AY 2020 Assignment 2 [8 Marks]

Date / Time	26 September 2020 – 16 October 2020 23:59
Course	**[M1522.600] Computer Programming**
Instructor	Youngki Lee

- You can refer to the Internet or other materials to solve the assignment, but you *__SHOULD NOT__* discuss the question with anyone else and need to code **ALONE**.
- We will use the automated copy detector to check the possible plagiarism of the code between the students. The copy checker is reliable so that it is highly likely to mark a pair of code as the copy even though two students quickly discuss the idea without looking at each other's code. Of course, we will evaluate the similarity of a pair compared to the overall similarity for the entire class.
- We will do the manual inspection of the code. In case we doubt that the code may be written by someone else (outside of the class), we reserve the right to request an explanation about the code. We will ask detailed questions that cannot be answered if the code is not written by yourself.
- If one of the above cases happen, you will get 0 marks for the assignment and may get a further penalty. Please understand that we will apply these methods for the fairness of the assignment.

- Download and unzip “HW2.zip” file from the ETL. “HW2.zip” file contains skeleton codes for Question 1 (in the “problem1” directory) and Question 2 (in the “problem2” directory).
- Do not modify the overall directory structure after unzipping the file, and fill in the code in appropriate files. It is okay to add new directories or files if needed.
- Once you have completed your homework, compress the “HW2” directory in a single zip file named “**20XX-XXXXX.zip**” (your student ID) and upload it to ETL as you submit the solution for the lab tests. Contact the TA if you are not sure of the process. Make sure to double-check if your final zip file is properly submitted. If the submission file name is wrong, you get a 5% deduction.
- Java Collection Framework is allowed.
- Do not use any external libraries.

Contents

Submission Guidelines

1. You should submit your code on eTL.
2. After you extract the zip file, you must have a HW2/ directory. The submission directory structure should be as shown in the table below. Note that src/ directories are added to each of the problem directories compared to Assignment1.
3. You can create additional directories or files in each src/ directory.
4. You can add additional methods or classes, but do not remove or change signatures of existing methods.
5. Compress the “HW2” directory and name the file “20XX-XXXXX.zip” (your student ID).
6. The Autolab server for Assignment 2 will be available in a week.

Submission Directory Structure (Directories or Files can be added)

- Inside HW2/ directory, there should be problem1/ and problem2/ directory.

Directory Structure of Problem 1	Directory Structure of Problem 2

problem1/
└── src/
  └── security/...
  └── bank/
  └── event/...
  ├── MobileApp.java
  ├── Session.java
  ├── SessionManager.java
  ├── Bank.java
  └── BankAccount.java
  └── Test.java
	problem2/
	└── src/
	  └── hand/
	  └── agent/
	  ├── Agent.java
	  ├── Buyer.java
	  └── Seller.java
	  └── market/
	  └── Market.java
	  └── Test.java

Question 1: Secure Banking [5 Marks]

Objectives: Develop a secure mobile banking service that supports financial transactions like deposit, withdrawal, and transfer with secure transactions.

Description: “Bank of SNU” plans to open an online banking service to enable a range of financial transactions through a mobile banking application. You are asked to implement this service with Java applying the Object-Oriented Programming (OOP) concept.

The problem consists of three parts. Firstly, you will implement a simple form of `Bank` class that supports various transactions. Secondly, you will implement the `Session` class to minimize the effort of authentication for multiple transactions. Finally, you will implement a `MobileApp` class and emulate secure transactions between `MobileApp` and the `Bank` classes. Note: we will not implement a real mobile app nor communication across different devices; they are just conceptual entities. All the implementations will be done within a single Java program.

Notes

- Feel free to add new classes if necessary.
- Feel free to modify member attributes or implementations of methods of the given classes in the skeleton code unless we instruct otherwise.
- However, **DO NOT** modify the signature of the given methods (i.e., return type, method name, and parameter types). The exact signature of methods will be used for the final evaluation.
- You do not need to consider corner cases that we did not describe.
- Test cases are introduced as the `Test.java`.

Question 1.1: Bank Account [2 Marks]

Objective:

- Implement six member methods of the `BankAccount` class in the `bank` package (i.e., `BankAccount`, `authenticate`, `deposit`, `withdraw`, `receive`, `send`)
- Implement five member methods of the `Bank` class in the `bank` package (i.e., `deposit`, `withdraw`, `transfer`, `getEvents`, `getBalance`)

Description: On creating a personal savings account, a `BankAccount` object is created to manage the account information of a client. The `Bank` class is responsible for storing and managing multiple `BankAccount` objects for all clients. A client can `createAccount`, `deposit`, `withdraw` and `transfer` from his/her account through a `Bank` object; a client cannot directly access the `BankAccount` object.

Event Class Description

- Use the provided Event class and its subclasses in the bank.event package to implement Bank and BankAccount classes. Event classes are used to keep track of the history of transactions. Upon each transaction, an appropriate Event object is created and stores the information regarding the transaction.
- There are four subclasses of the Event class: DepositEvent, WithdrawEvent, SendEvent, and ReceiveEvent.
- Please DO NOT modify source codes of the Event classes. We would stick to what we have provided for the final evaluation, even if you modified the package.

BankAccount Class Specifications

Implement the following methods to handle different transactions. The class will also manage the history of transactions using the Event class described above; upon each transaction, an Event object is created and stored in the events array. Assume that the event array can store up to 100 events, and no more than 100 events are stored per BankAccount.

- BankAccount(String id, String password, int balance)
  - Construct the BankAccount object and initialize its id, password and balance attributes with the given parameter values.
- boolean authenticate(String password)
  - Check if the account's password is equal to the given password.
  - Return true if and only if the password strings are equal.
- void deposit(int amount)
  - Add the amount to the balance, and add a DepositEvent object to the events array.
- boolean withdraw(int amount)
  - Check if the balance is larger than or equal to the amount. If yes, subtract the amount from the balance, add a WithdrawEvent object to the events array, and return true.
  - Otherwise, return false.
- void receive(int amount)
  - Add the balance by the amount, and add a ReceiveEvent object to the events array.
- boolean send(int amount)
  - Check if the balance is larger than or equal to the amount. If yes, subtract the amount from the balance, add a SendEvent object to the events array, and return true.
  - Otherwise, return false.

Bank Class Specifications

Implement the following member methods. You will need to use appropriate member methods of the BankAccount class (Consider implementing BankAccount Class first!). All methods except for createAccount require a password authentication prior to the corresponding transaction. Assume that the maximum number of bank accounts that a bank manages is 100.

- public void createAccount(String id, String password) and
  public void createAccount(String id, String password, int initBalance)
  - This method is provided.
  - Create a BankAccount object with the given account id, password, and the initial balance. If the initial balance is not given, set the initial balance to 0.
  - The negative initBalance is not considered for the final evaluation.
- public boolean deposit(String id, String password, int amount)
  - Authenticate the client with the id and password.
  - If the authentication is not successful, do nothing, and return false.
  - If the authentication is successful, add the amount to the balance and return true.
  - Use the deposit method of the BankAccount class.
- public boolean withdraw(String id, String password, int amount)
  - Authenticate the client with the id and password.
  - If the authentication is not successful, do nothing, and return false.
  - If the authentication is successful, subtract the amount from the account's balance.
  - Return false if there is not enough balance to withdraw. Otherwise return true.
  - Use the withdraw method of the BankAccount class.
- public boolean transfer(String sourceId, String password, String targetId, int amount)
  - Authenticate the source account with the sourceId and password.
  - If the authentication is not successful, do nothing, and return false.
  - If the authentication is successful, transfer the amount to the targetId's account.
  - Return false if the amount is larger than the balance of the source account or there is no account with the given targetId. Otherwise, return true.
  - Use the send and receive method of the BankAccount class.
- public Event[] getEvents(String id, String password)
  - Authenticate the client with the id and the password.
  - Return null if the authentication fails.
  - Return the array of Events that were **recorded** upon the deposit, withdraw, and transfer method calls.
    - The returned array should not contain null.
    - More recent Events must be located **before** the older Events in the array.
- public int getBalance(String id, String password)
  - Authenticate the client with the id and the password.
  - Return the balance of the corresponding account. Return -1 if the authentication fails.

Question 1.2: One-time Authentication with Sessions [1 Marks]

Objectives:

- Implement the three methods (deposit, withdraw, transfer) of the Bank class in the bank package.
- Implement the three methods (deposit, withdraw, transfer) of the Session class in the bank package.
- Implement a method (expireSession) of the SessionManager class in the bank package.

Description: In Question 1.1, it was cumbersome to pass an account id and the password for authentication upon every transaction. Now, we will simplify this process using a new feature called a “session”. A user is provided with a session after the initial authentication (via the provided generateSession method of the SessionManager), and all following transactions can be performed with the session without explicit authentications; more specifically, upon the session generation, a session key is created and the key is used in the following transactions (instead of using a password and id). A session expires after a certain number of transactions (via the expireSession method), and the expired session cannot be used for transactions.

Bank Class Specifications

This class should be extended to process various transactions using a session. Implement the following three methods using the provided getAccount method.

- boolean deposit(String sessionkey, int amount)
    - Find the bank account corresponding to the given sessionkey. Use the getAccount method to retrieve the bank account with the sessionkey.
    - Add the amount to the account balance, and return true.
- boolean withdraw(String sessionkey, int amount)
    - Find the bank account corresponding to the given sessionkey.
    - Return false if there is not enough balance to withdraw.
    - Otherwise, subtract the amount from the bank account’s balance, and return true.
- boolean transfer(String sessionkey, String targetId, int amount)
    - Find the bank account corresponding to the given sessionkey.
    - Return false if the amount is larger than the balance of the bank account or there is no account with the given targetId.
    - Otherwise, transfer the amount from the bank account to the account with the targetId, and return true.

`Session` Class Specifications

This class enables a client to perform various transactions using the session. Implement the following three methods. Constructor of the class is already implemented. The member attribute `sessionkey` and `bank` is set in the constructor. Use the above `Bank` class methods for implementation.

- `public boolean deposit(int amount)`
    - Return false only if the session has expired.
    - Otherwise, call the `bank`'s `deposit` with the `sessionkey` and `amount` and return its output.
- `public boolean withdraw(int amount)`
    - Return false if the session has expired.
    - Otherwise, call the `bank`'s `withdraw` with the `sessionkey` and `amount` and return its output.
- `public boolean transfer(int targetId, int amount)`
    - Return false if the session has expired.
    - Otherwise, call the `bank`'s `transfer` with the `sessionkey`, `targetId` and `amount` and return its output.

`SessionManager` Class Specifications

This class is responsible for generating and expiring a session for a client. Implement the following method.

- `public static void expireSession(Session session)`
    - Expire the `session`. All the method calls through the expired `session` should not do any actions.

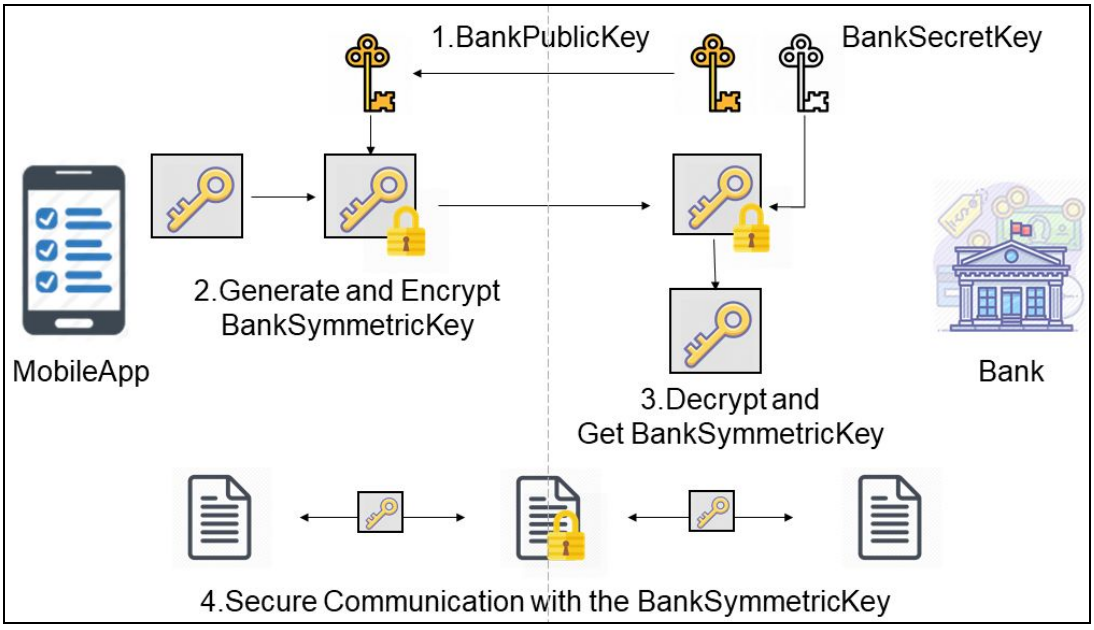
Question 1.3: Secure Mobile Banking [2 Marks]

Objectives:

- Implement four methods (`sendSymKey`, `deposit`, `withdraw`, and `processResponse`) of the `MobileApp` class in the `bank` package.
- Implement two methods (`processRequest` and `fetchSymKey`) of the `Bank` class in the `bank` package.

Descriptions: In Question 1.1 and 1.2, we assumed scenarios where customers directly access the bank management system through the `Bank` class. Now, we would like to help the customers to access the banking service through a mobile application (represented with the `MobileApp` class). Note that this is not a real mobile application; rather, it is a java class emulating the behavior of a mobile application. In addition, to make the banking service secure and prevent hackers from manipulating the client's financial transactions, we would like to enable secure transactions between the `MobileApp` object and the `Bank` object; again, we are not really implementing secure networking, but just emulating secure transactions to practice the OOP concept.

Before you implement the methods, you will need to understand a new concept, a 'handshake protocol' (See. https://youtu.be/sEkw8ZcxtFk?t=166), which is a standard way to establish a secure transaction channel. The following figure and texts describe key steps of the protocol.



When a MobileApp needs to communicate with a Bank for secure transactions, it first creates a secure channel through the following 'handshaking' steps.

1. The Bank generates two keys: BankPublicKey and BankSecretKey. It sends BankPublicKey to the MobileApp, and stores the BankSecretKey internally.
2. The MobileApp generates BankSymmetricKey and encrypts it with the received BankPublicKey. As a result, the Encrypted<BankSymmetricKey> is created and transmitted to the Bank.
3. The Bank decrypts the received Encrypted<BankSymmetricKey> with the BankSecretKey (created in Step 1) to obtain the BankSymmetricKey.

Once the handshake is complete, both the Bank and the MobileApp have a shared BankSymmetricKey, which is used to encrypt and decrypt the data for the subsequent transactions. The MobileApp and the Bank can use the Encrypted<T> class to encrypt and decrypt the data with BankSymmetricKey, respectively.

In the skeleton code, the 'handshake protocol' is implemented via the handshake method of the Protocol class in the security package. For a MobileApp to send transaction requests to a Bank, it first needs to perform Protocol.handshake, and then uses the Protocol.communicate method to conduct follow-up transactions. Note: we will test your submission with these two methods of Protocol class.

When you look at the handshake and communicate methods, you can see that they are implemented by calling adequate member methods of the MobileApp class and the Bank class. Thus, the main goal of this problem is to fill in these member methods to make the handshake and communicate methods fully working.

Before jumping into the implementation, you may want to carefully look at the three provided classes, the Protocol class, the Encrypted<T> class and the Message class. We already implemented these three classes, and you do not have to modify them.

Protocol Class Descriptions

This class is responsible for enabling i) the handshake between a mobile application and a bank and ii) secure communications between the mobile application and the bank for subsequent transactions.

- public static void handshake(MobileApp mobileApp, Bank bank)
    - This method implements the handshake protocol (step 1-3).
    - It invokes the bank.getPublicKey (step 1), mobileApp.sendSymKey (step 2), bank.fetchSymKey (step 3) in a sequence.
- public static boolean communicate(Deposit deposit, MobileApp mobileApp, Bank bank,

int amount)
    - This method enables a secure deposit transaction through secure communication.
    - The first argument is used to identify the type of the transaction.
    - In this method, mobileApp.deposit, bank.processRequest, and mobileApp.processResponse are invoked in sequence. It is your job to implement three methods to make the secure deposit successful.
- public static boolean communicate(Withdraw withdraw, MobileApp mobileApp, Bank bank, int amount)
    - This method enables the secure withdrawal transaction through secure communication.
    - The first argument is used to identify the type of the transaction.
    - In this method, mobileApp.withdraw, bank.processRequest, and mobileApp.processResponse are invoked in sequence. It is your job to implement three methods to make the secure withdrawal successful.

Encrypted<T> Class Descriptions

This class is responsible for encrypting and decrypting T-typed data using a proper key.

- public Encrypted(T obj, [BankSymmetricKey/BankPublicKey] key)
    - This method emulates the encryption of the given obj with the key. Specifically, it stores a T object as a private attribute, which can be only accessed with the corresponding key.
    - The key could be either a BankSymmetricKey object (used for transactions) or BankPublicKey object (used for handshake).
- public T decrypt([BankSymmetricKey/BankSecretKey] key)
    - This method emulates the decryption of the stored encrypted object. Specifically, it retrieves the stored T object only if the key is the right key.
    - The key could be either a BankSymmetricKey object (used for transactions) or BankSecretKey object (used for handshake).
    - The data encrypted with a BankPublicKey object can only be decrypted with the **paired** BankSecretKey object.
    - The data encrypted with a BankSymmetricKey object can only be decrypted with the **same** BankSymmetricKey object.
    - If the key does not match, it returns null, indicating the failure of the decryption.

Message Class Descriptions

This class is used to format the information of a transaction when a MobileApp makes a transaction request to the Bank. The class has the following attributes. Also, it provides a constructor to initialize the attributes and getters to access individual attributes.

- String requestType: The type of the transaction request. It can be either “deposit” or

"withdraw".
- String id, password: The authentication information of the customer.
- int amount: The argument for the deposit and withdraw calls.

Now, you are ready to implement the methods of the MobileApp class and the Bank class. See the specifications below for details. Note: Please do not modify the source code in the security package. We will use the original security package for the final evaluation.

MobileApp Class Specifications

Implement the following methods to support secure transactions. Every MobileApp object is initialized with a unique String AppId generated by the getAppId method.

- public MobileApp(String id, String password)
  - This method is provided.
  - Sign in to the mobile application with the given id and password.
  - Sets the member attribute id and password.
- public Encrypted<BankSymmetricKey> sendSymKey(BankPublicKey publickey)
  - This method performs the step 2 of the handshake protocol, i.e., encrypting a BankSymmetricKey with the publickey and sending it to the bank.
  - You need to generate a random string with the randomUniqueStringGen method, and create a BankSymmetricKey object with it.
    - You should store the created BankSymmetricKey object for further communications.
  - Then, you need to encrypt the created BankSymmetricKey object with the given publickey and return the Encrypted<BankSymmetricKey>.
- public Encrypted<Message> deposit(int amount)
  - This method constructs an encrypted message to deposit the money. The encrypted message is used by the processRequest method of the Bank class.
  - You should create a Message object with the String "deposit", id, password and amount.
  - Then, you need to encrypt the Message object with the BankSymmetricKey object(generated by the sendSymKey), and return the Encrypted<Message>.
- public Encrypted<Message> withdraw(int amount)
  - This method constructs an encrypted message to withdraw the money. The encrypted message is used by the processRequest method of the Bank class.
  - You should create a Message object with the String "withdraw", id, password and amount.
  - Then, you need to encrypt the Message object with the BankSymmetricKey object (generated from the sendSymKey), and return the Encrypted<Message>.
- public boolean processResponse(Encrypted<Boolean> obj)
  - This method decrypts the encrypted response from the Bank.
  - Return false if the obj is null.

    - Otherwise, decrypt the obj with the BankSymmetricKey object (generated from the sendSymKey). If decryption fails return false, otherwise return the value of the decrypted output.

Bank Class Specifications

Implement the following two member methods: fetchSymKey and processRequest. Note that the getPublicKey method is already implemented.

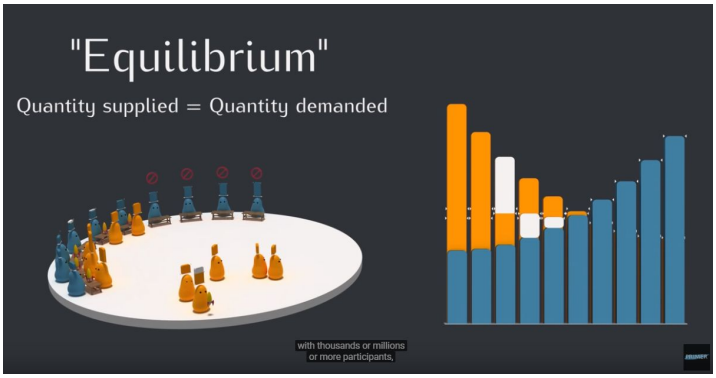
- public BankPublicKey getPublicKey()
    - This method is provided.
    - Generate a (BankPublicKey, BankSecretKey) key **pair**.
        - Note that the Encrypted<T> object encrypted with a BankPublicKey object can only be decrypted with the **paired** BankSecretKey object.
    - Store the BankSecretKey object to the member attribute secretkey and return the BankPublicKey object.
- public void fetchSymKey (Encrypted<BankSymmetricKey> encryptedkey, String AppId)
    - This method performs the step 3 of the handshake protocol, i.e., decrypting an encrypted BankSymmetricKey with the BankSecretKey object to retrieve it.
    - Decrypt the encryptedkey with the secretkey, and store the decrypted BankSymmetricKey object. Note that the BankSymmetricKey object should be stored together with the AppId, so that the correct keys can be found for different mobile applications.
        - Assume that the maximum number of handshakes is 10,000.
        - If fetchSymKey is called multiple times for the same AppId, the old BankSymmetricKey object should be replaced with the new one.
    - If the encryptedkey is null, or decryption fails with the BankSecretKey, do not store anything.
- public Encrypted<Boolean> processRequest(Encrypted<Message> messageEnc, String AppId)
    - This method processes the encrypted request from the MobileApp and returns the encrypted response.
    - Find the BankSymmetricKey object corresponding to the AppId.
        - If the BankSymmetricKey does not exist for a given AppId, return null.
    - Decrypt the messageEnc with the BankSymmetricKey object.
        - If the messageEnc is null or decryption fails with the BankSymmetricKey, return null.
    - Retrieve the request information from the decrypted Message object and call the appropriate Bank methods. The final evaluation only considers message objects with "deposit" and "delete" requests.
    - Fetch the boolean result of the invoked method, encrypt it with the BankSymmetricKey object and return it.

Question 2: Invisible Hand [3 Marks]

Objective: In this problem, we will simulate a simple free-market economy and demonstrate the equilibrium state.

Description: There is a market. There are sellers and buyers that have a specific role in it. There are 10 rounds of exchanges every day. In each round, sellers and buyers are tied in pairs to exchange an item. To make things simple, let's assume that there is only one type of item. Also, each buyer(seller) can buy(sell) at most one item a day; for instance, if a buyer purchases an item in round 5, he cannot buy any more from round 6.

Note: You'll face a new Java Collection class called `ArrayList` in the skeleton code of this problem. ArrayList is a variable length Collection that works like an Array. You can use `E get(int index)`, `E set(int index, E element)` and `int size()` to use the same functionality of Array like [] and `int length()`. Additional information is available in [ArrayList(Java SE 11 & JDK 11)].



Question 2-1: Greedy Humans [1 Marks]

Objective: Implement `willTransact` and `reflect` methods in `Buyer` and `Seller` classes (see Buyer.java and Seller.java). Note that the base class `Agent` is given.

Description: Just like our daily lives, each buyer and seller has the following two parameters.

- **Price limit**
  - Seller has its own value that is the lower bound of the price that the seller wants to sell.
  - Buyer has its own value that is the upper bound of the price that the seller wants to pay.
- **Expected price**
  - Even if the buyer's budget is sufficient, the buyer will want to buy an item at the lowest possible price. Likewise, the seller wants to sell it at the highest possible price. You can consider this as more realistic, desired prices to sell or buy.

How are price limits and expected price used? In each round, sellers and buyers make decisions only based on the expected prices. However, at the end of each day, they have a time of reflection. During reflection, they adjust the expected price depending on whether they were able to make a deal on that day. In particular, the expected prices may be adjusted but will never go beyond the price limits.

- If they could make a deal, they would want a better price the next day. That is, the seller will raise the expected price, and the buyer will lower it.
- Conversely, if they couldn't make a deal, the seller will lower the expected price, and the buyer will raise it. If the adjusted price goes beyond the price limit, it is set to the price limit.

Read the skeleton code and complete implementation of the following 4 methods.

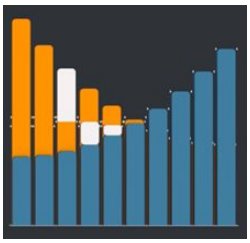
`Buyer` Class Specifications

- `boolean willTransact(double price)`
  - It returns true if and only if it hasn't made a transaction that day and the price is less than or equal to its expected price. Do not care about floating point errors.
- `void reflect()`
  - If it made a transaction that day, decrement `expectedPrice` field of `Agent` class (i.e., superclass of `Buyer` and `Seller`) by the value of `adjustment` field. Otherwise, increment it by `adjustment`. However, if adjusted `expectedPrice` is greater than its value of `priceLimit` field, set `expectedPrice` equal to `priceLimit`. Finally, in all cases, reset `hadTransaction` member variable to false for the next day.

`Seller` Class Specifications

- `boolean willTransact(double price)`
  - It returns true if and only if it hasn't made a transaction that day and the price is greater than or equal to its expected price. Do not care about floating point errors.
- `void reflect()`
  - Very similar to the method described above. Just reverse the direction of change of expected price so that it makes sense for the logic of sellers.

Question 2-2: Free Market [1 Marks]



Objective: Simulate the free market using the Buyer and Seller classes. The simulation period is 1,000 days. As described, there are 10 rounds a day. In each round, buyers and sellers will be paired and try to exchange an item; pairs must be decided with the given matchedPairs(int day, int round) method. For this, you should implement the following simulate method in Market.java.

Market Class Explanation

- double simulate()
    - Repeat the following for 1000 times (=1000 days)
        - Repeat the following for 10 times (=10 rounds)
            - Get matched <seller, buyer> pairs using matchedPairs(int day, int round) method.
            - For each matched pair, the seller will suggest his expected price to the buyer. If the buyer is satisfied by the price, call the makeTransaction method of both objects.
        - call the reflect method of every buyer and seller. You should call this method even on the last day.
    - You may return any double value for now. Question 2-3 will specify what to return.

- List<Pair<Seller, Buyer>> matchedPairs(int day, int round)
    - It returns a list of <seller, buyer> pairs, for the specified day and round.
    - In fact, the intention of this function is to randomly match buyers and sellers. However, unlike real randomness, this function always returns the same result for a given day and round.

Question 2-3: Equilibrium [1 Marks]

Objective: Economists found that that the average price of an item converges to the value corresponding to the intersection of two curves, each made from the price limits of buyers and sellers. Here we will use this property to find the intersection of two polynomials (average of the prices of all exchanges).

Implement or modify the following 3 methods.

`Market` Class Specification

- `double simulate()`
    - It now has to return the average of the prices of all exchanges made in the last(=1000th) day
    - That is, if there were 3 seller-buyer pairs who made a deal on the last day, each at the price of 300, 400, 500, then the function should return 400.
- `List<Buyer> createBuyers(int n, List<Double> f)`
    - Create and return n buyers described by a polynomial f. Each buyer $i = 1, 2, \ldots, n$ should have a price limit of $f(i/n)$ (Note that index i starts from 1 not 0).
        - E.g., when n=3, there should be 3 buyers with a price limit of $f(0.333\ldots)$, $f(0.666\ldots)$, and $f(1)$.
    - $f = [a_n, \ldots, a_1, a_0]$ represents $f(x) = a_n x^n + \ldots + a_1 x + a_0$. It is guaranteed that the length of f is at least 1.
        - Don't worry about underflow or overflow. Test cases will not test these.
- `List<Seller> createSellers(int n, List<Double> f)`
    - It is the same as above except for changing the `Buyer` to `Seller`.