Vietnam National University of HCMC
International University
School of Computer Science and Engineering

Arrays & ArrayLists

(IT069IU)

Le Duy Tan, Ph.D.

ldtan@hcmiu.edu.vn

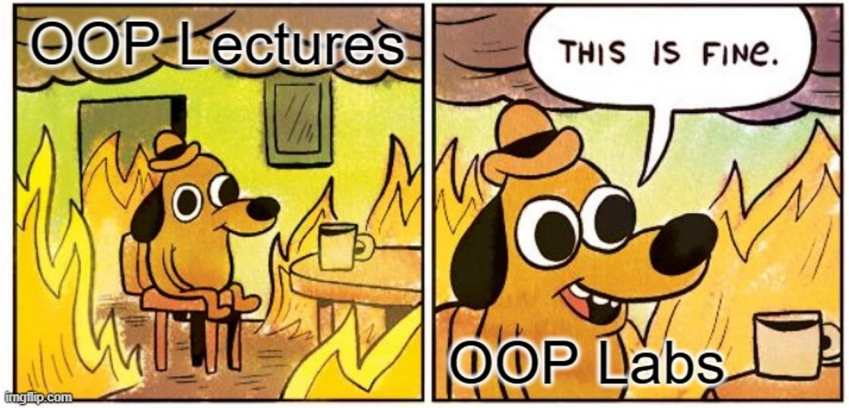
leduytanit.com

Previously,

- **Casting** types.
- Control flow statements:
  - Decision making statements:
    - **If**
    - **If...else**
    - **Switch**
  - Loop statements:
    - **While**
    - **Do while**
    - **For**
  - Jump statements:
    - **Break** statement
    - **Continue** statement
- **Static** Keyword
  - Static Method
  - Static Variable
- **Overloading**
  - Method Overloading
  - Constructor Overloading
- **Final** Keyword
  - Constant Variable

Agenda

- **Increment and Decrement Operators**
- **Scope of Declarations**
  - this keyword
- **Array:**
  - Declare and Create Array
  - Loop through Array
  - Pass Arrays to Methods
    - Pass by Value vs Pass by Reference
  - Multidimensional Arrays
  - Class Arrays for helper methods
- **ArrayList** (Collections Class)
- **Array vs ArrayList**



Increment and Decrement Operators

- Like C, Java has two unary operators:
  - Increment operator **++**
  - Decrement operator **--**

Operator	Operator name	Sample expression	Explanation
++	prefix increment	++a	Increment a by 1, then use the new value of a in the expression in which a resides.
++	postfix increment	a++	Use the current value of a in the expression in which a resides, then increment a by 1.
--	prefix decrement	--b	Decrement b by 1, then use the new value of b in the expression in which b resides.
--	postfix decrement	b--	Use the current value of b in the expression in which b resides, then decrement b by 1.

- For example, the three assignment statements:

```
passes = passes + 1;
failures = failures + 1;
studentCounter = studentCounter + 1;
```

- can be written more concisely with compound assignment operators as

```
passes += 1;
failures += 1;
studentCounter += 1;
```

- prefix increment operators as

```
++passes;
++failures;
++studentCounter;
```

- postfix increment operators as

```
passes++;
failures++;
studentCounter++;
```

```
1  // Fig. 4.15: Increment.java
2  // Prefix increment and postfix increment operators.
3
4  public class Increment
5  {
6     public static void main(String[] args)
7     {
8        // demonstrate postfix increment operator
9        int c = 5;
10       System.out.printf("c before postincrement: %d%n", c); // prints 5
11       System.out.printf("    postincrementing c: %d%n", c++); // prints 5
12       System.out.printf(" c after postincrement: %d%n", c); // prints 6
13
14       System.out.println(); // skip a line
15
16       // demonstrate prefix increment operator
17       c = 5;
18       System.out.printf(" c before preincrement: %d%n", c); // prints 5
19       System.out.printf("     preincrementing c: %d%n", ++c); // prints 6
20       System.out.printf("  c after preincrement: %d%n", c); // prints 6
21    }
22 } // end class Increment
```

```
c before postincrement: 5
    postincrementing c: 5
 c after postincrement: 6

 c before preincrement: 5
     preincrementing c: 6
  c after preincrement: 6
```

Scope of Declaration

Scope of Declarations

- You've seen declarations of various **Java entities**, such as **classes**, **methods**, **variables** and **parameters**.
- **The scope of declaration** refers to the **places in the code** that **an entity is accessible.**
- There are generally **three main "levels" of scope**:
  - If something has **class scope**, it means it is **accessible across an entire class**. For examples:
    - **Variables (class variables, instance variables), Constants, and Methods.**
  - If something has **method scope**, it means it is **accessible across an entire single method**, but not before the line on which it is declared. For example:
    - **Local variables** are **declared inside a method**. **(method variables)**
    - **Parameters** in **method header**
  - If something has **block scope** or **inner block scope**, then it is **accessible within just a section of the block.** These variables "go out of scope" at the end of the loop. For example:
    - **Initialized variables** in a **for loop** header.
    - **Local variable** declared inside **while loop**.

Simple Example of Scope

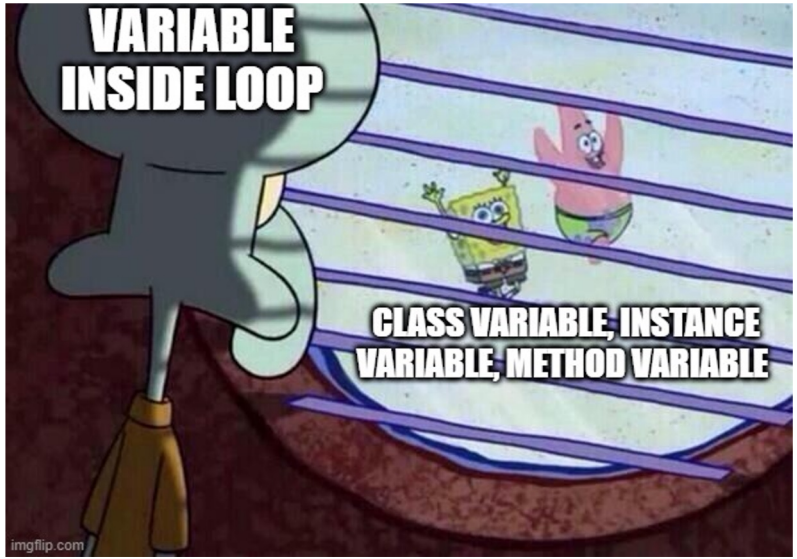
- In this method, the **variable a** has **method scope**, but the **variable x** has **block scope**.
- The **variable a can be used throughout the method**, because it has method scope.
- The **variable x can be used within the loop itself, but not after the loop**. It won't even compile. It has fallen out of scope.

```
public void doSomething(){
    int a = 0;
    for(int x=0; x<5; x++){
        System.out.println(x);
    }
    x = -5; // This won't compile as the variable 'x' is "out of scope".
    a = -5; // This is ok since the variable 'a' is still in scope of the method.
}
```

Another Example of Scope

- Interestingly, we can **reuse the same variable name (x) for a different variable**, and the **two won't conflict because both x variable only exists within each loop.**

```
public void doSomething(){
    for(int x = 0; x < 5; x++)
        System.out.println(x);

    for(int x = 0; x < 10; x++)
        System.out.println(x);
}
```



Name Hiding

- When **two variables are in different scopes and they have the same name.**
- When this happens, **the variable in the smaller scope "hides" the variable in the larger scope.** This is a situation called **name hiding.**
- **One solution** is to just always **choose a different names** for local variables and parameters compared to the class-level instance variable.
- **"this"** is a special reference to the current instance that a method or constructor is running in. It's a quick and easy way to **reach up into the class scope** and work with instance variables and even methods that belong to the class.

Choose different names for parameters

```
public class Student {
    private int id;
    private String name;

    public Student(int inputID, String inputName){
        id = inputID;
        name = inputName;
    }
}
```

Choose same names but use this keyword to refer to class-level variables

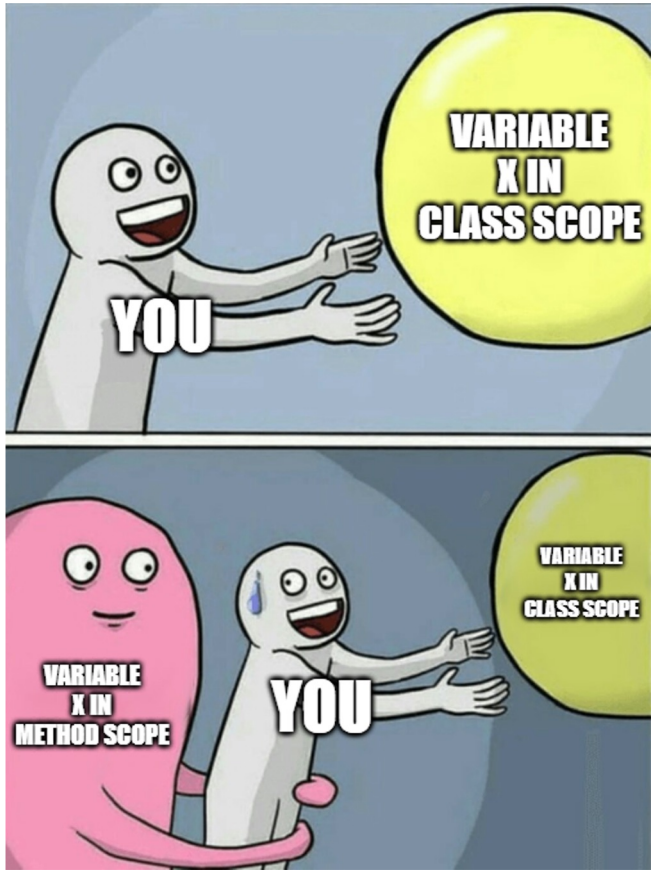
```
public class Student {
    private int id;
    private String name;

    public Student(int id, String name){
        this.id = id;
        this.name = name;
    }
}
```

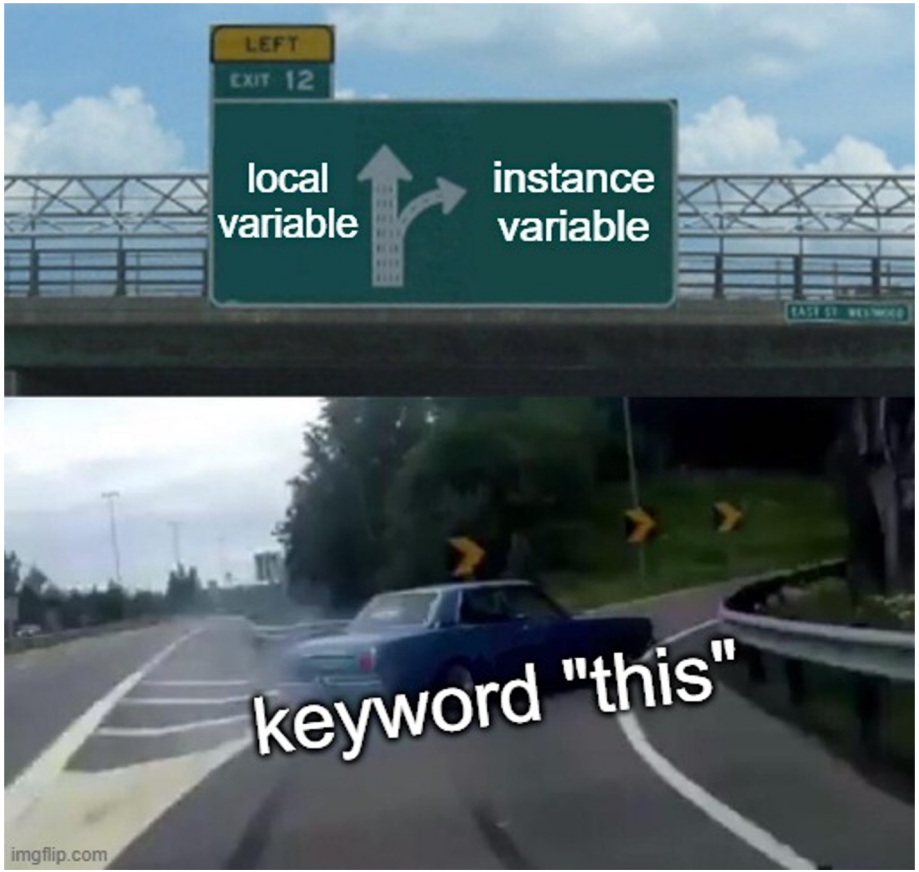
```java
// Fig. 6.9: Scope.java
// Scope class demonstrates field and local variable scopes.

public class Scope
{
   // field that is accessible to all methods of this class
   private static int x = 1;

   // method main creates and initializes local variable x
   // and calls methods useLocalVariable and useField
   public static void main(String[] args)
   {
      int x = 5; // method's local variable x shadows field x

      System.out.printf("local x in main is %d%n", x);

      useLocalVariable(); // useLocalVariable has local x
      useField(); // useField uses class Scope's field x
      useLocalVariable(); // useLocalVariable reinitializes local x
      useField(); // class Scope's field x retains its value

      System.out.printf("%nlocal x in main is %d%n", x);
   }

   // create and initialize local variable x during each call
   public static void useLocalVariable()
   {
      int x = 25; // initialized each time useLocalVariable is called

      System.out.printf(
         "%nlocal x on entering method useLocalVariable is %d%n", x);
      ++x; // modifies this method's local variable x
      System.out.printf(
         "local x before exiting method useLocalVariable is %d%n", x);
   }

   // modify class Scope's field x during each call
   public static void useField()
   {
      System.out.printf(
         "%nfield x on entering method useField is %d%n", x);
      x *= 10; // modifies class Scope's field x
      System.out.printf(
         "field x before exiting method useField is %d%n", x);
   }
} // end class Scope
```

```
local x in main is 5

local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 1
field x before exiting method useField is 10

local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26

field x on entering method useField is 10
field x before exiting method useField is 100

local x in main is 5
```

[Question] Can you tell which variable "hides" which variable and which scope?

Meme about Scope of Declaration





Array & ArrayList

Store multiple items of the same type

- Imagine you have a high score board in a game, with 10 high scores on it. Using only the tools we know so far, you can create one variable for every score you want to keep track of:

```
int score1 = 100;
int score2 = 95;
int score3 = 86;
// more and more ...
```

- What if you had 10,000 scores? Too many variable!!!

Array is here to save you!

- with 10 high scores, you want to keep track of:

```
int score1 = 100;
int score2 = 95;
int score3 = 86;
// more and more ...
```

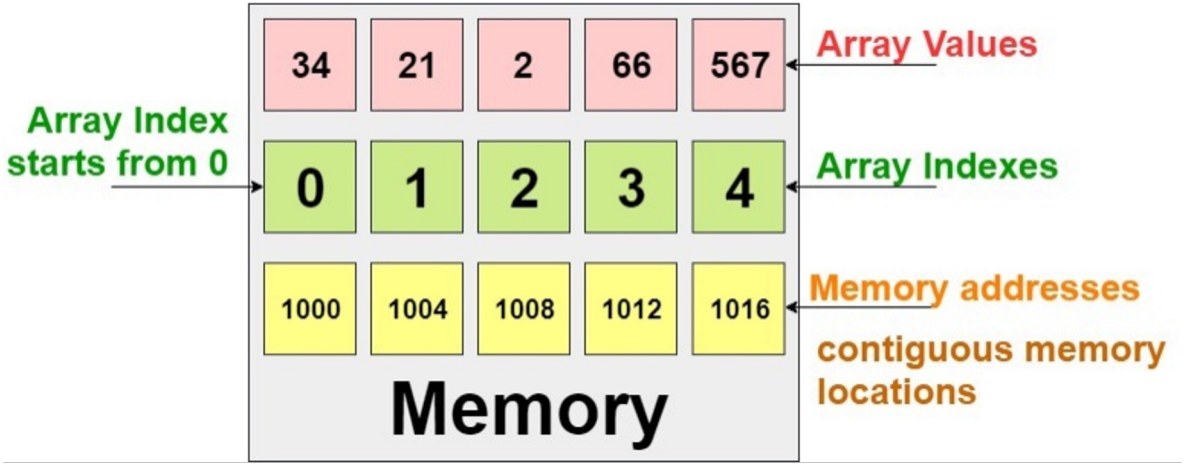


```
int[] scores = new int[10];
```

Array

- An array is **a group of variables** (called elements or components) containing **values of the same type**.
- Arrays are **objects**, so they're considered **reference types**.

```
int[ ] x = new int[5];
x[0] = 34;
x[1] = 21;
x[2] = 2;
x[3] = 66;
x[4] = 567;
```



Array Declaration Syntax

- Array Declaration (array variable points to null):

data_type[] *arrayName*;

- Allocate actual memory for array members:

arrayName = new *data_type*[*size*];

- You can do both steps in one line:

data_type[] *arrayName* = new *data_type*[*size*];

For Example:

- you declare a new array with two steps:

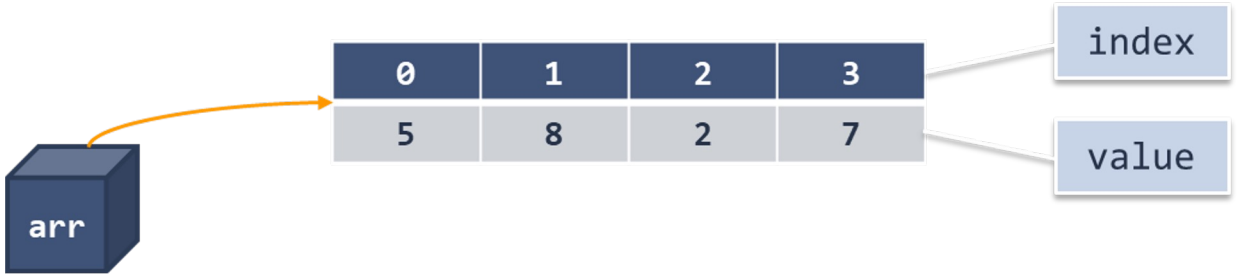
```
int[] c; // declare the array variable
c = new int[12]; // create the array; assign to array variable
```

- You can declare a new array of 12 integers in one step like this:

```
int[] c = new int[12];
```

Array Initializer Example

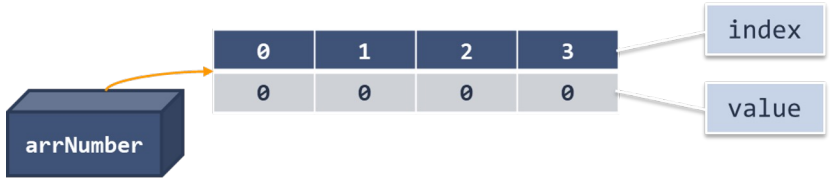
```
int[] arr;
arr = new int[4];
arr[0] = 5;
arr[1] = 8;
arr[2] = 2;
arr[3] = 7;
```



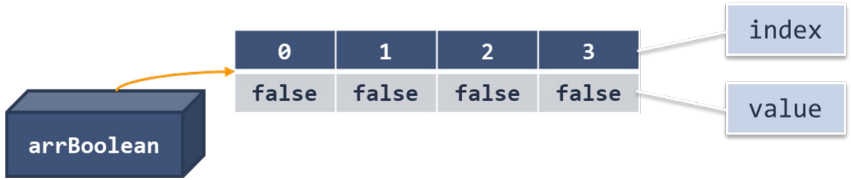
Default Value for Array Element

- **When an array is created**, each of its elements receives a **default value**. For example:
  - **zero** for the **numeric** primitive-type elements
  - **false** for **boolean** elements
  - **null** for **references**

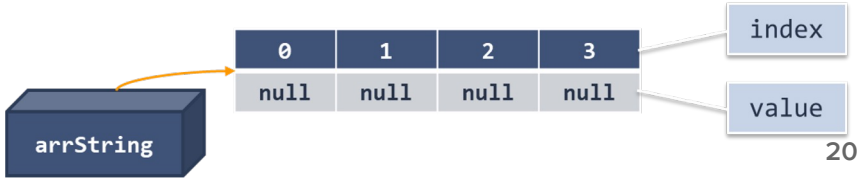
```
int[] arrNumber = new int[4];
```




```
boolean[] arrBoolean = new boolean[4];
```




```
String[] arrString = new String[4];
```



Readability for Array Declaration

- A program **can create several arrays in a single declaration**. The following declaration reserves 100 elements for b and 27 elements for x:

```
String[] b = new String[100], x = new String[27];
```

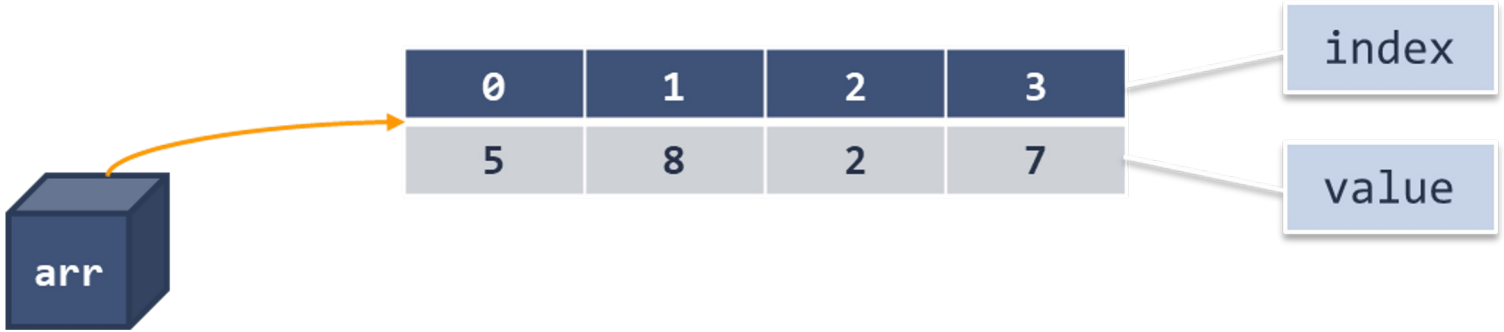
- **For readability**, we **prefer to declare only one variable at a time**:

```
String[] b = new String[100]; // create array b
String[] x = new String[27]; // create array x
```

Array Initializer

- You can create an array and initialize its elements with an **array initializer.**
- An **array initializer is** a comma-separated list of expressions (called an initializer list) enclosed in braces

```
int[] arr = {5, 8, 2, 7};
```



Access Element in Array using Index

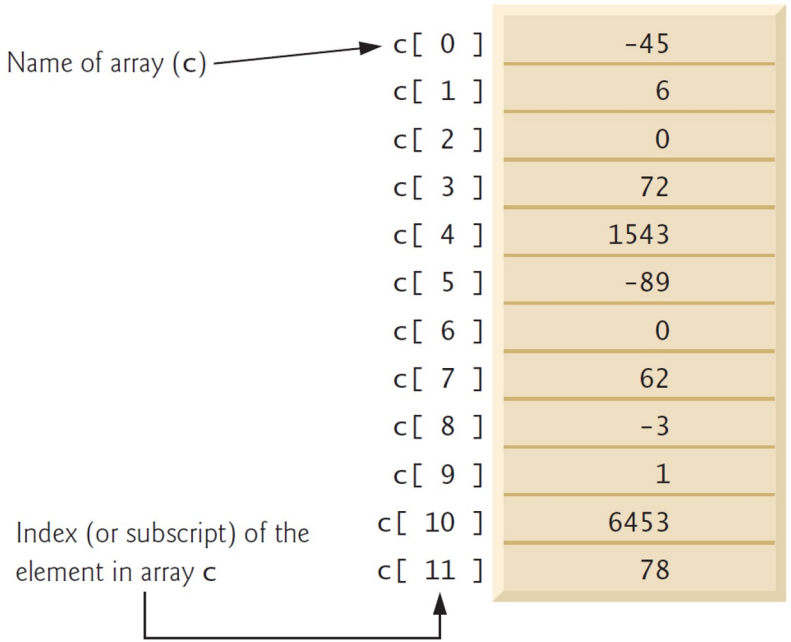
int[ ] c = {-45, 6, 0, 72, 1543, -89, 0, 62, -3, 1, 6453, 78}

```
int a =5, b = 6;
c[a+b] +=2;
```

What will be changed in the array?

```
int sum = c[0] + c[1] + c[2];
```

What is the output of sum?



Array Length Usefulness

```java
int[] scores = new int[5];
System.out.println("The size of is " + scores.length);
System.out.println("Old Array:");
for (int i=0; i<scores.length; i++){
    System.out.printf("%d ",scores[i]);
}
scores[1]=99;
scores[4]=88;
System.out.println("\nNew Array:");
for (int i=0; i<scores.length; i++){
    System.out.printf("%d ",scores[i]);
}
```

Output:

```
The size of is 5
Old Array:
0 0 0 0 0
New Array:
0 99 0 0 88
```

Loop Through 1-Dimensional Array

- Use for loop:

```
int[] myArray= {1, 2, 3, 4, 5, 6};

for (int index=0; index < myArray.length; index++){
        System.out.printf("%d ", myArray[index]);
}
```

The Same Output:

```
1 2 3 4 5 6
```

- Use enhanced for loop:

```
int[] myArray= {1, 2, 3, 4, 5, 6};

for (int element: myArray){
        System.out.printf("%d ",element);
}
```

[Question] What is the advantage and disadvantage of for loop and enhanced for loop to loop through array?

Initialize Array Example

InitArray.java

```
1  // Fig. 7.2: InitArray.java
2  // Initializing the elements of an array to default values of zero.
3
4  public class InitArray
5  {
6     public static void main(String[] args)
7     {
8        // declare variable array and initialize it with an array object
9        int[] array = new int[10]; // create the array object
10
11       System.out.printf("%s%8s%n", "Index", "Value"); // column headings
12
13       // output each array element's value
14       for (int counter = 0; counter < array.length; counter++)
15          System.out.printf("%5d%8d%n", counter, array[counter]);
16    }
17 } // end class InitArray
```

Index	Value
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0

Array Initializer Example

Index	Value
0	32
1	27
2	64
3	18
4	95
5	14
6	90
7	70
8	60
9	37

InitArray.java

```
1  // Fig. 7.3: InitArray.java
2  // Initializing the elements of an array with an array initializer.
3
4  public class InitArray
5  {
6     public static void main(String[] args)
7     {
8        // initializer list specifies the initial value for each element
9        int[] array = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
10
11       System.out.printf("%s%8s%n", "Index", "Value"); // column headings
12
13       // output each array element's value
14       for (int counter = 0; counter < array.length; counter++)
15          System.out.printf("%5d%8d%n", counter, array[counter]);
16
17 } // end class InitArray
```

Sum Array Example

```
1  // Fig. 7.4: InitArray.java
2  // Calculating the values to be placed into the elements of an array.
3
4  public class InitArray
5  {
6     public static void main(String[] args)
7     {
8        final int ARRAY_LENGTH = 10; // declare constant
9        int[] array = new int[ARRAY_LENGTH]; // create array
10
11       // calculate value for each array element
12       for (int counter = 0; counter < array.length; counter++)
13          array[counter] = 2 + 2 * counter;
14
15       System.out.printf("%s%8s%n", "Index", "Value"); // column headings
16
17       // output each array element's value
18       for (int counter = 0; counter < array.length; counter++)
19          System.out.printf("%5d%8d%n", counter, array[counter]);
20    }
21 } // end class InitArray
```

Index	Value
0	2
1	4
2	6
3	8
4	10
5	12
6	14
7	16
8	18
9	20

Enhanced for Statement

- The enhanced for statement i**terates through the elements of an array without using a counter**, thus **avoiding the possibility of "stepping outside" the array.**
- where **parameter** has a **type** and an **identifier** (e.g., int number), and **arrayName** is the **array through which to iterate.**

```
for (parameter : arrayName)
    statement
```

- Instead, using counter in a for loop to loop through an array:

```
for (int counter = 0; counter < array.length; counter++)
   total += array[counter];
```

- We can just use:

```
for (int number : array)
   total += number;
```

Enhanced For Loop Example

```
1  // Fig. 7.12: EnhancedForTest.java
2  // Using the enhanced for statement to total integers in an array.
3
4  public class EnhancedForTest
5  {
6     public static void main(String[] args)
7     {
8        int[] array = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
9        int total = 0;
10
11       // add each element's value to total
12       for (int number : array)
13          total += number;
14
15       System.out.printf("Total of array elements: %d%n", total);
16    }
17 } // end class EnhancedForTest
```

```
Total of array elements: 849
```

Pass Arrays to Methods

- You can **pass an array to a method** just like **you pass a variable to a method**.

For example:

- If **array hourlyTemperatures** is declared as:

```
double[] hourlyTemperatures = new double[24];
```

- The **method call** can be:

```
modifyArray(hourlyTemperatures);
```

- **Method header** might be written as:

```
void modifyArray(double[] b)
```

```java
public class TemperatureCaculation {
    public static void main(String[] args) {
        double [] hourlyTemperatures = {34, 24, 25, 26, 37, 34, 52};
        double averageTemp = calculateAverage(hourlyTemperatures);
        System.out.printf("The average: %.2f\n", averageTemp);
        convertToFahrenheit(hourlyTemperatures);
        System.out.print("C->F: ");
        for (double temp: hourlyTemperatures){
            System.out.printf("%.2f ", temp);
        }
    }

    public static double calculateAverage(double[] localArray){
        double sum=0;
        for (double temp: localArray){
            sum+=temp;
        }
        return sum/localArray.length;
    }

    public static void convertToFahrenheit(double[] tempC){
        for (int index=0; index< tempC.length; index++){
            tempC[index] = tempC[index]/2+30;
        }
    }
}
```

Output:

```
The average: 33.14
C->F: 47.00 42.00 42.50 43.00 48.50 47.00 56.00
```

[Question] Why do we use an enhanced loop for calculateAverage method and a normal loop for convertToFahrenheit method?

Pass-By-Value vs. Pass-By-Reference

```java
public static void main(String[] args) {
    int x = 8;
    System.out.println("X Start: " + x);
    System.out.println("Local X: " + addTwo(x));
    System.out.println("X End: " + x);
}

public static int addTwo(int local_x){
    local_x+= 2;
    return local_x;
}
```

```
X Start: 8
Local X: 10
X End: 8
```

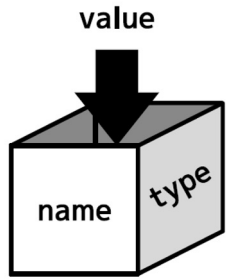
```java
public static void main(String[] args) {
    int[] x = {8};
    System.out.println("X Start: " + x[0]);
    System.out.println("Local X: " + addTwo(x)[0]);
    System.out.println("X End: " + x[0]);
}

public static int[] addTwo(int[] local_x){
    local_x[0] += 2;
    return local_x;
}
```

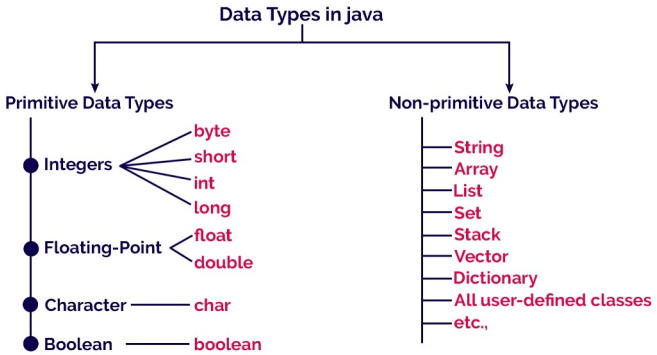
```
X Start: 8
Local X: 10
X End: 10
```

[Question] Which example passes by value to a method or which passes by reference to a method?

Let's revisited: Variable

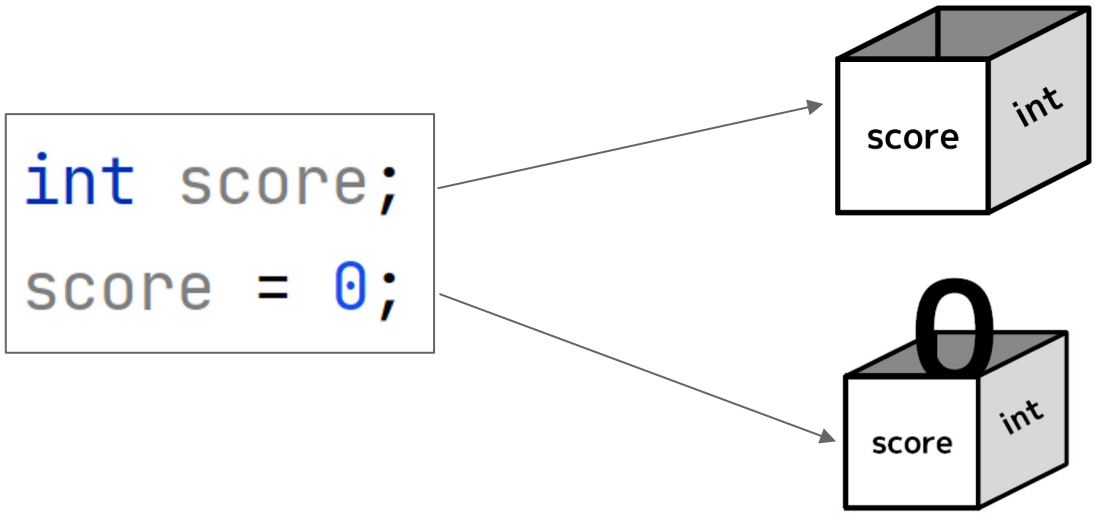
- Remember, a variable is a place in memory where you can store information.
- It's like a little box or bucket to put stuff in.
- Each **variable** has a **name** and a **type**.
- For **primitive** data type:
  - The value of a variable stores the actual value of the data.
- For **reference** data type:
  - The value of a variable stores the memory address (reference) to the location of the data. Similar to pointer address





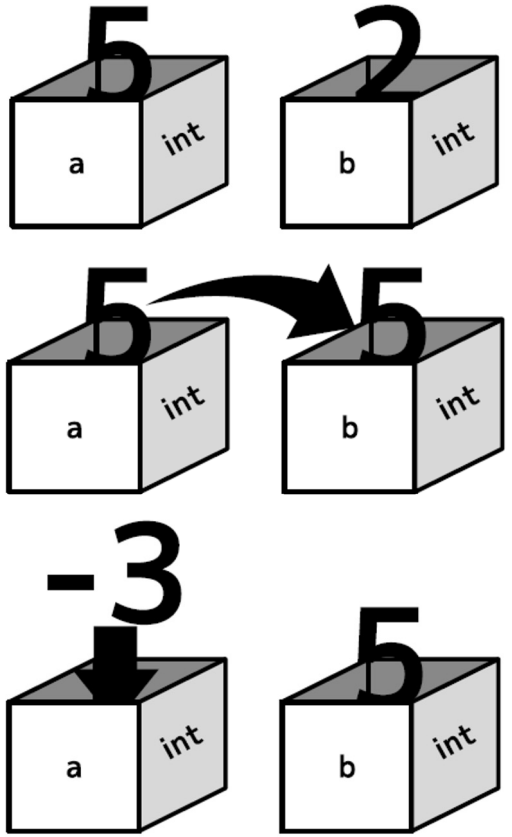
Primitive Data Type

- For **primitive** data type:
  - The value of a variable stores the **actual value of the data**.



Primitive Data Type

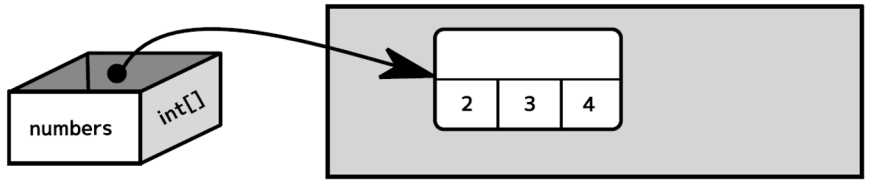
```
int a = 5;
int b = 2;
b = a;
a = -3;

System.out.println(a);
System.out.println(b);
```



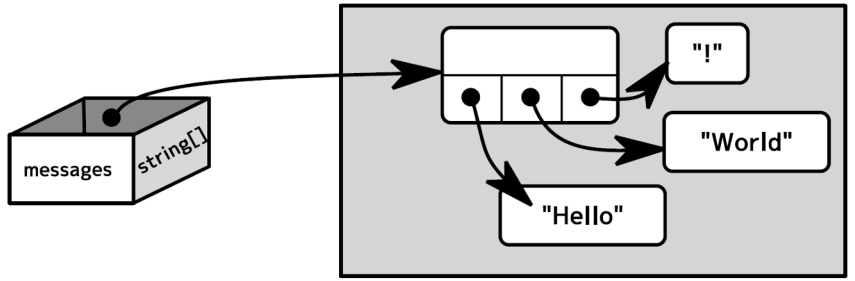
Array is a Reference Data Type

- For **reference** data type:
  - The value of a variable stores the **memory address (reference)** to the location of the data. **Similar to pointer address in C.**

```
int[] numbers =  { 2, 3, 4 };
```




```
String[] messages = {"Hello", "World", "!"};
```

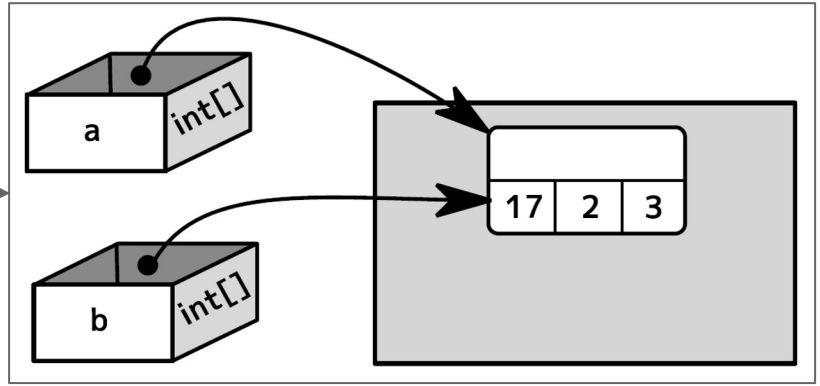


Compared This

```java
int a = 3;
int b = a;
b++;
System.out.println(a);
System.out.println(b);
```

[Question] Can you guess what is the output for each case?

```java
int[] a = new int[] { 1, 2, 3 };
int[] b = a;
b[0] = 17;
System.out.println(a[0]);
System.out.println(b[0]);
```



```java
// Fig. 7.13: PassArray.java
// Passing arrays and individual array elements to methods.

public class PassArray
{
   // main creates array and calls modifyArray and modifyElement
   public static void main(String[] args)
   {
      int[] array = { 1, 2, 3, 4, 5 };

      System.out.printf(
         "Effects of passing reference to entire array:%n" +
         "The values of the original array are:%n");

      // output original array elements
      for (int value : array)
         System.out.printf("   %d", value);

      modifyArray(array); // pass array reference
      System.out.printf("%n%nThe values of the modified array are:%n");

      // output modified array elements
      for (int value : array)
         System.out.printf("   %d", value);

      System.out.printf(
         "%n%nEffects of passing array element value:%n" +
         "array[3] before modifyElement: %d%n", array[3]);

      modifyElement(array[3]); // attempt to modify array[3]
      System.out.printf(
         "array[3] after modifyElement: %d%n", array[3]);
   }

   // multiply each element of an array by 2
   public static void modifyArray(int[] array2)
   {
      for (int counter = 0; counter < array2.length; counter++)
         array2[counter] *= 2;
   }

   // multiply argument by 2
   public static void modifyElement(int element)
   {
      element *= 2;
      System.out.printf(
         "Value of element in modifyElement: %d%n", element);
   }
} // end class PassArray
```

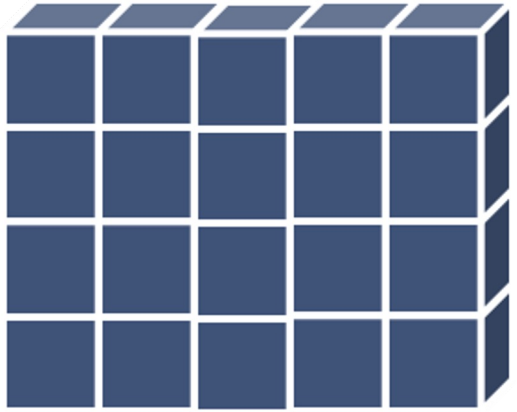
```
Effects of passing reference to entire array:
The values of the original array are:
   1   2   3   4   5

The values of the modified array are:
   2   4   6   8   10

Effects of passing array element value:
array[3] before modifyElement: 8
Value of element in modifyElement: 16
array[3] after modifyElement: 8
```

[Question] Why the modifyArray method change the original array but modifyElement method does not change the value of the original element of the array?

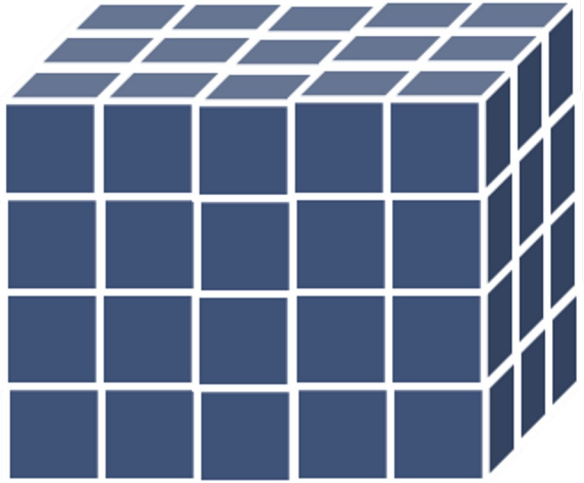
The Nature of Multidimensional Array

- Two dimensional array is one type of multidimensional array.

1-Dimensional Array



2-Dimensional Array



3-Dimensional Array

Multidimensional Arrays

- **Multidimensional arrays with two dimensions** are often used to represent tables of values with data arranged in **rows and columns.**
- To **identify a particular table element**, you **specify two indices.**
- By convention, **the first identifies the element's row and the second its column**.

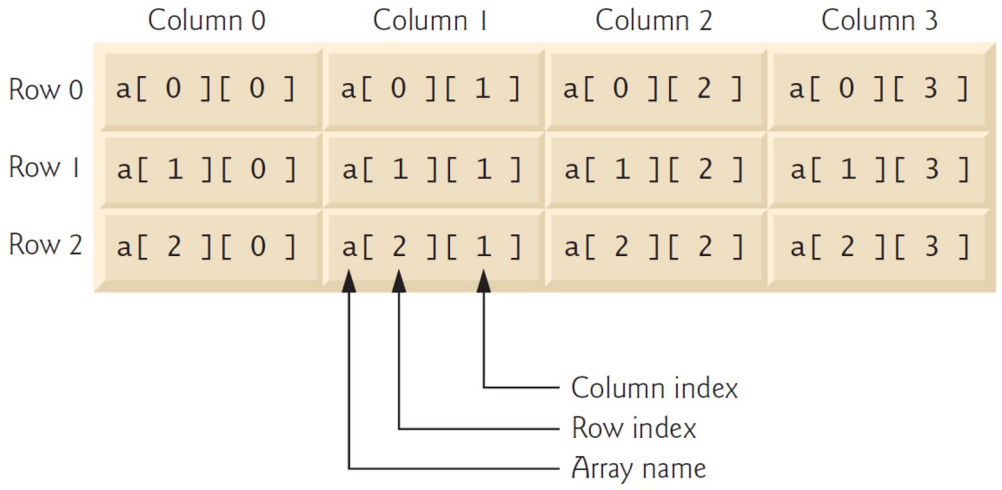


Fig. 7.16 | Two-dimensional array with three rows and four columns.

2-Dimensional Array Declaration Syntax

- Array Declaration (array variable points to null):

data_type[][] *arrayName*;

- Allocate actual memory for array members:

arrayName = new *data_type*[*size*][*size*];

- You can do both steps in one line:

data_type[][] *arrayName* = new *data_type*[*size*][*size*];

For Example:

- you declare a new array with two steps:

```
int[][] c;
c = new int[3][5];
```

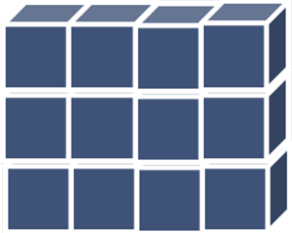
- You can declare a new 2-dimensional array of in one step like this:

```
int[][] c = new int[3][5];
```

Creating Two-Dimensional Arrays with Array-Creation Expressions

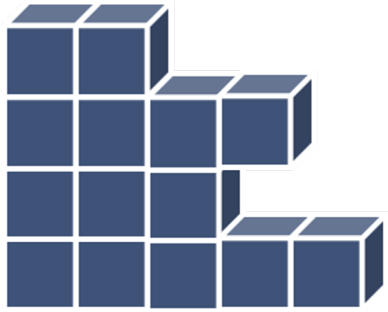
- Here is to declare array b and assign it a reference to a three-by-four array:

```
int[][] b = new int[3][4];
```



- A **multidimensional array in which each row has a different number of columns can be created as follows**:

```
int[][] b = new int[4][];
b[0] = new int[2];
b[1] = new int[4];
b[2] = new int[3];
b[3] = new int[5];
```



Default Values of 2-Dimensional Array

- **Like the same rule with one dimensional array**, each of its elements receives a **default value**. For example:
  - **zero** for the **numeric** primitive-type elements
  - **false** for **boolean** elements
  - **null** for **references**

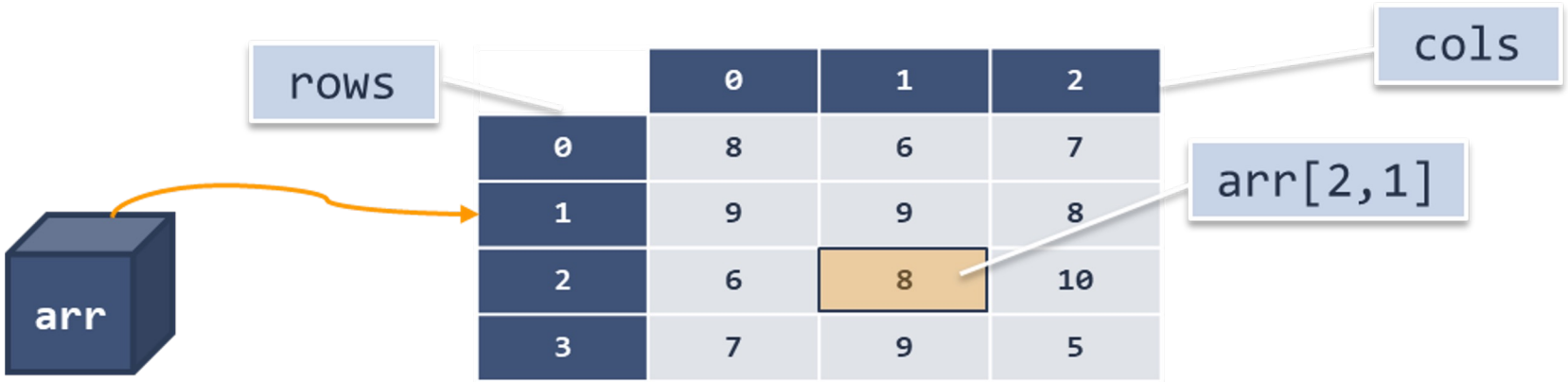
```
int[][] arrNumber = new int[4][3];
```

	0	1	2
0	0	0	0
1	0	0	0
2	0	0	0
3	0	0	0

Multiple Dimension Array Index

- **Row index** usually starts from top to bottom, starting with 0.
- **Column index** usually starts from left to right, starting with 0.
- To **access each element in the array**, we need to specific:

arrayName[row index, column index]



Nested Array Initializers

- Like one-dimensional arrays, multidimensional arrays can be **initialized with array initializers in declarations**.

```
int[ ][ ] myArray = { {8,6,7}, {9,9,8}, {6,8,10}, {7,9,5} };
```

	0	1	2
0	8	6	7
1	9	9	8
2	6	8	10
3	7	9	5

Nested Array Initializers

- In fact, **the lengths of the rows in array** b are **not required to be the same**:

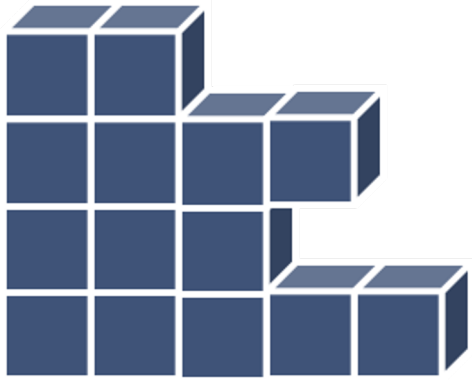
```
int[ ][ ] myArray = { {1, 2}, {3, 4, 5} };
```

	0	1	2
0	1	2	
1	3	4	5

Two-Dimensional Array Length

```
int[][] b = new int[4][0];

b[0] = new int[2];
b[1] = new int[4];
b[2] = new int[3];
b[3] = new int[5];

System.out.printf("Number of rows: %d\n", b.length);
System.out.printf("Number of columns in 1st row: %d\n", b[0].length);
System.out.printf("Number of columns in 2nd row: %d\n", b[1].length);
System.out.printf("Number of columns in 3rd row: %d\n", b[2].length);
System.out.printf("Number of columns in 4th row: %d\n", b[3].length);
```

```
Number of rows: 4
Number of columns in 1st row: 2
Number of columns in 2nd row: 4
Number of columns in 3rd row: 3
Number of columns in 4th row: 5
```



Loop through 2-dimensional array

- Use for loop:

```
int[][] myArray={{1, 2},{3},{4,5,6}};

for (int row=0; row < myArray.length; row++){
    for (int column=0; column < myArray[row].length; column++){
        System.out.printf("%d ",myArray[row][column]);
    }
    System.out.println();
}
```

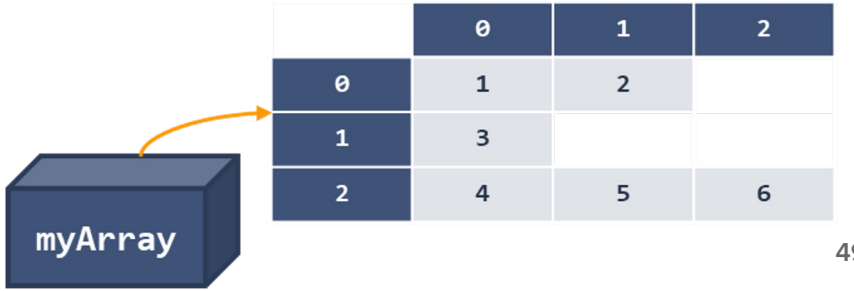
- Use for each loop:

```
int[][] myArray={{1, 2},{3},{4,5,6}};

for (int[] row: myArray){
    for (int column: row){
        System.out.printf("%d ",column);
    }
    System.out.println();
}
```

The Same Output:

```
1 2
3
4 5 6
```

[**Question**] What is the advantage and disadvantage of for loop and enhanced for loop to loop through array?



Pass 2-Dimensional Array to a Method

```
1  // Fig. 7.17: InitArray.java
2  // Initializing two-dimensional arrays.
3
4  public class InitArray
5  {
6     // create and output two-dimensional arrays
7     public static void main(String[] args)
8     {
9        int[][] array1 = {{1, 2, 3}, {4, 5, 6}};
10       int[][] array2 = {{1, 2}, {3}, {4, 5, 6}};
11
12       System.out.println("Values in array1 by row are");
13       outputArray(array1); // displays array1 by row
14
15       System.out.printf("%nValues in array2 by row are%n");
16       outputArray(array2); // displays array2 by row
17    }
18
19    // output rows and columns of a two-dimensional array
20    public static void outputArray(int[][] array)
21    {
22       // loop through array's rows
23       for (int row = 0; row < array.length; row++)
24       {
25          // loop through columns of current row
26          for (int column = 0; column < array[row].length; column++)
27             System.out.printf("%d  ", array[row][column]);
28
29          System.out.println();
30       }
31    }
32 } // end class InitArray
```

Output:

```
Values in array1 by row are
1  2  3
4  5  6

Values in array2 by row are
1  2
3
4  5  6
```

Variable-Length Argument Lists

- With **variable-length argument lists**, you can **create methods that receive an unspecified number of arguments.**
- A type followed by an ellipsis (...) in a method's parameter list indicates that the method receives a variable number of arguments of that particular type.
- This **use of the ellipsis** can **occur only once in a parameter list**, and the ellipsis, together with **its type** and the **parameter name**, must be **placed at the end of the parameter list.**
- Method header look like:

```
public static double average(double... numbers)
```

- Method call now can take any number of parameters:

```
average(5);
average(5, 7);
average(5, 7, 10);
average(5, 7, 10, 15);
```

Variable-Length Argument Lists Example

```
1  // Fig. 7.20: VarargsTest.java
2  // Using variable-length argument lists.
3
4  public class VarargsTest
5  {
6     // calculate average
7     public static double average(double... numbers)
8     {
9        double total = 0.0;
10
11       // calculate total using the enhanced for statement
12       for (double d : numbers)
13          total += d;
14
15       return total / numbers.length;
16    }
17
18    public static void main(String[] args)
19    {
20       double d1 = 10.0;
21       double d2 = 20.0;
22       double d3 = 30.0;
23       double d4 = 40.0;
24
25       System.out.printf("d1 = %.1f%nd2 = %.1f%nd3 = %.1f%nd4 = %.1f%n%n",
26          d1, d2, d3, d4);
27
28       System.out.printf("Average of d1 and d2 is %.1f%n",
29          average(d1, d2) );
30       System.out.printf("Average of d1, d2 and d3 is %.1f%n",
31          average(d1, d2, d3) );
32       System.out.printf("Average of d1, d2, d3 and d4 is %.1f%n",
33          average(d1, d2, d3, d4) );
34    }
35 } // end class VarargsTest
```

Output:

```
d1 = 10.0
d2 = 20.0
d3 = 30.0
d4 = 40.0

Average of d1 and d2 is 15.0
Average of d1, d2 and d3 is 20.0
Average of d1, d2, d3 and d4 is 25.0
```

Variable-Length Argument with Other Arguments Example

```
public class SomeCalculationTest {

    public static void main(String[] args){
        System.out.println(someCalculate(4, 2, 5, 6, 7, 8));
    }

    public static double someCalculate(double add, double multiply, double... numberArray){
        double sum=0;
        for (double temp: numberArray){
            sum+=temp;
        }
        sum+=add;
        sum*=multiply;
        return sum;
    }
}
```

[**Question**] Can you guess what is the output and explain why is that?

Using Command-Line Arguments

```
public static void main(String[] args)
```

- Remember, we have:
- When an **application is executed using the java command**, **Java passes the command-line arguments that appear after the class name in the java** command to the application's main method **as Strings in the array args.**
- The **number of command-line arguments is obtained by accessing the array's length attribute**. **Common uses of commandline arguments include passing options and filenames to applications**
- The command passes three arguments, 5, 0 and 4, to the application InitArray.

```
java InitArray 5 0 4
```

Using Command-Line Arguments Example

```java
public class InitArray {

    public static void main(String[] args) {
        // check number of command-line arguments
        if (args.length != 3)
            System.out.println(
            "Error: Please re-enter the entire command " +
            "including an array size, initial value and increment.");
        else
        {
            // get array size from first command-line argument
            int arrayLength = Integer.parseInt(args[0]);
            int[] array = new int[arrayLength];

            // get initial value and increment from command-line arguments
            int initialValue = Integer.parseInt(args[1]);
            int increment = Integer.parseInt(args[2]);

            // calculate value for each array element
            for (int counter = 0; counter < array.length; counter++)
                array[counter] = initialValue + increment * counter;

            System.out.printf("%s%8s%n", "Index", "Value");

            // display array index and value
            for (int counter = 0; counter < array.length; counter++)
                System.out.printf("%5d%8d%n", counter, array[counter]);
        }
    }
}
```

If you try to execute without command-line arguments:

```
java InitArray
Error: Please re-enter the entire command, including
an array size, initial value and increment.
```

For this program, you need to execute with command-line arguments:

```
java InitArray 5 0 4
Index   Value
    0       0
    1       4
    2       8
    3      12
    4      16
```

```
java InitArray 8 1 2
Index   Value
    0       1
    1       3
    2       5
    3       7
    4       9
    5      11
    6      13
    7      15
```

Class Arrays - Helper Methods for Arrays

- **Class Arrays** helps you avoid reinventing the wheel by **providing static methods for common array manipulations.**
- These methods include **sort** for sorting an array, **binarySearch** for searching an element in a sorted array, **equals** for comparing arrays and **fill** for placing values into an array.
- These methods are overloaded for primitive-type arrays and for arrays of objects.

```
int[] arr = {5, 3, 2, 6, 1, 8};

Arrays.sort(arr);

System.out.println(Arrays.toString(arr));
```

```
[1, 2, 3, 5, 6, 8]
```

https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html

Class Arrays Example

```
1  // Fig. 7.22: ArrayManipulations.java
2  // Arrays class methods and System.arraycopy.
3  import java.util.Arrays;
4
5  public class ArrayManipulations
6  {
7     public static void main(String[] args)
8     {
9        // sort doubleArray into ascending order
10       double[] doubleArray = { 8.4, 9.3, 0.2, 7.9, 3.4 };
11       Arrays.sort(doubleArray);
12       System.out.printf("%ndoubleArray: ");
13
14       for (double value : doubleArray)
15          System.out.printf("%.1f ", value);
16
17       // fill 10-element array with 7s
18       int[] filledIntArray = new int[10];
19       Arrays.fill(filledIntArray, 7);
20       displayArray(filledIntArray, "filledIntArray");
21
22       // copy array intArray into array intArrayCopy
23       int[] intArray = { 1, 2, 3, 4, 5, 6 };
24       int[] intArrayCopy = new int[intArray.length];
25       System.arraycopy(intArray, 0, intArrayCopy, 0, intArray.length);
26       displayArray(intArray, "intArray");
27       displayArray(intArrayCopy, "intArrayCopy");
28
29       // compare intArray and intArrayCopy for equality
30       boolean b = Arrays.equals(intArray, intArrayCopy);
31       System.out.printf("%n%nintArray %s intArrayCopy%n",
32          (b ? "==" : "!="));
```

Output:

```
doubleArray: 0.2 3.4 7.9 8.4 9.3
filledIntArray: 7 7 7 7 7 7 7 7 7 7
intArray: 1 2 3 4 5 6
intArrayCopy: 1 2 3 4 5 6

intArray == intArrayCopy
intArray != filledIntArray
Found 5 at element 4 in intArray
8763 not found in intArray
```

[Question] Can you find out which method in Arrays Class we use in this program?

```
33
34          // compare intArray and filledIntArray for equality
35          b = Arrays.equals(intArray, filledIntArray);
36          System.out.printf("intArray %s filledIntArray%n",
37             (b ? "==" : "!="));
38
39          // search intArray for the value 5
40          int location = Arrays.binarySearch(intArray, 5);
41
42          if (location >= 0)
43             System.out.printf(
44                "Found 5 at element %d in intArray%n", location);
45          else
46             System.out.println("5 not found in intArray");
47
48          // search intArray for the value 8763
49          location = Arrays.binarySearch(intArray, 8763);
50
51          if (location >= 0)
52             System.out.printf(
53                "Found 8763 at element %d in intArray%n", location);
54          else
55             System.out.println("8763 not found in intArray");
56       }
57
58       // output values in each array
59       public static void displayArray(int[] array, String description)
60       {
61          System.out.printf("%n%s: ", description);
62
63          for (int value : array)
64             System.out.printf("%d ", value);
65       }
66    } // end class ArrayManipulations
```

Output:

```
doubleArray: 0.2 3.4 7.9 8.4 9.3
filledIntArray: 7 7 7 7 7 7 7 7 7 7
intArray: 1 2 3 4 5 6
intArrayCopy: 1 2 3 4 5 6

intArray == intArrayCopy
intArray != filledIntArray
Found 5 at element 4 in intArray
8763 not found in intArray
```

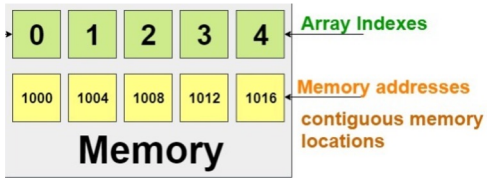
[Question] Can you find out which method in Arrays Class we use in this program?

Limitation of Array in Java

- Just like C, **the size of an array can't be changed.**
- The **dimension of an array is determined the moment the array is created**, and **cannot be changed later on.**
- **If you want a bigger array you have to instantiate a new one, and copy elements from the old array to the new one.** (Like in the below code)
- **The array occupies an amount of memory that is proportional to its size, independently of the number of elements that are actually there.**
- If we want to keep the elements of the collection ordered, and insert a new value in its correct position, or remove it, then, for each such operation we may need to move many elements. this is **very inefficient**.

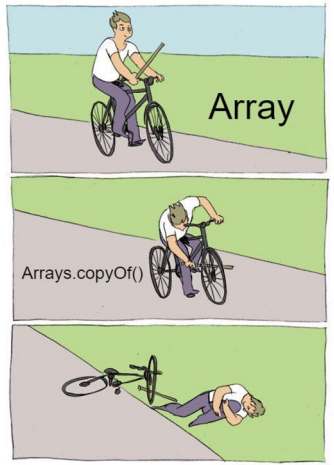
Extend array by two more elements (inconvenient way):

```
int[] oldArray = new int[] { 10, 11, 12 };

// allocating space for 5 integers
int[] newArray = Arrays.copyOf(oldArray, newLength: oldArray.length + 2);

// adding new integers at index 3, 4
newArray[3] = 13;
newArray[4] = 14;

System.out.println(Arrays.toString(newArray));
```



Output:

```
[10, 11, 12, 13, 14]
```



Collections & Class ArrayList

Resizable Collections

ArrayList

- The Java API provides several **predefined data structures**, called **collections**, used to **store groups of related objects**.
- ArrayList is a **resizable array** implementation in java. ArrayList **grows dynamically** and ensures that there is **always a space to add elements**.
- The collection class **ArrayList<T>** (package java.util) provides a **convenient solution** to this problem—it can **dynamically change its size** to accommodate **more elements**. The **T** (by convention) is a **placeholder**.

ArrayList to store String elements:

```
ArrayList<String> list;
```

ArrayList to store Integer elements:

```
ArrayList<Integer> integers;
```

ArrayList Methods

Method	Description
`add`	Adds an element to the *end* of the `ArrayList`.
`clear`	Removes all the elements from the `ArrayList`.
`contains`	Returns `true` if the `ArrayList` contains the specified element; otherwise, returns `false`.
`get`	Returns the element at the specified index.
`indexOf`	Returns the index of the first occurrence of the specified element in the `ArrayList`.
`remove`	Overloaded. Removes the first occurrence of the specified value or the element at the specified index.
`size`	Returns the number of elements stored in the `ArrayList`.
`trimToSize`	Trims the capacity of the `ArrayList` to the current number of elements.

Fig. 7.23 | Some methods and properties of class `ArrayList<T>`.

ArrayList & Array Example

```java
int[] arr = new int[3];
arr[0] = 1;
arr[1] = 2;
arr[2] = 3;
// We cannot add more elements to array arr[]

// ArrayList
// Need not to specify size
ArrayList<Integer> arrL = new ArrayList<Integer>();

// Adding elements to ArrayList object
arrL.add(1);
arrL.add(2);
arrL.add(3);
arrL.add(4);
// We can add more elements to arrL

// Print and display Arraylist elements
System.out.println(arrL);
// Print and display array elements
System.out.println(Arrays.toString(arr));
```

Output:

```
[1, 2, 3, 4]
[1, 2, 3]
```

ArrayList

Array

ArrayList Example

```
1  // Fig. 7.24: ArrayListCollection.java
2  // Generic ArrayList<T> collection demonstration.
3  import java.util.ArrayList;
4
5  public class ArrayListCollection
6  {
7     public static void main(String[] args)
8     {
9        // create a new ArrayList of Strings with an initial capacity of 10
10       ArrayList<String> items = new ArrayList<String>();
11
12       items.add("red"); // append an item to the list
13       items.add(0, "yellow"); // insert "yellow" at index 0
14
15       // header
16       System.out.print(
17          "Display list contents with counter-controlled loop:");
18
19       // display the colors in the list
20       for (int i = 0; i < items.size(); i++)
21          System.out.printf(" %s", items.get(i));
22
23       // display colors using enhanced for in the display method
24       display(items,
25          "%nDisplay list contents with enhanced for statement:");
26
27       items.add("green"); // add "green" to the end of the list
28       items.add("yellow"); // add "yellow" to the end of the list
29       display(items, "List with two new elements:");
30
31       items.remove("yellow"); // remove the first "yellow"
32       display(items, "Remove first instance of yellow:");
```

Output:

```
Display list contents with counter-controlled loop: yellow red
Display list contents with enhanced for statement: yellow red
List with two new elements: yellow red green yellow
Remove first instance of yellow: red green yellow
Remove second list element (green): red yellow
"red" is in the list
Size: 2
```

```
33
34          items.remove(1); // remove item at index 1
35          display(items, "Remove second list element (green):");
36
37          // check if a value is in the List
38          System.out.printf("\"red\" is %sin the list%n",
39             items.contains("red") ? "": "not ");
40
41          // display number of elements in the List
42          System.out.printf("Size: %s%n", items.size());
43       }
44
45       // display the ArrayList's elements on the console
46       public static void display(ArrayList<String> items, String header)
47       {
48          System.out.printf(header); // display header
49
50          // display each element in items
51          for (String item : items)
52             System.out.printf(" %s", item);
53
54          System.out.println();
55       }
56    } // end class ArrayListCollection
```

Output:

```
Display list contents with counter-controlled loop: yellow red
Display list contents with enhanced for statement: yellow red
List with two new elements: yellow red green yellow
Remove first instance of yellow: red green yellow
Remove second list element (green): red yellow
"red" is in the list
Size: 2
```

Array [ ] vs ArrayList < >

- **Array cannot change the length of array** once created while **ArrayList can grow** to meet the demand.
- We **cannot store primitives in ArrayList, it can only store objects**. But array can contain both primitives and objects in Java. So you need to use alternative reference data type for primitive data types. (int -> Integer)
- **Array is faster** for reading and processing than ArrayList.

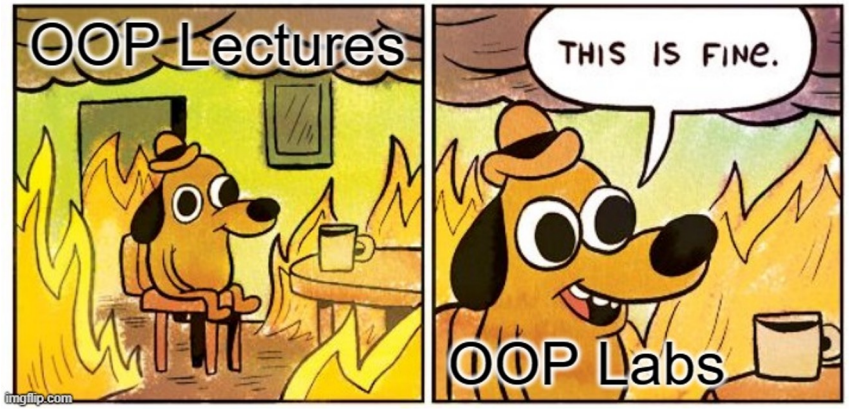
https://stackoverflow.com/questions/716597/array-or-list-in-java-which-is-faster

Which situation below need array or arrayList?

- A ticket in the 649 game with 6 numbers?
- A 3-dice game? Such as, tài xỉu.
- A classroom with a list of students like Edusoft system?
- An ecommerce order with any amount of items like on Shopee.
- Can you think of other examples?

Recap

- **Increment and Decrement Operators**
- **Scope of Declarations**
  - this keyword
- **Array:**
  - Declare and Create Array
  - Loop through Array
  - Pass Arrays to Methods
    - Pass by Value vs Pass by Reference
  - Multidimensional Arrays
  - Class Arrays for helper methods
- **ArrayList** (Collections Class)
- **Array vs ArrayList**

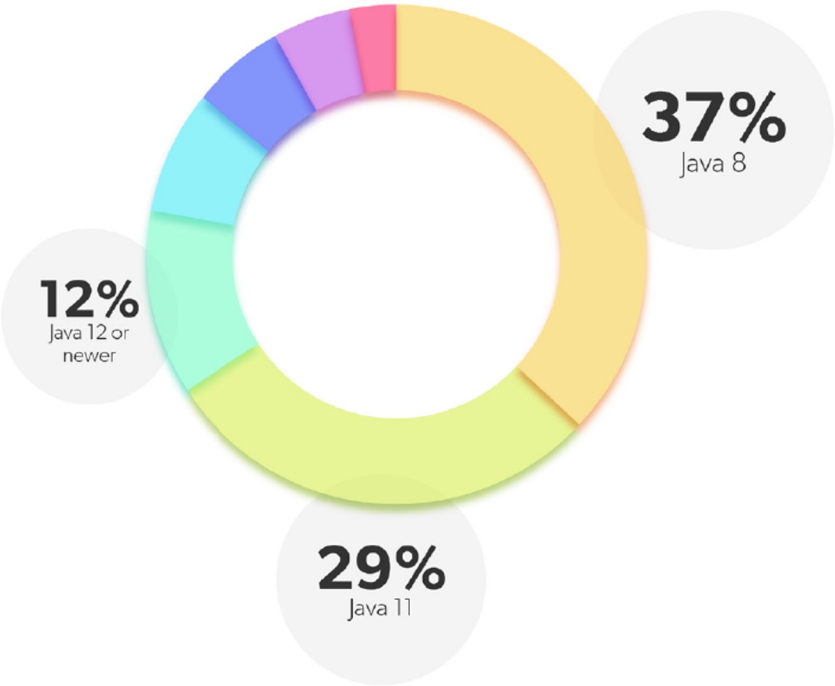


Java Developer Productivity Survey 2022

To get you started with labs this week

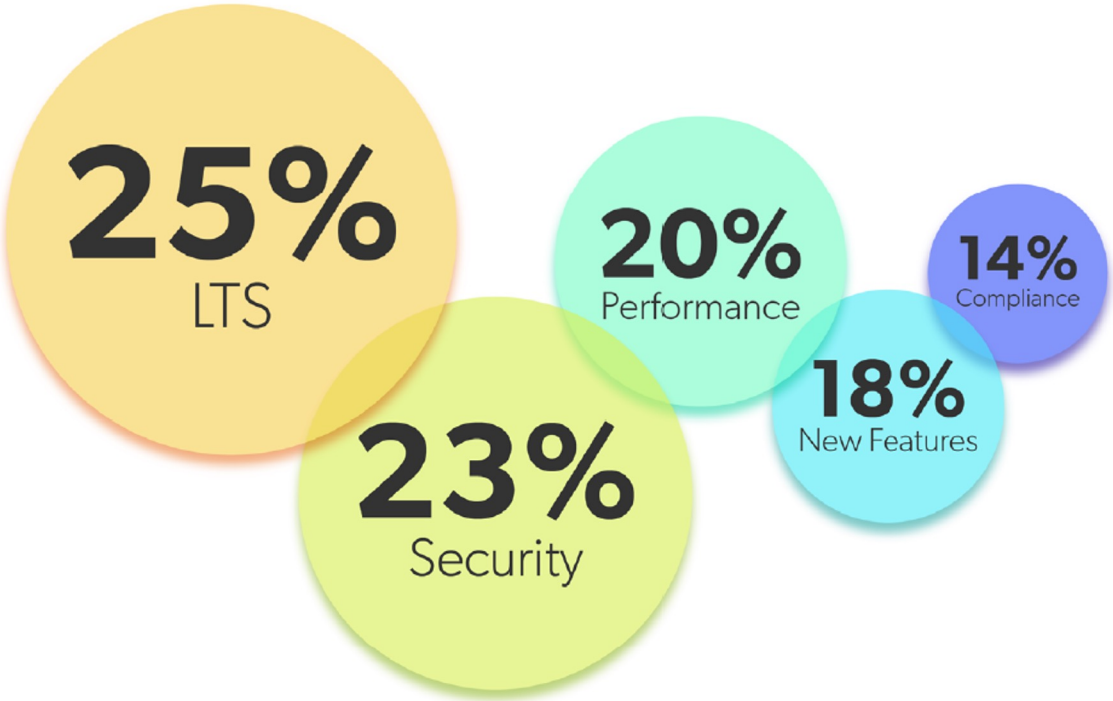
https://www.jrebel.com/resources/java-developer-productivity-report-2022

Which JDK Programming Languages are You Using in Your Main Application?



- Java 8 **37%**
- Java 11 **29%**
- Java 12 or newer **12%**
- Kotlin **8%**
- Groovy **6%**
- Java 7 or older **5%**
- Scala **3%**

Which Factors Influence Your Decision to Upgrade JDK Versions?



LTS **25%**

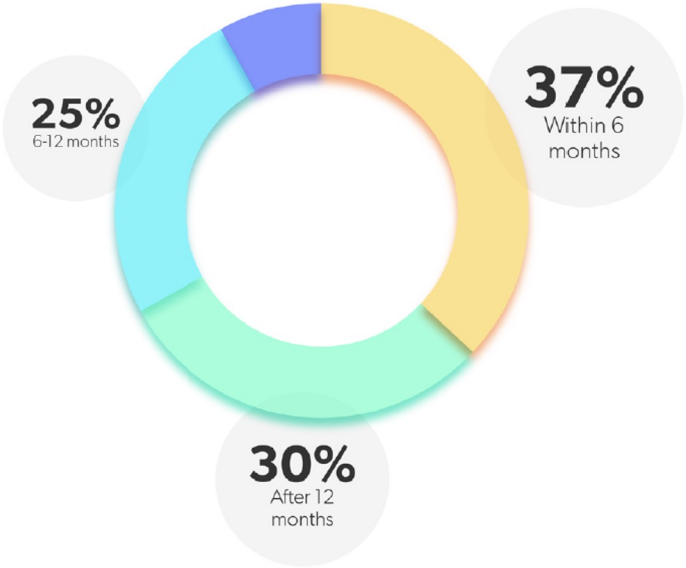
Security **23%**

Performance **20%**

New Features **18%**

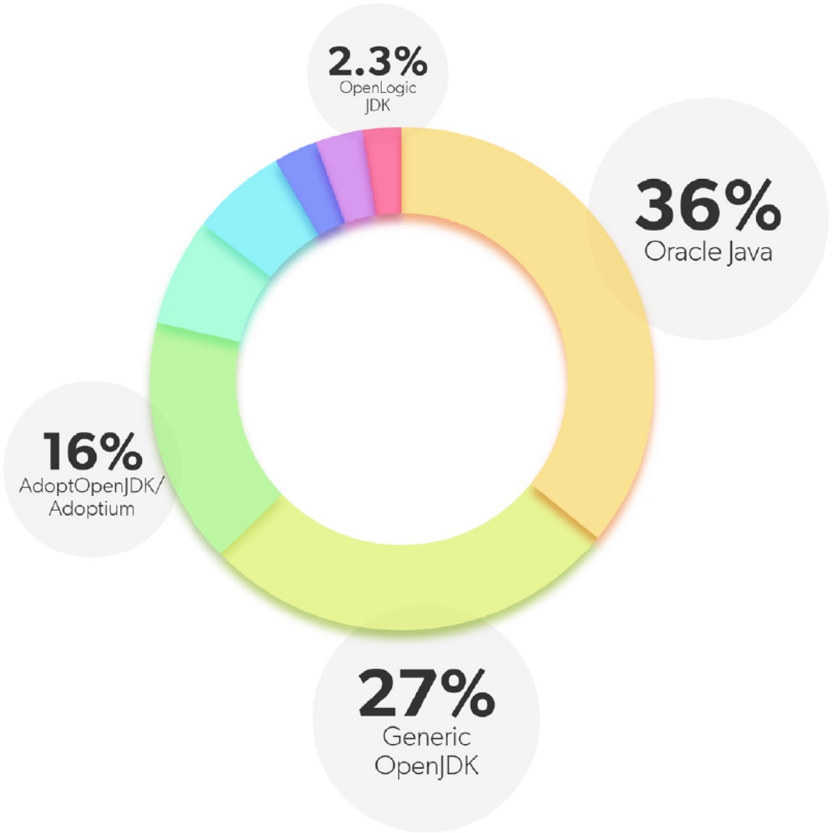
Compliance **14%**

When Will You Upgrade to JDK 17?



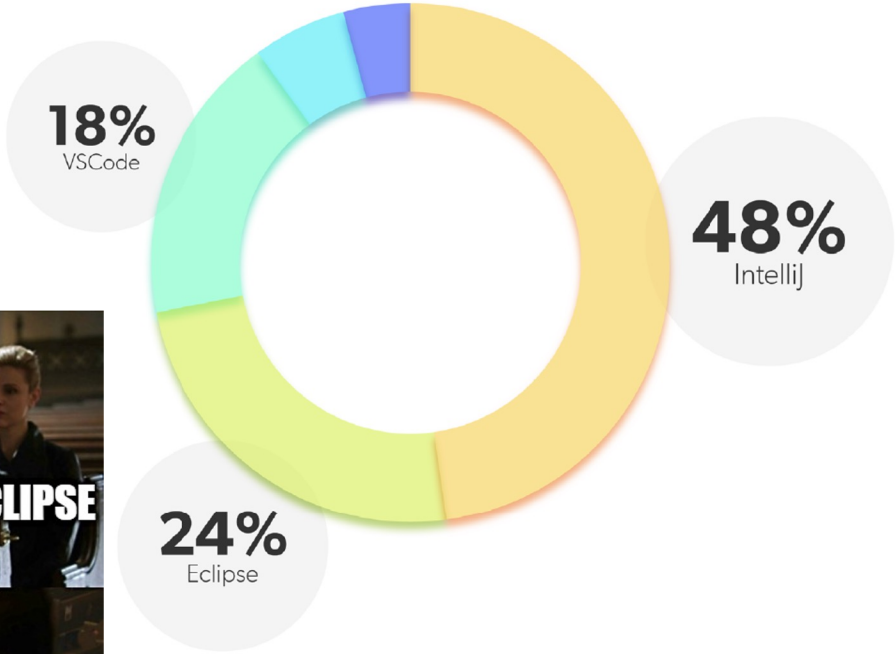
- Within the next 6 months **37%**
- Sometime after the next 12 months **30%**
- Between 6-12 months **25%**
- Never **8%**

What JRE/JDK Distribution Do You Use?



- Oracle Java **36%**
- Generic OpenJDK **27%**
- AdoptOpenJDK/Adoptium **16%**
- Amazon Corretto **7%**
- Azul Zulu **6%**
- GraalVM **3%**
- Other **3%**
- OpenLogic JDK **2.3%**

What Developer IDE Do You Use Professionally?



IntelliJ **48%**

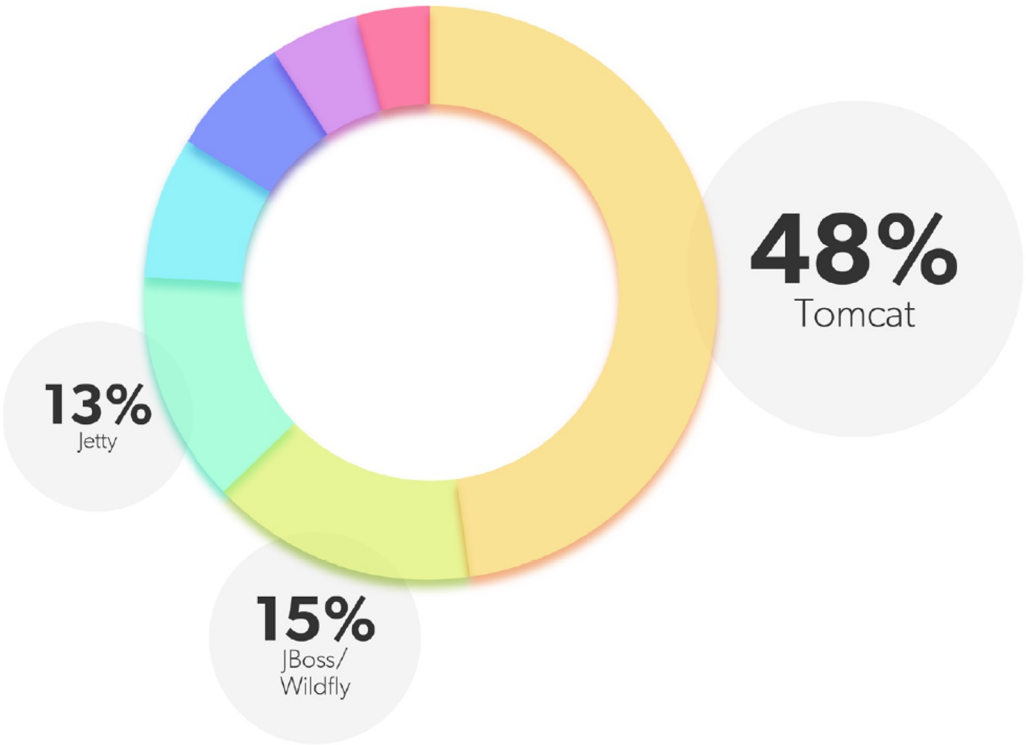
Eclipse **24%**

VSCode **18%**

Netbeans **6%**

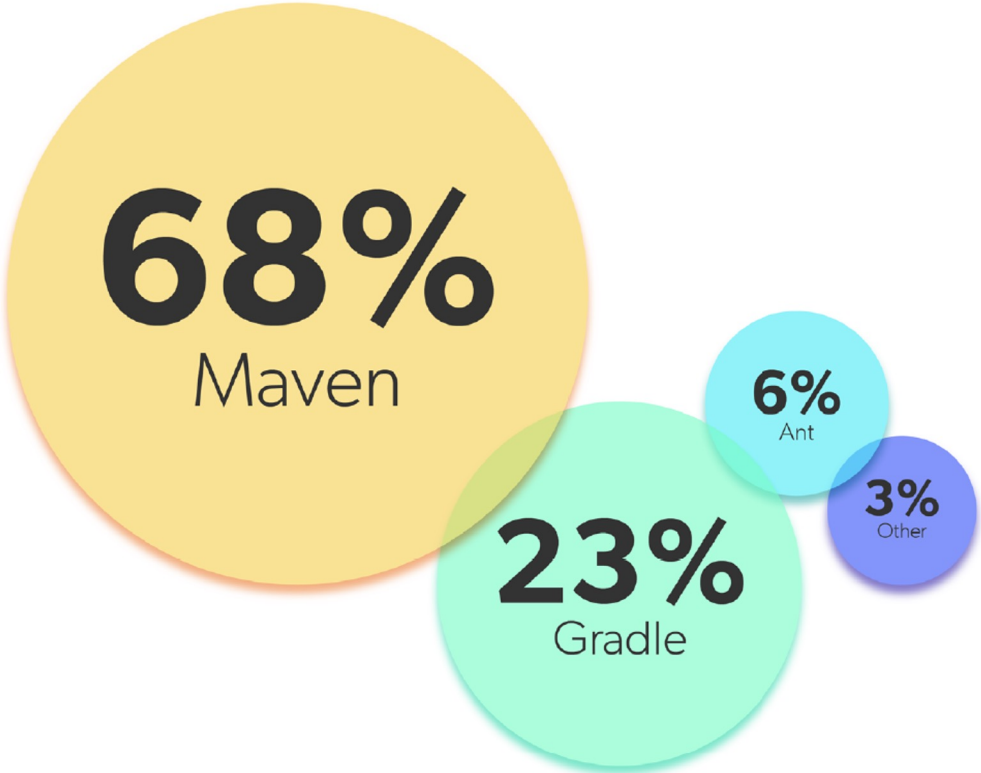
Other **4%**

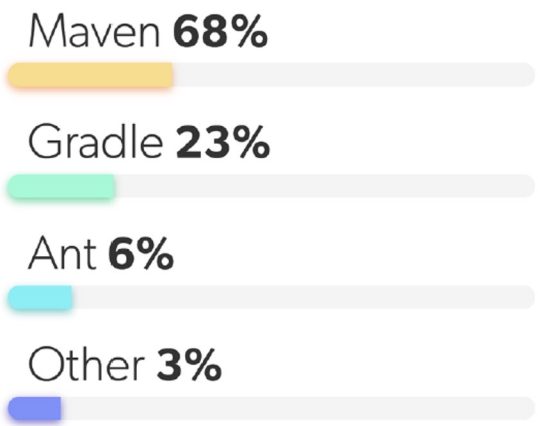
What Application Server Do You Use on Your Main Application?



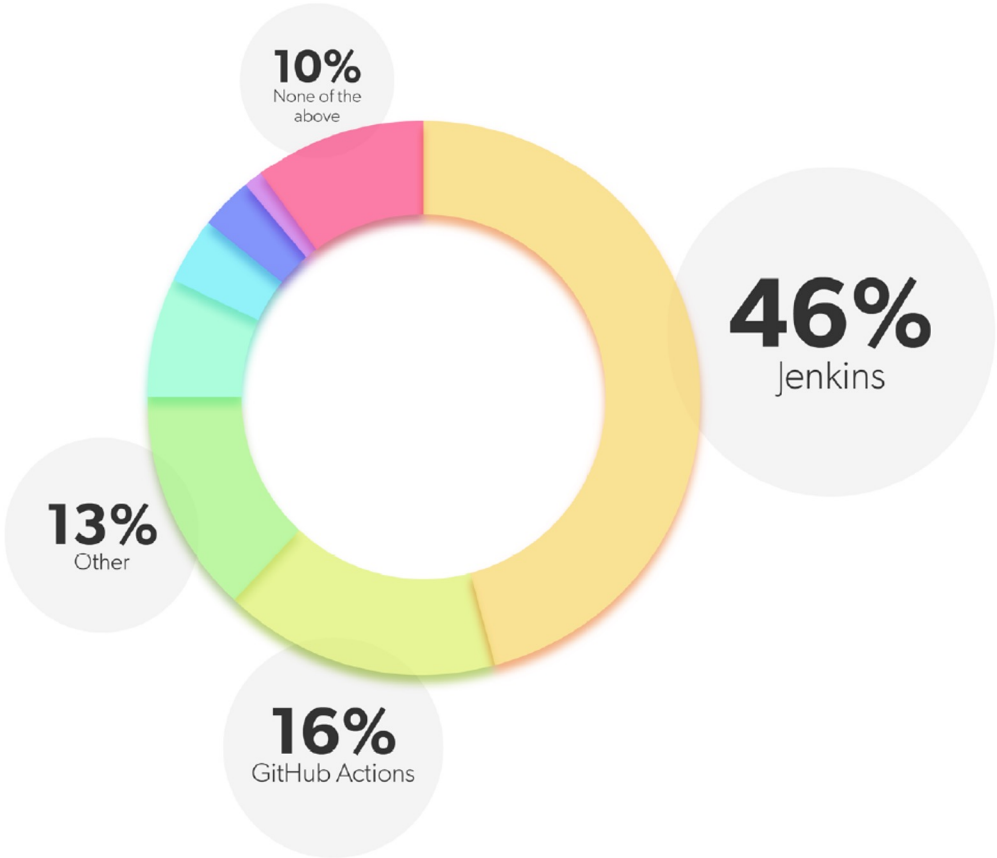
- Tomcat **48%**
- JBoss/Wildfly **15%**
- Jetty **13%**
- Other **8%**
- WebLogic **7%**
- WebSphere **5%**
- GlassFish **4%**

What Build Tool Do You Use in Your Main Application?



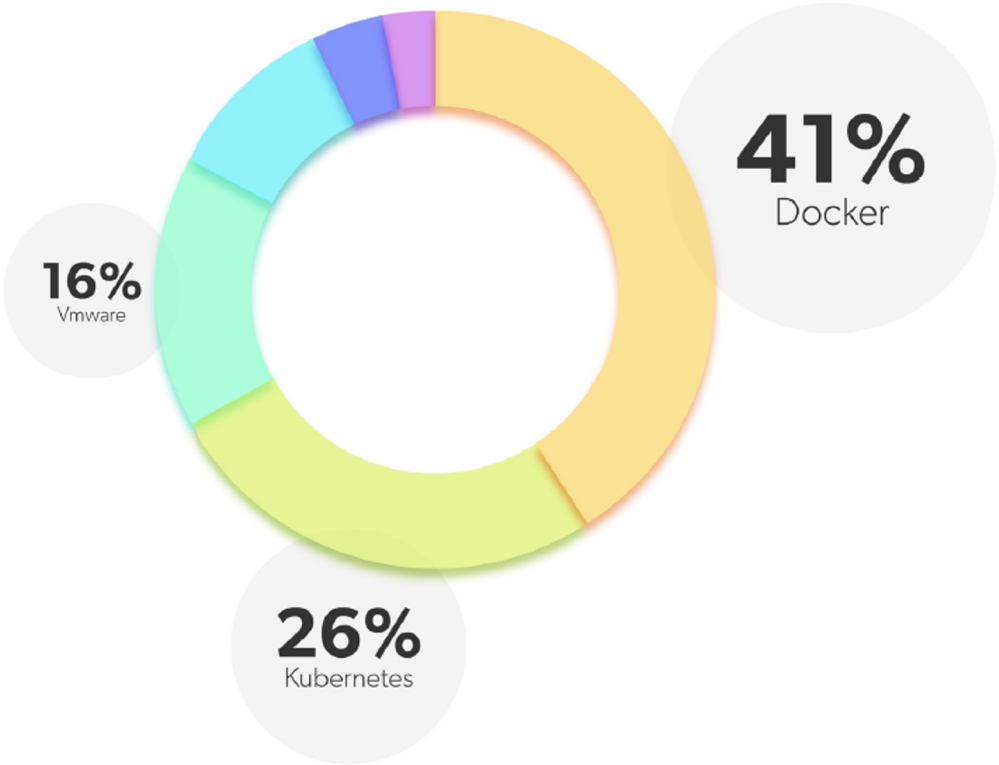


Which CI/CD Technologies Are You Using?



- Jenkins **46%**
- GitHub Actions **16%**
- Other **13%**
- Bamboo **7%**
- TeamCity **4%**
- Circle CI **3%**
- Travis CI **1%**
- None of the above **10%**

Which Virtual Machine Platform Do You Use?

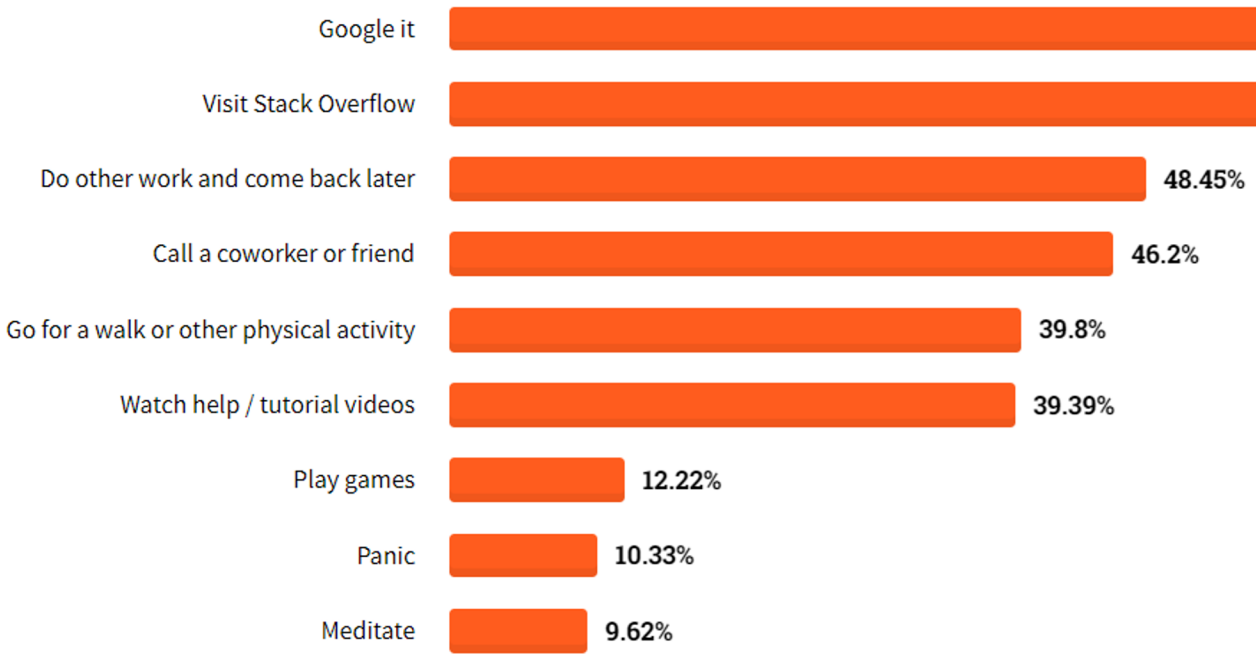


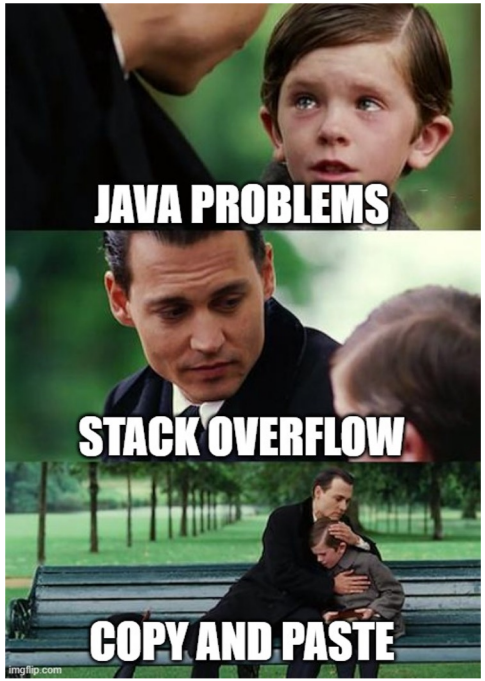
- Docker **41%**
- Kubernetes **26%**
- Vmware **16%**
- N/A **10%**
- Other **4%**
- Vagrant **3%**

Stack Overflow Developer Survey 2021

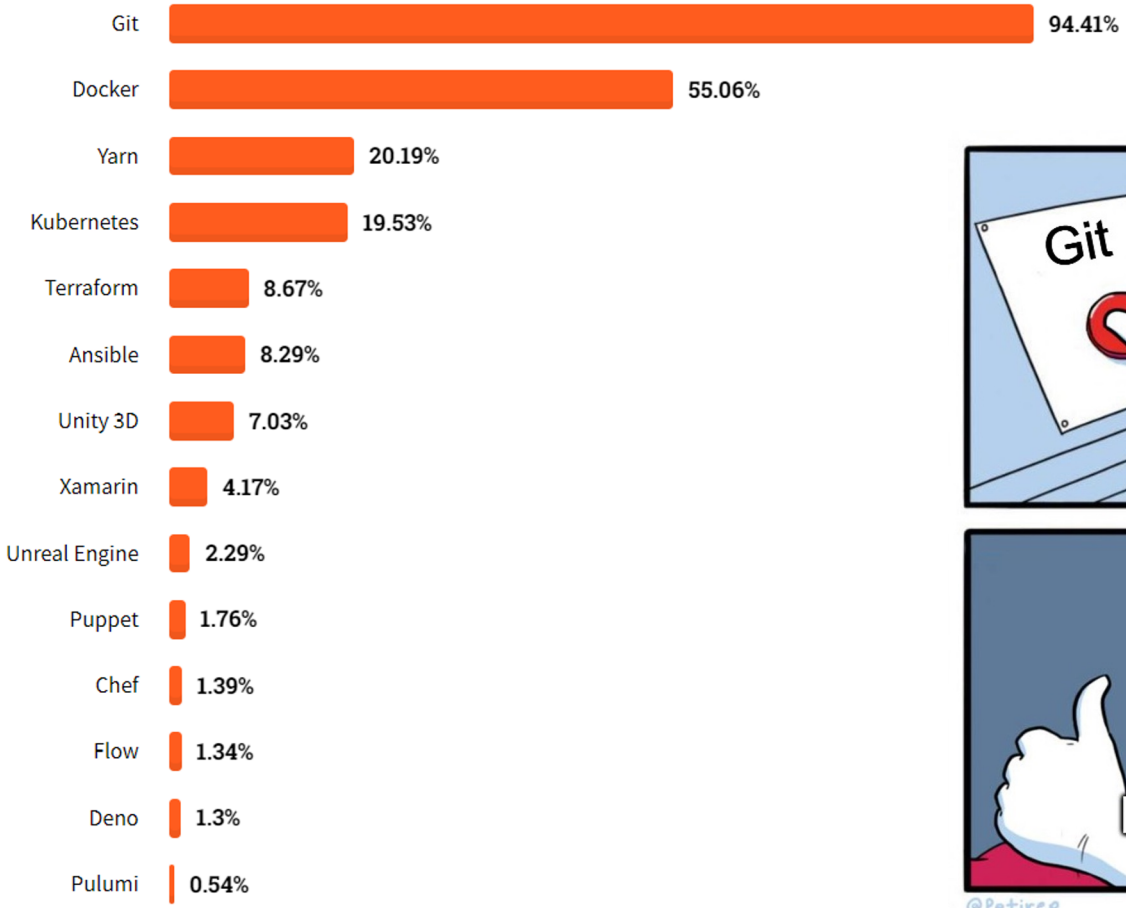
https://insights.stackoverflow.com/survey/2021#experience-learn-code

What do you do when you get stuck?





Popular Tools



Thank you for your listening!