U6L1: Array Creation and Access

One Variable, Many Values

We've all written code like this before:

```java
int score1;
int score2;
int score3;
```

This can get tedious, and the only thing associating the related data are the identifiers chosen by the programmer. There is nothing internal to Java linking these values.
Arrays allow us to organize related data into a single variable, for example:

```java
int[] scores;
```

We are able to store as many scores as we need in this single variable.

Declaring an Array

We declare an array in Java by adding square brackets **[]** after the data-type and assigning an identifier to the array. In the example below we declare an array of `int`s with the name `testScores`

```java
int[] testScores;
```

Arrays can contain any Java data-type, as long as all the values are the same type. Arrays can contain primitives or reference values.

```java
double[] heights; // primitive
String[] names; // reference
Color[] colors; // reference
```

Creating an Array

Arrays in Java are fixed in size. They *cannot* grow and shrink like arrays in other languages. This means we must declare the size when an array is first created. To create an array we use the `new` keyword followed by the type with the size of the array enclosed in square brackets.

```java
int[] scores = new int[3];
```

When the program is run, the JVM will set aside memory to store three integer values. These values will default to `0` initially.

Default Values in Java Arrays

numeric types: 0
booleans: false
objects: null

Accessing values of an array

The individual elements of an array can be accessed with square brackets and an *index* representing the position of the element in the array. The first element of an array is always index 0.

```java
// declare new int array with 3 elements
int[] scores = new int[3];

// assigns values to the scores array
scores[0] = 76;
scores[1] = 100;
scores[2] = 83;

// print the first value
System.out.println(scores[0]);

// assign the second value to a new variable
int n = scores[1];
```

Valid index range

An array may be indexed with an integer between 0 and 1 less than the length of the array. Any index outside of this range will result in a `ArrayIndexOutOfBoundsException`.

```
int[] arr = new int[3];
arr[0] = 10; // sets first element to 10
arr[-1] = 5; // will NOT compile (negative)
arr[3] = 5; // will NOT compile (greater than length - 1)
```

length attribute: every Java array comes with a built in *final* attribute called *length* that holds the amount of elements in the array. *NOTE:* This is not a method `.length()` like it is when we get the length of a String, it is an attribute. Therefore, we do not include parentheses.

```
int[] arr = new int[10];
System.out.println(arr.length); // will print 10
```

Using an array initializer

An *array initializer* is a shortcut for creating arrays in Java. Array initializers consist of curly brackets with comma separated values.

```java
int[] scores = {76, 100, 83};
```

This does the same thing as the code in the previous slide. Java infers the size of the array by the number of elements inside the brackets. It is interesting to note that `{76. 100, 83}` is used to create an integer array object however it is **not** an actual array itself (*Java does not have array literals*). This distinction is most noticable when using methods that take arrays as parameters, or return arrays. See example below:

```java
public class ArrayDemo {

    public static int getFirst(int[] arr){

        // code in class

    }

    public static double[] randomTriplet() {

        // code in class
    }

    public static void main(String[] args) {

        // code in class
    }
}
```

Arrays are objects

Arrays in Java are objects, even when the elements they contain are primitive. This can lead to some confusion if we forget that the array variable contains a reference value. Consider the following example:

```java
String[] names = {"Shaggy", "Scooby", "Velma", "Daphne", "Fred"};
```

Suppose I want to make my own copy of the array doing the following:

```java
String[] myNames = names;
```

Now if I want to replace the name "Scooby" with "Scrappy" I could write:

```java
myNames[1] = "Scrappy";
```

What do you think the following will print?

```java
System.out.println(myNames[1]);
System.out.println(names[1]);
```

Using the .clone() method

In the previous example, `names` and `myNames` were aliases to the same array. Changes to 1 affected the other. When we want to create a true copy of an array, we utilize the `.clone()` method.

```
String[] names = {"Shaggy", "Scooby", "Velma", "Daphne", "Fred"};

String[] myNames = names.clone();
myNames[1] = "Scrappy";

System.out.println(myNames[1]);
System.out.println(names[1]);
```

LAB 022 Starter Code

```java
import java.util.Arrays;

public class Student{

    public static final int NUM_GRADES = 5;
    private String name;
    private int[] grades;

    // constructor

    // methods

}
```

LAB-022 TODO: Student Class

Create the following constructor and instance methods

Constructor

Student(String name): takes a string used to set the `name` attribute. Initialize the `grades` array here. Use `NUM_GRADES` to set the size.

Instance Methods

getGrade(): Takes an integer representing the position in the `grades` array to return.
setGrade(): Takes two integers. The first represents the position in the array to set and the second represents the value of the grade (no validation required).
improved(): Return whether or not the Student's last grade was greater than their first grade.
getAverage(): Return the average of the 5 grades in the array as a double.
droppedMin(): Will return a new array of integers containing only 4 grades with the minimum grade dropped. *HINT:* This problem is easier if we sort the array. You may use the static method `Arrays.sort(arr)` where `arr` is the name of the array you are sorting. **NOTE:** DO NOT sort the `grades` array directly or our `improved()` method will be useless! Instead you must make a copy of the array first.

LAB-022 TODO: Test Code

Write a separate test class to verify your methods.

```java
public class StudentTest{
    public static void main(String[] args){
        // create two students

        // populate both students 5 grades using the setGrade() method
        // make sure 1 student "improves" and 1 does not
        // make sure 1 student has two min grades

        // test your getter by calling the getGrade() method at least once

        // test your improved() method by calling it on each student

        // test your getAverage() method by calling it on each student

        // test your droppedMin() by calling it on each student and printing the array
        // feel free to use Arrays.toString(arr) to print.

    }
}
```