COP 3330 – Object Oriented Programming in Java
Arrays (of primitives and Strings)

Storing Many Values

All languages have a facility to store many items without using a different variable name for each. C has arrays and Python has lists. Java has arrays as well, but they do work a bit differently than either arrays in C or lists in Python.

All arrays are dynamically allocated (not primitives). All identifier names for arrays are references. (Every identifier name that isn't a primitive type is a reference type in Java.) To allocate an array of a particular primitive, we do the following:

```
type[] varname = new type[int_expr];
```

For example, if we wanted to make an integer array, allscores, of size 2n, where n is an integer variable storing a well-defined value, we would type:

```
int[] allscores = new int[2*n];
```

From this point on, we can use allscores exactly like a list or an array in other languages. Java does not support negative indexing, so you can only index an array from index 0 to the length of the array-1.

The way to access the length of an array is through a special member variable, length. The expression for the length of allscores is:

```
allscores.length
```

Once we start learning classes we'll use the dot operator a lot. Typically, the dot operator allows you to access a field in a class. An array isn't a class, it's its own thing, but we do use the dot operator for this purpose in this particular context.

In Java, unlike C, all arrays of numeric types are initialized to 0 without explicitly writing the code to do so.

Just like other languages, the typical way to process each item in an array is as follows:

```
for (int i=0; i<allscores.length; i++) {
    // Do something with allscores[i].
}
```

Here is a code snipped that adds up all of the values stored in an integer array:

```
int sum = 0;
for (int i=0; i<allscores.length; i++)
    sum += allscores[i];
```

Grades Example

As a static method, we could express this as follows:

```
public static int sumValues(int[] array) {
    int sum = 0;
    for (int i=0; i<array.length; i++)
        sum += array[i];
}
```

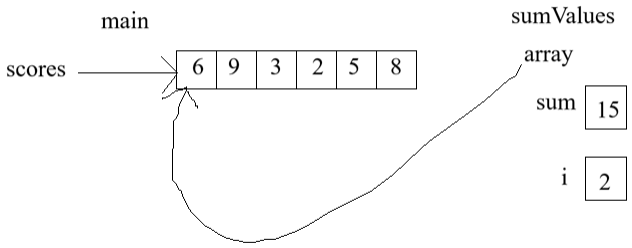
Now, to test this method to see that it actually works, we'll write a print method, which prints out each value in an array:

```
public static void print(int[] array) {
    for (int i=0; i<array.length; i++)
        System.out.print(array[i]+" ");
    System.out.println();
}
```

We can write a main method to test the sumValues method. Here's our bare minimum program to test this code above:

```
import java.util.*;
public class testscores {

    public static void main(String[] args) {

        Scanner stdin = new Scanner(System.in);
        int n = stdin.nextInt();

        int[] scores = new int[n];
        for (int i=0; i<n; i++)
            scores[i] = stdin.nextInt();

        print(scores);
        int sum = sumValues(scores);
        System.out.println("Sum of values = "+sum);
    }

    public static int sumValues(int[] array) {
        int sum = 0;
        for (int i=0; i<array.length; i++)
            sum += array[i];
    }

    public static void print(int[] array) {
        for (int i=0; i<array.length; i++)
            System.out.print(array[i]+" ");
        System.out.println();
    }
}
```

Let's take a quick look at the picture of how the method call occurs and the mechanics of passing an array to a method:



This snapshot is while the method sumValues is executing right after it finishes the loop iteration where i = 1 and i has been incremented to 2. Notice that since an array isn't a primitive, we must pass a variable name that is a reference to an array to the sumValues method, and then the formal parameter, array, will reference the array from the main method passed to it.

We'll extend this example as follows: find the average of the data, the standard deviation of the data, and we'll apply two different types of changes to an original set of grades. We'll also pay careful attention to how methods build upon themselves. Now that we've written a sumValues method, an average method is quite simple:

```
public static double average(int[] array) {
    return sumValues(array)*1.0/array.length;
}
```

This is typical with methods, where you call one from another, building upon its work.

Now, let's build on the average method to help calculate standard deviation. The standard deviation of a set of numbers is the square root of the variance of those numbers. The definition of variance of a set of numbers is to take the difference of each number from the average, square that difference and add those differences up. Then, take that sum and divide by the total number of values. We'll look at the code in a second, but before we do, let's see a different way to loop through the values in an array:

```
for (int x: array) {
    // refer to x in here, but you can't change
    // the values in array.
}
```

Here's the code for the standard deviation method:

```
public static double stddev(int[] array) {

    double avg = average(array);
    double total = 0;
    for (int x: array)
        total += Math.pow(x-avg, 2);
```

```
    return Math.sqrt(total/array.length);
}
```
Finally, let's investigate two methods teacher's use to remap grades to different percentages:

1. "Curve" to highest grade
2. "Compression

and write methods to help us achieve these modifications to scores.

The idea of a curve is that you take the highest grade in the class and "make it a 100." So let's say the highest grade in the class was a 93. To make this a 100, we would add 7 to it. So, to curve the grades, we'll add 7 to everyone's grade. To algorithmically, we need to do the following:

1. Find the maximum grade.
2. Subtract the value from step 1 from 100.
3. Add the value from step 2 to all the test scores.

Step 1 is a classic task, so let's write a method for it, using a bit of help from the Math method max:

```
public static int maxValue(int[] array) {
    int res = array[0];
    for (int i=1; i<array.length; i++)
        res = Math.max(res, array[i]);
    return res;
}
```

Again, notice the similar structure to other methods. One difference is that we initialized our answer to the first array value, so that our loop starts at the following index (1) instead of 0. Alternatively, we could have written the body of the for loop as:

```
if (array[i] > res)
    res = array[i];
```

Now that we have this method written, we can go ahead and write our curve method, calling the maxValue method:

```
public static void curve(int[] scores) {

    int top = maxValue(scores);
    int toAdd = 100 - top;

    for (int i=0; i<scores.length; i++)
        scores[i] += toAdd;
}
```

Our final method will be compressing the scores. This also takes the idea of looking at the difference between a grade and 100, but applies a different adjustment to each grade. For each grade x, take it's difference from 100, so 100 – x. Now, divide this value by 2, using integer

division. Finally add this adjusted value to the score. It's called a compression because it's compressing the difference between each grade and the theoretical top grade by 1/2.

Here's the code that accomplishes this grade adjustment:

```
public static void doubleCompress(int[] scores) {

    for (int i=0; i<scores.length; i++) {
        int from100 = 100 - scores[i];
        scores[i] += from100/2;
    }
}
```

For these last two examples, notice that when we call these methods from another method (say main), these methods have references to the original array and make the changes directly in that original array, which is precisely what makes these methods useful.

Letter Frequency Example

Let's look at another example with an integer array where we count up the frequency of each of the letters in a string. We'll make our method take in a string and return an integer array of size 26, where index 0 stores the number of 'a's, index 1 the number of 'b's, etc., and index 25 stores the number of 'z's. The key is understanding that Ascii values for letters are stored contiguously and that we can just subtract Ascii values to get the relative 0 to 25 numeric value. Here is the method:

```
public static int[] getFreq(String s) {
    s = s.toLowerCase();
    int[] freq = new int[26];
    for (int i=0; i<s.length(); i++) {
        if (s.charAt(i)>='a' && s.charAt(i)<='z')
            freq[s.charAt(i)-'a']++;
    }

    return freq;
}
```

The if statement just makes sure that we don't get an array out of bounds error with a non-letter character. The toLowerCase method has to be called on the String (doesn't exist for the char because char is a primitive), so we do that to avoid case work.

Now, it might be fun to view this data in a bar graph. One bar graph would be horizontal, and this is pretty easy to print out. We just go letter by letter, and for each letter, we print a star for each occurrence of it.

Here, we have to go from integer (0 to 25) back to character. If the integer i stores the 0 to 25 value, here is the expression for the corresponding lowercase character:

```
(char)('a'+i)
```

On the next page, we can see the method that prints the bar graph of the data:

```
// Pre-req: freq is a frequency array of size 26, where each
// value represents the number of a's, b's, ..., z's in a String.
public static void printBarGraph(int[] freq) {
    for (int i=0; i<26; i++) {
        System.out.print((char)('a'+i)+"\t");
        for (int j=0; j<freq[i]; j++)
            System.out.print("*");
        System.out.println();
    }
}
```

We separate our printing of each row into three different tasks:

1. Printing the row header (letter).
2. Printing the stars representing the frequency of that letter.
3. Printing the newline character.

For even more fun, we can print a vertical bar graph. For this one, the labels are at the bottom and the bars go up and down. The tricky part here is that when we print a line, we must print some spaces (for less frequent characters) and some stars. What we can do is loop backwards through the frequencies and use an if statement to determine, if for a particular letter and a particular number, whether or not we should print a star or a space. In addition, we need to find the maximum value in the frequency array, so we can call our old maxValue method to help us.

Here's that code:

```
// Pre-req: freq is a frequency array of size 26, where each
// value represents the number of a's, b's, ..., z's in a String.
public static void printVerticalBarGraph(int[] freq) {

    // i represents the number of stars for this row.
    for (int i=maxValue(freq); i>0; i--) {

        for (int j=0; j<26; j++) {
            if (freq[j] >= i)
                System.out.print("*");
            else
                System.out.print(" ");
        }
        System.out.println();
    }

    for (int i=0; i<26; i++)
        System.out.print((char)('a'+i));
    System.out.println();
}
```

Notice that the maxValue method is useful to help us figure out the top row of the chart. We can start at that integer and count down. Then, for each "scan line", we loop through the 26 letters, and for each letter, we ask if it appeared at least or more times than the current row number we're at. If so, we print a star, if not, we'll print a space. Finally, like always, at the end of the row, we print a newline character. At the very bottom, we have our letter column headers.

Example: Arrays of Strings (and Arrays class documentation)

Let's conclude this lecture with a short example showing the usefulness of an array of Strings, and along the way, we'll learn about a very convenient built in method.

This program will ask the user to enter several names (for now we'll do all lowercase letters only) and will sort those names in alphabetical order.

In the program we'll first ask the user how many names they will enter. Then we'll create the array:

```
Scanner stdin = new Scanner(System.in);
System.out.println("How many names to enter?");
int n = stdin.nextInt();

String[] names = new String[n];
```

From here, let's ask the user to enter the names and read them in one by one:

```
System.out.println("Please enter, one name per line.");
System.out.println("Lowercase letters only.");
for (int i=0; i<n; i++)
    names[i] = stdin.next();
```

There is a class called Arrays that has some methods for Arrays. Here is a partial listing of those methods:

Modifier and Type	Method	Description
`static int`	`binarySearch(int[] a, int key)`	Returns the index where key is found in the sorted array a, returns a negative integer otherwise. (Note: The full documentation indicates which negative integer is returned.)
`static int[]`	`copyOf(int[] original, int newLength)`	Returns a new copy of the array original with the length newLength. If newLength is less than the original, the array is truncated. Otherwise, null values (0) are filled into the extra spots in the new array.
`static boolean`	`equals(int[] a, int[] a2)`	Returns true if both a and a2 are the same length and all corresponding entries are the same value. (a[0]=a2[0], a[1]=a2[1], etc.)
`static void`	`fill(int[] a, int val)`	Sets all values in the array a to val.
`static void`	`sort(int[] a)`	Sorts the values of in a in ascending order.
`static void`	`sort(Object[] a)`	Sorts the specified array of objects into ascending order, according to the natural ordering of its elements.

Notice that there's no array of Strings listed. But, Object is a Java class from which ALL other classes inherit. Thus, this means we can use the sort method on any object in its "natural ordering." For Strings, that built in ordering is called lexicographical ordering via Ascii value. Since in our example we'll ask the user to only use lowercase letters, lexicographical ordering will mean alphabetical ordering.

To sort, we'll just do this:

```
Arrays.sort(names);
```

The class Arrays resides in java.util, so our usual package import will work just fine. Here's the completed program:

```
import java.util.*;

public class SortNames {

    public static void main(String[] args) {

        // Get the number of names.
        Scanner stdin = new Scanner(System.in);
        System.out.println("How many names to enter?");
        int n = stdin.nextInt();

        // Make the array of Strings.
        String[] names = new String[n];

        // Read in the names.
        System.out.println("Please enter, one name per line.");
        System.out.println("Lowercase letters only.");
        for (int i=0; i<n; i++)
            names[i] = stdin.next();

        // Sort them.
        Arrays.sort(names);

        // Printout the sorted list.
        System.out.println("Here is the list of names sorted.");
        for (int i=0; i<n; i++)
            System.out.println(names[i]);
    }
}
```

Arrays can be used for many, many things, but these are two classical examples that help illustrate both the flexible use of arrays and the benefit of writing methods that operate on arrays and build on each other.