CS2030

Lab 4

AY24/25 Sem 2, Week 8

Fadhil Peer Mohamed <f_p_m@u.nus.edu>

Pang Yang Yi <pang.yy@u.nus.edu>

Lab Admin

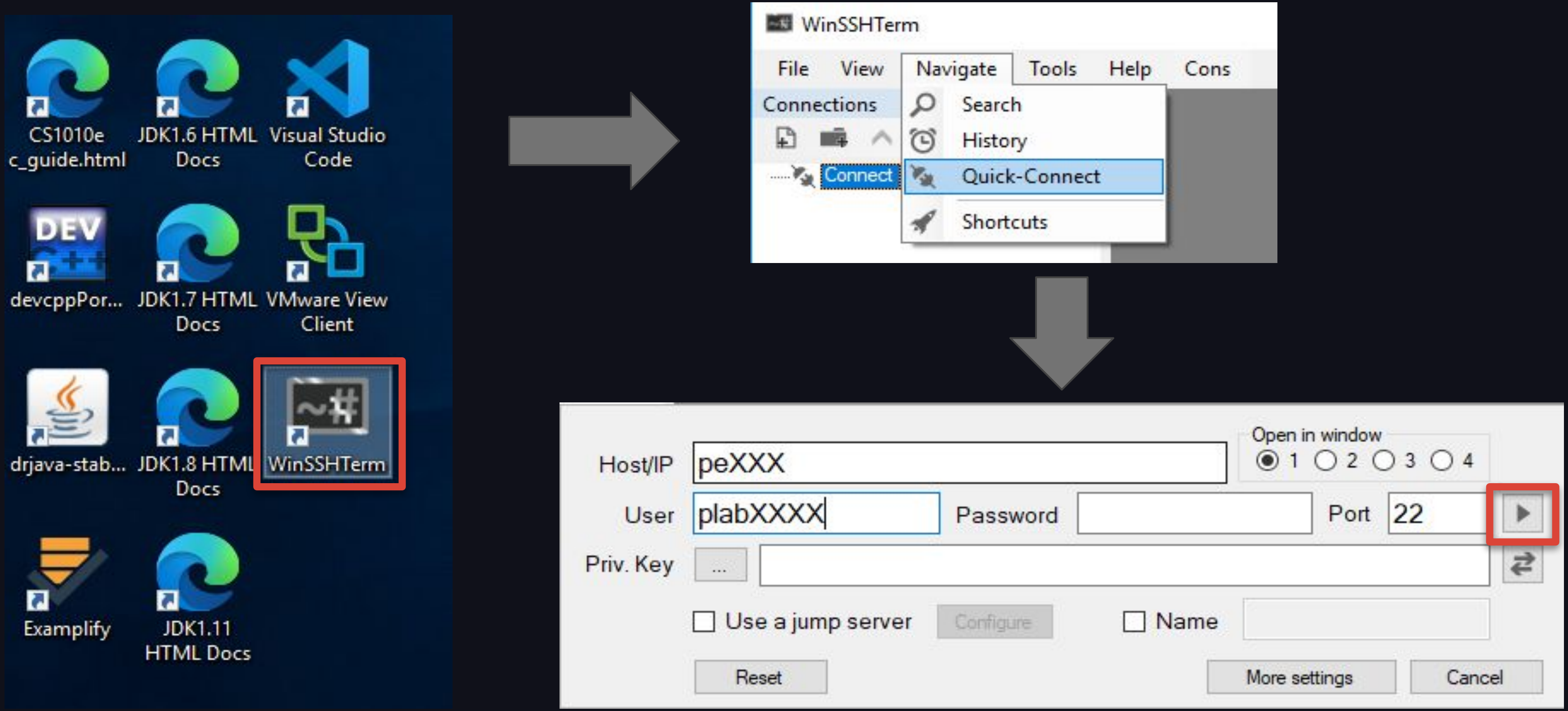
Admin Stuff

Log in to the lab device

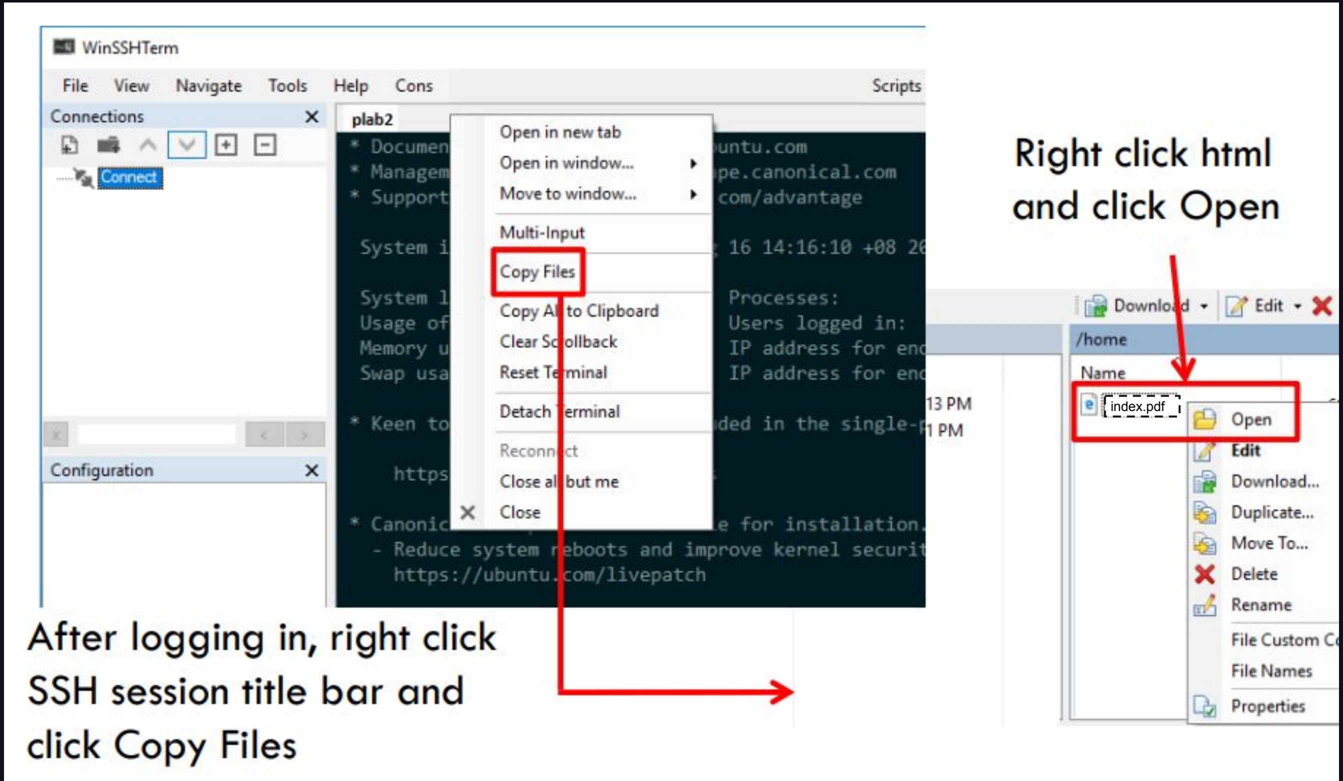
Username: nusstu\e{0/1}xxxxxxxx
Password: <your nusstu password>

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**

Timeline

WEEK	LAB	DUE
8	4 (Last Project Milestone)	
9	5	*none! :)*
10	Well-being Day	5 - 27 Mar 2359 (Thur)
11	Mock PA#2	Project - 6 Apr 2359 (Sun)
12	PA#2	Mock PA#2 - 10 Apr 2359 PA#2 Moderation - 25 Apr 2359 (Last day of reading week)

Teaching Interest

Let us know if you’re interested in becoming a TA for the upcoming semester.

The following experience is desirable in a TA application:

- Either have completed a computing-related internship or
- Have completed SOC’s Orbital Programme

Recap

Access Modifiers

If a method/constructor is only used in the class, it is good practice to make the method/constructor private.

For example, private constructors in **View**, constructors that help you “update” object instances, or helper methods that help you compartmentalise implementations.

Compartmentalisation

```java
1 usage
String run() {
    Shop initShop = new Shop(numOfServers);

    // generate list of arrivals
    List<ArriveEvent> arrivalEvents = arrivals.stream() Stream<Pair<...>>
            .map(arrival -> new ArriveEvent(new Customer(arrival.t(), arrival.u().t(),
                    arrival.u().u()), arrival.u().t())) Stream<ArriveEvent>
            .toList();

    PQ<Event> initPQ = new PQ<~>(arrivalEvents);

    State initState = new State(initPQ, initShop);

    // generate output string
    String output = Stream.iterate(initState,
                    state -> !state.isEmpty(),
                    state -> state.next().get()) Stream<State>
            .map(state -> state.toString()) Stream<String>
            .filter(str -> !str.isEmpty())
            .reduce( identity: "", (x, y) -> x + y + "\n");

    return output;
}
```

Consider this example implementation of the **Simulator**

Notice how the **run**() method becomes rather long even though it is only effectively doing two things?

Compartmentalisation

```
1 usage
private List<ArriveEvent> initArrivalList() {
    return arrivals.stream() Stream<Pair<...>>
            .map(arrival -> new ArriveEvent(new Customer(arrival.t(), arrival.u().t(),
                    arrival.u().u()), arrival.u().t())) Stream<ArriveEvent>
            .toList();
}

1 usage
private String simulateOutputString(State initState) {
    return Stream.iterate(initState,
                    state -> !state.isEmpty(),
                    state -> state.next().get()) Stream<State>
            .map(state -> state.toString()) Stream<String>
            .filter(str -> !str.isEmpty())
            .reduce( identity: "", (x, y) -> x + y + "\n");
}
```

We can use private helper methods like this and compartmentalise the **run**() method’s implementation

Compartmentalisation

With the new private methods and some cleanup…

```java
String run() {
    // generate list of arrivals
    List<ArriveEvent> arrivalEvents = initArrivalList();

    State initState = new State(new PQ<Event>(arrivalEvents), new Shop(numOfServers));

    return simulateOutputString(initState);
}
```

See how much easier it is to read?

Compartmentalisation

```java
String run() {
    // generate list of arrivals
    List<ArriveEvent> arrivalEvents = initArrivalList();

    State initState = new State(new PQ<Event>(arrivalEvents), new Shop(numOfServers));

    return simulateOutputString(initState);
}
```

This also allows for easier debugging:

If something fails, you can immediately check its specific function and debug from there

Any changes you make will also be isolated to that specific part without affecting the rest of the code

Project Design Considerations

Event Priority

Event priority should be determined by order of eventTime, breaking times by Customer arrivalTimes, as stated in the previous project lab:

You should implement Comparable for Customers to achieve this as well. Any other form of comparisons will be penalised.

Implicit Typechecking

OOP places a strong focus on **Polymorphism**, where we design methods to have different behaviours.

This is part of abstraction so as to not reveal too many details to the client.

This also means that the function calls to Events should be the same, but each Event should behave differently

Implicit Typechecking

Ideally, you **should not be using an additional attribute** such as eventPriority or eventType to help your State determine which Event it is.

Methods that return a String or Integer to help the State determine the exact Event are also undesirable.

Inheritance

Instead of defining the same common attributes for all your Events (e.g. eventTime, Customer), you can define them in the base Event class and simply use super()

This also applies to methods like compareTo() where method behaviour is common throughout the subclasses

Grading

Incorrect determination of Event Priority and instances of Implicit Type Checking are considered design violations, and **will be penalised**

Design will be graded manually, so to get a good individual project grade (worth 15% of your overall course grade), you need to get an A with good design.

Plagiarism

Friendly reminder:

Projects should be done **independently**.

We **will** be checking for plagiarism.

Lab 4

Task Overview

Introduction of:

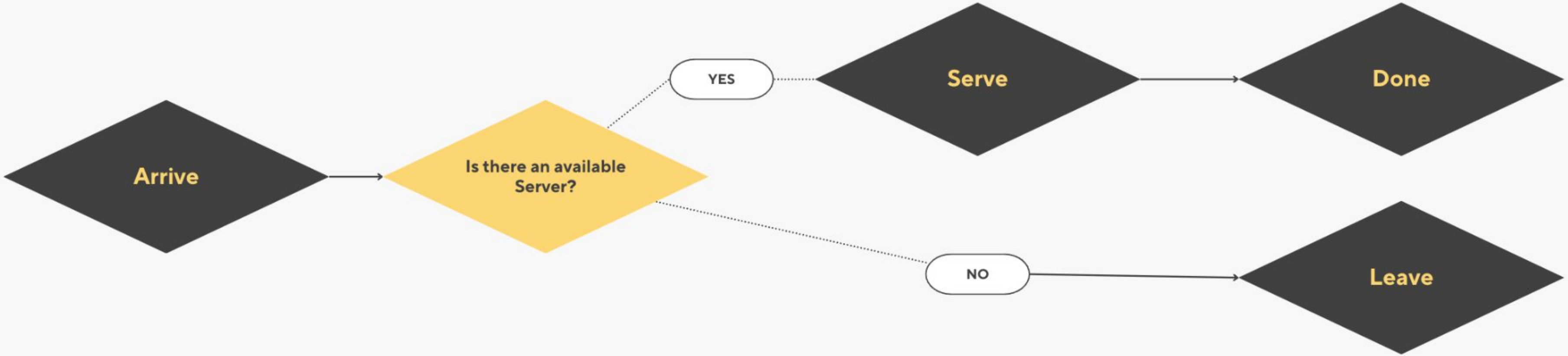
- **Queues** to each Server
- **Wait** Event
- Simulation Statistics
- On-Demand Service Timings

Disclaimer

Slides might not be entirely accurate with regards to sample runs shown later as changes might have been made to the index.pdf file.

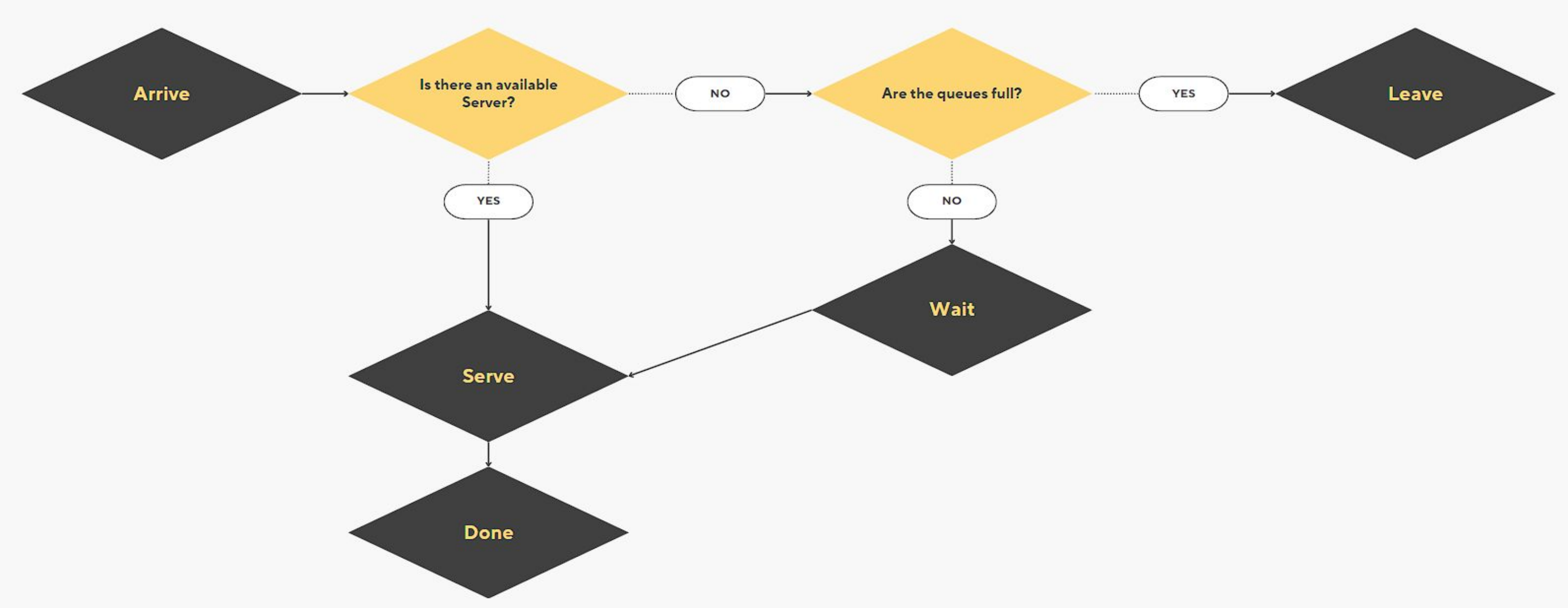
Treat these as an example to understand the flow of how things should work, and refer to your index.pdf in case of updates.

Visualisation: Lab 3



Remember this?

Visualisation: Lab 4



Now Customers can Wait if there's a Server that they can queue at!

Server Queues and Wait

Like how we queue for everything and anything, Customers can also queue at Servers to be Served.

Each Server should now have a queue at which Customers can queue at, with a maximum queue length **specified in the input**.

Customers will look for the **first available queue** (and **not the shortest!**)

As an example, given an input of one server with a maximum queue length of 1 (i.e. only one waiting customer is allowed), followed by three customer arrivals and *assuming that service times is* `1.0`,

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

Server Queues and Wait

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
```

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 Customers who have been served
2. the number of Customers served
3. the number of Customers 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]
```

The **run**() method now returns a Pair<String, String>

The first String would be your simulation output

The second String would be the simulation statistics

On-Demand Service Timings

Notice that not all Customers who arrive get served. To make our simulation more realistic, we want to provide service time data **only if** the Customer is being served.

To facilitate on-demand data (or delayed data), we make use of the Supplier interface which species an abstract method get() to be defined by its implementation class

On-Demand Service Timings

```java
class DefaultServiceTime implements Supplier<Double> {
    // this is only an example that returns a default (standard)
    // service time of 1.0
    // note that service time can be some random value!
    public Double get() {
        return 1.0;
    }
}
```

This is a simple implementation for demo purposes, your index.pdf has a separate one that you can read through to get a better understanding

On-Demand Service Timings

As an example:

```
DefaultServiceTime svcTimeSupplier = new DefaultServiceTime();
Double svcTime = svcTimeSupplier.get();
```

We will be using this for our project, where it will be passed into your Shop class. You may decide where you ultimately store the Supplier, but the Customer should have nothing to do with handling service times.

On-Demand Service Timings

```java
class Shop {
    private final Supplier<Double> serviceTime;
     // constructor
    Shop(..., Supplier<Double> serviceTime) {
        ...
    }
    ...
    public Double getServiceTime() {
        return this.serviceTime.get();
    }
    ...
}
```

getServiceTime(), or any call to get() from the supplier should only be invoked (called) when a Customer is served! **(IMPORTANT)**

Optionals

- We are removing all bans on Optional methods only in the Simulator class (i.e. you can use any and all methods)

- However, you may not expose the Optional in any other class.

Optionals

- That being said, you may no longer use orElse() and orElseGet() in any classes outside of the Simulator

- Instead, you may use the or() method that takes in a Supplier of an Optional

Design Tips

- Make sure you only invoke `getServiceTime()` in one location. Invoking the method call in multiple locations may result in a wrong service time (extremely important for grading!)
- Avoid having multiple `Events` with the same `Customer` in the `PQ`. The `PQ` should only have one `Event` per `Customer`.
- Consider how you would emulate a queue for each `Server`
  - Hint: it is not necessary to use an `List` or a `PQ`

Design Tips

Something to think about:

How are you going to determine when the Wait becomes a Serve?

What happens when there are multiple Customers waiting?

Deadline: **6 Apr 2359**