# Programming Exercise 8:
## Arrays

**Purpose:** Introduction to arrays

**Background readings from textbook:** Liang, Sections 7.1–7.2.

Due date for section 001: Monday, March 7 by 10 am (NOTE: this is during spring break)
Due date for section 002: Wednesday, March 9 by 10 am (NOTE: this is during spring break)

---

**Overview**

The variables that we have used in our programs only store single values. We refer to such variables as *scalar* variables. We are also interested in storing groups of values using what is referred to in Java as *Collection* types. The most common Collection type is called the *array*. Where a scalar stores one value of the declared type (e.g., int), the array stores many values of that declared type. If we think of a basic scalar variable as a box that is capable of holding a single object at a time, then we can think of an array as a sort of egg carton that is capable of holding numerous objects of the same type. Arrays can be defined to hold as many values of a given type as needed. Notice the restriction of the *same* type. This is known as a *homogenous* structure in that all items stored there must be the same type. We learn later of *heterogeneous* structures that can store multiple values of different types. In Python for instance, the array was called a tuple but the Python tuple was heterogeneous.

Why do we want to use arrays? Let's motivate this with a problem. Earlier in the semester we saw that with a while loop, we could input a list of numbers to compute the sum, and then after getting all the numbers, we could compute the average. Imagine that, given the average, we want to know how many of the inputs were less than the average. Remember in the while loop, we are using a single variable to input the value and add it to sum. We can only compute the average after we receive all of the values. But at that point, we will have discarded all of the inputs except the last one, so how can we go back and compare each input to the average to see how many are less than it? There are three solutions to this problem.

1. Ask the user to re-input the list again. This is inefficient not to mention frustrating to the user especially if there are a lot of values.
2. Store each input value in its own variable. Let's imagine that there are four inputs and we are going to store them in value1, value2, value3 and value4. We input the four, we compute the average as (value1+value2+value3+value)/4.0 and then we test each of value1 against average, value2 against average, etc, counting any that are less than. This is fine for four values. What if we need to do this with 10 values? Its more effort but still doable. What about 10,000 values? Now this solution is tiresome. What if, when we write the program, we don't know how many values there will be? It might be 1, it might be 10 million, it might be anywhere in between. Now this solution doesn't work.
3. Use an array whose size can vary based on how many values the user has.

So we want to use arrays because there are many problems where, upon inputting or computing a value, we need to retain the value for later portions of the program. Among the reasons for using an

array are to sort the list of items to be ordered and to later search through the list for values of interest. Searching and sorting are traditional programming problems and both usually require arrays. We focus on these in the next lab.

**Part 1: Java Examples**

In Java, arrays are objects and so you must declare the variable of the type and then create an instance of it (instantiate it) using the `new` command. To declare an array, use the notation *`type`*`[ ] `*`name`*`;` where `type` is the *type* of value you are storing in the array (e.g., int). When instantiating the array, use *`name`* `= new` *`type`*`[`*`number`*`];` where *type* is again that same type and where *number* is either a literal integer such as 10 or an int variable storing the size of the array. As with other Java declarations, you can combine the two into one statement. Here are two examples.

```
int[] values = new int[1000];           // using one statement with
                                        // a preset number of items

String[] names;                         // declaring first
System.out.print("How many names do you want to store?   ");
int x = input.nextInt();
names=new String[x];                    // instantiating later with
                                        // a size specified by the user
```

The array can be passed as a parameter to a method but otherwise to use the array, you must access one element of the array. You do this by *indexing* into the array. The notation is *`name`*`[`*`index`*`]` where *index* is either a literal integer or an int variable storing the location in the array that you want to access. Here is the tricky part though, the index is a number between 0 and size-1 rather than 1 to size. For instance, legal indices for the array `values` from above are 0 to 999, not 1 to 1000.

In order to simplify access into an array, we often use a loop. The following code would be used to input all n values into the `names` array from above and then print them out afterward in reverse order.

```
for(int i=0;i<n;i++)
{
       System.out.print("Enter the next name:   ");
       names[i]=input.next();
}
for(int i=n-1;i>=0;i--)
       System.out.println(names[i]);
```

Why do you suppose we are starting our counting at n-1 instead of n? See if you can figure that out.

Here is another example where we are inputting the array (in this case, an array of ints) from a separate method. We pass to the method two paramaters, the Scanner (we will call it `input`) and the number of items we expect to input into the array. Notice that the return type of the method is `int[ ]` meaning that it returns an array of ints.

```
public static int[] getArray(Scanner input, int numItems)
{
       int[] temp = new int[numItems];
       for(i=0;i<numItems;i++)
              temp[i]=input.nextInt();
       return temp;
}
```

Here is another example that's a bit trickier. In this code, we will store each letter of the alphabet (lower case) in 26 array elements of type char. The tricky part is understanding what `(char) ('a'+i)` does. See if you can figure it out.

```
char[] alphabet = new char[26];
for( int i = 0; i < 26; i++ )
       alphabet[i] = (char) ('a' + i );
System.out.println( "The 10th letter of the alphabet is "
                           + alphabet[9] + "." );
```

**Part 2: Common Pitfalls**

1.
```
int[] list = new int[100];
for( int i = 1; i <= 100; i++ )
       list[i] = 2*i + 1;
```

*Runtime error!*

*When the loop reaches i =100, the statement list[i] will cause a runtime error*

2.
```
public static int getArray(Scanner input, int size)
{
       int[] temp = new int[n];
       for(int i=0;i<size;i++)
          temp[i] = input.nextInt();
       return temp;
}
```

*Syntax error!*

*The return type should be int[ ] instead of int because we are returning temp, an entire array!*

**Part 3: Problem**

You will write a program to input a number of values into an array and then process the array by passing it to various methods that will compute and return information we are interested in. These include:

1. the average of the values in the array
2. the standard deviation of the values in the array
3. the number of items in the array less than the average
4. whether the array's values are in sorted increasing order or not

First, you need to write main. Your main method will declare all variables and then call all of the other methods. Your variables are as follows:

- The int array (you will need no more than 100, so instantiate the array to be new int[100])
- The number of items in the array
- A Scanner for input

Next, main will input from the user the number of values that the user wants to input. Store this in the number of items in the array. Verify that the number is between 1 and 100. If outside of that range, output an error message and end the program, otherwise continue. Next, main will call a method to input the values into the array. Pass to this method your Scanner and the number of items to input and the method will have a local variable for the array. It will input the array and return it

using a return statement. In main, you will have an assignment statement that might look like this where array is your int array, input is your Scanner and num is the number of elements to input.

```
array=getArray(input, num);
```

Then, main will call the other methods (listed in 1-4 above), passing it both the array and the number of elements. Those methods will return either a double (#1 and #2), an int (#3) or a boolean (#4). You can either store the resulting values in variables and output them later, or output them directly. Here are two examples assuming the array is called values and the number of elements is called num.

```
double standdev = getStdDev(values, num);
System.out.println("The standard deviation of the " + num +
     " values in the array is " + standdev);
```

or

```
System.out.println("The standard deviation of the " + num +
     " values in the array is " + getStdDev(values, num));
```

For method #4, you would either store the result in a boolean and use it in an if-else statement or call the method right from your if condition. For instance:

```
boolean sorted=isArraySorted(array, num);
if(sorted) System.out.println("...");
else System.out.println("...");
```

or

```
if(isArraySorted(array, num) System.out.println("...");
else System.out.println("...");
```

To compute the average, use a for loop that will iterate through the array adding each element to a sum variable and then compute and return the double value for sum/num.

The standard deviation first requires that you compute the average. Assuming our array is values and the average is avg, the standard deviation is computed as:

$$StdDev = \sqrt{\frac{(values[0]-avg)^2 + (values[1]-avg)^2 + \cdots + (values[num-1]-avg)^2}{n-1}}$$

Or, in Java, it would look like this:

```
Math.sqrt ((Math.pow((values[0]-avg),2) +
     Math.pow((values[1]-avg),2) + … +
     Math.pow((values[num-1]),2)) / n-1)
```

You will need to use a loop in this method to compute for each i from 0 to num-1 the value Math.pow(values[i]-avg),2) and add it to a sum. Then, compute Math.sqrt(sum/n-1). However, if n is 0 there are no values at all and if n is 1, you will get an error. So if n is not at least 2, return the value -1000. This will indicate that there were not enough values in the array to compute the stddev.

To determine the number of items in the array less than the average, pass the method the array, the number of elements and the average. Use a for loop and compare each a[i] to avg. Use an if statement in the for loop. If a[i]<avg, then you would add 1 to a counter. The method then returns the counter.

To determine if the array is in sorted order, you iterate through the array testing each pair of values, i and i+1. If you find that values[i]>values[i+1] for any i then the array is not already sorted. Here is

pseudocode for how to do this. Notice that the for loop iterates from 0 to num-2 instead of num-1. The reason for this is that the last item (at num-1) has nothing to be compared to.

```
for i from 0 to n-2
     if values[i]>values[i+1] return false;
          // found mismatch, stop immediately
return true;   // made it through the array
          // without a mismatch
```

Note that any output of double values should be formatted with DecimalFormat.

**Part 4: Test Your Program**

Run your program inputting the following values:

Number of items: 5

Items: 16 25 81 80 24

Output:

```
Average:           45.20
Std Dev:           32.41
Less than Avg:     3
Array is not in sorted order
```

**Part 5: Run Your Program and Submit Results**

Run your program on the following seven sets of inputs.

1. 5 inputs: 16 25 81 80 24 (the output for this is shown above to test)
2. 10 inputs: 1000 1001 1111 1222 1775 1776 1777 1888 1997 1998
3. 1 input: 1000 (this should cause an error message that it cannot compute standard deviation)
4. 10 inputs: 1 2 3 9 8 7 4 6 12 15
5. 0 inputs (this should cause error messages that the number of inputs is out of range)
6. 20 inputs: 21 3 20 5 30 1 33 14 17 18 10 4 48 50 22 25 6 47 8 19
7. 10 inputs: 8 7 8 7 8 7 9 8 9 7

Copy your outputs (not the inputs) for runs 2-7 and either paste the outputs (not the inputs) at the bottom of your source code in comments or paste them into a separate text file. Print your source code and outputs and hand them in or email them both to your instructor.