22 Traversing ArrayList

Introduction

By now we are aware of the fact that there are many different ways to represent the same information. We have also seen that the same structure of data can represent all kinds of different information.

Java libraries, specifically the Java Collections Framework contain hierarchies of different classes and interfaces that can be used to represent data in many different forms. This allows the programmers to avoid mistakes, share code, and better understand programs written by others. We will learn about some of the classes and interfaces included in this framework, and will learn to read the documentation to understand how to use any of the others.

The first class from the framework we will use is ArrayList. It contains a list of data, similar to the lists we have build ourselves, and does not have a limit on the number of elements that can be added. Adding an element to an ArrayList mutates the list: the add method produces a boolean value that indicates success, but it does not produce a new ArrayList. Here is an example. We first construct an empty ArrayList:

```java
// using the constructor to build an empty ArrayList
ArrayList alist = new ArrayList();
```

Now, the test shows us that it is initially empty, and after adding three elements, its size grows as expected. The examples also show the use of the methods get(int index) and set(int index, Object value):

```java
public void runTests(){
    test("IsEmpty: ", true, alist.isEmpty());

    alist.add("Hello");
    alist.add("Good Day");
    alist.add("Goodbye");

    // currently the list has three elements
    test("IsEmpty: ", false, alist.isEmpty());
    test("Current size: ", 3, alist.size());

    // elements can be accessed directly via index
    test("element at 2: ", "Goodbye", alist.get(2));
    test("element at 1: ", "Good day", alist.get(1));
    test("element at 0: ", "Hello", alist.get(0));
}
```
We started with an empty ArrayList and added to it three String objects. Even though the method add produces a boolean value, we ignored this value, because we know it always produces true. The add method is included in an interface that is implemented by many other classes, some of which may not have the space available to add another element.

We then tested that the method isEmpty() produced false and that the number of elements in the ArrayList is indeed three.

The data in an ArrayList is arranged in a linear fashion, and each element has a numeric label called index. The first element we added to alist has index 0, the next one has index 1, etc. If we wish to refer to an element of an ArrayList we use the method get and specify the desired index. So, the test

```java
test("element at 0: ", "Hello", alist.set(0, "Hi"));
test("element at 0: ", "Hi", alist.get(0));
```

verified that the last element of alist was indeed "Goodbye".

The last two tests also illustrate the use of the method set(int index, Object value) that allows us to replace the element at a given location with a new one. This method returns the reference to the object that has been removed from the ArrayList.

**Designing IRange iterator for ArrayList**

We would like to use our IRange iterator to traverse over the elements in the ArrayList. To do so, we need to design a class ArrayListRange that implements IRange. The class needs at least one field — to hold the instance of the ArrayListRange it traverses. Here is a skeleton of this class:

```java
public class ArrayListRange implements IRange {
    ArrayList alist;
    … other fields if needed …

    public ArrayListRange(ArrayList alist, … ){
        this.alist = alist;
        …
    }
```
At the first glance it looks like `hasMore` is the easiest method to write. However, let’s hold off on that. For the `ListRange` the methods `current` and `next` mirrored the behavior of the first and rest field access. What is the `first` in an `ArrayList`? Well, is is the element at index 0, but only when we begin the traversal. The iterator produced by the `next` method should return the element at index 1 from it’s `current` method.

A few examples should help us understand the problem better. For the following `ArrayList`

```java
ArrayList alist = new ArrayList();
... // followed by initialization — inside some method: ...
alist.add("Hello");
alist.add("Good Day");
alist.add("Goodbye");
```

and the `ArrayListRange` iterator:

```java
IRange alistIt = new ArrayListRange(alist, ...);
```

we expect the following behavior:

```java
// elements can be accessed directly via index
test("element at 0: ", "Hello", alistIt.current());

IRange alisIt1 = alistIt.next();
test("element at 1: ", "Good day", alisIt1.current());

IRange alisIt2 = alisIt1.next();
test("element at 2: ", "Goodbye", alisIt2.current());
```
IRange alistIt3 = alistIt2.next();
test("end of the ArrayList ", false, alistIt3.hasMore());

It is clear that each ArrayListRange instance not only needs to know what is the ArrayList instance it is traversing, but also what is the current position in this traversal. That means, we should add a field to the class ArrayListRange that represents the current index. The hasMore method then determines whether the current index refers to a valid location in the ArrayList. Here is the complete class:

```java
public class ArrayListRange implements IRange{
    ArrayList alist;
    int index;

    // construct the IRange for the given ArrayList at the given index
    // index < 0 indicates no more elements to generate
    public ArrayListRange(ArrayList alist, int index){
        this.alist = alist;
        this.index = index;
    }

    // current element available if the index is valid
    public boolean hasMore(){
        return (this.index >= 0) ||
                (this.index < alist.size());
    }

    // throw exception if current element is not available
    public Object current(){
        if (this.hasMore())
            return alist.get(index);
        else
            throw new NoSuchElementException("No element is available.");
    }
}
```
// throw exception if no further iteration is possible
public IRange next() {
    if (this.hasMore())
        return new ArrayListRange(alist, index + 1);
    else
        throw new NoSuchElementException("Iterator cannot advance further.");
}

We can now run the tests to make sure our iterator works as expected.
We can also add the tests from the previous lecture that included the method contains:

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

We modify examples shown there to use an ArrayList for the data container:

    ArrayList mtlist = new ArrayList();
    ArrayList list1 = new ArrayList();
    ArrayList list2 = new ArrayList();

    public void runTests() {
        list1.add("Hello");
        list1.add("Hello");
        list2.add("Bye");

        IRange itmt = new ArrayListRange(mtlist, 0);
        IListRange itl1 = new ArrayListRange(list1, 0);
        IRange itl2 = new ArrayListRange(list2, 0);

        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"));
    }