

Yeah java scoping is a bit more restrictive than python scoping

14:52

In java the variable scope is between the closest { above it and the closing } below it

14:53

```
for (int i = 1; i < numLoaders + 1; i++) {
    if (i % 3 == 0) {
        Service s = new Service(new RecycledLoader(i), cruise);
    } else {
        Service s = new Service(new Loader(i), cruise);
    }
    active = active.add(s);
    result = result.add(s);
}
continue;
```

aka only within the if statement

14:54

```
if (active.isEmpty() == true &&
expired.isEmpty() == true) {
    for (int i = 1; i < numLoaders + 1; i++)
{
    Service s;
    if (i % 3 == 0) {
        s = new Service(new
RecycledLoader(i), cruise);
    } else {
        s = new Service(new Loader(i),
cruise);
    }
    active = active.add(s);
    result = result.add(s);
}
continue;
```

edited 14:55

if you do this then it is within the whole for-loop

14:55

## Inheritance

Remember that methods with the same name can only exist if they are overridden or overloaded

(different return type or access type of method does will result in error)

Overloaded -> different **type/number/order** of arguments

Ensure that whatever is on the **right side** of the assignment aka **run-time type** is assigned to something **more general/parent class**

Preferably parent class 's methods do not return child class type

If right side is the parent class and its method returns a child class which is assigned to the child class on the left side

E.g. B extends A

A.give() returns B

B result = A.give() => will give compilation error, because if B has **its own child class methods**, result.method() won't work as **run-time type is of A which does not have the method**

**Design issues:** if you have cyclic dependency, create a new interface so that the class will now depend on the interface rather than the other class itself.

**Or**

If the method is the one causing the cyclic dependency, declare the class and method abstract  
Define the method instance outside under an anonymous inner class.

### Generics

Wrapper type conversions in List will not work

You can't new List() as List is an interface, must use new ArrayList

---

(d) `List<? super Integer> list = new ArrayList<int>();`

*Error. A generic type cannot be primitive type.*

Let's redefine the `max3` method to make use of the `Comparable` interface

```
<T> T max3(T a, T b, T c) {  
    T max = a;  
    if (b.compareTo(max) > 0) {  
        max = b;  
    }  
    if (c.compareTo(max) > 0) {  
        max = c;  
    }  
    return max;  
}
```

Does the above method work? What is the compilation error?

```
jshell> /open ...  
| Error:  
| cannot find symbol  
|   symbol:   method compareTo(T)  
|       if (b.compareTo(max) > 0) {  
|           ^-----^
```

Sub methods used under `max3` here has to be applicable and present in **all possible** classes of `T`

`<T extends Comparable<T>> T max3(T a, T b, T c) {`

Change to this ^

### Lazy Evaluation

Streams: Intermediate operations aka generate, iterate, limit vs Given a list.stream()

```
Supplier<Integer> supply = () -> urls.stream().map(x -> processUrl(x)).reduce(0, (x,y) -> x + y);
```

Optionals: `map(x -> () -> mapper.apply(this.get()))`, use of **orElseGet/or(takes in a Supplier that produces an Optional)**, change attributes to Supplier

`orElse(T) =>` the `T` refers to the `Optional<T>` that used `.orElse()`, not the first `Optional` in the pipeline, if you mapped it then used `orElse`, `orElse(T)` refers to the `R` returned from the map

1. Cache the result of the first evaluation, let `cache` be an `Optional` attribute, private but not `final`  
`Optional` because if you have not called `get()` yet, it will be null/empty

```

    T get() {
        T v = this.cache.orElseGet(this.supplier);
        this.cache = Optional.<T>of(v);
        return v;
    }

    <R> Lazy<R> flatMap(Function<T, Lazy<R>> mapper) {
        return Lazy.<R>of(() -> mapper.apply(this.get()).get());
    }

```

2. Or use overloaded methods, first method call will call .get() and then return the overloaded method with this value as the argument, **you use this when your method requires you to call the value multiple times in the pipeline**

```

abstract class Func {
    abstract int apply(int a);

    Func compose(Func other) {
        return new Func() {
            int apply(int x) {
                return Func.this.apply(other.apply(x)); // <-- take note!
            }
        };
    }
}

```

Remember to **add the class.** in front of any attributes or methods that you are using from the class itself

```

return new Predicate<T>() {
    public boolean test(T x) {
        return p1.test(x) && p2.test(x);
    }
}

```

Functional interface is an interface that has **only one** abstract method. You can only use lambda expressions if the interface you are implementing is a functional interface with a single

abstract method. If the interface has more than one abstract method or it's an abstract class, you use an anonymous inner class instead.

Anonymous inner classes are used to provide an implementation for a **static method in an interface**.

Static methods in an interface means **your instances of the interface** are unable to call these methods, only the interface itself is able to do so.

```
I.foo().foo(); // error  
I.foo();       // ok
```

Anonymous inner class will need to implement all the methods stated, able to have attributes and constructors.

```
static <E> ImList<E> of() {  
    return ImList.<E>of(List.of());  
}  
  
static <E> ImList<E> of(List<? extends E> list) {  
    return new ImList<E>() {  
  
        private final ArrayList<E> elems = new ArrayList<E>(list);  
  
        public ImList<E> add(E elem) {  
            List<E> newList = new ArrayList<E>(this.elems);  
            newList.add(elem);  
            return ImList.of(newList);  
        }  
    };  
}
```

```
abstract class Func {  
    abstract int apply(int x);  
}
```

```
Func f = new Func() {  
    int apply(int x) {  
        return 2 * x;  
    }  
}
```

```
Func g = new Func() {  
    int apply(int x) {  
        return 2 + x;  
    }  
}
```

Unable to use lambda expression ^

```
jshell> interface Func {  
...>     int apply(int a)  
...> }  
| created interface Func
```

3



```
jshell> Func f = x -> 2 * x;  
f ==> $Lambda$20/0x0000000800c0a000052cc8049
```

```

Func compose(Func other) {
    return new Func() {
        int apply(int x) {
            return Func.this.apply(other.apply(x)); // <-- take note!
        }
    };
}

```

Combining two streams often involve stream1.flatMap(x -> stream2.map( combining function))

Combining Optionals `this.solve(problemGet).or(() ->`  
`left(problemGet).solve(combiner) //Optional, compute first Optional`  
`.flatMap(leftS -> right(problemGet).solve(combiner) // Compute second`  
Optional within the first Optional and then map to combine both values together  
`.map(rightS -> combiner.apply(leftS,rightS))));`

Combining two functions => f.andThen(g) or f.compose(g)

Combining predicates => x -> p1.test(x) && p2.test(x)

=> p1.and(p2)  
=> p1.or(p2)

When you see an attribute being defined as an Optional, it means it might be null  
Optional return value inside it => orElse

Function to simulate BiFunction()=> `Function<? super E, Function<? super E, ? extends E>> f`

Function returns a function, use currying where you f.apply(a).apply(b)

Reduce(Terminal operation) returns type T (value stored in Stream), hence either your iterate/generate or your map has to return type T already.

**2 arguments Reduce returns type T, 3 arguments Reduce returns type U**  
**return words.stream()**  
`.reduce(0, (x,y) -> x+y.length(), (x,y) -> x+y);`

**Accumulator BiFunction takes in type T and type U and returns type U**  
**Binary Operator takes in two type T arguments and returns type T**

If you are required to know the indexes, use IntStream instead of Stream as you can use range() or its returning Integer

Shorten the list by half the size => IntStream. iterate(0, x < list.size(), x -> x + 2) => skip index

To get a List from Instream, you need to convert to stream first by either using boxed() or mapToObj()

```
Stream<String> stream = intStream.mapToObj(i -> "Number " + i);
```

To get a list from a stream, think of the final product which is a list of type T?, if so before you toList(), your Stream must already consist of that type T

```
<E> List<E> merging(List<? extends Pair<? extends E, ? extends E>> list) {  
    return list.stream().flatMap(x -> Stream.of(x.first(), x.second())).toList();  
}
```

### CompletableFuture

**Always assign completablefuture to a variable, so that you can at least .join() after**

thenApply() similar to map

thenCombine() and thenCompose() are similar to flatMap

**thenCombine()** is used to combine the results of two CompletableFuture instances that are independent of each other. This method takes two CompletableFuture instances as parameters and a **BiFunction** that combines their results.

**thenCompose()** is used to chain together multiple CompletableFuture instances that depend on each other. The result of the first operation is passed as a parameter to the second operation, and so on. thenCompose() returns a new CompletableFuture that completes with the result of the last operation in the chain.

CompletableFuture<A>.handle() used for exception handling -> takes in two inputs from the previous stage, -> if else statement to catch exceptions and return new object, check if result == NULL -> new A() if exception found

```
.handle((result, exception) -> {  
    if (exception != null) {  
        return -1;  
    } else {  
        return result * 2;  
    }  
});
```

thenAccept takes in a consumer, hence return type will be CompletableFuture<Void>, replaces thenApply if return value required is void

CompletableFuture.<A>completedFuture() is often used for **base cases, if else loops**

**Done message to signify all the async tasks have been completed, print at the end after you use join()**

If you want to split up paths, example  $f(a) = b$  and  $b$  is used for  $g(b) = c$  and  $h(b) = d$

Store  $f(a)$  as a CompletableFuture variable and use **thenApply** and **thenApplyAsync**  
**separately -> split up into two threads, will use the same one thread if we use thenApply for both and thus won't be done asynchronously**

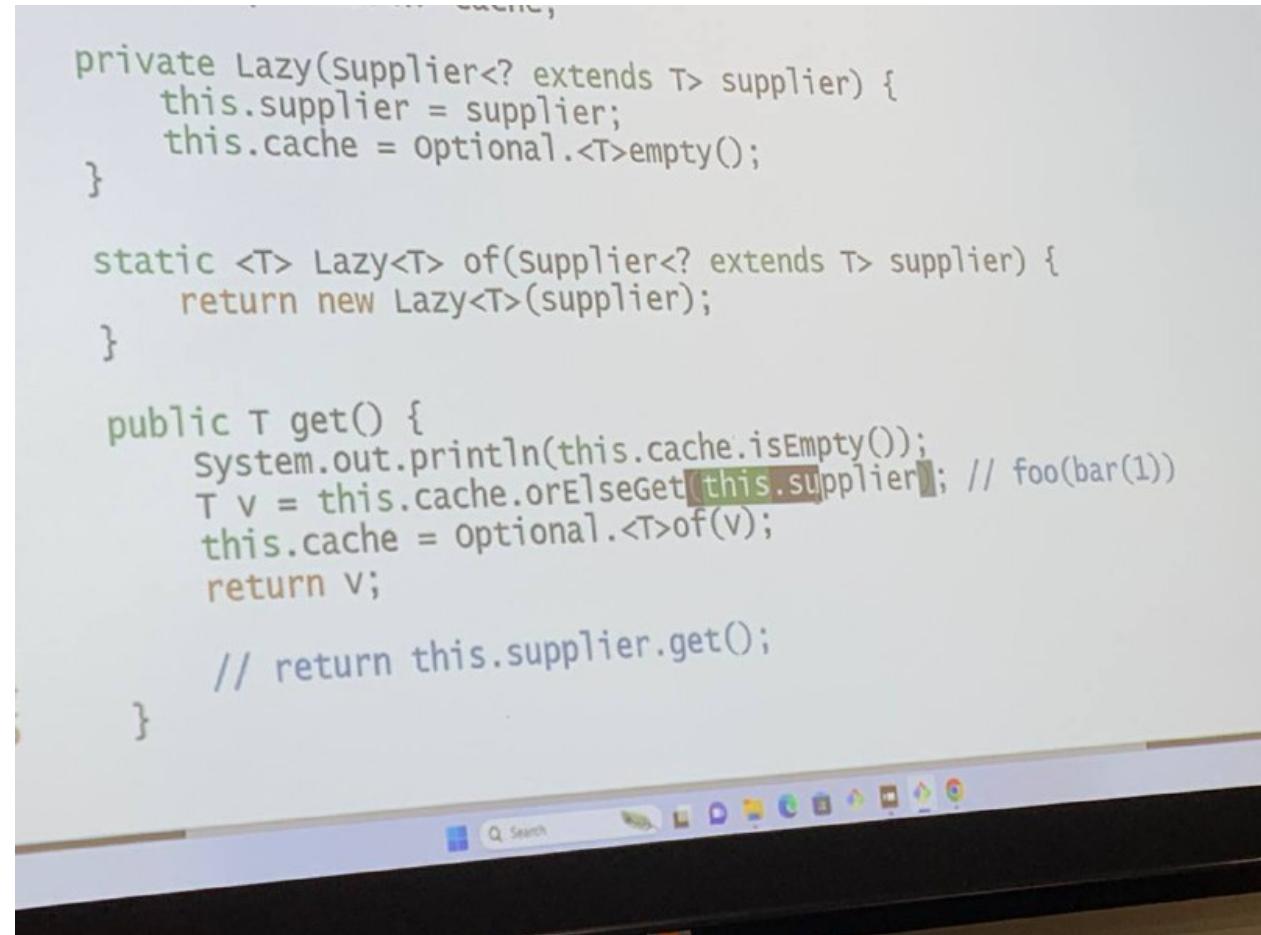
Return `cfc.thenCombine(cfd, (x,y) -> )`

### Joining List, Stream and CompletableFuture:

```
CompletableFuture<Integer> cf = urls.stream().map(x -> CompletableFuture
```

```
.supplyAsync(() -> processUrl(x)))  
.reduce(CompletableFuture.completedFuture(0),
```

```
(x,y) -> x.thenCombine(y, (a,b) -> a + b));
```



```
private Lazy(Supplier<? extends T> supplier) {  
    this.supplier = supplier;  
    this.cache = Optional.<T>empty();  
}  
  
static <T> Lazy<T> of(Supplier<? extends T> supplier) {  
    return new Lazy<T>(supplier);  
}  
  
public T get() {  
    System.out.println(this.cache.isEmpty());  
    T v = this.cache.orElseGet(this.supplier); // foo(bar(1))  
    this.cache = Optional.<T>of(v);  
    return v;  
    // return this.supplier.get();  
}
```

Should use `orElseGet(Supplier)` instead of `orElse(this.supplier.get())`

`orElseGet()` does not execute the supplier if a value is found in the optional, it's merely an address

`orElse()` executes the supplier via the `.get()`, hence executing it irregardless

Try to let `flatMap` and `Map` take in a supplier

The `orElseThrow()` method of [java.util.Optional class](#) in Java is used to get the value of this Optional instance if present. If there is no value present in this Optional instance, then this method throws the exception generated from the specified supplier.

Syntax:

```
public <X> T  
    orElseThrow(Supplier<X> exceptionSupplier)  
    throws X extends Throwable
```

**Parameters:** This method accepts [supplier](#) as a parameter of type X to throws an exception if there is no value present in this Optional instance.

**Return supplier:** This method returns the **value** of this Optional instance, if present. If there is no value present in this Optional instance, then this method throws the exception generated from the specified supplier.

```
jshell> mi.filter(x -> x % 2 == 0).orElseGet(() -> 2)  
$... ==> 2
```

```
jshell> mi.filter(x -> x % 2 == 1).orElseGet(() -> 2)  
$... ==> 1
```

```
jshell> Supplier<Optional<Circle>> supp = () -> {  
...>     System.out.println("beep!");  
...>     return Optional.<Circle>of(new Circle(new Point(0, 0), 1.0));}  
supp ==> $Lambda$23/0x000000000c13858@15615899
```

```
jshell> Supplier<Maybe<Integer>> suppMaybe = () -> Maybe.<Integer>of(2)
suppMaybe ==> $Lambda$24/0x00000008000adc48@5c7fa833

jshell> mo.filter(x -> x.hashCode() == 0).or(suppMaybe)
$... ==> Maybe[2]

jshell> mo.filter(x -> x.hashCode() == 1).or(suppMaybe)
$... ==> Maybe[1]
```

```
Optional.ofNullable(item).filter(i -> i.getCrystal() == Crystal.A)
.ifPresent(k -> player.getInventory.addItem(i));
```

```
jshell> primefactors(6)
$19 ==> [1, 2, 3, 6]

jshell> IntStream primefactors(int n) {
...>     return factors(n).filter(x -> isPrime(x));
...> }
|   created method primefactors(int)

jshell> primefactors(6).boxed().toList()
$21 ==> [2, 3]

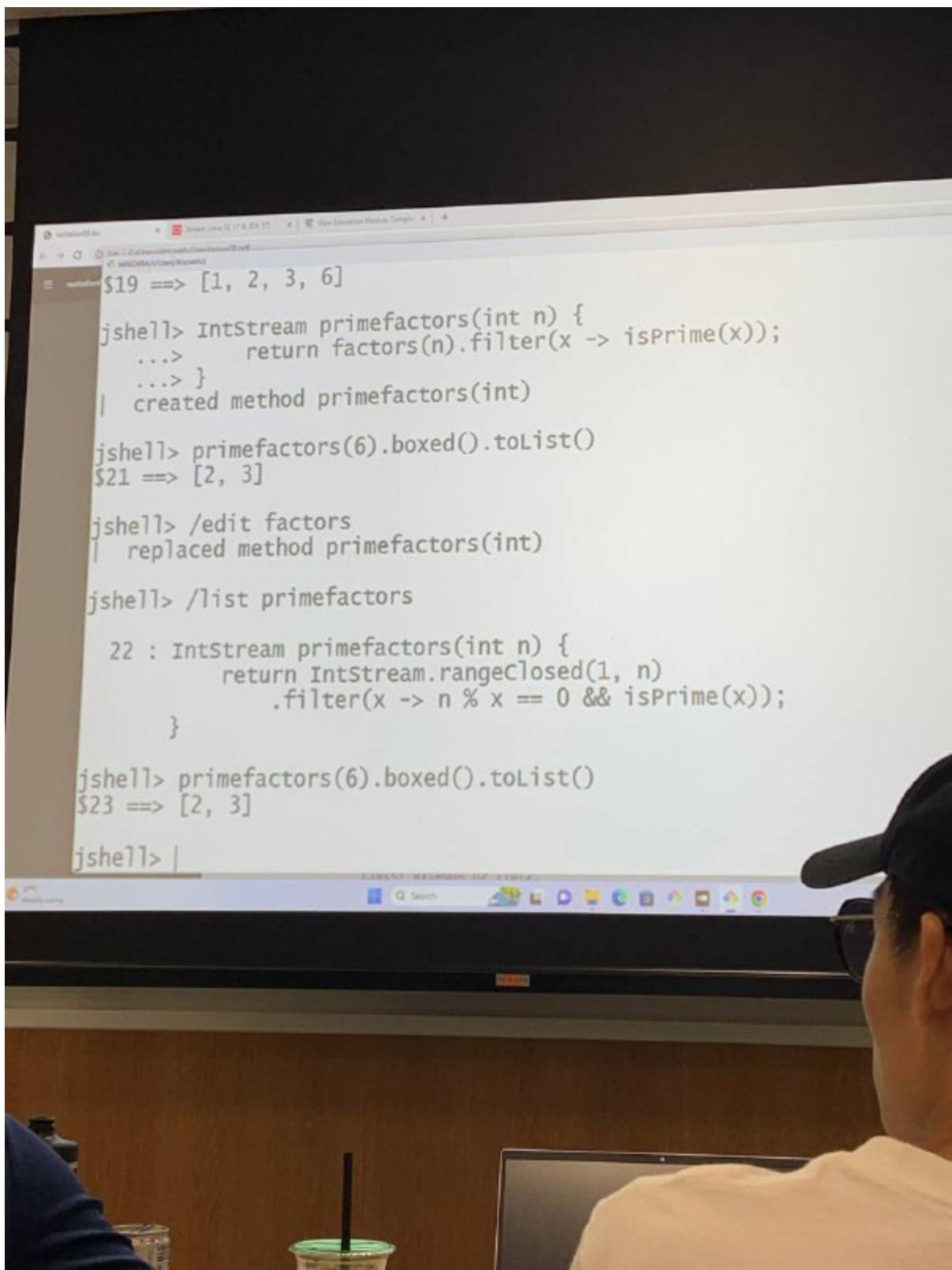
jshell> /edit factors
|   replaced method primefactors(int)

jshell> /list primefactors

22 : IntStream primefactors(int n) {
    return IntStream.rangeClosed(1, n)
        .filter(x -> n % x == 0 && isPrime(x));
}

jshell> primefactors(6).boxed().toList()
$23 ==> [2, 3]

jshell> |
```



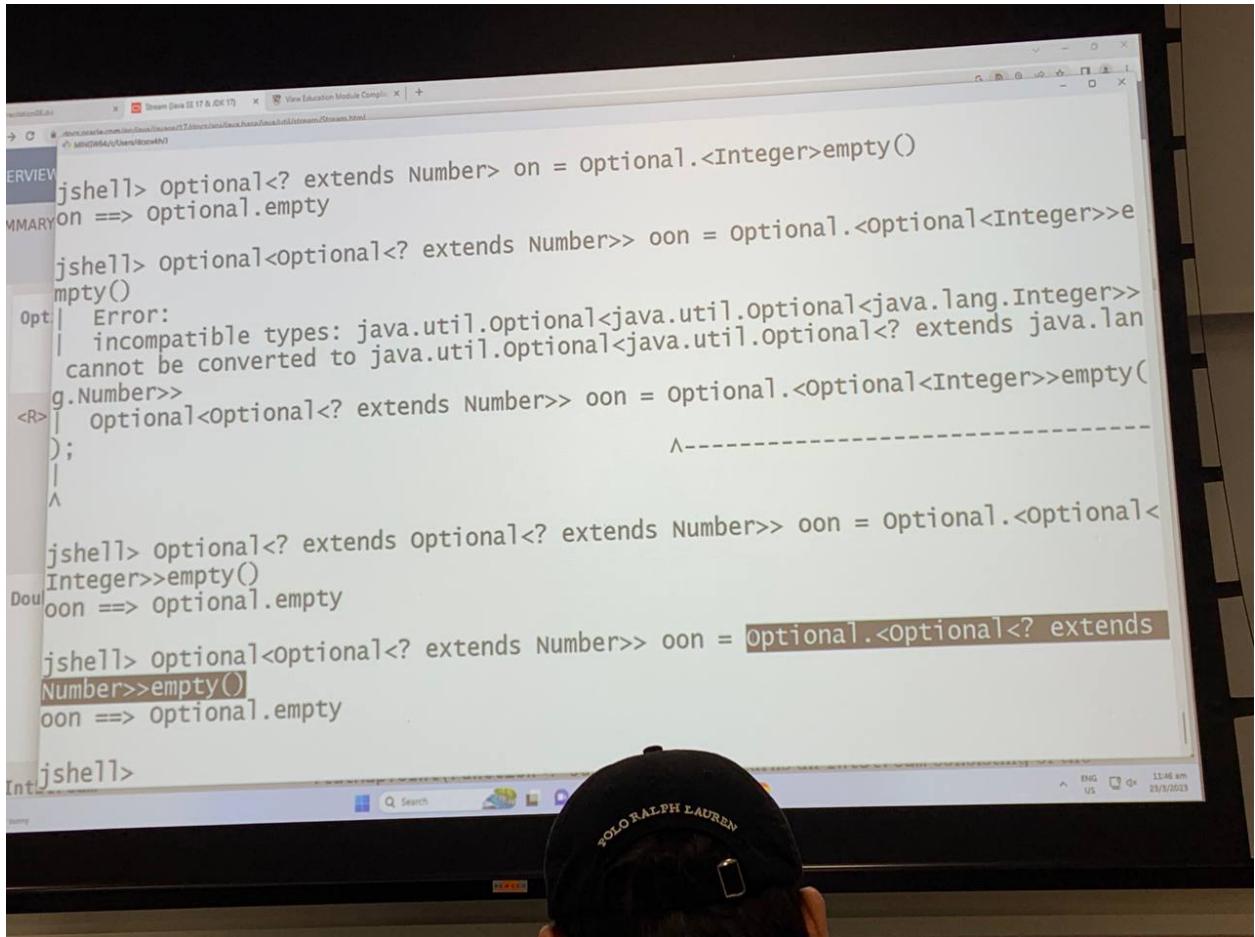
A screenshot of a Java REPL session in a browser window titled "jshell". The session shows the creation and use of two methods: `primefactors` and `omega`.

```
jshell> /edit factors
| replaced method primefactors(int)
jshell> /list primefactors
22 : IntStream primefactors(int n) {
    return IntStream.rangeClosed(1, n)
        .filter(x -> n % x == 0 && isPrime(x));
}
jshell> primefactors(6).boxed().toList()
$23 ==> [2, 3]
Long
jshell> LongStream omega(int n) {
    ...>     return IntStream.rangeClosed(1,n)
    ...>         .mapToLong(x -> primefactors(x).count());
    ...> }
| created method omega(int)
Opt:jshell> omega(10).boxed().toList()
$25 ==> [0, 1, 1, 1, 1, 2, 1, 1, 1, 2]
jshell> |
```

A person wearing a black Polo Ralph Lauren cap is looking at a computer screen displaying Java code in a terminal window. The code is being entered into a JShell session. The code defines a Stream<Integer> fibo(int n) method that uses a Stream API to generate Fibonacci numbers. The terminal window shows the command-line interface with various tabs like OVERVIEW, SUMMARY, and Intellisense.

```
jshell> stream<Integer> fibo(int n) {  
    ...>     return null;  
    ...> }  
| created method fibo(int)  
  
jshell> /edit fibo  
| modified method fibo(int), however, it cannot be invoked until class Pair is  
declared  
  
Dou  
jshell> /open Pair.java  
  
jshell> /list fibo  
Int  
Long  
Opt  
27 : Stream<Integer> fibo(int n) {  
    Pair<Integer, Integer> startPair = new Pair<Integer, Integer>(0, 1);  
    UnaryOperator<Pair<Integer, Integer>> nextPair = pair ->  
        new Pair<Integer, Integer>(pair.second(),  
            pair.first() + pair.second());  
    return Stream.<Pair<Integer, Integer>>iterate(startPair, nextPair)  
        .map(pair -> pair.second()) // Stream<Integer>  
        .limit(n);  
}  
  
Opt:jshell> fibo(10)
```

```
jshell> <T,U,R> Stream<R> product(List<? extends T> list1,
   ...> List<? extends U> list2,
   ...> BiFunction<? super T, ? super U, ? extends R> func) {
   ...>     return list1.stream()
   ...>         .flatMap(x -> list2.stream().map(y -> func.apply(x,y)));
   ...>     }
   | created method product(List<? extends T>,List<? extends U>,BiFunction<? super T, ? super U, ? extends R>)
jshell> product(List.of(1,2,3,4), List.of("A","B"), (x,y) -> x + ":" + y)
$32 ==> java.util.stream.ReferencePipeline$7@1b701da1
jshell> product(List.of(1,2,3,4), List.of("A","B"), (x,y) -> x + ":" + y).toList()
$33 ==> [1:A, 1:B, 2:A, 2:B, 3:A, 3:B, 4:A, 4:B]
jshell>
```



```
Roster add(String name, String module, String assessment, String grade) {
    Student s = super.get(name).orElse(new Student(name));
    Module m =
}
```

`.get(name)` here returns an `Optional<Student>`

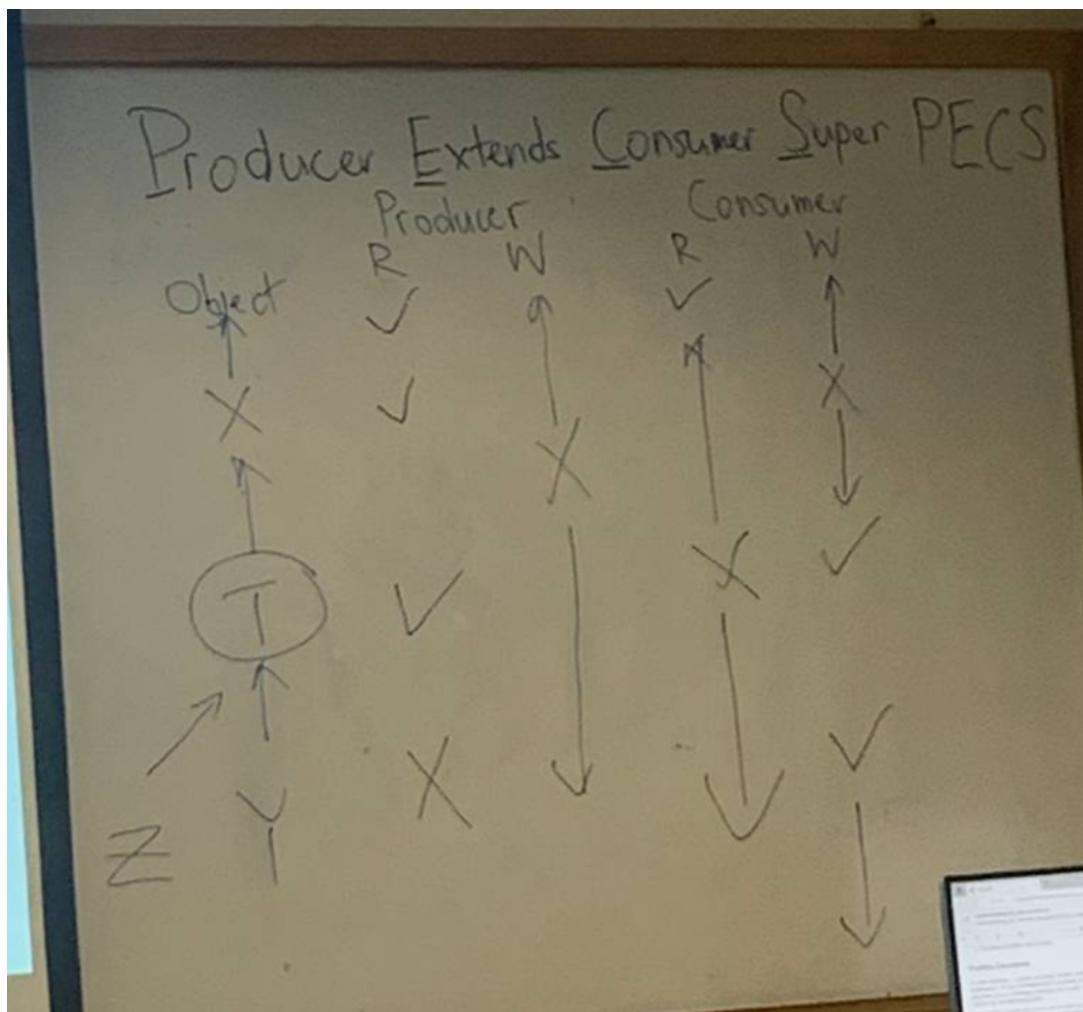
By adding the `.orElse()`, it means that if `get(name) != Null`, it will return the **VALUE** of the `Optional<Student>`, which is the `Student` object, else return the other value

## PECS

PECS: Producer **extends**, Consumer **super**

Underlying principle: Use `<? extends T>` if you want to read from a collection (producer), and `<? super T>` if you want to add to a collection `<? super T>`

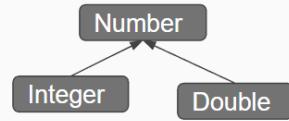
Use `<T>` if you want to read from and add to a collection



## Producer Extends

```
List<? extends Number> foo3 = new ArrayList<Number>(); // Number "extends"  
Number (in this context)  
List<? extends Number> foo3 = new ArrayList<Integer>(); // Integer extends Number  
List<? extends Number> foo3 = new ArrayList<Double>(); // Double extends Number
```

The above assignments are all valid.



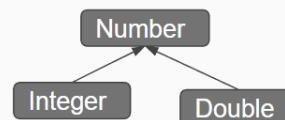
Reading from foo3 (must be legal for **all** possible foo3 assignments):

- Number can be read because any of the lists that could be assigned to foo3 contain a Number or a subclass of Number
- You cannot read an Integer because foo3 could be pointing to a List<Double>
- You cannot read a Double because foo3 could be pointing to a List<Integer>

## Producer Extends

```
List<? extends Number> foo3 = new ArrayList<Number>(); // Number "extends"  
Number (in this context)  
List<? extends Number> foo3 = new ArrayList<Integer>(); // Integer extends Number  
List<? extends Number> foo3 = new ArrayList<Double>(); // Double extends Number
```

The above assignments are all valid.



Writing to foo3 (must be legal for **all** possible foo3 assignments):

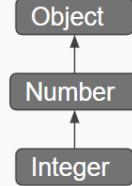
- You cannot add an Integer because foo3 could be pointing to a List<Double>
- You cannot add a Double because foo3 could be pointing to a List<Integer>
- You cannot add a Number because foo3 could be pointing to a List<Integer>

Conclusion: By using **extends**, only Numbers (or subclasses of Number) can be read from foo3 (therefore foo3 is a producer of Numbers)

## Consumer Super

```
List<? super Integer> foo3 = new ArrayList<Integer>(); // Integer is a  
"superclass" of Integer (in this context)  
List<? super Integer> foo3 = new ArrayList<Number>(); // Number is a superclass  
of Integer  
List<? super Integer> foo3 = new ArrayList<Object>(); // Object is a superclass  
of Integer
```

The above assignments are all valid.



Writing to foo3 (must be legal for **all** possible foo3 assignments):

- You can add an Integer (allowed by all 3 lists)
- You can add an instance of a **subclass** of Integer (allowed by all 3 lists)
- You cannot add a Double because foo3 might point to a List<Integer>
- You cannot add a Number because foo3 might point to a List<Integer>
- You cannot add an Object because foo3 might point to a List<Integer>

You can't make a Number into an Integer, you only can make an Integer into a Number (specific into general)

## Consumer Super

```
List<? super Integer> foo3 = new ArrayList<Integer>(); // Integer is a  
"superclass" of Integer (in this context)  
List<? super Integer> foo3 = new ArrayList<Number>(); // Number is a superclass  
of Integer  
List<? super Integer> foo3 = new ArrayList<Object>(); // Object is a superclass  
of Integer
```

The above assignments are all valid.

You can't read from a List<? super T> because you do not know what kind of List it points to (unless you assign the item to an Object instance; redundant as mentioned before). You can only add items of type T or a subclass of T to the list.

Conclusion: By using **super**, only Integers (or subclasses of Integer) can be added to foo3 (therefore foo3 is a consumer of Integers)

## Producer Extends, Consumer Super: Function<T , R>

Function<T , R>

What do T and R correspond to?

### Method Summary

All Methods	Static Methods	Instance Methods	Abstract Methods	Default Methods
Modifier and Type	Method		Description	
R	apply(T t)		Applies this function to the given argument.	

If I were to create a method that takes in a function, which bounded wildcard should I use?

1. Function<? extends T , ? extends R>
2. Function<? super T , ? super R>
3. Function<? super T , ? extends R>
4. Function<? extends T , ? super R>

Depends, but most often the use case will be 3, since the function is consuming a value of type T, and producing a value of type R.



An API (Application Programming Interface) is the way **your application communicates with some software component, typically a library**. A typical example is the Java API. It defines **lots of classes and methods that can be used** by your application.

Tell-don't-Ask: basically telling you to encapsulate all your computations into methods, then you just call the methods to do them for you

Private states, public getters and setters

Javac filename.java

Java filename

```
java -jar ./checkstyle-8.2-all.jar -c ./cs2030_checks.xml *.java
```

Use jshell on vim, /open file and then create new object to test your code in the java file

/exit

Or jshell filename then run the trial cases

```
java filename < filename.in (assign input file to test out in java file)
```

v shift v and control v

gg (go to the first line of file)

G (go to the last line of file)

gg=G fix indentation

W or b to jump words forward/backward

/any word) , N/n (to go the next occurrence of the word you are searching for)

vim -p file file (to open multiple files in multiple tabs)

:vsplit (split screen)

Control W, vim arrow keys to jump screens

**Esc + gg + dG** delete all lines

explorer.exe .

Compile-Time type: dictates the methods it can call (**Circle**)

Run-Time type: actual method called during run time (**FilledCircle**), referenced

Circle c = new FilledCircle()

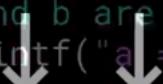
Can only call methods that belong to super/parent class, overriding methods in FilledCircle will be called instead of the parent's

**Static methods/attributes** can be accessed without creating an object of a class

Static binding uses Type information for binding (Compile-Time) - overloaded methods - calls the correct method of the **correct type argument**

**IF your run time object type has no overridden methods, only overloading methods, it will call the compile time type object's methods instead**

Dynamic binding uses Objects to resolve binding (Run-Time) - overridden methods - calls the correct method from the **correct object**

```
class Main {  
  
    public static void main(String[] args) {  
        Scanner sc = new Scanner(System.in);  
        int a = sc.nextInt();  
        double b = sc.nextDouble();  
  
        if (a < 0 || b < 0) {  
            return;  
        }  
  
        // See if a and b are correct.  
        System.out.printf("a = %d, b = %.2f\n", a, b);  
          
        // Print a ^ b.  
        System.out.println(Math.pow(a, b));  
    }  
}  
starting
```

Scan here is to read the input arguments

All under main()

```
// See if a and b are correct.  
System.out.printf("a = %d, b = %f\n", a, b);  
  
// Print a ^ b.  
System.out.println(Math.pow(a, b));  
  
// Print gcd(a, floor(b))  
System.out.println(findGcd(a, (int) b));  
  
// Print b fibonacci numbers > a.  
findFibonacci(a, (int) b)  
}
```

You can't just print the array out, you will get the memory space instead, need to use a for loop and use string concatenation “[“ + ..... Or you can just use the inbuilt function under the Arrays library

```
1 import java.util.Scanner;  
2 import java.util.Arrays;  
3  
4  
System.out.println(Math.pow(a, b));  
  
// Print gcd(a, floor(b))  
System.out.println(findGcd(a, (int) b));  
  
// Print b fibonacci numbers > a.  
// System.out.println(findFibonacci(a, (int) b));  
System.out.println(Arrays.toString(findFibonacci(a, (int) b)));  
}
```

findGCD, isFibonacci ..... are helper functions

```
static int[] findFibonacci(int a, int b) {  
    int[] result = new int[b];  
    int i = 0;  
    while (i < b) {  
        if (isFibonacci(a)) {  
            result[i++] = a;  
        }  
        ++a;  
    }  
    return result;  
}
```

```
    return result;  
}  
  
static boolean isFibonacci(int n) {  
    return isPerfectSquare(5 * n * n + 4) ||  
        isPerfectSquare(5 * n * n - 4);  
}  
  
static boolean isPerfectSquare(int n) {  
    int s = (int) Math.sqrt(n);  
    return s * s == n;  
}
```

Typecast/Convert the result of the square root into integer

Static keyword is to bind attribute/method to class

Static attribute is useful if you have many instances of that class and you want to update (**in the constructor**) a counter of the total number of all those instances; this static attribute which is bound to the class is updated each time the constructor is called.

Static method can be called without creating an object of the class

Main class: main logic of program

From the opened java file, enter :tabedit Main.java to open the other file on a separate tab

**Compile the Main.java and run it to run both files**

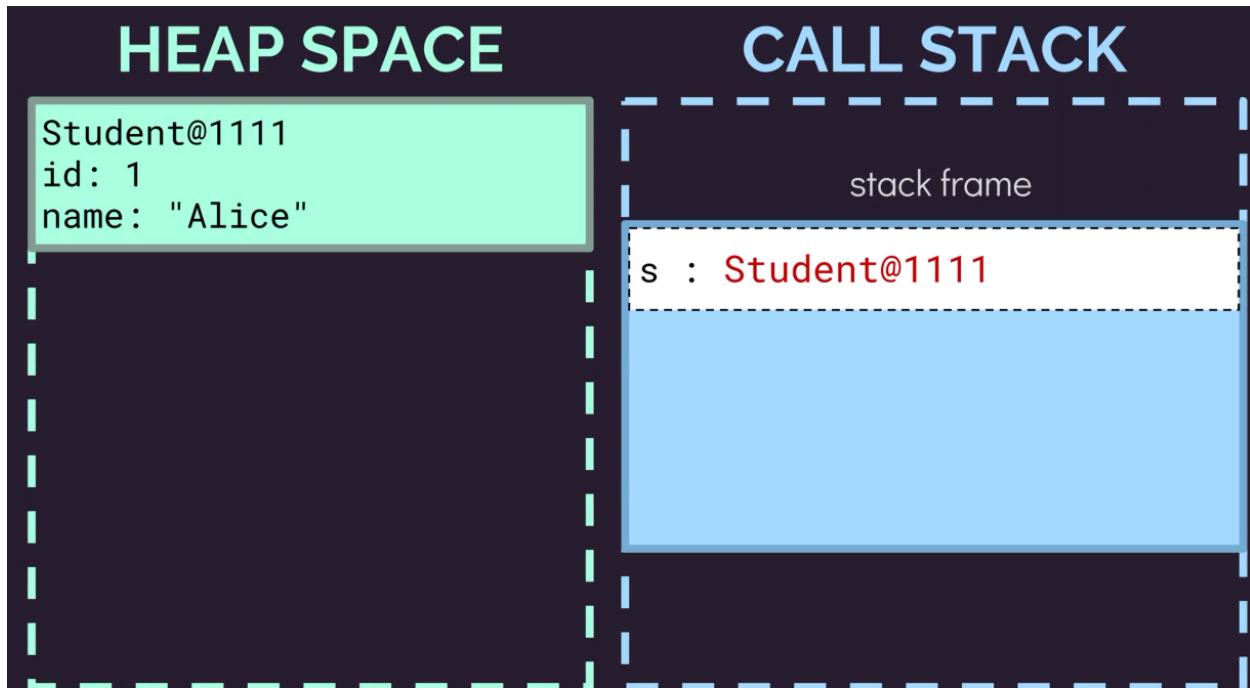
```
Student.java Main.java
1 class Main {
2
3     public static void main(String[] args) {
4         Student s = new Student(1, "Alice");
5         s.sayHello();
6         Student.createStudent().sayHello();
7     }
8 }
~
```

Stack: stores **references** of objects, stores **values** of primitive types (when  $j = i$  and  $i$  value is changed thereafter, only the value of  $i$  changes,  **$j$  remains as the old value of  $i$** )

```
public static void main(String[] args) {
    int a = 1;
    coolMethod(a);
    System.out.println(1);
}

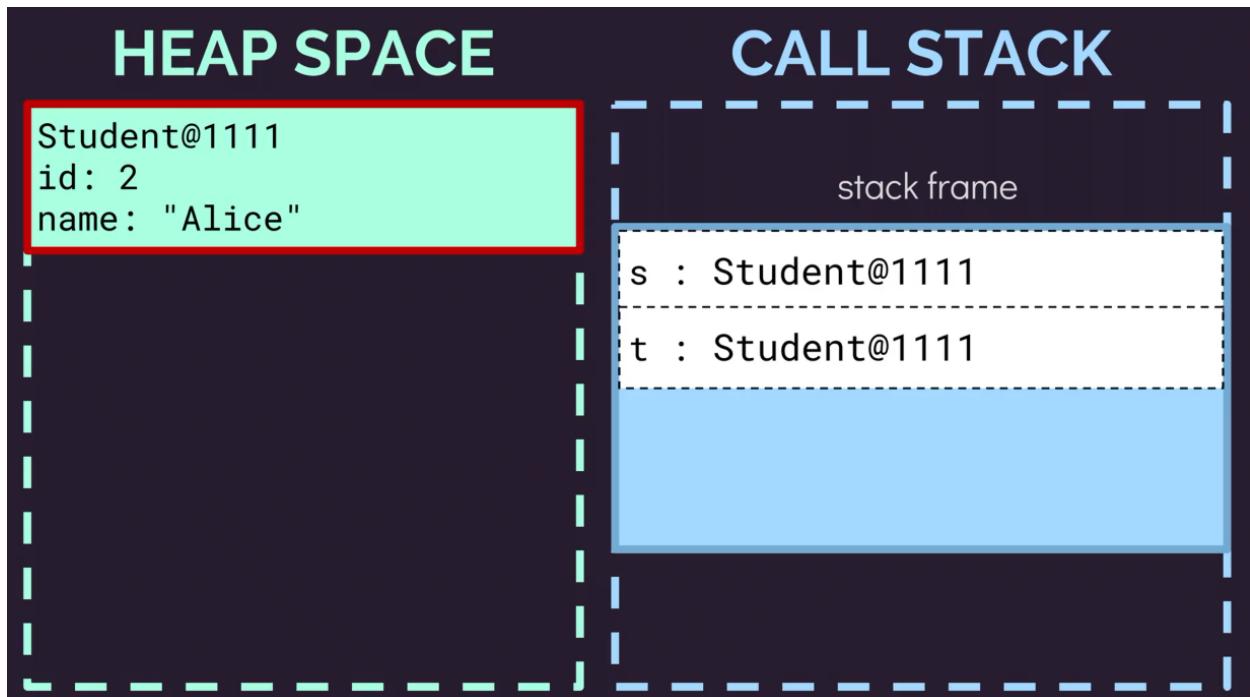
static void coolMethod(int a) {
    a = 2;
}
```

When coolMethod is called here, it changes the value of  $a$  to 2 temporarily and when the function is done, **it drops off the stack and the value of  $a$  is still 1**



When `s = t`, `s.id = 1`, `t.id = 1`

and when `s.id = 2`, `t.id` also changes to 2 as both `s` and `t` are pointing to the same memory location of the object



Arrays:

Memory stored under stack, array space stored under heap space

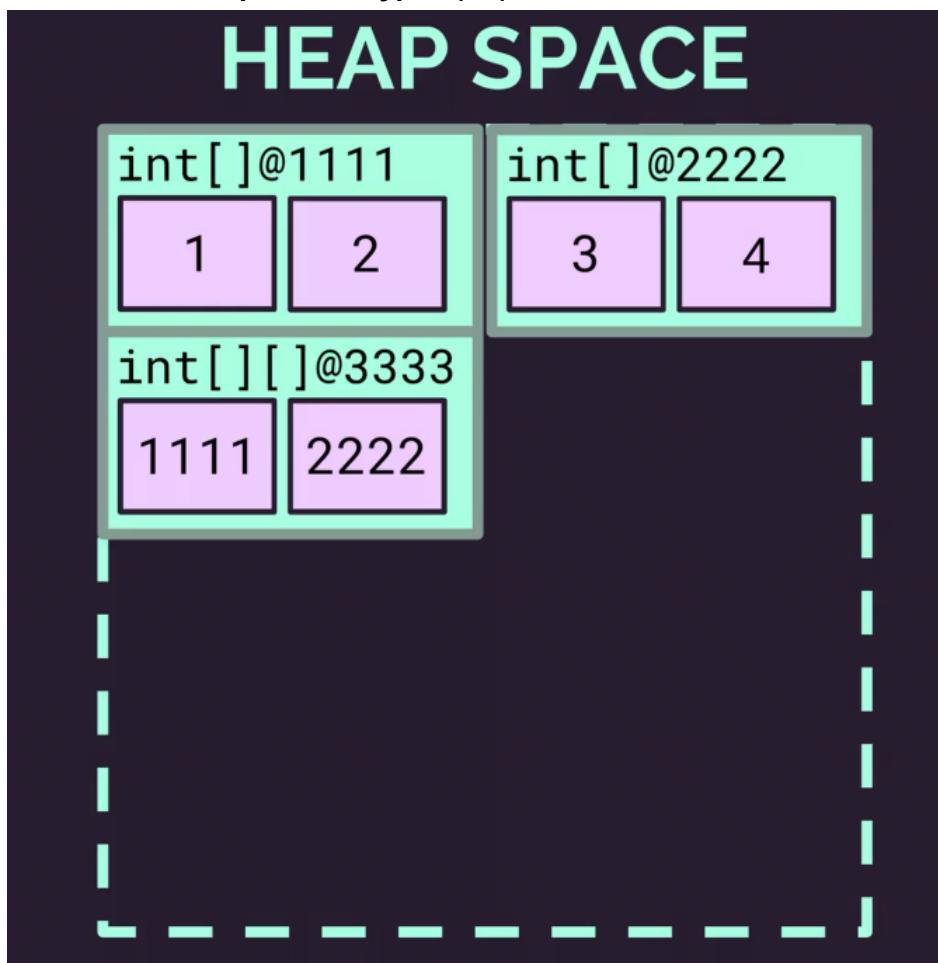
Multi-dimensional arrays:

```
Int [] [] c = new int [] [] {{1,2}, {3,4}};
```

==

```
Int [] [] c = int [] [] @3333;
```

1 2 3 4 stored as primitive types (int)



### Encapsulation

- Hide implementation details and state of objects from clients using access modifiers
- Can be achieved by declaring all the **variables in the class as private** and writing **public methods in the class to set and get** the values of variables.
- Changing of properties done under mutators/setters e.g. `this.curr = input`

### Abstraction

- It is used to hide the unnecessary information or data from the user but shows the essential data that is useful for the user.

Avoiding Cyclic Dependencies:

If class A has class B within itself as a property, then all methods involving class A should be contained within class A and **not within class B**.

**Has-a relationship:**

Circle has a point, the **point stored in the circle object is the memory value** which references to a point object

Point's properties and getters should be **private to prevent Circle object from accessing them directly**

**Avoid state-mutating void methods, return a new object instead**

Polymorphism

Children class 's override methods can only return objects that are **either the parent class or more specific**, can't be more general due to substitution principle (compile time type)

Interface & Abstract:

By default, all methods under Interface are public and abstract until we do not declare it as default and properties are static and final.

ANY methods declared under Interface state class must be public in the children classes and they must have those methods

Difference between **inheritance and interface** -> inheritance you can just use your parent class 's methods but as for interface, you have to define the method on your own and override it,

**Interface allows the children classes to inherit from more than one parent class, multiple inheritances**

```

1 interface Shape {
2     /* abstract */ public double getArea();
3 }

4 
5 
6 
7 
8 
9 
10 
11 
12 
13 

Shape.java
7         this.height = height;
8     }
9
10    @Override
11    public double getArea() {
12        return width * height;
13    }
Rectangle.java
6
7
8    @Override
9    public double getArea() {
10        return Math.PI * radius * radius;
11    }
Circle.java

```

If you want more than 1 abstract methods to be implemented in your “child” class, you use **Interface**

```

1 interface Shape {
2     abstract double getArea();
3 }
```

**abstract double getArea()** here means that your children classes need to override **getArea()**

Return type cannot be more general, only more specific

```

Scalable.java
1 class Rectangle implements Shape, Scalable {
2     private final int width;
3 }
```

**Under rectangle class, scale method can return rectangle object instead of a scalable object as a rectangle is also a scalable**

The screenshot shows a Java code editor with two files open:

- Scalable.java**:

```
1 interface Scalable {  
2     Scalable scale(int factor);  
3 }  
4  
5  
6  
7  
8  
9  
10  
11  
12 }
```
- Rectangle.java**:

```
13     }  
14  
15     @Override  
16     public Rectangle scale(int factor) {  
17         return new Rectangle(this.width * factor, this.height * factor);  
18     }  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48  
49  
50  
51  
52  
53  
54  
55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100  
101  
102  
103  
104  
105  
106  
107  
108  
109  
110  
111  
112  
113  
114  
115  
116  
117  
118  
119  
120  
121  
122  
123  
124  
125  
126  
127  
128  
129  
130  
131  
132  
133  
134  
135  
136  
137  
138  
139  
140  
141  
142  
143  
144  
145  
146  
147  
148  
149  
150  
151  
152  
153  
154  
155  
156  
157  
158  
159  
160  
161  
162  
163  
164  
165  
166  
167  
168  
169  
170  
171  
172  
173  
174  
175  
176  
177  
178  
179  
180  
181  
182  
183  
184  
185  
186  
187  
188  
189  
190  
191  
192  
193  
194  
195  
196  
197  
198  
199  
200  
201  
202  
203  
204  
205  
206  
207  
208  
209  
210  
211  
212  
213  
214  
215  
216  
217  
218  
219  
220  
221  
222  
223  
224  
225  
226  
227  
228  
229  
230  
231  
232  
233  
234  
235  
236  
237  
238  
239  
240  
241  
242  
243  
244  
245  
246  
247  
248  
249  
250  
251  
252  
253  
254  
255  
256  
257  
258  
259  
260  
261  
262  
263  
264  
265  
266  
267  
268  
269  
270  
271  
272  
273  
274  
275  
276  
277  
278  
279  
280  
281  
282  
283  
284  
285  
286  
287  
288  
289  
290  
291  
292  
293  
294  
295  
296  
297  
298  
299  
300  
301  
302  
303  
304  
305  
306  
307  
308  
309  
310  
311  
312  
313  
314  
315  
316  
317  
318  
319  
320  
321  
322  
323  
324  
325  
326  
327  
328  
329  
330  
331  
332  
333  
334  
335  
336  
337  
338  
339  
340  
341  
342  
343  
344  
345  
346  
347  
348  
349  
350  
351  
352  
353  
354  
355  
356  
357  
358  
359  
360  
361  
362  
363  
364  
365  
366  
367  
368  
369  
370  
371  
372  
373  
374  
375  
376  
377  
378  
379  
380  
381  
382  
383  
384  
385  
386  
387  
388  
389  
390  
391  
392  
393  
394  
395  
396  
397  
398  
399  
400  
401  
402  
403  
404  
405  
406  
407  
408  
409  
410  
411  
412  
413  
414  
415  
416  
417  
418  
419  
420  
421  
422  
423  
424  
425  
426  
427  
428  
429  
430  
431  
432  
433  
434  
435  
436  
437  
438  
439  
440  
441  
442  
443  
444  
445  
446  
447  
448  
449  
450  
451  
452  
453  
454  
455  
456  
457  
458  
459  
460  
461  
462  
463  
464  
465  
466  
467  
468  
469  
470  
471  
472  
473  
474  
475  
476  
477  
478  
479  
480  
481  
482  
483  
484  
485  
486  
487  
488  
489  
490  
491  
492  
493  
494  
495  
496  
497  
498  
499  
500  
501  
502  
503  
504  
505  
506  
507  
508  
509  
510  
511  
512  
513  
514  
515  
516  
517  
518  
519  
520  
521  
522  
523  
524  
525  
526  
527  
528  
529  
530  
531  
532  
533  
534  
535  
536  
537  
538  
539  
540  
541  
542  
543  
544  
545  
546  
547  
548  
549  
550  
551  
552  
553  
554  
555  
556  
557  
558  
559  
559  
560  
561  
562  
563  
564  
565  
566  
567  
568  
569  
569  
570  
571  
572  
573  
574  
575  
576  
577  
578  
579  
579  
580  
581  
582  
583  
584  
585  
586  
587  
588  
589  
589  
590  
591  
592  
593  
594  
595  
596  
597  
598  
599  
599  
600  
601  
602  
603  
604  
605  
606  
607  
608  
609  
609  
610  
611  
612  
613  
614  
615  
616  
617  
618  
619  
619  
620  
621  
622  
623  
624  
625  
626  
627  
628  
629  
629  
630  
631  
632  
633  
634  
635  
636  
637  
638  
639  
639  
640  
641  
642  
643  
644  
645  
646  
647  
648  
649  
649  
650  
651  
652  
653  
654  
655  
656  
657  
658  
659  
659  
660  
661  
662  
663  
664  
665  
666  
667  
668  
669  
669  
670  
671  
672  
673  
674  
675  
676  
677  
678  
679  
679  
680  
681  
682  
683  
684  
685  
686  
687  
688  
689  
689  
690  
691  
692  
693  
694  
695  
696  
697  
698  
699  
699  
700  
701  
702  
703  
704  
705  
706  
707  
708  
709  
709  
710  
711  
712  
713  
714  
715  
716  
717  
718  
719  
719  
720  
721  
722  
723  
724  
725  
726  
727  
728  
729  
729  
730  
731  
732  
733  
734  
735  
736  
737  
738  
739  
739  
740  
741  
742  
743  
744  
745  
746  
747  
748  
749  
749  
750  
751  
752  
753  
754  
755  
756  
757  
758  
759  
759  
760  
761  
762  
763  
764  
765  
766  
767  
768  
769  
769  
770  
771  
772  
773  
774  
775  
776  
777  
778  
779  
779  
780  
781  
782  
783  
784  
785  
786  
787  
788  
789  
789  
790  
791  
792  
793  
794  
795  
796  
797  
798  
799  
799  
800  
801  
802  
803  
804  
805  
806  
807  
808  
809  
809  
810  
811  
812  
813  
814  
815  
816  
817  
818  
819  
819  
820  
821  
822  
823  
824  
825  
826  
827  
828  
829  
829  
830  
831  
832  
833  
834  
835  
836  
837  
838  
839  
839  
840  
841  
842  
843  
844  
845  
846  
847  
848  
849  
849  
850  
851  
852  
853  
854  
855  
856  
857  
858  
859  
859  
860  
861  
862  
863  
864  
865  
866  
867  
868  
869  
869  
870  
871  
872  
873  
874  
875  
876  
877  
878  
879  
879  
880  
881  
882  
883  
884  
885  
886  
887  
888  
889  
889  
890  
891  
892  
893  
894  
895  
896  
897  
898  
899  
899  
900  
901  
902  
903  
904  
905  
906  
907  
908  
909  
909  
910  
911  
912  
913  
914  
915  
916  
917  
918  
919  
919  
920  
921  
922  
923  
924  
925  
926  
927  
928  
929  
929  
930  
931  
932  
933  
934  
935  
936  
937  
938  
939  
939  
940  
941  
942  
943  
944  
945  
946  
947  
948  
949  
949  
950  
951  
952  
953  
954  
955  
956  
957  
958  
959  
959  
960  
961  
962  
963  
964  
965  
966  
967  
968  
969  
969  
970  
971  
972  
973  
974  
975  
976  
977  
978  
979  
979  
980  
981  
982  
983  
984  
985  
986  
987  
988  
989  
989  
990  
991  
992  
993  
994  
995  
996  
997  
998  
999  
999  
1000  
1001  
1002  
1003  
1004  
1005  
1006  
1007  
1008  
1009  
1009  
1010  
1011  
1012  
1013  
1014  
1015  
1016  
1017  
1018  
1019  
1019  
1020  
1021  
1022  
1023  
1024  
1025  
1026  
1027  
1028  
1029  
1029  
1030  
1031  
1032  
1033  
1034  
1035  
1036  
1037  
1038  
1039  
1039  
1040  
1041  
1042  
1043  
1044  
1045  
1046  
1047  
1048  
1049  
1049  
1050  
1051  
1052  
1053  
1054  
1055  
1056  
1057  
1058  
1059  
1059  
1060  
1061  
1062  
1063  
1064  
1065  
1066  
1067  
1068  
1069  
1069  
1070  
1071  
1072  
1073  
1074  
1075  
1076  
1077  
1078  
1079  
1079  
1080  
1081  
1082  
1083  
1084  
1085  
1086  
1087  
1088  
1089  
1089  
1090  
1091  
1092  
1093  
1094  
1095  
1096  
1097  
1098  
1099  
1099  
1100  
1101  
1102  
1103  
1104  
1105  
1106  
1107  
1108  
1109  
1109  
1110  
1111  
1112  
1113  
1114  
1115  
1116  
1117  
1118  
1119  
1119  
1120  
1121  
1122  
1123  
1124  
1125  
1126  
1127  
1128  
1129  
1129  
1130  
1131  
1132  
1133  
1134  
1135  
1136  
1137  
1138  
1139  
1139  
1140  
1141  
1142  
1143  
1144  
1145  
1146  
1147  
1148  
1149  
1149  
1150  
1151  
1152  
1153  
1154  
1155  
1156  
1157  
1158  
1159  
1159  
1160  
1161  
1162  
1163  
1164  
1165  
1166  
1167  
1168  
1169  
1169  
1170  
1171  
1172  
1173  
1174  
1175  
1176  
1177  
1178  
1179  
1179  
1180  
1181  
1182  
1183  
1184  
1185  
1186  
1187  
1188  
1189  
1189  
1190  
1191  
1192  
1193  
1194  
1195  
1196  
1197  
1198  
1199  
1199  
1200  
1201  
1202  
1203  
1204  
1205  
1206  
1207  
1208  
1209  
1209  
1210  
1211  
1212  
1213  
1214  
1215  
1216  
1217  
1218  
1219  
1219  
1220  
1221  
1222  
1223  
1224  
1225  
1226  
1227  
1228  
1229  
1229  
1230  
1231  
1232  
1233  
1234  
1235  
1236  
1237  
1238  
1239  
1239  
1240  
1241  
1242  
1243  
1244  
1245  
1246  
1247  
1248  
1249  
1249  
1250  
1251  
1252  
1253  
1254  
1255  
1256  
1257  
1258  
1259  
1259  
1260  
1261  
1262  
1263  
1264  
1265  
1266  
1267  
1268  
1269  
1269  
1270  
1271  
1272  
1273  
1274  
1275  
1276  
1277  
1278  
1279  
1279  
1280  
1281  
1282  
1283  
1284  
1285  
1286  
1287  
1288  
1289  
1289  
1290  
1291  
1292  
1293  
1294  
1295  
1296  
1297  
1298  
1299  
1299  
1300  
1301  
1302  
1303  
1304  
1305  
1306  
1307  
1308  
1309  
1309  
1310  
1311  
1312  
1313  
1314  
1315  
1316  
1317  
1318  
1319  
1319  
1320  
1321  
1322  
1323  
1324  
1325  
1326  
1327  
1328  
1329  
1329  
1330  
1331  
1332  
1333  
1334  
1335  
1336  
1337  
1338  
1339  
1339  
1340  
1341  
1342  
1343  
1344  
1345  
1346  
1347  
1348  
1349  
1349  
1350  
1351  
1352  
1353  
1354  
1355  
1356  
1357  
1358  
1359  
1359  
1360  
1361  
1362  
1363  
1364  
1365  
1366  
1367  
1368  
1369  
1369  
1370  
1371  
1372  
1373  
1374  
1375  
1376  
1377  
1378  
1379  
1379  
1380  
1381  
1382  
1383  
1384  
1385  
1386  
1387  
1388  
1389  
1389  
1390  
1391  
1392  
1393  
1394  
1395  
1396  
1397  
1398  
1399  
1399  
1400  
1401  
1402  
1403  
1404  
1405  
1406  
1407  
1408  
1409  
1409  
1410  
1411  
1412  
1413  
1414  
1415  
1416  
1417  
1418  
1419  
1419  
1420  
1421  
1422  
1423  
1424  
1425  
1426  
1427  
1428  
1429  
1429  
1430  
1431  
1432  
1433  
1434  
1435  
1436  
1437  
1438  
1439  
1439  
1440  
1441  
1442  
1443  
1444  
1445  
1446  
1447  
1448  
1449  
1449  
1450  
1451  
1452  
1453  
1454  
1455  
1456  
1457  
1458  
1459  
1459  
1460  
1461  
1462  
1463  
1464  
1465  
1466  
1467  
1468  
1469  
1469  
1470  
1471  
1472  
1473  
1474  
1475  
1476  
1477  
1478  
1479  
1479  
1480  
1481  
1482  
1483  
1484  
1485  
1486  
1487  
1488  
1489  
1489  
1490  
1491  
1492  
1493  
1494  
1495  
1496  
1497  
1498  
1499  
1499  
1500  
1501  
1502  
1503  
1504  
1505  
1506  
1507  
1508  
1509  
1509  
1510  
1511  
1512  
1513  
1514  
1515  
1516  
1517  
1518  
1519  
1519  
1520  
1521  
1522  
1523  
1524  
1525  
1526  
1527  
1528  
1529  
1529  
1530  
1531  
1532  
1533  
1534  
1535  
1536  
1537  
1538  
1539  
1539  
1540  
1541  
1542  
1543  
1544  
1545  
1546  
1547  
1548  
1549  
1549  
1550  
1551  
1552  
1553  
1554  
1555  
1556  
1557  
1558  
1559  
1559  
1560  
1561  
1562  
1563  
1564  
1565  
1566  
1567  
1568  
1569  
1569  
1570  
1571  
1572  
1573  
1574  
1575  
1576  
1577  
1578  
1579  
1579  
1580  
1581  
1582  
1583  
1584  
1585  
1586  
1587  
1588  
1589  
1589  
1590  
1591  
1592  
1593  
1594  
1595  
1596  
1597  
1598  
1599  
1599  
1600  
1601  
1602  
1603  
1604  
1605  
1606  
1607  
1608  
1609  
1609  
1610  
1611  
1612  
1613  
1614  
1615  
1616  
1617  
1618  
1619  
1619  
1620  
1621  
1622  
1623  
1624  
1625  
1626  
1627  
1628  
1629  
1629  
1630  
1631  
1632  
1633  
1634  
1635  
1636  
1637  
1638  
1639  
1639  
1640  
1641  
1642  
1643  
1644  
1645  
1646  
1647  
1648  
1649  
1649  
1650  
1651  
1652  
1653  
1654  
1655  
1656  
1657  
1658  
1659  
1659  
1660  
1661  
1662  
1663  
1664  
1665  
1666  
1667  
1668  
1669  
1669  
1670  
1671  
1672  
1673  
1674  
1675  
1676  
1677  
1678  
1679  
1679  
1680  
1681  
1682  
1683  
1684  
1685  
1686  
1687  
1688  
1689  
1689  
1690  
1691  
1692  
1693  
1694  
1695  
1696  
1697  
1698  
1699  
1699  
1700  
1701  
1702  
1703  
1704  
1705  
1706  
1707  
1708  
1709  
1709  
1710  
1711  
1712  
1713  
1714  
1715  
1716  
1717  
1718  
1719  
1719  
1720  
1721  
1722  
1723  
1724  
1725  
1726  
1727  
1728  
1729  
1729  
1730  
1731  
1732  
1733  
1734  
1735  
1736  
1737  
1738  
1739  
1739  
1740  
1741  
1742  
1743  
1744  
1745  
1746  
1747  
1748  
1749  
1749  
1750  
1751  
1752  
1753  
1754  
1755  
1756  
1757  
1758  
1759  
1759  
1760  
1761  
1762  
1763  
1764  
1765  
1766  
1767  
1768  
1769  
1769  
1770  
1771  
1772  
1773  
1774  
1775  
1776  
1777  
1778  
1779  
1779  
1780  
1781  
1782  
1783  
1784  
1785  
1786  
1787  
1788  
1789  
1789  
1790  
1791  
1792  
1793  
1794  
1795  
1796  
1797  
1798  
1799  
1799  
1800  
1801  
1802  
1803  
1804  
1805  
1806  
1807  
1808  
1809  
1809  
1810  
1811  
1812  
1813  
1814  
1815  
1816  
1817  
1818  
1819  
1819  
1820  
1821  
1822  
1823  
1824  
1825  
1826  
1827  
1828  
1829  
1829  
1830  
1831  
1832  
1833  
1834  
1835  
1836  
1837  
1838  
1839  
1839  
1840  
1841  
1842  
1843  
1844  
1845  
1846  
1847  
1848  
1849  
1849  
1850  
1851  
1852  
1853  
1854  
1855  
1856  
1857  
1858  
1859  
1859  
1860  
1861  
1862  
1863  
1864  
1865  
1866  
1867  
1868  
1869  
1869  
1870  
1871  
1872  
1873  
1874  
1875  
1876  
1877  
1878  
1879  
1879  
1880  
1881  
1882  
1883  
1884  
1885  
1886  
1887  
1888  
1889  
1889  
1890  
1891  
1892  
1893  
1894  
1895  
1896  
1897  
1898  
1899  
1899  
1900  
1901  
1902  
1903  
1904  
1905  
1906  
1907  
1908  
1909  
1909  
1910  
1911  
1912  
1913  
1914  
1915  
1916  
1917  
1918  
1919  
1919  
1920  
1
```

```
> class IntComp implements Comparator<Integer> {  
>     public int compare(Integer i, Integer j) { return i - j; } }  
> inted class IntComp
```

Handwritten note: Foo Bar

list.iterator() will return an iterator object which is an interface

```
jshell> list  
list ==> [Circle with radius 100, Rectangle 2 x 3]  
  
jshell> for (Shape s : list) { System.out.println(s.getArea()); }  
31415.926535897932  
6.0  
  
jshell> list.iterator()  
$60 ==> java.util.ArrayList$Iterator@368239c8  
  
jshell> Iterator<Shape> iter = list.iterator()  
iter ==> java.util.ArrayList$Iterator@3b192d32  
  
jshell> iter.hasNext()  
$62 ==> true
```

Assign it to a new Iterator object

```
jshell> Iterator<Shape> iter = list.iterator()  
iter ==> java.util.ArrayList$Iterator@707f7052  
  
jshell> while (iter.hasNext()) { System.out.println(iter.next().getArea()); }  
31415.926535897932  
^
```

Stuff to look out for:

include a snd method to enable a terminal (e.g. a pager) to initiate the handshake with a host (e.g. a transmitter).

Terminal here is a parent class, so the snd method should not belong to pager

Prevent cyclic dependencies -> create new classes

Level 5 needs to track ID or sth from the v start -> takes in ImList of the objects that needs to be tracked or maybe ImList of String

Creating new classes for:

Polymorphism + Unique methods encapsulated within the children class helps create specificity whereby not all Hosts can carry out .rcv(), only TransSND can!!

Snd method takes in a host but not all hosts are allowed

if the host is undergoing a process and is incomplete, you get incompatible type, hence you can only take in completed hosts or new hosts

-> this means that snd takes in a children class of host

1. Ack extends Completed Host
2. Transmitter extends Completed Host

Tips to print out separate line by line:

public void method

Inside method System.out.println()

Cyclic Dependencies:

TransSND extends Host

**TransSND -> PagerRCV -> Host == no cyclic dependency**

PagerRCV(Term t, **Host h**) vs PagerRCV(Term t, **TransSND h**)

TransSND -> PagerRCV -> TransSND == cyclic dependency

RcvTerm takes in a Host, meaning that it can only use the behaviors that Host provide, that's why the dependency of RcvTerm is only on Host, and not SndHost

Try not to let the **abstract** class handle all the logic, create another class

Terminal -> TermlImp extends Terminal -> Pager extends TermlImp

Overloading constructors:

1. useful when you have methods that return a new Object of the same type
2. and when you need to track history aka passing in ImList as an input
3. the inputs are predefined in the example shown
4. to take in a flagger

```

private final boolean tracker;

Booking(Driver driver, Request request) {
    this.driver = driver;
    this.request = request;
    this.tracker = false;
}

Booking(Driver driver, Request request, boolean tracker) {
    this.driver = driver;
    this.request = request;
    this.tracker = tracker;
}

Driver getDriver() {

Case(Person person, PCR pcr) {
    this(person, new ImmutableList<>(List.of(pcr)), getValidTestProtocol(person, pcr), 0, new ImmutableList<>());
}

Case(Person person, ImmutableList<Test> testHistory, Protocol currentProtocol, int numOfDays, ImmutableList<Case> priorLineage) {
    this.person = person;
    this.testHistory = testHistory;
    this.currentProtocol = currentProtocol;
    this.numOfDays = numOfDays;
    this.priorLineage = priorLineage;
}
}

```

If the name of the class is alr at base form, e.g. Case and there's another type of case CloseCase, Case should just be the parent alr

Rec 5 generics:

Generic Invariance => The type parameters must match exactly, to protect against heap pollution.

In java, GenericType<Cat> and GenericType<Animal> have **absolutely nothing to do** with each other due to invariance. To force generic types in Java to be assignable to each other based on the generic parameter, we **need to use the wildcard**.

For this to happen in Java, we need to use the contravariance generic operator: <? super Cat>

If the Generic method(? extends Fruits) returns Fruits type(defined at the start aka method binded by Fruits type), even though you can sub in orange into the method, **the return type will still be binded as Fruits**.

**Hence, you cannot Orange o = method(orange).**

**Must change the Generic method to Comparable<? Super T>, then the above assignment is possible.**

Rec 6:

Function's input: E and smaller(within E) aka parents as well, ? super E

Function's output: must be at least R, ? extends R in order for it to be as general as it can

You do not want to have nested contexts, **aka do not do what's below**

Functon<String,ImList<Integer>>

F = x -> new ImList<Integer>(). add(x.length())

=> each input will be stored into a list/context

=> use flatMap() instead

- flatMap transforms each stream element into a stream of other elements (either zero or more) by taking in a function that produces another stream, and then *flattens* it

```
jshell> IntStream.of(1,2,3).
    ...> flatMap(x -> IntStream.rangeClosed(x,3).map(y -> x * y)).
    ...> forEach(x -> System.out.print(x + " "))
1 2 3 4 6 9
```

---

Function within flatmap takes in an int argument and returns an IntStream

Flatmap REPLACES the elements of the stream with ONLY the CONTENTS of the mapped stream

Function within flatmap is applied to the original IntStream Object (.of(1,2,3))

1, 2, 3 goes into inner IntStream as x

(1,3) -> map() -> apply function x\*y on every element 1 to 3, where y rep each element of the inner IntStream -> 1 \* 1, 1 \* 2, 1\* 3

(2,3) 2 \* 2, 2\* 3

(3,3) 3 \* 3

Map simply applies the map function on the elements of the stream

```
9  
jshell> IntStream.rangeClosed(1,10).map(x -> x + "")  
| Error:  
| incompatible types: bad return type in lambda expression  
|         java.lang.String cannot be converted to int  
| IntStream.rangeClosed(1,10).map(x -> x + "")  
|                                     ^----^
```

```
*--> java.util.stream.IntPipeline$3@4629104a  
jshell> IntStream.rangeClosed(1,10).boxed().map(x -> x + "")  
$4 ==> java.util.stream.ReferencePipeline$3@4629104a  
jshell> |
```

This works because you are boxing the primitive int objects up into wrapper objects and each of these has a `toString()` method, allowing you to concatenate