thing:

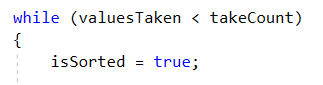


Let’s first implement the else clause. In there we have to check whether the new sorted dictionary does NOT contain as a key the current studentWithScore’s key. If so, we set the nextInOrder to the studentWithScore and set the isSorted to false.



Waiting up next is the if clause above. We take the int that our Comparison function returns, by passing to it the nextInOrder and the studentWithScore If the comparison result is greater than or equal to 0 and the dictionary that we use for the sorted students does NOT contain the key of the studentsWithScore’s key, we set the nextInOrder to the studentWithScore and the isSorted to false.  
  


Don’t forget to set the condition of the while loop to stop when we gather the needed students or else you are going to end up with an endless cycle.



Now that we are ready with the GetSortedStudents, we hope that the private OrderAndTake will also work correctly. So one last thing is left in the current class and it is to implement the public OrderAndTake.

Implement Public OrderAndTake Method

Here our only job is to decide how to choose which comparison type to use. That is why we do pretty much the same thing as in the public FilterAndTake. First we check if the comparisonType string is ascending and if so, call the private OrderAndTake, passing the in order comparison Func. If descending is chosen, call the same method with the descending order comparison Func. If none of the comparisons is chosen we display a new Exception message, which we should first add to the ExceptionMessages called InvalidComparisonQuery with a message “The comparison query you want, does not exist in the context of the current program!”

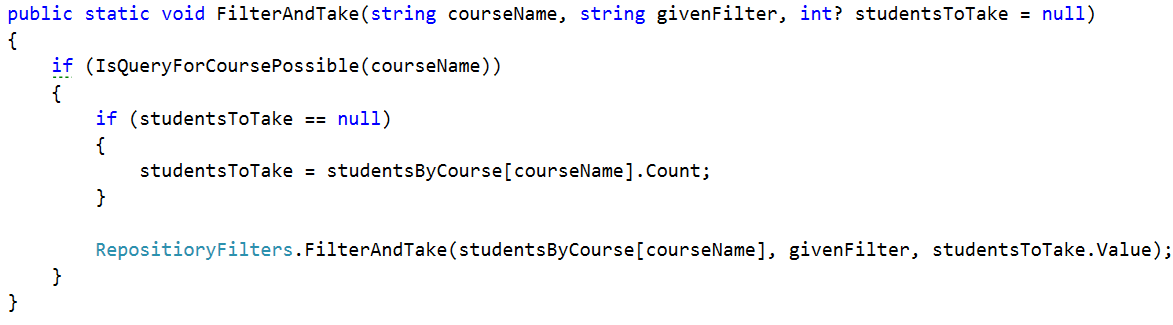


Part III: Student Repository’s Filters and Sorters

Implement Public FilterAndTake method

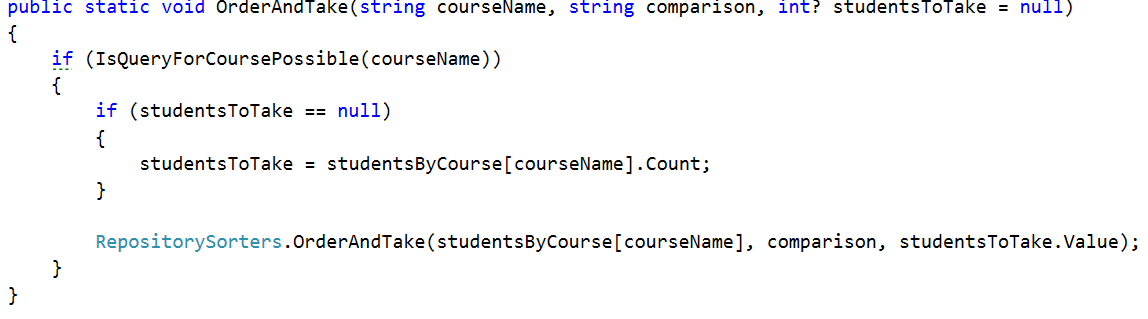
Since we are going to use the dictionary from the StudentsRepository class and it is private, we can easily take all that we need from the StudentsRepository by using it as a mediator between the command interpreter and the filters/sorters. So we are going to add two methods in this class. One that is called FilterAndTake and another one OrderAndTake. The filter follows the following signature:



If you’re wondering why the studentsToTake is nullable with a default value of null it’s because we want to call the method with giving it the parsed value and if it hasn’t been parsed (this happens in the command interpreter – we’ll get there soon) for example if the user has inputted “all”, we want to make sure we take the number of students in the current course and that is only possible once we’re in the StudentRepository class. If you are confused, don’t worry it’s harder to explain that to see it in code.   
  


Implement Public OrderAndTake method

The situation in the OrderAndTake method is pretty much the same as you can see:

  
Now that we have these methods we can easily communicate with the RepositoeryFilters indirectly using the StudentsRepository.

Part IV: Command Interpreter’s Filters and Sorters.

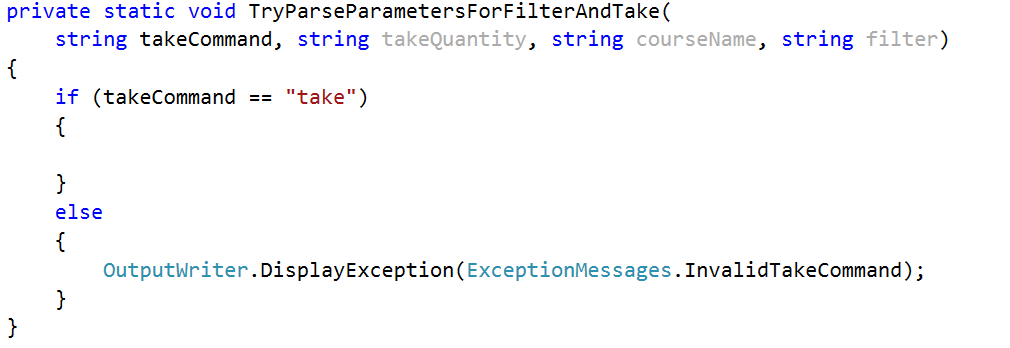
In the command interpreter we should make two methods called TryFilterAndTake and TryOrderAndTake that take input parameters, the same as all the other try methods in this class. After making them we should call them in the InterpredCommand method in the appropriate place.

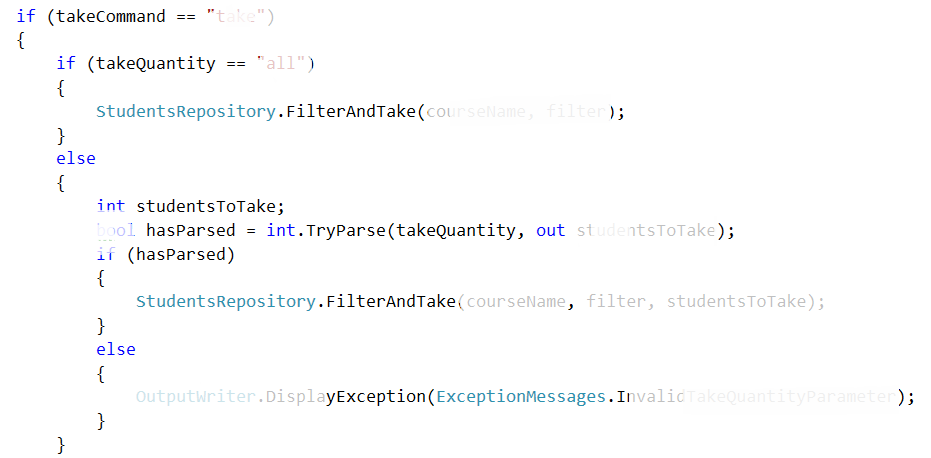
Implement Filtering Data Parsing in Command Interpreter

Let’s first look at the implementation of the TryFilterAndTake method. All we have to do there is check if the number of input parameters is 5 and if not, DisplayInvalidCommandMessage. If it is, we take the course name which is at index 1, the filter in lower case at index 2, the take command in lower case at index 3 and finally the take quantity in lower case at index 4. Finally we should pass all those parameters to a new method TryParseParametersForFilterAndTake.   
  


Actually the method we mentioned above does almost all of the validation of the parameters so let’s look at its implementation.

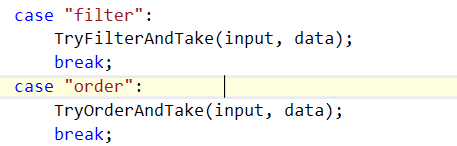
First we check if the take command is actually equal to the word “take” and if not we print an exception message on the output writer, which of course we should first add, called InvalidTakeQuantityParameter with a message “The take command expected does not match the format wanted!”



If this is the actual command then we have to check if the take quantity is “all”. If so, call FilterAndTake from the StudentsRepository without the last parameter for the quantity and therefore it is null by default, because we set it to a nullable int. However if that is not the case, we have to check if it is a number that can be parsed. If the number can be parsed, we get the result from the parse and call the FilterAndTake but including the last parameter. In the case where the number hasn’t been parsed we should display an exception for InvalidTakeQuantityParameter. All of the above should look something like this:  
  


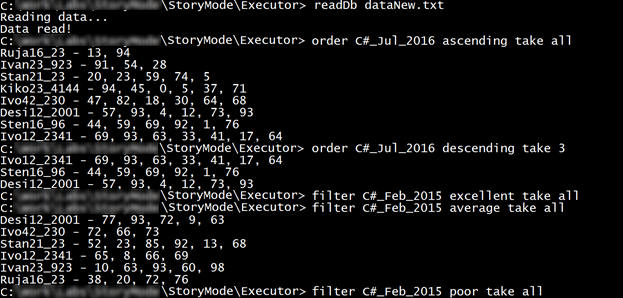
The implementation of the TryParseParametersForOrderAndTake is the same so we leave the implementation of this method to you.

Now if you’ve done everything and the situation in the switch case in the InterpredCommand method is the following :



Everything should be ok and we are ready to start reading from the input.

Next thing to do is read the dataNew.txt from where you’ve saved it and apply one sorting and one filtering.

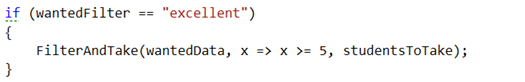


LINQ

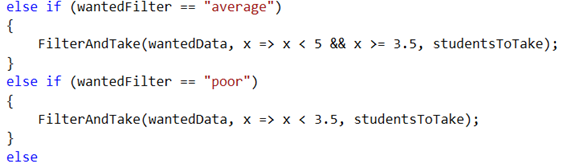
In the current piece we will see how we can implement some of the operations with less code thanks to the LINQ operators. In this way we will improve the readability of the project and that’s something you are obliged to do.

Change Predicate Methods with Lambda Expressions

The first thing we are going to change though is not related to LINQ. All we want to change here is in the RepositoryFilters in the public FilterAndTake. In the previous piece we created 3 methods that filter the given data. However, we have the possibility to delete the 15 rows that these methods take in the code. In the first method, where the wantedFilter is excellent we can easily express it through a predicate using lambda. We have only one input parameter so our statement in the call of the private FilterAndTake should look something like this:



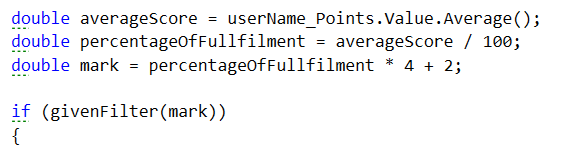
Next up are the average and the poor filter, however they are pretty much the same as the one above. For this reason we are not going to discuss them any further and just change the calls of the filter methods with the appropriate lambda expression:



Now we can easily delete the three methods from the repository filters class.

Use LINQ Average Instead of a Custom One

Another piece we can get rid of is the Average method and replace it with the operator that comes from the LINQ. In the private FilterAndTake we should change the averageMark’s name to averageScore and its value equal to the output of the just mentioned above operator. Next thing we need to make is a new variable called percentageOfFullfillments equal to the average score devided by the maximal score on a task which is 100. Finally, we should make one last variable that is the actual mark and it is equal to the percentage of fulfillment multiplied by 4 and summed with 2 after that. Here is how the private FilterAndTake method should look:



Changing Structure in Repository Sorters

I guess you wouldn’t be quite happy to hear that we’ve done the sorting the hard way. But now we can appreciate LINQ’s easiness and replace it with the easy and more readable way. This means that we can delete the private OrderAndTake method but just before that let’s extract only one method that we can reuse for our functionality. The new method for printing the sorted students declaration’s and implementation should look like this:

  
  
Now you can easily delete all the methods except the public OrderAndTake and the one we just extracted. So after the deletions of all occurrences, the class should look like this:



Whether the wanted filter is ascending or descending we want to call the PrintStudents method passing an already sorted dictionary.



First we are going to implement the ascending comparison and then we’ll need to only change one word and copy the rest of the sorting so that it works for descending as well.

Since we have a method for ordering a collection from LINQ, we are going to use it out of the box.

  
  
What the function wants is the criteria which to use for sorting and in our case we’ll pass to it the sum of the scores on the tasks.

  
  
Now we should take only the needed number of results:



Finally, we should convert it to a dictionary since after ordering it’s a collection of different type:



As we said we will copy this piece and only change the OdrerBy with a OrderByDescending.



Now you can test the functionality of the filters and sorters and see if they still work.

Creating folder structure

There are no more places where we can use LINQ so I suggest we use the current piece to put the structure of our project in order at least a little bit.

Let’s create 4 folders in the current project called IO, Judge, Repository, Static data. In the IO folder put all the following things:



In the Judge folder as you can imagine, we’ll put the Tester class:



In the repositories folder we’ll put everything related to the repository:



And finally in the Static data we should put the ExceptionMessages and the SessionData.



Well done! You have completed the tutorial for your BashSoft a.k.a DIY Judge. Although you may feel free to continue explore and experiment with adding new features.