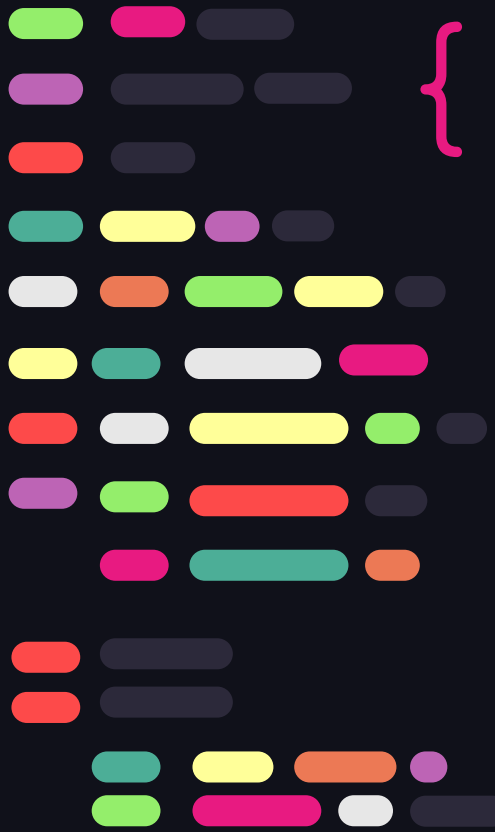


CS2030
Mock PA #1

AY25/26 Sem 1, Week 6

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

Lab Admin

Timeline

WEEK	**LAB**	**DUE**
6 (this week)	Mock PA#1 (19 Sep)	3 (Project Part 2)
Recess week		
7 (after recess week)	PA#1	Mock PA#1 Self Practice Exercises
8	4 (Project Part 3)	3 (Project Part 2 reopened) (Thurs) PA#1 moderation (Sun)

Project

Lab 3 will be reopened for submissions until the night before Lab 4. The tutors will give feedback on your current Lab 3 submissions, and you should refine your implementations accordingly.

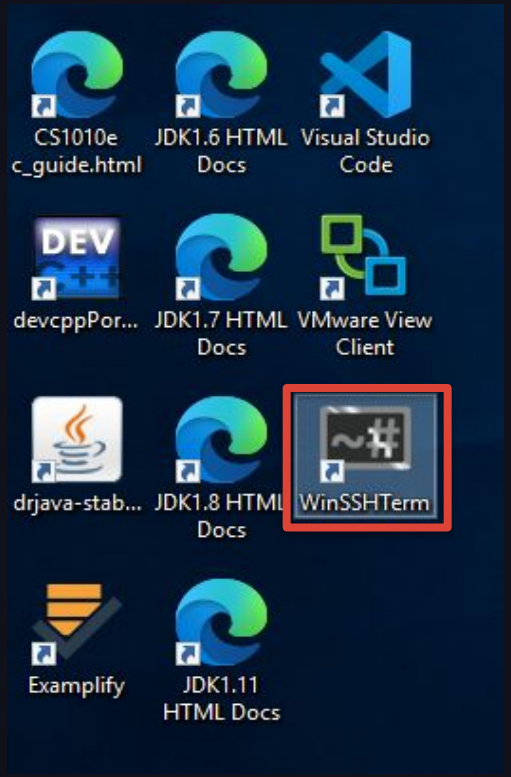
Admin Stuff

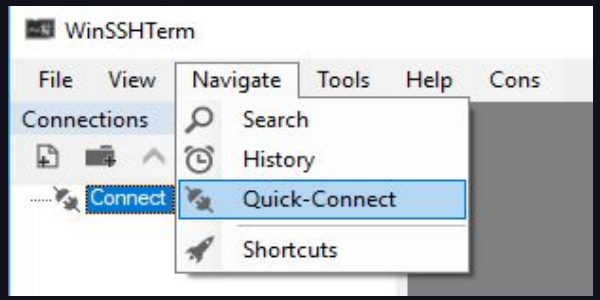
Log in to the lab device

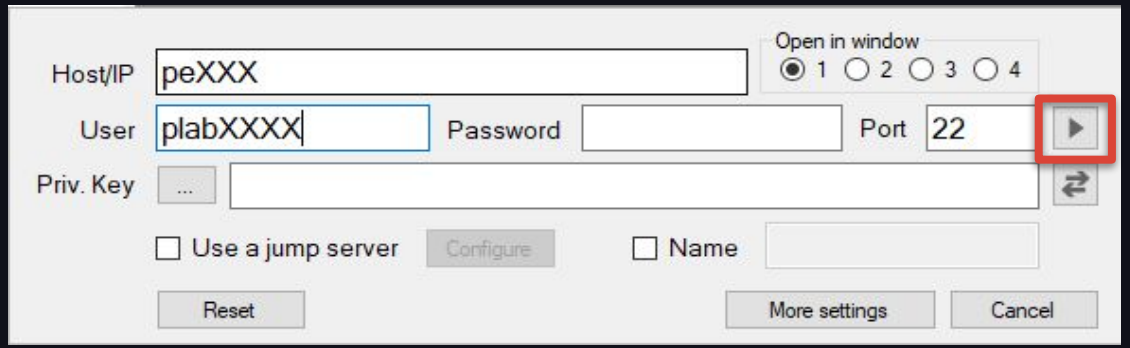
Username: nusstu\e{0/1}xxxxxxxx
Password: <your nusstu password>

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

Connecting to PE Node







Course Admin

Mid Sem Tutor Feedback

A survey has been released on Canvas for you to provide teaching feedback about the module so far

Responses are **anonymous**

Please fill it in so that we can continue to improve on the teaching :)

Recap

orElse()

- The orElse() method should be used to help you deal with the situation where your Maybe is empty, and not to prematurely strip the value out of the Maybe.
- You should think of this as the “else” part of an if…else statement

orElse()

```
maybe.map(x -> logic handling assuming value exists)
          .orElse(logic handling if value doesn't exist)
```

Basic example, but this should be the structure to approach your Maybe with.

orElse()

```
findServer().map(server -> server).orElse(new Server(-1));
```

This is bad practice, and beats the point of using Maybe.

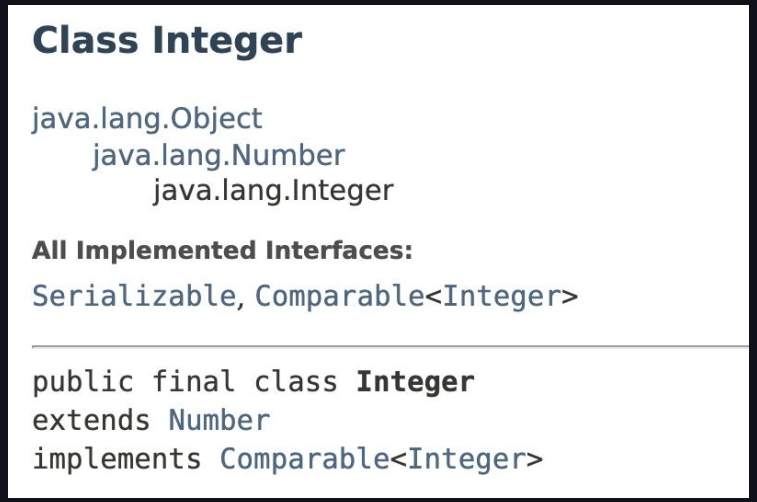
It is advisable that you start thinking of how to properly use Maybe.

Comparing Objects

- For Java to order your objects during sorting, there must be some form of priority defined
- There are 3 general ways to do this:
  - inbuilt .**sort**() where natural ordering is present
  - **Comparable** interface
  - use of a **Comparator**

Comparable

Some Java classes already implement **Comparable** (check Java API) Classes that you expect to have some kind of natural ordering probably implement **Comparable** (e.g. **String**, **Integer**, **Double** etc), which is why you can simply call .sort() on these



Class Integer

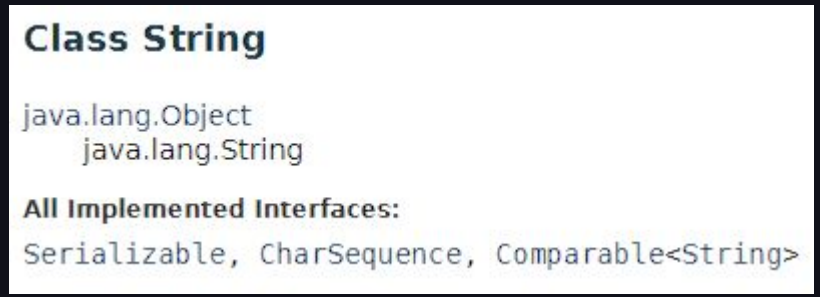
java.lang.Object
    java.lang.Number
        java.lang.Integer

All Implemented Interfaces:

Serializable, Comparable<Integer>

---

public final class **Integer**
extends Number
implements Comparable<Integer>



Class String

java.lang.Object
    java.lang.String

All Implemented Interfaces:

Serializable, CharSequence, Comparable<String>

Comparable

- Last lab, we learnt how to define priority between objects through the **Comparable** interface (to compare between **Event**s through **Event** times)

Comparable

- By implementing the **Comparable** interface in our **Event**s, Java now knows how to order your **Event**s
- e.g. given a **Stream** of **Event**s, you can sort it by calling the .sorted() function on the **Stream**

```
events.stream().sorted().toList();
```

Comparator

- We can also use a **Comparator** for ordering, which allows us to sort objects that do not implement the **Comparable** interface

From the Java API:

```
int compare(T o1, T o2)
```

Compares object o1 with object o2 for order. Returns a
- negative integer (less than)
- Zero (equal to)
- positive integer (greater than) than object o2.

Comparator

Given a **Person** class;

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

    // other code here

}
```

Comparator

We use a separate class to implement **Comparator**

```
class AgeComparator implements Comparator<Person> {
    // sort by age in ascending order
    public int compare(Person p1, Person p2) {
        return p1.age - p2.age;
    }
}
```

Comparator

We can create different **Comparator** classes to compare different things

```
class NameComparator implements Comparator<Person> {
    // sort by name in ascending order
    public int compare(Person p1, Person p2) {
        return p1.name.compareTo(p2.name);
    }
}
```

However, this requires public access to a classes' attributes, whereas Comparable will not require that.

Comparator

Comparator classes can also compare multiple attributes at once

```
class AgeNameComparator implements Comparator<Person> {
    // sort by age in ascending order
    // ties in age are sorted by name in ascending order
    public int compare(Person p1, Person p2) {
    if (p1.age == p2.age) {
        return p1.name.compareTo(p2.name);
    } else {
        return p1.age - p2.age;
    }
}
```

Comparator

- Now that we have a **Comparator** to define the sorting order, we pass in an instance of the **Comparator** as input to the **sorted**() function

- e.g. given a **Stream** of people:

```
Stream.of(new Person(..), ..) //initialise Stream of Persons
      .sorted(new NameComparator()) //sort with Comparator
      .toList();
```

Comparator

- **Comparator**s can also be defined using lambda syntax.
- e.g. this is identical to using NameComparator:

```
Stream.of(new Person(..), ..) //initialise Stream of Persons
        .sorted((p1, p2) -> p1.name.compareTo(p2.name)) //lambda
        .toList();
```

{ ..

Practical Assessments

} ..

Overview

- 90 minutes
- Coding in the PE node (same thing that you do every lab)
- PA 1 is worth 15%, PA 2 20%
- Makeup PA1 will only be available if there is a valid reason for absence
- There will be no moderation for Makeup PA1

How It Works

- 90mins to code your implementation during the session

- Code will be uploaded to CodeCrunch and you will be given a moderation period of about one week (usually slightly more) to modify your code for correctness

Grading (Moderation)

- PA Score will be based on the amount of modifications made: Lesser changes = Higher Score
- Full marks if you finish the PA within the 90min assessment period **and** pass all hidden test cases on CodeCrunch
- Checkstyle is **not** included for the PA

Grading

You **must** score full marks (100/100) on CodeCrunch by the end of the moderation period, or you will score `0` for your PA.

Grading Principle

(Your code should) **Convince us that you were on the right track during the 90min session, and if given more time you would have gotten the correct solution**

Grading

A walkthrough will be uploaded sometime after the PA. You may wish to refer to it to complete the task and achieve an ‘A’, but you should still stick to your own implementation to minimize changes.

Grading

- Comments will be ignored during grading
- Checkstyle will not be graded
- You still need one file per Java class
- Functions/Variables that are unused will be deleted during grading. If you have any of these, we suggest you to delete them anyway for your own sake to make the code clearer

Specifics

PA1 Topic coverage: **ALL topics covered in only labs and exercises before PA1.**

- You may only use the school device in front of you
- You may refer to hardcopy notes
- You can **only** refer to the Java 21 API on the desktop (no other websites/resources/CodeCrunch)
- **You are only allowed to code inside the PE nodes**
- You may not bring your own keyboard

IMPORTANT

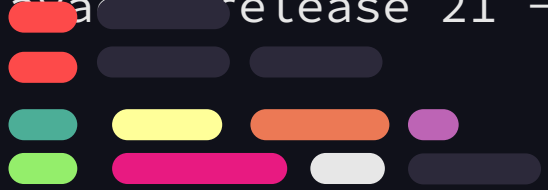
Please make sure to compile your code frequently.

We use it as a savepoint for your code should any technical fault occur.

Save your files on vim with :wq in command mode (press esc)

then, in the PE node, run:

```
ja[redacted]release 21 --enable-preview *.java
```



Tips

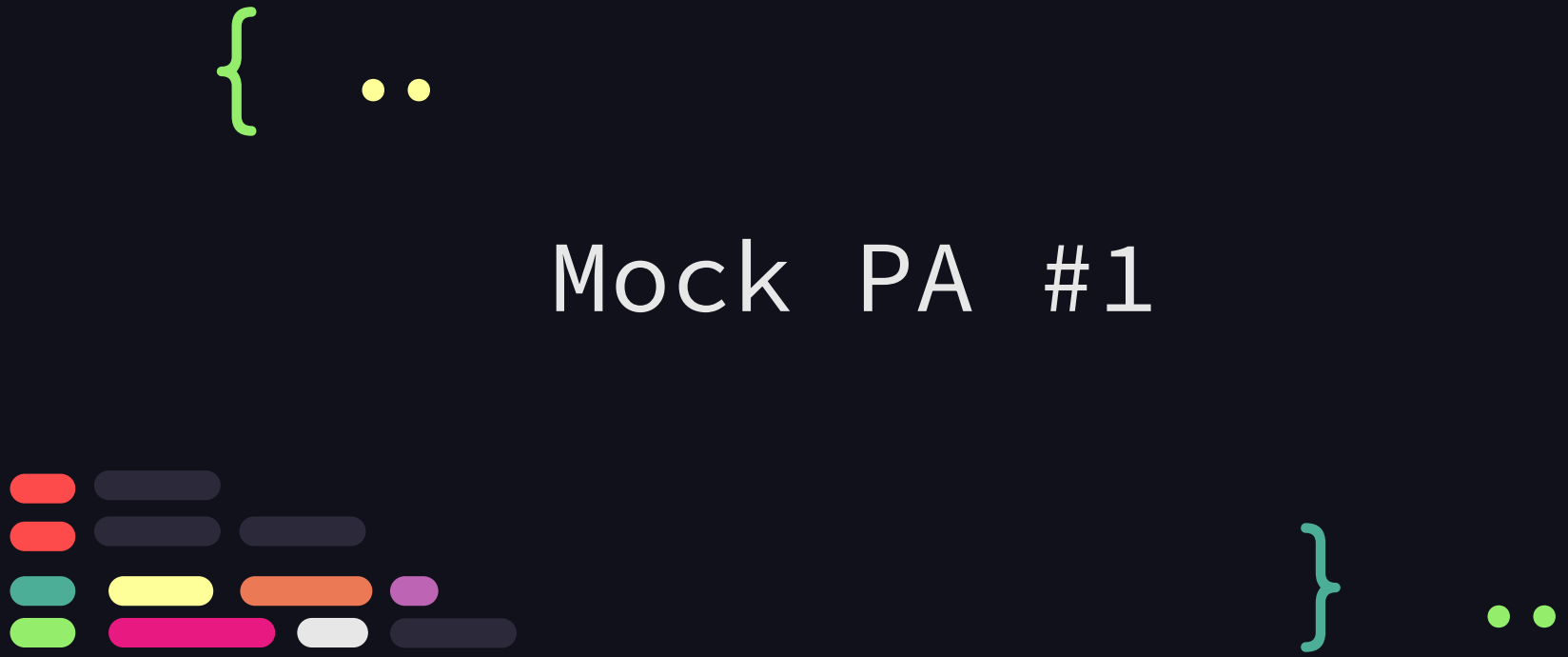
- Allocate some time to plan and think about your program's design before writing any code (**IMPORTANT!**)
- Read through all levels of the PA before proceeding to have a better understanding on what you are going to code (later levels may require you to tweak your implementation)

Tips

- The amount of modifications you make to your code will determine your score - Attempt other levels if you happen to be stuck, and do not be afraid to code out “imperfect” implementations/methods/classes to minimise the amount of changes you will need!

Tips

- Variable/Function/Class names should be meaningful to help your tutors understand your work during moderation!


Mock PA #1

Instructions

You are highly encouraged to read this portion of the index.pdf

This will guide you on what you're allowed to use for the task.

Take note!

There are altogether five levels. You are required to complete ALL levels.

You should keep to the constructs and programming discipline instilled throughout the module.

- Write each class/interface in its own file.
- Ensure that ALL object properties and class constants are declared private final.
- Ensure that that your classes are NOT cyclic dependent.
- Use only java libraries taught in class.
- You are NOT allowed to use java keywords/literals: `null`, instanceof (except within the typical `equals` method)
- You are NOT allowed to use Java reflection, i.e. `Object::getClasses` and other methods from `java.lang.Class`
- You are NOT allowed to use * wildcard imports.
- You are NOT allowed to define array constructs, e.g. `String[]` or using ellipsis, e.g. `String...`

You may assume that all tests provide valid arguments to methods; hence there is no need to validate method arguments.

Mock PA #1

No walkthrough today - this lab will be considered a mock PA

However, feel free to ask questions and suggest implementations

Try making use of some of the tips mentioned earlier!

Deadline: **2 Oct 2359** (Night before PA #1)