Lecture 9

FP Patterns

Functions with Multiple Inputs

```
(x,y,z)->(x+y+z)/3
```

Function<T,R>

BiFunction<T,U,R>

Currying

Haskell Curry

```
x ->
(y,z)->(x+y+z)/3
```

```
x ->
y -> (z ->(x+y+z)/3)
```

x -> y -> z ->

(x+y+z)/3

```
Function<T,Function<U,R>>
```

```
BiFunction<T,U,R>
```

```
Function<T,
Function<U,Function<V,R>>>
```

```
int add(x, y) {
   return x+y;
}
```

```
Function<Integer,Function<Integer,Integer>>
add = x -> y -> x+y;
```

```
add(1, 1); // normal method
add.apply(1,1) // BiFunction
```

```
add.apply(1).apply(1); // curried
```

```
add.apply(1).apply(1);
incr = add.apply(1);
decr = add.apply(-1);
```

```
double randomExpValue(Random rng, double rate) {
  return -Math.log(rng.nextDouble()) / rate;
}

double generateServiceTime() {
  return randomExpValue(rng, serviceRate);
}

double generateInterArrivalTime() {
  return randomExpValue(rng, arrivalRate);
}

:
```

```
BiFunction<Random, Double, Double> randomExpValue =
    (rng, rate) -> -Math.log(rng.nextDouble())/rate;

Supplier<Double> generateServiceTime = () ->
    randomExpValue.apply(rng, serviceRate);

Supplier<Double> generateInterArrivalTime = () ->
    randomExpValue.apply(rng, arrivalRate);

:
```

```
Function<Random, Function<Double, Double>>
randomExpValue =
    rng -> rate -> -Math.log(rng.nextDouble()) / rate;

Function<Double,Double> f =
    randomExpValue.apply(rng);

Supplier<Double> generateServiceTime = () ->
    f.apply(serviceRate);

Supplier<Double> generateInterArrivalTime = () ->
    f.apply(arrivalRate);

:
```

Functional Interface

aka

SAM Interface

```
@FunctionalInterface
interface FindServerStrategy {
    Server findServer(Shop shop);
}
```

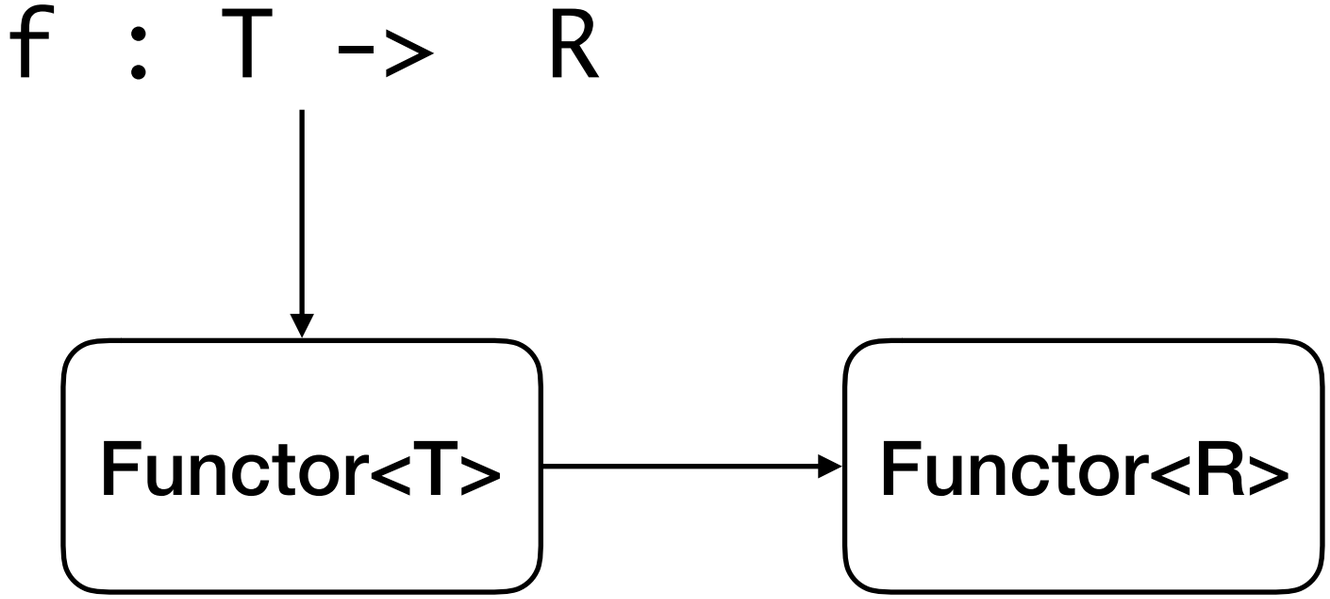
```
Function<Shop,Server> fss =
        s -> s.getRandomServer();

Function<Shop,Server>  fss =
        s -> s.getShortestQueue();

fss.apply(shop)
```

Functor

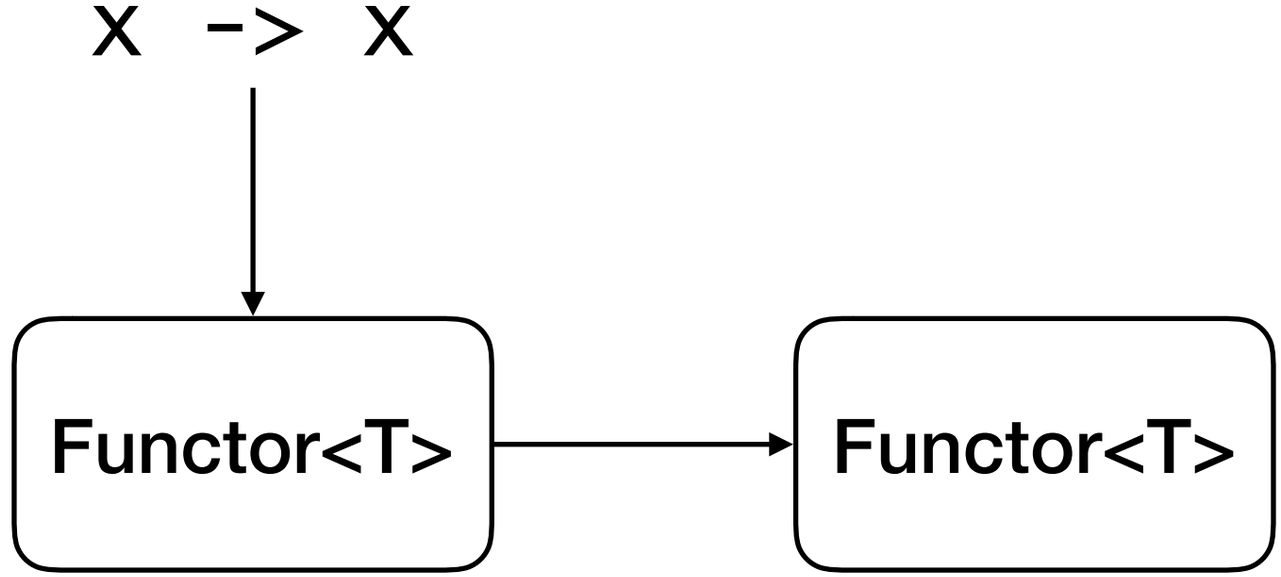
Explained in OO way



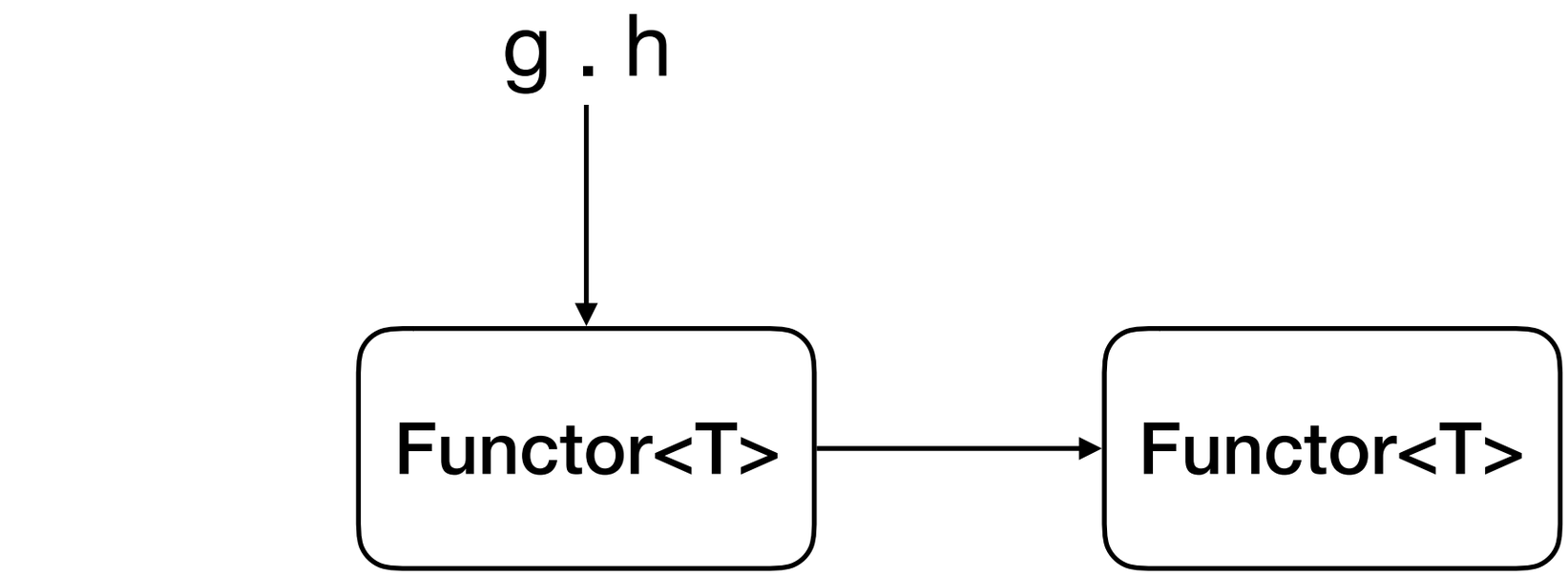
```
interface Functor<T> {
  public <R> Functor<R> f(Function<T,R> func);
}
```

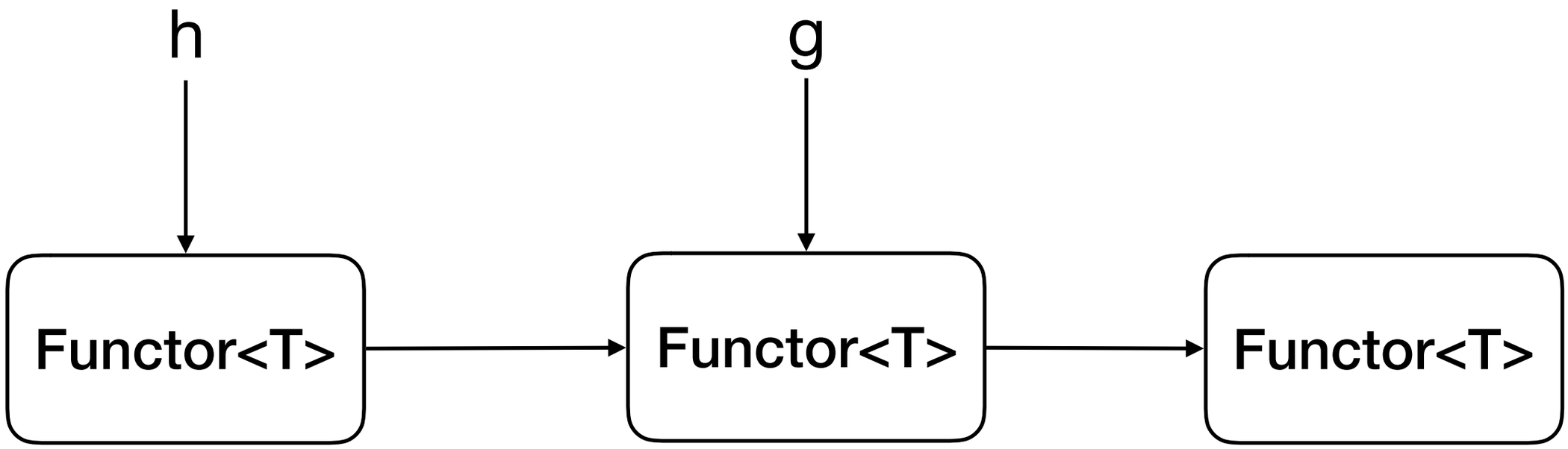
Functor Laws

Explained in OO way

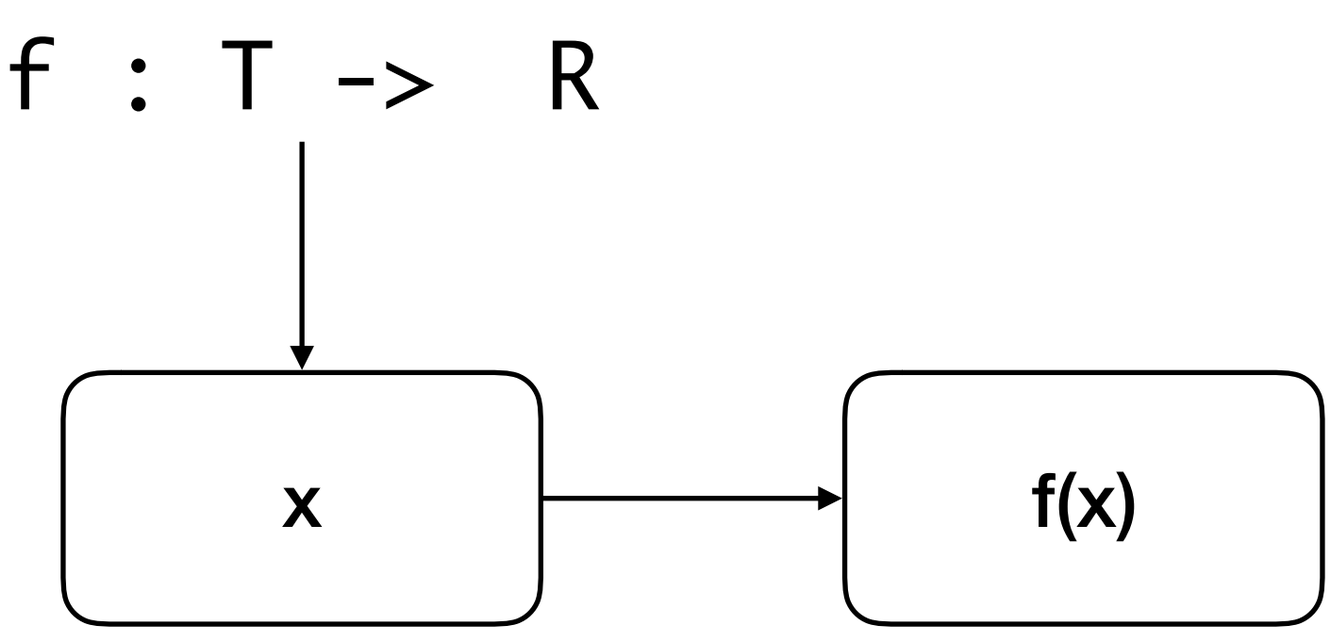


No change!


g . h
Functor<T>
Functor<T>


h
g
Functor<T>
Functor<T>
Functor<T>

“modify things in a box”



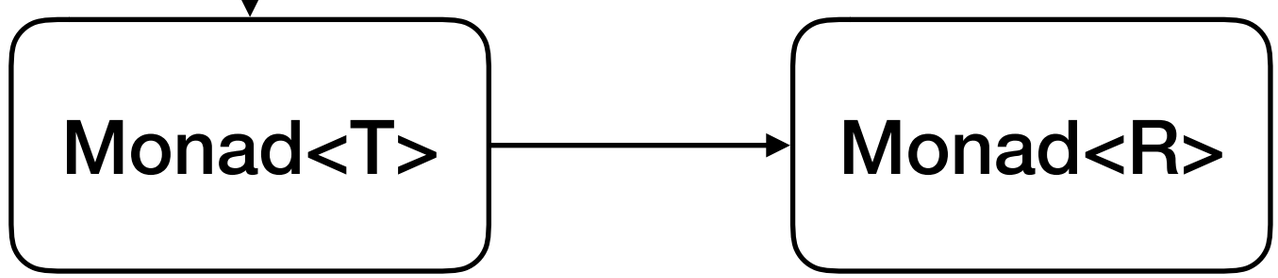
Functors

- LambdaList
- InfiniteList
- Stream
- Optional
- :

Monad

Explained in OO way

```
f : T -> Monad<R>
```



```
interface Monad<T> {
  public <R> Monad<R> f(Function<T,Monad<R>> func);
}
```

```
interface Monad<T> {
  public <R> Monad<R> f(Function<T,Monad<R>> func);
}
```

```
interface Stream<T> {
  public <R> Stream<R> flatMap(Function<T,Stream<R>> func);
}
```

Monad Laws

- `Monad.of(x).flatMap(f)`
  `= f(x)`

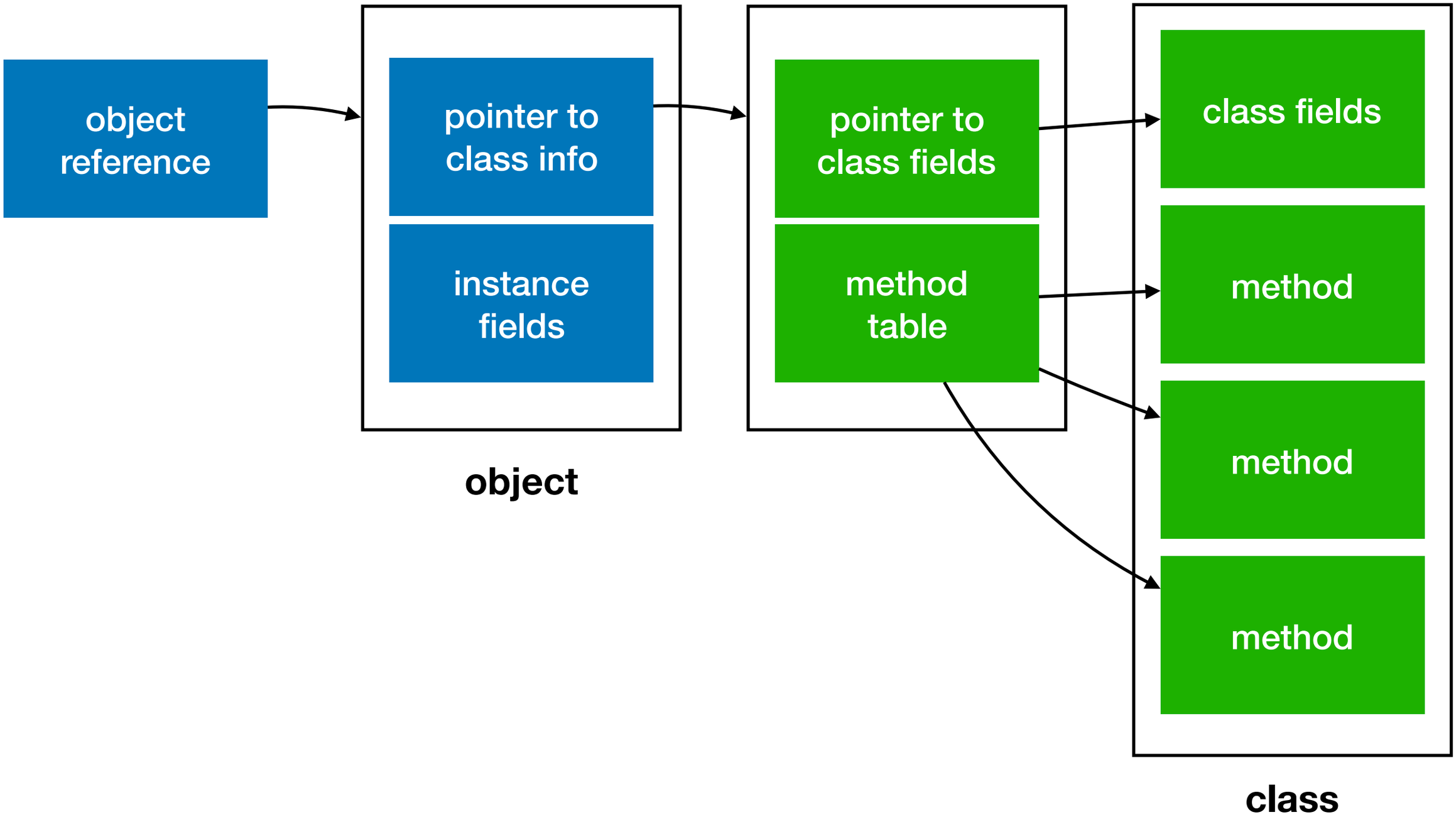
- `monad.flatMap(x -> Monad.of(x))`
  `= monad`

- `monad.flatMap(f).flatMap(g)`
  `= monad.flatMap(x -> f(x).flatMap(g))`

Monads

- Stream
- Optional

Using lambdas for policy / strategy (instead of polymorphism)


object reference
pointer to class info
instance fields
object
pointer to class fields
method table
class fields
method
method
method
class

Using lambdas as observers