Lab One: ABC Corporation's Elevator

Allison Juliette Snipes

Department of Computer Science, Johns Hopkins University

EN 605.203.81: Data Structures

Dr. Eleanor Chlan

Dr. Richard Cost

Dr. Scott Almes

Dr. Farhana Shah

October 06, 2020

Abstract

The purpose of lab one was to serve as a means to expose the student to stacks. In this lab, I had the opportunity to design and implement data structures, utilize input and output libraries to read and write to text files, and utilize getter and setter methods. Although there is no direct interaction with the user, via Scanner methods, the program relies on the user to enter the proper terminal commands to read input data from a text file. An alternate class, which holds the stack in a separate Java file, will serve as the means to simulate an elevator.

Program Design

My program was coded in the Eclipse Integrated Development Environment (IDE) using the Java 8 Standard Edition Development Kit (JDK). As per the project guidelines, dictated by Dr. Chlan and Dr. Cost, I did not utilize Graphical User Interfaces (GUIs) nor application builders to develop my project. My program's design is very simple and rudimentary, I utilized: one main class, a second class to create a passenger object via getter and setter methods, another class to create my stack, and finally various methods to control the elevator and passengers' actions.

It is important to note that the application's code begins by importing numerous Java utility and input/output libraries in the ABCElevator.java file. This main class imports the SnipesStack, which dictates the capacity of riders and the maximum floors of the building. It is important to note that the stack class utilizes a Last In First Out (LIFO) approach to load the passengers into the elevator (who are actually objects in the code). We have to utilize a stack that uses the LIFO principle as the last person in the elevator is constantly exiting and entering to allow for other riders behind them to exit. The main class utilizes two exceptions for error handling. These exceptions control against improper actions the user might accidentally do: Illegal Argument Exception, and IOException. Both of these exceptions provide a simple message to the user via the console, should they make a mistake with working the with

program. The Illegal Argument Exception controls against the user not providing the proper runtime arguments, and the IOException throws an error if the file that contains the passenger data cannot be read. These exceptions were utilized within an if statement and try-catch method respectively. Line 54 is responsible for grabbing the data file that is necessary to run the application (as it is a runtime argument needed for main to execute). Within the main class, I used a Buffered Reader to read text from the character input stream provided in the input text file. In order to write my program's output, I utilized both a Print Writer and File Writer to write the passenger objects via a text output stream to a file object.

My second class, called Passenger, creates the passenger object. It is quite a simple class that holds important information regarding each person, such as: their name, the floor they entered, and their desired floor to exit. This class utilizes getter and setter methods that stores data from the data text file given for the project. I stored critical data elements of the elevator's passengers in this fashion as my File Writer and Print Writer uses objects to write to an output file. Plus, this approach bypasses the need to hardcode each passenger (and data about them) into my program.

My passengerOff method serves to manipulate the stack of passengers based on their destination floor. I had other classes and methods that manipulated the stack, however it soon became very complicated and difficult to navigate through the complicated logic. This method controls for the number of times a person exits, when a person arrives at their destination floor, lists who is still on the elevator, and controls when the program should not pop the stack.

Finally at the end of my code I converted my passenger list, which was originally an object, back into a string. This method allows for the program to parse the string into the necessary data components needed for each passenger such as: their name, the floor they entered on, and the floor they will exit on. This information is then created as an object which will be returned in the method.

Analysis

I spent a large amount of time planning this project on paper before I started coding it in Eclipse. I went through countless of alternative approaches, which made sense to me, only to change my final design while coding (due to receiving runtime errors, logic errors, and my program not compiling) . On paper I had six different methods that I wanted to utilize to manipulate my stack: a method for when a person boarded the elevator, a method for when they exited the elevator, a method that tracked when a passenger temporarily exited, a method that tracked who took the stairs, a method that tracked when the elevator was either full or empty. In the long run, I abandoned all of the methods except for when a passenger wished to exit the elevator. It is important to note, that my original program did not utilize a separate file that contained my stack. It was after I received guidance on why I was experiencing certain errors that I learned that utilizing a separate file for my stack was the better approach.

Secondly, I did not need to use try-catch methods in my code. However, since there would be minimal input from the user—it was best to provide marginal error handling. This approach would assist me program in two major ways: it will teach me good coding habits and/or error handling practices, and demonstrates good troubleshooting skills to assist the user. When in doubt, programmers should aim to have the users' experience in mind (by keeping their applications simple, and providing user feedback when necessary).

While my program does not run properly, I am able to make some assumptions about the elevator's efficiency as it regards to time and space. Clearly being limited to a LIFO approach, when riding an elevator, is not efficient at all. The constant in-and-out would be frustrating to the passengers. The elevator's entry should be large enough to accommodate multiple people entering and exiting at any given time. I would estimate that the cost to run ABC Corporation's defective elevator would be at least

n^3; however this estimate is quite modest given that I am having some trouble with executing my code. I would recommend that ABC Company invests in upgrading the functioning elevator first to allow for a larger entry and exit for its passengers, and upgrade its broken elevator to do the same.

Reflection and Enhancements

While my project has complications with executing, I was still able to provide enhancements to my project as per the dictated guidelines. Within my code, I performed exception handling to provide basic feedback to the user (if they performed incorrect actions). I utilized try-catch methods to read from the program's data files, and provided the framework to write to an analysis file. In order to maintain version control, I utilized Github as a means to save my progress or revert to a previous versions of my project (should I encounter an irrevocable error). Lastly, within my SnipesStack, I made sure that the class had a maximum capacity of 5 riders and there was maximum of 5 floors in the building. This was obtained by setting both variables, fullCapacity and maxFloor, as final.

I learned a lot while coding this project. I was able to find my weaknesses, and identify concepts that I need to study again. Before this project, I did not know how to confidently plan or execute an assignment that required multiple classes. I am grateful that I am now able to grasp: when I should create a separate class to perform functions, how to import separate classes, utilize and import stacks, and how to better troubleshoot my code.

If I could do anything different, I would remove the restriction of not being able to import stacks. While, it ensures that the student understands the concepts of stacks—it makes the project very difficult for beginner programmers! I would do many things different in this project, however the most important thing I would do would be: to slow down and better understand the concepts of stacks, what to do should my program encounter errors when executing stacks, and ask for clarification and additional assistance when necessary.

Conclusion

In closing, this was not a trivial assignment! This project highlighted important concepts discussed in lecture by allowing the student to gain hands on experience in an Object Orientated Programming environment. As previously mentioned, my program did not properly execute due to logic errors. I was not able to resolve where the logic error was (even after receiving assistance from professors). While this assignment was challenging, I remain hopeful to resolve my error and resubmit the project at a later date.