CS2030 Lecture 10

The Art of Being Lazy: 2

Terence Sim

New abstraction

freeze(<expr>) → <expr> is not evaluated now, but postponed indefinitely. <expr> is wrapped in a *thunk.*

freeze is not a normal function. Its argument is not evaluated immediately.

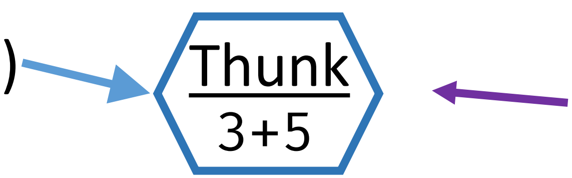
Its secret will be revealed shortly.

compare: int foo(int n) { … }

foo(3+5) → foo(8)

The argument (which is an expression) is eagerly evaluated BEFORE foo is called. But,

freeze(3+5)



The expression 3+5 is not evaluated but frozen in a thunk

New abstraction

`freeze(freeze( 3+5 ))` → **Thunk** `freeze(3+5)`

`thaw(thunk)` → forces the evaluation of the `<expr>` inside the thunk.

`thaw(freeze(freeze( 3+5 ))` → **Thunk** `3+5`

`thaw(thaw(freeze(freeze( 3+5 )))` → 8

Lazy Lists

```
public class LazyList<T> {
   private final T head;
   private final Thunk(LazyList<T>) tail;
   . . . }
```

Constructors:

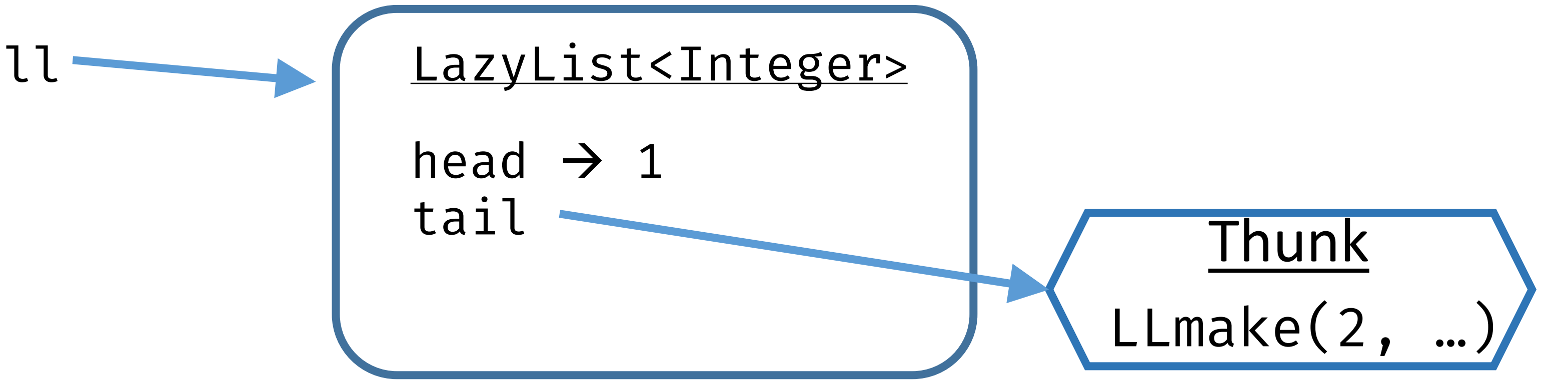
`LLmake(a, b) ≡ new LazyList<>(a, freeze(b))`

Note that a is eagerly evaluated, but b is frozen.

`LLmake` is syntactic sugar, and not a normal function.

Lazy Lists: example

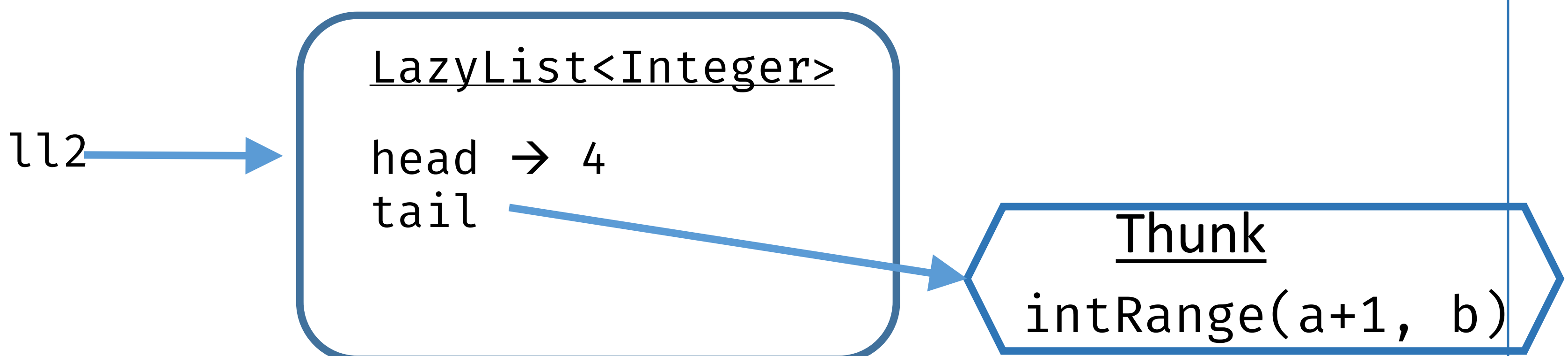
```
LazyList<Integer> ll =
      LLmake(1, LLmake(2, LazyList.makeEmpty()));
```



Integers from a (inclusive) to b (exclusive)

```
LazyList<Integer> intRange(int a, int b) {
    if (a >= b)
        return LazyList.makeEmpty();
    return LLmake(a, intRange(a+1, b)); }

var ll2 = intRange(4, 12);
```



map: apply function to every element

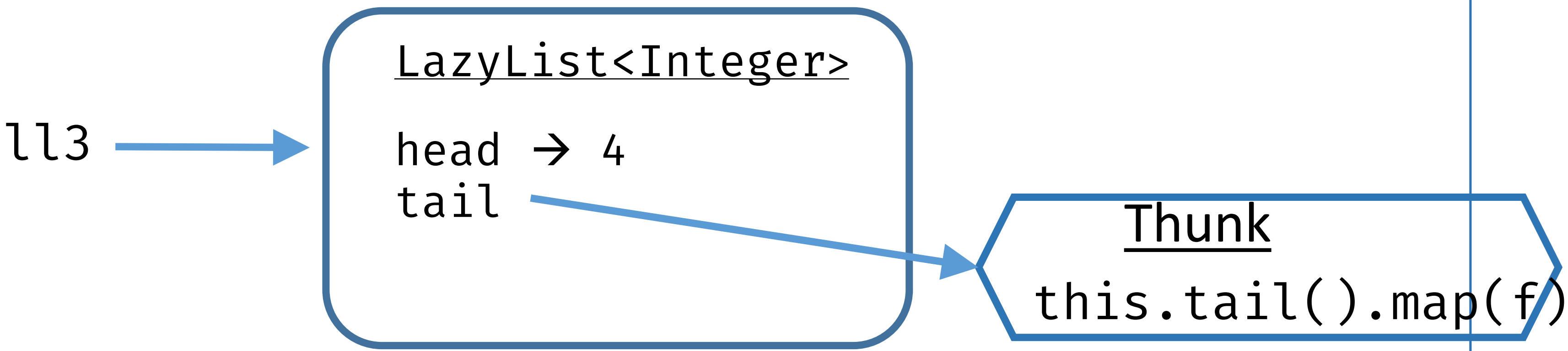
```
public <R> LazyList<R> map(Function<T,R> f) {
    if (this.isEmpty())
        return LazyList.makeEmpty();
    else
        return LLmake(f.apply(this.head()),
                      this.tail().map(f));
}
```

Note the use of recursion: we recursively apply `map` to the `tail` of the list.

But this is frozen because it is the 2nd argument of `LLmake`.

Compute the squares of 2,3,4,5

```
var ll3 = intRange(2,6).map(x -> x*x);
```

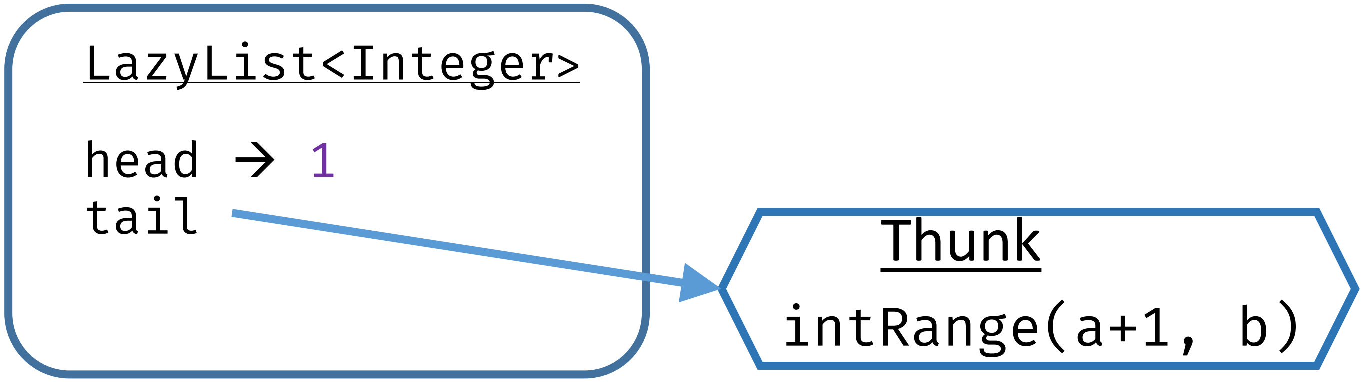


`filter`: keep elements that satisfy predicate

```
public LazyList<T> filter(Predicate<T> pred) {
    if (this.isEmpty())
        return this;
    else if (pred.test(this.head()))
        return LLmake(this.head(),
                      this.tail().filter(pred));
    else
        return this.tail().filter(pred);  }
```

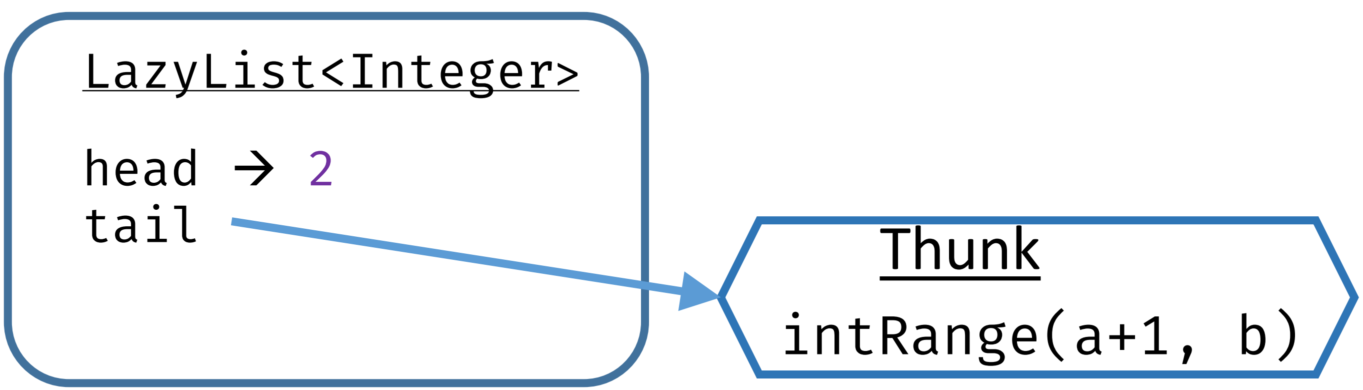
`filter`: keep elements that satisfy predicate

```
intRange(1,21).filter(x-> x%2==0)
```



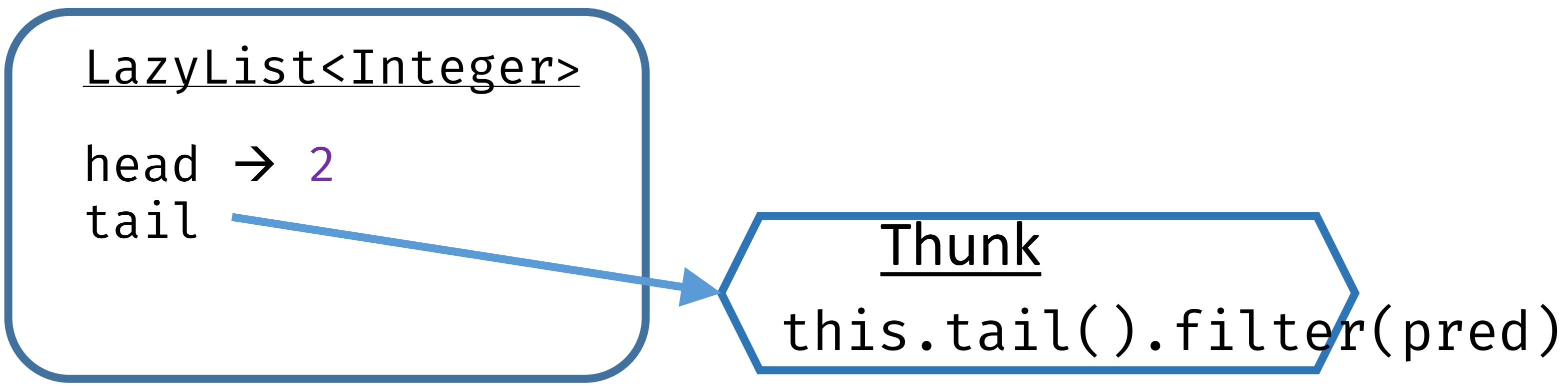
`pred.test(1) → false`

so, `intRange(2, 21)` will be called.



`filter`: keep elements that satisfy predicate

```
pred.test(2) → true
so, return LLmake(2, this.tail().filter(pred))
```



elementWiseCombine

Demo

```
public LazyList<T> elementWiseCombine> other,
            BinaryOperator<T> binOp) {
  if (this.isEmpty() || other.isEmpty())
    return LazyList.makeEmpty();
  else
    return LLmake(binOp.apply(this.head(),
                                    other.head()),
                 this.tail()
            .elementWiseCombine(other.tail(),
                                    binOp)); }

var ll = intRange(1, 11);
var twos = ll.map(x->2);
ll.elementWiseCombine(twos, (x,y)->x+y).print();
=> (* 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, *)
```

reduce

```
T reduce(T identity, BinaryOperator<T>
                               accumulator) {
    if (this.isEmpty())
      return identity;
    else
      return accumulator.apply(this.head(),
                 this.tail().reduce(identity,
                                    accumulator));
}
```

This combines all the elements, two at a time, via the generic accumulator. The resulting value is the “aggregate” of all elements.

eg: add the squares of 1, 2, ..., 20

```
var result = LazyList.intRange(1,21)
             .map(x->x*x)
             .reduce(0, (x,y)->x+y);

System.out.println(result);
=> 2870
```

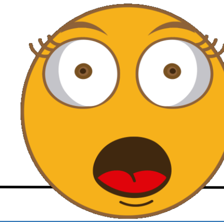
This computes:

$$1^2 + 2^2 + \ldots + 19^2 + 20^2$$

eg2: factorial may be expressed as

```
int factorial(int n) {
        return LazyList.intRange(1, n+1)
            .reduce(1, (x,y)-> x*y);
}
```

Infinite LazyLists!



Demo

```
LazyList<Integer> integersFrom(int n) {
     return LLmake(n, integersFrom(n+1)); }
integersFrom(1).limit(50).print();


=> (* 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27,
28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
41, 42, 43, 44, 45, 46, 47, 48, 49, 50, *)
```

```
LazyList<T> limit(long maxSize) {
   if (maxSize==0)
         return LazyList.makeEmpty();
   return LLmake(this.head(),
                 this.tail().limit(maxSize - 1));   }
```

Prime numbers

Of course, one way to check if a number n is prime is to see if n can be found in a list, L, of primes. To generate L, an ancient method called the Sieve of Eratosthenes may be used:

1. Start by listing all integers greater than 1. Call this list C. Also, let L be an empty list.
2. Take the first number $p = 2$ in C, and add it to L. This is the first prime.
3. In C, cross out all multiples of p.
4. Let p be the next uncrossed number in C. This is the next prime. Add it to L, and repeat from Step 3.

	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~
~~21~~	~~22~~	23	~~24~~	~~25~~	~~26~~	~~27~~	~~28~~	29	~~30~~

Prime sieve

Demo

```
LazyList<Integer> sieve(LazyList<Integer> s) {
   return LLmake(s.head(),
                 sieve(s.tail()
                       .filter(x->
                               x%s.head()!=0)));
}


var primes = sieve(integersFrom(2));
primes.limit(50).print();


=> (* 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101,
103, 107, 109, 113, 127, 131, 137, 139, 149, 151,
157, 163, 167, 173, 179, 181, 191, 193, 197, 199,
211, 223, 227, 229, *)
```

Another way to get integers

Demo

```
LazyList<Integer> integers;

integers = LLmake(1, integers.map(x->x+1));
```

Fibonacci numbers

Demo

The infinite sequence:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

The next number is the sum of the previous two.

```
LazyList<Integer> fib;
fib = LLmake(0,
        LLmake(1,
            fib.elementWiseCombine(fib.tail(),
                                    (x,y)->x+y)));
```

0	1	1	2	3	5			← fib
		0	1	1	2	3	5	← fib
		1	1	2	3	5		← fib.tail()

The BIG Reveal!

C macros to effect syntactic sugar

```
#define freeze(x)  ( ()->(x) )
#define LLmake(a, b) new LazyList<>((a),
                                        freeze(b))
#define Thunk(T)  Supplier<T>
```

Use a nullary lambda to freeze an expression:

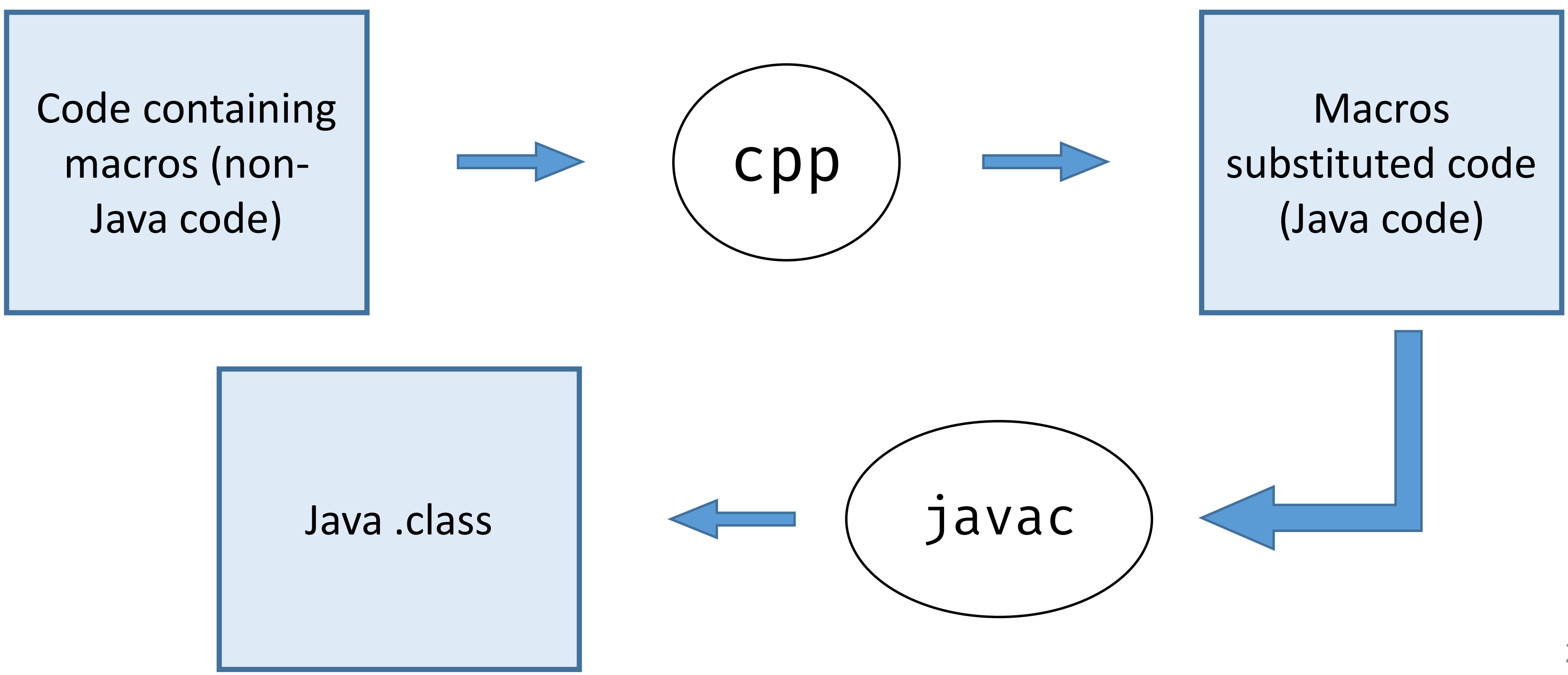
```
freeze(3+5)  ≡  ( () -> (3+5) )
```

```
T thaw(Thunk(T) ice) {
        return ice.get();
}
```

Using C pre-processor to convert macros

```
#define freeze(x)  ( ()->(x) )
#define LLmake(a, b) new LazyList<>((a),
                                  freeze(b))
#define Thunk(T)  Supplier<T>
```

This process is done by the bash script `jpp`



jpp script

```
PROC_DIR=processed
TMPFILE=$(mktemp /tmp/jpp.XXXX)
FNAME=$1

cat <<EOF > $TMPFILE
#define freeze(x)  ( ()->(x) )
#define LLmake(a, b) new LazyList<>((a), freeze(b))
#define Thunk(T)  Supplier<T>
EOF

cat $TMPFILE $FNAME | cpp -P - -o $PROC_DIR/$FNAME
javac -classpath $PROC_DIR -d $PROC_DIR $PROC_DIR/$FNAME
rm -f $TMPFILE
```

Let’s see an example

Before `jpp`

```
LazyList<Integer> integersFrom(int n) {
    return LLmake(n, integersFrom(n + 1));
}
```

After `jpp`

```
LazyList<Integer> integersFrom(int n) {
   return new LazyList<>((n), (()->(integersFrom(n +
1)) ));
}
```

Example 2

Before `jpp`

```
LazyList<Integer> sieve(LazyList<Integer> s) {
  return LLmake(s.head(),
                    sieve(s.tail().filter(x->
      (x%s.head()) != 0 )));
}
```

After `jpp`

```
LazyList<Integer> sieve(LazyList<Integer> s) {
  return new LazyList<>((s.head()), (()->
(sieve(s.tail().filter(x-> (x%s.head()) != 0 )))
));
}
```

Memoization

In computing, **memoization** is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again.

https://en.wikipedia.org/wiki/Memoization

Memo.java

```
public class Memo<T> {
    private boolean hasBeenRun;
    private T value;
    private Supplier<T> supplier;

    private Memo(Supplier<T> s) {
        this.hasBeenRun = false;
        this.value = null;
        this.supplier = s;  }

    public static <T> Memo<T>  make(Supplier<T> s) {
        return new Memo<>(Objects.requireNonNull(s));
    }
```

Memo.java

```
    public T get() {
        if (!hasBeenRun) {
            this.hasBeenRun = true;
            this.value = this.supplier.get();
        }
        return this.value;
    }
}
```

First time: invoke `supplier` to evaluate the expression, and store the result in `value`

Subsequent times: just return the `value`

Change the macros to use Memo

```
#define freeze(x)  Memo.make( ()->(x) )
#define LLmake(a, b) new LazyList<>((a), freeze(b))
#define Thunk(T)  Memo<T>
```

Let’s time the difference

Demo

```
timeIt(()-> fib.get(10), "fib 10");
timeIt(()-> fib.get(10), "fib 10");
timeIt(()-> fib.get(20), "fib 20");
timeIt(()-> fib.get(15), "fib 15");
```

```
fib 10 took 24 ms
fib 10 took 0 ms
fib 20 took 1 ms
fib 15 took 0 ms
```

Tracking #get

```
Memo Get Table:
head: 1, tail: thunk!Memo@546a03af: count-> 6
head: 1, tail: thunk!Memo@7b3300e5: count-> 6
. . .
head: 1597, tail: thunk!Memo@256216b3: count-> 3
head: 2, tail: thunk!Memo@721e0f4f: count-> 6
head: 21, tail: thunk!Memo@725bef66: count-> 6
head: 233, tail: thunk!Memo@649d209a: count-> 4
. . .
head: 55, tail: thunk!Memo@6e3c1e69: count-> 6 (fib 10)
head: 610, tail: thunk!Memo@357246de: count-> 4 (fib 15)
head: 6765, tail: thunk!Memo@16c0663d: count-> 1 (fib 20)
```

java.util.stream.Stream<T>

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html

LazyList<T> vs Stream<T>: similarities

Both are lazily evaluated

Both can be finite or infinite

Both have these functions:

map, flatmap, filter, forEach, empty, limit, reduce

LazyList<T> vs Stream<T>: differences

LazyList<T>	Stream<T>
Multiple use	Single use only stream is used up when terminal operation is applied
Can be memorized	No memorization
Can retrieve element by index	Cannot retrieve by index
Has Constructor, no converters	No constructor, but converters `List.stream()`, `String.lines()`, `String.chars()`, `Scanner.tokens()`

IntStream

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/IntStream.html

```
OptionalDouble result = IntStream.range(1,21)
            .map(x->x*x).average();

result.ifPresent(System.out::println);
=> 143.5
```

This computes: $\frac{1}{20}(1^2 + 2^2 + \ldots + 20^2)$

2 categories of stream operations:

Intermediate: This produces another lazily evaluated stream from the given stream, eg: map, flatmap, filter, limit

Terminal: This forces the eager evaluation of all the elements in the stream, and uses it up, eg: reduce, sum, collect, allMatch

Summary

Lazy Evaluation means a computation is postponed until the value is needed.

Advantages: avoids unnecessary computation, amortizes time complexity, allows infinite data structures

Lazy Evaluation may be achieved using nullary lambda expressions (ie. Supplier<T>)

LazyList<T> and Stream<T> are similar in many ways, and yet are different in important ways.

A Tutorial on Java Stream:

http://tutorials.jenkov.com/java-functional-programming/streams.html