

Functional Programming in Java

↳ have an input : domain
↳ maps to output : codomain.

} mathematical function

$$f: X \rightarrow Y$$

domain codomain

$$f(x) \Rightarrow a$$

input output

- referential transparency.

↳ $f(x) = a \Rightarrow x \leftrightarrow a$
Replace interchangeably.

↳ every time call $f(x) \Rightarrow a$ looks
↳ same result, deterministic.

- no side effects

Pure Functions

↳ deterministic, transparent, no side effects.

eg. don't print to screen
write to file
throw exceptions.
change other variables \Rightarrow might change state of function.
modify the values of the arguments.

↳ eg.

```
int square(int i) {  
    return i * i;  
}
```

// pure.

```
int add(int i, int j) {  
    return i + j;  
}
```

// pure

```
int div(int i, int j) {
```

return i/j; \Rightarrow if $j = 0$, then can throw exception.

}

\therefore not pure.

```
int incrCount(int i) {
```

return this count + i;

}

may not be final.

\therefore not deterministic as can return diff values per input

\therefore not pure.

↳ not referentially transparent

pure function \Rightarrow much easier to understand the result
of computing / reason about our complex programs.
 \Rightarrow relies to apply and compose.

method has to be associated with a class.

Function as First-Class Citizens in Java.

- need to instantiate an object in order to find the function group.

@FunctionalInterface

- interface with a single abstract method.

eg. interface Comparator

↳ can be used as an assignment target for lambda expression.

$(x, y) \rightarrow f(x, y)$ ^{arrow token} eg. $(x, y) \rightarrow x+y$.

only for
FunctionalInterface

only 1 method
that is abstract.
or we anonymous class

```
Comparator<String> comp = new Comparator<String>() {
    public int compare (String s1, String s2) {
        ...
    }
};
```

↓

```
Comparator<String> comp = (String s1, String s2) -> {
    return s1.length() - s2.length(); // block for multiple things.
}
```

↓

```
Comparator<String> comp = (s1, s2) -> s1.length() - s2.length();
```

initiate this object of type comparator.

rely on Java Type inference

↳ note: 1 argument no need parenthesis
eg. $comp = i \rightarrow i+1$

$(x_1, \dots, x_n) \mapsto \{ \dots \text{return } \dots \}$

need to know what is the function in the class

Curried Function

$(x, y) \rightarrow f(x, y)$

↓
 $[x \rightarrow [y \rightarrow f(x, y)]]$

eg. int x=1;

Transformer<Integer, Integer> t = y -> x+y;

↓

Transformer<Integer, Transformer<Integer, Integer>> t = x -> y -> x+y;

t.transform(4) ⇒ get back a lambda.
⇒ set value of x to be 4.

higher order function.

↓

t.transform(4).transform(5) ⇒ get back 9.

- eg. create a add-one function.

Transformer<Integer, Integer> addOne = t.transform(1);

take this and addOne to everything else.

Lambda as Closures.

Transformer <Idet, Double> dist = f → origin.distanceTo(p);

closure: store a function and its enclosing environment

↓

captured variable needs to be fixed / effectively final.

↓

= origin::distanceTo;

↓

method reference: distanceTo method on origin

↓

Compiler will check if the method has a single argument point

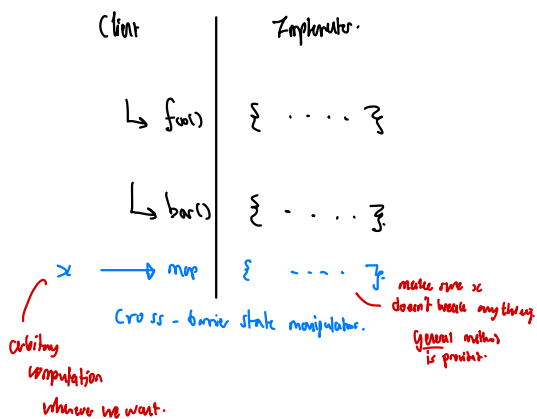
instance method: origin::distanceTo

class method: Box::of

↓

just a method that is written in another class

Abstraction Barrier



Lazy evaluation

↳ Delayed.

eg. interface Task {
void run();
}

↳ *function interface*

interface Runnable {
void run();
}

↳ *non-void function*

Producer <Task> p = () → 3;

↳ *no arg.*

'shell / kit

p.get()

Producer <Task> p = () → { System.out.println("Execute"); return 3; };

p.get() → Execute.

↳ *delayed; not evaluated.*

Lazy

↳ delayed evaluation until we need it.

↳ no need to recompute all computed values.

Logger.log(Logger.LogLevel.INFO, () → { System.out.println("Execute"); return "Hello World"; });

↓

now executed

= now executes until the log level is correct.

enum of 0, 1, 2

INFO, WARN, ERROR.

keep track of non-boolean state.

↓

will print out the name or thing.

Normalization: Store that value, then return if value asked for.

```

class Lazy<T> {
    private T value;
    private boolean evaluated;
    private Producer<T> producer;

    public Lazy(Producer<T> producer) {
        this.producer = producer;
        this.value = null;
        this.evaluated = false;
    }

    public T get() {
        // has the value been evaluated
        // ↳ lazy.
        if (!evaluated) {
            this.value = this.producer.get();
            this.evaluated = true;
            // must be deterministic => same value every time.
        }
        return this.value;
        // ↳ only the value that is stored will be returned again
    }
}

```