Functional Iterators, Continued

Last time, we started looking at functional iterators. I finished with a couple of questions. Let's answer them.

First, let's see how to use such iterators. The idea is to mimick what we do when we iterate over arrays. Here is a method to compute the sum of an integer array:

```
public static int computeArraySum (int[] arg) {
  int total=0;
  for (int i = 0; i < arg.length; i=i+1)
    total=total+arg[i];
  return total;
}
```

Here is a similar method to compute the sum of an integer stack. Notice the parallels with the above code.

```
public static int computeStackSum (Stack arg) {
  int total=0;
  for (IFunIterator i = arg.getFunIterator(); i.hasNext(); i=i.advance())
    total=total+i.next();
  return total;
}
```

Of course, it is possible to abstract away from this and give a method to simply compute the sum over any structure with an iterator.

```
public static int computeIterSum (IFunIterator arg) {
  int total=0;
  for (IFunIterator i=arg; i.hasNext(); i=i.advance())
    total=total+i.next();
  return total;
}
```

Now, as to the question of efficiency. The stack iterator we implemented last time was somewhat inefficient. It basically traversed the structure twice. Once to construct the initial

array, and the second time when actually iterating over the array. There are times when this is the best we can do easily.

However, here, we can do better. Suppose that the specification says that we should iterate over the stack in a top-to-bottom order. (The specification may well say that we do not care the order in which we iterate over a structure; in that case, feel free to pick to easiest one to implement.) In that case, we can do better. 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 IFunIterator getFunIterator () {
    return new StackFunIterator (this);
  }

  private class StackFunIterator implements IFunIterator {
    Stack stack;
    public StackFunIterator (Stack s) {
      stack = s;
    }
    public boolean hasNext () {
      return !(stack.isEmpty());
    }
    public next () {
      if (!this.hasNext())
        throw new NoSuchElementException ("In StackFunIterator");
      return stack.top();
    }
    public IFunIteratorInt advance () {
      if (!this.hasNext())
        throw new NoSuchElementException ("In StackFunIterator");
      return stack.pop().getFunIterator();
    }
  }
}
```

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 IFunIterator getFunIterator ();

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

    public IFunIterator getFunIterator () {
      return new EmptyStackFunIterator ();
    }
  }

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

    public IFunIterator getFunIterator () {
      return new PushStackFunIterator (this.topVal, this.rest);
    }
  }

  // a new nested class - iterators for empty stacks
  private static class EmptyStackFunIterator implements IFunIterator {
    public EmptyStackFunIterator () {}
    public boolean hasNext () {
      return false;
    }
    public int next () {
      throw new NoSuchElementException ("...");
    }
    public IFunIterator next () {
```

```
      throw new NoSuchElementException ("...");
    }
  }

  // a new nested class - iterators for nonempty stacks
  private static class PushStackFunIterator implements IFunIterator {
    private int topVal;
    private Stack rest;
    public PushStackFunIterator (int t, Stack r) {
      topVal = t;
      rest = r;
    }
    public boolean hasNext () {
      return true;
    }
    public int next () {
      return topVal;
    }
    public IFunIterator next () {
      return rest.getFunIterator();
    }
  }
}
```

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

What if we wanted to iterate the other way, that is, iterate over a stack in the bottom-to-top direction. If we use the array-mediated iterator of last time, the change is almost trivial. Instead of filling in the array from the left, we fill it in from the right:

```
  public IFunIterator getFunIterator () {
    int[] content = new int[this.length()];
    Stack current = this;
    for (int i = content.length-1; i >=0 ; i--) {
      content[i] = current.top();
      current = current.pop();
    }
    return new StackFunIterator (content,0);
  }
```

Everything else is the same. That's nice.

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 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 `next()` 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.

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

```
public interface IFunIterator {
  public boolean hasNext ();
  public Object next ();
  public IFunIterator 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 = ... // some stack constructed here
int total=0;
for (IFunIterator i=s.getFunIterator(); i.hasNext(); i=i.advance()) {
  Object obj = i.next();
  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 `next()` 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: we gain generality (having to define a single interface for all iterators), but lose compile-time checking for some run-time checking.

Up until a couple of years ago, that was the trade-off you had to live with, generality versus compile-time checking.

Generics provide a way to be both general and preserve compile-time checking. Let's focus on a specific use of generics for now, specifically, generic interfaces.

Here is the definition of a generic interface for functional iterators:

```
public interface IFunIterator<T> {
  public boolean hasNext ();
  public T next ();
  public IFunIterator<T> advance ();
}
```

Basically, the definition is as before, except for the funny `<T>` annotation. This defines interface `IFunIterator` with a type parameter `T`. (This is variously called a generic interface, or a parameterized interface, or a parameterized type.) Within `IFunIterator<T>` you can use `T` as if it were a type. When it actually comes time to create an object with interface `IFunIterator<T>`, you say `IFunIterator<Integer>` or `IFunIterator<Person>`. You construct new types from that `IFunIterator<T>`, and it is truly as if your type arguments get substituted for the type parameter. All of the `T`s become `Integer`s or `Person`s, you don't have to cast, and there is compile-time type checking everywhere.

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 `getFunIterator()` method in `Stack` becomes:

```
  public IFunIterator<Integer> getFunIterator() {
    .. // same code as before
  }
```

because the iterator returned has interface `IFunIterator` specialized at the `Integer` type.

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

```
  private static class StackFunIterator implements IFunIterator<Integer> {
    ..
  }
```

Thus, everywhere we use the `IFunIterator` 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 = ... // some stack constructed here
  int total=0;
```

```
for (IFunIterator<Integer> i=s.getFunIterator();
                                        i.hasNext(); i=i.advance()) {
  total = total + i.next();
return total;
```

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

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

Generic interfaces are extensively used in the Java Collections framework. In particular, the interface `Comparable` we defined last time has a generic version, with the following definition:

```
public interface Comparable<T> {
  public int compareTo (T other);
}
```

A class `C` implementing interface `Comparable<Integer>` guarantees that, at compile time, we only compare objects of class `C` to objects of class `Integer`.