

Arrays and Strings

Table of Contents

- Arrays
 - Array Methods and Tools
 - Array Iteration
 - Multi-dimensional arrays
 - Arrays of Objects
- Strings
 - Basic string operations
 - Substrings
 - String concatenation
 - String equality and references
 - String Modification

Arrays

- Array documentation
- Declaration: Brackets can be attached to the *type* or the *variable_name*

```
1 int[] nums;  
2 int nums[];
```

- Allocation: declaring an array doesn't initialise it, so you first need to allocate it
 - can use the **new** operator, declare an array of **<type>** values, storing up to **<size>** elements
 - primitives are initialised to "zero" (**int**: 0, **double**: 0.0, ...)
 - objects: initialised to **null**

```
1 <type>[] var = new <type>[<size>];
```

- can also specify initial values

```
1 <type>[] var = new <type>[]{element1, element2, ..., elementn};
```

- can use an already declared array to initialise a second,
- second array is an **alias** for the first array; they both refer to same values

```
1 <type>[] var = new <type>[<size>];
2 <type>[] var2 = var;
```

- any variable that stores a **non-primitive** value is a **pointer/reference**

Array Methods and Tools

```
1 int[] nums = new int[10]
2 int num = nums[0];           // array indexing
3 int x = intArray[10];        // gives out of bounds error: doesn't cause
    program to crash
4 int length = nums.length;    // array length is public instance variable
    (NB not usually available)
```

- Arrays library

```
1 import java.util.Arrays;
2 ...
3 System.out.println(Arrays.toString(nums));    // converting to a
    string
4 int[] nums = Arrays.copyOf(nums, nums.length); // create distinct copy
    of an array
5 Arrays.sort(nums)                           // in-place sort
6 Arrays.equal(nums, nums2);                   // equality: same
    length + holds same values
```

- Resizing

```
1 int[] intArray = new int[5];
2 intArray = new int[intArray.length + 3];
```

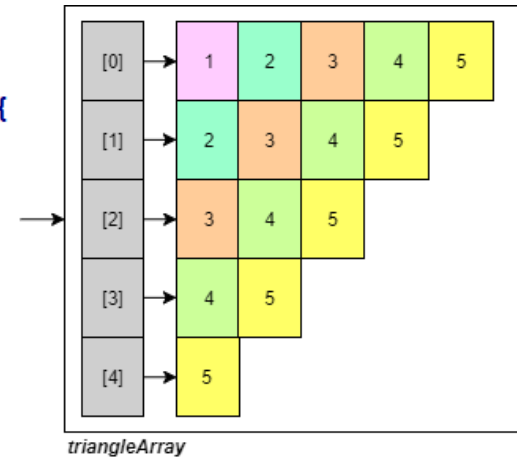
Array Iteration

- **for each loop:** can use to access each element of an iterable e.g. array when you are not modifying it

```
1 for (<type> var : <iterable>) {
2     // code block
3 }
```

Multi-dimensional arrays

```
int[][] triangleArray = {
    {1, 2, 3, 4, 5},
    {2, 3, 4, 5},
    {3, 4, 5},
    {4, 5},
    {5},
};
```



- treated as arrays of arrays
- declaration and initialisation: rectangular 2D array

```
1 int[][] nums = new int[100][10]; // array with 100 rows and 10 columns,
   each cell initialised to 0
```

- irregular array (e.g. triangular)

```
1 int[][] nums = new int[10][];
2 for (int i = 0; i < nums.length; i++) {
3     nums[i] = new int[<length of subarray>];
4 }
```

```
1 import java.util.Arrays;
2
3 public class Program {
4     public static void main(String args[]) {
5         final int NUM_ROWS = 5;
6         final int MAX_COLS = NUM_ROWS;
7
8         int[][] nums = new int[NUM_ROWS][]; // <- declaration of
           uninitialised 2D array
9
10        for (int i = 0; i < nums.length; i++) {
11            nums[i] = new int[NUM_ROWS - i];
12        }
13
14        for (int i = 0; i < NUM_ROWS; i++) {
15            System.out.println(Arrays.toString(nums[i]));
16        }
17    }
```

- Write a Java static method `computeDoublePowers` that accepts an integer `n` and returns an

array of **doubles** of that size. Your method should then fill that array with increasing powers of 2 (starting from 1.0)

```

1 import java.lang.Math;
2 // ...
3 public static double[] computeDoublePowers(int n) {
4     double[] nums = new double[n];
5     for (int i = 0; i < n; i++) {
6         arr[0] = Math.pow(2, i);
7     }
8     return nums;
9 }

```

```

int[][] triangleArray = {
    {1, 2, 3, 4, 5},
    {2, 3, 4, 5},
    {3, 4, 5},
    {4, 5},
    {5},
};

```

- Write a program that can generate the following 2D array:

```

1 public class IrregularArray {
2     public static void main(String[] args) {
3         final int HEIGHT = 5;
4         final int MAX_WIDTH = HEIGHT;
5         int[][] triangleArray = new int[HEIGHT][];
6         for (int i = 0; i < HEIGHT; i++) {
7             triangleArray[i] = new int[MAX_WIDTH - i];
8             for (int j = 0; j < MAX_WIDTH - i; j++) {
9                 triangleArray[i][j] = j+i+ 1;
10            }
11        }
12    }
13 }

```

Arrays of Objects

- arrays can also be used to store objects, but when you initialise the array it doesn't create the objects

```
1 // CircleArray.java
2 class CircleArray {
3     Circle[] circleArray = new Circle[3];
4     // create circle objects, store in array
5     for (int i = 0; i < circleArray.length; i++) {
6         circleArray[i] = new Circle(i, i, i+2);
7     }
8 }
```

Strings

- `Strings` are a class imported by default in Java
- cannot use single quotes for Java `Strings`

Basic string operations

```
1 String s = "Hello";
2 s.length();           // returns length of s (5)
3 s.toUpperCase();       // returns "HELLO"
4 s.toLowerCase();       // returns "hello"
5 s.split(" ");          // split by space character
```

e.g. What does this output?

```
1 String s = "Hello World";
2 s.toUpperCase();       // "HELLO WORLD"
3 s.replace("e", "i");   // "Hillo World"
4 s.substring(0,2);      // "He"
5 s += " FIVE";          // s = "Hello World FIVE"
6 System.out.println(s); // "Hello World FIVE"
```

Substrings

```
1 String substr = "el";
2 s.contains(substr);    // indicates if substr found in s
3 s.indexOf(substr);     // indicates index of first instance of
   substr; else -1
4 s.substring(arg1, arg2); // slice of string with indices [arg1, arg2
   - 1]
```

String concatenation

- Java has + operator overloaded for string concatenation

```
1 System.out.println("1 + 1 = " + 1 + 1);
2 // "1 + 1 = 11"
3 System.out.println("1 + 1 = " + (1 + 1));
4 // "1 + 1 = 2"
```

String equality and references

```
1 public class Program {
2     public static void main(String[] args) {
3         // 1. Two string literals
4         System.out.println("Hello" == "Hello"); // true
5
6         // 2. One literal, one variable
7         String s0 = "Hello";
8         System.out.println(s0 == "Hello"); // true
9
10        // 3. Two variables, given the same literal value
11        String s1 = "Hello";
12        String s2 = "Hello";
13        System.out.println(s1 == s2); // true
14
15        // 4. Two variables, with one creating a new "object"
16        String s3 = "Hello";
17        String s4 = new String("Hello");
18        System.out.println(s3 == s4); // false
19    }
20 }
```

- Java is built on **references** which act like pointers
- when you explicitly write a `String` (e.g. "Hello") it is effectively treated as a constant (string literal*) and stored separate to dynamic memory
 - this constant is only created once, in e.g. 1-3, the string is the same, irrespective of which variable it is in
- e.g. 4 creates a **new** `String`, which Java puts on the *heap* (dynamic memory)
 - `s3` is now pointing at different address than `s4`
- `==` applied to objects is actually comparing *address* of reference
- for string comparison, use `String.equals()`

String Modification

- strings are **immutable**: once created they cannot be modified
- all string methods return a string which you can then assign to a variable