Assignment 1

COMP 250   Fall 2018

posted: Sunday, Sept. 23, 2018
due: Monday, Oct. 8, 2018 at 23:59

The Teaching Assistants handling this assignment are Tabish Syed (email), Johannes Brustle (email), Matt Grenander (email), Qing Tian (email), Anand Kamat (email).
Their office hours will be posted on mycourses Announcements.

General Instructions

- **Submission instructions**
  - Late assignments will be accepted up to 2 days late and will be penalized by 10 points per day. Note that submitting one minute late is the same as submitting 23 hours late. We will deduct 10 points for any student who has to resubmit after the due date (i.e. late) irrespective of the reason, be it wrong file submitted, wrong file format was submitted or any other reason. This policy will hold regardless of whether or not the student can provide proof that the assignment was indeed "done" on time.
  - Don't worry if you realize that you made a mistake after you submitted : you can submit multiple times but only the latest submission will be evaluated. We encourage you to submit a first version a few days before the deadline (computer crashes do happen and myCourses may be overloaded during rush hours).
  - Please store all your files in a folder called "Assignment1", zip the folder and submit it to myCourses. Inside your zipped folder, there must be the following files.
    - `MarketProduct.java`
    - `Egg.java`
    - `Jam.java`
    - `Fruit.java`
    - `SeasonalFruit.java`
    - `Basket.java`
    - `Customer.java`

    Do not submit any other files, especially .class files. Any deviation from these requirements may lead to lost marks
- You will have to create all the above classes from scratch. The assignment shall be graded automatically. Requests to evaluate the assignment manually shall not be entertained, so

please make sure that you follow the instruction closely or your code may fail to pass the automatic tests. Note that for this assignment, you are NOT allowed to import any other class (including for example `ArrayList` or `LinkedList`). **Any failure to comply with these rules will give you an automatic 0.**

- We have included with these instruction a tester class, which is a mini version of the final tester used to evaluate your assignment. If your code fails those tests, it means that there is a mistake somewhere. Even if your code passes those tests, it may still contain some errors. We will test your code on a much more challenging set of examples. We therefore highly encourage you to modify the tester class and expand it.
- You will automatically get 0 if your code does not compile.
- Failure to comply with any of these rules will be penalized. If anything is unclear, it is up to you to clarify it by asking either directly a TA during office hours, or on the discussion board on myCourses.

Market Place

For this assignment you will write several classes to simulate an online Market place. Make sure to follow the instructions below very closely. Note that in addition to the required methods, you are free to add as many other **private** methods as you want (no other additional method is allowed).

[**10 points**] Write an `abstract` class `MarketProduct` which has the following `private` field:

- A `String` name

The class must also have the following `public` methods:

- A constructor that takes a `String` as input indicating the name of product and uses it to initialize the corresponding attribute.
- A `final getName()` method to retrieve the name of *this* `MarketProduct`.
- An `abstract` method `getCost()` which takes no input and returns an `int`. This method should be abstract (thus, not implemented) because how to determine the cost depends on the type of product.
- An `abstract` method `equals()` which takes an `Object` as an input and returns a `boolean`. This method should be abstract as well, since depending on the type of product different conditions should be met in order for two products to be considered equal.

[**25 points**] All of the followings must be subclasses of the `MarketProduct`:

- Write a class `Egg` derived from the `MarketProduct` class. The `Egg` class has the following `private` fields:
  - An `int` indicating the number of eggs.
  - An `int` indicating the price per dozen of these eggs. Note that the all the prices (throughout the assignment) are indicated in cents. For instance, 450 represents the amount $4.50.

  The `Egg` class has also the following `public` methods:

  - A constructor that takes as input a `String` with the name of the product, an `int` indicating the number required, and an `int` indicating the price of such product by the dozen. The constructor uses the inputs to create a `MarketProduct` and initialize the corresponding fields.
  - A `getCost()` method that takes no input and returns the cost of the product in cents. The cost is computed base on the number required and the cost per dozen. For instance, 4 large brown eggs at 380 cents/dozen cost 126 cents (the cost should be rounded down to the nearest cent). You may assume that cost of all `MarketProducts` fits within an int and therefore doesn't cause overflow.
  - An `equals()` method which takes as input an `Object` and return `true` if input matches `this` in type, name, cost and number of eggs. Otherwise the method returns `false`.

- Write a class `Fruit` derived from the `MarketProduct` class. The `Fruit` class has the following `private` fields:
  - A `double` indicating the weight in kg.
  - An `int` indicating the price per kg in cents.

  The `Fruit` class has also the following `public` methods:
  - A constructor that takes as input a `String` with the name of the product, a `double` indicating the weight in kg, and an `int` indicating the price per kg of such product. The constructor uses the inputs to create a `MarketProduct` and initialize the corresponding fields.
  - A `getCost()` method that takes no input and returns the cost of the product in cents. The cost is computed based on the weight and the price per kilogram. For instance, 1.25 kgs of asian pears at 530 cents per kg cost 662 cents.
  - An `equals()` method just like the `Egg` class, which matches type, name, weight and cost.
- Write a class `Jam` derived from the `MarketProduct` class. The `Jam` class has the following `private` fields:
  - An `int` indicating the number of jars.
  - An `int` indicating the price per jar in cents.

  The `Jam` class has also the following `public` methods:
  - A constructor that takes as input a `String` with the name of the product, an `int` indicating the number of jars, and an `int` indicating the price per jar of such product. The constructor uses the inputs to create a `MarketProduct` and initialize the corresponding fields.
  - A `getCost()` method that takes no input and returns the cost of the product in cents. The cost is computed based on the number of jars and their price. For instance, 2 jars of Strawberry jam at 475 cents per jar cost 950 cents.
  - An `equals()` method like in the previous classes.
- Write a class `SeasonalFruit` derived from the `Fruit` class. The `SeasonalFruit` class has no fields, but it has the following `public` methods:
  - A constructor that takes as input a `String` with the name of the product, a `double` indicating the weight in kg, and an `int` indicating the price per kg of such product. The constructor uses the inputs to create a `Fruit`.
  - A `getCost()` method that takes no input and returns the cost of the product in cents. Since this type of `Fruit` is in season, its original cost should receive a 15% discount. For instance, 0.5 kgs of McIntosh apples at 480 cents per kg cost 204 cents.

[**40 points**] Write a class `Basket` which has the following `private` field:

- An array of `MarketProduct`s.

The class must also have the following `public` methods:

- A constructor that takes no inputs and initialize the field with an empty array.
- A `getProducts()` method which takes no inputs and returns *a shallow copy* of the array (NOT a copy of the reference!) of `MarketProduct`s of *this* `Basket` (with the elements in the same order).
- An `add()` method which takes as input a `MarketProduct` and does not return any value. The method adds the `MarketProduct` at the end of the array of products of *this* `Basket`.
- A `remove()` method which takes as input a `MarketProduct` and returns a `boolean`. The method removes the **first** occurrence of the specified element from the array of products of *this* `Basket`. If no such product exists, then the method returns `false`, otherwise, after removing it, the method returns `true`. Note that this method removes the the whole product. For example, it is not possible to remove 0.25 Kg of McIntosh apples for a 0.5 Kg McIntosh apples `MarketProduct`. After the product has been remove from the array, the subsequent elements should be shifted down by one position, leaving no empty slot in the array.
- A `clear()` method which takes no inputs, returns no values, and empties the array of products of *this* `Basket`.
- A `getNumOfProducts()` method that takes no inputs and returns the number of products present in *this* `Basket`.
- A `getSubTotal()` method that takes no inputs and returns the cost (in cents) of all the products in *this* `Basket`.
- A `getTotalTax()` method that takes no inputs and returns the tax amount (in cents) to be paid based on the product in *this* `Basket`. Since we are in Quebec, you can use a tax rate of 15%. The tax amount should be rounded down to the nearest cent. Note that `Egg` and `Fruit` are tax free, so taxes should be paid only for `Jam`.
- A `getTotalCost()` method that takes no inputs and returns the total cost (in cents and after tax) of the products in *this* `Basket`.
- A `toString()` method that returns a `String` representing a receipt for *this* `Basket`. The receipt should contain a product per line. On each line the name of the product should appear, followed by its price separated by a tab character. After all the products have been listed, the following information should appear on each line:
  - An empty line
  - The subtotal cost
  - The total tax
  - An empty line
  - The total cost

Note that all the integer number of cents should be transformed into a `String` formatted in dollars and cents (you can write a helper method to do so if you'd like). If the number of cents is represented by an `int` that is less than or equal to 0, then it should be transformed into a `String` containing only the hyphen character (`"-"`). An example of a receipt is as follows:

```
Quail eggs  4.00
McIntosh apples 6.12
Asian Pears 4.24
Blueberry Jam 4.75

Subtotal    23.86
Total Tax   0.71

Total Cost  24.57
```

[**25 points**] Write a class `Customer` which has the following `private` fields:

- A `String` name
- An `int` representing the balance (in cents) of the customer
- A `Basket` containing the products the customer would like to buy.

The class must also have the following `public` methods:

- A constructor that takes as input a `String` indicating the name of the customer, and an `int` representing their initial balance. The constructor uses its inputs and creates an empty `Basket` to initialize the corresponding fields.
- A `getName()` and a `getBalance()` method which return the name and balance (in cents) of the customer respectively.
- A `getBasket()` method which returns the reference to the `Basket` of *this* `Customer` (no copy of the Basket is needed).
- An `addFunds()` method which takes an `int` as input representing the amount of cents to be added to the balance of *this* `Customer`. If the input received is negative, the method should throw an `IllegalArgumentException` with an appropriate message. Otherwise, the method will simply update the balance and return the new balance in cents.
- An `addToBasket()` method which takes a `MarketProduct` as input and adds it to the basket of *this* `Customer`. This method should not return anything.
- A `removeFromBasket()` method which takes a `MarketProduct` as input and removes it from the basket of *this* `Customer`. The method returns a boolean indicating whether of not the operation was successful.
- A `checkOut()` method which takes no input and returns the receipt for the customer as a `String`. If *this* `Customer`'s balance is not enough to cover the total cost of their

basket, then the method throws an `IllegalStateException`. Otherwise, *this* `Customer` is charged the total cost of the basket and a receipt is returned.