CS2030 Lecture 6

The Case Against `null`

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

Semester 1 2023 / 2024

Outline and Learning Outcome

- Appreciate that a **`null`** value is meaningless and may lead to `NullPointerException` as a side effect
- Know how to use Java's `Optional` class to encapsulate invalid or missing values
- Be able to define implementations of functional interfaces to support passing functions as first-class citizens
  - `Comparator`, `Predicate`, `Function`, etc.
- Be able to define anonymous inner classes and lambda expressions
- Be able to use `Optional` values *declaratively*

`null` and `NullPointerException`

- Suppose `createUnitCircle` is defined as

```
Circle createUnitCircle(Point p, Point q) {
    double d = p.distanceTo(q);
    if (d < EPSILON || d > 2.0 + EPSILON) {
        return null; // null is a Circle?
    }
    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);
}
```

and used in the following method pipeline:

$$\texttt{\{p,q\}} \in \texttt{(Point,Point)} \xrightarrow{\texttt{createUnitCircle(p,q)}} \texttt{c} \in \texttt{Circle} \xrightarrow{\texttt{c.contains((0.5, 0.5)}} \texttt{boolean}$$

- Will the above always produce a valid `boolean` outcome?

```
jshell> createUnitCircle(new Point(0, 0), new Point(0, 0)).contains(new Point(0.5, 0.5))
|  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)
```


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"

Java's **Optional** Class

- It is desirable that `createUnitCircle` takes in any value from its domain, and returns a range of values within a co-domain
  - `null` is not in the co-domain of `Circle`
- Requires a co-domain that includes valid circles and no circle
  - `Optional<T>` "wraps" around another object of type `T`
    - may contain a `T`, or maybe empty
    - static factory methods: `of` and `empty`

```
jshell> Optional.<Circle>of(new Circle(new Point(0.0, 0.0), 1.0))
$.. ==> Optional[Circle at (0.000, 0.000) with radius 1.0]

jshell> Optional.<Circle>empty()
$.. ==> Optional.empty

jshell> Optional.<Circle>ofNullable(null) // wrapping around null
$.. ==> Optional.empty
```


Returning an **Optional**

```
Optional<Circle> createUnitCircle(Point p, Point q) {
    double d = p.distanceTo(q);
    if (d < EPSILON || d > 2.0 + EPSILON) {
        return Optional.<Circle>empty();
    }
    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));
}
```

`(p,q) ∈ (Point,Point)` $\xrightarrow{\texttt{createUnitCircle(p,q)}}$ `c ∈ 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(10, 10))
$.. ==> Optional.empty

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


Chaining Methods to an **Optional**

- 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
- Just like any value, a function needs to be a *first-class citizen*
  - assign a function to a variable
  - pass a function as an argument to another method
  - return a function from another method

Ponder...

- Would this work?

```
createUnitCircle(p, q).someHigherLevelMethod(contains(new Point(0.5, 0.5)))
```

- Or something like this?

```
createUnitCircle(p, q).someHigherLevelMethod(new Containment())

class Containment implements SomeFunction<Circle> {
    boolean test(Circle c) {
        return c.contains(new Point(0.5, 0.5));
    }
}

class Optional<T> {
    ...
    Optional<T> someHigherLevelMethod(SomeFunction<T> k) {
        if (this.isEmpty() {
            return this; // if already empty, then remain empty
        } else if (k.test(this.get()) { // get() returns value in Optional
            return this; // if test is true, then return the Optional
        } else {
            return Optional.<T>empty(); // otherwise return Optional.empty
        }
    }
}
```

`Optional<T>::filter` as a Higher Order Method

- `filter` method takes in an implementation of a *functional interface* `Predicate<T>` that specifies the `test` method

```
jshell> class Containment implements Predicate<Circle> {
   ...>     public boolean test(Circle c) {
   ...>         return c.contains(new Point(0.5, 0.5));
   ...>     }
   ...> }
|  created class Containment

jshell> Predicate<Circle> pred = new Containment()
pred ==> Containment@506e1b77

jshell> createUnitCircle(new Point(0.0, 0.0), new Point(1.0, 1.0)).
   ...> filter(pred)
$.. ==> Optional[circle of radius 1.0 centred at point (0.000, 1.000)]

jshell> createUnitCircle(new Point(0.0, 0.0), new Point(-1.0, -1.0)).
   ...> filter(pred)
$.. ==> Optional.empty

jshell> createUnitCircle(new Point(0.0, 0.0), new Point(0.0, 0.0)).
   ...> filter(pred)
$.. ==> Optional.empty
```

- *cf.* `Comparator<T>` with method `compare(T o1, T o2)`

Anonymous Inner Class

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

```
jshell> Predicate<Circle> pred = new Predicate<Circle>() {
   ...>     public boolean test(Circle c) {
   ...>         return c.contains(new Point(0.5, 0.5));
   ...>     }
   ...> }
pred ==> 1@506e1b77

jshell> createUnitCircle(new Point(0.0, 0.0), new Point(1.0, 1.0)).
   ...> filter(pred)
$.. ==> Optional[circle of radius 1.0 centred at point (0.000, 1.000)]

jshell> createUnitCircle(new Point(0.0, 0.0), new Point(-1.0, -1.0)).
   ...> filter(pred)
$.. ==> Optional.empty

jshell> createUnitCircle(new Point(0.0, 0.0), new Point(0.0, 0.0)).
   ...> filter(pred)
$.. ==> Optional.empty
```

- Which part of the anonymous inner class is *really* useful?
  - Interface name (`Predicate`) does not add value
  - `Predicate` has a *single abstract method* (SAM)
    - method name `test` does not add 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`
- Lambda as implementation of a *functional (SAM) interface*

```
jshell> Predicate<Circle> pred = c -> c.contains(new Point(0.5, 0.5))
pred ==> $Lambda$20/0x00007f38c000a628@506e1b77

jshell> createUnitCircle(new Point(0.0, 0.0), new Point(1.0, 1.0)).
   ...> filter(pred)
$.. ==> Optional[circle of radius 1.0 centred at point (0.000, 1.000)]

jshell> createUnitCircle(new Point(0.0, 0.0), new Point(-1.0, -1.0)).
   ...> filter(pred)
$.. ==> Optional.empty

jshell> createUnitCircle(new Point(0.0, 0.0), new Point(0.0, 0.0)).
   ...> filter(pred)
$.. ==> Optional.empty
```


`Optional<T>::filter(? super T)`

- Suppose an appropriate `equals` method is defined in `Circle`
- Can we pass a `Predicate<Object>` to the `filter` method of `Optional<Circle>`?

```
jshell> Predicate<Object> pred = obj ->
   ...> obj.equals(new Circle(new Point(0.0, 0.0), 1.0))
pred ==> $Lambda$20/0x00007f032c00a618@506e1b77
```

- Yes! `Optional<T>::filter(Predicate<? super T>)`

```
jshell> createUnitCircle(new Point(-1.0, 0.0), new Point(1.0, 0.0)).
   ...> filter(pred)
$.. ==> Optional[circle of radius 1.0 centred at point (0.000, 0.000)]

jshell> createUnitCircle(new Point(0.0, 0.0), new Point(0.0, 0.0)).
   ...> filter(pred)
$.. ==> Optional.empty

jshell> createUnitCircle(new Point(-1.0, 0.0), new Point(1.0, 0.0)).
   ...> filter(obj -> obj.equals("some circle"))
$.. ==> Optional.empty
```

`Optional<T>::map` as a Higher Order Method

- `Optional<T>::map` transforms a type `T` value within an `Optional` to a type `R` value, while maintaining the `Optional`
  - takes in a `T`–to–`R` transformation via a `Function<T,R>`
  - returns `Optional<R>`
- `Function<T,R>` is a functional (SAM) interface with single abstract method: `R apply(T)`

```
jshell> Function<Circle, Boolean> fn = c -> c.contains(new Point(0.5, 0.5))
fn ==> $Lambda$20/0x0000000800c0a420@27973e9b

jshell> createUnitCircle(new Point(0, 0), new Point(1, 1)).map(fn)
$.. ==> Optional[true]

jshell> createUnitCircle(new Point(0, 0), new Point(-1, -1)).map(fn)
$.. ==> Optional[false]

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


`Optional<T>::map(? super T, ? extends R)`

- Can we pass `Function<Object,Integer>` to the `map` method of `Optional<Circle>` to return an `Optional<Integer>`?

```
jshell> Function<Object,Integer> fn = x -> x.toString().length()
fn ==> $Lambda$26/0x0000000800c14838@6f2b958e

jshell> Optional<Integer> opti = createUnitCircle(
   ...> new Point(0, 0), new Point(1, 0)).map(fn)
opti ==> Optional[40]

jshell> opti = createUnitCircle(new Point(0, 0), new Point(0, 0)).map(fn)
opti ==> Optional.empty
```

- How about returning an `Optional<Object>`?

```
jshell> Optional<Object> opto = createUnitCircle(
   ...> new Point(0, 0), new Point(1, 0)).map(fn)
opto ==> Optional[40]

jshell> opto = createUnitCircle(new Point(0, 0), new Point(0, 0)).map(fn)
opto ==> Optional.empty
```


Exercise: Finding Coverage

- With `createUnitCircle` returning `Optional<Circle>`, we can proceed to find the coverage over a list of points

```
int findCoverage(Optional<Circle> circle, ImList<Point> points) {
    int numOfPoints = 0;
    for (Point point : points) {
        numOfPoints = numOfPoints + circle
            .filter(c -> c.contains(point)) // Optional<Circle>
            .map(x -> 1) // Optional<Integer>
            .orElse(0); // Integer
    }
    return numOfPoints;
}
```

- Notice how `Optional` is used *declaratively*, as opposed to imperatively
  - **declarative**: *tell* `Optional` *what to do*
  - imperative: ask the value from `Optional` and do it yourself

Do Not Break Encapsulation in `Optional`! `>.<`

- Using `Optional` imperatively is considered bad code! `>.<`

```
int findCoverage(Optional<Circle> circle, ImList<Point> points) {
    int numOfPoints = 0;
    if (circle.isEmpty()) {
        return 0;
    } else {
        Circle c = circle.get(); // refrain from exposing the value in Optional! >.<
        for (Point point : points) {
            if (c.contains(point)) {
                numPoints = numPoints + 1;
            }
        }
        return numPoints;
    }
}
```

- Even though Java's `Optional` declares methods `get()`, `isEmpty()` and `isPresent()`with public access
  - **avoid using them! `>.<`**
- Also **avoid using `equals` method** to "check" the content between two `Optional`s! `>.<`

```
Optional<Circle> emptyCircle = Optional.<Circle>empty()
if (circle.equals(emptyCircle)) { // just another way of checking circle.isEmpty()! >.<
    return 0;
} else ...
```