Functional Programming Part 2

Terence Sim

Functor laws

A functor is a generic type that contains a thing (aka *payload*), and provides a constructor (`make`), and a `map` method that transforms the payload. These must satisfy 2 laws:

1. Identity:

```
make(t).map(x -> x).equals(make(t))
```

2. Associativity:

```
make(t).map(f).map(g).equals(
    make(t).map(f.andThen.g))
```

Functors in Java

Optional<T>:

- Separates logic for handling `null` values from normal processing.
- Constructor: `of`, Method: `map`

Stream<T>:

- Provides lazily evaluated lists
- Constructor: `of`, Method: `map`

ArrayList<T>:

- Stores multiple values which are accessible by a numeric index (via the `get` method).
- Provides easy way to handle multiple values without loops
- Constructor: `Arrays.asList`
- But missing a `map` method.

Easy to provide a `map` for `ArrayList`

```
<T,U> List<U> listMap(Function<T,U> f,
                      List<T> list) {
    List<U> newList = new ArrayList<>();
    for (T item : list)
        newList.add(f.apply(item));
    return newList;
}
```

`listMap` removes burden of looping

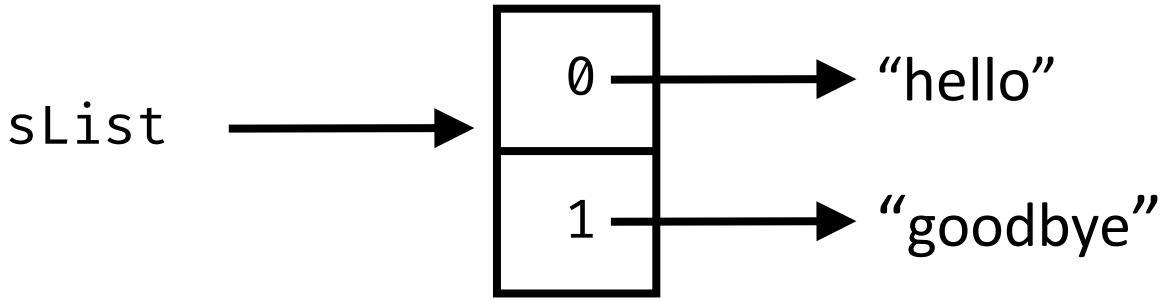
```
List<Integer> intList = Arrays.asList(1, 2, 3, 4, 5);

List<Integer> newList = listMap(x->x*x, intList);
⇒ [1, 4, 9, 16, 25]
```

Instead of this:

```
List<Integer> newList = new ArrayList<>();
for (int x : intList)
    newList.add(x*x);
```

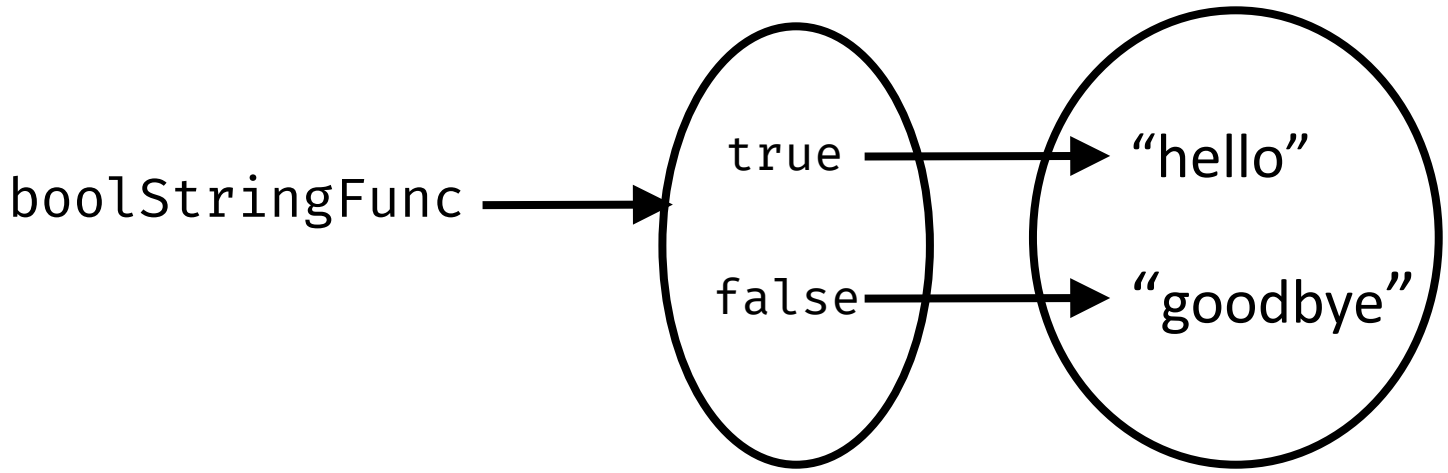
Visualizing `ArrayList`

```
List<String> sList = Arrays.asList("hello",
"goodbye");
```



Visualizing Function

```
Function<Boolean, String> boolStringFunc =
   v -> v ? “hello” : “goodbye”;
```



Interface Function<T,U>

Surprise! This is also a functor.

Think of it as an `ArrayList` whose index is of type `T`, and payload is of type `U`.

eg. The `boolStringFunc` stores "`hello`" at index `true`, and "`goodbye`" at index `false`.

```
Function<Boolean, String> boolStringFunc =
   v -> v ? "hello" : "goodbye";
```

The Constructor is Java's assignment of lambda expression, and the `map` method is `andThen`

```
boolStringFunc.andThen(String::length)
```

This is a `Function<Boolean, Integer>` that stores 5 at index `true`, and 7 at index `false`.

Functors aren't powerful enough

How can we add 2 Sandboxes of integers?

```
Sandbox<Integer> s1 = Sandbox.make(3);
Sandbox<Integer> s2 = Sandbox.make(5);
Sandbox<Integer> s3 = s1 + s2;
```

Monads

What happens if the Sandbox `map` method is given `f` with signature: `Function<T, Sandbox<U>>` ?

Then, `make(t).map(f)` returns `Sandbox<Sandbox<U>>`

That is, the transformed payload is now wrapped in a Sandbox inside another Sandbox. This is troublesome!

Thus, it is often useful to have a `flatmap` method, which "flattens" or unwraps one of the box. So:

`make(t).flatmap(f)` returns `Sandbox<U>`

The data type is then called a Monad.

Monads

A Monad is a parametrized type that contains a thing, along with a constructor (`of`, aka `unit`), and a method `flatmap` (aka `bind`).

`Optional<T>` and `Stream<T>` are also monads.

Monads must obey 3 laws. Please look them up:

https://medium.com/@afcastano/monads-for-java-developers-part-1-the-optional-monad-aa6e797b8a6e

Monads

How can we add 2 Sandboxes of integers?

```
Sandbox<Integer> s1 = Sandbox.make(3);
Sandbox<Integer> s2 = Sandbox.make(5);
Sandbox<Integer> s3 =
      s1.flatmap(t1 ->
                  s2.map(t2 -> t1 + t2));
```

We can generalize this concept …

combine aka liftM2

```
public <U,R> Sandbox<R> combine(Sandbox<U> s,
             BiFunction<T,U,R> binOp) {
   return this.flatmap(t1 ->
                          s.map(t2 ->
                            binOp.apply(t1, t2)));
}

Sandbox<Integer> s1 = Sandbox.make(3);
Sandbox<Integer> s2 = Sandbox.make(5);
Sandbox<Integer> s3 = s1.combine(s2, (x,y)->x+y);
```

Lecture Summary

Functional Programming is a style that emphasizes pure functions, and declarative coding.

FP makes code easier to reason about, test, debug, optimize, and parallelize.

Functors are generic boxes (aka context) that contain values (aka payload) and provide a useful service:

- Optional: handling of null values
- Sandbox: handling exceptions
- ArrayList: handling multiple values without looping

Monads are also common in FP.