JAVA Revise

Important Topics

Core Java

1.Java basics

- Features of java
- JVM,JRE,JDK
- Program structure

2.Datatypes and Variables

- Primitive vs Non-Primitive
- Type casting

3.Operators

- Arithmetic, Relational, Logical, Ternary

4. Control Statements

- if-else, switch
- Loops (for, while, do-while)
- break, continue

5. Arrays & Strings

- 1D, 2D arrays
- String, StringBuilder, StringBuffer
- String methods & immutability

6. OOP Concepts

- Class, Object, Constructor
- this & super keywords
- Inheritance
- Method Overloading & Overriding
- Polymorphism
- Encapsulation
- Abstraction (abstract class, interface)

7.Access Modifiers & Packages

8. Exception Handling

- try, catch, finally
- throw, throws
- Checked vs Unchecked exceptions

9. Collections Framework

- List, Set, Map
- ArrayList, LinkedList,
- HashSet, TreeSet, HashMap
- Comparable vs Comparator

10. Java Memory Management

- Stack vs Heap
- Garbage Collection

11. Java 8+ Features

- Lambda Expressions
- Functional Interfaces
- Stream API
- Default & Static methods in interfaces

12. Multithreading

- Thread class, Runnable interface
- Synchronization basics

Advanced Java (If targeting backend jobs)

1. **JDBC** (Database Connectivity)
2. **Servlets** (Request/Response flow)
3. **JSP** (Java Server Pages)
4. **Hibernate / JPA** (ORM basics)
5. **Spring / Spring Boot** (Basics, REST API)

Extras (for Java Full Stack)

1. HTML, CSS, JavaScript basics
2. MySQL or PostgreSQL basics
3. REST API creation & consumption
4. Git & GitHub basics
5. Basic deployment knowledge (Tomcat, AWS, etc.)

Answers

1. Java Basics

- **Object-Oriented** → Everything is treated as objects except primitive data types.
- **Platform-Independent** → "Write once, run anywhere" (compiled code runs on any OS with JVM).
- **Strongly Typed** → You must declare the type of every variable.
- **Secure** → No direct memory access like in C/C++, has built-in security features.

Features of java:

Feature	Explanation
Simple	Easy syntax, similar to C/C++ but without complex features like pointers.
Object-Oriented	Based on objects, classes, and reusability.
Platform-Independent	Bytecode can run on any OS with JVM.
Secure	No explicit pointer use, has runtime checks.
Robust	Strong memory management and exception handling.
Portable	Same code works anywhere without recompilation.
High Performance	Just-In-Time (JIT) compilation improves speed.
Multithreaded	Can execute multiple tasks in parallel.
Distributed	Can work with remote objects via RMI, web services.
Dynamic	Can load classes dynamically at runtime.

JVM, JDK, JRE

JVM (Java Virtual Machine)

- Executes **bytecode**.
- Provides platform independence.
- Handles memory management, garbage collection, and security.

JVM Tasks:

1. Loads bytecode.
2. Verifies it.
3. Executes via an interpreter or JIT compiler.

JRE (Java Runtime Environment)

- Contains **JVM** + **Java libraries** + **other components** to run Java applications.
- **Used to run** Java programs, not to develop them.

JDK (Java Development Kit)

- Contains **JRE** + **development tools** (compiler javac, debugger, jar tool, etc.).
- **Used to develop and run** Java programs.

Java Program Structure

```
public class HelloWorld {
// Main method - Entry point
public static void main(String[] args) {
System.out.println("Hello, Java!");
}
}
```

Breakdown:

1. **Class Declaration** → public class HelloWorld defines a class.
2. **Main Method** → public static void main(String[] args) is the entry point.
3. **Statements** → System.out.println("Hello, Java!"); prints output.

Execution Process

1. **Write code** → HelloWorld.java
2. **Compile** → javac HelloWorld.java → generates HelloWorld.class (bytecode).
3. **Run** → java HelloWorld → JVM executes the bytecode.

2.Datatypes and Variables

DataTypes:

Java is **strongly typed**, meaning every variable must have a declared type.
There are **two categories** of data types:

A. **Primitive Data Types** (8 types)
B. **Non-Primitive Data Types**

A. Primitive Data Types

Type	Size	Default Value	Example
byte	1 byte	0	`byte b = 10;`
short	2 bytes	0	`short s = 100;`
int	4 bytes	0	`int x = 500;`
long	8 bytes	0L	`long l = 100000L;`
float	4 bytes	0.0f	`float f = 5.5f;`
double	8 bytes	0.0d	`double d = 3.14;`
char	2 bytes	'\u0000'	`char c = 'A';`
boolean	1 bit	false	`boolean flag = true;`

B. Non-Primitive Data Types

- **String** → String name = "Java";
- **Arrays** → int[] numbers = {1, 2, 3};
- **Classes / Objects**
- **Interfaces**

Variables:

A **variable** is a container to store data.

Types of Variables:

- Local Variable
- Instance Variable (Non-static field)
- Static Variable (Class variable)

1.Local variables

- Declared inside a method, constructor, or block.
- Must be initialized before use.

```
void method() {
    int x = 10; // local variable
    System.out.println(x);
}
```

2.Instance Variable (Non-static field)

- Declared inside a class but outside methods.
- Belongs to the object.

```
class Student {
    String name; // instance variable
}
```

3. Static Variable (Class variable)

- Declared with static keyword.
- Shared among all objects of the class.

```
class Counter {
    static int count = 0;
}
```

Example program:

```
public class DataTypeExample {
    static int staticVar = 50; // Static variable
    String instanceVar = "Java"; // Instance variable

    public void show() {
        int localVar = 10; // Local variable
        System.out.println("Local: " + localVar);
        System.out.println("Instance: " + instanceVar);
        System.out.println("Static: " + staticVar);
    }

    public static void main(String[] args) {
        DataTypeExample obj = new DataTypeExample();
        obj.show();
    }
}
```

Typecasting

1.What is Typecasting?

Typecasting means **converting one data type into another**.
In Java, this is mainly used when you want to:

- Change a variable's type (e.g., int → double)
- Assign a value of one type to another type

2. Types of Typecasting in Java

A. Implicit Typecasting (Widening)

- **Smaller data type → Larger data type**
- Done automatically by Java (no data loss).
- Also called **Type Promotion**.

Example:

```
int num = 10;
double val = num; // int → double
System.out.println(val); // 10.0
```

Order of Widening:

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

B. Explicit Typecasting(Narrow)

- **Larger data type → Smaller data type**
- Must be done **manually** using (type) operator.
- May cause **data loss**.

Example:

```
double num = 9.78;
int val = (int) num; // double → int
System.out.println(val); // 9 (fraction lost)
```

Order of Narrowing:

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

3. Typecasting with Objects

Upcasting (Implicit)

- Subclass object assigned to a superclass reference.

- Done automatically.

```
class Animal {}
class Dog extends Animal {}

Animal a = new Dog(); // Upcasting
```

Downcasting (Explicit)

- Superclass reference cast to subclass.
- Must be done manually.

```
Animal a = new Dog();
Dog d = (Dog) a; // Downcasting
```

E xample Program

```
public class TypeCastingExample {
    public static void main(String[] args) {
        // Widening (Implicit)
        int x = 100;
        double y = x;
        System.out.println("Widening: " + y);

        // Narrowing (Explicit)
        double p = 99.99;
        int q = (int) p;
        System.out.println("Narrowing: " + q);

        // Object casting
        Animal a = new Dog(); // Upcasting
        Dog d = (Dog) a;      // Downcasting
        d.bark();
    }
}
class Animal {}
class Dog extends Animal {
    void bark() {
        System.out.println("Woof!");
    }
}
```

✅ **Key Interview Points**:

- Widening happens **automatically**; narrowing needs **explicit casting**.
- Narrowing can cause **precision/data loss**.
- With objects, casting is related to **inheritance hierarchy**.
- instanceof keyword is used before downcasting to **avoid ClassCastException**.

Operator	Meaning	Example
`==`	Equal to	`a == b`
`!=`	Not equal to	`a != b`
`>`	Greater than	`a > b`
`<`	Less than	`a < b`
`>=`	Greater or equal	`a >= b`
`<=`	Less or equal	`a <= b`

3.Operators

1. What is an Operator?

An **operator** is a symbol that performs an operation on one or more operands (variables/values).

Example:

```
int sum = 5 + 3; // '+' is the operator, 5 & 3 are operands
```

2. Types of Operators in Java

A. Arithmetic Operators

Used for mathematical calculations.

Example:

```
int a = 10, b = 3;
System.out.println(a + b); // 13
System.out.println(a % b); // 1
```

B. Relational (Comparison) Operators

Used to compare two values → returns true or false.

Operator	Description	Example
`+`	Addition	`a + b`
`-`	Subtraction	`a - b`
`*`	Multiplication	`a * b`
`/`	Division	`a / b`
`%`	Modulus (remainder)	`a % b`

C. Logical Operators

Used for logical conditions (works on boolean values).

Operator	Description	Example
`&&`	Logical AND	`a > 5 && b < 10`
`||`	Logical OR	`a > 5 || b < 10`
`!`	Logical NOT	`!(a > 5)`

D. Assignment Operators

Used to assign values.

Operator	Meaning	Example
`=`	Assign	`x = 5`
`+=`	Add & assign	`x += 3`
`-=`	Subtract & assign	`x -= 3`
`*=`	Multiply & assign	`x *= 3`
`/=`	Divide & assign	`x /= 3`
`%=`	Modulus & assign	`x %= 3`

E. Unary Operators

Works with only one operand.

Operator	Description	Example
`+`	Positive value	`+a`
`-`	Negation	`-a`
`++`	Increment	`a++` or `++a`
`--`	Decrement	`a--` or `--a`
`!`	Logical NOT	`!true`

F. Ternary Operator

Shortcut for if-else.

```
int a = 10, b = 20;
String result = (a > b) ? "A is greater" : "B is greater";
System.out.println(result);
```

G. Bitwise Operators

Work on bits.

Operator	Description	Example (a=5, b=3)
`&`	AND	`a & b` → `1`
`	`	OR
`^`	XOR	`a ^ b` → `6`
`~`	NOT	`~a` → `-6`
`<<`	Left shift	`a << 1` → `10`
`>>`	Right shift	`a >> 1` → `2`
`>>>`	Unsigned right shift	`a >>> 1`

H. instanceof Operator

Checks if an object is an instance of a class.

```
String s = "Hello";
System.out.println(s instanceof String); // true
```

Example Program

```
public class OperatorsExample {
    public static void main(String[] args) {
        int a = 10, b = 3;
        // Arithmetic
        System.out.println("Addition: " + (a + b));
        // Relational
        System.out.println("a > b: " + (a > b));
        // Logical
        System.out.println(a > 5 && b < 5);
```

```
        // Assignment
        a += 5;
        System.out.println("a after += 5: " + a);
        // Ternary
        String res = (a > b) ? "a is bigger" : "b is bigger";
        System.out.println(res);
        // Bitwise
        System.out.println("a & b: " + (a & b));
    }
}
```

4.Control Statements:

1. What are Conditional Statements?

Conditional statements allow you to **make decisions** in a program — so that certain code executes **only if a condition is true**.

Example:

```
if (marks >= 35) {
    System.out.println("Pass");
} else {
    System.out.println("Fail");
}
```

2. Types of Conditional Statements in Java

A. if Statement

Executes a block of code **only if** the condition is true.

```
int age = 18;
if (age >= 18) {
    System.out.println("You are eligible to vote.");
}
```

B. if-else Statement

Executes one block if the condition is true, otherwise another block.

```
int num = 5;
if (num % 2 == 0) {
    System.out.println("Even");
} else {
    System.out.println("Odd");
}
```

C. if-else-if Ladder

Checks multiple conditions **one by one**.

```
int marks = 85;
if (marks >= 90) {
    System.out.println("Grade A");
} else if (marks >= 75) {
    System.out.println("Grade B");
} else if (marks >= 50) {
    System.out.println("Grade C");
} else {
    System.out.println("Fail");
}
```

D. Nested if

An **if statement inside another if**.

```
int age = 25;
boolean hasID = true;

if (age >= 18) {
    if (hasID) {
        System.out.println("You can enter the club.");
    }
}
```

E. switch Statement

Used when you have multiple possible values for a single variable.

```
int day = 3;
switch (day) {
    case 1: System.out.println("Monday"); break;
    case 2: System.out.println("Tuesday"); break;
    case 3: System.out.println("Wednesday"); break;
    default: System.out.println("Invalid day");
}
```

3. Ternary Operator *(Shortcut for if-else)*

```
int age = 20;
String result = (age >= 18) ? "Adult" : "Minor";
System.out.println(result);
```

4. Example Program

```java
public class ConditionalExample {
    public static void main(String[] args) {
        int number = 10;

        // if-else
        if (number > 0) {
            System.out.println("Positive number");
        } else {
            System.out.println("Negative number");
        }

        // switch
        switch (number) {
            case 10:
                System.out.println("Number is 10");
                break;
            default:
                System.out.println("Number is not 10");
        }
    }
}
```

Loops:

1. What are Loops?

Loops are used to **execute a block of code repeatedly** until a certain condition is met.

2. Types of Loops in Java

A. for Loop

- Runs a block of code for a **known number of iterations**.

Syntax:

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

Example:

```java
for (int i = 1; i <= 5; i++) {
    System.out.println("Count: " + i);
}
```

B. while Loop

- Runs **while** the condition is true.
- **Used when the number of iterations is unknown**.

```
int i = 1;
while (i <= 5) {
    System.out.println(i);
    i++;
}
```

C. do-while Loop

- Executes **at least once**, then checks the condition.
- Useful when you want the code to run first before checking.

```
int i = 1;
do {
    System.out.println(i);
    i++;
} while (i <= 5);
```

D. for-each Loop (Enhanced for loop)

- Used to iterate over arrays or collections.
- Automatically handles indexes.

```
int[] numbers = {1, 2, 3, 4};
for (int num : numbers) {
    System.out.println(num);
}
```

3. Loop Control Statements

- **break** → exits the loop immediately.

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) break;
    System.out.println(i);
}
```

- **continue** → skips the current iteration and moves to the next.

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) continue;
    System.out.println(i);
}
```

- **Nested loops** → a loop inside another loop.

```
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 2; j++) {
        System.out.println("i=" + i + ", j=" + j);
    }
}
```

4. Example Program

```
public class LoopsExample {
  public static void main(String[] args) {
    // for loop
    for (int i = 1; i <= 3; i++) {
      System.out.println("For Loop: " + i);
    }
   // while loop
    int j = 1;
    while (j <= 3) {
      System.out.println("While Loop: " + j);
      j++;
    }
   // do-while loop
    int k = 1;
    do {
      System.out.println("Do-While Loop: " + k);
      k++;
    } while (k <= 3);
  // for-each loop
    int[] nums = {10, 20, 30};
    for (int num : nums) {
      System.out.println("For-each: " + num);
    }
  }
}
```

Break & Continue

Break Statement

- **Purpose** → Immediately exits from a **loop** or a **switch** statement.
- **When to use** → When you want to stop execution as soon as a certain condition is met.

Example with loop:

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) {
        break; // stops the loop completely
    }
    System.out.println(i);
}
```

Output:

```
1
2
```

Example with switch:

```
int day = 2;
switch (day) {
    case 1: System.out.println("Monday"); break;
    case 2: System.out.println("Tuesday"); break;
    default: System.out.println("Other day");
}
```

2. continue Statement

- **Purpose** → Skips the **current iteration** of the loop and jumps to the next iteration.
- **When to use** → When you want to skip certain values but continue the loop.

Example:

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) {
        continue; // skip printing 3
    }
    System.out.println(i);
}
```

Output:

```
1
2
4
5
```

3. Key Differences Between break & continue

Feature	break	continue
Function	Exits the loop completely	Skips current iteration only
Use case	When condition met, stop looping	When condition met, skip and move on
Effect	Ends the loop	Loop still runs for remaining iterations

4. Nested Loop Example

```
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
        if (j == 2) continue; // skip printing when j=2
        if (i == 3) break;    // stop inner loop when i=3
        System.out.println("i=" + i + ", j=" + j);
    }
}
```

5.Arrays & Strings

Array: An array in Java is a collection of elements of the same data type, stored in contiguous memory locations, and accessed using an index (starting from 0). It allows storing multiple values in a single variable instead of declaring separate variables for each value.

Example:

```
int[] numbers = {10, 20, 30, 40, 50};
System.out.println(numbers[0]); // Output: 10
```

Key Points:

- Fixed size once created.
- Index-based access (0 to length-1).
- Can store primitives (int, char, etc.) and objects (String, etc.).
- Stored in heap memory in Java.

Methods to create an Array

In Java, there are **two main ways** to create an array:

1. Declaration + Allocation

you first declare the array type, then allocate memory using new.

```
int[] arr = new int[5];  // size is fixed at 5
```

2. Declaration + Initialization with Values

You create and assign values directly (no new keyword needed here).

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

Extra Variations

- Separate Declaration and Allocation

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

- **Anonymous Array** (used directly without storing in a variable)

```
System.out.println(new int[]{1, 2, 3}[0]); // prints 1
```

Problems:

BASIC ARRAY PROGRAMS

1. Print all elements of an array

```
public class PrintArray {
    public static void main(String[] args) {
        int[] arr = {10, 20, 30, 40, 50};
        for (int num : arr) {
            System.out.print(num + " ");
        }
    }
}
```

Output:

```
10 20 30 40 50
```

2. Find the length of an array

```
public class ArrayLength {
    public static void main(String[] args) {
        int[] arr = {5, 10, 15, 20};
        System.out.println("Length: " + arr.length);
    }
}
```

Output:

```
Length: 4
```

3. Sum of all elements in an array

```java
public class SumArray {
    public static void main(String[] args) {
        int[] arr = {5, 10, 15};
        int sum = 0;
        for (int num : arr) {
            sum += num;
        }
        System.out.println("Sum: " + sum);
    }
}
```

Output:

```
Sum: 30
```

4. Find the largest element

```java
public class LargestElement {
    public static void main(String[] args) {
        int[] arr = {10, 45, 3, 99, 23};
        int max = arr[0];
        for (int num : arr) {
            if (num > max) {
                max = num;
            }
        }
        System.out.println("Largest: " + max);
    }
}
```

Output:

```
Largest: 99
```

5. Find the smallest element

```java
public class SmallestElement {
    public static void main(String[] args) {
        int[] arr = {10, 45, 3, 99, 23};
        int min = arr[0];
        for (int num : arr) {
            if (num < min) {
                min = num;
            }
        }
        System.out.println("Smallest: " + min);
    }
}
```

Output:

```
Smallest: 3
```

6. Reverse an array

```java
public class ReverseArray {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        for (int i = arr.length - 1; i >= 0; i--) {
            System.out.print(arr[i] + " ");
        }
    }
}
```

Output:

```
5 4 3 2 1
```

7. Copy elements from one array to another

```java
public class CopyArray {
    public static void main(String[] args) {
        int[] arr1 = {10, 20, 30};
        int[] arr2 = new int[arr1.length];
        for (int i = 0; i < arr1.length; i++) {
            arr2[i] = arr1[i];
        }
        System.out.println("Copied Array:");
        for (int num : arr2) {
            System.out.print(num + " ");
        }
    }
}
```

Output:

```
Copied Array:
10 20 30
```

INTERMEDIATE ARRAY PROGRAMS:

8. Find the second largest element

```
public class SecondLargest {
    public static void main(String[] args) {
        int[] arr = {10, 20, 4, 45, 99};
        int largest = Integer.MIN_VALUE;
        int secondLargest = Integer.MIN_VALUE;

        for (int num : arr) {
            if (num > largest) {
                secondLargest = largest;
                largest = num;
            } else if (num > secondLargest && num != largest) {
                secondLargest = num;
            }
        }
        System.out.println("Second Largest: " + secondLargest);
    }
}
```

Output:

```
Second Largest: 45
```

9. Find the second smallest element

```
public class SecondSmallest {
    public static void main(String[] args) {
        int[] arr = {10, 20, 4, 45, 99};
        int smallest = Integer.MAX_VALUE;
        int secondSmallest = Integer.MAX_VALUE;

        for (int num : arr) {
            if (num < smallest) {
                secondSmallest = smallest;
                smallest = num;
            } else if (num < secondSmallest && num != smallest) {
                secondSmallest = num;
            }
        }
        System.out.println("Second Smallest: " + secondSmallest);
    }
}
```

Output:

```
Second Smallest: 10
```

10. Sort an array (Ascending Order)

```java
import java.util.Arrays;

public class SortArray {
    public static void main(String[] args) {
        int[] arr = {5, 1, 4, 2, 8};
        Arrays.sort(arr);
        System.out.println("Sorted Array: " + Arrays.toString(arr));
    }
}
```

Output:

```
Sorted Array: [1, 2, 4, 5, 8]
```

11. Check if an array is sorted

```java
public class CheckSorted {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        boolean sorted = true;
        for (int i = 0; i < arr.length - 1; i++) {
            if (arr[i] > arr[i + 1]) {
                sorted = false;
                break;
            }
        }
        System.out.println(sorted ? "Array is sorted" : "Array is not sorted");
    }
}
```

Output:

```
Array is sorted
```

12. Linear Search(Search an element in an array)

```java
public class LinearSearch {
    public static void main(String[] args) {
        int[] arr = {10, 25, 30, 45};
        int key = 30;
        boolean found = false;
        for (int num : arr) {
            if (num == key) {
                found = true;
                break;
            }
        }
        System.out.println(found ? "Element found" : "Element not found");
    }
}
```

Output:

```
Element found
```

13. Binary Search (sorted array)

```java
import java.util.Arrays;

public class BinarySearch {
    public static void main(String[] args) {
        int[] arr = {10, 25, 30, 45, 50};
        int key = 30;
        int index = Arrays.binarySearch(arr, key);
        System.out.println(index >= 0 ? "Element found at index " + index : "Not found");
    }
}
```

Output:

```
Element found at index 2
```

14. Count frequency of each element

```java
public class FrequencyCount {
    public static void main(String[] args) {
        int[] arr = {1, 2, 2, 3, 3, 3};
        boolean[] visited = new boolean[arr.length];

        for (int i = 0; i < arr.length; i++) {
            if (!visited[i]) {
                int count = 1;
                for (int j = i + 1; j < arr.length; j++) {
                    if (arr[i] == arr[j]) {
                        visited[j] = true;
                        count++;
                    }
                }
                System.out.println(arr[i] + " occurs " + count + " times");
            }
        }
    }
}
```

Output:

```
1 occurs 1 times
2 occurs 2 times
3 occurs 3 times
```

15. Find duplicate elements

```
public class Duplicates {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 2, 4, 5, 1};
        System.out.print("Duplicate elements: ");
        for (int i = 0; i < arr.length; i++) {
            for (int j = i + 1; j < arr.length; j++) {
                if (arr[i] == arr[j]) {
                    System.out.print(arr[i] + " ");
                    break;
                }
            }
        }
    }
}
```

Output:

```
Duplicate elements: 1 2
```

16. Remove duplicates from a sorted array

```
public class RemoveDuplicates {
    public static void main(String[] args) {
        int[] arr = {1, 1, 2, 2, 3, 4, 4};
        int length = arr.length;
        int[] temp = new int[length];
        int j = 0;

        for (int i = 0; i < length - 1; i++) {
            if (arr[i] != arr[i + 1]) {
                temp[j++] = arr[i];
            }
        }
        temp[j++] = arr[length - 1];

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

Output:

```
1 2 3 4
```

17. Rotate array by k positions (left)

```java
public class RotateArray {
    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        int k = 2;
        int n = arr.length;
        int[] temp = new int[n];

        for (int i = 0; i < n; i++) {
            temp[(i + n - k) % n] = arr[i];
        }

        for (int num : temp) {
            System.out.print(num + " ");
        }
    }
}
```

OutPut:

```
3 4 5 1 2
```

18. Merge two sorted arrays

```java
import java.util.Arrays;

public class MergeSortedArrays {
    public static void main(String[] args) {
        int[] arr1 = {1, 3, 5};
        int[] arr2 = {2, 4, 6};
        int[] merged = new int[arr1.length + arr2.length];

        int i = 0, j = 0, k = 0;
        while (i < arr1.length && j < arr2.length) {
            if (arr1[i] < arr2[j]) {
                merged[k++] = arr1[i++];
            } else {
                merged[k++] = arr2[j++];
            }
        }
        while (i < arr1.length) merged[k++] = arr1[i++];
        while (j < arr2.length) merged[k++] = arr2[j++];

        System.out.println("Merged Array: " + Arrays.toString(merged));
    }
}
```

Output:

```
Merged Array: [1, 2, 3, 4, 5, 6]
```

ADVANCE ARRAY PROGRAMS

19. Find missing number in an array from 1 to n

```
public class MissingNumber {
    public static void main(String[] args) {
        int[] arr = {1, 2, 4, 5};
        int n = 5; // supposed range is 1 to 5
        int totalSum = n * (n + 1) / 2;
        int arrSum = 0;
        for (int num : arr) {
            arrSum += num;
        }
        System.out.println("Missing Number: " + (totalSum - arrSum));
    }
}
```

Output:

```
Missing Number: 3
```

20. Find all pairs whose sum is a given number

```
public class PairSum {
    public static void main(String[] args) {
        int[] arr = {2, 4, 3, 5, 7, 8, 9};
        int target = 7;
        System.out.println("Pairs with sum " + target + ":");
        for (int i = 0; i < arr.length; i++) {
            for (int j = i + 1; j < arr.length; j++) {
                if (arr[i] + arr[j] == target) {
                    System.out.println(arr[i] + ", " + arr[j]);
                }
            }
        }
    }
}
```

Output:

```
Pairs with sum 7:
2, 5
4, 3
```

21. Find intersection of two arrays

```java
public class ArrayIntersection {
    public static void main(String[] args) {
        int[] arr1 = {1, 2, 3, 4, 5};
        int[] arr2 = {3, 4, 5, 6, 7};
        System.out.print("Intersection: ");
        for (int i : arr1) {
            for (int j : arr2) {
                if (i == j) {
                    System.out.print(i + " ");
                }
            }
        }
    }
}
```

Output:

```
Intersection: 3 4 5
```

22. Find union of two arrays

```java
import java.util.*;

public class ArrayUnion {
    public static void main(String[] args) {
        int[] arr1 = {1, 2, 3, 4, 5};
        int[] arr2 = {3, 4, 5, 6, 7};
        Set<Integer> set = new LinkedHashSet<>();
        for (int num : arr1) set.add(num);
        for (int num : arr2) set.add(num);
        System.out.println("Union: " + set);
    }
}
```

Output:

```
Union: [1, 2, 3, 4, 5, 6, 7]
```

23. Move all zeros to the end

```
public class MoveZeros {
    public static void main(String[] args) {
        int[] arr = {0, 1, 0, 3, 12};
        int index = 0;
        for (int num : arr) {
            if (num != 0) {
                arr[index++] = num;
            }
        }
        while (index < arr.length) {
            arr[index++] = 0;
        }
        for (int num : arr) {
            System.out.print(num + " ");
        }
    }
}
```

Output:

```
1 3 12 0 0
```

24. Separate even and odd numbers

```
public class EvenOddSeparate {
    public static void main(String[] args) {
        int[] arr = {12, 17, 70, 15, 22, 65, 21, 90};
        int left = 0, right = arr.length - 1;
        while (left < right) {
            while (arr[left] % 2 == 0 && left < right) left++;
            while (arr[right] % 2 == 1 && left < right) right--;
            if (left < right) {
                int temp = arr[left];
                arr[left] = arr[right];
                arr[right] = temp;
            }
        }
        for (int num : arr) {
            System.out.print(num + " ");
        }
    }
}
```

Output:

```
12 90 70 22 15 65 21 17
```

25. Find majority element (more than n/2 times)

```java
public class MajorityElement {
    public static void main(String[] args) {
        int[] arr = {3, 3, 4, 2, 3, 3, 3};
        int candidate = findCandidate(arr);
        if (isMajority(arr, candidate)) {
            System.out.println("Majority Element: " + candidate);
        } else {
            System.out.println("No Majority Element");
        }
    }

    static int findCandidate(int[] arr) {
        int count = 0, candidate = 0;
        for (int num : arr) {
            if (count == 0) {
                candidate = num;
            }
            count += (num == candidate) ? 1 : -1;
        }
        return candidate;
    }
```

```java
    static boolean isMajority(int[] arr, int candidate) {
        int count = 0;
        for (int num : arr) {
            if (num == candidate) count++;
        }
        return count > arr.length / 2;
    }
}
```

Output:

```
Majority Element: 3
```

26. Kadane's Algorithm – Maximum Subarray Sum

```java
public class MaxSubarraySum {
    public static void main(String[] args) {
        int[] arr = {-2, 1, -3, 4, -1, 2, 1, -5, 4};
        int maxSum = arr[0], currentSum = arr[0];
        for (int i = 1; i < arr.length; i++) {
            currentSum = Math.max(arr[i], currentSum + arr[i]);
            maxSum = Math.max(maxSum, currentSum);
        }
        System.out.println("Maximum Subarray Sum: " + maxSum);
    }
}
```

Output:

```
Maximum Subarray Sum: 6
```

27. Find subarrays with given sum

```java
public class SubarrayWithSum {
    public static void main(String[] args) {
        int[] arr = {1, 4, 20, 3, 10, 5};
        int target = 33;
        for (int start = 0; start < arr.length; start++) {
            int sum = 0;
            for (int end = start; end < arr.length; end++) {
                sum += arr[end];
                if (sum == target) {
                    System.out.println("Subarray found from index " + start + " to " + end);
                }
            }
        }
    }
}
```

Output:

```
Subarray found from index 2 to 4
```

28. Check if two arrays are equal

```java
import java.util.Arrays;

public class ArraysEqual {
    public static void main(String[] args) {
        int[] arr1 = {1, 2, 3};
        int[] arr2 = {1, 2, 3};
        System.out.println(Arrays.equals(arr1, arr2) ? "Arrays are equal" : "Arrays are not equal")
    }
}
```

Output:

```
Arrays are equal
```

Strings: In Java, a String is an object that represents a sequence of characters. It is immutable, meaning once a String object is created, its value cannot be changed. Strings are widely used for storing and processing text, and Java provides the String class in the java.lang package with many built-in methods for manipulation.

Methods to create String

1. Using String Literals

```
String str = "Hello";
```

- Stored in the String Constant Pool (SCP) inside the heap memory.
- If the same literal already exists in SCP, the reference is reused (memory-efficient).

Example:

```
String s1 = "Java";
String s2 = "Java"; // Reuses same object from SCP
System.out.println(s1 == s2); // true
```

2.Using new Keyword

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

- Always creates a new object in the heap memory (even if the same value exists in SCP).

Example:

```
String s1 = new String("Java");
String s2 = new String("Java");
System.out.println(s1 == s2); // false (different objects)
System.out.println(s1.equals(s2)); // true (values are same)
```

Most Important Things in Strings (Interview Focus):

1.Strings are Immutable:

- Once created, a String object **cannot be changed**.
- Any modification (like concat(), replace(), etc.) creates a **new String object**.

```
String s = "Java";
s.concat(" World");
System.out.println(s); // Java (unchanged)
```

2. Memory Storage – String Constant Pool (SCP):

- String literals are stored in the SCP for memory efficiency.
- If a literal already exists, Java reuses the reference instead of creating a new object.

```
String s1 = "Hello";
String s2 = "Hello";
System.out.println(s1 == s2); // true (same reference from SCP)
```

3. Ways to Create Strings:

- **Literal way** (SCP, reused if exists)
- **new keyword** (always creates a new object in heap)

```
String s1 = "Java";
String s2 = new String("Java");
```

4. String Comparison:

- **==** → compares memory references.
- **.equals()** → compares values (content).

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

5. Commonly Used Methods:

Method	Purpose
`length()`	Returns number of characters
`charAt(int)`	Get character at index
`substring(int, int)`	Extracts part of a string
`toUpperCase() / toLowerCase()`	Changes case
`trim()`	Removes leading/trailing spaces
`replace(old, new)`	Replaces characters
`contains()`	Checks substring presence
`startsWith()/endsWith()`	Checks start/end
`split()`	Splits string into array
`indexOf()/lastIndexOf()`	Finds position of character

6. String Mutability Alternatives:

- **StringBuilder** → Mutable, Not Thread-Safe (Faster)
- **StringBuffer** → Mutable, Thread-Safe (Slower)

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

7. String Interning:

- intern() method forces a String into SCP for reuse.

```
String s1 = new String("Java");
String s2 = s1.intern();
String s3 = "Java";
System.out.println(s2 == s3); // true
```

8. String is a Final Class:

- Ensures immutability and security.
- You cannot extend the String class.

Problems

1. Reverse a String

```
public class ReverseString {
    public static void main(String[] args) {
        String str = "Santhosh";
        String reversed = "";
        for(int i = str.length() - 1; i >= 0; i--) {
            reversed += str.charAt(i);
        }
        System.out.println("Reversed: " + reversed);
    }
}
```

Output:

```
Reversed: hsohtnaS
```

2. Check Palindrome

```java
public class PalindromeCheck {
    public static void main(String[] args) {
        String str = "madam";
        String rev = new StringBuilder(str).reverse().toString();
        if(str.equals(rev)) {
            System.out.println("Palindrome");
        } else {
            System.out.println("Not Palindrome");
        }
    }
}
```

Output:

```
Palindrome
```

3. Count Vowels and Consonants

```java
public class VowelConsonantCount {
    public static void main(String[] args) {
        String str = "Java Programming";
        str = str.toLowerCase();
        int vowels = 0, consonants = 0;

        for(char ch : str.toCharArray()) {
            if(ch >= 'a' && ch <= 'z') {
                if("aeiou".indexOf(ch) != -1) vowels++;
                else consonants++;
            }
        }
        System.out.println("Vowels: " + vowels);
        System.out.println("Consonants: " + consonants);
    }
}
```

Output:

```
Vowels: 5
Consonants: 9
```

4. Count Words in a String

```java
public class WordCount {
    public static void main(String[] args) {
        String str = "Java is awesome";
        String[] words = str.trim().split("\\s+");
        System.out.println("Word Count: " + words.length);
    }
}
```

Output:

```
Word Count: 3
```

5. Remove Duplicate Characters

```java
public class RemoveDuplicates {
    public static void main(String[] args) {
        String str = "programming";
        String result = "";
        for(char ch : str.toCharArray()) {
            if(result.indexOf(ch) == -1) {
                result += ch;
            }
        }
        System.out.println("Without duplicates: " + result);
    }
}
```

Output:

```
Without duplicates: progamin
```

6. Find Duplicate Characters

```java
public class DuplicateCharacters {
    public static void main(String[] args) {
        String str = "programming";
        str = str.toLowerCase();
        int[] freq = new int[256];

        for(char ch : str.toCharArray()) {
            freq[ch]++;
        }

        System.out.print("Duplicate characters: ");
        for(int i = 0; i < freq.length; i++) {
            if(freq[i] > 1 && i != ' ') {
                System.out.print((char)i + " ");
            }
        }
    }
}
```

Output:

```
Duplicate characters: r g m
```

7. Anagram Check

```java
import java.util.Arrays;

public class AnagramCheck {
    public static void main(String[] args) {
        String str1 = "listen";
        String str2 = "silent";

        char[] arr1 = str1.toCharArray();
        char[] arr2 = str2.toCharArray();

        Arrays.sort(arr1);
        Arrays.sort(arr2);

        if(Arrays.equals(arr1, arr2)) {
            System.out.println("Anagram");
        } else {
            System.out.println("Not Anagram");
        }
    }
}
```

Output:

```
Anagram
```

8. Count Occurrence of Each Character

```java
public class CharacterFrequency {
    public static void main(String[] args) {
        String str = "hello";
        int[] freq = new int[256];

        for(char ch : str.toCharArray()) {
            freq[ch]++;
        }

        for(int i = 0; i < freq.length; i++) {
            if(freq[i] > 0) {
                System.out.println((char)i + " - " + freq[i]);
            }
        }
    }
}
```

Output:

```
h - 1
e - 1
l - 2
o - 1
```

9. Remove White Spaces

```java
public class RemoveSpaces {
    public static void main(String[] args) {
        String str = " Java   Programming ";
        str = str.replaceAll("\\s+", "");
        System.out.println("Without spaces: " + str);
    }
}
```

Output:

```
Without spaces: JavaProgramming
```

10. Swap Two Strings Without Temp Variable

```java
public class SwapStrings {
    public static void main(String[] args) {
        String a = "Hello";
        String b = "World";

        a = a + b;
        b = a.substring(0, a.length() - b.length());
        a = a.substring(b.length());

        System.out.println("a: " + a);
        System.out.println("b: " + b);
    }
}
```

Output:

```
a: World
b: Hello
```

OOPs Concepts:

1. Class in Java

Definition

A **class** is a blueprint or template for creating objects.
It contains:

- **Fields (variables)** → store data
- **Methods** → define behavior

Syntax

```
class ClassName {
    // fields
    // methods
}
```

Example

```
class Car {
    String color;
    void drive() {
        System.out.println("Car is driving");
    }
}
```

Real-World Example:

A **Car design** in a factory is the *class*.

2. Object in Java

Definition

An **object** is an instance of a class.
It has **state** (values of variables) and **behavior** (methods it can perform).

Syntax

```
ClassName obj = new ClassName();
```

Example

```
public class Main {
    public static void main(String[] args) {
        Car myCar = new Car(); // object creation
        myCar.color = "Red";   // setting state
        myCar.drive();         // calling behavior
    }
}
```

Real-World Example:

The **actual car** built from the design is the *object*.

3. Constructor in Java

Definition

A **constructor** is a special method used to initialize objects.

- Same name as the class
- No return type (not even void)
- Called automatically when object is created

Types

1. **Default Constructor** – no parameters
2. **Parameterized Constructor** – with parameters

Syntax

```
class ClassName {
    ClassName() { } // default
    ClassName(parameters) { } // parameterized
}
```

Example

```
class Car {
    String color;
    Car(String c) { // parameterized constructor
        color = c;
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car("Blue");
        System.out.println("Car color: " + myCar.color);
    }
}
```

Output:

```
Car color: Blue
```

4. this Keyword

Definition

this refers to the **current object** of the class.

- Used to differentiate between instance variables and parameters with the same name.

- Used to call another constructor in the same class.
- Used to pass the current object as a parameter.

Example 1: Variable Differentiation

```
class Student {
    String name;
    Student(String name) {
        this.name = name; // this refers to current object's variable
    }
    void display() {
        System.out.println("Name: " + name);
    }
}

public class Main {
    public static void main(String[] args) {
        Student s = new Student("Santhosh");
        s.display();
    }
}
```

Output:

```
Name: Santhosh
```

5. super Keyword

Definition

super refers to the parent class object.

- Used to call parent class constructor.
- Used to access parent class methods and variables.

Example 1: Calling Parent Constructor

```
class Animal {
    Animal() {
        System.out.println("Animal Constructor");
    }
}

class Dog extends Animal {
    Dog() {
        super(); // calls parent constructor
        System.out.println("Dog Constructor");
    }
}

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

Output:

```
Animal Constructor
Dog Constructor
```

Example 2: Calling Parent Method

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

class Dog extends Animal {
    void sound() {
        super.sound(); // call parent method
        System.out.println("Dog barks");
    }
}

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

Output:

```
Animal sound
Dog barks
```

✅ Quick Interview Summary

Concept	Purpose	Example
Class	Blueprint for objects	Car design
Object	Instance of class	Actual car
Constructor	Initializes object	Set color, model when buying
this	Refers to current object	`this.name = name`
super	Refers to parent object	`super()` calls parent constructor

Inheritance:

1. Definition

Inheritance in Java is a mechanism where one class (child/subclass) **acquires** the properties and methods of another class (parent/superclass).
It helps in **code reusability** and creating a hierarchical relationship between classes.

2. Syntax

```
class ParentClass {
    // properties and methods
}

class ChildClass extends ParentClass {
    // additional properties and methods
}
```

- **extends** → keyword used to inherit from another class.
- Parent → Superclass / Base class
- Child → Subclass / Derived class

3. Simple Code Example

Copy Edit

```
// Parent Class
class Animal {
    void eat() {
        System.out.println("Eating...");
    }
}

// Child Class (inherits from Animal)
class Dog extends Animal {
    void bark() {
        System.out.println("Barking...");
    }
}

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

Output:

```
Eating...
Barking...
```

4. Real-World Example

- **Parent Class:** Vehicle (properties: speed, fuel; methods: start, stop)
- **Child Classes:** Car, Bike, Bus — they automatically get **start** and **stop** methods from Vehicle but can also have their own features.
- **Analogy:** Like a son inherits his father's traits but also has his own qualities.

5. Types of Inheritance in Java

Java supports:

1. **Single Inheritance** → One child from one parent.
2. **Multilevel Inheritance** → Child from parent, grandchild from child.
3. **Hierarchical Inheritance** → Multiple children from one parent.

❌ Java does **not** support **multiple inheritance** with classes (but supports via interfaces).

Method Overloading & Overriding

Method Overloading:

1.Definition

- **Same method name**, **different parameters** (number or type).
- Happens **within the same class**.
- Decided at **compile time** → called **Compile-time Polymorphism**.

2.Syntax

```
class Example {
    void display(int a) { }
    void display(String b) { }
}
```

Example Code

```
class MathOperations {
    // Method 1
    int add(int a, int b) {
        return a + b;
    }

    // Method 2 (different parameter type)
    double add(double a, double b) {
        return a + b;
    }

    // Method 3 (different number of parameters)
    int add(int a, int b, int c) {
        return a + b + c;
    }
}

public class OverloadingExample {
    public static void main(String[] args) {
        MathOperations m = new MathOperations();
        System.out.println(m.add(5, 3));       // calls int version
        System.out.println(m.add(5.5, 3.2));   // calls double version
        System.out.println(m.add(1, 2, 3));    // calls 3-parameter version
    }
```

Output:

```
8
8.7
6
```

3.Real-world Example

Think of a **printer**:

- Same button ("Print") but can print **PDF**, **Word**, **Image**.
- Same name, different inputs.

Method Overridding:

1.Definition

- **Same method name**, **same parameters**, **different implementation** in child class.
- Happens **between parent and child classes**.
- Decided at **runtime** → called **Runtime Polymorphism**.

2.Rules

1. Must have same method signature (name + parameters).
2. Must be in different classes (parent-child relationship).
3. Cannot override private, final, or static methods.

Example Code

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

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

public class OverridingExample {
    public static void main(String[] args) {
        Animal a = new Dog(); // Parent reference, child object
        a.sound(); // Calls Dog's version at runtime
    }
}
```

Output:

```
Dog barks
```

3.Real-world Example

Think of a **mobile's camera app**:

- The base app has a **capture()** method.
- Different phone models **override** this method for better quality or extra features.

✅ Quick Difference Table

Feature	Overloading	Overriding
Parameters	Must be different	Must be same
Return Type	Can be different (with rules)	Must be same or covariant
Class Relationship	Same class	Parent-child classes
Binding Time	Compile time	Runtime
Polymorphism Type	Compile-time	Runtime

Polymorphism

1. Definition

Polymorphism means **"many forms"** — the ability of the same method or object to behave differently based on the situation.
It allows a single interface to represent different underlying forms (data types or methods).

In **Java**, polymorphism is mainly of **two types**:

1. **Compile-time Polymorphism** → Method **Overloading**
2. **Runtime Polymorphism** → Method **Overriding**

2. Types of Polymorphism

A. Compile-time Polymorphism (Static Binding)

- Achieved through **Method Overloading**.
- Decided at **compile time**.
- Same method name, different parameter list.

Example:

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

public class CompileTimePoly {
    public static void main(String[] args) {
        MathOp m = new MathOp();
        System.out.println(m.add(5, 3));     // calls int version
        System.out.println(m.add(2.5, 3.5)); // calls double version
    }
}
```

Output:

```
8
6.0
```

B. Runtime Polymorphism (Dynamic Binding)

- Achieved through **Method Overriding**.
- Decided at **runtime**.
- Parent class reference can point to child class object, and the child's method is called.

Example:

```
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

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

public class RuntimePoly {
    public static void main(String[] args) {
        Animal a = new Dog(); // Parent reference, child object
        a.sound(); // Calls Dog's method at runtime
    }
}
```

Output:

```
Dog barks
```

3. Real-World Example

- **Overloading analogy:**
  A person can **introduce** themselves in different ways — by name, by job, or by both.

- **Overriding analogy:**
  A base "Vehicle" has a method start().

  - In a **Car**, start() uses a key.
  - In an **Electric Car**, start() uses a button.
    The action is the same, but implementation differs.

4. Quick Comparison Table E

Feature	Compile-time Polymorphism	Runtime Polymorphism
Achieved By	Method Overloading	Method Overriding
Decision Time	Compile time	Runtime
Flexibility	Less flexible	More flexible
Binding	Static binding	Dynamic binding

Encapsulation

1.Definition: Encapsulation is the process of wrapping data (variables) and methods (functions) together into a single unit (class) and restricting direct access to the data from outside the class.
It's basically data hiding with controlled access using getters and setters.

2. Syntax:

```
class ClassName {
    // Step 1: Private variables (data hiding)
    private dataType variableName;

    // Step 2: Public getter method (read data)
    public dataType getVariableName() {
        return variableName;
    }

    // Step 3: Public setter method (write data)
    public void setVariableName(dataType value) {
        this.variableName = value;
    }
}
```

3. Rules of Encapsulation:

- **Variables should be private** → This hides them from outside classes.
- **Provide public getters & setters** → This gives controlled access to the private data.
- **Methods can have logic for validation** → Before setting a value, you can check rules.
- **Encapsulation is always implemented at the class level**.
- **You can restrict setting certain values** → e.g., age can't be negative.

3. Example Code:

```java
// Encapsulation Example
class Student {
    // Private data members
    private String name;
    private int age;

    // Getter for name
    public String getName() {
        return name;
    }

    // Setter for name
    public void setName(String name) {
        this.name = name;
    }

    // Getter for age
    public int getAge() {
        return age;
    }

    // Setter for age with validation
    public void setAge(int age) {
        if (age > 0) {  // validation
            this.age = age;
        } else {
            System.out.println("Age must be positive!");
        }
    }
}

// Main Class
public class Main {
    public static void main(String[] args) {
        Student s = new Student();

        s.setName("Rahul");
        s.setAge(21);

        System.out.println("Name: " + s.getName());
        System.out.println("Age: " + s.getAge());

        s.setAge(-5); // This will trigger validation message
    }
}
```

Output:

```
Name: Rahul
Age: 21
Age must be positive!
```

5. Real-World Example:

ATM Machine:

- Your bank account balance is **private** (hidden from you directly).
- You **request** money through the ATM (getter/setter methods).
- The ATM applies **validation**:
  - If your balance is sufficient → transaction allowed.
  - If not → transaction declined.

So, the **bank account class** hides the balance (private), and you can access it only through **controlled methods**.

Abstraction (abstract class, interface)

1. Definition: **Abstraction** is the process of **hiding implementation details** and **showing only the essential features** to the user.

Key point:

- **Encapsulation** → Hides data (variables).
- **Abstraction** → Hides implementation (methods).

2. Syntax:

Abstract Class

```java
abstract class ClassName {
    abstract void method1(); // abstract method (no body)
    void method2() {         // normal method
        System.out.println("Method with body");
    }
}
```

Interface

```java
interface InterfaceName {
    void method1(); // public & abstract by default
}
```

3. Rules

Abstract Class

1. Declared with the keyword abstract.
2. Can have abstract methods (without body) and concrete methods (with body).
3. Can have variables, constructors, static methods, final methods.
4. Cannot be instantiated directly — must be inherited.
5. Subclass must implement all abstract methods (unless subclass is also abstract).

Interface

1. Declared with the keyword interface.
2. All methods are public abstract by default (Java 7 and earlier).
3. From Java 8 → can have default and static methods with body.
4. From Java 9 → can have private methods.
5. Variables are public static final (constants) by default.
6. A class can implement multiple interfaces (supports multiple inheritance).

4. Example — Abstract Class

```java
abstract class Vehicle {
    abstract void start(); // abstract method

    void stop() { // concrete method
        System.out.println("Vehicle stopped");
    }
}

class Car extends Vehicle {
    @Override
    void start() {
        System.out.println("Car started with a key");
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Vehicle v = new Car(); // upcasting
        v.start();  // Car started with a key
        v.stop();   // Vehicle stopped
    }
}
```

Output:

```
Car started with a key
Vehicle stopped
```

5. Example — Interface:

```java
interface Animal {
    void sound(); // abstract method
}

class Dog implements Animal {
    public void sound() {
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal a = new Dog();
        a.sound(); // Dog barks
    }
}
```

Output:

```
Dog barks
```

6. Real-World Example:

Abstract Class

- Think of a **Bank** abstract class:
    - Abstract method: calculateInterest()
    - Concrete method: openAccount()
- Different banks implement calculateInterest() in their own way.

Interface

- Think of a **RemoteControl** interface:

- Methods: turnOn(), turnOff()

- TV, AC, Fan implement these methods differently but follow the same rules.

7. Quick Difference Table

Feature	Abstract Class	Interface
Methods	Abstract + Concrete	Abstract (default), Concrete (default/static from Java 8)
Variables	Instance, static, final	public static final only
Multiple Inheritance	No	Yes
Access Modifiers	Any	public only (methods)
Constructor	Yes	No

7.Access Modifiers & Packages

1.Definition: Access modifiers define how and where a variable, method, constructor, or class can be accessed from other parts of the program.

2.Types of Access Modifiers

Modifier	Within Same Class	Within Same Package	Subclass (Different Package)	Other Packages
private	✅ Yes	❌ No	❌ No	❌ No
default *(no keyword)*	✅ Yes	✅ Yes	❌ No	❌ No
protected	✅ Yes	✅ Yes	✅ Yes	❌ No
public	✅ Yes	✅ Yes	✅ Yes	✅ Yes

3.Syntax

```java
public class Test {
    private int privateVar = 10;
    int defaultVar = 20;            // default access
    protected int protectedVar = 30;
    public int publicVar = 40;

    public void display() {
        System.out.println("Private: " + privateVar);
    }
}
```

Key Points:

- **private** → Most restrictive, used for encapsulation.
- **default** → Package-private (if no modifier is given).
- **protected** → Visible to same package + subclasses in different packages.
- **public** → Accessible from anywhere.

4.Packages in Java:

Definition:A package is a way to group related classes, interfaces, and sub-packages together.
Think of it like a folder in your computer for organizing files.

5.Types of Packages:

- **Built-in Packages** → Provided by Java (e.g., java.util, java.io).
- **User-defined Packages** → Created by the programmer.

6.Syntax — Creating a Package:

```java
// File: MyClass.java
package mypackage;

public class MyClass {
    public void showMessage() {
        System.out.println("Hello from MyClass in mypackage!");
    }
}
```

7.Using a Package

```java
// File: Main.java
import mypackage.MyClass;

public class Main {
    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj.showMessage();
    }
}
```

Package Key Points

- Every .java file can have **only one package declaration**, and it must be the **first statement** in the file.
- **import** is used to use classes from other packages.

- Packages help avoid **naming conflicts** and make **code modular**.

8. Real-World Analogy

- **Access Modifiers** → Like a house's access system:
  - **private** → Only you can enter your bedroom.
  - **default** → Family members can enter the living room (same package).
  - **protected** → Relatives can enter if they are part of your family (subclass).
  - **public** → Anyone can enter the garden.
- **Packages** → Like folders in a cupboard, grouping related files together.

Exception Handling

1. Definition: Exception Handling in Java is a mechanism to handle runtime errors so that the normal flow of the program is maintained.

- It prevents the program from **crashing** when something unexpected happens.

2. Key Terms:

- **Exception** → An unwanted/unexpected event that occurs during the execution of a program.
- **Error** → Serious problem that cannot be recovered (e.g., OutOfMemoryError).
- **Throwable** → Superclass of all errors and exceptions.
- **Checked Exception** → Checked at compile time (e.g., IOException).
- **Unchecked Exception** → Occurs at runtime (e.g., ArithmeticException).

3. Exception Hierarchy:

```
java.lang.Object
    └─ java.lang.Throwable
        ├─ java.lang.Exception (Checked & Unchecked)
        └─ java.lang.Error (Serious problems)
```

4. Syntax for Exception Handling:

```java
try {
    // Code that may throw exception
} catch (ExceptionType e) {
    // Code to handle exception
} finally {
    // Code that will always execute (optional)
}
```

5.Example:

```java
public class Main {
    public static void main(String[] args) {
        try {
            int num = 10 / 0; // This will throw ArithmeticException
            System.out.println("This won't be printed");
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero: " + e.getMessage());
        } finally {
            System.out.println("Finally block executed");
        }
    }
}
```

Output:

```
Cannot divide by zero: / by zero
Finally block executed
```

6. Keywords in Exception Handling:

Keyword	Purpose
try	Defines a block of code to be tested for exceptions
catch	Defines a block to handle the exception
finally	Always executed, used for cleanup code
throw	Used to explicitly throw an exception
throws	Declares exceptions a method can throw

7. Example with throw and throws:

```java
class Test {
    void checkAge(int age) throws Exception {
        if (age < 18) {
            throw new Exception("Age must be 18 or above");
        }
        System.out.println("Access granted");
    }
}

public class Main {
    public static void main(String[] args) {
        try {
            new Test().checkAge(15);
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

8. Real-World Analogy:

Think of exception handling like **seatbelts in a car**:

- **try** → Driving normally (might face accident).
- **catch** → Seatbelt catches you during accident.
- **finally** → After the incident, you always stop and check damage.
- **throw** → You intentionally raise a red flag.
- **throws** → Warning label that says "This might be risky."

Checked Exception & Unchecked Exception

Checked Exception

- Exceptions that are **checked at compile time**.
- If a method might throw one, the compiler forces you to **handle it** (using try-catch or throws).
- Usually caused by **external factors** like files, database, or network.
- **Subclass of** java.lang.Exception (but NOT RuntimeException).

Unchecked Exception

- Exceptions that are **checked at runtime** (compiler doesn't check).
- No need to handle them explicitly (but you can).
- Usually caused by **programming mistakes** like dividing by zero or null pointer access.
- **Subclass of** java.lang.RuntimeException.

Feature	Checked Exception	Unchecked Exception
Checked at	Compile time	Runtime
Compiler enforcement	Must handle using `try-catch` or `throws`	No compiler requirement
Cause	External factors (I/O, DB, network)	Programming errors (null, index out of range)
Examples	`IOException`, `SQLException`, `FileNotFoundException`	`ArithmeticException`, `NullPointerException`, `ArrayIndexOutOfBoundsException`
Superclass	`Exception` (excluding `RuntimeException`)	`RuntimeException`

Types of Exceptions:

1. Main Categories

All exceptions in Java come from **java.lang.Throwable**

```
java.lang.Object
   └─ java.lang.Throwable
        ├─ java.lang.Exception   → Recoverable issues
        │       ├─ Checked Exceptions
        │       └─ Unchecked Exceptions
        └─ java.lang.Error       → Serious issues (non-recoverable)
```

2. Types of Exceptions

A. Checked Exceptions (Compile-Time Exceptions)

- Checked **by compiler** at compile time.
- Must be handled using try-catch or declared using throws.
- Usually caused by **external resources** (files, DB, network).

Examples:

- ❖ IOException
- ❖ SQLException
- ❖ FileNotFoundException
- ❖ ClassNotFoundException
- ❖ InterruptedException

B. Unchecked Exceptions (Runtime Exceptions)

- Not checked by compiler at compile time.
- Occur during program execution.
- Usually caused by programming mistakes.

Examples:

- ❖ ArithmeticException
- ❖ NullPointerException
- ❖ ArrayIndexOutOfBoundsException
- ❖ NumberFormatException
- ❖ IllegalArgumentException

C. Errors

- Serious problems beyond the control of the programmer.
- Usually **not handled** (program crashes).
- Related to **system failures**.

Examples:

- ❖ OutOfMemoryError
- ❖ StackOverflowError
- ❖ VirtualMachineError
- ❖ AssertionError

Real-World Analogy:

- **Checked Exception** → Forgetting your train ticket (ticket checker stops you before boarding).
- **Unchecked Exception** → Tripping over a rock while walking (unexpected, but could've been avoided).
- **Error** → Earthquake during the walk (beyond your control).

Collections Framework

1. Definition: The Java Collections Framework is a set of classes and interfaces in java.util package that provides ready-made data structures (like lists, sets, maps) and algorithms (like sorting, searching) to store, manipulate, and retrieve objects efficiently.

2. Collections Framework Hierarchy:

```
Iterable (interface)
   ↑
Collection (interface)
   ├── List (interface) → ArrayList, LinkedList, Vector, Stack
   ├── Set (interface) → HashSet, LinkedHashSet, TreeSet
   └── Queue (interface) → PriorityQueue, ArrayDeque

Map (interface) [Separate hierarchy]
   ├── HashMap, LinkedHashMap, TreeMap, Hashtable
```

3. Core Interfaces

Interface	Description	Example Implementations
List	Ordered, allows duplicates	ArrayList, LinkedList, Vector
Set	Unordered, no duplicates	HashSet, LinkedHashSet, TreeSet
Queue	Follows FIFO, can be priority-based	PriorityQueue, ArrayDeque
Map	Key-value pairs, keys unique	HashMap, LinkedHashMap, TreeMap

4. Example Usage

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        // List example
        List<String> names = new ArrayList<>();
        names.add("Rahul");
        names.add("Priya");
        names.add("Rahul"); // allows duplicates
        System.out.println("List: " + names);

        // Set example
        Set<String> uniqueNames = new HashSet<>(names); // removes duplicates
        System.out.println("Set: " + uniqueNames);

        // Map example
        Map<Integer, String> map = new HashMap<>();
        map.put(1, "Apple");
        map.put(2, "Banana");
        map.put(3, "Cherry");
        System.out.println("Map: " + map);
    }
}
```

Output:

```
List: [Rahul, Priya, Rahul]
Set: [Rahul, Priya]
Map: {1=Apple, 2=Banana, 3=Cherry}
```

5. Real-World Analogy

- **List** → A guest list at a party (order matters, duplicates allowed).
- **Set** → A list of unique voter IDs (no duplicates, order doesn't matter).
- **Queue** → People standing in a ticket line (FIFO).
- **Map** → Dictionary (word = key, definition = value).

6. Common Interview Points

- ArrayList vs LinkedList
- HashSet vs TreeSet
- Fail-fast vs Fail-safe Iterators
- Synchronized Collections vs Concurrent Collections

1. ArrayList vs LinkedList

1.Definition:

- **ArrayList** → Resizable array implementation of List
- **LinkedList** → Doubly linked list implementation of List

Feature	ArrayList	LinkedList
Data Structure	Dynamic array	Doubly linked list
Access (get/set)	Fast (O(1))	Slow (O(n))
Insertion/Deletion in middle	Slow (shifting elements)	Fast (changing links)
Memory usage	Less memory	More memory (extra node pointers)

Example:

```
List<String> arrayList = new ArrayList<>();
arrayList.add("A");
arrayList.add("B");

List<String> linkedList = new LinkedList<>();
linkedList.add("X");
linkedList.add("Y");

System.out.println("ArrayList: " + arrayList);
System.out.println("LinkedList: " + linkedList);
```

2. HashSet vs TreeSet

1. Definition:

- **HashSet** → Implements Set using hash table (unordered).
- **TreeSet** → Implements Set using a Red-Black Tree (sorted order).

Feature	HashSet	TreeSet
Order	No order	Sorted (natural order)
Performance	Faster (O(1))	Slower (O(log n))
Null elements	Allows one null	Does not allow null

Example:

```
Set<String> hashSet = new HashSet<>();
hashSet.add("B");
hashSet.add("A");

Set<String> treeSet = new TreeSet<>();
treeSet.add("B");
treeSet.add("A");

System.out.println("HashSet: " + hashSet);
System.out.println("TreeSet: " + treeSet);
```

Output:

```
HashSet: [A, B]  (order not guaranteed)
TreeSet: [A, B]  (sorted order)
```

3. HashMap vs Hashtable

Definition:

- **HashMap** → Non-synchronized map; allows null keys & values.
- **Hashtable** → Synchronized map; does not allow null keys or values.

Feature	HashMap	Hashtable
Thread safety	Not synchronized	Synchronized
Null key	Allows 1 null key	Does not allow null key
Null values	Allows multiple null values	Does not allow null values
Performance	Faster	Slower (due to synchronization)

Example:

```
Map<Integer, String> hashMap = new HashMap<>();
hashMap.put(1, "One");
hashMap.put(null, "NullKey");

Map<Integer, String> hashtable = new Hashtable<>();
hashtable.put(1, "One");
// hashtable.put(null, "NullKey"); // Throws NullPointerException

System.out.println("HashMap: " + hashMap);
System.out.println("Hashtable: " + hashtable);
```

4. Fail-fast vs Fail-safe Iterators

Definition:

- **Fail-fast** → Throws ConcurrentModificationException if collection is modified while iterating.
- **Fail-safe** → Works on a copy of the collection, so no exception occurs.

Feature	Fail-fast	Fail-safe
Modification during iteration	Throws exception	No exception
Examples	ArrayList, HashMap	CopyOnWriteArrayList, ConcurrentHashMap
Performance	Faster	Slower (due to copying)

Example (Fail-fast):

```
List<String> list = new ArrayList<>();
list.add("A");
list.add("B");

for (String s : list) {
    list.add("C"); // Causes ConcurrentModificationException
}
```

Example (Fail-safe):

```
CopyOnWriteArrayList<String> list = new CopyOnWriteArrayList<>();
list.add("A");
list.add("B");

for (String s : list) {
    list.add("C"); // No exception
}
System.out.println(list);
```

5. Synchronized Collections vs Concurrent Collections

Definition:

- **Synchronized Collections** → Achieved by using Collections.synchronizedXXX(); locks the entire collection.
- **Concurrent Collections** → Part of java.util.concurrent; allow thread-safe operations without locking the whole collection.

Feature	Synchronized Collections	Concurrent Collections
Locking	Whole collection locked	Partial locking / no locking
Performance	Slower	Faster in multi-threaded
Examples	Collections.synchronizedList(), Vector	ConcurrentHashMap, CopyOnWriteArrayList

Example (Synchronized Collection):

```
List<String> syncList = Collections.synchronizedList(new ArrayList<>());
syncList.add("A");
syncList.add("B");
```

Example (Concurrent Collection):

```
ConcurrentHashMap<Integer, String> map = new ConcurrentHashMap<>();
map.put(1, "One");
map.put(2, "Two");
```

Comparable vs Comparator

1. Definition:

Comparable:

- An interface used to **define the natural ordering** of objects.
- The class itself **implements** Comparable and overrides compareTo() method.
- Only **one** natural ordering can be defined for a class.

Comparator

- An interface used to **define custom ordering** of objects.
- A separate class **implements** Comparator and overrides compare() method.
- You can have **multiple** comparator implementations for different sorting logic.

2. Method Signatures:

Interface	Method Signature
Comparable	int compareTo(T o)
Comparator	int compare(T o1, T o2)

3. Difference Table:

Feature	Comparable	Comparator
Package	java.lang	java.util
Method	compareTo(Object)	compare(Object, Object)
Sorting	Natural ordering	Custom ordering
Changes to class	Class needs to implement Comparable	No change to original class
No. of sorting logics	One per class	Multiple possible

4. Comparable Example (Natural Order):

```
import java.util.*;

class Student implements Comparable<Student> {
    String name;
    int age;

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

    @Override
    public int compareTo(Student s) {
        return this.age - s.age; // sort by age
    }
```

```
    @Override
    public String toString() {
        return name + " - " + age;
    }
}

public class Main {
    public static void main(String[] args) {
        List<Student> list = new ArrayList<>();
        list.add(new Student("Rahul", 22));
        list.add(new Student("Priya", 20));

        Collections.sort(list);
        System.out.println(list);
    }
}
```

Output:

```
[Priya - 20, Rahul - 22]
```

5. Comparator Example (Custom Order):

```
import java.util.*;

class Student {
    String name;
    int age;

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

```java
    @Override
    public String toString() {
        return name + " - " + age;
    }
}

class NameComparator implements Comparator<Student> {
    @Override
    public int compare(Student s1, Student s2) {
        return s1.name.compareTo(s2.name); // sort by name
    }
}

public class Main {
    public static void main(String[] args) {
        List<Student> list = new ArrayList<>();
        list.add(new Student("Rahul", 22));
        list.add(new Student("Priya", 20));

        Collections.sort(list, new NameComparator());
        System.out.println(list);
    }
}
```

Output:

```
[Priya - 20, Rahul - 22]
```

6. Real-World Analogy:

- **Comparable** → Like a student roll number — the school decides the natural order.
- **Comparator** → Like sorting students by height, marks, or name — you decide custom orders.

JAVA memory management System

1.Definition

Java Memory Management is the process of allocating memory to variables, objects, and program instructions, and automatically freeing it when it's no longer needed.
It is handled by the **JVM** using **heap, stack, metaspace** areas, and **garbage collection** to reclaim unused memory.

2. Syntax

(There is no direct "syntax" for memory management because it's automatic in Java, but you can control it indirectly.)

Common JVM commands for memory settings:

1. Stack Memory

Definition:

Stack memory is the part of JVM memory that stores **method calls, local variables, and object references**.
It works on **LIFO (Last In, First Out)** principle.

Key Points:

- Each **thread** gets its own stack.
- Stores:
  - Method execution data.
  - Local variables (primitives & references).
- Memory is **freed automatically** when a method ends.
- Faster access compared to heap.
- Cannot store objects themselves (only **references** to objects stored in heap).

2. Heap Memory

Definition:

Heap memory is the part of JVM memory used to **store objects and class instances**.
It is shared by all threads and managed by **Garbage Collector (GC)**.

Key Points:

- Divided into:
  - **Young Generation** → New objects (Eden + Survivor spaces).
  - **Old Generation** → Long-lived objects.
  - **Metaspace** → Class metadata (Java 8+).
- Objects remain until no reference points to them.
- Access is slower than stack.
- GC automatically removes unused objects.

3. Diagram

```
┌──────────────────────────┐
│        Stack             │   → Per Thread
│ ┌────────────────────┐   │
│ │ Method Call Frames  │   │
│ │ Local Variables     │   │
│ │ Object References   │   │
│ └────────────────────┘   │
└──────────────────────────┘
          │
          ▼
┌──────────────────────────┐
│         Heap             │   → Shared by all threads
│ ┌────────────────────┐   │
│ │ Objects             │   │
│ │ Arrays              │   │
│ │ Class Instances     │   │
│ └────────────────────┘   │
└──────────────────────────┘
```

4. Example Code

```java
public class StackHeapExample {
    public static void main(String[] args) {
        int num = 10; // Stored in Stack
        String text = new String("Hello"); // text ref in Stack, object in Heap

        display(text);
    }

    static void display(String message) {
        String localMessage = message + " World"; // localMessage in Stack
        System.out.println(localMessage);
    }
}
```

5. Step-by-step memory flow

1. **num** → stored directly in Stack (primitive value 10).
2. **text** → reference in Stack, "Hello" object in Heap.
3. When display() runs:
    - message → reference in Stack pointing to "Hello" in Heap.
    - localMessage → reference in Stack, "Hello World" object in Heap.
4. After method ends → stack frames are removed, heap objects remain until GC removes them.

Garbage Collection

1. Definition

Garbage Collection in Java is the **automatic process of freeing memory** by removing objects that are no longer reachable or used in a program.
It is handled by the **Java Virtual Machine (JVM)** so developers don't need to manually deallocate memory (like free() in C).

2. Why GC is Needed

- Prevents **Memory Leaks** (holding unused objects in memory).
- Improves application performance by reclaiming unused space.
- Simplifies coding — no manual memory management.

3. How GC Works

1. **Object Creation** → Stored in Heap.

2. **Reference Lost** → Object becomes unreachable.
3. **GC Runs** → Finds unreachable objects and removes them.
4. **Memory Freed** → Heap space is reused.

4. GC Types in Java

- **Minor GC** → Cleans Young Generation (short-lived objects).
- **Major GC** → Cleans Old Generation (long-lived objects).
- **Full GC** → Cleans entire Heap (Young + Old Gen).

5. Syntax to Suggest GC

```
System.gc(); // Suggest garbage collection (not guaranteed)
Runtime.getRuntime().gc(); // Another way
```

6. Example

```
public class GarbageCollectionDemo {
    public static void main(String[] args) {
        GarbageCollectionDemo obj = new GarbageCollectionDemo();

        // Now obj points to null → object becomes eligible for GC
        obj = null;

        // Suggest garbage collection
        System.gc();
    }

    @Override
    protected void finalize() {
        System.out.println("Object is garbage collected");
    }
}
```

7. When Objects Become Eligible for GC

- Reference is set to null.
- Reference is reassigned to another object.
- Object is created inside a method and method ends.
- Object is in a scope that ends (stack frame removed).

8. Best Practices

- Avoid forcing GC with System.gc() in production — JVM is smarter at deciding.
- Always close unused resources (files, DB connections, sockets).
- Use profiling tools (VisualVM, JConsole) to monitor memory usage.

Java 8+ features

1.Java 8 Major Features (Released in 2014)

1. Lambda Expressions

- Allows you to write short, inline functions.
- **Syntax:**

```
(parameter) -> expression
```

Example:

```
List<String> names = Arrays.asList("A", "B", "C");
names.forEach(n -> System.out.println(n));
```

2. Functional Interfaces

- Interface with exactly **one abstract method**.
- Annotated with @FunctionalInterface.

Example:

```
@FunctionalInterface
interface MyFunc {
    void display();
}
```

3. Streams API

- Process collections in a functional style (filter, map, reduce).

Example:

```
List<Integer> nums = Arrays.asList(1, 2, 3, 4);
nums.stream()
    .filter(n -> n % 2 == 0)
    .forEach(System.out::println);
```

4. Default & Static Methods in Interfaces

- Interfaces can now have method implementations.

```java
interface MyInterface {
    default void greet() { System.out.println("Hello"); }
    static void info() { System.out.println("Static Method"); }
}
```

5. Method References

- Shorthand for calling existing methods.

```java
names.forEach(System.out::println);
```

6. Optional Class

- Avoids NullPointerException.

```java
Optional<String> name = Optional.ofNullable(null);
System.out.println(name.orElse("Default"));
```

7. New Date & Time API (java.time)

- Better than old java.util.Date.

```java
LocalDate today = LocalDate.now();
LocalTime time = LocalTime.now();
```

8. Collectors API

- Collect stream results into collections or other forms

```java
List<String> upper = names.stream()
    .map(String::toUpperCase)
    .collect(Collectors.toList());
```

9. Nashorn JavaScript Engine

- Run JavaScript inside Java (deprecated in later versions).

Multithreading

1. Definition

Multithreading in Java is the ability of a program to execute **two or more threads concurrently**, so tasks can run in parallel and improve performance.

A **thread** is the smallest unit of execution in a process.
Java supports multithreading through the **java.lang.Thread** class and **java.lang.Runnable** interface.

2. Advantages

- Efficient use of CPU.
- Faster execution for multiple tasks.
- Better performance in multi-core processors.
- Useful for background tasks (file downloading, I/O operations, etc.).

3. Life Cycle of a Thread

States:

1. **New** → Thread created but not started.
2. **Runnable** → Ready to run, waiting for CPU.
3. **Running** → Currently executing.
4. **Waiting / Timed Waiting** → Temporarily inactive.
5. **Terminated** → Finished execution.

4. Ways to Create a Thread

A. Extending Thread Class

```java
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread running: " + Thread.currentThread().getName());
    }
}

public class ThreadExample {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        t1.start(); // Starts a new thread
    }
}
```

B. Implementing Runnable Interface

```java
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Thread running: " + Thread.currentThread().getName());
    }
}

public class RunnableExample {
    public static void main(String[] args) {
        Thread t1 = new Thread(new MyRunnable());
        t1.start();
    }
}
```

C. Using Lambda (Java 8+)

```java
public class LambdaThread {
    public static void main(String[] args) {
        Thread t1 = new Thread(() -> {
            System.out.println("Thread running: " + Thread.currentThread().getName());
        });
        t1.start();
    }
}
```

5. Common Thread Methods

Method	Description
start()	Starts a new thread (calls run() internally)
run()	Contains thread code
sleep(ms)	Pauses thread for given time
join()	Waits for a thread to finish
yield()	Gives chance to other threads
setPriority()	Sets thread priority (1 to 10)

6. Example with Multiple Threads

```java
class Task extends Thread {
    public void run() {
        for(int i = 1; i <= 5; i++) {
            System.out.println(Thread.currentThread().getName() + " - " + i);
            try { Thread.sleep(500); } catch(Exception e) {}
        }
    }
}

public class MultiThreadDemo {
    public static void main(String[] args) {
        Task t1 = new Task();
        Task t2 = new Task();
        t1.start();
        t2.start();
    }
}
```

7. Multithreading Issues

- **Race Condition** – Multiple threads modifying shared data.
- **Deadlock** – Two threads waiting for each other forever.
- **Starvation** – Thread waits too long for CPU time.

8. Synchronization

- To avoid problems in shared data, use:

```
synchronized void methodName() {
    // only one thread at a time
}
```

Synchronization basics

1. Definition

Synchronization in Java is a mechanism to **control access to shared resources** by multiple threads.
It ensures that only **one thread at a time** can access a block or method, preventing **data inconsistency**.

2. Why Synchronization is Needed

Without synchronization:

- Two or more threads may **modify shared data at the same time**.
- Leads to **race conditions** and **unpredictable results**.

Example of a **problem without synchronization**:

```
class Counter {
    int count = 0;
    void increment() { count++; }
}

public class NoSyncExample {
    public static void main(String[] args) throws Exception {
        Counter c = new Counter();

        Thread t1 = new Thread(() -> {
            for(int i=0; i<1000; i++) c.increment();
        });

        Thread t2 = new Thread(() -> {
            for(int i=0; i<1000; i++) c.increment();
        });
```

```
        t1.start(); t2.start();
        t1.join(); t2.join();

        System.out.println("Count: " + c.count); // May be less than 2000 due to race condition
    }
}
```

3. Synchronization Types in Java

A. Synchronized Method

Only one thread can call this method at a time **on the same object**.

```
class Counter {
    int count = 0;
    synchronized void increment() { count++; }
}
```

B. Synchronized Block

Locks only the specified code block, not the whole method (more efficient).

```
class Counter {
    int count = 0;
    void increment() {
        synchronized(this) {
            count++;
        }
    }
}
```

C. Static Synchronization

When method is static, lock is on the **class** object, not an instance.

```
class Counter {
    static int count = 0;
    public static synchronized void increment() {
        count++;
    }
}
```

4. Example with Synchronization

```java
class Counter {
    int count = 0;
    synchronized void increment() { count++; }
}

public class SyncExample {
    public static void main(String[] args) throws Exception {
        Counter c = new Counter();

        Thread t1 = new Thread(() -> { for(int i=0; i<1000; i++) c.increment(); });
        Thread t2 = new Thread(() -> { for(int i=0; i<1000; i++) c.increment(); });

        t1.start(); t2.start();
        t1.join(); t2.join();

        System.out.println("Count: " + c.count); // Always 2000
    }
}
```

5. Key Points

- **Locking Mechanism**: Every object has an intrinsic lock (monitor).
- **Thread Safety**: Synchronization ensures one thread works on the resource at a time.
- **Performance Cost**: Too much synchronization can slow down the program.
- **Avoid Deadlocks**: Use consistent locking order and minimal lock scope.

ADVANCE JAVA

A. Java Database Connectivity (JDBC) — Deep Dive:

1. What is JDBC?

Definition:
JDBC is an API in Java that allows Java applications to connect to and interact with relational databases like MySQL, PostgreSQL, Oracle, etc.

- Think of it as a **bridge** between Java code and a database.

Package:

```
import java.sql.*;
```

2. JDBC Architecture

JDBC follows a **two-layer architecture**:

A. JDBC API Layer

- Interfaces and classes for developers to write database code.
- Examples: Connection, Statement, ResultSet.

B. JDBC Driver Layer

- Vendor-specific implementation to connect to the database.
- Types of Drivers:
    1. **Type-1:** JDBC-ODBC bridge (obsolete)
    2. **Type-2:** Native API driver
    3. **Type-3:** Network Protocol driver
    4. **Type-4:** Thin driver (Pure Java, most used today)

3. JDBC Steps (Core Flow)

1.Load the Driver:

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

2.Establish the Connection:

```
Connection con = DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/mydb", "root", "password");
```

3. Create a SQL Statement (or PreparedStatement):

```
Statement st = con.createStatement();
```

4. Execute Query:

```
ResultSet rs = st.executeQuery("SELECT * FROM employees");
```

5. Process Result:

```
while(rs.next()) {
    System.out.println(rs.getInt("id") + " " + rs.getString("name"));
}
```

4. JDBC Important Interfaces

Interface	Purpose
Driver	Interface for database drivers
Connection	Represents a connection to the database
Statement	Executes SQL queries (simple)
PreparedStatement	Executes parameterized queries (faster, prevents SQL injection)
CallableStatement	Executes stored procedures
ResultSet	Stores query results and allows iteration

5. Advanced JDBC Concepts

A. **PreparedStatement:**

- Safer and faster than Statement.

```
PreparedStatement ps = con.prepareStatement("INSERT INTO emp VALUES (?, ?)");
ps.setInt(1, 101);
ps.setString(2, "John");
ps.executeUpdate();
```

B. **Batch Processing:**

- Saves time by executing multiple queries at once.

```
PreparedStatement ps = con.prepareStatement("INSERT INTO emp VALUES (?, ?)");
ps.setInt(1, 102); ps.setString(2, "Alice"); ps.addBatch();
ps.setInt(1, 103); ps.setString(2, "Bob"); ps.addBatch();
ps.executeBatch();
```

C. Transactions

- Used when you want multiple queries to either **all succeed** or **all fail**.

```
con.setAutoCommit(false);
try {
    ps1.executeUpdate();
    ps2.executeUpdate();
    con.commit();
} catch(Exception e) {
    con.rollback();
}
```

D. Connection Pooling

Instead of creating a new connection every time (which is slow), reuse connections from a pool.

- Libraries: **HikariCP**, **Apache DBCP**, **C3P0**
- Example (with HikariCP):

```
HikariDataSource ds = new HikariDataSource();
ds.setJdbcUrl("jdbc:mysql://localhost:3306/mydb");
ds.setUsername("root");
ds.setPassword("password");
Connection con = ds.getConnection();
```

E. CallableStatement (Stored Procedures)

```
CallableStatement cs = con.prepareCall("{call getEmployee(?)}");
cs.setInt(1, 101);
ResultSet rs = cs.executeQuery();
```

✅ **Real-Life Analogy:**
Think of JDBC like ordering food from a restaurant:

- **Driver** = Waiter who understands both your language and the kitchen's
- **Connection** = Table where you sit
- **Statement** = Menu order form
- **SQL Query** = What you order

- **ResultSet** = The food served
- **close()** = Leaving the table and freeing resources

SERVLETS

1. What is a Servlet?

A **Servlet** is a Java program that runs on a web server, receives HTTP requests from clients (usually a browser), processes them, and sends back HTTP responses.

- Part of **Java EE / Jakarta EE** specification.
- Runs inside a **Servlet Container** (e.g., Apache Tomcat, Jetty).
- Mainly used for **dynamic web content**.

2. How Servlets Work (Request-Response Cycle)

Flow:

1. Client (browser) sends an HTTP request (e.g., clicking a link, submitting a form).
2. Servlet container receives the request.
3. Servlet processes it in service() method.
4. Servlet sends an HTTP response (HTML, JSON, XML, etc.).
5. Response goes back to client.

3. Servlet Lifecycle

A servlet goes through **three main stages**:

1. **Initialization** (init() method)
    - Called only once when servlet is first loaded.
    - Good place for resource setup.
2. **Service** (service() method)
    - Called for every request.
    - Dispatches to doGet() or doPost() depending on HTTP method.
3. **Destruction** (destroy() method)
    - Called when servlet is removed from service.
    - Good place for cleanup.

4. Basic Servlet Example

HelloServlet.java

```java
import java.io.*;
import jakarta.servlet.*;
import jakarta.servlet.http.*;

public class HelloServlet extends HttpServlet {
    @Override
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<h1>Hello from Servlet!</h1>");
    }
}
```

web.xml Mapping (if not using annotations)

```xml
<servlet>
    <servlet-name>hello</servlet-name>
    <servlet-class>HelloServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>hello</servlet-name>
    <url-pattern>/hello</url-pattern>
</servlet-mapping>
```

Annotation-based Mapping (Preferred in modern Java)

```java
@WebServlet("/hello")
public class HelloServlet extends HttpServlet {
    // doGet/doPost here
}
```

5. Key Servlet Classes & Interfaces

Interface/Class	Purpose
Servlet	Base interface for all servlets
GenericServlet	Abstract class implementing Servlet
HttpServlet	Abstract class for HTTP-specific servlets
HttpServletRequest	Represents the HTTP request data
HttpServletResponse	Represents the HTTP response data
ServletConfig	Servlet-specific configuration
ServletContext	Application-wide configuration

6. Handling Request Data

From **HTML form**:

```
<form action="login" method="post">
    <input type="text" name="username">
    <input type="password" name="password">
    <input type="submit">
</form>
```

From **Servlet**:

```
String user = request.getParameter("username");
String pass = request.getParameter("password");
```

7. Session Management in Servlets

A. Cookies:

```
Cookie c = new Cookie("user", "John");
response.addCookie(c);
```

B. HttpSession:

```
HttpSession session = request.getSession();
session.setAttribute("username", "John");
```

C. URL Rewriting:

```
response.sendRedirect("dashboard?user=John");
```

8. Filters & Listeners

- **Filters**: Pre-process and post-process requests (e.g., logging, authentication).
- **Listeners**: Event handlers for application, session, or request lifecycle.

9. Advantages of Servlets

- Platform-independent (pure Java).
- High performance compared to CGI.
- Reusable and maintainable.
- Can integrate with JDBC, JSP, frameworks like Spring MVC.

✅ Real-Life Analogy:

Think of a **Servlet** as the **receptionist** of a company:

- Receives your request (what you want).
- Passes it to the right department (business logic).
- Prepares the response (document, info, or report).
- Sends it back to you.

1. Where They Come From

Both **doGet()** and **doPost()** are methods from the **HttpServlet** class, which is part of the Servlet API (jakarta.servlet.http.HttpServlet).

- They handle **specific HTTP request methods** (GET or POST).
- They are **overridden** in your servlet to provide custom logic.

2. doGet() Method

Definition:

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
```

Purpose:

- Handles **HTTP GET requests** (default method when you type a URL in a browser or click a link).
- Typically used for **reading/fetching data** without modifying server state.
- Parameters are sent as **query strings** in the URL.

Example URL:

```
http://localhost:8080/app/user?id=101
```

In Servlet:

```
String id = request.getParameter("id");
```

Example Code:

```
@WebServlet("/getExample")
public class GetExampleServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

        String name = request.getParameter("name");
        response.setContentType("text/html");
        response.getWriter().println("<h1>Hello, " + name + " (GET)</h1>");
    }
}
```

3. doPost() Method

Definition:

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
```

Purpose:

- Handles **HTTP POST requests** (usually from forms with method="post").
- Used for **sending sensitive or large data**, like login credentials, file uploads, form submissions.
- Parameters are sent in the **request body** (not visible in URL).

Example HTML Form:

```
<form action="postExample" method="post">
    Name: <input type="text" name="name">
    <input type="submit">
</form>
```

Example Servlet:

```java
@WebServlet("/postExample")
public class PostExampleServlet extends HttpServlet {
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

        String name = request.getParameter("name");
        response.setContentType("text/html");
        response.getWriter().println("<h1>Hello, " + name + " (POST)</h1>");
    }
}
```

4. Key Differences Between doGet() and doPost()

Feature	doGet()	doPost()
Purpose	Fetch/read data	Send/create/update data
Data in URL?	Yes (query string)	No (hidden in body)
Data Limit	Limited (about 2048 characters)	Larger (depends on server)
Caching	Can be cached	Not cached by default
Security	Less secure (data visible in URL)	More secure (data hidden)
Idempotent?	Yes (safe to repeat without changing data)	No (repeating can change data)

JSP (Java Server Pages)

1. Definition of JSP

JSP (JavaServer Pages) is a technology used to create **dynamic web pages** by embedding Java code into HTML.

- **Part of**: Java EE (now Jakarta EE).
- **Purpose**: Simplify web page creation by mixing HTML with Java code.
- **Runs inside**: A servlet container (like Apache Tomcat).
- **Extension**: .jsp

When you run a JSP:

1. The server **compiles JSP → Servlet** internally.
2. The servlet generates the HTML/response.
3. The HTML is sent back to the client's browser.

2. Syntax

JSP uses **special tags** to embed Java code in HTML:

Tag Type	Syntax	Purpose
Scriptlet	<% code %>	Write Java code inside JSP
Expression	<%= expression %>	Output a value directly into HTML
Declaration	<%! code %>	Declare methods or variables
Directive	<%@ directive %>	Provide page settings/import classes
Comment	<%-- comment --%>	JSP comments (not sent to client)

3. Example – Basic JSP:

```
<%@ page language="java" contentType="text/html; charset=UTF-8" %>
<%@ page import="java.util.Date" %>
<html>
<head>
<title>JSP Example</title>
</head>
<body>
    <h1>Welcome to JSP!</h1>
    <p>Current Date and Time: <%= new Date() %></p>

    <%
        String name = "Santhosh";
        out.println("<p>Hello, " + name + "!</p>");
    %>
</body>
</html>
```

4. How JSP Works (Life Cycle):

- **Translation**: JSP file → Servlet .java file.
- **Compilation**: Servlet .java file → .class bytecode.
- **Execution**: Servlet handles request/response.
- **Response Sent**: HTML sent back to browser.

5. Advantages of JSP

- Easier to write HTML with embedded Java.
- Automatically converted into servlets — no need to manually write HttpServlet classes.
- Supports **tag libraries** (JSTL) for clean code.
- Can separate presentation (HTML) from business logic.

6. Example – Form Handling in JSP

form.jsp:

```
<html>
<body>
    <form action="welcome.jsp" method="post">
        Name: <input type="text" name="username">
        <input type="submit" value="Submit">
    </form>
</body>
</html>
```

welcome.jsp:

```
<%@ page language="java" contentType="text/html; charset=UTF-8" %>
<html>
<body>
    <h2>Hello, <%= request.getParameter("username") %>!</h2>
</body>
</html>
```

Ways to Implement Java in JSP:

A. Scriptlets – <% ... %>

Used for **writing Java statements** inside the JSP.

```
<%
    String name = "Santhosh";
    int age = 22;
    if (age >= 18) {
        out.println("<p>" + name + " is an adult.</p>");
    } else {
        out.println("<p>" + name + " is a minor.</p>");
    }
%>
```

B. Expressions – <%= ... %>

Used for **outputting a value directly** into the HTML.

```
<p>Today's date is: <%= new java.util.Date() %></p>
```

No need to write out.print() — JSP does it automatically.

C. Declarations – <%! ... %>

Used to **declare methods or variables** at the **class level** of the JSP-generated servlet.

```
<%!
    public String greet(String name) {
        return "Hello, " + name;
    }
%>

<p><%= greet("Santhosh") %></p>
```

Complete Example – Java in JSP

```
<%@ page import="java.util.*" %>
<html>
<body>
    <h2>Java in JSP Example</h2>

    <!-- Declaration -->
    <%!
        public String greet(String name) {
            return "Welcome, " + name + "!";
        }
    %>

    <!-- Scriptlet -->
    <%
        String user = "Santhosh";
        Date today = new Date();
    %>

    <!-- Expression -->
    <p><%= greet(user) %></p>
    <p>Today's date: <%= today %></p>
</body>
</html>
```

Best Practice

Even though JSP allows Java code directly inside HTML:

- **Avoid putting business logic in JSP** — keep it in Servlets or Java classes.
- Use **JSTL (JSP Standard Tag Library)** or **Expression Language (EL)** for cleaner code.
- Use JSP mainly for **presentation**, not heavy Java processing.

Hibernate / JPA (ORM basics)

1. What is ORM?

ORM (Object-Relational Mapping) is a technique that lets you **map Java objects to database tables** without writing SQL manually for CRUD operations.

- **Object** → Java Class
- **Relational** → Database Table
- **Mapping** → Connection between them

2. JPA and Hibernate

- **JPA**: *Java Persistence API* — a **specification** (set of rules) for ORM in Java.
- **Hibernate**: A **framework** that implements JPA (plus extra features).
- Think of JPA as an **interface**, Hibernate as a **class that implements it**.

3. Why use Hibernate / JPA?

✅ No need to write boilerplate SQL for CRUD
✅ Database-independent (change DB with minimal code changes)
✅ Easy mapping between Java classes and DB tables
✅ Supports caching, lazy loading, relationships (OneToMany, ManyToMany)

4. Basic Hibernate Flow

1. **Create Entity Class** (mapped to DB table)
2. **Configure Hibernate** (hibernate.cfg.xml or persistence.xml)
3. **Use Session / EntityManager** to perform DB operations

5. Example – Hibernate ORM

User.java (Entity)

```java
import jakarta.persistence.*;

@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String name;
    private String email;

    // Getters and Setters
}
```

hibernate.cfg.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <!-- DB Connection -->
        <property name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</property>
        <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/testdb</property>
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.connection.password">root</property>

        <!-- Hibernate properties -->
        <property name="hibernate.dialect">org.hibernate.dialect.MySQL8Dialect</property>
        <property name="hibernate.hbm2ddl.auto">update</property>
        <property name="show_sql">true</property>

        <!-- Entity Mapping -->
        <mapping class="User"/>
    </session-factory>
</hibernate-configuration>
```

Main.java

```java
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class Main {
    public static void main(String[] args) {
        SessionFactory factory = new Configuration().configure().buildSessionFactory();
        Session session = factory.openSession();

        session.beginTransaction();

        User user = new User();
        user.setName("Santhosh");
        user.setEmail("santhosh@example.com");

        session.save(user);

        session.getTransaction().commit();
        session.close();
        factory.close();
    }
}
```

6. JPA Version (EntityManager)

If using JPA:

```java
EntityManagerFactory emf = Persistence.createEntityManagerFactory("my-pu");
EntityManager em = emf.createEntityManager();

em.getTransaction().begin();
em.persist(new User("Santhosh", "santhosh@example.com"));
em.getTransaction().commit();
```

7. Key ORM Annotations

Annotation	Purpose
@Entity	Marks class as DB table mapping
@Table(name="table_name")	Maps class to specific table
@Id	Primary key
@GeneratedValue	Auto-generate primary key
@Column	Map field to column
@OneToOne, @OneToMany, @ManyToMany	Relationships

Spring / Spring Boot (Basics, REST API)

1. What is Spring Framework?

Spring is a **Java framework** that simplifies building enterprise applications.
Its core features are:

- **Dependency Injection (DI)** – Spring creates and manages objects for you.
- **Aspect-Oriented Programming (AOP)** – for cross-cutting concerns (logging, security).
- **Modular** – can use only the parts you need.
- **Integration** – works with JDBC, JPA, Hibernate, and more.

2. What is Spring Boot?

Spring Boot is a **simplified version of Spring** that removes boilerplate setup.

✅ Advantages:

- No XML configuration — uses **annotations**.
- **Embedded server** (Tomcat/Jetty) — run apps directly like normal Java apps.
- Built-in **starter dependencies** (spring-boot-starter-web, spring-boot-starter-data-jpa, etc.).
- Easy to create **REST APIs**.

Think of Spring Boot as: Spring + Auto Configuration + Embedded Server.

3. How Spring Boot Works

1. **Annotations configure your app** (@SpringBootApplication).
2. **Auto-configuration** sets up beans automatically.
3. Runs inside **embedded Tomcat**, so no manual deployment needed.

4. First Spring Boot REST API Example

Step 1 – Add Dependency (Maven pom.xml)

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Step 2 – Main Class

```java
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

Step 3 – REST Controller

```java
import org.springframework.web.bind.annotation.*;

@RestController
public class HelloController {

    @GetMapping("/hello")
    public String sayHello() {
        return "Hello from Spring Boot!";
    }

    @PostMapping("/greet")
    public String greetUser(@RequestParam String name) {
        return "Hello, " + name;
    }
}
```

Step 4 – Run

- Right-click → Run as Java Application
- Open browser:
    - GET → http://localhost:8080/hello
    - POST → Send request via Postman or HTML form to /greet.

5. Important Annotations

Annotation	Purpose
@SpringBootApplication	Main entry point of Spring Boot app
@RestController	Returns data (JSON/text) instead of HTML
@GetMapping, @PostMapping	Handle HTTP GET/POST requests
@RequestParam	Get query/form parameter

Annotation	Purpose
@PathVariable	Get value from URL path
@Autowired	Inject dependency automatically
@Service, @Repository	Mark service or DAO layers

6. REST API Basics in Spring Boot

A **REST API** is an HTTP-based interface where:

- **GET** → Read data
- **POST** → Create data
- **PUT** → Update data
- **DELETE** → Delete data

Example – Simple CRUD REST API:

```java
import org.springframework.web.bind.annotation.*;
import java.util.*;

@RestController
@RequestMapping("/users")
public class UserController {
    private List<String> users = new ArrayList<>();

    @GetMapping
    public List<String> getUsers() {
        return users;
    }

    @PostMapping
    public String addUser(@RequestParam String name) {
        users.add(name);
        return "User added: " + name;
    }

    @DeleteMapping("/{name}")
    public String deleteUser(@PathVariable String name) {
        users.remove(name);
        return "User deleted: " + name;
    }
}
```

7. Why Spring Boot is Popular

- Fast development (less setup, ready-to-use)
- Microservices-friendly

- Works with JPA, Hibernate, MySQL, MongoDB easily
- Built-in **REST API support**

REST API creation & consumption

1. What is a REST API?

REST (Representational State Transfer) is an architectural style for building APIs that communicate over HTTP.

- **Resource**: A data object (like User, Product).
- Uses **HTTP methods** for CRUD:
  - **GET** → Read
  - **POST** → Create
  - **PUT** → Update
  - **DELETE** → Delete
- **JSON** is the most common data format.

2. REST API Creation in Spring Boot

Step 1 – Add Dependencies (Maven)

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Step 2 – Main Application

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class RestApiApplication {
    public static void main(String[] args) {
        SpringApplication.run(RestApiApplication.class, args);
    }
}
```

Step 3 – Model (User.java)

```java
public class User {
    private int id;
    private String name;

    // Constructors
    public User() {}
    public User(int id, String name) {
        this.id = id;
        this.name = name;
    }

    // Getters & Setters
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
}
```

Step 4 – REST Controller

```java
import org.springframework.web.bind.annotation.*;
import java.util.*;

@RestController
@RequestMapping("/users")
public class UserController {
    private List<User> users = new ArrayList<>(List.of(
        new User(1, "Santhosh"),
        new User(2, "Naveen")
    ));

    @GetMapping
    public List<User> getAllUsers() {
        return users;
    }

    @GetMapping("/{id}")
    public User getUserById(@PathVariable int id) {
        return users.stream().filter(u -> u.getId() == id).findFirst().orElse(null);
    }

    @PostMapping
    public String addUser(@RequestBody User user) {
        users.add(user);
        return "User added: " + user.getName();
    }
```

```java
    @PutMapping("/{id}")
    public String updateUser(@PathVariable int id, @RequestBody User updatedUser) {
        for (User u : users) {
            if (u.getId() == id) {
                u.setName(updatedUser.getName());
                return "User updated: " + updatedUser.getName();
            }
        }
        return "User not found!";
    }

    @DeleteMapping("/{id}")
    public String deleteUser(@PathVariable int id) {
        users.removeIf(u -> u.getId() == id);
        return "User deleted with id " + id;
    }
}
```

3. Testing the API

You can test using:

- **Browser** → Only for GET
- **Postman** → For GET, POST, PUT, DELETE
- **curl** (command line)

Example in Postman:

- **GET** → http://localhost:8080/users
- **POST** → http://localhost:8080/users with JSON body:

```json
{
  "id": 3,
  "name": "Ravi"
}
```

4. REST API Consumption (Java Client)

You can consume REST APIs from:

- Java (using RestTemplate or HttpClient)
- JavaScript frontend (Fetch API, Axios)
- Mobile Apps

Java Example using RestTemplate:

```java
import org.springframework.web.client.RestTemplate;
import java.util.Arrays;
import java.util.List;

public class ApiConsumer {
    public static void main(String[] args) {
        RestTemplate restTemplate = new RestTemplate();
        String url = "http://localhost:8080/users";

        // GET all users
        User[] users = restTemplate.getForObject(url, User[].class);
        List<User> userList = Arrays.asList(users);
        userList.forEach(u -> System.out.println(u.getName()));

        // POST new user
        User newUser = new User(4, "Priya");
        String response = restTemplate.postForObject(url, newUser, String.class);
        System.out.println(response);
    }
}
```

5. Key Points

- **@RestController** → Makes a class a REST API controller.
- **@RequestMapping / @GetMapping / @PostMapping** → Handle HTTP requests.
- **@RequestBody** → Reads JSON from the request body into a Java object.
- **@PathVariable / @RequestParam** → Get values from URL or query params.
- Always use **JSON** for request/response in modern APIs.

Git

1.Definition:

Git is a **distributed version control system** used to track changes in source code during software development.

Key Points About Git

- **Version Control** → Keeps a history of changes so you can revert to older versions anytime.
- **Distributed** → Every developer has a complete copy of the repository on their machine.
- **Branching & Merging** → You can create separate branches for features and merge them later.
- **Offline Work** → You can commit changes locally without internet, then push later.

2.Common Git Commands

Command	Purpose
git init	Create a new Git repository in a folder
git clone <repo-url>	Download (clone) an existing repo
git status	Check changes in your working directory
git add <file>	Stage changes for commit
git commit -m "message"	Save changes to history
git push	Upload commits to remote repo
git pull	Download latest changes from remote
git branch	List branches
git checkout <branch>	Switch to another branch
git merge <branch>	Merge branch into current branch

GitHub

- **Definition:**

GitHub is a **cloud-based hosting service** for Git repositories.
It allows developers to store their code online, collaborate with others, and manage projects.

Key Points About GitHub

- Stores your Git repositories online.
- Makes **collaboration easy** via pull requests and code reviews.
- Provides **issue tracking** for bugs and feature requests.
- Supports **continuous integration (CI)** and deployment.
- Allows **public** and **private** repositories.

Git vs GitHub:

Git	GitHub
A tool for version control	A platform to host and share Git repositories
Works locally without internet	Requires internet to push/pull changes
Installed on your machine	Accessed via browser or Git commands
Maintained by open-source community	Owned by Microsoft