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.

<u>Generics</u>
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
```



```
jshell> Func f = x -> 2 * x;
f ==> $Lambda$20/0x0000000800c0a000@52cc8049
```

```
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(combing 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

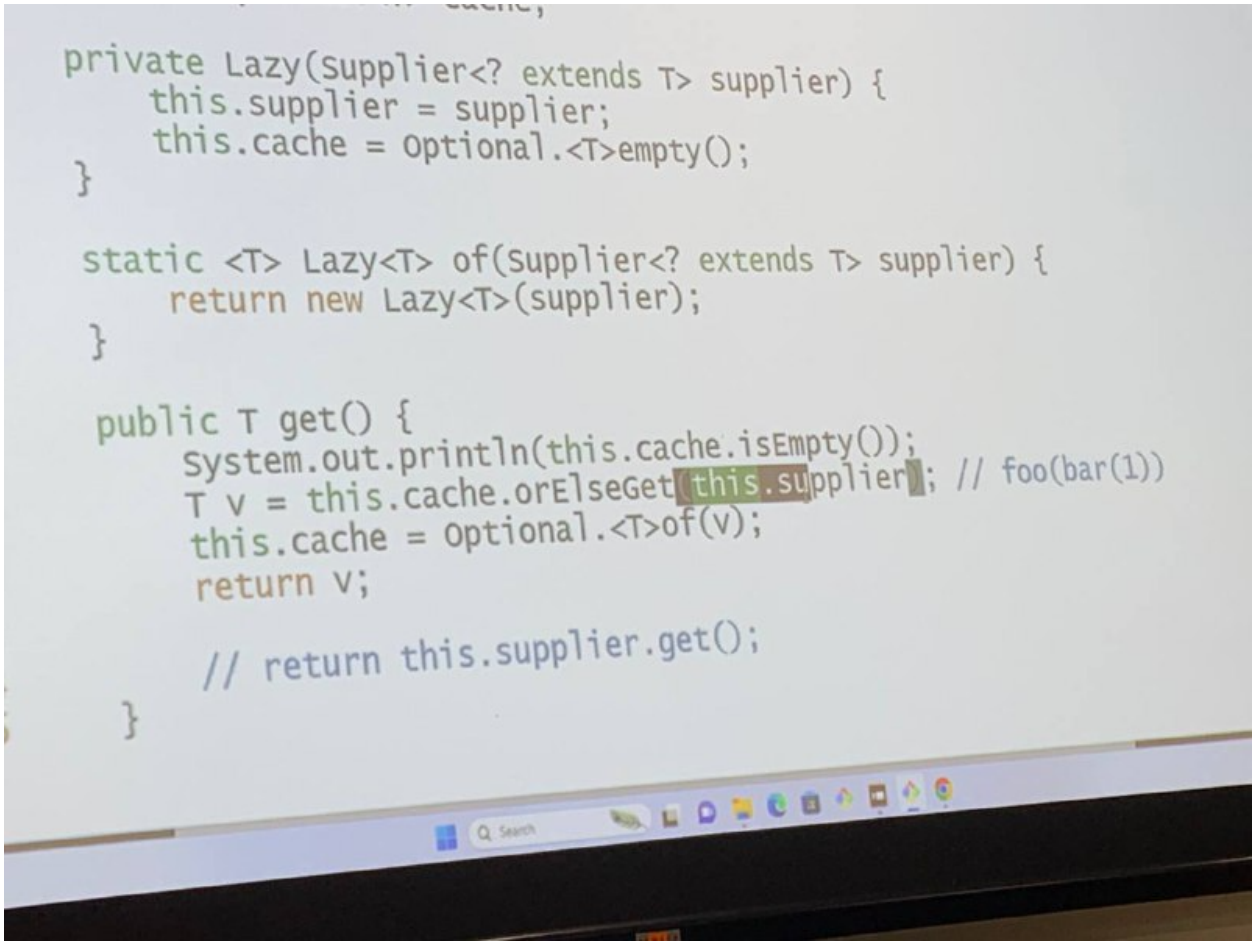
CompletableFuture.<A>completedFuture() is often used for **base cases, if else loops**

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));
```



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.

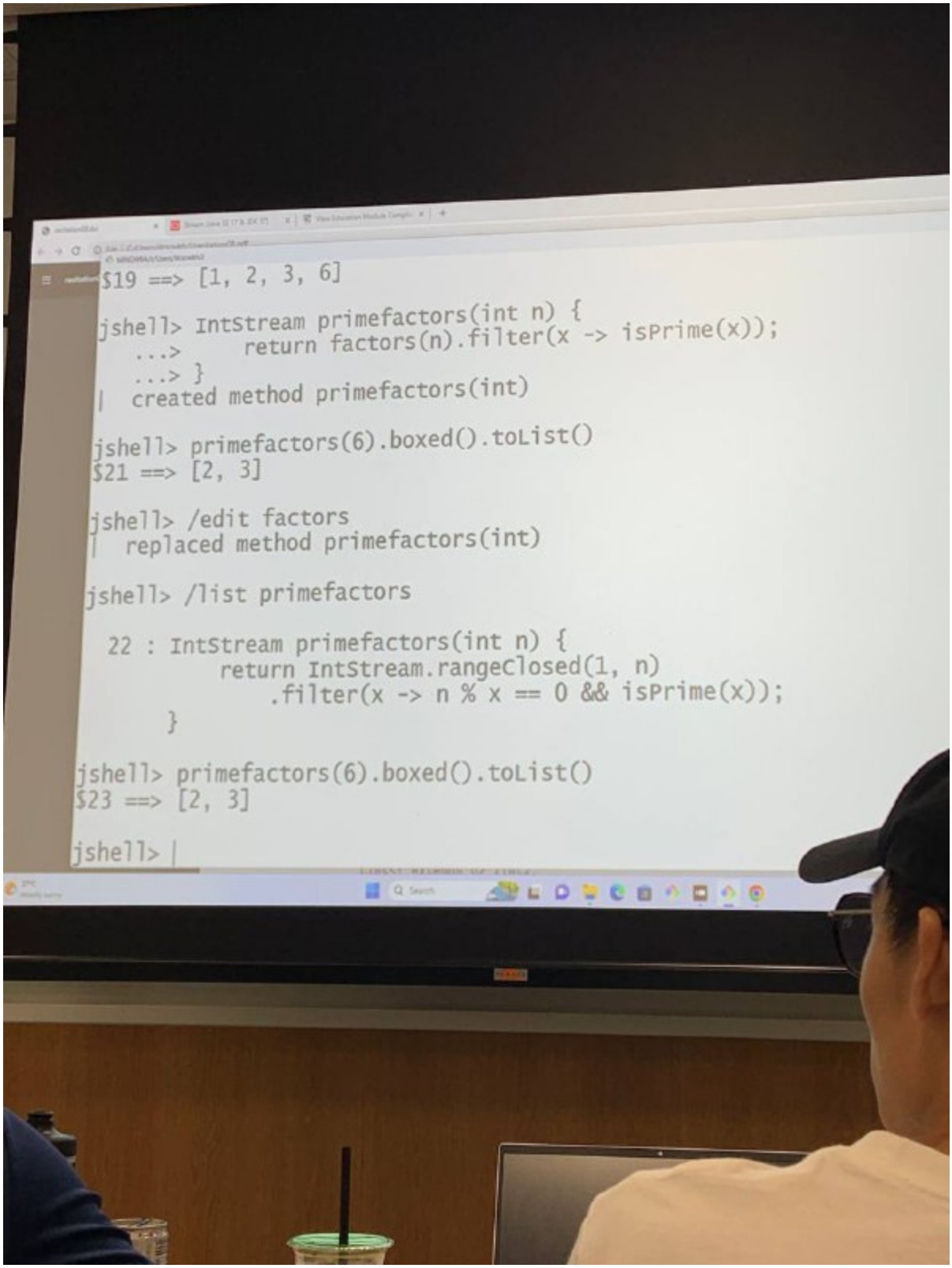
Return supplier: This method returns the **value** of this Optional instance, if present. If there is no value present in this Optional instance, then this method throws the exception generated from the specified supplier.

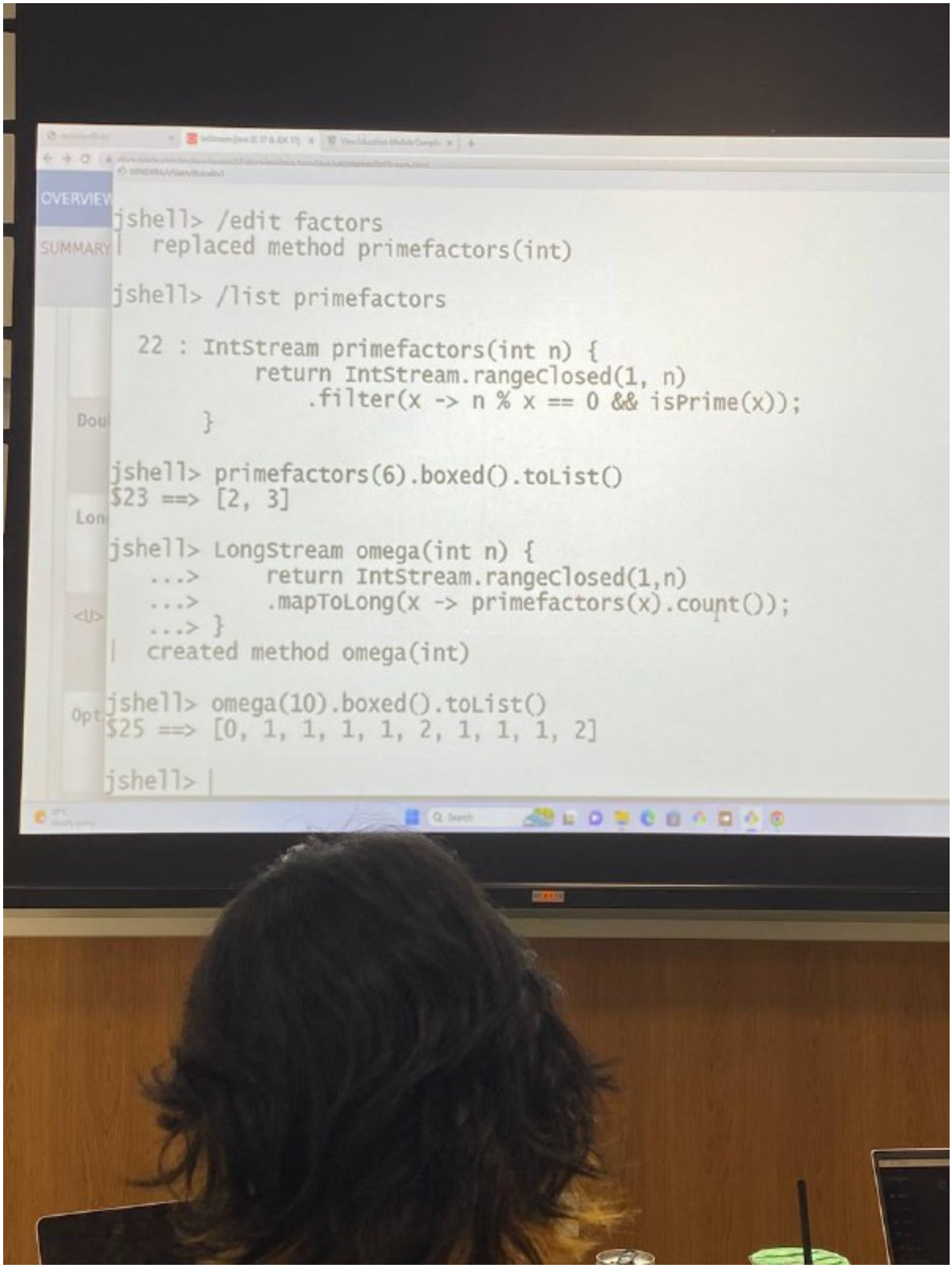
```
jshell> mi.filter(x -> x % 2 == 0).orElseGet(() -> 2)
$.. ==> 2

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

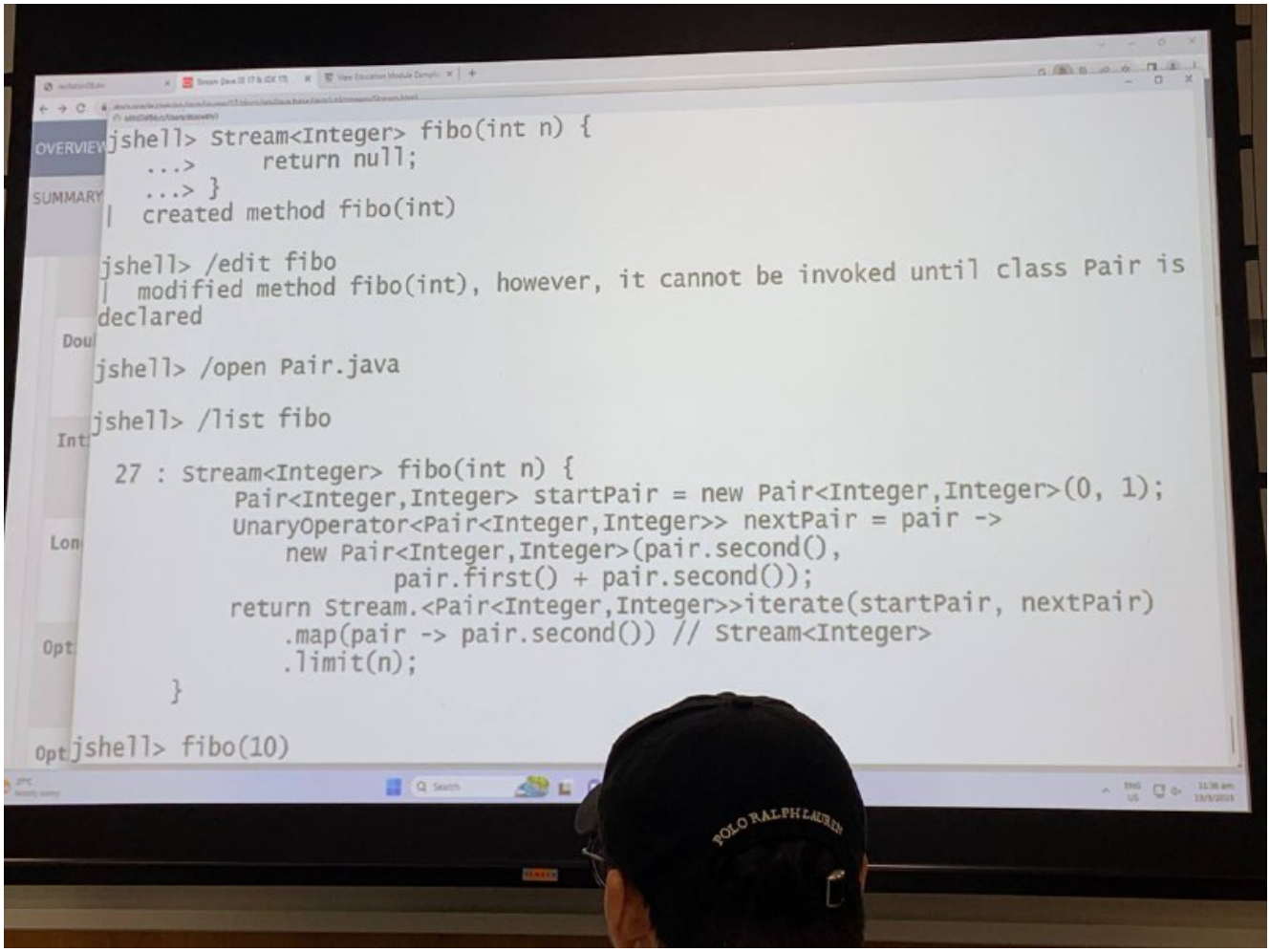
```
jshell> Supplier<Optional<Circle>> supp = () -> {
   ...>     System.out.println("beep!");
   ...>     return Optional.<Circle>of(new Circle(new Point(0, 0), 1.0));}
supp ==> $Lambda$23/0x0000000800c13858@15615099
```

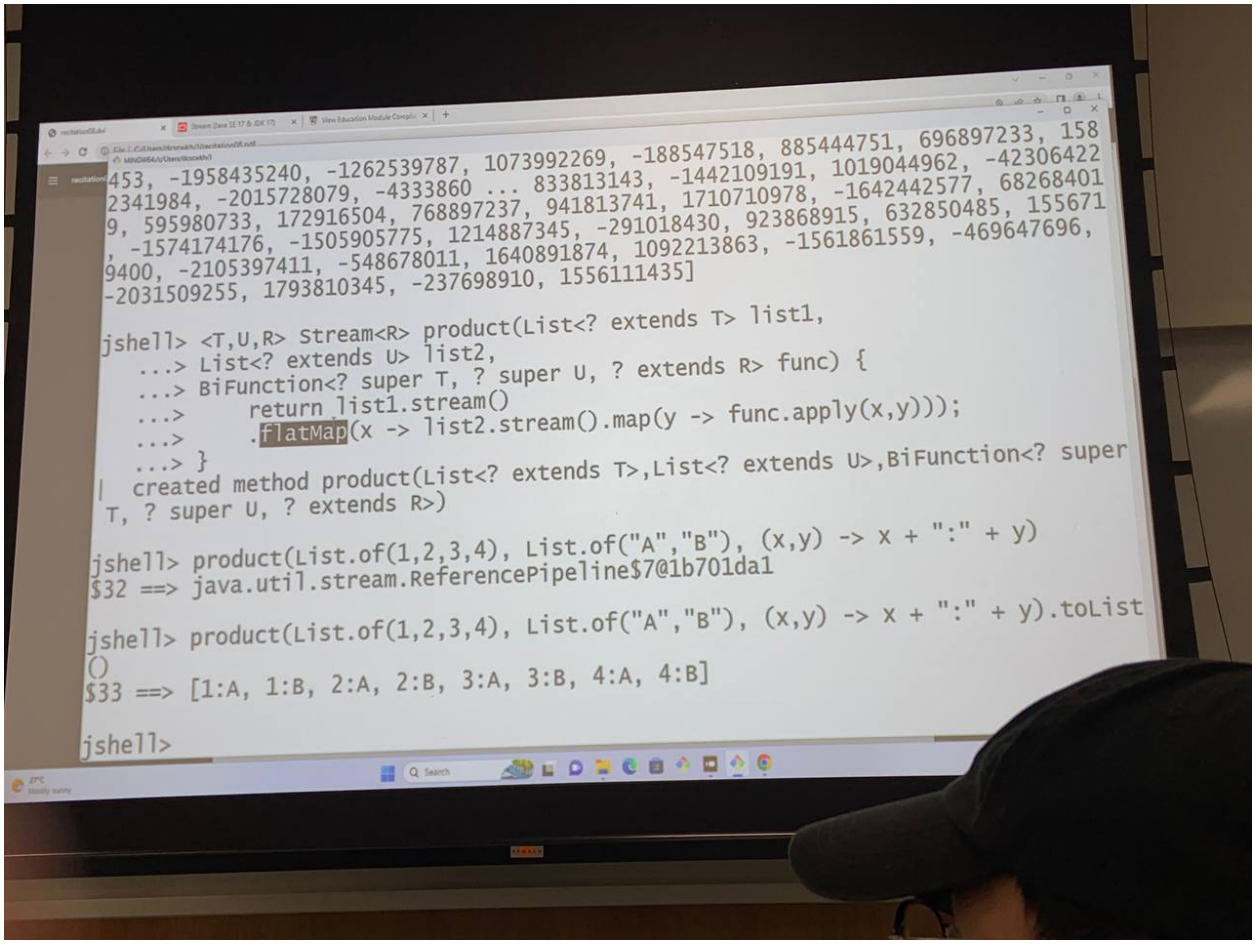
```
$.. ==> 1

jshell> Supplier<Maybe<Integer>> suppMaybe = () -> Maybe.<Integer>of(2)
suppMaybe ==> $Lambda$24/0x0000000800adc40@5c7fa833

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

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

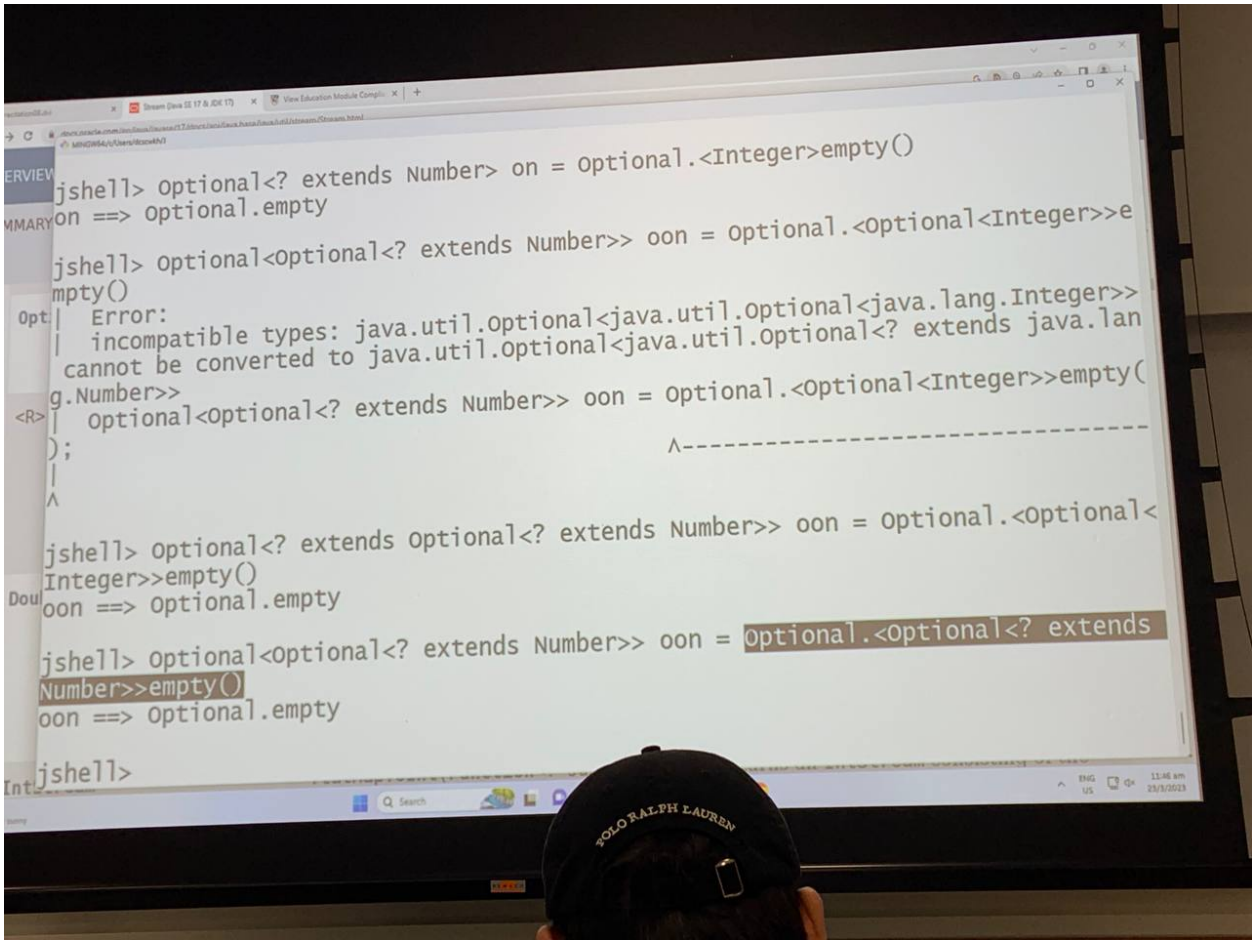
```
Optional.ofNullable(item).filter(i -> i.getCrystal() == Crystal.A)
    .ifPresent(k -> player.getInventory.addItem(i));
```


$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>


OVERVIEW
SUMMARY
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> LongStream omega(int n) {
...> return IntStream.rangeClosed(1,n)
...> .mapToLong(x -> primefactors(x).count());
...> }
| created method omega(int)
jshell> omega(10).boxed().toList()
$25 ==> [0, 1, 1, 1, 1, 2, 1, 1, 1, 2]
jshell>


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
jshell> /open Pair.java
jshell> /list fibo
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);
}
jshell> fibo(10)
POLO RALPH LAUREN


453, -1958435240, -1262539787, 1073992269, -188547518, 885444751, 696897233, 158
2341984, -2015728079, -4333860 ... 833813143, -1442109191, 1019044962, -42306422
9, 595980733, 172916504, 768897237, 941813741, 1710710978, -1642442577, 68268401
, -1574174176, -1505905775, 1214887345, -291018430, 923868915, 632850485, 155671
9400, -2105397411, -548678011, 1640891874, 1092213863, -1561861559, -469647696,
-2031509255, 1793810345, -237698910, 1556111435]
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 =
}
```

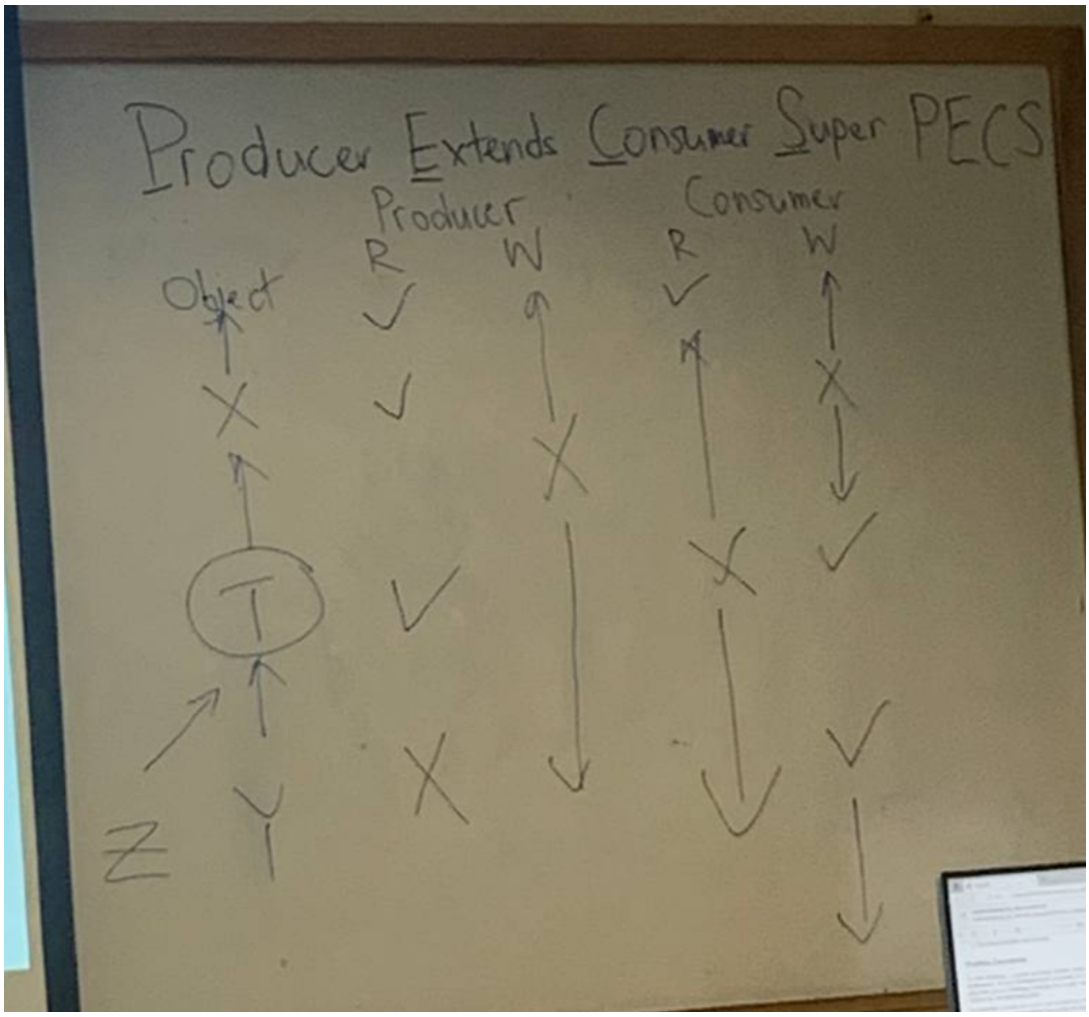
.get(name) here returns an Optional<Student>
By adding the .orElse(), it means that if get(name) != Null, it will return the **VALUE** of the Optional<Student>, which is the Student object, else return the other value

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>

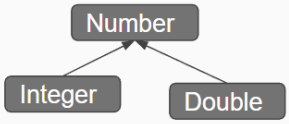
Use <T> if you want to read from and add to a collection



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.



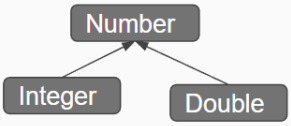
Reading from `foo3` (must be legal for **all** possible `foo3` assignments):

- `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.



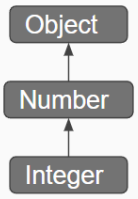
Writing to `foo3` (must be legal for **all** possible `foo3` assignments):

- 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>

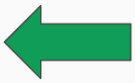
Function<T,R>
What do T and R correspond to?

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)

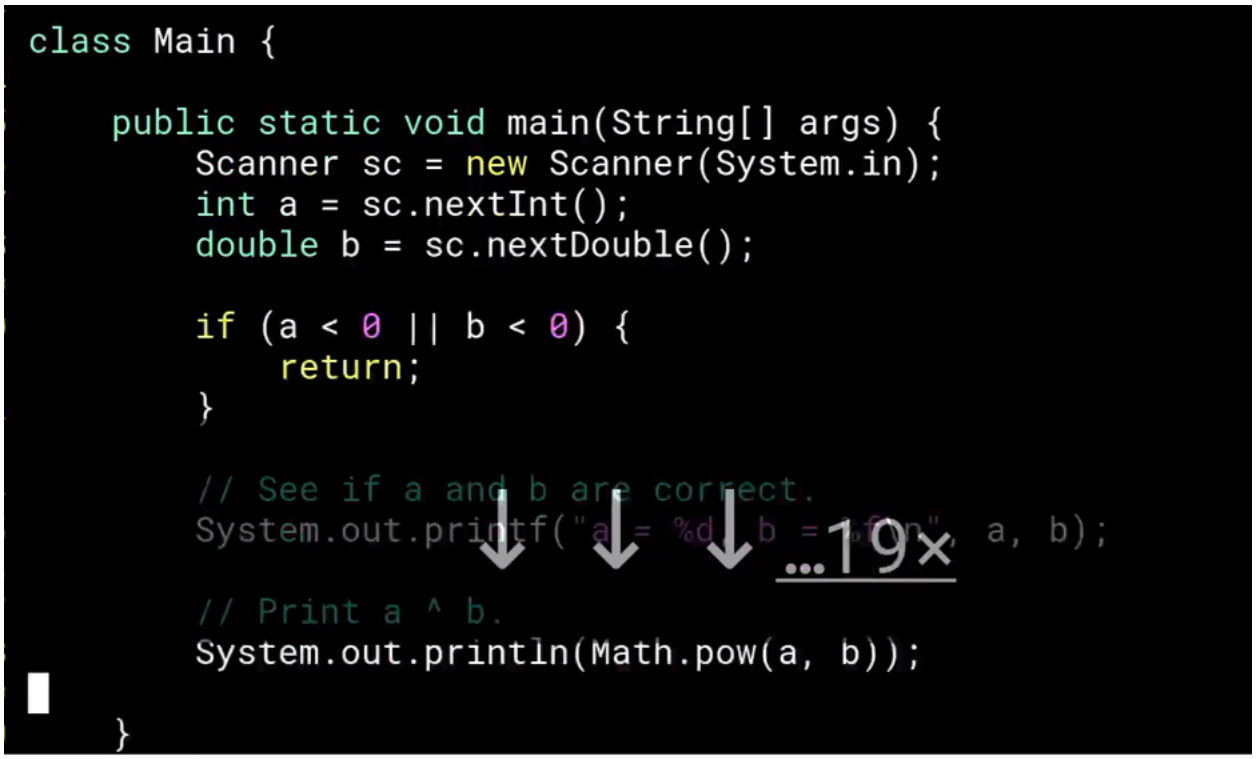
v shift v and control v
gg (go to the first line of file)
G (go to the last line of file)
gg=G fix indentation
W or b to jump words forward/backward

/(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**



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
```

```
    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

```java
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;
}
```

```java
        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;
    }
```

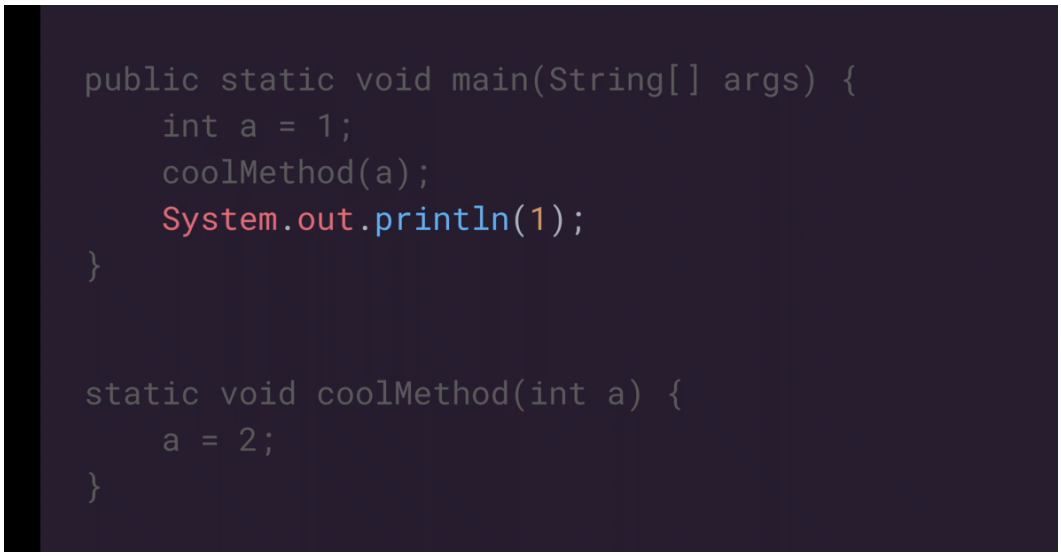
Typecast/Convert the result of the square root into integer

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.

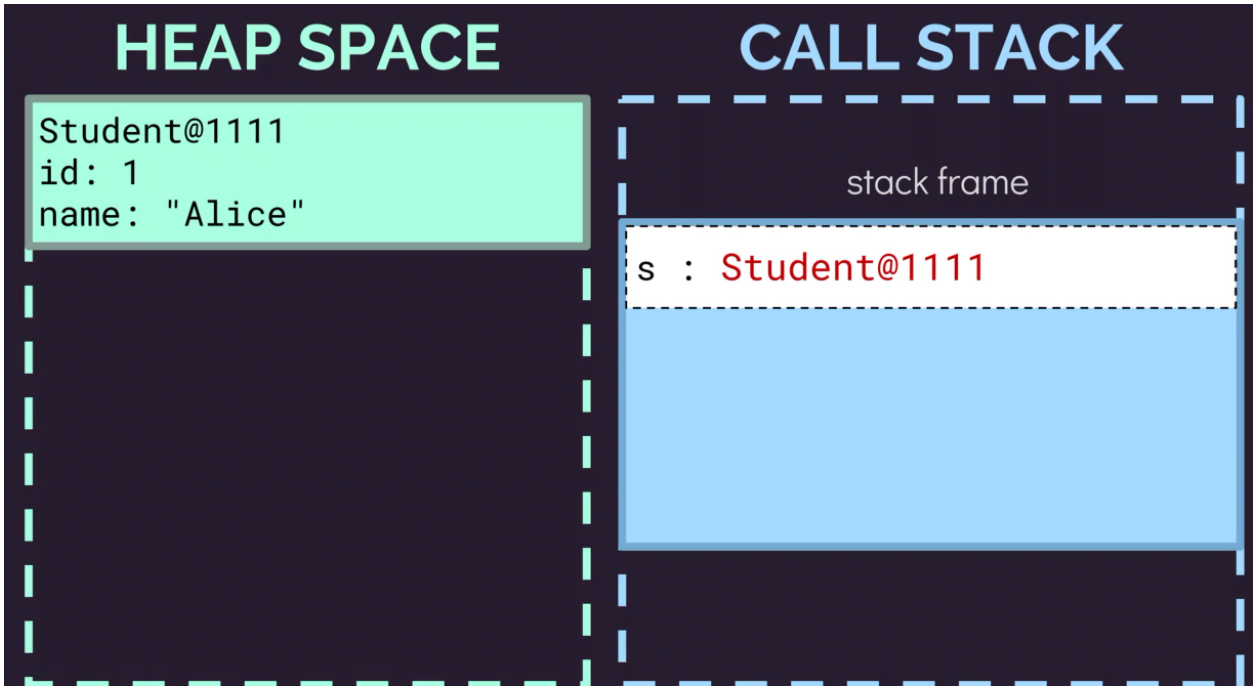
Static method can be called without creating an object of the class

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

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)**

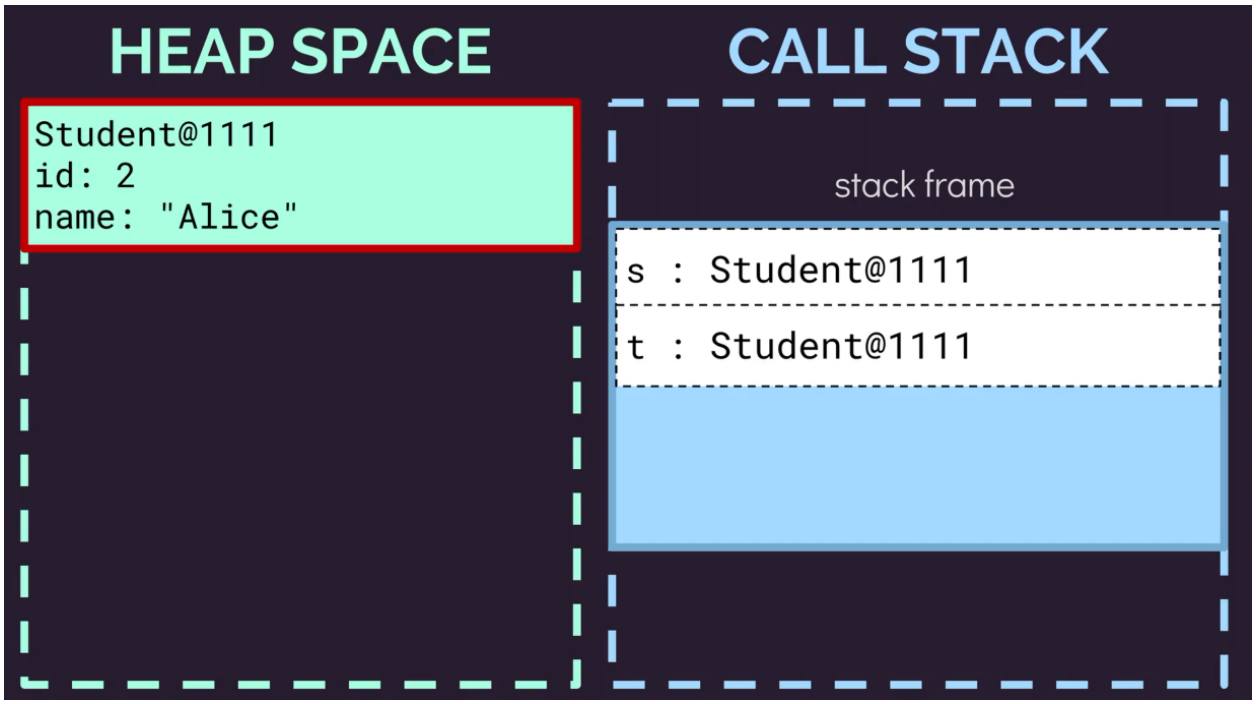


When coolMethod is called here, it changes the value of a to 2 temporarily and when the function is done, **it drops off the stack and the value of a is still 1**



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

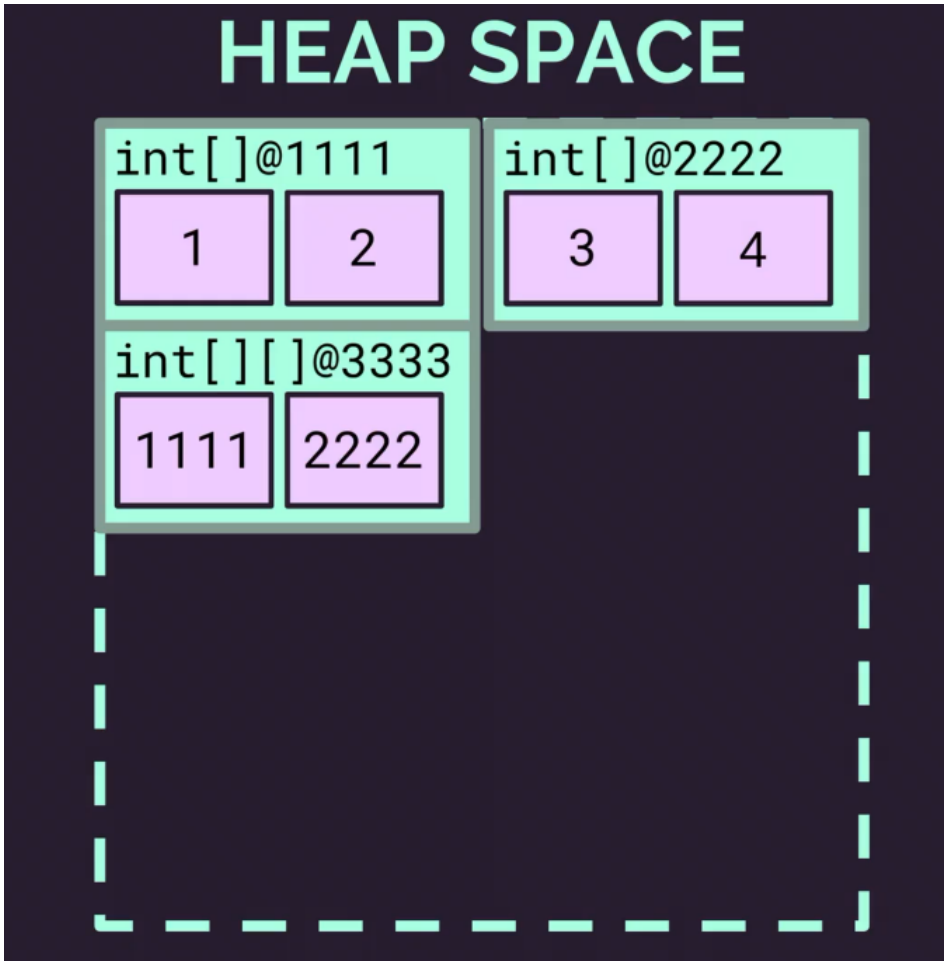


Arrays:

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

```java
interface Shape {
    /* abstract */ public double getArea();
}
```

Shape.java

```java
        this.height = height;
    }

    @Override
    public double getArea() {
        return width * height;
    }
```

Rectangle.java

```java
    }

    @Override
    public double getArea() {
        return Math.PI * radius * radius;
    }
```

Circle.java

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

```java
interface Shape {
    abstract double getArea();
}
```

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

<u>Return type cannot be more general, only more specific</u>

Scalable.java

```java
class Rectangle implements Shape, Scalable {
    private final int width;
```

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

```
1 interface Scalable {
2     Scalable scale(int factor);
3 }
```

```
Scalable.java                                    2,6           All
13      }
14
15      @Override
16      public Rectangle scale(int factor) {
17          return new Rectangle(this.width * factor, this.height * factor);
18      }
19
Rectangle.java                                   15,1          70%
```

List.of returns an immutable list, but it's useful if you are adding many elements, aka replaces multiple .add methods

```
ell> new ImList<Integer>(List.of(1,2,3,5,6)).add(4)
3 ==> [1, 2, 3, 5, 6, 4]
```

superinterface

25

Here's an example:

```
public interface A {
    void doSomething();
}

public interface B extends A {
    void doSomethingElse();
}
```

In this example, `A` is a *superinterface* of `B`. It's like being a superclass. Any class implementing `B` now also implements `A` automatically, and must provide an implementation of both `doSomething()` and `doSomethingElse()`.

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

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()
$60 ==> java.util.ArrayList$Itr@368239c8

jshell> Iterator<Shape> iter = list.iterator()
iter ==> java.util.ArrayList$Itr@3b192d32

jshell> iter.hasNext()
$62 ==> true
```

Assign it to a new Iterator object

```
jshell> Iterator<Shape> iter = list.iterator()
iter ==> java.util.ArrayList$Itr@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 -> TermImp extends Terminal -> Pager extends TermImp

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 ImList<>(List.of(pcr)), getValidTestProtocol(person, pcr), 0, new ImList<>());
}

Case(Person person, ImList<Test> testHistory, Protocol currentProtocol, int numOfDays, ImList<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.

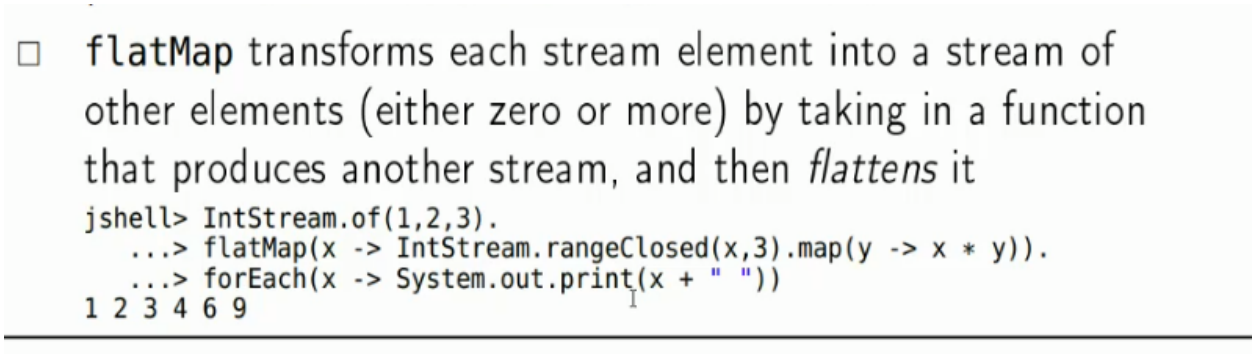
In java, GenericType<Cat> and GenericType<Animal> have **absolutely nothing to do** with each other due to invariance. To force generic types in Java to be assignable to each other based on the generic parameter, we **need to use the wildcard**.
For this to happen in Java, we need to use the contravariance 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**
Funcion<String,ImList<Integer>>
F = x -> new ImList<Integer>(). add(x.length())
=> each input will be stored into a list/context
=> use flatMap() instead



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 + "")
|                                       ^----^
```

```
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