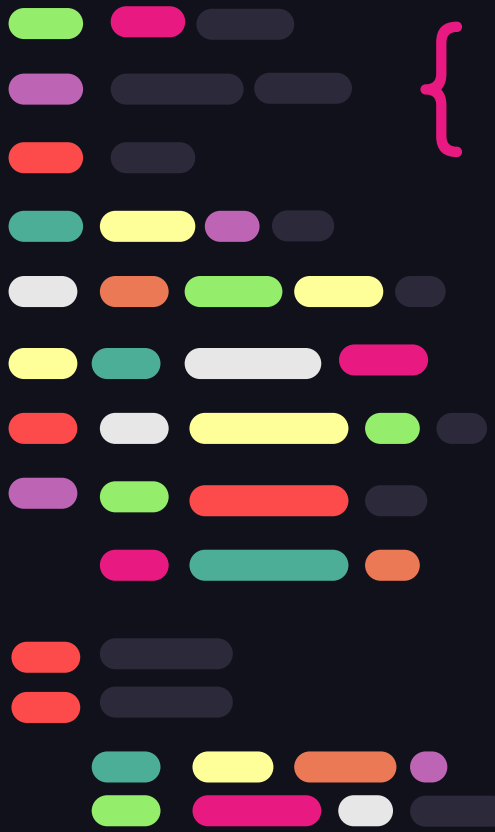


CS2030
Lab 2

AY24/25 Sem 2, Week 4

Fadhil Peer Mohamed <f_p_m@u.nus.edu>
Pang Yang Yi <pang.yy@u.nus.edu>

Lab Admin

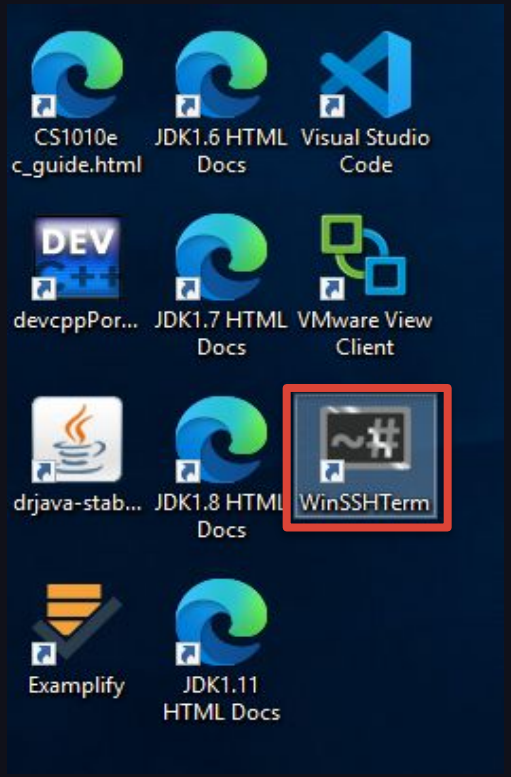
Admin Stuff

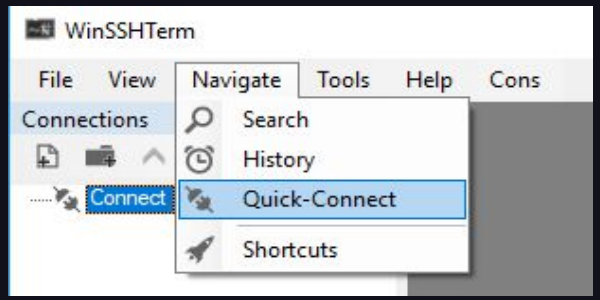
Log in to the lab device

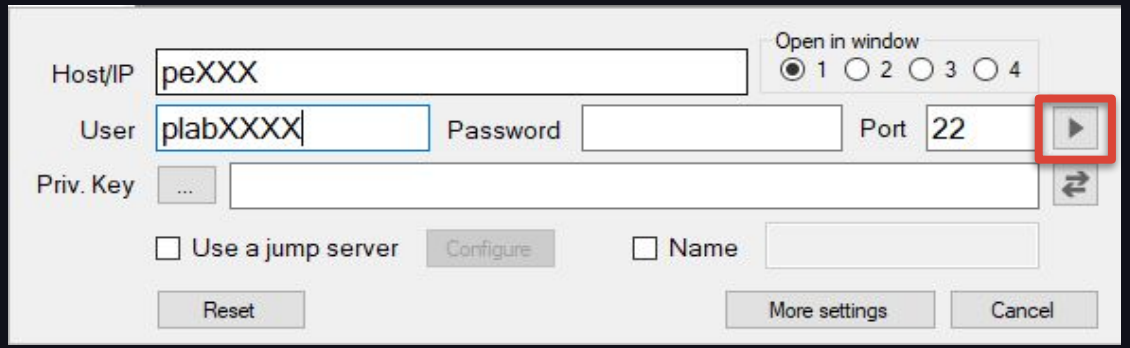
Username: nusstu\exxxxxxxx (e.g. nusstu\e1234567)
Password: <your canvas password>

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

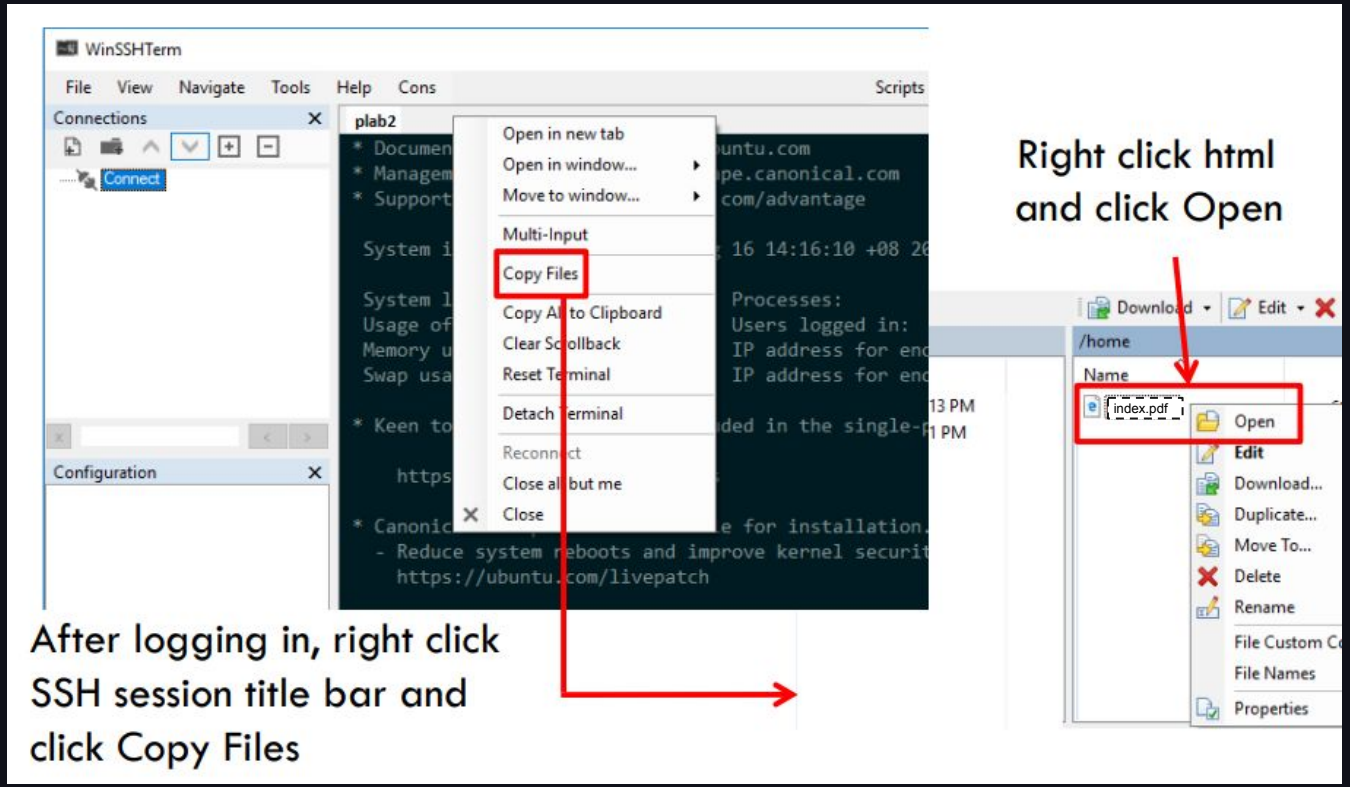
Connecting to PE Node







Viewing Questions



Questions are in the file labelled **index.pdf**

Coding Style

CS2030 enforces a rigid coding style. Deviating from this style (even by a little) will result in an instant F on CodeCrunch, no matter the correctness of your code.

This prepares you to work in a software engineering teams.

If everyone on the team follows the same style, the intent of the programmer can become clear (e.g., is this a class or a field?), code becomes more readable and less bug prone.

Coding Style

What is the CS2030 style, exactly?

Read more at https://www.comp.nus.edu.sg/~cs2030/style/

Can we automate this?

Checkstyle

Checkstyle is a tool that checks the compliance of certain files of code to a given coding standard. Coding standards are provided via *.xml files.

Setup, usage instructions, and CS2030's coding standard in XML form can be found at:
https://canvas.nus.edu.sg/courses/69900/pages/style-check

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 Java (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

Recap

Optional

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

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

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

Optionals

We will be revisiting the `map`, `flatMap`, `filter` and `orElse` methods

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 Optional.empty.

Optional - filter

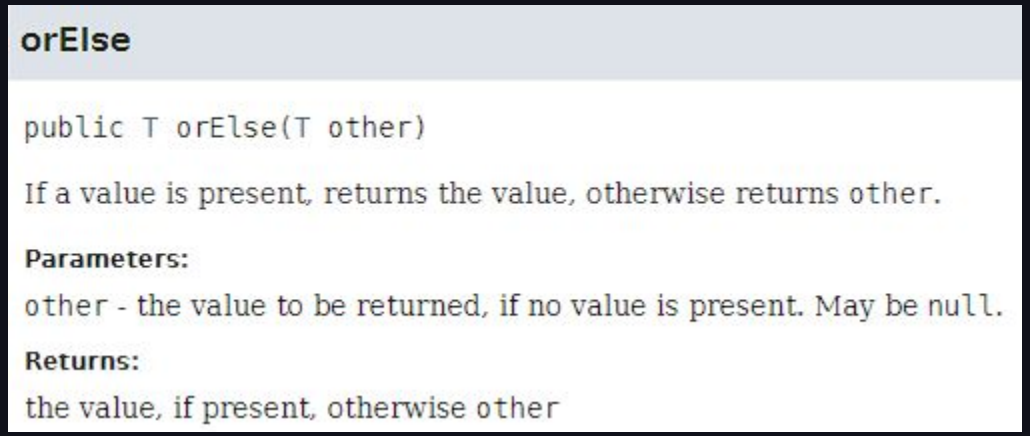
```
Optional<Integer> optInt = Optional.of(15);
Optional<Integer> moreThanTen = optInt.filter(val -> val > 10);
Optional<Integer> lessThanTen = optInt.filter(val -> val < 10);
```

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> optInt = Optional.of(15);

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

What would the result of value1 and value2 be?

Restrictions

- Since we're working with Optionals, the following methods are not allowed:
  - isPresent
  - isEmpty
  - get
  - equals
  - hashCode

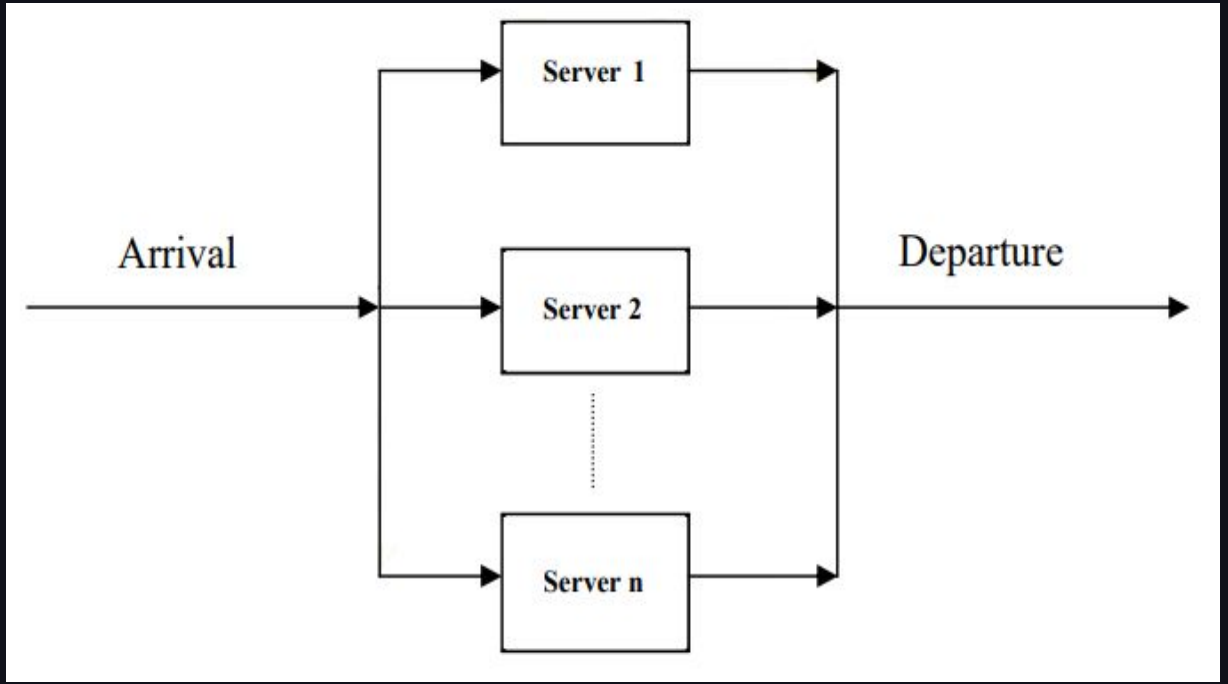
Lab 2
Project Part 1

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.

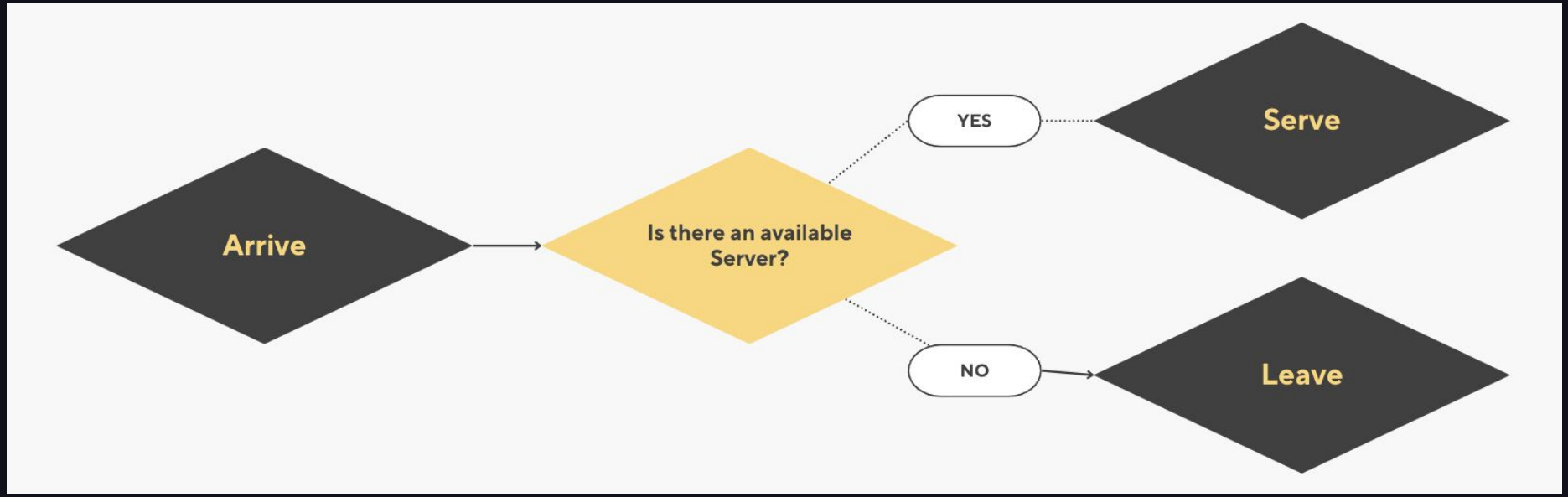
If all servers are busy, customer will just leave.



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, identified by an int, 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

```
String run() {
    State init = new State(new Shop(numOfServers));
    List<Customer> customers = arrivals.stream()
        .map(x -> new Customer(x.t(), x.u())).toList();
    Iterator<Customer> iter = customers.iterator();

    return Stream.<State>iterate(init, state -> state.next(iter.next()))
        .limit(numOfCustomers + 1) // including start state
        .map(state -> state.toString())
        .filter(state -> !state.isEmpty()) // ignore state with no simulation output, e.g. init state
        .reduce("", (x, y) -> x + y + "\n");
}
```

initial State

Customer iterator

generating States

reduction of toString()s

- The Stream has numCust + 1 States (since it includes the start)
- States are mapped to their toString()s before being reduced down into a single output

Simulator

```
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;
    }
```

Time it takes to serve a Customer

- It will also tell you the serviceTime - how long it takes to serve the Customer

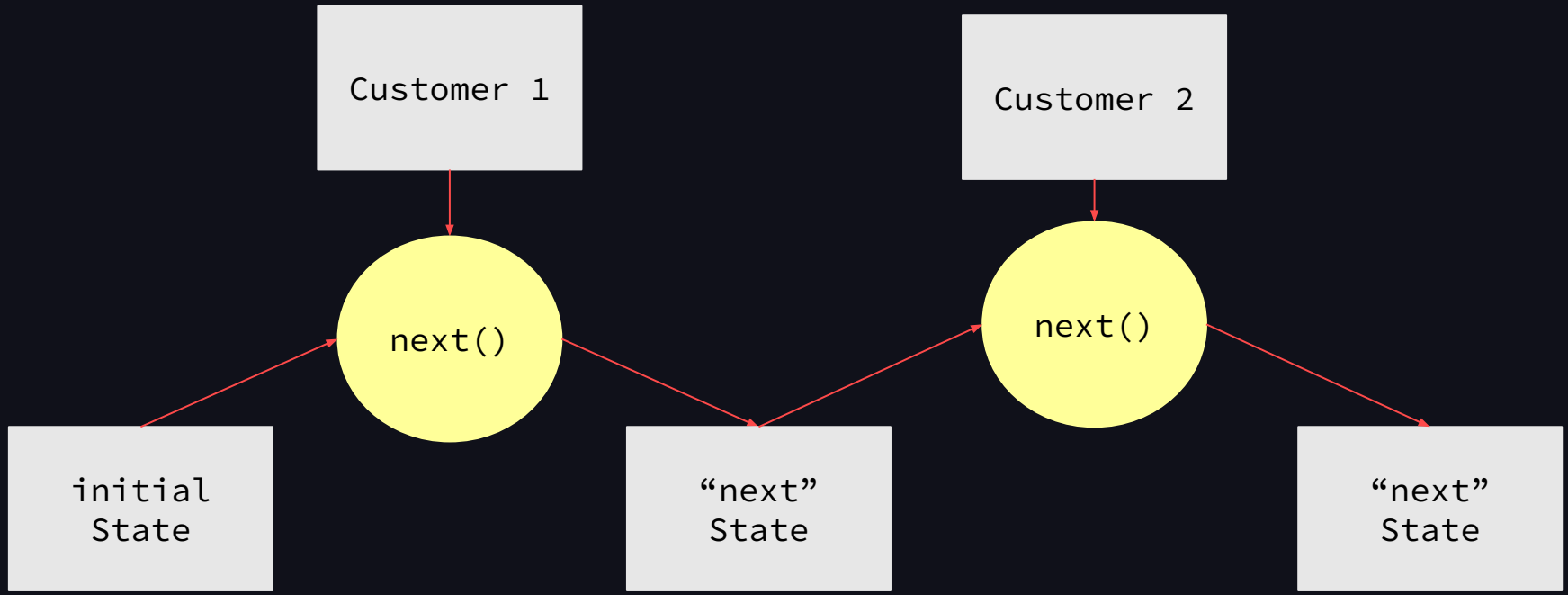
State

- Main task is to design the next() method to simulate through the States for each Customer
- Note that the Stream's reduction starts with an "empty" or "fresh" State, and the next() method takes in a Customer (iter.next() returns the next Customer)

```
return Stream.<State>iterate(init, state -> state.next(iter.next()))
```

Tips

Visualising how the Stream's iterate flow works



State

```
jshell> new State(new Shop(2))
$.. ==>

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

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

jshell> new State(new Shop(2)).next(new Customer(1, 1.0)).next(new Customer(2, 1.5))
$.. ==> customer 2 arrives
customer 2 served by server 2
```

- Note that that the `next()` method returns the next State that contains both the arrive and serve outputs in its `toString` at once.

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

Notes

- Unsure how to achieve some sort of behaviour? Stare at the API, maybe you'll find something useful…
- Use .jsh to open all the files

Deadline

Levels 1-3: **13 Feb (Thurs) 2359**

Levels 4-5: **20 Feb (Thurs) 2359**

The extended deadline is to allow you more time to clarify any doubts with your implementation. Please do not wait until the last minute to start level 4!

Deadline

Exercise 1: **02 Mar 2025**