



U6L1: Array Creation and Access

One Variable, Many Values

We've all written code like this before:

```
1 int score1;  
2 int score2;  
3 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:

```
1 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 `ints` with the name `testScores`

```
1 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.

```
1 double[] heights; // primitive
2 String[] names; // reference
3 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.

```
1 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.

```
1 // declare new int array with 3 elements
2 int[] scores = new int[3];
3
4 // assigns values to the scores array
5 scores[0] = 76;
6 scores[1] = 100;
7 scores[2] = 83;
8
9 // print the first value
10 System.out.println(scores[0]);
11
12 // assign the second value to a new variable
13 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`.

```
1 int[] arr = new int[3];
2 arr[0] = 10; // sets first element to 10
3 arr[-1] = 5; // will NOT compile (negative)
4 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.

```
1 int[] arr = new int[10];
2 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.

```
1 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:

```
1 public class ArrayDemo {
2
3     public static int getFirst(int[] arr){
4
5         // code in class
6
7     }
8
9     public static double[] randomTriplet() {
10
11         // code in class
12     }
13
14     public static void main(String[] args) {
15
16         // code in class
17     }
18 }
```

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:

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

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

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

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

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

What do you think the following will print?

```
1 System.out.println(myNames[1]);  
2 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.

```
1 String[] names = {"Shaggy", "Scooby", "Velma", "Daphne", "Fred"};
2
3 String[] myNames = names.clone();
4 myNames[1] = "Scrappy";
5
6 System.out.println(myNames[1]);
7 System.out.println(names[1]);
```

LAB 021

Follow along in class creating the starter code for the `Student` class to this activity.

Your Task: Add the following 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 grade to set to (no validation required).

improved(): Will return whether or not the Student's last grade was greater than their first grade.

droppedMin(): Will return a new array containing only 4 grades with the minimum grade dropped.

Starter Code:

// Complete in Class