

Java OOP Exam

Bakery



Overview

As we all love baked delicacies, today you were chosen to build a simple bakery software system. This system must have support for **baked foods**, **tables** and **drinks** in the bakery. The project will consist of **model classes** and a **controller class**, which manages the **interaction** between the **baked foods**, **drinks** and **tables**.

Setup

- Upload **only the bakery** package in every problem **except Unit Tests**
- **Do not modify the classes, 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

Task 1: Structure (50 points)

You are given **8** interfaces, and you must implement their functionality in the **correct classes**.

It is not required to implement your structure with **Engine**, **ConsoleReader**, **ConsoleWriter** and enc. It's good practice but it's not required.

There are **3** types of entities and 3 repositories in the application: **Table**, **BakedFood**, **Drink** and a **Repository** for each of them:

BakedFood

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

Data

- **name - String**
 - If the name is **null or whitespace**, throw an **IllegalArgumentException** with message **"Name cannot be null or white space!"**
- **portion - double**
 - If the portion is **less or equal to 0**, throw an **IllegalArgumentException** with message **"Portion cannot be less or equal to zero!"**

- **price - double**
 - If the portion is less or equal to 0, throw an **IllegalArgumentException** with message "Price cannot be less or equal to zero!"

Behavior

String toString()

Returns a **String** with information about **each food**. The returned String must be in the following format:

```
"{currentBakedFoodName}: {currentPortion - formatted to the second digit}g - {currentPrice - formatted to the second digit}"
```

Constructor

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

String name, double portion, double price

Child Classes

There are several concrete types of **BakedFood**:

Bread

The **Bread** has constant value for **InitialBreadPortion** - 200

Cake

The **Cake** has constant value for **InitialBreadPortion** - 245

Drink

BaseDrink is a **base class** for any **type of Drink** and it **should not be able to be instantiated**.

Data

- **name - String**
 - If the name is null or whitespace, throw an **IllegalArgumentException** with message "Name cannot be null or white space!"
- **portion - int**
 - If the portion is less or equal to 0, throw an **IllegalArgumentException** with message "Portion cannot be less or equal to zero!"
- **price - double**
 - If the portion is less or equal to 0, throw an **IllegalArgumentException** with message "Price cannot be less or equal to zero!"
- **brand - String**
 - If the name is null or whitespace, throw an **IllegalArgumentException** with message "Brand cannot be null or white space!"

Behavior

String toString()

Returns a **String** with information about **each drink**. The returned String must be in the following format:

```
"{current drink name} {current brand name} - {current portion}ml - {current price - formatted to the second digit}lv"
```

Constructor

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

String name, int portion, double price, String brand

Child Classes

There are several concrete types of **Drink**:

Tea

The **Tea** has **constant value** for **teaPrice - 2.50**

Water

The **Water** has **constant value** for **waterPrice - 1.50**

Table

BaseTable is a base **class** for different types of tables and **should not be able to be instantiated**

Data

- **foodOrders - Collection<BakedFood>** accessible only by the **base class**
- **drinkOrders - Collection<Drink>** accessible only by the **base class**
- **tableNumber - int** the table number
- **capacity - int** the table capacity.
 - It can't be **less than zero**. In these cases, throw an **IllegalArgumentException** with message **"Capacity has to be greater than 0"**
- **numberOfPeople - int** the count of people who want a table.
 - cannot be **less or equal to 0**. In these cases, **throw an IllegalArgumentException** with message **"Cannot place zero or less people!"**
- **pricePerPerson - double** the price per person for the table
- **isReserved - boolean** returns **true** if the **table** is **reserved**, otherwise **false**.
- **price - double** calculates the price for all people

Behavior

void reserve(int numberOfPeople)

Reserves the table with the count of people given.

void orderFood(BakedFood food)

Orders the provided food (think of a way to collect all the food which is ordered).

void orderDrink(Drink drink)

Orders the provided drink (think of a way to collect all the drinks which are ordered).

double getBill()

Returns the bill for all of the ordered drinks and food.

void clear()

Removes all the ordered drinks and food and finally frees the table sets the count of people and price to 0.

String getFreeTableInfo()

Return a String with the following format:

"Table: {table number}"

"Type: {table type}"

"Capacity: {table capacity}"

"Price per Person: {price per person for the current table}"

Constructor

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

int tableNumber, **int** capacity, **double** pricePerPerson

Child Classes

There are several concrete types of **Table**:

InsideTable

The **InsideTable** has **constant value** for **pricePerPerson** - 2.50

OutsideTable

The **OutsideTable** has **constant value** for **pricePerPerson** - 3.50

Repository

The repository holds information about the entity.

Data

- **models** - A collection of **T** (entity)

Behavior

void add(**T** model)

Adds an entity in the collection.

Collection<T> getAll()

Returns all entities (unmodifiable)

Child Repositories

TableRepository

T getByNumber(**int** tableNumber)

Returns an entity with that name.

FoodRepository

T getName(**String** name)

Returns an entity with that name.

DrinkRepository

T getName(**String** name)

Child Classes

Create **FoodRepositoryImpl**, **DrinkRepositoryImpl** and **TableRepositoryImpl** repositories.

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 must implement in the correct classes.

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

The first interface is **Controller**. You must implement a **ControllerImpl** class, which implements the interface and implements all its methods. The given methods should have the following logic:

Commands

There are several commands, which control the business logic of the application. They are stated below.

AddFood Command

Parameters

- **type** – String
- **name** – String
- **price** – double

Functionality

Creates a food with the correct type. If the food is created successful, returns:

"Added {baked food name} ({baked food type}) to the menu"

If a baked food with the given name already exists in the food repository, throw an **IllegalArgumentException** with message **"{type} {name} is already in the menu"**

AddDrink Command

Parameters

- **type** – String
- **name** – String
- **portion** – int
- **brand** - String

Functionality

Creates a drink with the correct type. If the drink is created successful, returns:

"Added {drinkName} ({drinkBrand}) to the drink menu"

If a drink with the given name already exists in the drink repository, throw an **IllegalArgumentException** with message **"{type} {name} is already in the menu"**

AddTable Command

Parameters

- **type** - String
- **tableNumber** - int
- **capacity** - int

Functionality

Creates a table with the correct type and returns:

"Added table number {tableNumber} in the bakery"

If a drink with the given name already exists in the drink repository, throw an `IllegalArgumentException` with message **"Table {tableNumber} is already added to the restaurant"**

ReserveTable Command

Parameters

- `numberOfPeople` - `int`

Functionality

Finds a table which is not reserved, and its capacity is enough for the number of people provided. If there is no such table returns:

"No available table for {numberOfPeople} people"

In the other case reserves the table and returns:

"Table {tableNumber} has been reserved for {numberOfPeople} people"

OrderFood Command

Parameters

- `tableNumber` - `int`
- `foodName` - `String`

Functionality

Finds the table with that number and the food with that name in the menu. If there is no such table returns:

"Could not find table with {tableNumber}"

If there is no such food returns:

"No {bakedFoodName} in the menu"

In other case orders the food for that table and returns:

"Table {tableNumber} ordered {bakedFoodName}"

OrderDrink Command

Parameters

- `tableNumber` - `int`
- `drinkName` - `String`
- `drinkBrand` - `String`

Functionality

Finds the table with that number and finds the drink with that name and brand. If there is no such table, it returns:

"Could not find table {tableNumber}"

If there isn't such drink, it returns:

"There is no {drinkName} {drinkBrand} available"

In other case, it orders the drink for that table and returns:

```
"Table {tableNumber} ordered {drinkName} {drinkBrand}"
```

LeaveTable Command

Parameters

- `tableNumber` - `int`

Functionality

Finds the table with the same table number. Gets the bill for that table and clears it. Finally returns:

```
"Table: {tableNumber}"
```

```
"Bill: {table bill:f2}"
```

GetFreeTablesInfo Command

Functionality

Finds all not reserved tables and for each table returns the table info.

GetTotalIncome Command

Returns the total income for the restaurant for all **completed bills**.

```
"Total income: {income:f2}lv"
```

Input / Output

You are provided with one interface, which will help 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:

- `AddFood {type} {name} {price}`
- `AddDrink {type} {name} {portion} {brand}`
- `AddTable {type} {tableNumber} {capacity}`
- `ReserveTable {numberOfPeople}`
- `OrderFood {tableNumber} {foodName}`
- `OrderDrink {tableNumber} {drinkName} {drinkBrand}`
- `LeaveTable {tableNumber}`
- `GetFreeTablesInfo`
- `GetTotalIncome`
- `END`

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

```
AddFood Bread White 2.90
AddDrink Water Spring 500 Divna
AddTable InsideTable 1 10
AddTable OutsideTable 2 20
ReserveTable 5
OrderFood 1 White
OrderDrink 1 Spring Divna
GetFreeTablesInfo
LeaveTable 1
GetTotalIncome
END
```

Output

```
Added White (Bread) to the menu
Added Spring (Divna) to the drink menu
Added table number 1 in the bakery
Added table number 2 in the bakery
Table 1 has been reserved for 5 people
Table 1 ordered White
Table 1 ordered Spring Divna
Table: 2
Type: OutsideTable
Capacity: 20
Price per Person: 3.50
Table: 1
Bill: 16.90
Total income: 16.90lv
```

Input

```
AddFood Bread Healthy 2.90
AddFood Bread Focaccia 4.90
AddFood Cake Choco 5.90
AddFood Cake Cherry -9.0
AddDrink Water Spring -500 Divna
AddDrink Water Sparkling 500 Perier
AddDrink Tea GreenTea 250 Lipton
AddDrink Tea HerbalTea 200 Bio
AddTable InsideTable 1 10
```



```
AddTable InsideTable 2 12
AddTable InsideTable 3 11
AddTable OutsideTable 4 20
AddTable OutsideTable 5 -2
AddTable OutsideTable 6 10
ReserveTable 5
ReserveTable 1
ReserveTable 2
OrderFood 1 Healthy
OrderFood 1 OrangeCream
OrderFood 2 Choco
OrderFood 3 Choco
OrderFood 4 Choco
OrderDrink 1 Spring Divna
OrderDrink 2 GreenTea Lipton
OrderDrink 2 Perier HerbalTea
OrderDrink 3 Spring Monin
GetFreeTablesInfo
LeaveTable 1
LeaveTable 2
GetTotalIncome
END
```

Output

```
Added Healthy (Bread) to the menu
Added Focaccia (Bread) to the menu
Added Choco (Cake) to the menu
Price cannot be less or equal to zero!
Portion cannot be less or equal to zero
Added Sparkling (Perier) to the drink menu
Added GreenTea (Lipton) to the drink menu
Added HerbalTea (Bio) to the drink menu
Added table number 1 in the bakery
Added table number 2 in the bakery
Added table number 3 in the bakery
Added table number 4 in the bakery
Capacity has to be greater than 0
Added table number 6 in the bakery
Table 1 has been reserved for 5 people
```

Table 2 has been reserved for 1 people
Table 3 has been reserved for 2 people
Table 1 ordered Healthy
No OrangeCream in the menu
Table 2 ordered Choco
Table 3 ordered Choco
Could not find table 4
There is no Spring Divna available
Table 2 ordered GreenTea Lipton
There is no Perier HerbalTea available
There is no Spring Monin available
Table: 4
Type: OutsideTable
Capacity: 20
Price per Person: 3.50
Table: 6
Type: OutsideTable
Capacity: 10
Price per Person: 3.50
Table: 1
Bill: 15.40
Table: 2
Bill: 10.90
Total income: 26.30lv

Task 3: Unit Tests (100 points)

You will receive a skeleton with one class inside. The class will have some methods, fields and constructors. Cover the whole class with unit test to make sure that the class is working as intended.