Assignment #4 [10 Marks]

Due Date	22 May 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 normalized 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 questions that cannot be answered if the code is not written by yourself.
- If any of the above cases happens, you will **get 0 marks** for the assignment and may get **a further penalty.**
- Download and unzip “HW4.zip” file from the NeweTL. “HW4.zip” file contains skeleton codes for Question 1 and 2 (in “problem1” and “problem2” directory, respectively).
- 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.
- When you submit, compress the “HW4” directory which contains “problem1” and “problem2” directories in a single zip file named “**{studentID}.zip**” (e.g. 2022-12345.zip) and upload it to NeweTL. 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 use external libraries.
- Built-in packages of JDK (e.g., java.util, java.io, java.nio) are allowed.
- Java Collections Framework is allowed.
- Note: we provide the basic test cases, and we will use a richer set of test cases for evaluation. We strongly encourage you to add more diverse test cases to make sure your application works as expected.

Contents

Submission Guidelines

1. You should submit your code on NeweTL.
2. Compress the “HW4” directory and name the file “{studentID}.zip”
3. After you extract the zip file, you must have a “HW4” directory. The submission directory structure should be as shown in the table below.
4. You can create additional directories or files in each “src” directory.
5. You can add additional methods or classes, but do not remove or change signatures of existing methods.

Submission Directory Structure (Directories or Files can be added)

- Inside the “HW4” directory, there should be “problem1” and “problem2” directory.

Directory Structure of Question 1	Directory Structure of Question 2
problem1/ └── data/... └── src/ └── attendancechecker/ └── Grader.java └── attendance/ ├── AttendanceList.java ├── Lecture.java └── Student.java └── utils/ └── Pair.java └── Test.java	problem2/ └── data/... └── src/ └── match/ ├── Betting.java └── Match.java └── user/ └── User.java └── server/ └── Server.java └── match/ ├── Pair.java ├── Config.java └── ErrorCode.java └── Test.java └── solution/…

Question 1: Attendance Checker [4 Marks]

Objective: Develop an auto-attendance checker for Computer Programming class.

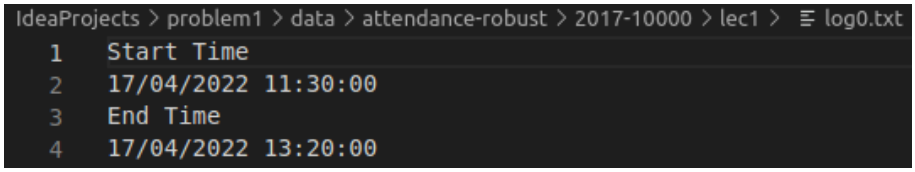
Description: You are a TA of a Computer Programming class. You don't have any program that automatically provides you the attendance scores of the students. You'll now implement an auto-attendance checker which returns the attendance score based on the students' Zoom log files.

You are required to write two methods in the **Grader** class: `gradeSimple()` for Question 1-1, `gradeRobust()` for Question 1-2. Both methods perform attendance checking; `gradeRobust()` can handle some corner cases while `gradeSimple()` does not. We separate two methods to prevent you from corrupting codes for each question. Do NOT change the skeleton codes except for Grader.java.

Question 1-1: Simple Checker [1.5 Marks]

Objective: Implement the `gradeSimple()` method in the Grader class.

Description: Your goal is to implement an attendance checker that checks students' zoom log files. A log file consists of "start time" and "end time" like in the picture below.



See below for the description of parameters and a return value of the `gradeSimple()` method.

- Parameters
  - `AttendanceList attendanceList`: AttendanceList object that contains the list of students
  - `String attendanceDirPath`: Path to the directory with zoom log data
- Return value
  - Type: Map<String,Map<String, Double>>
  - It returns the attendance score of each student. The Map maps student ID to another Map, which maps lecture name to attendance score.

Classes	Directory Structures
class AttendanceList ● List<Lecture> lectures ● List<Student> students	**@ data.problem.studentId.lecture** ● log0.txt

In Question 1-1, make the following assumptions for the simplicity of the problem:

- If there are multiple start and end times, simply accumulate the connection time of each start and end time pair.
- There's no student without a log file.
- There's only one log file.

For each log file, you should return whether the student was "absent", "late" or "attended". Students are considered "absent" (0 mark) if their connection time is less than 10% of the lecture time. Students are considered "late" if their connection time is less than 70% (0.5 mark) of the lecture time. Otherwise, students are considered "attended" (1.0 mark). Constructor of the lecture class uses the start and end time of a lecture for object creation.

Question 1-2: Robust Checker [2.5 Marks]

Objective: Implement the `GradeRobust()` method in the Grader class.

Description: There are some scenarios we should handle. Some students can experience lost internet connection. Some students might want to attend Zoom class with two or more devices (e.g. mobile for video and mic, desktop for watching). It's not possible to just ignore one log or accumulate connection times considering some corner cases. Here are some cases we should handle. You don't need to worry about other corner cases.

Case 1. Early Entrance

Students can enter the zoom before the lecture actually starts. You shouldn't count the time before the lecture's start time.

Case 2. Multiple Connections

Many students enter the zoom class with multiple devices. Fortunately, each connection will be recorded in different log files. You should calculate the "net connection time" of the students. For example, when a student transfers from mobile to desktop, two connection times can overlap. Therefore, you shouldn't simply add the two connection times.

IdeaProjects > New1 > data > attendance-robust > 2022-30002 > lec1 > log0.txt

```
1  Start Time
2  17/04/2022 12:50:00
3  End Time
4  17/04/2022 14:00:00
```

IdeaProjects > New1 > data > attendance-robust > 2022-30002 > lec1 > log1.txt

```
1  Start Time
2  17/04/2022 12:50:00
3  End Time
4  17/04/2022 14:00:00
```

Case 3. Excused Absence

Suppose excused absence generates a log file like in the picture below. If the log contains "Admitted", the student is considered "attended". Otherwise, the student is considered "absent".

log0.txt

IdeaProjects > cp2022 > hws > hw4 > problem1 > data > attendance-robust > 2022-12345 > lec1 > log0.txt

```
1  Refused
2
```

log0.txt

IdeaProjects > cp2022 > hws > hw4 > problem1 > data > attendance-robust > 2020-20001 > lec2 > log0.txt

```
1  Admitted
```

Question 2: Sports Betting Systems [6 Marks]

Objective: Develop a sports betting system.

Description: Many countries have legal sports betting systems to make profit and boost the popularity of the sports. There are also many portal and community sites that mimic these betting systems for fun. The fundamental idea of the sports betting system can be briefly summarized as below:

1. For every sports match, multiple sports bettings can be held depending on the result of the match. People can bet on “which team wins the match”, or “how many goals the team A can score”, etc. We’ll simplify them as betting on option1, option2, etc.
2. People can bet on multiple options of a single match. They can also bet additional coins if they want. A bet is valid only if it was done before the start of a match. All the late bids are refunded.
3. The system assigns **betting Ids** for all accepted bettings. Each betting option of each match has its own bettingIdMap. If a user is the **nth** person to bet on an option of a match, the user’s betting is assigned **n** as **betting Id**. The number of users who bet on other options of the same match doesn’t affect the id.
4. **Betting odds** are determined for each option as the ratio of total coin over coin bet on an option. For example, if one user bets 500 coins on option1 and another user bets 1000 coins on option2, odds are 3.00 (= (500 + 1000) / 500) for option1 and 1.50 (= (500 + 1000) / 1000) for option2. If no user bet on a certain option, assign 0 as its current odd.
5. During the **settlement** of a bet, a little commission(4%) is charged and the rest are distributed to the winners. In the example above, if option1 is selected as winner, the user who bet on option1 will receive 1440 (=0.96 * 1500 / 500 * 500) coins as a reward.
6. You’ll implement 1) match search system 2) betting and its management system 3) settlement system sequentially.

Note:

1. In this question, you will implement the methods in Server.java and User.java. **Do NOT change the signatures of the given methods**, but you can add/modify other parts. You can modify files or make new files inside any directories under “src” directory, unless mentioned otherwise.

Source Directory Structure

```
src/
└── user/
    └── User.java                // Implement this
└── match/
    ├── Betting.java
    └── Match.java
└── utils/
```

```
    ├── ErrorCode.java
    └── Pair.java
    └── Config.java
└── server/
    └── Server.java          // Implement this
└── Test.java                // Test cases are in this file.
```

2. There are five java classes (**Betting, Match, ErrorCodes, Pair, Config**) under the **match** and **utils** package. The specific use of each class will be explained in the sub-questions below. These classes determine the format of the output, and thus, you **shouldn't modify these five classes**.
3. There are example test cases in the **Test** class. You can run them from the main() methods. It provides you with the basic test cases for each sub-question. We encourage you to add more test cases to check the correctness of your code.

Question 2-1: Search Match Information [2 Marks]

Objective: Implement the List<Match> search(Map<String, Object> searchConditions, String sortCriteria) method in the Server class.

Description: The method returns the list of matches matching the search conditions (given in the Map) following the sorting criteria (given as the String).

The first parameter Map<String, Object> searchConditions can consist of four different search conditions listed below.

key	value type	value description
“sports”	String	Search by type of the sports of the match(e.g. Football). It should be case sensitive
“time”	String	Return matches whose match times are later than given time (e.g. for 2022/04/26 12:00, matches that start at 2022/04/26 14:45 should be returned)
“club”	String	Search matches with the name of a club. There are a home team and an away team for each match (e.g. LG Twins vs Dusan Bears). Return matches one of whose teams satisfy the search condition as described below.
“odds”	Double	Return matches whose highest betting odds are larger than the given value. Refer to the descriptions at the beginning of the question2 for the definition of betting odds.

- For the “**club**” criterion, it should be *case sensitive*. Also, find all the matches whose home or away team includes given “words separated with 1 empty space” in its name. For example, if you pass “Seoul” as a value, the result should contain all the matches related to “FC Seoul”. If you pass “Seoul FC”, all the matches one of whose team has both “Seoul” and “FC” in its name should be returned. Therefore, matches related to “FC Seoul” are returned again. However, if you pass “Seo”, it shouldn’t print the matches related to “FC Seoul”.
- **Multiple search conditions** can co-exist. Ignore keys other than “sports”, “time”, “odds” and “club”.

The second parameter String sortCriteria describes the sorting criteria of the returned List<Match>. It can have one of the following four values.

sortCriteria	Description
“sports”	Sort by the sports type in the dictionary order. If the sports types are equal, sort by the match Id in ascending order.

“club”	First, compare home team names and then compare away team names. If the names are equal, sort by the match ID in ascending order.
“time”	Sort by the betting deadline in the ascending order. If the betting deadlines are equal, sort by the match ID in ascending order.
“odds”	Sort by the highest betting odds in the ascending order. If the betting odds are equal, sort by the match ID in ascending order.

- Assume that match ID is unique for all the matches.
- If `sortCriteria` is null or empty String, sort by the match ID in ascending order.
- Assume that the `sortCriteria` string is one of {null, empty String, “club”, “sports”, “time”, “odds”}.
- Suppose the format of “sports” and “time” strings are always the same as the values given in the test cases. For example, you don’t need to handle the time written as “04/05/2022 17:23:30”, the sports written as “!Football?” or some other strange inputs.

Output type is `List<Match>`: The list of the `Match` instances satisfying the given search condition with the elements sorted by the given criteria.

The data directory structure and file format is as described below:

Data Directory Structure (Example)

```
data/
└── Matches/
    └── Football/
        └── 3/
            ├── 3_bettingBook.txt
            └── 3_info.txt
        └── 11/
            ├── 11_bettingBook.txt
            └── 11_info.txt
        └── …(match id)/
    └── Basketball/
        └── 5
            ├── 5_bettingBook.txt
            └── 5_info.txt
        └── 8/
            ├── 8_bettingBook.txt
            └── 8_info.txt
        └── …(match id)/
    └── …(sportsTypes)/
```

```
└── Users/
└── Users_backup/
      └── (Backup of the /data/Users)
└── Matches_backup/
      └── (Backup of the /data/Matches)
solution/
      └── (Data used for comparison at Test.java)
```

The content of each file is shown in the data contents table below. For a "(**match_id**)_info.txt" file, **the number of betting options**, **n**, means the number of options users can select. For example, if there are three options, "Team A wins", "draw", and "Team B wins", **n=3**. After n, **current betting odds** for each option are stored.

Total betting, means the number of bettings submitted. After one betting, additional bettings on the same option should be counted as 1. However, betting on another option should be counted independently. For example, even if a user bet on option1 of match1 for 7 times, total betting should increase by 1. Instead, if a user bet on option1 and option3, total betting should increase by 2. Odds are rounded to the second digit after the decimal point.

Data Contents		
	Format	Example
File Path	data/Matches/(Sports Types)/(match id)/(match id)_info.txt	data/Matches/Football/12/12_info.txt
File Content	(home team)|(away team)|(location)|(match time)|(**the number of betting options**)|(**current odds**[0])|(current odds[1])|(...)|(current odds[n-1])|(**total betting**)	FC Seoul|FC Anyang|Seoul Stadium|2022/04/26 15:22|3|2.63|3.87|2.77|300

For testing problem 2-1, Test.java will copy information text files from the solution directory. For problems 2-2 and 2-3, Test.java will use `collectBettings()` for building information and bettingBook text files.

Question 2-2: Betting System [3.5 Marks]

Objective: Implement the following four methods in the Server and User class:

1) int bet (int matchId, int bettingOption, int coin) in User class
2) int updateBettingId (int matchId, int bettingOption, int newBettingId) in User class
3) int collectBettings() in Server class
4) List<Betting> getBettingBook(int matchId) in Server class

Descriptions for User class methods: The bet() method allows a user to bet for a match. Users can bet on multiple options, multiple times. updateBettingId is used to update the content of a user's bettingIdMap to newBettingId.

bet() method returns the result of betting. Here are the return values that can arise:

Output of bet() Method	
ErrorCode	Description
SUCCESS	The betting is successful.
IO_ERROR	IOException is thrown during the execution of the method.
NOT_ENOUGH_COINS	User doesn't have enough coins to continue betting. There's COIN_PER_USER variable in the Config class.
OVER_MAX_BETTING	You can't bet too many coins on one match. The maximum coins to be bet per match is defined as a static variable MAX_COINS_PER_MATCH in the Config class.
NEGATIVE_BETTING	Input coin is **zero** or a negative integer.

- The ErrorCode class (in the skeleton code) defines constant integer values matching various types of errors. Use the ErrorCode class appropriately to return the error code. If no error occurs, place the bet and return SUCCESS. Otherwise, do not place the bet and return the proper error code. When multiple errors occur, return the error code with the lowest value.
- If there's no error, the bet method **writes the betting data to the "newBettings.txt" file** under the user's data directory. Check the format of "newBettings.txt" in the table below.
- Do not directly access "bettingBook.txt" files in the user class

Notes:

1. More specifically, the betting information of a user should be stored in a file (named "data/Users/(user id)/newBettings.txt"), as described in the tables below. You have to store the betting information in the "newBettings.txt" file for each bet method call.
2. Users can't withdraw or modify previous bettings.
3. When the bet method is called, you should update the user's bettingIdMap if needed.

bettingIdMap maps Pair <int matchId, int bettingOption> to int bettingId. Between the creation of a user's first betting on an option and the collection of that betting by server, -1 should be assigned as an Id. Once the betting is reflected to the betting book in the match directory, the ID in the bettingIdMap should be updated with updateBettingId() method. updateBettingId() method will be called at the collectBettings method. You can freely determine the return value of the updateBettingId() method.

4. We provide a resetDirs() method to clear all user and match directories and reload the default user information. You can freely use this method in Test.java to make appropriate test cases.
   The data directory structure and file format is as described below:

Data Directory Structure
data/ └── Matches/ └── Users/ └── 2010-22221 └── **newBettings.txt** └── 2012-22221 └── **newBettings.txt** └── ...(user id) └── Users_backup/ └── (Backup of the /data/Users) └── Matches_backup/ └── (Backup of the /data/Matches) solution/ └── (Data used for Test.java)

The content of newBettings.txt is as below:

Specification of Betting Data (newBetting.txt)		
	Format	Example
File Path	data/Users/(user id)/newBettings.txt	data/Users/2022-12345/newBettings.txt
File Content	(match id)|(betting option)|(coins bet) (match id)|(betting option)|(coins bet) (match id)|(betting option)|(coins bet) (match id)|(betting option)|(coins bet) ...	10|0|5000 10|1|1000 8|2|3000 1|1|2000 1|1|2000 3|0|3000bettingId

Descriptions for Server class methods: collectBettings() method is used for collecting the bettings under user folders, organizing the betting book for all the matches. It traverses the user directory and processes "newBettings.txt" files. Traverse order should be in the ascending order of userId (e.g. 2020-11111 earlier than 2022-00000).

Processed "newBettings.txt" files can be deleted. If a betting request is valid, corresponding "(match_id)_bettingBook.txt" will be updated and **betting ID** will be assigned to **bettingIdMap**. Otherwise, it's rejected and the user's bettingIdMap is updated with the **error code** and the coins are refunded. The return value of the collectBettings() method is ErrorCode.SUCCESS if there's no IO error. If an IO error occurs, return ErrorCode.IO_ERROR. However, if you want, you can freely assign the return value.

Betting ID is assigned to each betting in ascending order, starting from 1. **You should merge multiple bettings of the same user on the same option**. In this case, use previously assigned betting Id as its id, and write at the same line of the bettingBook. Each match has their own betting ID set. ~~for each betting option.~~

Here are some error codes that can arise:

Output of collectBettings() Method	
ErrorCode	Description
SUCCESS	The betting is successful.
IO_ERROR	IOException is thrown during the execution of the method.
INVALID_BETTING	Invalid betting number. For example, betting on the third option while the match has only two options.
MATCH_NOT_FOUND	The given match ID does not exist in the system.
LATE_BETTING	Current time is not earlier than match time. Current time is determined by the time when the server calls the collectBettings() method. (If a user bet at 14:00 for a match which is held at 14:00, you'll get refunded)

Betting book includes the information about the bettings submitted to the match. It includes the ids of users, which option they bet on and the number of coins they bet.

Specification of Betting book (X_bettingBook.txt)		
	Format	Example
File Path	data/Matches/(Sports Types)/(match id)_bettingBook.txt	data/Matches/Basketball/12_bettingBook.txt

File Content	(userId)|(betting option)|(coins bet)	2022-12345|0|4200 2020-27890|1|3000 2021-13579|2|4000 2022-27279|2|2000 2022-36334|0|3000

`getBettingBook()` method returns the list of bettings of a match.

Question 2-3: Settle a Match [0.5 Marks]

Objective: Implement the following method in the Server class:

1) `boolean settleMatch(int matchId, int winNumber)`

Description: Based on the result of the matches, the winner of the betting is determined. The system should return the appropriate number of coins to the winners. If there's no betting book, return false. If there's a betting book, 4% of the total coins are taken by the server as commission. Then, the rest are distributed to the winners proportional to the coins they bet. The rewards are rounded down at the decimal point (a coin can't be divided).