# Lab: Defining Classes and Methods

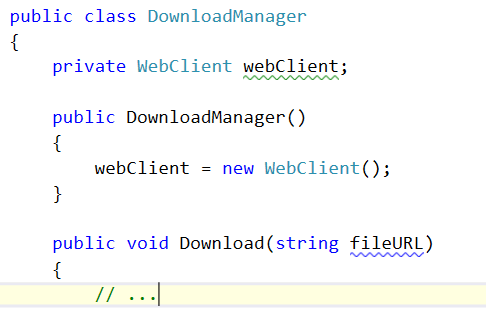
This document defines the lab overview for the ["C# OOP Basics" course @ Software University](https://softuni.bg/c-sharp-basics-oop). Please submit your solutions (source code) of all below described problems at the end of the course at [softuni.bg](https://softuni.bg/trainings/1373/c-sharp-basics-oop-june-2016).

# 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

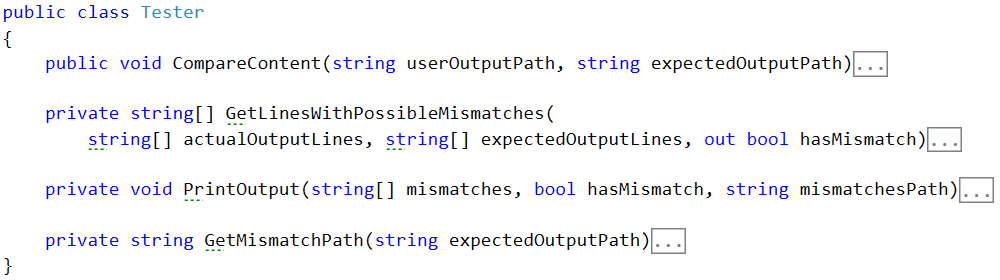
First we need to make the class and its methods **non-static**. Then we can make the **WebClient** into a field and initialize it in the constructor.



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.

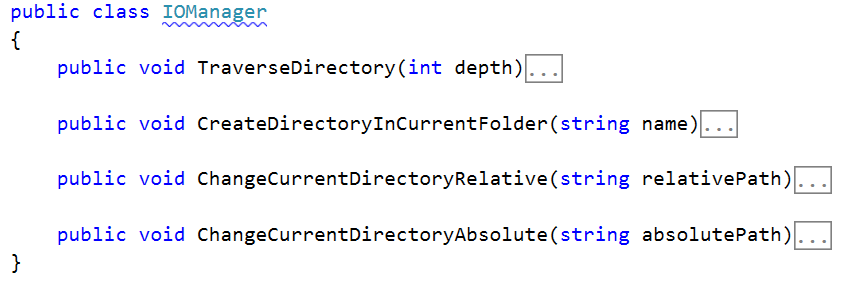
## Refactoring the Tester

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



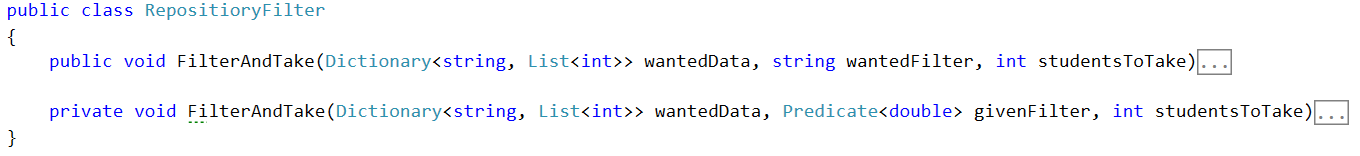
## Refactoring the IOManager

Same deal as the **Tester** class:



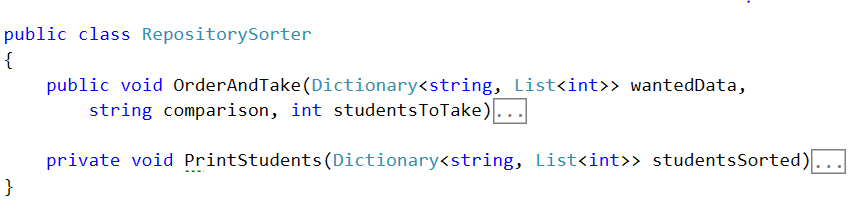
## Refactoring the RepositoryFilters

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.



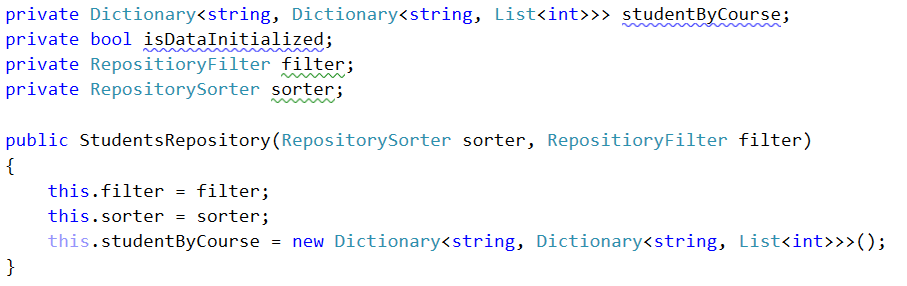
## Refactoring the RepositorySorters

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

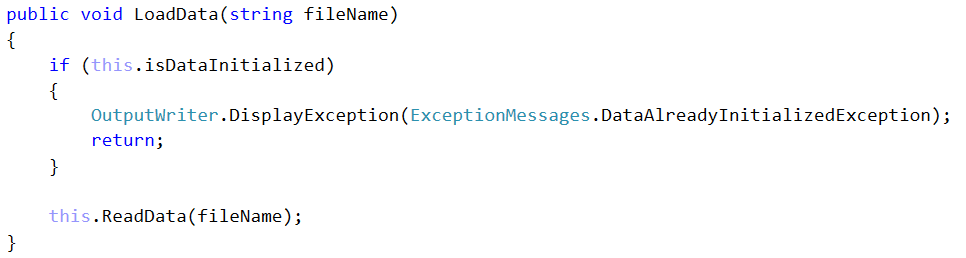


## Refactoring the StudentRepository

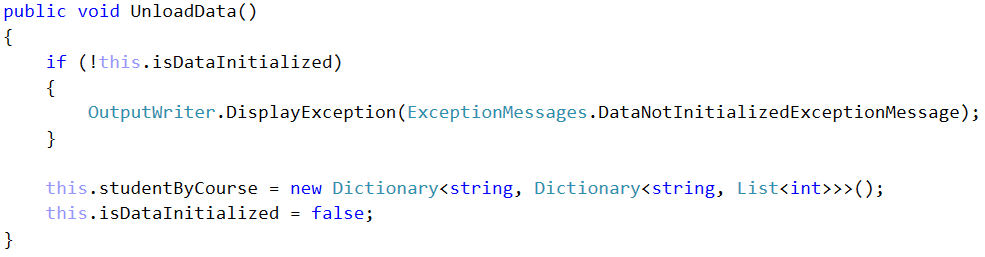
First remove every static word you see. Since we've changed the **RepositoryFilter** and **Sorter** we now have to make fields of these classes in the **StudentRepository.** And we can make instances of them in the constructor:



As you can see we have moved the initialization of the data structure to the constructor. So now we need to rename the **InitializeData** method to **LoadData** and remove the initialization from there, otherwise its implementation stays the same.



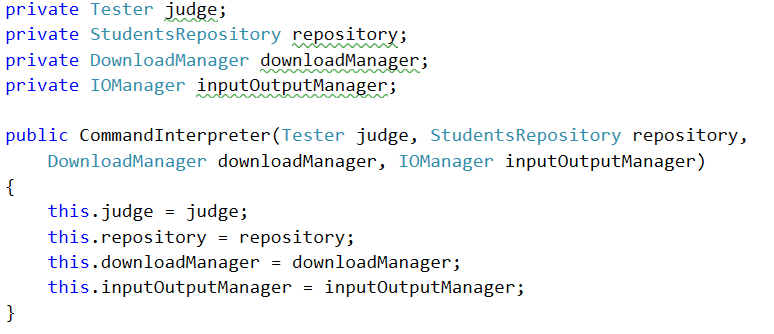
We will also create an **UnloadData** method which will do the exactly what is says:



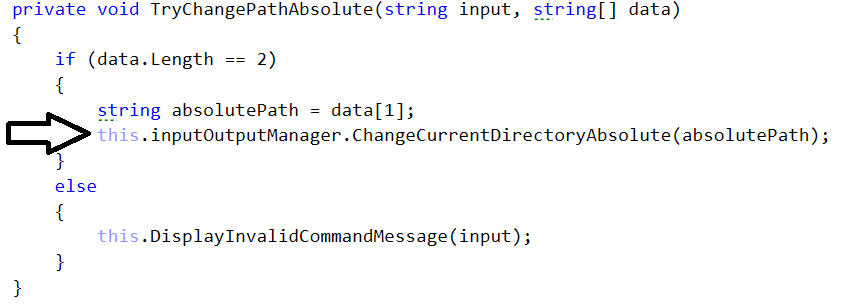
## Refactoring the CommandInterpreter

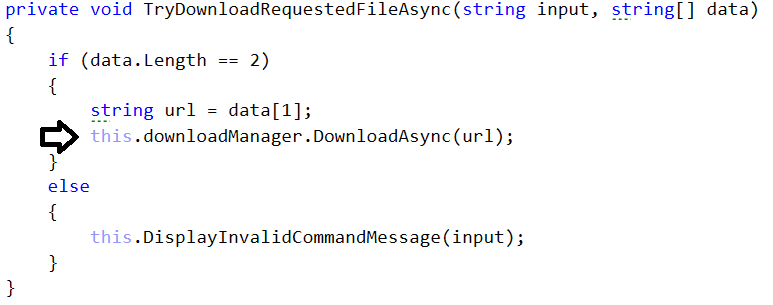
As always - start with removing every static word you find *(hint - use your IDEs find and replace all functionality and don't do it by hand)*. As you can see there are many errors in the error list of this file, so now we have to fix all of them. They appear because until now we've used only static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

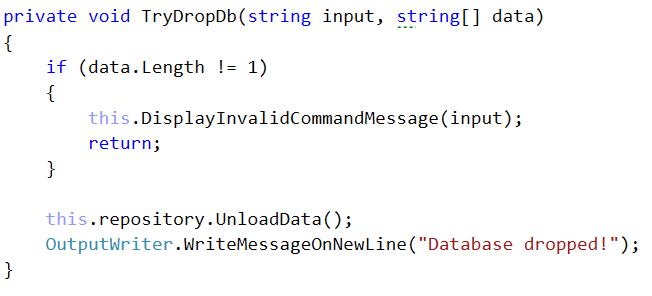


So now fix the method calls. Here are a few examples, do the others **yourself**:





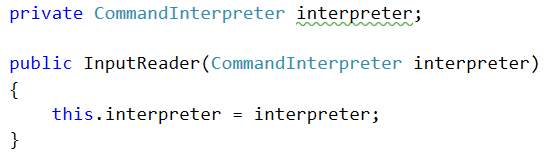
We also need to create a new command called "**dropdb**", so add such a case to the switch with the according method call. Then create the according method.



## Refactoring the Input Reader and the Main method

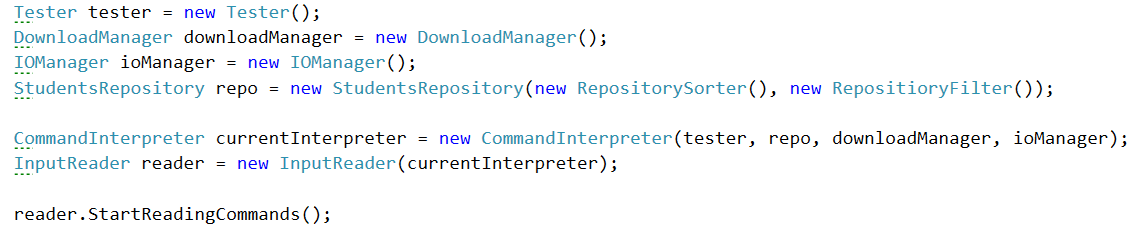
Start with the thing we always start with (psst… static).

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

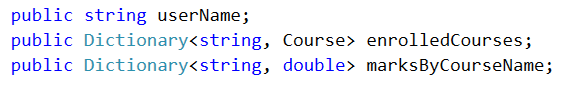
Before we start with the class - create a Folder which will hold all our models.

C:\Users\tilchev\Desktop\image_2016-06-22_13-56-07.png

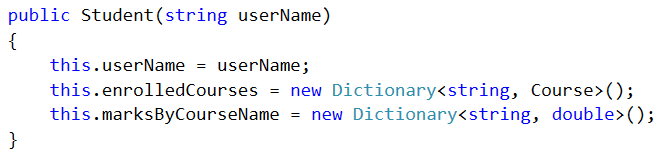
It will have fields (for now you can make them public, we will fix that in the encapsulation lab):

* userName
* enrolledCourses
* marksByCourseName

Try to make them yourself before looking at the screenshot.

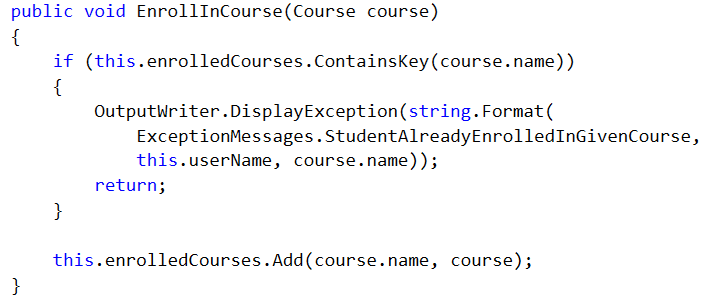


Make a **Constructor** which **initializes** all the **fields**.



Our class has the following **Methods**:

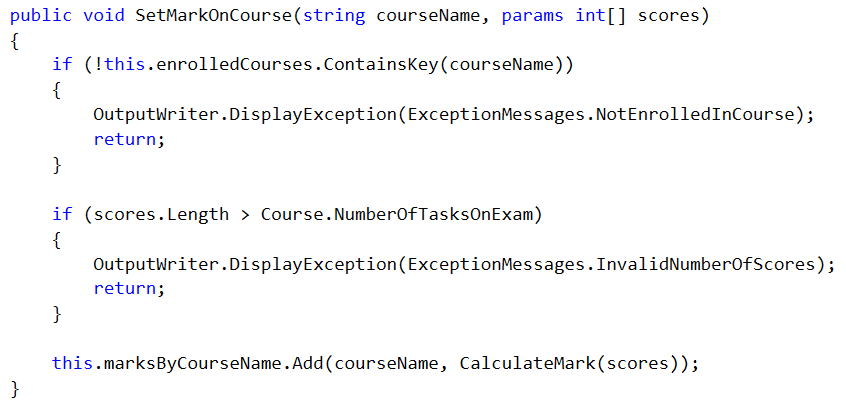
* **EnrollInCourse** - used for enrolling the current student in a certain course. Notice in the code bellow how we first check for the exceptional case(s). This is part of a programming approach called "Defensive programming". The message should be "The {0} already exists in {1}."



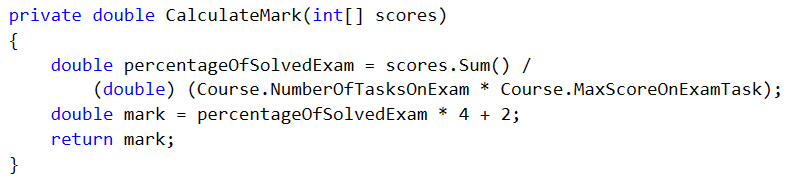
* **SetMarksInCourse** - used for setting the current students' average mark in a certain course. Notice how we use a **params** array to pass as many scores as we want to the method.

The message for **NotEnrolledInCourse** is: "Student must be enrolled in a course before you set his mark."

The message for **InvalidNumberOfScores** is: "The number of scores for the given course is greater than the possible."



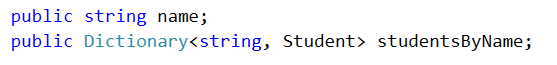
* **CalculateMark** - this is only a helper method to calculate the average mark from all the scores we get. As such we can leave it to be private. We use a certain formula to do this:



## Creating a class Course

It will have fields:

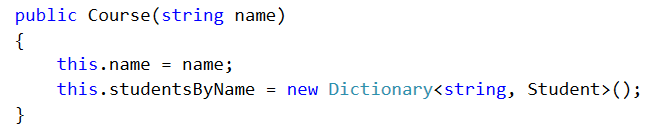
* name
* studentsByName



Constant fields:

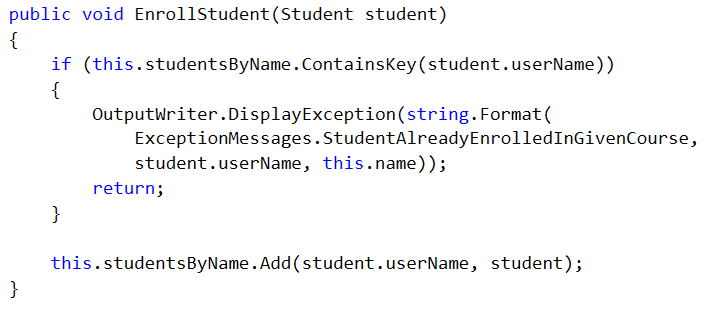


**Constructor**:



**Methods**:

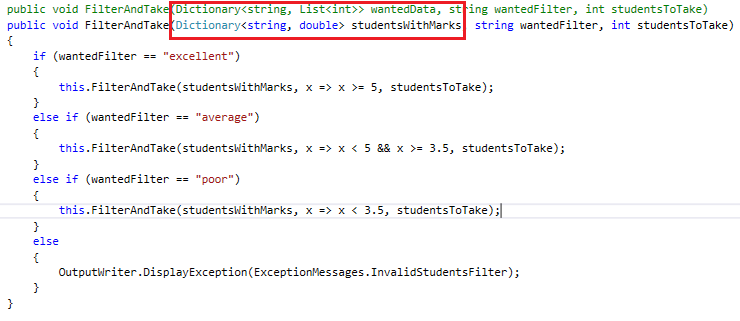
* **EnrollStudent** - this method will enroll a certain student in the current course.



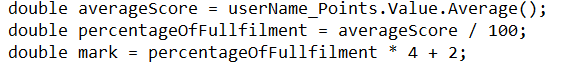
## Reworking the RepositoryFilter class

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. Lets start with the RepositoryFilter:

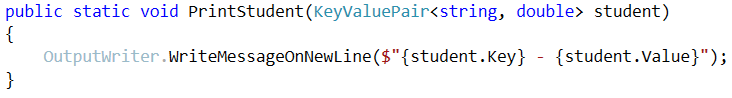
First off we need to change the Dictionary that we receive as a parameter in the **FilterAndTake** methods (both public and private ones).The value of the dictionary will now be simply a **double** which will represent the final mark of the student in the wanted course. We will also make the name of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



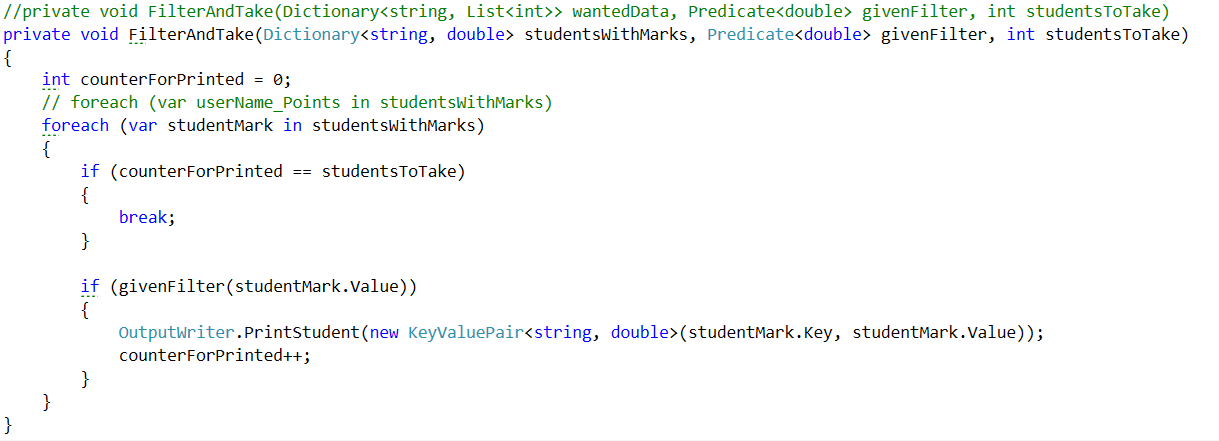
Do the same with the private method. Since we are going to receive a student with his mark we don't need to calculate the mark here. So we remove that code:



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

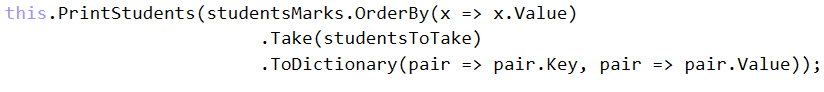


Now we need to return to the **RepositoryFilter** and give the **PrintStudent** method its corresponding arguments. Finally the method should look like this:



## Reworking the RepositorySorter class

First change the dictionary 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 **KeyValuePair**:



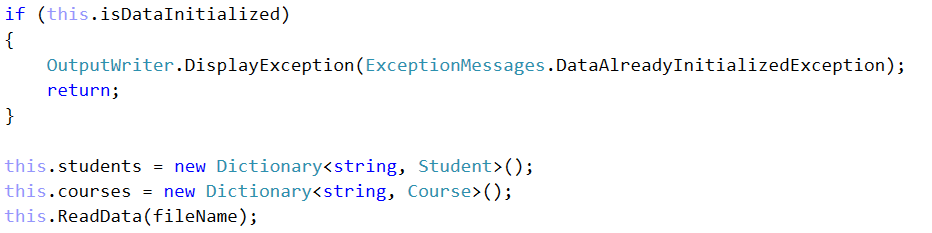
Fix the rest of the errors in the class yourself.

## Reworking the StudentsRepository class

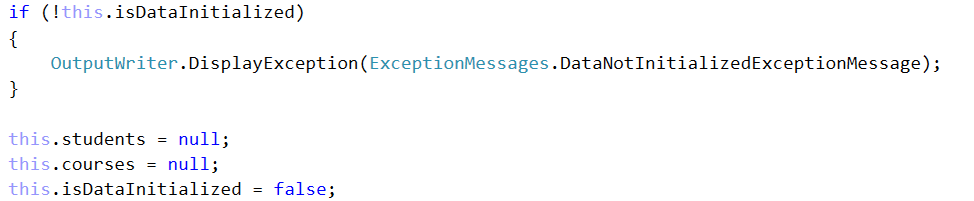
First off we need to delete the whole structure where we keep our data, because we will use our new shiny classes instead. The new data structure will be of two Dictionaries: one for the courses (**courseName -> Course**) and the other for the students (**studentName -> Student**).

C:\Users\tilchev\Desktop\image_2016-06-22_14-33-14.png

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

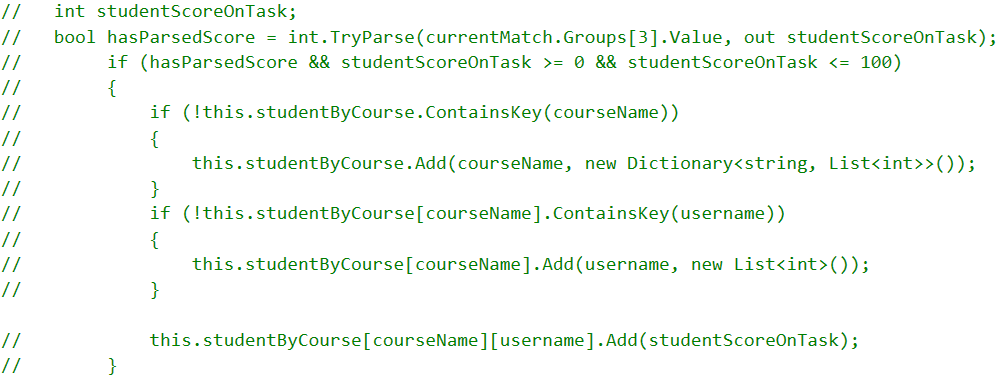


In the UnloadData method we will do the exact opposite - set them to null.



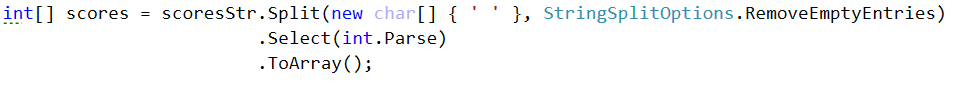
Next stop is the **ReadData** method. First change the regular expression because we've made some changes to the database. Now we will have students with both first and last name. Also, they can have from 1 to 5 scores from tasks. Here is the new regex: ([A-Z][a-zA-Z#\++]\*\_[A-Z][a-z]{2}\_\d{4})\s+([A-Za-z]+\d{2}\_\d{2,4})\s([\s0-9]+)

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 a boolean that told us whether the score has passed. It also had an Integer that held the actual value after the parsing. You can delete the following code block:

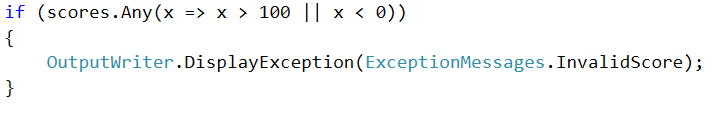


Now we have to take the third groups' value and save it in a string called scoresStr. Since we are going to make some unprotected parsing you are going to make a new try/catch block. The code in the try block will do the following:

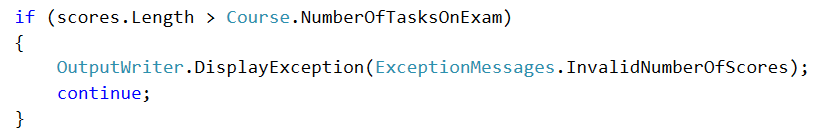
* Try to split, parse and collect the string in an integer array.



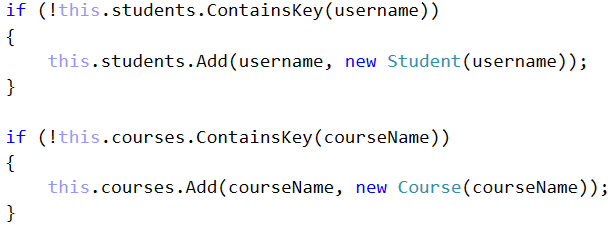
* Check whether any score is above 100 or below 0 and if so display the exception message: "The number for the score you've entered is not in the range of 0 - 100".



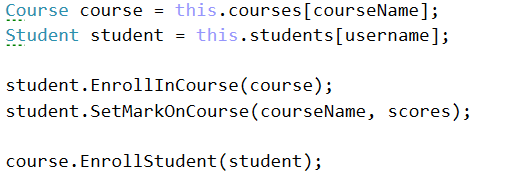
* Check whether the scores are more that the maximum **NumberOfTasksOnExam**



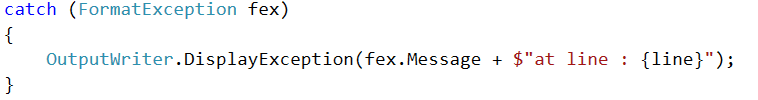
* Check if both of our data structures contain their corresponding key and if not add it with according new object:



* 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.



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 **GetStudentScoresFromCourse** method we need to change only the printing according to the new **KeyValuePair** we use for that. The tricky part is getting the value because we need to go deep.

wanted course -> wanted student inside course -> chosen students marks for course



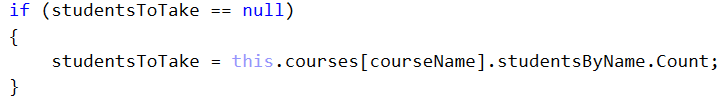
In the **GetAllStudentsFromCourse** **-** here the foreach needs to iterate over the students by name in the corresponding course:



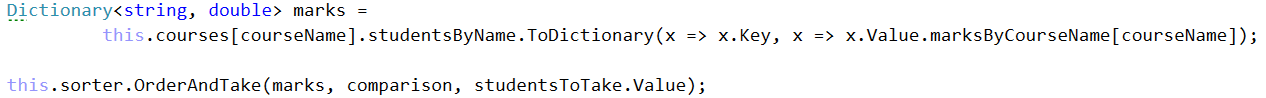
And in the foreach we can delete the print student method call because we can re-use the previous method.



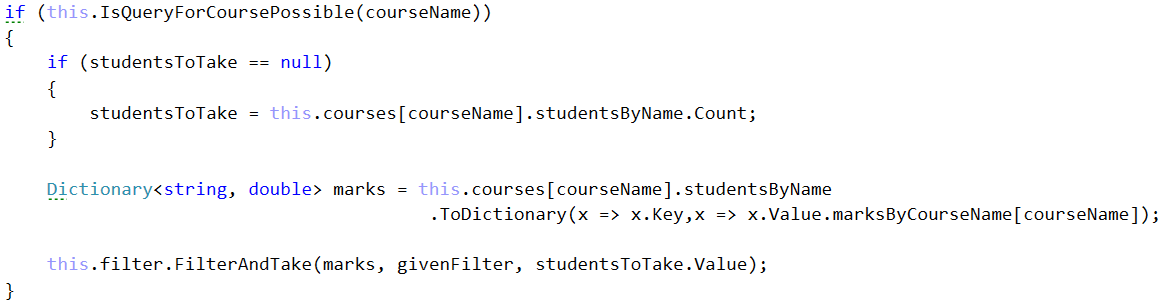
In the **OrderAndTake** method in its if-block where we check whether the students to take is null we should take the count of the students by name from the corresponding course.



In order to pass the dictionary with the marks in the order and take method in the sorter we first need to extract the marks. Here is how we do that - you must figure out why it's done like that on your own.

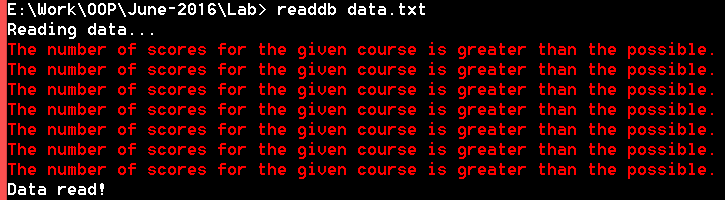


The last method we need to change in this class is the **FilterAndTake** Method. We make the same changes as the previous method.



Finally we shouldn’t have any more errors and we should easily be able to run the project.

The result of reading the database should look something like this:



The errors here are normal to appear. They are 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 we just for the sake of testing.