Functional Iterators II

Let's continue looking at functional iterators for stacks. (Not because stacks are especially interesting, but they're simple. For something more complicated, there's always the Picture ADT in homework 3...) Suppose that the specification says that we should iterate over the stack elements in a top-to-bottom order. In that case, we can do better than the array-based iterators in last lecture. In particular, we do not actually need to copy the content of the stack into an array—we can just use the stack itself in the iterator!

```
public class Stack {

  ... // original code

  // a new form of iterator
  public FuncIterator getFuncIterator () {
    return new StackFuncIterator (this);
  }

  private class StackFuncIterator implements FuncIterator {
    Stack stack;
    public StackFuncIterator (Stack s) {
      stack = s;
    }
    public boolean hasElement () {
      return !(stack.isEmpty());
    }
    public current () {
      if (!(this.hasElement()))
        throw new NoSuchElementException ("In StackFuncIterator");
      return stack.top();
    }
    public FuncIterator advance () {
      if (!(this.hasElement()))
        throw new NoSuchElementException ("In StackFuncIterator");
      return stack.pop().getFuncIterator();
    }
```

```
  }
}
```

Study the above code, and see why it works. In particular, note that the reason why this works at all is that both the iterator *and* the stack class are immutable. Because a stack, once created, never changes, we can freely pass it around to the iterator, who is free to iterate over it without worrying about some other part of the code changing that stack. In effect, this is equivalent to everyone getting their own copy of a stack when it is passed to them.

The above is better than the array-based iterator, but we can still do better. Indeed, the above code contains a lot of if-then-else statements that branch based on the representation (whether the stack is empty or not). As we saw using the recipe, we can eliminate these kind of if-then-else statements by appropriate use of subclassing. We can do the same here. Let's modify the recipe-based stack implementation.

```
public abstract class Stack {

  ... // original recipe-based implementation

  // new abstract method for getting the iterator
  public abstract FuncIterator getFuncIterator ();

  // the recipe-derived nested class for an empty stack
  private static class EmptyStack implements Stack {
    .. // original EmptyStack code

    public FuncIterator getFuncIterator () {
      return new EmptyStackFuncIterator ();
    }
  }

  // the recipe-derived nested class for a nonempty stack
  private static class PushStack implements Stack {
    .. // original PushStack code

    public FuncIterator getFuncIterator () {
      return new PushStackFuncIterator (this.topVal, this.rest);
    }
  }

  // a new nested class - iterators for empty stacks
  private static class EmptyStackFuncIterator implements FuncIterator {
    public EmptyStackFuncIterator () {}
```

```
    public boolean hasElement () {
      return false;
    }
    public int current () {
      throw new NoSuchElementException ("In EmptyStackFuncIterator");
    }
    public FuncIterator advance () {
      throw new NoSuchElementException ("In EmptyStackFuncIterator");
    }
  }

  // a new nested class - iterators for nonempty stacks
  private static class PushStackFuncIterator implements FuncIterator {
    private int topVal;
    private Stack rest;
    public PushStackFuncIterator (int t, Stack r) {
      topVal = t;
      rest = r;
    }
    public boolean hasElement () {
      return true;
    }
    public int current () {
      return topVal;
    }
    public FuncIterator advance () {
      return rest.getFuncIterator();
    }
  }
}
```

It is also possible to nest the `EmptyStackFuncIterator` class inside the `EmptyStack` class, and similarly nesting the `PushStackFuncIterator` class inside the `PushStack` class, but that seems somewhat overkill; they are already hidden away inside the `Stack` class.

Exercise: Can you come up with a more efficient way to write a bottom-to-top iterator , that does not involve multiple traversals of the stack?

Generic Interfaces

The functional iterator interface we have defined above is nice, but it is not very general. As defined, it can only be used to iterate over structures that yield integers. (Because of the definition of the `current()` method. If we wanted to define an iterator over structures that yields, say, Booleans, or `Person` objects, then we need to define a new iterator interface for that type. That's suboptimal, to say the least. Can we figure out a nice way to reuse the interface?

One way to get a general iterator interface is just to define it as:

```
// An alternative FuncIterator interface
public interface FuncIterator {
  public boolean hasElement ();
  public Object current ();
  public FuncIterator advance ();
}
```

We gain generality, but we lose compile-time checking. What do I mean? Suppose that we change `Stack` to implement the above interface. Consider how we would use such an iterator.

```
Stack s;
// construct stack s
int total=0;
FuncIterator i;
for (i=s.getFuncIterator(); i.hasElement(); i=i.advance()) {
  Object obj = i.current();
  Integer val = (Integer) obj;
  total = total + val;
}
return total;
```

The cast to integer in the middle of the code is a runtime check. If we somehow mess up the definition of stack iterators so that we sometimes return not an integer but, say, a Boolean, then we will only catch the bug at runtime, when we attempt to cast that Boolean into an integer and fail. This kind of bug cannot occur in the previous example, where we used an interface that had `current()` returning an integer, because the Java type checker would have caught an implementation of the iterator that does not always return an integer.

Bottom line: if use `Object` and subclassing, we get code reuse (having to define a single interface for all iterators), but trade-off compile-time checking for some run-time checking.

What we really want, however, is an interface that is *parameterized* by the type of result it returns.

```
public interface FuncIterator<T> {
  public boolean hasElement ();
  public T current ();
  public FuncIterator<T> advance ();
}
```

Basically, the definition is as before, except for the `<T>` annotation. This defines interface `FuncIterator` with a type parameter `T`. (In Java, this is called a *generic interface*, but also a parameterized interface, or a parameterized type.) Within `FuncIterator<T>` you can use `T` as if it were a type. When it actually comes time to use such an interface, you instantiate it at the type you require, say `FuncIterator<Integer>` or `FuncIterator<Person>`. Intuitively, `FuncIterator<Integer>` is as if you had written `FuncIterator` with `Integer` in place of every `T`. There is no need to cast, and types are checked at compile time.

Suppose we wanted to have the `Stack` class use the above interface for its iterator. First off, for simplicity, let's change all the uses of `int` in `Stack` to `Integer`. (Because of automatic boxing and unboxing, we do not actually need to change any code for this to work—only the types.) To use the above interface, the declaration of the `getFuncIterator()` method in `Stack` becomes:

```
public FuncIterator<Integer> getFuncIterator() {
  ... // same code as before
}
```

because the iterator returned has interface `FuncIterator<T>` specialized at the `Integer` type.

The iterator (in `StackFuncIterator`) is defined by:

```
private static class StackFuncIterator implements FuncIterator<Integer> {
  ... // same code as before
}
```

Thus, everywhere we use the `FuncIterator` interface, we get to specify exactly how to instantiate the `T` parameter in the definition.

To use the iterator is as simple as using the original iterator we defined:

```
Stack s;
// construct stack s
int total=0;
FuncIterator<Integer> i;
for (i=s.getFuncIterator(); i.hasElement(); i=i.advance()) {
```

```
    total = total + i.current();
  }
  return total;
```

We get both code reuse and compile-time type checking: we used a single definition for `FuncIterator`, and Java checks at compile time that the iterator always returns an `Integer`.

(Why did we have to redefine `Stack`s to use `Integer` instead of `int`? This is because a generic interface can only be instantiated using class type or an array type, not a primitive type.)

Generic interfaces are extensively used in the Java Collections framework.