Ex 3: Simulation III

Basic Information

- **Deadline:** 17 September 2024, Tuesday, 23:59 SGT
- **Difficulty:** ★★★★★★★

Prerequisite

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

Goal

This is a continuation of Programming Exercise 2. Programming Exercise 3 extends some of the requirements of Programming Exercise 2 and adds new entities 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 requirements with less effort.

Programming Exercise 3 also involves writing your generic classes.

Tasks

We have two extensions that we want to do on our coffee shop. Before we do that, we need to make sure that your Ex 1 code are sufficiently good.

Task 1: Address the Concern

Hopefully, by now your friendly TA should have written comments about your submission. You should address the concerns pointed by your TA. If those concerns remained unaddressed, the same deductions will be applied to the current submission too.

Task 2: Extend

Extension 1: Queueing with the Barista

The coffee shop has now decided to streamline its operations. It rearranges the layout and makes some space for queues at the counters with the barista. With that, customers can now wait at individual barista.

In this exercise, we will modify the simulation to add a queue to each barista/counter. Remember that each counter has exactly one barista so we can use counter and barista interchangeably. For simplicity, we will refer to the queue at the counter as barista queue.

The maximum queue length for each barista queue is l. All barista queues are independent. Recall from Exercise 2 that the maximum queue length for the entrance queue is denoted as m. m and l may be different.

When a customer arrives, a few scenarios could happen:

- If more than one counter is available, a customer will go to the counter with the smallest id (*just like in Exercise* 2).
- If none of the counters is available, but at least one of the barista queues is not full, the customer will join the counter with the shortest barista queue. If there are two counters with the same queue length, we break ties by going to the counter with the smaller id.
- If every barista queue has reached its maximum capacity of l, but the entrance queue is not full, then the customer will join the entrance queue.
- If every barista queue has reached its maximum capacity of l, and the entrance queue has reached its maximum capacity of m, then the customer will leave.

When a counter is done serving a customer and becomes available, the customer at the front of its barista queue will proceed to the counter for service. This event frees up a slot in the barista queue. One customer from the entrance queue may join the barista queue of that counter. It takes 0.05 time unit for a customer to move from the entrance queue to the barista queue.

Note that, the entrance queue is empty unless all barista queues have reached the maximum length. Furthermore, a counter cannot "steal" a customer from another barista queue. Once a customer joins a barista queue, it cannot switch to another barista queue.

Extension 2: Custom Orders

Recall that a customer comes into the coffee shop with an order. We will now add custom instructions on the orders. For simplicity, assume that all barista can accommodate all instructions. The custom instructions are always a single word (*i.e.*, not separated by a space).

Extension 3: Changes to Input

1. There is an additional input parameter, an integer l, indicating the maximum allowed length of the barista queue. This input parameter should be read immediately after reading the number of coffee shop counters and before the maximum allowed length of the entrance queue.
2. There is an additional input parameter at the end of each line in the input file that indicates the custom instructions for the order. This new input is always a string consisting only of alphabets. You should read the input using `Scanner::next()`.

CHANGES

Ex2.1.in

```
3 1 2
1.0 1.0 0
3.0 1.0 1
5.0 1.0 0
```

Ex3.1.in

```
3 1 0 2
1.0 1.0 0 Small
3.0 1.0 1 Medium
5.0 1.0 0 Large
```

In `Ex3.1.in`, the first line of the input has an additional integer input `0` as the third input on the first line. This specifies the length of the barista queue.

Additionally, for subsequent lines we have additional String inputs for each line.

- Line 2: The customer wants a "Small" order.
- Line 3: The customer wants a "Medium" order.
- Line 4: The customer wants a "Large" order.

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 their order*). As per all exercises, we assume that the input is correctly formatted.

Extension 4: Changes to Output

1. Whenever we print a counter/barista, we also print its barista queue.

From Ex3.13.out

```
1.700: C7 joined barista queue (at B1 [ C4 ])
```

2. Whenever we print an order, we print the custom instructions before the order. The custom instruction is enclosed within a parentheses.

From Ex3.13.out

```
3.100: C7 ordered (Medium) Coffee Espresso (by B1 [ ])
4.000: C3 served (Hot) Coffee Latte (by B0 [ C6 C9 ])
```

Task 3: Incorporate Generic Queue

Do NOT Edit

You are not allowed to edit `CoffeeQueue.java`.

Delete Queue.java

Please completely delete `Queue.java` before submitting.

We have given you a generic `CoffeeQueue<T>` class as an improvement to `Queue` class in Exercise 2 because `Queue` stores its elements as `Object` and is not type-safe. Update your program to incorporate `CoffeeQueue<T>` instead of `Queue`. You should not longer use `Queue` and you should delete `Queue.java`.

Our `CoffeeQueue<T>` is comparable to other `CoffeeQueue<T>`. This is done by implementing the interface `Comparable<CoffeeQueue<T>>` and providing an implementation of `compareTo`. We compare two `CoffeeQueue<T>` according to their sizes. Consider the method invocation below.

```
queue1.compareTo(queue2);
```

- If the result is negative, then `queue1` < `queue2`.
- If the result is positive, then `queue1` > `queue2`.

- Otherwise, `queue1` = `queue2`.

Task 4: Creating Generic Sequence

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

The `Seq<T>` class should support the following:

- `Seq<T>` takes in only a subtype of `Comparable<T>` as its type argument. That is, we want to parameterize `Seq<T>` with only a `T` that can compare with itself. Note that in implementing `Seq<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. If you need to suppress multiple warnings, you may use `@SuppressWarnings({"warning1", "warning2"})`.
- `Seq<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*). Recap the specification of `compareTo`. If `x.compareTo(y)` returns

  - *any negative value*, then `x` < `y`.
  - *zero*, then `x` = `y`.
  - *any positive value*, then `x` > `y`.
- `Seq<T>` supports a `toString` method. The code has been given to you in `Seq.java`.

You are encouraged to test your `Seq<T>` in jshell yourself. A sample test sequence is as follows:

```
jshell> /open Seq.java
jshell> Integer i
jshell> String s
jshell> Seq<Integer> a;
jshell> a = new Seq<Integer>(4);
jshell> a.set(0, 3);
jshell> a.set(1, 6);
jshell> a.set(2, 4);
jshell> a.set(3, 1);
jshell> a.set(0, "huat");
```

```
|  Error:
|  incompatible types: java.lang.String cannot be converted to
java.lang.Integer
|  a.set(0, "huat");
|           ^----^
jshell> i = a.get(0)
jshell> i
i ==> 3
jshell> i = a.get(1)
jshell> i
i ==> 6
jshell> i = a.get(2)
jshell> i
i ==> 4
jshell> i = a.get(3)
jshell> i
i ==> 1
jshell> s = a.get(0)
|  Error:
|  incompatible types: java.lang.Integer cannot be converted to
java.lang.String
|  s = a.get(0)
|      ^------^
jshell> i = a.min()
jshell> i
i ==> 1
jshell> a.set(3,9);
jshell> i = a.min()
jshell> i
i ==> 3
jshell> // try something not comparable
jshell> class A {}
jshell> Seq<A> a;
|  Error:
|  type argument A is not within bounds of type-variable T
|  Seq<A> a;
|        ^
jshell> class A implements Comparable<Long> { public int compareTo(Long
i) { return 0; } }
jshell> Seq<A> a;
|  Error:
|  type argument A is not within bounds of type-variable T
|  Seq<A> a;
|        ^
jshell> // try something comparable
jshell> class A implements Comparable<A> { public int compareTo(A a) {
return 0; } }
jshell> Seq<A> a;
jshell>
```

The file `SeqTest.java` helps to test your `Seq<T>` class.

```
username@pe111:~/ex3-username$ javac -Xlint:unchecked -Xlint:rawtypes
SeqTest.java
username@pe111:~/ex3-username$ java SeqTest
```

Task 5: Comparable Counter

Your class that encapsulates coffee counters must now implement `Comparable<T>` interface so that it can be compared with another instance of the same type. This way, it can also be used as type argument for `Seq<T>`.

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

Task 6: Updating the Simulation

By incorporating `CoffeeQueue<T>` and `Seq<T>`, your simulation will need to be updated. This potentially includes modifying `CoffeeCounter` (*the name may be different*) as well as adding new subclass of `Event`.

Finally, ensure that all your files can be compiled without warning even with `-Xlint:unchecked` and `-Xlint:rawtypes`.

Skeleton for Programming Exercise 3

We provide three classes for Exercise 3, the main `Ex3.java` (*which is simply* `Ex2.java` *renamed*), `CoffeeQueue.java` (*which is simply a generic version of* `Queue.java`), and `Seq.java`. You should not edit `Ex3.java` and `CoffeeQueue.java` but you need to fill in the blanks (*i.e.*, `// TODO`) for `Seq.java` as described above.

We also provide two new classes for testing: `CS2030STest.java` (*which you should be familiar with from Programming Exercise* 0) and `SeqTest.java`.

Do NOT Edit

You should **NOT** edit `Ex3.java`, `CoffeeQueue.java`, `CS2030STest.java`, or `SeqTest.java`. However, you may change the `@author` tag of these two files if you wish.

Additionally, if you have made changes to `Event.java`, `Simulation.java`, or `Simulator.java`, you will need to use the original file given for Programming Exercise 1 as those are not supposed to be edited.

Building on Programming Exercise 2

You are required to build on top of your Programming Exercise 2 submission for this exercise.

Assuming you have `ex2-username` and `ex3-username` under the same directory, and `ex3-username` is your current working directory, you can run

```
username@pe111:~/ex3-username$ cp -i ../ex2-username/*.java .
username@pe111:~/ex3-username$ rm -i Ex2.java
```

to copy all your Java code over and remove the main file for `Ex2`.

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

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

Compiling, Testing, and Debugging

Compiling

To compile your code, you can compile all the Java file.

```
username@pe111:~/ex3-username$ javac -Xlint:unchecked -Xlint:rawtypes *.java
```

To check for style,

```
username@pe111:~/ex3-username$ java -jar ~cs2030s/bin/checkstyle.jar -c ex3_style.xml *.java
```

Running and Testing

You may test your simulation code similarly to how you test your Programming Exercise 3.

Note that test cases 1 to 11 set l to 0, so there is no barista queue. Test cases 12 and 14 set l to non-zero and m to 0, so there is no entrance queue. Test cases 15 and 16 test scenarios with both entrance and barista queues.

Documentation (Optional)

Documenting your code with Javadoc is optional for Programming Exercise 2. It is, however, always a good practice to include comments to help readers understand your code.