Lab 5

Submission deadline: 2359, *Wednesday*, October 11, 2017.

You have more time than normal to complete Lab 5.

Prerequisites

This lab assumes that students:

- have already attempted Lab 4
- have an understanding of the multi-queues, multi-servers system being simulated
- are familiar with CS2030 coding style and javadoc documentation
- are comfortable with using Java Collections Framework
- are familiar with random number generation in `java.lang.Math`
- are familiar with static methods and variables
- are familiar with inheritance and interfaces

Learning Objectives

After completing this lab, students should develop a better appreciation of object-oriented programming, especially in:

- encapsulating data and behavior within classes
- inheriting common behavior from the parent class and implementing sub-class specific behaviors by overriding the parent class

- the ease of extending an existing program with new behaviors if the right encapsulations are used.

Setup

A sample code that solves Lab 4 is given. Read it carefully and understand all the classes and APIs given.

After which, you are free to choose, either:

- extend the given code for Lab 4 to solve Lab 5, or
- extend your own code for Lab 4 to solve Lab 5

I encourage you to do both, and compare how easy / difficult it is to extend the two versions of the code to implement the new requirements of Lab 5.

The sample code from Lab 4 is available on `cs2030-i` under the directory `~cs2030/lab04/sample`. The test data (`TESTDATA1.txt` .. `TESTDATA9.txt` for Lab 5 is available in `~cs2030/lab05`.

Task

For Lab 5, you will be asked to make changes to your Lab 4, to:

- Support different types of customers
- Support three new types of events

You are still required to

- follows the CS2030 Coding Style [../style/index.html]
- clearly documented with `javadoc` [../javadoc/index.html]

Take A Break

We have been working our servers in Lab 4 very hard. They do not get a break, even after serving one million customers! In this lab, we will allow our servers to take a break.

Here is how a server takes a break:

- After a server completes serving a customer, a server either takes a break with probability p_b, or serves the next customer with probability (1-p_b).
- The length of a server's break is random, and is drawn from an exponential distribution with rate parameter β.
- After the break is over, the server returns to its duty and serves the next customer (if available).

To implement this, you can create two new types of events, one for going for break, the other for returning from break, which you can create and schedule in the simulator with the logic above.

New Type of Customers

Let's call the customers we implemented in Lab 4 as *typical* customers. A typical customer, upon arrival, joins a random queue if none of the server can serve the customer immediately. Note that while in Lab 4, this means that none of the server is idle, in Lab 5, we have to consider that a server might be on a break. So a customer joins a random queue if every server is either busy or on a break. A customer, however, can queue up for a server that is on a break.

We will add two new types of customer in Lab 5. A kiasu [1] [#fn:1] customer is one that *joins the shortest queue* [2] [#fn:2] upon arrival if none of the server is available to serve it immediately. A pioneer [3] [#fn:3] customer is one that has highest priority to being served -- he/she always joins the queue at the front of the queue upon arrival, even if there is another pioneer customer at the front of the queue [4] [#fn:4] Pioneer customers chooses a queue to join randomly, since the length of the queue has no affect on them.

When the simulator generates a new customer, there is a p_k probability that the customer is a kiasu customer. There is another probability p_p that the customer is a pioneer customer. A typical customer is generated if a customer is neither a kiasu nor pioneer customer.

Switching Queues

We will introduce a new behavior to our customers, *regardless of which type they are*. A customer, at some random time after joining the queue, decides to look around and look for either: (i) an idle server who is not on a break, or (ii) the shortest queue to join. If an idle server who is not on a break is found, the customer leaves its current queue, goes to that server, and is served immediately. Otherwise, the customer looks for the shortest queue [5] [#fn:5] If joining the shortest queue would reduce the number of other customers in front of this customer, it will leave its current queue and join that shortest queue. Note that it does this even when the server for that shortest queue is on a break.

Regardless of whether a customer switch queue or not, it always try to do that again after some random time, until the customer is no longer in the queue.

You can implement this with another new type of "switch" event. Upon joining a queue, we schedule a "switch" event some time in the future (a random time interval drawn from exponential distribution just like Lab 3, with rate parameter σ). After every "switch" event, we schedule another one for the same customer, again, some random time in the future drawn from the same distribution.

Note that by the time the simulator gets to a "switch" event, it is possible that the customer has already left the queue (either being served or completed the service). In this case, we should just discard the "switch" event.

We focus on the case with multiple queues, although the resulting code should still work for the case of single queue, it is less interesting for this lab.

Implementation Hints

Random number generation

To do something randomly with probability p, you can use the `nextDouble()` method of `Random` class to throw a dice.

```
double dice = random.nextDouble();
if (dice < p) {
  doSomething();
}
```

To do something randomly with probability p, and something else with probability q, you can:

```
double dice = random.nextDouble();
if (dice < p) {
  doSomething();
} else if (dice > 1-q) {
  doSomethingElse();
}
```

To generate inter-switching time with parameter σ and break time with parameter β that follows the exponential distribution, you can use the same methods as Lab 3:

```
  -Math.log(rng.nextDouble()) / rate
```

where `rate` is either β or σ.

Collections

Java provides a utilitiy class called `Collections` [https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html], full of useful methods. You may find it useful for your Lab 5.

Input Files

With this change, the input file now contains additional *simulator parameters*. The first new lines are the same as Lab 4:

- The first line is an `int` value, indicating the seed to the simulator
- The next line is an `int` value, indicating the number of servers c
- The next line contains a `double` value, indicating the service rate μ
- The next line is an `int` value, indicating the number of queues -- it must be either 1 or c
- The next line is a `double` value, indicating the arrival rate λ
- The next line is a `double` value, indicating the total simulation time.

The new lines are:

- The next line contains two `double` values, seperated by a space, indicating the probability p_p and p_k respectively
- The next line is another `double` value, for σ
- The next line contains the probability that the server goes on a break after serving a customer p_b
- The last line contains the rate paramter for the length of the break, β.

Output

Like in previous labs, the last line of your program output should not be changed. It should remain as:

```
System.out.printf("%.3f %d %d", ..")
```

The two integer outputs are not as important here, but let's keep it there and print them anyway.

You can compare your output with the following tables [6] [#fn:6]:

TEST	Average Waiting Time	Customers	Switching	Break	
1	3.4 - 4.4	Only typical customers	Negligable	No	
2	2.7 - 3.7	Only kiasu customers	Negligable	No	
3	3.3 - 4.3	Only pioneers	Negligable	No	
4	3.1 - 4.2	Equal mix	Negligable	No	
5	~~2.2 - 3.0~~ 2.6 - 3.6	Equal mix	Some	No	
6	~~0.7 - 1.2~~ 1.6 - 2.4	Equal mix	Frequent	No	
7	~~0.9 - 1.5~~ 2.0 - 2.9	Equal mix	Frequent	Half of the time, short break	
8	~~1.4 - 2.3~~ 2.6 - 3.8	Equal mix	Frequent	Always, short break	
9	~~7 - 13~~ 10 - 16	Equal mix	Frequent	Always, long break	

Grading

This lab contributes another 4 marks to your final grade (100 marks).

- 1 mark for implementation of kiasu customer

- 1 mark for implementation of pioneer customer
- 1 mark for implementation of server breaks
- 1 mark for implementation of switching events

To get 1 mark, the implementation has to be correct in its logic and good enough in its design (right OO concepts applied).

You can get -0.5 mark deduction for serious violation of style and -0.5 mark deduction for not documenting your code (in Javadoc format) properly.

Submission

When you are ready to submit your lab, on `cs2030-i` , run the script

```
~cs2030/submit05
```

which will copy all files matching `*.java` (and nothing else) from your `~/lab05` directory on `cs2030-i` to an internal grading directory. We will test compile and test run with a tiny sample input to make sure that your submission is OK.

You can submit multiple times, but only the most recent submission will be graded.

---

1. "Afraid to loose" in Singlish
2. Breaking ties arbitarily
3. Pioneer as in "pioneer generation"
4. A queue consists of only pioneer customers is a last-in-first out queue.
5. Again, breaking ties arbitarily.
6. Your code might still be correct if it is slightly out of range -- try a different seed and see. If it gets consistently out of range, or is way out of range, then double check your logic.