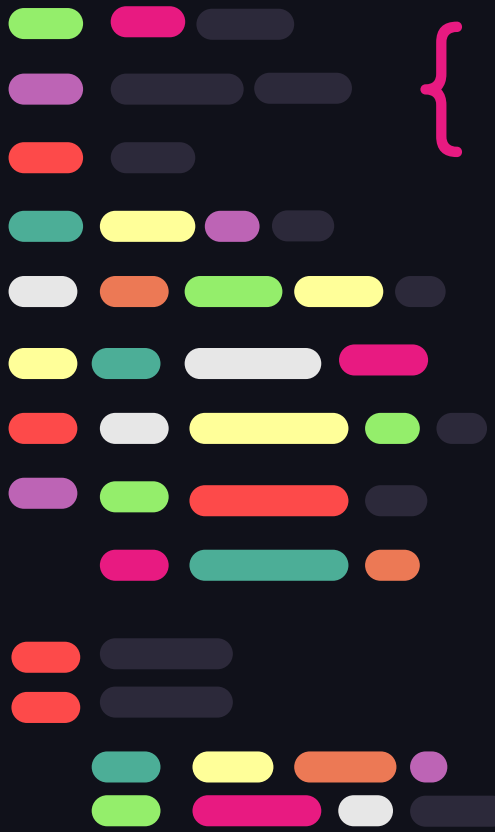


CS2030 Project

AY25/26 Sem 1, Week 9

Fadhil Peer Mohamed <f_p_m@u.nus.edu>
Abner Then <abner.then@u.nus.edu>

Lab Admin

Timeline

WEEK	LAB	DUE
9 (this week)	Project	4
10	5 (Computational context)	Project
11	6 (Computational context)	5
12	Mock PA2	6
13	PA2	Mock PA2

Project - Main

Main

```java
class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int numOfServers = sc.nextInt();
        int qmax = sc.nextInt();
        int numOfCustomers = sc.nextInt();
        Supplier<Double> serviceTime = () -> 1.0;
```

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

- **Scanner**: reads input. Check the API for documentation
- **Supplier<T>**: `() -> 1.0` is a lambda expression that takes in no inputs
  - Functional interface like **Function<T, R>** and **Predicate<T>**.
- Unsure what something from the Java library does? Check the API

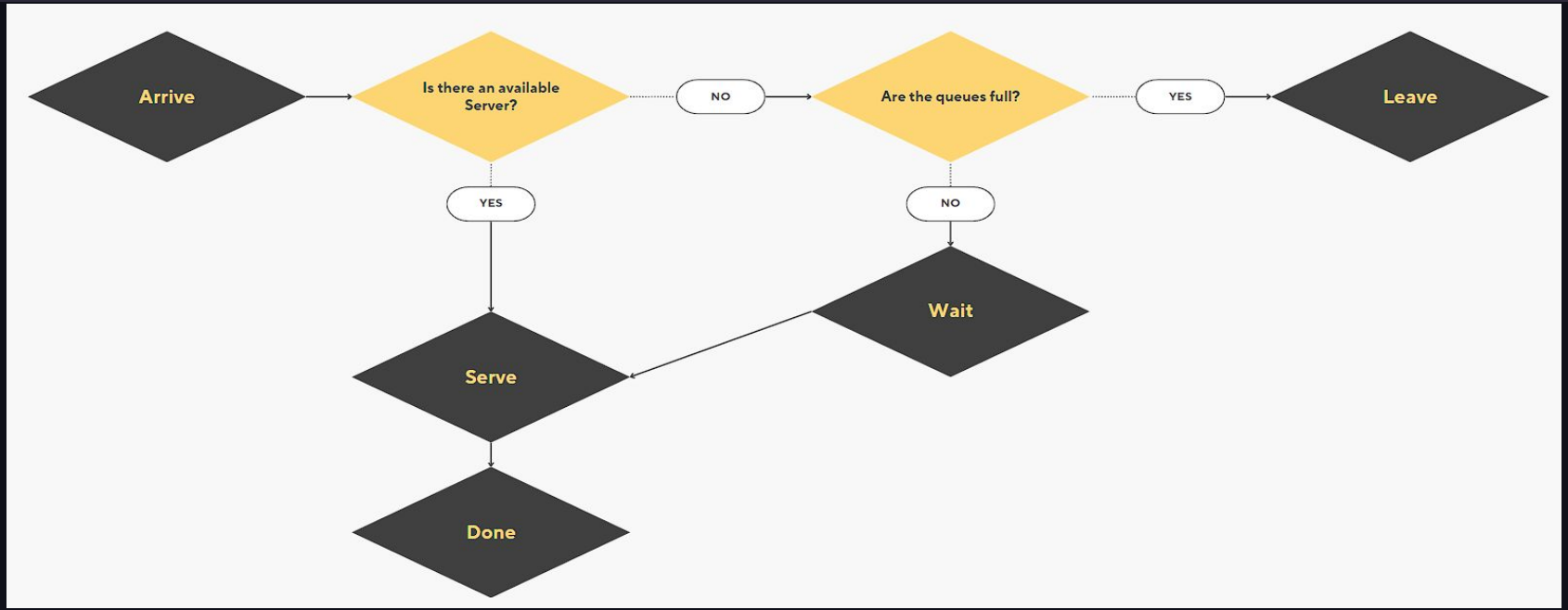
Main

```
new Simulator(numOfServers, qmax, serviceTime, numOfCustomers, arrivals)
    .run()
    .ifPresent(pair -> System.out.println(pair.t() + "\n" + pair.u()));
```

- The returned object from run() has a ifPresent method.
- This means that run returns a **Maybe**<>

Waiting and Queueing

Visualisation: Project



Now **Customer**s can wait if there's a **Server** that they can queue at!

Queueing

Customers can now queue at **Server**s to be served.

Servers now have a maximum queue length.

Maximum queue length **specified in the input**.

Queueing

Given the previous input, the output should look like this:

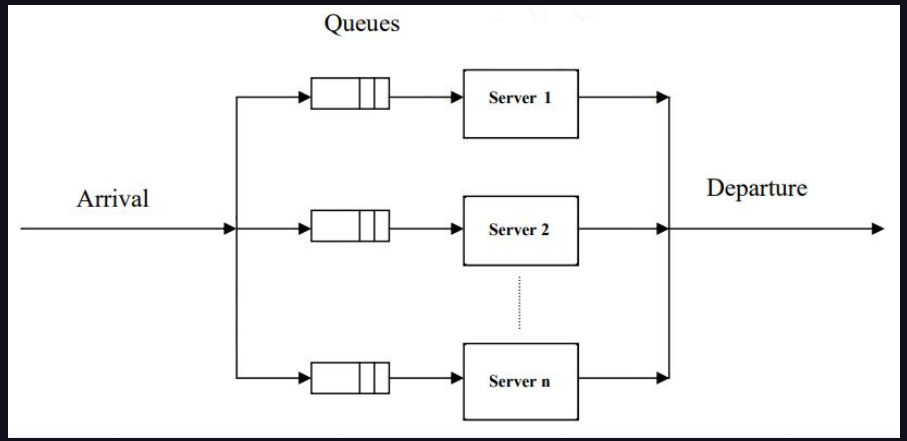
```
1 1 3
0.500
0.600
0.700
```

```
0.500 customer 1 arrives
0.500 customer 1 serves by server 1
0.600 customer 2 arrives
0.600 customer 2 waits at server 1
0.700 customer 3 arrives
0.700 customer 3 leaves
1.500 customer 1 done serving by server 1
1.500 customer 2 serves by server 1
2.500 customer 2 done serving by server 1
```

Note the addition of the **WaitEvent**, and the possibility that a **ServeEvent** may not happen at the same time as the **ArriveEvent**

Queueing

- It is not necessary to use a **List**, **Queue** or **PQ**
- Events are already ordered by a **PQ** somewhere else. Is that insufficient?
- Imagine you are at a polyclinic / hospital waiting for your turn
- Are you in a queue or a straight line waiting for your turn?
- How do you know it is your turn?



Simulation Statistics

We also want to be able to track how the **Simulator** fares for the given input. To do this, we keep track of:

1. the average waiting time for **Customer**s who have been served
2. the number of **Customer**s served
3. the number of **Customer**s who left without being served

```
0.500 customer 1 arrives
0.500 customer 1 serves by server 1
0.600 customer 2 arrives
0.600 customer 2 waits at server 1
0.700 customer 3 arrives
0.700 customer 3 leaves
1.500 customer 1 done serving by server 1
1.500 customer 2 serves by server 1
2.500 customer 2 done serving by server 1
```

using this example output,
the statistic are **[0.450 2 1]**

Simulation Statistics

After tracking the simulation statistics, we simply add it to the last line of the output as such:

```
0.500 customer 1 arrives
0.500 customer 1 serves by server 1
0.600 customer 2 arrives
0.600 customer 2 waits at server 1
0.700 customer 3 arrives
0.700 customer 3 leaves
1.500 customer 1 done serving by server 1
1.500 customer 2 serves by server 1
2.500 customer 2 done serving by server 1
```

```
0.500 customer 1 arrives
0.500 customer 1 serves by server 1
0.600 customer 2 arrives
0.600 customer 2 waits at server 1
0.700 customer 3 arrives
0.700 customer 3 leaves
1.500 customer 1 done
1.500 customer 2 serves by server 1
2.500 customer 2 done

[0.450 2 1]
```

Simulation Statistics

```
0.500 customer 1 arrives
0.500 customer 1 serves by server 1
0.600 customer 2 arrives
0.600 customer 2 waits at server 1
0.700 customer 3 arrives
0.700 customer 3 leaves
1.500 customer 1 done
1.500 customer 2 serves by server 1
2.500 customer 2 done

[0.450 2 1]
```

Simulator's **run**() method now returns a **Maybe<Pair<String, String>>**

The first **String** contains simulation output

The second **String** contains simulation statistics

Floating Point Numbers

```
if (double1 == double2) {
    // do something
}
```

Code looks familiar if trying to compare floating point numbers? (1.1, 3.14 etc)

Floating point numbers are represented differently in computers (you will learn more if you take CS2100), so the above code does not always work!

Floating Point Numbers

```
private static final double THRESHOLD = 1E-15; // 10^-15
…
if (Math.abs(double1 - double2) <= THRESHOLD) {
    // do something
}
```

We need to do something like this instead - Check that the difference between both numbers is smaller than some small threshold value when comparing “equality”

Math.abs takes the absolute (non-negative) value of the number passed into it

(basic)

Debugging Techniques

Compile Errors

```
Main.java:23: error: constructor Simulator in class Simulator cannot be applied to given types;
        State state = new Simulator(numOfServers, numOfCustomers, arrivals, sup).run();
                      ^
  required: int,int,List<Pair<Integer,Pair<Double,Double>>>
  found:    int,int,List<Pair<Integer,Pair<Double,Double>>>,Supplier<Double>
  reason: actual and formal argument lists differ in length
```

- ***Read*** compile errors
  - **Simulator**'s constructor expects 3 arguments with listed types, but 4 were provided
  - This means that either
    - **Simulator**'s constructor needs to take in a **Supplier<Double>**
    - **Main** should stop providing **Simulator**'s constructor with **Supplier<Double>**

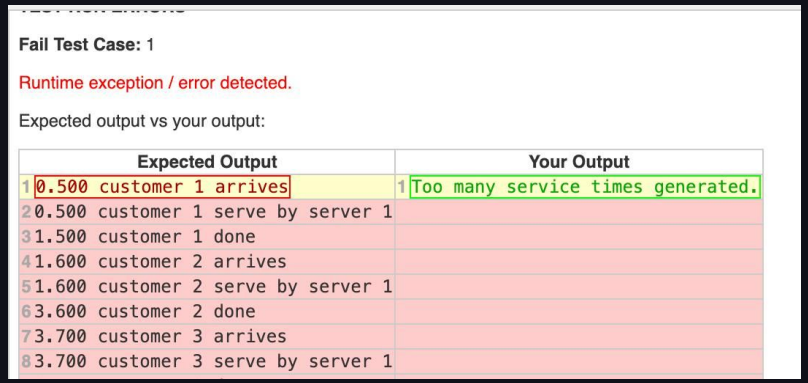
Compile errors

```
State.java:25: error: incompatible types: Maybe<State> cannot be converted to State
        return maybeState;
               ^
```

- ***Read*** compile errors
  - This error states that line 25 of State.java is returning a **Maybe<State>**, but that the method's return type is declared to be **State**
  - This means that either
    - The return type of the method should be changed
    - The return statement of the method should be changed

Too many service times?

```
Exception in thread "main" java.util.NoSuchElementException
        at java.base/java.util.Scanner.throwFor(Scanner.java:945)
        at java.base/java.util.Scanner.next(Scanner.java:1602)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2267)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2221)
        at Main.main(Main.java:8)
```



Fail Test Case: 1

Runtime exception / error detected.

Expected output vs your output:

	Expected Output		Your Output
1	0.500 customer 1 arrives	1	Too many service times generated.
2	0.500 customer 1 serve by server 1		
3	1.500 customer 1 done		
4	1.600 customer 2 arrives		
5	1.600 customer 2 serve by server 1		
6	3.600 customer 2 done		
7	3.700 customer 3 arrives		
8	3.700 customer 3 serve by server 1		

Caused by calling get() too many times. If only 3 customers are served, then only 3 service times are provided. Calling get() more than 3 times will cause an error because there are no more service times available

Too many service times?

- **Potential debugging strategies**
  a. Add a print statement to the **Supplier**.
     - Supplier prints something every time `get()` is called
     - Visually track how many times `get()` is called

  b. Print **State**'s **PQ<Event>** to console to check if **Event**s are being correctly added and polled

Too many service times?

```
Supplier<Double> serviceTime = () -> {
        System.out.println(“generating service time...”);
        return 1.0;
    }
```

Debugging techniques

- Come up with your own test cases to test specific parts
- Set **Customer** arrival times to exactly what you want to test
- E.g.
  - Serve immediately after done, 0.5 and 1.5
  - Exceed queue length, 0.1 0.2 0.3 0.4

Debugging techniques

- If a file has 100 lines and there is a bug
  - ***Binary Search***
  - Check intermediate variables at line 50.
    - If it is correct, the bug exists within lines 51 - 100.
    - Otherwise, the bug exists within lines 1 - 50
  - Repeat until you find the buggy line

Debugging techniques

- Test your code ***progressively***
- If you are coding classes **A, B, C, D, E**, and you only test at the end and find a bug - isolation is difficult
- If you code **A**, test **A**, code **B**, test **B**, and code **C** and find something wrong at the test of **C**, then the bug is at **C**

Project

Deadline: **23 Oct (Thurs) 2359**