AY 2022 Assignment 2 [8 Marks]

Date / Time	30 September 2022 – 21 October 2022 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, 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.
- For the first event of plagiarism, you will get 0 marks for the specific assignment. You will fail the course and be reported to the dean if you copy others' code more than once.
- Download and unzip "HW2.zip" file from the autolab. "HW2.zip" file contains skeleton codes for Question 1 (in the "problem1" directory) and Question 2 (in the "problem2" directory).
- When you submit, compress the "HW2" directory which contains "problem1" and "problem2" directories in a single zip file named "20XX-XXXXX.zip" (your student ID) and upload it to autolab as you submit the solution for the HW1. Contact the TA if you are not sure how to submit. Double-check if your final zip file is properly submitted. You will get 0 marks for the wrong submission format.
- 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.
- Java Collections Framework is allowed.
- Do not use any external libraries.

Contents

Submission Guidelines

1. You should submit your code on the autolab.
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.
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).

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
  ├── Client.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 by applying the Object-Oriented Programming (OOP) concept.

The problem consists of three parts. Firstly, implement a simple form of Bank, BankAccount, and Client classes that support various transactions. Secondly, implement the Session class to minimize the effort of authentication for multiple transactions. Finally, implement a MobileApp class and emulate secure transactions between MobileApp and the Bank classes. You 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 methods will be used for the 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 five member methods of the BankAccount class in the bank package (i.e., BankAccount, deposit, withdraw, receive, send)
- Implement six member methods of the Client class in the bank package (i.e., Client, getMembership, authenticate, setAuthenticatedFalse, findAccount, createAccount)
- Implement seven member methods of the Bank class in the bank package (i.e., createClient, createAccount, deposit, withdraw, transfer, getEvents, getBalance)

Description:

- The Bank class is responsible for storing and managing multiple Client objects.
- A Client object has client information such as id, password, membership, and possibly multiple BankAccount objects. A client obtains “VIP” membership if the total balance exceeds 10,000 across all his accounts, otherwise “Normal” membership. A client can createAccount, deposit, withdraw and transfer from its account through a Bank object.
- A BankAccount object is created to manage the account information of a client.

Event Class Description

- Use the provided Event class and its subclasses in the bank.event package to implement Bank and BankAccount classes.There are four subclasses of the Event class: DepositEvent, WithdrawEvent, SendEvent, and ReceiveEvent.
- 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.
- Please **DO NOT** modify source codes of the Event class and four subclasses of it.

BankAccount Class Specifications

Implement the following methods to handle different transactions. The class also manages 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 events array can store up to 100 Event objects, and no more than 100 objects are stored per BankAccount.

- BankAccount(String accountID, int balance)
  - Construct the BankAccount object and initialize its accountId, and balance attributes with the given parameter values.
- void deposit(int amount)
  - Add the amount to the balance, and add a DepositEvent object to the events array.
  - There will be no corner case with the amount value less than 0.
- boolean withdraw(int amount, String membership)
  - The withdrawal is done only when the sum of amount and fee do not exceed the balance. No fee is applied for a “VIP”. For a “Normal” member, the fee is 5.
  - If the balance is sufficient, subtract the amount and fee from the balance, add a WithdrawEvent object to the events array, and return true. Otherwise, return false.
  - There will be no corner case with the amount value less than 0.
- void receive(int amount)
  - Add the balance by the amount, and add a ReceiveEvent object to the events array.
  - There will be no corner case with the amount value less than 0.
- boolean send(int amount, String membership)
  - Similar to withdrawal, the transfer can be done when the sum of amount and fee do not exceed the balance. No fee is applied for a “VIP”. For a “Normal” member, the fee is 5.
  - If the balance is enough, subtract the amount and fee from the balance, add a SendEvent object to the events array, and return true. Otherwise, return false.
  - There will be no corner case with the amount value less than 0.

Client Class Specifications

Implement the following methods to manage the client's information (i.e., id, password, membership, accounts, isAuthenticated). You will need to use appropriate methods of the BankAccount class. It has an accounts array; BankAccount objects are created and stored in this array. Assume that the accounts array can store up to 10 BankAccount objects, and no more than 10 objects are stored per Client.

- Client(String ID, String password)
  - Construct the Client object and initialize its ID, and password attributes with the given parameter values.
- String getMembership()
  - Return the client's membership status. If the total balance across all accounts exceeds 10,000, return "VIP", otherwise, return "Normal".
- boolean authenticate(String password)
  - This method returns the authenticated value. If the client's password is equal to the given password, set the client's authenticated attribute to true. Else, do not change the authenticated.
- void expireAuthenticatedState()
  - Set the client's authenticated to false.
- BankAccount findAccount(int accountID)
  - If the client has a BankAccount object with the given accountID, return the BankAccount object. Else, return null.
- boolean createAccount(int accountID, int initBalance)
  - Create a BankAccount object with the given accountID and initBalance. If the creation is successful, add the BankAccount object to the accounts array and return true.
  - If the given accountID already exists for the client, do not create the account, and return false (ignore the request).
  - The negative initBalance is not considered for the evaluation.

Bank Class Specifications

Implement the following member methods. You will need to use appropriate methods of the BankAccount class and the Client class (Consider implementing BankAccount class and Client class first!). Assume that the maximum number of bank accounts that a client manages is 10, and the maximum number of clients that a bank manages is 100.

There are two types of tasks to be done before making a transaction; **(1) password authentication**, and **(2) check the existence of BankAccount**. For createAccount, deposit, withdraw, and transfer methods, do **password authentication** before the transaction (use the authenticate method in Client class). If the authentication is not successful, do nothing, and

return false. If and only if the authentication is successful, perform the transaction. For deposit, withdraw, and transfer methods, after the password authentication, check whether the client **has a BankAccount matching to the specified accountID**. If and only if there is one, perform the transaction. Otherwise, do nothing and return false.

After the transaction of createAccount, deposit, withdraw, or transfer method, **set the client's authenticated attribute to false** before ending the method (use the expireAuthenticatedState method in Client class).

- public void createClient(String id, String password)
  - Create a Client object with the given id and password, and add a Client object to clients array.
  - If the given id already exists in the bank, do not create the client.
- public void createAccount(String id, String password, int accountID) and public void createAccount(String id, String password, int accountID, int initBalance)
  - Create a BankAccount object for the client with the given id, password, and accountID.
- public boolean deposit(String id, String password, int accountID, int amount)
  - 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 accountID, int amount)
  - Return false if there is not enough balance to withdraw. Otherwise return true.
  - Use the withdraw method of the BankAccount class.

double check this b4 submitting (the loops)

- public boolean transfer(String sourceId, String password, int sourceAccountID, String targetId, int targetAccountID, int amount)
  - If there is no Client with targetID, return false. Similarly, if the Client object with targetID doesn't have a BankAccount with targetAccountID, return false.
  - If all prior actions are successful, transfer the amount to the target client's account.
  - Return false if there is not enough balance in sourceAccountID to transfer.
  - Use the send and receive method of the BankAccount class.
- public Event[] getEvents(String id, String password, int accountID)
  - Authenticate the client with the password, and then check whether the client has a BankAccount with accountID. If not, return null.
  - Return the array of Events that were **recorded** upon the deposit, withdraw, and transfer method calls in the corresponding BankAccount.
    - The returned array should not contain null.
    - More recent Events must be located **after** the older Events in the array.
- public int getBalance(String id, String password, int accountID)
  - Authenticate the client with the password, and then check whether the client has a BankAccount with accountID. If not, return -1.
  - Return the balance of the corresponding BankAccount.

Question 1-2: One-time Authentication with Sessions [1 Marks]

Objectives:

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

Description: In Question 1-1, it was cumbersome to pass an id, password and accountID 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 an id, password and accountID). A session expires after a certain number of transactions (=transLimit) (via the expireSession method), and the expired session cannot be used for transactions.

Bank Class Specifications

This class should be extended to process transactions using a session. Implement the following three methods. Use the getAccount method to retrieve the bank account with the sessionkey. There will be no corner case when a Bank fails to find a BankAccount object corresponding to the sessionkey.

- public boolean deposit(String sessionkey, int amount)
    - Find the bank account corresponding to the given sessionkey.
    - Add the amount to the account balance, and return true.
- public 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.
- public boolean transfer(String sessionkey, String targetId, int targetAccountID, 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 and targetAccountID.
    - Otherwise, transfer the amount from the bank account to the account with the targetId and targetAccountID, and return true.

Session Class Specifications

This class enables a client to perform various transactions using the Session object. Implement the following three methods. Note that there is a transLimit (=3), which is the maximum number of calls to these three methods. Constructor of the class is already implemented. The member attribute sessionkey and bank is already set in the constructor. Use the above Bank class methods for implementation.

- public boolean deposit(int amount)
    - Return false if the session has expired.
    - Otherwise, call the Bank's deposit method with the sessionkey and amount and return the method's output.
    - After calling the Bank's method, check if the number of calls for deposit/withdraw/transfer methods exceeds the transLimit. If it is equal or greater than transLimit, the session should be expired.
- public boolean withdraw(int amount)
    - Return false if the session has expired.
    - Otherwise, call the Bank's withdraw method with the sessionkey and amount and return the method's output.
    - After calling the Bank's method, check if the number of calls for deposit/withdraw/transfer methods exceeds the transLimit. If it is equal or greater than transLimit, the session should be expired.
- public boolean transfer(int targetId, int targetAccountID, int amount)
    - Return false if the session has expired.
    - Otherwise, call the Bank's transfer method with the sessionkey, targetId, targetAccountID and amount and return the method's output.
    - After calling the Bank's method, check if the number of calls for deposit/withdraw/transfer methods exceeds the transLimit. If it is equal or greater than transLimit, the session should be expired.

is a failed attempt to transfer/withdraw
also counted as a session?

SessionManager Class Specifications

This class is responsible for generating and expiring a session for a Client's BankAccount object. Implement the following method.

- public static void expireSession(Session session)
    - Expire the session. All the method calls through the expired session should be ignored.

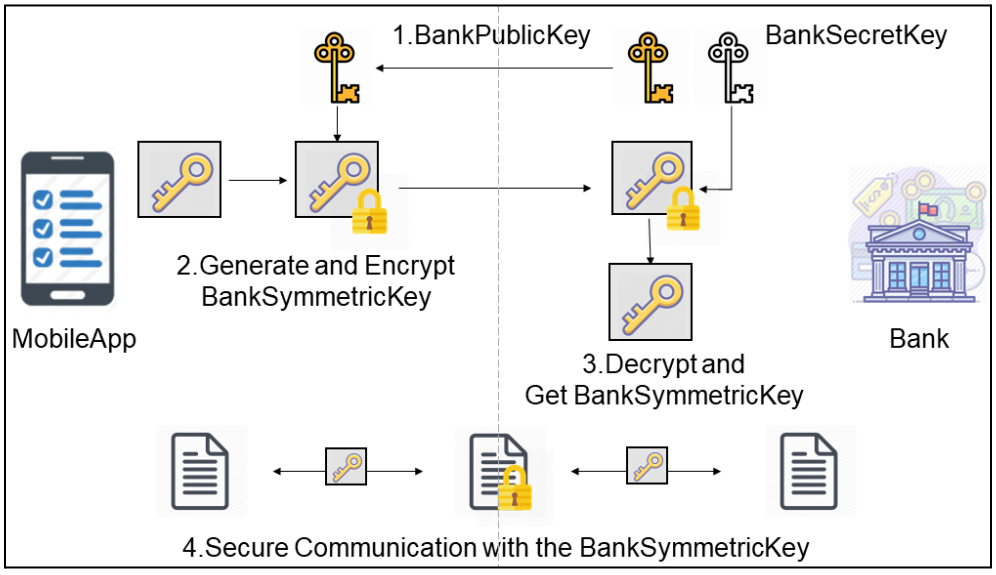
Question 1-3: Secure Mobile Banking [2 Marks]

Objectives:

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

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 provided 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 accountID: The customer’s account ID for the transaction.

- 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 randomUniqueStringGen method.

- public MobileApp(String id, String password, int AccountID)
  - 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, accountID and amount.
  - Then, you need to encrypt the Message object with the BankSymmetricKey object(generated by the sendSymKey), and return the Encrypted<Message>.

do i need to use sendSymKey() here again or is it alr saved in bankSymmetricKey

- 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, accountID 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, and the case where the BankSymmetricKey is null is not considered in the test). If decryption fails, return false, otherwise return the value of the decrypted output.

you mean a boolean? Which would obv be 1 if it didn’t return false?

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 "withdraw" requests.
  - Fetch the boolean result of the invoked method, encrypt it with the BankSymmetricKey object and return it.

can i make a new private var?

does my code do this

huh

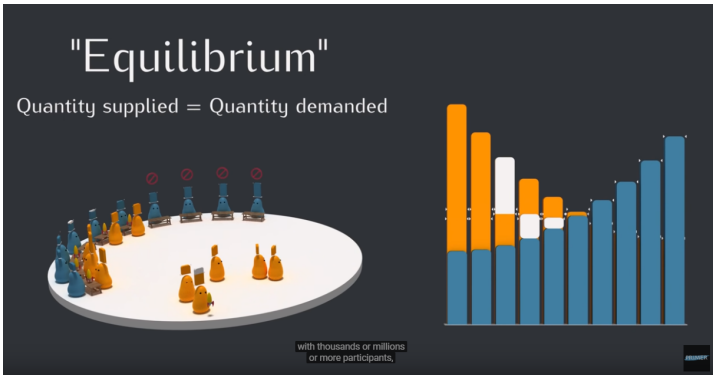
Question 2: Invisible Hand [3 Marks]

Objective: Develop a simulator to 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 the market. 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 once purchases an item in round 3, he/she cannot buy anymore from round 4.

Note:

- 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`.
- 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 two methods (i.e., willTransact and reflect) in Buyer and Seller classes.

Description: Just like our daily lives, each buyer and seller has its own price limit and expected price of a transaction. **Agent class is already provided** and it is the parent class of Buyer and Seller classes. It has five basic attributes. **DO NOT** modify them, but you can add if you need.

- protected double priceLimit
  - Seller has its own value which is the lower bound of the price that the seller wants to sell.
  - Buyer has its own value which is the upper bound of the price that the buyer wants to pay.
- protected double expectedPrice
  - Even if the buyer's priceLimit (=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.
- protected double adjustment
  - Amount that changes the expectedPrice. If there was no transaction on that day, Buyer and Seller change its expectedPrice.
- protected double adjustmentLimit
  - Limitation of the amount of adjustment.
- protected boolean hadTransaction
  - True when the transition is made on that day. Else, false.

Here's a more detailed description of how priceLimit and expectedPrice are used. In each round, each buyer and seller make decisions only based on their own expectedPrice. However, at the end of each day, they have a time of reflection. In reflection, they adjust the expectedPrice depending on whether they were able to make a deal on that day. In particular, the expectedPrice may be adjusted but will never go beyond the priceLimit.

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

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

Buyer Class Specifications

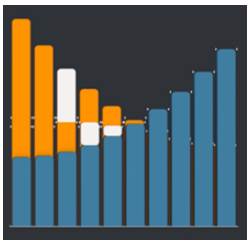
- public boolean willTransact(double price)
  - Return true if and only if (1) Buyer didn't make a transaction that day and (2) the price (the given parameter value) is equal to or less than its expectedPrice. Else, return false. Do not care about floating point errors.
- public void reflect()

    - If Buyer made a transaction that day, decrement expectedPrice by the value of adjustment.
        - After decrement expectedPrice, increase adjustment by 5. However, if adjustment exceeds the adjustmentLimit, set adjustment equal to adjustmentLimit
    - If Buyer didn't make a transaction, increment expectedPrice by the value of adjustment. However, if adjusted expectedPrice exceeds priceLimit, set expectedPrice equal to priceLimit.
        - If expectedPrice is increased by adjustment (not set to the priceLimit), decrement adjustment by 5.
        - If the decreased adjustment is less than 0, set the adjustment to 0.
    - Finally, in all cases, reset hadTransaction member variable to false for the next day.

Seller Class Specifications

- public boolean willTransact(double price)
    - Return true if and only if (1) Seller didn't make a transaction that day and (2) the price (the given parameter value) is greater than or equal to its expectedPrice. Else, return false. Do not care about floating point errors.
- public void reflect()
    - Very similar to the method described above. Just reverse the direction of change of expectedPrice 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 2,000 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 by using the given matchedPairs(int day, int round) method. Implement the following simulate method in Market class.

Market Class Specification

- public double simulate()
    - Repeat the following for 2000 times (=2,000 days)
        - Repeat the following for 10 times (=10 rounds)
            - Get a list of matched <Seller, Buyer> pairs using matchedPairs(int day, int round) method.
            - For each matched pair, the Seller will suggest its expectedPrice to the Buyer. If the Buyer is satisfied by the price, call the makeTransaction method of both Seller and Buyer objects.
        - Call the reflect method of every Seller and Buyer object. This method should be called even on the last day.

- Just return any double value for now. Question 2-3 will specify what to return.

- private List<Pair<Seller, Buyer>> matchedPairs(int day, int round)
  - As mentioned above, this method is already provided and **DO NOT** modify it. It returns a list of <Seller, Buyer> pairs, for the specified day and round.
  - In fact, the intention of this method is to randomly match buyers and sellers. However, unlike real randomness, this method always returns the same result for a given day and round.

Question 2-3: Equilibrium [1 Marks]

Objective: Economists found that the average price of an item converges to the value corresponding to the intersection of two curves, each made from the priceLimits 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 three methods.

Market Class Specification

- public double simulate()
  - Now, this method returns the average of the prices of all exchanges made on the last(=2000th) day
  - If there were three seller-buyer pairs who made a transaction on the last day, each at the price of 300, 400, 500, then this method should return 400.
- private List<Buyer> createBuyers(int n, List<Double> f)
  - Create and return a list of n Buyers constructed with priceLimits determined by a polynomial $f(x)$. Each Buyer with an index i should have a priceLimit of $f(i/n)$ ($i = 1, 2, ..., n$, note that the index i starts from 1, not 0).
    - E.g., when n=3, there should be 3 buyers with a price limit of $f(0.333...)$, $f(0.666...)$, and $f(1)$.
  - $f = [a_0, a_1, ..., a_n]$ represents $f(x) = a_n x^n + ... + a_1 x + a_0$. It is guaranteed that the length of f is at least 1.
    - Don't worry about underflow or overflow. Test cases ignore these cases.
    - You need to use methods in java.lang.Math to calculate $f(x)$.
      (This class doesn't need to be imported since java.lang package is imported by default)
- private List<Seller> createSellers(int n, List<Double> f)
  - It is the same as above except for changing the Buyer to Seller.