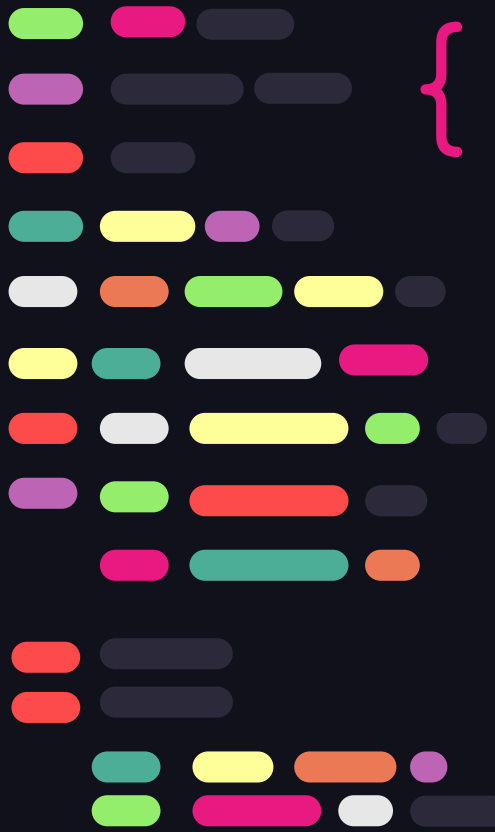


CS2030
Lab 2

AY25/26 Sem 1, Week 4

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

Lab Admin

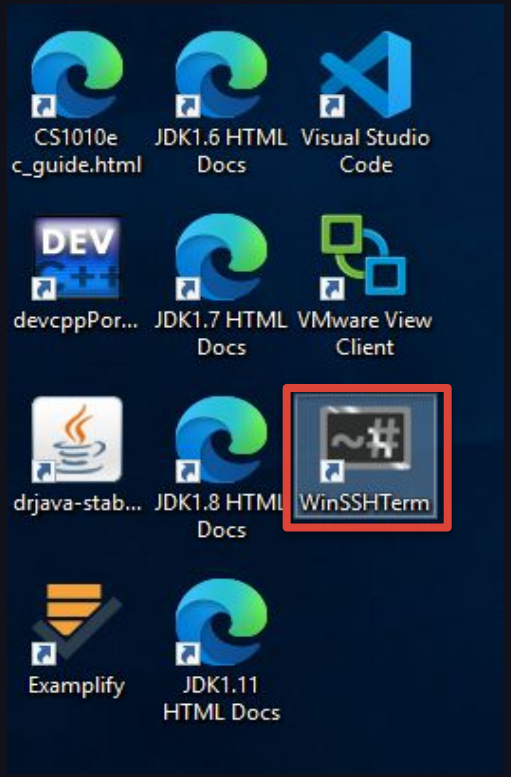
Admin Stuff

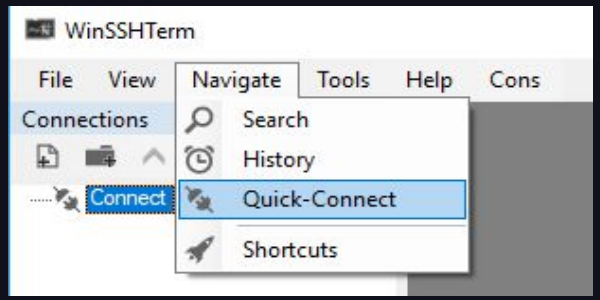
Log in to the lab device

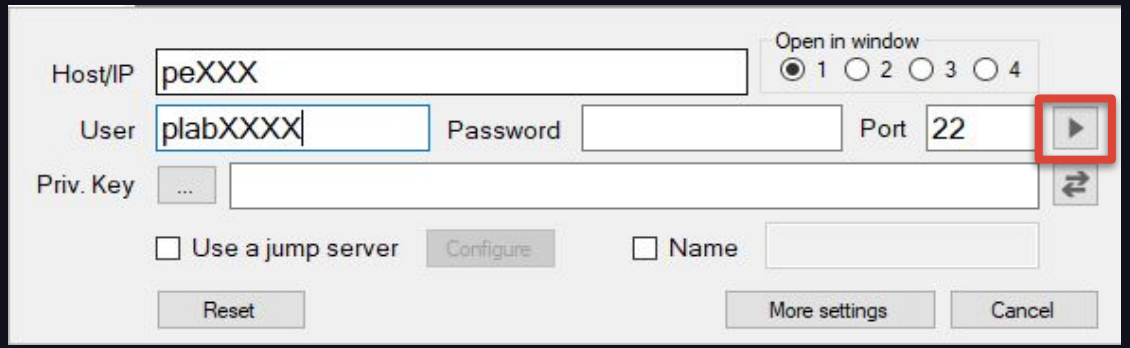
Username: nusstu\exxxxxxxx (e.g. nusstu\e1234567)
Password: <your canvas password>

Make sure that you are logged into **your** account and not someone else's, or you will be marked absent!

Connecting to PE Node

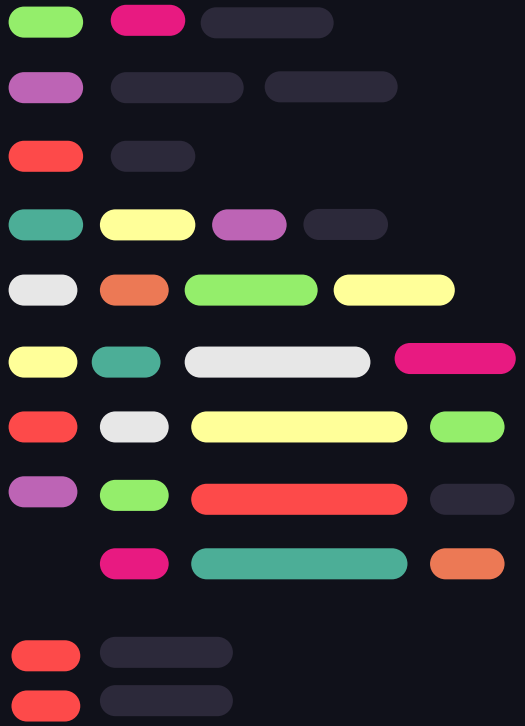






Gen AI Usage

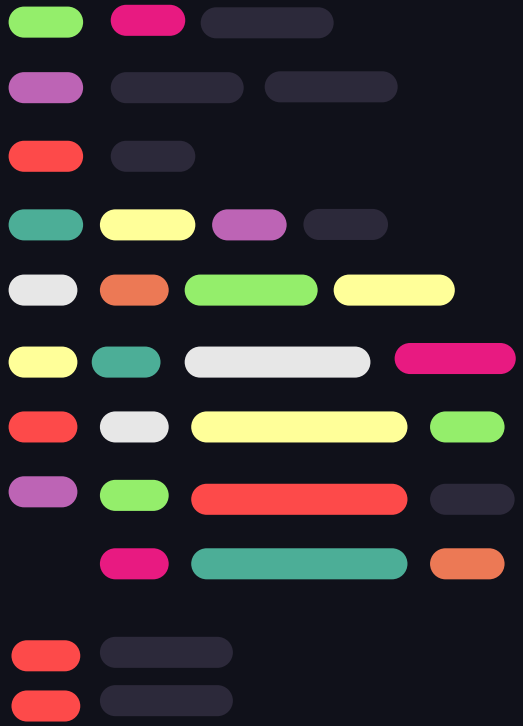
Use of Gen AI



Gen AI is a tool

- Use it to help with your learning, not do your homework for you. If it does your homework better than you, it can take your job too
- **Question** it on concepts you're **unsure** of
- **Challenge** it with **"why"/"why not"** questions *(e.g. "Why shouldn't I do X")*
- Get **help** with problem solving *(e.g. "What Stream method can I use to…")*

Use of Gen AI



Always attempt lab exercises on your own.

- Trains you to come up with **solutions**
- **Reinforces** concepts learnt in CS2030
- Prepares you for **PAs** and **Finals**

Use of Gen AI

Bad Prompts

- “Give me the code for this question”
- “Convert this for-loop into a stream”
- **pastes error message without reading** “Fix my code”
- **pastes lab document** “Do this for me”
  Please don’t do this, we can tell.

Use of Gen AI

Good Prompts

- "I’m considering approach A and approach B for solving X. Which is better and why?”
- “What are some Java Stream methods I can use to add indices to a stream of Strings?”
- “I get a NullPointerException when I do X. What is causing this?”
- “Does accessing an attribute of another instance of the same class violate the Tell-Don't-Ask principle?”

Use of Gen AI

Why is this important?

- **No access** to online resources during PAs and Finals
- Trains you to **write code** from **scratch** (Needed for PAs)
- More practice with **solving** various types of problems
- **Less time wasted** referencing notes during assessments
- ChatGPT is not good at CS2030

Use of Gen AI

Date Submitted
30 Aug 2025 14:27:32
30 Aug 2025 14:17:26
30 Aug 2025 02:50:54
30 Aug 2025 02:48:26
30 Aug 2025 02:39:07
30 Aug 2025 02:34:37
30 Aug 2025 02:27:55
30 Aug 2025 02:24:03
30 Aug 2025 02:18:28
30 Aug 2025 02:11:40
30 Aug 2025 01:52:55
30 Aug 2025 01:46:56
29 Aug 2025 23:42:13
29 Aug 2025 23:41:10
29 Aug 2025 23:36:10
29 Aug 2025 23:33:46
29 Aug 2025 23:32:05
29 Aug 2025 23:31:20
29 Aug 2025 23:29:12
29 Aug 2025 23:27:38
29 Aug 2025 23:26:46
29 Aug 2025 23:06:21
29 Aug 2025 23:03:17
29 Aug 2025 16:22:20

A reminder

Finals	40%
Individual Project	15%
Practical Assessment 1	15%
Practical Assessment 2	20%
Labs, self-practice exercises, class participation and peer learning activities	10%

Implementation Considerations

Magic Numbers

```
int numberOfMinutes = numberOfSeconds / 60;
```

Guessable that 60 is the number of seconds in a minute, but you only knew because of contextual knowledge (will not apply to other projects)

Thus, we refer to 60 as a magic number as prior context is required to understand the code.

We try to avoid magic numbers to make our code more readable.

Magic Numbers

```
private static final int NUMBER_OF_SECONDS_IN_ONE_MINUTE = 60;
…
int numberOfMinutes = numberOfSeconds / NUMBER_OF_SECONDS_IN_ONE_MINUTE;
```

We give magic numbers meaning by assigning them to constants

These variables typically have the `static` and `final` keywords, and are canonically written in UPPER_CASE (all caps, words separated with underscores)

Bonus: You only need to change one value if used in multiple places!

Floating Point Numbers

```
// Don’t need to know for this lab, but good to know for Ex 1

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 (more context: CS2100), so the above code will 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

Recap

Optional

The Optional class is a useful abstraction to deal with null values

Optional<T> defines an Optional that wraps around type T

e.g. Optional<Integer> defines an Optional that wraps around an Integer, etc.

Optional

```
// Creates an Optional<Integer>, encapsulating 1
Optional.<Integer>of(1);

// Creates an Optional<String>, encapsulating “Hello World!”
Optional.<String>of(“Hello World!”);
```

Optionals

We will go over the map, flatMap, filter methods. These are analogous to the stream methods with the same name.

Optional - map

The map function applies a function to the value inside the Optional (if any), and wraps the result of the function in a new Optional

Optional: map

```
// Maps Optional(1) to Optional(2)
Optional.<Integer>of(1).map(x -> x + 1);

// Maps Optional(1) to Optional("11")
Optional.<Integer>of(1).map(x -> "1" + x);

// Maps Optional(1) to Optional(Optional(1))
Optional.<Integer>of(1).map(x -> Optional.of(x));
```

Optional - flatMap

You use this when the function you are trying to apply on the value already returns an Optional

With a normal map, you would have Optional<Optional<value>>, since map wraps the result of the function in another Optional

Optional: flatMap

```
// Maps Optional(1) to Optional(2)
Optional.<Integer>of(1).flatMap(x -> Optional.of(x + 1));

// Maps Optional(1) to Optional(1)
Optional.<Integer>of(1).flatMap(x -> Optional.of(x));
```

Optional - filter

`Optional<T>`	**`filter`**`(`**`Predicate`**`<? super T> predicate)` If a value is present, and the value matches the given predicate, return an `Optional` describing the value, otherwise return an empty `Optional`.

The filter method applies a condition (Predicate) to the value inside the Optional

Optional - filter

`Optional<T>`	**`filter`**`(`**`Predicate`**`<? super T> predicate)` If a value is present, and the value matches the given predicate, return an `Optional` describing the value, otherwise return an empty `Optional`.

If the value is present and matches the predicate, it returns the Optional of that value

Otherwise, it returns an empty Optional.

Optional - filter

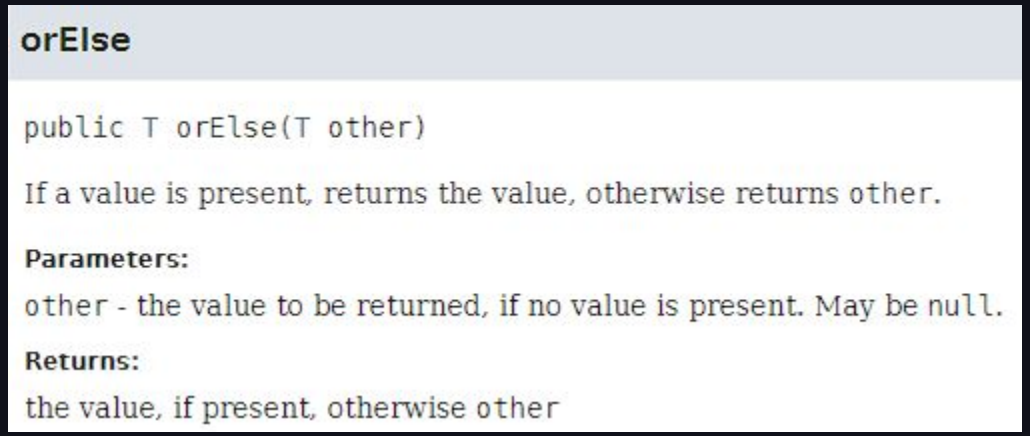
```
Optional<Integer> opt = Optional.of(15);
Optional<Integer> moreThanTen = opt.filter(val -> val > 10);
Optional<Integer> lessThanTen = opt.filter(val -> val < 10);
```

What would the result of moreThanTen and lessThanTen be?

Optional - orElse

The orElse method returns the value if present in the Optional, else returns a specified value of the same type instead.

Think of it as the “else” part of an “if-else” statement



orElse

`public T orElse(T other)`

If a value is present, returns the value, otherwise returns `other`.

Parameters:

`other` - the value to be returned, if no value is present. May be `null`.

Returns:

the value, if present, otherwise `other`

Optional - orElse

```
Optional<Integer> opt = Optional.of(15);

Integer value1 = opt.filter(x -> x > 10).orElse(1);
Integer value2 = opt.filter(x -> x < 10).orElse(2);
```

What would the result of value1 and value2 be?

isPresent & get

Consider these code snippets. Is there any point in using Optionals like this?

```
void greet(String name) {
    if (name == null) {
        System.out.println(“Please enter your name”);
    } else {
        System.out.println(“Hello ” + name);
}
void greet(String name) {
    Optional<String> opt = Optional.ofNullable(name);
    if (!opt.isPresent()) {
        System.out.println(“Please enter your name”);
    } else {
        System.out.println(“Hello ” + opt.get());
    }
}
```

orElse

If we cannot use get, then how do we take the value out of Optional?
Introducing orElse:

```
void greet(String name) {
    System.out.println(Optional.ofNullable(name)
        .map(x -> “Hello ” + x)
        .orElse(“Please enter your name”));
}
```

If a name is given, it will be mapped to Hello + name and returned.
Otherwise, the value specified in the orElse call be be returned.

orElse

```
void greet(String name) {
    String output = Optional.ofNullable(name)
        .map(x -> “Hello ” + x)
        .orElse(“”);
    if (output.equals(“”)) {
        System.out.println(“Please enter your name”);
    } else {
        System.out.println(output);
    }
}
```

This is a not how orElse should be used. Since orElse already functions as an if-else statement, the rest of the function does not require any if-else statements anymore.

Maybe<T>

- The Maybe<T> class is provided for you. It is similar to Optional
- Since we are using Maybe, there should be no null in your code
- Only a subset of Optional’s commands are provided

InfList<T>

- InfList is another provided class
- It has the same functionality as Stream, but streams can only be operated upon once, after which it will be closed. InfList allows you to operate on it as many times as you like

Maybe<T> & InfList<T>

For this lab, you will access the question through CodeCrunch on your laptops. Under level 0, you will see this command to generate HTML documentation for the provided Maybe & InfList classes. Download the 2 class files and run the command on your laptop to generate the documentation to know what methods are available to you, but continue to code on the lab PCs for practice.

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

```
$ javadoc -d doc Maybe.java InfList.java
```

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

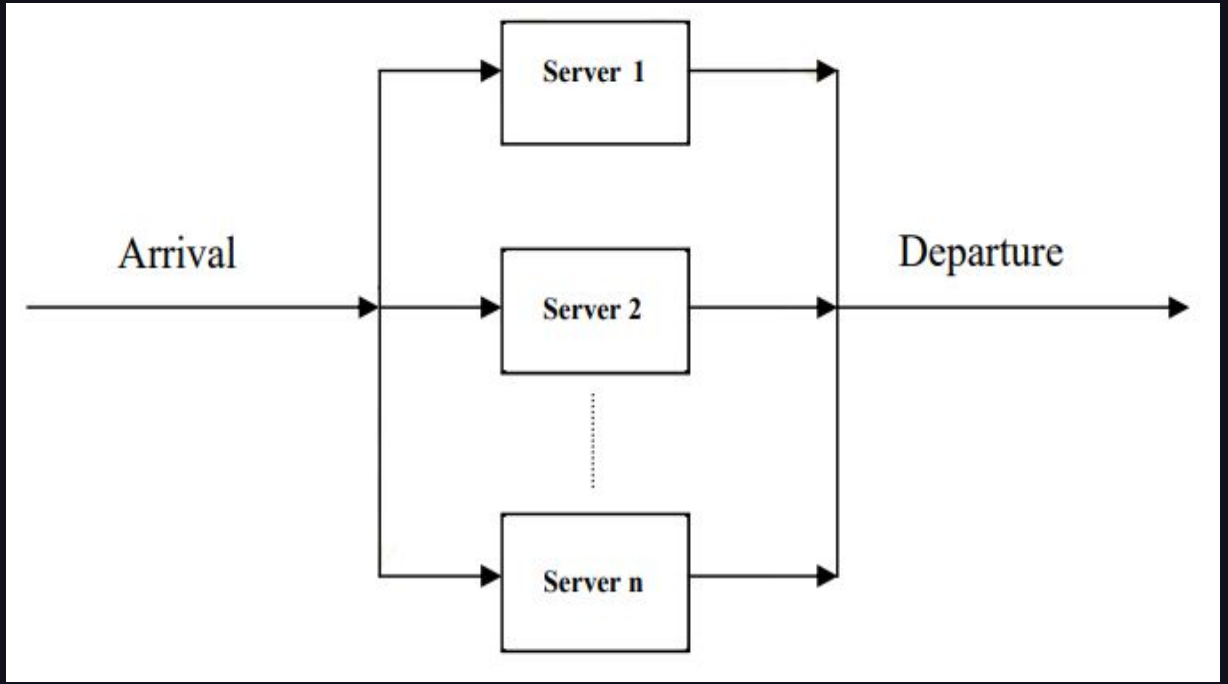
Lab 2
Project Part 1

Task

Simulate how Customers are served by servers.

Customer that arrives will look for the first available server, and he/she will be served for some time.

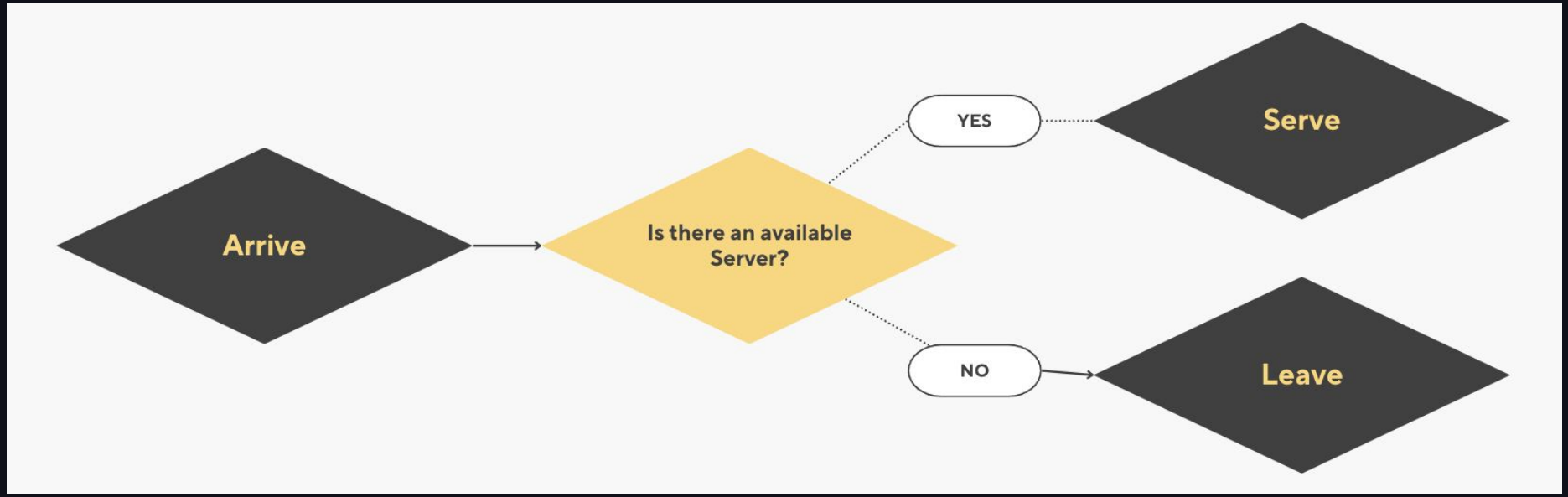
If all servers are busy, customer will just leave.



Specifics

- There are **n** servers and each server can only serve one customer at a time
- Each customer has a service time (time taken for the server to complete servicing the customer)
- Customer will scan servers from **1…n** and try to find an available server
  - If a server is able to serve the customer, it will serve the customer **immediately**
  - If no server is available, they will leave

Visualisation



Task Overview

- Customer class
- Server class
- Shop class - encapsulate a list of servers
- State class - to represent a state (or step) of the simulation

Customer

- Customers, identified by an `int`, will arrive at a certain timing (represented by a `double`)
- has a `canBeServed(time)` method that checks if the Customer has already arrived and can be served
- also has a `serveTill(serviceTime)` method that returns the time that the Customer will be finished, given the amount of time needed for service

Note

Focus on a tell-don’t-ask principle when designing your code...

Avoid exposing your attributes!

Server

- Servers, containing an int identifier, may only serve one Customer at a time, and is always available starting from time 0.0
- Need to manage timing (in order to know if the Server can serve a Customer)
- has a canServe(cust) method to determine if the Server is available to serve a given Customer, and a serve(cust, svcTime) that serves the customer for a given service time

Shop

- Where we manage the Servers (note that there can be no Servers)
- has a `findServer`(cust) method that finds the first server in the shop that can serve the customer
  - since there may be no (available) Servers, the method should return an Optional<Server>
- has an `update`(updatedServer) method that updates the old server with the updatedServer

State

- Represents a state (or step) of the Simulation we are modelling
  - e.g. Arrive/Serve/Leave
- Will be how you manage between states
- Will also be where you generate your outputs

Simulator

- Provided to you, used to run the Simulation with different params
- These slides are to explain to you how the Simulator works so you can better code the State - you do not need to code the Simulator
- In the run method, we start with an initial State along with an iterator of Customers
- Then a Stream of States are created with the State's next method that takes in a Customer which generates the next State
- We map each State into its toString() before reducing them into one final output

Simulator

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

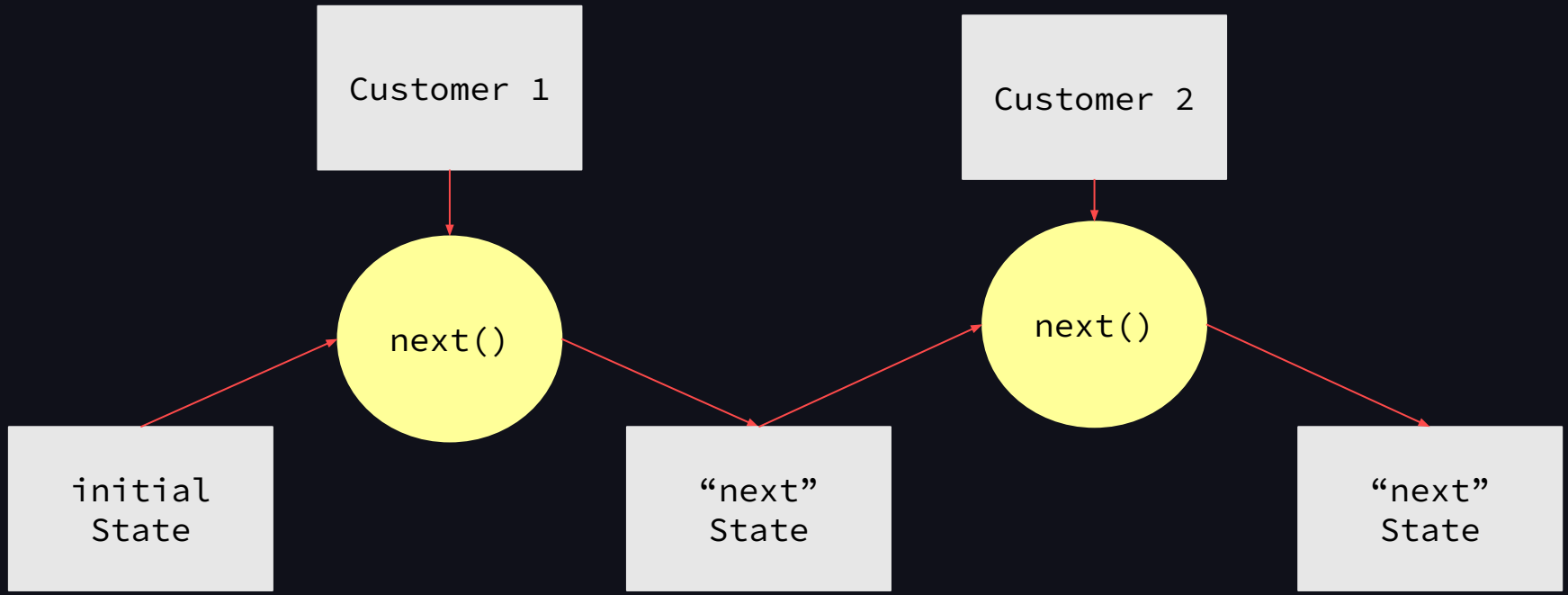
- The initial state is created as init
- State.next(customer) is repeatedly called starting with the initial state

State

Your task is to design the State class which has a next(cust) method that takes in a customer and combines the current state and the current customer to produce the next State

State

Visualising how the states reduce into the final state



Outputs

```
jshell> System.out.println(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
```

Example 1: All Served

```
$ cat 1.in
3 3              // Number of servers and customers
1 0.500          // Customer ID, Customer Arrival Time
2 0.600          // We are assuming a 1.0 service time
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
```

Example 2: Customer Leaves

```
$ cat 3.in
2 3             // Number of servers and customers
1 0.500         // Customer ID, Customer Arrival Time
2 0.600         // We are assuming a 1.0 service time
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
```

Tips

Focus on modelling the solution as a proper Object-Oriented solution:

- **Abstraction:**
  - Think about how to implement the solution using low-level data and methods
  - Keep in mind that clients will only use the high-level data type and methods
- **Encapsulation:**
  - Think about how to structure your solution such that it hides information/data from the client and only allowing access through methods provided by the implementor

Tip

- Unsure how to achieve some sort of behaviour? Stare at the API, maybe you'll find something useful...

Deadline

Deadline: **11 Sep (Thurs) 2359**