SIENAcollege
Computer Science

Computer Science 210
Data Structures
Siena College
Fall 2016

Topic Notes: Arrays Review

As you know from your prerequisite work, Java includes a fundamental programming language construct called an *array*.

In mathematics, we can refer to large groups of numbers (for example) by attaching subscripts to names. We can represent a set of numbers n_1, n_2,... An array lets us do the same thing with computer languages.

Suppose we wish to have a group of elements all of which have type `KindOfItem` and we wish to call the group `things`. Then we write the declaration of `things` as

```
KindOfItem[] things;
```

The only difference between this and the declaration of a single item of type `KindOfItem` is the occurrence of "`[]`" after the type.

Like all other objects, a group of elements needs to be created:

```
things = new KindOfItem[25];
```

Again, notice the square brackets. The number in parentheses (25) indicates the number of slots to create, each of which can hold one of the elements. We can now refer to individual elements using subscripts. However, in programming languages we cannot easily set the subscripts in a smaller font placed slightly lower than regular type. As a result we use the ubiquitous "`[ ]`" to indicate a subscript. If, as above, we define `things` to have 25 elements, they may be referred to as:

```
things[0], things[1], ..., things[24]
```

We start numbering the subscripts at `0`, and hence the last subscript is one smaller than the total number of elements. Thus in the example above the subscripts go from 0 to 24.

One warning: When we initialize an array as above, we only create slots for all of the elements, we do not necessarily fill the slots with elements. Actually, the default values of the elements of the array are the same as for instance variables of the same type. If `KindOfItem` is an object type, then the initial values of all elements is `null`, while if it is `int`, then the initial values will all be `0`. Thus you will want to be careful to put the appropriate values in the array before using them (especially before sending message to them! – that's a `NullPointerException` waiting to happen).

The following is an example of a Java application that uses arrays of `String`, `double`, and `int`.

See Example: GradeRangeCounter

There are a few items here we haven't used much this semester (the `Scanner`) but which you have seen before. There are also examples of arrays declared and initialized as `final`, and an example of an array of `int` allocated with `new`.

Inserting and Removing with Arrays

We have already seen that there is quite a bit to keep track of when using arrays, especially when objects are being added. We need to manage both the size of the array and the number of items it contains. If it fills, we either need to make sure we do not attempt to add another element, or reconstruct the array with a larger size.

As a wrapup of our initial discussion of arrays, let's consider two more situations and how we need to deal with them: adding a new item in the middle of an array, and removing an item from the end.

For these examples, we will not use graphical objects, just numbers. Arrays can store numbers just as well as they can store references to objects.

Suppose we have an array of `int` large enough to hold 20 numbers.

The array would be declared as an instance variable:

```
private int[] a;
```

along with another instance variable indicating the number of `int`s currently stored in `a`:

```
private int count;
```

and constructed and initialized:

```
a = new int[20];
count = 0;
```

At some point in the program, `count` contains 10, meaning that elements 0 through 9 of `a` contain meaningful values.

Now, suppose we want to add a new item to the array. So far, we have done something like this:

```
a[count] = 17;
count++;
```

This will put a 17 into element 10, and increment the `count` to 11.

But suppose that instead, we want to put the 17 into element 5, and without overwriting any of the data currently in the array. Perhaps the array is maintaining the numbers in order from smallest to largest.

In this case, we'd first need to "move up" all of the elements in positions 5 through 9 to instead be in positions 6 through 10, add the 17 to position 5, and then increment `count`.

If the variable `insertAt` contains the position at which we wish to add a new value, and that new value is in the variable `val`:

```
for (int i=count; i>insertAt; i--) {
   a[i] = a[i-1]
}
a[insertAt] = val;
count++;
```

Now, suppose we would like to remove a value in the middle. Instead of "moving up" values to make space, we need to "move down" the values to fill in the hole that would be left by removing the value.

If the variable `removeAt` contains the index of the value to be removed:

```
for (int i=removeAt+1; i<count; i++) {
  a[i-1] = a[i];
}
count--;
```

The loop is only necessary if we wish to maintain relative order among the remaining items in the array. If that is not important (as is often the case with our graphical objects), we might simply write:

```
a[removeAt] = a[count-1];
count--;
```

In circumstances where we are likely to insert or remove into the middle of an array during its lifetime, it usually makes sense to take advantage of the higher-level functionality of the `ArrayList`, which we will be considering soon.