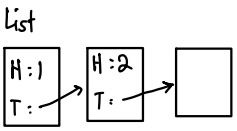



```
class EagerList<T> {
    private T head;
    private EagerList<T> tail;
    ⋮
}
```

↓

EmptyList

Methods:

```
static <T> EagerList<T> generate(T t, int size) {
    // generate(1,0); ⇒ returns EmptyList.
    if (size == 0)                          // base case.
        return empty();
    else
        return new EagerList<>(t, generate(t, size-1));
}
```

finite in size

```
static <T> EagerList<T> iterate(T init, BooleanCondition<? super T> cond, Transformer<? super T, ? ext T> op) {
    // iterate(1, i -> i<3, i->i+1)
    ⇒ a list 1, 2.
    if (!cond.test(init))                   // base case.
        return empty();
    return new EagerList<>(init, iterate(op.transform(init), cond, op));
}
```

must be a bound

apply to init

```
<R> EagerList<R> map(Transformer<? super T, ? ext R> mapper) {
    return new EagerList<>(mapper.transform(this.head()), this.tail().map(mapper));
}
```

↘ base case handled for empty. ↗

```
EagerList<T> filter(BooleanCondition<? super T> cond) {
    if (cond.test(this.head()))
        return new EagerList<>(this.head(), this.tail().filter(cond));
    else
        return this.tail().filter(cond);
}
```

Delayed Evaluation.

↳ Take advantage of Producer functional interface.

↳ InfiniteList.

↳ wrap head and tail w Producer Object.

↳ their evaluation delayed.

↳ Producer<T> head.
Producer<InfiniteList<T>> tail; } delayed

↳ thinking about type constraint

↳ no empty Stream abstraction.

↳ how to handle?

Methods.

```
<T> InfiniteList<T> generate(Producer<T> producer)
        return new InfiniteList<>(producer, () -> generate(producer));

<T> InfiniteList<T> iterate(T init, Transformer<T,T> next)
        // iterate(1, x -> x+1)
        return new IntList<>(() -> init, () -> iterate(next.transform(init), next));

<R> IntList<R> map(Transformer<? super T, ? ext R> mapper)
        return new IntList<>(() -> mapper.transform(this.head()), () -> this.tail().map(mapper));
                                                    all delayed!

IntList<T> filter(BooleanCondition<? super T> cond)
        if (cond.test(this.head())
            return new IntList<>(this.head, () -> this.tail().filter(cond));
        else
            return this.tail().filter(cond);
                    not delayed!
                    head is immediately produced!
                    => eager!
```

InfList<T> filter(BooleanCondition<? super T> cond)

```
Produces<T> tempHead = () -> {
    if (cond.test(this.head()))
        return this.head();
    else
        return null;
};
```

"demo"

alt. build an abstraction to handle if elements is present ⇒ Maybe<T>?

bad idea ⇒ stop us from having an inf list w/ value null.

```
return new InfList<>(tempHead, () -> this.tail().filter(cond));
```

⇓

change how head() and tail() work.

⇓

check if this.head.produce == null

⇒ tail.produce().head()

↳ get next elem in list.

|| for head().

⇒ grab next elem.

↳ this.tail.produce.tail()

↳ next tail

|| for tail()

InfList.iterate(0, x->x+2) . map(x->x+1) . map(x->x*2) . head()

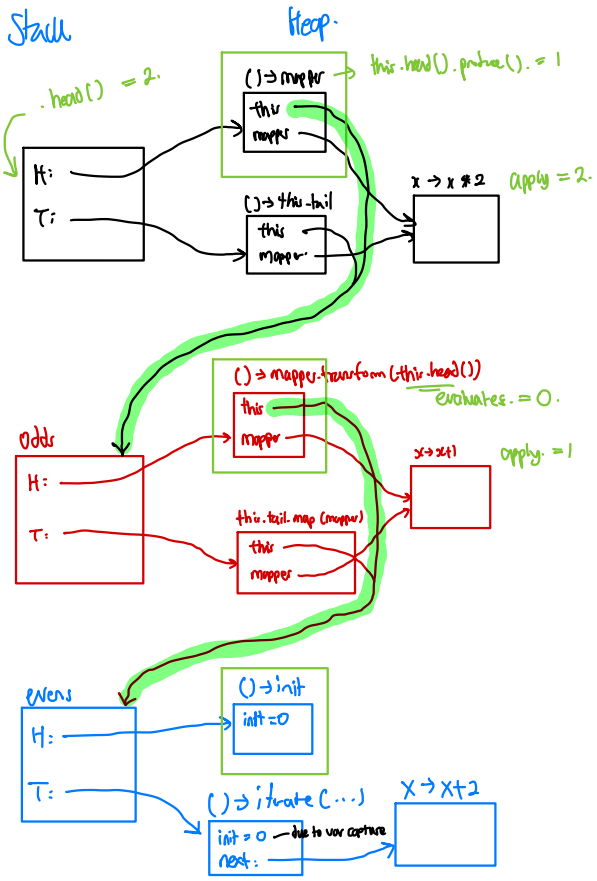
Odds

evens

new InfList object.

No computation!

Chain reaction!



Function

BooleanCondition<T>	→	Predicate<T>
Producer<T>	→	Supplier<T>
Transformer<T,R>	→	Function<T,R>
Transformer<T,T>	→	UnaryOp<T>
Combiner<S,T,R>	→	BiFunction<S,T,R>
Maybe<T>	→	Optional<T>
Lazy<T>	→	??
InfiniteList<T>	→	Stream<T>

Stream

A stream pipeline:

↳ operation:

1. data source → eg. of, generate, iterate → also Arrays::stream
2. intermediate ops. → eg. map, filter, count.
3. terminal ops. → evaluation. reduce, noneMatch, allMatch, anyMatch.
   ↘ forEach.
   ↓
   takes in Consumer.

↳ eg.

```
Stream.of(1,2,3).forEach(x -> System.out(x))
```

for each element do

Truncate Stream?

↳ takeWhile(Predicate<? super T>)
Longest stream where elements matches the predicate.

↳ limit(maxSize)

↳ distinct()
- will only give you a stream containing a distinct elements.

↳ sorted()

Stateful intermediate ops. ⇒ take into account of some states ⇒ keep track of all elems in stream

```
"".lines()
```
⇒ Convert to stream.

FlatMap.

↳ transforms every elem in the stream into another stream ⇒ results are flatten and concatenated together.

Memorization?

Peek
↳ looks into Stream.

Streams only evaluated once! i.e.

↳ cannot reevaluate again

```
Stream.iterate(0, x->x+1).peek(System.out).takeWhile(x -> x<5).forEach(x->{});
```

1 did not take into account of this.

```
0
1
2
3
4
5
```