

Exercises: Encapsulation

This document defines the exercises for ["Java OOP Basics" course @ Software University](#). Please submit your solutions (source code) of all below described problems in [Judge](#).

Problem 1. Class Box

You are given a geometric figure box with parameters length, width and height. Model a class Box that 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).

On the first three lines you will get the length, width and height. On the next three lines print the surface area, lateral surface area and the volume of the box:

Note

Add the following code **at the start** of your main method and submit it to Judge.

```
public static void main(String[] args) {  
  
    Class boxClass = Box.class;  
    Field[] fields = boxClass.getDeclaredFields();  
    System.out.println(Arrays.asList(fields)  
        .stream()  
        .filter(f -> Modifier.isPrivate(f.getModifiers()))).count());  
}
```

The output on the console should be **3**. If you defined the class correctly, the test should pass.

Examples

Input	Output
2 3 4	3 Surface Area - 52.00 Lateral Surface Area - 40.00 Volume - 24.00
1.3 1 6	3 Surface Area - 30.20 Lateral Surface Area - 27.60 Volume - 7.80

Problem 2. Class Box Data Validation

A box's side should not be zero or a negative number. Expand your class from the previous problem by adding data validation for each parameter given to the constructor. Make a private setter that performs data validation internally.

Examples

Input	Output
2 -3 4	Width cannot be zero or negative.

Problem 3. 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 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."

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 `Animal` class by declaring it **private**.

Step 4. Submit Code to Judge

Submit your code as a **zip file** in Judge. Zip only the **src folder**. Make sure you have a **public Main class** with a **public static void main** method in it.

Examples

Input	Output
-------	--------



© Software University Foundation (softuni.org). This work is licensed under the [CC-BY-NC-SA](https://creativecommons.org/licenses/by-nc-sa/4.0/) license.

Follow us:



Page 2 of 8

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

Problem 4. 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 name and 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 ("[Person name] can't afford [Product name]").

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 is 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 empty name: (empty name exception message "Name cannot be empty") break the program with an appropriate message. See the examples below:

Examples

Input	Output
Pesho=11;Gosho=4 Bread=10;Milk=2; Pesho Bread Gosho Milk Gosho Milk Pesho Milk END	Pesho bought Bread Gosho bought Milk Gosho bought Milk Pesho can't afford Milk Pesho - Bread Gosho - Milk, Milk
Mimi=0 Kafence=2 Mimi Kafence	Mimi can't afford Kafence Mimi - Nothing bought
Jeko=-3 Chushki=1; Jeko Chushki END	Money cannot be negative

Problem 5. *Pizza Calories

A Pizza is made of a dough and different toppings. You should model a class Pizza which should have a name, dough and toppings as fields. Every type of ingredient should have its own class. Every ingredient has different properties: the dough can be white or wholegrain and in addition it can be crispy, chewy or homemade. The toppings can be of

type meat, veggies, cheese or sauce. Every ingredient should have a weight in grams and a method for calculating its calories according its type. Calories per gram are calculated through modifiers. Every ingredient has 2 calories per gram as a base and a modifier that gives the exact calories. For example, a white dough has a modifier of 1.5, a chewy dough has a modifier of 1.1, which means that a white chewy dough weighting 100 grams will have $100 * 1.5 * 1.1 = 330.00$ total calories.

Your job is to model the classes in such a way that they are **properly encapsulated** and to provide a public method for every pizza that **calculates its calories according to the ingredients it has**.

Step 1. Create a Dough Class

The base ingredient of a Pizza is the dough. First you need to create a class for it. It has a flour type which can be white or wholegrain. In addition, it has a baking technique which can be crispy, chewy or homemade. A dough should have a weight in grams. The calories per gram of a dough are calculated depending on the flour type and the baking technique. Every dough has 2 calories per gram as a base and a modifier that gives the exact calories. For example, a white dough has a modifier of 1.5, a chewy dough has a modifier of 1.1, which means that a white chewy dough weighting 100 grams will have $(2 * 100) * 1.5 * 1.1 = 330.00$ total calories. You are given the modifiers below:

Modifiers:

- White – 1.5;
- Wholegrain – 1.0;
- Crispy – 0.9;
- Chewy – 1.1;
- Homemade – 1.0;

Everything that the class should expose is a getter for the calories per gram. Your task is to create the class with a proper constructor, fields, getters and setters. Make sure you use the proper access modifiers.

Step 2. Validate Data for the Dough Class

Change the internal logic of the Dough class by adding a data validation in the setters.

Make sure that if invalid flour type or an invalid baking technique is given a proper exception is thrown with the message "Invalid type of dough."

The allowed weight of a dough is in the range [1..200] grams. If it is outside of this range throw an exception with the message "Dough weight should be in the range [1..200]."

Exception Messages

- "Invalid type of dough."
- "Dough weight should be in the range [1..200]."

Make a test in your main method that reads Doughs and prints their calories until an "END" command is read.

Examples

Input	Output
Dough White Chewy 100 END	330.00
Dough Tip500 Chewy 100 END	Invalid type of dough.
Dough White Chewy 240 END	Dough weight should be in the range [1..200].

Step 3. Create a Topping Class

Next you need to create a Topping class. It can be of four different types – meat, veggies, cheese or a sauce. A topping has a weight in grams. The calories per gram of a topping are calculated depending on its type. The base calories per gram are 2. Every different type of topping has a modifier. For example, meat has a modifier of 1.5, so a meat topping will have 1.5 calories per gram ($1 * 1.5$). Everything that the class should expose is a getter for calories per gram. You are given the modifiers below:

Modifiers:

- Meat – 1.2;
- Veggies – 0.8;
- Cheese – 1.1;
- Sauce – 0.9;

Your task is to create the class with a proper constructor, fields, getters and setters. Make sure you use the proper access modifiers.

Step 4. Validate Data for the Topping Class

Change the internal logic of the Topping class by adding a data validation in the setter.

Make sure the topping is one of the provided types, otherwise throw a proper exception with the message "Cannot place [name of invalid argument] on top of your pizza."

The allowed weight of a topping is in the range [1..50] grams. If it is outside of this range throw an exception with the message "[Topping type name] weight should be in the range [1..50]."

Exception Messages

- "Cannot place [name of invalid argument] on top of your pizza."
- "[Topping type name] weight should be in the range [1..50]."

Make a test in your main method that reads a single dough and prints its calories.

Examples

Input	Output
Dough White Chewy 100 Topping meat 30 END	330.00 72.00
Dough White chewy 100 Topping Krenvirshi 500 END	330.00 Cannot place Krenvirshi on top of your pizza.
Dough White Chewy 100 Topping Meat 500 END	330.00 Meat weight should be in the range [1..50].

Step 5. Create a Pizza Class!

A Pizza should have a name, some toppings and a dough. Make use of the two classes you've made earlier. Also a pizza should have public getters for its name, number of toppings and the total calories. The **total calories** are **calculated by summing the calories of all the ingredients a pizza has**. Create the class using a proper constructor, expose a method for adding a topping, a public setter for the dough and a getter method for the total calories.

The input for a pizza consists of several lines. On the first line is the **pizza name** and the **number of toppings it has**. On the second line you will get input for the **dough**. On the next lines, you will receive every topping the pizza has. **The number of lines for the toppings** will correspond to the number of toppings declared on the first line.

If creation of the pizza was **successful** print on a single line the name of the pizza and the **total calories** it has.

Step 6. Validate Data for the Pizza Class

The name of the pizza should not be an empty string. Also it should not be longer than 15 symbols. If it doesn't fit this throw an exception with the message "Pizza name should be between 1 and 15 symbols."

The number of toppings should be in range [0...10]. If not, throw an exception with the message "Number of toppings should be in range [0..10]."

Your task is to print the name of the pizza and the total calories it has according to the examples below.

Examples

Input	Output
Pizza Meatless 2 Dough Wholegrain Crispy 100 Topping Veggies 50 Topping Cheese 50 END	Meatless - 370.00 Calories.
Pizza Meatfull 5 Dough White cheWY 200	Meatfull - 1028.00 Calories.

Topping Meat 50 Topping Cheese 50 Topping meat 20 Topping sauce 10 Topping Meat 30 END	
Pizza Bulgarian 20 Dough Tip500 Balgarsko 100 Topping Sirene 50 Topping Cheese 50 Topping Krenvirsh 20 Topping Meat 10 END	Number of toppings should be in range [0..10].
Pizza Bulgarian 2 Dough Tip500 Balgarsko 100 Topping Sirene 50 Topping Cheese 50 Topping Krenvirsh 20 Topping Meat 10 END	Invalid type of dough.
Pizza Bulgarian 2 Dough White Chewy 100 Topping Sirene 50 Topping Cheese 50 Topping Krenvirsh 20 Topping Meat 10 END	Cannot place Sirene on top of your pizza.

Problem 6. **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 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 all of the 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 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.

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 to add a player to a missing team, print "Team [team name] does not exist."
- If you receive a command to show stats 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