Lab 3: Simulation 3

- Deadline: 13 September 2022, Tuesday, 23:59
- Marks: 4%

Prerequisite:

- Completed Lab 2
- Caught up to Unit 25 of Lecture Notes
- Familiar with CS2030S Java style guide

Goal

This is a continuation of Lab 2. Lab 3 changes some of the requirements of Lab 2 and adds some new things to the world that we are simulating. The goal is to demonstrate that, when OO-principles are applied properly, we can adapt our code to changes in the requirement with less effort.

Lab 3 also involves writing your own generic classes.

Queueing at The Counters

Despite adding an entrance queue, the shop is still losing customers. With CNY coming, the shop decided to rearrange the layout and make some space for queues at the counters. With that, customers can now wait at individual counters.

In this lab, we will modify the simulation to add a counter queue to each counter. If all the counters are busy when a customer arrives, the customer will join a queue and wait. When a counter becomes available, the customer at the front of the queue will proceed to the counter for service. Each counter queue has a maximum queue length of L. If every counter queue has reached its maximum capacity of L, then an arriving customer has to wait at the entrance queue.

Just like Lab 2, the entrance queue has a maximum queue length of m. If there are already m customers waiting in the entrance queue, an arriving customer will be turned away.

With the addition of counters, there is a change to the customer behavior in choosing which counter to join:

- If more than one counter available, a customer will go to the counter with the smallest id (just like Lab 2)
- If none of the counters is available, then the customer will join the counter with the shortest queue. If there are two counters with the same queue length, we break ties with their id.

Note that, when a counter is done serving a customer, one customer from the entrance queue may join the counter queue of that counter.

Building on Lab 2

You are required to build on top of your Lab 2 submission for this lab.

Assuming you have `lab2-<username>` and `lab3-<username>` under the same directory, and `lab3-<username>` is your current working directory, you can run

```
cp -i ../lab2-<username>/*.java .
rm -i Lab2.java
```

to copy all your Java code over.

If you are still unfamiliar with Unix commands to navigate the file system and processing files, please review our Unix guide.

You are encouraged to consider your tutor's feedback and fix any issues with your design for your Lab 2 submission before you embark on your Lab 3.

Skeleton for Lab 3

We provide five files for Lab 3: - the main `Lab3.java` (which is simply `Lab2.java` renamed) - `QueueTest.java` to test your `Queue<T>` class, - `ArrayTest.java` to test your `Array<T>` class, - `CS2030STest.java`, which is the CS2030S test library, and - `Array.java`, which is the skeleton file for `Array<T>`.

Except for `Array.java`, these files should not be modified for this lab.

Your Tasks

We suggest you solve this lab in the following order.

1. Make `Queue` a generic class

The class `Queue` given to you in Lab 2 stores its elements as `Object` references, and therefore is not type-safe. Now that you have learned about generics, you should update `Queue` to make it a generic class `Queue<T>`.

You are encouraged to test your `Queue<T>` in `jshell` yourself. A sample test sequence can be found under `outputs/QueueTest.out`.

The file `QueueTest.java` helps to test your `Queue<T>` class (see "Running and Testing" section below).

```
javac -Xlint:rawtypes QueueTest.java
java QueueTest
```

2. Create a generic `Array<T>` class

Let's call the class that encapsulates the counter `ServiceCounter` (you may name it differently). We have been using an array to store the `ServiceCounter` objects. In Lab 3, you should replace that with a generic wrapper around an array. In other words, we want to replace `ServiceCounter[]` with `Array<ServiceCounter>`. You may build upon the `Array<T>` class from the notes -- Unit 25.

The `Array<T>` class you build must support the following:

- `Array<T>` takes in only a subtype of `Comparable<T>` as its type argument. That is, we want to parameterize `Array<T>` with only a `T` that can compare with itself. Note that in implementing `Array<T>`, you will find another situation where using raw type is necessary. You may, for this case, use `@SuppressWarnings("rawtypes")` at *the smallest scope possible* to suppress the warning about raw types.
- `Array<T>` must support the `min` method, with the following descriptor:

  T min()

`min` returns the minimum element (based on the order defined by the `compareTo` method of the `Comparable<T>` interface).

- `Array<T>` supports a `toString` method. The code has been given to you in `Array.java`.

You are encouraged to test your `Array<T>` in `jshell` yourself. A sample test sequence can

be found under `outputs/ArrayTest.out`.

The file `ArrayTest.java` helps to test your `Array<T>` class (see "Running and Testing" section below).

```
javac -Xlint:rawtypes ArrayTest.java
java ArrayTest
```

3. Make Your `ServiceCounter` Comparable to Itself

Your class that encapsulates the service counter must now implement the `Comparable<T>` interface so that it can compare with itself and it can be used as a type argument for `Array<T>`.

You should implement `compareTo` in such a way that `counters.min()` returns the counter that a customer should join (unless all the counter queues have reached maximum length).

4. Update Your Simulation

By incorporating `Queue<T>`, `Array<T>`, `ServiceCounter`, modify your simulation so that it implements the shop with counter queues as described above.

5. Other Changes Needed

We also need to make the following changes to the input and output of the program.

1. There is an additional input parameter, an integer L, indicating the maximum allowed length of the counter queue. This input parameter should be read immediately *after* reading the number of service counters and *before* the maximum allowed length of the entrance queue.
2. Now that we have two types of queues, if a customer joins the entrance queue, the customer along with the queue *before* joining should be printed as such:

   ```
   1.400: C3 joined shop queue [ C1 C2 ]
   ```

3. The counter queue will be printed whenever we print a counter.

   ```
   1.200: C2 joined counter queue (at S0 [ C1 ])
   2.000: C0 service done (by S0 [ C1 C2 ])
   ```

Following CS2030S Style Guide

Like Lab 2, you should also make sure that your code follows the given Java style guide

Assumptions

We assume that no two events involving two different customers ever occur at the same time (except when a customer departs and another customer begins its service, or when a customer is done and another customer joins the counter queue from the entrance queue). As per all labs, we assume that the input is correctly formatted.

Compiling, Testing, and Debugging

Compilation

To compile your code,

```
$ javac -Xlint:rawtypes *.java
```

To check for style,

```
$ java -jar ~cs2030s/bin/checkstyle.jar -c ~cs2030s/bin/cs2030_checks.xml
*.java
```

Running and Testing

You may test your simulation code similarly to how you test your Lab 2.

Test Cases

A series of test cases `Lab3.x.in` and `Lab3.x.out` are provided. Test cases for `x` = 1 to 10 duplicate the corresponding test cases of Lab 2, with the input format updated to allow additional input of L (max counter queue length). We set L to 0 in all these test cases. After your update your simulation to add counter queues, your code should still work for the scenarios in Lab 2 (except for small differences in the input and output format).

Test case x = 11 to 13 are test cases without entrance queue (m = 0). The rest of the test cases test scenarios with both entrance and counter queues.

Grading

This lab is worth 16 marks and contributes 4% to your final grade. The marking scheme is as follows:

- `Queue<T>`: 1 mark
- `Array<T>`: 3 marks
- Comparable counters: 1 mark
- Using Queue, Array, counters correctly in simulation: 2 marks
- Style: 2 marks
- Correctness: 3 marks
- OO Design: 4 marks

Note that the style marks is conditioned on evidence of efforts in solving Lab 3. Simply resubmitting your Lab 2 solution as Lab 3 does not automatically earn you 2 style marks.

Code that cannot be compiled will receive 0.

Penalty for Unnecessary Raw Types and Abuse of @SuppressWarnings

We penalize heavily (-1 marks per instance) for each unnecessary use of raw types and for each abuse of `@SuppressWarnings`.

For Lab 3, you are allowed at most one instance of raw type in the constructor of `Array<T>` and one use of `@SuppressWarnings("rawtypes")` in the smallest scope, immediately above the use of raw type.

Submission

Upload the following files to CodeCrunch.

- Lab3.java
- *any other* `.java` *files you use*