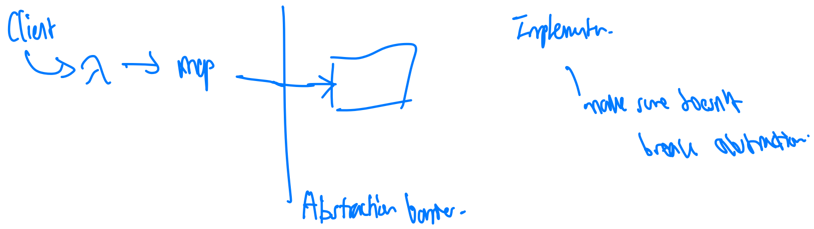
PE2 → unit 36 (Monad)

↳ Lab 6. incl. infiniteList (simpler)

↳ Recitation 9.

↳ 9 Apr 9-12pm.

Function as a cross-barrier state manipulator




int incr(int x)

↳ return x+1;

int abs(int x)

↳ return x>0 ? x : -x;

composition: abs(incr(incr(-3)))

⇒ 1

Logs information of method? ⇒ For debugging...

Pair <Integer, String> incrWithLog(int x)

↳ return new Pair<>(incr(x), "incr "+x);

incrWithLog(4)

⇒ (5, incr 4)

Cannot compose these!

eg. absWithLog(incrWithLog(-3))

⇒ Incompatible types.

⇓

instead.

Pair<Integer,String> absWithLog(Pair<Integer,String> p)

↳ new Pair<>(abs(p.first), p.second + " abs " + p.first)

⇓

p = [-3, ]

incrWithLog(p) — bringing along side info.

⇒ [-2, incr -3]. ⇒ abstract out?

incrWithLog(p)

⇒ [-1, incr -3 incr -2].

# Loggable

↳ takes a value.

↳ side info about value.

↳ Loggable.of(-3).
    ⇒ value: -3, log:

↳ Loggable.of(-3).incrWithLog().incrWithLog()
    ⇒ value: -1, log: incr -3; incr -2;

↳ OOP way.

↳ new operation = new method?
    NOT GREAT!

↳ Functions of Loggable depends on implementer.

```
Loggable map(Transformer<Integer, Integer> t)
    ↳ return new Loggable(t.transform(this.value), this.log);
```

⇓

```
Loggable.of(-3).map(x → incr(x))
    ⇒ value: -2, log:

Loggable.of(-3).map(x → incr(x)).map(x → x-1)
    ⇒ value: -3, log:
```

No Log info!

map to keep track of side-info.

```
* Loggable flatMap(Transformer<Integer, Loggable> t)
      Loggable l = t.transform(this.value);
      ↳ return new Loggable(l.value, this.log + l.log);
```

can be generic.
i.e. value ⇒ <T>.

⇓

```
Loggable incrWithLog(int x)
Loggable absWithLog(int x)
```

Cannot compose.

⇓

```
Loggable.of(-3).flatMap(x ⇒ incrWithLog(x))
               .flatMap(y → absWithLog(y))
```

implementer handles what happens when a function is passed in

⇒ value: 2, log: incr -3; abs -2;

---

Box.of(x)

$x \longrightarrow \boxed{x}$

box.map(x → f(x))

$\boxed{x} \rightarrow \boxed{f(x)}$

box.flatMap(x → $\boxed{f(x)}$)

main diff.

$\boxed{x} \rightarrow \boxed{f(x)}$

side-info

Maybe<T> : item might be missing

Lazy<T> : item is evaluated on demand.

Loggable<T> : item is logged.

Monad.

InfiniteList<T> : items in a Lazily-evaluated list

Array<T> : items in an array.

Box<T> : item in a box.

# Monad

## 1. Left Identity Law

Monad.of(x).flatMap(y → f(y))

is equivalent to

f(x).

Eg. Loggable.of(4).flatMap(x → incrWithLog(x)) — should not do anything weird

⇒ value: 5, log: incr 4;

incrWithLog(4) — same value & side-info.

⇒ value: 5, log: incr (4).

Monad.of(x).flatMap(y → $\boxed{f(y)}$)

$x \to \boxed{x} \to \boxed{f(y)}$

same as.

$x \to f(x)$.

2 method at least

* ⇒ of

* ⇒ flatMap.

\+ obeys the 3 laws.*

⇒ ∴ can do various operations on them.

⇒ order doesn't matter.

⇒ of doesn't change anything

## 2. Right Identity Law.

monad.flatMap(y → Monad.of(y)) — should not change anything about monad.

is equivalent to

monad

monad.flatMap(y → $\boxed{y}$)

$\boxed{x} \to \boxed{x}$

Eg. foo(3)

⇒ value: 4, log: abs 4; incr 3;

foo(3).flatmap(x → Loggable.of(x)) — same.

⇒ value: 4, log: abs 4; incr 3;

## 3. Associative Law

monad.flatMap(x → f(x)).flatMap(x → g(x))

is equivalent to

monad.flatMap(x → f(x).flatMap(x → g(x)))

Analogous to:

(A + B) + C

A + (B + C)

bar(-3)

⇒ smth...

Loggable.of(-3).flatMap(x → incrWithLog(x)).flatMap(x → absWithLog(x))

⇒ value: 2, log: abs -2; incr -3;

same as

Loggable.of(-3).flatMap(x → incrWithLog(x) .flatMap(x → absWithLog(x)))

monad.flatMap(x → $\boxed{f(x)}$.flatMap(y → $\boxed{g(y)}$)) vs monad.flatMap(x → $\boxed{f(x)}$).flatMap(y → $\boxed{g(y)}$)

$\boxed{x} \to \boxed{g(f(x))}$

$\boxed{x} \to \boxed{f(x)} \to \boxed{g(f(x))}$

# Functor

→ of

→ map.

→ obeys 2 laws.

functor.map(x → x) "identity law"

is just

functor.

functor.map(x → f(x)).map(x → g(x))

is just

functor.map(x → g(f(x))) just like in maths.

eg. A Box.

# Parallel & Concurrent Programming

Parallel → multiple processors/cores run things independent on each other

Concurrent → task will run on diff threads.

## Parallel Streams.

↳ allows us to use parallel computing.

```
boolean isPrime(int n)
↳ return IntStream.range(2, (int)Math.sqrt(n)+1)
                  .noneMatch(x -> n % x == 0)
```

↳ noneMatch: terminal op.

```
IntStream.range(2_030_000, 2_040_000)
    .filter(x -> isPrime(x))
```

⇒ doesn't do anything yet.

```
IntStream.range(2_030_000, 2_040_000)
    .filter(x -> isPrime(x)).forEach(Sys.out).
```

⇓ make it parallel.

```
IntStream.range(2_030_000, 2_040_000)
    .parallel()
    .filter(x -> isPrime(x))
    .forEach(System.out::println)
```

⇓ give diff order of numbers.

⇓ cannot guarantee the order since multi-threaded

```
IntStream.range(2_030_000, 2_040_000)
    .parallel()        → not waiting for another result to end.
    .filter(x -> isPrime(x))
    .sequential()      ⇒ order them.
    .forEach(System.out::println)
```

⇒ order doesn't matter if use count()

A lot of problem req. other things to come back.

## Interference : one of the stream operations modifies the source of the stream during execution of terminal operation

```
List<String> list = new ArrayList<>(List.of("Luke", "Leia", "Han"));

list.stream().forEach(sysout)
```

don't want to change anything in it i.e. interfere

⇒ Luke
Leia
Han

```
list.stream().peek(name -> { name.equals("Han") ? list.add("chewie"); }).forEach(i->{})
```

⇓
ConcurrentModificationException
⇓
cannot interfere with its own element.

## Stateful vs Stateless

↳ result depends on any state that might change during the execution of the stream

↳ eg. generate, map ⇒ depends on state of input

⇒ parallelising this might lead to incorrect output.

⇒ additional work might be needed to ensure state updates are visible to all parallel subtasks.

## Side-Effects : might lead to incorrect results in parallel execution.

List<Integer> list = odd no. from 1-19.

```
list.parallelStream().filter(x -> isPrime(x)).forEach(sysout).
```

13
11
17
19
5
7
3
1

```
list.parallelStream().filter(x -> isPrime(x)).forEach(x -> result.add(x))
```

add to non-thread safe
⇒ get diff result.

Use a thread-safe datastructure.
or Collectors.

ie. `list.parallelStream().filter(x -> isPrime(x)).collect(Collectors.toList())`

## Associativity

Parallelising reduce → combiner
↘ accumulator

meets the 3 rules below.

```
Stream.of(1,2,3,4)
    .reduce(1, (x,y) -> x*y,
               (x,y) -> x*y);
```

will get back

→

```
Stream.of(1,2,3,4).parallel()
    .reduce(1, (x,y) -> x+y,
               (x,y) -> x+y);
```

⇓
might get diff numbers

doesn't meet the rules.

1. Combiner.apply(identity, i) must be equal to i.

   (x,y)-> x+y. ⇒ not work.
   x*y ⇒ will work.

2. Combiner and accumulator must be associative.
   ⇒ work independent of order they come

3. Combiner and accumulator must be compatible.
   ⇒ ie. combiner.apply(u, accumulator.apply(identity, t)) ≡ accumulator.apply(u, t).

## Performance?, ⇒ parallel is faster 100%.

↓
only if it should be parallelise ⇒ only some makes sense.

⇒ cost of setting up thread takes longer than task to compute?