**Exercise: Encapsulation**

Problems for exercise and homework for the ["C# OOP" course @ SoftUni"](https://softuni.bg/trainings/3008/csharp-oop-october-2020).

You can check your solutions here: <https://judge.softuni.bg/Contests/1498/Encapsulation-Exercise>

* **Class Box Data**

You are given a geometric figure box with parameters **length**, **width** and **height**. Model a class **Box** that can be instantiated by the same **three parameters**. Expose to the outside world **only methods for its surface area**, **lateral surface area** and its **volume** (formulas: <http://www.mathwords.com/r/rectangular_parallelepiped.htm>).

A box’s **side** should **not be zero or a negative number**. Аdd **data validation** for each **parameter** given to the **constructor**. Make a **private setter** that performs data **validation** **internally**.

**Input**

* On the **first three lines** you will get the **length**, **width** and **height**.

**Output**

* On the **next three lines** print the **surface area**, **lateral surface area** and the **volume** of the box:

**Examples**

|  |  |
| --- | --- |
| **Input** | **Output** |
| 2  3  4 | Surface Area - 52.00  Lateral Surface Area - 40.00  Volume - 24.00 |
| 1.3  1  6 | Surface Area - 30.20  Lateral Surface Area - 27.60  Volume - 7.80 |
| 2  -3  4 | Width cannot be zero or negative. |

* **Animal Farm**

For this problem you have to **download** the provided **skeleton**.

You should be familiar with **encapsulation** already. For this problem, you’ll be working with the **AnimalFarm project**. It contains a class **Chicken**. **Chicken** contains several **fields**, a **constructor**, several **properties** and **methods**. Your task is to **encapsulate** or **hide** anything that is **unintended for viewing** or **modification** from **outside** 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 the 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 an **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 safely be **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** |
| Maria  10 | Chicken Maria (age 10) can produce 1 eggs per day. |
| Maria  17 | Age should be between 0 and 15. |

* **Shopping Spree**

Create two classes: **class** **Person** and **class** **Product**. Each person should have a **name**, **money** and a **bag** **of products**. Each product should have a **name** and a **cost**. Name cannot be an **empty string**. Money cannot be a **negative number**.

Create a program in which **each command** corresponds to a **person buying a product**. If the person can **afford** a product, **add** it to his bag. If a person **doesn’t have enough** money, print an **appropriate** **message** ("**{personName} can't afford {productName}**").

On the **first two lines** you are given **all people** and **all products**. After all purchases print **every person** in the order of **appearance** and **all products** that he has **bought** also in order of **appearance**. If **nothing was bought**, print the name of the person followed by "**Nothing bought**".

In case of **invalid input** (negative money Exception message: "**Money cannot be negative**") or an empty name (empty name Exception message: "**Name cannot be empty**") **break** the program with an appropriate message. See the examples below:

**Examples**

|  |  |
| --- | --- |
| **Input** | **Output** |
| Peter=11;George=4  Bread=10;Milk=2;  Peter Bread  George Milk  George Milk  Peter Milk  END | Peter bought Bread  George bought Milk  George bought Milk  Peter can't afford Milk  Peter - Bread  George - Milk, Milk |
| Maria=0  Coffee=2  Maria Coffee  END | Maria can't afford Coffee  Maria - Nothing bought |
| John=-3  Peppers=1;Tomatoes=2;Cheese=3  John Peppers  John Tomatoes  John Cheese  END | Money cannot be negative |