ARRAYS AND FOR LOOPS

In many cases, processing an array means applying the same operation to each item in the array. This is commonly done with a `for` loop. A loop for processing all the elements of an array `A` has the form:

```
// do any necessary initialization
for (int i = 0; i < A.length; i++) {
   . . . // process A[i]
}
```

Suppose, for example, that `A` is an array of type `double[]`. Suppose that the goal is to add up all the numbers in the array. An informal algorithm for doing this would be:

```
Start with sum = 0;
Add A[0] to sum;   (process the first item in A)
Add A[1] to sum;   (process the second item in A)
  .
  .
  .
Add A[ A.length - 1 ] to sum;   (process the last item in A)
```

Putting the obvious repetition into a loop, this becomes:

```
double sum;  // The sum of the numbers in A.
sum = 0;     // Start with 0.
for (int i = 0; i < A.length; i++)
   sum += A[i];  // add A[i] to the sum, for
                 //     i = 0, 1, ..., A.length - 1
```

Note that the continuation condition, "`i < A.length`", implies that the last value of `i` that is actually processed is `A.length-1`, which is the index of the final item in the array. It's important to use "<" here, not "<=", since "<=" would give an array index out of bounds error. There is no element at position `A.length` in `A`.

Eventually, you should just about be able to write loops similar to this one in your sleep. I will give a few more simple examples. Here is a loop that will count the number of items in the array `A` which are less than zero:

```
int count;  // For counting the items.
count = 0;  // Start with 0 items counted.
for (int i = 0; i < A.length; i++) {
   if (A[i] < 0.0)   // if this item is less than zero...
      count++;       //     ...then count it
}
// At this point, the value of count is the number
// of items that have passed the test of being < 0
```

Replace the test "`A[i] < 0.0`", if you want to count the number of items in an array that satisfy some other property. Here is a variation on the same theme. Suppose you want to count the number of times that an item in the array `A` is equal to the item that follows it. The item that follows `A[i]` in the array is `A[i+1]`, so the test in this case is "`if (A[i] == A[i+1])`". But there is a catch: This test cannot be applied when `A[i]` is the last item in the array, since then there is no such item as `A[i+1]`. The result of trying to apply the test in this case would be an *ArrayIndexOutOfBoundsException*. This just means that we have to stop one item short of the final item:

```
int count = 0;
for (int i = 0; i < A.length - 1; i++) {
   if (A[i] == A[i+1])
      count++;
}
```

Another typical problem is to find the largest number in `A`. The strategy is to go through the array, keeping track of the largest number found so far. We'll store the largest number found so far in a variable called`max`. As we look through the array, whenever we find a number larger than the current value of `max`, we change the value of `max` to that larger value. After the whole array has been processed, `max` is the largest item in the array overall. The only question is, what should the original value of `max` be? One possibility is to start with `max` equal to `A[0]`, and then to look through the rest of the array, starting from `A[1]`, for larger items:

```
double max = A[0];
for (int i = 1; i < A.length; i++) {
   if (A[i] > max)
      max = A[i];
}
// at this point, max is the largest item in A
```

(There is one subtle problem here. It's possible in Java for an array to have length zero. In that case, `A[0]` doesn't exist, and the reference to `A[0]` in the first line gives an array index out of bounds error. However, zero-length arrays are normally something that you want to avoid in real problems. Anyway, what would it mean to ask for the largest item in an array that contains no items at all?)

As a final example of basic array operations, consider the problem of copying an array. To make a copy of our sample array `A`, it is **not** sufficient to say

```
double[] B = A;
```

since this does not create a new array object. All it does is declare a new array variable and make it refer to the same object to which `A` refers. (So that, for example, a change to `A[i]` will automatically change`B[i]` as well.) Remember that arrays are objects, and array variables hold pointers to objects; the assignment `B = A` just copies a pointer. To make a new array that is a copy of `A`, it is necessary to make a new array object and to copy each of the individual items from `A` into the new array:

```
double[] B = new double[A.length]; // Make a new array object,
                                   //   the same size as A.
for (int i = 0; i < A.length; i++)
   B[i] = A[i];   // Copy each item from A to B.
```

Copying values from one array to another is such a common operation that Java has a predefined subroutine to do it. The subroutine, `System.arraycopy()`, is a static method in the standard `System` class. Its declaration has the form

```
public static void arraycopy(Object sourceArray, int
sourceStartIndex,
        Object destArray, int destStartIndex, int count)
```

where `sourceArray` and `destArray` can be arrays with any base type. Values are copied from `sourceArray` to `destArray`. The `count` tells how many elements to copy. Values are taken from`sourceArray` starting at position `sourceStartIndex` and are stored in `destArray` starting at position `destStartIndex`. For example, to make a copy of the array, `A`, using this subroutine, you would say:

```
double B = new double[A.length];
System.arraycopy( A, 0, B, 0, A.length );
```

Source : http://math.hws.edu/javanotes/c7/s2.html