T-444-USTY, Programming Assignment 1

March 1st, 2019

1 Instructions

- You **must handin** this assignment through Canvas or it will **not** be graded.
- Late handins will **not** be graded.

1.1 Handout instructions

The project is on Canvas. There is a ZIP archive with the base program. The base program contains few files.

com.ru.usty.elevator

- ElevatorMainProgram.java
- ElevatorScene.java

com.ru.usty.elevator.visualization

- ElevatorMainProgram.java
- ElevatorScene.java

In Eclipse you can do File-import-general-existing projects into workspace. Then pick the folder and import the project. Make sure you do refactor-rename on the project to change it to your own names. To run the program you need to install an older version of Java (Java SE 6). Instructions will be uploaded on Canvas.

1.2 Handin instructions

Once you have finished the project, re-archive the folder and return it into the Canvas assignment system (code, 0 points), and upload the design document PDF to the main assignment. Students can work on this assignment in groups of up to three. If working in a group, students need to handin the assignment as a group on Canvas.

1.2.1 Design document

The design document should contain the following

- The semaphores that will be made and used in the system
- The events (elevator stops on floor, elevator leaves floor, person enters floor, etc.) that happen in the system
- What operations are called on the semaphores when these events occur

Any images and system descriptions are also welcome.
The design document should state what decisions were made, what classes are made, why and what their purpose is.

2 The Problem

People enter a building and wish to use an elevator to travel between floors. Each elevator is implemented as a thread and each person is implemented as a thread. At any given time the system will include a separate thread for each elevator and each person that has entered the system but not yet finished its run by exiting at the floor it wishes to travel to.

The thread should wait at the appropriate places and unlock certain lock at the appropriate times, to make sure that other threads can also conclude their run correctly.

In the design work you can assume that the elevator threads are already running and that an outside source starts the person threads and lets them know where they are and where they wish to go. You still need to design the running functionality of the threads.

Following are the different levels of complexity:

- **Single elevator (50 %)**: two floors, everyone enters at the bottom floor and exits at the top floor.
- **Single elevator, any number of floors (Intermediate)**: everyone enters at the bottom floor and exit at any other floor. The person-thread knows from the beginning at which floor it wishes to exit.
- **Single elevator,two floors (Intermediate)**: persons can enter at either floor and exit at the other floor.
- **Single elevator, any number of floors (30 %)**: persons can enter and exit at any floor. The person-thread knows from the beginning at which floor it wishes to exit.
- **Any number of elevators, any number of floors (20 %)**: persons enter and exit at any floor. Persons enter at a floor and can use any elevator to travel.

Each system can be described either for a single person in an elevator at a time, or for 6 people at a time in an elevator. You can decide when this level of complexity is added, but in the end each system should allow for 6 people in an elevator at a time.

Feel free to add any intermediate levels of complexity if this helps you get your head around the problem. You can also skip levels in the final document if you feel they are included in other levels, but take care not to add too much at a time, so you have better overview over what causes errors and problems in your system.

Bonus Points

- **(5 %)**: Solve test case 5 in TestSuite without having to starve certain floors until other floors are empty.
- **(5 %)**: Implement the idea of up/down buttons, and have people only enter elevators when they are going in the correct direction.
- **(5 %)**: Make the entire system thread safe, by verifying mutual exclusion on all shared variables.

Comments in the project code describe what needs to be done where, and what is allowed, but also look at the following process description for better descriptions of the problem.

3 The Base Program

When you open the base project on Canvas, it contains the following files.

com.ru.usty.elevator

- ElevatorMainProgram.java
- ElevatorScene.java

com.ru.usty.elevator.visualization

- ElevatorGraphics.java
- TestSuite.java

This project contains a visualizer to help you debugg your code.

3.1 ElevatorMainProgram

Use this class to run and test your code. This class contains a function call to TestSuite:

```
TestSuite.runTest(0);
```

This line of code runs test case number 0. Replace the number enclosed by the parentheses to run a different test case. You can do whatever you want there, as long as it handles being switched out again for our own main() function when grading.

3.2 ElevatorGraphics

The class ElevatorGraphics should be left alone, as it is only there to display the state of your system.

3.3 TestSuite

You don't need to add anything to the class TestSuite, but if you wish to make new test cases you can change the functions `initScene`, which decide how many floors and elevators are in the system, and `runScene` which is run in the thread that adds people and decide where they are added and which floor they wish to exit at. When grading, these functions will be switched out for new ones.

3.4 ElevatorScene

In the java project there are 4 classes. Your main entry point to the system is through the class ElevatorScene. It contains several functions that the system calls at different times. The definitions of those functions can not be changed but you can do anything with the code inside them. You can also add functions to this class and you can of course add classes to the system to your heart's content.

```
public class ElevatorScene {

  public static final int VISUALIZATION_WAIT_TIME = 500;  //milliseconds

  private int numberOfFloors;
  private int numberOfElevators;
  ArrayList<Integer> personCount;
  ArrayList<Integer> exitedCount = null;
  public static Semaphore exitedCountMutex;

  //Base function: definition must not change
  public void restartScene(int numberOfFloors, int numberOfElevators) {}

  //Base function: definition must not change
  public Thread addPerson(int sourceFloor, int destinationFloor) {}

  //Base function: definition must not change, but add your code
  public int getCurrentFloorForElevator(int elevator) {}

  //Base function: definition must not change, but add your code
  public int getNumberOfPeopleInElevator(int elevator) {}

  //Base function: definition must not change, but add your code
  public int getNumberOfPeopleWaitingAtFloor(int floor) {}

  //Base function: definition must not change, but add your code if needed
  public int getNumberOfFloors() {}

  //Base function: definition must not change, but add your code if needed
  public void setNumberOfFloors(int numberOfFloors) {}

  //Base function: definition must not change, but add your code if needed
  public int getNumberOfElevators() {}
```

```
  //Base function: definition must not change, but add your code if needed
  public void setNumberOfElevators(int numberOfElevators) {}

  //Base function: no need to change unless you choose not to "open the doors" sometimes
  //         even though there are people there
  public boolean isElevatorOpen(int elevator) {}

  //Base function: no need to change, just for visualization
  public boolean isButtonPushedAtFloor(int floor) {}

  //Person threads must call this function to let the system know that they have exited.
  //Person calls it after being let off elevator but before it finishes its run.
  public void personExitsAtFloor(int floor) {}

  //Base function: no need to change, just for visualization
  public int getExitedCountAtFloor(int floor) {}
}
```

The most important functions are two.

1. `restartScene` is where you make your scene, make new threads for each elevator, make and initialize the semaphores needed and prepare everything for accepting people into the system.

2. `addPerson` is where a new person-thread is made with a person that enters at a specific floor and wishes to exit at another floor. These floors are variables in the addPerson function.

In addition there are a few functions that need to be changed so that they return correct values. For example, the return values are hard coded for `getCurrentFloorForElevator` and `getNumberOfPeopleInElevator`, both of which take a parameter stating for which elevator a value is needed.

You can implement these functions in way you wish, but the integer values for the parameter are indexes from 0 to number of elevators - 1 (0 in all levels of complexity except the last).

Look at the rest of the functions in the class `ElevatorScene` and decide whether they need to change.

The list `peopleCount` holds an integer for each level. If you wish to use this list you must make sure it is updated correctly when people enter elevators and when new people enter floors.
You can keep track of this in way you wish, but you must make sure that all the functions, e.g. `getNumberOfPeopleWaitingAtFloor` return the correct value so that the graphical visualization displays everything correctly.

Also realize that more than one thread may call some of these functions. Make sure that mutual exclusion is used where needed, that is, that while one thread is writing to a variable or adding to a list, no other thread is accessing it at the same time. There is no need to accommodate multiple readers. Simply one thread wither reading or writing at a time is fine.

4 The Process

Before you start coding, design your solution. Start with the simplest implementation.

4.1 Design

It is good to start by drawing a simple diagram of the problem you want to solve. This would first be an image of a single elevator that can be on one of two floors, along with a line of people entering on the bottom floor and an exit for people on the top floor. People only enter on the bottom floor and exit on the top floor.

Next you can write a list of events that can happen in the system, e.g. elevator leaves top floor, elevator stops at bottom floor, person is created, person is allowed into elevator, etc.
When you feel this list is ready it is time to list, for each event, what the thread in question (person or elevator) need to wait for at this point, and what locks need to be opened when this event happens.

The list might look like this:

- Person enters the system
  - Person-thread needs to wait for an elevator to stop at the floor
  - ...
- Elevator stops at bottom floor
  - Elevator-thread needs to unlock threads waiting at the floor
  - ...
- Elevator leaves top floor
  - ...
- ...

For each such line where a thread needs to wait or unlock other threads you must make the following decisions:

- Can this wait or unlock be achieved with a simple operation on a semaphore?
- ... or does it need a counter or access to a shared memory object?
- Can you use a semaphore already in the system?
- ... or do you need to add a semaphore? Add it to your diagram.

And without doubt you will sometimes need to make other decisions as well. Finally go through this system, your diagram and list of events, and make sure every wait and unlock adds up.
Make sure no thread is starved and that no locks cause deadlocks, but also that no thread can get through the system without waiting for the appropriate timing.
Once you have one such system described you can add the events need to solve the next level of complexity, and go through the steps again for that system.

4.2 Coding

Start by looking at the Java class `Semaphore`. Also look at the library `java.util.concurrent`, if you wish to use other methods than only counting semaphores in the more complex parts of the system.

To implement a thread it is best to make a class, e.g. Elevator or Person, that implements the `Runnable` interface. Then you can construct a new instance of Thread, which takes a new instance of your class as a parameter, and call `start()` for the thread. The functionality of the thread is then implemented in the `run()` function of the class implementing `Runnable`.

If any class needs to access `ElevatorScene` to update variables there you can make a constructor on that class that takes `ElevatorScene` as a parameter and sets its own instance to that (which is a **reference** to the main scene). You can then send `this` into the constructor.
Alternatively you can make a globally accessible instance of `ElevatorScene` that gets that same value. Just make sure that **only one instance** of `ElevatorScene` is used at any time, and that it is the instance that all classes needing it have access to.

4.2.1 Threads for elevators

Threads for elevators need to live in the system from the time `restartScene` is called and until `restartScene` is called again.

It would be good, in the beginning of that function, to collect all currently living elevator threads (if any), give them a signal to stop and then `join()` them before beginning to build the threads needed for the next scene.

The elevator threads must move between floors, up or down one at a time, and control the locks or constraints that make sure people only enter and exit at the correct time (and at the correct floor).

Each time an elevator switches floors, after finishing any operations needed when it enters a new floor, it must wait `VISUALIZATION_WAIT_TIME` milliseconds. This is done by calling `Thread.sleep(ElevatorScene.VISUALIZATION_WAIT_T` as can be seen in the `TestSuite` class.

After this wait the elevator performs any operations needed before leaving a floor and then switches floors again. This wait time is the time person-threads have to enter the elevator. Person-threads do not need to implement any sleeping of their own.

4.2.2 Threads for persons

Threads for persons need to exist from the time `addPerson` is called until they have both entered an elevator and exited it at the correct floor. The person-thread can simply finish its run then.

You should return the instance of Thread that runs the person in the function `addPerson` (when the thread is created). That way `TestSuite` will handle waiting for every thread to finish, so that the next scene doesn't suddenly start before everything was finished in the previous one, in case several scenes are being run sequentially.

A person-thread simply needs to wait for there being room for it in an elevator, then to wait for the correct elevator to let it out at the correct floor and finally call any operations, possibly to let know that the room is free again, but this depends on your implementation of the semaphores, and who calls which operation to lock/unlock them, and when.

A person thread must call `personExitsAtFloor(floor)` with the correct value of exit floor when it's let out of the elevator, before it finishes. That way people will be shown after they have exited the elevators.

4.3 Additional information

You should think about how different parts of this problem fit with the problems we have discussed in lectures, e.g. **Mutual Exclusion** and **Producer-Consumer**, and then what is the producer and what the consumer and what semaphores are needed to control access to all resources (and which re- sources they are).
You should also make sure to solve all mutual exclusion problems that may come up as side effects to the main problem, due to multiple threads needing access to the same information (e.g. the graphical visualization) to keep track of the state of the system at each time.

Take note that even though big parts of the project can be solved by setting semaphores, it can also be beneficial to keep track of certain things in shared variables in memory, extra counters and such.

In some levels of complexity you can also look at the possibility of making extra classes, even extra threads, that keep track of and connect certain data and threads, again with counters or semaphores that can be used to synchronize the main threads.

All such solutions shall then also be described in the accompanying document, to make sure that teachers spot students' brilliance in solving these complex problems.

Have a good time!