CSCI 1933 Lab 11

Discrete Event Simulation

Lab Rules

You may work individually or with *one* partner in lab. We suggest that you have a TA evaluate your progress on each milestone before you move onto the next. The labs are designed to be completable by the end of lab. If you are unable to complete all of the milestones by the end of lab, you have **until the last office hours on Monday** to get those checked off by **any** of the course TAs. We suggest you get your milestones checked off as soon as you complete them since Monday office hours tend to become crowded. If you worked with a partner, both of you must be present when getting checked off to receive credit. You will only receive credit for the milestones you have checked off by a TA. There is nothing to submit to Moodle for this lab.

Attendance

Per the syllabus, students with more than 3 unexcused lab absences over the course of the semester will automatically fail the course. The TAs will take attendance during lab; it is expected that students will be actively working on the lab material. *Your physical presence in lab is not sufficient to be marked as present for the purposes of attendance.*

Lab Overview

In this week's lab you will be creating a ferry simulator. The ferry you will be simulating is similar to a traditional ferry going between three islands. Here are some things to know before you get started:

- The ferry has one stop at each of three islands and goes in a circle (ex. 0,1,2,0 and so on). Islands will be represented as `int`s in 0,1,2.
- The ferry will stop at each island even if no one is at the island waiting for it.
- When a person gets on the ferry, they know what island they want to get off at. Each island is equally likely to be chosen, except they will never get off at the island they got on from.
- The ferry stops for one second for each person that gets on or off at that island.
- The ferry can hold 60 people at a time. People get off the ferry before people get on, but if the ferry is full they can't get on and must wait for it to come around again.
- The time between people arriving at a given island to start waiting for the ferry is n seconds, where n is a random number on the interval $[5, 15]$.

Download the supporting files from Moodle. You can refer to the comments at the tops of the files to see what each of them contains. You will be modifying the following files in this lab:

- `FerrySim.java`
- `Ferry.java`
- `PassengerArrivalEvent.java`
- `FerryEvent.java`

The rest of this page is intentionally left blank

1 Who wants to ride the ferry?

Background

In `FerrySim.java`, there are queues to keep track of passengers waiting on each island. In this section, you'll create events to add passengers to these queues. You'll be using `PQ` and implementing `IEvent`.

Events are actions that will run at some point in the future, relative to the current time of the agenda. Notice that `PQ` does not operate on `FerryEvent` objects, but on `IEvent` objects. `IEvent` is simply an interface that provides a `run()` method. This means we can put multiple types of events in the agenda, that have different run methods.

Passengers are represented by the Passenger class. We've already implemented this class for you; notice that each Passenger object has a pickupIsland (passed to the constructor) and a dropoffIsland (randomly chosen, different from pickupIsland).

> **Note:** `Passenger`s are saved in queues (`Q`), and `Event`s are saved in priority queues (`PQ`). This way, passengers exit their lines first-come-first-serve, and events are scheduled and occur according to their scheduled time.

Your tasks

Let's make an event that creates `Passenger`s for each island. Implement `public void run()` in the `PassengerArrivalEvent` class. This method will create a new `Passenger` with the correct pickupIsland and add it to the correct queue. It will also add a new `PassengerArrivalEvent` to our agenda for the same island at a time that is a random number between 5 and 15 (inclusive).

Now that we have our `PassengerArrivalEvent` class made, let's add it to main (in `FerrySim.java`). First, let's add a new `PassengerArrivalEvent` to our agenda for island 0.

If we hit run as it is, we don't have any useful output telling us that our code works. Let's edit our `PassengerArrivalEvent`'s `run()` function so that when we run our main, our output will be something similar to:

```
Passenger Event Island: 0, Current Time is: 0.0, Next Passenger in: 6
Passenger Event Island: 0, Current Time is: 6.0, Next Passenger in: 10
Passenger Event Island: 0, Current Time is: 16.0, Next Passenger in: 9
Passenger Event Island: 0, Current Time is: 25.0, Next Passenger in: 6
```

Also, add a print statement in `main` to print out the final length of the line at island 0, something

like:

```
Island 0, Number of Passengers in line: 105
```

Once you're sure that your `PassengerArrivalEvent` works, add a `PassengerArrivalEvent` for each island to the agenda, and print out the final length of each of their queues.

Note: You can reduce the amount of time in the `main` by modifying the `while` loop to allow for easier debugging.

Milestone 1: Show a TA the printout of your `main` with the messages from your PassengerArrivalEvents and the final length of each queue.

2 All Aboard CSCI1933

Having `Passengers` are great and all, but they still have no way to get from island to island! Let's fix that problem by completing our `Ferry` class. Here are the requirements for `Ferry`:

- The ferry can hold up to 60 `Passengers`. You can choose what kind of data structure you want to use to store these Passengers. Make sure that a max of 60 passengers can be stored and that **ONLY** `Passengers` can be stored. To protect our `Passengers` from anything dangerous (ex. Passengers leaving the ship midway or unwanted passengers boarding the ship at sea), our data structure should be `private`.
- `public boolean addPassenger(Passenger p)` – Adds a `Passenger` to this ferry. Returns true if the `Passenger` was added successfully.
- `public Passenger[] removePassengersAtIsland(int island)` – Removes all of the `Passenger` -s that are trying to get dropped off at a island and returns them in an array.
- `public boolean isFull()` – Returns true if ship is full.

Now our `Ferry` needs a way to move from island to island. We'll do this by creating a `FerryEvent`! The `FerryEvent` will have similarities to `PassengerArrivalEvent` but will work on a `Ferry` object instead of creating `Passengers`. Here are requirements for `FerryEvent`:

- `private int curIsland` – This is what island the `Ferry` that this Event is for is at.
- `private Ferry ship` – This is the Ferry object that we will remove and board passengers on.

- `public void run()` – This function will first remove all `Passengers` that want to get off at our current island, board new `Passengers`, and move our ferry to the next island.
- You will need a way to set `island` and `ship`.
- Ferries will drop off `Passengers` before they pick them up.
- Conditions for departure are: there are no passengers available for pickup OR this ferry is full.
- It takes 60 seconds to get from one island to another.
- To move our Ferry nicely (so that passengers that arrive while boarding is commencing can also board), we will need to play with scheduling times of this event. Essentially, we will schedule multiple FerryEvents during a ferry's stop at an island for the various phases: dropping off passengers, picking up passengers until a departure requirement is met, and traveling. See the FerryEvent.java class comments for an explanation of how to do this.
- It takes one second for each passenger to board or deboard.

We've written `Ferry` and `FerryEvent`, now it's time to use it. Edit the main method so that your `Ferry` will do its thing. Use some print statements as before to verify that your `Ferry` is working as intended.

Milestone 2: Once again, show a TA the printout of your `main`, this time with your FerryEvent print statements included.

3 Using the Simulation

Now that we have all of our pieces together, let's play around with the simulation a bit. First, try modifying the capacity of your Ferry. What if it can hold 40 Passengers? How about 80? What if we have one island that has two PassengerEvents, so Passengers arrive twice as often at that island? Try adding another Ferry to your simulation to see how that affects the lengths of the lines at the end of the simulation.

Milestone 3: To get checked off for this last step, demonstrate a simulation that has all lines < 10 in length at the end of the simulation, one simulation that has all lines > 200, and one simulation that has variation in the length of the lines (for example, island 2 consistently has about 2 times as many people as island 1).

The rest of this page is intentionally left blank