220

Iterators and HOFs

Dr Robert Chatley - rbc@imperial.ac.uk

@rchatley **#doc220**

```
List<String> list = Arrays.asList("quick", "brown", "fox");

void processList() {

    for (int i = 0; i < list.size(); i++) {
        String element = list.get(i);
        process(element);
    }

}
```

Let's look at a very simple loop over the contents of a list. Here we see that we are pulling a lot of the detail of the implementation out of the list. To operate the loop we are using an integer index into the array and revealing some of the detail - to some extent breaking encapsulation.

```
List<String> list = Arrays.asList("quick", "brown", "fox");

void processList() {

    Iterator<String> i = list.iterator();
    while(i.hasNext()) {
        String element = i.next();
        process(element);
    }
}
```

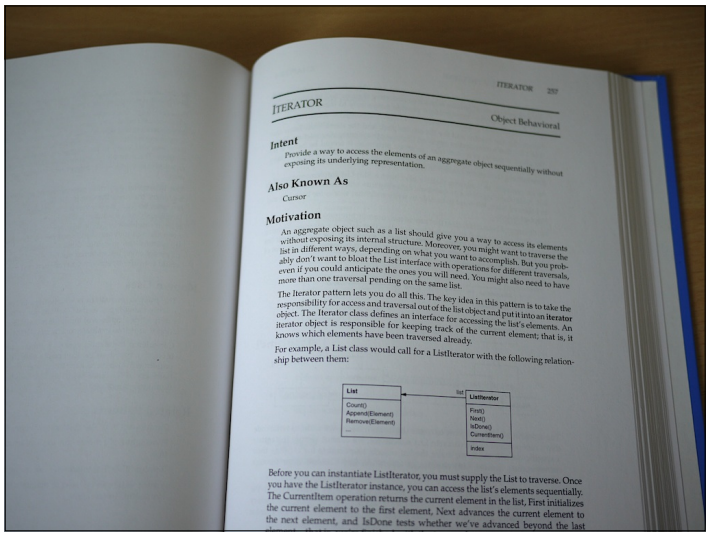
Here we see how we can use an iterator to abstract this, and to iterate through the list without using an integer index - but it's the same list and the same data underneath. Here we change the loop from a for() to a while(), and use the i.hasNext() and i.next() methods to traverse the data.

```
List<String> list = Arrays.asList("quick", "brown", "fox");

void processList() {

    for (Iterator<String> i = list.iterator(); i.hasNext(); ) {
        String element = i.next();
        process(element);
    }
}
```

Alternatively we can use the same iterator methods, but back in a for() structure. This is often a better formation, as the scope of the iterator variable is local to the loop, whereas in the while() example it is in scope for the rest of the method.



ITERATOR

Object Behavioral

Intent

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Also Known As

Cursor

Motivation

An aggregate object such as a list should give you a way to access its elements without exposing its internal structure. Moreover, you might want to traverse the list in different ways, depending on what you want to accomplish. But you probably don't want to bloat the List interface with operations for different traversals, even if you could anticipate the ones you will need. You might also need to have more than one traversal pending on the same list.

The Iterator pattern lets you do all this. The key idea in this pattern is to take the responsibility for access and traversal out of the list object and put it into an Iterator object. The Iterator class defines an interface for accessing the list's elements. An iterator object is responsible for keeping track of the current element; that is, it knows which elements have been traversed already.

For example, a List class would call for a ListIterator with the following relationship between them:

Before you can instantiate ListIterator, you must supply the List to traverse. Once you have the ListIterator instance, you can access the list's elements sequentially. The CurrentItem operation returns the current element in the list, First initializes the current element to the first element, Next advances the current element to the next element, and IsDone tests whether we've advanced beyond the last

The iterator is a common pattern - we saw it back in the first week in the Gang of Four Design Patterns book.

Iterator

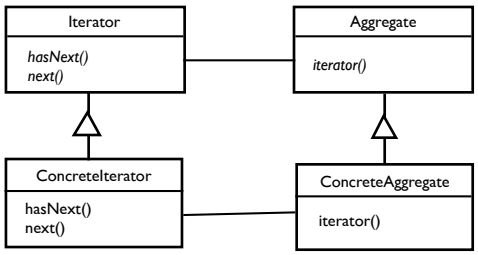
A way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Iterators can be *internal* or *external*.
Also known as a cursor.

#doc220

Standard collections in many languages have iterators, and we may write our own data structures and provide iterators over them. There are two main types of iterator, which we will look at: *internal* and *external*.

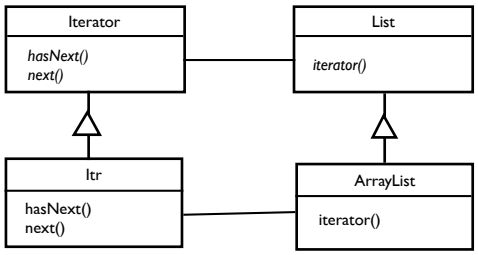
External Iterator: pattern



#doc220

First of all, the external iterator. This is the one that we saw in the Java example on the previous slides. We ask the aggregate (the collection) to give us an iterator, which returns an iterator implementation particular to that aggregate - i.e. it knows how to traverse the elements of the aggregate and expose them one at a time.

External Iterator: example



#doc220

In the ArrayList example, an ArrayList implements List, and as part of this contract must provide an Iterator. When you call the iterator() method on an ArrayList it returns an instance of an inner class ArrayList.Itr which implements Iterator and knows about the internal structure of the ArrayList.

```
List<String> list = Arrays.asList("quick", "brown", "fox");

void processList() {

    for (String element : list) {
        process(element);
    }
}
```

any Iterable can be looped-over in this way

Java also has a special for loop construct known as the foreach loop which can take not only any List, but any Iterable. A structure that is Iterable just has to provide an iterator() method returning an Iterator. All Java Collection types implement Iterable, but you can create your own datatype that is Iterable without it being a subtype of Collection.

Transforming a List

```
List<Integer> list = Arrays.asList(1, 2, 3, 4);

void processList() {

    for (int element : list) {
        square(element);
    }
}
```

Now, consider the situation where we have a list of values and we want to apply some transformation to each of the values - in the example above we have a list of integers and we want to square all of them.

```
class Squarer {

    List<Integer> list = Arrays.asList(1, 2 , 3, 4);

    List<Integer> squareAllValues(List<Integer> input) {
        List<Integer> result = new ArrayList<Integer>();
        for (Integer elem : input) {
            result.add(square(elem));
        }
        return result;
    }

    int square(int n) {
        return n * n;
    }

    void run() {
        List<Integer> results = squareAllValues(list);
        for (Integer result : results) {
            System.out.println(result);
        }
    }

}
```

The previous implementation didn't really do what we wanted - here's a fuller version. Here we can see that squareAllValues iterates over each value, applies the square() function to it, and aggregates the results in a new list, which is then returned. The actual mathematical function has been separated out into its own method, but it is still closely coupled to the iteration. What if we wanted to perform a different operation on the elements?

Passing a Function

```
square :: Int -> Int
square x = x * x

map square [1,2,3,4]
```

```
def square(x)
  x * x
end

list = [1,2,3,4]
list.map { | x | square x }
```

In Haskell we can use a higher-order function - map - define the square function as a first-class entity, and pass it to map as a parameter, for map to apply it to every element of the list. In this way the square() function and the map() function are defined separately and can be used separately.

In Ruby a method isn't a first class citizen like it is in Haskell, but we can pass a *block* as a parameter to another method. In the example above the block is defined in curly braces and says: given an x, square that x. The list type defines a map method that takes the block, applies it to each element, and returns a new list.

Passing a Function

```
function square(n) {
  return n*n;
}

list = [1,2,3,4];
list.map(square);
```

```
List<Integer> list = Arrays.asList(1, 2, 3, 4);

List<Integer> results = list.map(new Function() {
     int apply(int n) { return n * n; }
});
```

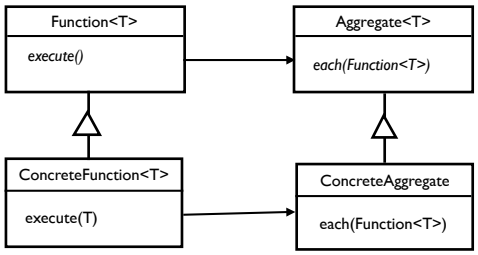
we would like to write something like this…

???

JavaScript actually behaves a bit more like Haskell in this case, as functions are first-class in JavaScript. To a large extent it can be considered a functional language. Arrays define a map() function which takes another function as a parameter.

If we wanted to apply this pattern in Java (before Java 8), it would be a bit more cumbersome. We'd have to write something like the above. Can we do that?

Internal Iterator: example

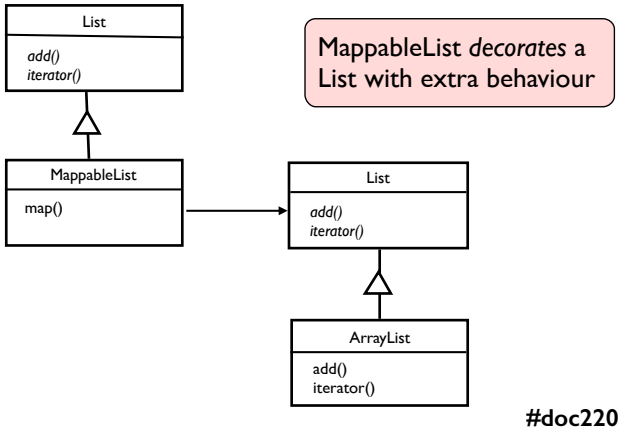


Pass a *strategy* for processing each element

#doc220

This pattern, of passing the operation in to the aggregate in order to be applied to each element is known as an *internal* iterator. Here we pass a Function to the Aggregate's each() (or map() etc) method, and have the aggregate iterate over the contents internally. This is a composition, and the function objects follow the Strategy pattern. There may be many different strategies for processing the same elements.

MappableList wraps List



MappableList *decorates* a List with extra behaviour

#doc220

We might want to add behaviour on to an existing class to make it work in this style. Here is an example where we create a MappableList, which is a List (and so is Iterable etc) but also supports map(). We make use of an ArrayList internally in the MappableList, and delegate most of the methods through to the internal one, but *decorate* the list's functionality with a map() method.

```
class MappableList<T> implements List<T> {

    private final List<T> delegate;

    MappableList(List<T> delegate) {
        this.delegate = delegate;
    }

    @Override
    public int size() {
        return delegate.size();
    }

    @Override
    public boolean isEmpty() {
        return delegate.isEmpty();
    }

    // etc, etc... delegate all public methods

    public List<T> map(Function<T> mapper) {
        List<T> result = new ArrayList<T>();
        for (T elem : delegate) {
            result.add(mapper.applyTo(elem));
        }
        return result;
    }

}
```

Here is an outline of a basic way that we could implement a map() method.

Libraries that implement a functional style for Java

Google's Guava:
https://github.com/google/guava

Daniel Worthington-Bodart's TotallyLazy:
https://totallylazy.com/

Here are some resources for fuller libraries of Java code written to support using this functional style. You might not want to write Java this way all of the time, but it is instructive to know that you can.
