Object-Oriented Design **Lecture 17**
CSU 370 Fall 2008 (Pucella) **Tuesday, Nov 18, 2008**

Higher-Order Programming

A common operation that we want to do for collections (aggregate data types) is to transform all the elements in the collection in some way. For instance, if we have a list of integers, we may want to transform every integer in the list and, say, double it.

Languages such as Scheme (or ML) make this very easy, because they let you pass functions as arguments to other functions. This is called *higher-order programming*.[1] In Scheme, for instance, we can write a function `map` that applies a function to every element of a list:

```
(define (map f l)
  (cond ((empty? l) l)
        (else (cons (f (first l)) (map (rest l))))))
```

To call such a function, you pass it a function, say one that adds 5 to its input:

```
(map (lambda (x) (+ x 5)) (list 1 2 3 4 5))
```

to get the list `(6 7 8 9 10)`.

The question is, how can we do something similar in Java? For the sake of concreteness, let us consider defining a map operation on stacks. (Of course, as we know, stacks are just linked lists anyways, so whatever we do here will apply almost immediately to a linked list ADT.)

Basically, we want to add to the Stack ADT a new operation that will map a function over all the elements of the stack, to produce a new stack. In other words, we want an operation with the following signature (I am using the generic stack ADT we saw last time):

```
public Stack<T> map (Function<T,T>);
```

[1] You may wonder why the terminology. A function that takes a base type (such as an integer) and produces a base type is often called a first-order function. A function that takes a first-order function as an argument and returns a base type is often called a second-order function. A function that takes a second-order function as an argument and returns a base type is often called a third-order function, and so on. In general, any function that expects an argument that is not a base type is at least a second-order function, and that makes it a higher-order function. In an OO language, because objects are base types but can contain methods (which are just functions), this distinction is a bit fuzzy, but the terminology has remained.

Intuitively, `Function<T,T>` is the type of functions taking values of type `T` and returning values of type `T`. So what can that type be? Well, we don't have that many choices. It has to be a class type, or an interface type. So a function is a class. What operations are defined on functions? Putting it differently, you have a function, what's pretty much the only thing you can do with it? Call it, giving it some arguments. So that's the one operation that `Function<T,T>` supplies. Do we know what the default implementation of an arbitrary function is? No, then this calls for an interface. So here it is:

```
public interface Function<T,U> {
  public U call (T arg);
}
```

This is a parameterized interface (or a generic interface), and note that it is parameterized by two types, the argument type and the result type of the `call` method.

To create a function, we need to define a class that implements the interface. Different classes will create different functions. Consider the function we saw above, that in Scheme I wrote as `(lambda (x) (+ x 5))`. Here is a class that corresponds to it:

```
public class AddFive implements Function<Integer,Integer> {

  private AddFive () {};

  public static AddFive create () {
    return new AddFive();
  }

  public Integer call (Integer i) {
    return i + 5;
  }
}
```

The class, `AddFive` has a creator to create an instance of the class, and the `call` operation required by the interface (note the types). Thus, `AddFive.create()` is something that could be passed to the `map` operation on stacks:

```
  Stack<Integer> foo = // create some stack here
  Stack<Integer> bar = foo.map(AddFive.create());
```

This will create a stack `bar` that looks like `foo` except every element is five more than in `foo`.

Here are two more examples of functions we can create.[2] First, a function that adds a constant to its input. The constant to add to the input is given as an argument when the instance of the class is created:

```
public class AddConstant implements Function<Integer,Integer> {
  private int constant;
  private AddConstant (int c) { constant = c; }

  public static AddConstance create (int c) {
    return new AddConstance(c);
  }

  public Integer call (Integer i) {
    return i + constant;
  }
}
```

To create a function that adds 10 to its input, we simply invoke `AddConstant.create(10)`—this creates something we can pass to `map` and that will add 10 to each element of the stack.

Another example to show that we are not restricted to creating functions that return a value of the same type as the input:

```
public class IntToString implements Function<Integer,String> {
  private InttoString () {};

  public static IntToString create () {
    return new IntToString();
  }

  public String call (Integer i) {
    return i.toString();
  }
}
```

Of course, this function cannot be passed to `map`—we'll fix that later by introducing a more general mapping function.

Okay, so now we know how to represent functions. Let's worry about how to implement map. Here is the signature for `Stack<T>`:

```
  public static <U> Stack<U> empty ();
```

[2]Not given in class.

```
  public static <U> Stack<U> push (Stack<U>, U);
  public boolean isEmpty ();
  public T top ();
  public Stack<T> pop ();
  public Stack<T> map (Function<T,T>);
```

It's as before, except with the additional `map` operation. What about the specification? Well, it's just like the old specification for the operations we had before, along with a new specification for `map`:

```
  Stack.empty().isEmpty() = true
  Stack.push(s,i).isEmpty() = false
  Stack.push(s,i).top() = i
  Stack.push(s,i).pop() = s

  // New specification for map
  Stack.empty().map(f) = Stack.empty()
  Stack.push(s,i).map(f) = Stack.push(s.map(f),f.call(i))
```

Make sure you understand the specification. Once you do, then implementing it, using the recipe, is near trivial:

```
abstract public class Stack<T> {

  public static <U> Stack<U> empty () {
    return new EmptyStack<U> ();
  }

  public static <U> Stack<U> push (Stack<U> s, U i) {
    return new PushStack<U> (s,i);
  }

  abstract public boolean isEmpty ();
  abstract public T top ();
  abstract public Stack<T> pop ();

  abstract public Stack<T> map (Function<T,T> f);

}


// Concrete subclass for empty()
//
```

```
class EmptyStack<T> extends Stack<T> {

  public EmptyStack () {};

  public boolean isEmpty () { return true; }

  public T top () {
    throw new IllegalArgumentException("top() invoked on empty stack");
  }

  public Stack<T> pop () {
    throw new IllegalArgumentException("pop() invoked on empty stack");
  }

  public Stack<T> map (Function<T,T> f) {
    return Stack.empty();
  }
}


// Concrete subclass for push()
//
class PushStack<T> extends Stack<T> {

  Stack<T> rest;
  T topVal;

  public PushStack (Stack<T> s, T i) {
    rest = s;
    topVal = i;
  }

  public boolean isEmpty () { return false; }

  public T top () { return this.topVal; }

  public Stack<T> pop () { return this.rest; }

  public Stack<T> map (Function<T,T> f) {
    return Stack.push(this.pop().map(f), f.call(this.top()));
  }
}
```

See, nothing to it. The most difficult bit was figuring out how to represent functions.

As I pointed out above, this `map` operation is rather limited. It can only transform a `Stack<T>` into a `Stack<T>`. We may want to more generally transform a `Stack<T>` into a `Stack<U>`, if we have a function that maps `T`s into `U`s. For instance, we may want to map a `Stack<Integer>` into a `Stack<String>` by converting every integer to a string via the function represented by `IntToString` above. I claim that we can do this using an operation with the following signature:

```
  public <U> Stack<U> genmap (Function<T,U>);
```

Exercise: *Give a specification for `genmap`. (Hint: easy!) Implement `genmap` by adding the suitable methods to the Stack implementation above. (Hint: also easy, if you can get the type annotations right!)* □

We implemented maps for stacks here. We can easily do so for other ADTs, as long as they are aggregates. On homework 5, you will implement a variant of maps for movies. As a further exercise, consider the Tree ADT, a generic variant of what you were asked to implement on the midterm. The `Tree<T>` ADT has the following signature and specification:

```
  public static <U> Tree<U> empty ();
  public static <U> Tree<U> node (U, Tree<U>, Tree<U>);
  public boolean isEmpty ();
  public T root ();
  public Tree<T> left ();
  public Tree<T> right ();

  Tree.empty().isEmpty() = true
  Tree.node(v,l,r).isEmpty() = false
  Tree.node(v,l,r).root() = v
  Tree.node(v,l,r).left() = l
  Tree.node(v,l,r).right() = r
```

Exercise: *add an operation*

```
  public Tree<T> map (Function<T,T>);
```

to the ADT, and give its specification. Then, implement the ADT, including `map`. Add and implement a corresponding `genmap` too. □

You should be able to see how you can implement most of the higher-order functions that you saw in 211/213 this way—folds (or reduces), filters, etc.

There is another use for higher-order programming. Instead of implementing higher-order operations in ADTs, we may want to transform values as they are processed, say, in iterators.

Suppose that you have an ADT with a functional iterator defined for it, that simply produces a sequence of values when requested. Suppose you have an instance `A` of that ADT. We may care for the sequence of values from `A` but transformed somehow. We could do so by taking `A` and mapping the transformation onto it. Another way is to define a "adapter" that sits between the stream of values coming from the iterator, and transforming them before giving them to the requester. You can think of this as an iterator-mapping adapter that takes an iterator and a function, and acts as an iterator itself. When asked for a value, it asks the underlying iterator for `A` for a value, apply the function, and returns the result to the requester.

Here is some code that does this. It implements an ADT `MapIterator` whose only operations are the iterator operations. It is highly generic, so pay attention!

```
public class MapIterator<T,U> implements FuncIterator<U> {
  private FuncIterator<T> incoming;
  private Function<T,U> function;

  private MapIterator (FuncIterator<T> inc, Function<T,U> fun) {
    incoming = inc;
    function = fun;
  }

  public static <V,W> MapIterator<V,W> create (FuncIterator<V> inc,
                                               Function<V,W> fun) {
    return new MapIterator<V,W>(inc,fun);
  }

  public boolean hasElement () { return incoming.hasElement(); }

  public U current () { return function.call(incoming.current()); }

  public FuncIterator<U> advance () {
    return new MapIterator<T,U> (incoming.advance(), function);
  }
}
```

To use one of these beasts, suppose that you have `A` as above, that implements interface `FuncIterator<Integer>`—meaning that you can invoke `A.current()` to get the current element in the structure, and `A.advance()` to point to the next element, etc. If you are interested in all the integers in the structure, plus 5, then you can simply create a `MapIterator` and hook it up to `A`'s iterator as follows:

```
  FuncIterator<Integer> iterPlus5 = MapIterator.create(A,AddFive.create());
```

Now, if `A` contains values 1,2,3,4 (in the order given by iterating over `A`), then `iterPlus.current()` returns 6, `iterPlus.advance().current()` returns 7, etc.

This kind of iterator adapter comes in handy when you work with iterators a lot.