COMPUTER SCIENCE 21A (FALL, 2022)
DATA STRUCTURES AND ALGORITHMS
DUE: FRIDAY, SEP 9TH, 11.59PM
PROGRAMMING ASSIGNMENT 0 – UP AND DOWN

Overview:

In this assignment, you will be simulating a building that contains a single elevator. However, this elevator is somewhat simplified: it only moves a person to a floor and then the person will remain on that floor forever. That is, the elevator cannot pick people up from any floor besides the lobby.

People will be entered into a building going to a specific floor. Two things need to happen. As a person arrives at the building, the elevator may or may not be in the lobby. To pick them up, the elevator (1) must be sent to the lobby to pick them up and then (2) bring them to their floor.

An elevator has a maximum number of people it can "carry" at a time. For instance, suppose an elevator can carry 3 people at a time and 5 people arrived in the lobby to be serviced by the elevator. The elevator would bring the first 3 people to their floors, then return to the lobby and bring the last 2 people to their floors. The order in which a group of people are brought to their floors is the order in which they requested use of the elevator (i.e., first-come-first-serve). Thus, the elevator will not be very efficient.

You will be coding an object-oriented solution, where you will need to implement the following classes. More details regarding their implementation will follow.

Building: A Building should hold an array of Floors and an instance of an Elevator. A Building will have a lobby and then floors numbered 1, 2, up to some maximum.

Floor: A Floor must contain an array of Person objects that are currently on the Floor.

Elevator: An Elevator will contain an array of Jobs where each Job holds a Person and the number of the floor which they desire to go to. **All printing done in your program outside of Simulation.java should be done in the `processJob()` method of this class.** This will be discussed in more detail on page 4.

Person: A Person has a first and last name and information regarding their location in a Building.

Job: A Job is used to represent a "request" to the elevator. This class holds a reference to a Person and a floor number. You can use a null Person to represent a Job that calls the Elevator to the lobby.

Simulation: This class should contain a main method where you simulate a Building. You should instantiate a Building, and several Person instances. Then you should simulate

various arrivals of people arriving at different times and requesting to use the Building's Elevator. It is recommended that you use this method to test your classes thoroughly. **This is the only class where you should put print statements outside of Elevator.**

Implementation Details:

In general, the client program should be able to create Person instances and a Building instance and then be able to add people to the Building's Elevator and start it in various orders. For instance, one may choose to add a Person to the Building's Elevator and start it. After this, two more Person instances could be added to the Building's Elevator before starting it, and so on. The necessary details of each class that you will be creating to provide this functionality are described below.

Note: You must implement the following methods following these exact signatures. If a class does not have a specific constructor declaration, feel free to pass as many parameters into that class' constructor as you see fit. You may use additional methods in your classes in order to have an organized and modular solution.

If you draw a simple diagram of how these classes will interact and what the flow of the program will be before you start coding, it will be easier to complete this assignment.

Building.java

- `public Building(int numFloors)-` This constructs a Building that has some number of floors. When you construct the Building, its Elevator should be at the lobby.

- `public boolean enterElevatorRequest(Person person, int floor)` — This method will handle the request made by a Person to enter this Building and be taken to some floor. This will involve ensuring that the Person's desired floor can be reached by this Building's Elevator. If the Elevator cannot reach the Person's desired floor, return false. If the elevator can reach the person's desired floor, return true. **This method should ensure that the Elevator will never process this Person's request if it is not valid.**

- `public void startElevator()` — This will call a method in this Building's Elevator instance that should process all of its current Jobs.

- `public void enterFloor(Person person, int floor)` — This method should save a reference of a Person in the Floor with the provided floor number.

Person.java

- `public Person(String firstName, String lastName)` - This will construct a Person with a given first and last name.

- `public boolean enterBuilding(Building building, int floor)` – This method will enter this Person into a Building's Elevator with a provided destination floor number. This should return true if the Person can be brought by the elevator to 'floor' and false if not. *You may assume a Person will only ever try to enter one Building*.

- `public String getLocation()` – This should return this Person's location as a String. To do this, you should maintain a field that tracks this Person's location in some way. When this method is called it should return the following **exact Strings**:
    - "Waiting to be serviced" from the time this Person is validated by a Building (to be processed) to the time they arrive on their desired Floor.
    - "In Lobby" if this Person is in a Building's lobby and will **never** be serviced by the Building's Elevator.
    - "In Floor x" if this Person is on Floor number x.

Job.java

This class may be implemented how you choose. Its primary purpose is to store requests for the Elevator to service. Each request comprises of a Person and the number of the floor they wish to be taken to.

Elevator.java

When implementing methods that create and process Jobs, remember that Jobs are handled first-come-first-serve. This class includes a single static field `maxOccupants` which specifies the maximum number of people an Elevator can bring to their Floors at a time. **You must be using this static field in your implementation.** You cannot remove it, but feel free to test your code for values other than 3.

- `public Elevator(// whatever params you want)` – This should construct an Elevator **on the lobby** with no current Jobs.

- `public void createJob(Person person, int floor)` – This method should add a new job to be completed by this Elevator given a Person and their desired floor number.

- `public void processAllJobs()` – This method should remove jobs one by one and process them.

- `public void processJob(Job job)` – This method should process a Job. This will involve moving this Elevator from its current floor to the Floor indicated by the provided Job. Once the desired Floor is reached, you should use the `exit()` method to move the Job's Person to their destination Floor in their

Building.
You must print the Elevator's location before you move it and after each time you move the Elevator by a floor (either up or down).
  - When the Elevator is in the lobby print "Elevator at Lobby".
  - When the Elevator is on floor x, print "Elevator at floor x"

For instance, a job processing could involve moving this Elevator from the lobby to floor 2. The printed output for this would look like:

```
Elevator at Lobby
Elevator at floor 1
Elevator at floor 2
```

A job processing may also involve moving an Elevator down to a floor. Say the Elevator starts on floor 3 and needs to go to floor 1. The printed output for this would look like:

```
Elevator at floor 3
Elevator at floor 2
Elevator at floor 1
```

- `public void exit(Person person, int floor)` – This method should remove a Person from this Elevator onto their Floor in the Building the Person is in.

Floor.java

- `public void enterFloor(Person person)` – This method should save a reference of the Person within this Floor object

Simulation.java

This class should be used to run your Simulation. In this class, you should only be constructing Person and Building objects. You should not be interacting with your Elevator class at all. The methods you should be using are: `enterBuilding()` to enter a Person into a Building and `startElevator()` to have the Elevator in your Building begin processing its Jobs.

Lastly, you must write a `toString()` method with your own implementation for each class.

If you think that you need extra classes, feel free to implement them. However, make sure to explain why you added them and what advantage they give your program in your README file. We are not looking for a uniquely right solution, so use your own judgement as a programmer and **describe your logic in a README.txt file** in addition

to thoroughly commenting your code.

JUnit Tests

In the "tests" package of the Eclipse project distributed on LATTE, there are 5 JUnit test files corresponding to the 5 classes you must write. **You must write JUnit tests for at least 2 of these classes.** It is recommended that you use your classes' `toString()` methods to write the tests.

If you write JUnit tests for additional classes, you will receive extra credit.

Debugging

The Eclipse debugger would be quite useful for this assignment. If you are not comfortable with the debugger, you should use print statements along with classes' `toString()` methods to isolate the source of an error.

However, when you submit your assignment, make sure that you remove all print statements besides the required ones in `processJob()`.

Important Notes

You should NOT import any libraries, such as java.util.Arrays, or Java collections in your code. The only Java-provided data structure that you will be using is an array.

Lastly, double-check the following:
1) Outside of Simulation.java, make sure you are printing **only** in `processJobs()` and that you are printing the **exact** Strings described on page 4 before moving the Elevator and after every time it is moved by 1 Floor.
2) Make sure that you are returning the **exact** Strings in `getLocation()` described on page 3.

Submission Details

- For every non-test method you write, you must provide a **proper JavaDoc** comment using `@param`, `@return`, etc.
- At the top of **every file** that you submit please leave a comment with the following format. Make sure to include **each of these 6 items.**

```
/**
 * <A description of the class>
 * Known Bugs: <Explanation of known bugs or "None">
 *
 * @author Firstname Lastname
 * <your Brandeis email>
 * <Month Date, Year>
```

```
 * COSI 21A PA0
 */

>Start of your class here<
```

- Use the provided skeleton Eclipse project.
- Submit your assignment via Latte as a .zip file by the due date. It should contain:
  1) Your solution in an Eclipse project.
  2) Your README.txt (more on this on page 2)
- Name your .zip file **LastnameFirstname-PA0.zip**
- Please check the syllabus for the late submission policy.