# Exercises: C# OOP Introduction

This document defines the **exercise assignments** for the [“CSharp DB Advanced” course @ Software University](https://softuni.bg/trainings/1741/databases-advanced-entity-framework-october-2017).

1. Defining Classes – Methods and Constructors

## Oldest Family Member

Create class **Person** with properties **Name** and **Age**. Create a class **Family**. The class should have **list of people**, method for **adding** members (**void AddMember(Person member)**) and a method **returning** the **oldest** family **member** (**Person GetOldestMember())**. Write a program that reads name and age for **N** people and **add them to the family**. Then **print** the **name** and **age** of the oldest member.

Your methods should have the **same exact** names as shown above!

### Note

Add the following code to your main method before your code and submit it to Judge.

|  |
| --- |
| MethodInfo oldestMemberMethod = typeof(Family).GetMethod("GetOldestMember");  MethodInfo addMemberMethod = typeof(Family).GetMethod("AddMember");  if(oldestMemberMethod == null || addMemberMethod == null)  {  throw new Exception();  } |

If you defined the class correctly, the test should pass.

### Examples

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Input** | **Output** |  | **Input** | **Output** |
| 3  Pesho 3  Gosho 4  Annie 5 | Annie 5 |  | 5  Steve 10  Christopher 15  Annie 4  Ivan 35  Maria 34 | Ivan 35 |

## Date Modifier

Create a class **DateModifier**, which stores the difference of the days between two Dates. It should have a method, which takes **two string parameters** **representing a date** as strings and **calculates the** difference in the days between them.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 1992 05 31  2016 06 17 | 8783 |
| 2016 05 31  2016 04 19 | 42 |

## Raw Data

You are the owner of a courier company and want to make a system for tracking your cars and their cargo. Define a class **Car** that holds information about **model, engine, cargo** and a **collection of exactly 4 tires**. The engine, cargo and tire **should be separate classes**. Create a constructor that receives all information about the **car**, creates, and initializes its inner components (engine, cargo and tires).

On the first line of input, you will receive a number **N** - the amount of cars you have. On each of the next **N** lines, you will receive information about a car in the format “**<Model> <EngineSpeed> <EnginePower> <CargoWeight> <CargoType> <Tire1Pressure> <Tire1Age> <Tire2Pressure> <Tire2Age> <Tire3Pressure> <Tire3Age> <Tire4Pressure> <Tire4Age>”** where the speed, power, weight and tire age are **integers**, tire pressure is a **double.**

After the **N** lines you will receive a single line with one of 2 commands “**fragile**” or “**flammable**” , if the command is “**fragile**” print all cars whose **Cargo Type is “fragile”** with a tire whose **pressure is** **< 1**, if the command is “**flammable**” print all cars whose **Cargo Type is “flamable”** and have **Engine Power > 250**. The cars should be printed in order of appearing in the input.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 2  ChevroletAstro 200 180 1000 fragile 1.3 1 1.5 2 1.4 2 1.7 4  Citroen2CV 190 165 1200 fragile 0.9 3 0.85 2 0.95 2 1.1 1  fragile | Citroen2CV |
| 4  ChevroletExpress 215 255 1200 flammable 2.5 1 2.4 2 2.7 1 2.8 1  ChevroletAstro 210 230 1000 flammable 2 1 1.9 2 1.7 3 2.1 1  DaciaDokker 230 275 1400 flammable 2.2 1 2.3 1 2.4 1 2 1  Citroen2CV 190 165 1200 fragile 0.8 3 0.85 2 0.7 5 0.95 2  flammable | ChevroletExpress  DaciaDokker |

1. Encapsulation and Validation

## Animal Farm

You should be familiar with encapsulation already. For this problem, you’ll be working with the **Animal Farm project**. It contains a class **Chicken**. Chicken contains several fields, a constructor, several properties and several methods. Your task is to encapsulate or hide anything that is not intended to be viewed or modified from outside of the class.

### Step 1. Encapsulate Fields

Fields should be **private**. Leaving fields open for modification from outside the class is potentially dangerous. Make all fields in the Chicken class private.

In case the value inside a field is needed elsewhere, use **getters** to reveal it.

### Step 2. Ensure Classes Have a Correct State

Having **getters and setters** is useless if you don’t actually use them. The Chicken constructor modifies the fields directly which is wrong when there are suitable setters available. Modify the constructor to fix this issue.

### Step 3. Validate Data Properly

Validate the chicken’s **name** (it cannot be null, empty or whitespace). In case of **invalid name**, print exception message – "Name cannot be empty.".

Validate the **age** properly, minimum and maximum age are provided, make use of them. In case of **invalid age**, print exception message "Age should be between 0 and 15."

Don’t forget to handle properly the possibly thrown exceptions.

### Step 4. Hide Internal Logic

If a method is intended to be used only by descendant classes or internally to perform some action, there is no point in keeping them **public**. The **CalculateProductPerDay()** method is used by the **ProductPerDay()** public getter. This means the method can be safely hidden inside the Chicken class by declaring it **private**.

### Step 5. Submit Code to Judge

Submit your code as a zip file in Judge. Zip everything except the bin and obj folders within the project and submit the single zip file in judge.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Mara  10 | Chicken Mara (age 10) can produce 1 eggs per day. |
| Mara  17 | Age should be between 0 and 15. |

## Football Team Generator

A football team has variable **number of players**, a **name** and a **rating**. A player has a **name** and **stats,** which are the basis for his skill level. The stats a player has are **endurance**, **sprint**, **dribble**, **passing** and **shooting**. Each stat can be an **integer** in the range [0..100]. The overall skill level of a player is calculated as the average of his stats. Only the name of a player and his stats should be visible to the entire outside world. Everything else should be hidden.

A team should expose a name, a rating (calculated by the average skill level of all players in the team and rounded to the integer part only) and methods for adding and removing players.

Your task is to model the team and the players following the proper principles of Encapsulation. Expose only the properties that needs to be visible and validate data appropriately.

Your application will receive commands until the "END" command is given. The command can be one of the following:

* **"Team;<TeamName>"** – add a new team;
* **"Add;<TeamName>;<PlayerName>;<Endurance>;<Sprint>;<Dribble>;<Passing>;<Shooting>"** – add a new player to the team;
* **"Remove;<TeamName>;<PlayerName>"** –remove the player from the team;
* **"Rating;<TeamName>"** – print the team rating, rounded to an integer.

### Data Validation

* A name cannot be null, empty or white space. If not, print "A name should not be empty.".
* Stats should be in the range 0..100. If not, print "[Stat name] should be between 0 and 100.".
* If you receive a command to remove a missing player, print "Player [Player name] is not in [Team name] team.".
* If you receive a command for a missing team, print "Team [team name] does not exist."

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Team;Arsenal  Add;Arsenal;Kieran\_Gibbs;75;85;84;92;67  Add;Arsenal;Aaron\_Ramsey;95;82;82;89;68  Remove;Arsenal;Aaron\_Ramsey  Rating;Arsenal  END | Arsenal – 81 |
| Team;Arsenal  Add;Arsenal;Kieran\_Gibbs;75;85;84;92;67  Add;Arsenal;Aaron\_Ramsey;195;82;82;89;68  Remove;Arsenal;Aaron\_Ramsey  Rating;Arsenal  END | Endurance should be between 0 and 100.  Player Aaron\_Ramsey is not in Arsenal team.  Arsenal - 81 |
| Team;Arsenal  Rating;Arsenal  END | Arsenal – 0 |

## Online Radio Database

Create an online radio station database. It should keep information about all added songs. On the first line you are going to get the number of songs you are going to try to add. On the next lines you will get the songs to be added in the format **<artist name>;<song name>;<minutes:seconds>**. To be valid, every song should have an artist name, a song name and length.

Design a custom exception hierarchy for invalid songs:

* InvalidSongException
  + InvalidArtistNameException
  + InvalidSongNameException
  + InvalidSongLengthException
    - InvalidSongMinutesException
    - InvalidSongSecondsException

### Validation

* Artist name should be between 3 and 20 symbols.
* Song name should be between 3 and 30 symbols.
* Song length should be valid numbers in format "m:s".
* Song minutes should be between 0 and 14.
* Song seconds should be between 0 and 59.

### Exception Messages

|  |  |
| --- | --- |
| **Exception** | **Message** |
| InvalidSongException | "Invalid song." |
| InvalidArtistNameException | "Artist name should be between 3 and 20 symbols." |
| InvalidSongNameException | "Song name should be between 3 and 30 symbols." |
| InvalidSongLengthException | "Invalid song length." |
| InvalidSongMinutesException | "Song minutes should be between 0 and 14." |
| InvalidSongSecondsException | "Song seconds should be between 0 and 59." |

**Note**: Check validity in the order artist name -> song name -> song length

### Output

If the song is added, print "**Song added.**". If you **can’t add a song**, print an **appropriate exception message**. On the last two lines print the **number of songs added** and the **total length of the playlist** in format **{Playlist length: 0h 7m 47s}.**

### Examples

|  |  |
| --- | --- |
| **Exception** | **Message** |
| 3  ABBA;Mamma Mia;3:35  Nasko Mentata;Shopskata salata;4:123  Nasko Mentata;Shopskata salata;4:12 | Song added.  Song seconds should be between 0 and 59.  Song added.  Songs added: 2  Playlist length: 0h 7m 47s |
| 5  Nasko Mentata;Shopskata salata;14:59  Nasko Mentata;Shopskata salata;14:59  Nasko Mentata;Shopskata salata;14:59  Nasko Mentata;Shopskata salata;14:59  Nasko Mentata;Shopskata salata;0:5 | Song added.  Song added.  Song added.  Song added.  Song added.  Songs added: 5  Playlist length: 1h 0m 1s |

1. Inheritance

## Mordor’s Cruelty Plan

Gandalf the Gray is a great wizard but he also loves to eat and lack of food makes him loose his capability of fighting the dark. The Mordor’s orcs have asked you to design them a program which keeps tack of Gandalf’s mood, so they could predict the outcome of battles with him. When Gandalf is hungry he gets angry and he cannot fight well. The orcs have a spy, who has told them which foods Gandalf eats and how they affect his mood. Here is the list:

* **Cram**: 2 points of happiness;
* **Lembas**: 3 points of happiness;
* **Apple**: 1 point of happiness;
* **Melon**: 1 point of happiness;
* **HoneyCake**: 5 points of happiness;
* **Mushrooms**: -10 points of happiness;
* **Everything else**: -1 point of happiness;

Gandalf’s moods are:

* **Angry** - below -5 points of happiness;
* **Sad** - from -5 to 0 points of happiness;
* **Happy** - from 1 to 15 points of happiness;
* **JavaScript** - when happiness points are more than 15;

The task is simple. Model an application which calculates Gandalf’s mood, based on all of the food it receives as input. After you are done, print on the first line the total happiness points Gandalf had collected and on the second line – his mood.

### Input

The input comes from the console. It will hold a single line: all the types of food Gandalf has eaten separated by a whitespace.

### Output

Print on the console Gandalf`s happiness points and the **mood**,corresponding to the points.

### Constraints

* The characters in the input string will be no more than: **1000.**
* The food count would be in the range **[1…100]**.
* Time limit: 0.3 sec. Memory limit: 16 MB.

### Note

Try to implement a factory pattern. You should have two factory classes – **FoodFactory** and **MoodFactory**. And their task is to produce objects (e.g. **FoodFactory**, produces – **Food** and the **MoodFactory** - **Mood**). Try to implement abstract classes (e.g. classes which can’t be instantiated directly)

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Cram melon honeyCake Cake | 7  Happy |
| gosho, pesho, meze, Melon, HoneyCake@; | -5  Sad |

## Animals

Create a hierarchy of **Animals**. Your task is simple: there should be a base class which all others derive from. Your program should have 3 different animals – **Dog**, **Frog** and **Cat**. Let’s go deeper in the hierarchy and create two additional classes – **Kitten** and **Tomcat**. **Kittens are female and Tomcats are male!** We are ready now, but the task is not complete. Along with the animals, there should be also a class which classifies its derived classes as sound producible. You may guess that all animals are sound producible. The only one mandatory functionality of all sound producible objects is to **ProduceSound()**. For instance, the dog should bark.

Your task is to model the hierarchy and test its functionality. Create an animal of all kinds and make them produce sound.

On the console, you will be given some lines of code. Each two lines of code, represents animals and their names, age and gender. On the first line there will be the kind of animal, you should instantiate. And on the next line, you will be given the name, the age and the gender. Stop the process of gathering input, when the command **“Beast!”** is given.

### Output

* On the console, print for each animal you’ve instantiated, its info on three lines. On the first line, print:  
   {**Kind of animal**}
* On the second line print it’s {**name**} {**age**} {**gender**}
* On the third line print the sounds it produces: {**ProduceSound()**}

### Constraints

* Each **Animal** should have **name**, **age** and **gender**
* **All** **properties**’ values should **not be blank** (e.g. name, age and so on…)
* If you enter invalid input for one of the properties’ values, throw exception with message: **“Invalid input!”**
* Each animal should have a functionality to **ProduceSound()**
* Here is example of what each kind of animal should produce when, **ProduceSound()** is called
  + **Dog: “BauBau”**
  + **Cat: “MiauMiau”**
  + **Frog: “Frogggg”**
  + **Kittens: “Miau”**
  + **Tomcat: “Give me one million b\*\*\*h"**

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Cat  Macka 12 Female  Dog  Sharo 132 Male  Beast! | Cat  Macka 12 Female  MiauMiau  Dog  Sharo 132 Male  BauBau |
| Frog  Sashky 12 Male  Beast! | Frog  Sashky 12 Male  Frogggg |
| Frog  Invalidcho -2 Male  Frog  Validcho 2 Male  Beast! | Invalid input!  Frog  Validcho 2 Male  Frogggg |