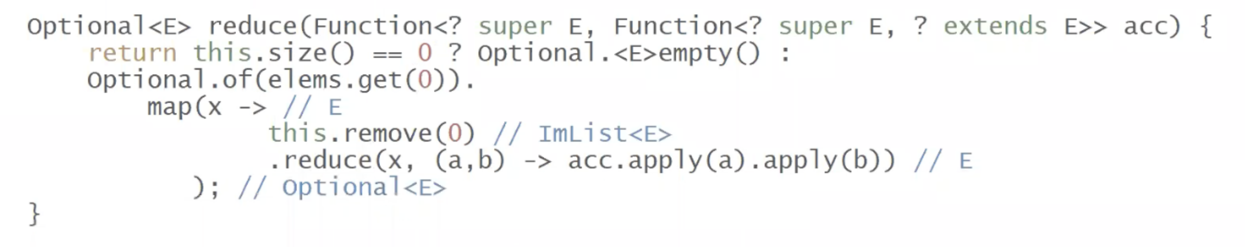
Generics (CS2030)



```
Optional<E> reduce(Function<? super E, Function<? super E, ? extends E>> acc) {
    return this.size() == 0 ? Optional.<E>empty() :
    Optional.of(elems.get(0)).
        map(x -> // E
                this.remove(0) // ImList<E>
                .reduce(x, (a,b) -> acc.apply(a).apply(b)) // E
           ); // Optional<E>
}
```

Predicate<T>

boolean test(T t)

Example:

Predicate<String> pred = x → x.isEmpty();

pred.test(””) // returns true

Commonly used in Optional<T>.filter(Predicate<? super T> predicate)

If a value is present, and the value matches the given predicate, return an Optional describing the value, otherwise return an empty Optional.

Lambda example:

Optional<Circle> u;

u.filter(x → x.contains(new Point(5.0, 5.0));
// returns u if it contains the Point else returns Optional.empty()

Consumer<T>

void accept(T t)

Example:

Consumer<String> con = x → System.out.println(x);

con.accept(”Hello World”) // prints Hello World

Used in Optional<T>.ifPresent(Consumer<? super T> consumer)

If a value is present, invoke the specified consumer with the value, otherwise do nothing.

Lambda example:

Optional<Circle> u;

u.ifPresent(x → System.out.println(x));

Supplier<T>

T get()

Example:

Supplier<Integer> sup = () → 1

int i = sup.get(); // returns 1

Commonly used in Optional<T>.or **

Lambda example:

Function<T, U>

U apply(T t)

Example:

Function<Integer, String> func = x → String.format(”%d is a number”, x);

String str = func.apply(5); // str = “5 is a number”;

Used in Optional.map(map(Function<? super T,? extends U> mapper)

If a value is present, apply the provided mapping function to it, and if the result is non-null, return an Optional describing the result.

Lambda example:

```
Optional<Number>.map(x → x.toString().Length());
```

The Lambda can be broken down into:

```
Function<Object, Integer> g = x → x.ToString().length();
```

When to use flatMap when to use map?

Example:

Not a very good example cause weird to get a Optional<Boolean> in this context

```
Optional<Boolean> contains(Point p) {
    return Optional.<Boolean>of(this.centre.distanceTo(p) < this.radius);
}
Function<Circle, Optional<Boolean>> f = x → x.contains(new Point(0.5, 0.5));
```

Circle.map(f) will return a Optional[Optional[Boolean]]

This is when you should use flatMap

When you are mapping a context to the same context

map takes out the value from LHS , changes it to RHS, wraps RHS in an Optional

“When map takes in a resultant that is the context itself”

Another example:

```
Fraction add(Fraction other) {
    Optional<Num> a = this.opt.map(x -> x.first());
    Optional<Num> b = this.opt.map(x -> x.second());
    Optional<Num> c = other.opt.map(x -> x.first());
    Optional<Num> d = other.opt.map(x -> x.second());
    Optional<Num> ad = a.flatMap(x -> d.map(y -> x.mul(y)));
    Optional<Num> bc = b.flatMap(x -> c.map(y -> x.mul(y)));
    Optional<Num> denom = b.flatMap(x -> d.map(y -> x.mul(y)));
    Optional<Num> numerator = ad.flatMap(x -> bc.map(y -> x.add(y)));
    Optional<Frac> f = numerator.flatMap(x -> denom.map(y -> Frac.of(x,y)));
    return new Fraction(f);

}
```

```
Optional<Num> bc = b.flatMap(x -> c.map(y -> x.mul(y)));
```

x = Num within b

y = Num within c

x.mul(y) = b * c

c.map(y → x.mul(y)) → returns b * c wrapped in an Optional<Num>

hence you need to use flatMap

When do you need to declare type

```
static <T> Maybe<T> of(T value) {

    return new Maybe<T>;

}
```

“Looking for something to bind to <T>” thats why you need the declaration of <T> in the method

this is because static generic methods can be declared without an instance, by declaring type ensures that Maybe is of type T