CS2030S Recitation

Week 9: Problem Set 6

brian

2025-10-15

National University of Singapore

Recap

Recap: Maybe

- `NullPointerException`s are annoying
- Bake the possibility of being “nothing” within the type of the object
- Conceptually just a box
  - `None` represents having no value (empty box)
  - `Some` is a “box” with the value inside
- Use APIs to interact with the value inside
  - Can chain API calls since they always return a `Maybe<T>`

Recap: Maybe APIs

- `of`: Creates a `Maybe` containing our value (or `None` if given a `null`)
  - “Lifting” a type `T` into type `Maybe<T>`
- `map`: Takes a function (`T -> U`)
  - If `Some`, apply function on the value
  - If `None`, propogate the `None`
- `filter`: Takes a predicate function
  - If `Some`, apply function and convert to `None` if function returns false
  - If `None`, propogate the `None`

Recap: More APIs

- `flatMap`: Takes in `f: T -> Maybe<U>`
  - If `Some`, apply `f` and flatten the maybe
  - If `None`, propogate the `None`
- `orElse`: Takes in `f: () -> U`
  - If `Some`, return value
  - If `None`, return `f()`
- `ifPresent`: Takes in `f: T -> ()`
  - If `Some`, consume the value with `f`
  - If `None`, propogate the `None`

Recap: Anonymous Class

- Declare a local class and instantiate in one statement
- Has the form `new X(arguments) { body }`
  - `X` is the class/interface that you inherit from
  - body is the methods of that class, just no constructor

Recap: Functions and λ-functions

- If an anonymous class implements an interface with one method
- Then it is kinda like a function (only one method to call)
- λ-function is an *anonymous* function
- Can replace these functional interface with lambda expressions
  - `(arguments) -> { body }`
  - Can omit type of variables and { } if it is a single return statement
- Stack and heap treates anonymous functions as anonymous classes
- More concepts like currying and closure can be seen in notes

Q1: Finding internship

Rewrite using functional style using `Maybe` (single return statement)

```java
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();
  }
}
```

Q1: Finding internship

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

- This is taken care of with `of`
  - `Maybe.of(r)`

Q1: Finding internship

```
Maybe<Internship> match(Resume r) {
  :
  Maybe<List<String>> optList = r.getListOfLanguages();
  :
}
```

- We see that the return type of `getListOfLanguages` is a `Maybe`
  - Hint that we should use `flatMap`
  - `.flatMap(x -> x.getListOfLanguages())`

Q1: Finding internship

```
Maybe<Internship> match(Resume r) {
  :
  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();
  }
}
```

- If `None`, stays `None` so we just can continue normally with mapping etc
  - Use filter to check if contains "Java"
  - `.filter(lst -> lst.contains("Java"))`

Q2: Draw stack and heap diagram

```java
class A {
  private int x;

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

With the following in main:

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

The End

bye!