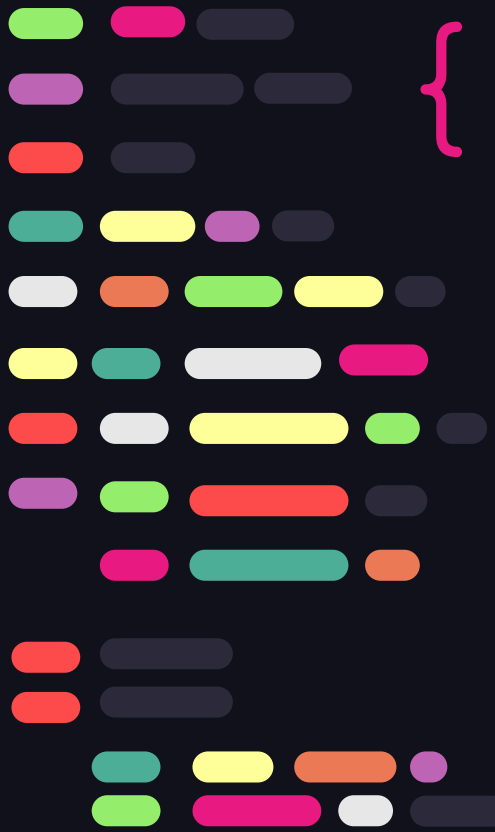


CS2030

Lab 4

AY25/26 Sem 1, Week 8

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

Lab Admin

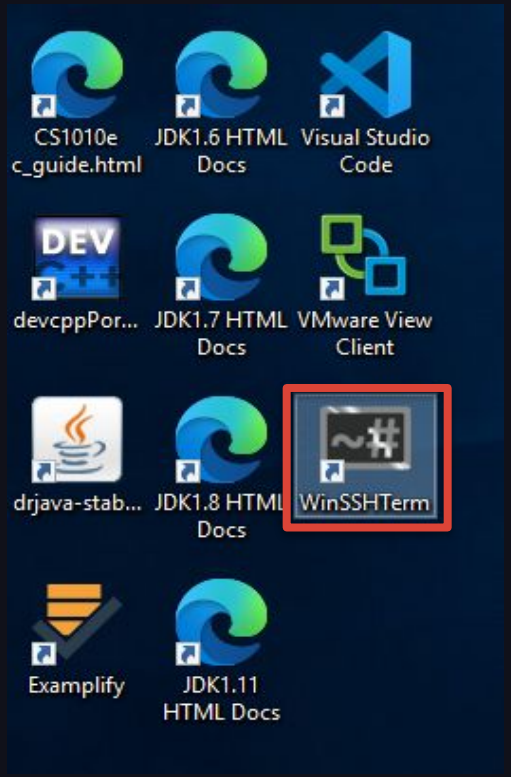
Admin Stuff

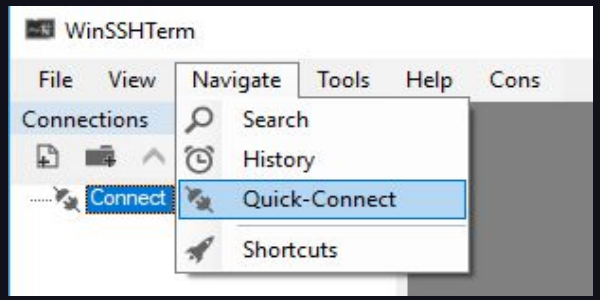
Log in to the lab device

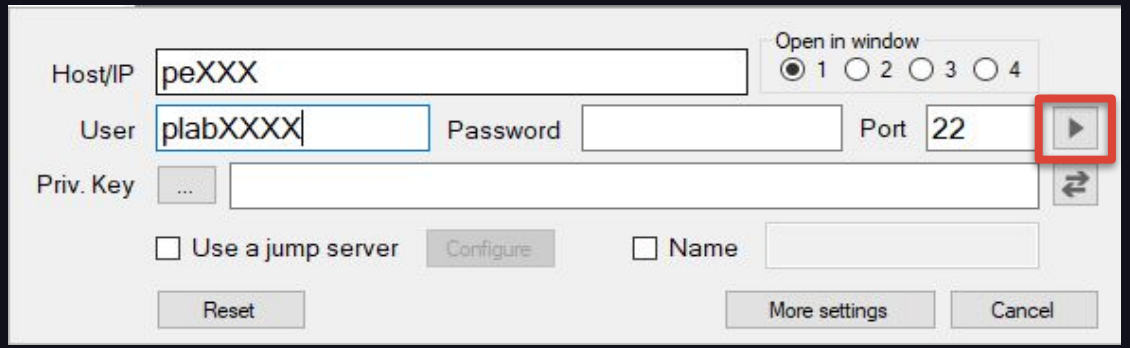
Username: nusstu\e{0/1}xxxxxxxx
Password: <your nusstu password>

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

Connecting to PE Node







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

Project Design Considerations

Access Modifiers

If a method/constructor is only used within a class and not by any test cases,

- **Make the method/constructor private**

Event Priority

Event priority is determined as follows

- Earlier events have higher priority
- If multiple events occur at the same time,
    - the event whose customer arrives earlier has highest priority

Event Priority

- **Event** implements **Comparable<Event>** to enable comparisons.
- To enable comparison of **Customer**s by arrival time, **Customer** can implement **Comparable<Customer>** to streamline code.

Implicit Typechecking

Strong focus on Polymorphism

- override 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 **Event**s should be the same, but each **Event** should behave differently

Implicit Typechecking

You should not be using an additional attribute to determine the exact subtype of an **Event** within any part of your program.

Methods that return a **String** or **Integer** (or any other proxy) to help the **State** determine the exact **Event** are also undesirable.

Inheritance

Instead of defining the same common attributes for all your **Event**s (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.

Supplier<T>

a crash course

Supplier<T>

- Simple functional interfaces you've seen before
  - **Function<T, R>**
    - **R** apply(**T** t) // input (**T**) -> output (**R**)
  - **Predicate<T>**
    - **boolean** test(**T** t) // input (**T**) -> **boolean**

Supplier<T>

- Functional interfaces have **Single Abstract Methods**
- This enables implementation via lambdas
  - e.g. `x -> x + 1` specifies all the logic needed for a **Function<Integer, Integer>**
  - `x -> x % 2 == 0` specifies all the logic needed for a **Predicate<Integer>**

Supplier<T>

- (Yet another) Functional interface from java.util.function

- **Single Abstract Method**
  - `T get() // no input -> output (of type T)`
- Lambda syntax
  - `() -> {expression}`

Supplier<T>

- Examples of **Supplier**s in lambda form
  - **Supplier<Double>** doubleSupp = () -> 1.0;
  - **Supplier<Integer>** intSupp = () -> 400;
  - **Supplier<String>** greetSupp = () -> “Hello World!”;

Supplier<T>

- Naturally, Supplier can also be implemented by concrete classes

```
class DefaultServiceTime implements Supplier<Double> {
    public Double get() {
        return 1.0;
    }
}
```

- **Logically identical to** `() -> 1.0`

Supplier<T>

- Why **Supplier**?
- **Supplier**s contain instructions for a computation
- However, they **do NOT execute** those instructions **until** the `get()` method is called
- **Lazy evaluation!**

Supplier<T>

- **Lazy chaining** of computations
  - **Supplier<Integer>** suppOne = () -> 1;
  - **Supplier<Integer>** suppPlusOne = () -> suppOne.get() + 1;

Project

Task Overview

Introduction of:

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

On-Demand Service Timings

In the previous project milestone, we assigned a serviceTime to each customer, even if they are not served.

- `new Customer(1, 1.0, 1.0)`

In this example, customer 3 leaves

- Customer 3's serviceTime has gone unused

```
0.5 customer 1 arrives
0.5 customer 1 serve by server 1
0.6 customer 2 arrives
0.6 customer 2 serve by server 2
0.7 customer 3 arrives
0.7 customer 3 leaves
1.5 customer 2 done
1.5 customer 4 arrives
1.5 customer 4 serve by server 2
1.6 customer 1 done
1.6 customer 4 done
1.6 customer 5 arrives
1.6 customer 5 serve by server 1
1.7 customer 6 arrives
1.7 customer 6 serve by server 2
1.8 customer 5 done
2.0 customer 6 done
```

On-Demand Service Timings

- Not all **Customer**s who arrive get served.
- To make our simulation more realistic, we provide service time data **only when** the **Customer** is served.
- We will thus make use of **Supplier**

On-Demand Service Timings

```
class DefaultServiceTime implements Supplier<Double> {
    // this returns a default service time of 1.0
    // Your implementation will be tested against more complex
    // Suppliers.
    public Double get() {
        return 1.0;
    }
}
```

This is a simple implementation for demo purposes.

On-Demand Service Timings

As an example:

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

- This supplier will be passed into your **Shop** class.
- **Customer** should **NOT** handle service times.

On-Demand Service Timings

```
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)**

On-Demand Service Timings

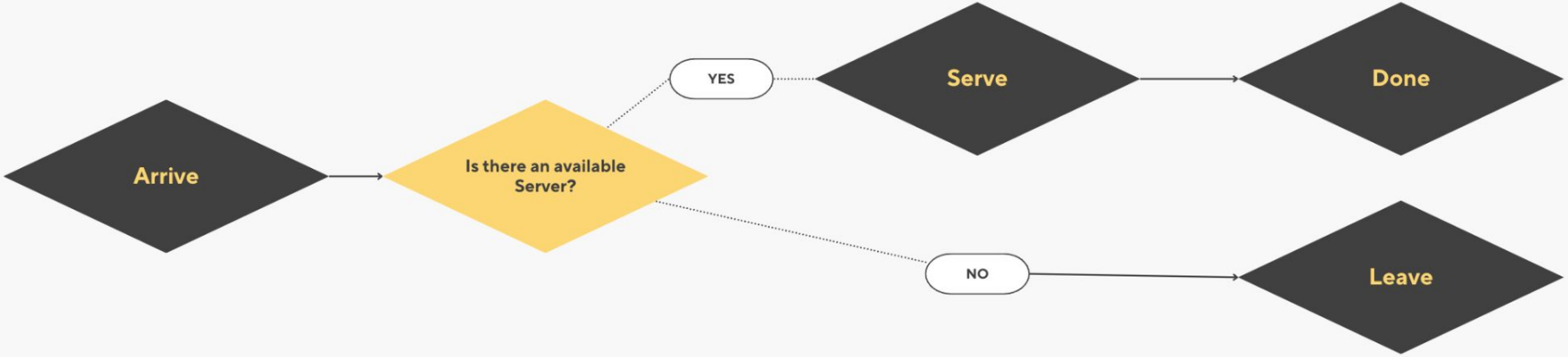
```
class DefaultServiceTime implements Supplier<Double> {

    public Double get() {
        System.out.println(“generating service time...”);
        return 1.0;
    }
}
```

This allows for easier tracing of get() calls in your code.

On-Demand Service Timings

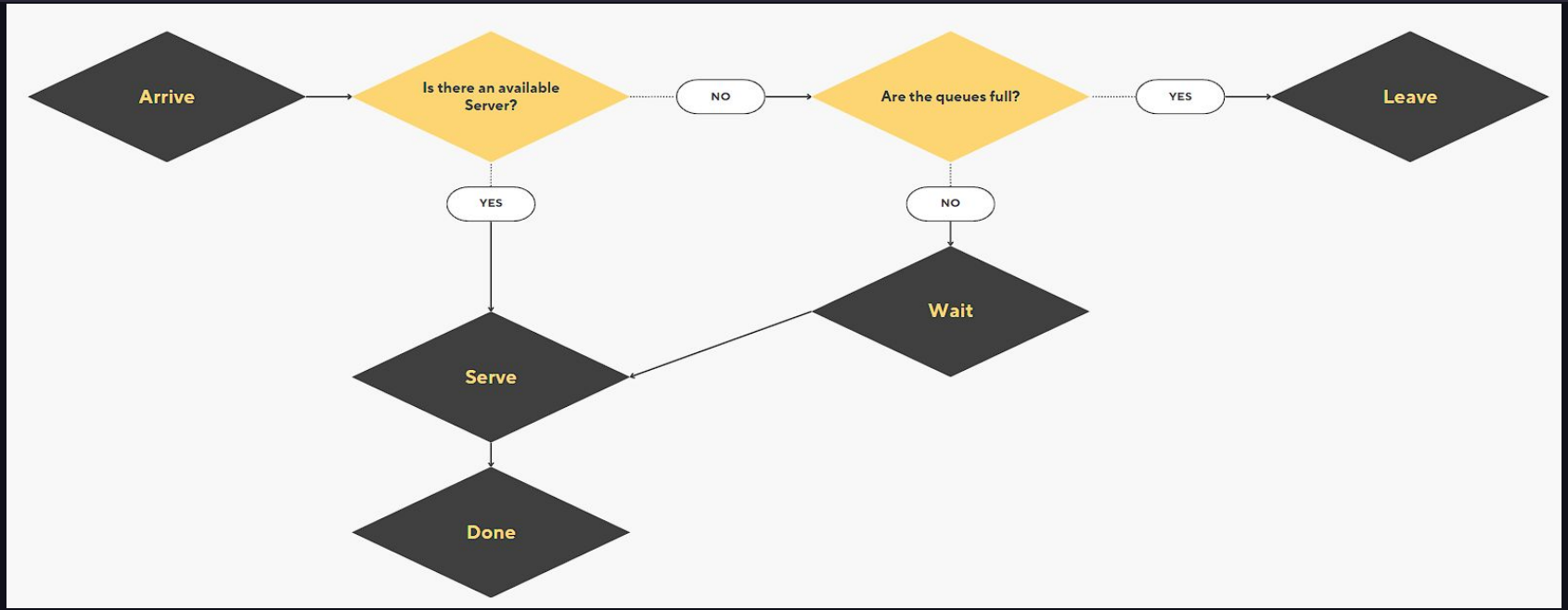
Deadline: **16 Oct (Thurs) 2359**

Visualisation: Lab 3



Remember this?

Visualisation: Project



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

Server Queues and Wait

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

Servers now have a maximum queue length.

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

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
```

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

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 Pair<String, String>

The first **String** contains simulation output

The second **String** contains simulation statistics

Updated Main

A new **Main** class with **DefaultServiceTime** has been provided.

Adjust your implementation as required.

Note that your program will be tested against test cases where service times could be different when serving different **Customer**s.

Design Tips

- Make sure you only invoke `get()` once per customer served, only when the **Customer** is served. Invoking the method call in multiple locations may result in a wrong service time (extremely important for grading!)
- Avoid having multiple **Event**s with the same **Customer** in the **PQ**.
- At any given time, the **PQ** should only have one **Event** per **Customer**.

Design Tips

- Consider how you would emulate a queue for each **Server.** Think about real-life queue systems and how they can be represented.
  Hint: it is not necessary to use an **List** or a **PQ**. Your events already have an order
- You may implement additional **Event** classes if required

Design Tips

Something to think about:

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

What happens when there are multiple **Customer**s waiting?

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