Object-Oriented Design — **Lecture 10**
CS 3500 Spring 2010 (Pucella) — **Friday, Feb 12, 2010**

10 Design Pattern: Functional Iterators

Subclassing enables the use of standardized interfacs for implementing specific functionality. Standardized interfaces enable code reure (on the client side) because it becomes possible to write functions against those standardized interfaces that will work for every subclass of those interfaces.

Let's look at an example. First, some terminology. An *aggregate* is a data type that contains objects, such as a list data type, or a tree data type, or a hash table data type, or a stack data type. Lists, arrays, and trees are exceedingly common aggregates.

A common operation that we want to do with aggregates is to get all the elements in the aggregate and do something to them: sum them up if they're integers, count them, find their maximum value or their minimum value, compute their average, print them out or display them in some meaningful way. And we want to do this in such a way that we can define general operations for, say, sum all the elements in an aggregate, independently of the kind of aggregate it is.

We will do so by implementing an operation in every aggregate that creates a special kind of object called an *iterator*, which we can think of as a finger pointing to an element of the aggregate, that can give you the value pointed to, and that knows how to "move" to the next element in the aggregate. (For some aggregates, the notion of which element is next is more natural than others.)

So what does an iterator look like? We focus for now on *functional iterators* over aggregates containing integers. A functional iterator is a subclass of the following class — which we will represent as an interface in Java:

```
public interface FuncIteratorInt {
  public boolean hasElement ();
  public int element ();
  public FuncIteratorInt moveToNext ();
}
```

Functional is often used as a synonym for immutable. A functional iterator provides a method `hasElement()` for asking whether we are done iterating over the elements of the underlying aggregate, a method `element()` for getting the current element of the aggregate we have not looked at yet, and a method `moveToNext()` that moves the iterator to the next element after the current one. Note that the `moveToNext()` method returns a new iterator that can be queried for that next element, because our functional iterators are immutable. Once a functional iterator is created, it always points to the same element of the aggregate.[1] One point here is that different aggregates will require quite different iterators — there is no notion of a default implementation of an iterator that works for all aggregates. This is one reason why we use an interface.

Once we have the notion (and interface) of a functional iterator, we can write functions to use them, such as:

```
public static int sumElements (FuncIteratorInt it) {
  if (it.hasElement())
    return it.element() + sumElements(it.moveToNext());
  else
    return 0;
}
```

The function uses recursion to get at all the elements provided by the iterator. The following function, in contrast, uses a while-loop:

```
public static void printElements (FuncIteratorInt it) {
  FuncIteratorInt curr = it;
  while (curr.hasElement()) {
    System.out.println("Element = " + curr.element());
    curr = curr.moveToNext();
  }
}
```

The above functions should work independently of the aggregate we're using (trees, lists, graphs, etc) as long as they define a functional iterator that implements the `FuncIteratorInt` interface.

Lists are aggregate, so let's define a functional iterator for lists. In general, an aggregate will not prescribe the order in which elements should be iterated over. Some structures have a natural ordering (e.g., lists from first to last, arrays from first to last), while others have many natural orders (e.g., trees in breadth-first order or depth-first order with both pre-, in-, and postorder access to the elements), and others have no natural order at all (e.g., graphs). For lists, the easiest order is the natural order of the list.

The first thing we need to do is add an operation to the List ADT that gives us a functional iterator for a list. The revised signature is:

[1]Java comes with an iterator interface in its basic API that is mutable; it does not have an `moveToNext()` method, and querying for the current element mutates the iterator so that querying it for the current element again returns the next element. This mutability makes it somewhat more difficult to reason about iteration. We will return to mutable iterators later.

```
public static List empty ();
public static List cons (int, List);

public boolean isEmpty ();
public int first ();
public List rest ();
public String toString ();
public FuncIteratorInt funcIterator ();
```

The specification for the first four operations is as before, and the specification for the `funcIterator()` operation is trickier, because it creates a new "thing", the iterator. But if you think of an iterator as an ADT with the three operations specified by the `FuncIteratorInt` interface, then there are two creators for functional iterators for lists: `empty().funcIterator()`, and `cons(i,L).funcIterator()`. We can still apply our technique for writing specifications to that:

```
empty().funcIterator().hasElement() = false
empty().funcIterator().element() ---> EXCEPTION
empty().funcIterator().moveToNext() ---> EXCEPTION

cons(i,L).funcIterator().hasElement() = true
cons(i,L).funcIterator().element() = i
cons(i,L).funcIterator().moveToNext() = L.funcIterator()
```

The only real tricky spec equation is the last one — if you get an iterator for a list that starts with element i in front of L, and you move that iterator to the next element, then that should be the same as taking L and taking its iterator.

Now that `funcIterator()` is an operation in our ADT, we can implement it following the Specification Design Pattern we saw a couple of lectures ago. We already have an implementation of `List` that follows that Design Pattern, so it's a simple matter to add an abstract method to the `List` abstract class, and an `funcIterator()` method to both the `EmptyList` and `ConsList` concrete subclasses. Those methods simply create instances of two new classes, implementing functional iterators for empty lists and for cons lists, respectively, each implementing the iterator operations according to the specification above.

```
// An implementation of the List ADT using the
// Specification Design Pattern

public abstract class List {   // abstract == cannot instantiate this class

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

  public static List cons (int i, List L) {
    return new ConsList(i,L);
  }

  public abstract boolean isEmpty ();
  public abstract int first ();
  public abstract List rest ();
  public abstract String toString ();

  public abstract FuncIteratorInt funcIterator ();
}


class EmptyList extends List {
  public EmptyList () {    // the ONE place where we have public constructor
  }

  public boolean isEmpty () {
    return true;
  }

  public int first () {
    throw new RuntimeException("empty().first()");
  }

  public List rest () {
    throw new RuntimeException("empty().rest()");
  }

  public String toString () {
    return "";
  }

  public FuncIteratorInt funcIterator () {
    return new EmptyFuncIteratorInt();
  }
}


class ConsList extends List {
  private int f;
  private List r;

  public ConsList (int i, List l) {    // the ONE place where we have public constructor
    f = i;
    r = l;
  }

  public boolean isEmpty () {
    return false;
  }

  public int first () {
    return f;
  }

  public List rest () {
    return r;
  }

  public String toString () {
    return f + " " + r.toString();
  }

  public FuncIteratorInt funcIterator () {
    return new ConsFuncIteratorInt(f,r);
  }
}
```

The two new classes that are needed, `EmptyFuncIteratorInt` implementing functional iterators for empty lists and `ConsFuncIteratorInt` implementing functional iterators for cons lists, live in the same file as `List`:

```
class EmptyFuncIteratorInt implements FuncIteratorInt {
    public EmptyFuncIteratorInt () {}

    public boolean hasElement () {
        return false;
    }

    public int element () {
        throw new java.util.NoSuchElementException("EmptyIterator.element()");
    }

    public FuncIteratorInt moveToNext () {
        throw new java.util.NoSuchElementException("EmptyIterator.moveToNext()");
    }

}


class ConsFuncIteratorInt implements FuncIteratorInt {
    private int f;
    private List r;

    public ConsFuncIteratorInt (int farg, List rarg) {
        f = farg;
        r = rarg;
    }

    public boolean hasElement () {
        return true;
    }

    public int element () {
        return f;
    }

    public FuncIteratorInt moveToNext () {
        return r.funcIterator();
    }
}
```

It is of course possible to nest the `EmptyFuncIteratorInt` class inside the `EmptyList` class, and similarly nesting the `ConsFuncIteratorInt` class inside the `ConsList` class. Or we can just nest everything inside the `List` class, as before.

The exception thrown when trying to get at the current element when there is no such element, or advancing the iterator passed the end of the list is the one that the Java API requires its iterators to throw, so I have done so here for consistency.

Exercise (hard): *Can you come up with a functional iterator for lists that iterates from the last element of the list to the first? How about a functional iterator that iterate over a list from the first element to the last, but going through all the odd-positioned elements first, then all the even-positioned elements?*