# 1. How to Print Output in Java

## ◈ Key Concepts:

- `System.out.print()` → Prints text without moving to the next line.
- `System.out.println()` → Prints text and moves to the next line.
- `System.out.printf()` → Prints formatted output using format specifiers.

## ◈ Syntax:

1. `System.out.print("text");`
2. `System.out.println("text");`
3. `System.out.printf("format", values);`
    - `%d` → integer
    - `%f` → float/double
    - `%s` → string
    - `%.2f` → float with 2 decimal places

## ✓ Points to Remember:

- Every Java print statement ends with a semicolon `;`.
- `println()` adds a line break automatically.
- `printf()` gives control over output formatting.
- Use escape characters like `\n` (newline), `\t` (tab).

## ◈ Example:

**Q:** Print "Hello" and "World" on separate lines

**A:**

```java
System.out.println("Hello");
System.out.println("World");
```

---

## 2. What is a Variable in Java?

### ◈ Key Concepts:

- Variables are containers used to store data values.
- Java requires you to declare the type of variable before using it.
- The type defines what kind of data the variable can hold.

### ◈ Formulae (Syntax):

1. `datatype variableName = value;`
2. Examples:
   - `int age = 20;`
   - `String name = "Rahul";`
   - `float price = 99.5f;`

### ✓ Points to Remember:

- Variable names are case-sensitive and follow camelCase convention.
- Cannot start with numbers or use Java keywords.
- Use `int`, `double`, `char`, `boolean`, `String`, etc., based on the data type.
- Variables can be declared without assigning values initially.

### ◈ Example:

**Q:** Declare a string and an integer variable, then print them
**A:**

```java
String city = "Mumbai";
int population = 20000000;
System.out.println(city + " has population of " + population);
```

---

## 3. What are Conditionals in Java?

### ❓ Key Concepts:

- Conditionals allow a program to make decisions based on certain conditions.
- Common conditional statements in Java are:
  - `if`
  - `if-else`
  - `if-else if-else`
  - `switch`

## Syntax:

```
// if statement
if (condition) {
    // code block
}

// if-else
if (condition) {
    // block if true
} else {
    // block if false
}

// if-else if-else
if (condition1) {
    // block 1
} else if (condition2) {
    // block 2
} else {
    // default block
}

// switch statement
switch (expression) {
    case value1:
        // code
        break;
    case value2:
        // code
        break;
    default:
        // default code
}
```

## 4. What are Loops in Java?

### ⯑ Key Concepts:

- Loops are used to execute a block of code repeatedly.
- Java has three main types of loops:
  - `for` loop — when number of iterations is known.
  - `while` loop — when the condition is checked before the block runs.
  - `do-while` loop — executes the block at least once, then checks the condition.

### ⯑ Syntax:

```java
// for loop
for (initialization; condition; update) {
    // code to run
}

// while loop
while (condition) {
    // code to run
}

// do-while loop
do {
    // code to run
} while (condition);
```

---

## 5. How to Take Input in Java?

### ⯑ Key Concepts:

- **Input:** Receiving data from the user during program execution.
- **Scanner Class:** A built-in Java class ( `java.util.Scanner` ) used to read input from the keyboard.
- **Object Creation:** Create a `Scanner` object to use its methods like `nextInt()` , `nextLine()` .

## ◈ Syntax:

1. **Import Scanner:** `import java.util.Scanner;` (at the top of the file)
2. **Create Scanner Object:** `Scanner sc = new Scanner(System.in);`
3. **Read Input:**
    - `int num = sc.nextInt();` → Reads an integer
    - `double val = sc.nextDouble();` → Reads a decimal
    - `String text = sc.nextLine();` → Reads a line of text

## ✓ Points to Remember:

- Always import `Scanner` before using it.
- `System.in` connects the Scanner to the keyboard.
- After `nextInt()`, add `sc.nextLine()` to clear the buffer before reading a string.
- Close the Scanner with `sc.close();` when done (good practice).

## ◈ Example:

**Q:** How do you take a user's name (string) and age (integer) as input and print them?
**A:**

```java
import java.util.Scanner;

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

        System.out.print("Enter your name: ");
        String name = sc.nextLine(); // Reads string first

        System.out.print("Enter your age: ");
        int age = sc.nextInt(); // Reads integer

        System.out.println("Name: " + name + ", Age: " + age);

        // Scenario: Integer first, then String
        System.out.print("Enter your age: ");
        age = sc.nextInt(); // Reads integer
        sc.nextLine(); // Clears leftover newline

        System.out.print("Enter your name: ");
        name = sc.nextLine(); // Reads string

        System.out.println("Name: " + name + ", Age: " + age);
        sc.close();
    }
}
```

---

## 📃 Functions in Java (aka Methods)

In Java, a **function** is called a **method**. It is a **block of code** that performs a specific task and runs only when called.

---

## ✅ Why Use Functions?

- 🔁 Reusability of code
- 📦 Modular design
- 💠 Cleaner and more readable code
- 🛠 Easy to debug and test

---

## 📌 Basic Syntax

```
returnType functionName(parameters) {
  // code to execute
  return value; // if returnType is not void
}
```

---

## 🚀 Example

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

```java
// Calling the function
int result = add(5, 3); // result = 8
```

---

## 🔤 Types of Functions in Java

| Type | Description |
|---|---|
| 📥 Parameterized | Accepts arguments |

| Type | Description |
|---|---|
| ⊘ Non-Parameterized | Doesn’t take any parameters |
| 🔁 With Return | Returns a value |
| ◇ Void (No Return) | Performs a task but returns nothing (`void`) |

---

## Example: All Variants

```
// 1. No parameters, no return
public void greet() {
  System.out.println("Hello!");
}

// 2. With parameters, no return
public void greetUser(String name) {
  System.out.println("Hello, " + name + "!");
}

// 3. No parameters, with return
public int getDefaultAge() {
  return 18;
}

// 4. With parameters and return
public int square(int x) {
  return x * x;
}
```

---

## ❓ Calling Methods

```
greet();                     // Calls method without parameters
greetUser("Alice");          // Passes argument
int age = getDefaultAge();   // Captures returned value
System.out.println(square(4)); // Output: 16
```

---

## ⚙ Access Modifiers

| Modifier | Description |
|---|---|
| `public` | Accessible from anywhere |
| `private` | Accessible only within the same class |
| `protected` | Accessible in same package or subclass |
| (default) | Package-private (no keyword) |

---

## 📃 Static vs Non-Static Methods

- **Static Method**: Belongs to the class
- **Non-Static Method**: Belongs to an object instance

```
public static void show() { ... } // No need to create object
public void display() { ... }     // Need object to call
```

---

## ❓ Return Statement

```
return value;
```

- Ends the function
- Sends back result (if not `void`)

---

## 🛡 Best Practices

- 📛 Use meaningful function names
- ❓ Keep functions small and focused
- ♻️ Reuse logic through functions
- 📚 Document with comments and JavaDoc

---

## ❓ Interview Tip:

> "In Java, methods (functions) allow **modular programming**, making code more reusable, testable, and maintainable."

---

# 🔢 Number Systems in Programming (Java Focus)

In programming and computer science, a **number system** defines how numbers are represented and manipulated. Java and most other programming languages support **multiple number systems**, mainly:

| Number System | Base | Digits Used | Common Use |
|---|---|---|---|
| Binary | 2 | 0, 1 | Low-level programming, bitwise |
| Octal | 8 | 0–7 | Legacy systems |
| Decimal | 10 | 0–9 | Human-readable numbers |
| Hexadecimal | 16 | 0–9 and A–F | Memory addressing, colors |

---

## ✅ Why Should Programmers Learn Number Systems?

- 🔧 Bitwise operations
- 🎁 Data compression and encoding
- 🎨 Color representation (Hex)
- 📍 Memory and address manipulation
- 💠 Understanding how the computer processes data

---

## 📌 Java Support for Number Systems

```java
int decimal = 100;          // Decimal
int binary = 0b1101;        // Binary (prefix 0b)
int octal = 0123;           // Octal (prefix 0)
int hex = 0x1A3F;           // Hexadecimal (prefix 0x)
```

---

## 🔄 General Rules for Converting Number Systems

| From | To | Rule / Steps |
|---|---|---|
| Decimal → Binary / Octal / Hex | Divide the number by 2 / 8 / 16 repeatedly. Write down the remainders in reverse order. | |
| Binary → Decimal | Multiply each bit by powers of 2 from right to left, then add. | |
| Octal → Decimal | Multiply each digit by powers of 8 from right to left, then add. | |
| Hex → Decimal | Multiply each digit by powers of 16 from right to left, then add (A=10 to F=15). | |
| Binary → Octal | Group bits in 3s (right to left), convert each group to octal digit. | |
| Binary → Hex | Group bits in 4s (right to left), convert each group to hex digit. | |
| Octal / Hex → Binary | Convert each digit into 3-bit (Octal) or 4-bit (Hex) binary. | |

## Conversion Example

### Decimal to Binary

```
Decimal: 13
13 ÷ 2 = 6, remainder = 1
6 ÷ 2 = 3, remainder = 0
3 ÷ 2 = 1, remainder = 1
1 ÷ 2 = 0, remainder = 1

Binary = 1101
```

### Binary to Decimal

```
Binary: 1101
= 1×2³ + 1×2² + 0×2¹ + 1×2⁰
= 8 + 4 + 0 + 1 = 13
```

---

## Java Methods for Conversion

```
Integer.toBinaryString(13);  // "1101"
Integer.toOctalString(13);   // "15"
Integer.toHexString(13);     // "d"

Integer.parseInt("1101", 2); // 13
Integer.parseInt("15", 8);   // 13
Integer.parseInt("d", 16);   // 13
```

---

## ⯑ Summary Table

| System | Base | Prefix | Java Example |
|---|---|---|---|
| Decimal | 10 | None | `int x = 100;` |
| Binary | 2 | `0b` | `int x = 0b1010;` |
| Octal | 8 | `0` | `int x = 012;` |
| Hexadecimal | 16 | `0x` | `int x = 0x1F;` |

---

## 💬 Real-Life Applications

| Application | Number System |
|---|---|
| File permissions | Octal |
| IP/MAC addresses | Hexadecimal |
| Color Codes (#fff) | Hexadecimal |
| Bitmasking/flags | Binary |
| Calculations | Decimal |

---

> 🔑 Mastering number systems builds the foundation for understanding how data is stored, processed, and optimized in programming.

# 📚 Arrays in Java – Complete Notes

In Java, an **array** is a **container object** that holds a fixed number of values of a **single data type**. Arrays are used to store multiple values in a **single variable**, instead of declaring separate variables for each value.

---

## ✓ Key Characteristics of Arrays

| Feature | Description |
|---|---|
| Fixed Size | Size is set when the array is created and cannot change. |
| Zero-based Indexing | First element is at index `0`, last at `length - 1`. |
| Homogeneous Elements | All elements must be of the same data type. |
| Stored in Contiguous Memory | Array elements are stored next to each other in memory. |

---

## Array Declaration and Initialization

### Syntax

```
dataType[] arrayName;            // Declaration
arrayName = new dataType[size]; // Memory allocation
```

### Combined Declaration and Allocation

```
int[] numbers = new int[5]; // Array of size 5
```

### Initialize with Values

```
int[] marks = {90, 85, 88, 76, 95};
```

---

## 📐 Accessing and Modifying Elements

```
System.out.println(marks[0]); // Access first element

marks[2] = 100;               // Modify 3rd element
```

> ⚠ Accessing an index out of bounds will throw `ArrayIndexOutOfBoundsException` .

---

## 🌀 Iterating Over Arrays

### 🔸 Using `for` loop

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

### 🔸 Using `for-each` loop

```
for (int mark : marks) {
    System.out.println(mark);
}
```

---

## 📏 Array Properties

| Property | Description |
|---|---|
| `length` | Returns size of array (no `()` like methods) |
| `index` | Starts from `0` and ends at `length - 1` |

```
System.out.println(marks.length); // 5
```

---

## ◇ Types of Arrays

### 1 One-Dimensional Array

```
int[] arr = new int[5];
```

### 2 Multi-Dimensional Array (Matrix)

```
int[][] matrix = new int[3][4]; // 3 rows, 4 columns

matrix[0][0] = 1;

for (int i = 0; i < 3; i++) {
  for (int j = 0; j < 4; j++) {
    System.out.print(matrix[i][j] + " ");
  }
  System.out.println();
}
```

---

## ◇ Use Cases

- Storing student grades
- Representing matrices
- Data tables in games
- Lookup tables

---

## ❗ Limitations of Arrays

| Limitation | Alternative |
|---|---|
| Fixed size (non-resizable) | Use `ArrayList` |
| Can hold only one data type | Use `Object[]` or Collections |
| No built-in functions (e.g. sort, search) | Use utility classes like `Arrays` |

---

## 🛠 Utility Methods – `java.util.Arrays`

```java
import java.util.Arrays;

int[] arr = {5, 3, 9, 1};
Arrays.sort(arr);                          // Sorts the array
System.out.println(Arrays.toString(arr)); // Print elements

int index = Arrays.binarySearch(arr, 3); // Binary Search
```

---

## 🔄 Common Interview Questions

| Question | Concept Tested |
|---|---|
| Reverse an array | Looping logic |
| Find largest/smallest element | Conditional checking |
| Check for duplicates | Nested loops / HashSet |
| Sort an array | Sorting algorithms / Arrays.sort |
| Rotate array elements | Index manipulation |

---

## ◈ Mini Exercise

```java
// Print sum of array elements
int[] nums = {2, 4, 6, 8};
int sum = 0;

for (int n : nums) {
    sum += n;
}
System.out.println("Sum = " + sum);
```

---

> ◈ Arrays are the building blocks of data structures. Mastering them will give you a strong foundation for learning Lists, Stacks, Queues, and more!

## ◈ How Arrays Are Stored in Memory in Java

In Java, arrays are **objects** stored in the **heap memory**, and they are accessed through **reference variables** stored in the **stack**. Let's understand this in detail.

---

### ◈ Components of Array Storage

When you declare and initialize an array:

```java
int[] arr = new int[5];
```

Java stores the array in two parts:

| Part | Memory Location | Description |
|---|---|---|
| Reference variable ( `arr` ) | Stack | Holds the reference (address) to the array |
| Actual array object | Heap | Contains array metadata and elements |

---

## 🔹 Memory Representation

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

**Heap Memory (Contiguous Allocation for Elements):**

| Index | Address | Value |
|---|---|---|
| 0 | 0x100 | 10 |
| 1 | 0x104 | 20 |
| 2 | 0x108 | 30 |
| 3 | 0x10C | 40 |
| 4 | 0x110 | 50 |

- If `int` takes 4 bytes, each value is stored 4 bytes apart.
- The reference variable `arr` (in the stack) points to the base address `0x100` of the array in the heap.

---

## Array Memory Layout Summary

```
[Stack]
+------------+
| arr: 0x100 |
+------------+

[Heap]
+--------+--------+--------+--------+--------+
|  10    |  20    |  30    |  40    |  50    |
+--------+--------+--------+--------+--------+
  0x100   0x104    0x108    0x10C    0x110
```

---

## Key Points

- Arrays are **objects** in Java, even if they store primitive types.
- The **length** property is stored with the array metadata in the heap.
- Java automatically **bounds-checks** arrays; accessing out-of-bounds throws `ArrayIndexOutOfBoundsException`.
- Arrays in Java are always **contiguously stored**, ensuring efficient access via index.

---

> Tip: Use `System.identityHashCode(arr)` to get the memory reference hash (not exact memory address) of the array.

---

## 🔷 💠 Example: Shared Reference Behavior in Arrays

```java
int[] arr = new int[5];
arr[0] = 33;
arr[1] = 47;
arr[2] = 59;
arr[3] = 67;
arr[4] = 98;

System.out.print(arr[2]); // Output: 59

int[] two = arr;          // 'two' now references the same array
two[2] = 200;

System.out.print(arr[2]); // Output: 200
```

### 🔍 Explanation

- ✅ `arr` and `two` both refer to the **same memory location** in heap.
- ✅ When we assign `two = arr`, we are copying the **reference**, not the array itself.
- ✅ Modifying `two[2] = 200` changes the value at index 2 in the original array too.
- ✅ That's why `arr[2]` also becomes `200`.

---

## Memory Visualization

```
[Stack]
+-----------+      +-----------+
| arr       |-----> |           |
| two       |-----> |  [Heap]   |
+-----------+      |-----------|
                   | [0] = 33  |
                   | [1] = 47  |
                   | [2] = 200 |
                   | [3] = 67  |
                   | [4] = 98  |
                   +-----------+
```

---

## ⚠ Key Takeaway

> In Java, assigning one array to another does **not copy values**, it **copies the reference**, so both variables point to the **same memory block** in heap.

This concept is crucial when working with arrays and object references in Java!

# 🔄 Shared Reference Behavior of Arrays When Passed to a Function in Java

In Java, when you pass an array to a method, you're **passing the reference to the array object**, not a separate copy of the array. This means **modifications inside the method affect the original array**.

---

## 📌 Example: Passing Array to a Method

```java
public class Main {
    public static void modifyArray(int[] arr) {
        arr[1] = 999; // Modify index 1
    }

    public static void main(String[] args) {
        int[] numbers = {10, 20, 30};

        System.out.println("Before: " + numbers[1]); // Output: 20
        modifyArray(numbers);
        System.out.println("After: " + numbers[1]);  // Output: 999
    }
}
```

---

## Explanation

| Step | Action |
|---|---|
| 1️⃣ | `numbers` is declared and initialized in `main`. |
| 2️⃣ | `modifyArray(numbers)` passes the **reference** to the `modifyArray` method. |
| 3️⃣ | Inside the method, `arr[1] = 999` modifies the actual array in **heap memory**. |
| 4️⃣ | After the method call, `numbers[1]` is now `999` in the original array. |

---

## ⯑ Memory Representation

```
[Stack - main()]                 [Stack - modifyArray()]
+--------------------+          +--------------------+
| numbers (ref)      | ------>  | arr (ref)          |
+--------------------+          +--------------------+
         |                                |
         |                                |
         V                                V
              [ Heap Memory - Shared Array ]
              +--------+--------+--------+
              | 10     | 999    | 30     |
              +--------+--------+--------+
               index 0  index 1  index 2
```

## ✓ Key Points

- Arrays in Java are **passed by value**, but that value is a **reference** to the object.
- Changes made inside the function reflect **outside the function**, as both point to the **same array**.
- This is known as **shared reference behavior**.

## ⚠ Gotcha

If you reassign the reference inside the method (e.g., `arr = new int[]{1,2,3};`), it **won't affect** the original array because you're changing what the local reference points to — not the original object.

```java
public static void modifyArray(int[] arr) {
    arr = new int[]{1, 2, 3}; // This does NOT affect the original array
}
```

> 💡 To truly copy an array and avoid affecting the original, use `Arrays.copyOf()` or `array.clone()`.

---

> ❓ Use this knowledge to carefully manage side effects when passing arrays to functions.

# ❓ Object References, Shallow Copy vs Deep Copy in Java

In Java, **objects are not passed or assigned directly**, but via **references**. This leads to behaviors like **shared modification**, especially with **mutable objects** like arrays or custom classes.

---

## 🔗 What is an Object Reference?

An **object reference** is a variable that **stores the memory address** of an object in the heap, not the object itself.

```java
class Student {
    String name;
}

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student();
        s1.name = "Alice";

        Student s2 = s1; // s2 points to the same object
        s2.name = "Bob";

        System.out.println(s1.name); // Output: Bob
    }
}
```

◈ **Explanation**: `s1` and `s2` both point to the **same memory location**, so a change via `s2` reflects in `s1`.

---

## ◈ Shallow Copy

A **shallow copy** copies the reference of an object — **not its actual content**. So the original and copy share the **same inner objects**.

## 📌 Example

```
class Student {
    String name;
}

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student();
        s1.name = "Alice";

        Student s2 = s1; // Shallow copy

        s2.name = "Bob";

        System.out.println(s1.name); // Output: Bob
    }
}
```

## 📌 Characteristics

| Feature | Shallow Copy |
|---|---|
| Memory allocation | Shared |
| Performance | Faster |
| Side Effects | High |
| Suitable for | Immutable or simple objects |

---

## ❓ Deep Copy

A **deep copy** creates a **completely new copy** of the object and all its nested objects — no shared memory.

## 📌 Example Using Constructor

```java
class Student {
    String name;

    Student(String name) {
        this.name = name;
    }

    // Deep copy constructor
    Student(Student s) {
        this.name = new String(s.name);
    }
}

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student("Alice");
        Student s2 = new Student(s1); // Deep copy

        s2.name = "Bob";

        System.out.println(s1.name); // Output: Alice
    }
}
```

## 📌 Characteristics

| Feature | Deep Copy |
|---|---|
| Memory allocation | Independent |
| Performance | Slower |
| Side Effects | None |
| Suitable for | Mutable or complex objects |

---

## Array Deep vs Shallow Example

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

// Shallow copy
int[] shallow = original;

// Deep copy
int[] deep = original.clone();

shallow[0] = 99;
deep[1] = 88;

System.out.println(Arrays.toString(original)); // [99, 2, 3]
System.out.println(Arrays.toString(shallow));  // [99, 2, 3]
System.out.println(Arrays.toString(deep));     // [1, 88, 3]
```

---

## When to Use What?

| Use Case | Type |
|---|---|
| Simple, performance-critical task | Shallow Copy |
| Handling mutable or nested objects | Deep Copy |
| Preventing unintended changes | Deep Copy |
| Working with immutable objects | Shallow Copy |

---

> **Key Takeaway**: Java uses **reference semantics**. Understand when you're copying **data vs reference**, and use **deep copy** when isolation of data is essential.

# 📚 Stacks in Java

A **stack** is a **linear data structure** that follows the **Last In, First Out (LIFO)** principle. This means that the **last element added** to the stack is the **first one to be removed**.

---

## 🔹 Why Use a Stack?

✅ Supports **undo/redo** operations (e.g., text editors)
✅ Manages function calls in **recursion**
✅ Used in **expression evaluation** (e.g., parsing expressions)
✅ Backtracking (e.g., **maze solving, browser history**)

---

## ✅ Stack Operations

| Operation | Description |
|---|---|
| `push(x)` | Adds element `x` to the top of the stack |
| `pop()` | Removes and returns the top element |
| `peek()` | Returns the top element **without removing** it |
| `isEmpty()` | Returns `true` if stack is empty |
| `size()` | Returns the number of elements in the stack |

---

## 🛠 Implementing Stack in Java

Java provides **two ways** to implement a stack:

## 1️⃣ Using `Stack` Class (Java Collection Framework)

```java
import java.util.Stack;

public class Main {
    public static void main(String[] args) {
        Stack<Integer> stack = new Stack<>();

        stack.push(10);
        stack.push(20);
        stack.push(30);

        System.out.println(stack.peek()); // 30
        System.out.println(stack.pop());  // 30
        System.out.println(stack.isEmpty()); // false
    }
}
```

✅ **Pros**: Built-in, optimized
❌ **Cons**: Synchronized (slower for multi-threading)

---

## 2️⃣ Implementing Stack Using an Array (Manual Approach)

```java
class StackArray {
    private int[] arr;
    private int top;
    private int capacity;

    public StackArray(int size) {
        arr = new int[size];
        capacity = size;
        top = -1;
    }

    public void push(int x) {
        if (top == capacity - 1) {
            System.out.println("Stack Overflow");
            return;
        }
        arr[++top] = x;
    }

    public int pop() {
        if (top == -1) {
            System.out.println("Stack Underflow");
            return -1;
        }
        return arr[top--];
    }

    public int peek() {
        return (top == -1) ? -1 : arr[top];
    }

    public boolean isEmpty() {
        return top == -1;
    }
}

public class Main {
    public static void main(String[] args) {
        StackArray stack = new StackArray(5);
        stack.push(10);
        stack.push(20);

        System.out.println(stack.peek()); // 20
        System.out.println(stack.pop());  // 20
        System.out.println(stack.isEmpty()); // false
    }
}
```

✓ **Pros**: Faster, thread-safe

✕ **Cons**: Fixed size, needs resizing

---

## 🔄 Stack Using Linked List (Dynamic)

```java
class Node {
    int data;
    Node next;
}

class StackLinkedList {
    private Node top;

    public StackLinkedList() {
        this.top = null;
    }

    public void push(int x) {
        Node newNode = new Node();
        newNode.data = x;
        newNode.next = top;
        top = newNode;
    }

    public int pop() {
        if (top == null) {
            System.out.println("Stack Underflow");
            return -1;
        }
        int value = top.data;
        top = top.next;
        return value;
    }

    public int peek() {
        return (top == null) ? -1 : top.data;
    }

    public boolean isEmpty() {
        return top == null;
    }
}

public class Main {
    public static void main(String[] args) {
        StackLinkedList stack = new StackLinkedList();

        stack.push(10);
        stack.push(20);

        System.out.println(stack.peek()); // 20
        System.out.println(stack.pop());  // 20
        System.out.println(stack.isEmpty()); // false
    }
}
```

✅ **Pros**: Dynamic size, no overflow
❌ **Cons**: More memory usage (extra pointers)

---

## 🚀 Stack Applications

| Application | Use Case |
|---|---|
| Function Calls | Call Stack in recursion |
| Undo/Redo | Text editors |
| Parentheses Matching | Syntax validation |
| Postfix & Prefix Evaluation | Expression parsing |
| DFS (Depth First Search) | Graph traversal |

---

## 💡 Key Takeaways

1️⃣ **Stack follows LIFO** (Last In, First Out).
2️⃣ Java provides `Stack<T>` **class**, but manual implementations offer more flexibility.
3️⃣ **Array-based stacks** are faster but have a fixed size.

4️⃣ **Linked list stacks** are dynamic but use extra memory.
5️⃣ Stacks are useful in **recursion, expression evaluation, and backtracking**.

---

> 🔥 **Tip:** Always use `try { pop(); } catch (EmptyStackException e) {}` when working with Java's `Stack` class to handle errors safely.

## Infix, Postfix, and Prefix Notations in Java

In mathematical expressions, operators and operands can be arranged in different ways, leading to three main notations: **Infix**, **Postfix**, and **Prefix**. Understanding these notations is crucial for expression evaluation, parsing, and conversion, especially in Java, where stacks are often used for such operations.

---

## 1. Infix Notation

### Definition:

- In **infix notation**, the operator is placed **between** operands.
- This is the standard way humans write mathematical expressions.
- Example:

  ```
  (3 + 5) * 2
  ```

### Evaluation in Java:

- Infix expressions follow **operator precedence** and **associativity** rules.
- Java evaluates infix expressions directly using arithmetic operators and parentheses.
- **Example in Java:**

  ```java
  int result = (3 + 5) * 2; // result = 16
  System.out.println(result);
  ```

- **Limitations:**

    - Requires parsing and precedence handling when evaluated from a string.
    - Cannot be easily processed by computers without additional logic.

---

## 2. Postfix Notation (Reverse Polish Notation - RPN)

### Definition:

- In **postfix notation**, the operator is placed **after** the operands.
- No need for parentheses since the order of operations is unambiguous.
- Example:

  ```
  3 5 + 2 *
  ```

  This is equivalent to `(3 + 5) * 2`.

### Evaluation in Java (Using Stack):

- Postfix expressions can be evaluated using a **stack**:
  i. Scan the expression from left to right.
  ii. Push operands onto the stack.
  iii. When an operator is encountered, pop two operands, apply the operator, and push the result back.
- **Example in Java:**

```java
import java.util.*;

public class PostfixEvaluation {
    public static int evaluatePostfix(String exp) {
        Stack<Integer> stack = new Stack<>();

        for (char ch : exp.toCharArray()) {
            if (Character.isDigit(ch)) {
                stack.push(ch - '0'); // Convert char to int
            } else {
                int v2 = stack.pop();
                int v1 = stack.pop();
                switch (ch) {
                    case '+': stack.push(v1 + v2); break;
                    case '-': stack.push(v1 - v2); break;
                    case '*': stack.push(v1 * v2); break;
                    case '/': stack.push(v1 / v2); break;
                }
            }
        }
        return stack.pop();
    }

    public static void main(String[] args) {
        String postfix = "35+2*"; // (3+5)*2
        System.out.println(evaluatePostfix(postfix)); // Output: 16
    }
}
```

- **Advantages:**
    - No need for precedence handling.
    - Easy to evaluate using stacks.

## 3. Prefix Notation (Polish Notation)

### Definition:

- In **prefix notation**, the operator is placed **before** the operands.
- Like postfix, no parentheses are required.
- Example:

  ```
  * + 3 5 2
  ```

  This is equivalent to `(3 + 5) * 2`.

### Evaluation in Java (Using Stack):

- Prefix expressions can be evaluated similarly to postfix:
  1. Scan the expression **right to left**.
  2. Push operands onto the stack.
  3. When an operator is encountered, pop two operands, apply the operator, and push the result back.
- **Example in Java:**

```java
import java.util.*;

public class PrefixEvaluation {
    public static int evaluatePrefix(String exp) {
        Stack<Integer> stack = new Stack<>();

        for (int i = exp.length() - 1; i >= 0; i--) {
            char ch = exp.charAt(i);
            if (Character.isDigit(ch)) {
                stack.push(ch - '0');
            } else {
                int v1 = stack.pop();
                int v2 = stack.pop();
                switch (ch) {
                    case '+': stack.push(v1 + v2); break;
                    case '-': stack.push(v1 - v2); break;
                    case '*': stack.push(v1 * v2); break;
                    case '/': stack.push(v1 / v2); break;
                }
            }
        }
        return stack.pop();
    }

    public static void main(String[] args) {
        String prefix = "*+352"; // (3+5)*2
        System.out.println(evaluatePrefix(prefix)); // Output: 16
    }
}
```

- **Advantages:**
    - No need for precedence handling.
    - Useful in **compilers and expression evaluation**.

## Conversion Between Notations

| Conversion | Algorithm Used |
|---|---|
| Infix → Postfix | **Shunting-yard algorithm** (Uses stack) |
| Infix → Prefix | Reverse infix → Convert to postfix → Reverse result |
| Postfix → Infix | Process using stack, insert operators at correct places |
| Prefix → Infix | Process right-to-left using stack |

- **Example: Converting Infix to Postfix in Java**

```java
import java.util.*;

public class InfixToPostfix {
    public static int precedence(char ch) {
        if (ch == '+' || ch == '-') return 1;
        if (ch == '*' || ch == '/') return 2;
        return -1;
    }

    public static String infixToPostfix(String exp) {
        Stack<Character> stack = new Stack<>();
        StringBuilder result = new StringBuilder();

        for (char ch : exp.toCharArray()) {
            if (Character.isDigit(ch)) {
                result.append(ch);
            } else if (ch == '(') {
                stack.push(ch);
            } else if (ch == ')') {
                while (!stack.isEmpty() && stack.peek() != '(')
                    result.append(stack.pop());
                stack.pop(); // Remove '('
            } else {
                while (!stack.isEmpty() && precedence(ch) <= precedence(stack.peek()))
                    result.append(stack.pop());
                stack.push(ch);
            }
        }

        while (!stack.isEmpty())
            result.append(stack.pop());

        return result.toString();
    }

    public static void main(String[] args) {
        String infix = "3+5*2";
        System.out.println(infixToPostfix(infix)); // Output: "352*+"
    }
}
```

---

## Comparison Table

| Notation | Expression Example | Evaluation Complexity | Ease of Use | Usage |
|---|---|---|---|---|
| **Infix** | `(3 + 5) * 2` | Medium (Handles precedence) | Easy for humans | Used in daily math |
| **Postfix** | `3 5 + 2 *` | Fast (Stack-based) | Hard to write manually | Used in compilers, calculators |
| **Prefix** | `* + 3 5 2` | Fast (Stack-based) | Hard to write manually | Used in AI, parsing |

---

## Conclusion

- **Infix notation** is human-friendly but requires precedence handling.
- **Postfix notation** is easier to evaluate using stacks, making it suitable for **compilers** and **calculators**.
- **Prefix notation** is useful in **recursive computations** and **expression trees**.

Java provides powerful **stack-based solutions** for evaluating and converting expressions between these notations, making it a core concept in **data structures, algorithms, and compilers**. 🚀

## Fundamental Conversions Between Strings, Characters, and Numbers in Java

Understanding how to efficiently convert between **strings, characters, and numeric values** is crucial for handling data operations in Java. This note establishes a **common base** for these conversions, ensuring a structured understanding for all future operations.

---

## 1. Converting a Numeric Character in a String to an Integer

💡 **Subtract `'0'` from a character digit to get its integer value.**

### Why?

- Characters are stored as **ASCII/Unicode values**.
- `'0'` (zero character) has an ASCII value of **48**.
- Any digit character `'0'` to `'9'` has a corresponding ASCII value from **48 to 57**.
- Subtracting `'0'` extracts the actual numeric value.

### Example:

```
char digit = '7';
int num = digit - '0'; // '7' (ASCII 55) - '0' (ASCII 48) = 7
System.out.println(num); // Output: 7
```

### Usage:

✔ Extracting integer values from numeric characters in strings.
✔ Efficient for handling **single-digit** character conversions.

---

## 2. Converting a String Representation of a Number to an Integer

💡 **Use `Integer.parseInt(str)` or `Integer.valueOf(str)`.**

### Example:

```java
String numStr = "123";
int number = Integer.parseInt(numStr);  // Converts "123" → 123
System.out.println(number); // Output: 123
```

✔ Works for **multi-digit** numbers.
✔ **`Integer.valueOf(str)`** returns an `Integer` object instead of `int`.

---

## 3. Converting a Single Digit Integer to a Character

💡 **Add `'0'` to an integer to get its character equivalent.**

### Why?

- Just as subtraction (`'7' - '0'`) extracts a numeric value,
- Adding `'0'` shifts an integer into its ASCII character range.

### Example:

```java
int num = 5;
char digitChar = (char) (num + '0'); // 5 + ASCII 48 = '5'
System.out.println(digitChar); // Output: '5'
```

✔ Efficient for **single-digit numbers**.

---

## 4. Converting Any Number to a String

💡 **Concatenate with `""` or use `String.valueOf(num)`.**

## Examples:

```java
int num = 123;
String str1 = num + "";  // Implicit conversion using concatenation
String str2 = String.valueOf(num); // Explicit conversion

System.out.println(str1); // Output: "123"
System.out.println(str2); // Output: "123"
```

✔ Works for **all numeric types (`int`, `double`, `float`, etc.)**
✔ **Preferred:** `String.valueOf(num)`, as it avoids unnecessary concatenation.

---

## 5. Converting a Character to a String

💡 **Concatenate with `""` or use `Character.toString(ch)`.**

## Example:

```java
char ch = 'A';
String str1 = ch + "";  // Implicit conversion
String str2 = Character.toString(ch); // Explicit conversion

System.out.println(str1); // Output: "A"
System.out.println(str2); // Output: "A"
```

✔ Useful for handling **single characters in string operations**.

---

## 6. Converting a String to a Character Array

💡 **Use `toCharArray()` to break a string into individual characters.**

## Example:

```java
String word = "Hello";
char[] charArray = word.toCharArray();

System.out.println(Arrays.toString(charArray)); // Output: [H, e, l, l, o]
```

✔ Useful for iterating over characters in a string.

---

## Conclusion: Universal Conversion Rules

| Conversion | Approach |
|---|---|
| String → Integer | `Integer.parseInt(str)` |
| String → Character Array | `str.toCharArray()` |
| Single Character → Integer | `ch - '0'` |
| Integer → Single Character | `(char) (num + '0')` |
| Number → String | `num + ""` OR `String.valueOf(num)` |
| Character → String | `ch + ""` OR `Character.toString(ch)` |

## Key Takeaways:

1. **Subtract** `'0'` to convert a numeric character to an integer.
2. **Add** `'0'` to convert an integer to a numeric character.
3. **Concatenation (** `"" +` **)** is a quick way to convert any number or character to a string.
4. **Use** `String.valueOf()` for efficient numeric-to-string conversion.
5. **Use** `toCharArray()` for character-level string processing.

This foundational understanding will help in **string manipulations, numerical operations, and type conversions** across Java programs. 🚀

# 🔷 Object-Oriented Programming (OOP) in Java

## ✅ What is OOP?

Object-Oriented Programming (OOP) is a programming paradigm that organizes code using **objects** and **classes**. It provides modularity, reusability, and scalability.

---

## ✅ Key OOP Principles

| Principle | Description |
|---|---|
| **Encapsulation** | Wrapping data (variables) and code (methods) into a single unit (class) and restricting direct access. |
| **Abstraction** | Hiding implementation details and exposing only necessary features through interfaces or abstract classes. |
| **Inheritance** | Enabling one class (child) to inherit properties and behavior from another class (parent). |
| **Polymorphism** | Allowing methods to have multiple forms (method overloading and method overriding). |

---

## ✅ OOP Concepts with Examples

### 🔷 1. Class & Object

- A **class** is a blueprint for creating objects.
- An **object** is an instance of a class.

## Example

```java
class Car {
    String brand;
    int speed;

    void display() {
        System.out.println("Brand: " + brand + ", Speed: " + speed);
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.brand = "Toyota";
        myCar.speed = 120;
        myCar.display();
    }
}
```

---

## 🔹 2. Encapsulation

- **Protects data** using private variables and provides access via public methods.

### Example

```
class Person {
    private String name;

    public void setName(String newName) {
        name = newName;
    }

    public String getName() {
        return name;
    }
}

public class Main {
    public static void main(String[] args) {
        Person p = new Person();
        p.setName("John");
        System.out.println(p.getName());
    }
}
```

---

## 🔷 3. Abstraction

- **Hides** unnecessary details using `abstract` classes or `interface`.

### Example (Abstract Class)

```java
abstract class Animal {
    abstract void makeSound();
}

class Dog extends Animal {
    void makeSound() {
        System.out.println("Bark!");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.makeSound();
    }
}
```

---

## 🔹 4. Inheritance

- **Allows a child class** to inherit properties and behaviors from a parent class.

### Example

```java
class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("Dog barks.");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.eat();  // Inherited method
        d.bark();
    }
}
```

---

## 🔹 5. Polymorphism

- **Method Overloading** (same method name, different parameters)
- **Method Overriding** (subclass changes the behavior of a parent class method)

## Example (Method Overloading)

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

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

public class Main {
    public static void main(String[] args) {
        MathOperations obj = new MathOperations();
        System.out.println(obj.add(5, 10));      // Calls int version
        System.out.println(obj.add(5.5, 10.5));  // Calls double version
    }
}
```

### Example (Method Overriding)

```java
class Animal {
    void makeSound() {
        System.out.println("Some sound...");
    }
}

class Dog extends Animal {
    void makeSound() {
        System.out.println("Bark!");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal a = new Dog();
        a.makeSound();  // Calls Dog's overridden method
    }
}
```

---

## ✅ Key Advantages of OOP

✔ **Modularity** – Code is divided into smaller, reusable units.
✔ **Reusability** – Inheritance allows reusing code.
✔ **Security** – Encapsulation hides sensitive data.
✔ **Flexibility** – Polymorphism enables dynamic method execution.
✔ **Maintainability** – Code is structured and easy to modify.

---

> 💡 **Tip:** Mastering OOP concepts is essential for writing scalable and efficient Java applications.

# Queue in Java (Detailed Explanation)

## 1. What is a Queue?

A **Queue** is a **linear data structure** that follows the **First-In-First-Out (FIFO)** principle. This means that elements are added at the **rear** (end) and removed from the **front** (beginning).

📌 **Key Characteristics of a Queue:**
✔ **FIFO (First In, First Out)** – The first element added is the first to be removed.
✔ **Insertion (Enqueue)** – Performed at the **rear** of the queue.
✔ **Deletion (Dequeue)** – Performed from the **front** of the queue.
✔ **Used in real-world scenarios** such as **task scheduling, buffering, and breadth-first search (BFS)** in graphs.

---

## 2. Queue Implementation in Java

### Java provides the `Queue` interface in the `java.util` package.

It is implemented by different classes such as:
✔ `LinkedList` (Doubly Linked List-based Queue)
✔ `PriorityQueue` (Heap-based Priority Queue)
✔ `ArrayDeque` (Resizable Array-based Queue)

---

## 3. Queue Interface in Java

📌 The `Queue<E>` interface extends the `Collection<E>` interface and provides methods for queue operations.

## Queue Methods in Java

| Method | Description |
|---|---|
| `add(E e)` | Inserts an element into the queue (throws an exception if full). |
| `offer(E e)` | Inserts an element into the queue (returns `false` if full). |
| `remove()` | Removes and returns the head of the queue (throws an exception if empty). |
| `poll()` | Removes and returns the head of the queue (returns `null` if empty). |
| `element()` | Retrieves but does not remove the head of the queue (throws an exception if empty). |
| `peek()` | Retrieves but does not remove the head of the queue (returns `null` if empty). |

---

## 4. Implementing a Queue using `LinkedList`

Java's `LinkedList` class implements the `Queue` interface.

```java
import java.util.LinkedList;
import java.util.Queue;

public class QueueExample {
    public static void main(String[] args) {
        // Create a Queue
        Queue<Integer> queue = new LinkedList<>();

        // Enqueue (Add elements)
        queue.add(10);
        queue.add(20);
        queue.offer(30);  // `offer()` is safer

        // Display queue
        System.out.println("Queue: " + queue); // [10, 20, 30]

        // Dequeue (Remove elements)
        System.out.println("Removed: " + queue.poll()); // 10
        System.out.println("Queue after removal: " + queue); // [20, 30]

        // Peek (Check front element)
        System.out.println("Front element: " + queue.peek()); // 20
    }
}
```

✅ **Output:**

```
Queue: [10, 20, 30]
Removed: 10
Queue after removal: [20, 30]
Front element: 20
```

---

## 5. Implementing a Queue using `ArrayDeque`

- `ArrayDeque` **is faster** than `LinkedList` and is preferred for queues.

```java
import java.util.ArrayDeque;
import java.util.Queue;

public class ArrayDequeExample {
    public static void main(String[] args) {
        Queue<String> queue = new ArrayDeque<>();

        queue.offer("Apple");
        queue.offer("Banana");
        queue.offer("Cherry");

        System.out.println("Queue: " + queue); // [Apple, Banana, Cherry]
        System.out.println("Peek: " + queue.peek()); // Apple
        System.out.println("Removed: " + queue.poll()); // Apple
        System.out.println("Queue after removal: " + queue); // [Banana, Cherry]
    }
}
```

---

## 6. Implementing a `PriorityQueue` in Java

A **Priority Queue** processes elements based on **priority instead of FIFO**.

```java
import java.util.PriorityQueue;
import java.util.Queue;

public class PriorityQueueExample {
    public static void main(String[] args) {
        Queue<Integer> priorityQueue = new PriorityQueue<>();

        priorityQueue.offer(50);
        priorityQueue.offer(20);
        priorityQueue.offer(40);
        priorityQueue.offer(10);

        System.out.println("Priority Queue: " + priorityQueue);
        // Order is based on natural sorting, but internal structure is not a simple list.

        System.out.println("Polled element: " + priorityQueue.poll()); // Removes 10 (smallest)
        System.out.println("Queue after poll: " + priorityQueue); // [20, 50, 40]
    }
}
```

✅ **Output:**

```
Priority Queue: [10, 20, 40, 50]
Polled element: 10
Queue after poll: [20, 50, 40]
```

✔ `PriorityQueue` **orders elements in ascending order by default**.
✔ **For custom ordering**, use a **Comparator**.

---

## 7. Implementing a Custom Queue using an Array

```java
class CustomQueue {
    private int[] arr;
    private int front, rear, size, capacity;

    public CustomQueue(int capacity) {
        this.capacity = capacity;
        arr = new int[capacity];
        front = 0;
        rear = -1;
        size = 0;
    }

    public void enqueue(int value) {
        if (size == capacity) {
            System.out.println("Queue is full");
            return;
        }
        rear = (rear + 1) % capacity;
        arr[rear] = value;
        size++;
    }

    public int dequeue() {
        if (size == 0) {
            System.out.println("Queue is empty");
            return -1;
        }
        int removedValue = arr[front];
        front = (front + 1) % capacity;
        size--;
        return removedValue;
    }

    public int peek() {
        return (size == 0) ? -1 : arr[front];
    }

    public boolean isEmpty() {
        return size == 0;
    }
}

public class QueueArrayExample {
    public static void main(String[] args) {
        CustomQueue queue = new CustomQueue(5);
        queue.enqueue(10);
        queue.enqueue(20);
        queue.enqueue(30);

        System.out.println("Front element: " + queue.peek()); // 10
        System.out.println("Dequeued: " + queue.dequeue()); // 10
        System.out.println("Front element after dequeue: " + queue.peek()); // 20
    }
}
```

---

## 8. Applications of Queue in Java

✔ **Task Scheduling** – OS Process Scheduling (Round-Robin Scheduling).
✔ **Print Spoolers** – Managing print jobs in a queue.
✔ **Breadth-First Search (BFS)** – Graph traversal.
✔ **Call Center Management** – Handling customer service calls in order.
✔ **Message Queues** – RabbitMQ, Kafka, etc.

---

## 9. Summary

- **Queue is a FIFO-based data structure.**
- **Implemented using `LinkedList`, `ArrayDeque`, and `PriorityQueue`.**
- **Use `PriorityQueue` for ordered processing.**
- **Use `ArrayDeque` for faster queue operations.**
- **Custom queue can be implemented using an array.**

🚀 **Queues are widely used in real-world applications for efficient processing and**

**scheduling tasks!**

Declaring a queue like this:

```java
Queue<Integer> que = new ArrayDeque<>();
```

**is done for flexibility and adherence to best practices in Java programming.** Let's break it down step by step.

---

## 1. Why Use `Queue<Integer>` Instead of `ArrayDeque<Integer>`?

Declaring the queue as:

```java
Queue<Integer> que = new ArrayDeque<>();
```

**instead of:**

```java
ArrayDeque<Integer> que = new ArrayDeque<>();
```

**is beneficial because:**

### a) Follows the "Program to an Interface" Principle

- `Queue` is an **interface** in Java that represents a queue behavior.
- `ArrayDeque` is a **concrete implementation** of the `Queue` interface.
- By using `Queue<Integer>`, you make your code **more flexible** and **loosely coupled** to the specific implementation.
- If you later decide to use a different queue implementation ( `LinkedList`, `PriorityQueue` ), you can change only the right-hand side.

✅ **Example of flexibility:**

```
Queue<Integer> que = new LinkedList<>();  // Now using LinkedList instead of ArrayDeque
```

This **would not be possible** if you declared:

```
ArrayDeque<Integer> que = new ArrayDeque<>();
```

because `LinkedList` cannot be assigned to `ArrayDeque`.

---

## b) Encourages Code Reusability and Extensibility

- If your code relies on the `Queue` interface, it can work with **any queue implementation** without modifications.
- This is useful in real-world applications where you might need to switch from `ArrayDeque` to `PriorityQueue` or a custom queue.

✅ **Example: Easy switching of queue implementations**

```
public void processQueue(Queue<Integer> queue) {
    queue.offer(10);
    queue.offer(20);
    System.out.println(queue.poll()); // Process and remove first element
}
```

Now, you can pass **any queue type** (`ArrayDeque`, `LinkedList`, `PriorityQueue`) to this method:

```
processQueue(new ArrayDeque<>());
processQueue(new LinkedList<>());
processQueue(new PriorityQueue<>());
```

🚀 **This makes the code modular and reusable!**

---

## 2. Why `new ArrayDeque<>()` Instead of `new Queue<>()`?

- `Queue` is **an interface**, so you **cannot instantiate it directly**.
- `ArrayDeque` is **a concrete class** that implements `Queue`, so it can be instantiated.

✅ **Correct:**

```
Queue<Integer> que = new ArrayDeque<>();
```

❌ **Incorrect (won't compile):**

```
Queue<Integer> que = new Queue<>(); // ❌ Error: Queue is abstract and cannot be instantiated
```

---

## 3. Why Not Use `LinkedList` for Queue?

You could also declare:

```
Queue<Integer> que = new LinkedList<>();
```

But `LinkedList` is **slower** than `ArrayDeque` because:

- `ArrayDeque` does not allow nulls, avoiding `NullPointerException` risks.
- `ArrayDeque` performs **faster** than `LinkedList` due to **better memory locality** and **less overhead**.
- `LinkedList` has **extra overhead** for maintaining node pointers.

🚀 **For a pure queue, `ArrayDeque` is the best option.**

---

## Conclusion

✅ `Queue<Integer> que = new ArrayDeque<>();`

- **Follows best practices** ("program to an interface").
- **Keeps code flexible** (can switch queue types easily).
- **Uses `ArrayDeque` because it is more efficient** than `LinkedList`.
- **Cannot instantiate `Queue` directly** since it's an interface.

🚀 **Always prefer this declaration for best flexibility and performance!**

# LinkedList in Java (Detailed Explanation)

## 1. What is a LinkedList?

A **LinkedList** is a **linear data structure** in which elements (nodes) are stored **non-contiguously** in memory. Each node contains:

1. **Data** (the actual value)
2. **Pointer (Reference)** to the next node in the list

Java provides `LinkedList` as a **class** in the `java.util` package, which implements both the **List** and **Deque** interfaces.

📌 **Key Features of LinkedList in Java:**
✔ **Doubly Linked List** implementation (each node has pointers to both next and previous nodes).
✔ **Efficient insertions and deletions** compared to `ArrayList`.
✔ **Supports FIFO (Queue) and LIFO (Stack) operations** using `Deque`.

---

## 2. LinkedList Class in Java

The `LinkedList` class implements:

- `List` **Interface** → Supports indexing and allows duplicate elements.
- `Deque` **Interface** → Allows efficient insertion/removal from both ends (acts as a queue or stack).

### Class Declaration

```
public class LinkedList<E> extends AbstractSequentialList<E>
        implements List<E>, Deque<E>, Cloneable, Serializable
```

---

## 3. Creating a LinkedList in Java

### Syntax

```
LinkedList<Type> list = new LinkedList<>();
```

## Example

```java
import java.util.LinkedList;

public class LinkedListExample {
    public static void main(String[] args) {
        // Creating a LinkedList of Strings
        LinkedList<String> list = new LinkedList<>();

        // Adding elements
        list.add("Apple");
        list.add("Banana");
        list.add("Cherry");

        // Printing the LinkedList
        System.out.println("LinkedList: " + list);
    }
}
```

✅ **Output:**

```
LinkedList: [Apple, Banana, Cherry]
```

---

## 4. Important LinkedList Methods

| Method | Description |
|---|---|
| `add(E e)` | Adds element at the end of the list |
| `addFirst(E e)` | Adds element at the beginning |
| `addLast(E e)` | Adds element at the end (same as `add()`) |
| `remove()` | Removes the first element |
| `removeFirst()` | Removes the first element |

| Method | Description |
|---|---|
| `removeLast()` | Removes the last element |
| `getFirst()` | Retrieves the first element |
| `getLast()` | Retrieves the last element |
| `peekFirst()` | Retrieves first element (returns `null` if empty) |
| `peekLast()` | Retrieves last element (returns `null` if empty) |
| `pollFirst()` | Retrieves and removes the first element |
| `pollLast()` | Retrieves and removes the last element |

---

## 5. Adding Elements in a LinkedList

### Using `add()`

```
LinkedList<Integer> numbers = new LinkedList<>();
numbers.add(10);
numbers.add(20);
numbers.add(30);
System.out.println(numbers); // [10, 20, 30]
```

### Using `addFirst()` and `addLast()`

```
numbers.addFirst(5);
numbers.addLast(40);
System.out.println(numbers); // [5, 10, 20, 30, 40]
```

---

## 6. Removing Elements in a LinkedList

### Using `remove()`, `removeFirst()`, `removeLast()`

```java
numbers.removeFirst(); // Removes 5
numbers.removeLast();  // Removes 40
System.out.println(numbers); // [10, 20, 30]
```

### Using `poll()`, `pollFirst()`, `pollLast()` (Safer)

```java
System.out.println(numbers.poll()); // 10 (removes and returns first element)
System.out.println(numbers.pollLast()); // 30 (removes last)
```

📌 `poll()` is safer than `remove()` because it returns `null` if the list is empty.

---

## 7. Accessing Elements

### Using `getFirst()` and `getLast()`

```java
System.out.println("First: " + numbers.getFirst()); // 20
System.out.println("Last: " + numbers.getLast());   // 20
```

### Using `peek()` (Safer)

```java
System.out.println("Peek First: " + numbers.peekFirst()); // 20
System.out.println("Peek Last: " + numbers.peekLast());   // 20
```

📌 `peek()` returns `null` instead of throwing an exception if the list is empty.

---

## 8. Iterating Over a LinkedList

### Using a `for` loop

```
for (int i = 0; i < numbers.size(); i++) {
    System.out.print(numbers.get(i) + " ");
}
```

### Using an Enhanced `for` loop

```
for (Integer num : numbers) {
    System.out.print(num + " ");
}
```

### Using an Iterator

```
Iterator<Integer> iterator = numbers.iterator();
while (iterator.hasNext()) {
    System.out.print(iterator.next() + " ");
}
```

### Using `forEach()`

```
numbers.forEach(num -> System.out.print(num + " "));
```

---

## 9. LinkedList as a Queue

The `LinkedList` class implements the `Queue` interface, making it usable as a **FIFO (First-In-First-Out) queue**.

```java
Queue<String> queue = new LinkedList<>();

queue.offer("A");
queue.offer("B");
queue.offer("C");

System.out.println(queue.poll()); // A (removes first element)
System.out.println(queue.peek()); // B (checks front)
```

---

## 10. LinkedList as a Stack

The `LinkedList` class implements the `Deque` interface, making it usable as a **LIFO (Last-In-First-Out) stack**.

```java
Deque<Integer> stack = new LinkedList<>();

stack.push(10);
stack.push(20);
stack.push(30);

System.out.println(stack.pop()); // 30 (removes last element)
System.out.println(stack.peek()); // 20 (checks top)
```

---

## 11. Difference Between `ArrayList` and `LinkedList`

| Feature | ArrayList | LinkedList |
|---|---|---|
| **Implementation** | Dynamic array | Doubly linked list |
| **Access Time** | Fast (O(1)) | Slow (O(n)) |
| **Insertion/Deletion** | Slow (O(n)) | Fast (O(1) at head/tail) |

| Feature | ArrayList | LinkedList |
|---|---|---|
| **Memory Usage** | Less (no extra pointers) | More (extra space for pointers) |
| **Best For** | Read-heavy applications | Insert/Delete-heavy applications |

---

## 12. When to Use LinkedList?

✅ **Use LinkedList when:**

- You **frequently add/remove** elements at the **beginning/middle** of the list.
- You need a **FIFO queue** or **LIFO stack**.

❌ **Avoid LinkedList when:**

- You need **fast access by index** (`ArrayList` is better).
- Memory usage is a concern (LinkedList uses extra space for pointers).

---

## 13. Summary

- **LinkedList is a doubly linked list** that allows fast insertions and deletions.
- Implements **List, Queue, and Deque**, so it can be used as a **list, queue, or stack**.
- **Slower than ArrayList for random access** but **faster for insertions/deletions**.
- **Memory overhead is higher** due to extra node pointers.
- **Use it when frequent modifications are required**, otherwise prefer `ArrayList`.

🚀 **Mastering `LinkedList` helps in implementing stacks, queues, and graph traversal algorithms like BFS!**

# Sorting Algorithms

Sorting algorithms are used to rearrange a given array or list elements in a specified order, typically ascending or descending. Below are some of the most common sorting algorithms, their working principles, and their time complexities.

## 1. Bubble Sort

### Description:

Bubble Sort repeatedly compares adjacent elements and swaps them if they are in the wrong order. This process is repeated until the array is sorted.

### Algorithm:

1. Iterate through the array.
2. Compare adjacent elements and swap if necessary.
3. Repeat the process for each element until no swaps are needed.

### Time Complexity:

- **Best Case (Already Sorted):** $O(n)$
- **Average Case:** $O(n^2)$
- **Worst Case (Reversed Order):** $O(n^2)$

---

## 2. Selection Sort

### Description:

Selection Sort divides the array into sorted and unsorted parts. It repeatedly finds the minimum element from the unsorted section and swaps it with the leftmost unsorted element.

### Algorithm:

1. Find the minimum element in the unsorted portion.
2. Swap it with the first unsorted element.
3. Move the boundary of the sorted section one step right.

4. Repeat until the entire array is sorted.

### Time Complexity:

- **Best Case:** $O(n^2)$
- **Average Case:** $O(n^2)$
- **Worst Case:** $O(n^2)$

---

## 3. Insertion Sort

### Description:

Insertion Sort builds the sorted array one element at a time by picking the next element and placing it in the correct position.

### Algorithm:

1. Start with the second element and compare it with the first.
2. Insert it into the correct position.
3. Repeat for all remaining elements.

### Time Complexity:

- **Best Case (Already Sorted):** $O(n)$
- **Average Case:** $O(n^2)$
- **Worst Case (Reversed Order):** $O(n^2)$

---

## 4. Merge Sort

### Description:

Merge Sort follows the divide and conquer approach. It recursively splits the array into halves, sorts them, and then merges the sorted halves.

### Algorithm:

1. Divide the array into two halves.
2. Recursively sort both halves.
3. Merge the sorted halves.

### Time Complexity:

- **Best Case:** O(n log n)
- **Average Case:** O(n log n)
- **Worst Case:** O(n log n)

---

## 5. Quick Sort

### Description:

Quick Sort selects a pivot element, partitions the array around the pivot, and recursively sorts the partitions.

### Algorithm:

1. Pick a pivot element.
2. Partition the array so that elements smaller than the pivot go left and larger ones go right.
3. Recursively apply the process to left and right subarrays.

### Time Complexity:

- **Best Case:** O(n log n)
- **Average Case:** O(n log n)
- **Worst Case (When pivot is smallest or largest element):** O($n^2$)

---

## 6. Heap Sort

### Description:

Heap Sort builds a binary heap from the array and extracts the largest element iteratively to get a sorted array.

### Algorithm:

1. Build a max heap.
2. Swap the root with the last element and reduce heap size.
3. Heapify the root.
4. Repeat until sorted.

### Time Complexity:

- **Best Case:** O(n log n)
- **Average Case:** O(n log n)
- **Worst Case:** O(n log n)

---

## 7. Counting Sort (For Small Integer Ranges)

### Description:

Counting Sort works by counting the frequency of each element and placing them in the correct order.

### Algorithm:

1. Count occurrences of each element.
2. Compute the prefix sum of counts.
3. Place elements in their correct position using the count array.

## Time Complexity:

- **Best Case:** O(n + k)
- **Average Case:** O(n + k)
- **Worst Case:** O(n + k)
  Where `k` is the range of input values.

---

## 8. Radix Sort (For Numbers and Strings)

### Description:

Radix Sort processes elements digit by digit using a stable sorting algorithm (e.g., Counting Sort).

### Algorithm:

1. Start with the least significant digit.
2. Sort numbers based on the current digit.
3. Move to the next digit and repeat until all digits are processed.

### Time Complexity:

- **Best Case:** O(nk)
- **Average Case:** O(nk)
- **Worst Case:** O(nk)
  Where `k` is the number of digits in the largest number.

---

## Conclusion

- **For small arrays:** Insertion Sort or Selection Sort can be useful.
- **For large datasets:** Merge Sort or Quick Sort is preferred.
- **For nearly sorted data:** Insertion Sort is efficient.

- **For integer sorting in a limited range:** Counting Sort or Radix Sort works best.

Choosing the right sorting algorithm depends on the problem constraints, input size, and time complexity considerations.

## Bubble Sort Algorithm

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The process is repeated until the list is sorted.

### Working Principle

1. Compare adjacent elements in the array.
2. Swap them if they are in the wrong order (i.e., the left element is greater than the right element).
3. Move to the next adjacent pair and repeat.
4. After one full pass, the largest element will be at the last position.
5. Repeat the process for the remaining elements until the entire array is sorted.

### Time Complexity

- **Best Case (Already Sorted):** $O(n)$
- **Worst Case (Reversed Order):** $O(n^2)$
- **Average Case:** $O(n^2)$

## Basic Bubble Sort Implementation in Java

```java
public class BubbleSort {
    public static void bubbleSort(int[] arr) {
        int n = arr.length;
        for (int i = 0; i < n - 1; i++) {
            for (int j = 0; j < n - 1 - i; j++) {
                if (arr[j] > arr[j + 1]) {
                    // Swap arr[j] and arr[j+1]
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                }
            }
        }
    }

    public static void printArray(int[] arr) {
        for (int num : arr) {
            System.out.print(num + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        int[] arr = {5, 2, 9, 1, 5, 6};
        System.out.println("Original Array:");
        printArray(arr);

        bubbleSort(arr);

        System.out.println("Sorted Array:");
        printArray(arr);
    }
}
```

## Explanation:

1. The **outer loop** runs $n - 1$ times because after each pass, the largest element is placed at its correct position.
2. The **inner loop** iterates from index `0` to `n - 1 - i` because after each pass, the largest elements are already sorted at the end.
3. Inside the inner loop, we compare adjacent elements and swap them if needed.

---

# Optimized Bubble Sort (With Early Termination)

A key optimization is to track whether any swaps were made in the inner loop. If no swaps occur in a pass, the array is already sorted, and we can terminate early.

```java
public class OptimizedBubbleSort {
    public static void bubbleSort(int[] arr) {
        int n = arr.length;
        boolean swapped;

        for (int i = 0; i < n - 1; i++) {
            swapped = false;
            for (int j = 0; j < n - 1 - i; j++) {
                if (arr[j] > arr[j + 1]) {
                    // Swap elements
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                    swapped = true;
                }
            }
            // If no swaps occurred, the array is already sorted
            if (!swapped) break;
        }
    }

    public static void printArray(int[] arr) {
        for (int num : arr) {
            System.out.print(num + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5, 6};
        System.out.println("Original Array:");
        printArray(arr);

        bubbleSort(arr);

        System.out.println("Sorted Array:");
        printArray(arr);
    }
}
```

### Why is this optimized?

- If the array is **already sorted**, the algorithm will terminate in **O(n)** time instead of **O(n²)**.
- The `swapped` variable checks if any swaps occurred. If not, the loop stops early, improving efficiency.

---

## Advantages and Disadvantages of Bubble Sort

### Advantages:

✅ Simple and easy to implement.
✅ Works well for nearly sorted data (**O(n) best case with optimization**).

### Disadvantages:

❌ **Slow** for large datasets (**O(n²) time complexity**).
❌ Not efficient compared to other sorting algorithms like **Quick Sort, Merge Sort, or Heap Sort**.

---

## Conclusion

- Bubble Sort is a basic sorting algorithm, good for teaching purposes but inefficient for large datasets.
- The optimized version improves performance by terminating early if no swaps occur.
- For better efficiency, use **Merge Sort, Quick Sort, or Heap Sort** for larger datasets.

Sure! Here's a detailed explanation of **Selection Sort** in Java, including how it works, its time complexity, and a sample implementation.

---

## ✅ What is Selection Sort?

Selection Sort is a simple comparison-based sorting algorithm. It works by dividing the array into two parts:

- **Sorted part** (initially empty)
- **Unsorted part** (initially the whole array)

In each pass:

1. Find the **minimum element** from the unsorted part.
2. Swap it with the **first element** of the unsorted part.
3. Move the boundary of the sorted part one step forward.

---

## 🔄 How Selection Sort Works (Step-by-step)

Example:
Input array: `[29, 10, 14, 37, 13]`

### Pass 1:

- Minimum in `[29, 10, 14, 37, 13]` → 10
- Swap 10 and 29 → `[10, 29, 14, 37, 13]`

### Pass 2:

- Minimum in `[29, 14, 37, 13]` → 13
- Swap 13 and 29 → `[10, 13, 14, 37, 29]`

### Pass 3:

- Minimum in `[14, 37, 29]` → 14 (already in correct place)

### Pass 4:

- Minimum in `[37, 29]` → 29
- Swap 29 and 37 → `[10, 13, 14, 29, 37]`

Sorted array ✅

---

## ⁉ Time Complexity

| Case | Time |
|---|---|
| Best Case | $O(n^2)$ |
| Average Case | $O(n^2)$ |
| Worst Case | $O(n^2)$ |

### Space Complexity:

- **O(1)** (in-place sorting)

---

## �💻 Java Code: Selection Sort

```java
public class SelectionSort {
    public static void selectionSort(int[] arr) {
        int n = arr.length;

        // One by one move boundary of unsorted subarray
        for (int i = 0; i < n - 1; i++) {
            // Find the index of the minimum element
            int minIndex = i;
            for (int j = i + 1; j < n; j++) {
                if (arr[j] < arr[minIndex]) {
                    minIndex = j;
                }
            }

            // Swap the found minimum with the first element
            int temp = arr[minIndex];
            arr[minIndex] = arr[i];
            arr[i] = temp;
        }
    }

    public static void printArray(int[] arr) {
        for (int val : arr) {
            System.out.print(val + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        int[] arr = {29, 10, 14, 37, 13};

        System.out.println("Original Array:");
        printArray(arr);

        selectionSort(arr);

        System.out.println("Sorted Array:");
        printArray(arr);
    }
}
```

## 📝 Key Points

- Selection Sort makes **fewer swaps** compared to Bubble Sort.
- It’s **not stable** (doesn’t preserve order of equal elements).
- Not suitable for large datasets due to **O(n²)** time.

# Insertion Sort in Java – Detailed Explanation

Insertion Sort is a simple and efficient sorting algorithm that works similarly to sorting playing cards in your hand. It builds the sorted array one element at a time by taking each element and inserting it into its correct position.

## ✅ How Insertion Sort Works

1. Assume the first element is already sorted.
2. Pick the next element.
3. Compare it with the elements in the sorted part.
4. Shift the larger elements to the right.
5. Insert the picked element at the correct position.
6. Repeat for all elements.

## 🔄 Step-by-step Example

Given array: `[9, 5, 1, 4, 3]`

| Pass | Current Element | Sorted Part Before Insert | Sorted Part After Insert |
|---|---|---|---|
| 1 | 5 | `[9]` | `[5, 9]` |
| 2 | 1 | `[5, 9]` | `[1, 5, 9]` |
| 3 | 4 | `[1, 5, 9]` | `[1, 4, 5, 9]` |
| 4 | 3 | `[1, 4, 5, 9]` | `[1, 3, 4, 5, 9]` |

Sorted array ✅

---

## ❓ Time Complexity

| Case | Time Complexity |
|---|---|
| Best Case | **O(n)** (Already sorted) |
| Average Case | **O($n^2$)** |
| Worst Case | **O($n^2$)** (Reversed order) |

### Space Complexity:

- **O(1)** (in-place sorting)

---

## �💻 Java Code: Insertion Sort

```java
public class InsertionSort {
    public static void insertionSort(int[] arr) {
        int n = arr.length;

        for (int i = 1; i < n; i++) {
            int key = arr[i]; // Current element
            int j = i - 1;

            // Move elements of arr[0..i-1] that are greater than key one position ahead
            while (j >= 0 && arr[j] > key) {
                arr[j + 1] = arr[j];
                j--;
            }

            // Insert key at its correct position
            arr[j + 1] = key;
        }
    }

    public static void printArray(int[] arr) {
        for (int num : arr) {
            System.out.print(num + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        int[] arr = {9, 5, 1, 4, 3};

        System.out.println("Original Array:");
        printArray(arr);

        insertionSort(arr);

        System.out.println("Sorted Array:");
        printArray(arr);
    }
}
```

## 📝 Key Points

- **Best case runs in O(n) time** (if already sorted).
- **Stable sort** (preserves order of equal elements).
- **Efficient for small datasets**, but slow for large ones (**$O(n^2)$**).

---

## When to Use Insertion Sort?

✅ When the array is **small** or **almost sorted**.
❌ Avoid for large datasets (use **Merge Sort** or **Quick Sort** instead).

---

Sure! Let's break down **Quick Sort** in Java with a beginner-friendly explanation, step-by-step logic, and code example.

---

## 📌 What is Quick Sort?

**Quick Sort** is a **divide-and-conquer** sorting algorithm that:

1. Picks a **pivot** element.
2. **Partitions** the array so that:
   - Elements **less than or equal to** the pivot go to the left,
   - Elements **greater** than the pivot go to the right.
3. **Recursively** applies the same process to the left and right parts.

---

## 🔁 How Quick Sort Works:

1. Choose a **pivot** (usually the last element).
2. Rearrange the array so:

    - All elements smaller than pivot go left,
    - All elements greater go right.
3. Recursively apply Quick Sort to subarrays.

---

## 🔧 Java Code for Quick Sort:

```java
public class QuickSort {

    // Helper function to swap elements
    public static void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }

    // Partition function
    public static int partition(int[] arr, int low, int high) {
        int pivot = arr[high]; // pivot is the last element
        int i = low - 1; // index of smaller element

        for (int j = low; j < high; j++) {
            if (arr[j] <= pivot) {
                i++;
                swap(arr, i, j);
            }
        }

        swap(arr, i + 1, high); // place pivot in correct position
        return i + 1; // return pivot index
    }

    // Quick sort function
    public static void quickSort(int[] arr, int low, int high) {
        if (low < high) {
            int pi = partition(arr, low, high); // partitioning index

            quickSort(arr, low, pi - 1);  // sort left part
            quickSort(arr, pi + 1, high); // sort right part
        }
    }

    // Main method
    public static void main(String[] args) {
        int[] arr = {10, 7, 8, 9, 1, 5};
        int n = arr.length;
        quickSort(arr, 0, n - 1);

        System.out.println("Sorted array:");
        for (int num : arr) {
            System.out.print(num + " ");
        }
    }
}
```

---

## 💹 Output:

```
Sorted array:
1 5 7 8 9 10
```

---

## ⏱ Time Complexity:

| Case | Time |
|---|---|
| Best Case | O(n log n) |
| Average Case | O(n log n) |
| Worst Case | O($n^2$) |

> Worst case happens when pivot is always the smallest or largest element (unbalanced partition).

---

## ✅ Key Points:

- **In-place** algorithm (no extra array).
- **Unstable** sort (relative order not preserved).

- **Fast in practice** for large datasets with random data.

---

Sure! Let's break down **Quick Select** in a beginner-friendly and visual way.

---

## 💡 What is Quick Select?

Quick Select is an efficient **selection algorithm** used to find the **kth smallest** or **kth largest** element in an **unsorted array**.

> 🔍 It is similar to Quick Sort, but instead of sorting the whole array, it focuses only on the part where the **kth element** could be.

---

## 🔷 When to Use It?

You want to find:

- 1st smallest = **Minimum**
- nth smallest = **Maximum**
- n/2-th smallest = **Median**
- kth largest = `n - k` smallest

✅ Without sorting the entire array.

---

## ⚙ How Does Quick Select Work?

1. **Pick a Pivot** (usually the last element).
2. **Partition** the array:
    - Elements **less than or equal** to pivot go to the left.
    - Elements **greater** than pivot go to the right.

3. After partitioning, check the **pivot’s final index**:
    - If it’s **equal to k**, return that element.
    - If it’s **greater than k**, the kth smallest is on the **left** side.
    - If it’s **less than k**, the kth smallest is on the **right** side.

> 🎯 The idea is to **narrow down** the search space.

---

## Example:

```
int[] arr = {7, 10, 4, 3, 20, 15};
int k = 3;
```

We want the **3rd smallest** element.

Steps:

1. Choose a pivot (say 15).
2. Partition the array → `[7, 10, 4, 3, 15, 20]`
3. Pivot 15 is at index 4 (0-based). We want index `k-1 = 2`.
4. Recurse on the left part: `[7, 10, 4, 3]`
5. Repeat until you find the element at index 2: **7**

---

## Time Complexity

| Case | Time |
|---|---|
| Average | **O(n)** |
| Worst (bad pivots every time) | **O($n^2$)** |

> But much faster than sorting the whole array (which is O(n log n)).

---

## ❓ Why is Quick Select Efficient?

Because it **only works on one side** of the array after partitioning. It doesn't sort everything—just what's necessary.

---

## ✅ In Summary

| Feature | Value |
|---|---|
| Purpose | Find kth smallest/largest |
| Based on | Quick Sort partitioning |
| Time Complexity (Avg) | O(n) |
| Modifies array? | Yes (in-place) |
| Stable? | No |

---

Sure! Let's break down **Counting Sort** in a very simple and beginner-friendly way 👇

---

## ❓ What is Counting Sort?

**Counting Sort** is a **non-comparison-based sorting algorithm** that works well when:

- The elements are **integers**
- The range of elements is **not very large**

Instead of comparing elements like Quick Sort or Merge Sort, it **counts how many times each value appears** and uses that to build the sorted array.

---

## 🎁 Key Idea

> Count how many times each number appears, and then place them in order using that count.

---

## 📊 How It Works (Step by Step)

Let's sort:

```
arr = {4, 2, 2, 8, 3, 3, 1}
```

### Step 1: Find the maximum element (say `max = 8`)

### Step 2: Create a count array of size `max + 1`

```
count = new int[9] // index 0 to 8
```

### Step 3: Count the frequency of each number in `arr`

```
arr:    [4, 2, 2, 8, 3, 3, 1]
count:  [0, 1, 2, 2, 1, 0, 0, 0, 1]
 index:  0  1  2  3  4  5  6  7  8
```

### Step 4: Build the sorted array

Go through the `count` array, and for each index, place it as many times as it appeared.

```
Sorted: [1, 2, 2, 3, 3, 4, 8]
```

---

## ✓ Code in Java

```java
import java.util.*;

public class CountSort {

    public static void countSort(int[] arr) {
        int max = Arrays.stream(arr).max().getAsInt();

        int[] count = new int[max + 1];

        // Count frequency
        for (int num : arr) {
            count[num]++;
        }

        // Fill original array
        int idx = 0;
        for (int i = 0; i < count.length; i++) {
            while (count[i]-- > 0) {
                arr[idx++] = i;
            }
        }
    }

    public static void main(String[] args) {
        int[] arr = {4, 2, 2, 8, 3, 3, 1};
        countSort(arr);

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

## ⏱️ Time and Space Complexity

| Aspect | Complexity |
|---|---|
| Time | O(n + k) |
| Space | O(k) |

Where:

- `n` = size of input array
- `k` = range of input values (max value)

---

## ⚠️ When to Use / Avoid

✅ Use Counting Sort if:

- You have **small range** integers (e.g., 0–100)
- You want **linear time sorting**

🚫 Avoid if:

- Numbers are **very large** (e.g., 1 to 1,000,000)
- You have **negative numbers** (need extra logic)
- Input is **not integers**

Absolutely! Let's break down **Radix Sort** step-by-step in a super easy way 👇

---

## ❓ What is Radix Sort?

**Radix Sort** is a **non-comparison-based sorting algorithm** that sorts numbers **digit by digit**, starting from the **least significant digit (LSD)** to the **most significant digit (MSD)**.

It uses a **stable sorting algorithm**, like **Counting Sort**, as a subroutine to sort digits.

---

## 🔁 How It Works

Imagine sorting the numbers:

```
[170, 45, 75, 90, 802, 24, 2, 66]
```

We'll sort them by:

1. Units digit (1s place)
2. Tens digit (10s place)
3. Hundreds digit (100s place)

At each step, we group and sort based on that digit.

---

## ❓ Step-by-Step Example:

**Initial Array:**

```
[170, 45, 75, 90, 802, 24, 2, 66]
```

### ✅ Step 1: Sort by 1s digit

```
[170, 90, 802, 2, 24, 45, 75, 66]
```

### ✅ Step 2: Sort by 10s digit

```
[802, 2, 24, 45, 66, 170, 75, 90]
```

## ✅ Step 3: Sort by 100s digit

```
[2, 24, 45, 66, 75, 90, 170, 802]
```

**Final Sorted Array 🎉**

---

## ✓ Java Code for Radix Sort

```java
public class RadixSort {

    // A utility function to get the maximum value in arr[]
    static int getMax(int[] arr) {
        int max = arr[0];
        for (int i : arr) {
            if (i > max)
                max = i;
        }
        return max;
    }

    // A function to do counting sort of arr[] according to the digit represented by exp.
    static void countingSort(int[] arr, int exp) {
        int n = arr.length;
        int[] output = new int[n];
        int[] count = new int[10]; // digits 0-9

        // Count occurrences
        for (int i = 0; i < n; i++) {
            int digit = (arr[i] / exp) % 10;
            count[digit]++;
        }

        // Prefix sum for stable sort
        for (int i = 1; i < 10; i++) {
            count[i] += count[i - 1];
        }

        // Build output array (go backwards for stability)
        for (int i = n - 1; i >= 0; i--) {
            int digit = (arr[i] / exp) % 10;
            output[count[digit] - 1] = arr[i];
            count[digit]--;
        }

        // Copy to original array
        for (int i = 0; i < n; i++) {
            arr[i] = output[i];
        }
    }

    static void radixSort(int[] arr) {
        int max = getMax(arr);

        // Sort for each digit: 1s, 10s, 100s, ...
        for (int exp = 1; max / exp > 0; exp *= 10) {
            countingSort(arr, exp);
        }
    }

    // Driver code
    public static void main(String[] args) {
        int[] arr = {170, 45, 75, 90, 802, 24, 2, 66};
        radixSort(arr);

        System.out.println("Sorted array:");
        for (int num : arr) {
            System.out.print(num + " ");
        }
    }
}
```

---

## ⏱ Time and Space Complexity

| Aspect | Complexity |
|---|---|
| Time | O(n × k) |
| Space | O(n + k) |

Where:

- `n` = number of elements
- `k` = number of digits in the maximum number (log10(max))

---

---

## ⚠ Notes

- Only works for **non-negative integers** by default.
- Can be extended to handle **negative numbers** with extra logic.
- Radix Sort is great when:
    - You're sorting **large number of integers**
    - The values are **not too large** in digit length

---

Sure! Here's the updated table with a **note highlighting the most efficient sorting algorithm(s)** based on typical use cases:

---

## 📊 Time and Space Complexities of Sorting Algorithms

| Sorting Algorithm | Best Case Time | Average Case Time | Worst Case Time | Space Complexity | Stable? |
|---|---|---|---|---|---|
| **Bubble Sort** | O(n) | O(n²) | O(n²) | O(1) | Yes |
| **Selection Sort** | O(n²) | O(n²) | O(n²) | O(1) | No |
| **Insertion Sort** | O(n) | O(n²) | O(n²) | O(1) | Yes |
| **Merge Sort** | O(n log n) | O(n log n) | O(n log n) | O(n) | Yes |
| **Quick Sort** | O(n log n) | O(n log n) | O(n²) | O(log n) | No |
| **Heap Sort** | O(n log n) | O(n log n) | O(n log n) | O(1) | No |
| **Counting Sort** | O(n + k) | O(n + k) | O(n + k) | O(k) | Yes |
| **Radix Sort** | O(nk) | O(nk) | O(nk) | O(n + k) | Yes |
| **Bucket Sort** | O(n + k) | O(n + k) | O(n²) | O(n + k) | Yes |

| Sorting Algorithm | Best Case Time | Average Case Time | Worst Case Time | Space Complexity | Stable? |
|---|---|---|---|---|---|
| **Tim Sort** (Java's default) | O(n) | O(n log n) | O(n log n) | O(n) | Yes |

---

## 📝 Note:

✅ **Tim Sort**, which is a hybrid of Merge Sort and Insertion Sort (used in Java and Python's built-in sorting), is considered one of the **most efficient general-purpose sorting algorithms** for real-world data due to its stability and performance on partially sorted data.

For specific use cases:

- Use **Counting/Radix/Bucket Sort** when the input is constrained (e.g., integers in a known range).
- Use **Quick Sort** for performance, but **Merge Sort** or **Tim Sort** when stability is important.

# 📄 Java String – Detailed Notes

Strings are a fundamental part of Java programming. This guide provides a thorough overview of strings in Java, their properties, and common operations.

---

## 📌 Table of Contents

---

## 🔷 What is a String?

- A `String` in Java is an object that represents a sequence of characters.
- It is part of the `java.lang` package.
- Strings are **immutable**, meaning once created, their value cannot be changed.

```
String str = "Hello, World!";
```

---

## 🔷 String Declaration & Initialization

### ✅ Using string literal:

```
String str1 = "Hello";
```

### ✅ Using `new` keyword:

```
String str2 = new String("Hello");
```

> 🔸 `str1` will refer to an object from the **String pool**
> 🔸 `str2` will create a **new object** in the heap, even if an identical one exists in the pool.

---

## 🔷 String Immutability

- Once a `String` object is created, its contents **cannot be changed**.
- Any method that seems to modify a string actually returns a **new string**.

```java
String str = "Hello";
str.concat(" World");
System.out.println(str); // Output: Hello

String newStr = str.concat(" World");
System.out.println(newStr); // Output: Hello World
```

---

## 🔷 String Pool

- Java optimizes memory usage with a **String pool** in the heap.
- All string literals are stored in this pool.
- When a new literal is created, Java checks if it already exists in the pool before creating a new one.

```java
String a = "Java";
String b = "Java";
System.out.println(a == b); // true
```

---

## 🔷 Commonly Used String Methods

| Method | Description |
|---|---|
| `length()` | Returns the length of the string |
| `charAt(int index)` | Returns character at specified index |
| `substring(int start)` | Returns substring from start index |

| Method | Description |
|---|---|
| `substring(int start, int end)` | Returns substring between indices |
| `toLowerCase()` | Converts string to lowercase |
| `toUpperCase()` | Converts string to uppercase |
| `trim()` | Removes leading and trailing whitespace |
| `equals(String another)` | Compares content |
| `equalsIgnoreCase(String)` | Case-insensitive comparison |
| `contains(CharSequence)` | Checks if sequence exists |
| `replace(old, new)` | Replaces characters/substring |
| `split(String regex)` | Splits string by regex |
| `indexOf(char)` | Returns index of first occurrence |
| `lastIndexOf(char)` | Returns last index |
| `isEmpty()` | Checks if string is empty (`length() == 0`) |

---

## 🔹 String Comparison

### ✔ Using `==` (Reference Comparison)

```java
String a = "Test";
String b = "Test";
System.out.println(a == b); // true
```

### ✅ Using `.equals()` (Value Comparison)

```java
String a = new String("Test");
String b = new String("Test");
System.out.println(a.equals(b)); // true
```

### ✅ Case-insensitive comparison

```java
a.equalsIgnoreCase(b);
```

---

## 🔹 String Concatenation

### ✅ Using `+` operator

```java
String fullName = "John" + " " + "Doe";
```

### ✅ Using `concat()` method

```java
String fullName = "John".concat(" Doe");
```

### ✅ Performance Note:

- For **multiple or large** string modifications, prefer `StringBuilder` or `StringBuffer`.

---

## 🔹 StringBuffer vs StringBuilder

| Feature | `StringBuffer` | `StringBuilder` |
|---|---|---|
| Mutability | Mutable | Mutable |

| Feature | `StringBuffer` | `StringBuilder` |
|---|---|---|
| Thread Safe | Yes (synchronized) | No |
| Performance | Slower (due to sync) | Faster |
| Use Case | Multithreaded apps | Single-threaded apps |

```
StringBuffer sb = new StringBuffer("Hello");
sb.append(" World");
System.out.println(sb); // Hello World
```

---

## 🔹 String Formatting

```
String name = "Alice";
int age = 25;

String formatted = String.format("Name: %s, Age: %d", name, age);
System.out.println(formatted); // Name: Alice, Age: 25
```

Other specifiers:

- `%s` → String
- `%d` → Integer
- `%f` → Float
- `%.2f` → Float with 2 decimal places

---

## ❓ Important String Interview Questions

1. What is the difference between `==` and `.equals()` in Java?
2. Why are Strings immutable in Java?
3. What is the String constant pool?

4. How does `StringBuilder` improve performance?
5. Explain the difference between `String`, `StringBuilder`, and `StringBuffer`.
6. How does Java handle memory with Strings?

Here's the explanation in **Markdown format** covering:

1. How to take String input in Java
2. The difference between `next()` and `nextLine()`

---

# 🗒 Taking String Input in Java

In Java, the most common way to take input from the user is using the `Scanner` class from the `java.util` package.

---

## 🔹 Importing Scanner

Before using Scanner, you must import it:

```java
import java.util.Scanner;
```

---

## 🔹 Creating a Scanner Object

```java
Scanner sc = new Scanner(System.in);
```

---

## 🔹 Taking String Input

### ✅ Using `next()`

```
System.out.print("Enter a word: ");
String word = sc.next();
System.out.println("You entered: " + word);
```

🔸 `next()` reads input **only until the first space**.
🔸 It does **not read the whole line**.

### ✅ Using `nextLine()`

```
System.out.print("Enter a sentence: ");
String sentence = sc.nextLine();
System.out.println("You entered: " + sentence);
```

🔸 `nextLine()` reads **the entire line**, including spaces, until the user hits Enter.

---

## 🔄 Difference Between `next()` and `nextLine()`

| Feature | `next()` | `nextLine()` |
|---|---|---|
| Reads | Word/token (up to whitespace) | Whole line (until Enter key is pressed) |
| Stops at | Space, tab, or newline character | Newline character |
| Includes spaces | ❌ No | ✅ Yes |
| Use case | Reading single words or tokens | Reading full-line input (e.g., full names) |

## ⚠ Common Issue

If you use `nextInt()` or `next()` before `nextLine()`, it may skip the input.

### Example:

```
int age = sc.nextInt();     // reads number
sc.nextLine();              // consumes leftover newline
String name = sc.nextLine(); // now reads correctly
```

Always handle the newline when switching from numeric/token input to `nextLine()`.

## ✅ Sample Program

```
import java.util.Scanner;

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

        System.out.print("Enter your first name: ");
        String firstName = sc.next();

        sc.nextLine(); // clear buffer

        System.out.print("Enter your full address: ");
        String address = sc.nextLine();

        System.out.println("First Name: " + firstName);
        System.out.println("Address: " + address);
    }
}
```

In Java, understanding the nuances of `String` handling is crucial for writing efficient and effective code. Let's delve into the concepts of **interning**, **immutability**, the use of the `new` **keyword**, **memory implications**, **performance considerations**, and the differences between `equals()` **and** `==` when working with strings.

## 1. String Immutability

In Java, `String` objects are **immutable**, meaning once a `String` object is created, its value cannot be changed. Any operation that seems to modify a `String` actually results in the creation of a new `String` object. For example:

```java
String str = "Hello";
str = str.concat(" World");
System.out.println(str); // Outputs "Hello World"
```

In this code, `str.concat(" World")` creates a new `String` object with the value `"Hello World"`, and `str` now references this new object. The original `"Hello"` string remains unchanged.

**Benefits of Immutability:**

- **Security:** Immutable objects are inherently thread-safe, reducing synchronization issues in concurrent applications.
- **Performance:** Facilitates caching and reuse, as immutable objects can be safely shared.
- **String Pooling:** Supports the concept of the string pool, optimizing memory usage by reusing instances.

## 2. String Interning

**String interning** is a method of storing only one copy of each distinct `String` value in a common pool, known as the **string pool**. When a `String` is interned, it ensures that all identical `String` literals share the same memory reference.

**How It Works:**

```java
String str1 = "Java";
String str2 = "Java";
System.out.println(str1 == str2); // Outputs true
```

In this example, both `str1` and `str2` refer to the same object in the string pool.

However, when using the `new` keyword:

```java
String str3 = new String("Java");
System.out.println(str1 == str3); // Outputs false
```

Here, `str3` creates a new `String` object in the heap, not in the string pool, resulting in a different memory reference.

### Using `intern()` Method:

To manually add a `String` to the string pool, you can use the `intern()` method:

```java
String str4 = new String("Java").intern();
System.out.println(str1 == str4); // Outputs true
```

This ensures that `str4` refers to the interned string in the pool.

### Benefits of String Interning:

- **Memory Efficiency:** Reduces memory usage by avoiding duplicate `String` objects.
- **Performance:** Allows for faster comparisons using `==` since it compares references.

### Considerations:

While interning can save memory, excessive interning, especially of large strings or a vast number of unique strings, can lead to increased memory consumption and potential performance issues. It's essential to use interning judiciously.

## 3. Using the `new` Keyword with Strings

When you create a `String` using the `new` keyword, a new object is instantiated in the heap memory, regardless of whether an identical `String` exists in the string pool:

```java
String str = new String("Hello");
```

This approach bypasses the string pool and creates a distinct object, which can lead to increased memory usage if not managed carefully.

## 4. Memory Implications

The immutability and interning of strings have significant memory implications:

- **String Pooling:** By storing literals in the string pool, Java conserves memory by reusing instances.
- **Heap Usage:** Creating strings with the `new` keyword increases heap memory usage, as each instantiation creates a new object.

Efficient use of strings, especially in large applications, is crucial to prevent excessive memory consumption.

## 5. Performance Considerations

While string interning can improve performance by enabling faster reference comparisons, it's essential to be cautious:

- **Interning Overhead:** The process of interning has its own overhead. Excessive interning can lead to performance degradation.
- **Garbage Collection:** Interned strings may not be garbage collected, leading to potential memory leaks if not handled properly.

Therefore, interning should be used when there's a clear benefit, such as with a high number of duplicate strings.

## 6. Comparing Strings: `equals()` vs. `==`

Understanding the difference between `equals()` and `==` is vital:

- `==` **Operator:** Compares object references to check if they point to the same memory location.

```java
String str1 = "Java";
String str2 = "Java";
System.out.println(str1 == str2); // true, same reference in string pool
```

- `equals()` **Method:** Compares the content of the strings, regardless of their memory references.

```java
String str3 = new String("Java");
System.out.println(str1.equals(str3)); // true, same content
```

In summary, use `==` to check if two references point to the same object and `equals()` to compare the actual content of the strings.

By comprehending these aspects of Java's `String` class, developers can write more efficient and reliable code, making informed decisions about memory and performance trade-offs.

**Interning** is a concept primarily related to programming languages and memory optimization. Let's explore it in depth, covering the **what**, **why**, **how**, **when**, **how to avoid**, and **implications** of interning.

---

## 🔍 What is Interning?

**Interning** is a method of **storing only one copy of each distinct immutable value**, which must be **shared across multiple references**. When two variables have the same value, interning ensures they **reference the same memory location** instead of duplicating memory for identical content.

Common in:

- **Strings** (most commonly)
- **Integers** (especially small ones)
- **Enums or Symbols** (in languages like Ruby, JavaScript, or Lisp)

## Example in Python:

```python
a = "hello"
b = "hello"
print(a is b)  # True — both point to the same interned string object
```

> The `is` operator checks if both refer to the same object in memory.

---

## 💡 Why is Interning Used?

1. **Memory Optimization**:
   Repeating the same immutable values across an app (e.g., `"status": "active"`) creates unnecessary duplicates. Interning allows **one shared copy**, saving memory.
2. **Performance Boost**:
   Comparing object references (`is`) is faster than comparing actual values (`==`). So interning helps with **faster comparisons**, especially in dictionaries, sets, etc.
3. **String Deduplication**:
   Useful when thousands of strings are the same (e.g., in parsing source code or HTML, logs, or compilers).

---

## 🛠 How Does Interning Work?

## String Interning

### Python:

Python automatically interns:

- Strings that look like identifiers (`'hello'`, `'Python3'`)
- Strings used frequently (cached)

But not always:

```python
x = "hello world"
y = "hello world"
print(x is y)  # Might be False — not interned automatically
```

To force interning:

```python
import sys
x = sys.intern("hello world")
y = sys.intern("hello world")
print(x is y)  # True
```

### Java:

Java uses a string pool:

```java
String a = "hello";
String b = "hello";
System.out.println(a == b); // true — same memory
```

But if you create new object:

```
String a = new String("hello");
String b = "hello";
System.out.println(a == b); // false
```

To intern:

```
String a = new String("hello").intern();
```

---

## ⊘ How to Avoid Interning?

In most cases, **interning is beneficial**, but if you're doing memory-sensitive work or security-critical comparisons, you may want to **avoid or be cautious** with interning.

### When Avoiding Interning Might Be Needed:

1. **Security**:
   Don't rely on `is` for secure comparisons:

   ```
   password == input_password  # Use ==, not is
   ```

2. **Confusion and Bugs**:
   Using `is` for string comparisons might break code when strings are not interned.
3. **Unintended Sharing**:
   If mutability were allowed (which it isn't for strings), changing one value would affect others.
4. **Overuse of Interning**:
   If you intern too many values, the intern pool can grow unnecessarily large — especially in Java.

---

## ⚠ Implications of Interning

### ✅ Advantages:

- Reduces memory usage (especially for large sets of identical strings)
- Speeds up comparisons
- Used by many standard libraries and compilers

### ❌ Disadvantages:

- Can lead to **subtle bugs** if developers rely on `is` instead of `==`
- Extra cost of interning if not done carefully
- Interned objects live longer (may increase GC pressure in Java)
- May **leak memory** if not handled properly (e.g., custom interning in long-running apps)

---

## 💬 When Does Interning Happen Automatically?

| Language | Interning Applies To | Happens Automatically? |
|---|---|---|
| Python | Strings, Small Integers (-5 to 256) | Partially |
| Java | Strings | Yes (string pool) |
| JavaScript | Symbols ( `Symbol("key")` ) | Yes |
| Ruby | Symbols ( `:key` ) | Yes |
| C# | String literals | Yes |

---

## 💡 Best Practices

- Use `==` for value comparisons, not `is`
- Intern only if you're dealing with a large number of repeated immutable values

- Use built-in interning (like Python’s `sys.intern()`) when needed
- Avoid manual interning in memory-sensitive environments unless you profile and benchmark

---

## 🔁 Summary

| Concept | Description |
|---|---|
| **Interning** | Sharing a single memory location for identical immutable values |
| **Used for** | Optimization (memory + speed) |
| **Common in** | Strings, numbers, enums |
| **Avoid when** | Security checks, overuse, unintended behavior |
| **Implication** | Can boost performance but may cause subtle bugs or memory leaks |

---

## 🔍 What is Immutability?

**Immutability** refers to the **inability to change an object after it is created**.

In simpler terms:

> "Once an immutable object is created, its internal state cannot be modified."

In Java:

- **Immutable objects** have **final fields**, **no setters**, and **no methods** that modify their state.
- Instead of modifying, you create a **new object** with the new state.

---

## 🔐 Characteristics of Immutable Objects in Java

To make a class immutable:

1. Declare the class as `final` (so it can't be subclassed).
2. Make all fields `private` and `final`.
3. No setters or methods that modify internal state.
4. Initialize all fields via constructor.
5. Ensure **deep copies** of mutable fields are made (to prevent shared references).
6. Don’t allow the reference of mutable objects to escape.

---

## ✅ Example of an Immutable Class in Java

```java
public final class Person {
    private final String name;
    private final int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() { return name; }
    public int getAge() { return age; }
}
```

If someone needs a new `Person` with a different age:

```java
Person p1 = new Person("Alice", 25);
Person p2 = new Person(p1.getName(), 26);
```

---

## ❓ Why Use Immutability?

1. **Thread Safety**:
   No synchronization needed — immutable objects can't be changed by multiple threads.
2. **Simpler Code**:
   Easy to reason about — no side effects or unexpected changes.
3. **Reliable Hashing**:
   Safe to use as **keys in hash-based collections** (`HashMap`, `HashSet`) since hashcode doesn't change.
4. **Functional Programming**:
   Encourages a **pure functional style** (no side effects, predictable output).
5. **Safe Sharing**:
   Immutable objects can be freely shared or cached without defensive copies.

---

## ⚠ Implications & Trade-offs

| Aspect | Positive | Trade-offs |
|---|---|---|
| **Thread Safety** | No concurrency bugs | Might need to recreate objects often |
| **Security** | Cannot be tampered with | Higher memory usage for new instances |
| **Ease of Debugging** | No accidental state changes | Can lead to object explosion (many instances) |
| **Use in Collections** | Safe as keys in `HashMap` | Deep copies are needed for mutable objects inside |
| **Garbage Collection** | Predictable lifespan | Might increase short-lived objects if not optimized |

---

## 🔄 Common Immutable Classes in Java

| Class | Description |
|---|---|
| `String` | Most well-known immutable class |
| `Integer`, `Long`, `Double`, etc. | Immutable wrapper types |
| `BigDecimal`, `BigInteger` | Used for accurate numerical computations |
| `LocalDate`, `LocalTime`, `LocalDateTime` | From `java.time` API (Java 8+) |
| `Optional<T>` | Immutable container for optional values |

---

## ⚡ Impact of Immutability in Java (with Examples)

### 1. In Collections and Keys

```java
Map<Person, String> map = new HashMap<>();
Person p = new Person("John", 30);
map.put(p, "Engineer");

// If p was mutable and we changed its fields, it may not be found again
```

Immutability guarantees that the `hashCode()` and `equals()` remain consistent during its lifetime.

---

### 2. In Multi-threading

```java
class UserService {
    private final User user; // if User is immutable, no synchronization needed
}
```

No synchronization = better performance and simpler code.

---

## 3. Functional Java (Java 8+)

Immutability supports use of:

- `Stream API`
- `Optional`
- `Collectors`
- Lambda expressions

All rely on pure functions and non-mutating behavior.

---

## 4. Defensive Copies (Deep Immutability)

If an immutable class has a mutable field:

```java
public final class Employee {
    private final Date dateOfJoining;

    public Employee(Date date) {
        this.dateOfJoining = new Date(date.getTime()); // defensive copy
    }

    public Date getDateOfJoining() {
        return new Date(dateOfJoining.getTime()); // defensive copy again
    }
}
```

Without this, callers could modify the internal `Date`.

---

## 🔷 Summary Table

| Feature | Mutable | Immutable |
|---|---|---|
| State Changes | Allowed | Not allowed after creation |
| Thread Safety | Needs synchronization | Naturally thread-safe |
| Memory Usage | Potentially lower | May be higher |
| Debugging | Complex due to shared state | Easier — no side effects |
| Object Sharing | Risky | Safe |
| Use in Collections | Can break key-based lookups | Reliable |

---

## 🔚 Conclusion

Immutability in Java is a **powerful design choice**:

- It simplifies multithreading.
- Encourages clean, reliable, and functional code.
- Essential in building secure and bug-free systems.

But it comes with **trade-offs** like potential performance overhead due to object recreation and memory usage — which can often be offset using **builder patterns**, **value objects**, or **record classes** in Java 14+.

---

## 🔁 What Does It Mean in Context of `String`?

### ➤ "Reference is mutable":

- The variable (reference) holding the `String` **can be changed to point to another object**.

### ➤ "Instance is not":

- The actual `String` object in memory **cannot be changed** once it is created.

---

## ❓ Example:

```
String str = "Hello";    // str points to "Hello"
str = "World";           // str now points to "World"
```

### Breakdown:

1. `"Hello"` is created and `str` references it.
2. Then `"World"` is created, and **`str` is reassigned** to reference it.
3. **`"Hello"` remains unchanged** in memory (still exists if referenced elsewhere).

✔ **Reference ( `str` ) is mutable** — it changed from pointing to `"Hello"` to `"World"`
✕ **The `String` instance itself is not mutable** — you cannot change the characters inside `"Hello"`

---

## ✕ You Cannot Do This:

```
String str = "Hello";
str.setCharAt(0, 'Y'); // ✕ Compilation error — no such method
```

Because:

- `String` has **no methods that modify its internal character array**
- It does **not allow in-place modification**

---

## 🔁 All String Operations Create New Objects

```java
String original = "Java";
String updated = original.concat(" Programming");

System.out.println(original); // "Java"
System.out.println(updated);  // "Java Programming"
```

✅ `original` is unchanged — because `concat()` returns a **new String**

---

## 🔍 Internally: Why Is String Immutable?

- `private final char[] value` — Java stores characters in a final array.
- Fields are `private` and **cannot be modified** once initialized.
- This makes `String` safe for:
  - Hashing (used in `HashMap` keys)
  - Caching (String pool)
  - Thread-safety (no shared modification)

---

## 📌 Final Summary

| Concept | Explanation |
|---|---|
| **Reference is mutable** | You can reassign a `String` variable to a different `String` object. |
| **Instance is not** | Once a `String` object is created, its internal characters cannot be changed. |

So:

```java
String name = "Alice";
name = "Bob"; // ✅ mutable reference

// But:
name.charAt(0) = 'X'; // ❌ Error! Can't mutate String content
```

---

---

## 📌 What is `StringBuilder`?

`StringBuilder` is a **mutable** sequence of characters — unlike `String`, which is **immutable**.

> 🔥 In short: If you need to frequently **modify**, **append**, or **delete** characters in a string, use `StringBuilder`.

---

## ❓ Why `StringBuilder` Exists

Imagine doing this:

```java
String str = "Hello";
str += " World"; // creates a new String object
str += "!";      // again, creates a new one
```

- Each `+=` creates **a new `String` object** in memory because `String` is immutable.
- This leads to **performance issues** in loops or heavy concatenations.

That's where `StringBuilder` helps — it modifies the **same object in-place**.

---

## 🔍 Basic Syntax and Methods

```java
StringBuilder sb = new StringBuilder("Hello");

sb.append(" World");       // "Hello World"
sb.insert(5, ",");         // "Hello, World"
sb.delete(5, 6);           // "Hello World"
sb.setCharAt(0, 'Y');      // "Yello World"
sb.reverse();              // "dlroW olleY"
```

## 🔑 Common Methods:

| Method | Description |
|---|---|
| `append(String s)` | Adds to the end |
| `insert(int offset, s)` | Inserts at position |
| `delete(int start, end)` | Removes a portion |
| `replace(int, int, s)` | Replaces part with another string |
| `reverse()` | Reverses the character sequence |
| `setCharAt(int, char)` | Changes a character at index |
| `toString()` | Converts back to a regular `String` |

---

## ⚙ How it Works Internally

```java
StringBuilder sb = new StringBuilder("Hello");
```

- Internally, it uses a **resizable character array** (`char[]`).
- It keeps track of:
  - **Capacity** – size of the internal array

  - **Length** – number of characters currently used

## Example Behind-the-Scenes:

```
sb → ┌──────────────────────────────┐
     │ char[] value = ['H', 'e', 'l', 'l', 'o', '', '', '', '', '', ...]
     │ capacity = 16 (default or more)
     │ length = 5
     └──────────────────────────────┘
```

If you append more characters than the current capacity, it **doubles** (like ArrayList).

---

## 🔄 `StringBuilder` vs `StringBuffer`

| Feature | `StringBuilder` | `StringBuffer` |
|---|---|---|
| Mutability | Mutable | Mutable |
| Thread Safety | ❌ Not synchronized | ✅ Synchronized (thread-safe) |
| Performance | ✅ Faster (single thread) | ❌ Slower (because of sync) |

> ✔ Use `StringBuilder` in **single-threaded** code
> ✔ Use `StringBuffer` if you **need thread safety**

---

## 📊 Performance Difference

```java
// Concatenating in loop — bad with String
String s = "";
for (int i = 0; i < 10000; i++) {
    s += i;
}

// Better with StringBuilder
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 10000; i++) {
    sb.append(i);
}
```

Result: `StringBuilder` can be **10x–100x faster** than `String` in such scenarios.

---

## ✅ When to Use `StringBuilder`

| Scenario | Recommendation |
|---|---|
| Concatenating strings in a loop | ✅ Use `StringBuilder` |
| Modifying or reversing strings | ✅ Use `StringBuilder` |
| Single-threaded environment | ✅ Use `StringBuilder` |
| Thread-safe environment | ❌ Use `StringBuffer` or other sync-safe structure |

---

## 🔶 Summary

- `StringBuilder` is **mutable**, unlike `String`.
- It's efficient for string manipulation (append, insert, delete).
- Stores characters in an internal **char array** that grows as needed.

- Much faster than `String` for repeated modifications.

---

## 🔤 Character Conversion Notes: ASCII Logic

### 💡 Understanding ASCII-based Case Conversion

In Java (and most programming languages), characters are internally represented using ASCII values.

### 🔁 Basic Observations

```
'p' - 'a' = 'P' - 'A'
```

This means the **distance** between a lowercase letter and `'a'` is the same as the distance between the corresponding uppercase letter and `'A'`.

---

### 🔄 Conversion Formulas

#### ✅ Convert Uppercase to Lowercase:

```
lc = 'a' + (uc - 'A');
```

📌 Example:
If `uc = 'C'`, then
`lc = 'a' + ('C' - 'A') = 'a' + 2 = 'c'`

#### ✅ Convert Lowercase to Uppercase:

```
uc = 'A' + (lc - 'a');
```

📌 Example:

If `lc = 'd'`, then
`uc = 'A' + ('d' - 'a') = 'A' + 3 = 'D'`

---

## 📌 General Summary

- `lc = 'a' + uc - 'A'` → **Uppercase to Lowercase**
- `uc = 'A' + lc - 'a'` → **Lowercase to Uppercase**

These formulas are helpful for:

- Character case conversion without built-in methods
- ASCII math-based challenges
- Understanding underlying character operations

---

# Java ArrayList Notes

## What is an ArrayList?

- A resizable array in Java.
- Part of the `java.util` package.
- Grows and shrinks dynamically.
- Can store objects (not primitives directly).

---

## Import Statement

```java
import java.util.ArrayList;
```

---

## Declaration and Initialization

```
ArrayList<Integer> list = new ArrayList<>();
ArrayList<String> names = new ArrayList<>();
```

---

## Adding Elements

```
list.add(10);           // adds 10 at the end
list.add(1, 20);        // inserts 20 at index 1
```

---

## Removing Elements

```
list.remove(2);                         // removes element at index 2
list.remove(Integer.valueOf(10));    // removes value 10 (first occurrence)
```

---

## Updating Elements

```
list.set(0, 99);        // sets index 0 to 99
```

---

## Accessing Elements

```
int val = list.get(1);  // gets value at index 1
```

---

## Size

```
int size = list.size(); // returns number of elements
```

---

## Looping

```
for (int i = 0; i < list.size(); i++) {
    System.out.println(list.get(i));
}

for (int val : list) {
    System.out.println(val);
}
```

---

## Searching

```
list.contains(30);       // returns true if 30 is present
list.indexOf(30);        // returns index of 30 or -1 if not found
```

---

## Clearing All Elements

```
list.clear();
```

---

## Notes

- Maintains insertion order
- Allows duplicate values

- Cannot store primitive types directly (use wrapper classes like `Integer`, `Double`, etc.)