Associativity in Java

Definition

Associativity in Java defines **the order in which operators of the same precedence are evaluated**.
When two or more operators with the **same precedence** appear in an expression, **associativity** tells us whether Java will evaluate them from **left-to-right** or **right-to-left**.

OPERATOR	TYPE	ASSOCIAVITY
( ) [ ] . ->		left-to-right
++ -- + - ! ~ (*type*) * & sizeof	Unary Operator	right-to-left
* / %	Arithmetic Operator	left-to-right
+ -	Arithmetic Operator	left-to-right
<< >>	Shift Operator	left-to-right
< <= > >=	Relational Operator	left-to-right
== !=	Relational Operator	left-to-right
&	Bitwise AND Operator	left-to-right
^	Bitwise EX-OR Operator	left-to-right
|	Bitwise OR Operator	left-to-right
&&	Logical AND Operator	left-to-right
||	Logical OR Operator	left-to-right
? :	Ternary Conditional Operator	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Operator	right-to-left
,	Comma	left-to-right

Example:

```
int result = 100 / 5 * 2;
// First: 100 / 5 = 20
// Then: 20 * 2 = 40
// Output = 40
```

Resulting Data Type in Java (Type Promotion in Expressions)

Definition

In Java, when arithmetic operations are performed between **different data types**, the smaller type is **promoted** to a larger type before evaluation.

This rule is called **Type Promotion** or **Type Conversion in Expressions**.

Key Rules:

- **byte, short, and char → promoted to int** before any operation.
  - Example:
  - `short s1 = 5, s2 = 10;`
  - `int result = s1 + s2; // result is int, not short`
- **If one operand is long → result becomes long**.
  - Example:
  - `int a = 10;`
  - `long b = 20;`
  - `long result = a + b; // result is long`
- **If one operand is float → result becomes float**.

Common Examples:

- **int + short → int**
- `int a = 5;`
- `short b = 2;`
- `int result = a + b; // result is int`
- **byte + byte → int**
- `byte x = 10, y = 20;`
- `int result = x + y; // result is int`
- **char + int → int**
- `char c = 'A'; // Unicode 65`
- `int result = c + 1; // 65 + 1 = 66, result is int`
- **int + long → long**
- `int a = 10;`
- `long b = 100L;`
- `long result = a + b; // result is long`
- **long + float → float**
- `long l = 100L;`
- `float f = 3.5f;`
- `float result = l + f; // result is float`

Type Promotion Hierarchy:

```
byte → short → int → long → float → double
```

When different types are mixed in an operation, Java **promotes them to the largest type in the chain**.

Arrays in Java

1. Definition

- An array in Java is a **collection of elements of the same data type** stored in a contiguous memory location.
- It is used to store multiple values in a single variable, instead of declaring separate variables for each value.

2. Key Features

- Fixed size: The size of an array is defined at the time of creation and cannot be changed.
- Homogeneous elements: All elements must be of the same type.
- Index-based: Each element can be accessed by its index (starting from `0`).

3. Declaration and Initialization

There are two ways:

Declaration:

```
int[] arr;    // preferred way
int arr[];    // also valid
```

Memory allocation:

```
arr = new int[5];   // creates an array of size 5
```

Declaration + Initialization together:

```
int[] arr = new int[5];    // default values (0 for int, null for objects)
```

Initialization with values:

```
int[] arr = {10, 20, 30, 40, 50};
```

4. Accessing Elements

- Using index:

```
System.out.println(arr[0]);   // prints first element
```

- Changing value:

```
arr[2] = 100;   // updates 3rd element
```

5. Array Length

- `arr.length` gives the total size of the array.

```
System.out.println(arr.length);
```

6. Types of Arrays

1. **One-Dimensional Array**

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

3. **Two-Dimensional Array (Matrix)**

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

Access element:

```
System.out.println(matrix[1][0]);  // prints 3
```

5. **Multidimensional Array**
   - Arrays inside arrays (3D or more).

```
int[][][] arr3D = new int[2][3][4];
```

7. Traversing Arrays

- Using **for loop**:

```
for(int i=0; i<arr.length; i++) {
    System.out.println(arr[i]);
}
```

- Using **for-each loop**:

```
for(int num : arr) {
    System.out.println(num);
}
```

8. Default Values in Arrays

- Numeric types → `0`
- char → `'\u0000'` (null character)
- boolean → `false`
- objects → `null`

9. Advantages

- Stores multiple values under one name.
- Easy to access using indexes.
- Better performance for fixed-size data.

10. Limitations

- Fixed size (cannot grow/shrink dynamically).
- Only stores elements of the same type.
- For dynamic data, **ArrayList** is preferred.

What is a 2D Array in Java

- A **2D array** is like a **table (rows and columns)**.

- It is an **array of arrays**.
  Example: `int[][] arr = new int[3][4];`
  → This creates a table with 3 rows and 4 columns.

Declaration of 2D Array

There are two ways:

1. **Declaration + Size Allocation**

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

- 3 rows, 4 columns
- Default values = `0`

2. **Declaration + Initialization**

```
int[][] arr = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
```

- 3x3 matrix initialized directly.

Accessing Elements

```
arr[0][2];   // Row 0, Column 2 → 3
arr[2][1];   // Row 2, Column 1 → 8
```

Traversing a 2D Array

Using **nested for loops**:

```
for (int i = 0; i < arr.length; i++) {           // rows
    for (int j = 0; j < arr[i].length; j++) {    // columns
        System.out.print(arr[i][j] + " ");
    }
    System.out.println();
}
```

Input a 2D Array (Using Scanner)

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        int[][] arr = new int[2][3]; // 2 rows, 3 columns

        // Input
        for (int i = 0; i < 2; i++) {
            for (int j = 0; j < 3; j++) {
                arr[i][j] = sc.nextInt();
            }
        }
```

```
        // Output
        for (int i = 0; i < 2; i++) {
            for (int j = 0; j < 3; j++) {
                System.out.print(arr[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

Key Points

- `arr.length` → number of rows
- `arr[i].length` → number of columns in row `i`
- Default values in int array = `0`
- 2D arrays can also be **jagged** (rows having different column sizes).

Arrays Utility Class in Java

Java provides a built-in class **`java.util.Arrays`** that contains **static methods** to work with arrays.
Instead of writing code from scratch, we can use this class to perform **common operations** like sorting, searching, comparing, and filling arrays.

Common Methods of **`Arrays`** Class

1. `sort()`

- Sorts the array elements in ascending order.

```
import java.util.Arrays;

public class Demo {
    public static void main(String[] args) {
        int[] arr = {5, 2, 8, 1};
        Arrays.sort(arr);
        System.out.println(Arrays.toString(arr)); // [1, 2, 5, 8]
    }
}
```

2. `binarySearch()`

- Searches an element in a **sorted array**.
- Returns **index** if found, otherwise returns **negative value**.

```
int[] arr = {1, 2, 5, 8};
int index = Arrays.binarySearch(arr, 5);
System.out.println(index); // 2
```

3. `equals()`

- Compares two arrays (element by element).
- Returns true if both arrays are of same length and content.

```
int[] a1 = {1, 2, 3};
```

```
int[] a2 = {1, 2, 3};
System.out.println(Arrays.equals(a1, a2)); // true
```

4. `fill()`

- Fills the entire array with a given value.

```
int[] arr = new int[5];
Arrays.fill(arr, 10);
System.out.println(Arrays.toString(arr)); // [10, 10, 10, 10, 10]
```

5. `copyOf() and copyOfRange()`

- Copies elements from one array into another.

```
int[] arr = {1, 2, 3, 4, 5};
int[] copy = Arrays.copyOf(arr, 3); // [1, 2, 3]
int[] range = Arrays.copyOfRange(arr, 1, 4); // [2, 3, 4]
```

6. `toString()`

- Returns a string representation of the array.

```
int[] arr = {1, 2, 3};
System.out.println(Arrays.toString(arr)); // [1, 2, 3]
```

Methods in Java

- A **method** is a block of code that performs a specific task.
- It helps in **code reusability** and **better organization**.
- Java methods are similar to functions in C/C++.

Syntax of a Method

```
returnType methodName(parameters) {
    // method body
    return value;  // if returnType is not void
}
```

Example:

```
int add(int a, int b) {
    return a + b;
}
```

Types of Methods

- **Predefined Methods (Built-in)**
  - Already provided by Java libraries.
  - **Example:**

    `Math.sqrt(25)` → returns 5.0
    `System.out.println("Hello")`

- **User-defined Methods**
  - Created by programmer as per requirement.

- **Example:**

```
void greet() {
    System.out.println("Welcome to Java!");
}
```

Method Signature

- Method name + parameter list = **Method Signature**
  Example: `add(int a, int b)`

Method Calling:

To execute a method, we **call** it.

```
class Main {
    static void greet() {
        System.out.println("Hello!");
    }

    public static void main(String[] args) {
        greet(); // calling method
    }
}
```

Return Type:

`void` → no value returned.
Any data type (`int`, `double`, `String`, etc.) → returns value.

```
int square(int num) {
    return num * num;
}
```

Parameters and Arguments:

- **Parameters**: variables defined inside method declaration.
- **Arguments**: actual values passed while calling.

```
int sum(int a, int b) {  // parameters
    return a + b;
}

sum(5, 10);  // arguments
```

Method Overloading:

Same method name but **different parameters** (number or type).

```
int add(int a, int b) {
    return a + b;
}
double add(double a, double b) {
    return a + b;
}
```

Static vs Non-static Methods:

Static Method: Can be called without creating object.
Example: `Math.max(5, 10)`
Non-static Method: Needs object to be called.

```
class Test {
    void display() {
        System.out.println("Non-static method");
    }
    public static void main(String[] args) {
        Test obj = new Test();
        obj.display(); // object required
    }
}
```

What is Varargs

`Varargs` (variable-length arguments) allow a method to accept **zero or multiple arguments** of the same type.
Declared using `...` (three dots).

```
void printNumbers(int... nums) {
    for (int n : nums) {
        System.out.print(n + " ");
    }
}
```

Call:

```
printNumbers(1, 2, 3, 4);  // accepts multiple arguments
```

Method Overloading with Varargs:

We can overload methods that use varargs by **changing parameter types** or **number of parameters**.
But, **confusion may occur** if normal parameter methods and varargs look similar.

Example 1: Different Parameter Types

```
class Test {
    void show(int... a) {
        System.out.println("int varargs method");
    }
    void show(String... s) {
        System.out.println("String varargs method");
    }

    public static void main(String[] args) {
        Test t = new Test();
        t.show(1, 2, 3);        // calls int version
        t.show("A", "B", "C");  // calls String version
    }
}
```

Example 2: Normal Parameter + Varargs

```
class Test {
    void display(int a, int b) {
        System.out.println("Normal method: " + (a + b));
    }
    void display(int... nums) {
        System.out.println("Varargs method, count = " + nums.length);
    }

    public static void main(String[] args) {
        Test t = new Test();
        t.display(5, 10);        // calls normal method (exact match)
        t.display(1, 2, 3, 4, 5); // calls varargs method
```

```
    }
}
```

Recursion in Java

- **Recursion** is a process in which a method calls **itself** directly or indirectly.
- Used for solving problems that can be broken into **smaller sub-problems of the same type**.

Syntax of Recursive Method:

```
returnType methodName(parameters) {
    // base condition
    if (condition) {
        return value;
    }
    // recursive call
    return methodName(modifiedParameters);
}
```

Key Terms:

1. **Base Case** → The condition that stops recursion.
   (Without base case, recursion will go infinite → `StackOverflowError`).
2. **Recursive Case** → The part where method calls itself.

Example 1: Factorial Using Recursion

```
class RecursionDemo {
    static int factorial(int n) {
        if (n == 0 || n == 1) {   // base case
            return 1;
        }
        return n * factorial(n - 1);  // recursive call
    }

    public static void main(String[] args) {
        System.out.println(factorial(5)); // 120
    }
}
```

Example 2: Fibonacci Using Recursion

```
class RecursionDemo {
    static int fib(int n) {
        if (n <= 1) {   // base case
            return n;
        }
        return fib(n - 1) + fib(n - 2);  // recursive case
    }

    public static void main(String[] args) {
        System.out.println(fib(6)); // 8
    }
}
```