

CS2030 Lecture 7

The *Maybe* Context

Henry Chia (hchia@comp.nus.edu.sg)

Semester 2 2022 / 2023

null and NullPointerException

```
Circle createUnitCircle(Point p, Point q) {
    double d = p.distanceTo(q);
    if (d < EPSILON || d > 2.0 + EPSILON) {
        return null; // null is a Circle?
    } else {
        Point m = p.midPoint(q);
        double mp = Math.sqrt(1.0 - Math.pow(p.distanceTo(m), 2.0));
        double theta = p.angleTo(q);
        m = m.moveTo(theta + Math.PI / 2.0, mp);
        return new Circle(m, 1.0);
    }
}

jshell> Point p = new Point(0.5, 0.5)
p ==> (0.500, 0.500)
jshell> createUnitCircle(new Point(0, 0), new Point(0, 0)).contains(p)
Exception java.lang.NullPointerException: Cannot invoke
"REPL.$JShell$13$Circle.contains(REPL.$JShell$11$Point)"
because the return value of
"REPL.$JShell$14.createUnitCircle(REPL.$JShell$11$Point, REPL.$JShell$11$Point)" is null
at (#5:1)
```

Circle $\xRightarrow{\text{contains}(p)}$ boolean

1 / 16

3 / 16

Outline and Learning Outcome

- Understand that **null** values are meaningless and may lead to `NullPointerException`s and why they should be avoided
- Know how to use Java's `Optional` class to handle **null** values
- Understand how *higher order functions* can be used to support **cross-barrier manipulation**
- Be able to define anonymous inner classes and lambda expressions
 - Appreciate that functions are first-class citizens
- Understand the concept of a *computation context*
- Able to define implementations of Java functional interfaces
- Appreciate `map` versus `flatMap`

2 / 16

My Billion Dollar Mistake...

"I call it my billion-dollar mistake. It was the invention of the null reference in 1965. I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement."

– Sir Charles Antony Richard Hoare
aka Tony Hoare

His friend, Edsger Dijkstra's response:

"If you have a null reference, then every bachelor who you represent in your object structure will seem to be married polyamorously to the same person Null"

4 / 16

A Maybe Context: Java Optional

- A *context* with connotations of *maybe* that “wraps” around another object of type T, i.e. maybe a T or maybe empty
- **Optional** static methods: **of** and **empty**

```
jshell> Optional.of("cs2030") // type-inference to Optional<String>
$.. ==> Optional[cs2030]

jshell> Optional.<String>of("cs2030") // type-witness to Optional<String>
$.. ==> Optional[cs2030]

jshell> Optional.<Object>of("cs2030") // type-witness to Optional<Object>
$.. ==> Optional[cs2030]

jshell> Optional.<String>empty() // Optional<String>
$.. ==> Optional.empty

jshell> Optional.empty() // type-inference to Optional<Object>
$.. ==> Optional.empty

jshell> Optional.<String>ofNullable(null) // how about of(null)?
$.. ==> Optional.empty
```

Returning a Context

- Redefine `createUnitCircle` to return `Optional<Circle>`

```
Optional<Circle> createUnitCircle(Point p, Point q) {
    double d = p.distanceTo(q);
    if (d < EPSILON || d > 2.0 + EPSILON) {
        return Optional.<Circle>empty();
    } else {
        Point m = p.midPoint(q);
        double mp = Math.sqrt(1.0 - Math.pow(p.distanceTo(m), 2.0));
        double theta = p.angleTo(q);
        m = m.moveTo(theta + Math.PI / 2.0, mp);
        return Optional.<Circle>of(new Circle(m, 1.0));
    }
}
```

```
jshell> createUnitCircle(new Point(0, 0), new Point(1, 1))
$.. ==> Optional[Circle at (0.000, 1.000) with radius 1.0]

jshell> createUnitCircle(new Point(0, 0), new Point(10, 10))
$.. ==> Optional.empty

jshell> createUnitCircle(new Point(0, 0), new Point(0, 0))
$.. ==> Optional.empty
```

Chaining Methods to a Context

- Chaining with a `contains` method gives a compilation error:

```
jshell> createUnitCircle(new Point(0, 0), new Point(1, 1)).contains(new Point(0, 1))
Error:
cannot find symbol
symbol: method contains(Point)
createUnitCircle(new Point(0, 0), new Point(1, 1)).contains(new Point(0, 1))
^
```

- Need to pass the `contains` method into `Optional` via a **higher-order function**
 - a function that takes in another function
- A function is a *first-class citizen*, i.e. just like any value/object
 - assign a function to a variable
 - pass a function as an argument to another method
 - return a function from another method

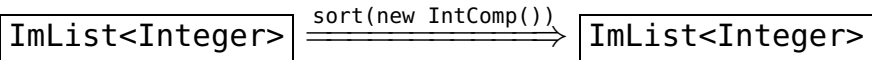
Cross-Barrier Manipulation

- **Cross-barrier manipulation** — where the client defines a function that is passed to the context for execution
- Recap: the `sort` method in `ImList<E>` that takes in a `Comparator<? super E>`

```
jshell> ImList<Integer> list123 = ImList.<Integer>of().add(1).add(2).add(3)
list123 ==> [1, 2, 3]

jshell> class IntComp implements Comparator<Integer> {
...>     @Override
...>     public int compare(Integer x, Integer y) {
...>         return y - x;
...>     }
...> }
| created class IntComp

jshell> list123.sort(new IntComp())
$.. ==> [3, 2, 1]
```



Anonymous Inner Class

- Define an *anonymous inner class* instead of a concrete class

```
jshell> Comparator<Integer> comp = new Comparator<Integer>() {  
...>     @Override  
...>     public int compare(Integer x, Integer y) {  
...>         return y - x;  
...>     }  
...> }  
comp ==> 1@20e2cbe0  
jshell> list123.sort(comp)  
$.. ==> [3, 2, 1]
```

- Which part of the anonymous inner class is *really* useful?
 - Interface name (**Comparator**) does not add value
 - Comparator** is a SAM (*single abstract method*) interface
 - there is only one abstract method in **Comparator**
 - method name **compare** does not add value

Computation Context

- A *computation context* wraps around a value, and abstracts away computations associated with the context
 - a “safe box” in which functions can be safely executed
 - e.g. **Optional** is a computation context that handles invalid or missing values
- A computation context comprises:
 - a way to wrap the parameter within the box, e.g. using **of**
`Optional<Integer> oi = Optional.<Integer>of(1)`
 - a way to pass a behaviour into the box via a *higher order method* (method that takes in another method) so that it can be applied to the parameter value

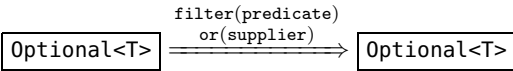
Lambda Expression

- Lambda syntax: (*parameterList*) -> {*statements*}
 - inferred parameter type with body: (x, y) -> { **return** x * y; }
 - body contains a single return expression: (x, y) -> x * y
 - only one parameter: x -> 2 * x
 - no parameter: () -> 1
- A lambda is a implementation of some *functional interface* — interface with single abstract method (SAM)
 - Comparator** is a functional interface with SAM **compare**

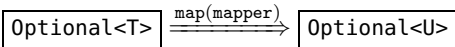
```
jshell> Comparator<Integer> comp = (x, y) -> y - x  
comp ==> $Lambda$15/0x00000001000a9440@68be2bc2  
jshell> list123.sort(comp)  
$.. ==> [3, 2, 1]
```
 - Other useful functional interfaces: **Supplier**, **Consumer**, **Predicate**, **Function**, **BiFunction**, etc.

Some Higher Order Methods in Optional

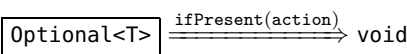
- `Optional<T> filter(Predicate<? super T> predicate)`
- `Predicate<T>` with a single abstract method: **public boolean** test(T t)
- `Optional<T> or(Supplier<? extends Optional<? extends T>> supplier)`
- `Supplier<T>` with a single abstract method: **public T** get()



- `<U> Optional<U> map(Function<? super T,? extends U> mapper)`
- `Function<T,R>` with a single abstract method: **public R** apply(T t)



- void** ifPresent(Consumer<? **super** T> action)
- `Consumer<T>` with single abstract method: **public void** accept(T t)



filter Higher Order Method

- Defining a Predicate<Circle> pred as a function to be applied on a Circle c and returning true or false

```
jshell> Predicate<Circle> pred = c -> c.contains(new Point(0.5, 0.5))
pred ==> $Lambda$20/0x0000000800c0d000@7cd84586
```
- Passing pred to the filter method of Optional<Circle>

```
jshell> createUnitCircle(new Point(0, 0), new Point(1, 1))
$.. ==> Optional[Circle at (0.000, 1.000) with radius 1.0]
jshell> createUnitCircle(new Point(0, 0), new Point(0, 0))
$.. ==> Optional.empty
jshell> createUnitCircle(new Point(0, 0), new Point(1, 1)).
...> filter(pred)
$.. ==> Optional[Circle at (0.000, 1.000) with radius 1.0]
jshell> createUnitCircle(new Point(0, 0), new Point(0, 0)).
...> filter(pred)
$.. ==> Optional.empty
jshell> createUnitCircle(new Point(0, 0), new Point(1, 1)).
...> filter(x -> x.contains(new Point(5.0, 5.0)))
$.. ==> Optional.empty
```
- Optional<T>::filter takes in Predicate<? super T>

13 / 16

map as a Higher Order Method

- E.g. mapping Circle to a Boolean while maintaining the Optional context

```
jshell> Function<Circle, Boolean> f = x -> x.contains(new Point(0.5, 0.5))
f ==> $Lambda$20/0x0000000800c0a420@27973e9b
jshell> createUnitCircle(new Point(0, 0), new Point(1, 1)).map(f)
$.. ==> Optional[true]
jshell> createUnitCircle(new Point(0, 0), new Point(0, 0)).map(f)
$.. ==> Optional.empty
```
- Optional::map takes in Function<? super T, ? extends U>
 - T thing in Optional can be *read* as T or it's supertype
 - result of function can be anything U or it's subtype

```
jshell> Function<Object,Integer> g = x -> x.toString().length()
g ==> $Lambda$26/0x0000000800c14838@6f2b958e
jshell> Optional<Number> on = createUnitCircle(new Point(0, 0), new Point(1, 0)).map(g)
on ==> Optional[40]
jshell> Optional<Number> on = createUnitCircle(new Point(0, 0), new Point(0, 0)).map(g)
on ==> Optional.empty
```

15 / 16

or and ifPresent Higher Order Methods

- Optional<T>::ifPresent takes in Consumer<? super T>

```
jshell> Consumer<Circle> action = c -> System.out.println(c)
action ==> $Lambda$24/0x0000000800c13890@41975e01
jshell> createUnitCircle(new Point(0, 0), new Point(1, 0)).ifPresent(action)
Circle at (0.500, 0.866) with radius 1.0
jshell> createUnitCircle(new Point(0, 0), new Point(0, 0)).ifPresent(action) // skips the action
jshell> createUnitCircle(new Point(0, 0), new Point(0, 0)).
...> ifPresentOrElse(action, () -> { System.out.println("nil");}) // takes in Runnable too!
nil
```
- Optional<T>::or takes in Supplier<? extends Optional<? extends T>>

```
jshell> Supplier<Optional<Circle>> supp = () -> {
...> System.out.println("beep!");
...> return Optional.<Circle>of(new Circle(new Point(0, 0), 1.0));}
supp ==> $Lambda$23/0x0000000800c13058@15615099
jshell> createUnitCircle(new Point(0, 0), new Point(1, 0))
$.. ==> Optional[Circle at (0.500, 0.866) with radius 1.0]
jshell> createUnitCircle(new Point(0, 0), new Point(1, 0)).or(supp)
$.. ==> Optional[Circle at (0.500, 0.866) with radius 1.0]
jshell> createUnitCircle(new Point(0, 0), new Point(0, 0)).or(supp)
beep!
$.. ==> Optional[Circle at (0.000, 0.000) with radius 1.0] // supp is invoked on demand
```

14 / 16

map versus flatMap

- Suppose Circle::contains returns Optional<Boolean>

```
Optional<Boolean> contains(Point p) {
    return Optional.<Boolean>of(this.centre.distanceTo(p) < this.radius);
}
```
- Now consider the following mapping operation:

```
jshell> Function<Circle,Optional<Boolean>> f =
...> x -> x.contains(new Point(0.5, 0.5))
f ==> $Lambda$20/0x0000000800c0ae68@3941a79c
jshell> createUnitCircle(new Point(0,0), new Point(1,1)).map(f)
$.. ==> Optional[Optional[true]] // value of type Optional<Optional<Boolean>>
```
- Use flatMap instead of map

```
jshell> createUnitCircle(new Point(0,0), new Point(1,1)).flatMap(f)
$.. ==> Optional[true]
```
- Higher order method flatMap in Optional:

```
<U> Optional<U> flatMap(
    Function<? super T, ? extends Optional<? extends U>> mapper)
```

16 / 16