15 Design Pattern: Laziness

Consider the following problem: I'm asking you to take the list of the first 100 natural numbers (starting from 1), square each of them, add the resulting list of integers to itself three times, subtract 1 from every element of the resulting list, and then give me the 5th element of the result. You could implement these operations quite easily using any implementation of lists, and get the result equally easily. But you'll be doing an awful lot of work, considering that all you really want is the 5th element of the resulting list.

If you think about it for a second, all you really need to do is take the 5th element of the initial list (that is, 5), square it to get 25, add it to itself three times to get 75, and then subtract 1 to get 74. No waste work, no need to worry about applying expensive list operations to elements of the list that we'll never need.

So how can we implement lists, or in general data structures, where we avoid doing the work to compute values that we will end up not needing? The general approach for doing this is called *lazy construction*, or just *laziness*, and basically says that we do not do any work when we construct objects, we do all the work when we compute from objects (that is, observe values). The opposite of lazy construction is *eager construction*.

To get lazy construction for an ADT, the basic trick is to make all operations (except for operations that extract data from the ADT) into creators. Creators do not do any work. Then it's just a matter of defining the remaining operations that extract data, and those will do the work, for each of the creators we define. Let's look at an example.

15.1 Lazy Mapping for Lists

To illustrate the difference between lazy construction and eager construction, consider the List implementation with a `map()` operation from last time. Suppose we have a mapping transformation that does *a lot* of work. For the sake of simplicity, here, I'm taking the `Double` mapping transformation, but modified so that it takes a long time to compute (about ten seconds):

```
public class Double implements MappingFunction<Integer> {
  private Double () {}

  public static Double function () { return new Double(); }
```

```
  public Integer apply (Integer a) {
    try { Thread.sleep(10000); } catch (Exception e) {}
    return 2 * a; }
}
```

If we construct a sample list, map the doubling transformation over it, and print its fourth element, as follows:

```
List<Integer> lst =
  List.cons(1,List.cons(2,List.cons(3,List.cons(4,List.<Integer>empty()))));
List<Integer> lst_doubled = lst.map(Double.function());
System.out.println("Fourth element of doubled list = " +
                       lst_doubled.rest().rest().rest().first());
```

the result is printed, but after about 40 seconds:

```
Fourth element of doubled list = 8
```

It takes a long time to compute because the mapping transformation spends ten seconds per element, and the whole list gets transformed before the fourth element is accessed.

Consider the following alternate List ADT where `map()` is a lazy operation — that is, we make it a creator:

```
CREATORS     <A> List<A> empty ()
             <A> List<A> cons (A, List<A>)
             <A> List<A> map (MappingFunction<A>, List<A>)

ACCESSORS    boolean isEmpty ()
             A first ()
             List<A> rest ()
             List<A> append (List<A>)
             String toString ()
```

with specification:

```
empty().isEmpty() = true
cons(v,L).isEmpty() = false
map(mf,L).isEmpty() = L.isEmpty()

cons(v,L).first() = v
map(mf,L).first() = mf.apply(L.first())
```

```
  cons(v,l).rest() = L
  map(mf,L).rest() = map(mf,L.rest())

  empty().append(M) = M
  cons(v,L).append(M) = cons(v,L.append(M))
  map(mf,L).append(M)
    = empty()     if L.isEmpty()=true
    = cons(mf.apply(L.first()),map(mf,L.rest).append(M))  otherwise

  empty().toString() = ""
  cons(v,L).toString() = v + " " + L
  map(mf,L).toString() = mf.apply(L.first()) + " " + map(mf,L.rest()).toString()
```

Notice that the specification tells you when the work gets done — when we take the `first()` of a list. (Also, when we `append()`, or when we convert to a string.)

The Specification design pattern gives us the obvious implementation:

```
public abstract class List<A> {

  public static <A> List<A> empty () {
    return new EmptyList<A>();
  }

  public static <A> List<A> cons (A i, List<A> l) {
    return new ConsList<A>(i,l);
  }

  public static <A> List<A> map (MappingFunction<A> mf, List<A> l) {
     return new MapList<A>(mf,l);
  }

  public abstract boolean isEmpty ();
  public abstract A first ();
  public abstract List<A> rest ();
  public abstract List<A> append (List<A> l);
  public abstract String toString ();
}



class EmptyList<A> extends List<A> {
 public EmptyList () {}
```

```
  public boolean isEmpty () {
    return true;
  }
  public A first () {
    throw new Error ("first() on an empty list");
  }
  public List<A> rest () {
    throw new Error ("rest() on an empty list");
  }
  public List<A> append (List<A> l) {
      return l;
  }
  public String toString () {
      return "";
  }
}



class ConsList<A> extends List<A> {
  private A first;
  private List<A> rest;

  public ConsList (A f, List<A> r) {
    first = f;
    rest = r;
  }

  public boolean isEmpty () {
    return false;
  }
  public A first () {
    return first;
  }
  public List<A> rest () {
    return rest;
  }
  public List<A> append (List<A> l) {
      return List.cons(first,rest.append(l));
  }
  public String toString () {
```

```
    return first.toString() + "   " + rest.toString();
  }
}



class MapList<A> extends List<A> {
   private MappingFunction<A> mf;
   private List<A> lst;

   public MapList (MappingFunction<A> mf, List<A> l) {
      this.mf = mf;
      this.lst = l;
   }

   public boolean isEmpty () {
      return lst.isEmpty();
   }
   public A first () {
      return mf.apply(lst.first());
   }
   public List<A> rest () {
      return List.map(mf,lst.rest());
   }
   public List<A> append (List<A> l) {
      if (this.isEmpty())
         return l;
      return List.cons(mf.apply(lst.first()),List.map(mf,lst.rest()).append(l));
   }
   public String toString () {
      if (this.isEmpty())
         return "";
      return mf.apply(lst.first()).toString() + "   " + List.map(mf,lst.rest()).toString();
   }
}
```

If we try this implementation using the doubling transformation above, we get a much more efficient execution. Again, we construct a sample list, map the doubling transformation over it, and print its fourth element, as follows:

```
  List<Integer> lst =
    List.cons(1,List.cons(2,List.cons(3,List.cons(4,List.<Integer>empty()))));
```

```
List<Integer> lst_doubled = List.map(Double.function(),lst);
System.out.println("Fourth element of doubled list = " +
                       lst_doubled.rest().rest().rest().first());
```

the result is printed now after only ten seconds:

```
Fourth element of doubled list = 8
```

It is faster because we only apply the transformation when we access the element. If you have doubts, use the specifications for this ADT and the one from last time to see the difference in execution (using algebraic reasoning).

15.2 Example: Infinite Streams

Laziness is very powerful. In particular, it lets us work easily with infinite data. The classic example of infinite data is streams, which you can think of infinite lists. Of course, you cannot simply represent an infinite list directly by listing all its elements. Instead, a stream can be thought of simply as a "promise" to deliver its elements, if you ask for them. If you ask for the first 10 elements of the stream, it will compute them, and then give them to you. If you ask for the 200th element of the stream, it will compute it and give it to you. Until you ask for it, though, it is not explicitly represented. We can also create new streams from old streams, and here again, we do not do any work at creation time, only when we ask for elements of the stream.

We implement the following interface for polymorphic streams, `Stream<A>`:

```
CREATORS     <A> Stream<A> constant (A)
             Stream<Integer> intsFrom (int)
             <A> Stream<A> cons (A, Stream<A>)
             Stream<Integer> sum (Stream<Integer>, Stream<Integer>)

ACCESSORS    A head ()
             Stream<A> tail ()
```

where `constant(v)` creates a stream delivering always the same value `v`, `intsFrom(v)` creates a stream of all integers starting from a given number `v`, `cons(v,s)` creates a stream that starts with a given value `v` before delievring the elements of stream `s`, and `sum(s1,s2)` creates a stream of elements that are the pointwise sums of the elements of `s1` and `s2`. Here is the specification of the two operations `head` and `tail` that return the first element of a stream, and the stream obtained by removing the first element. Note that all the work is done in those two operations:

```
constant(i).head() = i
intsFrom(i).head() = i
cons(a,S).head() = a
sum(S1,S2).head() = S1.head() + S2.head()

constant(i).tail() = constant(i)
intsFrom(i).tail() = intsFrom(i+1)
cons(a,S).tail() = S
sum(S1,S2).tail() = sum(S1.tail(),S2.tail())
```

It is completely straightforward to implement the above ADT using the Specification design pattern:

```
public abstract class Stream<A> {

   public static <B> Stream<B> constant (B i) {
      return new StreamConstant<B>(i);
   }
   public static Stream<Integer> intsFrom (int i) {
      return new StreamIntsFrom(i);
   }
   public static <B> Stream<B> cons (B i, Stream<B> s) {
      return new StreamCons<B>(i,s);
   }
   public static Stream<Integer> sum (Stream<Integer> s1, Stream<Integer> s2) {
      return new StreamSum(s1,s2);
   }

   public abstract A head ();
   public abstract Stream<A> tail ();
}


class StreamConstant<A> extends Stream<A> {
   private A cnst;

   public StreamConstant (A i) {
      cnst = i;
   }

   public A head () {
      return cnst;
   }
```

```
    public Stream<A> tail () {
        return Stream.constant(cnst);
    }
}


class StreamIntsFrom extends Stream<Integer> {
    private int cnst;

    public StreamIntsFrom (int i) {
        cnst = i;
    }

    public Integer head () {
        return cnst;
    }
    public Stream<Integer> tail () {
        return Stream.intsFrom(cnst+1);
    }
}


class StreamCons<A> extends Stream<A> {
    private A first;
    private Stream<A> rest;

    public StreamCons (A i, Stream<A> s) {
        first = i;
        rest = s;
    }

    public A head () {
        return first;
    }
    public Stream<A> tail () {
        return rest;
    }
}


class StreamSum extends Stream<Integer> {
    private Stream<Integer> first;
```

```
    private Stream<Integer> second;

    public StreamSum (Stream<Integer> s1, Stream<Integer> s2) {
        first = s1;
        second = s2;
    }

    public Integer head () {
        return first.head() + second.head();
    }
    public Stream<Integer> tail () {
        return Stream.sum(first.tail(),second.tail());
    }
}
```

Adding a new operation to a lazy ADT corresponds to adding a new creator for that operation, and defining how the observers work on that creator. For instance, suppose we want to have a `filter()` operation on streams, that keep only those elements that satisfy a certain predicate. A predicate is a function that takes an element of the stream and that returns a Boolean true or false.

```
CREATOR     <A> Stream<A> filter (PredicateFunction<A>, Stream<A>)
```

As far as implementing predicates, we use the same technique we used when working with map and reduce operations, and define an interface for predicates:

```
public interface PredicateFunction<A> {
  public boolean apply(A v);
}
```

Thus, the only thing we can do with a predicate is apply it to a value. Here is an example of a predicate over integers, that returns true if and only if an integer is even:

```
public class Even implements PredicateFunction<Integer> {
  private Even () {}

  public static Even function() {
    return new Even();
  }

  public boolean apply (Integer v) {
    return (v % 2 == 0);
```

```
  }
}
```

As a slightly different example, consider the following predicate, that checks if a number is not divisible by a given number (supplied when we construct the predicate).

```
public class NotDivisibleBy implements PredicateFunction<Integer> {
  private int divisor;
  private NotDivisibleBy (int i) {
    divisor = i;
  }

  public static NotDivisibleBy function (Integer i) {
    return new NotDivisibleBy(i);
  }

  public boolean apply (int v) {
    return (!(v % divisor == 0));
  }
}
```

The specification for `filter()` is as follows:

```
  filter(pf,s).head()
     = s.head()                      if pf.apply(s.head())=true
     = filter(pf,s.tail()).head()    otherwise
  filter(pf,s).tail()
     = filter(pf,s.tail())           if pf.apply(s.head())=true
     = filter(pf,s.tail()).tail()    otherwise
```

Think about this for a bit. Note also that this specification tells us that something like `filter(false,s).head()`, where `false` is a predicate that always returns false, is not equal to anything — in practice, this means that the implementation is free to do whatever it wants, and in the implementation below, it will get stuck in an infinite loop. We add the creator to the `Stream<A>` class:

```
public abstract class Stream<A> {
  ...

  public static <B> Stream<B> filter (PredicateFunction<B> pf, Stream<B> s) {
     return new StreamFilter<B>(pf,s);
  }
```

```
  ...
}
```

We easily implement the concrete class corresponding to the `filter()` creator:

```
class StreamFilter<A> extends Stream<A> {
    private Stream<A> underlying;
    private PredicateFunction<A> predicate;

    public StreamFilter (PredicateFunction<A> pf, Stream<A> s) {
        underlying = s;
        predicate = pf;
    }

    public A head () {
        if (predicate.apply(underlying.head()))
            return underlying.head();
        else
            return Stream.filter(predicate,underlying.tail()).head();
    }
    public Stream<A> tail () {
        if (predicate.apply(underlying.head()))
            return Stream.filter(predicate,underlying.tail());
        else
        return Stream.filter(predicate,underlying.tail()).tail();
    }
}
```

Let's try out some sample streams. Let's define a function to print the first n elements of a stream of integers:

```
  public static <A> void printFirstN (Stream<A> s, int n) {
    Stream<A> temp = s;
    for (int i = 0; i < n; i++) {
      System.out.print(temp.head() + "  ");
      temp = temp.tail();
    }
    System.out.println();
  }
```

We can now try out something such as:

```
Stream<Integer> s1 = Stream.intsFrom(1);
Stream<Integer> s2 = Stream.filter(Even.function(),s1);

printFirstN(Stream.sum(s1,s2),20);
```

This prints the first 20 elements of the stream obtained by summing the stream starting from 1 (i.e., `1 2 3 4 5 ...`) and the stream starting from 1 where we've removed all the non-even elements (i.e., `2 4 6 8 10 ...`). This yields:

```
3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48 51 54 57 60
```

Make sure you understand what is happening.

As a cute example, we can try to compute, using (a variant of) the Sieve of Eratosthenes, the stream of all prime numbers. The sieve computes the list of prime numbers by essentially starting with all integers from 2 on, and then keeping 2 and removing all multiples of 2, then moving to the next unremoved integer (3), keeping it and removing all multiples of 3, moving to the next unremoved integer (5), keeping it and removing all multiples of 5, and so on. You can convince yourself that what you are left with is the stream of all prime numbres.

Here is the operation `sieve()`, expressed as a constructor:

```
CREATOR      <A> Stream<A> sieve (Stream<A>)
```

with specification:

```
sieve(s).head() = s.head()
sieve(s).tail() = sieve(filter(NotDivisibleBy.function(s.head()),s.tail()))
```

Again, we add the creator to the `Stream<A>` class:

```
public abstract class Stream {
 ...

 static Stream<Integer> sieve (Stream<Integer> s) {
    return new StreamSieve(s);
 }

 ...
}
```

and implement the corresponding concrete subclass:

```
class StreamSieve extends Stream<Integer> {
   private Stream<Integer> stream;

   public StreamSieve (Stream<Integer> s) {
      stream = s;
   }

   public Integer head () {
      return stream.head();
   }
   public Stream<Integer> tail () {
      Stream<Integer> multipleRemoved =
         Stream.filter(NotDivisibleBy.function(stream.head()), stream.tail());
      return Stream.sieve(multipleRemoved);
   }
}
```

We can now compute the stream of prime numbers:

```
Stream primes = Stream.sieve(Stream.intsFrom(2));
```

By looking at the first N elements of the stream, we can get all the primes we want, such as:

```
printFirstN(primes,20);
```

with output:

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71
```

Note, however, that this is far from being an efficient way for computing prime numbers, as you can tell immediately by trying to execute `printFirstN(primes,100);`.