Notes on Arrays and ArrayLists

March 31, 2009

These are lecture notes for CPSC 101-3 Winter 2009 (in particular 2009-04-01). These notes discuss topics from Chapter 7 of *Big Java* by Cay Horstman. 3rd edition.

1 A Comparison of Arrays and ArrayLists

In many ways arrays and `ArrayLists` are different. The former are built into the JAVA programming language, the latter are provided through a generic class in the `java.util` package. However, they are more similar than they are different, and perhaps it is worth starting with a comparison of the similarities. Table 1 shows how similar some operations are.

Arrays	`ArrayList<E>`
`T [] data =`	`ArrayList<T> data =`
`int [] data =`	*no direct equivalent*
`data[j] = x;`	`data.set(j,x);`
`x = data[j];`	`x = data.get(j);`
`data.length`	`data.size()`

Table 1: Common Features of Arrays and `ArrayLists`

The biggest difference between an array and an `ArrayList<>` is that an `ArrayList<>` object can change shape. In particular, an `ArrayList<E>`

offers the methods

`.add(x)`	add an `E` at the end.
`.add(i,x)`	add an `E` at spot `i`, shifting the previous contents from location `i` and higher upwards.
`.remove(i)`	remove the element at location `i`.
`.remove(x)`	remove the first occurrence of `x`, if present.
.clear()	empty the current contents

There are also methods to remove a range of data, or to insert an entire collection of data.

Although an array object cannot change shape, an array object variable can point at differently shaped objects over its lifetime. For instance,

```
double [] data = new double [500] ;
data = new double [30] ;
```

is perfectly legal.

2 Wrapper Classes

Each builtin type (`boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, and `long`) has a corresponding subclass of `Object`. Most of these classes are capitalized versions of the corresponding builtin type (*e.g.,* `Float`), with `Integer` and `Character` being the two exceptions.

The JAVA language now makes it very easy to convert between a builtin type and its corresponding object type; see section 7.3 of the text for details. Note that the objects of these classes are immutable, so that

```
Integer n = 35 ;
Integer m = n ;
++n ;
```

is actually safe. The compiler converts `++n` into `n = new Integer(1+ n.intValue ())`.

Arrays	`ArrayList<E>`
arrays can contain builtin types	`ArrayLists` must contain objects.
array *objects* have a fixed shape	`ArrayLists` can be grown and shrunk
array objects can have initial slots created by using a call `new Type[size]`	`ArrayList` objects can have initial space reserved by using a call `new ArrayList<Type>(size)`, but slots still must be created using the `.add()` method.
arrays implement no particular interface	`ArrayList<E>s` implement the interfaces `List<E>` and, indirectly, `Collection<E>`.
arrays are close to machine language and easier to interface with other programming languages through `native`	`ArrayList<>`'s have a certain amount of inefficiency, particularly with respect to builtin types.

Table 2: Differences between Arrays and `ArrayLists`

```
public static void removeOutliers(ArrayList<Double> data,
    double average, double stdDev)
    {
        int i,j,n ;
        n = data.size() ;
        for (i=0,j=0; j<n; ++j)
            {
                double zScore = (data.get(j)-average) / stdDev ;
                if (Math.abs(zScore) <= 3.0)
                    data.set(i++,data.get(j)) ;
            }
        for (;i<n;--n)
            data.remove(i) ;
    }
```

Figure 1: Removing Outliers in place

3 Examples

Here is the same problem solved thrice, once using arrays, and twice using `ArrayList<E>`s. The problem is to eliminate all of the data that is more than three standard deviations from some fixed value from an array.

First we do this in-place with an `ArrayList` as shown in Figure 1. The slightly tricky way of getting rid of the extra values is a little bit more efficient than the more straight-forward method shown in Figure 2. In Figure 1, it is always true that `i` is pointing at the next place that we want to put a value that we are keeping. After the loop, all of the values that we want to keep have been copied to a location less than `i`, so we can safely delete the rest of the data.

In Figure 2, we remove bad values as soon as we encounter them. Notice the idea of going backward when changing the shape.

What goes wrong if we write a forward-loop?

Finally, in Figure 3, we show how to handle the problem using builtin arrays. Here, we need to determine the size of the answer first, then cre-

```
public static void removeOutliers2(ArrayList<Double> data,
    double average, double stdDev)
    {
        int j,n ;
        n = data.size() ;
        for (j=n-1; j>=0; --j)
            {
                double zScore = (data.get(j)-average) / stdDev ;
                if (Math.abs(zScore) <= 3.0)
                    data.remove(j) ;
            }
    }
```

Figure 2: Another way to remove outliers in place

ate an array of the appropriate size, then copy the desired data into the answer.

```
public static double [] removeOutliers(double [] data,
    double average, double stdDev)
    {
        int i,j,n, m ;
        n = data.length ;
        m = 0 ;
        for (j=0; j<n; ++j)
            {
                double zScore = (data[j]-average) / stdDev ;
                if (Math.abs(zScore) <= 3.0)
                    ++m ;
            }
        double [] answer = new double [m] ;
        for (i=0,j=0; j<n; ++j)
            {
                double zScore = (data.get(j)-average) / stdDev ;
                if (Math.abs(zScore) <= 3.0)
                    {
                    answer[i++] = data[j] ;
                    }
            }
        return answer ;
    }
```

Figure 3: Removing outliers by copying