CS2030 Lecture 7

The Case Against the Null Reference

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

Semester 1 2019 / 2020

Lecture Outline

- Avoiding `NullPointerExceptions` and **`null`** in general
- The `Maybe` object
- Chaining methods to a `Maybe` object
- Anonymous inner classes revisited
  - Methods as first-class citizens
  - Lambdas expressions
- Java's `Optional` class
- Local class and variable capture
- `map` versus `flatMap`

Circle revisited... *yet again*

```java
public class Circle {
    private final Point centre;
    private final double radius;

    private Circle(Point centre, double radius) {
        this.centre = centre;
        this.radius = radius;
    }

    public boolean contains(Point point) {
        return centre.distanceTo(point) < radius + 1E-15;
    }

    static Circle getCircle(Point centre, double radius) {
        return (radius > 0) ? new Circle(centre, radius) : null;
    }

    @Override
    public String toString() {
       return centre.toString() + ", " + radius;
    }
}
```

- What happens to the following?

```java
Circle.getCircle(new Point(0, 0), -1).contains(new Point(0, 0))
```

`null`

"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

Maybe a Circle

- Creating a circle via `getCircle` may return a circle or nothing
- Need an object with connotations of **maybe** that "wraps" around another object of type `T`, i.e. maybe a `T`, or maybe empty

```
public class Maybe<T> {
    private final T thing;

    private Maybe() {
        thing = null;
    }

    public Maybe(T thing) {
        this.thing = thing;
    }

    public static <T> Maybe<T> empty() {
        return new Maybe<T>();
    }

    @Override
    public String toString() {
        return "Maybe[" + (thing == null ? "empty" : thing) + "]";
    }
}
```

Redefining the **getCircle** Method

```
public class Circle {
    ...
    static Maybe<Circle> getCircle(Point centre, double radius) {
        if (radius > 0)
            return new Maybe<Circle>(new Circle(centre, radius));
        else
            return Maybe.empty();
    }
```

- `getCircle` now returns a `Maybe<Circle>` object

```
jshell> Circle.getCircle(new Point(0, 0), 1)
$4 ==> Maybe[(0.0, 0.0), 1.0]

jshell> Circle.getCircle(new Point(0, 0), -1)
$5 ==> Maybe[empty]
```

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

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

Chaining Methods to a `Maybe` Object

- We expect more methods to be chained to a `Maybe` object
  - Let's first reduce the scope of these methods to only those that return **`void`**
  - Reason is that a **`void`** method effectively ends the chain
- Define `ifPresent` method in `Maybe` that takes an "action" as argument to be applied on the object encased within `Maybe`
- The method call should look something like this:

```
Circle.getCircle(new Point(0, 0), 1).ifPresent(..contains(new Point(0, 0)))
```

- But arguments to a method has always been values, i.e. either primitive values or object references
  - How do we pass an "action" to a method?

Taking Inspiration from **List.sort**

`List.sort(Comparable<? super E> comp)`

- *From a previous lecture...*

```
jshell> List<Integer> nums = new ArrayList<>();
nums ==> []

jshell> nums.add(3);
$2 ==> true

jshell> nums.add(1);
$3 ==> true

jshell> nums.add(2);
$4 ==> true

jshell> nums.sort(new Comparator<>() {
   ...> public int compare(Integer x, Integer y) {
   ...> return x - y;
   ...> }})

jshell> nums
nums ==> [1, 2, 3]

jshell>
```

Passing An Object of an Interface

- Define `Actionable` **`interface`** with **`abstract`** `doit` method

  ```
  interface Actionable<T> {
      void doit(T t);
  }
  ```

- Pass an `Actionable` object into `ifPresent`, e.g.

  -
    ```
    Circle.getCircle(new Point(0, 0), 1).ifPresent(
        new Actionable<>() {
            public void doit(Circle c) {
                System.out.println(c.contains(new Point(0, 0)));
            }
        })
    ```

  -
    ```
    Circle.getCircle(new Point(0, 0), 1).ifPresent(
        new Actionable<>() {
            public void doit(Circle c) {
                System.out.println(this);
            }
        })
    ```

Defining `ifPresent` Method in `Maybe` Class

```
public class Maybe<T> {
    ...
    public void ifPresent(Actionable<T> action) {
        if (thing != null) {
            action.doit(thing);
        }
    }
```

```
jshell> Circle.getCircle(new Point(0, 0), 1).ifPresent(new Actionable<>() {
   ...> public void doit(Circle c) {
   ...> System.out.println(c.contains(new Point(0, 0)));
   ...> }})
true
jshell> Circle.getCircle(new Point(0, 0), 1).ifPresent(new Actionable<>() {
   ...>
   ...> public void doit(Circle c) {
   ...> System.out.println(c.contains(new Point(1, 1)));
   ...> }})
false
jshell> Circle.getCircle(new Point(0, 0), -1).ifPresent(new Actionable<>() {
   ...> public void doit(Circle c) {
   ...> System.out.println(c.contains(new Point(0, 0)));
   ...> }})
jshell> Circle.getCircle(new Point(0, 0), 1).ifPresent(new Actionable<>() {
   ...> public void doit(Circle c) {
   ...> System.out.println(c);
   ...> }})
(0.0, 0.0), 1.0
```

From Anonymous Inner Class to Lambda

- Which part of the anonymous inner class is *really* useful?

```
new Actionable<>() {
    public void doit(Circle c) {
        System.out.println(c.contains(new Point(0, 0)));
    }
})
```

- Class name (`Actionable`) does not add value
- If there is only a single abstract method in the class, then the method name (`doit`) does not add value
- A **lambda expression** can be used as a short hand, e.g.

```
(Circle c) -> System.out.println(c.contains(new Point(0, 0)))
```

```
jshell> Circle.getCircle(new Point(0, 0), 1).ifPresent(
   ...> (Circle c) -> System.out.println(c.contains(new Point(0, 0))))
true
```

Lambda Expression

- Lambda syntax: `(`*parameterList*`) -> {`*statements*`}`
- Other lambda variants:
  - inferred parameter type: `(x, y) -> {return x * y;}`
  - body contains a single expression: `(x, y) -> x * y`
  - only one parameter: `x -> 2 * x`
- Most importantly, methods can now be treated as values! 😎
  - assign lambdas to variables
  - pass lambdas as arguments to other methods
  - return lambdas from methods

```
jshell> Actionable<Circle> action = c -> System.out.println(c.contains(new Point(1, 1)))
action ==> $Lambda$23/0x00000008000b4c40@3abbfa04

jshell> Circle.getCircle(new Point(0, 0), 1).ifPresent(action)
false

jshell> Circle.getCircle(new Point(0, 0), 2).ifPresent(action)
true
```

map-ping from One Value to Another

- The `ifPresent` method accepts as argument a lambda expression where the return type is **`void`**
- Define method `map` that accepts a lambda that returns a value
  - The interface needs two type parameters, one for the input and another for the output

    ```
    interface Mappable<T,R> {
        R apply(T t);
    }
    ```

```
public class Maybe<T> {
    ...
    public <R> Maybe<R> map(Mappable<T,R> mapper) {
        if (thing == null) {
            return Maybe.empty();
        } else {
            return new Maybe<R>(mapper.apply(thing));
        }
    }
```

map-ping from One Value to Another

```
jshell> Circle.getCircle(new Point(0, 0), 1).map(new Mappable<>() {
   ...> public Boolean apply(Circle c) {
   ...> return c.contains(new Point(0, 0));
   ...> }})
$9 ==> Maybe[true]

jshell> Circle.getCircle(new Point(0, 0), 1).map(
   ...> x -> x.contains(new Point(1, 1)))
$10 ==> Maybe[false]

jshell> Circle.getCircle(new Point(0, 0), -1).map(
   ...> x -> x.contains(new Point(1, 1)))
$11 ==> Maybe[empty]
```

- `map` converts a value from `Maybe<Circle>` to `Maybe<Boolean>`
- We can now extend the method chain further

```
jshell> Circle.getCircle(new Point(0, 0), 1).map(
   ...> x -> x.contains(new Point(1, 1))).ifPresent(x -> System.out.println(x))
false

jshell> Circle.getCircle(new Point(0, 0), -1).map(
   ...> x -> x.contains(new Point(1, 1))).ifPresent(x -> System.out.println(x))

jshell>
```

Java's `Optional` Class

- Since `Maybe` is so useful, Java has equivalent `Optional` class
- Some familiar methods of the `Optional` class
  - `public void ifPresent(Consumer<? super T> action)`
    - ▷ `Consumer<T>` is a *functional interface* with a single abstract method `accept(T t)`
  - `public <U> Optional<U> map(Function<? super T, ? extends U> mapper)`
    - ▷ `Function<T,R>` is a *functional interface* with a single abstract method `R apply(T t)`
  - `public Optional<T> filter(Predicate<? super T> predicate)`
    - ▷ `Predicate<T>` is a *functional interface* with a single abstract method `boolean test(T t)`

Java's **Optional** Class

- **static** methods that create `Optional` objects:
  - `Optional.of(T value)`
  - `Optional.empty()`
  - `Optional.ofNullable(T value)`

```
import java.util.Optional;

public class Circle {
    ...
    static Optional<Circle> getCircle(Point centre, double radius) {
        if (radius > 0)
            return Optional.of(new Circle(centre, radius));
        else
            return Optional.empty();
    }
```

Java’s `Optional` Class

```
jshell> Circle.getCircle(new Point(0, 0), 1)
$4 ==> Optional[(0.0, 0.0), 1.0]

jshell> Circle.getCircle(new Point(0, 0), -1)
$5 ==> Optional.empty

jshell> Circle.getCircle(new Point(0, 0), 1).ifPresent(x -> System.out.println(x))
(0.0, 0.0), 1.0

jshell> Circle.getCircle(new Point(0, 0), -1).ifPresent(x -> System.out.println(x))

jshell> Point p = new Point(1, 1)
p ==> (1.0, 1.0)

jshell> Circle.getCircle(new Point(0, 0), 1).map(x -> x.contains(p))
$6 ==> Optional[false]

jshell> Circle.getCircle(new Point(0, 0), 2).map(x -> x.contains(p))
$7 ==> Optional[true]

jshell> Circle.getCircle(new Point(0, 0), -1).map(x -> x.contains(p))
$8 ==> Optional.empty

jshell> Circle.getCircle(new Point(0, 0), 1).filter(x -> x.contains(p))
$9 ==> Optional.empty

jshell> Circle.getCircle(new Point(0, 0), 2).filter(x -> x.contains(p))
$10 ==> Optional[(0.0, 0.0), 2.0]

jshell> Circle.getCircle(new Point(0, 0), -1).filter(x -> x.contains(p))
$11 ==> Optional.empty
```

Local Class and Variable Capture

- Take a look at the following test in JShell

```
jshell> boolean flag = false
flag ==> false

jshell> Circle.getCircle(new Point(0, 0), 2).ifPresent(
   ...> x -> flag = x.contains(new Point(1, 1)))

jshell> flag
flag ==> true
```

- Suppose we write the above in a Java file and compile it

```
class Main {
    static boolean foo() {
        boolean flag = false;
        Circle.getCircle(new Point(0, 0), 2)
            .ifPresent(x -> flag = x.contains(new Point(1, 1)));
        return flag;
    }

    public static void main(String[] args) {
        System.out.println(foo());
    }
}
```

```
$ javac Main.java
Main.java:5: error: local variables referenced from a lambda expression
must be final or effectively final
            .ifPresent(x -> flag = x.contains(new Point(1, 1)));
                            ^
1 error
```

Local Class and Variable Capture

- Lambdas and anonymous classes declared inside a method are called *local classes*
- Local class (like local variable) is scoped within the method
  - has access to the variables of the enclosing method/class
  - the local class actually makes a copy of these variables inside itself, i.e. *the local class captures the local variables*
- What happens when the method returns? What happens to the object of the local class?
- Java only allows a local class to access variables that are explicitly declared final or effectively (or implicitly) final
  - An implicitly final variable is one that does not change after initialization

Local Class and Variable Capture

- Java memory model involving variable capture

Dealing with **`Optional<Point>`**

- Suppose **`Point`** maybe invalid for some reason
- Redefine **`Circle`** class with centre as an **`Optional<Point>`**

```java
import java.util.Optional;

public class Circle {
    private final Optional<Point> centre;
    private final double radius;

    private Circle(Optional<Point> centre, double radius) {
        this.centre = centre;
        this.radius = radius;
    }

    public boolean contains(Point point) {
        return centre.get().distanceTo(point) < radius + 1E-15;
    }

    static Optional<Circle> getCircle(Point centre, double radius) {
        if (radius > 0)
            return Optional.of(new Circle(Optional.of(centre), radius));
        else
            return Optional.empty();
    }

    public Optional<Point> getCentre() {
        return this.centre;
    }

    @Override
    public String toString() {
       return centre.toString() + ", " + radius;
    }
}
```

Dealing with `Optional<Point>`

- Let's create a circle via `getCircle`

  ```
  jshell> Circle.getCircle(new Point(0, 0), 1)
  $4 ==> Optional[Optional[(0.0, 0.0)], 1.0]
  ```

- Notice that we obtain an `Optional<Circle>` object in which the centre is a `Optional<Point>` object
- How do we get the `Optional<Point>` from an `Optional<Circle>` via the getCentre method?
  - Using `map`?

    ```
    jshell> Circle.getCircle(new Point(0, 0), 1).map(
       ...> x -> x.getCentre())
    $5 ==> Optional[Optional[(0.0, 0.0)]]
    ```

  - We actually get the point encased in two `Optional`s! i.e. `Optional<Optional<Point>>`

`flatMap`-ping from One Value to Another

- We need to “flatten” two `Optional`s into one
  - That is, to apply the mapping function on the value of an `Optional` and flatten the resulting nested `Optional`
- Use `flatMap` instead of `map`

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

- Example:

```
jshell> Circle.getCircle(new Point(0, 0), 1).flatMap(x -> x.getCentre())
$6 ==> Optional[(0.0, 0.0)]
```

- In the above, `flatMap` takes in a `Function<T,R>` where the input type parameter is a `Circle` and the output type parameter is an `Optional<Point>`

Lecture Summary

- ☐ Appreciate the avoidance of **`null`** in writing effective tests
- ☐ Appreciate the use of a `Maybe` type that wraps around values, and abstracts away the complexities of handling **`null`**
- ☐ Understand how Java functional interface with a single abstract method can be used to realize methods as first class citizens
- ☐ Familiarity with writing lambda expressions in place of anonymous inner classes
- ☐ Appreciate the difference between `map` and `flatMap`
- ☐ Know about the common functional interfaces and situations where they are used