CS2030S Recitation Problem Set 6

Method reference

Let's assume class `A` and method `h`, therefore `A::h`

- Static method:
  - $(a_1, a_2, \ldots, a_n) \rightarrow \mathrm{A.h}(a_1, a_2, \ldots a_n)$
- Non-static method:
  - $(a_1, a_2, \ldots, a_n) \rightarrow a_1.\mathrm{h}(a_2, \ldots, a_n)$

Maybe

- What is a maybe?
  - Maybe is a monad
- What is a monad?
  - A monad is just a monoid in the category of endofunctors, what's the problem? 😆

Maybe

- Ok thanks, but legit what is a monad
  - Comes from a branch of abstract mathematics called category theory
  - But for Software Engineers, just think of it as a box.

Maybe

- Ok but why do I need a box.
  - We want to abstract out `null` checks.
  - This "null" would be represented by the `None`.
- Now we can create APIs on Maybe to work on the value, generally,
  - if it is still a `Some` do whatever we would've normally done
  - if it is a `None` then we just propogate the `None`
- This allows us to chain operations *functional programming* 🤩

Maybe

- APIs (we'll use x as the value here)
  - `of`: Creates a `Maybe` containing x, or `None` if x was a `null`
  - `map`: Takes in $f : X \to Y$ and applies on x to produce a `Maybe` containing $f(x)$ (context is preserved)
  - `filter`: takes in $f : X \to B$, turns `Maybe` into a `None` if $f(x) = false$

Maybe

- More APIs
  - `flatMap`: Takes in $f : X \rightarrow Maybe < Y >$ and applies on x to produce a `Maybe` containing $f(x)$ and composes it to produce a `Maybe` containing $f(x)$ (contexts are composed)
  - `orElse`: Takes in $f : () \rightarrow \mathrm{X}$, if `Some` return x, else produce the value of the producer ie $f()$
  - `ifPresent`: Takes in $f : X \rightarrow \mathrm{void}$. Only if x is present then consume the x.

Question 1 (finally)

```
Maybe<Internship> match(Resume r) {
    if (r == null) {
        return Maybe.none();
    }

    Maybe<List<String>> optList = r.getListOfLanguages();
    List<String> list;

    if (optList.equals(Maybe.none())) {
        list = List.of();
    } else {
        list = optList.get(); // cannot call
    }

    if (list.contains("Java")) {
        return Maybe.of(findInternship(list));
    } else {
        return Maybe.none();
    }
}
```

Question 1 (Observations)

```
    if (r == null) {
        return Maybe.none();
    }
```

- Notice that this will be handled by the `of`? if `r` was null it would have correctly created a `None`.

```
return Maybe.of(r)
```

Question 1 (Observations)

```
    Maybe<List<String>> optList = r.getListOfLanguages();
    List<String> list;
```

- What is the type of `r::getListOfLanguages`?
  - Seems to be returning a `Maybe sth`
  - Sign that we should use `flatMap`

```
return Maybe.of(r).flatMap(x -> x.getListOfLanguages())
```

Question 1 (Observations)

```
    if (optList.equals(Maybe.none())) {
        list = List.of();
    } else {
        list = optList.get(); // cannot call
    }
```

- It's a bit trickier here. Let's break down what's happening
  - Since our previous `getListOfLanguages` could be `None`, we want to make sure we have an empty list. Else, we get the list from the `Maybe<List>`
- Not clear if we have to do anything right now.

```
return Maybe.of(r).flatMap(x -> x.getListOfLanguages())
```

Question 1 (Observations)

```
    if (list.contains("Java")) {
        return Maybe.of(findInternship(list));
    } else {
        return Maybe.none();
    }
```

- We need to see if the list has Java
  - A sign telling me to use `filter`
  - If filter fails, it should give a `None`
- If the `Maybe<List>` was a `None` we would return `None` as well.
- A sign is telling me to `filter` and then `map` because if we `map` a `None` we still get a `None`

Question 1: Putting it all together

```
return Maybe.of(r)
    .flatMap(x -> x.getListOfLanguages())
    .filter(lst -> lst.contains("Java"))
    .map(lst -> findInternship(lst));
```

Look at how elegant this is. 😆

I ❤️ FP

Question 2

```
class A {
    private int x;
    public A(int x) {
        this.x = x;
    }
    public int get() {
        // Line A
        return this.x;
    }
}
```

Draw the contents of the stack and heap at Line A.

```
A a = new A(5);
Producer<Integer> p = () -> a.get();
p.produce();
```

Question 2

- Remember that for stack and heap, we think of lambda functions as *syntatic sugar* for anonymous classes.
- In reality, not really anonymous classes
  - This is due to the lexical `this`.
  - `this` in lambda refers to the class containing the lambda.
  - `this` in a lexically replaced anonymous class refers to the instance of the anonymous class.
  - lexical `this` refers to the class containing the lambda.

Question 2

```
Producer<Integer> p = () -> a.get();
```

- Notice on this line, `a` needs to be captured.
  - Because `a` might be removed from the stack before `p.produce` is called.

Question 2