# Lab: Defining Classes and Methods

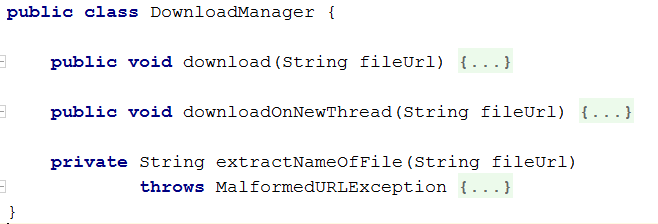
This document defines the lab overview for the ["Java OOP Basics" course @ Software University](https://softuni.bg/java-basics-oop). Please submit your solutions (source code) of all below described problems at the end of the course at [softuni.bg](https://softuni.bg/trainings/1375/java-basics-oop-june-2016).

# Introduction

We added a lot of functionality during the last course. However the code we wrote was only using static methods. Now the time has come for us to start following the principles of writing good OOP code. We are going to start by replacing some of the static members with instance ones. Note that we should start from the classes that **don't depend on any others**.

## Refactoring the Download Manager

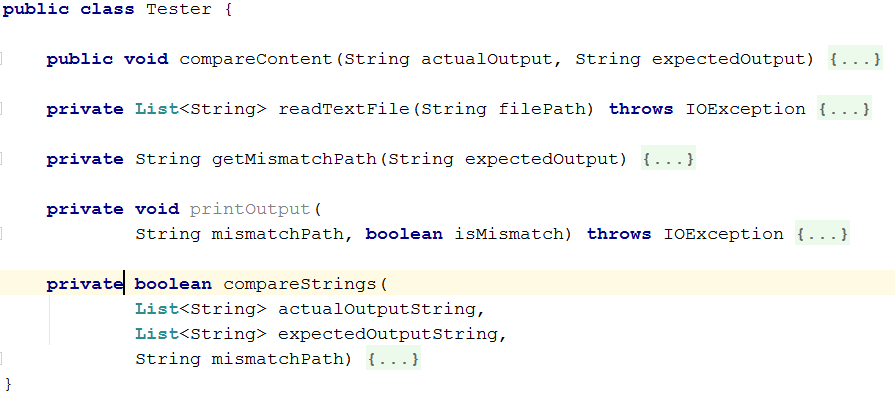
First we need to make the class and its methods non-static. We won't make a constructor because we can use the default one.



Now if you try to compile the project it should give you a big nasty error list. We must finish refactoring the other classes before you can run it again.

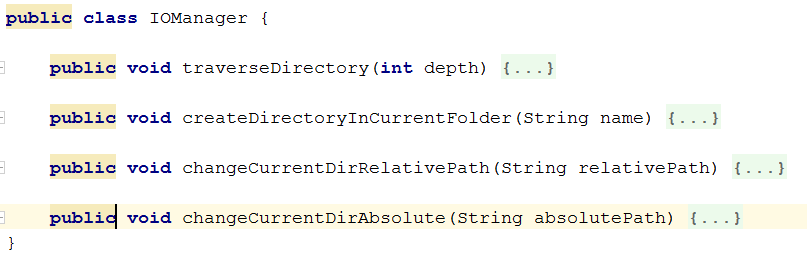
## Refactoring the Tester

Here there is also no need of a constructor so the only thing we have to do is make everything non-static.



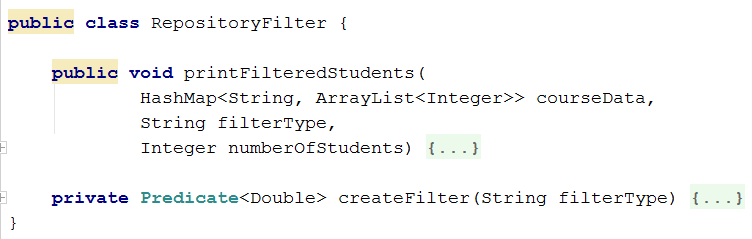
## Refactoring the IOManager

Same deal as the Tester class:



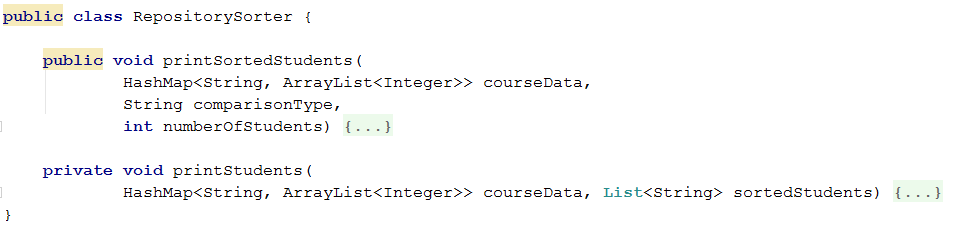
## Refactoring the RepositoryFilters

First rename the class to **RepositoyFilter** (without the s). We do this because now it is an instance class, instead of a static one. Also remove static from everywhere again.



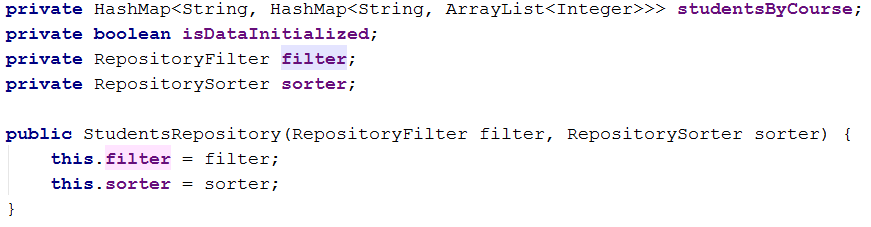
## Refactoring the RepositorySorters

Basically the same thing we did with the **RepositoryFilter** class.

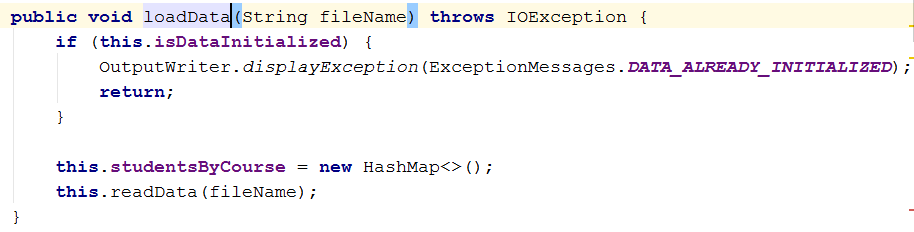


## Refactoring the StudentsRepository

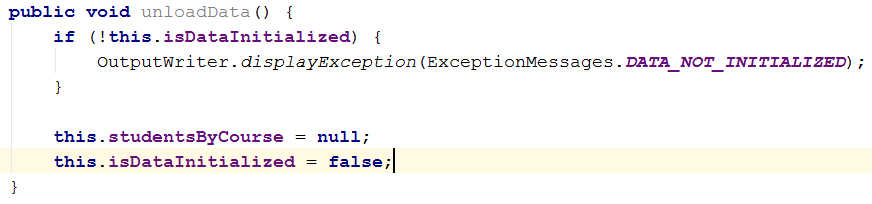
First remove every static word you see. Since we've changed the RepositoryFilter and Sorter we now have to make fields of these classes in the **StudentsRepository.** And we can make instances of them in the constructor:



Now we need to rename the **initializeData** method to **loadData** and remove the initialization from there, otherwise its implementation stays the same.



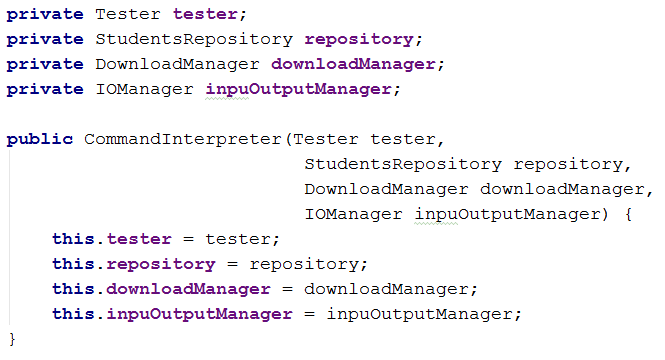
We will also create an **unloadData** method which will do the exactly what is says:



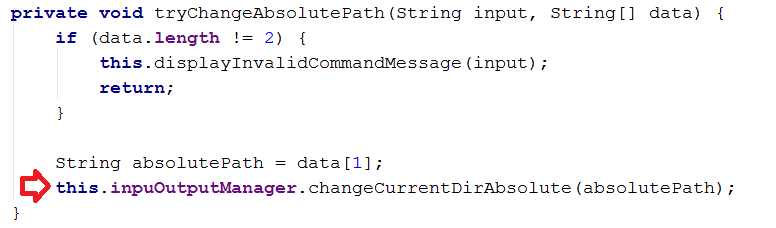
## Refactoring the CommandInterpreter

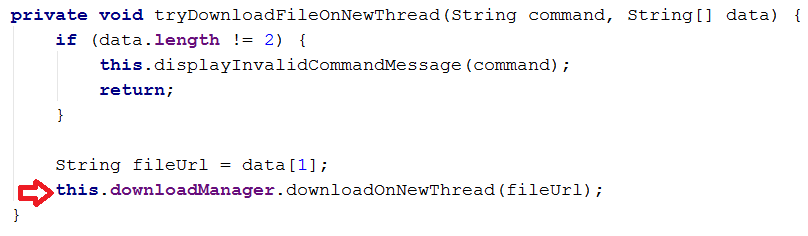
As always - start with removing every static word you find *(hint - use your IDEs find and replace all functionality and don't do it by hand)*. As you can see there are many errors in the error list of this file, so now we have to fix all of them. They appear because until now we've used only static classes and now we need to replace them with instances of these classes.

To do this we will create **fields** in the **CommandInterpreter** and **set** them in its **constructor**:

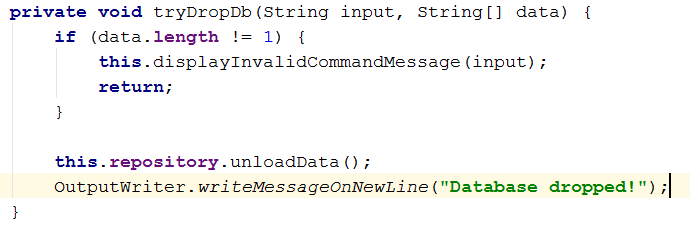


So now fix the method calls. Here are a few examples, do the others **yourself**:





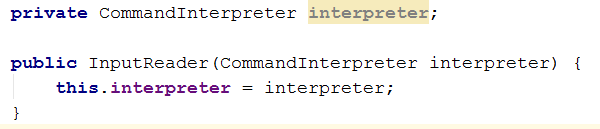
We also need to create a new command called "**dropdb**", so add such a case to the switch with the according method call. Then create the according method.



## Refactoring the Input Reader and the Main method

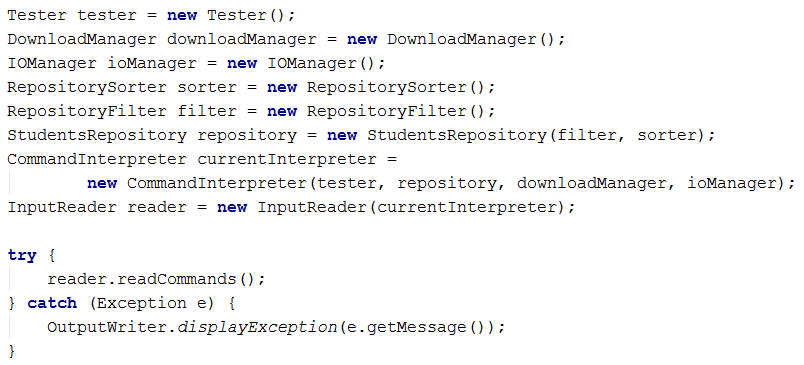
Start with the thing we always start with (psst… static).

Also we have to make a **constructor** for the class. It will receive as parameters a **CommandInterpreter**. We also have to make an appropriate field.



By now you shouldn't have any errors related to our classes.

In the Main method of our application we need to initialize everything trough its constructor.



## Creating a class Student

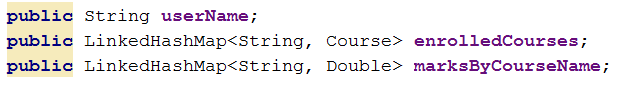
Before we start with the class - create a Folder which will hold all our models.



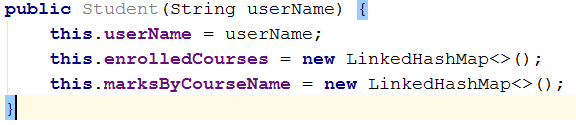
It will have fields (for now you can make them public, we will fix that in the encapsulation lab):

* userName
* enrolledCourses
* marksByCourseName

Try to make them yourself before looking at the screenshot.

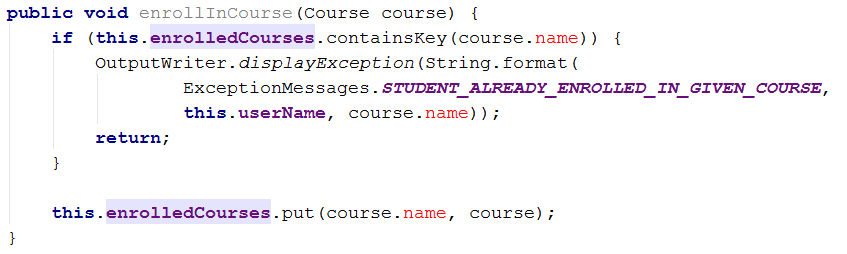


Make a **Constructor** which **initializes** all the **fields**.



Our class has the following **Methods**:

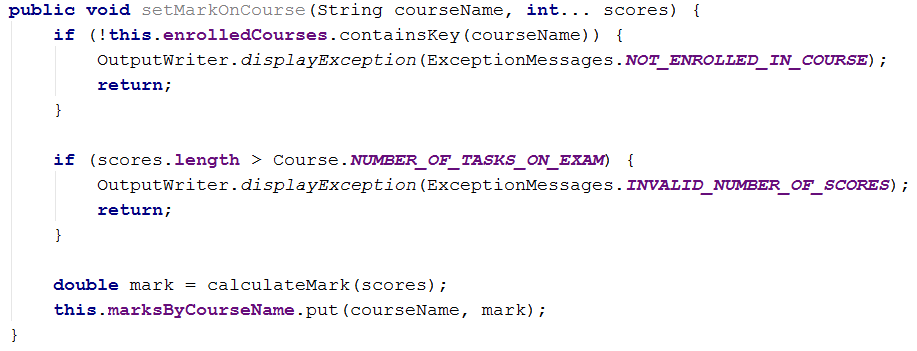
* enrollInCourse - used for enrolling the current student in a certain course. Notice in the code bellow how we first check for the exceptional case(s). This is part of a programming approach called "Defensive programming". The message should be "The {0} already exists in {1}."



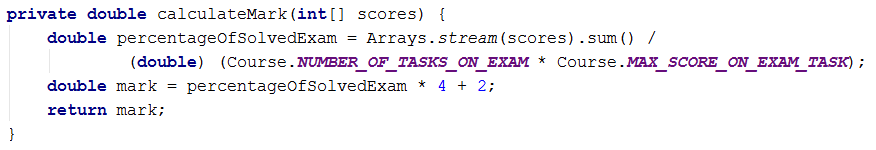
* setMarksInCourse - used for setting the current students' average mark in a certain course. Notice how we use a **params** array to pass as many scores as we want to the method.

The message for **NOT\_ENROLLED\_IN\_COURSE** is: "Student must be enrolled in a course before you set his mark."

The message for **INVALID\_NUMBER\_OF\_SCORES** is: "The number of scores for the given course is greater than the possible."



* calculateMark - this is only a helper method to calculate the average mark from all the scores we get. As such we can leave it to be private. We use a certain formula to do this:



## Creating a class Course

It will have fields:

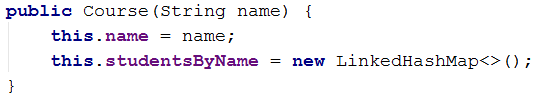
* name
* studentsByName



Constant fields:

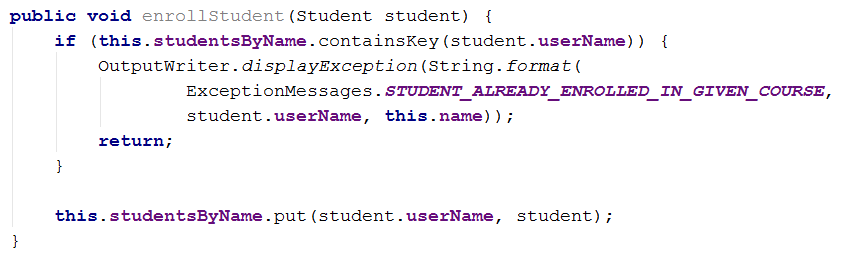


Constructor:



Methods:

* EnrollStudent - this method will enroll a certain student in the current course.



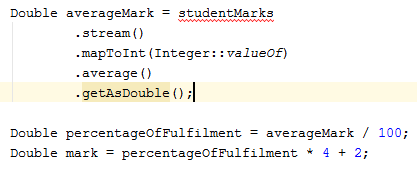
## Reworking the RepositoryFilter class

Now that we've made ourselves nice classes for Students and Courses we can use them to simplify all the classes in the Repository folder. Let's start with the RepositoryFilter:

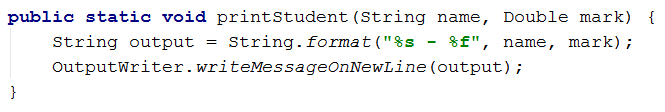
First off we need to change the Map that we receive as a parameter in the printFilteredStudents method (both public and private ones).The value of the map will now be simply a **double** which will represent the final mark of the student in the wanted course. We will also make the name of the map more descriptive:



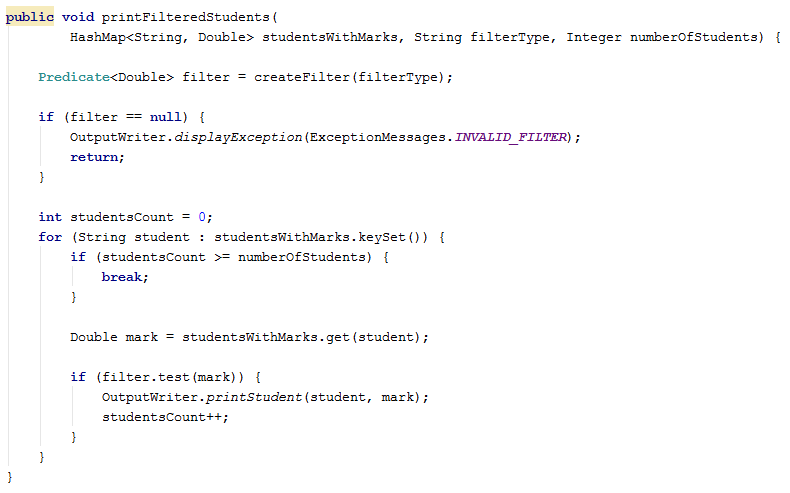
Do the same with the private method. Since we are going to receive a student with his mark we don't need to calculate the mark here. So we remove that code:



After you've deleted them you will notice that the printStudent command gives an error. That's because it waits for a String and List<Integer> so we need to change that to String and Double:

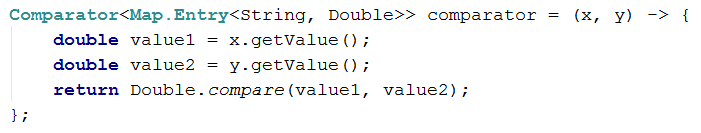


Now we need to return to the RepositoryFilter and give the printStudent method its corresponding arguments. Finally the method should look like this:



## Reworking the RepositorySorter class

First change the map parameter in the methods the same way as in the Filter. Next we want to change the lambda expression for ordering to match our new Map.Entry:



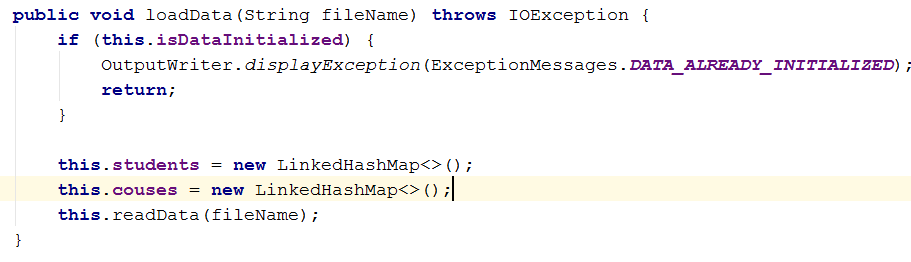
Fix the rest of the errors in the class yourself.

## Reworking the StudentsRepository class

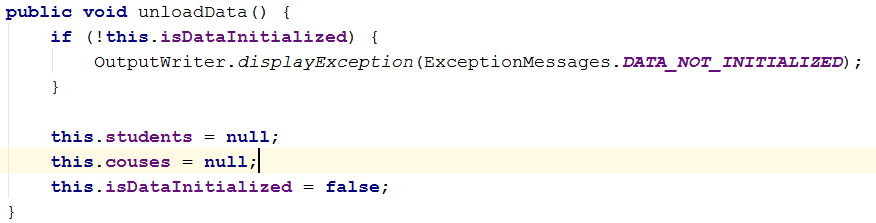
First off we need to delete the whole structure where we keep our data, because we will use our new shiny classes instead. The new data structure will be of two Maps: one for the courses (courseName -> Course) and the other for the students (studentName -> Student).



Next thing we need to do is initialize the two structures in the **loadData** method. Here is an example:



In the **unloadData** method we will do the exact opposite - set them to null.

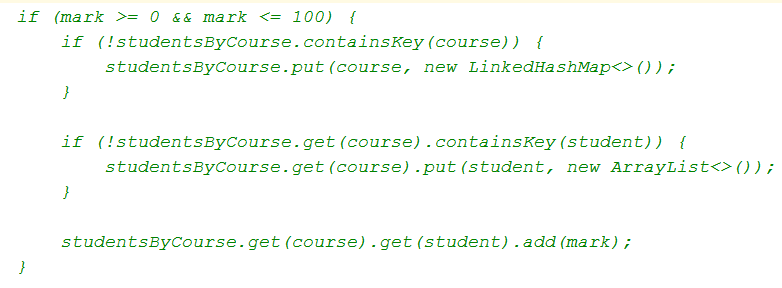


Next stop is the **readData** method. First change the regular expression because we've made some changes to the database. Now we will have students with both first and last name. Also, they can have from 1 to 5 scores from tasks. Here is the new regex: ([A-Z][a-zA-Z#\+]\*\_[A-Z][a-z]{2}\_\d{4})\s+([A-Za-z]+\d{2}\_\d{2,4})\s([\s0-9]+)

Now since we're changing the way we get the scores for each task we need to handle the parsing of the third group of the match. **Until now we had an Integer that held the actual value after the parsing. Replace it with a String which will keep all the numbers from the third group.**

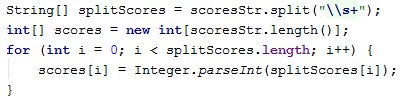


You can delete the following code block:

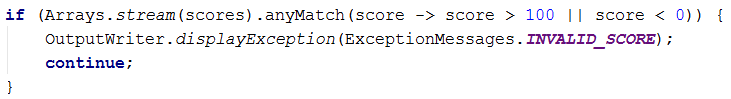


Since we are going to make some unprotected parsing you are going to make a new try/catch block. The code in the try block will do the following:

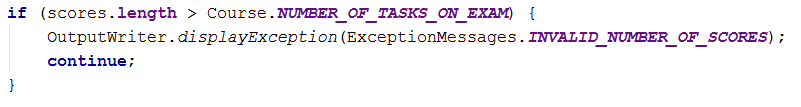
* Try to split, parse and collect the string in an integer array.



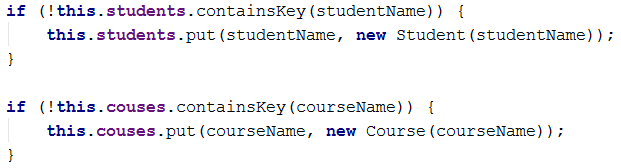
* Check whether any score is above 100 or below 0 and if so display the exception message: "The number for the score you've entered is not in the range of 0 - 100".



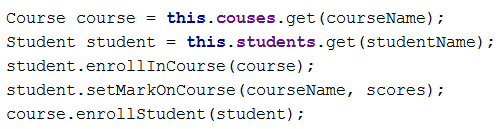
* Check whether the scores are more that the maximum NumberOfTasksOnExam:



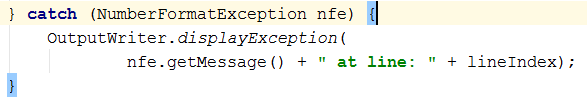
* Check if both of our data structures contain their corresponding key and if not add it with according new object:



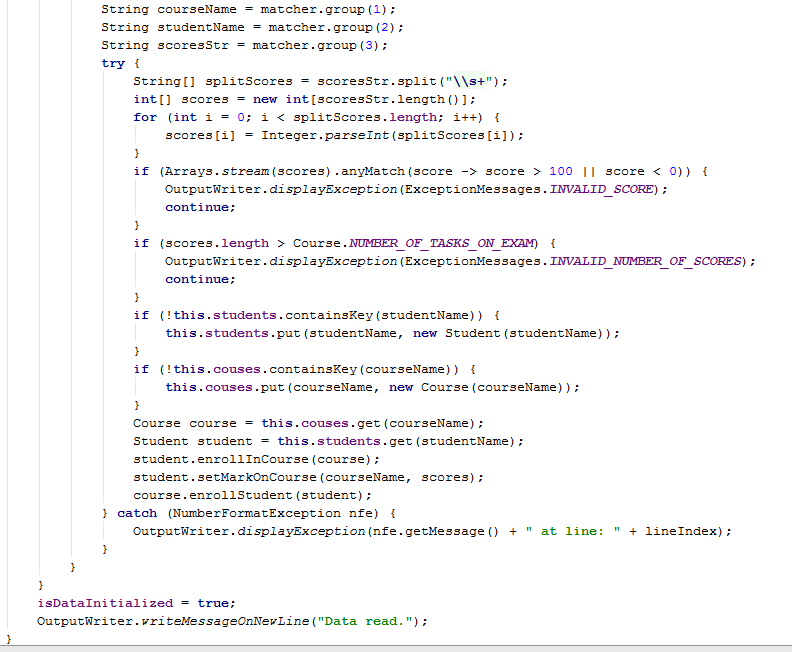
* Finally call the methods we made in the Course and Student classes so that our base is up.



In the catch block we will display the **FormatException** we might throw.



At the end the altered part of the method (without the regex - don't forget to change it too) should look something like this:



Now we have to change the **isQueryForCoursePossible** method. Change the data structure in which we have to check whether such a course exists:

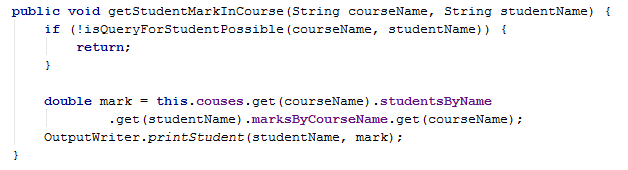


In the **isQueryForStudentPossible** we should basically do the same, but first we should take the wanted course and from it the studentsByName structure. Then we check whether it contains the wanted student:



In the **getStudentMarksInCourse** method we need to change only the printing according to the new way we print a student. The tricky part is getting the value because we need to go deep. You can also change the name of the method because now we only get one mark - **getStudentMarkInCourse**

**wanted course -> wanted student inside course -> chosen students mark for course**



Next up is the **getStudentsByCourse** method **-** here the for each loop needs to iterate over the students by name in the corresponding course:



And in the cycle body we can delete the print student method call because we can re-use the previous method.

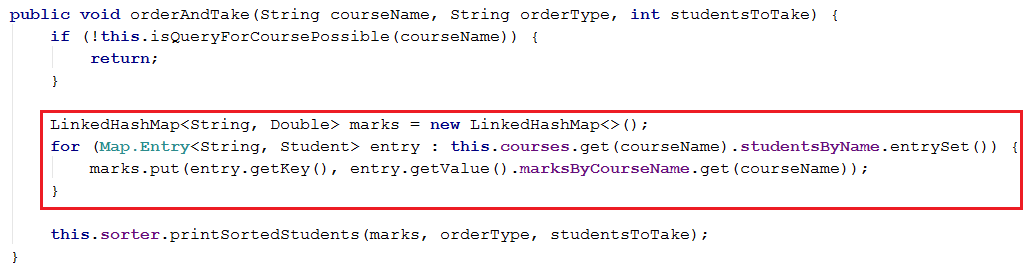


The **printFilteredStudents** and **printOrderedStudents** methods we probably won't need any more so just delete them.

The **2 parameter** **filterAndTake** and **orderAndTake** methods just need to have their studentsToTake variables changed to take the appropriate size:



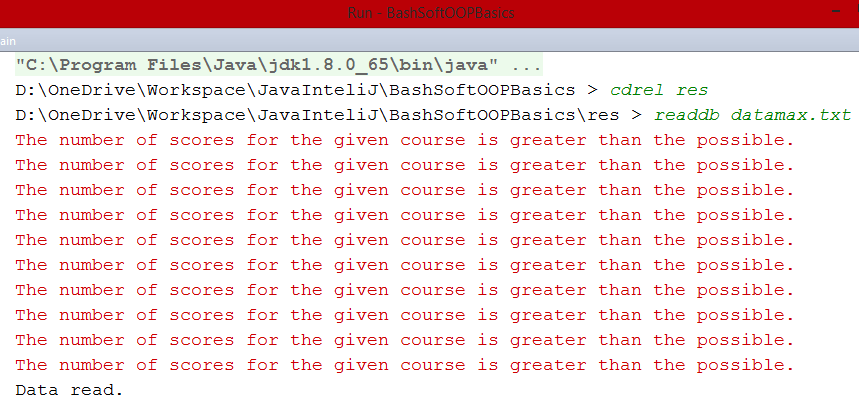
Their **3 parameter** counterparts need fixes in order to pass the right map with the marks to their corresponding **filter** or **sorter** class methos. We first need to extract the marks. Here is how we do that:



The last method we need to change in this class is the **filterAndTake** Method. We make the same changes as the previous method.

## Testing

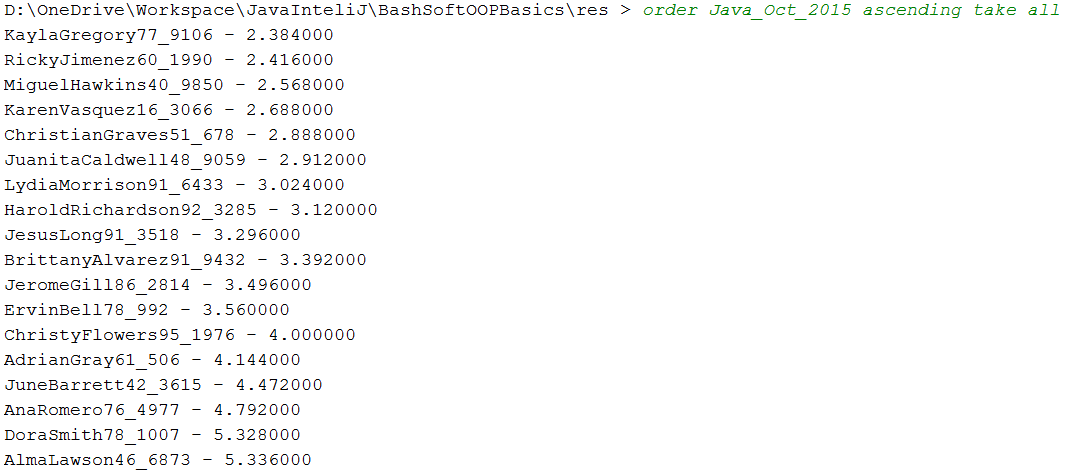
Finally we shouldn’t have any more errors and we should easily be able to run the project. Let's test if we broke something. If you use the provided **datamax.txt** file the result of reading the database should look something like this:



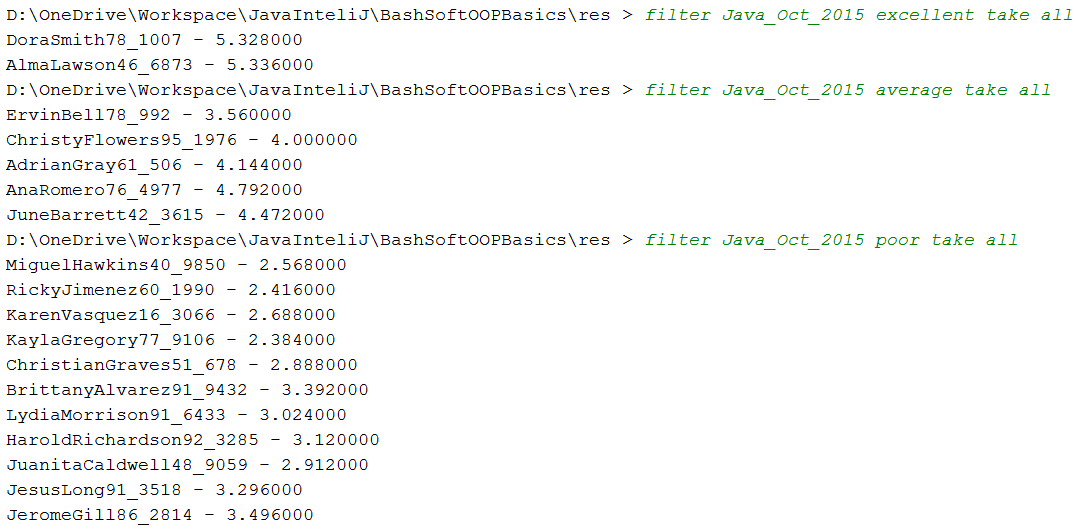
The errors here are normal. They appear because there are some entries in the end of the file which contain more than 5 scores on tasks. If you like, you can delete them. They were just for the sake of testing.

You can also test other functionality like:

* Sorting



* Filtering



Congratulations you completed the first Lab in the OOP Basics course!