OOP CLASS: MARKETPLACE CASE STUDY

Mosi-oa-Tunya

GOLD COIN

1 Introduction

In this project, you are going to implement a basic marketplace model for the RBZ. This model consists of **traders**, **wallets**, **transactions**, **currencies**, and **the market**. For simplicity we have two currencies in the market: USD and GoldCoin (Mosi-oa-Tunya Gold Coins). Traders can give buying or selling orders to the market, if they can afford.

In our market model, there are two priority queues for storing orders. In a priority queue, elements having highest priority serve before. The priority rules are written in the next sections. With these rules, your market implementation should make transactions if these two priority queues' tops overlap.

We are expecting from your code, to give proper outputs for given inputs.

2 Packages & Classes:

You are provided with the general structure of the project below. In this context, there are 2 packages and 8 classes. The details of the classes, functionalities to be implemented and overall operations of the project with the corresponding format are presented in the following sections. You need to implement these classes according to the information provided in this document.

2.1. Package: **executable**

2.1.1. Main.java

The Main class should be implemented in a way that it should read corresponding parameters and events from the input file, and log the results to the output file. The input and output file paths are given in arguments of the program.

You should create a random object with the given seed. In order to access Random object everywhere, you may define it as a static object:

```
public static Random myRandom;
```

2.2. Package: **elements**

2.2.1. Market.java

There is only one market throughout the program. The fields and methods to be implemented in Market.java include but not limited to:

```
private PriorityQueue<SellingOrder> sellingOrders;
private PriorityQueue<BuyingOrder> buyingOrders;
private ArrayList<Transaction> transactions;
public void giveSellOrder(SellingOrder order);
public void giveBuyOrder(BuyingOrder order);
public void makeOpenMarketOperation(double price);
public void checkTransactions(ArrayList<Trader> traders);
public Market(int fee);
```

The additional data for implementing these methods are given in the "Implementation Details" section.

2.2.2. Order.java

Order is the parent class of BuyingOrder.java and SellingOrder.java. Every Order should have:

```
double amount;
double price;
int traderID;
public Order(int traderID, double amount, double price)
```

2.2.3. SellingOrder.java

SellingOrder.java should be a child of Order.java. In order to implement PriorityQueue<SellingOrder>, the SellingOrder method implements Comparable, and compareTo(SellingOrder e) should be overridden. The comparison logic of SellingOrder is given in "implementation details" in this document.

2.2.4. BuyingOrder.java

BuyingOrder.java should be a child of Order.java. In order to implement PriorityQueue<BuyingOrder>, the BuyingOrder method implements Comparable, and compareTo(BuyingOrder e) should be overridden. The comparison logic of BuyingOrder is given in "implementation details" in this document.

2.2.5. Trader.java

Each Trader object created in the program has an ID and a wallet. The IDs of the traders are starting from 0 and incremented when a trader object is created. A trader can give buying or selling orders into the market. However if the trader cannot afford that order, there is nothing to do.

The fields and signature of the methods that should be implemented for the Trader class include but not limited to:

```
private int id;
private Wallet wallet;
public Trader(double USD, double GoldCoin);
public int sell(double amount, double price, Market market);
public int buy(double amount, double price, Market market);
public static int numberOfUsers = 0;
```

2.2.6. Wallet.java

Each Wallet object keeps the amounts of USD and GoldCoin. The necessary fields and signature of the methods to be implemented for the Wallet class include but not limited to:

```
private double USD;
private double GoldCoin;
private double blockedUSD;
private double blockedGoldCoin;
public Wallet(double USD, double GoldCoin);
```

2.2.7. Transaction.java

Each Transaction object includes a buying and a selling order.

```
private SellingOrder sellingOrder;
private BuyingOrder sellingOrder;
```

3 Implementation Details:

★ PriorityQueue functionality in the Market:

There are two distinct PriorityQueue you have. One of them is for SellingOrders, the other is for BuyingOrders.

- **PriorityQueue<SellingOrder>** - When you poll() this PQ you should get the SellingOrder that has the lowest price. If two orders have the same price then you get the SellingOrder which has the highest amount. If it is still the same, you get the SellingOrder that has the lowest tradersID.
- **PriorityQueue<BuyingOrder>** - When you poll() this PQ you should get the BuyingOrder that has the highest price. If two orders have the same price then you get the BuyingOrder which has the highest amount. If it is still the same, you get the BuyingOrder that has the lowest tradersID.

At the end of each query, the market checks whether the prices at the PQs are overlapping. If there is an overlap, the market should make transactions with the top of PQs, until there is no overlapping.

★ Query#666 (make open market operation):

- In this query, Trader#0 (The System) gives buying or selling orders for setting the current price of GoldCoin to the given price.
- The final price can differ from the given price. The market tries to converge as much as possible.
- You should create system's orders until:

```
■ price of buyingOrders.peek()< price given by the system
■ price of sellingOrders.peek()> price given by the system
```

★ Handling Orders:

- After a BuyingOrder is given, USD reserved for this order (amount*price) cannot be used again. Therefore, you should store this amount in the blockedUSD of the trader's wallet.
- After a SellingOrder is given, GoldCoin reserved for this order (amount) cannot be used again. Therefore, you should store this amount in the blockedGoldCoin of the trader's wallet.
- When making transactions the blocked currencies should disappear.
- When making transactions the amount of BuyingOrder and the amount of SellingOrder are possibly different. In that case you should divide an order into two parts. One goes to transaction, the other returns back to its PriorityQueue.
- When making transactions the price of BuyingOrder and the price of Selling order might differ. If so, you should take the SellingOrder's price as the Transaction price. However, if this is the case, the amount of USD or GoldCoin blocked for extra price should return to the wallet.

★ Market Fee:

- The market fee is an integer representing how much commission the market receive from the transaction per thousand.
- Conducting a transaction, Seller gets `(amount*price*(1-fee/1000)`.

4 Input & Output:

- The first line of the input file specifies the random seed that you have to use to generate random numbers in the project.: A
- The second line of the input file specifies the initial transaction fee of the marketplace, the number of users, and number of queries: B C D
- The next C lines represent the initial USD and GoldCoin asset in each trader's wallet.

```
<USD_amount> <GoldCoin_amount>
<USD_amount> <GoldCoin_amount>
```

```
...
<USD_amount> <GoldCoin_amount>
<USD_amount> <GoldCoin_amount>
```

The next D lines are the queries. There are exactly 14 types of events.

Trader specific queries:

➔ 10: give buying order of specific price:
`10 <trader_id> <price> <amount>`
Example: `10 1 3 34`
Trader#1 tries to give a buying order for 34 GoldCoin with 3*34 USD.
➔ 11: give buying order of market price:
`11 <trader_id> <amount>`
Example: `11 1 34`
Trader#1 tries to give a buying order for the cheapest 34 GoldCoin order in the market.
Note: This is similar to Query#10, but it takes the current selling price as price.
Note: If there is no current selling price then increment the number of invalid queries.
➔ 20: give selling order of specific price:
`20 <trader_id> <price> <amount>`
Example: `20 1 3 34`
Trader#1 tries to give a selling order with 34 GoldQoin with 3 USD for each.
➔ 21: give selling order of market price:
`21 <trader_id> <amount>`
Example: `21 1 34`
Trader#1 tries to give a selling order for the most expensive 34 GoldCoin order in the market.
Note: This is similar to Query#20, but it takes the current buying price as price.
➔ 3: deposit a certain amount of USD to wallet:
`3 <trader_id> <amount>`
Example: `3 1 43`
Trader#1 deposits 43 USD to wallet#1.
➔ 4: withdraw a certain amount of USD from wallet:
`4 <trader_id> <amount>`
Example: `4 1 43`
Trader#1 withdraws 43 USD from wallet#1
➔ 5: print wallet status:
`5 <trader_id>`
Prints "Trader <traderID>: <trader_s_USD>$ <trader_s_GoldCoin>GC"
Example: Trader 5: 34.0$ 23.0GC

System queries:
➔ 777: give rewards to all traders:
`777` - When this query is read, the system creates and distributes random amounts of GoldCoin to all traders. for each trader add `myRandom.nextDouble()*10` GoldCoin to the trader's wallet.
➔ 666: make open market operation:
`666 <price>` - When this query is read, the system compensates buying or selling orders in order to set the price of GoldCoin to the given price. This query can increase or decrease the total amount of GoldCoin in the market.
➔ 500: print the current market size:
`500` - Prints "`Current market size: <total_$_in_buying_pq>`

`<total_GoldCoin_in_selling_pq>"` to the output file.

→ 501: print number of successful transactions:

`501` - Prints "`Number of successful transactions: <num_of_successful_transaction>`" to the output file.

→ 502: print the number of invalid queries:

`502` - Prints `<number_of_invalid_queries>` to the output file.

Note: If a trader is unable to satisfy the query's liabilities, then the number of invalid queries is incremented by one.

→ 505: print the current prices:

`505` - Prints; "`Current prices: <cp_buying> <cp_selling> <cp_average>` to the output file.

Note: if one of the PriorityQueues is empty then the price related to it, is not included in the average current price.

→ 555: print all traders' wallet status

`555` - Prints "<trader_s_USD> $ <trader_s_PQ" of all traders.

NB

- The method signatures and field names should be exactly the same with the ones specified in this document.
- You can implement and utilize additional helper methods of your choice.
- You should implement getters and setters, if it is necessary.
- Please keep in mind that providing the necessary accessibility and visibility is important: you should not implement everything as public, even though the necessary functionality is implemented. The usage of appropriate access modifiers and other Java keywords (super, final, static etc.) play an important role in this project since there will be partial credit specifically for the software design.
- You should document your code in Javadoc style including the class implementations, method definitions (including the parameters and return if available etc.), and field declarations. You are not going to submit a documentation file generated by Javadoc.
- You may use Java's PriorityQueue classes as well as you may implement yours.
- Do not make assumptions about the numbers and the size of the input.
- All amounts and prices are given in float.
- The current price is calculated:
  - the top of selling priority queue when buying from the market
  - the top of buying priority queue when selling from the market
  - the average of the tops of buying and selling priority queues when the system prints the current price. If one of the PriorityQueues is empty then the price related to it, is not included in the average current price.
- Trader#0 is reserved for the system. If you test the code without awareness of this, your output can differ from what you expect.
- Do not forget to use the given random seed.
- If a trader does not have enough assets for buying, selling, or withdrawing, the number of invalid queries increases.