# Lab: Functional Programming

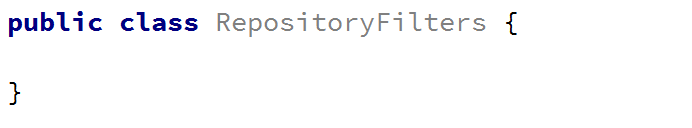
In the current lab we are going to make some filters and implement a sorting algorithm so we can see how functional programming can be helpful here. The filters and sort types are described in the strings lab, however let’s revise them again. We said that we are going to make a filter for a given course in order to extract some or all poor/average/excellent students and print them on the current output in the output writer. After that we are going to sort the data by given criteria (ascending/descending) and again take some or all students from the query.

Let’s first stat by making two public static classes called repositoryFilters and repositorySorters.

# Part I: Filtered Students Query

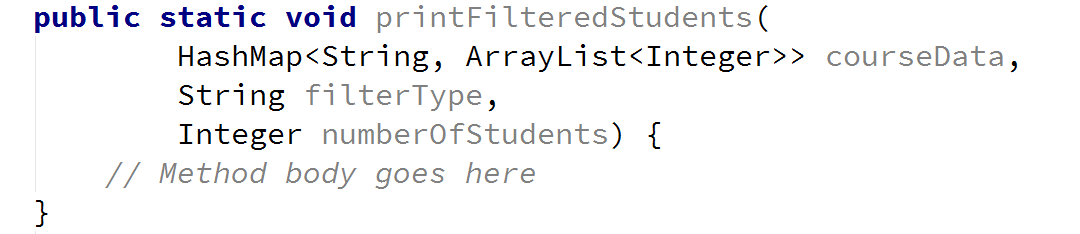
## Implement Filters

The first thing to do is to make one more class called RepositoryFilters.



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 printFilteredStuents(). Since we are going to filter students from a given course, we need to get as parameter the dictionary that corresponds to the students with their scores for a given 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 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 printFilteredStudents() should look like this:



Now we need to create a **predicate** which will actually going to be our filter. We can use a method called createFilter for this purpose.



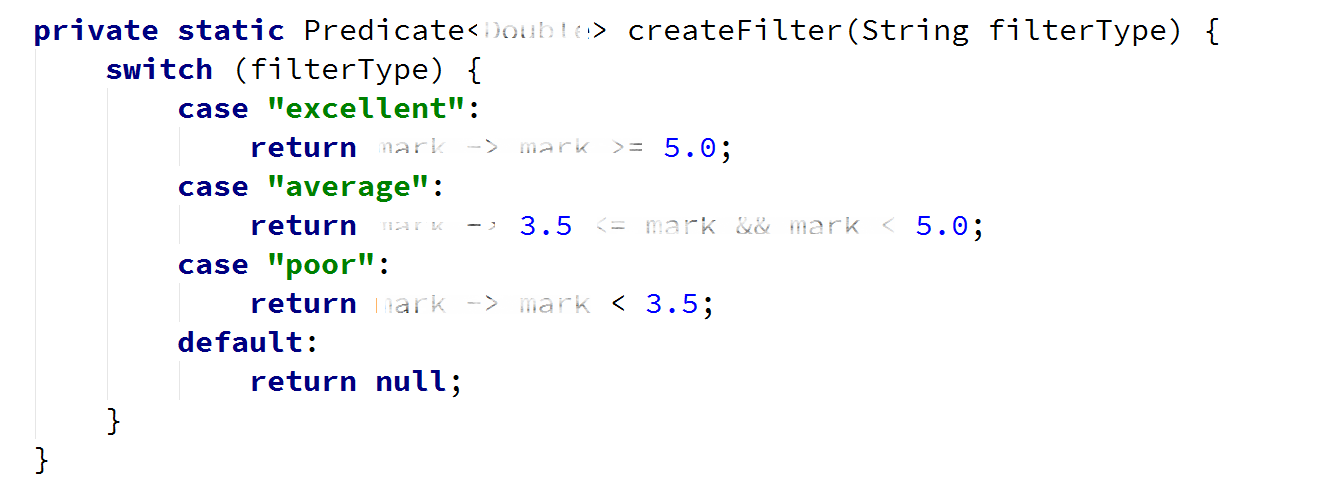
It will receive only the string which tells us what filter do we need and within a switch-case statement we will create and return a predicate. Since we are receiving a mark as a parameter, it’s in the range from 2 to 6, so it’s up to use, which mark is excellent, which is average and which is poor. We suggest that you return true for an excellent mark if it is more than or equal to 5.00, return true for an average mark if It is more than 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. So we have three cases:

Excellent students -> with average grades ≥ 5.0;

Average students -> with average grades within the range 3.5 ≤ avg < 5.0;

Poor students -> with average grades < 3.5

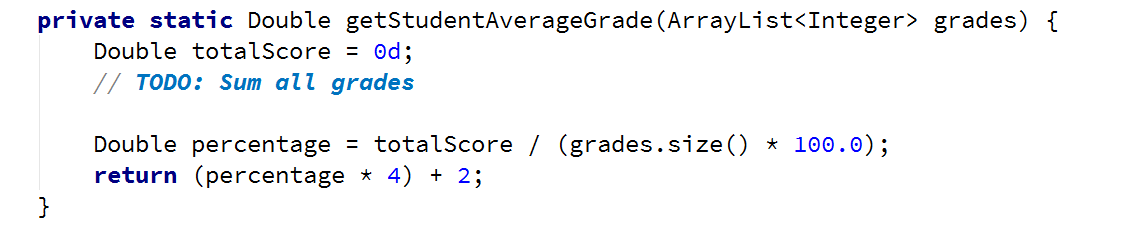
And the method should look like:



## Calculate Average Grades

There is one more helper method we need to make in order to do the job. It’s called getStudentAverageGrade() 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 good return type of the method.

After we’ve implemented the signature of the method it’s time for the implementation. First we’ll need a variable to hold the total score for all the tasks. Next thing we should do is to iterate through the list and add each value to the total score. Finally, after the loop we should take the percentage of the sum of all grades and divide it by their count. Now we have the percentage from all possible points and we can easily calculate our mark by the formula (percentage \* 4) + 2. If you’ve done everything correct, by now your implementation of the method for the calculation of the average mark should be something pretty close to this:

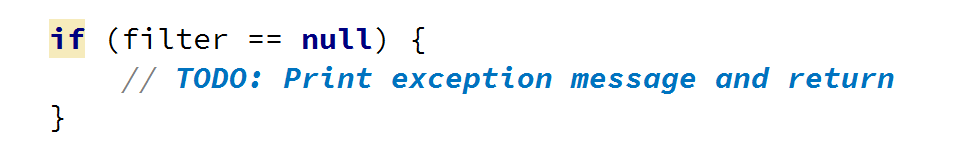


Just complete the TODO by yourself.

## Filter and Print Students

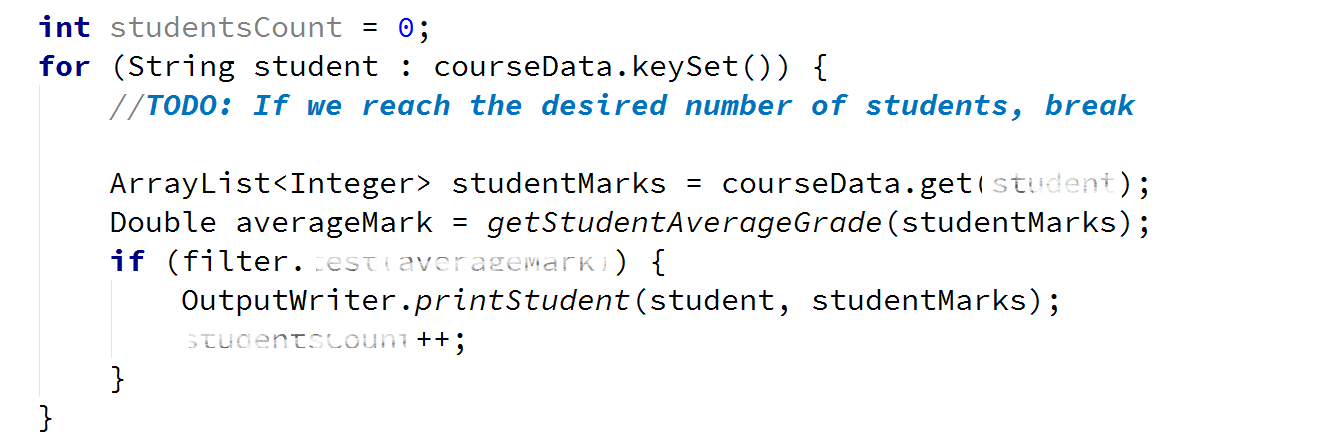
Now that we are done with the helper methods, it’s time to move to the actual place where the filtering is done and that is the filterAndTake() method.

First thing we are going to need in the method is to check if the predicate that we previously created is actually created. If it is equal to null, obviously we failed to create one so we will print an exception message. Create a new exception message in the appropriate class (Exception Messages) called INVALID\_FILTER containing the string “Invalid filter.” and if the predicate is equal to null print it. There should be also a return statement so the method stops there.



Next, we create a variable which will hold the count of students that we already printed and then start iterating through all the entries in the hash map. For each student, we calculate its average mark using the method we implemented above, as we pass to it, the value of the key-value pair that give us the current iteration of the dictionary.

Finally, we check if the average mark, passed to the given filter, returns true. And if that is so. We print the student on the output writer using print student method and increment the counter. By now the implementation of the method we are talking about should look like this:

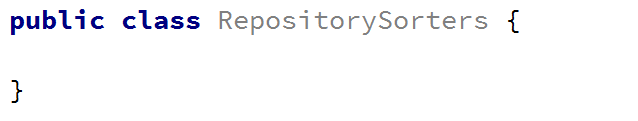


We should be done with the filtering repositories class and it is time to move on to the sorting class.

# Part II: Sorted Students Query

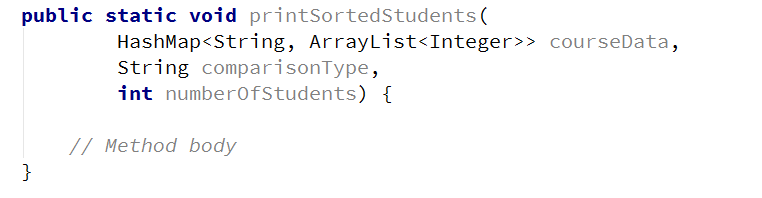
## Implement Sorters

Now let’s make another class RepositorySorters.



In the sorter repository, we need a method that is going to be a public static void, called printSortedStudents(). Since we are going to sort students from a given course, we need to receive the hash map that corresponds to the students with their scores from the course. Another thing the method has to receive is which sorter to use. Since we are reading strings from the InputReader, we can pass them to this method as a string and here in the RepositorySorters class we can 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 printSortedStudents() should look like this:

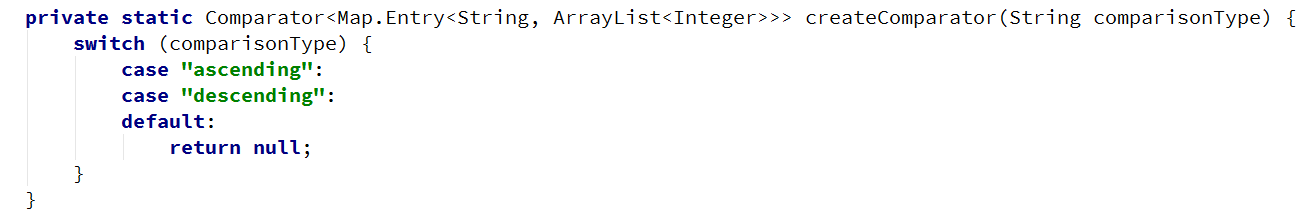


## Create Custom Comparator

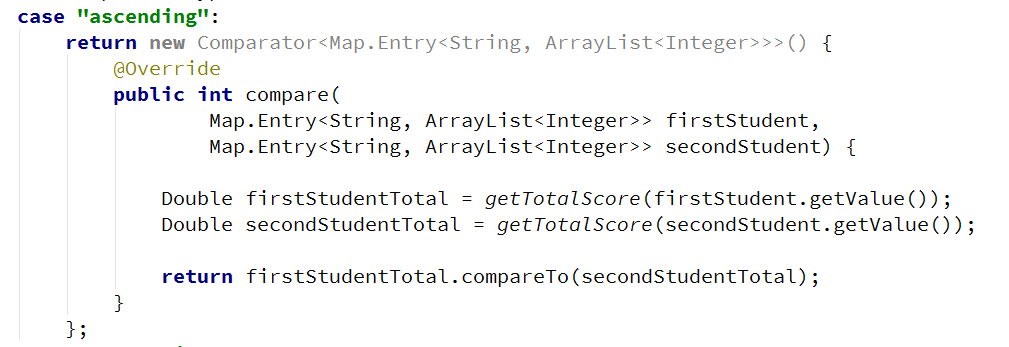
Since the public method receives the type of sort as a string, we need to create a custom comparator that will do the actual sorting. We can leave the actual creation to another method and the custom comparator will compare map entries so we can initialize it like this:



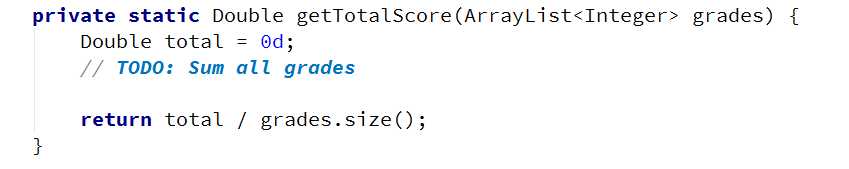
Now we have to define the method that will create our comparators. It will get as parameters the string with the comparison type and then in a switch-case statement choose and return the appropriate one:



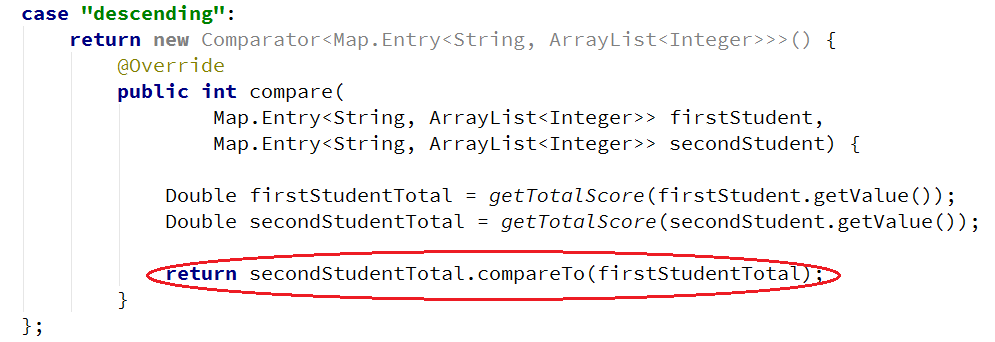
Then we need to create a Comparator, override its default compare method and then return it. The comparator will receive two map entries and compare them by the average score of a student:



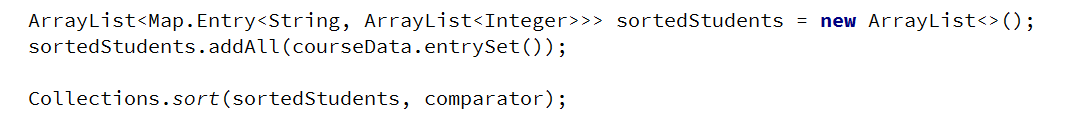
For this we need to create one helper method getTotalScore() that will just get the sum of all grades and then divide it to the grades count:



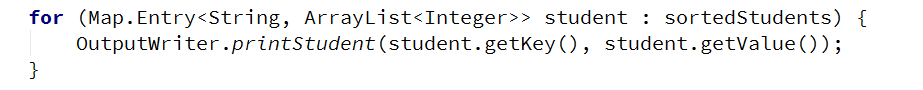
The case where we need to compare by descending grades is the same as the above, but we compare them in reverse order:



Now is the time to use the comparator that we’ve created. We can do this by adding all elements of the hash map to an array list and then use the method Collections.sort() by passing to it our array list and the comparator:



And the last thing that we need to do is to print all sorted students:



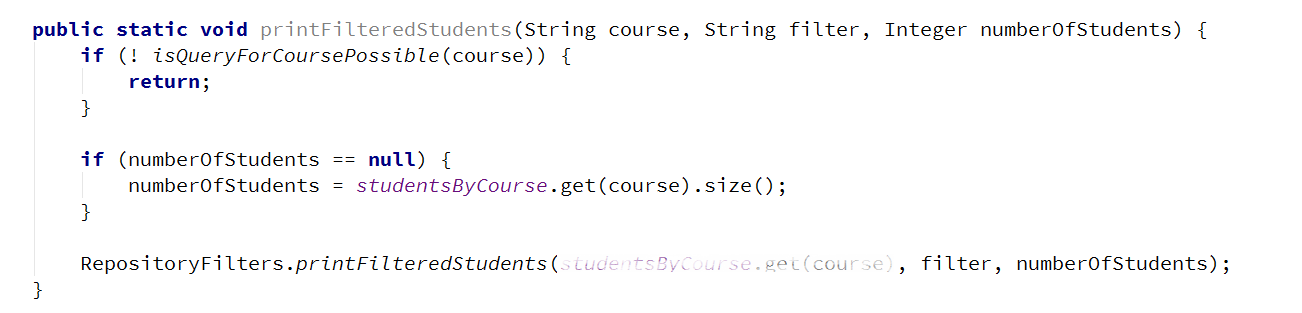
# Part III: Adding Functionality

## Extend the StudentsRepository Class

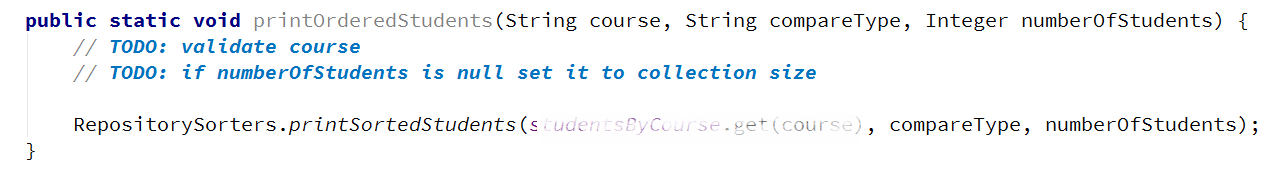
Since we are going to use the hash map from the StudentsRepository class and it is private, we can easily take all that we need from the StudentsRepository by using it as a mediator between the command interpreter and the filters/sorters. So what we are going to make are two methods in this class. One called printFilteredStudents() and one printOrderedStudents(). The filter follows the following signature:



Now we can take advantage of Java primitive type boxing. You already know that the Integer class is nullable so if we want, we can call the method from the command with null instead of a number and get all students from a given course, not a fixed number. Within the method we need to verify that the course is valid and if the value of the numberOfStudents variable is null we will set it to the size of the hash map, so we can take all students. After that we just need to call the corresponding method that we already created in the RepositoryFilters class. And the method should look like this:



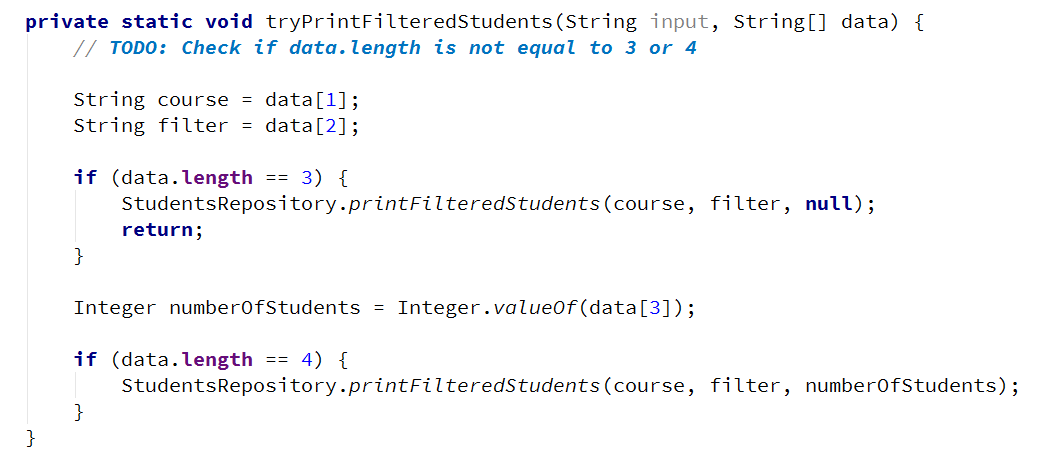
printOrderedStudents() method is exactly the same but we will call it from RepositorySorters class so try to complete the TODOs by yoursef:



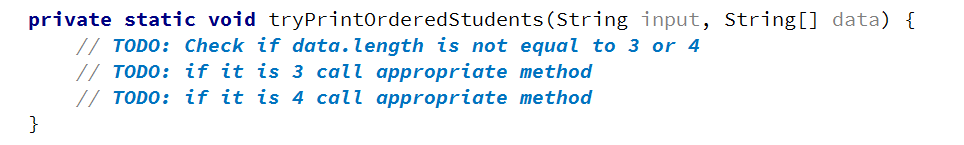
## Extend the Command Interpreter

In the command interpreter we should make two methods called tryPrintFilteredStudents() and tryPrintOrderStudents 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.

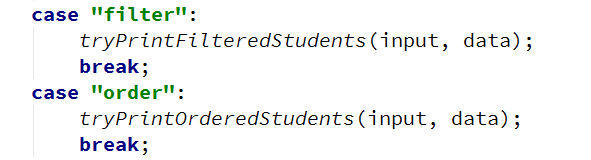
Let’s first look at the implementation of the tryPrintFilteredStudents() method. All we have to do there is check if the number of input parameters are 3 or 4 and if not, displayInvalidCommandMessage. If they are, we take the course name which is at index 1, the filter in lower case at index 2 and the number of students at index 3.



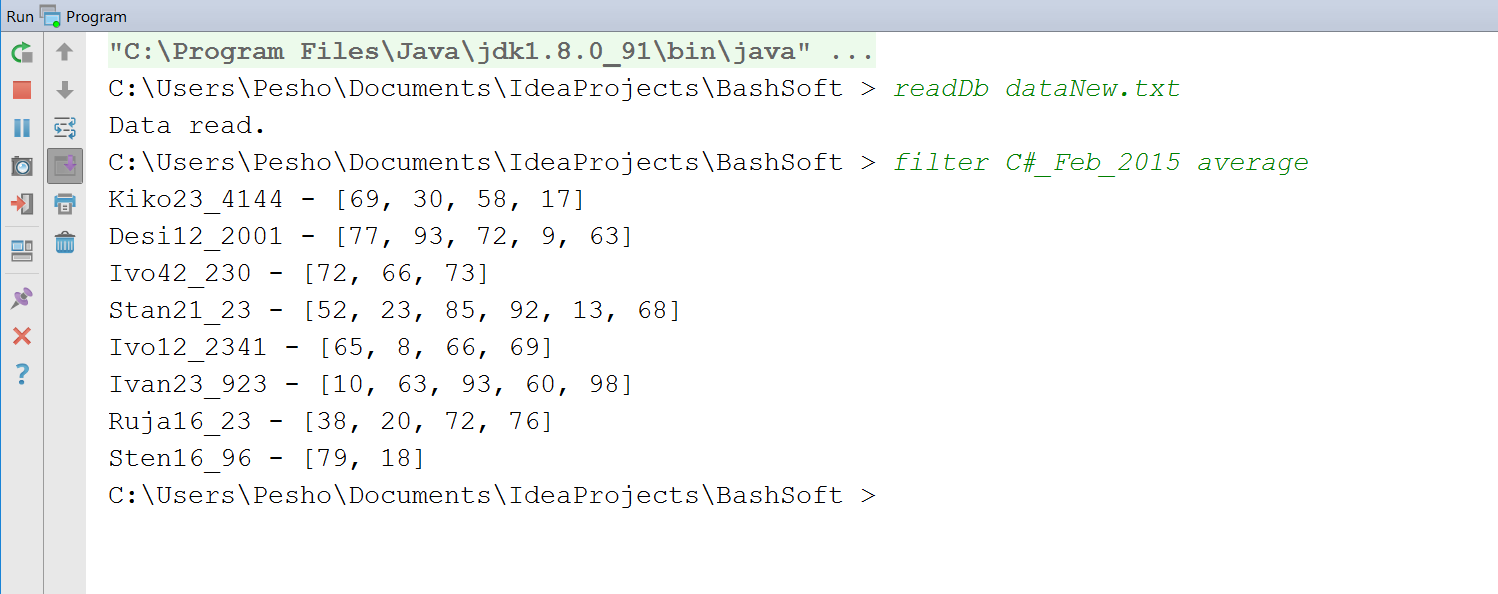
The situation with the tryPrintOrderedStudents() is the same the implementation of this method is left to you.



Now if you’ve done everything and the situation in the switch case in the interpretCommand() 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.



You can now play with and test the rest of the functionality that we have just implemented!

Congratulations! You’ve successfully completed the lab exercise for Functional Programming.