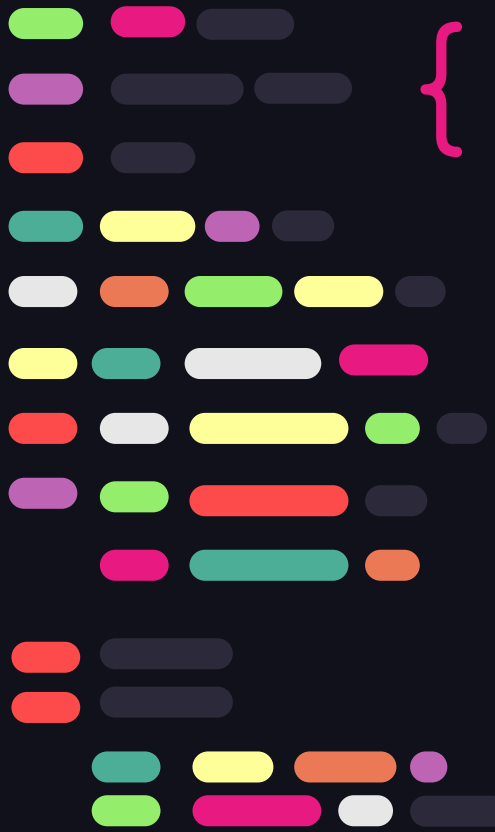


CS2030

Lab 3

AY24/25 Sem 2, Week 5

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

Lab Admin

Timeline

WEEK	**LAB**	**DUE**
5 (this week)	3 (Project Part 2)	2 (Project Part 1 Level 3/5)
6 (next week)	Mock PA#1 (21 Feb)	2 (Project Part 1 Level 5/5)
Recess week		Self Practice Exercises (tentative) 3 (Project Part 2) Mock PA#1
7 (after recess week)	PA#1	

Mock PA #1

If you plan on continuing the course, it is highly recommended that you attend next week's Mock PA

The session will serve to help you get used to the Practical Assessment on Week 7 once you come back from recess week

Plagiarism

Plagiarism is a VERY serious academic offense.

NUS Plagiarism Policy states that the minimum penalty for cases of plagiarism and cheating in tests/examinations/graded assignments that have been assessed to be of ‘Moderate’ severity would be that of a `Fail` grade for the affected course.

https://myportal.nus.edu.sg/studentportal/student-discipline/all/docs/NUS-Plagiarism-Policy.pdf

Plagiarism

Your lab submissions should always be done independently.

Do **NOT** share your code with others - Discussions are fine, but you will never know if they blatantly copy-paste your code and you become complicit to the plagiarism offence.

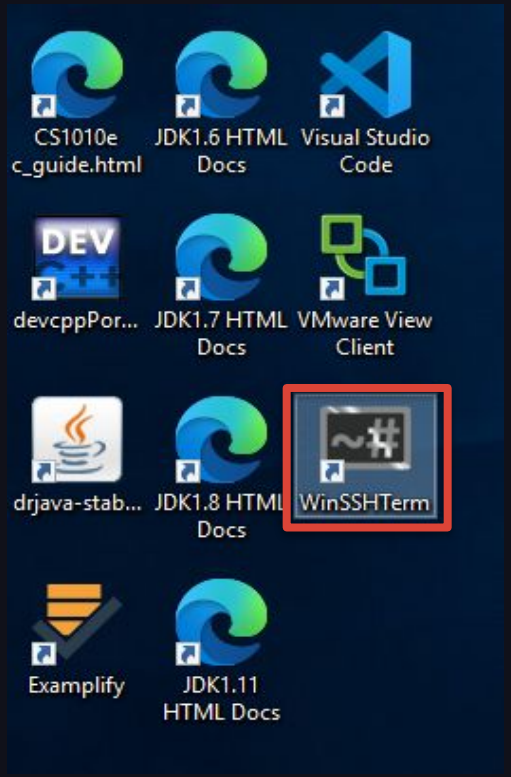
Admin Stuff

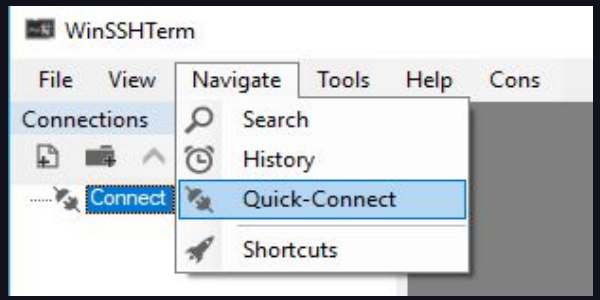
Log in to the lab device

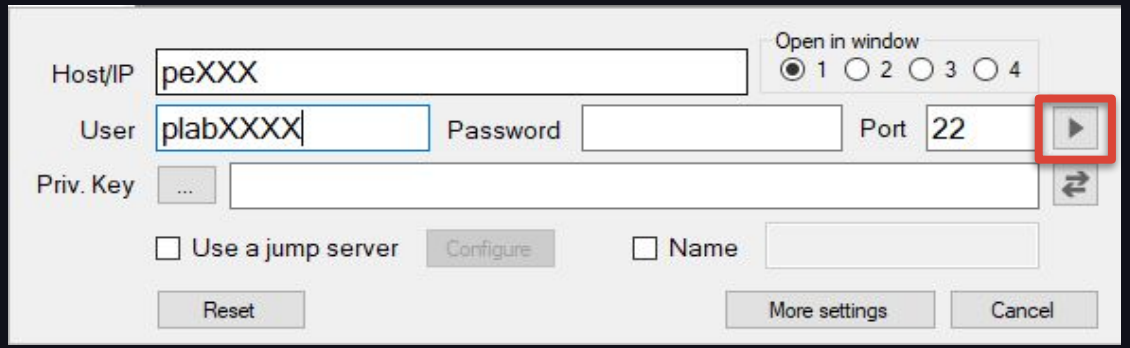
Username: nusstu\e{0/1}xxxxxxxx
Password: <your nusstu password>

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

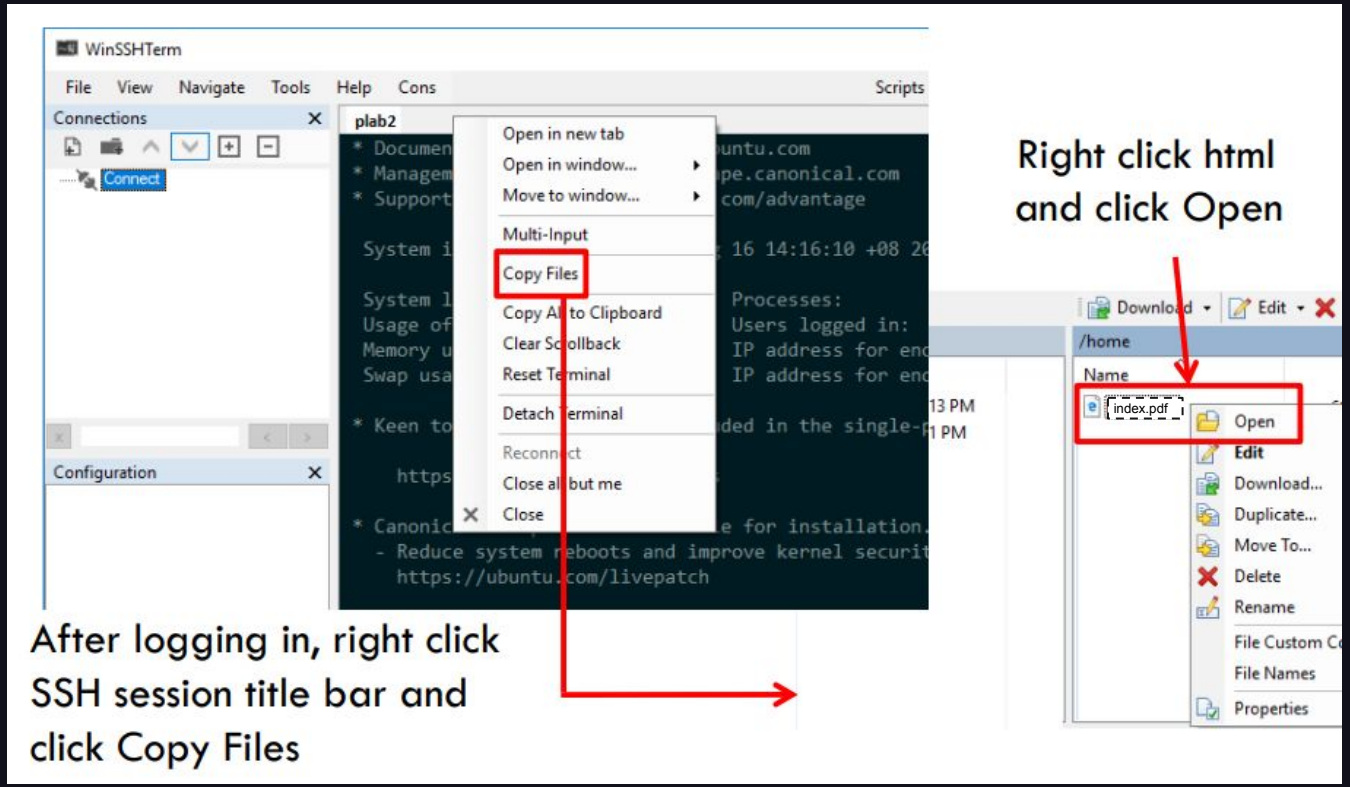
Connecting to PE Node







Viewing Questions



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

Session Overview

- Implementation Considerations
- Recap (Overloaded constructors, Polymorphism)
- Lab Task Overview

Implementation Considerations

Working with Lists

- We often stream lists to transform our data before converting it back

- use .toList() instead of .collect(Collectors.toList) to do this

- The former returns an immutable implementation of a List, while the Collectors method returns a mutable ArrayList

Tell-Don’t-Ask

Unnecessary getters are good indication of violating the principle. Common violation:

- Having `Shop` call getters in `Server` and `Customer` to check if the server is free to serve the customer. Tell the `Server` to check if `Customer` can be served instead

Some getters are necessary, e.g. `get()` in `List`, or complex objects like `Customer` (`getServiceTime()` etc) will need getters to be functional. Avoid them where you can.

Variable Naming

Use more descriptive variable names

```
public class Customer {
    private final double at;              // not advisable
    private final double serviceTime;     // better var name
}
```

Spell out your variables fully with camelCase (e.g. arrivalTime)

Variable Names

```java
public class Customer {
    private final double arrivalTime;

    public Customer(double at) {
        arrivalTime = at;
        }
    }
```

```java
public class Customer {
    private final double arrivalTime;

    public Customer(double arrivalTime) {
        this.arrivalTime = arrivalTime;
        }
    }
```

Notice how both the constructor input variable and the attribute variable are named arrivalTime?

You can do this by differentiating with the keyword this, instead of changing the input variable name to `at`.

Line Wrapping

Wrap your lines instead of letting them get too long

Usually appropriate to wrap lines after operators or at appropriate junctures for Strings (e.g. after a full stop)

```
e.g. Server tempServer = new Server(serverId,
                                    serverAvailableTiming,
                                    customer,
                                    customerArrivalTiming)
```

vs `Server tempServer = new Server(serverId, serverAvailableTiming, customer, customerArrivalTiming)`

Abstraction

- Break up long method implementations into multiple smaller methods. Do not dump everything into one method.
- If needed, you can **abstract** out long lines of code into **private** helper functions
- This will help you when it comes to debugging and is important in writing code with good design

Data Hiding

- Always declare class and instance variables as **private**
- This is to **hide implementation details** from clients
- This is part of **encapsulation** whereby only classes and instances that need to know about the data will have access to them

Immutability

- Make objects immutable so that they cannot be tampered with
- Internal state of object stays constant throughout

Immutability

- Use the **final** keyword for instance variables and use **constructors** to get new object instances
  - e.g. `Customer setX(double newX)` doesn't modify *X* but instead creates a new customer with the new *X*
- No modifying instance variables!
  - e.g. a `void setX(double newX)` method that directly modifies `this.X`
  - Do not change the state of immutable objects once they are created/initialised

Recap

Overloaded Constructors

- We use overloaded constructors to “update” object instances by returning a new instance with updated parameters

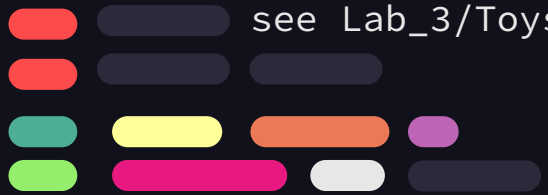
- Constructor overloading is having two (or more) constructors in a class that differ in the number or type of their parameters

Overloaded Constructors

Assume we have a Pet class with the following attributes

```
class Pet {
    private final String name;
    private final String gender;
    private final int age;
```

see Lab_3/Toys/OverloadedConstructors/implementation1 for details



Overloaded Constructors

A “default” constructor can look something like this

```
Pet(String name) {
    this.name = name;
    this.gender = "Unknown";
    this.age = 0;
}
```

Overloaded Constructors

We can have additional constructors that help us pass in a gender or an age

This means we can pass in the same Pet name and a different age or gender along with the same name to “update” those parameters

```
Pet(String name, String gender) {
    this.name = name;
    this.gender = gender
    this.age = 0;
}

Pet(String name, int age) {
    this.name = name;
    this.age = age;
    this.gender = "Unknown";
}
```

see Lab_3/Toys/OverloadedConstructors/implementation2 for details

Overloaded Constructors

If you already have an instance of a Pet defined, you can use constructors like these to update a given attribute or property

```
Pet(Pet p, int age) {
    this.name = p.name;
    this.gender = p.gender;
    this.age = age;
}

Pet(Pet p, String gender) {
    this.name = p.name;
    this.gender = gender;
    this.age = p.age;
}
```

see Lab_3/Toys/OverloadedConstructors/implementation3 for details

Which means your class can look like this with just constructors

(But you often won't need this many, please only code in what you need)

```java
public class Pet {
    private final String name;
    private final String gender;
    private final int age;

    Pet(String name, String gender, int age) {
        this.name = name;
        this.gender = gender;
        this.age = age;
    }

    Pet(String name, String gender) {
        this.name = name;
        this.gender = gender;
        this.age = 0;
    }

    Pet(String name, int age) {
        this.name = name;
        this.gender = "Unknown";
        this.age = age;
    }

    Pet(Pet p, int age) {
        this.name = p.name;
        this.gender = p.gender;
        this.age = age;
    }

    Pet(Pet p, String gender) {
        this.name = p.name;
        this.gender = gender;
        this.age = p.age;
    }

    @Override
    public String toString() {
        return String.format("Pet named %s, with gender %s and age %d", name, gender, age);
    }
}
```

In a more compact form:

```java
public class Pet {
    private final String name;
    private final String gender;
    private final int age;

    Pet(String name, String gender, int age) {
        this.name = name;
        this.gender = gender;
        this.age = age;
    }

    Pet(String name, String gender) {
        this(name, gender, 0);
    }

    Pet(String name, int age) {
        this(name, "Unknown", age);
    }

    Pet(Pet p, int age) {
        this(p.name, p.gender, age);
    }

    Pet(Pet p, String gender) {
        this(p.name, gender, p.age);
    }

    @Override
    public String toString() {
        return String.format("Pet named %s, with gender %s and age %d", name, gender, age);
    }
}
```

see Lab_3/Toys/OverloadedConstructors/implementation4 for details

Polymorphism

Greek-derived word that means “having multiple forms”, happens when classes are related by inheritance

Consider this code that tries to model Payment methods:

```
class Payment { }

class Card extends Payment { }

class Cash extends Payment { }

class BankTransfer extends Payment { }
```

```
void processPayment(List<Payment> payments) {
    for (Payment payment : payments) {
        if (payment instanceof Cash) {
            System.out.println("Dig wallet for cash...");
        } else if (payment instanceof Card) {
            System.out.println("swipe some cards...");
        } else if (payment instanceof BankTransfer) {
            System.out.println("Getting account number...");
        }
    }
}

// To run the code
processPayment(List.of(new Card(), new Cash(), new BankTransfer()));
```

Polymorphism

see Lab_3/Toys/Polymorphism/badExample/ for details

```
class Payment { }

class Card extends Payment { }

class Cash extends Payment { }

class BankTransfer extends Payment { }
```

```
1  class Main {
2
3      public static void main(String[] args) {
4          processPayment(List.of(new Card(), new Cash(), new BankTransfer()));
5      }
6
7      public void processPayment(List<Payment> paymentList) {
8          for (Payment payment : paymentList) {
9              if (Payment instanceof Cash) {
10                 System.out.println("Dig wallet for cash");
11             } else if (Payment instanceof Card) {
12                 System.out.println("swipe card");
13             } else if (Payment instanceof BankTransfer) {
14                 System.out.println("Getting bank account number,");
15             }
16         }
17     }
18 }
```

What if we want to add another payment method (e.g. QR)? We will have to create a new class and modify the `processPayment()` method, which is not ideal

Polymorphism

Consider this implementation instead:

```
abstract class Payment {
    abstract void pay();
}

class Card extends Payment {
    @Override
    void pay() {
        System.out.println("swipe some cards...");
    }
}

class Cash extends Payment {
    @Override
    void pay() {
        System.out.println("Dig wallet for cash...");
    }
}

class BankTransfer extends Payment {
    @Override
    void pay() {
        System.out.println("Getting account number...");
    }
}
```

```
void processPayment(List<Payment> payments) {
    for (Payment payment : payments) {
        payment.pay();
    }
}

// To run the code:
processPayment(List.of(new Card(), new Cash(), new BankTransfer()));
```

See how there's only a call to the pay() method? Let each subclass @Override for their implementation!

Polymorphism

This allows you to simply create a new class, and all you have to do is @Override the pay() method!

```
class PayLah extends Payment {
    @Override
    void pay() {
        System.out.println("Loading camera...")
    }
}
```

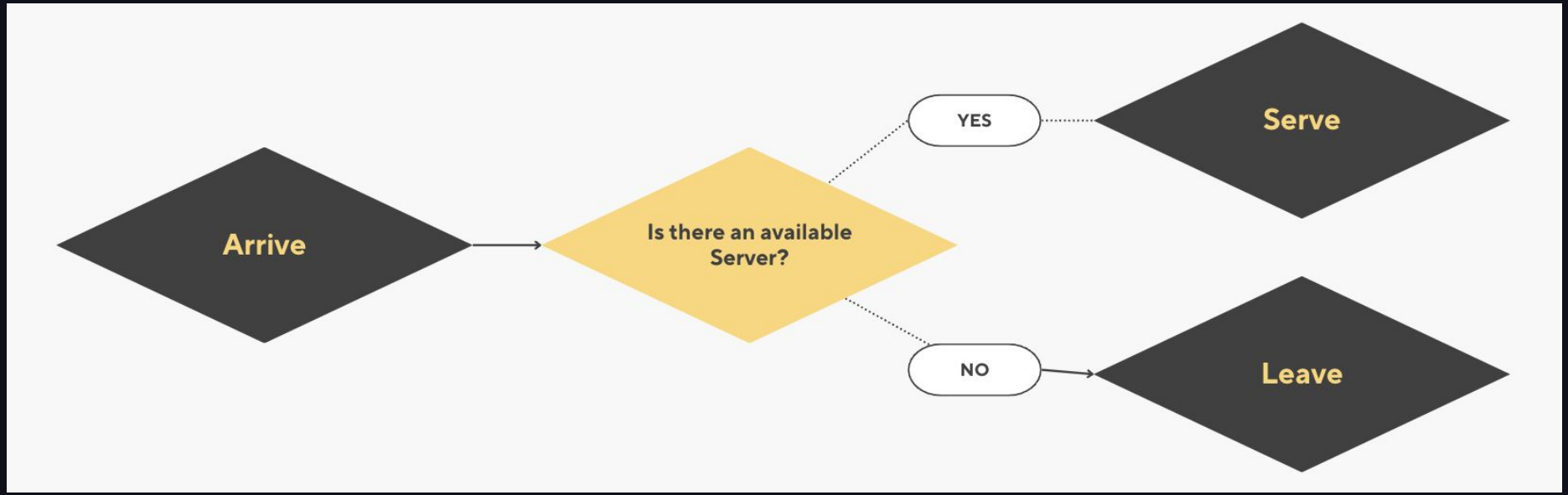
see Lab_3/Toys/Polymorphism/betterExample/ for details

```
abstract class Payment {
    abstract void pay();
}

class Card extends Payment {
    @Override
    void pay() {
        System.out.println("swipe some cards...");
    }
}

class Cash extends Payment {
    @Override
    void pay() {
        System.out.println("Dig wallet for cash...");
    }
}

class BankTransfer extends Payment {
    @Override
    void pay() {
        System.out.println("Getting account number...");
    }
}
```

```
void processPayment(List<Payment> payments) {
    for (Payment payment : payments) {
        payment.pay();
    }
}

// To run the code:
processPayment(List.of(new Card(), new Cash(), new BankTransfer()));
```

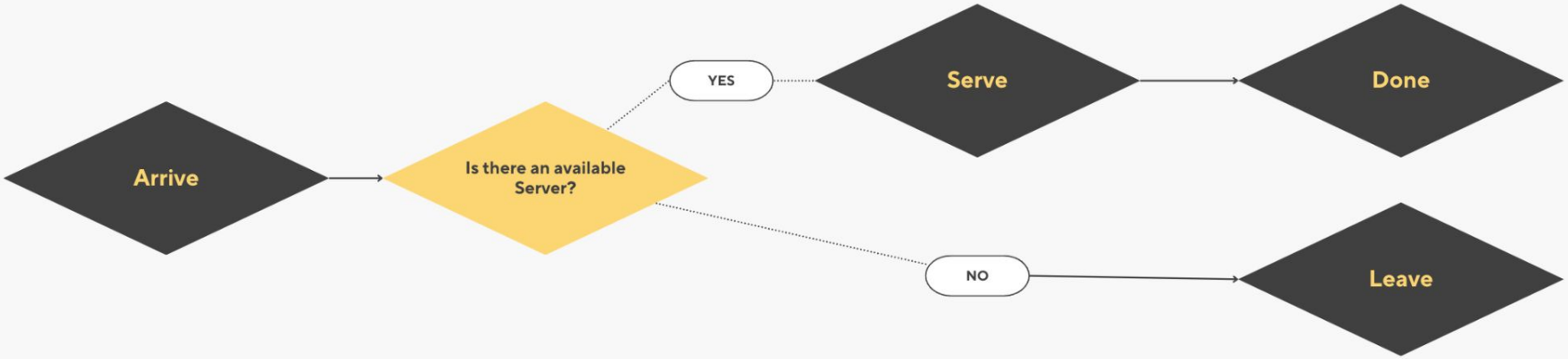
Lab 3

Visualisation: Lab 2



Remember this?

Visualisation: Lab 3



See them as Events instead!

(with the addition of Done after Serve)

Done happens after the Server is *done* serving the Customer

Task Overview

Introduction of:

- **Event Classes**
- **Done** Event
- **Priority Queue**

Event Classes

- We are now modelling the events in our project as individual classes

- e.g. **Customer** arrives -> **ArriveEvent**
  Customer gets served -> **ServeEvent**
  Customer leaves -> **LeaveEvent**

Event Classes: Done

- We are also introducing the **Done** event, where a server is done serving a customer
- which means you can expect:
  ServeEvent -> (after some time) -> **DoneEvent**

Event Classes: next()

- Like how we used **next()** in **State** to process the **Customer** interactions, you will need to implement a **next()** to process **Events**

- the **next()** method should return the new **Event** as well as an updated **Shop** in a **Pair<Event, Shop>**

Event Classes: next()

- e.g. calling **next()** on an **ArriveEvent** would either lead to a **ServeEvent** or **LeaveEvent**, and **next()** on **Serve** -> **Done**
- **next().t()** is the **Event**, **next().u()** is the updated **Shop**

```
jshell> Event e1 = new ArriveEvent(new Customer(1, 1.0, 1.0), 1.0)
e1 ==> 1.0 customer 1 arrives

jshell> Event e2 = e1.next(new Shop(2)).t()
e2 ==> 1.0 customer 1 serve by server 1

jshell> Shop s2 = e1.next(new Shop(2)).u()
s2 ==> [server 1, server 2]

jshell> Event e3 = e2.next(s2).t()
e3 ==> 2.0 customer 1 done
```

Priority Queues

5	Other integers...

Imagine that we have a **Priority Queue** of integers and define the **highest priority** to be that of the biggest integer.

We insert the integers 1, 2, 3, 4 and 5 into the above **PQ**

Regardless of the order of insertion, 5 will be the first element in the **PQ**

Priority Queues

4	Other integers...

Note: Elements in the **PQ** are **NOT** necessarily sorted in order. A PQ only guarantees that the first element is the **highest priority** one.

```
for (Element e : pq) {
    // The elements may not necessarily be sorted in order!
}
```

So how do we define **highest priority**?

Comparable Interface

Let objects be **Comparable** to each other so that an order can established

Implement a compareTo method within the class to specify how comparison should work for relative ordering

From the Java API:

```
int compareTo(T obj)
```

Compares this instance of object to `obj`for order. Returns a
- negative integer (less than)
- Zero (equal to)
- positive integer (greater than) than obj.

Comparable

```
class Person {
    int age;
    String name;

    // other code here

}
```

Imagine that we have a **Person** class

Comparable

```
class Person implements Comparable<Person> {
    // sort by age in ascending order
    public int compareTo(Person otherPerson) {
        return this.age - otherPerson.age;
    }
}
```

You can define your own compareTo function, which is especially useful if you're working with custom classes

Comparable

```
class Person implements Comparable<Person> {
    // sort by age in ascending order
    public int compareTo(Person otherPerson) {
        return Integer.compareTo(this.age, otherPerson.age);
    }
}
```

Here, we are using Java's inbuilt compareTo function to compare the Integers

Note that since our attributes are of the primitive int, we have to call on the wrapper Integer class' compareTo

Immutable Priority Queues

```
// Since we have defined a compareTo method for Person,
// the PQ knows how to compare between Person 1 and Person 2
PQ<Person> pq = new PQ<Person>();
```

Difference between Java PriorityQueue vs immutable PQ:

- Java PriorityQueue allows you to mutate the list without changes.
- Immutable PQ returns a Pair object when polled, first attribute storing the item removed from PQ, second item storing the modified PQ.

Priority Queue - Pairs

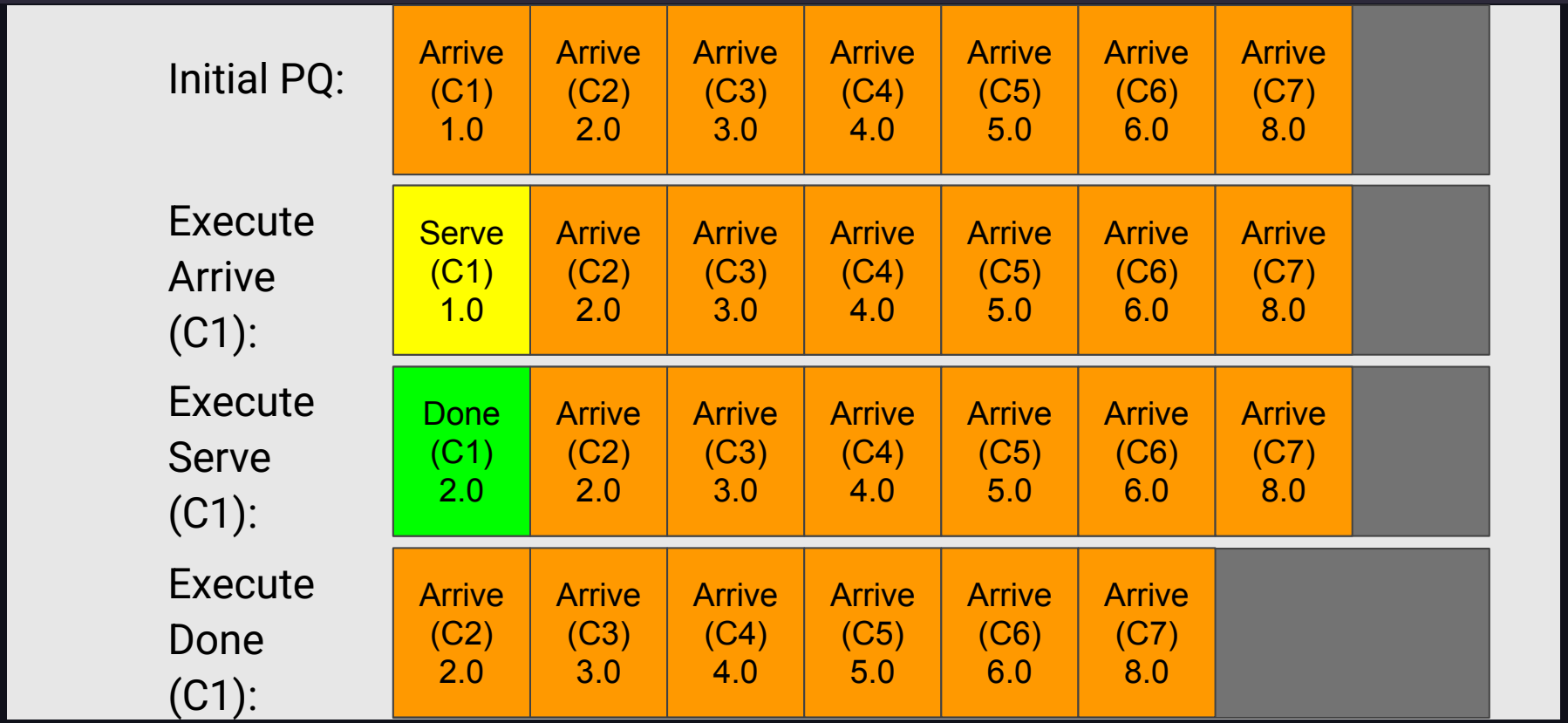
```
// Get the first element (removed from the PQ)
Pair<Optional<Person>, PQ<Person>> pair = pq.poll();
Optional<Person> person = pair.t();
pq = pair.u();

// Add a new element (pq.add is an alternative)
PQ<Person> pq = pq.add(new Person());

// Check whether the PQ is empty
boolean isEmpty = pq.isEmpty();
```

Event Priority Queue

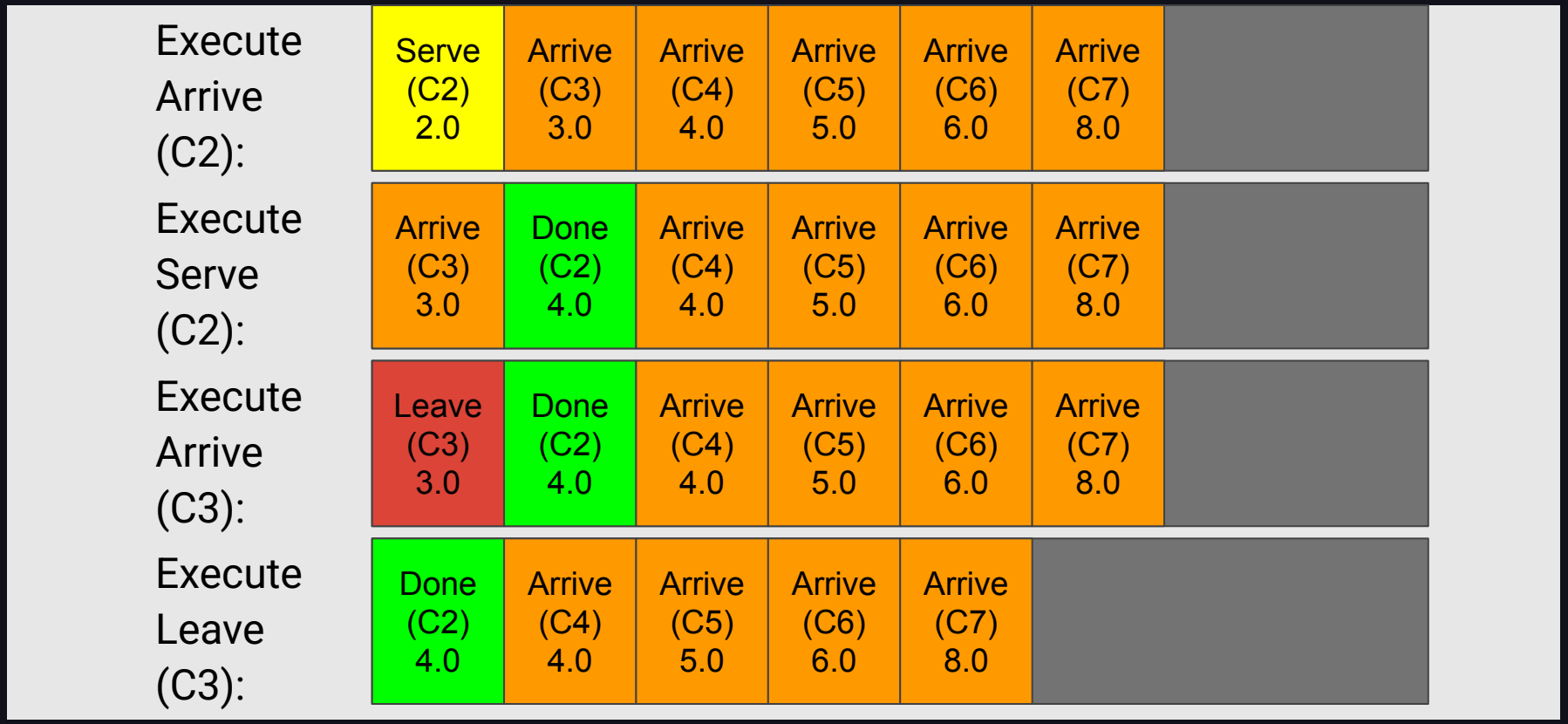
- We will be using a Priority Queue in order to manage and process Events in a right way

- This means you will now have a PQ of Events, which will be in the State

PQ Illustration



PQ Illustration



Event Priority Queue

- note when calling **next()** on the **State** now returns a State that has a singular output (event), instead of returning the entirety of a **Customer**'s **Event**s in the previous lab

```
jshell> new State(pq, new Shop(2)).next()
$.. ==> 1.0 customer 1 arrives

jshell> new State(pq, new Shop(2)).next().next()
$.. ==> 1.0 customer 1 serve by server 1

jshell> new State(pq, new Shop(2)).next().next().next()
$.. ==> 2.0 customer 1 done
```

Main and Simulator

- Given the changes to the project thus far, a new Main file has been given to you

- Write the Simulator class with your project design in mind so as to carry out the simulation

Main and Simulator

We will only be testing for Level 4
Use Levels 1 to 3 to guide your implementation to it

```
$ cat 1.in
3 3
1 0.5 1.0
2 0.6 1.0
3 0.7 1.0

$ cat 1.in | java --enable-preview Main
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 serve by server 3
1.5 customer 1 done
1.6 customer 2 done
1.7 customer 3 done
```

```
$ cat 3.in
2 3
1 0.5 1.0
2 0.6 1.0
3 0.7 1.0

$ cat 3.in | java --enable-preview Main
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 1 done
1.6 customer 2 done
```

Task Overview

- Event Classes
  - ArriveEvent, ServeEvent, DoneEvent, LeaveEvent
- State class
- Simulator class
- Minor changes to other classes

Deadline

2 Mar (Sunday) 2359

(end of recess week)

Deadlines

Lab 2 (Level 4/5)	: 20 Feb (Thursday)
Lab 3	: 02 Mar (Sunday)
Exercise 1	: 02 Mar (Sunday)
Exercise 2	: 02 Mar (Sunday)


MAY THE FORCE BE
WITH YOU