Design Pattern: Functional Iterators

There are two main ways, in practice, in which interfaces are used in Java programming:

(1) To capture functionality orthogonal to the natural class hierarchy; and

(2) To define a class for which there is no natural default implementation.

Last time, we saw an example of (1); the notion of a salaried person was somewhat orthogonal to the natural hierarchy of people in a university setting. (In other words, it kind of cuts across the hierarchy.)

What about item (2) above, classes for which there is no default implementation?

Again, 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. Arrays are typical aggregates. Arrays have the added advantage that there is nice built-in syntax for them, including easy ways of iterating over all the elements of an array using a `for` loop.

This idea of iteration can be generalized to all aggregates. An *iterator* is an object whose sole purpose is to make it easy to iterate over all the elements of an aggregate.

What's an iterator? A (functional) iterator is an object that implements the following interface:

```
public interface FuncIterator {
  public boolean hasElement ();
  public int current ();
  public FuncIterator advance ();
}
```

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 `current()` for getting the current element of the aggregate we have not looked at yet, and a method `advance()` that advances the iterator past the current element so that we can look at the element after than. The `advance` method returns a new iterator

that can be queried for that next element.[1] The 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. Thus, we use an interface instead of a class above.

Stacks are aggregate, so let's define an iterator for stacks. The first thing we need to do is add a method to the `Stack` class that gives us a functional iterator to iterator over the implicit stack argument. That iterator creates a new instance of a stack iterator, which is a class that we define nested inside the `Stack` class. (It does not need to be nested inside, but since we only ever create instances of it from within the class, we might as well in order not to pollute the namespace.)

There are many ways of implementing iterators. Let's pick a fairly simple minded one that we will improve next time. Roughly, we create an iterator by passing it an array containing all the objects to iterate over.

> Diversion: arrays
>
> An array is a data structure that allows constant-time access to the elements in the array. Arrays have a fixed size.
>
> An array of values of type `T` has type `T[]`. Any type `T` can be used. A new array of size 10, containing values of type `T[]` is created using `new T[10]`. If `a` is an array of size n, the elements of the arrays can be accessed using `a[0]`, ..., `a[n-1]`. (Note that indexing starts with 0.)
>
> Initially, an array is created with all elements initialized to either 0 (in case of integer or float arrays), `false` (in case of Boolean arrays), or `null` (in case of object arrays). To specify an initialized array, one can use special syntax such as:
>
> ```
> int[] a = {0,1,1,2,3,5,8,13}
> ```
>
> This creates an array of size 8, initialized with the supplied values.
>
> Arrays are in some sense objects, and they do have fields. A useful field is `length`, which returns the length of the array. Thus, in the above example, `a.length` evaluates to 8.
>
> A useful operation on arrays is iterating over all the elements of the array. A `for` loop can be used for that. There are two flavors of `for` loops that can be used. The first is C-like. Suppose that `sa` is a `String` array, initialized with some strings, and suppose that we wanted to print out all the strings in the array, one per line.

[1] Java comes with an iterator interface in its basic API that is mutable; it does not have an `advance()` method, and query for the current element in the interface mutates the iterator under the hood so that querying it for the next element returns yet a new element. This mutability, as usual, makes it somewhat more difficult to reason about iteration (but also sometimes more efficient—the usual tradeoff). We will come back to mutable iterators later.

```
for (int i = 0; i<sa.length; i++)
  System.out.println sa[i];
```

The `for` loop specifies three things: a declaration for the variable that will hold the current index when iterating through the array, along with an initial value (here, `int i = 0`), a test for when to continue iterating (here, we continue iterating as long as as `i < sa.length`, that is, as long as the index has not reached the end of the array), and how to update the index at the end of every iteration (here, we increment the index by one, `i++`).

A cleaner way to express this `for` loop, one that does not require us to worry about indices into the array (which always opens the door to either going past the end of the array, or stopping short before the array ends) is to use the second form, called a `for-each` loop:

```
for (String s : sa)
  System.out.println s;
```

The idea is that variable `s`, of type `String`, repeatedly gets bound to each element of array `sa` in turn.

We create such an array in the `getFuncIterator()` method that will create the iterator instance. Let's hack on the `Stack` class, adding to it a (private) `length()` method.

```
public class Stack {
  ... // same code as before

  private int length () {
    if (this.isEmpty())
      return 0;
    else
      return (1 + this.pop().length());
  }

  public FuncIterator getFuncIterator () {
    int[] content = new int[this.length()];
    Stack current = this;
    for (int i = 0; i < content.length; i++) {
      content[i] = current.top();
      current = current.pop();
    }
    return new StackFuncIterator (content,0);
  }
```

```
  private static class StackFuncIterator implements FuncIterator {
    private int[] content;
    private int index;

    // constructor takes the array of content and the initial index
    public StackFuncIterator (int[] c, int i) {
      content = c;
      index = i;
    }

    public boolean hasElement () {
      return (this.index < content.length);
    }

    public int current () {
      if (this.hasElement())
        return content[this.index];
      throw new java.util.NoSuchElementException ("Stack iterator empty");
    }

    public FuncIterator advance () {
      if (this.hasElement())
        return new StackFuncIterator (content,index+1);
      throw new java.util.NoSuchElementException ("Stack iterator empty");
    }
  }
}
```

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 stack is the one that the Java API requires its iterator to throw, so I have done so here for consistency.

Here is a method to compute the sum of an integer stack.

```
public static int computeStackSum (Stack arg) {
  FuncIterator i;
  int total=0;
  for (i = arg.getFuncIterator(); i.hasElement(); i=i.advance())
    total=total+i.current();
  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 (FuncIterator arg) {
  FuncIterator i;
  int total=0;
  for (i=arg; i.hasElement(); i=i.advance())
    total=total+i.current();
  return total;
}
```

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 FuncIterator getFuncIterator () {
    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 StackFuncIterator (content,0);
  }
```

Everything else is the same. That's nice.

The stack iterator we implemented above is 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.

Question for next time: how can you do it more efficiently?