Exercises: Encapsulation

This document defines the exercises for "Java OOP" 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 fields **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).

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.

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

Box
- length: double - width: double - height: double
+ Box (double length, double width, double height) - setLength(double): void - setWidth(double): void - setHeight(double): void + calculateSurfaceArea (): double + calculateLateralSurfaceArea (): double + calculateVolume (): double

Examples

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

Problem 2. Animal Farm

You should be familiar with encapsulation already. For this problem, you'll need to create class called **Chicken**. Chicken should contain several **fields**, a **constructor**, and several **methods**. Your task is to encapsulate or hide anything that is not intended to be viewed or modified from outside the class.

Chicken
- name: String - age: int
+ Chicken(String, int) - setName(String) : void - setAge (int): void + productPerDay (): double + toString(): Override - calculateProductPerDay() : double

Chicken lives for **15 years**. Chicken have **name** for sure, at least **1 symbol** long. Chicken producing eggs:

- First 6 years it produces 2 eggs per day [0 - 5]
- Next 6 years it produces 1 egg per day [6 - 11]
- And after that it produces 0.75 eggs per day

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 method. This means the method can safely be hidden inside the **`Chicken`** class by declaring it **private**.

Step 4. Submit Code to Judge

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

Examples

Input	Output
`Mara` `10`	`Chicken Mara (age 10) can produce 1.00 eggs per day.`

Mara 17	Age should be between 0 and 15.
Gosho 6	Chicken Gosho (age 6) can produce 1.00 eggs per day.

Problem 3. 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. Be careful about **white space** in name. Money and cost cannot be a **negative** number.

Person
- name: String - money: double - products: List<Product>
+ Person (String , double) - setName (String) : void - setMoney (double) : void + buyProduct (Product) : void + getName(): String

Product
- name: String - cost: double
+ Product (String, double) - setCost (double): void - setName (String): void + getName(): String + getCost (): double

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"**.

Read commands till you find line with **"END"** command. 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 END	Mimi can't afford Kafence Mimi – Nothing bought
Jeko=-3 Chushki=1 Jeko Chushki END	Money cannot be negative

Hint

Judge does not work with **`isBlank()`** method. You can use **`trim().isEmpty()`**.

Problem 4. 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.

Pizza
- name: String - dought: Dough - toppings: List<Topping>
+ Piza (String, int numberOfToppings) - setToppings(int) : void - setName(String) : void + setDough(Dough) : void + getName(): String + addTopping (Topping) : void + getOverallCalories () : double

Every ingredient has **different fields**: 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.

Dough
- flourType: String - bakingTechnique: String - weight: double
+ Dought (String, String, double) - setWeight(double): void - setFlourType(String): void - setBakingTechnique(String): void + calculateCalories (): double

Topping
- toppingType: String - weight: double
+ Topping (String, double) - setToppingType (String): void - setWeight (double): void + calculateCalories (): double

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**.

Dough Modifiers	Toppings Modifiers
• White – 1.5; • Wholegrain – 1.0; • Crispy – 0.9; • Chewy – 1.1; • Homemade – 1.0;	• Meat – 1.2; • Veggies – 0.8; • Cheese – 1.1; • Sauce – 0.9;

For example, **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**.

For example, **meat** has a modifier of **1.2**, which means that a meat weighting **50 grams** will have (2 * 50) * 1.2 = 120.00 total calories.

Data Validation

Data Validation must be in the order of the Input Data.

- If invalid flour type or an invalid baking technique is given an exception is thrown with the message "**Invalid type of dough.**".
- If dough weight is outside of range [1..200] throw an exception with the message "**Dough weight should be in the range [1..200]**."
- If topping is not one of the provided types throw an exception with the message "**Cannot place {name of invalid argument} on top of your pizza.**"
- If topping weight is outside of range [1..50] throw an exception with the message "**{Topping type name} weight should be in the range [1..50].**".
- If name of the pizza is **empty, only whitespace** or longer than 15 symbols throw an exception with the message "**Pizza name should be between 1 and 15 symbols.**".
- If number of topping is outside of range [0..10] throw an exception with the message "**Number of toppings should be in range [0..10].**".

The input for a pizza consists of several lines:

- On the first line is the **pizza name** and the **number of toppings it has** in format:
  - Pizza {**pizzaName**} {**numberOfToppings**}
- On the second line you will get input for the **dough** in format:
  - Dough {**flourType**} {**bakingTechnique**} {**weightInGrams**}
- On the next lines, you will receive every topping the pizza has, until an**"END"** command is given:
  - Topping {**toppingType**} {**weightInGrams**}

If creation of the pizza was **successful** print on a single line the name of the pizza and the **total calories** it has, rounded to the second digit after the decimal point.

Examples

Input	Output
Pizza Meatless 2 Dough Wholegrain Crispy 100 Topping Veggies 50 Topping Cheese 50 END	Meatless - 370.00
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	Invalid type of dough.

Topping Krenvirsh 20 Topping Meat 10 END	
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 5. **Football Team Generator

A football team has variable number of players, a name and a rating.

Team
- name: String - players: List<Player>
+ Team (String) - setName(String) : void + getName(): String + addPlayer(Player) : void + removePlayer(String) : void + getRating() : double

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 skil**l 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.

Player
- name: String - endurance: int - sprint: int - dribble: int - passing: int - shooting: int
+ Player (String, int, int, int, int, int) - setName(String) : void + getName(): String - setEndurance (int) : void - setSprint (int) : void - setDribble (int) : void - setPassing (int) : void - setShooting (int) : void + overallSkillLevel() : double

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 fields that needs to be visible and validate data appropriately.

Input

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 a closest 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 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