Problem 1. Creating a set of error messages to display if a function produces such an error:

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** and **static**. The only things we are going to **put** in this class are **public const strings with** a given **name and** it's corresponding **message**:

```
public static class ExceptionMessages
{
    public const string ExampleExceptionMessage = "Example messsage!";
```

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

Problem 2. Creating a data structure for the Bashsoft

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.

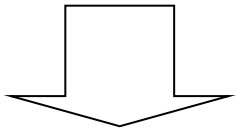
First thing you have to do is to **open** your **project** from the previous assignment and **set up** a **class** in which you will store your data. You have to create **a new class**, following the steps from the previous piece of the story. This class will be called "**Data**" and has to be **static**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

```
public static class Data
{

}
```

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:

```
public static bool isDataInitialized = false;
private static Dictionary<string, Dictionary<string, List<int>>> studentsByCourse;
```



```
Dictionary<course_name, Dictionary<user_name, scoresOnTasks>>
```

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 make some of the data **searching and filtration throughout public methods** that give to the other world the basic operations needed over the **SoftUni** system's data.

Problem 3. Initializing and saving our 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 `DataAlreadyInitialisedException` that we need to **add first in** the **ExceptionMessages class**. It's **message should be**: `Data is already initialized!` The implementation of the method for the initialization should look like this:

```
public static void InitializeData()
{
    if (!isDataInitialized)
    {
        OutputWriter.WriteMessageOnNewLine("Reading data...");
        studentsByCourse = new Dictionary<string, Dictionary<string, List<int>>>();
        ReadData();
    }
    else
    {
        OutputWriter.WriteMessageOnNewLine(ExceptionMessages.DataAlreadyInitializedException);
    }
}
```

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 **from** the **input** and **save** it **in** our **data structure.**

```
private static void ReadData()
{
    string input = Console.ReadLine();

    while (!string.IsNullOrEmpty(input))
    {
        string[] tokens = input.Split(' ');
        string course = tokens[0];
        string student = tokens[1];
        int mark = int.Parse(tokens[2]);

        // TODO: Add the course and student if they don't exist
        // TODO: Add the mark
    }
}
```

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.

```
if (!studentsByCourse.ContainsKey(course))
{
    studentsByCourse.Add(course, new Dictionary<string, List<int>>());
}

if (!studentsByCourse[course].ContainsKey(student))
{
    studentsByCourse[course].Add(student, new List<int>());
}

studentsByCourse[course][student].Add(mark);
input = Console.ReadLine();
```

Finally **after** the **while loop** we need to **set** the **`isDataInitialized`** to **`true`** and **`print "Data read!" on a new line`**!

```
isDataInitialized = true;
OutputWriter.WriteMessageOnNewLine("Data read!");
```

Problem 4. Making security checks available before retrieving data from the data structure

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, so 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 our selves** 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 `IsQueryForStudentPossible`. `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) (the user name of the student)`**. Their definition should look like the following:

```
private static bool IsQueryForCoursePossible(string courseName)...

private static bool IsQueryForStudentPossible(string courseName, string studentUserName)...
```

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 it's 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 it's message is : "The data structure must be initialised first in order to make any operations with it." :

```
private static bool IsQueryForCoursePossible(string courseName)
{
    if (isDataInitialized)
    {
        return true;
    }
    else
    {
        OutputWriter.DisplayException(ExceptionMessages.DataNotInitializedExceptionMessage);
    }

    return false;
}
```

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!"`**

```
if (studentsByCourse.ContainsKey(courseName))
{
    return true;
}
else
{
    OutputWriter.DisplayException(ExceptionMessages.InexistingCourseInDataBase);
}
```

Now that we've implemented the first method for the checks, it's time for it's 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`** :

```
private static bool IsQueryForStudentPossible(string courseName, string studentUserName)
{
    if (IsQueryForCoursePossible(courseName) && studentsByCourse[courseName].ContainsKey(studentUserName))
    {
        return true;
    }
    else
    {
        OutputWriter.DisplayException(ExceptionMessages.InexistingStudentInDataBase);
    }

    return false;
}
```

Now that we are ready with the security checks we are ready to proceed with the next step.

Problem 5. Displaying 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 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 **KeyValuePair** of **string** (user name) and 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:

```
public static void PrintStudent(KeyValuePair<string, List<int>> student)
{
    OutputWriter.WriteMessageOnNewLine(string.Format($"{student.Key} - {string.Join(", ", student.Value)}"));
}
```

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.

Problem 6. 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** the **him on** the output and so we give a new student to the **Output writer** in order to be printed:

```
public static void GetStudentScoresFromCourse(string courseName, string username)
{
    if (IsQueryForCoursePossible(courseName, username))
    {
        OutputWriter.PrintStudent(new KeyValuePair<string, List<int>>(username, studentsByCourse[courseName][username]));
    }
}
```

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**

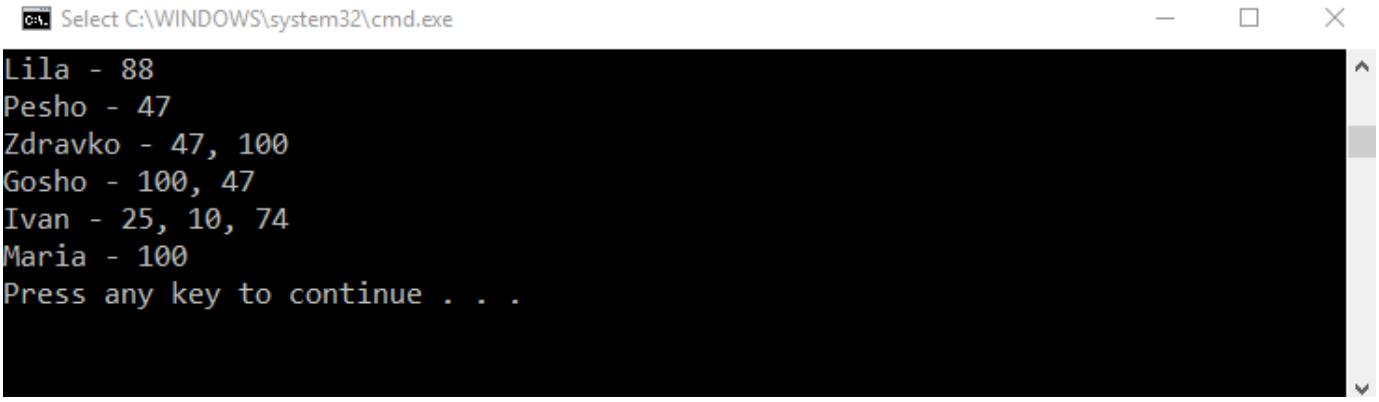
```
public static void GetAllStudentsFromCourse(string courseName)
{
    if (IsQueryForCoursePossible(courseName))
    {
        OutputWriter.WriteMessageOnNewLine($"{courseName}:");
        foreach (var studentMarksEntry in studentsByCourse[courseName])
        {
            OutputWriter.PrintStudent(studentMarksEntry);
        }
    }
}
```

Problem 7. Test your code

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

```
static void Main()
{
    StudentsRepository.InitializeData();
    StudentsRepository.GetAllStudentsFromCourse("Unity");
}
```

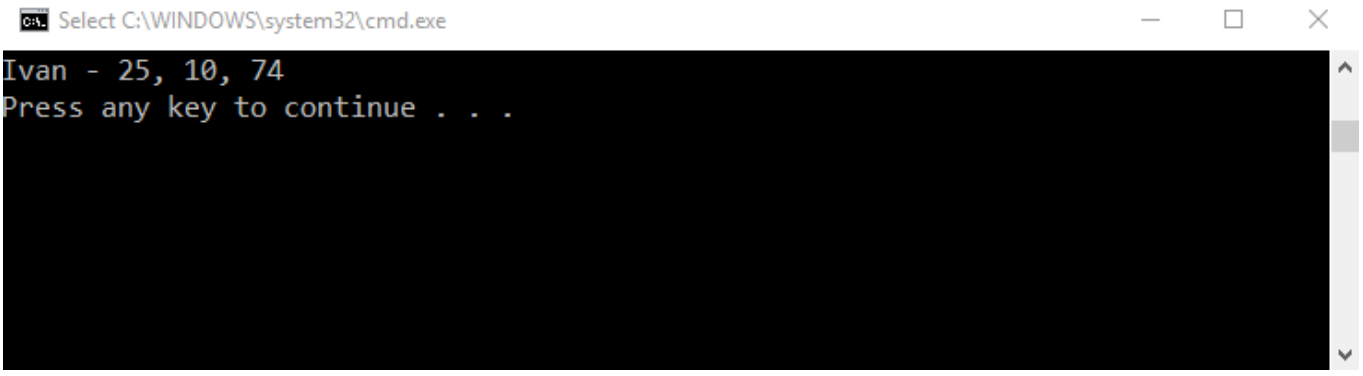
And the result should look like this:



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

```
static void Main()
{
    StudentsRepository.InitializeData();
    StudentsRepository.GetStudentScoresFromCourse("Ivan", "Unity");
}
```

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.