Functional Collection Patterns in Java

How I Learned to Stop Worrying and Love Functional Programming

by Bill Six

Table of Contents

1 Collections

While many software developers are familiar with the Iterator pattern from the classic "Design Patterns" book, the Iterator pattern just scratches the surface of the abstractions available on collections. Classic languages such as Scheme, Common Lisp, and Smalltalk-80 provide much more powerful abstractions over collections. These abstractions are available in Java, they're just a bit more verbose.

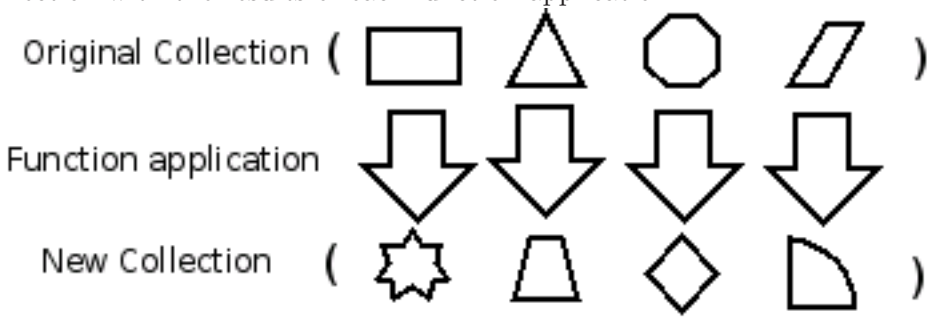
The patterns listed in this paper are all dependent on the availability of high-order functions. A high-order function is a function that takes in a function as an argument, or returns a function. By the end of this paper I hope to show that the use of high-order functions is extremely useful for separating the collection's concerns from the user of the collection.

There are three main categories of operations on collections:

- Map
- Filter
- Reduce

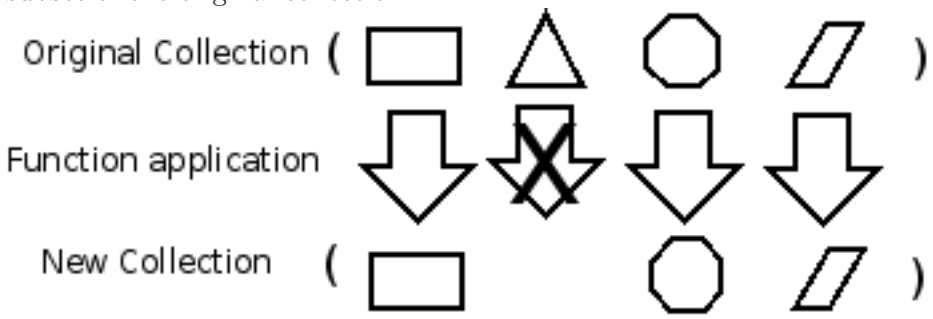
1.1 Map

The Map pattern evaluates a high-order function on all elements of the collection. It returns a new collection with the results of each function application.



1.2 Filter

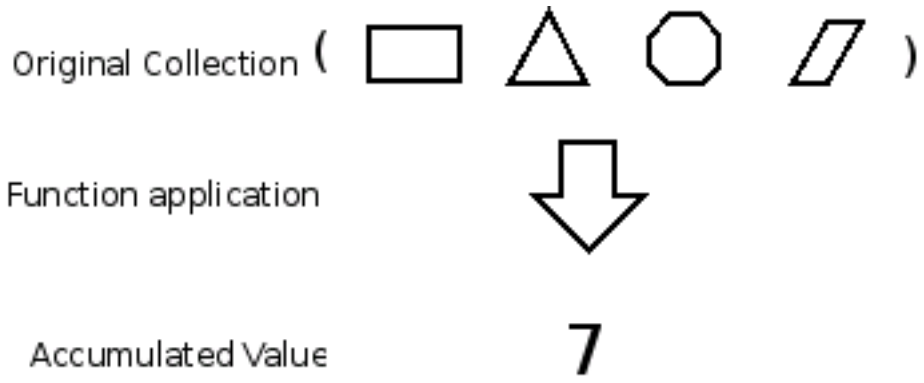
The Filter pattern evaluates a predicate[1] on each of the elements, returning a new collection which is subset of the original collection.



[1] a function which returns a Boolean

1.3 Reduce

The Reduce pattern evaluates a function on all elements of the collection, returning a scalar value. For instance, in the following diagram, Reduce is applied to sum the area of each shape.



2 Java Examples

This section will show how to use anonymous inner classes as high-order functions to implement the Map, Filter, and Reduce patterns.

2.1 High-order functions

High-order functions are available in Java by using anonymous inner classes[1].

```
interface HighOrder
{
    public int value(int i);
}
public class Anon
{
    int i;
    public Anon(int i)
    {
        this.i = i ;
    }
    public void modifyI(HighOrder highOrder)
    {
        i = highOrder.value(i);
    }
    public static void main(String[] args)
    {
        Anon anonymous = new Anon(5);
        anonymous.modifyI(new HighOrder(){
                public int value(int i)
                {
                   return i+1 ;
                }
        });

        System.out.println(anonymous.i);
    }
}

  result => 6
```

2.2 Map example

First, we need an interface that defines our high-order function. This function takes an arbitrary object, and returns an object.

```
public interface Expression {
    public Object value(Object element);
}
```

Suppose I have a collection of lowercase letters that I want to convert to uppercase letters.

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.List;
```

[1] Anonymous inner classes aren't quite as flexible as Lisp/Smalltalk-80's closures, but I won't open that Pandora's box

```
import junit.framework.TestCase;

public class MapTest extends TestCase {

 public void testCollect() {
   List strings = new ArrayList();
   strings.add("a");
   strings.add("e");
   strings.add("b");
   strings.add("s");
   strings.add("d");

   List upperCaseLetters = (List) Map.overCollection(strings).collect(
     new Expression(){
     public Object value(Object element) {
        return ((String)element).toUpperCase();
     }});

   assertTrue(upperCaseLetters.size() == 5);
   assertTrue(upperCaseLetters.get(0).equals("A"));
   assertTrue(upperCaseLetters.get(1).equals("E"));
   assertTrue(upperCaseLetters.get(2).equals("B"));
   assertTrue(upperCaseLetters.get(3).equals("S"));
   assertTrue(upperCaseLetters.get(4).equals("D"));
  }
}
```

After looking at this example, you may wonder why not just get an Iterator from the List, and create a new List manually. You could do that, and you could make a good argument for why it would be simpler.

I prefer the high-order function approach because I don't have to set any state, reducing the possibility of iteration-related bugs. The act of iteration over the collection and the creation of the new collection is the responsibility of the Map class. My responsibility as the implementer of the Expression interface is to say how to transform each element of the old collection into the element of the new collection.

2.3 Filter example

Since Filter requires a predicate to filter out certain elements of the collection, our previous high-order function won't work. So we define a new interface which returns a Boolean

```
public interface BooleanExpression {
   public boolean value(Object element);
}
```

There are a few different ways that you might want to filter a collection. You may want to only return elements where the predicate is true. This is done using the select method of the Filter class. The detect method is a special case of the select method where you're only interested in the first match. Lastly, the reject method only returns elements when the predicate is false.

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.List;

import junit.framework.TestCase;
```

```
public class FilterTest extends TestCase {

    List strings ;

    public void setUp()
    {
      strings = new ArrayList();
      strings.add("a");
      strings.add("e");
      strings.add("B");
      strings.add("s");
      strings.add("B");
    }

    public void testSelect() {
       List newList = (List) Filter.overCollection(strings).select(
         new BooleanExpression(){
         public boolean value(Object element) {
            return ((String)element).charAt(0) == 'B';
         }});

        assertTrue(newList.size() == 2);
        assertTrue(newList.get(0).equals("B"));
        assertTrue(newList.get(1).equals("B"));
    }

    public void testDetect() {
        String b = (String) Filter.overCollection(strings).detect(
           new BooleanExpression(){
           public boolean value(Object element) {
             return ((String)element).charAt(0) == 'B';
           }});
        assertTrue(b.equals("B"));
    }

    public void testReject() {
        List newList = (List) Filter.overCollection(strings).reject(
           new BooleanExpression(){
           public boolean value(Object element) {
             return ((String)element).charAt(0) == 'B';
           }});

        assertTrue(newList.size() == 3);
        assertTrue(newList.get(0).equals("a"));
        assertTrue(newList.get(1).equals("e"));
        assertTrue(newList.get(2).equals("s"));
    }
}
```

2.4 Reduce example

The Reduce pattern is useful when you want to return a scalar value from a collection. We need a new interface whose method takes in an accumulator in addition to an element of the collection. If this doesn't make sense, think about the example where you need to sum the area of various shapes. You need to have some value to accumulate the results into.

```
public interface ReduceExpression {
```

```
    public Object value(Object element, Object accumulator);
}
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.List;

import junit.framework.TestCase;

public class ReduceTest extends TestCase {

    public void testReduce() {
      List strings = new ArrayList();
      strings.add("a");
      strings.add("e");
      strings.add("b");
      strings.add("s");
      strings.add("d");

      String result = (String) Reduce.overCollectionWithInitialValue(strings,"").reduce(
         new ReduceExpression(){
         public Object value(Object element, Object accumulator){
           return accumulator.toString() + element.toString();
         }
      });
       assertTrue(result.equals("aebsd"));
    }
}
```

Notice that even though I have an accumulator variable, I'm not actually setting the accumulator to the new value, I'm just returning it. That's because my responsibility is only to provide the new value of the accumulator, the Reduce class takes care of setting state.

2.5 Combining Expressions

These abstractions are powerful on their own, but even more so when combined. Let's say you want to concatenate all of the lowercase letters of a collection together. First, you use the Filter pattern to remove all uppercase letters. After, you apply the Reduce pattern to aggregate the elements into one string.

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.List;

import junit.framework.TestCase;

public class CombineTest extends TestCase {

    public void testCombine() {
      List strings = new ArrayList();
      strings.add("a");
      strings.add("e");
      strings.add("B");
      strings.add("s");
      strings.add("B");
```

```
        List newList = (List) Filter.overCollection(strings).remove(
          new BooleanExpression(){
          public boolean value(Object element) {
             return ((String)element).charAt(0) == 'B';
          }});

        String result = (String) Reduce.overCollectionWithInitialValue(newList,"").reduce(
           new ReduceExpression(){
           public Object value(Object element, Object accumulator){
             return accumulator.toString() + element.toString();
           }
        });
         assertTrue(result.equals("aes"));
      }
  }
```

At this point, it's my opinion that simplicity scale is leaning towards the high-order function approach.

3 Conclusion

High-order functions can be used to help separate orthogonal concerns. Map, Filter, and Reduce are responsible for iterating over the collection and collecting the results of the high-order function. The user of Map, Filter, and Reduce provides a high-order function that the collection needs.

The combination of static typing and the lack of concise high-order functions makes the use of Map, Filter, and Reduce very verbose in Java[1]. Martin Fowler blogged about how he tried to use Smalltalk-80 style closures in Java, but decided against their use because of the verbosity. I agree for the most part; just not in this case. I'm not going to try and define Smalltalk-80 style control structures using anonymous inner classes; that would be asinine. When I program in Java, I've got to control my Alien Hand Syndrome.

In my humble opinion - the use of high-order functions to simplify iteration on collections in Java a trade-off that you need to decide for yourself. The inherent complexity of the three patterns are there whether you use high-order functions or not. You can use high-order functions in Java to create orthogonal code, albeit verbose. Or you can write non-verbose code at the expense of having possibly buggy, repeated code throughout the code-base

[1] and we didn't even discuss the complexities of trying to use closures over non-final local variables

4 Map, Filter, and Reduce implementations

```
public class Map {
       private Collection collection;

       private Map(Collection collection)
       {
          this.collection = collection;
       }

       public static Map overCollection(Collection c)
       {
          return new Map(c);
       }

       public static Map overArray(Object[] array)
       {
          return overCollection(Arrays.asList(array));
       }

       public Collection collect(Expression e)
       {
                Collection result = new ArrayList();
                Iterator iterator = collection.iterator();
                while (iterator.hasNext()) {
                      Object item = e.value(iterator.next());
                      if (item != null) result.add(item);
                }
                return result;
       }
}

public class Filter {
       private Collection collection;

       private Filter(Collection collection)
       {
                this.collection = collection;
       }

       public static Filter overCollection(Collection c)
       {
                return new Filter(c);
       }

       public static Filter overArray(Object[] array)
       {
                return overCollection(Arrays.asList(array));
       }

       public Object select(BooleanExpression booleanExpression)
       {
                Collection result = new ArrayList();
                Iterator iterator = collection.iterator();
                while (iterator.hasNext()) {
                      Object item = iterator.next();
                      if (booleanExpression.value(item)) result.add(item);
                }
```

```
                return result;
        }

        public Object detect(BooleanExpression booleanExpression) {
                Iterator iterator = collection.iterator();
                while (iterator.hasNext()) {
                        Object item = iterator.next();
                        if (booleanExpression.value(item)) return item;
                }
                return null;
        }

        public Object remove(BooleanExpression booleanExpression)
        {
                Collection result = new ArrayList();
                Iterator iterator = collection.iterator();
                while (iterator.hasNext()) {
                        Object item = iterator.next();
                        if (!booleanExpression.value(item)) result.add(item);
                }
                return result;
        }
}

public class Reduce {
        private Collection collection;
        protected Object result;

        private Reduce(Collection collection, Object result)
        {
                this.collection = collection;
                this.result = result;
        }

        public static Reduce
            overCollectionWithInitialValue(Collection c, Object initialValue)
        {
                return new Reduce(c, initialValue);
        }

        public static Reduce
            overArrayWithInitialValue(Object[] array, Object initialValue)
        {
                return overCollectionWithInitialValue(Arrays.asList(array), initialValue);
        }

        public Object reduce(ReduceExpression e)
        {
                Iterator iterator = collection.iterator();
                while (iterator.hasNext()) {
                        result = e.value(result, iterator.next());
                }
                return result;
        }
}
```

5 Common Lisp Examples

Here are the same examples shown in ANSI Common Lisp. Some people say Lisp stands for Lots of Silly Parenthesis, and that Lisp is used for artificial intelligence work. The following shows a comparison of the number of parenthesis each implementation requires. And notice that nowhere in these examples am I solving an AI problem.

5.1 Map example

```
(mapcar #'char-upcase '(#\a #\e #\b #\s #\d))
;; => (#\A #\E #\B #\S #\D)

Number of parenthesis in Lisp: 4
Number of parenthesis in Java (minus assertations): 32
```

5.2 Filter example

```
(remove-if-not #'lower-case-p '(#\a #\e #\B #\s #\B))
;; => (#\a #\e #\s)

Number of parenthesis in Lisp: 4
Number of parenthesis in Java (minus assertations): 34
```

5.3 Reduce example

```
(reduce #'(lambda (s1 s2) (concatenate 'string s1 s2))
        (mapcar #'string '(#\a #\e #\b #\s #\d)))
;; => "aebsd"

Number of parenthesis in Lisp: 12
Number of parenthesis in Java (minus assertations): 30
```

5.4 Combination example

```
(reduce #'(lambda (s1 s2) (concatenate 'string s1 s2))
        (mapcar #'string
                (remove-if #'(lambda (c) (char= c #\B))
                           '(#\a #\e #\B #\s #\B))))
;; => "aes"

Number of parenthesis in Lisp: 20
Number of parenthesis in Java (minus assertations): 42
```