



Middlebury

CSCI 201: Data Structures

Spring 2025

Lecture 2M: Arrays

Goals for today:

- Do a mini-introduction to objects and use the **new** keyword.
- Create and use fixed-size arrays to store many values of the same type.
- Modify objects by using a **reference** to the underlying object.
- Create and use multi-dimensional arrays.
- Use the debugger to inspect the value stored in variables.



Remember the **String** example from last class? Here's a variation.

```
1 public class StringEquals {  
2     public static void main(String[] args) {  
3  
4         String s = "Middlebury";  
5         String a = s.substring(0, 4); // "Midd"  
6         String b = "Midd";  
7  
8         boolean sameByEqualityOperator = (a == b);  
9         boolean sameByEqualsMethod = a.equals(b);  
10  
11        System.out.println("sameByEqualityOperator: " + sameByEqualityOperator);  
12        System.out.println("sameByEqualsMethod: " + sameByEqualsMethod);  
13    }  
14 }
```

Why did this happen? What is **==** actually comparing?

A **String** is a class. An instance of a **String** is an **Object**.

We have been creating **Strings** like this (special syntax for **String**s):

```
1 String s = "Middlebury";
```

But we can also create **Strings** like this:

```
1 String s = new String("Middlebury");
```

Because strings are *objects*.

- Objects are created using the **new** keyword.
- When objects are created, the resulting variable is a *reference variable*.
- A reference variable is really an address to a place in memory where the object is stored.
- We can pass around this address to make changes to the underlying object.
- The **==** operator will check if the addresses are the same.

Back to our **String** example.

```
1 public class StringEquals {  
2     public static void main(String[] args) {  
3  
4         String a = new String("Midd");  
5         String b = new String("Midd");  
6  
7         boolean sameByEqualityOperator = (a == b);  
8         boolean sameByEqualsMethod = a.equals(b);  
9  
10        System.out.println("sameByEqualityOperator: " + sameByEqualityOperator);  
11        System.out.println("sameByEqualsMethod: " + sameByEqualsMethod);  
12    }  
13 }
```

Introducing fixed-size arrays! Size is fixed upon creating the array.

Unlike lists in **Python**,
where the size can change.



```
1 values = []
2 for i in range(10):
3     values.append(i)
```

We'll see dynamically-sized arrays in **Java** next week.

Fixed-sized arrays can be used to store multiple values of the same type.

```
1 public class ArrayExamples {  
2     public static void main(String[] args) {  
3  
4         int[] values1 = new int[6]; // creates an array of integers with 6 values  
5         values1[0] = 311;  
6         values1[1] = 312;  
7         values1[2] = 315;  
8         values1[3] = 318;  
9         values1[4] = 333;  
10        values1[5] = 467;  
11  
12        int[] values2 = {311, 312, 315, 318, 333, 467};  
13        int[] values3 = new int[]{311, 312, 315, 318, 333, 467};  
14    }  
15 }
```

Retrieve the length using **.length**.

We're not calling a method! So no **()**. We're accessing a property.

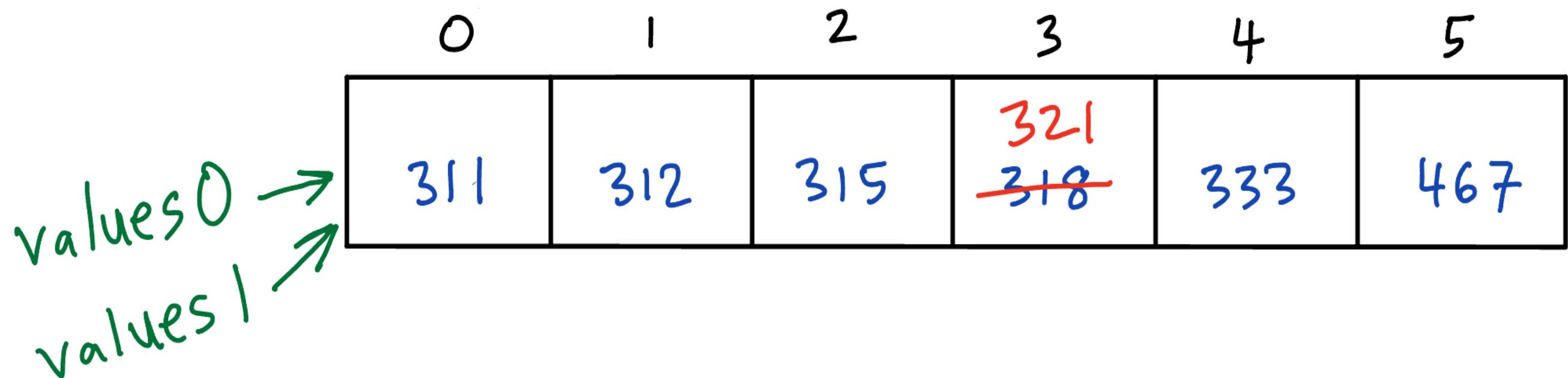
Enhanced **for**-loops (or *for-each* loops)

```
1 int[] intValues = {311, 312, 315, 318, 333, 467};  
2 for (int value : intValues) {  
3     System.out.println("value = " + value);  
4 }  
5  
6 String[] stringValues = {"apple", "banana", "pear", "orange"};  
7 for (String value : stringValues) {  
8     System.out.println("value = " + value);  
9 }
```

This can be done with objects that are *iterable* (like arrays).

The variables we declare for arrays are also reference variables.

```
1 int[] values0 = {311, 312, 315, 318, 333, 467};  
2  
3 int[] values1 = values0; // values1 has the same reference as values0  
4  
5 values1[3] = 321; // change item in values1  
6  
7 System.out.println(values0[3]); // change is reflected in values0
```



Multi-dimensional arrays: can be rectangular or jagged.

```
1 // rectangular: all rows have the same size
2 int nRows = 10;
3 int nCols = 20;
4 int[][] matrix = new int[nRows][nCols];
5 for (int i = 0; i < nRows; i++) {
6     for (int j = 0; j < nCols; j++) {
7         matrix[i][j] = 1; // some value
8     }
9 }
10
11 // jagged: each row has a different size
12 char[][] triangle = new char[nRows][]; // nRows arrays, initially null
13 for (int i = 0; i < triangle.length; i++) {
14     triangle[i] = new char[i + 1];
15     for (int j = 0; j < triangle[i].length; j++)
16         triangle[i][j] = '*';
17 }
```

Using the VS Code debugger

The screenshot shows the VS Code interface with the following components:

- Code Editor:** Displays a snippet of Java code with a red dot at line 41 indicating a breakpoint. A tooltip says "Click to add a breakpoint".
- Toolbar:** Shows standard menu items: Go, Run, Terminal, Window, Help, along with icons for file operations like Open, Save, and Close.
- Run View:** A dropdown menu under the Run tab containing various debugging commands: Start Debugging (selected), Run Without Debugging, Stop Debugging, Restart Debugging, Open Configurations, Add Configuration..., Step Over, Step Into, Step Out, Continue, Toggle Breakpoint, New Breakpoint, Enable All Breakpoints, Disable All Breakpoints, and Remove All Breakpoints.
- Variables Explorer:** A sidebar titled "VARIABLES" showing the "Local" scope. It lists variables and their values:
 - args = String[0]@8
 - values = int[6]@9
 - 0 = 311
 - 1 = 312
 - 2 = 315
 - 3 = 318
 - 4 = 333
 - 5 = 467
 - intValues = int[6]@10
 - stringValues = String[4...]
 - values0 = int[6]@12
 - values1 = int[6]@12
 - nRows = 10
 - nCols = 20
 - matrix = int[10]@13
 - i = 0
 - j = 0

Group exercises

See Practice 3 problems