# C# 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** Bakeryproject in every problem **except** **Unit Tests**
* **Do not modify the interfaces or their namespaces**
* 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** or **properties** in the concrete class than the interface has defined
* Make sure you have **no public fields** anywhere

## Task 1: Structure (50 points)

For this task's evaluation logic in the methods isn’t included.

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

There are **3** types of entities in the application: **BakedFood, Drink, Table**.

### BakedFood

The Food is a **base class** for any **type of food** and it **should not be able to be instantiated**.

#### Data

* Name – **string** (If the name is **null** or **whitespace** throw an **ArgumentException** with the message **"Name cannot be null or white space!")**
* Portion – **int** (can’t be **less or equal to 0**. In these cases, throw an **ArgumentException** with the message **"**Portion cannot be less or equal to zero**")**
* **Price** – **decimal** (can’t be **less or equal to 0.** In these cases, throw an **ArgumentException** with the message **"**Price cannot be less or equal to zero!"**)**

Once a baked food is initialized, it shouldn't be possible to change its properties' values.

#### Behavior

##### string ToString()

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

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

#### Constructor

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

**string name, int portion, decimal price**

#### Child Classes

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

* Bread – with the constant value for InitialBreadPortion - 200
* Cake - with the constant value for InitialCakePortion - 245

### Drink

The Drink 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 **ArgumenException** with the message **"Name cannot be null or white space!"**)
* Portion – **int** (if the portion is **less than** or **equal** to **0**, throw an **ArgumentException** with the message **"**Portion cannot be less or equal to zero**"**)
* **Price** – **decimal** (if the price is **less than** or **equal** to **0**, throw an **ArgumentException** with the message **"**Price cannot be less or equal to zero!**"**)
* **Brand - string** (If the brand is **null** or **whitespace** throw **ArgumentException** with the message **"Brand cannot be null or white space!"**)

Once a drink is initialized, it shouldn't be possible to change its properties' values.

#### 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 **drink** should take the following values upon initialization:

**string name, int portion, decimal price, string brand**

#### Child Classes

There are several concrete types of **drink**, which have **different prices**:

* Tea – with a constant value for TeaPrice - 2.50
* Water – with a constant value for WaterPrice - 1.50

### Table

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

#### Data

* FoodOrders – a collection of foods accessible only by the base class.
* DrinkOrders – a collection of drinks accessible only by the base class.
* TableNumber – **int** the table number
* Capacity – **int** the table capacity(capacity can’t be **less than zero**. In these cases, throw an ArgumentException with **the** message "Capacity has to be greater than 0")
* NumberOfPeople – int the count of people who want a table (number of people cannot be **less** or **equal** **to 0**. In these cases, throw an ArgumentException **with the message** "Cannot place zero or less people!")
* PricePerPerson – decimal the price per person for the table
* IsReserved – bool returns true if the table is reserved
* Price – calculated property, which calculates the price for all people

#### Constructor

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

**int tableNumber, int capacity, decimal pricePerPerson**

#### Child Classes

There are several concrete types of **tables**, which have **different prices**:

* InsideTable – with a constant value for InitialPricePerPerson - 2.50
* OutsideTable – with a constant value for InitialPricePerPerson - 3.50

#### Behavior

##### void Reserve(int numberOfPeople)

Reserve the table with the count of people given.

##### void OrderFood(IFood food)

Order the provided food (think of a way to collect all the food that is ordered).

##### void OrderDrink(IDrink drink)

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

##### decimal GetBill()

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

##### void Clear()

Removes all of the ordered drinks and food and finally frees the table and sets the count of people 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}"

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

The first interface is **I**Controller. You must create a Controllerclass, which implements the interface and implements all of its methods. The constructor of Controllerdoes not take any arguments. The given methods should have the logic described for each in the Commands section.

### Commands

There are several commands which control the business logic of the application. They are stated below. Also, the controller class holds all **bakedFoods**, **drinks** and **tables**:

* **bakedFoods** – List of foods – foods offered by the restaurant
* **drinks** – List of drinks – the drinks the restaurant offers
* **tables** – List of tables – all tables in the restaurant

The main functionality is represented by these **public** **methods**:

Also,

**NOTE: The** Controller **class should not handle any exceptions. That should be the responsibility of the class, which reads the commands and passes them to the** Controller**.**

### 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 – decimal

##### Functionality

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

"Added {baked food name} ({baked food type}) to 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 successfully, returns:

#### "Added {drink name} ({drink brand}) to the drink menu"

#### AddTable Command

##### Parameters

* Type - string
* TableNumber – int
* Capacity - int

##### Functionality

Creates a table with the correct type and returns:

"Added table number {table number} in the bakery"

#### ReserveTable Command

##### Parameters

* NumberOfPeople – int

##### Functionality

Finds a table that 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 {table number} has been reserved for {numberOfPeople} people"

#### OrderFood Command

##### Parameters

* TableNumber - int
* FoodName - string

##### Functionality

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

"Could not find table {tableNumber}"

If there is no such food returns:

"No {bakedFoodName} in the menu"

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

"Table {tableNumber} ordered {bakedFoodName}"

#### OrderDrink Command

##### Parameters

* TableNumber - int
* DrinkName – string
* DrinkBrand - string

##### Functionality

Find the table with that number and find 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 a 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 orders.

"Total income: {income:f2}lv"

## Input / Output

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

You are given the **Engine** class with written logic in it. In order for the code to be **compiled**, some parts are **commented**, **don't forget to comment them out**. The **try-catch block** is also **commented** for the program to **throw exceptions and for you to see them**, **comment it out** when you are **ready** with this too.

### 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 Cake Choco 12.90**  **AddDrink Water Spring -500 Divna**  **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 2 Choco**  **OrderFood 3 Choco**  **OrderFood 4 Choco**  **OrderDrink 1 Spring Divna**  **OrderDrink 2 Spring Divna**  **OrderDrink 2 Spring Herbal**  **OrderDrink 3 Spring Monin**  **GetFreeTablesInfo**  **LeaveTable 1**  **LeaveTable 2**  **GetTotalIncome**  **END** |
| **Output** |
| **Added Healthy (Bread) to the menu**  **Added Choco (Cake) to the menu**  **Portion cannot be less or equal to zero**  **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**  **Table 2 ordered Choco**  **Table 3 ordered Choco**  **Table 4 ordered Choco**  **There is no Spring Divna available**  **There is no Spring Divna available**  **There is no Spring Herbal 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: 15.40**  **Total income: 30.80lv** |

## Task 3: Unit testing (100 points)

You will receive a skeleton with two classes – **Item** and **BankVault**. The class that you need to test is **BankVault**. Cover the whole class with unit tests to make sure that the class is working as intended.