21 Abstracting Traversals: Iterators

Introduction

We know by now that the same data may be represented in several different ways. We can save the information in a list, or in a binary search tree (BST), or maybe, there are other ways of keeping track of data. However, in each case we need to be able to perform the same basic operations. We can insert an item to a binary search tree, or add an item to a list, determine the number of elements in a tree or in a list, find the first item, or the structure that represents the rest of the items.

Another relevant observation is that over the time as we kept designing new methods for manipulating lists of data our classes kept growing and changing constantly. We had a choice of either carrying with us all the methods we already designed, or having several different variants of the same structures of data, each with methods relevant to our current problem.

Both of these problems suggest that we seek an abstraction. The first one suggests that the structure of the data and our way of interacting with this structure can be abstracted, so that we can interact in a uniform way with several different structures.

Abstracting over the structure of data

Suppose we defined an interface that allowed us to add items to a collection of data and to determine the number of elements in this collection. The interface would be:

```
// an interface to represent the construction of a data set
interface DataSet{

   // add the given object to this data set
   DataSet add(Object obj);

   // determine the number of elements in this data set
   int size();
}
```

It would be easy to modify our list structures and the binary tree structure to implement this interface. For the lists, we do the following:

```
// in the class ALoObj:

  // add the given object to this data set
  DataSet add(Object obj){
    return new ConsLoObj(obj, this);
  }

  // determine the number of elements in this data set
  abstract int size();

  // in the class MTLoObj:
  int size(){
    return 0;
  }

  // in the class ConsLoObj:
  int size(){
    return 1 + this.rest.size();
  }
```

though we may think of a better way of dealing with the size. For the BST the *add* method would be the same as the original *insert* and again, the size is the same as the count of nodes we designed earlier. However, now we can build either of these structures using the same methods in the same way.

We will see more of these kinds of abstractions when we discuss the *Java Collections Framework*.

Abstracting over the traversals

The second problem is a bit harder. Each method that processed a list of data engaged every element of the list in some computation, one element at a time. We may have been just counting them, of selecting those that satisfied some predicate, or producing a new value from the data contained in the original data element (map).

The typical structure of the program was:

```
// in the class ALoObj:
abstract Object method(...);
```

```
// in the class MTLoObj:
Object method(...){
  return baseValue;
}

// in the class ConsLoObj:
Object method(...){
  return result using
              ... (this.first) ...
              ... this.rest.method(...)...);
}
```

Not all parts were present for all problems, but it is clear that we needed to be able to process the first item and to have access to the rest. Let us recall similar methods in Scheme. The general structure of a function defined for a list-like data was:

```
(define (fcn alist)
  (cond
    [(empty? alist) ... produce base-value ...]
    [(cons? alist)  ... produce result using
                                  ... (first alist) ...
                                  ... (fcn (rest alist)) ...]))
```

We would like to be able to design a method in our *Examples* class that consumes a list structure and produces a result that is computed by examining each element of that list. We need to design a mechanism that will allow us to access the data in the list in the desired orderly fashion.

We start by defining the desired interface that allows us to observe the contents of a list-like structure:

```
// functional iterator for a linear traversal of a data structure
interface IRange{

  // is there a current element available in the structure
  boolean hasMore();

  // produce the current element of the structure
  Object current();

  // produce an iterator for the rest of this structure
  IRange next();
}
```

We have selected the names for these methods to be similar to the Java *Iterator* interface. Alternately, we could select the names of these methods to match the names for fields, methods, and scheme functions used in our programs. The alternative definition of this interface (providing the same information and functionality) would be

```
// functional iterator for  a linear traversal of a data structure
interface FIterator{

  // is the structure empty
  boolean isEmpty();

  // produce the first (current) element of the structure
  Object getFirst();

  // produce an iterator for the rest of this structure
  IRange getRest();
}
```

We will design a class that for a given list of *Object*s implements the IRange interface.

First, it must have access to the given list. That means, is will contain a field that represents the list we are traversing.

```
// represent the traversal of a list of Object-s
class AListRange implements IRange{

  ALoObj alist;

  AListRange(ALoObj alist){
    this.alist = alist;
  }
  ...
```

We know the class must contain all the methods defined in the *IRange* interface. We already have the purpose statements. Let us make some examples of the use of these methods:

```
ALoObj mtlist = new MTLoObj();
ALoObj list1 = new ConsLoObj("Hello", mtlist);
ALoObj list2 = new ConsLoObj("Bye",list1);

AListRange itmt = new AListRange(mtlist);
AListRange itl1 = new AListRange(list1);
AListRange itl2 = new AListRange(list2);
```

```
test("Test hasMore:", false, mtlist.hasMore());
test("Test hasMore:", true, itl1.hasMore());

test("Test current:", "Hello", itl1.current());
test("Test current:", "Bye", itl2.current());
// test("Test current:", "Error", itmt.current());

test("Test next:", itmt, itl1.next());
test("Test next:", itl1, itl2.next());
// test("Test next:", "Error", itmt.next());
```

Next, we need to design the method *hasMore*(). It should return true or false depending on whether *alist* is an instance of the *MTLoObj* or of the *ConsLoObj*. We get the following:

```
// is there a current element available in the structure
boolean hasMore(){
  return alist instanceof ConsLoObj;
}
```

We run into a bit of a trouble when designing the *current* method. If we knew that *alist* was an instance of *ConsLoObj*, the method could just produce *alist.first*. But is it is invoked with an empty *alist*, it is an error! Java has a mechanism for signalling errors of different kinds by *throwing Exceptions*. There is a whole hierarchy of classes that extend the base *Exception* class. For now we are only interested in errors that can be detected only when the program is running. There are all represented by subclasses of the class *RuntimeException*. The class *NoSuchElementException* seems to fit the problem we encounter when the program attempts to use the current element of an empty list, or to advance to the rest of the empty list. To signal the error we construct a new instance of the NoSuchElementException class and supply as the argument a message that explains the reason for the error:

```
// produce the current element of the structure
Object current(){
  if (alist.hasMore())
    return ((ConsLoObj)alist).first;
  else
    throw new NoSuchElementException(
                  "No current element available");
}
```

Producing the iterator that traverses the rest of *alist* follows the same pattern:

```
// produce the iterator for the rest of the structure
IRange next(){
  if (alist.hasMore())
    return ((ConsLoObj)alist).rest;
  else
    throw new NoSuchElementException(
                "No next iterator available");
}
```

We need to run our examples to see that the implementation of our iterator works correctly.

Designing methods with iterators

We are now ready to think about methods that consume a list of data. To make things more concrete, assume we want to determine whether the list contains the given element.

The method purpose and header will be:

```
// does the structure traversed with the given iterator
// contain the given object
boolean contains(IRange it, Object obj);
```

We make examples using the data defined earlier:

```
ALoObj mtlist = new MTLoObj();
ALoObj list1 = new ConsLoObj("Hello", mtlist);
ALoObj list2 = new ConsLoObj("Bye",list1);

AListRange itmt = new AListRange(mtlist);
AListRange itl1 = new AListRange(list1);
AListRange itl2 = new AListRange(list2);

test("Test contains:", false, contains(itmt, "Hello"));
test("Test contains:", true, contains(itl1, "Hello"));
test("Test contains:", false, contains(itl1, "Bye"));
test("Test contains:", true, contains(itl2, "Hello"));
test("Test contains:", false, contains(itl2, "Hi"));
```

Next we think of the template. There is no relevant object that invokes this method - as the method can easily be defined within any class that knows about the *AListRange*. The *Object obj* contains only the methods defined for all objects. We are interested in the method *equals*. The *IRange* interface does not provide any fields, only methods. But there is only one method we can invoke without any danger: *it.hasMore*(). However, if *it.hasMore*() produces true we have two additional methods available: *it.current*() and *it.next*(). The template then is:

```
... obj ...
... obj.equals(...) ...
... it ...
... it.hasMore() ...
    ... if ((it.hasMore())
        ... it.current() ...
        ... it.next() ...
        ... contains(it.next(), anyObject) ...
```

We can now complete the method body:

```
if (it.hasMore())
   if (obj.equals(it.current()))
      return true;
   else
      return contains(it.next(), obj);
else
   return false;
```

Comparison with the earlier design

To gain some insight into how this method commputes, we compare it to the earlier solutions, both within the *ALoObj* class hierarchy and written in Scheme. Within the *ALoObj* class hierarchy the method *contains* is designed as follows:

```
// in the class ALoObj:
abstract Object contains(Object obj);

// in the class MTLoObj:
Object contains(Object obj){
   return false;
}
```

```
// in the class ConsLoObj:
Object contains(Object obj){
  if (obj.equals(this.first)
    return true;
  else
    return this.rest.contains(obj);
}
```

It is clear that the clause following the true branch of **if** (*it.hasMore*()) corresponds to the code in the class *ConsLoObj* and the false branch corresponds to the *MTLoObj* case. We see the same when comparing our code to the function definition in Scheme:

```
(define (contains alist obj)
  (cond
    [(empty? alist) false]
    [(cons? alist)
      (cond
        [(equal? obj (first alist)) true]
        [else (contains (rest alist) obj)])]))
```

The empty clause produces false and the nonempty clause continues with another conditional that produces true if match has been found and otherwise recurs with the rest of the list.

```
// represent the traversal of a list of Object-s
class AListRange implements IRange{

  ALoObj alist;

  AListRange(ALoObj alist){
    this.alist = alist;
  }

  // is there a current element available in the structure
  boolean hasMore(){
    return alist instanceof ConsLoObj;
  }

  // produce the current element of the structure
  Object current();

  // produce an iterator for the rest of this structure
  IRange next();

}
```