## Restructure the Course & Student app

Create a new “**Education**” **namespace**, based on your app’s namespace. ConsoleApp2.Education for example. Move your **Course**, **Student** and **Academy** objects to that namespace. Use `**using**` so that your objects and exceptions are visible from the root namespace. Also create a Helper sub-namespace of **Education** which will hold an empty class **AcademyHelper** (you’ll use it later on).

## Complex

Create a **Complex** class which represents a complex number, it should have 2 public **properties, Real** and **Imaginary**. The numbers should be able to be added with **+** or subtracted with -, using the rules for addition (**(m + ni) + (p + qi) = (m + p) + (n+q)i**) and subtraction (**(m + ni) - (p + qi) = (m - p) + (n - q)i**) . When converted to a string, the number should return “{real} + {imaginary}i”.

Input:  
On the first 2 lines, 2 complex numbers are entered in the format **{real}** **{imaginary}.**On the next line, **add** or **sub** is entered, which picks the action that should be executed.

Output:  
Print the result of the **addition** or **subtraction** in the format {real} +/- {imaginary}i

Example

|  |  |
| --- | --- |
| **Input** | **Output** |
| 3 4  2 6  add | 5 + 10i |

## Custom exceptions

Create 4 exceptions: **PersonAgeException**, **StudentIsBusy**, **StudentNotFound, CourseNotFound.**Replace your original exceptions with these 4, and filter the exceptions with different catch blocks.

## Grades

Improve your academy app, by adding support for grades, and a set of completed tasks for every student. A task should have a **float** for a grade, name of the task, date when it was created  
( hint: use DateTime ). Ensure that every task is unique by it’s name.

The input starts with **N** number of courses on the first line.  
On every **N**-th line a course is added in the format **courseName**//**capacity**.  
After that on the next line the user will enter **M** as the number of students.  
A new student is created and enrolled in a course, by his name and the course id,  
 by using the format **studentId**//**courseId .**After that on every next line, a completed **task** is entered in this format  
 **studentId** **courseId** **taskName**//**score**until `quit` is entered.

The program should output the **students** which have an average of **no less** than 95% score on their courses tasks, with the students sorted by name ascending and then by their score   
(use the LINQ ThenBy() ).   
And finally the program should output the top 3 or less courses, in which the students have 95% or higher scores, sorted by the course’s name and total task count ( hint: use a helper method from your **AcademyHelper** together with your **Academy** object, to sum the student tasks, for a given course) .

|  |  |
| --- | --- |
| **Input** | **Output** |
| 2  C#//30  F#//20  5  Ivan//0  Todor//1  Georgi//0  Nikolay//0  Pesho//0  0 0 task1//100  0 0 task2//100  0 1 task2//99.9  0 1 task3//99.8  quit | Ivan 100  Todor 99.85  C#  F# |

## Polymorphic vehicles

Write a program that has 2 vehicles **Car** and **Truck** and will be able **drive** and **refuel** them. **Car** and **truck** both have **fuel quantity**, **fuel consumption** **in liters** **per km** and can be **driven given distance** and **refueled with given liters.** But in the summer both vehicles use air conditioner and their **fuel consumption** per km is **increased** by **0.9** liters for the **car** and with **1.6** liters for the **truck**. Also the **truck** has a tiny hole in his tank and when it gets **refueled** it gets only **95%** of given **fuel**. The **car** has no problems when refueling and adds **all given fuel to its tank.** If vehicle cannot travel given distance its fuel does not change.

**Input**:

* On the first line - information about the car in format {Car {fuel quantity} {liters per km}}
* On the second line – info about the truck in format {Truck {fuel quantity} {liters per km}}
* On third line - number of commands N that will be given on the next N lines
* On the next N lines – commands in format
* Drive Car {distance}
* Drive Truck {distance}
* Refuel Car {liters}
* Refuel Truck {liters}

**Output**:

After each Drive command print whether the Car/Truck was able to travel given distance in format if it’s successful:

Car/Truck travelled {distance} km

Or if it is not:

Car/Truck needs refueling

Finally print the remaining fuel for both car and truck **rounded 2 digits after floating point** in format:

**Car: {liters}   
Truck: {liters}**

Example

|  |  |
| --- | --- |
| **Input** | **Output** |
| Car 15 0.3  Truck 100 0.9  4  Drive Car 9  Drive Car 30  Refuel Car 50  Drive Truck 10 | Car travelled 9 km  Car needs refueling  Truck travelled 10 km  Car: 54.20  Truck: 75.00 |