Functional Programming In Java.

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

} mathematical function

$f : X \to Y$.

/ \

domain codomain

$f(x) \Rightarrow a$

/ |

| input | output.

- referential transparency.
  - $f(x) = a \Rightarrow f(x) \leftrightarrow a$.
    replace interchangeablely.
  - every time call $f(x) \Rightarrow a$ back.
    ↳ same results, deterministic.
- no side effects.

Pure Functions

↳ deterministic, transparent, no side effect:

eg. don't print to screen
| write to file
| throw exceptions.
| change other variables ⇒ 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;    ⇒ if j = 0, then can throw exception.
}                     ∴ not pure.

int incrCount(int i) {
    return this.count + i;
}
```

this.count ⇓ may not be final.
∴ not deterministic as can return diff value per input
∴ not pure.
↳ not referentially transparent

pure function ⇒ much easier to understand the result of computing / reason about our complex programs.
⇒ easier 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 pass the function around.

@FunctionalInterface:

- Interface with a single abstract method.

eg. interface Comparator.

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

arrow token.

$(x, y) \to f(x, y)$ eg. $(x, y) \to x+y$.

only for FunctionalInterface

only 1 method. that is abstract.

else we anonymous class

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

⇓

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

⇓

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

instantiate this object of type comparator.

rely on Java Type Inference;

↳ note: 1 argument no need parenthesis.

eg. `cmp = i -> i+1;`

$(x_1 \dots x_n) \to \{ \ \dots \ \text{return} \dots\dots \};$

↳ need to know what is the function in the class

Curried Function

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

⇓

$[x \to [y \to f(x, y)]]$

eg.
```
int x=1;
Transformer<Integer, Integer> t= y -> x+y;
```

⇓

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

higher order function.

`t.transform(4)` ⇒ get back a lambda.

⇒ set value of x to be 4.

⇓

`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.

closure: stores a function and its enclosing environment

```
Transformer <Point, Double> dist = p -> origin.distanceTo(p);
```

captured variable needs to be final / effectively final.

```
= origin::distanceTo ;
```

Compiler will check if the method has a single argument point

method reference: distanceTo method on origin.

instance method: origin::distanceTo

class method: Box::of

just a method that is written in another class

Abstraction Barrier

Client | Implementer.

↳ foo() | { . . . . }

↳ bar() | { . . . . }.

x → map | { . . . . }.

Cross - barrier state manipulator.

arbitrary computation whenever we want.

make sure x doesn't break anything

General method is provided.

Lazy evaluation

↳ Delayed.

functional interface

```
eg. interface Task {            interface Producer <T> {
        void run();                     T get();
    }                               }
```

nullary function

```
Producer <Integer> p = () -> 3;
```

no args.

jshell / list

```
p.get()
```

```
Producer <Integer> p = () -> { System.out.println("Execute"); return 3; }.
```

↳ delayed; not evaluated.

p.get() ⇒ Execute.

Lazy

↳ delayed evaluation until we need it.

↳ no need to recompute alr. computed value.

```
Logger.log( Logger.Loglevel. INFO. , () -> {System.out.println("Execute") return "Hello World"; });
```

⇓ never executed

∴ never evaluate until the log level is correct.

enum e{ 0, 1, 2 }

⇓ INFO, WARNING, ERROR.

keep track of non-boolean state

⇓ will print out the name as strings.

Memoization: Store that value, then return it when 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() {
        if (evaluated) {
            this.value = this.producer.get();
            this.evaluated = true;
        }
        return this.value;
    }
}
```

has the value been evaluated
↳ lazy.

must be deterministic ⇒ same value every time.

only the value that is stored will be returned again