Homework 2

CSCIE-55

Fall 2018

Fred Evers

# Homework 2: Expanded Elevator Model w/ Java Exceptions / Zoo animal behaviors

Due: Monday, October 1st 11:59 Eastern time

Grading: 15%

Last Modified: Wednesday, 25-Aug-2018 11:39:58 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.]

Your submission should include the following:

* An executable .jar file.
* A README.txt file describing how to run the program from the .jar file.
* JavaDoc for each class placed in a directory inside the jar named ‘docs’.
* A zip file named <lastName>\_<firstName>\_hw2.zip containing the above.

# Part 1.

# This homework extends Homework 1 in the following ways:

# All of the following classes, as well as the refactored [or recreated] Elevator.java class from HW 1, should be put into a package named **cscie55.hw2**

# There are three new types [classes] of object.

# Building

# Floor

# ElevatorFullException

## **Overview**

# The Elevator from last assignment that will be refactored and reused. Be sure the package name matched the hw2 specification.

# The Elevator moves passengers between Floors. The Elevator and all of the Floors are part of a Building.

# The Elevator will limit the number of passengers it can carry, throwing an ElevatorFullException if an attempt is made to board passengers past this limit.

# Instead of writing a main(String[] args) method, your implementation will be tested by a JUnit test that you can download from these links:

* http://www.java2s.com/Code/Jar/j/Downloadjunit411jar.htm
* http://www.java2s.com/Code/Jar/h/Downloadhamcrestcore13jar.htm

# While it is recommended that you extend your Homework 1 code, that is not required, and you may prefer to start Homework 2 from scratch. Either approach is fine. The requirements given below are written as an extension of the Homework 1 requirements.

## **Tasks**

# Code organization

# The Elevator, Floor and Building classes all go into the package **cscie55.hw2.** If you define any additional classes, they should also go into this package (and should not be declared public).

# **The Elevator class**

# The Elevator moves exactly as it did last time, stopping at Floors to disembark passengers, and to pick up new passengers. But now, if a passenger tries to board the Elevator when it is already loaded to capacity, an ElevatorFullException will be thrown. That should prevent the passenger from boarding. [It's up to move to deal with the exception, rather that throwing it up the stack.] The Elevator will pick up the passengers that could not be accommodated when it next visits the Floor.

# Starting with Elevator from homework 1 or from scratch, add the following the **public** methods and fields**\*** to the Elevator class:

# int getCurrentFloor() : Return the Elevator's current floor number. I.e., this is the number of the floor reached by the last call to Elevator.move().

# int getPassengers(): Return the number of passengers currently on the Elevator.

# void boardPassenger(int destinationFloorNumber) ***throws ElevatorFullException***: Board a passenger who wants to ride to the indicated floor. Note that this method boards a single passenger and may throw an ElevatorFullException.

# Elevator.CAPACITY is a **final static** field that stores the number of passengers that the Elevator can accommodate. Set it to 10.

# Replace the Elevator() constructor by Elevator(Building building). (Hint: You may find it useful to record the building in a new field of Elevator.)

# **\***[Note that the term field is used here as synonymous with the terms "instance variable" and "state variable".]

# **The Floor class**

# This is a new class, representing one of the floors that the Elevator can visit. It should have the following public methods:

# int getPassengersWaiting(): Returns the number of passengers on the Floor who are waiting for the Elevator.

# void waitForElevator() : Called when a passenger on the Floor wants to wait for the Elevator. Calling this should cause the Elevator to stop the next time it moves to the Floor. NB: For this homework assume that passengers waiting for the Elevator on floors 2 and above should all be boarded as going to the first floor. (We'll drop this assumption in the next homework.)

# Floor(Building building, int floorNumber): This is the Floor constructor.

# You will need to decide what fields the Floor class should have. Note that to meet the first two requirements a Floor object must retain a piece of state that records the number of passengers it has waiting for an Elevator.

# **The Building class**

# This is a new class, which keeps track of one Elevator and multiple Floors. It provides access to these objects for tests. Building has the following public methods and fields.

# Elevator getElevator() : Returns the building's Elevator

# Floor getFloor(int floorNumber): Returns the Floor object for the given floor number.

# FLOORS: A **static final** field storing the number of floors in the building. Set the value to 7.

# Building(): The Building constructor creates an Elevator, and one floor for each floor number. (Move the variable currentFloor that was in the Elevator class in homework 1 to the Building class.)

# These specifications are requirements. As such, they specify things that must be done in any correct implementation. You are, of course, free to add other methods (as long as they are not public) and any fields (that should probably not be public).

# **The ElevatorFullException class**

# This needs to be a public class, extending java.lang.Exception. ElevatorFullException, like any other exception, must be thrown when something out of the ordinary happens. Because we do not want this exception to end execution of your program, some other code must catch ElevatorFullException and handle the situation, (i.e., that the elevator cannot board a passenger when it is full).

# In this case, Elevator.boardPassenger(int destinationFloorNumber) throws ElevatorFullException when the Elevator is already at capacity.

# Elevator.boardPassenger(int destinationFloorNumber) is called when the Elevator stops on a Floor, some passengers leave, and the passengers waiting on that floor try to board. This code must catch the exception, e.g.

# try {

# ...

# elevator.boardPassenger(...); // Method may throw ElevatorFullException

# // The passenger boarded successfully

# ...

# } catch (ElevatorFullException efe) { // catch exception

# // handle exception

# // The passenger was unable to board because the elevator is full.

# ...

# }

## **Design Considerations and Gotchas**

# It should be clear from the requirements that there must be communication between Floor objects and Elevator objects. This is in order that the Elevator knows when there are passengers waiting to board when it arrives at a Floor and that the Elevator can tell its current Floor when it has boarded one of the waiting passengers. In other words, your model must maintain data integrity. There are different design approaches to achieve this. It's up to you which approach to take, so long as it works and so long as the code that implements it is easy to understand.

# Array sizing and indexing present potentially fatal gotchas. If you create an array to represent data for 7 Floors and you size the array to be 7 cells, its 0th element will map to floor number 1 and its 6th element will correspond to floor # 7. When you want the data for the ith floor it will be in cell i-1. The key to avoid indexing errors is careful attention to preserve this mapping.

## **Testing**

# In Homework 1, you demonstrated that your code was working by writing ‘test harness’ - a main(String[] args) method that created an Elevator, set up some passengers, and then called Elevator.move() and generated output to show that things were working properly.

# A more common approach to testing is to write self-testing code. You write a test that sets up initial conditions, moves the Elevator a few times, and then examines the state of the Elevator to see that all is as expected. A program like this has no output if all works well. However, if any state is not as expected, this results in an assertion failure, which is printed, and that test is then reported as failing.

# The most common self-testing framework for Java is JUnit.

# As before, submit the output you get when you run your program. If all goes well, there should be minimal output, e.g.

# JUnit version 4.11

# ....

# Time: 0.006

# OK (4 tests)

# If you don't see something like this, and there is output indicating tests failing, and containing stack traces, then tests are failing. Figure out what's going wrong and keep debugging until you get a clean run.

# **Getting started with JUnit**

# For general background on JUnit see page “Unit Testing with Junit” linked to the assignment

# The first thing to do is to download the test code. [Note: You may not modify this code. If it fails to compile, your implementation has violated the requirements. If any tests fail, there are errors in your code.] The class of this Java code is cscie55.hw2test.ElevatorTest. Note that the package is cscie55.hw2test.

In order for Java to find the classes it is to run, it employs what is called the ‘classpath’. The classpath is a like a url: it is a location pointer indicating where the needed resources can be located on the computer that hosts the program [and the resources needed]

You will need to download two jar files, and include them on your classpath. The two jar files are:

* http://www.java2s.com/Code/Jar/j/Downloadjunit411jar.htm
* http://www.java2s.com/Code/Jar/h/Downloadhamcrestcore13jar.htm

# Your code should be structured something like this:

src/

lib/

junit-4.11.jar

hamcrest-core-1.3.jar

cscie55/

hw2/

all HW2 classes

hw2test/

ElevatorTest.java

I recommend a directory beneath your *Java Source Root* [which above is named ‘src’] named ‘lib’. Your classpath must include both these jar files as well as directions to the classes in your src and test directories. (For jar files, your classpath must include each jar file explicitly. If you just include a directory containing the jar files, then the jar files themselves will not be found.) This makes four separate paths in your classpath, one each explicitly for the 2 jar files, and the 2 separate paths to locate the class files in the two packages, hw2 and hw2test.

Because you are running a junit test, the class you run is **org.junit.runner.JUnitCore.** **That is, the main method is in JUnitCore**. There is one argument, the name of the test: **cscie55.hw2test.ElevatorTest**

For local [on your machine] running of tests, you can create a temporary environment variable named CLASSPATH as follows:

CLASSPATH=<path\_to\_lib\_directory>/junit-4.11.jar: <path\_to\_lib\_directory>/ hamcrest-core-1.3.jar: <path\_to\_Java\_Source\_Root\_directory>/hw2: <path\_to\_Java\_Source\_Root\_directory>/hw2test

On my Linux machine, the command to run the test code is as follows:

**java -classpath $CLASSPATH org.junit.runner.JUnitCore cscie55.hw2test.ElevatorTest**

You will need to adjust this for your environment and the location of the homework classes on your machine if they differ from above.

Further design considerations. Homework 2 is a big expansion of the simulation model. Structurally there are three new classes. In terms of data, the big change is the presence of passengers waiting on floors. The big change in behavior is that Elevator.move() now must take into account the need to board waiting passengers when it arrives at a Floor, that is, if there are any such waiting passengers, even if there is no one on board destined for the new current floor. And when it stops there it must board only as many passengers as it can accommodate, meaning that some passengers will be left waiting on the Floor, until the next time the Elevator arrives there.

How does a Floor acquire waiting passengers? That's what waitForElevator() does. Further, the Floor must keep track of how many passengers are waiting, and be prepared to tell the Elevator when it arrives there. Notice this means Elevator and Floor must communicate.

As the Elevator boards a passenger waiting on a Floor, how does the Floor learn to decrement the count of waiting passengers? We will need a method by which an Elevator tells a Floor it has taken one of its waiting passengers. Does this method have to be public? The homework states that no further public methods may be added. You don't have to make a method public for other classes in the same package to have access: absent an explicit qualifier public private protected a member is package protected, i.e. accessible to every class in the same package

# Part 2.

Refactor [move] your animal classes from HW 1 into a directory structure as follows:

<whatever\_you\_like>/

src/

main/

java/

cscie55/

hw2/

zoo/

animals/

Lion.java // animal classes go here

. . .

Revise the package names of your existing classes to reflect the relocation as follows: ***cscie55.hw2.zoo.animals***

* Create at least one Constructor for each class and initialize any variables in the constructor.

**Add 3** **methods** to each of your Homework 1 zoo animal classes.

* One should be named '**eat**' and it should **return a String** that **takes the 'favoriteFoods' data from assignment 1 and displays it as proof that it is eating wel**l. "Yum! Salmon" might be appropriate for a grizzly bear, for example.
* Another method should be '**speak**'. This method too should **return a String** indicating what sound the animal makes. [What does a lion say?]
* The third should be '**play**'. This method when called should **return a String** representing a sound that your animals make while having fun. This one is pretty open. Make something up!