Queue Simulator

Programming Techniques – Homework 1

2ND Year, 2ND Semester, Group 30422

Student Name: Pop Elisa Teea

Technical University of Cluj-Napoca

Computer Science and Automation Department

Problem Definition

**Task: The application is meant to simulate a real-life situation of queues and how a series of clients are arriving for a service, entering queues, waiting, being served and finally leaving the queue. It tracks the time the customers spend waiting in queues and outputs the average waiting time.**

The stated problem can be reached by solving the following sub-problems:

* Implementing each queue using threads
* Choosing the shortest queue to place a new customer at
* Preparing the data for the simulated results

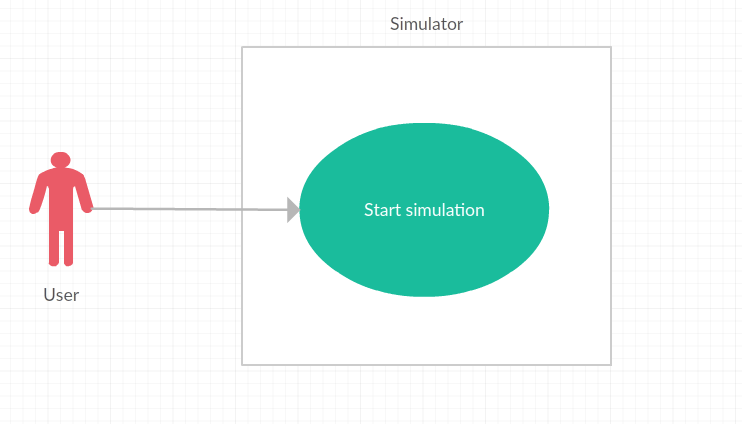
Project specifications:

* Programming language used: Java
* Project SDK: 1.8 ( java version “1.8.0\_191”)
* Project language level: 8 – Lambdas, type annotations, etc.
* GUI: JavaFX
* Program used: IntelliJ Idea
* Git : Bitbucket

Problem Analysis

Use-case diagrams

This application has the intention to visually simulate a queue and give the user the ability to analyze it based in the introduced inputs



Actors: the User   
Use case title: Start the simulation  
Short summary: after introducing the data, the simulation is started  
Preconditions: The user has to introduce valid data for all the given fields ( arrival time interval for customers, service time interval for customers, number of queues, simulation time and specified time which computes values – average waiting time, average service time - for the given interval)

Design and Programming

The project is structured using packages, separating the UI-based classes from the back-end. The back-end classes are the following: : WorkingQueue, Scheduler, Customer, Executor, whereas the UI based ones are Model, ViewProgress, ViewMain, and Utils.

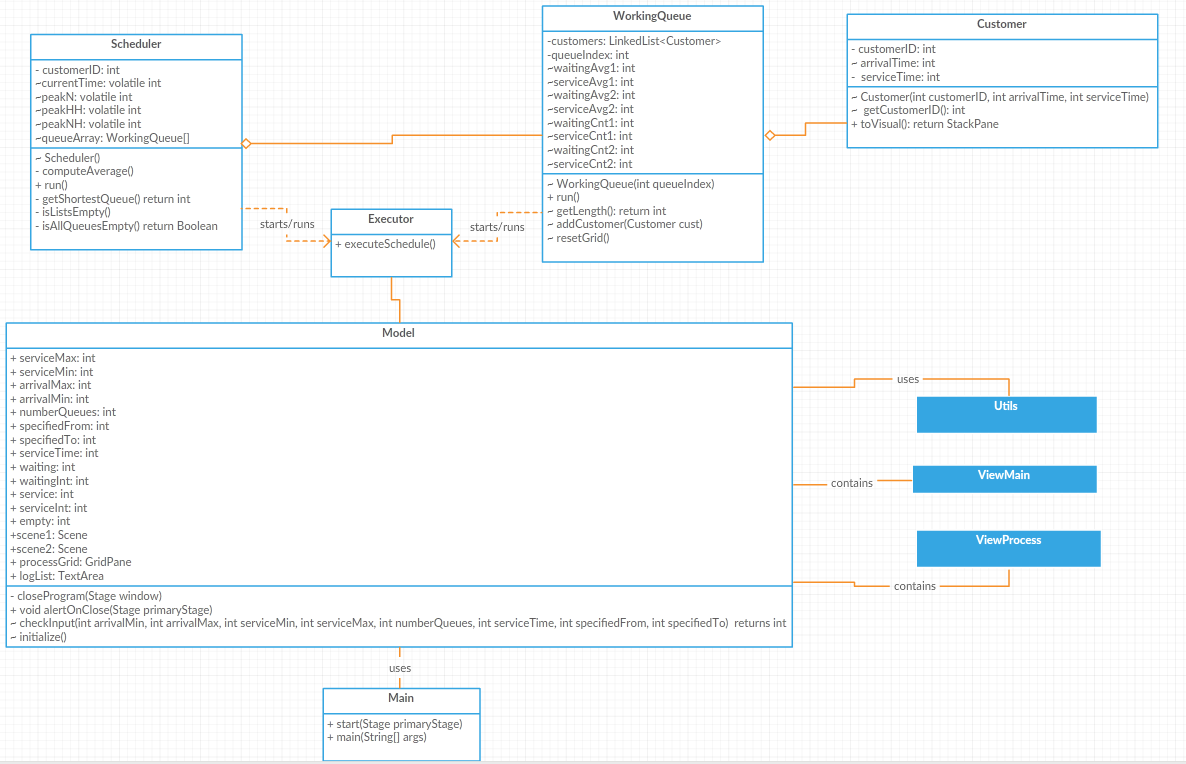
entities

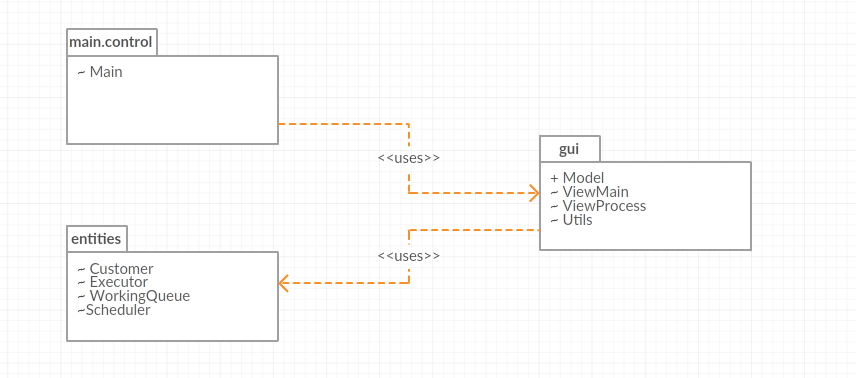
* It contains the Customer class, forming the WorkingQueue class. Another class is Scheduler, which generates Customers for WorkingQueue to process. And finally, Executor starts the threads of Scheduler and WokringQueue which simulate the queue.

gui

* As the name suggests, this package covers all the UI-related elements.
* Due to the fact that the MVC architecture is difficult and counter-intuitive to implement with JavaFX because usually we’re working with the Scene Builder so the UI contains a Controller and a .fxml file which is the view. This is the reason why why the application only has a Model (Model-Controller) and a View.
* For this implementation, I used two views due to the fact that the scenes change. ViewMain is used for the main screen, where you introduce the data to be simulated. ViewProcess displays the real-time simulation using the data introduced in the first scene.

main.control

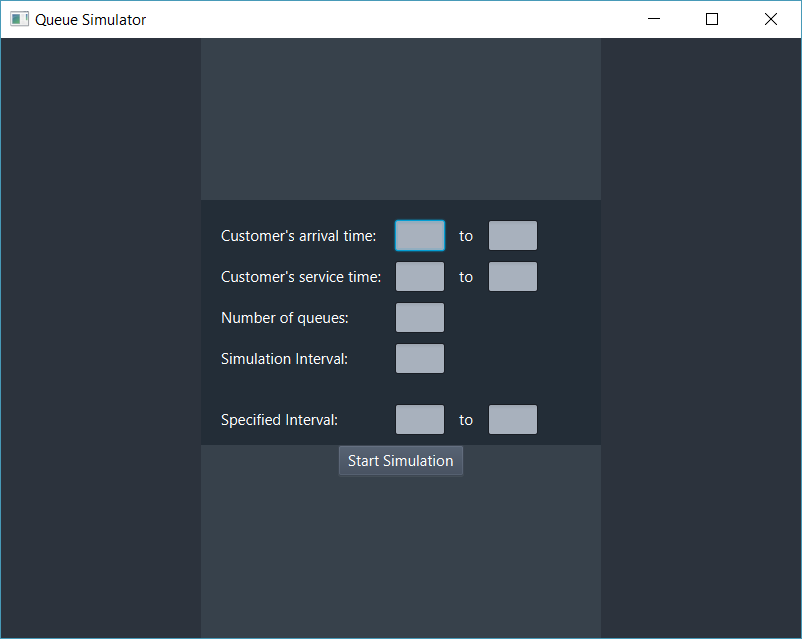
* This package holds a sole class: Main class. This is used for setting the stage and starting the application.



Implementation

The user interface

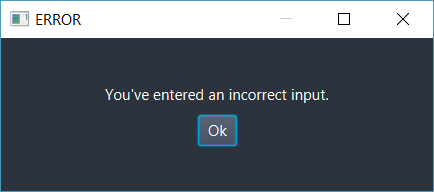
The main scene which is set in ViewMain:



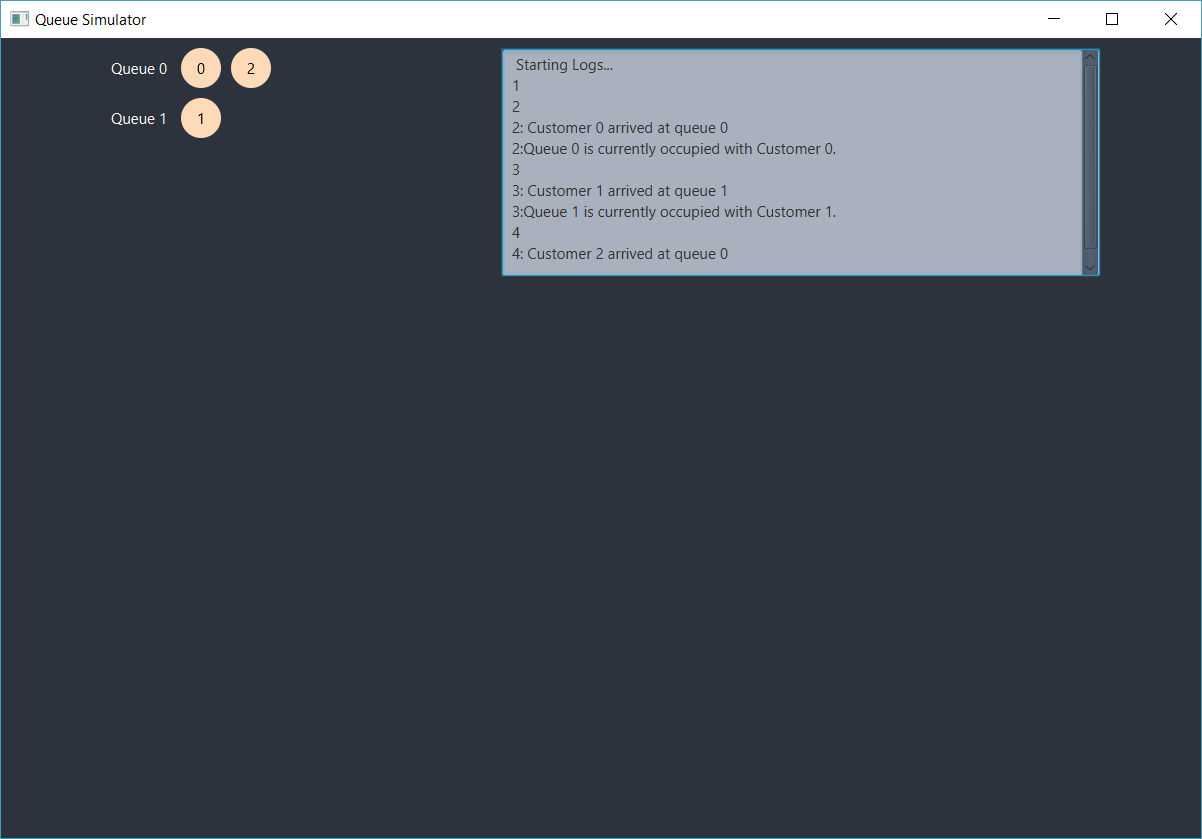
It’s a relatively simple scene, having used one GridPane to place the fields: 8 Labels, 8 TextFields and a Button which sets the next scene given that the inputs are correct.

* Customer’s arrival time: is the interval of time in which customers will appear in the simulation.
* Customer’s service time: is the interval of time in which the time a customer spends at the “Checkout” ( before leaving the queue ) is generated randomly.
* Number of queues: represents the number of queues that will be available for the generated customers.
* Simulation interval: indicates after how many seconds the simulation will stop generating customers.
* Specified interval: indicates the interval from which another set of data will be generated

The only requirement for starting the simulation is having non-empty fields. If the condition is not met, an error will occur:



Given that the inputs are correct, the simulation will start immediately after pressing the button. The second scene presented below is the ViewProgress:



As the simulation starts immediately, a part of the simulation is shown. The log area in the GUI translates in text what is shown visually, on the screen. When the simulation ends, information about waiting time, service time, empty time of the first three queues, peak time of the process and the average of service time and waiting time of the first three queues in the given specified interval from the first scene is written.

The Model

The model not only transmits information between the GUI and the entity package, but also sets the scenes of the user interface and communicates with the Main class.  
 It has several static fields that ties the GUI to the back-end functions, out of which there are *serviceMax*, *serviceMin*, *arrivalMax*, *arrivalMin*, *numberQueues*, *specifiedFrom*, *specifiedTo*, *serviceTime* which contain data introduced in the main scene. Another set of static integers is: *waiting*, *waitingInt*, *service*, *serviceInt*, *empty* which are used to compute the analysis data of the queue which will be shown after the simulation has terminated. The most notable static declaration of this class is logList, which is a TextArea containing the log displayed on the second scene.

Methods:

- closeProgram(Stage window) and alertOnClose(Stage primaryStage) : after pressing the quit button in the up right corner of the application, a message will appear asking if you really want to close the program. This is mainly used to prevent the user from accidentally closing the application and losing the process.

--checkInput(int arrivalMin, int arrivalMax, int serviceMin, int serviceMax, int numberQueues, int serviceTime, int specifiedFrom, int specifiedTo): as the name suggests, it checks if the input introduced in the first scene is correct and updates them accordingly in the first set of static integers mentioned above. The metod returns 1 if the input is correct and 0 otherwise.

- initialize(): calls the Executor’s only method in order to start the simulation. This is placed in the method because a GUI class has to call it once a button is pressed.

The Customer

It is used to represent customers and it only has getters. A mentionable getter is the one returning a visual element in order to be represented on the screen during the simulation.

The WorkingQueue

Is the proper queue of this application. It has a LinkedList containing customers. The reason why LinkedList was preferred over BlockingQueue is the ability to access every element without having to remove them. This is used for resetting the queues when a new operation is made on it.

Notable methods:

- run(): is used in the thread pool which will be presented shortly after this class. It sets information about current operation in the log and computed the queue analysis results. After customers are added in the LinkedList by Scheduler, the first customer is taken and the threads waits until its service time runs out. Afterwards, another message is shown in the log and the customer is deleted from this queue’s list.

- resetGrid() deletes the queues’ grid and places all the elements that are currently present in queues.

The Scheduler

It contains an array of WorkingQueues and manages the Customers wich are randomly generated each second.

Methods:

- isAllQueuesEmpty() : checks if the Scheduler’s array of WorkingQueues contains only empty queues. If so, this method is used to stop the scheduler’s thread.

- getShortestQueue() : gets the queue which has the least customers in it. This is used in the Scheduler thread to check at which queue a newly generated customer is to be placed.

- isListsEmpty() : checks if any of the first three queues are empty. It is used for computing the average empty time.

- run() : updates the log with every passing second and has a set chance to generate a cunstomer each second. If a customer is to be generated, it is placed accordingly at the most beneficial queue for it. It also resets the queue grid because a new customer is shown. After it has no more customers to generate, it simply updates the log with the current second and waits for the queues to be done with the customers.

- computeAverage() : is used to compute the final results that will be shown in the log after the whole simulation is finished.

The Executor

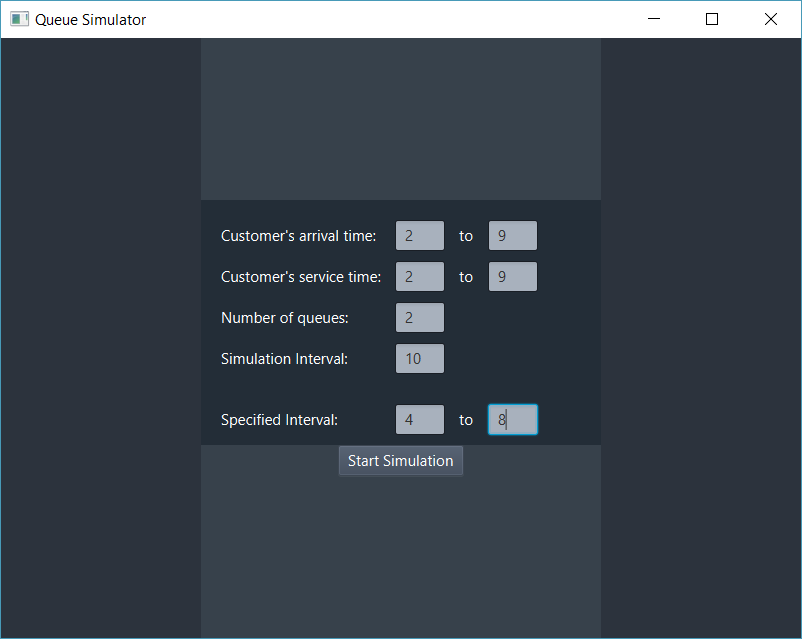
It contains a sole static method, executeSchedule(). Since Schedule and WorkingQueue implements Runnable, they are unable to change anything in the GUI elements. In order to be able to change it, the runnables must be placed in a thread, and Thred.start() has to be called. This makes the runnable safely call Platform.runLater() which will update the GUI accordingly. Also, ObservableObject, which is the main data structure used for visual elements in JavaFX is not thread safe. So regular data structures were used instead (LinkedList, String).

Results

Since the program contains threads and random elements, it is too complicated to test or debug. Instead, the log located in the second scene that starts at the same time with the simulation will help the users see what actually happens.

For the following simulation, the following data will be placed in the main screen:

I will screenshot a few sequences of the simulation and provide some of the missing logs:

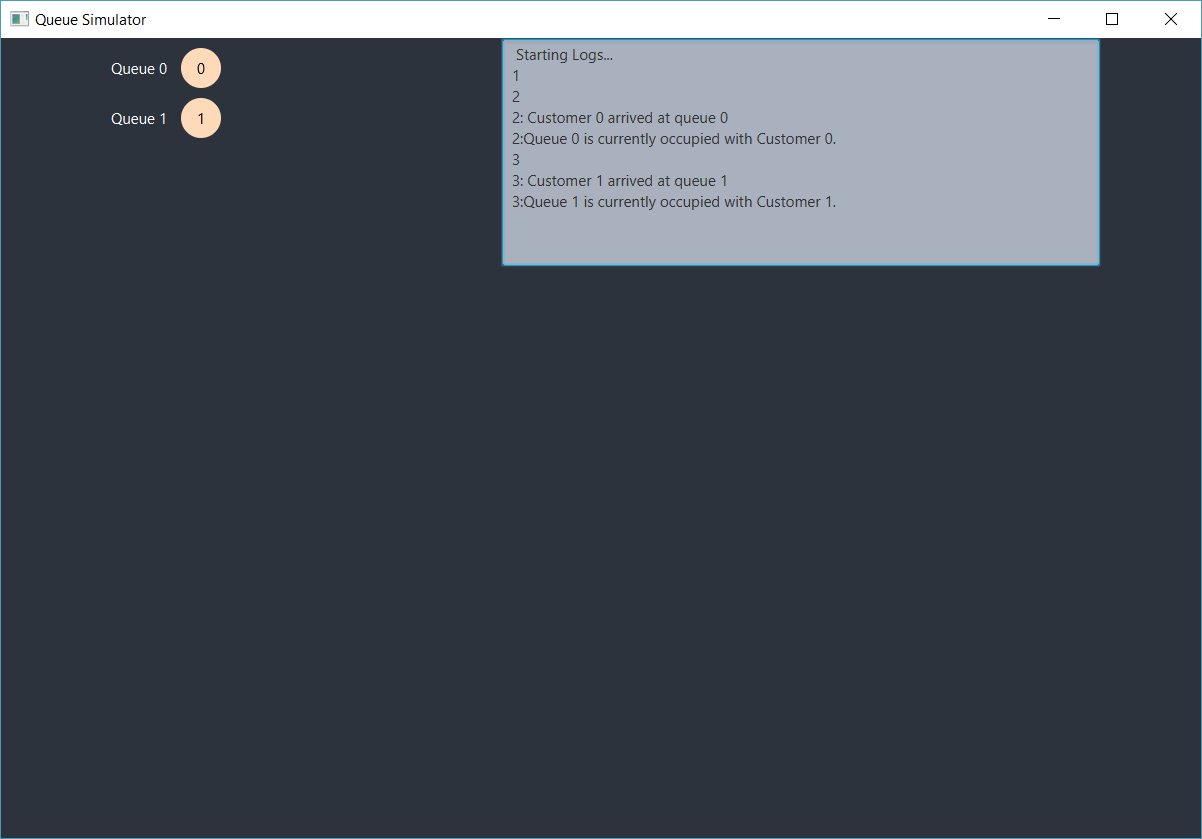


. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .. . . . . .. . . . . . . . . . . . .. . . . . . . . .

. . . . .. . . . . . . . . .. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .. .. . . .. . . . . . . . . . . . . . .

.



4

4: Customer 2 arrived at queue 0

4:Queue 0 finished working with Customer 0.

5:Queue 0 is currently occupied with Customer 2.

5:Queue 1 finished working with Customer 1.

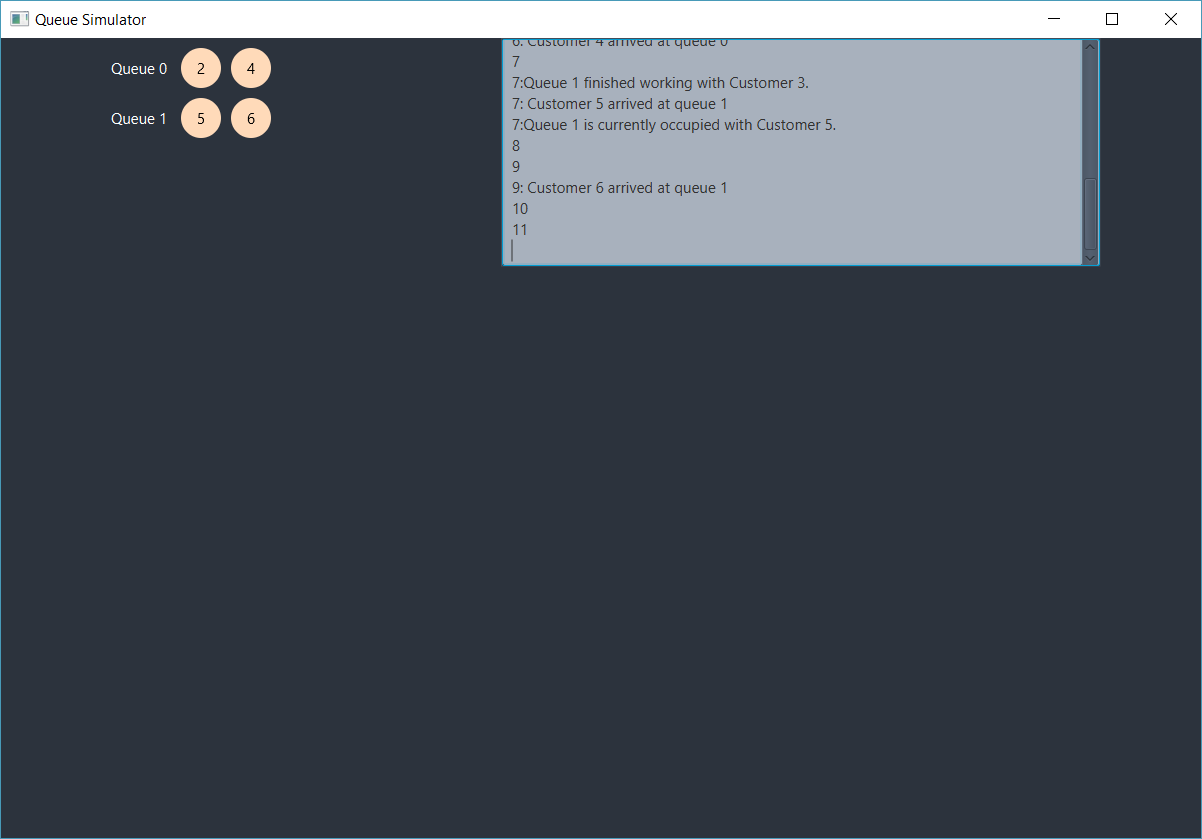
5

5: Customer 3 arrived at queue 1

5:Queue 1 is currently occupied with Customer 3.

6

6: Customer 4 arrived at queue 0

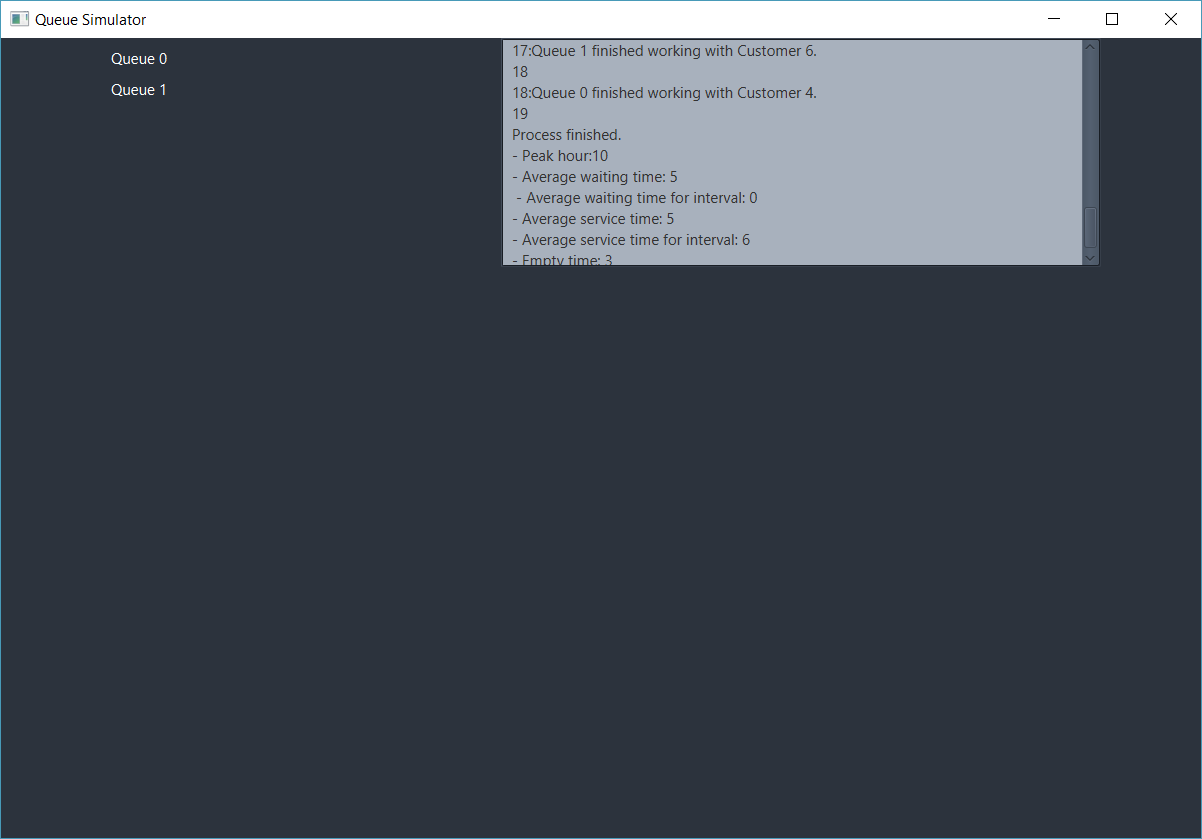


11:Queue 0 finished working with Customer 2.

11:Queue 0 is currently occupied with Customer 4.

11:Queue 1 finished working with Customer 5.

11:Queue 1 is currently occupied with Customer 6. // (12, 13 … until 17)



Conclusions

After finishing this project, I can say I’ve learned a very important aspect about Java and OOP overall, and that is threads and multi threading. Another important aspect was learning how to change the GUI from multiple threads, which is counter intuitive for someone new to the concept of “threads”. JavaFX itself uses a single thread for the UI, therefore the interface could only be changed from a FX thred. This makes the FX thread ( Platform.runLater() ) very easy to flood and this results in the stage to freeze/ stop responding. Runnables can’t call the FX Thread, so the only option is to place the runnables in threads and call .start() instead of .run().

Further improvements for this program can be:   
- Logs that will help more with debugging   
- Selecting queues to calculate the data for average value at will. This application only computer the values for the first three threads.  
- Computed data showing in real time on the UI and a “Pause” and “Resume” button to check the data needed in the exact second we pause.

References

<https://creately.com>

<https://stackoverflow.com/>

<https://www.vogella.com/tutorials/JavaConcurrency/article.html>

<https://www.tutorialspoint.com/java/java_multithreading.htm>

<https://www.geeksforgeeks.org/multithreading-in-java/>

<https://www.developer.com/java/data/multithreading-in-javafx.html>

Appendix

1. Cover …………………………………………………………………….. Page 1
2. Problem Definition ……………………………………………………... Page 2
3. Problem Analysis... ……………………………………………………... Page 2
4. Design and Programming ... ……………………………………………. Page 3

* entities ……………….…………………………………………... Page 3
* gui ………………………………………………………………... Page 3
* main.control ….…………………………………………………... Page 4
* UML. ……………………………………………………………… Page 4

1. Implementation…………………………………………………………… Page 5
   * The user interface ……………………………………………...… Page 5
   * The Model ………………………………………………………... Page 7
   * The Customer …………………………………………………..... Page 7
   * The WokringQueue ………………………………………………. Page 7
   * The Scheduler ……………………………………………………. Page 8
   * The Executor ……………………………………………………... Page 8
2. Results ………………………………………………………………...….. Page 9
3. Conclusions ………………………………………………..…………….. Page 12
4. References ………………………………………………………………... Page 12
5. Appendix ……………………………………………………………...….. Page 12