AY 2020 Assignment 4 [10 Marks]

Date / Time	5 November 2021 – 19 November 2021 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 questions with anyone else and need to code **ALONE**.
- We will use the automated copy detector to check the possible plagiarism of the code between the students. The copy checker is reliable so that it is highly likely to mark a pair of code as the copy even though two students quickly discuss the idea without looking at each other's code. Of course, we will evaluate the similarity of a pair compared to the overall similarity for the entire class.
- We will do the manual inspection of the code. In case we doubt that the code may be written by someone else (outside of the class), we reserve the right to request an explanation about the code. We will ask detailed questions that cannot be answered if the code is not written by yourself.
- If one of the above cases happen, you will get 0 marks for the assignment and may get a further penalty. Please understand that we will apply these methods for the fairness of the assignment.
- Download and unzip “HW4.zip” file from the autolab. “HW4.zip” file contains skeleton codes for Question 1 (in the “problem1” directory) and Question 2 (in the “problem2” directory).
- Do not modify the overall directory structure after unzipping the file, and fill in the code in appropriate files. It is okay to add new directories or files if needed.
- When you submit, compress the “HW4” 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 HW1 ~ HW3. 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.

Contents

Question 1. CPTA [5 Marks]
1-1. Simple Grader [1 Marks]: Collections, File I/O
1-2. Robust Grader [4 Marks]: Collections, File I/O

Question 2. Course Registration System [5 Marks]
2-1: Search Course Information [1 Marks]: Collections, File I/O
2-2: Placing a Bid [2 Marks]: Collections, File I/O
2-3: Confirming Bids [2 Marks]: Collections, File I/O

Submission Guidelines

1. You should submit your code on autolab.
2. Compress the “HW4” directory and name the file “20XX-XXXXX.zip” (your student ID).
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.

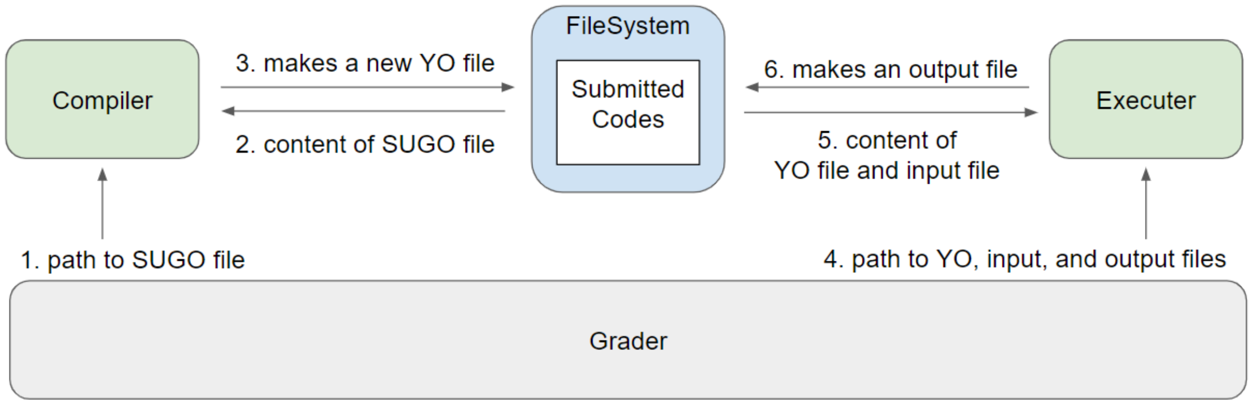
Directory Structure of Question 1	Directory Structure of Question 2
problem1/	problem2/
└── data/...	└── data/...
└── src/	└── src/
└── cpta/	├── course/...
├── environment/...	├── utils/...
├── exam/...	└── server/
├── exceptions/...	└── **Server.java**
└── **Grader.java**	└── Test.java
└── Test.java	

Question 1: CPTA [5 Marks]

Objective: Develop an auto-grading system for Computer Programming class.

Description: You are a TA of a Computer Programming class. You have to implement an auto-grading system that compiles, executes, and grades students' submissions.

- Note: we provide the basic test cases for the Autolab, 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.



To simulate this, we will use a *virtual* compiler and executer. You are provided with the following two classes that imitate the behavior of real ones.

Compiler Class Specifications

Our compiler can understand an imaginary language named SUGO. It converts a source code file (with the extension .sugo) into an executable file (with the extension .yo). The class provides one public method:

- public void compile(String filePath)
  - Compiles the specified SUGO file and creates a YO file of the same name in the directory that the SUGO file is in.
    - For example, compiler.compile("codes/hello.sugo") makes a new file named hello.yo in the directory codes/.
    - If a YO file of the same name already exists, it will be overwritten.
  - It throws CompileErrorException if the SUGO code contains syntax errors. There is no way you can prevent this and must handle it.
  - It throws InvalidFileTypeException if the specified file does not have .sugo extension or the number of periods(.) in the file name is not one.
  - It throws FileSystemRelatedException if an error occurs while reading or writing a file.

Executer Class Specifications

This class runs a YO program and writes the program output to a new file. It also has one public method provided:

- public void execute(String targetFilePath, String inputFilePath, String outputFilePath)
    - Executes a .yo file specified by targetFilePath with the program input given by a file at inputFilePath, and creates a new file containing the program output at outputFilePath.
        - If there is already a file at outputFilePath, it will be overwritten.
    - It throws RunTimeErrorException if the program with the given input results in an error.
    - It throws InvalidFileTypeException if the specified file does not have a .yo extension or the number of periods(.) in the file name is not one.
    - It throws FileSystemRelatedException if an error occurs while reading or writing a file.

You are required to write two methods in the Grader class: gradeSimple for Question 1-1 and gradeRobust for Question 1-2. Both methods perform grading; gradeRobust handles diverse corner cases while gradeSimple does not. We separate two methods to prevent you from corrupting codes for Question 1-1 while solving Question 1-2. Do NOT change the skeleton codes except for Grader.java.

Question 1-1: Simple Grader [1 Marks]

Objective: Implement the gradeSimple method in the Grader class.

Description: Your goal is to implement a grader that grades students' submitted codes according to the specification of an exam. An exam consists of several problems, and each problem will be tested with several test cases.

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

- Parameters
    - ExamSpec examSpec: It contains information about the given exam.
    - String submissionDirPath: It points to the directory where students' submitted codes are located. ※*) Throughout Question 1, it is guaranteed that a string variable whose name ends with "dirPath", including this, ends with '/'(file separator of the platform)*
- Return value
    - Type: Map<String, Map<String, List<Double>>>
    - It returns a Map that maps a student ID to a Map, which maps a problem ID to a list of scores for individual test cases.
        - <studentId, <problemId, list of <scores of individual test cases>>
        - A list of scores for individual test cases should be in ascending order of test case ID.

Classes	Directory Structures
class ExamSpec ● List<Problem> problems ● List<Student> students **class Problem** ● String id ● String testCasesDirPath ● List<TestCase> testCases ● String targetFileName *// below are not used in Question 1-1* ● String wrappersDirPath ● Set<String> judgingTypes **class TestCase** ● String id ● String inputFileName ● String outputFileName ● double score **class Student** ● String id ● String name	**@ Problem.testCasesDirPath** ● 1.in (= one of inputFileNames) ● 1.out (= one of outputFileNames) ● … **@ submissionDirPath** (in gradeSimple/gradeRobust) ● 2019-12345 (= one of Student.ids) ○ problem1 (= one of Problem.ids) ■ Problem1.sugo (= Equal to Problem.targetFileName) ○ problem2 ■ … ○ ... ● 2019-12346 ○ … ● ...

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

- The immediate sub-directories of the submission directory contain students' codes (***'student submission directory'*** hereafter).
  - Student submission directories are named with student IDs.
- The immediate sub-directories of the student submission directories contain students' codes for each problem (***'problem submission directory'*** hereafter).
  - Problem submission directories are named with problem IDs.
- Every student submits codes for all problems.
- Each problem submission directory contains exactly one .sugo file named Problem.targetFileName.
  - The file contains the program that compiles and runs without any errors.
  - You can leave the catch block empty to handle the exceptions thrown by the compile method or execute method.

For each test case, you should compare (1) the output of a student's program and (2) the desired output, which can be found in an output file of the test case. The two files are considered the same when the contents are literally the same. Give a score specified in the test case for the correct answer, and a 0 for the wrong answer.

Hint. Instead of reading a file line by line, consider fetching the whole content in one string.

Note: Files in the data directory may change after running the test cases. The original data files are available in the backup directories (i.e., *data/exam-simple-backup* for *data/exam-simple*, *data/exam-robust-backup* for *data/exam-robust*) in case you want to test your code with the original data files. Alternatively, you can use the resetData method in Test.java to reset the data files to the original states before you run your own test cases.

Question 1-2: Robust Grader [4 Marks]

Objective: Implement the gradeRobust method in the Grader class.

Description: There are many unexpected situations in the real world. Now you are required to upgrade your grader so that it can deal with various corner cases in the submitted codes. In Question 1-2, there are 4 groups of corner cases to handle. You must handle all cases in each group to make it count.

Group 1. Errors

- **Compile error**: Compiler.compile method may throw exception (CompileErrorException, InvalidFileTypeException, FileSystemRelatedException). In this case, all test cases in the problem should get 0 points.
- **Runtime error**: Executer.execute method may throw exception (RunTimeErrorException, InvalidFileTypeException, FileSystemRelatedException). In this case, a corresponding test case should get 0 points.

Group 2. White Spaces and Upper/Lower Cases

Sometimes, we want to be generous about subtle mistakes like printing trailing whitespaces or confusion of lower and upper cases. More specifically, we would like to consider the following three cases (Note: they are not mutually exclusive). We may tolerate all three cases for generous grading. Or, we may want to be strict about all three cases. Problem.judgingTypes specifies which case we want to be generous about. If Problem.judgingTypes is null or an empty set, it indicates that the grading will be strict about all three cases.

- **Leading whitespaces**: If Problem.judgingTypes includes Problem.LEADING_WHITESPACES, a student’s output and the desired output are considered the same if the two outputs contain the same string and the only difference is the existence of additional whitespaces at the start of the string.
- **Ignore whitespaces**: If Problem.judgingTypes includes Problem.IGNORE_WHITESPACES, remove all whitespaces in a student’s output and the desired output before you compare those.
- **Case-insensitive**: If Problem.judgingTypes includes Problem.CASE_INSENSITIVE, do

not differentiate lower-cased letters and upper-cased letters.
- **Ignore Special Characters**: If Problem.judgingTypes includes Problem.IGNORE_SPECIAL_CHARACTERS, remove all characters except alphabets(a~z, A~Z), numbers(0~9), and whitespaces.

In this assignment, "whitespaces" are defined as spaces (' ') and hard tabs ('\t'). Assume that the inputs do not contain other characters that are generally considered as whitespaces such as '\n', '\r', 'U+FEFF', etc.

Group 3. Multiple Source Files

Not all programs are written in a single file. Also, some problems may require TAs to write additional wrapper codes to thoroughly test students' codes.

- **Student codes**: If a problem submission directory contains multiple .sugo files, you should compile them all before you execute the .yo file specified by Problem.targetFileName. If there is a compilation error for any of the submitted files in the directory, it should be handled as specified in Group 1 above.
- **Wrapper codes**: If Problem.wrappersDirPath is not null, copy all .sugo files in that directory to the problem submission directory before compilation; assume that there is no .sugo file with the same name in the submission directory.

Assume that the values of Problem.wrappersDirPath and Problem.targetFileName are always valid. Also, assume that all .sugo files in a problem submission directory are relevant codes and need to be compiled.

Group 4. Submission Format Errors

- **Wrong directory name**: The name of a student submission directory may not be the same as the student ID, but contains it. In this case, there is no penalty.
    - For example, *2019-12345-LT3* instead of *2019-12345*
    - Assume that a student ID is NOT included more than once in directory names.
- **Wrong directory structure**: Sometimes, a problem submission directory contains exactly one directory of an arbitrary name that contains source codes. In this case, there is no penalty.
    - For example, *problem3/src/Main.sugo* instead of *problem3/Main.sugo*
    - We only test the case where there is **only a single additional directory in the submission directory**. Thus, you may ignore the cases with several additional directories. Also, there is no more nested additional directory inside the first single additional directory.
    - Multiple files can exist inside and outside of the additional directory.
        - For example, *problem3/src/Main.sugo* and *problem3/Test.sugo*
    - You have to first move all the files inside the additional directory to outside, then compile.

    - If there are files with the same names inside and outside of the additional directory, overwrite the file when you move the file inside the additional directory. For example, if there are both *problem3/src/Main.sugo* and *problem3/Main.sugo*, move *problem3/src/Main.sugo* to *problem3/Main.sugo*, and then compile.
- **No submission**: Some directories and files in the student submission directories can be missing. Even a student submission directory itself can be absent. You should give 0 points for all problems of which submissions are incomplete.
    - Even when the whole student submission directory is missing, the result of gradeRobust should contain their scores, filled with 0's.
- **Submitted only .yo files**: Some students might have submitted .yo files instead of .sugo files. For each problem submission directory, if there is at least one .yo file without a .sugo file of the same name, use it during execution but cut the scores by half for all test cases for the problem.
    - Check the existence after you copy all the wrapper codes (Group 3) and the files in the additional directory in the submission directory (if it exists).
- **Submitted unnecessary .yo files**: However, if there are both a .sugo and a .yo file of the same name in a problem submission directory, compile the .sugo file to overwrite the .yo file and use it. In this case, there is no penalty.

Question 2: Course Registration System [5 Marks]

Objective: A course registration system based on mileage bidding, not a first-come-first-served basis.

Description: Unlike Seoul National University, many universities run their course registration systems based on mileage bidding. The idea can be briefly summarized as follows.

1. This system provides a certain amount of mileage (e.g., 72) to individual students every semester. The student can place a bid for a course with a certain amount of mileage. The bid amount will be decided based on how much he wishes to take the course.
2. At a certain point, course registrations will be confirmed based on all students' bids. Students who bid higher mileages have the priority for the course registration.
3. You will implement key functions of the course registration system as guided by this document.

Note:

1. Please start with the skeleton code and test data given in the "src" directory. As usual, import the code and data in the Intellij IDEA.
2. In this question, you will implement the methods in Server.java, which is located in the "src/server" directory. 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.

Directory Structure

```
src/
 └── course/
      ├── Bidding.java
      └── Course.java
 └── utils/
      ├── ErrorCode.java
      ├── Pair.java
      └── Config.java
 └── server/
      └── Server.java            // Implement this
 └── Test.java                   // Test cases are in this file.
```

3. There are five java classes (`Bidding`, `Course`, `ErrorCodes`, `Pair`, `Config`) under the `course` 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 SHOULD NOT modify these five classes.
4. 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 Course Information [1 Marks]

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

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

- Inputs:
  - The first parameter `Map<String, Object> searchConditions` can describe three different search conditions as below.

key	value type	value description
"dept"	String	The department providing the course
"ay"	Integer	The academic year for the course
"name"	String	Name of the course

    - If the `searchConditions` is `null` or empty, then it means there are no specific search criteria. In this case, you should return the list of entire courses sorted by the sorting criteria.
    - For the "dept" criterion, you should find the courses that have identical values to the search value.
    - For the "ay" criterion, you should find the courses that have lower or equal values to the search value.
    - The "name" criterion is specially handled. For instance, when the search string is "Computer Engineering", you should consider two keywords "Computer" and "Engineering" split by the space. Then, you should find the courses that contain both keywords, i.e., "Computer" and "Engineering" in the course name. To generalize this, you first need to identify multiple keywords split by the space from the search string, then find the courses containing all keywords in their names. The search should be case-sensitive. When the search string is an empty string, you should return an empty list.
    - For keyword matching, check if the search keyword is identical to one of the words in the course name. For instance, when the search string is "Comp", the course named "Computer Engineering" SHOULD NOT be contained in the search result.
    - If multiple search conditions co-exist, you should search for the courses that match all the specified conditions.
    - Assume that the keys are one of the three strings, i.e., "dept", "ay", "name".

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

sortCriteria	Description
"id"	Sort by the course id in ascending order.
"name"	Sort by the course name in the dictionary order. If the names are equal, sort by the course id in ascending order.
"dept"	Sort by the department name in the dictionary order. If the department names are equal, sort by the course id in ascending order.
"ay"	Sort by the academic year in ascending order. If the academic years are equal, sort by the course id in ascending order.

    - Assume that course id is unique for all the courses.
    - If sortCriteria is null or empty String, sort by the course id.
    - Assume that the sortCriteria string is one of {null, empty String, "id", "name", "dept", "ay"}.
- Outputs:
  - List<Course>: The list of the class Course instances satisfying the given search condition with the elements sorted by the given criteria.
    - Note: use the Course class in the skeleton code.

Note:

1. To solve this problem, you first need to load all the course information using an appropriate data structure. Use the Course class (in the course package) to store the information of the course; do NOT modify the Course class as described above. Assume that the course data does not change after the Server class instance is created.

Course Class
- public String courseName - public String college - public String department - public String academicDegree - public String instructor - public String location - public int courseId - public int academicYear - public int credit - public int quota

- public Course(int courseId, String college, String department, String academicDegree, int academicYear, String courseName, int credit, String location, String instructor, int quota)
- public String toString()
- public boolean equals(Object obj)

2. Course information is given as files. They are organized as the following directory structure below. The “problem2” directory includes the example data. Each text file contains the attributes of a course separated by a vertical bar (see the table below for more details).

Directory Structure

```
data/
└── Courses/
    └── 2020_Spring/
        └── Engineering/
            ├── 11.txt
            ├── 3.txt
            └── …(course id).txt
        └── Fine Arts/
            ├── 5.txt
            ├── 8.txt
            └── …(course id).txt
        └── Humanities/
            └── …(course id).txt
        └── …(College Names)
├── Users/
└── Users_backup/
```

	Format	Example
File Path	data/Courses/2020_Spring/(College Name)/(course id).txt	data/Courses/2020_Spring/Humanities/12.txt
File Content	(department)|(academic_degree)|(academic_year)|(course_name)|(credit)|(location)|(instructor)|(quota)	Aesthetics|Bachelor|4|Topics in French Aesthetics|3|014-207|Milne, Peter W|40

- Assume that each attribute does not include a vertical bar.
- Assume that the first line of the course information text file includes all the information.
- Assume that the course information is formatted correctly; you don’t have to consider wrong formats such as a wrong attribute order, missing attributes, and wrong separators.

Question 2-2: Placing a Bid [2 Marks]

Objective: Implement the following two methods in the Server class:

1) int bid(int courseId, int mileage, String userId)
2) Pair<Integer, List<Bidding>> retrieveBids(String userId)

Description: The bid method allows a student to place a bid for a course while the retrieveBids method returns the information of all placed bids. Note: we describe the bid method first and the retrieveBids method later, but you may want to implement them in reverse order after reading the entire question.

Firstly, implement the following bid method.
int bid(int courseid, int mileage, String userId)

- Inputs:
  - int courseId: Course id that the user wants to place a bid.
  - int mileage: The amount of mileage the user wants to bid.
  - String userId: User id to specify the user.
- Outputs:
  - int: Error code specifying the status of the method call.

ErrorCode	Description
SUCCESS	The bidding is successful.
IO_ERROR	IOException is thrown during the execution of the method.
OVER_MAX_MILEAGE	The sum of previously bid mileages and the current bid mileage exceeds the maximum allowable mileage. Note that the maximum mileage is defined as a static variable MAX_MILEAGE in the Config class.
OVER_MAX_COURSE_MILEAGE	The given mileage exceeds the maximum allowable mileage per course. Note that the maximum mileage per course is defined as a static variable MAX_MILEAGE_PER_COURSE in the Config class.
OVER_MAX_COURSE_NUMBER	The number of bid courses exceeded the maximum allowable number of courses. Note that the maximum number of courses is defined as a static variable MAX_COURSE_NUMBER in the Config class.
NEGATIVE_MILEAGE	The mileage is a negative integer.
NO_COURSE_ID	The given course id does not exist in the system.

USERID_NOT_FOUND	The given user id does not exist in the system.

- 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 bid and return SUCCESS. Otherwise, do not place the bid and return the proper error code.
- When multiple errors occur, return the error code with the lowest value.

Note:

1. Each user gets a maximum amount of mileage to be spent in total. The maximum mileage is defined in the Config class (as a static variable MAX_MILEAGE = 72).
2. There is maximum allowable mileage for each bid, which is defined in the Config class (as a static variable MAX_MILEAGE_PER_COURSE = 18).
3. The bid information should be stored in files (not in the memory only). The bid information should not be removed even if the server has been stopped and restarted.
4. More specifically, the bid information of a user should be stored in a file (named “data/Users/(user id)/bid.txt”), as described in the tables below. You have to save the bid information in the “bid.txt” file for each bid method call. Note that there is already bid information in the pre-existing “bid.txt” file, and you have to append the new bid information in the file for each bid method call.
5. There is a maximum allowable number of courses that each user can bid, which is defined in the Config class (as a static variable MAX_COURSE_NUMBER = 8).
6. Assume that a valid student has a directory named with his/her ID under the “Users” directory, and has the file “bid.txt” in it. If the corresponding directory for a user id does not exist, it means that no such student exists. In such a case, the USERID_NOT_FOUND error should be returned. You may want to consider defining a User class to manage the user information.
7. If the input parameter userId is null, the USERID_NOT_FOUND error should be returned.
8. If there is a valid user id directory with no “bid.txt” file, the bidding is ignored. This case, IO_ERROR error should be returned.
9. If the user already has a bid for the same course, you should replace the existing bid with the new amount.
10. If the user bids 0 mileage for the previous bid course, then the previous bid should be canceled. The canceled bidding should be removed from both the bidding list in the memory and "bid.txt".
11. If the user bids 0 mileage for the new course, then ignore the bid.
12. We provide a resetUserDirs method to clear all user directories and reload the default user information. You can freely use this method in Test.java to make appropriate test cases.

Directory Structure

data/
├── Courses/
└── Users/
    └── 2010-22221
        └── **bid.txt**
    └── 2012-22221
        └── **bid.txt**
    └── ...(user id)
└── Users_backup/
    └── (Backup of the /data/Users)

	Format	Example
File Path	data/Users/(user id)/bid.txt	data/Users/2018-22233/bid.txt
File Content	(course id)|(mileage) (course id)|(mileage) (course id)|(mileage) (course id)|(mileage) ...	10|17 9|15 8|15 1|1 3|2

- This is the format of the initially given "bid.txt" file.
- Initial bid information has no duplicate course ids, but can be unsorted.

Secondly, implement the following retrieveBids method.
Pair<Integer, List<Bidding>> retrieveBids(String userId)

- Inputs:
  - String userId: User id to specify the user.
- Outputs:
  - Pair<Integer, List<Bidding>>: The key is the error code (defined in ErrorCode class). The value is the list of previously placed bids. The list of the biddings does not need to be sorted.

ErrorCode	Description
SUCCESS	Successfully retrieved the user's bids.
USERID_NOT_FOUND	The given user id does not exist in the system.
IO_ERROR	IOException is thrown during the execution of the method. Handle this only if necessary. This will not be tested.

- When multiple errors occur, return the error code with the lowest value.
- If the input parameter userId is null, the USERID_NOT_FOUND error should be returned.
- When an error occurs (USERID_NOT_FOUND or IO_ERROR), the key of the output pair should be the error code, and the value of the output pair should be an empty list.

Note:

1. Use the Bidding class (described in the table below) and Pair class to generate the output. Refer to the skeleton code for the Pair class (it is very simple!).

Bidding Class
- public int courseId - public int mileage
- public Bidding(int courseId, int mileage) - public String toString() - public boolean equals(Object obj)

Question 2-3: Confirming Bids [2 Marks]

Objective: Implement the following two methods in the Server class.

1) boolean confirmBids()
2) Pair<Integer, List<Course>> retrieveRegisteredCourse(String userId)

Description: The confirmBids method determines who will be registered for which courses based on the bids placed so far.

boolean confirmBids()

- Outputs:
  - boolean:
    - **false**, if IOException is thrown during the execution of the method.
    - **true**, otherwise.

The clearing logic is as follows.

1. The basic rule is that students who bid larger mileage to a course will be registered for the course if the quota is limited.
2. If multiple students bid the same mileage for a course and all of them cannot fit into the quota of the course, the student who spends less mileage in total has the priority. If this second criterion cannot break the tie, a student with the preceding user id (in the dictionary order. e.g. 2012-12345 precedes 2012-12435) gets the priority.
3. The confirmed registrations (i.e., who is registered for which courses) should be stored in

files. Similar to the bids, the information of the confirmed registrations should not be removed even if the server has been stopped and restarted.
4. Feel free to create a file to store the confirmed courses for a user under the corresponding user directory. You can freely decide the file format.
5. All stored bids should be removed after the confirmation. Remove all bid information in every users' "bid.txt" files whether the registration is successful or not.
6. Similar to the bids, you can use resetUserDirs method to clear all user directories and reload the default user information. You can freely use this method in Test.java to make appropriate test cases.
7. You can assume that confirmBids will not be called consecutively before the user information is reset; we will only consider a single round of bidding and clearing. Multiple rounds of bidding are not considered.

Secondly, implement the retrieveRegisteredCourse method.

Pair<Integer, List<Course>> retrieveRegisteredCourse(String userId)

- inputs:
  - String userId: User id specifying the user.
- outputs:
  - Pair<Integer,List<Course>>: The key is the error code. The value is the list of the Course instances that are confirmed to be registered. The list does not need to be sorted.

ErrorCode	Description
SUCCESS	Successfully returned the user's registered courses.
USERID_NOT_FOUND	The given user id does not exist in the system.
IO_ERROR	IOException is thrown during the execution of the method.

Note:
- Assume that retrieveRegisteredCourse is called only after confirmBids method is invoked.
- If the input parameter userId is null, the USERID_NOT_FOUND error should be returned.
- When an error occurs (USERID_NOT_FOUND or IO_ERROR), the key of the output pair should be the error code, and the value of the output pair should be an empty list.