Object-Oriented Design — Lecture 17
CS 3500 Fall 2010 (Pucella) — Tuesday, Nov 16, 2010

17 Design Pattern: Adapters

One of the design pattern that we have been using — although I have not really called it that — to provide sequential access to the elements of an aggregate structure is the *Stream* design pattern, using the following interface;

```
trait Stream[A] {

  def hasElement ():Boolean
  def head ():A
  def tail ():Stream[A]

  // Derived operations

  def print ():Unit = {
    if (hasElement()) {
      println("  " + head());
      tail().print()
    }
  }

  def printN (n:Int):Unit =
    if (hasElement())
      if (n > 0) {
        println("  " + head())
        tail().printN(n-1)
      }
      else
        println("  ...")

  def sequence (st:Stream[A]):Stream[A] =
    new Sequence(this,st)

  private class Sequence (st1:Stream[A], st2:Stream[A]) extends Stream[A]
    {
    def hasElement ():Boolean = {
      st1.hasElement() || st2.hasElement()
    }
    def head ():A =
      if (st1.hasElement())
        st1.head()
      else
        st2.head()
    def tail ():Stream[A] =
      if (st1.hasElement())
        new Sequence(st1.tail(),st2)
      else
        st2.tail()
  }

  def zip[B] (st2:Stream[B]):Stream[Pair[A,B]] =
    new Zip[B](this,st2)

  private
  class Zip[B] (st1:Stream[A],st2:Stream[B]) extends Stream[Pair[A,B]] {
    def hasElement ():Boolean = {
      st1.hasElement() && st2.hasElement()
    }
    def head ():Pair[A,B] = Pair.create(st1.head(),st2.head())
    def tail ():Stream[Pair[A,B]] = st1.tail().zip(st2.tail())
  }

  def map[B] (f:(A)=>B):Stream[B] =
    new Map[B](this,f)

  private
  class Map[B] (st:Stream[A], f:(A)=>B) extends Stream[B] {
    def hasElement ():Boolean = st.hasElement()
    def head ():B = f(st.head())
    def tail ():Stream[B] = st.tail().map(f)
  }

  def filter (p:(A)=>Boolean):Stream[A] =
    new Filter(this,p)

  private
  class Filter (st:Stream[A], p:(A)=>Boolean) extends Stream[A] {
    def findNext (s:Stream[A]):Stream[A] =
      if (s.hasElement()) {
        if (p(s.head()))
          s
        else
          findNext(s.tail())
      } else
        s
    def hasElement ():Boolean = findNext(st).hasElement()
    def head ():A = findNext(st).head()
    def tail ():Stream[A] = new Filter(findNext(st).tail(),p)
  }
}
```

(This is an example of a trait providing a rich interface — as we saw in the lecture on multiple inheritance.)

Streams are immutable. This is reflected in the interface by having an explicit method to make the stream move to the next element. In particular, repeatedly calling the `first()` method always returns the same answer.

For the sake of examples, recall our implementation of binary trees with streams:

```
object BinTree {

  def empty[T] ():BinTree[T] = new Empty[T]()

  def node[T] (n:T, l:BinTree[T], r:BinTree[T]):BinTree[T] =
    new Node[T](n,l,r)

  private class Empty[T] extends BinTree[T] {

    def isEmpty ():Boolean = true

    def root ():T =
      throw new RuntimeException("BinTree.empty().root()")
    def left ():BinTree[T] =
      throw new RuntimeException("BinTree.empty().left()")
    def right ():BinTree[T] =
      throw new RuntimeException("BinTree.empty().right()")

    def size ():Int = 0

    // canonical methods?

    override def toString ():String = "-"
    // stream methods
    def hasElement ():Boolean = false
    def head ():T =
      throw new RuntimeException("BinTree.empty().head()")
    def tail ():Stream[T] =
      throw new RuntimeException("BinTree.empty().tail()")

  }

  private class Node[T] (n:T, l:BinTree[T], r:BinTree[T]) extends BinTree
    [T] {

    def isEmpty ():Boolean = false

    def root ():T = n
    def left ():BinTree[T] = l
    def right ():BinTree[T] = r

    def size ():Int = 1 + l.size() + r.size()

    // canonical methods?

    override def toString ():String = n + "[" + l + "," + r + "]"

    // stream methods
    def hasElement ():Boolean = true

    def head ():T = n

    def tail ():Stream[T] = new Pair[T](l,r)
  }

  private class Pair[T] (fst:Stream[T],snd:Stream[T]) extends Stream[T] {

    // stream methods
    def hasElement ():Boolean = {
      fst.hasElement() || snd.hasElement()
    }

    def head ():T =
      if (fst.hasElement())
```

```
        fst.head()
      else
        snd.head()

    def tail ():Stream[T] =
      if (fst.hasElement())
        new Pair[T](fst.tail(),snd)
      else
        snd.tail()
  }
}


abstract class BinTree[T] extends Stream[T] {

  def isEmpty ():Boolean
  def root ():T
  def left ():BinTree[T]
  def right ():BinTree[T]
  def size ():Int
}
```

Additionally, consider the following stand-alone stream creators that can be used to create infinite streams:

```
object Stream {

  // EMPTY stream

  def empty[T] ():Stream[T] =
    new Empty[T]

  private class Empty[T] extends Stream[T] {
    def hasElement ():Boolean = false
    def head ():T = throw new RuntimeException("Stream.empty().head()")
    def tail ():Stream[T] = throw new RuntimeException("Stream.empty().
   tail()")
  }

  // CONSTANT stream

  def constant[T] (v:T):Stream[T] =
    new Constant[T](v)
  private class Constant[T] (v:T) extends Stream[T] {
    def hasElement ():Boolean = true
    def head ():T = v
    def tail ():Stream[T] = this
  }

  // INT-FROM stream

  def intsFrom (v:Int):Stream[Int] =
    new IntsFrom(v)

  private class IntsFrom (v:Int) extends Stream[Int] {
    def hasElement ():Boolean = true
    def head ():Int = v
    def tail ():Stream[Int] = new IntsFrom(v+1)
  }
}
```

Our goal today is to look at mutable version of streams, and develop techniques for going back and forth between streams and their mutable versions.

17.1 Iterators

In Java and in Scala (and in many other languages), the standard way of getting sequential access to the elements of an aggregator class is to use an *iterator*.

Iterators are similar in spirit to streams. The big difference is that they are *mutable*:

```
  trait Iterator[+A] {
    def hasNext:Boolean
    def next ():A
  }
```

(Never mind the `+` in front of the type parameter — I'll explain it next lecture. Also, the `Iterator` trait defines several derived operations to provide a rich interface to iterators. We'll focus only on the two abstract methods here.)[1] Method `hasNext()` is similar to `hasElement()` in streams: it returns whether there is an element left to return from the iteration. Method `next()` returns the next element in the iteration, like `first()`, except is also automatically moves the iterator to the next object to be delivered. The important

[1]Another thing to note is that `hasnext` is a method that takes no arguments. We haven't seen those, but they're exactly what you would expect: a method that requires no argument, and not even parentheses.

thing to recognize is that calling `next()` does not produce a new instance of the iterator to point to the next element, unlike streams: the current instance of the iterator is mutated so that it now *points* to the next element. This means, among other things, that invoking `next()` twice on the same iterator will generally yield two different values.

Why do we care about iterators at all, aside from the fact that we need to know about them to iterate over the Collections classes? Putting it differently, if you already have streams for a particular ADT you've designed, why should you care about iterators?

One advantage of iterators is that Scala (and Java) offers *syntactic support* for them. Consider how you would use an iterator, for instance to iterate over all the elements of an array:

```
val arr:Array[Integer] = ...  // some code to create an array

val it:Iterator[Integer] = arr.iterator(); // get an iterator on the array
while (it.hasNext()) {
  val nxt = it.next();
  // some code acting on each integer in the array, e.g.:
  println("Entry = " + nxt.toString());
}
```

A class with a method `iterator` that returns an iterator can implement trait `Iterable[T]`. Class `Array[T]`, for instance, implements `Iterable[T]`. When you have a class implementing `Iterable[T]`, we can use a special for loop, called a foreach loop, as follows:

```
val arr:Array[Integer] = ...// some code to create an array

for (item <- arr)
  System.out.println ("Entry = " + item.toString());
```

This is exactly equivalent to the while loop above. In fact, you can think of the Scala statement

```
for (ITEM_VAR:TYPE <- COLL_EXP)
  STATEMENT
```

as shorthand for the longer

```
val it:Iterator[TYPE] = COLL_EXP.iterator
while (it.hasNext()) {
  val ITEM_VAR:TYPE = it.next();
  STATEMENT;
}
```

Having both streams and iterators around is a bit of a mess. For instance, you might care about using a library that implements a stream,while your code has utility procedures that use iterators, or vice versa.

It is of course possible to rewrite code, or reimplement libraries to use either streams or iterators, whichever your code is expecting.

Another possibility is to write a little class that *adapts* the objects returned by the library to interact better with your application. This is a general enough occurrence that we often call this an *Adapter design pattern*, although it is so obvious that the term seems like overkill.

17.2 An Adapter for Iterators

Consider the `BinTree[T]` implementation at the beginning of this lecture. We could implement an `iterator` method in `BinTree[T]` that returns an `Iterator[T]` instance, but this would require us to rewrite a lot of stream-like code, and we already have streams implemented for lists.

So let's write an adapter class that wraps around an arbitrary stream and yields an iterator that iterates over the same values as the stream.

```
class StreamToIterator[T] (s:Stream[T]) extends Iterator[T] {

  // store the current stream,
  // update whenever asked for an element

  private var current:Stream[T] = s

  def hasNext:Boolean = current.hasElement()

  def next ():T =
    if (current.hasElement()) {
      val result = current.head()
      current = current.tail()
      result
    } else {
      throw new RuntimeException("StreamToIterator.next()")
    }
}
```

First, note that the above adapter is not specific to streams over lists — it works for *any* stream, and turns any such stream into an iterator. Second, note the implementation of `next()`, which needs to update the field holding the stream so that on the next call to `next()` another object is extracted from the iteration. You should understand the above code, perhaps using the framework I described last lecture for modeling mutability. To use this adapter class, we only need to create an instance of it, passing in a stream.

The easiest way to do so is simply to add an `iterator` method to the `Stream` trait:

```
trait Stream[A] {

  ...

  def iterator:Iterator[A] = new StreamToIterator(this)
}
```

As an example, suppose we have a function

```
  def printN[T] (it:Iterator[T],n:Int):Unit =
    if (it.hasNext) {
      if (n>0) {
        print(" [" + it.next() + "]");
        printN(it,n-1)
      } else
        println(" ...")
    } else
      println("")
```

that prints the first n elements delivered by an iterator, and consider the following sample code;

```
    val T = BinTree
    val T1:BinTree[String] = T.node("the",
                                    T.node("quick",
                                           T.node("brown",T.empty(),T.empty()),
                                           T.empty()),
                                    T.node("fox",
                                           T.node("jumps",T.empty(),T.empty()),
                                           T.node("over",
                                                  T.node("the",
                                                         T.node("lazy",
                                                                T.empty(),
                                                                T.empty()),
                                                         T.node("dog",
                                                                T.empty(),
                                                                T.empty())),
                                                  T.empty())))
    println("T1 = " + T1)
    println("Elements T1 (as a stream) = ")
    T1.print()
    val iter = T1.iterator
    println("T1 iterator = ")
    printN(iter,3)
    println("T1 length iterator (continued) = ")
    printN(stringLengthIterator(iter),3)
    println("T1 iterator (continued) = ")
    printN(iter,3)
    println("")
```

```
T1 = the[quick[brown[-,-],-],fox[jumps[-,-],over[the[lazy[-,-],dog[-,-]],-]]]
Elements T1 (as a stream) =
  the
  quick
  brown
  fox
  jumps
  over
  the
  lazy
  dog
T1 iterator =
 [the] [quick] [brown] ...
T1 length iterator (continued) =
 [3] [5] [4] ...
T1 iterator (continued) =
 [the] [lazy] [dog] ...
```

We see that the iterator `iter` is mutable: repeated calls to `printN()` using that same iterator yields different results.

17.3 An Adapter for Streams

The above shows that we can adapt a stream into an iterator. How about the other way around? More precisely, suppose we have an application geared towards using streams, and we want to use one of the Collections classes, which all implement iterators.

It turns out to be a bit more complicated to adapt iterators into streams, and not very robust, because as we saw last time, mutability is contagious. Here's a stab at this:

```
class IteratorToStream[T] (it:Iterator[T]) extends Stream[T] {
  // INVARIANT: we better make sure we do not call
  // next() more than once

  val element:Option[T] =
    if (it.hasNext)
      Option.some(it.next())
    else
      Option.none()

  def hasElement ():Boolean = element.isSome()

  def head ():T =
    if (element.isSome())
      element.valOf()
    else
      throw new RuntimeException("IteratorToStream.head()")

  def tail ():Stream[T] =
    if (element.isSome())
      // note: 'it' has already been moved to the next() element!
      new IteratorToStream[T](it)
    else
      throw new RuntimeException("IteratorToStream.tail()")
}
```

This adapter uses the `Option` ADT, which we saw on the midterm and in a homework.The `Option` ADT lets us avoid the use of `null`, which is always error prone.

The idea behind the above adapter is that we have to make sure that we never call `next()` more than once for every element of the underlying iterator, because whenever we call `next()`, the iterator mutates. We therefore call `next()` once when we construct the stream, storing the value (if there is one — if there is none, we also record that fact, using `Option.none()`) away so that when we ask for `head()` we do not need to query the underlying iterator, we just return the value we stored away. When we move the stream to the next element, we simply construct a new adapter for the underlying iterator, which by that point is already pointing to the next element to be returned anyways.

Again, stare at the above code, and convince yourself that it works, and try to understand how it works. When you do, then you have nailed how mutation works.

Let's test the above adapter by constructing a stream from the iterators of our `BinTree[T]` implementation (which themselves were constructed from streams, so that's two layers of indirection here), continuing on with the example from last section:

```
    val str = new IteratorToStream(T1.iterator)
    println("Stream from iterator from stream on T1 = ")
    str.print()
```

which yields:

```
Stream from iterator from stream on T1 =
  the
  quick
  brown
  fox
  jumps
  over
  the
  lazy
  dog
```

as expected.

I claimed above that `IteratorToStream` is not very robust. What I mean is that there are ways to disrupt the result of iterating using a stream obtained by `IteratorToSteeam`. The underlying iterator is shared with the resulting iterator, meaning that we can mutate the underlying iterator, and the result of the mutation will be visible in the stream. Study the following example, again continuing the sample code above:

```
    println("Messing with iteration")
    val iter2 = T1.iterator
    val str2 = new IteratorToStream(iter2)
    println(" str2.head() = " + str2.head())
    println("Advancing underlying iterator")
    val dummy = iter2.next()
    println(" str2.head() = " + str2.head())
    println(" str2.tail().head() = " + str2.tail().head())
```

This yields:

```
Messing with iteration
 str2.head() = the
Advancing underlying iterator
 str2.head() = the
 str2.tail().head() = brown
```

By accessing the iterator `iter2`, we made the `str2` stream skip an element — it thinks the element after `the` is `brown`, and not `quick`. That's not good.