Java OOP Exam – 05 August 2022

Judge link: https://judge.softuni.org/Contests/3424/Java-OOP-Retake-Exam-18-April-2022

1. Overview

The zoo has always been an interesting destination for young and old. You have to create a **zoo** project, which keeps track of the animal in the areas in the zoo. The **Areas** have **Animals** with different environmental requirements. Your task is to add, feed, and take care of the animals.

2. Setup

- Upload **only the zoo** package in every task **except Unit Tests.**
- **Do not modify the interfaces or their packages.**
- Use **strong cohesion** and **loose coupling.**
- **Use inheritance and the provided interfaces wherever possible**.
    - This includes **constructors**, **method parameters,** and **return types.**
- **Do not** violate your **interface implementations** by adding **more public methods** in the concrete class than the interface has defined.
- Make sure you have **no public fields** anywhere.

3. Task 1: Structure (50 points)

You are given **3** interfaces and you must implement their functionalities in the **correct classes**.

There are **3** types of entities in the application: **`Area`, `Animal`, and `Food`.** There should also be **`FoodRepository`**.

Food

`BaseFood` is a **base class** of any **type of food** and it **should not be able to be instantiated**.

Data

- **`calories`** - **`int`**
- **`price`** - **`double`**
    - The price of the food.

Constructor

A **Food** should take the following values upon initialization:

```
(int calories, double price)
```

Child Classes

There are two concrete types of **Food**:

Vegetable

Has **50 `calories`** and its **price** is **5**.

The constructor should take no values upon initialization.

Meat

Has **70 `calories`** and its **price** is **10**.

The constructor should take no values upon initialization.

Animal

`BaseAnimal` is a **base class** of any **type of animal** and it **should not be able to be instantiated**.

Data

- **`name`** - **`String`**
  - If the name **is null or whitespace,** throw a **`NullPointerException`** with a message:
    `"Animal name cannot be null or empty."`
  - All names are unique.
- **`kind`** - **`String`**
  - If the type **is null or whitespace,** throw a **`NullPointerException`** with a message:
    `"Animal kind cannot be null or empty."`
- **`kg`** - **`double`**
  - The kilograms of the **`Animal.`**
- **`price`** - **`double`**
  - The price of the **`Animal.`**
  - If the price is below or equal to **0,** throw an **`IllegalArgumentException`** with a message:
    `"Animal price cannot be below or equal to 0."`

Behavior

```
abstract void eat()
```

The **`eat()`** method increases the **`Animal's`** kilograms. Keep in mind that some breeds of **`Animal`** can implement the method differently.

Constructor

An **`Animal`** should take the following values upon initialization:

```
(String name, String kind, double kg, double price)
```

Child Classes

There are several concrete types of **`Animal`**:

AquaticAnimal

Has **initial kilograms of 2.50.**

Can only live in `WaterArea`!

The constructor should take the following values upon initialization:

```
(String name, String kind, double price)
```

Behavior

`void eat()`

- The method **increases** the animal's kilograms by **7.50**.

TerrestrialAnimal

Has **initial kilograms of 5.50**.

Can only live in `LandArea`!

The constructor should take the following values upon initialization:

```
(String name, String kind, double price)
```

Behavior

`void eat()`

- The method **increases** the animal's kilograms by **5.70**.

Area

`BaseArea` is a **base class** of any **type of `Area`** and it **should not be able to be instantiated**.

Data

- **`name`** - **`String`**
  - If the name **is null or whitespace,** throw a **`NullPointerException`** with a message: **`"Area name cannot be null or empty."`**
  - All names are unique.
- **`capacity`** - **`int`**
  - The **number** of **`Animal`** an **`Area`** **can have.**
- **`foods`** - **`Collection<Food>`**
- **`animals`** - **`Collection<Animal>`**

Behavior

Constructor

An **`Area`** should take the following values upon initialization:

```
(String name, int capacity)
```

`int sumCalories()`

Returns the sum of **each food's calories** in the **`Area`.**

`void addAnimal(Animal animal)`

Adds an **`Animal`** in the **`Area`** if there is the **capacity** for it.

If there is **not enough capacity** to **add** the **`Animal`** in the **`Area`** **throw an `IllegalStateException`** with **the following message:**

- **`"Not enough capacity."`**

`void removeAnimal(Animal animal)`

Removes an **`Animal`** from the **`Area`**.

`void addFood(Food food)`

Adds a **Food** in the **`Area`**.

`void feed()`

The **`feed()`** method **feeds all animals** in the area.

`String getInfo()`

Returns a **`String`** with **information** about the **`Area`** in the format below. If the **`Area`** **doesn't have an animal**, print **`"none"`** instead.

```
"{areaName} ({areaType}):
Animals: {animalName1} {animalName2} {animalName3} (…) / Animals: none
Foods: {foodsCount}
Calories: {sumCalories}"
```

*Note: Use **`System.lineSeparator()`** for a new line.*

Child Classes

There are 2 concrete types of **`Area`**:

WaterArea

Has **10 capacity.**

The constructor should take the following values upon initialization:

```
(String name)
```

LandArea

Has **25 capacity.**

The constructor should take the following values upon initialization:

```
(String name)
```

FoodRepository

The **`FoodRepositoryImpl`** is a **repository** for the **foods** that are in the **area**.

Data

- **`foods`** - **`Collection<Food>`**

Behavior

`void add(Food food)`

- **Adds food** to the **collection**.

`boolean remove(Food food)`

- **Removes food** from the **collection**. **Returns `true`** if the deletion was **successful**, **otherwise** - **`false`**.

`Food findByType(String type)`

- **Returns** the **first food** of the **given type**, if there is. **Otherwise**, returns **null**.

Task 2: Business Logic (150 points)

The Controller Class

The business logic of the program should be concentrated around several **commands**. You are given interfaces, which you have to implement in the correct classes.

Note: The `ControllerImpl` class SHOULD NOT handle exceptions! The tests are designed to expect exceptions, not messages!

The first interface is **`Controller`**. You must create a **`ControllerImpl`** class, which implements the interface and implements all its methods. The constructor of **`ControllerImpl`** does **not take** any **arguments**. The given methods should have the following logic:

Data

You need to keep track of some things, this is why you need some private fields in your controller class:

- **`foodRepository`** - **`FoodRepository`**
- **`areas`** - a **`Collection of Area`**

Commands

There are several **commands**, which control the **business logic** of the **application**. They are **stated below**. The **`Area` name** passed to the methods will **always** be **valid**!

AddArea Command

Parameters

- **`areaType`** - **`String`**
- **`areaName`** - **`String`**

Functionality

Adds an **`Area`** in the collection. **Valid** types are: "**`WaterArea`**" and "**`LandArea`**".

If the **`Area` type** is **invalid**, you have to **throw a `NullPointerException`** with **the following message:**

- **`"Invalid area type."`**

If the **`Area`** is **added successfully**, the method should **return** the following **`String`**:

- **`"Successfully added {areaType}."`**

BuyFood Command

Parameters

- **`type`** - **`String`**

Functionality

Creates a **food** of the **given type** and **adds** it to the **`FoodRepository`**. **Valid** types are: "**`Vegetable`**" and "**`Meat`**". If the food **type** is **invalid**, throw an **`IllegalArgumentException`** with a message:

- **`"Invalid food type."`**

The **method** should **return** the following **string** if the **operation** is **successful**:

- **`"Successfully added {foodType}."`**

FoodForArea Command

Parameters

- **areaName - String**
- **foodType - String**

Functionality

Adds the desired **Food** to the **Area** with the **given name**. You have to remove the **Food** from the **FoodRepository** if the insert is **successful**.

If there is **no such food**, you have to **throw an IllegalArgumentException** with **the following message**:

- **"There isn't a food of type {foodType}."**

If **no exceptions** are **thrown return** the **String**:

- **"Successfully added {foodType} to {areaName}."**

AddAnimal Command

Parameters

- **areaName** - **String**
- **animalType** - **String**
- **animalName** - **String**
- **kind** - **String**
- **price** - **double**

Functionality

Adds the desired **Animal** to the **Area** with the **given name**. **Valid Animal** types are "**AquaticAnimal**", and "**TerrestrialAnimal**".

If the **Animal type** is **invalid**, you have to **throw an IllegalArgumentException** with **the following message**:

- **"Invalid animal type."** - if the **Animal type** is **invalid.**

If **no errors** are **thrown**, **return** one of the following strings:

- **"Not enough capacity."** - if there is **not enough capacity** to **add** the **Animal** in the **Area.**
- **"The external living environment is not suitable."** - if the **Animal cannot live** in the **Area**
- **"Successfully added {animalType} to {areaName}."** - if the **Animal** is **added successfully** in the **Area**

FeedAnimal Command

Parameters

- **areaName** - **String**

Functionality

Feeds all **Animal** in the **Area** with the given name.

Returns a **string** with information about **how many animals** were **successfully fed**, in the following **format**:

- **"Animals fed: {fedCount}"**

CalculateKg Command

Parameters

- **areaName** - **String**

Functionality

Calculates the value of the **Area** with the given name. It is calculated by the sum of all **Animal**'s kilograms in the **Area**.

Return a **string** in the following **format**:

- **"The kilograms of Area {areaName} is {value}."**
  - The **value** should be **formatted** to the **2nd decimal place**!

GetStatistics Command

Functionality

Returns information about each area. You can use the overridden **.getInfo Area** method.

```
"{areaName} ({areaType}):
Animals: {animalName1} {animalName2} {animalName3} (…) / Animals: none
Foods: {foodCount}
Calories: {areaCalories}

{areaName} ({areaType}):
Animals: {animalName1} {animalName2} {animalName3} (…) / Animals: none
Foods: {foodCount}
Calories: {areaCalories}

(…)"
```

*Note: Use **System.lineSeparator()** for a new line.*

Exit Command

Functionality

Ends the program.

Input / Output

You are provided with one interface, which will help you with the correct execution process of your program. The interface is **Engine** and the class implementing this interface should read the input and when the program finishes, this class should print the output.

Input

Below, you can see the **format** in which **each command** will be given in the input:

- **AddArea {areaType} {areaName}**
- **BuyFood {foodType}**
- **FoodForArea {areaName} {foodType}**
- **AddAnimal {areaName} {animalType} {animalName} {animalKind} {price}**
- **FeedAnimal {areaName}**
- **CalculateKg {areaName}**
- **GetStatistics**

- `Exit`

Output

Print the output from each command when issued. If an exception is thrown during any of the commands' execution, print the exception message.

Examples

Input

`AddArea WaterArea CoralReef`
`BuyFood Vegetable`
`BuyFood Vegetable`
`BuyFood Meat`
`FoodForArea CoralReef Vegetable`
`FoodForArea CoralReef Vegetable`
`FoodForArea CoralReef Meat`
`AddAnimal CoralReef AquaticAnimal Balboa Turtle 17.38`
`AddAnimal CoralReef AquaticAnimal Shelby Carp 14.14`
`FeedAnimal CoralReef`
`CalculateKg CoralReef`
`FeedAnimal CoralReef`
`GetStatistics`
`Exit`

Output

`Successfully added WaterArea.`
`Successfully added Vegetable.`
`Successfully added Vegetable.`
`Successfully added Meat.`
`Successfully added Vegetable to CoralReef.`
`Successfully added Vegetable to CoralReef.`
`Successfully added Meat to CoralReef.`
`Successfully added AquaticAnimal to CoralReef.`
`Successfully added AquaticAnimal to CoralReef.`
`Animals fed: 2`
`The kilograms of Area CoralReef is 20.00.`
`Animals fed: 2`
`CoralReef (WaterArea):`
`Animals: Balboa Shelby`
`Foods: 3`
`Calories: 170`

Input

`AddArea WaterArea DeepOcean`
`AddAnimal DeepOcean AquaticAnimal Fudukazi Dolphin 205.90`
`AddAnimal DeepOcean AquaticAnimal Raphael Dolphin 199.97`
`AddArea LandArea Savannah`
`AddAnimal Savannah TerrestrialAnimal LadyBoxworthy Leopard 603.32`
`AddAnimal DeepOcean Reptile Oscar Lizard 57.51`
`BuyFood Vegetable`
`BuyFood Grass`
`FoodForArea DeepOcean Plankton`
`FoodForArea DeepOcean Vegetable`
`BuyFood Meat`
`FoodForArea Savannah Meat`

```
FeedAnimal Savannah
FeedAnimal DeepOcean
AddAnimal Savannah TerrestrialAnimal Rufus Elephant 400.99
AddAnimal Savannah TerrestrialAnimal Shyman Zebra -16.90
GetStatistics
Exit
```

Output

```
Successfully added WaterArea.
Successfully added AquaticAnimal to DeepOcean.
Successfully added AquaticAnimal to DeepOcean.
Successfully added LandArea.
Successfully added TerrestrialAnimal to Savannah.
Invalid animal type.
Successfully added Vegetable.
Invalid food type.
There isn't a food of type Plankton.
Successfully added Vegetable to DeepOcean.
Successfully added Meat.
Successfully added Meat to Savannah.
Animals fed: 1
Animals fed: 2
Successfully added TerrestrialAnimal to Savannah.
Animal price cannot be below or equal to 0.
DeepOcean (WaterArea):
Animals: Fudukazi Raphael
Foods: 1
Calories: 50
Savannah (LandArea):
Animals: LadyBoxworthy Rufus
Foods: 1
Calories: 70
```

Task 3: Unit Tests (100 points)

You will receive a skeleton with three classes inside – **Main**, **Animal,** and **PetStore**. **PetStore** class will have some methods, fields, and constructors. Cover the whole class with the unit test to make sure that the class is working as intended. In Judge, you upload **.zip** to **petStore (**with **PetStoreTests** inside**)** from the **skeleton**.