JAVA Basics:

Java is a widely used, object-oriented programming language known for its platform independence(WORA), robustness, and security. Here are some key basics:

**1. Features of Java**

* **Platform Independent:** "Write Once, Run Anywhere" (WORA) using the Java Virtual Machine (JVM).
* **Object-Oriented:** Everything in Java is based on objects and classes.
* **Simple and Secure:** No explicit pointers, built-in security features.
* **Multithreading:** Supports concurrent execution of multiple tasks.
* **Automatic Memory Management:** Uses garbage collection to free memory.

### 🔹 Java Execution Flow:

Source Code (.java)

↓

Compiler (javac)

↓

Bytecode (.class)

↓

JVM

↓

Machine Code (Output)

## 🔄 Summary Table

| **Component** | **Role** |
| --- | --- |
| **JDK** | Java Development Kit - for development (includes JRE + tools) |
| **JRE** | Java Runtime Environment - to run Java programs |
| **JVM** | Java Virtual Machine - executes bytecode |
| **Compiler (javac)** | Converts source code to bytecode |
| **Bytecode** | Intermediate code executed by JVM |

Object:

It is a real wold Entity having identity,state and Behaviour

Instance of class

Ex:Car

State(member var):Wheel,Gear,breaketc

Behaviour(method):We can drive car

Class:

In Java, **a class is a blueprint** for creating objects. It defines variables (fields) and methods (functions) that objects will have.

Group of object

Structure of class:

class Car

{

// Fields (Variables)

String brand = "Toyota"; int speed = 120;

// Method (Function)

void displayCarInfo()

{

System.out.println("Brand: " + brand);

System.out.println("Speed: " + speed + " km/h"); } }

Creating Object of the Class:

public class Main {

public static void main(String[] args) {

Car myCar = new Car(); // Creating an object

myCar.displayCarInfo(); // Calling a method

}

}

A class is a group of objects which have common properties.

A class in Java contains:

Constructors

Methods

Fields

Blocks

Nested class and interface

**2. Basic Structure of a Java Program**

public class HelloWorld {

public static void main(String[] args) {

System.out.println("Hello, World!");

}

}

* public class HelloWorld → Class declaration.
* public static void main(String[] args) → Main method, the entry point of the program.
* System.out.println("Hello, World!"); → Prints output to the console.

**Parameters used in First Java Program**

* **class** keyword is used to declare a class in java.
* **public** keyword is an access modifier. It means it is visible to all.
* **static** is a keyword. If we declare any method as static, it is known as the static method. The core advantage of the static method is that there is no need to create an object to invoke the static method.
* **void** means it doesn't return any value.
* **main** represents the starting point of the program.
* **String[] args** is used for command line argument.
* **System.out.println()** is used to print statement.
* Here, System is a class, out is the object of PrintStream class, println() is the method of PrintStream class.

### ****Data Types in Java****

A **data type** in Java specifies the type of data a variable can store. Java has two main categories of data types:

1. **Primitive Data Types** (Basic types like int, float, char, etc.)
2. **Non-Primitive Data Types** (Objects, Arrays, Strings, etc.)

## **1. Primitive Data Types**

These are the most basic data types built into Java.

| **Data Type** | **Size** | **Default Value** | **Description** |
| --- | --- | --- | --- |
| byte | 1 byte | 0 | Stores small integers (-128 to 127) |
| short | 2 bytes | 0 | Stores medium integers (-32,768 to 32,767) |
| int | 4 bytes | 0 | Stores whole numbers (-2³¹ to 2³¹-1) |
| long | 8 bytes | 0L | Stores large integers (-2⁶³ to 2⁶³-1) |
| float | 4 bytes | 0.0f | Stores decimal numbers (up to 7 decimal places) |
| double | 8 bytes | 0.0d | Stores large decimal numbers (up to 16 decimal places) |
| char | 2 bytes | '\u0000' | Stores a single character (Unicode) |
| boolean | 1 bit | false | Stores true or false |

### ****Example Usage****

public class DataTypesExample {

public static void main(String[] args) {

int num = 10;

double price = 99.99;

char grade = 'A';

boolean isJavaFun = true;

System.out.println("Number: " + num);

System.out.println("Price: " + price);

System.out.println("Grade: " + grade);

System.out.println("Java is fun: " + isJavaFun);

}

}

**Output:**

Number: 10

Price: 99.99

Grade: A

Java is fun: true

* **Non-Primitive Data Types:**
  + String, Arrays, Classes, Interfaces

### ****JVM (Java Virtual Machine)****

#### **Definition:**

JVM (**Java Virtual Machine**) is an abstract machine that enables a computer to run Java programs. It acts as a runtime environment that converts Java **bytecode** into machine-specific code.

**How JVM Works?**

1. **Java Source Code (.java)** → Written by the programmer.
2. **Compilation (Javac - Java Compiler)** → Converts the .java file into **bytecode** (.class file).
3. **JVM Execution** → The .class file (bytecode) is executed by the JVM, which converts it into machine code using the Just-In-Time (JIT) compiler.

**JVM Architecture**

JVM has the following main components:

bbbbn

1. **Class Loader**
   * Loads .class files into memory.
   * Performs **verification, linking, and initialization** of classes.
2. **Runtime Memory Areas (JVM Memory Structure)**
   * **Method Area** → Stores class-level information (method code, static variables).
   * **Heap Area** → Stores objects and instance variables.
   * **Stack Area** → Stores method execution details and local variables.
   * **PC Register** → Stores the address of the current instruction being executed.
   * **Native Method Stack** → Stores native (non-Java) method calls.
3. **Execution Engine**
   * **Interpreter** → Executes bytecode line by line (slow).
   * **JIT Compiler (Just-In-Time)** → Converts bytecode into native code for faster execution.
4. **Garbage Collector (GC)**
   * **Manages memory by automatically removing unused objects** to free up space.

Finalize:

The finalize method in Java is used to perform **cleanup operations** before an object is **garbage collected**. It is a **protected method** of the Object class, and it can be overridden in a class.

**Java Component- JRE**

JRE stands for Java Runtime Environment.

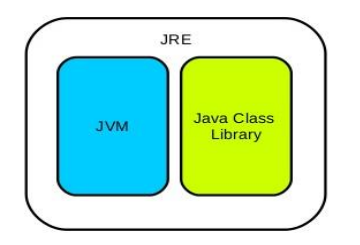
It plays a key role while executing any java application.

It is a collection of tools that together allow the development of applications and provide a runtime

environment.

The JVM is a part of JRE.

This is like JVM, platform-dependent.



**Java Component- JDK**

JDK stands for Java Development Kit.

It includes Development Tools to provide an environment to develop your Java programs and JRE to execute

your java code.

In order to create, compile and run Java program you would need JDK installed on your computer.

### ****JVM vs JRE vs JDK****

| **Component** | **Description** |
| --- | --- |
| **JVM (Java Virtual Machine)** | Runs Java bytecode, converts it to machine code |
| **JRE (Java Runtime Environment)** | Contains JVM + libraries required to run Java programs |
| **JDK (Java Development Kit)** | Contains JRE + development tools (compiler, debugger) |

### ****JVM is Platform-Independent****

Java code can run on **any OS (Windows, Mac, Linux, etc.)** because the JVM is different for each platform, but the bytecode remains the same. This is what makes Java **"Write Once, Run Anywhere" (WORA).**

### ****Example: JVM Execution Flow****

public class HelloWorld {

public static void main(String[] args) {

System.out.println("Hello, JVM!");

}

}

* **Compilation:** javac HelloWorld.java → Produces HelloWorld.class (bytecode).
* **Execution:** java HelloWorld → JVM loads and runs bytecode.

### finalize ****Keyword in Java****

#### **Definition:**

The finalize method in Java is used to perform **cleanup operations** before an object is **garbage collected**. It is a **protected method** of the Object class, and it can be overridden in a class.

### ****How**** finalize() ****Works?****

* The **Garbage Collector (GC)** calls finalize() just before destroying an object.
* It is used for **closing resources** like files, database connections, etc.
* It **does not guarantee** immediate execution, as GC runs at an unpredictable time.

### ****Example of**** finalize()

class Example {

// Constructor

Example() {

System.out.println("Object Created");

}

// Overriding finalize() method

@Override

protected void finalize() {

System.out.println("Object is being garbage collected!");

}

}

public class Main {

public static void main(String[] args) {

Example obj1 = new Example();

Example obj2 = new Example();

obj1 = null; // Eligible for garbage collection

obj2 = null; // Eligible for garbage collection

// Requesting JVM to run Garbage Collector

System.gc();

System.out.println("Main method ends");

}

}

### ****Possible Output:****

Object Created

Object Created

Main method ends

Object is being garbage collected!

Object is being garbage collected!

(Note: The order of garbage collection messages may vary.)

### ****Key Points About**** finalize()

1. **Called by Garbage Collector (GC)** before removing an object.
2. **Used for resource cleanup**, but not a reliable method for closing resources.
3. **Deprecated in Java 9** because it is unpredictable and inefficient.
4. **Alternatives:** Use try-with-resources or finally blocks for proper resource management.

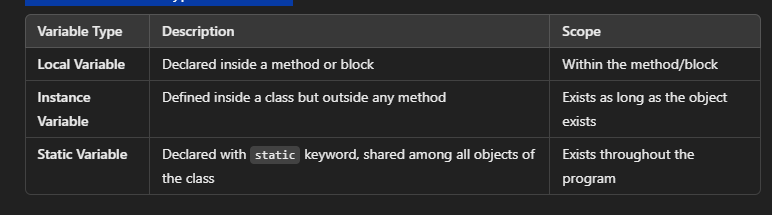
### ****Variables in Java****

#### **Definition:**

A **variable** in Java is a container that holds a value that can change during the execution of a program. Every variable has a **type**, **name**, and **scope**.

## **1. Types of Variables in Java**

Java has three main types of variables:



**Local Variables**

* Declared inside a **method, constructor, or block**.
* Only accessible within that method/block.
* **Must be initialized before use** (No default value).

**Instance Variables (Non-Static Variables)**

* Declared inside a class but **outside any method**.
* Each **object has its own copy** of instance variables.
* **Automatically initialized** with default values.

**Static Variables (Class Variables)**

* Declared with the **static keyword** inside a class.
* **Shared among all objects** of the class.
* **Memory is allocated only once** (when the class is loaded).

Naming Convention:

 Start with a **letter**, \_ (underscore), or $ (dollar sign).

 Use **letters, digits, \_, $** after the first character.

 Case-sensitive (age and Age are different).

**Operator in JAVA**

In Java, operators are used for evaluation of expressions.

**Operator** in java is a **symbol** that is used to perform operations.

Java supports the following types of operators:

1. Unary Operator (++,--)

2. Arithmetic Operator (+,-,\*,%,/)

3. Shift Operator (<<,>>)

4. Relational Operator (OR,AND)

5. Bitwise Operator (|,&)

6. Logical Operator (<=.<=,==,!=)

7. Ternary Operator(cond:True:false)

8. Assignment Operator(=)

**Arrays in JAVA**

•Array is an object

An array is a collection of similar types of elements.

• An array is a container that holds data of one single type.

For example, you can create an array that can hold

100 values of int type.

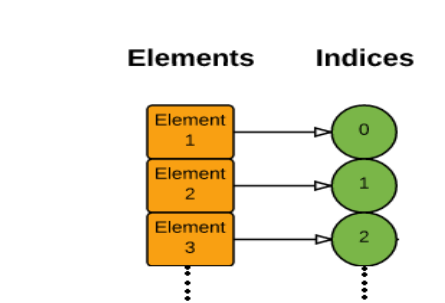
To make it more clear, a pictorial representation of array elements with their corresponding index values is

shown below.

•

The first element of the array is stored at the 0th index, second element of the array is stored at 1st index and

so on as shown in given figure.



**Array Syntax**

**We can create Array in 2 way**

**1.Dyanamic (literal way)**

•In Java, an array is an object of a dynamically generated class.

• Once the array is created, its length is fixed.

• Syntax to define array :

**data type [ ] array name;**

• **Example : int [ ] num;**

num = new int[10];

• Here, **num** array can hold 10 values of data type int.

**2.Static(using new keyword)**

**Syntax**

datatype arrayname[]=new datatype[size]

int[ ] num = new int[10];

value will be assign at the time of execution

new keyword using for assign the size of the array

There are two types of array in JAVA

* 1. Single Dimensional Array :
* A **one-dimensional (1D) array** is a simple list of elements stored in contiguous memory locations.

It have only one row

Ex: public class OneDArray { public static void main(String[] args) { int[] numbers = {10, 20, 30, 40, 50}; System.out.println("First Element: " + numbers[0]); // Output: 10 } }

2. Multi Dimensional Array :

A **two-dimensional (2D) array** is a matrix-like structure with rows and columns.

Having multiple rows and column

int[][] matrix = new int[3][3];

// 3x3 matrix

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

one more classification is there in Multidiamentinal aray-🡪Jagged Array

A **jagged array** is a 2D array where the number of columns in each row **can be different**.

int[][] jagged = new int[3][];

jagged[0] = new int[]{1, 2};

jagged[1] = new int[]{3, 4, 5};

jagged[2] = new int[]{6};

**Cloning of Arrays**

• Cloning creates copies that are clones of the original elements or reference elements.

• Cloning arrays are of two types shallow copy and deep copy in Java.

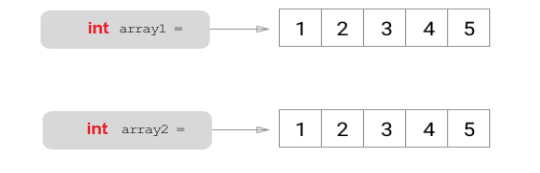
• In a single-dimensional array, a deep copy creates the clones of the original elements or reference elements.

• In a multi-dimensional array, a shallow copy is created, which means both arrays are pointing to the same memory address.

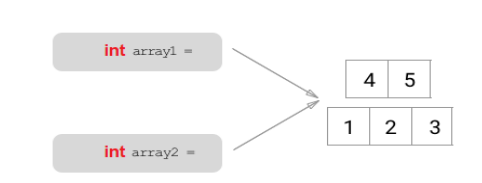
• Cloning shallow copy and deep copy in Java are the ways of copying the attributes of one object into another of the same type.

these arrays, you will be modifying both array

Deep Copy:



Shallow Copy:



STRING:

A **string** in Java is a **sequence of characters** stored as an **object** of the String class. Unlike primitive data types (int, char, etc.), a string is an **immutable object**, meaning (once created, it cannot be changed.)

There are two ways to create a String in Java

1. **String literal**

2. **Using new keyword**

1. **String literal :**

Java String literal is created by using double-quotes.

**Example**:

**String s = “king";**

String objects are stored in a special memory area known as the "string constant pool".

This is the most common way of creating the string.

2. **Using new keyword:**

String object can be created using new operator like java class.

**Example:**

**String s = new String(“king”);**

It creates two objects in String pool and in heap

Also one reference variable ‘s’ is created that will refer to the object in the heap.

**Java String Pool:** Java String pool refers to collection of Strings which are stored in heap memory. So whenever a new

object is created. It will check whether the new object is already present in the pool or not. If it is present, then same

reference is returned to the variable else new object will be created in the String pool and the respective reference

will be returned.

Here is a list of **String** methods in Java:

1. length()
2. toUpperCase()
3. toLowerCase()
4. concat()
5. substring(int beginIndex)
6. substring(int beginIndex, int endIndex)
7. contains(String str)
8. equals(String anotherString)
9. equalsIgnoreCase(String anotherString)
10. startsWith(String prefix)
11. endsWith(String suffix)
12. replace(char oldChar, char newChar)
13. replace(CharSequence target, CharSequence replacement)
14. split(String regex)
15. split(String regex, int limit)
16. trim()
17. indexOf(String str)
18. indexOf(String str, int fromIndex)
19. lastIndexOf(String str)
20. lastIndexOf(String str, int fromIndex)
21. charAt(int index)
22. isEmpty()
23. isBlank() *(Java 11+)*
24. toCharArray()
25. format(String format, Object... args)
26. valueOf(Object obj)
27. matches(String regex)
28. compareTo(String anotherString)
29. compareToIgnoreCase(String anotherString)
30. repeat(int count) *(Java 11+)*
31. strip() *(Java 11+)*

## **1. What are StringBuffer and StringBuilder?**

Both StringBuffer and StringBuilder are **mutable** classes used for modifying strings dynamically. Unlike String, which is **immutable**, these classes allow modification without creating new objects, making them **faster and memory-efficient** for string manipulation.

ENUM:

An **enum** (short for enumeration) is a special data type in Java that represents a fixed set of constants. It is used to define a collection of predefined values, making the code more readable and preventing invalid values.

### ****2. How to Declare an Enum?****

Enums are declared using the enum keyword. Each value in an enum is a constant and written in uppercase by convention.

# **Control Flow in Java**

Control flow in Java determines the execution order of statements in a program. Java provides three types of control flow statements:

1. **Decision-Making Statements (Conditional Statements)**
2. **Looping Statements (Iteration Statements)**
3. **Branching Statements (Jump Statements)**

### ****1.1 if Statement****

Executes a block of code if the condition is true.

int num = 10;

if (num > 0) {

System.out.println("Positive number");

}

### ****1.2 if-else Statement****

Executes different blocks based on whether the condition is true or false.

int num = -5;

if (num > 0) {

System.out.println("Positive number");

} else {

System.out.println("Negative number");

}

### ****1.3 if-else if Ladder****

Checks multiple conditions sequentially.

int num = 0;

if (num > 0) {

System.out.println("Positive number");

} else if (num < 0) {

System.out.println("Negative number");

} else {

System.out.println("Zero");

}

### ****1.4 Nested if Statement****

An if statement inside another if.

int age = 20;

int weight = 55;

if (age > 18) {

if (weight > 50) {

System.out.println("Eligible to donate blood");

}

}

### ****1.5 switch Statement****

Used when multiple conditions are checked for a single variable.

int day = 3;

switch (day) {

case 1: System.out.println("Sunday"); break;

case 2: System.out.println("Monday"); break;

case 3: System.out.println("Tuesday"); break;

default: System.out.println("Invalid day");

}

## ****2. Looping Statements (Iteration)****

Used to repeat a block of code multiple times.

### ****2.1 for Loop****

Executes a block multiple times.if the number of iteration is fixed we can use For loop

1.Initialization

2.Condition

3.Increment or Decrement

Ex:

for (int i = 1; i <= 5; i++) {

System.out.println(i);

}

### ****2.2 Enhanced for Loop (for-each)****

### ****To iterate Array Elements****

int[] numbers = {10, 20, 30, 40};

for (int num : numbers) {

System.out.println(num);

}

### ****2.3 while Loop****

Executes a block while a condition is true. if the number of iteration is not fixed we can use While or Do while

int i = 1;

while (i <= 5) {

System.out.println(i);

i++;

}

### ****2.4 do-while Loop****

Executes at least once, then checks the condition.

int i = 1;

do {

System.out.println(i);

i++;

} while (i <= 5);

## **3. Branching Statements (Jump Statements)**

Used to change the normal flow of execution.

### ****3.1 break Statement****

Exits a loop or switch statement.

for (int i = 1; i <= 5; i++) {

if (i == 3) {

break; // Exits loop when i == 3

}

System.out.println(i);

}

### ****3.2 continue Statement****

Skips the current iteration and moves to the next.

for (int i = 1; i <= 5; i++) {

if (i == 3) {

continue; // Skips printing 3

}

System.out.println(i);

}

### ****3.3 return Statement****

Exits from a method.

public class ReturnExample {

public static void main(String[] args) {

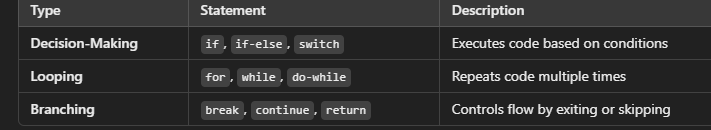
System.out.println("Before return");

return;

// System.out.println("After return"); // This will not be executed

}

}



Methods:

**Methods :** In Java, a method is like a function, runs when it calls. Methods can be called anywhere in the program.

Advantages:-

1. **Code reusability** methods means we can use code many times by declaring only once.

2. **Code optimization** methods means we can reduce the line of code.

• Methods should be declared inside the class.

• It should be defined by method name followed by parenthesis ().

• Some predefined methods in java, is **System.out.println**(), etc.

• A syntax for creating method is shown below.

Access modifier return Type method name()

{

}

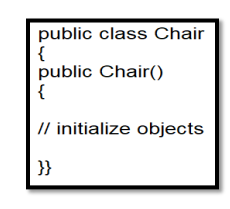
Ex:public int show()

{

}

**Constructor:**

**Constructor:** It is a block of codes similar to the method. It is a special type of method. Every class contains a minimum of one constructor. Constructor name must be the same as its class name.



Difference between Constructor and Method:

1.Constructor name is same as Class name where method name Can be anything

2.Method is having return type but Constructor Doesn’t have return type

### ****6. Object-Oriented Programming (OOP) in Java****

* **Class and Object**

class Car {

String brand = "Toyota";

}

public class Main {

public static void main(String[] args) {

Car myCar = new Car(); // Creating an object

System.out.println(myCar.brand);

}

}

* **Encapsulation:** Data hiding using private variables and getters/setters.
* **Inheritance:** One class inherits from another using extends.
* **Polymorphism:** Same method behaving differently in different classes.
* **Abstraction:** Hiding details using abstract classes or interfaces.

### ****Classes in Java****

In Java, **a class is a blueprint** for creating objects. It defines variables (fields) and methods (functions) that objects will have.

## **1. Defining a Class**

A class in Java is defined using the class keyword.

### ****Example of a Simple Class****

class Car {

// Fields (Variables)

String brand = "Toyota";

int speed = 120;

// Method (Function)

void displayCarInfo() {

System.out.println("Brand: " + brand);

System.out.println("Speed: " + speed + " km/h");

}

}

## **2. Creating Objects from a Class**

An object is an instance of a class. You create objects using the new keyword.

public class Main {

public static void main(String[] args) {

Car myCar = new Car(); // Creating an object

myCar.displayCarInfo(); // Calling a method

}

}

**Output:**

Brand: Toyota

Speed: 120 km/h

## **3. Constructors in Java**

A **constructor** is a special method that is called when an object is created. It initializes the object’s properties.

### ****Example of a Constructor****

class Car {

String brand;

int speed;

// Constructor

Car(String b, int s) {

brand = b;

speed = s;

}

void displayCarInfo() {

System.out.println("Brand: " + brand);

System.out.println("Speed: " + speed + " km/h");

}

}

public class Main {

public static void main(String[] args) {

Car myCar = new Car("Honda", 150); // Creating object with constructor

myCar.displayCarInfo();

}

}

**Output:**

Brand: Honda

Speed: 150 km/h

## **4. Types of Classes in Java**

### ****a) Normal Class****

A regular class with methods and fields (like Car above).

### ****b) Abstract Class****

An abstract class **cannot be instantiated** and must be inherited. It can have abstract (without body) and non-abstract methods.

abstract class Animal {

abstract void makeSound(); // Abstract method

void sleep() {

System.out.println("Sleeping...");

}

}

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();

d.sleep();

}

}

**Output:**

Bark!

Sleeping...

### ****c) Final Class****

A final class **cannot be extended (inherited)**.

final class Vehicle {

void drive() {

System.out.println("Driving...");

}

}

// This will cause an error

// class Car extends Vehicle { }

public class Main {

public static void main(String[] args) {

Vehicle v = new Vehicle();

v.drive();

}

}

## **5. Inheritance (Extending a Class)**

Inheritance allows a class to inherit properties and methods from another class using the extends keyword.

class Animal {

void eat() {

System.out.println("This animal eats food.");

}

}

class Dog extends Animal {

void bark() {

System.out.println("The dog barks.");

}

}

public class Main {

public static void main(String[] args) {

Dog myDog = new Dog();

myDog.eat(); // Inherited method

myDog.bark(); // Own method

}

}

**Output:**

This animal eats food.

The dog barks.

## **6. Encapsulation (Using Private Variables)**

Encapsulation is **hiding data** by making variables private and accessing them using getter and setter methods.

class Person {

private String name;

// Setter Method

void setName(String newName) {

name = newName;

}

// Getter Method

String getName() {

return name;

}

}

public class Main {

public static void main(String[] args) {

Person p = new Person();

p.setName("Alice");

System.out.println("Name: " + p.getName());

}

}

**Output:**

Name: Alice

## **7. Static Members in a Class**

* static members **belong to the class itself, not to any object**.
* A static method can be called without creating an object.

class MathUtils {

static int square(int x) {

return x \* x;

}

}

public class Main {

public static void main(String[] args) {

System.out.println(MathUtils.square(5)); // No object needed

}

}

**Output:**

25

## **Summary**

| **Feature** | **Description** |
| --- | --- |
| **Class** | Blueprint for creating objects |
| **Object** | Instance of a class |
| **Constructor** | Initializes an object when created |
| **Inheritance** | Allows a class to derive properties from another class |
| **Encapsulation** | Hides data using private variables |
| **Static Members** | Belong to the class, not to instances |

## **2. Non-Primitive Data Types**

These are more complex data types that refer to objects.

| **Data Type** | **Description** |
| --- | --- |
| String | Represents a sequence of characters |
| Array | Collection of elements of the same type |
| Class | Blueprint for creating objects |
| Interface | Defines a contract for classes |
| Enum | Represents a fixed set of constants |

### ****Example Usage****

public class NonPrimitiveExample {

public static void main(String[] args) {

String name = "Java";

int[] numbers = {10, 20, 30}; // Array

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

System.out.println("First number: " + numbers[0]);

}

}

**Output:**

Name: Java

First number: 10

### ****Key Differences: Primitive vs Non-Primitive Data Types****

| **Feature** | **Primitive Data Types** | **Non-Primitive Data Types** |
| --- | --- | --- |
| **Memory Efficiency** | More efficient | Less efficient (Objects require more memory) |
| **Stored In** | Stack memory | Heap memory |
| **Examples** | int, char, boolean | String, Array, Class |
| **Operations** | Direct operations | Methods required |

### ****Better Alternative to**** finalize()****: Try-With-Resources****

Instead of finalize(), **use try-with-resources** to close resources explicitly:

import java.io.\*;

public class TryWithResourcesExample {

public static void main(String[] args) {

try (FileWriter file = new FileWriter("test.txt")) {

file.write("Hello, World!");

} catch (Exception e) {

System.out.println(e);

}

}

}

### ****Summary****

| **Feature** | **Description** |
| --- | --- |
| **finalize() Purpose** | Cleanup before garbage collection |
| **Who Calls It?** | JVM Garbage Collector |
| **Can We Call It Manually?** | Yes, using finalize(), but not recommended |
| **Is It Reliable?** | No, GC timing is unpredictable |
| **Java 9 Status** | Deprecated |
| **Alternatives** | Try-with-resources, finally block |

### ****7. Constant Variables (****final ****Keyword)****

To make a variable **unchangeable**, use the final keyword.

#### **Example of Final Variable**

java

CopyEdit

class Example {

final int MAX\_VALUE = 100; // Constant variable

void display() {

// MAX\_VALUE = 200; // ❌ Error (Cannot change final variable)

System.out.println("Max Value: " + MAX\_VALUE);

}

}

### ****Summary****

| **Concept** | **Description** |
| --- | --- |
| **Local Variable** | Declared inside a method/block, must be initialized |
| **Instance Variable** | Defined in a class, belongs to objects, gets default values |
| **Static Variable** | Shared among all objects, belongs to class |
| **Final Variable** | Cannot be changed after assignment |

**Packages in java**

Package is a kind of bundle or container or library, where we put one or more Java classes, interfaces, and other related entities/information.

It means bundling the multiple related program files at one place. Package is the first statement of any Java program.

Packages can be categorized into two categories,

1.built in package

2.user defined package.

**Built-in packages:** The already predefined package by the java compiler is known as built-in packages.

Some of the commonly used built-in packages in java are as follows:

-

**java.lang:** It contains language support classes.

-

**java.util:** It contains utility classes such as vectors, lists, hash tables, etc.

-

**java.awt:** It contains classes for the graphic user interface.

-

**java.applet:** It contains a set of classes for applets.

-

**java.net:** It contains a set of network classes.

-

**java.io:** It contains classes for input and output operation.

•

**2.User defined packages :** These are the packages that are defined by the user. Now we will see how the

packages are created and used in java.

Select a suitable name for the package to be created.

-

Name of the package must be same as the directory under which this file is saved.

-

Declare the name of the package with the “**package**” keyword.

-

Define a public class inside that package.

**Java Access Modifiers**

Java access modifiers are used to provide access control in java.

Access modifiers are used with Classes as well as Class variables and methods.

It is allowed to use only public or default access modifiers with java classes.

Java provides three types of access control through Keywords

1. Private

2. Protected

3. Public

Access modifier is a keyword that we use to set the visibility or scope or define the boundary of variable,

method, and class.

This is also known as specifier.

Default is the default access modifier when we do not write any modifier with class declaration.

1.Default modifier makes a class accessible within the same package

2.“**private**” then it will be accessible only inside the same class. This is the most restricted

access and the class member will not be visible to the outer world.

3. **public**” then it can be accessed from anywhere. Also member variable or method is

accessed globally.

4. If class member is “**protected**” then it will be accessible only to the classes in the same package and to

the subclasses.

**Super keyword**

The super() keyword should always be the first statement of the constructor.

The super keyword is used to call the constructor from a super class.

This super() will call the default constructor from the super class.

**this keyword**

The **this** keyword represents the members (for example, variables or methods) from same class.

**this** works in a similar manner as super keyword.

In the constructor, we use the **this()** keyword to call or refer a constructor, which is defined within the same class. this should be the first statement of a constructor. If you write **this()** as second

statement or later, then the system will generate a compilation error.

A class can have any number of constructors and with the help of the **this()** keyword, a constructor can

communicate with other constructors within the class.

It will avoid Name collision

**Method Overloading**

It allows the class to have more than one method having the same name, if their argument lists are different.

It is not possible by changing the return type of methods.

In order to overload a method, the argument lists of the methods must differ in either of the following:

1. **Number of argument :**

It is allowed within the class given that the number of arguments

are not the same.

**Example :**

**Int max(int, int)** // 2 argument

**Int max (int, int, int)** // 3 argument

2. **Data type of argument:**

It is allowed within the class given that at least one pair of arguments are of different data type.

**Example :**

**max (int, int)** // same data type

**max(int, float)** // different data type

**Data type of argument :** We have two methods with the name **max**(), one with argument of int type

and another method with the argument of int & float type.

OOPS:

**Method overriding & overridden**

Declaring a method in sub class which is already present in parent class is known as method overriding.

Overriding is done so that a child class can give its own implementation to a method which is already provided by the parent class.

In this case the method in parent class is called overridden method and the method in child class is called overriding method.

A method declared in **child class** but it is already present in the **parent class** is known as method overriding.

•

The method declared in the parent class is called overridden method and the method in the child class is

called the overriding method.

•

Method overriding is used for runtime polymorphism.

•

A method declared static cannot be overridden but can be redeclared.

# **Encapsulation in Java**

## **1. What is Encapsulation?**

Encapsulation is one of the four fundamental principles of **Object-Oriented Programming (OOP)** in Java. It is the technique of **hiding the internal details of a class** and allowing access only through well-defined methods.

### ****Key Features of Encapsulation:****

✅ **Data Hiding** – Prevents direct access to instance variables.  
✅ **Controlled Access** – Uses getter and setter methods for data manipulation.  
✅ **Improved Security** – Restricts unauthorized modifications.  
✅ **Easier Maintenance** – Code changes are localized within the class.

## **2. How to Achieve Encapsulation in Java?**

Encapsulation is implemented by:

1. **Declaring instance variables as private** (to restrict direct access).
2. **Providing public getter and setter methods** (to allow controlled access).

### ****Example of Encapsulation****

class Person {

private String name; // Private variable

private int age; // Private variable

// Getter method for name

public String getName() {

return name;

}

// Setter method for name

public void setName(String name) {

this.name = name;

}

// Getter method for age

public int getAge() {

return age;

}

// Setter method for age

public void setAge(int age) {

if (age > 0) { // Adding validation

this.age = age;

} else {

System.out.println("Age must be positive.");

}

}

}

public class EncapsulationExample {

public static void main(String[] args) {

Person p = new Person();

p.setName("John Doe");

p.setAge(25);

System.out.println("Name: " + p.getName());

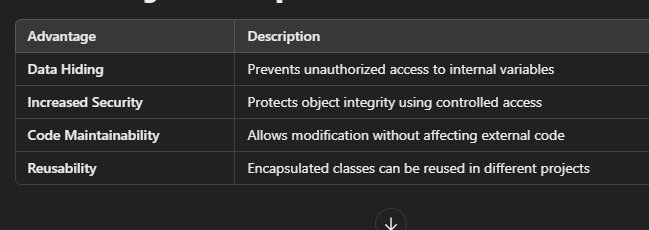
System.out.println("Age: " + p.getAge());

}

}

**Output:**

makefile



**Abstraction**

Abstraction is a process where you show only “relevant functional details” and “hide irrelevant details” of an object from the user.

Abstraction reduces the complexity of the view of objects to the user.

It increases security as information is kept hidden.

Abstraction is one of the most important features of oops.

Abstraction is a process of hiding the implementation details and only showing the functionality to the user.

Abstraction is created using abstract classes and interfaces.

There are two ways to achieve abstraction in java are as follows:

1. Abstract class

2. Interface

**Abstract class**

A class that is declared using “**abstract**” keyword is known as abstract class.

An Abstract class is a class that contains abstract as well as concrete methods.

We cannot create the object of an abstract class.

An abstract class can have static methods and constructors.

A class derived from the abstract class must implement all those methods that are declared as abstract in the parent class.

An abstract class can have an abstract and non abstract method.

(abstract method-🡪no body, No Abstract method->having body))

A class can not be declared with both final and abstract keywords, because final keyword is used to prevent overriding whereas abstract methods need to be overridden.

An abstract method does not have implementation and body.

An abstract method does not have implementation and body. It defines only the signature of the method

Interface:

An **interface** in Java is a blueprint of a class that contains **only abstract methods (until Java 7)** and **default, static methods (from Java 8)**. It is used to achieve **full abstraction** and **multiple inheritance and Hybrid**  in Java.

### ****Key Features of an Interface:****

✅ **Defines only method signatures (no implementation in Java 7 and earlier)**  
✅ **Implemented by classes using the implements keyword**  
✅ **Supports multiple inheritance (a class can implement multiple interfaces)**  
✅ **Cannot be instantiated (like abstract classes)**  
✅ **Can contain default and static methods (Java 8+ only)**

Ex:

interface Animal

{

void eat(); // Abstract method (no body)

void sleep();

}

**Functional Interface:interface having only 1 method**

Marker Interface (Interface with No Methods)

**Inheritance**

The process in which one class (object) acquires all the properties and behaviors (fields and methods) of

another class (parent) is known as inheritance.

When we inherit from an existing class, we can reuse fields and methods of the parent class.

We can declare new fields in the subclass that are not in the superclass.

In real life, a child inherits the properties from his father just like that inheritance represents a parent child

relationship which is also known as IS-A relationship.

In inheritance “extends” keyword is used to inherit a class. (The extends keyword defines that we are making a new class that is derived from an existing class.

)

Use “implements” to inherit interface

HAS-A simply mean the use of instance variables that are references to other objects. For example, Maruti has Engine, or House has Bathroom.

Some terms used in Inheritance are :

**Class :** It is a blueprint from which objects are created.

**Sub Class/Child Class:** It is a class which inherits the other class. It is also called a derived class,

extended class, or child class.

**Super Class/Parent Class:** The class whose properties and functionalities are used (inherited) by

another class is called as base class or a parent class.

**Reusability:** We can use the same fields and methods already defined in the previous class.

**Extends keyword :** It indicates that you are making a new class that derives from an existing class.

**Super keyword :** It is used to access methods of the parent class.

**This keyword :** It is used to access methods of the current class.

**Types of Inheritance :**

There are five types of Java Inheritance.

1. Single Inheritance

2. Multilevel Inheritance

3. Hierarchical Inheritance

4. Multiple Inheritance

5. Hybrid Inheritance

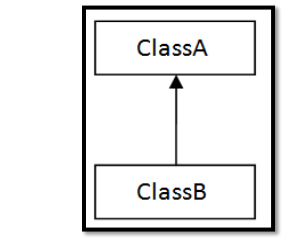
**Note:** Java does not support Multiple and Hybrid Inheritance with classes. Multiple and Hybrid inheritance can

only be achieved only through Interfaces.

**Single Inheritance**

**Single Inheritance :** In single inheritance, the features and methods of the parent class are inherited by a

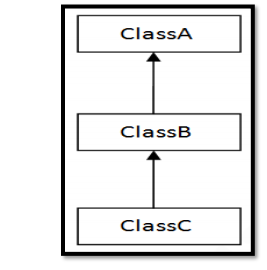
single child class.



**Multilevel Inheritance**

•

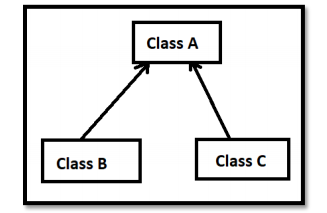
In multilevel inheritance, a child class will be inheriting a parent class as well as the child class, and also parent class for some other class.



**Hierarchical Inheritance**

**Hierarchical Inheritance :** In hierarchical inheritance, more than one subclasses inherit the properties,

behavior, features and methods from a single parent class.

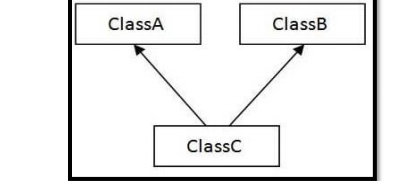


**Multiple Inheritance**

•

**Multiple Inheritance :** In Multiple inheritance one class can have more than one superclass and inherit

properties, behavior, features and methods from all parent classes.



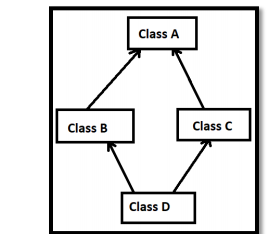
**Hybrid Inheritance**

•

**Hybrid Inheritance :** Hybrid Inheritance is a combination of both Single Inheritance and Multiple Inheritance.

•

Since in Java Multiple Inheritance is not supported directly we can achieve Hybrid inheritance also through Interfaces only.



**Polymorphism in Java**

It is the process of representing one form in multiple forms known as **Polymorphism,** in simple words it is the **OOPs** feature that allows to perform a single action but in different ways.

Polymorphism is derived from two greek words **poly** and **morphs**.

The word “**poly**” means many and “**morphs**” means forms.

So polymorphism means many forms.

There are mainly two types of polymorphism in Java as shown below:

1.compile-time polymorphism

-2.Runtime polymorphism

**Compile time Polymorphism /Early Binding/Static (Overloading)**

Compile time polymorphism is also known as static polymorphism or early binding.

Static polymorphism in java is achieved by method or function or operator overloading.

When there are multiple functions with same name but different arguments then these functions are said to be overloaded.

In order to overload a method, the argument lists of the methods must differ in either of the following:

Method Overloading allows to have more than one method having the same name, if the arguments of

methods are different in number, sequence and data types of parameters.

It is a compile time process. :

The method definition and method call are linked during the compile time.

Actual object is not used for binding.

Program execution is faster.

The binding of static, private and final methods are done at compile time

**Runtime Polymorphism**

Dynamic polymorphism is a process in which a call to an overridden method is resolved at runtime, thats why

it is called runtime polymorphism.

Runtime polymorphism is also known as dynamic or late binding.

Runtime polymorphism in java is achieved by method overriding.

Overriding allows a child class to implement a method that is already provided by one of its parent class.

**Late Binding**

•

It is a run time process.

•

The method definition and method call are linked during the run time.

•

Actual object is used for binding.

•

For example: Method overriding.

•

Program execution is slower.

•

The binding of static, private and final methods are not done.

# **Exception:**

An Exception is an unwanted event that interrupts the normal flow of the program.

•

Errors are generated while writing a programming code.

•

So these errors are displayed at compile time.

•

Some of these errors do not show up at compile time but interrupts the normal flow of execution at run time.

•

These errors are known as Exceptions in programming.

•

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

•

This is something that every programmer faces at any point of coding. They can occur from different kind of

scenarios like entering the wrong data by user, hardware failure, network failure, class not found, out of

memory, etc.

•

Exception Handling is a mechanism to handle runtime errors.

•

The main advantage of exception handling is to maintain the normal flow of the application.

•

All exception and errors types are sub classes of class **Throwable.**

•

Suppose if an exception is not handled, it may lead to a system failure. That is why handling an exception is very important.

**Java Exception Keywords**

•

Below 5 keywords are used to handle exceptions in Java.

1. try

2. catch

3. finally

4. throw

## 5. throws**1. What is Exception Handling?**

**Exception handling** in Java is a mechanism that handles runtime errors, ensuring that the normal flow of a program is maintained. When an exception occurs, Java creates an exception object that contains details about the error, and the program can **catch** and **handle** it appropriately.

### ****Key Concepts:****

✅ **Exception** → An unexpected event that disrupts program flow.  
✅ **Try-Catch Block** → Handles exceptions gracefully.  
✅ **Finally Block** → Always executes, even if an exception occurs.  
✅ **Throws & Throw** → Used for explicitly throwing exceptions.  
✅ **Custom Exceptions** → User-defined exceptions for specific scenarios.

## **2. Types of Exceptions in Java**

Java exceptions are categorized into **Checked**, **Unchecked**, and **Errors**.

### ****2.1 Checked Exceptions (Compile-Time Exceptions)****

* Occur at compile time and must be handled using try-catch or throws.
* Examples: IOException, SQLException, FileNotFoundException.

**Throw**:throw is used to explicitily create and throw an instance of an exception.it will handle a specific error

Used within in the method or block of code

Can only throw one exceptio at a time

Throw throws an exception

**Throws:**

It is used in method signature to declare that method might throw one or more exception

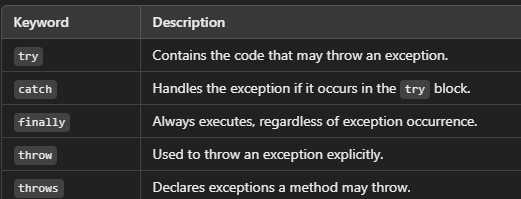
Throws ****declares that a method might throw an exception****

### ****2.2 Unchecked Exceptions (Runtime Exceptions)****

* Occur at runtime and do not require explicit handling.
* Examples: NullPointerException, ArrayIndexOutOfBoundsException, ArithmeticException.

### ****2.3 Errors****

* Occur due to system failures and cannot be handled.
* Examples: OutOfMemoryError, StackOverflowError.



## **Creating Custom Exceptions**

You can create **your own exceptions** by extending the Exception class.

**Java Collections Framework**

Collections framework provides a set of interfaces and classes to implement various data structures and

algorithms.

•

Collections framework is contained in java.util package.

•

It allows the programmers to program at the interfaces, instead of the actual implementation.

•

A well designed framework can improve your productivity and provide ease of maintenance.

•

The Java Collection Framework package (java.util) contains:

1. A set of interfaces

2. Implementation classes

3. Algorithms (like sorting and searching)

•

Java Collections are similar to containers that consists of multiple items in a single unit. for e,g. collections of

books, list of names etc.

•

Collections framework provides unified architecture for manipulating and representing collections.

•

Java collection framework can perform following activity :

-

Add objects to collection

-

Remove objects from collection

-

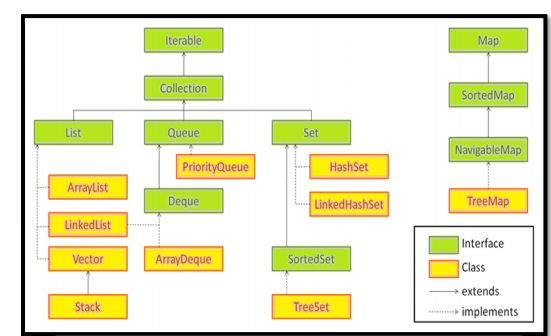
Search for an object in collection

-

Retrieve/get object from collection

•

Collection framework contains different types of collections such as lists, sets, maps, stacks, queues, etc.



**Multithreading in Java:**

**Multithreading** is a Java feature that allows **concurrent execution** of two or more threads to maximize CPU utilization. It helps in **parallel processing** by running multiple tasks simultaneously.

### ****Key Features of Multithreading in Java****

✅ **Threads share a common memory space** (efficient memory usage)  
✅ **Independent execution** (one thread's failure doesn’t affect others)  
✅ **Better performance** (especially in multi-core processors)  
✅ **Useful for tasks like gaming, web servers, and real-time applications**

**Threads in JAVA**

•

Thread is the smallest execution unit of a process and a process may have many threads that are executing at he same time.

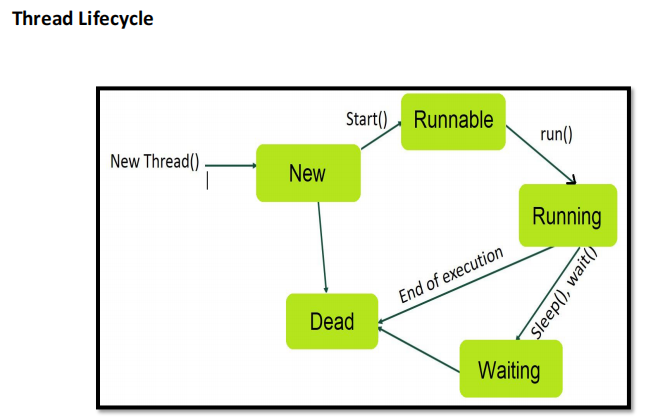
•

Thread has its own execution path within the process and shares the memory of the process with other

threads, which are running in the same process.

•

Thread doesn’t allocate any memory, but it uses the memory allocated by its process; this helps faster and efficient communication between threads within the same process



**Thread Synchronization**

•

Synchronization is a keyword in the Java programming language that facilitates the programmer to control

threads that are sharing data.

**DeadLock**

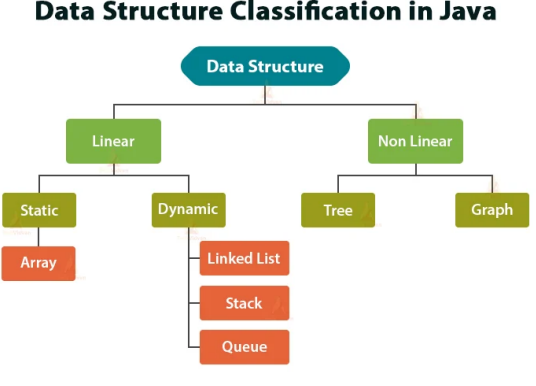
•

Deadlock in Java is a part of multithreading.

•

Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another

thread and second thread is waiting for an object lock that is acquired by first thread.



Linear Data Structures – The elements arranged in a linear fashion are called Linear Data Structures. Here, each element is connected to one other element only. The linear data structure is a single level data structure. Linear Data Structures are as follows:

● Arrays

● Single dimensional Array

● Multidimensional Array

● Stack

● Queue

● Linked List:● Singly-linked list ● Doubly Linked list ● Circular Linked List

Non-Linear Data Structures: The non-linear data structure does not arrange the data in a sequential manner as in linear data structures. Non-linear data structures are the multilevel data structure. Examples of non linear data structures are - Tree and Graph. Here each node can be connected to multiple nodes

## ****Stack****

A **Stack** is a **linear data structure** that follows the **LIFO** principle:

**Last In, First Out**  
The element that is **added last** will be the **first** to be **removed**.

## 🔹 Real-Time Examples:

Undo/Redo functionality in editors

Browser history

Call stack in recursion

Backtracking algorithms (e.g., maze solver)

| **Operation** | **Description** |
| --- | --- |
| push() | Add item to top of stack |
| pop() | Remove item from top |
| peek() | View item at top without removing |
| isEmpty() | Check if stack is empty |

Ex:

import java.util.Stack;

public class StackExample {

public static void main(String[] args) {

Stack<String> stack = new Stack<>();

// Push elements

stack.push("Java");

stack.push("Python");

stack.push("C++");

// Stack: [Java, Python, C++]

System.out.println("Top element: " + stack.peek()); // C++

// Pop element

String removed = stack.pop(); // Removes C++

System.out.println("Removed: " + removed);

// Check if empty

System.out.println("Is stack empty? " + stack.isEmpty());

// Print all elements

System.out.println("Current Stack: " + stack); // [Java, Python]

}

}

**Stack creation using Array :**

class MyStack {

private int[] stack;

private int top;

private int capacity;

public MyStack(int size) {

stack = new int[size];

capacity = size;

top = -1;

}

public void push(int item) {

if (top == capacity - 1) {

System.out.println("Stack Overflow");

return;

}

stack[++top] = item;

}

public int pop() {

if (top == -1) {

System.out.println("Stack Underflow");

return -1;

}

return stack[top--];

}

public int peek() {

return stack[top];

}

public boolean isEmpty() {

return top == -1;

}

}

**Reversing a string using stack**

public class ReverseStringUsingStack {

public static void main(String[] args) {

String input = "hello";

Stack<Character> stack = new Stack<>();

for (char c : input.toCharArray()) {

stack.push(c);

}

StringBuilder reversed = new StringBuilder();

while (!stack.isEmpty()) {

reversed.append(stack.pop());

}

System.out.println("Reversed: " + reversed.toString()); // "olleh"

}

}

## **🔗 **LinkedList in Data Structures****

A **Linked List** is a linear data structure where each element (called a **node**) contains:

**1.Data**

**2.Reference to the next node** (and optionally the previous one)

Unlike arrays, linked lists do not use contiguous memory.

## 🧱 **Types of LinkedList**

### ****Singly Linked List****

Each node has:

1.data

2.next → points to the next node  
🔁 Traversal only in one direction( Only **forward)**

**Ex:**

### **head ➝ [10| ] ➝ [20| ] ➝ [30| ] ➝ null**

### ****2.Doubly Linked List****

Each node has:

1.data

2.prev → points to the previous node

3.next → points to the next node  
🔁 Traversal in both directions(📌**forward& Backward)**

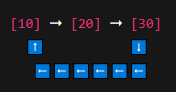
### 

### ****3.Circular Linked List****

Last node points back to the head

Can be singly or doubly circular  
🔄 Good for circular queue, round-robin schedulers

Ex:



## ⚙️ **Basic Operations**

| **Operation** | **Description** | **Time Complexity** |
| --- | --- | --- |
| insertFirst() | Insert at the beginning | O(1) |
| insertLast() | Insert at the end | O(n) (O(1) if tail maintained) |
| deleteFirst() | Remove from the beginning | O(1) |
| deleteLast() | Remove from the end | O(n) |
| search(key) | Find node with value = key | O(n) |
| reverse() | Reverse the entire list | O(n) |
| display() | Print the list | O(n) |

## 🔨 Core Operations in a Linked List

### 1. Insert

At beginning

At end

At a specific position

### 2. Delete

From beginning

From end

From a given position

### 3. Traverse (Print elements)

**Java Built in LinkedList:**

import java.util.\*;

public class BuiltInLinkedList {

public static void main(String[] args) {

LinkedList<String> list = new LinkedList<>();

list.add("Apple");

list.add("Banana");

list.addFirst("Mango");

list.addLast("Orange");

System.out.println(list); // [Mango, Apple, Banana, Orange]

list.remove("Banana");

System.out.println(list); // [Mango, Apple, Orange]

}

}

### ****1.Singly Linked List:Ex****

class SinglyNode {

int data;

SinglyNode next;

SinglyNode(int data) {

this.data = data;

this.next = null;

}

}

class SinglyLinkedList {

SinglyNode head;

void insert(int data) {

SinglyNode newNode = new SinglyNode(data);

if (head == null) {

head = newNode;

} else {

SinglyNode temp = head;

while (temp.next != null) {

temp = temp.next;

}

temp.next = newNode;

}

}

void display() {

SinglyNode temp = head;

while (temp != null) {

System.out.print(temp.data + " -> ");

temp = temp.next;

}

System.out.println("null");

}

}

public class A {

public static void main(String[] args) {

// TODO Auto-generated method stub

// Singly Linked List

SinglyLinkedList sll = new SinglyLinkedList();

sll.insert(10); sll.insert(20); sll.insert(30);

sll.display();

}

}

**Doubly LinkedList:Ex**

package DesignPatternDemo;

class DoublyNode {

int data;

DoublyNode prev, next;

DoublyNode(int data) {

this.data = data;

this.prev = null;

this.next = null;

}

}

class DoublyLinkedList {

DoublyNode head;

void insert(int data) {

DoublyNode newNode = new DoublyNode(data);

if (head == null) {

head = newNode;

} else {

DoublyNode temp = head;

while (temp.next != null) {

temp = temp.next;

}

temp.next = newNode;

newNode.prev = temp;

}

}

void displayForward() {

DoublyNode temp = head;

System.out.print("Forward: ");

while (temp != null) {

System.out.print(temp.data + " <-> ");

temp = temp.next;

}

System.out.println("null");

}

void displayBackward() {

DoublyNode temp = head;

if (temp == null) return;

// Go to last node

while (temp.next != null) {

temp = temp.next;

}

// Print in reverse

System.out.print("Backward: ");

while (temp != null) {

System.out.print(temp.data + " <-> ");

temp = temp.prev;

}

System.out.println("null");

}

}

public class A {

public static void main(String[] args) {

DoublyLinkedList dll = new DoublyLinkedList();

dll.insert(100); dll.insert(200); dll.insert(300);

dll.displayForward();

dll.displayBackward();

}

}

**Circular:Ex**

package DesignPatternDemo;

class CircularNode {

int data;

CircularNode next;

CircularNode(int data) {

this.data = data;

this.next = null;

}

}

class CircularLinkedList {

CircularNode head = null;

CircularNode tail = null;

void insert(int data) {

CircularNode newNode = new CircularNode(data);

if (head == null) {

head = newNode;

tail = newNode;

newNode.next = head; // Circular link

} else {

tail.next = newNode;

tail = newNode;

tail.next = head; // Maintain circularity

}

}

void display() {

if (head == null) return;

CircularNode temp = head;

do {

System.out.print(temp.data + " -> ");

temp = temp.next;

} while (temp != head);

System.out.println("(back to head)");

}

}

public class A {

public static void main(String[] args) {

// Circular Linked List

CircularLinkedList cll = new CircularLinkedList();

cll.insert(1); cll.insert(2); cll.insert(3);

cll.display();

}

## ✅ What is a Queue?

A **Queue** is a **linear data structure** that follows the **FIFO (First-In-First-Out)** principle.  
The element added first is removed first.

## ✅ Java Queue Interface

In Java, the Queue is an **interface** in the java.util package. It is extended by:

LinkedList

PriorityQueue

ArrayDeque

## ✅ Common Operations

| **Operation** | **Description** |
| --- | --- |
| add() | Adds element to the queue |
| offer() | Adds element (returns false if full, used in bounded queues) |
| remove() | Removes and returns head element |
| poll() | Removes and returns head, returns null if empty |
| peek() | Returns head element without removing |
| element() | Like peek but throws exception if empty |

## ✅ Example 1: Using LinkedList as a Queue

import java.util.\*;

public class QueueExample {

public static void main(String[] args) {

Queue<Integer> queue = new LinkedList<>();

// Enqueue elements

queue.add(10);

queue.add(20);

queue.add(30);

System.out.println("Initial Queue: " + queue);

// Peek at front

System.out.println("Front element: " + queue.peek());

// Dequeue elements

System.out.println("Removed: " + queue.remove());

System.out.println("Queue after remove: " + queue);

}

}

## ✅ Example 2: Real-Time Use Case – **Customer Service Queue**

import java.util.\*;

public class CustomerServiceQueue {

public static void main(String[] args) {

Queue<String> customerQueue = new LinkedList<>();

customerQueue.add("Customer 1");

customerQueue.add("Customer 2");

customerQueue.add("Customer 3");

while (!customerQueue.isEmpty()) {

System.out.println(customerQueue.peek() + " is being served.");

customerQueue.poll(); // Remove served customer

}

}

}

## ✅ Example 3: PriorityQueue (Elements served by priority)

import java.util.\*;

public class PriorityQueueExample {

public static void main(String[] args) {

PriorityQueue<Integer> pq = new PriorityQueue<>();

pq.add(50);

pq.add(20);

pq.add(30);

pq.add(10);

System.out.println("Priority Queue (natural order):");

while (!pq.isEmpty()) {

System.out.println(pq.poll());

}

}

}

## ✅ Example 4: ArrayDeque (Faster than LinkedList, for both Queue & Stack)

import java.util.\*;

public class ArrayDequeExample {

public static void main(String[] args) {

Deque<String> deque = new ArrayDeque<>();

deque.offer("A");

deque.offer("B");

deque.offer("C");

System.out.println("Deque as Queue: " + deque);

System.out.println("Removed from front: " + deque.poll());

System.out.println("After removal: " + deque);

}

}

## ✅ When to Use Queue?

Print Job Scheduling

Customer Service Systems

CPU Task Scheduling

Call Center Ticketing

Order Processing Systems

## ****Searching Algorithms:****

Searching algorithms in **Data Structures and Algorithms (DSA)** are techniques used to find a particular element in a collection (like an array, list, etc.). In Java, two main types of searching algorithms are widely used:

1. Linear Search
2. Binary Search

**Linear Search:**

Linear Search scans every element of the collection sequentially until the target element is found or the collection ends.

### ****Steps (Step by Step):****

Start from the first element of the array.

Compare the current element with the target element.

If they match, return the index of the element.

If they don’t match, move to the next element.

Repeat until you find the element or reach the end of the array.

If not found, return -1.

## **2. Binary Search**

### ****Definition:****

Binary Search is an efficient algorithm that works on sorted arrays. It repeatedly divides the search range in half.

### ****Steps (Step by Step):****

Ensure the array is sorted.

Set two pointers:

low = 0 (start index)

high = n-1 (last index)

Find the middle element:

mid = (low + high) / 2

Compare the middle element with the target:

If arr[mid] == key, return mid.

If arr[mid] > key, search the left half (high = mid - 1).

If arr[mid] < key, search the right half (low = mid + 1).

Repeat until low > high.

If not found, return -1.

## ✅ **1. Linear Search Dry Run**

### ****Example:****

Array: {10, 20, 30, 40, 50}  
Target (key): 30

### ****Step-by-Step Execution:****

| **Step** | **Index (i)** | **arr[i]** | **Compare with Key (30)** | **Action** |
| --- | --- | --- | --- | --- |
| 1 | 0 | 10 | 10 != 30 | Move to next element |
| 2 | 1 | 20 | 20 != 30 | Move to next element |
| 3 | 2 | 30 | 30 == 30 | **Found → return index 2** |

✔️ **Output:** Element found at index 2  
🕒 **Time Complexity:** O(3) → O(n) in general.

## ✅ **2. Binary Search Dry Run**

### ****Example:****

Array: {10, 20, 30, 40, 50} (sorted)  
Target (key): 40

### ****Step-by-Step Execution:****

| **Step** | **Low** | **High** | **Mid** | **arr[mid]** | **Compare with Key (40)** | **Action** |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 4 | 2 | 30 | 30 < 40 | Search right → low = mid + 1 (3) |
| 2 | 3 | 4 | 3 | 40 | 40 == 40 | **Found → return index 3** |

✔️ **Output:** Element found at index 3  
🕒 **Time Complexity:** O(log₂5) ≈ O(3)

| **Type** | **Description** | **Time Complexity** |
| --- | --- | --- |
| **Linear Search** | Check each element one by one | O(n) |
| **Binary Search** | Divide and conquer on **sorted** array | O(log n) |

## 🔸 **1. Linear Search**

### 📌 Real-life Example: Searching for a name in an unorganized guest list.

public class LinearSearch {

public static int linearSearch(int[] arr, int key) {

for (int i = 0; i < arr.length; i++) {

if (arr[i] == key)

return i; // Found at index

}

return -1; // Not found

}

public static void main(String[] args) {

int[] nums = {11, 25, 39, 42, 53};

int key = 39;

int index = linearSearch(nums, key);

System.out.println(index != -1 ? "Found at index: " + index : "Not found");

}

}

## 🔸 **2. Binary Search**

Works only on a **sorted array**

### 📌 Real-life Example: Searching a word in a dictionary (ordered list).

public class BinarySearch {

public static int binarySearch(int[] arr, int key) {

int start = 0, end = arr.length - 1;

while (start <= end) {

int mid = (start + end) / 2;

if (arr[mid] == key)

return mid;

else if (key < arr[mid])

end = mid - 1;

else

start = mid + 1;

}

return -1; // Not found

}

public static void main(String[] args) {

int[] nums = {10, 20, 30, 40, 50, 60};

int key = 50;

int index = binarySearch(nums, key);

System.out.println(index != -1 ? "Found at index: " + index : "Not found");

}

}

## 🔸 Bonus: Recursive Binary Search

public class RecursiveBinarySearch {

public static int binarySearch(int[] arr, int key, int start, int end) {

if (start > end) return -1;

int mid = (start + end) / 2;

if (arr[mid] == key) return mid;

else if (key < arr[mid]) return binarySearch(arr, key, start, mid - 1);

else return binarySearch(arr, key, mid + 1, end);

}

public static void main(String[] args) {

int[] nums = {5, 15, 25, 35, 45};

int key = 25;

int index = binarySearch(nums, key, 0, nums.length - 1);

System.out.println(index != -1 ? "Found at index: " + index : "Not found");

}

}

## 🛠️ When to Use Which?

| **Algorithm** | **When to Use** |
| --- | --- |
| Linear Search | Small/unsorted datasets |
| Binary Search | Large/sorted datasets |

public class LinearSearch {

public static int linearSearch(String[] arr, String key) {

for (int i = 0; i < arr.length; i++) {

if (arr[i].equalsIgnoreCase(key)) // Case-insensitive match

return i;

}

return -1; // Not found

}

public static void main(String[] args) {

String[] names = {"Arun", "Meena", "Rahul", "Priya", "Amit"};

String key = "Rahul";

int index = linearSearch(names, key);

System.out.println(index != -1 ? "Found at index: " + index : "Name not found");

}

}

**Sorting:**

## ✅ 1. **Bubble Sort**

### 🔧 ****Concept:****

Repeatedly **compare adjacent elements** and **swap if out of order**.

Largest elements "bubble" to the end in each pass.

### 🔄 ****Working Example:****

Array: [5, 3, 8, 4]

Pass 1: [3, 5, 4, 8]

Pass 2: [3, 4, 5, 8]

Pass 3: Already sorted

### ⏱ ****Time Complexity:****

Best: O(n) (already sorted)

Worst: O(n²)

Space: O(1) (in-place)

### ✅ ****Use Case:****

Educational purpose or very small datasets.

## ✅ 2. **Selection Sort**

### 🔧 ****Concept:****

**Find the minimum element** and put it at the beginning.

Repeat for next index.

### 🔄 ****Working Example:****

Array: [29, 10, 14, 37]

Pass 1: Find min 10, swap with 29 → [10, 29, 14, 37]

Pass 2: Find min 14, swap with 29 → [10, 14, 29, 37]

### ⏱ ****Time Complexity:****

Best/Worst/Average: O(n²)

Space: O(1)

### ❌ ****Use Case:****

Rare in practice; easy to understand but inefficient.

## ✅ 3. **Insertion Sort**

### 🔧 ****Concept:****

Build the final sorted array one item at a time by **inserting each element into the correct position**.

### 🔄 ****Working Example:****

Array: [4, 3, 2, 10]

3 < 4 → [3, 4, 2, 10]

2 < 3 → [2, 3, 4, 10]

### ⏱ ****Time Complexity:****

Best: O(n) (already sorted)

Worst: O(n²)

Space: O(1)

### ✅ ****Use Case:****

Good for **nearly sorted or small data** (e.g., sorting live data entry).

## ✅ 4. **Merge Sort** (Divide and Conquer)

### 🔧 ****Concept:****

**Divide** array into halves → recursively sort → **merge** back

### 🔄 ****Working Example:****

Array: [38, 27, 43, 3]  
Split → [38, 27], [43, 3]  
Sort → [27, 38], [3, 43]  
Merge → [3, 27, 38, 43]

### ⏱ ****Time Complexity:****

Best/Worst: O(n log n)

Space: O(n)

### ✅ ****Use Case:****

Large datasets or guaranteed performance needs (e.g., external sorting).

## ✅ 5. **Quick Sort** (Divide and Conquer)

### 🔧 ****Concept:****

Choose a **pivot**, partition array into elements < pivot and > pivot, then **recursively sort** sub-arrays.

### 🔄 ****Working Example:****

Array: [10, 80, 30, 90], Pivot = 90  
Partition → [10, 80, 30] (all < 90)  
Recurse on [10, 80, 30]

### ⏱ ****Time Complexity:****

Best/Average: O(n log n)

Worst: O(n²) (bad pivot)

Space: O(log n) (recursive calls)

### ✅ ****Use Case:****

**In-memory fast sorting**, especially when average-case performance matters.

## ✅ 6. **Java Built-in Sort**

### 🔧 Arrays.sort() for primitives:

Uses **Dual Pivot Quick Sort** (O(n log n))

### 🔧 Arrays.sort() for objects:

Uses **TimSort** (Hybrid of Merge + Insertion sort)

### ✅ ****Use Case:****

Most real-world projects use this as it's **optimized and robust**.

## ⚖️ Comparison Table

| **Algorithm** | **Best Case** | **Worst Case** | **Stable** | **In-place** | **Use Case** |
| --- | --- | --- | --- | --- | --- |
| Bubble Sort | O(n) | O(n²) | ✅ | ✅ | Educational |
| Selection Sort | O(n²) | O(n²) | ❌ | ✅ | Rarely used |
| Insertion Sort | O(n) | O(n²) | ✅ | ✅ | Small/Nearly sorted arrays |
| Merge Sort | O(n log n) | O(n log n) | ✅ | ❌ | Large data / linked list |
| Quick Sort | O(n log n) | O(n²) | ❌ | ✅ | Fastest on average in memory |
| TimSort | O(n) | O(n log n) | ✅ | ❌ | Built-in, real-world use |

## ✅ **What is Sorting?**

Sorting is the process of **arranging elements in a particular order** — most commonly **ascending** or **descending**.

## 🔢 Common Sorting Algorithms in Java

| **Algorithm** | **Time Complexity (Best/Worst)** | **Stable?** | **Use Case** |
| --- | --- | --- | --- |
| Bubble Sort | O(n) / O(n²) | ✅ Yes | Educational, small datasets |
| Selection Sort | O(n²) / O(n²) | ❌ No | Easy to implement, not efficient |
| Insertion Sort | O(n) / O(n²) | ✅ Yes | Small or nearly sorted data |
| Merge Sort | O(n log n) / O(n log n) | ✅ Yes | Large data, guaranteed performance |
| Quick Sort | O(n log n) / O(n²) | ❌ No | Fast in practice for large datasets |
| Arrays.sort() | O(n log n) (Dual-Pivot QS) | ✅ Yes | Built-in Java method, optimized |

## 🔹 **1. Bubble Sort**

public class BubbleSort {

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

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

swapped = true;

}

}

if (!swapped) break; // Optimization

}

}

public static void main(String[] args) {

int[] arr = {64, 25, 12, 22, 11};

bubbleSort(arr);

System.out.println("Sorted array:");

for (int num : arr) System.out.print(num + " ");

}

}

## 🔹 **2. Insertion Sort**

public class InsertionSort {

public static void insertionSort(int[] arr) {

for (int i = 1; i < arr.length; i++) {

int key = arr[i];

int j = i - 1;

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j--;

}

arr[j + 1] = key;

}

}

public static void main(String[] args) {

int[] arr = {10, 7, 4, 3, 1};

insertionSort(arr);

System.out.println("Sorted array:");

for (int n : arr) System.out.print(n + " ");

}

}

## 🔹 **3. Selection Sort**

public class SelectionSort {

public static void selectionSort(int[] arr) {

for (int i = 0; i < arr.length - 1; i++) {

int min = i;

for (int j = i + 1; j < arr.length; j++) {

if (arr[j] < arr[min]) {

min = j;

}

}

// Swap

int temp = arr[min];

arr[min] = arr[i];

arr[i] = temp;

}

}

public static void main(String[] args) {

int[] arr = {29, 10, 14, 37, 13};

selectionSort(arr);

System.out.println("Sorted array:");

for (int n : arr) System.out.print(n + " ");

}

}

## 🔹 **4. Merge Sort**

public class MergeSort {

public static void mergeSort(int[] arr, int left, int right) {

if (left < right) {

int mid = (left + right) / 2;

mergeSort(arr, left, mid);

mergeSort(arr, mid + 1, right);

merge(arr, left, mid, right);

}

}

public static void merge(int[] arr, int left, int mid, int right) {

int n1 = mid - left + 1;

int n2 = right - mid;

int[] L = new int[n1];

int[] R = new int[n2];

for (int i = 0; i < n1; ++i) L[i] = arr[left + i];

for (int j = 0; j < n2; ++j) R[j] = arr[mid + 1 + j];

int i = 0, j = 0, k = left;

while (i < n1 && j < n2)

arr[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];

while (i < n1) arr[k++] = L[i++];

while (j < n2) arr[k++] = R[j++];

}

public static void main(String[] args) {

int[] arr = {38, 27, 43, 3, 9, 82, 10};

mergeSort(arr, 0, arr.length - 1);

System.out.println("Sorted array:");

for (int n : arr) System.out.print(n + " ");

}

}

Explanation:

Initial Call

java

[38, 27, 43, 3, 9, 82, 10]

mergeSort(arr, 0, 6);

left = 0, right = 6, mid = (0+6)/2 = 3

Split into:

Left Half → mergeSort(arr, 0, 3)

Right Half → mergeSort(arr, 4, 6)

Step 3: Breaking Down Recursion

First Half: mergeSort(arr, 0, 3)

left=0, right=3, mid=(0+3)/2=1

Again split:

Left → mergeSort(arr, 0, 1)

Right → mergeSort(arr, 2, 3)

Sub-call: mergeSort(arr, 0, 1)

left=0, right=1, mid=0

Calls:

mergeSort(arr, 0, 0) → Single element, returns

mergeSort(arr, 1, 1) → Single element, returns

Now merge(0,0,1):

L=[38], R=[27]

Merge → [27,38]

Result after merging → [27,38,43,3,9,82,10]

Sub-call: mergeSort(arr, 2, 3)

left=2, right=3, mid=2

Calls:

mergeSort(arr, 2, 2) → returns

mergeSort(arr, 3, 3) → returns

merge(2,2,3):

L=[43], R=[3]

Merge → [3,43]

Result → [27,38,3,43,9,82,10]

Merge (0,1,3)

L=[27,38], R=[3,43]

Merge → [3,27,38,43]

Result → [3,27,38,43,9,82,10]

Step 4: Second Half: mergeSort(arr, 4, 6)

left=4, right=6, mid=5

Split into:

mergeSort(arr, 4, 5)

mergeSort(arr, 6, 6)

mergeSort(arr, 4, 5)

left=4, right=5, mid=4

Calls:

mergeSort(arr, 4, 4) → returns

mergeSort(arr, 5, 5) → returns

merge(4,4,5):

L=[9], R=[82]

Merge → [9,82]

Result → [3,27,38,43,9,82,10]

merge(4,5,6)

L=[9,82], R=[10]

Merge → [9,10,82]

Result → [3,27,38,43,9,10,82]

Step 5: Final Merge (0,3,6)

L=[3,27,38,43], R=[9,10,82]

Merge:

Compare elements one by one:

3 < 9 → 3

27 > 9 → 9

27 > 10 → 10

27 < 82 → 27

38 < 82 → 38

43 < 82 → 43

Remaining 82

Final Sorted Array →

csharp

Copy

Edit

[3, 9, 10, 27, 38, 43, 82]

✅ Final Output

php

Copy

Edit

Sorted array:

3 9 10 27 38 43 82

## 🔹 **5. Quick Sort**

public class QuickSort {

public static void quickSort(int[] arr, int low, int high) {

if (low < high) {

int pi = partition(arr, low, high);

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

public static int partition(int[] arr, int low, int high) {

int pivot = arr[high];

int i = (low - 1);

for (int j = low; j < high; j++) {

if (arr[j] < pivot) {

i++;

// Swap

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

// Swap pivot

int temp = arr[i + 1];

arr[i + 1] = arr[high];

arr[high] = temp;

return i + 1;

}

public static void main(String[] args) {

int[] arr = {10, 80, 30, 90, 40, 50, 70};

quickSort(arr, 0, arr.length - 1);

System.out.println("Sorted array:");

for (int n : arr) System.out.print(n + " ");

}

}

Explanation:

Step-by-Step Execution (for input {8, 4, 7, 3, 10, 2})

➤ First call: quickSort(arr, 0, 5)

Pivot = 2

Partition:

All elements are > 2, so no swaps.

Swap 2 with 8 → {2, 4, 7, 3, 10, 8}

Pivot index = 0

➤ Recursive call on left: quickSort(arr, 0, -1) → nothing happens

➤ Recursive call on right: quickSort(arr, 1, 5)

Pivot = 8

Elements < 8 are: 4, 7, 3 → swapped ahead

After partition → {2, 4, 7, 3, 8, 10}

Pivot index = 4

➤ Left: quickSort(arr, 1, 3)

Pivot = 3

Partition:

4 > 3 → no swap

7 > 3 → no swap

3 swaps with 4 → {2, 3, 7, 4, 8, 10}

Pivot index = 1

➤ Right: quickSort(arr, 2, 3)

Pivot = 4

Partition:

7 > 4 → no swap

4 swaps with 7 → {2, 3, 4, 7, 8, 10}

Pivot index = 2

Now array is fully sorted → {2, 3, 4, 7, 8, 10}

## 🔹 **6. Built-in Sort (Arrays.sort)**

import java.util.Arrays;

public class BuiltInSort {

public static void main(String[] args) {

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

Arrays.sort(arr);

System.out.println("Sorted using Arrays.sort():");

for (int n : arr) System.out.print(n + " ");

}

}

**sorting of Strings and custom objects (like Student)** in Java using Arrays.sort() and Collections.sort() with **Comparator** and **Comparable** interfaces.

## ✅ 1. **Sorting String Arrays (Alphabetical Order)**

import java.util.Arrays;

public class SortStrings {

public static void main(String[] args) {

String[] names = {"Meena", "Arun", "Priya", "Zara", "Bala"};

Arrays.sort(names); // Alphabetical order

System.out.println("Sorted names:");

for (String name : names)

System.out.print(name + " ");

}

}

## ✅ 2. **Sorting List of Strings (Using Collections.sort)**

java

CopyEdit

import java.util.\*;

public class SortStringList {

public static void main(String[] args) {

List<String> cities = Arrays.asList("Mumbai", "Delhi", "Chennai", "Kolkata");

Collections.sort(cities);

System.out.println("Sorted cities:");

for (String city : cities)

System.out.print(city + " ");

}

}

## ✅ 3. **Sorting Custom Objects (Student class)**

### 🎯 Sort Students by Name and then by Marks

### 🔹 ****Student Class with Comparable (sort by name)****

class Student implements Comparable<Student> {

String name;

int marks;

Student(String name, int marks) {

this.name = name;

this.marks = marks;

}

// Sort by name

public int compareTo(Student other) {

return this.name.compareTo(other.name);

}

public String toString() {

return name + " - " + marks;

}

}

### 🔹 ****Main Class: Sort by Name (default)****

import java.util.\*;

public class SortStudentByName {

public static void main(String[] args) {

List<Student> list = new ArrayList<>();

list.add(new Student("Rahul", 75));

list.add(new Student("Amit", 85));

list.add(new Student("Meena", 65));

list.add(new Student("Priya", 90));

Collections.sort(list); // Uses compareTo (by name)

System.out.println("Students sorted by name:");

for (Student s : list)

System.out.println(s);

}

}

### ✅ 4. ****Sorting with Comparator (Sort by marks)****

import java.util.\*;

class SortStudentByMarks {

public static void main(String[] args) {

List<Student> list = new ArrayList<>();

list.add(new Student("Rahul", 75));

list.add(new Student("Amit", 85));

list.add(new Student("Meena", 65));

list.add(new Student("Priya", 90));

// Sort by marks (ascending)

list.sort(Comparator.comparingInt(s -> s.marks));

System.out.println("Students sorted by marks:");

for (Student s : list)

System.out.println(s);

}

}

### ✅ 5. ****Descending Sort by Marks****

list.sort((s1, s2) -> Integer.compare(s2.marks, s1.marks)); // Descending

## 🔚 Summary

Use Arrays.sort() for arrays and Collections.sort() for lists.

Use Comparable for natural ordering (like name).

Use Comparator for custom sorting (like marks, age, etc.).

You can chain comparisons or sort by multiple fields.

## ****Functional Interface****

A **functional interface** is an interface that has **exactly one abstract method**.  
It can have:

Any number of **default** methods

Any number of **static** methods

But only **one** abstract method

💡 They are the backbone of **lambda expressions** because a lambda needs a single method to implement.

### ****Syntax****

@FunctionalInterfaceinterface MyFunction {

void execute(); // single abstract method

}

💡 @FunctionalInterface is **optional** but **recommended** — it forces compile-time checking so no extra abstract methods can be added.

## **2. Types of Functional Interfaces**

Java provides some predefined functional interfaces in java.util.function package.  
They fall mainly into **four categories**:

| **Type** | **Abstract Method** | **Description** | **Example Lambda** |
| --- | --- | --- | --- |
| **Predicate<T>** | boolean test(T t) | Tests a condition and returns boolean | x -> x > 10 |
| **Function<T,R>** | R apply(T t) | Takes input of type T and returns R | str -> str.length() |
| **Supplier<T>** | T get() | No input, returns a value | () -> Math.random() |
| **Consumer<T>** | void accept(T t) | Takes input and performs action, no return | x -> System.out.println(x) |

### ****3. Real-Time Example with All Four Types****

import java.util.function.\*;

import java.util.\*;

public class FunctionalInterfaceTypes {

public static void main(String[] args) {

// Predicate - Check if order amount is high

Predicate<Double> isHighValue = amount -> amount > 10000;

System.out.println(isHighValue.test(15000.0)); // true

// Function - Convert order amount to discounted amount

Function<Double, Double> applyDiscount = amount -> amount \* 0.9;

System.out.println(applyDiscount.apply(20000.0)); // 18000.0

// Supplier - Generate random order ID

Supplier<Integer> orderIdGenerator = () -> new Random().nextInt(1000);

System.out.println(orderIdGenerator.get());

// Consumer - Print order details

Consumer<String> printOrder = order -> System.out.println("Processing order: " + order);

printOrder.accept("Order#101");

}

}

## **4. Benefits in Real Time**

**Predicate** → used in filtering collections

**Function** → used in mapping data (converting one type to another)

**Supplier** → used in lazy loading or generating IDs

**Consumer** → used in logging, printing, sending notifications