Assignment #2 [8 Marks]

Due Date	21 April 2022 23:59
Course	**[M1522.000600] Computer Programming**
Instructor	Jae W. Lee

- You are allowed to discuss with fellow students to understand the questions better, but your code must be **SOLELY YOURS**. You **MUST NOT** show your code to anyone else, or vice versa.
- 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 like github), we reserve the right to request an explanation about the code. We will ask detailed Problems that cannot be answered if the code is not written by yourself.
- If one of the above cases happens, 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 newetl. The 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.
- **Do NOT use ANY external libraries or methods** (even java.util.* or collection framework) which are not already imported. All “import” you need is already written in the skeleton code. But you can use any method which doesn’t need additional import(e.g. java.lang.* such as java.lang.Object class) such as Math.random() or String.split() or toString(). But check whether the method is supported by the JAVA version we use (11.0.14) before using it. If the method does not work on 11.0.14 version, you could get 0 score. Please don’t ask about the allowed methods or libraries, just do based on this paragraph.

Contents

Notes

- Feel free to add new classes if necessary.
- Feel free to add member attributes (don't modify the access modifiers of given attributes) or modify implementations of methods of the given classes in the skeleton code unless we instruct otherwise.
- However, **DO NOT** modify the package name, the signature of the given methods (i.e., return type, method name, and parameter types) or additionally extend class or implement interface to classes. **The exact given package name and signature of methods will be used for the final evaluation.** We don't have responsibility for disadvantages from modifying skeleton code.
- If you have any questions about whether you should take care of a specific corner case, contact TA through Slack.
- Test cases are introduced as Test.java. Please test your code with this testbench before submission. Of course, we will replace testbench with new Test.java with more complicated cases when we score your codes.
- Description document is pretty long, but don't be afraid! It is long due to the kind explanation of the code step by step. Keep calm and read carefully, then it will not be that difficult.

Submission Guidelines

1. For submission, compress the entire "problem1" and "problem2" directory in a single zip file.

2. The directory structure of your submitted file (after unzipping) should look like the following. **When you extract the zip file, there must be the 20xx-xxxxx/ directory.**
3. Name the compressed file “20XX-XXXXX.zip” (your student ID).
4. Submit your code on the neweTL.
5. Please submit only .java files. Delete files or folders like .iml / MACOSX/ DS_store / out / .idea before submit.
6. Double-check if your final zip file is properly submitted.
7. Note that **you will get 0 marks for the wrong submission format**.

Submission Directory Structure (Directories or Files can be added)

- Inside 20xx-xxxxx/ directory, there should be problem1/ and problem2/ directory.
- Refer to the screenshot below.

Directory Structure of Problem 1	Directory Structure of Problem 2
problem1/ └── src/ ├── security/... └── bank/ ├── event/... ├── MobileApp.java ├── Bank.java └── BankAccount.java └── Test.java	problem2/ └── src/ ├── map/ └── player/ ├── animal/ ├── Rabbit.java └── Turtle.java └── human/ ├── Runner.java └── Swimmer.java └── simulator/ ├── Map.java └── Simulator.java └── Test.java

Question 1: Secure Banking [4 Marks]

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

Descriptions: “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 a `MobileApp` class and emulate secure transactions between `MobileApp` and `Bank` classes. Finally, you will implement a compensation transaction for the hacked account. 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.

Question 1.1: Bank Account [1 Mark]

Objective:

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

Descriptions: 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 Descriptions

- 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 class and four subclasses of the Event class. 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. You don't have to consider (100+α) events per BankAccount case.

- 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 exactly equal.
    - "password" "password " are different due to space : return false
- void deposit(int amount, int transId)
  - 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, int transId)
  - 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.
  - There will be no corner case with the amount value less than 0.
- void receive(int amount, int transId)
  - 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, int transId)
  - 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.

    - There will be no corner case with the amount value less than 0.

`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)`
    - 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.
    - If the given id already exists in the bank, do not create the account.(ignore the request)
    - The negative initBalance is not considered for the final evaluation.
- `public boolean deposit(String id, String password, int amount, int transId)`
    - Authenticate the client with the `id` and `password`.
    - If the authentication is not successful or there is no account with the given id, 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, int transId)`
    - Authenticate the client with the `id` and `password`.
    - If the authentication is not successful or there is no account with the given id, 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, int transId)`
    - Authenticate the source account with the `sourceId` and `password`.
    - If there is no account with the given `sourceID` or `targetId`, do nothing, and return false.
    - 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
    - Use the `send` and `receive` method of the `BankAccount` class.
    - Since transfer is one transaction request, transIds of send and receive method in transfer method are same.

- public Event[] getEvents(String id, String password)
    - Authenticate the client with the id and the password.
    - Return null if the authentication fails or there is no account with the given id.
    - Return the array of Events that were **recorded** upon the deposit, withdraw, and transfer method calls.
        - The returned array should not contain null.
        - Return null if there has been no event since creation of the account.
        - More recent Events must be located **after** 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.

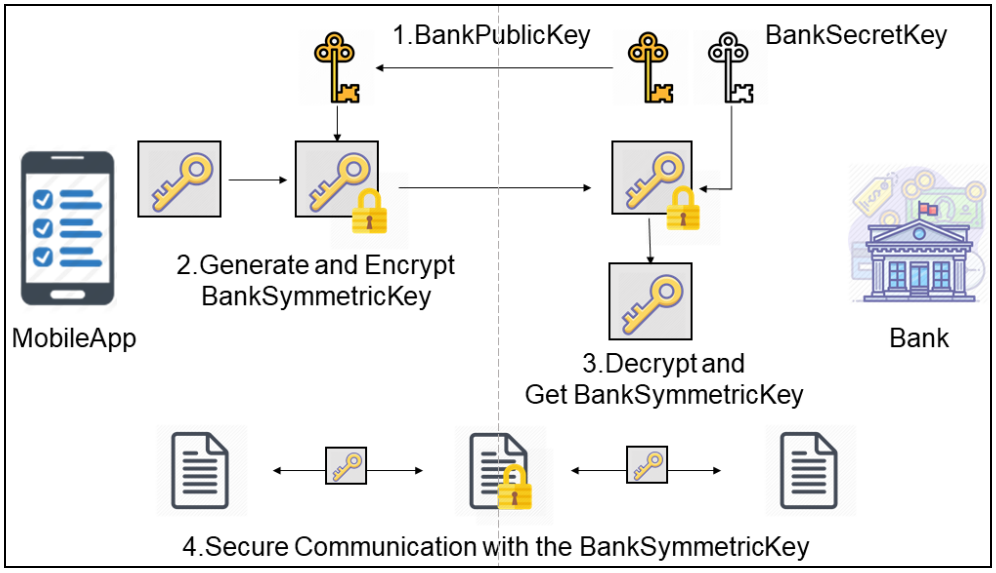
Question 1.2: Secure Mobile Banking [1 Mark]

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: For 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.
- public static boolean communicate(Compensate compensate, MobileApp mobileApp, Bank bank, String questionAnswer, int[] transIdList)
    - This method enables the secure compensation transaction through secure communication. (Problem 1.3)
    - The first argument is used to identify the type of the transaction.

    - In this method, mobileApp.requestCompensate, bank.processRequest, and mobileApp.processResponse are invoked in sequence. It is your job to implement three methods to make the secure compensation 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 provided 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” or concatenated question/answer string.
- String id, password: The authentication information of the customer.
- int amount: The argument for the deposit and withdraw calls.
- String qna : Question and answer for secondary authentication.
- int transId: Identifier for each transaction. This is a positive integer number and assigned to each transaction from one.
- int[] transIdList: Requested transaction Id list for compensation in Problem 1.3.

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. MobileApp server manages every transaction and allocates transId for each transaction. We emulate this briefly by the static attribute curTransId. curTransId indicates current transaction Id and you increase this variable by one when you create a deposit or withdraw Message object. curTransId is always a positive number and allocated consecutively from one.

- 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 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”(case sensitive), id, password and amount and transId. TransId is the identifier code for each transaction and the MobileApp class manages it. Note that curTransId is a static member. It increases by one after the first transaction message is constructed. TransId should be a positive integer number.
    - 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”(case sensitive), id, password and amount and trandId. TransId is the identifier code for each transaction and the MobileApp class manages it. Note that curTransId is a static

member. It increases by one after the first transaction message is constructed. TransId should be a positive integer number.
    - 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 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. You don't have to consider processing more than 10,000 handshakes such as exception handling for the 10,001st handshake.
        - 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", "withdraw" and “compensate”(Q 1.3)(all case sensitive) requests.
    - Fetch the boolean result of the invoked method, encrypt it with the BankSymmetricKey object and return it.

Question 1.3: Hacking Detection [2 Marks]

Objectives: Implement four methods (createAccount, compensate, secondaryAuthenticate, requestCompensate)

Descriptions: Someone found that his/her account had been hacked, and part of his/her balance was withdrawn. He/She calls the bank for compensation. So with the transaction IDs he/she told, the bank compensates for the unintended withdrawal to his/her account. But of course, the bank checks whether he/she is the right owner of the account by a question that he/she optionally answered when he/she made his account.

Bank Class Specifications

- public void createAccount(String id, String password, int initBalance, String question, String answer)
    - This overloaded method creates a secure bankAccount with input parameters.
    - For simplicity, only two questions are allowed for authentication. Answer consists of alphabets only.
        - Q1. “BestProfessor” A. “JWLee”, etc.
        - Q2. “BestTA” A. “KSS”, “JWS”, etc.
        - Note there is no space in question or answer string. Also those are case sensitive.(e.g. Wrong question or answer: “BestTa”, “BestProfessor“, “wonsukjang”)
    - If an account with a given id is already created and succeeds authentication with

input password, ignore initBalance and just add or update the question and answer attribute of the account.
    - Do nothing for illegal question or answer.
    - Do nothing for other cases. (Just finish this method)
- public boolean compensate (String id, String password, String questionAnswer, int[] transIdList)
    - If secondary authentication with questionAnswer is successful, check transaction log from Event class and sum robbed balance by matching transID array customer sent. Then return true.
        - If the transId in transIdList does not match the account's event or event is not a WithdrawEvent, just ignore that transId.
    - If authentication is not successful, return false.

BankAccount Class Specifications

- boolean secondaryAuthenticate(String questionAnswer)
    - Slice questionAnswer string into two(question, answer) pieces. They are separated by "," and there is no space in the string.
    - If the question and answer provided by the customer match those stored in the account, return true. Else, return false.
    - If the account has never been set to the question and answer, return false.

MobileApp Class Specifications

- public Encrypted<Message> requestCompensate (String question, String answer, int[] transIdList)
    - Assume customers can check the transID of their transaction through mobileApp.
    - Send an encrypted transID array to the bank.
    - This method constructs an encrypted message to compensate for the money. The encrypted message is used by the processRequest method of the Bank class.
    - You should create a Message object with String "compensate"(case sensitive) , id, password, the concatenated string of question and answer with ","("question,answer") and transIdList which has transIds of transactions he/she wants to get compensation for.
    - Then, you need to encrypt the Message object with the BankSymmetricKey object (generated from the sendSymKey), and return the Encrypted<Message>.

Question 2: Relay race [4 Marks]

Objective: In this problem, we will simulate a simple relay race with humans and animals.

Description: Humans(runner, swimmer) and animals(rabbit, turtle) do the relay race. Each player class can be constructed at most two. There is one water region and the rest is a land region. Some players can't swim so they need to pass the baton to a swimmable player before the start point of the water region. If they arrive at the water region without passing the baton, they throw up(=give up) the race. Even if one team gives up the race, the other team keeps the race. Meanwhile, the simulator checks the race and notices players of their current situation. Humans boost their speed if distance between them and animals is less than specified value. If both teams throw up the game or one team reaches the goal, the game ends with a finish log.

Question 2-1: Map, Eyesight and Message [1 Mark]

Objectives: Implement `Map`, `Eyesight` and `Message` class.

Descriptions: Players move on the `Map` defined. Players can see which is in their `Eyesight`. `Messages` are announced by a simulator and humans can hear it.

`Map` Class Specifications

- public Map(double waterStart, double waterEnd, double mapEnd)
    - The Starting point of the map is zero. So the size of the map is equal to `mapEnd`. `mapEnd` is in the range of $0<mapEnd\leq100$. We will not test with 100+ of `mapEnd` so just ignore that case. There is only one water region on the map, from `waterStart` to `waterEnd`. Water region can start from zero, and can end at `mapEnd`. Size of the water region should be a non-zero positive value. You don't have to handle the zero-water region case.
- public boolean getOnWater(double position)
    - If the player is in the water region, return true. The start point of the water region should be included and the end point should be excluded, that means, the player is considered as the player is on the water region when the player's position is 20 for water region 20~30.(Water region : [20,30))

`Eyesight` Class Specifications

Each player has their own `Eyesight`. `Eyesight` checks the distance to the next player or the region boundary. `Eyesight`'s visible `range` is 3. `nextPlayerPosition` is the position of the next player who will be passed the baton by this `Eyesight`'s owner.

- public Eyesight(Map map)
    - Constructor of this class with the `Map` the race proceeds.
- public void setNextPlayerPosition(double position)
    - set `nextPlayerPosition` of `Eyesight` class.

- public double getDistanceToBoundary(double playerPosition)
    - playerPosition is the position of the player who has this Eyesight.
    - If the player is in the water region, return the distance to the water region endpoint.
    - If the player is on the land region preceding the water region, return the distance to the water region start point.
    - If the player is on the land region next to the water region, return the distance to the final line.
    - For all cases, if the calculated distance is larger than the range, return the range value, 3.
- public double getDistanceToNextPlayer(double playerPosition)
    - playerPosition is the position of the player who has this Eyesight.
    - If the distance to the next player is less than the range, return that distance. Else, return the range value, 3.

Message Class Specifications

Message is a deliverable object from simulator to humans. It has four basic member attributes.

- private Player currentHuman : Current human team player who runs or swims
- private Player currentAnimal : Current animal team player which runs or swims
- private Map map : The map that the race is in progress.
- private int time : Time step when this message is created.

Now implement the next few methods.

- public Message(Player human, Player animal, Map map, int time)
    - Constructor of this class.
- public double getDistance()
    - Calculate the distance between currentHuman and currentAnimal. Return value should be non-negative value.
- public String toString()
    - Overriding method of toString method. There are six cases of message, so implement **accurately** as follows. We will evaluate your submitted code by printed output texts. Please check the exact format including space, comma, parentheses, square brackets etc. **No partial score will be given for tiny mistakes.** String inside the parentheses in the format is variable. Bolden red colored **s** in the format indicates space(just one space(" "). Not including tab or several spaces like " "). Use each player(Runner, Swimmer, Rabbit, Turtle)'s toCustomString method(Note that toString method of Player class invokes this method) for *human/animal's description*.(Refer to the problem 2.2 description)
        - If both teams are in the middle of the race without both teams giving up. This case includes the case that one team gives up and one team is still running.
            - Format : "(time):**s**[RUNNING]**s**Human**s**team**s**:**s**(*currentHuman)'s description***s**is**s**at**s***(currentHuman)'s position***s**/**s**Animal**s**team**s**:**s***(currentAnimal)'s*

*descriptions*is**s**at**s***(currentAnimal)'s position*"
        - Example : "16: [RUNNING] Human team : 2nd human player, runner is at 58 / Animal team : 4th animal player, turtle is at 43"
    - If currentHuman and currentAnimal are at the starting position
        - Format : "(time):**s**[READY]**s**Human**s**team**s**:**s***(currentHuman)'s descriptions*/**s**Animal**s**team**s**:**s***(currentAnimal)'s descriptions*are**s**at**s**0"
        - Example : 0: [READY] Human team : 1st human player, swimmer / Animal team : 1st animal player, rabbit are at 0
    - If currentHuman reaches at the end point earlier than animal team
        - Format : "(time):**s**[FINISH]**s**Human**s**team**s**wins"
    - If currentAnimal reaches at the end point earlier than human team
        - Format : "(time):**s**[FINISH]**s**Animal**s**team**s**wins"
    - If currentHuman and currentAnimal reach the end point at the same time.
        - Format:"(time):**s**[FINISH]**s**Both**s**teams**s**reach**s**the**s**goal**s**at**s**the**s**same**s**time"
    - If both teams throw up the race
        - Format : "(time):**s**[FINISH]**s**Both**s**teams**s**throw**s**up**s**the**s**race"

Question 2-2: Various Players [1.5 Marks]

Objectives: Implement Player class with abstract classes, subclasses and interfaces.

Descriptions: There are Human and Animal players and you should implement 4 classes(Runner, Swimmer, Rabbit, Turtle). Each of them has methods of their way of moving(move(), swim()) and they have a Swimmable(Turtle, Swimmer) interface or Throwable(Rabbit, Runner) interface. Plus, they have their own speed and it could be changed as their environment and racing condition.

Read the skeleton code and complete implementation of the following 7 classes. Some methods are already implemented and don't modify them.

Interface Class Specifications

There are two interface classes. Swimmable is for Swimmer and Turtle, and Throwable is for Runner and Rabbit. These classes are already implemented.

- public interface Swimmable
    - Swimmable players can swim at the water region. It invokes a swim method. You should implement the swim method with proper details for Swimmable players.
- public interface Throwable
    - Non-swimmable players throw up the race when they cannot pass the baton to the swimmable player until they reach the starting point of the water region. You should implement the throwUp method with proper details for Throwable players.

Player Class Specifications

Player class is a parent class of Human and Animal class. Below are member attributes of this class. Note that the Player class has a Comparable interface.

- double position, velocity : player's position and velocity
- double nextPlayerPosition : Position of the next player. If this player is the last player, nextPlayerPostion is the map endpoint position.
- boolean currentPlayer : True when this player is moving with the baton. Zero value means this player is waiting for the baton or finished his race after passing over the baton.
- Eyesight eyesight : Player's Eyesight class object.
- int playerNum : playerNum indicates the player number in the team.(ex. 3rd player in the human team : playerNum is 3)
- String playerType : It indicates that this player is "Animal" or "Human". We don't consider any other type or typo case in tests like "human" or "Animan". Note this is final attribute.

Now implement the following few methods.

- public Player(double velocity, Map map, String playerType)
- public Player(double position, double velocity, double nextPlayerPosition, Map map, String playerType)
    - Constructors are already provided. If position value is not specified, position is set to zero for default.
- public setNextPlayerPosition(double position)
    - Set the next player's position who will be passed the baton by this player in Player class and Player's Eyesight
    - This provided method is called by the Simulator class before the start of the race because the Simulator manages the sequence of the players.
- abstract public void move()
    - Player can move for its velocity from the current position. Note that this is an abstract method.
- abstract public boolean getThrowUp()
    - If a player throws up the race, return 1. Else, return 0. Note that this is an abstract method.
- abstract public void hear(Message message)
    - Humans can hear the message and they boost their speed in a specific case. Note that this is an abstract method.
- public void passBaton(Player nextPlayer)
    - Set currentPlayer attribute to false and nextPlayer's currentPlayer attribute to true.
    - This method will be invoked in Simulator class.
- protected double getMovableDistance(double velocity)
    - Return movable distance at this time step. Basically, the player can move for velocity, but you should consider two cases of limitation.
        - Movable distance can be less than velocity if distance to next player is shorter than value of velocity.
        - Movable distance can be less than velocity if distance to the next

region(land to water, water to land, land to finish line) is shorter than the value of velocity.
    - To check these limitations, use methods of Eyesight class.

Human Class Specifications

- Human class is the parent class of Runner and Swimmer class. This abstract class is already provided.

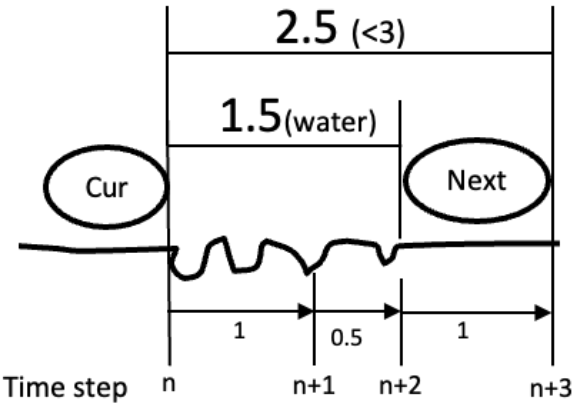
Animal Class Specifications

- Animal class is the parent class of Rabbit and Turtle class. Animals can't understand the words, so hear method is empty. This abstract class is already provided.

Runner Class Specifications

- public Runner(Map map)
- public Runner(double position, Map map)
    - Set Runner's position and velocity and the map Runner runs on. Runner's default velocity is 2. If position is not specified, set position value to zero.
- public void hear(Message message)
    - If the distance to the current animal player is less than 2 (i.e.,|*Runner's position* - *Animal's position*|<2) and the Runner can see(e.g., in his/her eyesight range) the end line(mapEnd) or next player, he/she boosts his/her velocity to 2.5 until he/she hears the next message.
- public void move(Map map)
    - Runner moves to the goal with velocity per time step. If the Runner cannot move further because of water, he/she stops at the boundary of water and throws up the game. Then, wait for the animal team until they reach the goal or give up the race. But there is one exception. If the next player or end line(mapEnd) is in the sight range of Runner, Runner doesn't give up and is carried along by the wave to the next player or end line(mapEnd) with very low speed, 1. Also, if distance to the boundary of water/land region is less than velocity, Runner just moves to the boundary at that time step. **Note** : Runner can't swim (just moved by the wave), so you don't need to implement a swim method, just set the velocity to 1 for this movement. Also, since Runner is not swimming (just floating) in the water, so Runner can hear the Message on the water region.

        - Example for your understanding



            - Runner can see the next player, so drifts away with velocity 1.
            - Runner moves just for 0.5(<1) because he/she reaches the boundary.
            - Runner's velocity is reverted to 2(or 2.5) on the land region, but moves just for 1 because he/she reaches the next player.
    - If distance to the next player is less than velocity, the Runner moves to the next player's position. Note : You already implemented a related method in Player class.
- public String toCustomString()
    - Follow this format and return. Be cautious of small mistakes like st,nd,th.
        - Format : "Athshumansplayer,srunner"
        - Example : "1st human player, runner" (O) "1st human player, runner"(X)
- public boolean throwUp(double position, Map map)
    - Runner throws up the game when he/she reaches the start point of the water region. But as mentioned above, if Runner can see the next player or end line(mapEnd), he/she will not throw up and try to overcome the water barrier.
- public boolean getThrowUp()
    - Return the result of the throwUp method.

Swimmer Class Specifications

Swimmer can't hear while swimming. Swimmer can run and swim but speed in those cases are different.

- public Swimmer(Map map)
- public Swimmer(double position, Map map)
    - Set Swimmer's position and velocity and the map he/she runs or swims on.

Swimmer's default running velocity is 1.5 and default swimming speed is 2. If position is not specified, set position value to zero.

- public void hear(Message message)
    - If the Swimmer is on the land region and the distance to the current animal player is less than 2 (i.e.,|*Runner's position - Animal's position*|<2) and if he/she can see the end line(mapEnd) or next player, he/she boosts his/her running speed to 2 until he/she hears the next message.
- public void move(Map map)
    - Swimmer moves to the goal with velocity per time step on the land and swims for swimSpeed on the water. But if Swimmer starts a race on the land region, Swimmer needs one time step to warm up at the water region start point before entering the water region. Swimmer doesn't give up the race in any situation. If distance to the next player is less than velocity for land region and swimSpeed for water region, the Swimmer moves to the next player's position. Also, if distance to the boundary of water/land region is less than swimSpeed/velocity, Swimmer just moves to the boundary at that time step. Implement and use the swim method.
    - Of course you could implement this method which makes a desirable result of the whole operation without the swim method, but for practicing the interface concept, implement the swim method as follows and use it here. There are test cases for each method(e.g., only test Swimmer.swim()) so we strongly recommend to follow each method description.
- public void swim()
    - If Swimmer needs to warm up, he/she doesn't move and stays in his/her position.
    - If distance to the next player is less than swimSpeed, the Swimmer moves to the next player's position.
- public boolean getThrowUp()
    - Just return false. Swimmer never gives up.
- public String toCustomString()
    - Follow this format and return.
        - Format : "Ath**s**human**s**player,**s**swimmer"
        - Example : "2nd human player, swimmer"(O) "2th human player, swimmer"(X)

Rabbit Class Specifications

- public Rabbit(Map map)
- public Rabbit(double position, Map map)

    - Set Rabbit's position and velocity and map it runs on. Rabbit's default velocity is 3. If position is not specified, set position value to zero.
- public void move(Map map)
    - Rabbit moves to the goal with velocity per time step. If the Rabbit cannot move further because of water, it stops at the boundary of water and throws up the game. Then, it waits for the human team until they reach the goal or give up the race. If distance to the next player is less than velocity, the Rabbit moves to the next player's position. Note : You already implemented a related method in Player class.
- public String toCustomString()
    - Follow this format and return.
        - Format : "Ath**s**animal**s**player,**s**rabbit"
        - Example : "1st animal player, rabbit"
- public boolean throwUp(double position, Map map)
    - If the Rabbit is on the point that it cannot move ahead due to the water, return true. Else, return false.
- public boolean getThrowUp()
    - Return the result of the throwUp method.

Turtle Class Specifications

- public Turtle(Map map)
- public Turtle(double position, Map map)
    - Set Turtle's position and velocity and the map it runs or swims on. Turtle's default running velocity is 1 and default swimming speed is 2.5. If position is not specified, set position value to zero.
- public void move(Map map)
    - Turtle moves to the goal with velocity per time step on the land and swims for swimSpeed on the water. It doesn't give up the race in any situation. If distance to the next player is less than velocity for land region and swimSpeed for water region, the turtle moves to the next player's position. Use the swim method.
    - Of course you could implement this method which makes a desirable result of the whole operation without the swim method, but for practicing the interface concept, implement the swim method as follows and use it here.
- public void swim()
    - If distance to the next player is less than swimSpeed, the Turtle moves to the next player's position.
- public boolean getThrowUp()

    - Just return false. Turtle never gives up.
- public String toCustomString()
    - Follow this format and return.
        - Format : “Ath**s**animal**s**player,**s**turtle”
        - Example : “3rd animal player, turtle”

Question 2-3: Simulator [1.5 Marks]

Objectives : Implement Simulator class.

Description : Simulator lets the race proceed and pass the baton. Also, the Simulator snapshots the current racing situation for every time step and announces it to all. Evaluation of this sub problem includes checking various conditions of race simulation output. So if your code of Problem 2.1 and 2.2 is wrong, you could not get a full score for Problem 2.3 even though you implement a perfect Simulator class.

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). Again, do not use this class in any other class except for Simulator.

Simulator Class Specifications

Note that maxTeamPlayerNum is 4. Each kind of player can be constructed at most 2. Simulator has two ArrayList for managing each team’s players. currentHuman is moving human player who has baton and currentAnimal is moving animal player which has baton. raceLogForEval stores the race log and it will be used to score this problem.

- public Simulator(ArrayList<Player> humanPlayers, ArrayList<Player> animalPlayers, Map map)
    - Update map, currentHuman, currentAnimal attributes of this class.
    - Each ArrayList of human and animal are constructed outside of the Simulator class. But the Simulator sorts those arrays in ascending order of starting position of human/animalPlayers. Then the Simulator allocates Human/AnimalPlayerNum and NextPlayerPosition for each player. You don’t have to consider the case that several team players have equal starting positions. We will test with unique starting positioned players in one team. Of course different team players can have the same starting position.
    - Also, we will not consider the case that the start position of any player is equal to the map end position. (No player on the map end point at time 0)

    - Note that Human and Animal classes implement a Comparable interface.
- public void snapshot()
    - Update currentHuman and currentAnimal attributes if changed.
    - If one player reaches the next player's position, invoke passBaton method of Player class.
- public Message talk(int time)
    - Check the racing situation and create a Message to announce it to the players.
    - Simulator also notices race start and race finish.
    - This method is invoked for every time step from zero to race finish.
    - This method is already implemented.
- public boolean getRaceFinish()
    - Return raceFinish flag to notice the testbench that race ends.
- public void simulate()
    - From time step 0, the Simulator makes the currentAnimal and currentHuman move, and update currentHuman and currentAnimal attributes if needed for every time step until one of the teams finishes the race or both teams throw up the race. Messages should be created by talk method from time step 0 to race finish time(0 and finish time included), which means [READY] and [FINISH] Messages should be created once. Human players' hear method is invoked for every time step from 0 except for the last message which notices race finish. Time step increases by one after both teams' players finish their move or stay(throw up case) for that time step. Set the raceFinish flag to 1 when the race finishes.
    - If the number of players of team exceeds maxTeamPlayerNum or Animal/Human player is mis-allocated in Human/Animal team or no Player of any team exists at position 0 at time 0, then don't go ahead the simulation(just finish the simulate method) and just store "[ERROR]**s**Team**s**building**s**error" to raceLogForEval[0] and set the raceFinish flag to 1.