Object-Oriented Programming: Homework #4

Time

- Assigned on 4/24/2023 (Monday); Due at 09:00 on 5/8/2023 (Monday)

Submission

- Source Code Submit on iLearning.
- Your program code for this homework should be submitted as a single .java file named HW4_{Student ID}.java (**DO NOT** submit the entire Eclipse project, only the source files).
- In addition, please delete the package before uploading the file.

Objectives

- Work with multiple classes.
- Use UML diagram as outline for creating classes, fields, and methods.
- Practice incremental programming.

Program Descriptions

- Please write a program in Java to create a backend for organizing data for a store. The program will be based on the UML diagram as shown in Fig. 1. You will create the three classes `Store`, `Product`, and `Customer`. However, to develop code, you will need to create a "driver" program(s) in Java `main` method that you can run to test your code. In the driver program, you can create sequences of statements to test your three classes.

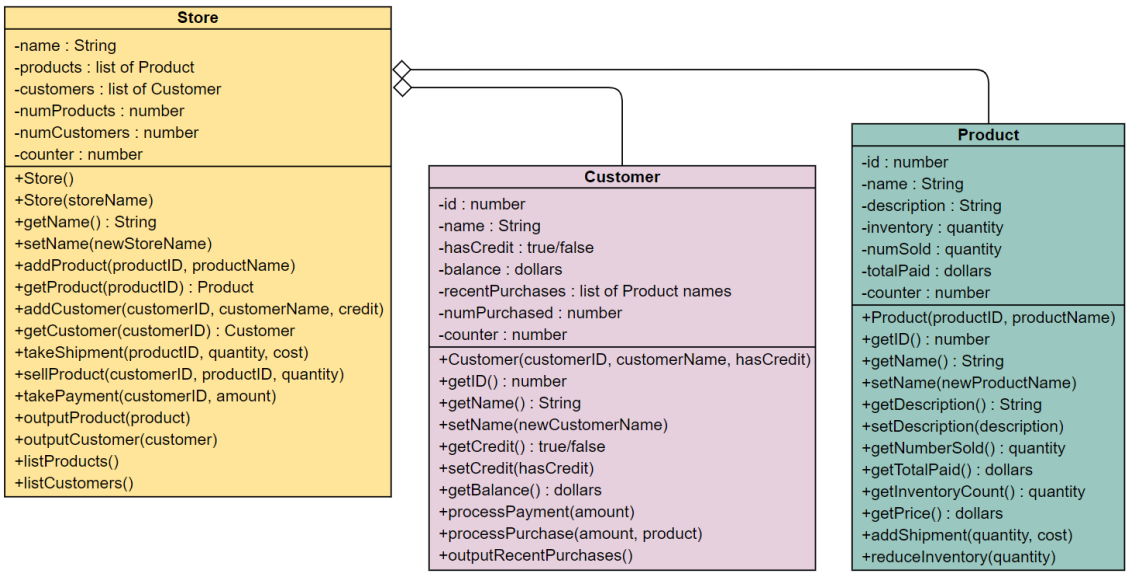

Figure 1: The UML diagram for the store backend.

Program Specifications

- Based on the UML diagram in Fig. 1, create the three classes: Product, Customer, and Store.
    - Include all attributes (data members) as indicated in the UML diagram.
    - Implement the constructors and the methods / member functions listed below.
- We have attached a starting code "HW4.java" on iLearning system. The Product class has been **mostly** done for you and includes an example of a driver program. Note that the starting code is just for your reference. Your final submission should follow the specification listed in this section. In addition, you will have to create **your own driver programs**, or extend the one in "HW4.java", to **test the other classes** as you develop them. See the end of the document for some guidance on creating driver programs.
- A lot of the design has been done for you in the UML Class diagram provided.
    - Outline your steps for creating a driver program for testing the classes.
    - Additional Components: Create test cases to ensure the class behaves correctly when created and or updated.
- Think about which member functions should not change the objects. Those member function declarations should be modified and marked as `final` for all classes. This will be important when creating the output functions.
- Product Class:
    - Attributes
        - `id`, `inventory`, and `numSold` should all be `int`.
        - `name` and `description` should be `String`.
        - `totalPaid` should be `double`.
        - `counter` should be a `static int`.
    - Methods
        - `Product(int productID, String productName)`
            - ✓ Initialize the values of data members: `id`, `inventory`, `numSold`, and `totalPaid`.
            - ✓ Initialize the product name by the `setName` method.
        - `int getID()`
            - ✓ Return the value of `id`.
        - `String getName()`
            - ✓ Return the value of `name`.
        - `void setName(String productName)`
            - ✓ The value of `productName` cannot be an empty string, i.e., "". If it is, the value of `counter` should be incremented, and the `name` should be set to the default value "Product <num>" where <num> is the value of `counter`.
        - `String getDescription()`
            - ✓ Return the value of `description`.
        - `void setDescription(String description)`
            - ✓ Set the value of `description`.
        - `int getNumberSold()`
            - ✓ Return the value of `numSold`.

        - `double getTotalPaid()`
            - ✓ Return the total of all shipment costs over time. See `addShipment()` function.
        - `int getInventoryCount()`
            - ✓ Return the value of `inventory`.
        - `boolean addShipment(int shipmentQuantity, double shipmentCost)`
            - ✓ Add `shipmentQuantity` to `inventory` and increase `totalPaid` by `shipmentCost`. Do not replace `totalPaid`, just increase its value. Return true.
            - ✓ If you get a negative `shipmentQuantity` or a negative `shipmentCost`, make no changes and return false.
        - `boolean reduceInventory(int purchaseQuantity)`
            - ✓ Decrease `inventory` by `purchaseQuantity` and increase `numSold` by `purchaseQuantity` and return true.
            - ✓ If there is not enough `inventory`, do nothing and return false.
            - ✓ If the `purchaseQuantity` is negative, do nothing and return false.
        - `double getPrice()`
            - ✓ This function will calculate the current price based on the average cost per item over time plus a 25% markup.
                (1) Price: `(totalPaid / (inventory + numSold)) * 1.25`
                (2) Return `-1` if you cannot calculate a price (e.g. division by zero)
                (3) Warning: avoid integer division!
- Customer Class:
    - Attributes
        - `id` should be `int`
        - `name` should be a `String`
        - `credit` should be a `boolean`
            - ✓ True means `balance` can become negative.
            - ✓ False means `balance` cannot become negative.
            - ✓ If they have `credit` and they make a purchase with insufficient funds in their balance, the purchase is allowed. Otherwise, they are limited to purchases that can be paid by their `balance`.
        - `balance` should be a `double`
        - `recentPurchases` should be an array that can hold up to 5 elements of `String`.
            - ✓ Note that you will need an additional private data member `numPurchased` to track the size of the array. This is not in the UML diagram, but becomes necessary due to the implementation details.

- `counter` should be a `static int`
  - ✓ This is an additional private data member for the class. It is not part of the UML diagram, but is necessary to create default names that attempt to be unique.

- Methods
  - `Customer(int customerID, String name, boolean credit)`
    - ✓ Initialize the values of data members: `id`, `credit`, `balance`, and `numPurchased`.
    - ✓ Initialize the customer's name by the `setName` method.
  - `int getID()`
    - ✓ Return the value of `id`.
  - `String getName()`
    - ✓ Return the value of `name`.
  - `void setName(String customerName)`
    - ✓ The value of `customerName` cannot be an empty string, i.e., "". If it is, the value of `counter` should be incremented, and the `name` should be set to the default value "Customer <num>" where <num> is the value of `counter`.
  - `boolean getCredit()`
    - ✓ Return the value of `credit`.
  - `void setCredit(boolean hasCredit)`
    - ✓ Set the value of `credit`.
  - `double getBalance()`
    - ✓ Return the value of `balance`.
  - `boolean processPayment(double amount)`
    - ✓ Add `amount` to `balance` and return true.
    - ✓ If `amount` is negative, do nothing and return false.
  - `boolean processPurchase(double amount, Product product)`
    - ✓ If the customer has `credit`: subtract `amount` from `balance`. Recall that `balance` can be negative if `credit` is true.
    - ✓ If the customer does not have `credit`:
      - (1) If the `balance` is greater than or equal to the `amount`, then subtract amount from `balance` and return true. Otherwise, do nothing and return false.
      - (2) Recall, `balance` is not allowed to be negative if `credit` is false.
    - ✓ If the purchase occurs, then add as first product name in `recentPurchases`.
      - (1) If it is already in list do not add it again.
      - (2) If the array is full, you can lose the information for the least recent item in the array. The idea is you have up to the 5 most recent products purchased in the array.
    - ✓ If amount is negative, return false.

        - `void outputRecentPurchases()`
            - ✓ Output information about each product recently purchased.
- Store Class
    - Attributes
        - `name` should be a `String`.
        - `products` should be an array that can hold up to 100 elements of type `Product`.
        - `customers` should be an array that can hold up to 100 elements of type `Customer`.
        - `numProducts` and `numCustomers` should be `int`.
        - `counter` should be a `static int`.
            - ✓ This is an additional private data member for the class. It is not part of the UML diagram, but is necessary to create default names that attempt to be unique.
    - Methods
        - `Store()`
            - ✓ Initialize the values of data members: `numProducts` and `numCustomers`.
            - ✓ Initialize the store name by the `setName` method.
        - `Store(String name)`
        - `String getName()`
        - `void setName(String storeName)`
            - ✓ The value of `storeName` cannot be an empty string, i.e., "". If it is, the value of `counter` should be incremented, and the `name` should be set to the default value "Store <num>" where <num> is the value of `counter`.
        - `boolean addProduct(int productID, String productName)`
            - ✓ If this `productID` already belongs to another product, return false.
            - ✓ If there are no more slots for a new product, return false. Otherwise, create a new Product, add it to products, and return true.
        - `Product getProduct(int productID)`
            - ✓ Find the matching product and return its corresponding object.
            - ✓ If the product does not exist, return null.
        - `boolean addCustomer(int customerID, String customerName, boolean credit)`
            - ✓ If this `customerID` already belongs to another customer, return `false`.
            - ✓ If there are no more slots for a new customer, return false. Otherwise, create a new `Customer`, add it to `customers` and return true.
            - ✓ If an argument is not provided for the `credit` parameter, set it to false by default.
        - `Customer getCustomer(int customerID)`
            - ✓ Find the matching customer and return its corresponding object.

- ✓ If the customer does not exist, return null.

- `boolean takeShipment(int productID, int quantity, double cost)`
  - ✓ Find matching `Product`.
  - ✓ If the product is not in list of `products` do nothing and return false. Otherwise, update the product with the shipment quantity and cost.
- `boolean sellProduct(int customerID, int productID, int quantity)`
  - ✓ Make the sale only if it is allowed and return true. Otherwise, do nothing and return false. In this context, allowed means that both the function calls of the product's `reduceInventory()` and the customer's `processPurchase()` have returned true.
  - ✓ Warning: Do not change the product or customer if both cannot be done successfully. Since calling one of the functions (which results in changes that cannot be undone) might return false. You have to ensure both will return true, before actually calling each function.
  - ✓ Note that the difference between `int quantity` (input parameter to `sellProduct`) and `double amount` (input parameter to `processPurchase`) is `quantity` refers to the number of items that will be sold, while `amount` refers to the total price that the customer will pay for the purchase. Therefore, in order to call `processPurchase`, you will need to calculate `amount` by using `quantity` and the price of the product that is being sold.
- `boolean takePayment(int customerID, double amount)`
  - ✓ If the customer does not exist, do nothing and return false.
  - ✓ Find matching customer, process the payment and return true.
- `void outputProducts(Product product)`
  - ✓ Output information about each product.
- `void outputCustomers(Customer customer)`
  - ✓ Output information about each customer.

- Output Functions
    - Create outputs for the Product and Customer classes. Recall, these are helpers for those classes.

Product output example
Product Name: NCHU Koozie **Product ID: 32498** **Description: A great way to keep a canned beverage cold.** **Inventory: 83** **Number Sold: 17** **Total Paid: 103.75** **Price: 1.0375**

Customer output example
Customer Name: Jason Lin **Customer ID: 2198123** **Has Credit: true** **Balance: -228.33** **Products purchased --** **NCHU T-Shirt** **NCHU Koozie**

Driver Program Guidance

- Your driver program will help you test and debug your objects as you create them. A nice thing about using a driver program, is that you do not have to validate inputs since you can ensure you are only entering the values you want to test. In many cases, you can just use a literal value instead of taking in input. Remember, this program is only for testing, so you do not have to create a rock-solid program for testing. However, member functions may need to do validation as indicated.
- Initially, you will want to create and test the Product class since it does not rely on another class. So, testing means creating and being able to see values in in the class and trying both valid and invalid inputs. We will show you some steps in how the testing evolves for the Product class. Also, as you code you should be constantly compiling every time you type fresh code to catch problems as soon as possible. It is much easier to catch errors as you go rather than write tons of code and then go back and debug. Taking that approach can double, triple, or even quadruple your development time.
- I have included a sample driver program with the starting code. This testing is **not complete**, but represents a solid starting point with coverage of most items. Edge cases are probably the most glaring omission in testing so far.
- Note: it is helpful to label what you are outputting especially as the amount of code grows. If you are methodical, develop your code incrementally, and compile and test frequently, you will produce code much faster than when you try to code everything first and then go back and fix the problems.