NUS | Computing
National University of Singapore

CodeCrunch

## CS2030 (2410) Lab #2

### Tags & Categories

Tags:

Categories:

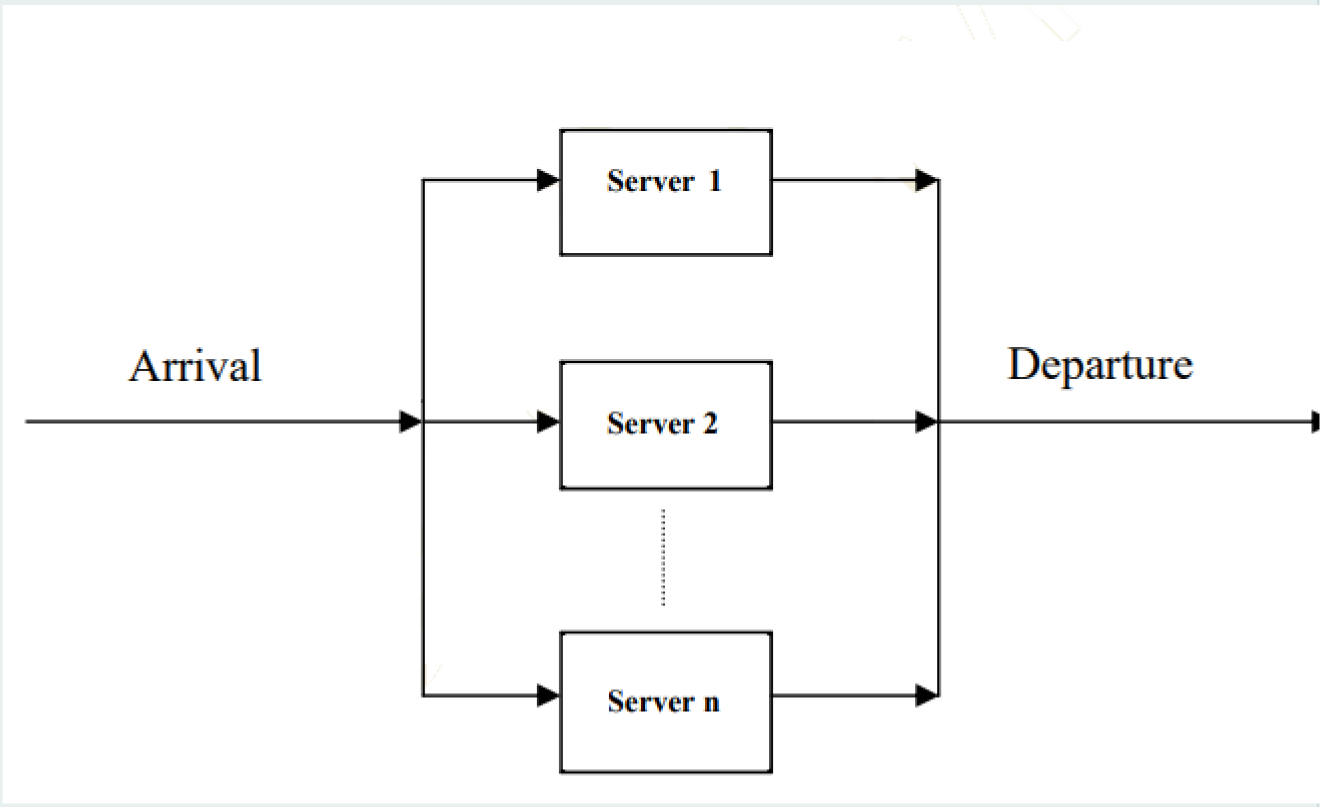
### Related Tutorials

### Task Content

## Lab #2

We would like to simulate how customers are being served by servers in a shop. Specifically, a customer that arrives will look for the first available server so as to be served for some amount of time. On seeing that all servers are busy, a new arriving customer will just leave.

The following depicts a schematic of the simulation:



- There are *n* servers;
- Each server can serve one customer at a time;
- Each customer that is served has a service time (time taken to serve the customer);
- When a customer arrives:
  - the servers are scanned from 1 to n to find the first one that is available (not serving any customer) and gets served immediately.
  - if all servers are serving customers, then the customer just leaves.

As an example, you are given an input of two servers and three customers, followed by the the integer identifier and floating-point arrival times of the customers,

```
2 3
1 0.500
2 0.600
3 0.700
```

Assuming that the service time to service each customer is constant at `1.0`, a simulation can be performed with the following output that shows how each customer is served.

```
customer 1 arrives
customer 1 served by server 1
customer 2 arrives
customer 2 served by server 2
customer 3 arrives
customer 3 leaves
```

The `Pair` record and `Main` program has been provided for you.

```
record Pair<T,U>(T t, U u) {}
```

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

static final double SERVICE_TIME = 1.0;

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

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

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

For each simulation, a `Simulator` object is created with a constructor that takes in the number of servers, the number of customers, a list of paired values comprising the customer id and arrival time, and a fixed service time.

## Level 1

Write the `Customer` class with a constructor that takes in an identifier of type `int` and an arrival time of type `double`.

```
jshell> new Customer(1, 1.0)
$.. ==> customer 1

jshell> Stream.of(new Customer(1, 1.0), new Customer(2, 2.0)).
   ...> forEach(x -> System.out.println(x))
customer 1
customer 2
```

Include the method `canBeServed` that takes in a `time`, and returns `true` when the customer arrives exactly or later than `time`. That is to say, when the customer arrives, it has a chance to be served.

```
jshell> Stream.of(new Customer(1, 1.0), new Customer(2, 2.0)).
   ...> map(c -> c.canBeServed(2.0)).
   ...> toList()
$.. ==> [false, true]
```

Also include the method `serveTill` that takes in a service time, and returns the time of the end of service.

```
jshell> Stream.of(new Customer(1, 1.0), new Customer(2, 2.0)).
   ...> map(c -> c.serveTill(1.0)).
   ...> toList()
$.. ==> [2.0, 3.0]
```

## Level 2

Write the `Server` class with a constructor that takes in an integer identifier of the server. Each new server is ready to serve the next customer.

```
jshell> new Server(1)
$.. ==> server 1
```

Write a method `serve` that allows a server to serve a customer for a service time specified by the supplier. As an example, the following creates a new server #1 to serve customer #10 which arrives at 1.0, for a service time of 2.0.

```
jshell> new Server(1).serve(new Customer(10, 1.0), 2.0)
$.. ==> server 1
```

In the above, server #1 is available to serve from time `0.0`, starts serving customer #10 at `1.0` and ends the service at `3.0`. The server will only be available to serve another customer from time `3.0` onwards.

Write a method `canServe` to determine if a server can serve a customer. As an example, the following creates a new server #1 and determines if he/she can serve customer #1 which arrives at 1.0.

```
jshell> new Server(1).canServe(new Customer(1, 1.0))
$.. ==> true
```

In the following, server #1 serves customer #1 at 1.0 and becomes available at `3.0`.

```
jshell> new Server(1).serve(new Customer(1, 1.0), 2.0).canServe(new Customer(2, 3.0))
$.. ==> true
```

You can also use streams to test your implementation.

```
jshell> Stream.of(new Customer(1, 1.0), new Customer(2, 2.0)).
   ...> map(c -> new Server(1).canServe(c)).
   ...> toList()
$.. ==> [true, true]

jshell> Stream.of(new Customer(1, 1.0), new Customer(2, 2.0)).
   ...> map(c -> new Server(1).serve(new Customer(10, 1.0), 1.0).canServe(c)).
   ...> toList()
$.. ==> [false, true]

jshell> Stream.of(new Customer(1, 1.0), new Customer(2, 2.0)).
   ...> map(c -> new Server(1).serve(new Customer(10, 1.0), 2.0).canServe(c)).
   ...> toList()
$.. ==> [false, false]
```

## Level 3

Write the class `Shop` with a constructor that takes in the number of servers so as to encapsulate all the servers. Each server is identified by a running number starting from `1`. Note that a shop can have no servers.

```
jshell> new Shop(2)
$.. ==> [server 1, server 2]

jshell> new Shop(0)
$.. ==> []
```

Write a method `findServer` that takes in a customer and finds the first server in the shop that can serve the customer. Since a shop can be empty (or all servers are busy to serve the customer), the method should return an `Optional<Server>`.

```
jshell> new Shop(2).findServer(new Customer(1, 1.0))
$.. ==> Optional[server 1]

jshell> new Shop(0).findServer(new Customer(1, 1.0))
$.. ==> Optional.empty
```

Write a method `update` that takes in a server as argument and updates the corresponding server (with same identifier) in the shop with this new one. You may add more methods in the `Server` class to check for the same servers.

```
jshell> new Shop(2).update(new Server(1).serve(new Customer(1, 1.0), 1.0))
$.. ==> [server 1, server 2]

jshell> new Shop(2).update(new Server(1).serve(new Customer(1, 1.0), 1.0)).
   ...> findServer(new Customer(2, 1.0))
$.. ==> Optional[server 2]
```

Notice in the last example that after updating server #1 to serve a customer at 1.0 until 2.0, the available server to serve customer #2 arriving at 1.0 is now server #2. Here is an example when both servers are busy.

```
jshell> new Shop(2).update(new Server(1).serve(new Customer(1, 1.0), 1.0)).
   ...> update(new Server(2).serve(new Customer(2, 1.0), 1.0)).
   ...> findServer(new Customer(3, 1.0))
$.. ==> Optional.empty
```

## Level 4

The following `Simulator` class is provided for you.

```
import java.util.List;

class Simulator {
    private final int numOfServers;
    private final int numOfCustomers;
    private final List<Pair<Integer,Double>> arrivals;
    private final double serviceTime;

    Simulator(int numOfServers, int numOfCustomers,
        List<Pair<Integer,Double>> arrivals, double serviceTime) {
        this.numOfServers = numOfServers;
        this.numOfCustomers = numOfCustomers;
        this.arrivals = arrivals;
        this.serviceTime = serviceTime;
    }

    State run() {
        Shop shop = new Shop(numOfServers);
        return arrivals.stream()
            .map(x -> new State(shop, new Customer(x.t(), x.u())))
            .reduce(new State(shop), (x, y) -> x.next(y));
    }
}
```

Here is a sample run.

```
jshell> List<Pair<Integer,Double>> arrivals = List.of(new Pair<>(1, 0.5),
   ...> new Pair<>(2, 0.6), new Pair<>(3, 0.7))
arrivals ==> [Pair[t=1, u=0.5], Pair[t=2, u=0.6], Pair[t=3, u=0.7]]

jshell> new Simulator(2, 3, arrivals, 1.0).run()
$.. ==> customer 1 arrives
customer 1 served by server 1
customer 2 arrives
customer 2 served by server 2
customer 3 arrives
customer 3 leaves
```

Your task is to complete the `State` class. An instance of `State` represents a state (or step) of the simulation. From the `run` method above, each customer is mapped to a state which encapsulates a shop and the customer.

```
jshell> new State(new Shop(2), new Customer(1, 1.0))
$.. ==> customer 1 arrives
```

After mapping, a reduction is done starting with a state comprising an empty shop, and performs a reduction via the `next` method. Serving of customers to generate the next shop are all encapsulated within a state. Here is an example of the first reduction step.

```
jshell> new State(new Shop(2)).next(new State(new Shop(2), new Customer(1, 1.0)))
$.. ==> customer 1 arrives
```

```
customer 1 served by server 1
```

For simplicity, you may assume from this point onwards that service time for each customer is strictly `1.0`.

## Level 5

This is merely a verification step. Compile your program together with the given `Main` class.

Sample runs of the program is given below. You should create your own input files using `vim`.

```
$ 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 0.500
2 0.600
3 0.700

$ cat 1.in | java --enable-preview Main
customer 1 arrives
customer 1 served by server 1
customer 2 arrives
customer 2 served by server 2
customer 3 arrives
customer 3 served by server 3

$ cat 2.in
3 6
1 0.500
2 0.600
3 0.700
4 1.500
5 1.600
6 1.700

$ cat 2.in | java Main
customer 1 arrives
customer 1 served by server 1
customer 2 arrives
customer 2 served by server 2
customer 3 arrives
customer 3 served by server 3
customer 4 arrives
customer 4 served by server 1
customer 5 arrives
customer 5 served by server 2
customer 6 arrives
customer 6 served by server 3

$ cat 3.in
2 3
1 0.500
2 0.600
3 0.700

$ cat 3.in | java --enable-preview Main
customer 1 arrives
customer 1 served by server 1
customer 2 arrives
customer 2 served by server 2
customer 3 arrives
customer 3 leaves

$ cat 4.in
2 3
1 0.500
2 0.600
3 1.500

$ cat 4.in | java --enable-preview Main
customer 1 arrives
customer 1 served by server 1
customer 2 arrives
customer 2 served by server 2
customer 3 arrives
customer 3 served by server 1
```

It is interesting to note that Unix also makes use of pipelining to pipe the input of a file (e.g. `cat 1.in`) via the pipe