Project 6: Sophocles Software Call Center Sim

1 Task

Sophocles Software is launching soon and is building a customer support call center. They want you to build them a simulation that shows how call and support tech queueing will work. While you have a grand plan for a fancy simulation, this project will encompass only the first steps toward that goal.

Create a *realistic*, mocked customer list of 1,000 customers[1], including customer number, first name, last name, email address, and phone number. Store this data in a simple text file, in a format of your choosing.

Create a *realistic*, mocked tech list of 50 support techs, including id number, first name, last name, user name, and schedule. Assume techs have one day off during the week; come up with a good way record schedules in the file.

When the program runs, ask the user for simulation data including how many customers to put in the initial queue (e.g., 20). Also ask them the call interval to use (e.g., 1.5 seconds), and how many calls/iterations to simulate (e.g., 30) before terminating the simulation[2]. These interactions should be bulletproofed, i.e., it should do data type and range checking and re-prompt as necessary[3].

Then read in the customers and techs into respective ArrayLists. Pick the specified number of random customers (but don't include any single customer multiple times) and put them into a queue; this simulates calls waiting when the call center opens. Pick a random tech schedule (simulating a random day of the week), and add all techs working that schedule to a queue. Clearly display all information about the simulation as a "header," on a blank terminal screen, before details are shown per below instructions.

At the specified timer interval, pair the next queued customer with the next queued tech. Show that pairing as output on the terminal window. Then put a random customer into the queue (simulating an incoming call), then put the tech back into the tech queue.

Continue the simulation until the specified number of iterations is reached. At that point, tell the user that the simulation has ended.

(project handout continues…)

[1] Use Mockaroo or some other similar site, to generate realistic data. You may need to tweak this data, or generate additional data, in Excel.
[2] Note that the number of calls to simulate can be greater than the number of customers in the initial queue.
[3] This should use typical data validation loops in which the user is trapped in a loop until they provided data that complies with the stated request (i.e., is of the right type and in the specified range).

2 Code Implementation

2.1 Queue

Create a generic HashSetQueue class. It *must not* implement Java's Queue interface, but instead use linked list concepts/code[4] and implement the following methods: add, remove, peek, isEmpty, clear, and size. In the class, use a HashSet (Java's version is okay for the non-extra credit version) as part of the queue, ensuring that no object gets duplicated in the queue. Think carefully about what queue methods need HashSet integration; several of them do.

As always with generics, use Xlint to check for "unsafe" compiler warnings, which usually indicate misuse. And, as always, don't arbitrarily suppress warnings; fix them, instead[5]. As you did before with some other projects, ensure that these methods focus on *client data* and not *nodes*; client code should interact with techs and customer objects, not nodes themselves. You will of course need to implement and use nodes, but this isn't something that client code needs to be aware of.

2.2 Simulation and Main

Create a simulation class that accepts as constructor parameters the data the user enters in Main. In this case, it is okay for the simulation class to display data on the console. Main's job is to focus on UI and handoff to the simulation.

3 Extra Credit

Create and use your own HashSet (or use the author's, and make sure you understand every line of code). Override the hashCode method in Tech and Customer classes, leveraging the uniqueness of IDs.

4 Design Documentation

Submit the usual diagrams (Class and Object).

5 Code Implementation

5.1 Additional Building Blocks You'll Need

- Queues
- Timers
- Mocked data
- ArrayList
- HashSet (okay to use Java's for this)

5.2 Style

Follow the Course Style Guide. You'll lose points on every assignment if you fail to do so.

[4] Don't include a linked list class here; adapt your linked list code into a newly created Queue class.
[5] Remember that one issue at an early/fundamental point can cause a cascade of other errors; fix from the top.

6 Testing

Since you're writing a generic HashSetQueue, you'll need to fully test that class. Otherwise, no other testing should be necessary as other classes are either (1) well tested from previous projects, or (2) too simple to merit the investment at this point.

7 Submitting Your Work

Follow the course's Submission Guide when submitting your project.

8 Grading Matrix and Points Values

Area	**Value**	**Evaluation**
Design docs	10%	Did you submit initial design documents, and were they in reasonable shape to begin coding? Did you submit final design documents, and were they a good representation of the final version of the project?
Realistic mocked data	5%	Was all data realistic, including customer numbers and tech employee numbers?
ArrayList usage	5%	Was your own ArrayList used, and used well within the project?
Generic queue implemented/used	25%	Was a generic queue created and implemented? Did it have only the methods a queue should offer? Did any unchecked warnings remaining?
HashSet	5%	Was the hash set used appropriately and properly? Was it well integrated with queue methods?
Customer/Tech classes	5%	Were these classes created properly?
Timer usage	15%	Was the timer properly created and used?
Main, user interaction, display	15%	Was it possible to run "happy path" simulations? Were the inputs bulletproofed so that no bad data types or ranges would cause exceptions?
Testing HashSetQueue class	5%	Was a test class created for the queue class? Were all possible methods tested? Was all state tested?
Style/internal documentation	10%	Was the -Xdoclint run clean? Did you use JavaDoc notation, and use it properly? Were other elements of style (including the Style Guide) followed?
Extra credit: HashSet/hashCode	2%	Is a non-Java version of HashSet used, and used properly? Was hashCode overridden in the Tech and Customer classes, per instructions?
Total	**102%**	

Code that does not compile or run won't be graded