Date : 3/1/2026

CORE JAVA ASSIGNMENT.

Theory Assignment

1. Introduction to Java
Theory:
o History of Java
o Features of Java (Platform Independent, Object-Oriented, etc.)
o Understanding JVM, JRE, and JDK
o Setting up the Java environment and IDE (e.g., Eclipse, IntelliJ)
o Java Program Structure (Packages, Classes, Methods)

1. History of Java

- Java was developed by **James Gosling and his team at Sun Microsystems** in **1991**.
- Originally called **Oak**, it was designed for **embedded systems** like set-top boxes.
- Later renamed **Java** in **1995**, inspired by Java coffee.
- Java gained popularity due to the growth of the **Internet**, as it allowed programs to run on different platforms.
- In **2010**, **Oracle Corporation** acquired Sun Microsystems and took over Java.

Key Goal of Java:
👉 *Write Once, Run Anywhere (WORA)*

---

2. Features of Java

1. Platform Independent

- Java code is compiled into **bytecode**, not machine code.
- Bytecode runs on the **Java Virtual Machine (JVM)** of any operating system.

2. Object-Oriented

- Java follows **OOP concepts**:
    - Class & Object
    - Inheritance
    - Polymorphism
    - Encapsulation
    - Abstraction

3. Simple

- Syntax similar to C/C++ but without complex features like pointers.
- Automatic memory management using **Garbage Collection**.

4. Secure

- No direct access to memory.
- Bytecode is verified by JVM.
- Used widely in banking and enterprise applications.

5. Robust

- Strong exception handling.
- Automatic garbage collection.
- Strong type checking.

6. Multithreaded

- Supports multiple threads running simultaneously.

- Improves CPU utilization.

7. Portable

- Bytecode is platform-neutral.
- Same `.class` file runs on all systems.

8. High Performance

- Uses **Just-In-Time (JIT) Compiler**.

---

3. Understanding JVM, JRE, and JDK

JVM (Java Virtual Machine)

- Executes Java bytecode.
- Converts bytecode into machine code.
- Platform-dependent but bytecode is platform-independent.

Functions of JVM:

- Class loading
- Bytecode verification
- Execution
- Memory management

JRE (Java Runtime Environment)

- Provides environment to **run** Java programs.
- Contains:
    - JVM
    - Core libraries
    - Supporting files

👉 Used when you **only want to run** Java programs.

JDK (Java Development Kit)

- Used to **develop** Java programs.
- Contains:
    - JRE
    - Compiler (`javac`)
    - Debugger
    - Development tools

👉 Required for **writing and compiling** Java programs.

Relationship Diagram

```
JDK
 └── JRE
      └── JVM
```

---

4. Setting Up Java Environment and IDE

Steps to Install Java:

1. Download **JDK** from Oracle or OpenJDK website.
2. Install JDK.
3. Set **Environment Variables**:
    - `JAVA_HOME`
    - Add `bin` folder to `PATH`

Verify installation:

```
java -version
```

4.

Popular Java IDEs

1. Eclipse

- Free and open source
- Easy for beginners
- Strong plugin support

2. IntelliJ IDEA

- Smart code completion
- Faster development
- Community (free) and Ultimate versions

3. NetBeans

- Simple UI

- Good for academic use

---

5. Java Program Structure

Basic Java Program

```
package mypackage;

public class HelloWorld {

    public static void main(String[] args) {
        System.out.println("Hello, Java!");
    }
}
```

Explanation of Components

1. Package

- Used to group related classes.
- Avoids name conflicts.

```
package mypackage;
```

---

2. Class

- Blueprint of an object.
- Must match file name.

```
public class HelloWorld {
}
```

---

3. Method

- Block of code performing a task.
- `main()` is the **entry point** of Java program.

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

---

4. Statements

- Instructions executed by JVM.

```
System.out.println("Hello, Java!");
```

Key Rules

- Every Java program must have **at least one class**.
- Execution starts from the **main() method**.
- File name must match the **public class name**.

---

2. Data Types, Variables, and Operators

Theory:

o Primitive Data Types in Java (int, float, char, etc.)

o Variable Declaration and Initialization

o Operators: Arithmetic, Relational, Logical, Assignment, Unary, and Bitwise

o Type Conversion and Type Casting

1. Primitive Data Types in Java

Java has **8 primitive data types**. They store **simple values** and are not objects.

Data Type	Size	Default Value	Description	Example
`byte`	1 byte	0	Small integer	`byte b = 10;`
`short`	2 bytes	0	Medium integer	`short s = 100;`
`int`	4 bytes	0	Integer (most used)	`int x = 25;`
`long`	8 bytes	0L	Large integer	`long l = 50000L;`
`float`	4 bytes	0.0f	Decimal value	`float f = 3.14f;`
`double`	8 bytes	0.0	Large decimal	`double d = 99.99;`
`char`	2 bytes	'\u0000'	Single character	`char c = 'A';`
`boolean`	1 bit	false	True or false	`boolean flag = true;`

👉 **Note:** Java uses **Unicode**, so `char` takes 2 bytes.

---

2. Variable Declaration and Initialization

Variable Declaration

Declaring a variable means specifying its **type and name**.

```
int age;

float salary;
```

Variable Initialization

Assigning a value to a variable.

```
age = 20;

salary = 15000.50f;
```

Declaration + Initialization

```
int marks = 85;

char grade = 'A';
```

Types of Variables

1. **Local Variable**
    - Declared inside methods

- No default value

2. **Instance Variable**
    - Belongs to object
    - Declared inside class
3. **Static Variable**
    - Shared among all objects
    - Declared using `static`

---

3. Operators in Java

1. Arithmetic Operators

Operator	Meaning	Example
`+`	Addition	`a + b`
`-`	Subtraction	`a - b`
`*`	Multiplication	`a * b`
`/`	Division	`a / b`
`%`	Modulus	`a % b`

```
int a = 10, b = 3;
```

```
System.out.println(a % b); // 1
```

---

2. Relational Operators

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

```
System.out.println(a > b); // true
```

---

3. Logical Operators

Operator	Meaning

`&&`	Logical AND
`` ` ``	
`!`	Logical NOT

```
boolean x = true, y = false;
System.out.println(x && y); // false
```

---

4. Assignment Operators

Operator	Example	Meaning
`=`	`a = 5`	Assign
`+=`	`a += 2`	`a = a + 2`
`-=`	`a -= 2`	`a = a - 2`
`*=`	`a *= 2`	`a = a * 2`

`/=`	`a /= 2`	`a = a / 2`

---

5. Unary Operators

Operator	Meaning
`++`	Increment
`--`	Decrement
`+`	Unary plus
`-`	Unary minus
`!`	Logical NOT

```
int x = 5;

System.out.println(++x); // 6
```

---

6. Bitwise Operators

Operator	Meaning

`&`	Bitwise AND
`` ` ``	`` ` ``
`^`	Bitwise XOR
`~`	Bitwise NOT
`<<`	Left shift
`>>`	Right shift

```
int a = 5;   // 0101
int b = 3;   // 0011
System.out.println(a & b); // 1
```

---

4. Type Conversion and Type Casting

1. Type Conversion (Implicit / Widening)

- Automatic conversion
- Smaller data type → Larger data type

```
int a = 10;
double d = a;  // int → double
```

Order:

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

---

2. Type Casting (Explicit / Narrowing)

- Manual conversion
- Larger data type → Smaller data type
- Possible data loss

```
double d = 10.75;

int a = (int)d; // 10
```

Example of Type Casting

```
int x = 130;

byte b = (byte)x;

System.out.println(b); // -126
```

3. Control Flow Statements

Theory:

o If-Else Statements

o Switch Case Statements

o Loops (For, While, Do-While)

o Break and Continue Keywords.

3. Control Flow Statements in Java

Control flow statements decide the **order in which statements are executed** in a Java program.

---

1. If–Else Statements

Used to execute code based on a **condition**.

(a) Simple `if`

```
if (condition) {
    // statements
}
```

Example

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

(b) `if-else`

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

Example

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

(c) `else if` ladder

Used to check **multiple conditions**.

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

(d) Nested `if`

```
if (a > b) {
    if (a > c) {
        System.out.println("a is largest");
    }
}
```

---

2. Switch Case Statements

Used when multiple choices depend on **one variable**.

Syntax

```
switch (expression) {
```

```
    case value1:
        statements;
        break;
    case value2:
        statements;
        break;
    default:
        statements;
}
```

Example

```
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");
}
```

Important Points

✔ `break` prevents fall-through
✔ `default` is optional
✔ Works with `int`, `char`, `String`, `enum`

---

3. Loops in Java

Loops are used to **repeat a block of code**.

(a) `for` Loop

Used when number of iterations is **known**.

Syntax

```
for (initialization; condition; increment/decrement) {
    statements;
}
```

Example

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

(b) `while` Loop

Used when number of iterations is **unknown**.

Syntax

```
while (condition) {
    statements;
}
```

Example

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

(c) `do-while` Loop

Executes the loop **at least once**.

Syntax

```
do {
    statements;
} while (condition);
```

Example

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

Loop Comparison

Loop	Condition Check	Minimum Execution
for	Before loop	0 times
while	Before loop	0 times
do-while	After loop	1 time

---

4. Break and Continue Keywords

`break`

- Terminates the loop or switch statement.

Example

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

```
    if (i == 3)
        break;
    System.out.println(i);
}
```

Output:
1
2

---

continue

- Skips the current iteration and continues with next.

Example

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

Output:
1
2
4
5

---

4. Classes and Objects Theory:

o Defining a Class and Object in Java

o Constructors and Overloading

o Object Creation, Accessing Members of the Class

o this Keyword

4. Classes and Objects in Java

Java is a **pure object-oriented programming language**, where everything is designed using **classes and objects**.

---

1. Defining a Class and Object in Java

Class

- A **class** is a blueprint or template used to create objects.
- It contains:
    - **Variables** (data members)
    - **Methods** (member functions)

Syntax of Class

```
class ClassName {
    // variables
    // methods
}
```

Example

```
class Student {
    int rollNo;
    String name;

    void display() {
        System.out.println(rollNo + " " + name);
    }
}
```

Object

- An **object** is an instance of a class.
- It represents real-world entities.
- Objects occupy **memory at runtime**.

Creating an Object

```
Student s1 = new Student();
```

---

2. Constructors and Overloading

Constructor

- A constructor is a **special method** used to initialize objects.
- Constructor name must be **same as class name**.
- It has **no return type** (not even `void`).
- Automatically called when an object is created.

Types of Constructors

(a) Default Constructor

- Takes no parameters.

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

(b) Parameterized Constructor

- Takes parameters to initialize data members.

```
class Student {
    int rollNo;
    String name;

    Student(int r, String n) {
        rollNo = r;
        name = n;
    }
}
```

Constructor Overloading

- Multiple constructors with **different parameter lists**.
- Example of **compile-time polymorphism**.

```
class Student {
    Student() {
        System.out.println("Default");
    }

    Student(int r) {
        System.out.println("Roll No: " + r);
    }

    Student(int r, String n) {
        System.out.println(r + " " + n);
    }
}
```

---

3. Object Creation and Accessing Members

Object Creation

```
Student s1 = new Student(101, "Rahul");
```

Accessing Data Members

```
s1.rollNo = 101;
s1.name = "Rahul";
```

Accessing Methods

```
s1.display();
```

Complete Example

```
class Student {
    int rollNo;
    String name;

    void display() {
        System.out.println(rollNo + " " + name);
    }

    public static void main(String[] args) {
        Student s1 = new Student();
        s1.rollNo = 1;
        s1.name = "Amit";
        s1.display();
    }
}
```

---

4. `this` Keyword

Meaning of `this`

- `this` is a **reference variable**.
- Refers to the **current object** of the class.

Uses of `this` Keyword

---

1. To Distinguish Instance Variables from Parameters

```
class Student {
    int rollNo;

    Student(int rollNo) {
        this.rollNo = rollNo;
    }
}
```

---

2. To Invoke Current Class Method

```
class Demo {
    void show() {
        System.out.println("Show method");
    }

    void display() {
        this.show();
    }
}
```

---

3. To Invoke Current Class Constructor

```
class Test {
    Test() {
        this(10);
        System.out.println("Default constructor");
```

```
    }

    Test(int x) {
        System.out.println(x);
    }
}
```

---

4. To Pass Current Object as Argument

```
class Example {
    void show(Example obj) {
        System.out.println("Method called");
    }

    void call() {
        show(this);
    }
}
```

---

5. Methods in Java
Theory:
o Defining Methods
o Method Parameters and Return Types
o Method Overloading
o Static Methods and Variables

5. Methods in Java

A **method** is a block of code that performs a specific task.
Methods help in **code reusability, modularity, and readability**.

---

1. Defining Methods

Syntax

```
accessModifier returnType methodName(parameters) {
```

```
    // method body
}
```

Example

```
class Demo {
    void display() {
        System.out.println("Hello Java");
    }
}
```

Components of a Method

- **Access Modifier** – `public`, `private`, `protected`
- **Return Type** – Data type of value returned (`int`, `void`, etc.)
- **Method Name** – Name of the method
- **Parameters** – Values passed to method
- **Method Body** – Code to be executed

---

2. Method Parameters and Return Types

Method with Parameters

```
void add(int a, int b) {
    System.out.println(a + b);
}
```

Method with Return Type

```
int square(int x) {
```

```
    return x * x;
}
```

Method without Return Type (`void`)

```
void showMessage() {
    System.out.println("Welcome");
}
```

Calling a Method

```
Demo d = new Demo();
d.add(10, 20);
int result = d.square(5);
```

---

3. Method Overloading

Definition

Method overloading means **having multiple methods with the same name but different parameter lists**.

✔ Number of parameters
✔ Type of parameters
✔ Order of parameters

Example of Method Overloading

```
class MathOperation {

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

    int add(int a, int b, int c) {
```

```
        return a + b + c;
    }

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

Important Points

✔ Return type alone **cannot** overload a method
✔ Overloading occurs at **compile time**
✔ It is an example of **compile-time polymorphism**

---

4. Static Methods and Variables

Static Variables

Definition

- Declared using `static` keyword
- Shared among **all objects** of the class
- Only **one copy** exists

Example

```
class Student {
    static String college = "ABC College";
    int rollNo;
}
```

Static Methods

Definition

- Belong to the **class**, not object
- Can be called **without creating an object**
- Can access **only static data**

Example

```
class Demo {

    static void show() {
        System.out.println("Static method");
    }

    public static void main(String[] args) {
        Demo.show();
    }
}
```

Rules for Static Methods

✔ Cannot use `this` keyword
✔ Cannot directly access non-static members
✔ Can be overloaded

Static vs Non-Static

Static	Non-Static
Belongs to class	Belongs to object
Memory allocated once	Memory per object

Accessed using class name	Accessed using object

---

6. Object-Oriented Programming (OOPs) Concepts
Theory:
o Basics of OOP: Encapsulation, Inheritance, Polymorphism, Abstraction
o Inheritance: Single, Multilevel, Hierarchical
o Method Overriding and Dynamic Method Dispatch.

6. Object-Oriented Programming (OOPs) Concepts in Java

Java follows the **Object-Oriented Programming (OOP)** paradigm, which focuses on **objects**, **classes**, and **reusability**.

---

1. Basics of OOP Concepts

(a) Encapsulation

Definition

Encapsulation means **wrapping data (variables) and methods together into a single unit (class)** and restricting direct access to data.

How It Is Achieved

- Declare variables as `private`
- Provide `public` getter and setter methods

Example

```
class Student {
    private int marks;

    public void setMarks(int m) {
        marks = m;
    }

    public int getMarks() {
        return marks;
    }
}
```

Advantages

✔ Data security
✔ Controlled access
✔ Improved maintainability

(b) Inheritance

Definition

Inheritance allows one class to **acquire the properties and methods of another class**.

- `extends` keyword is used
- Promotes **code reusability**

Example

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

class Dog extends Animal {
```

```
    void bark() {
        System.out.println("Barking");
    }
}
```

(c) Polymorphism

Definition

Polymorphism means **one name, many forms**.

Types

1. **Compile-time Polymorphism** – Method Overloading
2. **Run-time Polymorphism** – Method Overriding

Example

```
class Shape {
    void draw() {
        System.out.println("Drawing shape");
    }
}

class Circle extends Shape {
    void draw() {
        System.out.println("Drawing circle");
    }
}
```

(d) Abstraction

Definition

Abstraction means **hiding implementation details and showing only essential features**.

Achieved Using

- **Abstract classes**
- **Interfaces**

Example (Abstract Class)

```
abstract class Vehicle {
    abstract void start();
}

class Car extends Vehicle {
    void start() {
        System.out.println("Car starts");
    }
}
```

---

2. Types of Inheritance in Java

(a) Single Inheritance

One child class inherits from **one parent class**.

```
class A {
    void show() {
        System.out.println("Class A");
    }
}

class B extends A {
}
```

(b) Multilevel Inheritance

A class inherits from another derived class.

```
class A {
    void display() {
        System.out.println("A");
    }
}

class B extends A {
}

class C extends B {
}
```

(c) Hierarchical Inheritance

Multiple child classes inherit from **one parent class**.

```
class A {
    void show() {
        System.out.println("Parent");
    }
}

class B extends A {
}

class C extends A {
}
```

Note:

❌ Java does **not** support **multiple inheritance using classes**
✔ Supported using **interfaces**

---

3. Method Overriding

Definition

Method overriding occurs when a **subclass provides a specific implementation of a method already defined in its superclass**.

Rules

✔ Method name must be same
✔ Parameter list must be same
✔ Inheritance must exist
✔ Occurs at **runtime**

Example

```
class Bank {
    int getRate() {
        return 5;
    }
}

class SBI extends Bank {
    int getRate() {
        return 7;
    }
}
```

---

4. Dynamic Method Dispatch

Definition

Dynamic Method Dispatch is a mechanism by which a **call to an overridden method is resolved at runtime**, not compile time.

Example

```
class A {
    void show() {
        System.out.println("Class A");
    }
}

class B extends A {
    void show() {
        System.out.println("Class B");
    }

    public static void main(String[] args) {
        A obj = new B();  // Upcasting
        obj.show();      // Calls B's method
    }
}
```

Output

```
Class B
```

---

7. Constructors and Destructors

Theory:

o Constructor Types (Default, Parameterized)

o Copy Constructor (Emulated in Java)

o Constructor Overloading

o Object Life Cycle and Garbage Collection

7. Constructors and Destructors in Java

In Java, **constructors** are used to initialize objects, while **destructors do not exist explicitly**. Java uses **Garbage Collection** for object destruction.

---

1. Constructor Types

What is a Constructor?

- A constructor is a **special method** used to initialize an object.
- Constructor name must be **same as class name**.
- It has **no return type**.
- Automatically called when an object is created using `new`.

(a) Default Constructor

Definition

- A constructor with **no parameters**.
- If no constructor is defined, Java provides a **default constructor** automatically.

Example

```
class Student {
    Student() {
        System.out.println("Default Constructor called");
    }
}
```

(b) Parameterized Constructor

Definition

- A constructor that accepts **parameters**.
- Used to initialize data members with given values.

Example

```
class Student {
    int rollNo;
    String name;

    Student(int r, String n) {
        rollNo = r;
        name = n;
    }
}
```

---

2. Copy Constructor (Emulated in Java)

Explanation

- Java does **not have a built-in copy constructor** like C++.
- Copy constructor behavior is **emulated** by passing an object as a parameter to a constructor.

Example

```
class Student {
    int rollNo;
    String name;

    Student(int r, String n) {
        rollNo = r;
        name = n;
    }

    // Emulated copy constructor
    Student(Student s) {
```

```
        rollNo = s.rollNo;
        name = s.name;
    }
}
```

Usage

```
Student s1 = new Student(101, "Amit");
Student s2 = new Student(s1);
```

---

3. Constructor Overloading

Definition

- Multiple constructors in the same class with **different parameter lists**.
- Helps initialize objects in **different ways**.

Example

```
class Demo {

    Demo() {
        System.out.println("Default Constructor");
    }

    Demo(int x) {
        System.out.println("Value: " + x);
    }

    Demo(int x, int y) {
        System.out.println("Sum: " + (x + y));
    }
}
```

Key Points

✔ Constructor name must be same
✔ Parameter list must differ
✔ Return type does not matter (constructors have none)

---

4. Object Life Cycle in Java

The life cycle of an object consists of **four stages**:

1. Creation

- Object is created using `new` keyword.

```
Student s = new Student();
```

---

2. Usage

- Object is used to access methods and variables.

---

3. Dereferencing

- Object is no longer referenced.

```
s = null;
```

---

4. Garbage Collection

- JVM automatically destroys unused objects.
- Memory is reclaimed by **Garbage Collector (GC)**.

---

5. Garbage Collection in Java

What is Garbage Collection?

- Automatic memory management process.
- Removes objects that are **no longer referenced**.

Ways an Object Becomes Eligible for GC

✔ Reference set to `null`
✔ Reassigned to another object
✔ Object created inside method (local object)

`finalize()` Method (Deprecated)

```
protected void finalize() {
    System.out.println("Object destroyed");
}
```

Constructors vs Destructors (Important Difference)

Constructor	Destructor
Used to initialize object	Not explicitly available
Called automatically on object creation	GC destroys object
Can be overloaded	Cannot be defined

---

8. Arrays and Strings

Theory:

o One-Dimensional and Multidimensional Arrays

o String Handling in Java: String Class, StringBuffer, StringBuilder

o Array of Objects

o String Methods (length, charAt, substring, etc.)

8. Arrays and Strings in Java

Arrays and Strings are used to store and manipulate **collections of data** efficiently.

---

1. Arrays in Java

Definition

An **array** is a collection of elements of the **same data type** stored in contiguous memory locations.

(a) One-Dimensional Array

Declaration

```
int[] a;
```

Initialization

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

Declaration + Initialization

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

Accessing Elements

```
System.out.println(a[0]); // 10
```

Example

```
int[] marks = {60, 70, 80};
for (int i = 0; i < marks.length; i++) {
    System.out.println(marks[i]);
}
```

(b) Multidimensional Array

Declaration

```
int[][] a = new int[2][3];
```

Initialization

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

Example

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

---

2. String Handling in Java

Java provides **three classes** for string handling:

(a) String Class

Characteristics

✔ Immutable (cannot be changed)
✔ Stored in **String Constant Pool**

Example

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

(b) StringBuffer

Characteristics

✔ Mutable
✔ Thread-safe (synchronized)
✔ Slower than StringBuilder

Example

```
StringBuffer sb = new StringBuffer("Java");
sb.append(" Programming");
System.out.println(sb);
```

(c) StringBuilder

Characteristics

✔ Mutable
✔ Not thread-safe
✔ Faster than StringBuffer

Example

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

String vs StringBuffer vs StringBuilder

Feature	String	StringBuffer	StringBuilder
Mutability	Immutable	Mutable	Mutable
Thread-safe	Yes	Yes	No
Performance	Slow	Medium	Fast

---

3. Array of Objects

Definition

An array that stores **objects of a class**.

Example

```
class Student {
    int rollNo;
    String name;

    Student(int r, String n) {
        rollNo = r;
        name = n;
    }
}

class Test {
    public static void main(String[] args) {
        Student[] s = new Student[2];

        s[0] = new Student(1, "Amit");
        s[1] = new Student(2, "Ravi");

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

```
            System.out.println(s[i].rollNo + " " + s[i].name);
        }
    }
}
```

---

4. String Methods in Java

Common String Methods

Method	Description	Example
`length()`	Returns string length	`s.length()`
`charAt(int)`	Character at index	`s.charAt(1)`
`substring(int)`	Substring from index	`s.substring(2)`
`substring(int, int)`	Substring range	`s.substring(1,4)`
`toUpperCase()`	Convert to uppercase	`s.toUpperCase()`
`toLowerCase()`	Convert to lowercase	`s.toLowerCase()`
`equals()`	Compare content	`s1.equals(s2)`
`equalsIgnoreCase()`	Ignore case	`s1.equalsIgnoreCase(s2)`
`indexOf()`	First occurrence	`s.indexOf('a')`
`replace()`	Replace characters	`s.replace('a','o')`
`trim()`	Remove spaces	`s.trim()`

Example

```
String s = " Java Programming ";
```

```
System.out.println(s.length());          // 18
System.out.println(s.trim());             // Java Programming
System.out.println(s.substring(5));       // Programming
System.out.println(s.charAt(2));          // a
```

---

9. Inheritance and Polymorphism
Theory:

o Inheritance Types and Benefits
o Method Overriding
o Dynamic Binding (Run-Time Polymorphism)
o Super Keyword and Method Hiding

9. Inheritance and Polymorphism in Java

Inheritance and polymorphism are **core OOP concepts** that help achieve **code reusability, flexibility, and maintainability**.

---

1. Inheritance

Definition

Inheritance is a mechanism where a **child class acquires properties and methods of a parent class**.

- Uses the `extends` keyword
- Supports **IS-A relationship**

```
class Parent {
    void show() {
        System.out.println("Parent class");
    }
}

class Child extends Parent {
```

```
}
```

---

2. Types of Inheritance in Java

(a) Single Inheritance

One child class inherits from one parent class.

```
class A {
    void display() {
        System.out.println("A");
    }
}

class B extends A {
}
```

(b) Multilevel Inheritance

Inheritance chain of multiple levels.

```
class A {
    void show() {
        System.out.println("A");
    }
}

class B extends A {
}

class C extends B {
}
```

(c) Hierarchical Inheritance

Multiple child classes inherit from the same parent class.

```
class A {
    void show() {
        System.out.println("Parent");
    }
}

class B extends A {
}

class C extends A {
}
```

(d) Multiple Inheritance (Not supported using classes)

- Java does **not support multiple inheritance with classes**
- Achieved using **interfaces**

---

3. Benefits of Inheritance

✔ Code reusability
✔ Reduced redundancy
✔ Easy maintenance
✔ Supports polymorphism
✔ Improves readability

---

4. Method Overriding

Definition

Method overriding occurs when a **child class provides its own implementation of a method defined in the parent class**.

Rules

✔ Same method name
✔ Same parameter list
✔ Inheritance must exist
✔ Return type should be same or covariant
✔ Access level should not be more restrictive

Example

```
class Bank {
    int getRate() {
        return 5;
    }
}

class SBI extends Bank {
    int getRate() {
        return 7;
    }
}
```

---

5. Dynamic Binding (Run-Time Polymorphism)

Definition

Dynamic binding means that the **method call is resolved at runtime**, based on the **object type**, not reference type.

Example

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

```
}

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

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

Output

```
Dog barks
```

Key Points

✔ Reference type = Parent
✔ Object type = Child
✔ Method resolved at runtime

---

6. `super` Keyword

Definition

`super` refers to the **immediate parent class object**.

Uses of `super`

(a) Access Parent Class Variable

```
class A {
```

```
    int x = 10;
}

class B extends A {
    int x = 20;

    void show() {
        System.out.println(super.x); // 10
    }
}
```

(b) Call Parent Class Method

```
class A {
    void display() {
        System.out.println("Parent method");
    }
}

class B extends A {
    void display() {
        super.display();
        System.out.println("Child method");
    }
}
```

(c) Call Parent Class Constructor

```
class A {
    A() {
        System.out.println("Parent constructor");
    }
}

class B extends A {
    B() {
        super();
```

```
        System.out.println("Child constructor");
    }
}
```

---

7. Method Hiding

Definition

Method hiding occurs when a **static method in a child class has the same signature as a static method in the parent class**.

Important Points

✔ Applies only to **static methods**
✔ Resolved at **compile time**
✔ Not polymorphism

Example

```
class A {
    static void show() {
        System.out.println("Parent static method");
    }
}

class B extends A {
    static void show() {
        System.out.println("Child static method");
    }

    public static void main(String[] args) {
        A a = new B();
        a.show();
    }
}
```

Output

```
Parent static method
```

Method Overriding vs Method Hiding

Method Overriding	Method Hiding
Applies to instance methods	Applies to static methods
Run-time binding	Compile-time binding
Supports polymorphism	Does not support polymorphism

---

10. Interfaces and Abstract Classes

Theory:

o Abstract Classes and Methods

o Interfaces: Multiple Inheritance in Java

o Implementing Multiple Interfaces

10. Interfaces and Abstract Classes in Java

Interfaces and abstract classes are used to achieve **abstraction** and **multiple inheritance** in Java.

---

1. Abstract Classes and Methods

Abstract Class

Definition

An **abstract class** is a class declared using the `abstract` keyword that **cannot be instantiated**.

Characteristics

✔ May contain **abstract and non-abstract methods**
✔ May contain **constructors**
✔ Can have instance variables
✔ Supports **inheritance**

Syntax

```
abstract class ClassName {

    abstract void method1(); // abstract method

    void method2() {

        System.out.println("Concrete method");

    }

}
```

Abstract Method

Definition

A method declared without implementation is called an **abstract method**.

```
abstract void display();
```

Example

```
abstract class Shape {

    abstract void draw();

}


class Circle extends Shape {

    void draw() {

        System.out.println("Drawing Circle");

    }

}
```

Key Rules

✔ Abstract class must be inherited
✔ Child class must implement all abstract methods
✔ Abstract methods cannot be `final` or `static`

---

2. Interfaces in Java

Interface

Definition

An **interface** is a blueprint of a class that contains **abstract methods and constants**.

- Declared using `interface` keyword
- Achieves **100% abstraction** (until Java 7)

Syntax

```
interface InterfaceName {

    void show();     // abstract method

}
```

Interface Features

✔ All methods are **public and abstract** by default
✔ All variables are **public, static, final**
✔ No constructors
✔ Supports **multiple inheritance**

Example

```
interface Animal {

    void sound();

}


class Dog implements Animal {

    public void sound() {

        System.out.println("Dog barks");

    }
```

```
}
```

---

3. Multiple Inheritance in Java (Using Interfaces)

Java does **not support multiple inheritance using classes** to avoid ambiguity, but it **supports multiple inheritance using interfaces**.

Example

```
interface A {

    void show();

}


interface B {

    void display();

}


class C implements A, B {

    public void show() {

        System.out.println("Show method");

    }


    public void display() {

        System.out.println("Display method");
```

```
    }
}
```

---

4. Implementing Multiple Interfaces

Syntax

```
class ClassName implements Interface1, Interface2 {
    // implementations
}
```

Example

```
interface Printable {
    void print();
}

interface Scannable {
    void scan();
}

class Machine implements Printable, Scannable {
    public void print() {
```

```
        System.out.println("Printing");
    }

    public void scan() {
        System.out.println("Scanning");
    }
}
```

Abstract Class vs Interface

Abstract Class	Interface
Can have abstract and concrete methods	Only abstract methods (Java 7)
Supports single inheritance	Supports multiple inheritance
Can have constructors	Cannot have constructors
Variables can be any type	Variables are public static final
Uses `extends`	Uses `implements`

---

11. Packages and Access Modifiers

Theory:

o Java Packages: Built-in and User-Defined Packages

o Access Modifiers: Private, Default, Protected, Public

o Importing Packages and Classpath

11. Packages and Access Modifiers in Java

Packages and access modifiers help in **organizing code**, **controlling access**, and **improving security** in Java programs.

---

1. Java Packages

Definition

A **package** is a namespace that groups **related classes and interfaces**.

Advantages of Packages

✔ Code organization
✔ Avoids class name conflicts
✔ Improves reusability
✔ Provides access control

---

2. Types of Packages

(a) Built-in Packages

Provided by Java API.

Common Built-in Packages

Package	Purpose
`java.lang`	Core classes (String, Math, System)
`java.util`	Utilities (Scanner, ArrayList)
`java.io`	Input/Output operations
`java.sql`	Database connectivity
`java.net`	Networking

👉 `java.lang` is imported **automatically**.

(b) User-Defined Packages

Created by programmers.

Creating a Package

```
package mypackage;


public class Demo {
```

```
    public void show() {
        System.out.println("User defined package");
    }
}
```

Compile with Package

```
javac -d . Demo.java
```

---

3. Access Modifiers in Java

Access modifiers control the **visibility** of classes, methods, and variables.

Types of Access Modifiers

(a) Private

- Accessible **only within the same class**
- Most restrictive

```
class Test {
    private int x = 10;
}
```

(b) Default (No Modifier)

- Accessible **within the same package**
- Also called **package-private**

```
class Test {

    int x = 20; // default access

}
```

(c) Protected

- Accessible within:
  ✔ Same package
  ✔ Subclasses in different packages

```
class Test {

    protected int x = 30;

}
```

(d) Public

- Accessible **from anywhere**
- Least restrictive

```
public class Test {

    public int x = 40;

}
```

Access Modifier Scope Table

Modifier	Same Class	Same Package	Subclass	Outside Package
Private	✔	✖	✖	✖
Default	✔	✔	✖	✖
Protected	✔	✔	✔	✖
Public	✔	✔	✔	✔

---

4. Importing Packages

Import Statement

Used to access classes defined in other packages.

Import Single Class

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

Import Entire Package

```
import java.util.*;
```

Without Import (Fully Qualified Name)

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

---

5. Classpath in Java

Definition

Classpath tells the JVM where to find **.class files and packages**.

Setting Classpath

Temporarily

```
set CLASSPATH=.;C:\myclasses
```

Permanently

- Set through **Environment Variables**

---

12. Exception Handling

Theory:

o Types of Exceptions: Checked and Unchecked

o try, catch, finally, throw, throws

o Custom Exception Classes

12. Exception Handling in Java

Exception handling is a mechanism to **handle runtime errors** so that the **normal flow of the program is not disrupted**.

---

1. Exception in Java

Definition

An **exception** is an unwanted or unexpected event that occurs during program execution and **disrupts normal flow**.

Examples:

- Divide by zero
- Array index out of bounds
- File not found

---

2. Types of Exceptions

(a) Checked Exceptions

Definition

- Checked at **compile time**
- Must be **handled or declared**

Examples

- `IOException`
- `SQLException`
- `ClassNotFoundException`

```
try {
    FileReader fr = new FileReader("abc.txt");
} catch (IOException e) {
    System.out.println(e);
}
```

(b) Unchecked Exceptions

Definition

- Checked at **runtime**
- Subclasses of `RuntimeException`

Examples

- `ArithmeticException`
- `NullPointerException`
- `ArrayIndexOutOfBoundsException`

```
int a = 10 / 0; // ArithmeticException
```

Checked vs Unchecked Exceptions

Checked	Unchecked
Checked at compile time	Checked at runtime
Must be handled	Handling optional
Extend `Exception`	Extend `RuntimeException`

---

3. `try`, `catch`, `finally`

`try` Block

Contains code that may cause an exception.

```
try {
    int a = 10 / 0;
}
```

`catch` Block

Handles the exception.

```
catch (ArithmeticException e) {
    System.out.println("Error: " + e);
}
```

`finally` Block

- Always executed
- Used for cleanup (closing files, connections)

```
finally {
    System.out.println("Finally block executed");
}
```

Example

```
try {
```

```
    int a = 10 / 2;
} catch (Exception e) {
    System.out.println(e);
} finally {
    System.out.println("Done");
}
```

---

4. `throw` Keyword

Definition

Used to **explicitly throw an exception**.

```
throw new ArithmeticException("Invalid operation");
```

Example

```
int age = 15;
if (age < 18) {
    throw new ArithmeticException("Not eligible to vote");
}
```

---

5. `throws` Keyword

Definition

Used to **declare exceptions** that a method may pass to the calling method.

```
void readFile() throws IOException {
    FileReader fr = new FileReader("abc.txt");
}
```

Example

```
class Test {
    static void show() throws Exception {
        throw new Exception("Exception occurred");
    }

    public static void main(String[] args) {
        try {
            show();
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

`throw` vs `throws`

throw	throws
Used to throw exception	Used to declare exception
Used inside method	Used in method signature
Can throw only one	Can declare multiple

---

6. Custom Exception Classes

Definition

User-defined exception classes created by **extending Exception or RuntimeException**.

Example (Checked Custom Exception)

```
class InvalidAgeException extends Exception {

    InvalidAgeException(String msg) {

        super(msg);

    }

}
```

Using Custom Exception

```
class Test {

    static void checkAge(int age) throws InvalidAgeException {

        if (age < 18)

            throw new InvalidAgeException("Age must be 18 or above");

    }


    public static void main(String[] args) {

        try {

            checkAge(16);

        } catch (InvalidAgeException e) {

            System.out.println(e.getMessage());

        }

    }

}
```

Example (Unchecked Custom Exception)

```
class MyException extends RuntimeException {

    MyException(String msg) {

        super(msg);

    }

}
```

---

13. Multithreading
Theory:
o Introduction to Threads
o Creating Threads by Extending Thread Class or Implementing Runnable Interface
o Thread Life Cycle
o Synchronization and Inter-thread Communication

13. Multithreading in Java

Multithreading allows **multiple threads (smaller units of a process)** to run **concurrently**, improving **CPU utilization and performance**.

---

1. Introduction to Threads

What is a Thread?

- A **thread** is the smallest unit of execution within a process.
- Java supports **multithreading** using the `Thread` class and `Runnable` interface.

Advantages of Multithreading

✔ Better CPU utilization
✔ Faster program execution
✔ Improved responsiveness
✔ Resource sharing

Process vs Thread

Process	Thread
Heavyweight	Lightweight

Separate memory	Shared memory
Slower communication	Faster communication

---

2. Creating Threads in Java

Java provides **two ways** to create threads.

(a) Extending the `Thread` Class

Steps

1. Extend `Thread` class
2. Override `run()` method
3. Call `start()` method

Example

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running");
    }

    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start();
    }
}
```

(b) Implementing the `Runnable` Interface

Steps

1. Implement `Runnable` interface
2. Override `run()` method
3. Create `Thread` object and call `start()`

Example

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Thread running using Runnable");
    }

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

Thread Class vs Runnable Interface

Thread Class	Runnable Interface
Extends `Thread`	Implements `Runnable`
No multiple inheritance	Supports multiple inheritance
Less flexible	More flexible

👉 **Runnable is preferred** in real applications.

---

3. Thread Life Cycle

A thread passes through **different states** during execution.

Thread Life Cycle States

1. **New**
    - Thread object created

```
Thread t = new Thread();
```

2.
3. **Runnable**
    - After calling `start()`
    - Ready to run, waiting for CPU
4. **Running**
    - Thread scheduler selects the thread
5. **Blocked / Waiting**
    - Waiting for resource, sleep, or I/O
6. **Terminated (Dead)**
    - Thread completes execution

Life Cycle Flow (Exam Description)

`New → Runnable → Running → Blocked/Waiting → Runnable → Terminated`

---

4. Synchronization

What is Synchronization?

Synchronization ensures that **only one thread accesses a shared resource at a time**, preventing **data inconsistency**.

Why Synchronization?

- To avoid **race condition**
- To maintain **data integrity**

Synchronized Method

```
class Table {
    synchronized void printTable(int n) {
        for (int i = 1; i <= 5; i++) {
            System.out.println(n * i);
        }
    }
}
```

Synchronized Block

```
synchronized(this) {
    // critical section
}
```

Key Points

✔ Synchronization uses **object lock**
✔ Only one thread can enter synchronized block at a time
✔ Reduces performance slightly

---

5. Inter-Thread Communication

Definition

Inter-thread communication allows threads to **communicate with each other** and **avoid busy waiting**.

Methods Used

All are defined in **Object class**:

Method	Purpose
`wait()`	Thread waits
`notify()`	Wakes one waiting thread
`notifyAll()`	Wakes all waiting threads

Example

```
class Customer {
    int amount = 1000;

    synchronized void withdraw(int amt) {
        if (amount < amt) {
            try {
                wait();
            } catch (Exception e) {}
        }
        amount -= amt;
        System.out.println("Withdraw completed");
    }

    synchronized void deposit(int amt) {
        amount += amt;
        System.out.println("Deposit completed");
        notify();
    }
}
```

---

14. File Handling

Theory:

o Introduction to File I/O in Java (java.io package)

o FileReader and FileWriter Classes

o BufferedReader and BufferedWriter

o Serialization and Deserialization

14. File Handling in Java

File handling allows Java programs to **store data permanently** in files and **read data back** when needed.
Java supports file I/O mainly through the **`java.io` package**.

---

1. Introduction to File I/O in Java (`java.io` package)

What is File I/O?

- **Input**: Reading data from a file
- **Output**: Writing data to a file

Java treats files as **streams of bytes or characters**.

Important Classes in `java.io`

Class	Purpose

`File`	Represents file or directory
`FileReader`	Reads character data
`FileWriter`	Writes character data
`BufferedReader`	Fast reading of text
`BufferedWriter`	Fast writing of text
`ObjectInputStream`	Deserialization
`ObjectOutputStream`	Serialization

File Class Example

```
File f = new File("data.txt");

System.out.println(f.exists());
```

---

2. FileReader and FileWriter Classes

These classes are used for **character-based file operations**.

FileWriter

Purpose

- Writes characters to a file
- Creates file if it does not exist

Example (Writing to File)

```
import java.io.*;


class WriteFile {
    public static void main(String[] args) throws IOException {
        FileWriter fw = new FileWriter("abc.txt");
        fw.write("Hello Java");
        fw.close();
    }
}
```

FileReader

Purpose

- Reads characters from a file

Example (Reading from File)

```java
import java.io.*;


class ReadFile {

    public static void main(String[] args) throws IOException {

        FileReader fr = new FileReader("abc.txt");

        int ch;

        while ((ch = fr.read()) != -1) {

            System.out.print((char) ch);

        }

        fr.close();

    }

}
```

Limitations

❌ Slow for large files
❌ Reads character by character

---

3. BufferedReader and BufferedWriter

These classes improve performance by using **buffering**.

BufferedWriter

Example

```
import java.io.*;


class WriteBuffered {

    public static void main(String[] args) throws IOException {

        BufferedWriter bw = new BufferedWriter(new
FileWriter("data.txt"));

        bw.write("Buffered Writing");

        bw.newLine();

        bw.write("Java File Handling");

        bw.close();

    }

}
```

BufferedReader

Example

```
import java.io.*;


class ReadBuffered {

    public static void main(String[] args) throws IOException {

        BufferedReader br = new BufferedReader(new
FileReader("data.txt"));

        String line;
```

```
        while ((line = br.readLine()) != null) {
            System.out.println(line);
        }
        br.close();
    }
}
```

FileReader/FileWriter vs BufferedReader/BufferedWriter

FileReader / FileWriter	BufferedReader / BufferedWriter
Slower	Faster
Reads character by character	Reads line by line
No buffering	Uses buffer

---

4. Serialization and Deserialization

Serialization

Definition

Serialization is the process of **converting an object into a byte stream** so that it can be stored in a file or transferred over a network.

- Uses `ObjectOutputStream`
- Class must implement `Serializable` interface

Example (Serialization)

```
import java.io.*;


class Student implements Serializable {

    int rollNo;

    String name;


    Student(int r, String n) {

        rollNo = r;

        name = n;

    }

}


class SerializeDemo {

    public static void main(String[] args) throws Exception {

        Student s = new Student(1, "Amit");
```

```
        ObjectOutputStream oos =
            new ObjectOutputStream(new FileOutputStream("obj.txt"));
        oos.writeObject(s);
        oos.close();
    }
}
```

Deserialization

Definition

Deserialization is the process of **converting byte stream back into an object**.

- Uses `ObjectInputStream`

Example (Deserialization)

```
import java.io.*;

class DeserializeDemo {
    public static void main(String[] args) throws Exception {
        ObjectInputStream ois =
            new ObjectInputStream(new FileInputStream("obj.txt"));

        Student s = (Student) ois.readObject();
```

```
        System.out.println(s.rollNo + " " + s.name);

        ois.close();

    }

}
```

---

15. Collections Framework
Theory:
o Introduction to Collections Framework
o List, Set, Map, and Queue Interfaces
o ArrayList, LinkedList, HashSet, TreeSet, HashMap, TreeMap
o Iterators and ListIterators

15. Collections Framework

The **Java Collections Framework (JCF)** provides a **standardized architecture** for storing, manipulating, and accessing groups of objects efficiently.

---

1. Introduction to Collections Framework

What is a Collection?

A **collection** is an object that can store **multiple elements** as a single unit.

Why Collections Framework?

✔ Dynamic size (unlike arrays)
✔ Built-in data structures
✔ Improved performance
✔ Reusable and consistent APIs

Major Components of JCF

1. **Interfaces** – List, Set, Queue, Map
2. **Classes** – ArrayList, HashSet, HashMap, etc.
3. **Algorithms** – Searching, sorting (Collections class)

Hierarchy (Simplified)

```
Iterable
   |
Collection
   |
---------------------------------
|        |            |           |
List     Set         Queue       (Map – separate)
```

---

2. Core Interfaces

(a) List Interface

Features

- Ordered collection
- Allows **duplicate elements**
- Index-based access

Common Implementations

- `ArrayList`
- `LinkedList`

Example

```
List<Integer> list = new ArrayList<>();
list.add(10);
```

```
list.add(20);
list.add(10);
```

(b) Set Interface

Features

- **No duplicate elements**
- Unordered collection (except TreeSet)

Common Implementations

- `HashSet`
- `TreeSet`

Example

```
Set<String> set = new HashSet<>();
set.add("A");
set.add("A"); // ignored
```

(c) Queue Interface

Features

- Follows **FIFO (First In First Out)**
- Used in scheduling, buffering

Common Implementation

- `LinkedList`
- `PriorityQueue`

Example

```
Queue<Integer> q = new LinkedList<>();
q.add(1);
q.add(2);
q.remove();
```

(d) Map Interface (Not a Child of Collection)

Features

- Stores data in **key-value pairs**
- Keys are **unique**
- Values can be duplicate

Common Implementations

- `HashMap`
- `TreeMap`

Example

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "Java");
map.put(2, "Python");
```

---

3. Important Collection Classes

ArrayList

Features

✔ Dynamic array
✔ Fast random access
❌ Slower insertion/deletion

Example

```
ArrayList<String> al = new ArrayList<>();
al.add("Java");
al.add("C++");
```

LinkedList

Features

✔ Doubly linked list
✔ Fast insertion/deletion
❌ Slower access

Example

```
LinkedList<Integer> ll = new LinkedList<>();
ll.add(10);
ll.addFirst(5);
```

HashSet

Features

✔ No duplicates
✔ No order maintained
✔ Fast performance

Example

```
HashSet<Integer> hs = new HashSet<>();
hs.add(10);
hs.add(20);
```

TreeSet

Features

✔ Sorted order
✔ No duplicates
❌ Slower than HashSet

Example

```
TreeSet<Integer> ts = new TreeSet<>();
ts.add(30);
ts.add(10);
```

HashMap

Features

✔ Key-value pairs
✔ One null key allowed
✔ Unordered

Example

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

TreeMap

Features

✔ Sorted by key
✔ No null key
❌ Slower than HashMap

Example

```
TreeMap<Integer, String> tm = new TreeMap<>();
tm.put(3, "C");
tm.put(1, "A");
```

---

4. Iterators and ListIterators

Iterator Interface

Purpose

Used to **traverse elements** of a collection.

Methods

- `hasNext()`
- `next()`
- `remove()`

Example

```
Iterator<String> it = al.iterator();
while (it.hasNext()) {
    System.out.println(it.next());
}
```

ListIterator Interface

Features

✔ Traverses in **both directions**
✔ Can modify elements
✔ Only for **List**

Methods

- `hasPrevious()`
- `previous()`
- `add()`
- `set()`

Example

```
ListIterator<String> li = al.listIterator();
while (li.hasNext()) {
    System.out.println(li.next());
}
```

Iterator vs ListIterator

Iterator	ListIterator
One direction	Two directions
For all collections	Only for List
Cannot add elements	Can add elements

---

16. Java Input/Output (I/O)
Theory:

o Streams in Java (InputStream, OutputStream)
o Reading and Writing Data Using Streams
o Handling File I/O Operations

16. Java Input / Output (I/O)

Java I/O provides mechanisms to **read data from input sources** and **write data to output destinations** using **streams**.

---

1. Streams in Java

What is a Stream?

A **stream** is a **sequence of data** flowing from a **source** to a **destination**.

Types of Streams

(a) Byte Streams

- Used for **binary data** (images, audio, video)
- Base classes:
    - `InputStream`
    - `OutputStream`

(b) Character Streams

- Used for **text data**
- Base classes:
    - `Reader`

- `Writer`

Stream Hierarchy (Simplified)

```
InputStream                OutputStream
     |                          |
FileInputStream          FileOutputStream
BufferedInputStream      BufferedOutputStream
```

---

2. InputStream and OutputStream

InputStream

Purpose

- Reads **byte data**
- Abstract class

Important Methods

Method	Description
`read()`	Reads one byte
`read(byte[])`	Reads multiple bytes
`close()`	Closes stream
