# Lab: Sets and Maps

This document defines the lab overview for the ["Java Advanced" course @ Software University](https://softuni.bg/trainings/1377/advanced-java-may-2016). Please submit your solutions (source code) of all below described problems at the end of the course at [softuni.bg](https://softuni.bg/trainings/1377/advanced-java-may-2016).

# Part I: Creating a Data Structure

## Creating some 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 such messages that are used often.  
The class should be named ExceptionMessages and is public. The only things we are going to put in this class are **public static final strings** with a given name and its corresponding message:

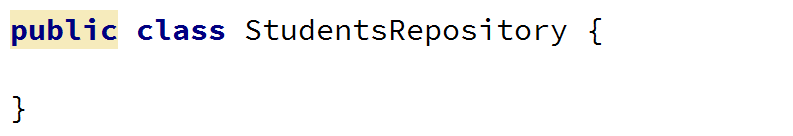


So from now on, every time we have to add a message you should follow the format described above.

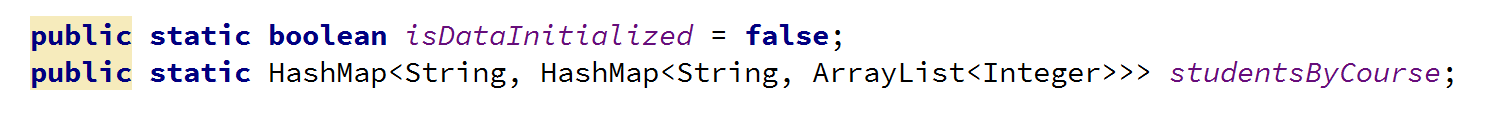
## Creating a data structure

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 open your project from the previous assignment and 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 needs to be static and public. By now you should be somewhere around here:



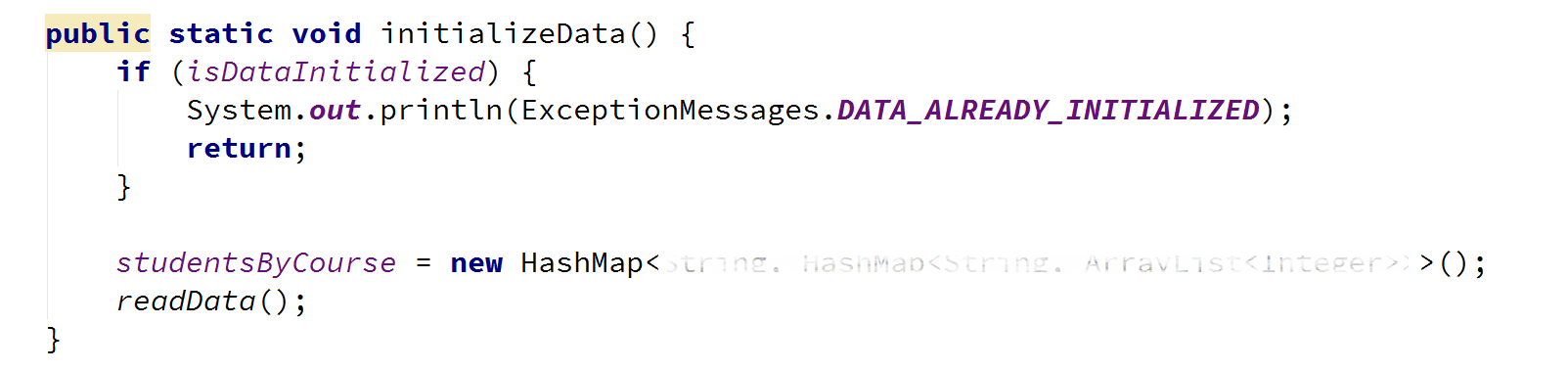
Now it is time to decide what data structure to define for our application in order to be able to make fast operations and have easy access to your data. Since we have to save different courses, the students in those courses have unique usernames and list of grades, we can save them in two nested dictionaries 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 that we haven’t put private in front of our data structure but you need not to be distracted by such things, you may fix it after the High Quality Code course.

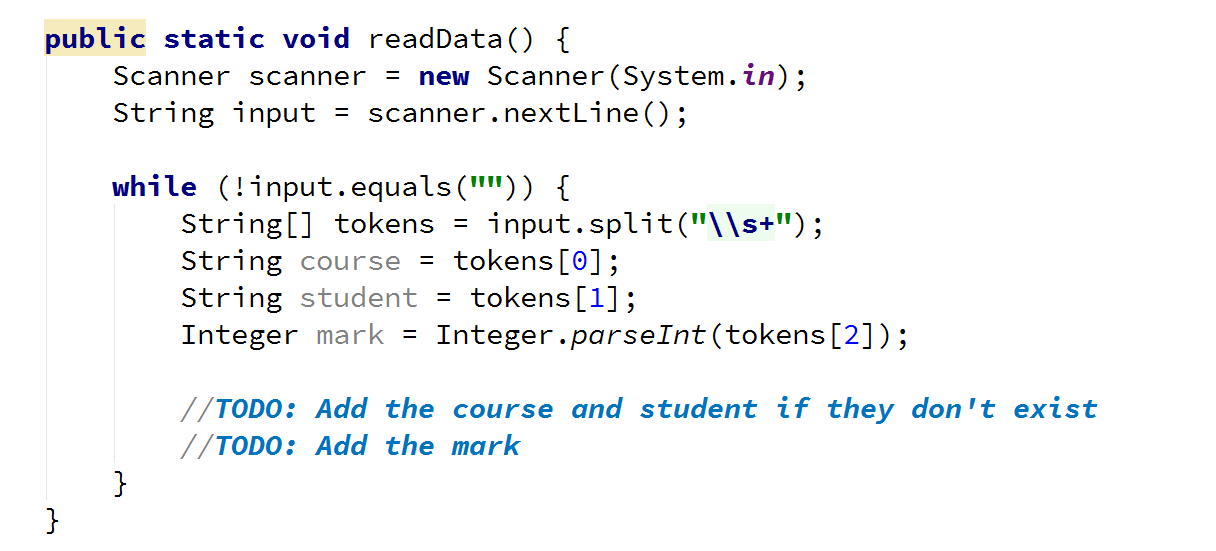
## Initializing and Saving our Data

Now 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** and sets our boolean variable to true.

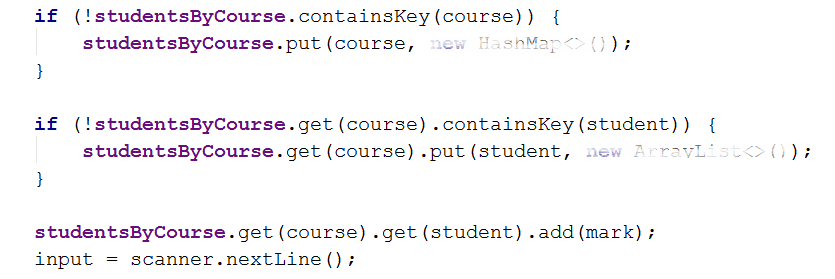


Now it’s time to fill the private readData() method (the data will always be valid). It is private because we do not want to be reachable out of our class.

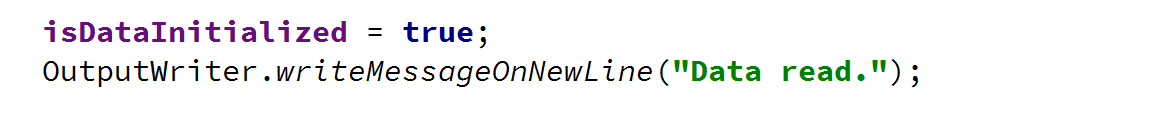
All we are going to do, is to **read from the console until an empty line is read.** The data you need to read is in the data.txt file given with the current document. We also need to extract the information we need and save it in our data structure.



Now we need to check **if our course and student already exists in the data**. If we don’t do this, we are going 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.



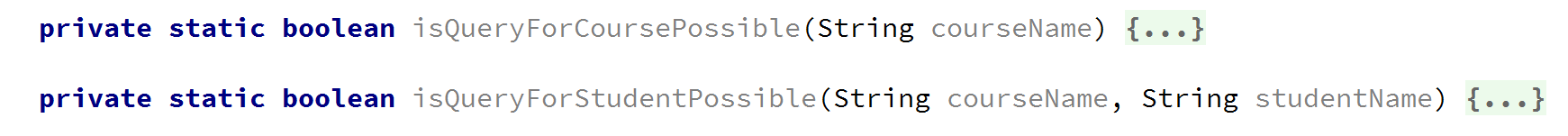
Finally, after the while loop we need to **set the isDataInitialized to true** and print that the data is read on a new line.



## Making Security Checks

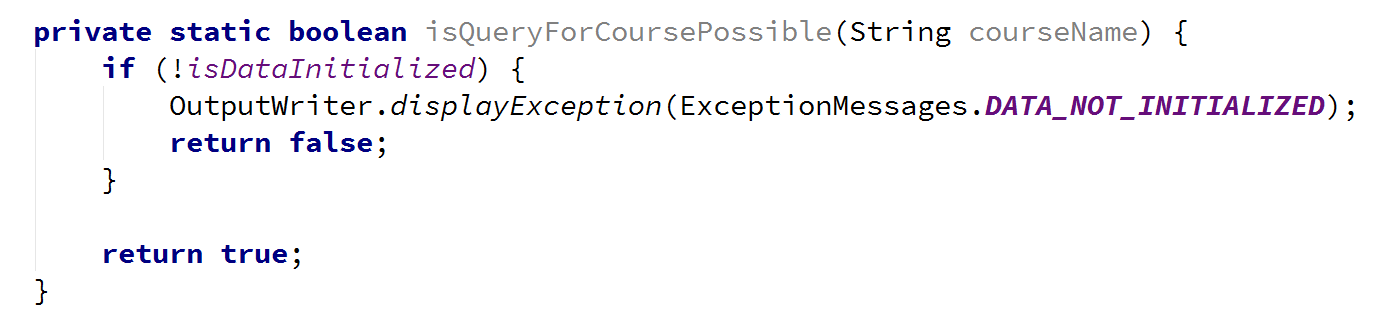
Since we are going to **make queries to the data structure** in this BashSoft piece and also in some others along the track of the course, it would be a good idea to make a method for the security checks in order to retrieve some data for a given course or for a given student in some course. This way we will save ourselves the writing of the 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 takes one parameter (the course name) and the second one takes two parameters (the course name and the user name of the student). Their signature should look like this:

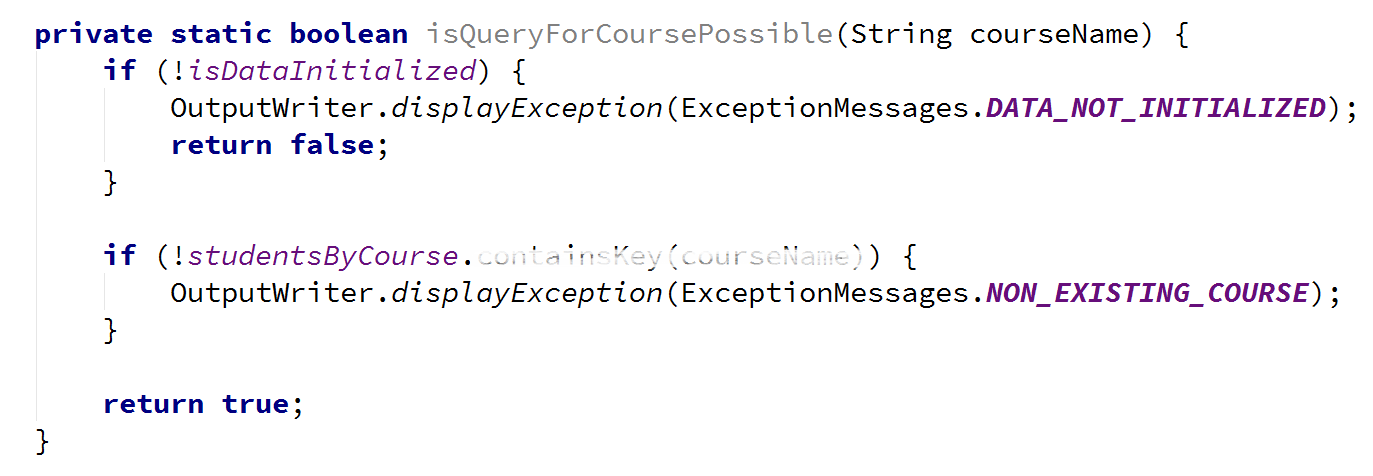


Since the second method will have to do half of the checks for the course that are done in the first method we will reuse the first one and for this reason we are starting with its implementation.

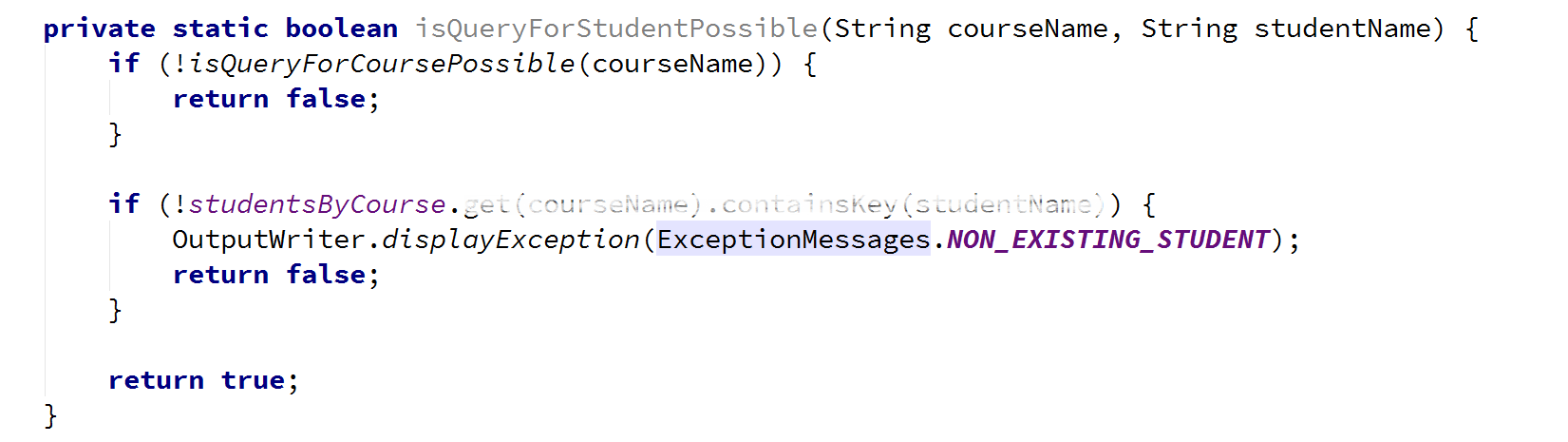
First thing we need to check in order to search for the given course name, is whether the data structure is actually initialized. If you haven’t already, create a new message in the ExceptionsMessages that is called DATA\_NOT\_INITIALIZED and its message should be: “Data is not initialized.”



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. Now we have to add this check, if the data structure does not contain the key, we return and we display an exception that we’ll need to add in the ExceptionsMessages called NON\_EXISTENT\_COURSE with the following message: “Course does not exist.” Otherwise we return true.



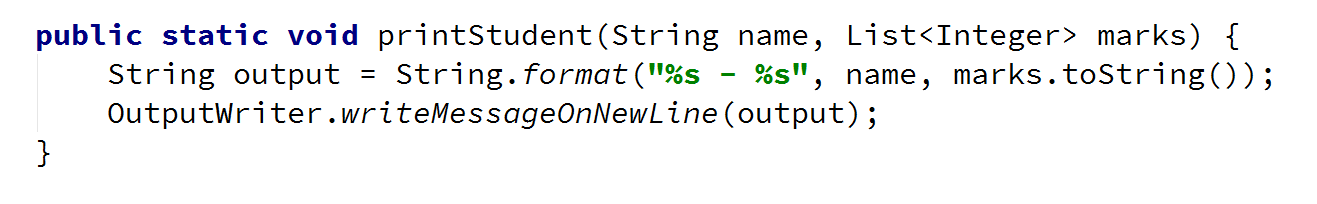
Now that we’ve implemented the first method for the checks, it’s time for its sidekick. As we’ve said we will reuse the check from the first method and also **add a new check** for whether the given student user name exists in the data structure of the university. If it is present, we return true, if it is not we **display** an **exception** that we’ll need to add in the **ExceptionsMessages** called **NON\_EXISTING\_STUDENT** with the following message: “User does not exist.” and return false:



# Part II: Saving Information

## Displaying a Student Entry

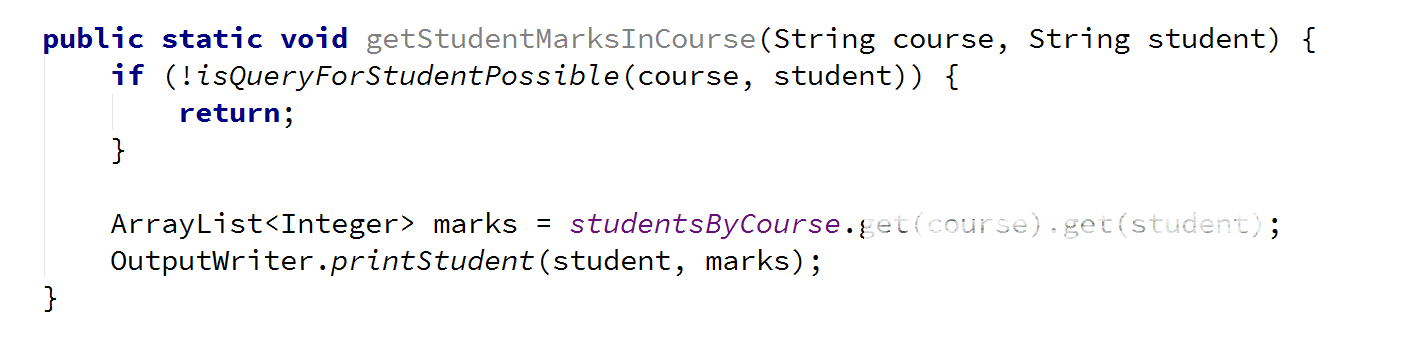
Before we continue with reading of the data, there is just one last thing we might add in order to make our life easier. Since now we have two methods that are going to display student somehow and we might have more things that need to display student after a filter or a sorting for example, by implementing 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 String (student) and ArrayList<Integer> (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 things are displayed on the standard output. The implementation of the method should be as follows:



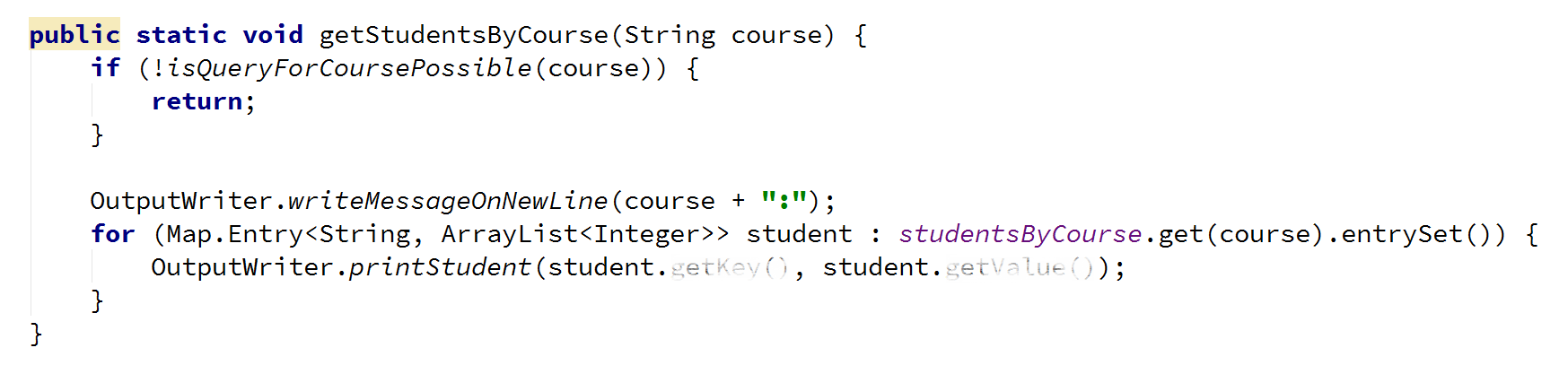
Now that we are ready with the displaying of a student are ready to proceed with the actual reading of the data from the data structure.

## Reading Information from Our Data

The most basic operations for extracting information will be to **get all the scores of a student in a course and get all students from a given course**. We need to define two methods. Let’s start with the first one. It should be public static with return type void. Its 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:

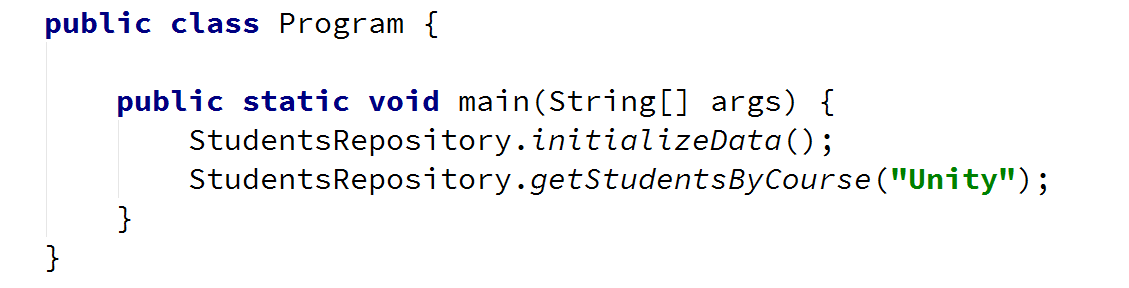


The other method is a little bit more complex. It gets all students from a given course if the query for course is possible and then write the course name followed by two dots and after that we foreach the collection with students from the given course and print all of the students:

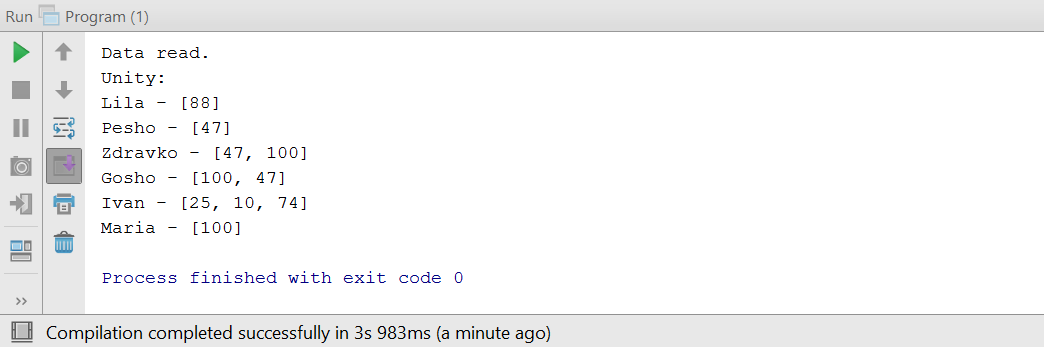


## Testing your code

If you put the given input (from data.txt) and get all the students from the unity course, query should look like this:



And the result should look like this:



Congratulations! You’ve successfully completed the lab exercises!