NUS | Computing
National University of Singapore

CodeCrunch

CS2030 (2410) Lab #3

Tags & Categories

Tags:

Categories:

Related Tutorials

Task Content

Lab #3

The server and customer interactions from the previous lab can be broken down into a series of activities or events. Based on whether the server is available or busy serving another customer, there are two event transitions:

- `ARRIVE → SERVE`: a new customer arrives, then gets served immediately; or
- `ARRIVE → LEAVE`: a new customer arrives, then leaves.

We include another event transition:

- `SERVE → DONE`: a server is done serving a customer.

Notice that when a customer arrives and a server is available, the `SERVE` event follows right after `ARRIVE` and the `DONE` event occurs sometime in future.

```
ARRIVE → SERVE → after some time... → DONE
```

An event occurs at a particular time, and each event alters the state of the system and may generate more events. We call this a *discrete event simulator* as states remain unchanged between two events, which allows the simulation to jump from the time of one event to another.

Processing events *in the right way* requires the use of an event priority queue. While Java provides the `PriorityQueue` class which is a mutable collection (like `ArrayList`), we have provided our own `PQ` class that is immutable, as well as an accompanying `Pair` class.

The programs `PQ.java` and `Pair.java` are given. Specifically, `PQ.java` includes javadoc comments. To automatically generate HTML documentation from the comments, issue the command:

```
$ javadoc -d doc PQ.java
```

You may then navigate through the documentation from `allclasses-index.html` found in the `doc` directory.

We have seen how strings can be compared based on the *natural order* by using the `compareTo` method.

```
jshell> "one".compareTo("two")
$.. ==> -5
```

This allows a stream of strings to be sorted.

```
jshell> Stream.of("one", "two", "three").
   ...> sorted().
   ...> toList()
$.. ==> [one, three, two]
```

Similarly, a priority queue can be used to prioritize elements in the queue based on the natural order. The following shows how an empty PQ of strings can be constructed.

```
jshell> PQ<String> pq = new PQ<String>()
pq ==> []
```

We can add elements to the priority queue via the add method.

```
jshell> pq.add("one")
$.. ==> [one]

jshell> pq // note that pq is immutable
pq ==> []

jshell> pq = pq.add("one").add("two").add("three")
pq ==> [one, two, three]
```

Values can be returned from PQ via the poll method. Each call to poll gives a pair of values comprising the element polled from the PQ wrapped in an Optional, and the remaining PQ.

```
jshell> pq.poll()
$.. ==> Pair[t=Optional[one], u=[three, two]]

jshell> pq.poll().u().poll()
$.. ==> Pair[t=Optional[three], u=[two]]

jshell> pq.poll().u().poll().u().poll()
$.. ==> Pair[t=Optional[two], u=[]]

jshell> pq.poll().u().poll().u().poll().u()
$.. ==> []

jshell> pq.poll().u().poll().u().poll().u().poll()
$.. ==> Pair[t=Optional.empty, u=[]]
```

You may notice that the output of elements in a priority queue is not necessarily in order. What matters most is the values returned from the poll method.

We are now ready to process our events using PQ. As usual, we create arrival events for all customer arrivals, with each customer having both an arrival time and a service time (although the customer may not be served). Rather than processing them right away, we now add them into a priority queue that is ordered in terms of events. Events are ordered by earliest occurrence of the event, and in the case of ties, by order in which the customer arrives. You may assume that no two customers arrive at the same time.

Having included all arrival events in the priority queue, we can begin processing the queue:

- poll an event from the queue
- if necessary, generate the next event using the current state of the servers in the shop and add into the queue

Repeate the above until the queue is empty. Remember to update the server along the way. Here is an example of how we can iterate through the priority queue.

```
jshell> pq
pq ==> [one, two, three]

jshell> Stream.iterate(pq, pq -> !pq.isEmpty(), pq -> pq.poll().u()).
   ...> map(pq -> pq.poll().t().orElse("")).
   ...> toList()
$.. ==> [one, three, two]
```

Level 1

To start off the simulation, we need a prority queue of events PQ<Event> comprising of arrival events for each cutomer.

Define the Event and ArriveEvent classes. Specifically an event requires an event time, so that we can obtain the earliest event from the priority queue upon polling. This can be seen from the construction of an ArriveEvent event that takes in a customer (comprising of an identifier, the arrival time and the service time) and an event time (which is the same as the arrival time).

```
jshell> PQ<Event> pq = new PQ<Event>().add(new ArriveEvent(new Customer(1, 1.0, 1.0), 1.0)).
   ...> add(new ArriveEvent(new Customer(2, 2.0, 1.0), 2.0))
pq ==> [1.0 customer 1 arrives, 2.0 customer 2 arrives]
```

```
jshell> pq.poll()
$.. ==> Pair[t=Optional[1.0 customer 1 arrives], u=[2.0 customer 2 arrives]]

jshell> pq.poll().u().poll()
$.. ==> Pair[t=Optional[2.0 customer 2 arrives], u=[]]

jshell> pq.poll().u().poll().u().poll()
$.. ==> Pair[t=Optional.empty, u=[]]

jshell> pq.poll().u().poll().u().isEmpty()
$.. ==> true
```

Level 2

From the arrival event, we need to determine the next event, which could either be a serve event or a leave event based on the status of the servers in the shop.

As such you will need to define a `next` method that takes in a `Shop` and return the next event and an updated shop within a pair `Pair<Event,Shop>`.

Here is an example of customer `1` arriving at `1.0` to a shop with two fresh servers which results in a serve event.

```
jshell> new ArriveEvent(new Customer(1, 1.0, 1.0), 1.0).next(new Shop(2)).t()
s ==> 1.0 customer 1 serve by server 1
```

The following example depicts customer `1` arriving at `1.0` to a shop with two servers where only the first server is serving customer `2`.

```
jshell> new ArriveEvent(new Customer(1, 1.0, 1.0), 1.0).next(new Shop(2).
   ...> update(new Server(1).serve(new Customer(2, 1.0, 1.0)))).t()
$.. ==> 1.0 customer 1 serve by server 2
```

The example below depicts customer `1` arriving at `1.0` to a shop with only one server which is serving customer `2`. Note that the resulting event is a leave event.

```
jshell> new ArriveEvent(new Customer(1, 1.0, 1.0), 1.0).next(new Shop(1).
   ...> update(new Server(1).serve(new Customer(2, 1.0, 1.0)))).t()
$.. ==> 1.0 customer 1 leaves
```

The following shows a sequence of state transitions starting from an arrival event to a done event.

```
jshell> Event e1 = new ArriveEvent(new Customer(1, 1.0, 1.0), 1.0)
e1 ==> 1.0 customer 1 arrives

jshell> Event e2 = e1.next(new Shop(2)).t()
e2 ==> 1.0 customer 1 serve by server 1

jshell> Shop s2 = e1.next(new Shop(2)).u()
s2 ==> [server 1, server 2]

jshell> Event e3 = e2.next(s2).t()
e3 ==> 2.0 customer 1 done

jshell> Shop s3 = e2.next(s2).u()
s3 ==> [server 1, server 2]
```

You may define your serve, leave and done events in any way you wish. You only need to ensure that the output is consistent with the ones shown above.

Alternatively, you may also have the `next` method return `Pair<Optional<Event>,Shop>`.

Level 3

In the previous lab, you have seen how the `State` can be used to set up a specific configuration of the simulation, so that the `next` method can be used to test the rest of the simulation for correctness. To do the same for this lab, you will need to include the `PQ` and set up the events correctly.

```
jshell> PQ<Event> pq = new PQ<Event>().add(new ArriveEvent(new Customer(1, 1.0, 1.0), 1.0)).
   ...> add(new ArriveEvent(new Customer(2, 2.0, 1.0), 2.0))
pq ==> [1.0 customer 1 arrives, 2.0 customer 2 arrives]

jshell> new State(pq, new Shop(2)).next()
```

```
$.. ==> 1.0 customer 1 arrives

jshell> new State(pq, new Shop(2)).next().next()
$.. ==> 1.0 customer 1 arrives
1.0 customer 1 serve by server 1

jshell> new State(pq, new Shop(2)).next().next().next()
$.. ==> 1.0 customer 1 arrives
1.0 customer 1 serve by server 1
2.0 customer 1 done

jshell> System.out.println(
   ...>    Stream.iterate(new State(pq, new Shop(2)),
   ...>       state -> !state.isEmpty(),
   ...>       state -> state.next())
   ...> .map(x -> x + "\n")
   ...> .reduce("", (x, y) -> y))
1.0 customer 1 arrives
1.0 customer 1 serve by server 1
2.0 customer 1 done
2.0 customer 2 arrives
2.0 customer 2 serve by server 1
3.0 customer 2 done
```

Level 4

You are given the following `Main` class.

```
import java.util.List;
import java.util.Scanner;
import java.util.stream.Stream;

void main() {
    Scanner sc = new Scanner(System.in);
    int numOfServers = sc.nextInt();
    int numOfCustomers = sc.nextInt();

    sc.nextLine(); // removes trailing newline
    List<Pair<Integer,Pair<Double,Double>>> arrivals = sc.useDelimiter("\n")
        .tokens()
        .map(line -> {
            List<String> token = Stream.of(line.split(" ")).toList();
            return new Pair<Integer,Pair<Double,Double>>(Integer.parseInt(token.get(0)),
                    new Pair<Double,Double>(Double.parseDouble(token.get(1)),
                        Double.parseDouble(token.get(2))));
        }).toList();

    State state = new Simulator(numOfServers, numOfCustomers, arrivals).run();
    System.out.println(state);
}
```

Write the `Simulator` class so as to carry out the simulation. Sample runs of the program is given below.

```
$ javac --release 21 --enable-preview Main.java
Note: Main.java uses preview features of Java SE 21.
Note: Recompile with -Xlint:preview for details.

$ cat 1.in
3 3
1 0.5 1.0
2 0.6 1.0
3 0.7 1.0

$ cat 1.in | java --enable-preview Main
0.5 customer 1 arrives
0.5 customer 1 serve by server 1
0.6 customer 2 arrives
0.6 customer 2 serve by server 2
0.7 customer 3 arrives
0.7 customer 3 serve by server 3
1.5 customer 1 done
1.6 customer 2 done
1.7 customer 3 done
```

```
$ cat 2.in
3 6
1 0.5 1.0
2 0.6 1.0
3 0.7 1.0
4 1.5 1.0
5 1.6 2.0
6 1.7 3.0

$ cat 2.in | java --enable-preview Main
0.5 customer 1 arrives
0.5 customer 1 serve by server 1
0.6 customer 2 arrives
0.6 customer 2 serve by server 2
0.7 customer 3 arrives
0.7 customer 3 serve by server 3
1.5 customer 1 done
1.5 customer 4 arrives
1.5 customer 4 serve by server 1
1.6 customer 2 done
1.6 customer 5 arrives
1.6 customer 5 serve by server 2
1.7 customer 3 done
1.7 customer 6 arrives
1.7 customer 6 serve by server 3
2.5 customer 4 done
3.6 customer 5 done
4.7 customer 6 done

$ cat 3.in
2 3
1 0.5 1.0
2 0.6 1.0
3 0.7 1.0

$ cat 3.in | java --enable-preview Main
0.5 customer 1 arrives
0.5 customer 1 serve by server 1
0.6 customer 2 arrives
0.6 customer 2 serve by server 2
0.7 customer 3 arrives
0.7 customer 3 leaves
1.5 customer 1 done
1.6 customer 2 done

$ cat 4.in
2 3
1 0.5 1.0
2 0.6 1.0
3 1.5 1.0

$ cat 4.in | java --enable-preview Main
0.5 customer 1 arrives
0.5 customer 1 serve by server 1
0.6 customer 2 arrives
0.6 customer 2 serve by server 2
1.5 customer 1 done
1.5 customer 3 arrives
1.5 customer 3 serve by server 1
1.6 customer 2 done
2.5 customer 3 done

$ cat 5.in
2 6
1 0.5 1.1
2 0.6 0.9
3 0.7 0.7
4 1.5 0.1
5 1.6 0.2
6 1.7 0.3

$ cat 5 | java --enable-preview Main
0.5 customer 1 arrives
```

```
0.5 customer 1 serve by server 1
0.6 customer 2 arrives
0.6 customer 2 serve by server 2
0.7 customer 3 arrives
0.7 customer 3 leaves
1.5 customer 2 done
1.5 customer 4 arrives
1.5 customer 4 serve by server 2
1.6 customer 1 done
1.6 customer 4 done
1.6 customer 5 arrives
1.6 customer 5 serve by server 1
1.7 customer 6 arrives
1.7 customer 6 serve by server 2
1.8 customer 5 done
2.0 customer 6 done
```

Submission (Course)

Select course: CS2030 (2024/2025 Sem 1) - Programming Methodology II

Your Files:

BROWSE

SUBMIT (only .java, .c, .cpp, .h, .jsh, and .py extensions allowed)

To submit multiple files, click on the Browse button, then select one or more files. The selected file(s) will be added to the upload queue. You can repeat this step to add more files. Check that you have all the files needed for your submission. Then click on the Submit button to upload your submission.