Homework 1

CSCIE-55

Fall 2019

Fred Evers

# Homework 1: Elevator Simulation 1

Due: Monday, September 16th 11:59 Eastern time

Grading: 15%

Last Modified: Friday, 6-Sept-2019 7:39:58 EDT

# Specification of Requirements

[Note: Requirements are not suggestions, they are requirements. Your work should meet them strictly and completely. Do not supplement or modify them. Do not try to design a better Elevator. In this respect the assignments mimic real world software tasks. (Think ‘Product Management’ 😊)]

In this assignment, you will begin to develop a simulation for a building's elevator system. The major objects you will deal with in assignments for this class are:

Elevator: Carries Passengers between Floors. Has a limited capacity.

Building: Contains multiple Floors and one Elevator. (While multiple Elevators are possible, the assignments will only deal with one.)

Floor: Passengers use the Elevator to spend some time on a Floor, and then use the Elevator to go to another Floor.

Person: A generic representation with fields/properties needed for our domain model.

Passenger: A Person who enters and leaves the Building. While inside the Building, a Passenger is either on a Floor, or in the Elevator, moving between Floors.

And more to come…

## First Assignment: Create the Elevator class.

**In the first homework, you will only be concerned with the Elevator class.** (Other classes will be added in later assignments.) This initial version of the Elevator class will keep track of the number of passengers destined for each floor, and whether a stop on that floor is required. Any number of passengers may board the elevator. Elevator capacity will be limited in later assignments.

The Elevator starts on the ground floor and goes up. When it reaches the top floor it starts going down. When it has gone to one cycle, from top to bottom, it stops and should empty any passengers. When moving in either direction, it continues moving, stopping only at destination floors. That is, a stop on a floor is necessary if there are passengers destined for that floor. [In later assignments, a passenger on a floor may summon the Elevator, requiring it to stop there, even if the Elevator is not carrying any passengers who want to get off there.]

When the Elevator stops on a destination floor, it should discharge the passengers who wanted to go there and change its state to reflect the fact that the floor is no longer a destination. Then it moves on to the next floor, ultimately stopping at the bottom.

## Tasks

**Fork and Clone homework\_1 from BitBucket.**

**Elevator class**

Write an Elevator class as follows:

You will find a ‘stub’ file for an Elevator class in the provided directory src/main/java/cscie55/hw1/elevator. Use the package name ***cscie55.hw1.elevator*** in your Elevator.java class. The class should have a no-argument constructor that sets up the elevator's state. [Note that there is no main() method in this class. Instead, the main method will be in the ElevatorTest class described after the Elevator class.]

* Define a constant, (i.e., a static final field), for the number of floors in the building, and set it to 7. Use it where appropriate.
* Define a field for tracking the Elevator's current floor
* Define a field for tracking the Elevator's direction of travel.
* Define an array-valued field for tracking, for each floor, the number of passengers destined for that floor.
* Define a move() method which, when called, modifies the Elevator's state, (i.e., updates the fields appropriately):
* Increments/decrements the current floor, i.e. the Elevator moves one floor at a time.
* Modifies the direction of travel, if the ground floor or top floor has been reached.
* Clears the array entry tracking the number of passengers destined for the floor that the elevator has just arrived at.
* For each floor, if it stops or not, the move() method prints out the status of the Elevator [see toString() method below]
* Define a boardPassenger(int destinationFloor) method
* This method adds to the Elevator one passenger destined for the indicated floor.
* Define a toString() method to aid in debugging and testing.
* The String returned by toString() should indicate the number of passengers on board, and the current floor. So each time the toString() method is called, it returns something like: Floor *n*: *n* passengers

This list defines requirements of the Elevator class. Your submission should therefore define all the fields and methods specified above, with appropriate access modes (private, public, etc.) You are free to provide additional fields and methods that you may find to be useful.

In this homework, the only reason to stop at a floor is that a passenger wants to get off there. However, in future assignments, passengers on a floor who want to leave the building (for example) will summon the Elevator, and then the Elevator must stop there, even if nobody on the Elevator wants to stop there.

**Testing**

To demonstrate that your Elevator class is working properly, you must create tests. You will find an ElevatorTest.java ‘stub’ file in provided directory src/test/java/cscie55/hw1/elevator.

Create the ElevatorTest class with package name ***cscie55.hw1.elevator***.

The main() method in this class should do the following:

Create an instance of an Elevator object.

Board two passengers for the 3nd floor, and one for the 5th floor.

Move the Elevator from the ground floor to the top floor, and then back to the ground floor.

Print the state of the elevator before the first move, and after each move. Your output should look like this:

Floor 1: 3 passengers

Floor 2: 3 passengers

Floor 3: 1 passenger

Floor 4: 1 passenger

Floor 5: 0 passengers

Floor 6: 0 passengers

Floor 7: 0 passengers

Floor 6: 0 passengers

Floor 5: 0 passengers

Floor 4: 0 passengers

Floor 3: 0 passengers

Floor 2: 0 passengers

Floor 1: 0 passengers

**Compile**

To compile these files you will have to resort to a trick. [You *could* compile them individually, but getting the ElevatorTest class to compile would be a chore since the compiler doesn’t know how to find the Elevator class.] The javac command can be instructed to compile a list of files present in a file using a special flag.

First, navigate to your java source root directory [‘src’]. Then in your bash shell do:

*find . -name \\*.java > files.txt*

This will generate a file in your current directory named files.txt [the choice of name and extension is irrelevant]. Then, from the same directory do:

*javac @files.txt*

This will compile all of the files with .java extensions beneath the current directory. And the compiler puts all of the classes in a single memory space, so your references will resolve. [Of course, the code has to be written correctly…]

**Run during development**

Since the main method is in your ElevatorTest class, and since ElevatorTest depends on Elevator class, you will need to tell the java program how to find Elevator.class. There are ways to do this in linux:

Create a symlink between the Elevator.class file and the directory where ElevatorTest.class is located.

Assignments/homework\_1/src/test/java (master)

$ cd ../../

Assignments/homework\_1/src (master)

$ ln -s main/java/cscie55/hw1/elevator/Elevator.class test/java/cscie55/hw1/elevator/Elevator.class

[I could not get symlinks to work in my Git bash shell. If you dig around on Google you may find a solution.]

Alternatively, copy the compiled Elevator.class file int the directory where ElevatorTest.class is located. [That is: test/java/cscie55/hw1/elevator/] **But remember to remove it before you create your jar file.**

Create jar.

Happily, the same trick we used to compile also works with creating jars.

To create a file that lists the path to all .java file AND all .class file, cd up the common parent directory “homework\_1/src”. Then do:

***find . -name \\*.java -o -name \\*.class > files.txt***

Then, create your jar as follows.

***jar cvf ../evers\_fred\_hw\_1.jar @files.txt***

[Note that I put “../” in front of the jar file name, which tells the jar command to put the generated jar one level up in the ‘homework\_1’ directory. This is optional, but I like it because it is a bit cleaner.]

You should see output like this:

added manifest

adding: main/java/cscie55/hw1/elevator/Elevator.class(in = 2223) (out= 1245)(deflated 43%)

adding: main/java/cscie55/hw1/elevator/Elevator.java(in = 8151) (out= 1998)(deflated 75%)

adding: test/java/cscie55/hw1/elevator/ElevatorTest.class(in = 1171) (out= 674)(deflated 42%)

adding: test/java/cscie55/hw1/elevator/ElevatorTest.java(in = 2933) (out= 837)(deflated 71%)

**Documentation**

For each method in your classes, put comments using the block comment style:

**/\*** some description of the purpose of the method **\*/**

For this assignment, edit your README file providing an overview of how your methods solved the problem started in the requirements.

**Submit.**

Do: ***git add*** for the **generated jar**. Also do ***git add*** for the **README** file. Since you have added your source .java files to the jar file, everything you need to submit for this assignment is inside there.

Do: git commit -m “put a short message here reminding you what this is”

Then do ***git push*** to send your work to BitBucket.

Finally, go to Canvas and add your jar and a text based copy of your README.