LEARNER'S PICK

AKTU | GATE | JEE | NEET | CBSE | CODING

OOPS WITH JAVA

(BCS403)

UNIT-01+02+03+04+05

ONE SHOT

Unit-01

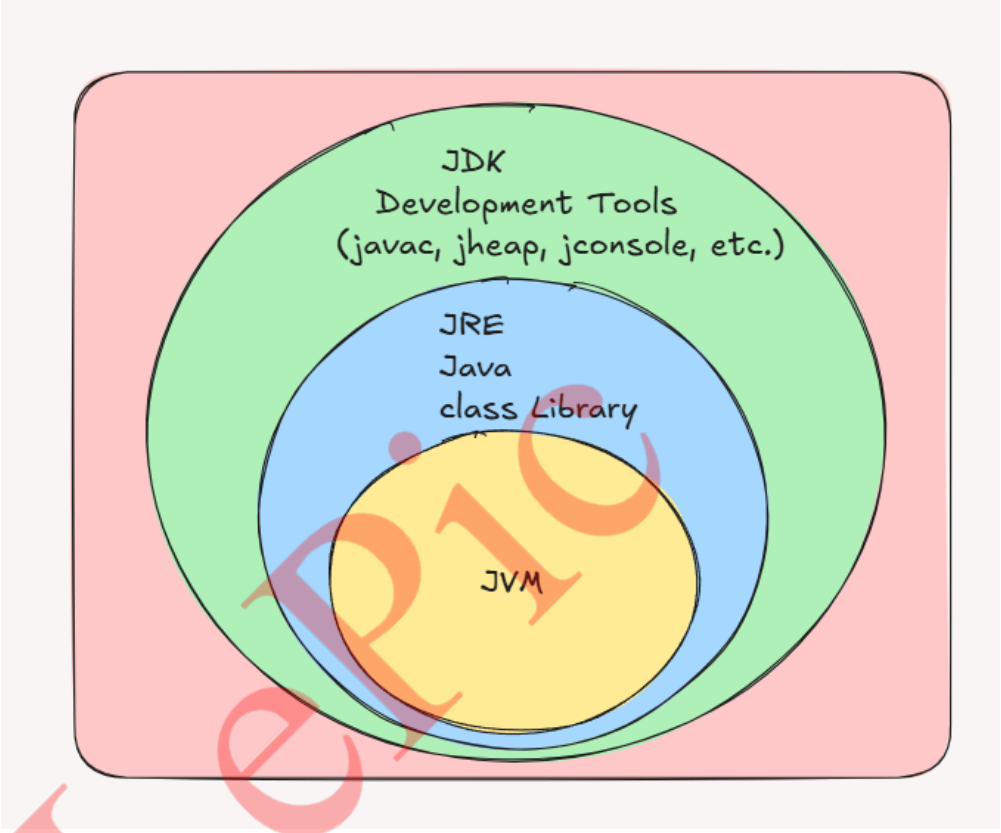
Ques →	Describe the architecture of Java and explain the roles of JVM, JRE, and JDK. How do they interact in the program lifecycle?

Ans →

Architecture of Java

→ Java architecture is designed to be platform-independent, secure, and robust. It follows a "Write Once, Run Anywhere" approach.
→ The architecture includes:

- Java Source Code (.java files)
- Compiler (javac)
- Bytecode (.class files)
- Java Virtual Machine (JVM)



Roles of JVM, JRE, and JDK

1. JVM (Java Virtual Machine)

→ JVM is the core part of Java architecture.
→ It runs Java bytecode on any platform by converting it to machine-specific code.
→ It performs tasks like:

- Loading code
- Verifying code
- Executing code
- Providing runtime environment

→ JVM is platform-dependent (each OS has its own JVM).

2. JRE (Java Runtime Environment)

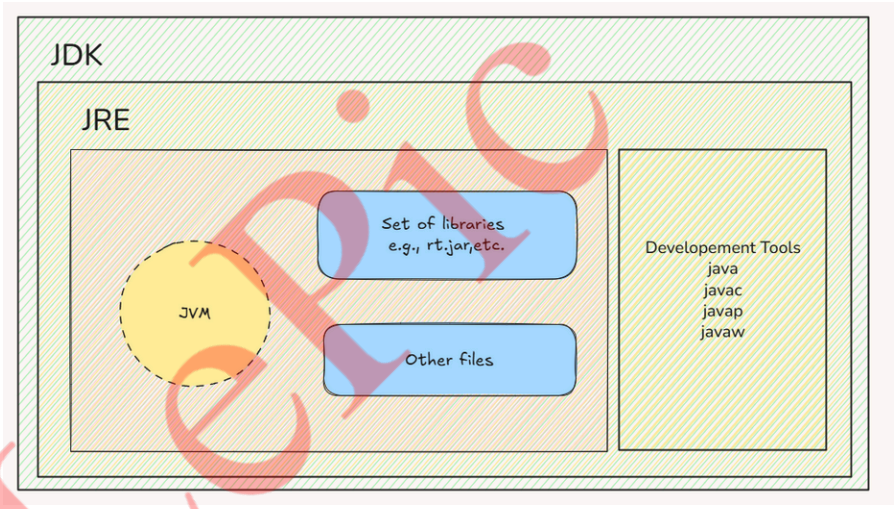
→ JRE provides an environment to run Java applications.
→ It includes:

- JVM
- Core Java libraries
- Other supporting files
  → It does **not** contain development tools (like compiler).
  → It is used to **run** Java programs, not to develop them.

3. JDK (Java Development Kit)

→ JDK is a complete package for Java development.
→ It includes:

- JRE
- Development tools (compiler, debugger, etc.)
  → It is used to **develop and compile** Java applications.

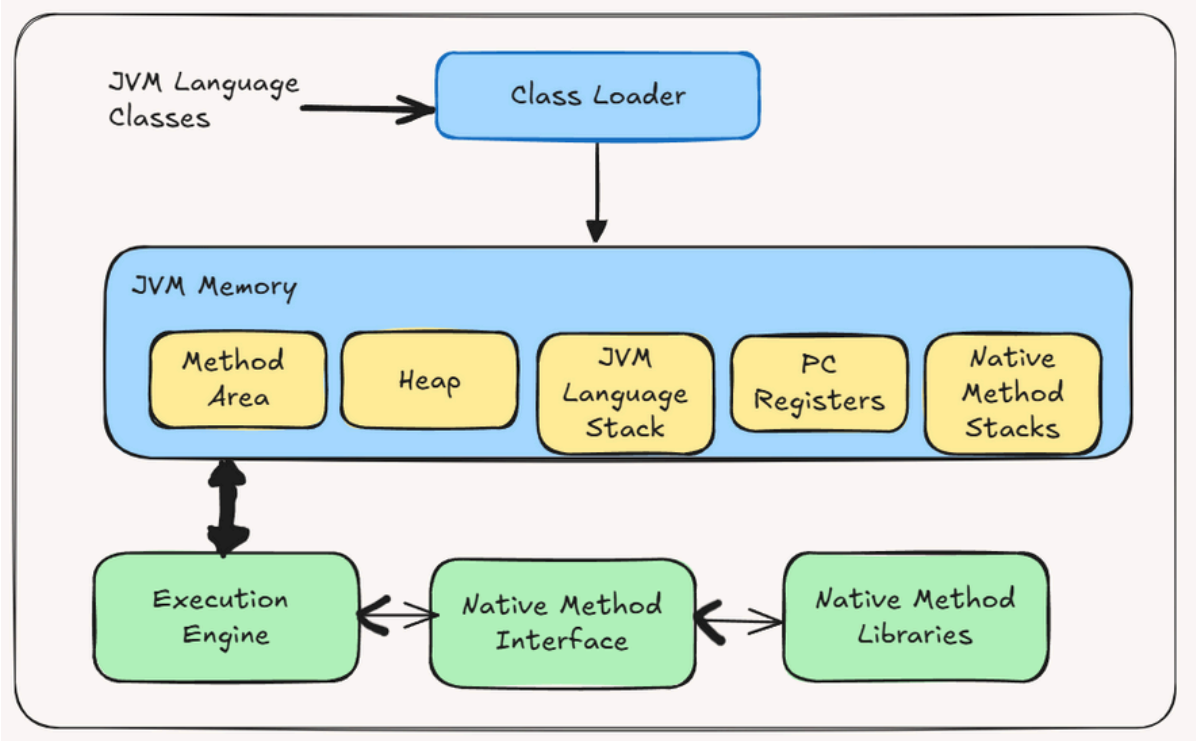


Interaction in Program Lifecycle

→ **Step 1:** Write Java code in .java file (using JDK tools).
→ **Step 2:** Compile the code with javac, which converts it into bytecode (.class file).
→ **Step 3:** JVM executes the bytecode with the help of JRE.
→ **Step 4:** The program runs on any platform where JVM is installed.

Ques →	**With a neat diagram, explain the internal working of JVM. How does it ensure platform independence in Java?**

Internal Working of JVM



→ **JVM (Java Virtual Machine)** is the engine that runs Java bytecode on any machine.
→ It provides a **runtime environment** and handles:

- Loading of class files
- Bytecode verification
- Code execution
- Memory management (via Garbage Collector).

Components of JVM

1. Class Loader Subsystem

→ Loads .class files (bytecode) into memory.
→ Performs:

Ans →

- Loading
- Linking (Verification, Preparation, Resolution)
- Initialization

2. Runtime Data Areas

→ **Method Area:** Stores class structure like metadata, static variables.
→ **Heap:** Stores objects and their instance variables.
→ **Stack:** Stores frames for each method call, local variables, partial results.
→ **Program Counter (PC) Register:** Stores address of the current instruction.
→ **Native Method Stack:** Supports native (non-Java) method execution.

3. Execution Engine

→ Executes bytecode instructions.
→ Includes:

- **Interpreter**: Reads and executes bytecode line by line.
- **JIT (Just-In-Time) Compiler**: Improves performance by compiling bytecode to native machine code at runtime.

4. Native Method Interface (JNI)

→ Allows JVM to call and execute native methods written in languages like C/C++.

5. Native Libraries

→ Collection of native libraries required for execution.

How JVM Ensures Platform Independence

→ Java source code is compiled into **bytecode (.class)**, which is not platform-specific.
→ JVM interprets this bytecode and converts it into **machine-specific code** for the underlying OS.
→ Each platform (Windows, Linux, Mac) has its own **implementation of JVM** to handle this conversion.

Ques →	**Explain the step-by-step Java program compilation and execution process. What roles do javac and java commands play?**

Ans →

Step 1: Writing the Java Program

- You create a Java source code file using a text editor (like Notepad, VS Code) and save it with a .java extension.
- **Example:** HelloWorld.java

Step 2: Compilation using javac command

- javac is the **Java Compiler** that translates the source code (written in .java file) into **bytecode**.
- Bytecode is stored in a .class file and is platform-independent.
- Command:

  javac HelloWorld.java

- **Output:** Creates HelloWorld.class file containing bytecode.

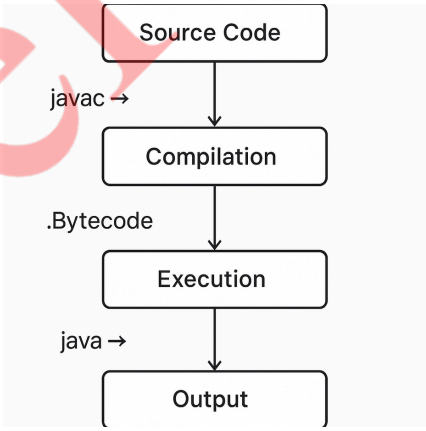
Step 3: Loading the bytecode by JVM

- The **Java Virtual Machine (JVM)** loads the .class file into memory.

Step 4: Execution using java command

- java is the **Java Application Launcher** that starts the JVM and runs the program.
- Command:

  java HelloWorld



#(Note: Do not include .class extension in the command.)

- The JVM interprets the bytecode or uses the **Just-In-Time (JIT)** compiler to convert it into machine code specific to the host operating system and executes it.

Step 5: Displaying Output

- The program runs, and any output (like System.out.println) is displayed on the console.

Roles of javac and java commands

→ **javac**

- Translates human-readable Java source code into bytecode (.class files).
- Performs **syntax checking** during compilation.

→ **java**

- Starts the JVM to execute the compiled bytecode.
- Loads classes, allocates memory, and manages program execution.

Ques →	**What is the role of the final keyword in Java? Explain how it applies to variables, methods, and classes with code examples.**

Ans →

The final keyword in Java is used to declare **constants**, prevent **method overriding**, and stop **class inheritance**. Its role varies based on where it is applied:

→ 1. Final Variables

- A final variable cannot be changed once assigned.
- Acts like a **constant**.

Example:

final int MAX_VALUE = 100;

MAX_VALUE = 200; // Error: cannot assign a value to final variable

- For **reference variables**, the object it points to can be modified, but the reference itself cannot be changed.

Example:

final int[] nums = {1, 2, 3};

nums[0] = 10; // Allowed

nums = new int[5]; // Error: can't change reference

→ 2. Final Methods

- A final method **cannot be overridden** by subclasses.
- Used to prevent changing core logic of a method in child classes.

Example:

```
class Parent {
    final void show() {
        System.out.println("Final method in Parent");
    }
}

class Child extends Parent {
    void show() {  // Error: cannot override final method
        System.out.println("Trying to override");
    }
}

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

```
ERROR!
Main.java:8: error: show() in Child cannot override show() in Parent
    void show() {  // Error: cannot override final method
         ^
  overridden method is final
1 error
```

→ 3. Final Classes

- A final class **cannot be extended.**
- Prevents inheritance to protect implementation.

Example:

```java
final class Vehicle {
    void run() {
        System.out.println("Vehicle is running");
    }
}

class Car extends Vehicle {  // Error: cannot inherit from final class
    void drive() {
        System.out.println("Car is driving");
    }
}

public class Main {
    public static void main(String[] args) {
        Car car = new Car();
        car.run();
        car.drive();
    }
}
```

```
ERROR!
Main.java:7: error: cannot inherit from final Vehicle
class Car extends Vehicle {  // Error: cannot inherit from final class
                  ^
1 error
```


Ques →	**Illustrate the use of static variables, static methods, and static blocks in Java. What are their execution rules and use cases?**

1. Static Variables

- Declared using the static keyword in a class.
- Shared across **all objects** of the class (only **one copy** exists).
- Useful for common properties that should be the same for every object.

Example:

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

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

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

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student(1);
        Student s2 = new Student(23);

        s1.display();
        s2.display();
    }
}
```

Output:

1 ABC College

23 ABC College

Execution Rule:

→ Memory allocated once when the class is loaded.

→ Shared across all instances.

Use Case:

→ Store data common to all objects (e.g., company name, school name).

2. Static Methods

- Declared using static keyword.
- Can be called **without creating an object.**
- Can **only access static data** (directly) and cannot use this or super.

Example:

Ans →

```java
class Utility {
    static void greet() { // static method
        System.out.println("Hello from static method");
    }
}

public class Main {
    public static void main(String[] args) {
        Utility.greet(); // Calling without object
    }
}

```

Output:

Hello from static method

Execution Rule:

→ Invoked using the class name directly.
→ Cannot access non-static variables/methods directly.

Use Case:

→ Utility methods like Math.sqrt(), Math.max(), etc.

3. Static Blocks

- Used for **initializing static data**.
- Executed **only once**, when the class is loaded into memory.

Example:

```java
class Demo {
    static {  // static block
        System.out.println("Static block executed");
    }

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

public class Main {
    public static void main(String[] args) {
        Demo d1 = new Demo();  // static block and constructor executed
        Demo d2 = new Demo();
    }
```

Output:

Static block executed

Constructor executed

Constructor executed

Execution Rule:

→ Runs **before the constructor** and only once when the class is loaded.

Use Case:

→ Initialize static variables, load configurations, or run startup logic.

Ques →	**Differentiate between compile-time and run-time polymorphism with suitable code . Why is polymorphism essential?**

Ans →	Compile-Time Polymorphism	Run-Time Polymorphism
	Also known as **Method Overloading**.	Also known as **Method Overriding**.
	Resolved **at compile time** (early/static binding).	Resolved **at runtime** (late/dynamic binding).
	Achieved by **methods with same name but different parameter lists** in the same class.	Achieved by a **subclass redefining a method** of its parent class.
	Cannot be achieved using inheritance.	Requires **inheritance** between parent and child classes.
	The method to be called is **fixed at compile time.**	The method to be called is **determined at runtime** based on object type.
	Example: Method Overloading.	**Example:** Method Overriding.
	Faster because resolution happens at compile time.	**Slightly slower** due to runtime resolution.

Compile-time Polymorphism (Method Overloading)

```java
class Calculator {
    // Method overloading
    int add(int a, int b) {
        return a + b;
    }

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

public class CompileTimePolymorphism {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println("Sum of integers: " + calc.add(5, 10));      // Calls int version
        System.out.println("Sum of doubles: " + calc.add(5.5, 10.5));   // Calls double version
    }
}
```

Output:

Sum of integers: 15

Sum of doubles: 16.0

Run-time Polymorphism (Method Overriding)

```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 RunTimePolymorphism {
    public static void main(String[] args) {
        Animal obj = new Dog();  // Reference of parent, object of child
        obj.sound();             // Calls Dog's sound() method
    }
}

```

Output:

Dog barks

Why is Polymorphism Essential?

→ 1. Code Reusability

- Allows you to write **generalized code** that works for different types of objects.
- Example: A single method can process objects of different classes.

→ 2. Flexibility and Maintainability

- You can **extend existing code** without modifying it.
- Makes it easier to update and maintain the program.

→ 3. Supports Method Overriding (Dynamic Behavior)

- Allows objects to **decide which method implementation to execute at runtime**.
- Essential for implementing **runtime flexibility** in applications.

→ 4. Simplifies Code Structure

- You can **program to interfaces/superclasses** instead of writing specific code for each subclass.

Ques →	**Explain the concept of abstraction in Java. How is it achieved using abstract classes and interfaces?**

Ans →

Abstraction

- Abstraction is the process of **hiding the implementation details** and showing only the **essential features** of an object.
- It focuses on **what an object does** instead of **how it does it**.
- Abstraction helps in reducing complexity and increases code reusability.

→ Abstraction Achieved in Java:-

Abstraction in Java is achieved in two ways:

1. Using Abstract Classes

- An **abstract class** is declared using the abstract keyword.
- It can have both **abstract methods** (without implementation) and **concrete methods** (with implementation).
- Abstract classes **cannot be instantiated directly**.
- Subclasses must provide implementations for all abstract methods.

Example:

```java
abstract class Animal {
    abstract void sound();

    void eat() {
        System.out.println("Animal is eating");
    }
}

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

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

Output:

Dog barks

Animal is eating

2. Using Interfaces

- An **interface** is a blueprint of a class.
- All methods in an interface are **abstract by default** (until Java 7).
- A class implements an interface using the implements keyword.
- Interfaces support **multiple inheritance** in Java.

Example:

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

class Bike implements Vehicle {
    public void start() {
        System.out.println("Bike starts with a kick");
    }
}

public class Main {
    public static void main(String[] args) {
        Vehicle v = new Bike();
        v.start();
    }
}

```

Output:

Bike starts with a kick

Ques →	**Differentiate between import and static import in Java. Give examples that show when and why static import is preferred.**

Ans →	Import	Static Import
	Used to access classes and interfaces from other packages.	Used to access static members (methods, variables) of a class directly.
	Requires the class name to access static members.	Allows accessing static members without the class name.
	Syntax: import package.ClassName;	Syntax: import static package.ClassName.*;
	Example: Math.sqrt(16);	Example: sqrt(16); (after static import of Math)
	Present since the beginning of Java.	Introduced in Java 5 for cleaner syntax.
	Does not improve code readability for frequent static member calls.	Improves readability when using many static members.

Static Import in Java

- Introduced in Java 5.
- Allows you to access static members (fields and methods) of a class **directly without class name prefix**.
- Saves typing and makes code cleaner for frequently used static members.

When Static Import is Preferred

→ 1. For Utility Methods (Cleaner Code)

Example **Without Static Import**:

```java
import java.lang.Math;

public class Main {
    public static void main(String[] args) {
        double result = Math.sqrt(25) + Math.pow(2, 3);
        System.out.println(result);
    }
}

```

```
Output

13.0

=== Code Execution Successful ===
```

Example **With Static Import**:

```java
import static java.lang.Math.*;

public class Main {
    public static void main(String[] args) {
        double result = sqrt(25) + pow(2, 3); // Cleaner and shorter
        System.out.println(result);
    }
}
```

Why preferred?

- Eliminates repetitive Math. prefix for every static method call.
- Useful when working with many static utility methods.

→ 2. For Constants

Example **Without Static Import**:

```
import java.awt.Color;

public class Main {
    public static void main(String[] args) {
        System.out.println(Color.RED);
    }
}
```

Output

```
java.awt.Color[r=255,g=0,b=0]

=== Code Execution Successful ===
```

Example **With Static Import**:

```
import static java.awt.Color.*;

public class Main {
    public static void main(String[] args) {
        System.out.println(RED); // Cleaner
    }
}
```

Why preferred?

- Simplifies code when **using multiple static constants**.

→ 3. For Testing Frameworks (JUnit, etc.)

Example **Without Static Import**:

```
import org.junit.Assert;


public class TestExample {

    public void test() {
```

```
        Assert.assertEquals(5, 2 + 3);
    }
}
```

Example **With Static Import**:

```
import static org.junit.Assert.*;

public class TestExample {
    public void test() {
        assertEquals(5, 2 + 3); // Simpler
    }
}
```

Why preferred?

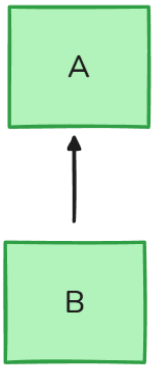
- Reduces clutter and makes **unit tests more readable**.

Ques →	**Ques-Explain the various types of inheritance in java with suitable diagram**

Types of Inheritance in Java

→ 1. Single Inheritance

- A class inherits from one parent class.
- **Java supports this.**



Single Inheritance

Example:

```java
class Vehicle {
    public void start() {
        System.out.println("Vehicle is starting");
    }
    }

class Car extends Vehicle {
    public void start() {
        System.out.println("Car is starting");
    }
}

class Demo {
    public static void main(String[] args) {
        Vehicle vehicle = new Vehicle();
        vehicle.start();

        Car car = new Car();
        car.start();
    }
}
```

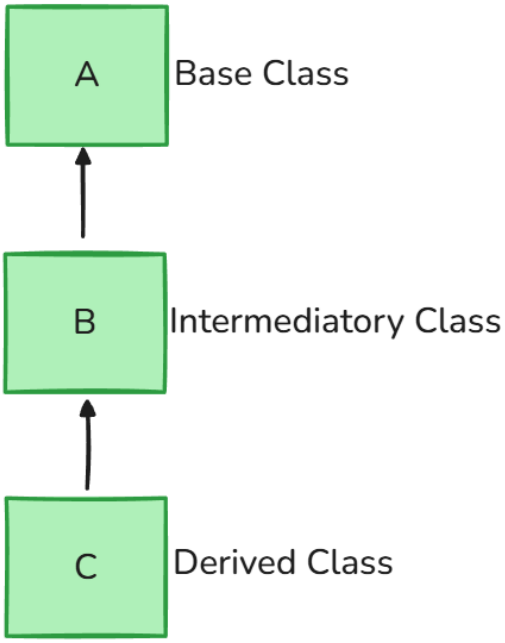
Output:

Vehicle is starting

Car is starting

→ 2. Multilevel Inheritance

- A class inherits from another class, and another class inherits from it.
- Forms a **chain of inheritance**.



Multilevel Inheritance

Example:

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

class A {
    public void display() {
        System.out.println("Bhaag Milkha Bhaag");
    }
}

class B extends A {
    public void soon() {
        System.out.println("Chak de phatte");
    }
}

class C extends B {
    public void exit() {
        System.out.println("Padh lo Beta padh lo");

    }
}
class Demo {
    public static void main(String[] args) {
        C c = new C();
        c.display();
        c.soon();
        c.exit();
    }
}

```

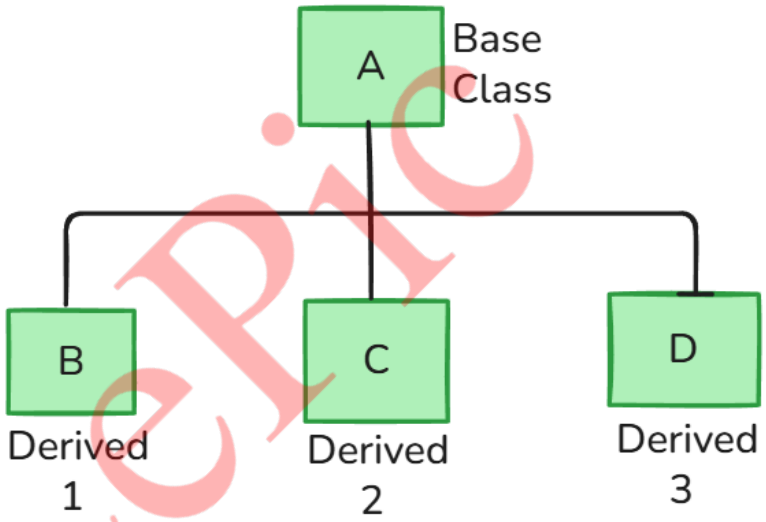
Output:

Bhaag Milkha Bhaag
Chak de phatte
Padh lo Beta padh lo

Ans →

→ 3. Hierarchical Inheritance

- Multiple classes inherit from a **single parent class.**



Example:

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

class Child1 extends Parent {
    void display1() {
        System.out.println("Child1 class");
    }
}

class Child2 extends Parent {
    void display2() {
        System.out.println("Child2 class");
    }
}

class Main {
    public static void main(String args[]) {
        Child1 c1 = new Child1();
        c1.show();
        c1.display1();
        Child2 c2 = new Child2();
        c2.show();
        c2.display2();
    }
}
```


Output:

Parent class

Child1 class

Parent class

Child2 class

→ 4. Multiple Inheritance

- A class inherits from **more than one class**.
- **Java does not support multiple inheritance** with classes to avoid ambiguity (Diamond Problem).

- But it supports multiple inheritance with interfaces.

Example with interfaces:

```java
interface A {
    void methodA();
}

interface B {
    void methodB();
}

class C implements A, B {
    public void methodA() {
        System.out.println("Method A");
    }
    public void methodB() {
        System.out.println("Method B");
    }
}

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

Unit-02.

Ques →	**Ques-Differentiate between Exceptions and Errors in Java. Classify Java exceptions into checked and unchecked categories with examples.**

Ans →	Exceptions	Errors
	Represent issues that occur **due to logical mistakes or external conditions** in the program.	Represent **serious system-level problems** beyond the control of the program.
	These are **recoverable**. The program can handle them using try-catch blocks.	These are **unrecoverable**. The program usually **cannot handle them**.
	Defined in the **java.lang.Exception** hierarchy.	Defined in the **java.lang.Error** hierarchy.
	Examples: NullPointerException, IOException, ArithmeticException.	**Examples:** OutOfMemoryError, StackOverflowError, VirtualMachineError.
	Programmers are expected to handle exceptions in the code.	Programmers are **not expected to handle errors** as they are system failures.
	Used for **application-level problems**.	Used for **system-level problems** (like JVM or hardware issues).

Classification of Java Exceptions

In Java, all exceptions are part of the Throwable hierarchy and are divided into **Checked** and **Unchecked** exceptions:

Checked Exceptions

- Checked at **compile-time**.
- The compiler forces you to **handle** these exceptions using try-catch or throws.
- Occur due to **external factors** like file handling, database operations, or network issues.

Example:

```java
import java.io.*;

public class CheckedException {
    public static void main(String[] args) {
        try {
            FileReader file = new FileReader("test.txt"); // May throw FileNotFoundException
        } catch (FileNotFoundException e) {
            System.out.println("File not found: " + e.getMessage());
        }
    }
}

```

Unchecked Exceptions

- Checked at **runtime**.
- The compiler **does not force you** to handle them.
- Usually caused by **logic errors** in the program.

Example:

```java
public class UncheckedExample {
    public static void main(String[] args) {
        int arr[] = {1, 2, 3};
        System.out.println(arr[5]); // Throws ArrayIndexOutOfBoundsException
    }
}
```

Ques →	**Ques-Write a program that demonstrates the use of try, catch, finally, and throw. Explain the role of each keyword in the exception handling process.**

Ans →

→ try, catch, finally, and throw

```java
public class ExceptionDemo {
    // Method that throws an exception
    static void validateAge(int age) {
        if (age < 18) {
            throw new ArithmeticException("Age is less than 18. Not
                allowed!");
        } else {
            System.out.println("Valid age: Access granted.");
        }
    }

    public static void main(String[] args) {
        try {
            System.out.println("Checking age...");
            validateAge(15); // This will throw an exception
        } catch (ArithmeticException e) {
            System.out.println("Exception caught: " + e.getMessage());
        } finally {
            System.out.println("Finally block executed: Closing
                resources if any.");
        }
    }
}
```

Output

```
Checking age...
Exception caught: Age is less than 18. Not allowed!
Finally block executed: Closing resources if any.

=== Code Execution Successful ===
```

Program Flow

1. **try block**
   → Code in try is executed first.
   → If validateAge(15) throws an exception, control moves to the catch block.
2. **catch block**
   → Handles ArithmeticException and prints the exception message.
3. **finally block**
   → Executes regardless of whether an exception occurs or not.
   → Used to close resources (like files, DB connections).
4. **throw keyword**
   → Used in validateAge() to throw a custom exception when age < 18.

LePic

Ques →	**Ques- Explain the “throws” Keyword in Java with a suitable code.How is it differ from “throw” keyword.**

Ans →

→ The throws keyword in Java is used to declare exceptions in a method signature.
→ It tells the compiler and the caller of the method that this method might throw an exception.
→ The actual throwing of the exception (if needed) is done using the throw keyword

Usage

→ Used in the method declaration to inform the caller about **checked exceptions**.
→ Checked exceptions must either be **handled using try-catch** or **declared using throws**.
→ Syntax:

```
returnType methodName() throws ExceptionType1, ExceptionType2 {
    // method body
}
```

```
import java.io.*;

class Test {
    void readFile() throws IOException {
        FileReader fr = new FileReader("data.txt");
        System.out.println("Reading done.");
    }

    public static void main(String[] args) {
        Test t = new Test();
        try {
            t.readFile();
        } catch (IOException e) {
            System.out.println("Exception handled: " + e);
        }
    }
}
```

Output

```
Exception handled: java.io.FileNotFoundException: data.txt (No such file or directory)
```

throw	throws
Used to actually throw an exception.	Used to declare that a method might throw an exception.
Used inside the method body.	Used with the method signature.
Can throw only one exception at a time.	Can declare multiple exceptions, separated by commas.
Followed by an exception object.	Followed by exception class names.
Mostly used for unchecked exceptions.	Commonly used for checked exceptions.
Syntax: throw new ExceptionType();	Syntax: returnType methodName() throws ExceptionType
Causes immediate termination of the method.	Passes the handling responsibility to the caller.

Ques →	Ques-Discuss the process of creating a user-defined exception in Java. Illustrate with a program that throws a custom exception if age < 18.

Ans →

How to Create a User-Defined Exception in Java?

- In Java, you can create your own exception by **extending the Exception class** (for checked exceptions) or **RuntimeException** (for unchecked exceptions).
- This allows you to throw **custom exceptions** with meaningful messages.

Steps to Create a Custom Exception

→ **1.** **Create a custom exception class** by extending Exception.
→ 2. Throw the custom exception in your program logic using throw.
→ 3. Handle it using try-catch.

```java
// Step 1: Create a custom exception
class UnderAgeException extends Exception {
    public UnderAgeException(String message) {
        super(message);
    }
}

// Main class
public class CustomExceptionExample {
    // Step 2: Method that throws custom exception
    static void validateAge(int age) throws UnderAgeException {
        if (age < 18) {
            throw new UnderAgeException("Age is less than 18. Not eligible to vote.");
        } else {
            System.out.println("Eligible to vote.");
        }
    }

    public static void main(String[] args) {
        try {
            validateAge(28); // Pass age < 18 to trigger exception
        } catch (UnderAgeException e) {
            System.out.println("Exception caught: " + e.getMessage());
        }
    }
}

```


Ques →

Ques-Write a program to read data from a text file and write it to another file.

Ans →

```java
import java.io.*;

public class FileCopy {
    public static void main(String[] args) {
        // Source and destination file paths
        String sourceFile = "source.txt";
        String destFile = "destination.txt";

        try {
            // Create FileReader to read from source file
            FileReader reader = new FileReader(sourceFile);

            // Create FileWriter to write to destination file
            FileWriter writer = new FileWriter(destFile);

            int character;
            // Read character by character from source and write to destination
            while ((character = reader.read()) != -1) {
                writer.write(character);
            }

            // Close the streams
            reader.close();
            writer.close();

            System.out.println("File copied successfully.");
        } catch (IOException e) {
            System.out.println("An error occurred: " + e.getMessage());
        }
    }
}
```

Ques → **Ques-Describe the classes provided by Java under java.io for performing file I/O. How are BufferedReader and BufferedWriter different from FileReader and FileWriter?**

Ans →

→ Java provides several classes in the java.io package to perform **file input/output operations**.
→ These classes handle reading from and writing to **files, streams, and data sources**.

Commonly Used File I/O Classes

→ File
→ Represents the name of a file or directory. Used to **create, delete, or inspect file properties**, but not for reading or writing.

→ FileReader
→ Used to **read character data** from a file (one character at a time). Suitable for **text files**.

→ FileWriter
→ Used to **write character data** to a file. Automatically creates the file if it does not exist.

→ BufferedReader
→ Wraps around FileReader to **read text efficiently** (line-by-line using a buffer). Reduces number of I/O operations.

→ BufferedWriter
→ Wraps around FileWriter to **write text efficiently** using a buffer. Improves performance in large writes.

→ FileInputStream
→ Reads **raw byte data** from a file (used for binary files like images, PDFs, etc.).

→ FileOutputStream
→ Writes **raw byte data** to a file.

→ PrintWriter
→ Used to **write formatted text** to a file like System.out, but for file output.

BufferedReader vs FileReader

LePic

BufferedReader	FileReader
Reads text using a buffer, making it more efficient.	Reads text character by character, less efficient.
Suitable for reading large files or multiple lines.	Suitable for simple, small file reading.
Provides method like readLine() to read a full line.	Does not support readLine(); reads one character at a time.
Needs to wrap another reader (e.g., FileReader).	Can be used directly with a file name.
Fewer disk accesses → better performance.	More disk accesses → slower performance.

BufferedWriter vs FileWriter

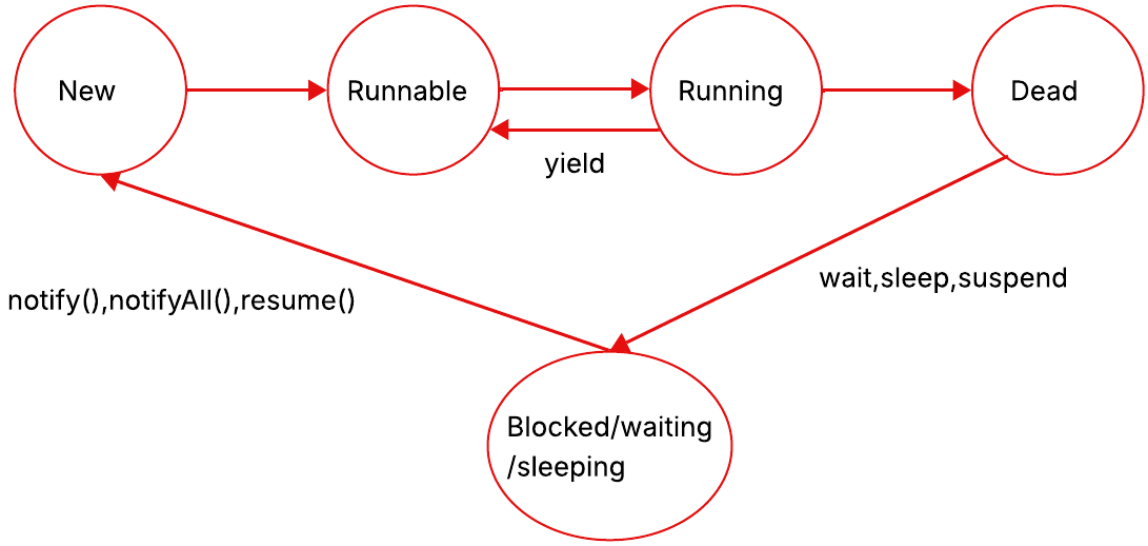
BufferedWriter	FileWriter
Writes text using a buffer, improving speed.	Writes text directly, character by character.
Suitable for writing large amounts of data.	Suitable for simple or short text writing.
Has newLine() method to write line breaks easily.	Does not have newLine() method.
Wraps another writer like FileWriter.	Can be used directly with a file name.
Fewer writes to disk → faster performance.	More writes to disk → slower performance.

Ques →	**Ques-Compare byte streams and character streams in Java.**

Ans →	Byte Stream	Character Stream
	Handles raw binary data (8-bit bytes)	Handles text data (16-bit characters)
	Uses InputStream and OutputStream classes	Uses Reader and Writer classes
	Suitable for binary files like images, audio, etc.	Suitable for text files like .txt, .csv
	Not aware of character encoding	Aware of character encoding (like UTF-8, UTF-16)
	Faster for non-text (binary) data	Better for text processing
	Example: FileInputStream, BufferedOutputStream	Example: FileReader, BufferedWriter

Ques →	What is a thread in Java? Explain the life cycle of a thread with the help of a well-labelled diagram and explanation.

Thread in Java refers to a lightweight **sub-process** or **smallest unit of execution** in a program. It is part of **multithreading**, which allows concurrent execution of two or more parts of a program for maximum CPU utilization.



Ans →

1. New State

→ A thread enters the New state when it is created.
→ It has not started running yet.
→ At this point, the thread object exists in memory, but it is not active.

2. Runnable State

→ When the start method is called, the thread enters the Runnable state.
→ In this state, the thread is ready to run but waiting for the CPU to schedule it.
→ The thread is placed in the runnable queue by the thread scheduler.
→ The actual execution will depend on the operating system's thread scheduling policy.

3. Running State

→ The thread enters the Running state when the CPU assigns it time.
→ The thread's run method begins executing.
→ Only one thread can be in the Running state on a single-core processor at a time.
→ The thread will continue running until it either finishes execution or is moved to another state.

4. Blocked / Waiting / Sleeping State (Non-Runnable States)

→ These are intermediate states where the thread is temporarily inactive.

- **Blocked State**:
  → The thread wants to enter a critical section (like a synchronized block), but another thread holds the lock.
  → It remains blocked until the lock is released.
- **Waiting State**:
  → The thread is waiting indefinitely for another thread's signal.

→ It has released the lock and is paused until it is notified by another thread.

- **Sleeping State**:
  → The thread is deliberately paused for a fixed time.
  → After the time ends, it becomes runnable again.

5. Back to Runnable from Blocked/Waiting/Sleeping

→ A sleeping thread becomes runnable again once the specified time ends.
→ A waiting thread returns to the runnable state when another thread calls notify or notifyAll on the object it was waiting on.
→ A blocked thread becomes runnable once the lock it was waiting for is released.

6. Yield (Running → Runnable)

→ The yield method is used when a running thread voluntarily gives up the CPU.
→ It moves back to the runnable state.
→ The thread scheduler then decides whether to run this thread again or pick another thread of equal priority.
→ It is used to improve fairness among threads of the same priority.

7. Dead State

→ A thread enters the Dead state when its task is completed.
→ This happens either when the run method finishes execution or the thread is terminated due to an exception.
→ A thread in the Dead state cannot be restarted.

Ques →	**Ques-Explain the following-** **a.Thread Synchronisation** **b.Thread Priorities** **c.Multithreading**

a.Thread Synchronization

Need of Thread Synchronization

→ In Java, we can run multiple threads at the same time (this is called multithreading).
→ But sometimes, more than one thread tries to access the **same resource** (like a file, variable, or database).
→ This creates **concurrency problems** – where threads interfere with each other's work, causing **unexpected or wrong results**.

Example to Understand the Problem:

→ Suppose two threads are writing to the **same file** at the same time.
→ One thread is still writing, and at that moment the second thread starts writing too.
→ This may lead to **data corruption**, where one thread's data may overwrite the other's.
→ Or imagine, while one thread is **opening** a file, another one is **closing** it – this could crash the program.

Why Synchronization is Needed:

→ To prevent such issues, we must **control the access** to shared resources.
→ We need to ensure that **only one thread can access the resource at a time**.
→ This process is called **thread synchronization**.

How Java Handles It – Using Monitors:

→ In Java, every object has a built-in **monitor** (also called a lock).
→ A thread can **lock** the monitor when it wants to access a synchronized part of the code.
→ Once a thread locks the monitor, **no other thread can enter the synchronized code** that uses the same monitor.
→ The thread **releases the lock** when it finishes, allowing another thread to take the lock.

Synchronized Block in Java

→ Java provides a **simple way** to manage multithreading problems using **synchronized blocks**.
→ These blocks are used to **protect shared resources** – like variables, files, or objects – that are accessed by multiple threads.

Why Use Synchronized Blocks?

→ If two or more threads try to access the **same resource at the same time**, it may lead to wrong results.
→ To avoid this, Java lets you use **synchronized blocks** to allow **only one thread at a time** to execute a critical section (i.e., the code that accesses the shared resource).

```
synchronized(object_reference) {
    // code that needs to be synchronized
}
```

→ object_reference is the reference to any object whose **monitor (lock)** is used.
→ The thread **locks the monitor** of the object before entering this block.
→ While one thread is inside the synchronized block, **other threads are blocked** from entering any synchronized block locked on the same object.
→ Once the thread **exits the block**, it **releases the lock**, and other waiting threads can enter.

b.Thread Priorities

Thread Priorities in Java

→ In Java, each thread has a **priority** which helps the thread scheduler decide **which thread to run first**, when multiple threads are ready.
→ In simple terms, think of priority like **a number that shows how "urgent" a thread's task is.**
→ The priority is represented using **integers from 1 to 10**, where:

→ Higher number = **Higher priority**
→ Lower number = **Lower priority**

Predefined Constants for Thread Priorities:

→ **Thread.MIN_PRIORITY**
→ Sets the minimum priority of a thread.
→ Value = 1

→ **Thread.NORM_PRIORITY**
→ Sets the default (normal) priority of a thread.
→ Value = 5

→ **Thread.MAX_PRIORITY**
→ Sets the maximum priority of a thread.
→ Value = 10

How to Get and Set Priority of a Thread:

→ Java provides two methods to manage thread priorities:

Ans →

1. **public final int getPriority()**
   → Returns the **current priority** of the thread.
   → **Example:** int p = thread.getPriority();
2. **public final void setPriority(int newPriority)**
   → Sets the thread's priority to newPriority.
   → You must pass a value between **1 and 10**, otherwise it throws IllegalArgumentException.
   → **Example:** thread.setPriority(8);

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Running thread: " + Thread.currentThread().getName() +
                           ", Priority: " + Thread.currentThread().getPriority());
    }
}

public class TestPriority {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();
        MyThread t3 = new MyThread();

        t1.setPriority(Thread.MIN_PRIORITY); // 1
        t2.setPriority(Thread.NORM_PRIORITY); // 5
        t3.setPriority(Thread.MAX_PRIORITY); // 10

        t1.start();
        t2.start();
        t3.start();
    }
}
```

```
Output
Running thread: Thread-0, Priority: 1
Running thread: Thread-2, Priority: 10
Running thread: Thread-1, Priority: 5
```

c.Multithreading

→ Multithreading is a feature in Java that allows **multiple threads to run at the same time**.
→ It helps in **executing two or more parts of a program concurrently**, which leads to **better CPU usage**.
→ Each thread is a **lightweight process** – it runs inside the main program and shares memory with other threads.
→ This makes multithreading faster and more efficient than running separate processes.

Terms:

→ **Thread** → A part of a program that runs independently but shares memory with other threads.
→ **Process** → A running instance of a program.

→ **Threads within a process** → Share memory and resources, making communication easy.

Why Use Multithreading?

→ Improves the **performance** of programs (especially on multi-core CPUs)
→ Allows **parallel execution** (e.g., downloading a file while browsing other data)
→ Makes applications **more responsive** (e.g., in GUI-based apps)

Different Ways to Create Threads in Java

Java provides **two main ways** to create a thread:

→ 1. Extending the Thread class

→ You create a new class that extends Thread and override its run() method.
→ Then create an object of your class and call start() to run the thread.

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

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

Output

```
Thread is running using Thread class.
```

→ 2. Implementing the Runnable interface

→ You create a class that implements Runnable and override its run() method.
→ Then create a Thread object and pass your Runnable object to it, then call start().

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

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

Output

```
Thread is running using Runnable interface.
```

Ques →	**Ques-Discuss the concept of inter-thread communication. How do wait(), notify(), and notifyAll() methods work together to coordinate threads?**

Inter-thread Communication in Java

→ In multithreading, multiple threads may need to **communicate or coordinate** with each other to work properly.
→ This is called **inter-thread communication**.
→ Java provides three methods to help threads communicate:
→ wait()
→ notify()
→ notifyAll()

Why is Inter-thread Communication Needed?

→ Sometimes a thread needs to **pause and wait** for another thread to complete a task.
→ **Example:** A **consumer thread** should wait until the **producer thread** puts data into a shared buffer.
→ Without communication, threads may **keep running blindly**, leading to wasted CPU time or inconsistent results.

How it Works

→ All three methods belong to the **Object class** (because every object has a monitor/lock in Java).
→ These methods must be called **inside a synchronized block or method**.

1. wait() Method

→ Causes the current thread to **wait (pause execution)** until another thread calls notify() or notifyAll() on the same object.
→ It also **releases the lock** held by the thread so other threads can acquire it.
→ Syntax: object.wait();
→ Must be in a synchronized block on that object.

→ **Example use:**
→ A thread finds that the condition it needs (like data being available) is not met, so it calls wait().

2. notify() Method

→ Wakes up **one waiting thread** that has called wait() on the same object.
→ Only **one random thread** is chosen if multiple threads are waiting.
→ The awakened thread can continue **only after** it regains the lock.
→ Syntax: object.notify();

3. notifyAll() Method

→ Wakes up **all threads** that are waiting on the object's monitor.
→ Only one thread will acquire the lock at a time (the others must wait).
→ Syntax: object.notifyAll();

How They Work Together – Simple Flow:

1. One thread (Thread A) checks a condition (e.g., buffer is empty).
2. If the condition is not met, it calls wait() and **pauses**.
3. Another thread (Thread B) does some work (e.g., adds data) and then calls notify() or notifyAll().
4. The waiting thread (Thread A) **wakes up**, reacquires the lock, and **continues** from where it paused.

Example – Producer-Consumer

Ans →

```
class SharedData {
    int data;
    boolean available = false;

    synchronized void produce(int value) {
        while (available) {
            try { wait(); } catch(Exception e) {}
        }
        data = value;
        available = true;
        System.out.println("Produced: " + data);
        notify();
    }

    synchronized void consume() {
        while (!available) {
            try { wait(); } catch(Exception e) {}
        }
        System.out.println("Consumed: " + data);
        available = false;
        notify();
    }
}

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

        Thread producer = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                obj.produce(i);
                try { Thread.sleep(100); } catch (Exception e) {}
            }
        });

        Thread consumer = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                obj.consume();
                try { Thread.sleep(100); } catch (Exception e) {}
            }
        });

        producer.start();
        consumer.start();
    }
}
```

Output

```
Produced: 1
Consumed: 1
Produced: 2
Consumed: 2
Produced: 3
Consumed: 3
Produced: 4
Consumed: 4
Produced: 5
Consumed: 5
```

Unit-03

Ques →	**Ques-Define “functional Interface” and “lambda expression”. Implement any 4 built-in functional Interfaces**

Functional Interface

- A **functional interface** is an interface in Java that contains **exactly one abstract method.**
- It can have multiple **default** or **static** methods.
- Functional interfaces are used extensively in **Lambda expressions** and **Streams API**.
- Annotated with @FunctionalInterface (optional but recommended).

Example:

```
@FunctionalInterface

interface MyInterface {

   void display(); // single abstract method

}
```

Lambda Expression

- A **Lambda Expression** is a **short-cut syntax** to provide an implementation for the abstract method of a functional interface.
- It removes the need for writing anonymous inner classes.

Syntax:

```
(parameter_list) -> { body }
```

Example:

```
MyInterface obj = () -> System.out.println("Hello from Lambda!");

obj.display();
```

4 Built-in Functional Interfaces

Java provides functional interfaces in java.util.function package.

1. Consumer<T>

- Represents an operation that takes an input and **returns nothing**.

```java
import java.util.function.Consumer;

public class ConsumerExample {
    public static void main(String[] args) {
        Consumer<String> print = s -> System.out.println("Hello, " + s
            );
        print.accept("Java");
    }
}
```

Ans →

Output:

Hello, Java

2. Supplier<T>

- Represents a function that takes **no input but returns a result**.

```java
import java.util.function.Supplier;

public class SupplierExample {
    public static void main(String[] args) {
        Supplier<Double> randomValue = () -> Math.random();
        System.out.println("Random Value: " + randomValue.get());
    }
}
```

Output:

Random Value: 0.012402340021491876

3. Predicate<T>

- Represents a function that takes an input and returns a **boolean** result.

```java
import java.util.function.Predicate;

public class PredicateExample {
    public static void main(String[] args) {
        Predicate<Integer> isEven = n -> n % 2 == 0;
        System.out.println("Is 10 even? " + isEven.test(10));
    }
}
```

Output:

Is 10 even? true

4. Function<T, R>

- Represents a function that takes an input and returns an output.

```java
import java.util.function.Function;

public class FunctionExample {
    public static void main(String[] args) {
        Function<String, Integer> stringLength = s -> s.length();
        System.out.println("Length of 'Java': " + stringLength.apply
            ("Java"));
    }
}
```

Output:

Length of 'Java': 4

Ques →	**What is the role of the @FunctionalInterface annotation? Write a program that uses a custom functional interface with lambda implementation.**

Ans →

→ @FunctionalInterface is an annotation introduced in **Java 8**.

→ It is used to **mark an interface as a functional interface**, which means:

→ **The interface must contain only one abstract method.**

Why Use It?

→ Ensures that the interface is valid for use with **lambda expressions.**
→ Gives a **compile-time error** if more than one abstract method is added.
→ Improves **readability** and **intent clarity** for the developer.

```java
@FunctionalInterface
interface Calculator {
    int operate(int a, int b);
}

public class LambdaDemo {
    public static void main(String[] args) {
        Calculator addition = (a, b) -> a + b;
        Calculator multiplication = (a, b) -> a * b;

        System.out.println("Sum: " + addition.operate(5, 3));
        System.out.println("Product: " + multiplication.operate(5, 3));
    }
}
```

Output:

```
Sum: 8
Product: 15
```

→ Calculator is a **functional interface** because it has **one abstract method**: operate(int, int).

→ Two **lambda expressions** are created for:

- addition: (a, b) -> a + b
- multiplication: (a, b) -> a * b

→ Each lambda is passed to the operate() method and produces the result.

Ques → **Differentiate between intermediate and terminal operations in the Stream API. Demonstrate with the use of filter, map, and collect.**

Ans →	Intermediate Operations	Terminal Operations
	Return a new stream	Return a final result (value or collection)
	Do not trigger execution	Trigger execution of the stream pipeline
	Can be chained	Only one terminal operation allowed
	Examples: map(), filter(), sorted()	Examples: collect(), forEach(), count()

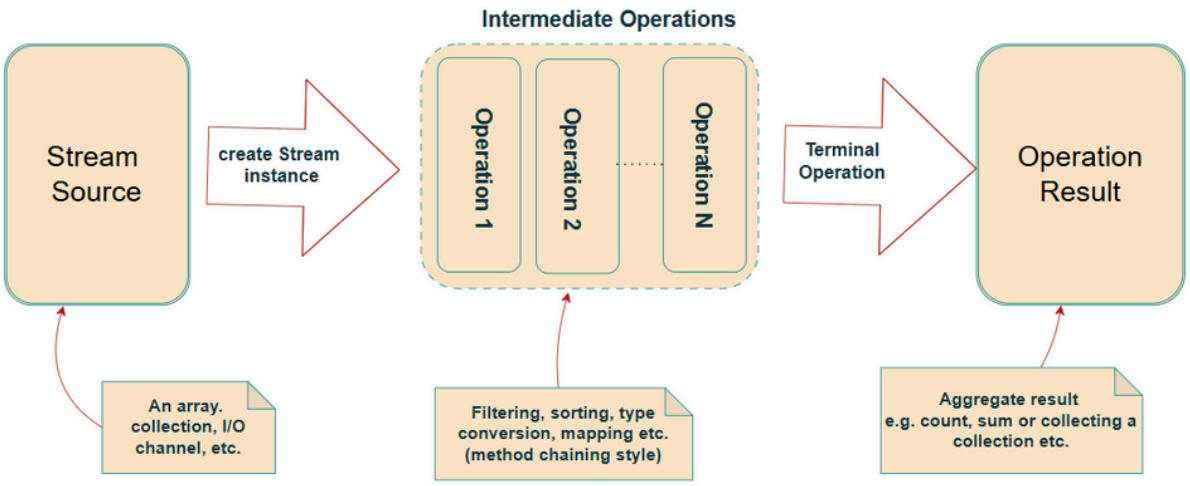
```java
import java.util.*;
import java.util.stream.*;

public class StreamExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Aditya", "Nitya", "Niharika", "Yashvander");

        List<String> result = names.stream()
                                   .filter(name -> name.startsWith("N"))
                                   .map(String::toUpperCase)
                                   .collect(Collectors.toList());

        System.out.println("Filtered and Uppercase Names: " + result);
    }
}
```

Output:

```
Filtered and Uppercase Names: [NITYA, NIHARIKA]
```


Ques →	Explain the architecture and working of Java Stream API.

Ans →



Architecture of Java Stream API

The Stream API in Java (introduced in Java 8) is designed for **processing collections of objects** in a **functional and declarative style**.

Components of Stream API Architecture:

→ 1. Data Source

- The starting point of streams.
- Can be a **Collection** (List, Set), Array, or I/O channel.

→ 2. Stream Pipeline

A stream pipeline consists of **three parts**:

a. Intermediate Operations

- Transform a stream into another stream.
- Examples: map(), filter(), sorted().
- These are **lazy** and not executed until a terminal operation is invoked.

b. Terminal Operations

- Produces a result or side-effect and **triggers the pipeline execution**.
- Examples: forEach(), collect(), reduce().

c. Short-circuiting Operations

- Stops processing once a certain condition is met.
- Examples: limit(), findFirst(), anyMatch().

Working of Stream API

1. A **stream is created** from a data source.
2. One or more **intermediate operations** are applied (these are **lazy**).
3. A **terminal operation** is applied, which executes the stream pipeline.
4. Streams process data **sequentially** or **parallel** based on configuration.

Example: Stream API in Action:

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

public class StreamExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("John", "Jane", "Jack",
            "Jill");

        // Stream pipeline: filter, map, and forEach
        names.stream()
            .filter(s -> s.startsWith("J"))       // Intermediate
                operation
            .map(String::toUpperCase)             // Intermediate
                operation
            .forEach(System.out::println);        // Terminal
                operation
    }
}
```

Output

```
JOHN
JANE
JACK
JILL
```


Ques →	**What is the forEach() method in Java?Explain with a suitable code.**

Ans →

→ forEach() is a **method in Java 8 and above**, used to **iterate through elements in a collection** like List, Set, etc.

→ It is part of the **Iterable interface** and is also used with **streams.**

→ You pass a **lambda expression or method reference** to forEach() that defines what to do with each element.

Why use forEach()?

→ Makes your code **more concise** and readable than traditional for or while loops.

→ Ideal when you want to **perform an action on each element** in a collection.

```java
import java.util.*;

public class ForEachExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Aditya", "Niharika", "Nitya");

        names.forEach(name -> {
            System.out.println("Hello, " + name);
        });
    }
}
```

Output:

```
Hello, Aditya
Hello, Niharika
Hello, Nitya
```

→ names is a List of strings.
→ forEach() goes through each element and executes the lambda name -> System.out.println(...).
→ The lambda takes each name one by one and prints a greeting.

Ques →	**How does “static method” differs from “default method.**

LePic

Ans →	Static Method	Default Method
	Declared using the static keyword in **interfaces or classes**.	Declared using the default keyword in **interfaces only**.
	Belongs to the **interface/class** and **cannot be overridden** in implementing classes.	Belongs to the **interface** but **can be overridden** in implementing classes.
	Can be called **using the interface name** (for interface static methods).	Called using an **object of the implementing class.**
	Cannot be inherited by implementing classes.	**Inherited** by implementing classes.
	Used to provide **utility/helper methods** in interfaces.	Used to provide **default implementations** in interfaces.

	Example:
	`interface Test {`
	`  static void display() {`
	`  System.out.println("Static method");`
	`  }`
	`}`
	
	`public class Main {`
	`  public static void main(String[] args) {`
	`  Test.display(); // Called using interface name`
	`  }`
	`}`
	
	Output:
	Static method
		Example:
		`interface Test {`
		`  default void display() {`
		`  System.out.println("Default method");`
		`  }`
		`}`
		
		`class Demo implements Test {}`
		
		`public class Main {`
		`  public static void main(String[] args) {`
		`  Demo obj = new Demo();`
		`  obj.display(); // Called using object`
		`  }`
		`}`
		
		Output:
		Default method

Ques →	**Explain Base-64 encoding and decoding with a suitable code.**

Ans →

- **Base64 encoding** is a way of converting **binary data** (like images, files, or text) into a **textual format** using a set of 64 characters:
  A–Z, a–z, 0–9, +, /
- It is commonly used to:
  → Encode data for safe transmission over **text-based protocols** (like email, JSON, XML).
  → Store binary data in text files.

Decoding is the reverse process of converting Base64 encoded text back to the original data.

Base64 Encoding in Java

Example:

```java
import java.util.Base64;

public class EncodeExample {
    public static void main(String[] args) {
        String original = "HelloJava";

        // Encoding
        String encoded = Base64.getEncoder().encodeToString(original.getBytes());
        System.out.println("Encoded: " + encoded);
    }
}
```

Output:

Encoded: SGVsbG9KYXZh

Base64 Decoding in Java

Example:

```java
import java.util.Base64;

public class DecodeExample {
    public static void main(String[] args) {
        String encoded = "SGVsbG9KYXZh";

        // Decoding
        byte[] decodedBytes = Base64.getDecoder().decode(encoded);
        String decoded = new String(decodedBytes);

        System.out.println("Decoded: " + decoded);
    }
}
```

Output:

Decoded: HelloJava

- **Encoding:** Converts bytes to Base64 string.
  Base64.getEncoder().encodeToString(byte[])
- **Decoding:** Converts Base64 string back to bytes.
  Base64.getDecoder().decode(String)
- Useful for encoding images, passwords (before hashing), and transmitting binary data safely.

Variants in Base64

- **Basic:** Default encoding/decoding.
- **URL Safe:** Replaces + and / with - and _ for URLs.
  Base64.getUrlEncoder()
- **MIME:** Adds line breaks for long Base64 data.
  Base64.getMimeEncoder()

Ques →	**Explain “type annotations” and “repeating annotations” with suitable example.**

Ans →

1. Type Annotations in Java

→ Introduced in Java 8.

→ Allows annotations to be used anywhere a type is used, not just on declarations.

→ Useful for static analysis tools, frameworks, and advanced type-checking.

Example:

```java
import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

// Using target annotation to annotate a type
@Target(ElementType.TYPE_USE)

// Annotating a function
@interface A{}

public class Main {

    public static void main(String[] args) {

    // Annotating a variable
    @A String string = "Hello";
    System.out.println(string);
     abc();
    }

    // Annotating return type of a function
    static @A int abc() {

        System.out.println("This function's  return type is annotated");

        return 0;
    }
}

```

Output:

Hello

This function's return type is annotate

Repeating Annotations in Java

→ Introduced in Java 8.

→ Allows applying the same annotation multiple times on a single declaration.

→ Achieved using a container annotation.

Example:

```java
import java.lang.annotation.Repeatable;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

// Repeating annotation
@Retention(RetentionPolicy.RUNTIME)
@interface Roles {
    Role[] value();
}

// Repeating annotation
@Repeatable(Roles.class)
@Retention(RetentionPolicy.RUNTIME)
@interface Role {
    String value();
}

// Applying repeating annotations
@Role("Admin")
@Role("User")
public class RepeatingAnnotationExample {
    public static void main(String[] args) {
        Role[] roles = RepeatingAnnotationExample.class.getAnnotationsByType(Role.class);
        for (Role r : roles) {
            System.out.println("Role: " + r.value());
        }
    }
}
```

Output:

Role: Admin

Role: User

Ques →	**Briefly describe the “Yield Keyword”. Provide a suitable code to show the working of yield keyword in switch Expression.**

Ans →

yield keyword in Java

- The yield keyword was introduced in Java 13 as part of switch expressions.
- It is used to return a value from a case block in a switch expression.
- It helps make switch expressions concise and more powerful than traditional switch statements.

Key Features of yield:

→ Works only in switch expressions (not in old switch statements).
→ Helps return a value from multi-line case blocks.
→ Replaces break when you need to pass a value.

Example Program: Using yield in Switch Expression

```java
public class YieldExample {
    public static void main(String[] args) {
        String grade = "B";

        String result = switch (grade) {
            case "A" -> "Excellent";
            case "B" -> {
                System.out.println("Processing for grade B");
                yield "Good"; // yield returns "Good" as result
            }
            case "C" -> "Average";
            default -> "Invalid grade";
        };

        System.out.println("Result: " + result);
    }
}
```

Output

Processing for grade B

Result: Good

Ques →	**Explain the following terms-** **a.Sealed Classes.** **b.text blocks** **c.records** **d.local variable type inference**

Ans →

a. Sealed Classes

→ Introduced in **Java 15 (preview)** and **Java 17 (standard)**.

→ A **sealed class** restricts which classes can extend or implement it.

→ You define which specific classes are allowed to be subclasses.

Syntax Example:

```
public sealed class Vehicle permits Car, Truck {}

final class Car extends Vehicle {}
final class Truck extends Vehicle {}
```

b. Text Blocks

→ Introduced in **Java 15**.

→ Allows you to write **multi-line strings** using """ triple quotes.

→ Makes writing long strings like HTML, SQL, or JSON **cleaner and more readable**.

Example:

```
String html = """
    <html>
        <body>
            <h1>Hello, Devansh!</h1>
        </body>
    </html>
    """;
```


c. Records

→ Introduced in **Java 16**.

→ A record is a **special class** for storing immutable data (like a data holder or DTO).

→ It automatically provides:

- Constructor
- Getters
- toString(), equals(), and hashCode()

Example:

```
public record Student(int id, String name) {}


Student s = new Student(101, "Aditya");
System.out.println(s.name());  // Aditya
```

d. Local Variable Type Inference

→ Introduced in **Java 10** using the keyword var.

→ Allows you to **declare local variables without specifying the type** — compiler infers it automatically.

Example:

```
var name = "Niharika";       // inferred as String
var age = 24;               // inferred as int
var list = new ArrayList<String>();  // inferred as ArrayList<String>
```

Unit-04

Ques →	Differentiate between the Collection and Collections classes in Java.

Ans →	Collection	Collections
	It is an interface in the java.util package.	It is a utility class in the java.util package.
	It is the root interface of the Collection Framework.	It contains static utility methods for operations on collections.
	Used to represent a group of objects as a single unit (like List, Set).	Used to perform operations like sorting, searching, reversing, etc.
	Examples: List, Set, Queue are subinterfaces of Collection.	Examples: Collections.sort(list), Collections.reverse(list)
	Cannot be instantiated directly (since it's an interface).	Can be used directly through static methods.
	Example: Collection<String> list = new ArrayList<>();	Example: Collections.sort(myList);

Ques →	Describe the working of the Iterator interface. How is it different from a ListIterator

Iterator Interface

→ Iterator is a **Java interface** used to **traverse (iterate) over elements of a collection** one by one.

→ It works with **any collection** that implements the Collection interface like ArrayList, HashSet, etc.

Methods of Iterator

→ **hasNext()** → Returns true if the iteration has more elements
→ **next()** → Returns the next element in the collection
→ **remove()** → Removes the current element (optional operation)

Example

```java
import java.util.*;

public class IteratorExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Aditya", "Nitya", "Niharika");

        Iterator<String> it = names.iterator();

        while (it.hasNext()) {
            String name = it.next();
            System.out.println(name);
        }
    }
}
```

Output:

```
Aditya
Nitya
Niharika
```

Ans →

ListIterator

→ ListIterator is a **specialized iterator** for **lists only** (like ArrayList, LinkedList).

→ It allows:

- **Forward and backward traversal**
- **Modifying elements while iterating**
- **Getting the index** of current position

```java
import java.util.*;

public class ListIteratorExample {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>(List.of("A", "B", "C"));

        ListIterator<String> lit = names.listIterator();

        System.out.println("Forward:");
        while (lit.hasNext()) {
            System.out.println(lit.next());
        }

        System.out.println("Backward:");
        while (lit.hasPrevious()) {
            System.out.println(lit.previous());
        }
    }
}
```

Output:

```
Forward:
A
B
C
Backward:
C
B
A
```

Iterator	ListIterator
Can be used with all collections like Set, List, Queue	Can be used only with List types (e.g., ArrayList, LinkedList)
Supports only forward traversal	Supports both forward and backward traversal
Does not allow adding or updating elements	Allows adding and modifying elements during iteration
Provides only hasNext(), next(), and remove()	Provides hasNext(), next(), hasPrevious(), previous(), add(), set(), remove()
Cannot access element index	Can access index using nextIndex() and previousIndex()
Always starts from beginning	Can start from any index using listIterator(index)

Ques →	How is Stack implemented using Java's Collection Framework? Write a program to demonstrate basic push/pop operations.

Ans →

```java
import java.util.Stack;

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


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

        System.out.println("Stack after pushes: " + stack);


        int popped = stack.pop();
        System.out.println("Popped element: " + popped);

        System.out.println("Stack after pop: " + stack);
    }
}
```

Output:

```
Stack after pushes: [10, 20, 30]
Popped element: 30
Stack after pop: [10, 20]
```


Ques → **Compare ArrayList, LinkedList, and Vector**

LePic

Ans →	ArrayList	LinkedList	Vector
	Based on dynamic array	Based on doubly linked list	Based on dynamic array
	Fast for accessing elements	Slow for access (needs traversal)	Fast for accessing elements
	Insertion/deletion is slow (due to shifting)	Insertion/deletion is fast (node links adjusted)	Insertion/deletion is slow (same as ArrayList)
	Not synchronized (not thread-safe)	Not synchronized (not thread-safe)	Synchronized (thread-safe)
	Better for read-heavy operations	Better for insert/delete-heavy operations	Slower due to thread safety
	Grows by 50% of current size	Grows by adding new nodes	Grows by doubling its size
	Allows multiple null values	Allows multiple null values	Allows multiple null values
	Preferred in single-threaded access	Preferred for frequent add/remove	Preferred in multi-threaded access

Ques →	What is the difference between HashSet, LinkedHashSet, and TreeSet? Explain with examples.

LePic

Ans →	HashSet	LinkedHashSet	TreeSet
	Does not maintain any order of elements	Maintains insertion order	Maintains elements in sorted (natural) order
	Fastest for search, insert, and delete	Slightly slower than HashSet due to ordering	Slowest among the three (uses tree structure)
	Allows one null element	Allows one null element	**Does not allow null** (throws NullPointerException if used)
	Backed by a hash table	Backed by a linked hash table	Backed by a Red-Black Tree
	Best when order doesn't matter	Best when insertion order matters	Best when sorted data is needed

1. HashSet Example

```java
import java.util.*;

public class Demo {
    public static void main(String[] args) {
        HashSet<String> set = new HashSet<>();
        set.add("Apple");
        set.add("Banana");
        set.add("Mango");
        set.add("Apple");
        System.out.println(set);
    }
}
```

Output

```
[Apple, Mango, Banana]
```

2. LinkedHashSet Example

```
import java.util.*;

public class Demo {
    public static void main(String[] args) {
        LinkedHashSet<String> set = new LinkedHashSet<>();
        set.add("Apple");
        set.add("Banana");
        set.add("Mango");
        set.add("Apple");
        System.out.println(set);
    }
}
```

Output

```
[Apple, Banana, Mango]
```

3. TreeSet Example

```
import java.util.*;

public class Demo {
    public static void main(String[] args) {
        TreeSet<String> set = new TreeSet<>();
        set.add("Banana");
        set.add("Apple");
        set.add("Mango");
        System.out.println(set);
    }
}
```

Output

```
[Apple, Banana, Mango]
```

Ques →	How does HashSet ensure uniqueness of elements? Explain the role of equals() and hashCode()

Ans →

How HashSet Ensures Uniqueness

→ HashSet does **not allow duplicate elements.**
→ Internally, it uses a **HashMap** to store data.
→ When you add an element, HashSet uses the following steps to decide **if it's already present or not.**

Procedure

→ **Step 1: Compute hashCode**
→ When you add an element, Java first calls the **hashCode()** method of that object.
→ This gives an integer (called a hash) that tells where the element should be stored in memory (i.e., in a bucket).

→ **Step 2: Compare using equals()**
→ If two elements have the **same hash code**, Java doesn't assume they are the same.
→ It then calls the **equals()** method to check **actual content equality.**
→ Only if equals() returns true, the object is considered a duplicate and **won't be added** again.

Role of hashCode()

→ hashCode() returns an int value that represents the **bucket index.**
→ It **narrows down** where to look for possible duplicates.
→ But it's not enough alone — two different objects can have the same hash code (called a collision).

Role of equals()

→ equals() does the **actual comparison** between two objects.
→ Even if two objects have the same hash code, equals() must return true for one to be rejected as a duplicate.
→ If equals() returns false, the object is **considered different** and is added.

```java
import java.util.*;

class Student {
    int id;
    String name;

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

    public boolean equals(Object o) {
        Student s = (Student) o;
        return this.id == s.id && this.name.equals(s.name);
    }

    public int hashCode() {
        return id + name.hashCode();
    }
}

public class Demo {
    public static void main(String[] args) {
        HashSet<Student> set = new HashSet<>();
        set.add(new Student(1, "Aditya"));
        set.add(new Student(1, "Aditya"));
        System.out.println(set.size());
    }
}
```

Ques →	**Describe how custom objects can be sorted in Java using Comparable and Comparator**

Sorting with Comparable

→ Use Comparable when you want the **object itself** to define how it should be sorted.
→ You must **implement the Comparable interface** and override the compareTo() method.
→ This is called **natural sorting** (like ascending order of numbers or alphabetical order of names).

Example: Sorting Students by Roll Number (Ascending)

```
import java.util.*;

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

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

    public int compareTo(Student s) {
        return this.roll - s.roll;
    }

    public String toString() {
        return roll + " " + name;
    }
}
public class Demo {
    public static void main(String[] args) {
        List<Student> list = new ArrayList<>();
        list.add(new Student(3, "Niharika"));
        list.add(new Student(1, "Nitya"));
        list.add(new Student(2, "Aditya"));

        Collections.sort(list);

        for (Student s : list) {
            System.out.println(s);
        }
    }
}
```

Output

```
1 Nitya
2 Aditya
3 Niharika
```

Ans →

Sorting with Comparator

→ Use Comparator when you want to **sort objects in different ways,** without changing the class itself.

→ You **pass the sorting logic separately** using compare() method.

→ Ideal for **sorting by different fields,** like name, marks, or in reverse order.

```java
import java.util.*;

class Student {
    int roll;
    String name;

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

    public String toString() {
        return roll + " " + name;
    }
}

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

public class Demo {
    public static void main(String[] args) {
        List<Student> list = new ArrayList<>();
        list.add(new Student(3, "Aditya"));
        list.add(new Student(1, "Nitya"));
        list.add(new Student(2, "Niharika"));

        Collections.sort(list, new NameComparator());

        for (Student s : list) {
            System.out.println(s);
        }
    }
}
```

Output

```
3 Aditya
2 Niharika
1 Nitya
```

Ques →	**What is the need for the Java Collection Framework? How does it solve the limitations of arrays in Java?**

Ans →

Why Java Collection Framework Is Needed

→ Arrays in Java are useful for storing multiple elements, but they come with several limitations.
→ The **Java Collection Framework (JCF)** was introduced to solve these problems by providing a set of **ready-made classes and interfaces** to handle groups of objects more flexibly and efficiently.

Limitations of Arrays in Java

→ Fixed Size
→ Once an array is created, its size **cannot be changed.**

→ Only Homogeneous Data
→ Arrays can hold **only one type** of data (e.g., only integers or only strings).

→ Lacks Built-in Methods
→ Arrays do not provide useful methods like **sorting, searching, insertion, deletion**, etc.

→ Difficult to Insert/Delete
→ You must **manually shift elements,** which is error-prone and inefficient.

→ No Type Safety with Object Arrays
→ Using Object[] loses type safety; can lead to runtime errors.

How Java Collection Framework Solves These Problems

→ Resizable
→ Collections like ArrayList or LinkedList can **grow or shrink dynamically.**

→ Supports Heterogeneous Objects (with Object)
→ Collections can store any type of object and even allow **custom types** using generics.

→ Rich Set of Methods
→ Provides methods for **sorting, searching, inserting, removing,** etc.

→ Type Safety with Generics
→ Using generics, you can create type-safe collections (e.g., ArrayList<String>).

→ More Data Structures Available
→ Offers ready-to-use implementations of **List, Set, Queue, Map,** etc., unlike arrays which are just linear.

→ Better Performance for Complex Tasks
→ Collections like HashMap, TreeSet, or PriorityQueue are optimized for specific operations.

Unit-05

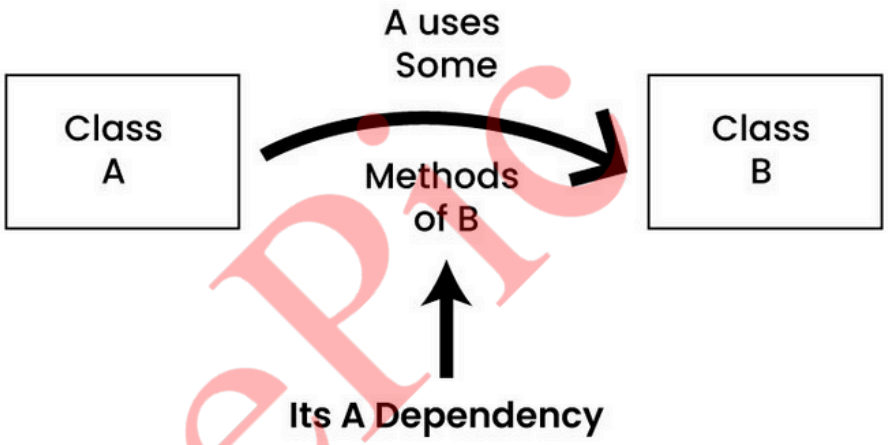
Ques →	**Explain the concept of Dependency Injection (DI) in Spring.**

Ans →

In object-oriented programming, the Dependency Injection (DI) design pattern is a technique that reduces the connection between system components, making the code more modular, testable, and maintainable. Classes frequently rely on other classes to carry out their tasks in a typical software program.

For Example: Car class might depend on a Engine class to run. Without DI, the Car class would directly create or manage the Engine instance within its code, which makes the two classes tightly coupled. This approach can create problems, particularly when you need to test, extend, or modify the classes in the future.

- Dependency Injection solves this problem by injecting the dependencies (like the Engine ones in the Car example) into the class from an external source, rather than having the class create them.
- In simpler terms, DI allows you to "inject" the things a class needs (its dependencies) from the outside, instead of letting the class create or manage them itself.



In Dependency Injection, the dependencies of a class are injected from the outside, rather than the class creating or managing its dependencies internally. This pattern has four main roles:

→ Client:

- The client is the component or class that depends on the services provided by another class or module.
- The Client does not provide dependencies, it only receives them from the Injector. (The Injector is responsible for providing dependencies, not the Client).

→ Service:

- The service is the component or class that provides a particular functionality or service that the client depends on.

- It focuses on offering particular functionality and is made to be independent of the clients.

→ Injector:

- Instances of services must be created and injected into the client by the injector.
- It is aware of the dependencies of the client and provides the necessary services during runtime.

→ Interface:

- The interface defines the contract or set of methods that a service must implement.
- Clients rely on these interfaces rather than specific implementations, promoting flexibility and the ability to swap implementations.

Applicability

Below are the key scenarios where dependency injection is a valuable approach:

→ Loose Coupling and Reusability: Objects don't create their own dependencies, breaking tight connections and making them more independent.

→ Testability: Inject mock or test doubles for dependencies, allowing you to test individual objects in isolation without relying on external systems or services.

→ Maintainability and Flexibility: Dependency injection frameworks often manage dependencies, making it easier to track and configure them.

→ Scalability and Extensibility: In large-scale applications, DI helps manage complex dependency graphs and enables easier scaling and extension.

→ Cross-Cutting Concerns: Inject services for logging, security, caching, or other cross-cutting concerns that are used across multiple components, avoiding code duplication and promoting a consistent approach.

Ques →	**What is Inversion of Control (IoC) in the Spring Framework? How does the IoC container manage the lifecycle of Spring Beans?**

Ans →

Inversion of Control (IoC)

→ Inversion of Control (IoC) is a **design principle** used to remove **tight coupling** between objects.
→ It means: **Instead of the object creating its dependencies**, the **control of creating and injecting dependencies is given to a container**.

→ In Spring, the **IoC container** is responsible for:
→ Creating objects (beans)
→ Injecting dependencies
→ Managing the entire bean lifecycle

How Spring IoC Container Works

→ The Spring IoC container is the **core part** of the framework.
→ It reads configuration (XML, annotations, or Java config) to know which beans to create and how to inject them.

→ Two main types of containers:
→ BeanFactory (basic, lazy loading)
→ ApplicationContext (advanced, eager loading, used more often)

Bean Lifecycle Managed by IoC Container

Here's how the Spring IoC container manages the **full lifecycle** of a Spring bean:

→ 1. Instantiation
→ Spring creates an object of the class (the bean).

→ 2. Populate Properties (Dependency Injection)
→ Spring injects the dependencies (using setter or constructor injection).

→ 3. Set Bean Name (Aware interface)
→ If the bean implements BeanNameAware, Spring gives it its name.

→ 4. Set Bean Factory (Aware interface)
→ If the bean implements BeanFactoryAware, it gets access to the factory that created it.

→ 5. Pre-Initialization (BeanPostProcessor)
→ Spring allows custom logic to run before initialization.

→ 6. Initialization
→ If the bean implements InitializingBean or defines a custom init-method, it is called now.

→ 7. Post-Initialization (BeanPostProcessor)
→ Spring allows additional processing after initialization.

→ **8. Bean is Ready to Use**
→ The bean is now available for the application to use.

→ **9. Destruction**
→ When the application shuts down, Spring calls destroy() or custom destroy-method.

Ques → **Describe the different bean scopes in Spring: Singleton, Prototype, Request, Session, Application**

Ans →

1. Singleton (Default Scope)

→ Only **one instance** of the bean is created **per Spring container**.
→ All requests for that bean will return the **same object**.
→ **Best for stateless beans** (shared across application).

2. Prototype

→ A **new bean instance** is created **every time** it is requested from the container.
→ Useful when you want to create **stateful or independent beans**.

3. Request (Web-aware Scope)

→ A **new bean instance** is created for **every HTTP request**.
→ Used in **web applications** to keep request-specific data.

4. Session (Web-aware Scope)

→ One bean instance is created **per HTTP session**.
→ All requests within the same session share the same object.

5. Application (Web-aware Scope)

→ One bean instance is created for **entire ServletContext (application)**.
→ Shared across **all sessions and requests**.

Scope	Bean Created	Lifetime	Use Case
Singleton	Once per container	Entire app	Shared configuration or service
Prototype	Every request	Short	Custom logic, dynamic use
Request	Once per HTTP request	Per request	Request-specific data
Session	Once per HTTP session	Per session	User-specific state
Application	Once per app context	App-wide	Shared data for all users

Ques →	How can @PostConstruct and @PreDestroy annotations be used to manage bean lifecycle?

Ans →

→ Spring provides lifecycle interfaces like InitializingBean and DisposableBean,
→ But using annotations like @PostConstruct and @PreDestroy makes the code **cleaner and easier to maintain.**

→ These annotations are part of **javax.annotation** package and are recognized by the Spring **IoC container.**

What @PostConstruct Does

→ Runs a method **immediately after dependency injection is done** (after the bean is created and properties are set).
→ Commonly used for **initial setup**, loading resources, or validation.

Example:

```
public class MyBean {
    public void init() {
        System.out.println("Bean is initialized.");
    }

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

Output:

```
Bean is initialized.
```

What @PreDestroy Does

→ Runs a method **just before the bean is destroyed** (when the Spring container is shutting down).
→ Useful for **cleanup tasks** like closing connections or releasing resources.

Example:

```
@Component
public class MyBean {

    @PreDestroy
    public void cleanup() {
        System.out.println("Bean is being destroyed.");
    }
}
```

```
Bean is being destroyed.
```

Ques →	**What is Autowiring in Spring? Explain different autowiring modes.**

Ans →

Autowiring

→ **Autowiring** is a feature in Spring that allows the **automatic injection of bean dependencies** without explicitly using @Bean or XML wiring.
→ It saves you from writing boilerplate code to manually link beans.
→ Spring automatically resolves and injects the **right bean** into a class **based on type, name, or constructor**.

Need of Autowiring

→ Reduces configuration code
→ Makes your classes **loosely coupled**
→ Promotes **Dependency Injection** best practices

Different Autowiring Modes in Spring

Spring provides **five types of autowiring modes** (when using XML). For annotation-based configuration (@Autowired), most common modes are byType and constructor.

1. no (default)

→ **Autowiring is disabled.**
→ You need to manually wire dependencies using ref in XML or setter methods.

→ **Used when:** You want full control over bean wiring.

2. byName

→ Spring looks for a **bean with the same name** as the property name and injects it.
→ Works with **setter injection**.

Example:

```
<bean id="engine" class="com.car.Engine"/>
<bean id="car" class="com.car.Car" autowire="byName"/>
```

3. byType

→ Spring searches for a **bean of the same type** as the dependency and injects it.
→ **Fails if more than one bean of same type** is available (ambiguous).

Example:

```
<bean id="engine1" class="com.car.Engine"/>
<bean id="car" class="com.car.Car" autowire="byType"/>
```

4. constructor

→ Spring injects dependency using the **constructor** instead of setter.
→ It matches the constructor's parameter **by type** (and optionally by name).

Example:

```
<bean id="engine" class="com.car.Engine"/>
<bean id="car" class="com.car.Car" autowire="constructor"/>
```

Annotation-Based Autowiring (@Autowired)

→ More commonly used in modern Spring applications (including Spring Boot)

```
@Component
public class Car {

    @Autowired
    private Engine engine;
}
```

→ Spring will **autowire Engine** based on type (by default).
→ If multiple beans of same type exist, use @Qualifier.

Ques →	**Define Aspect-Oriented Programming (AOP) in Spring. How is it useful in cross-cutting concerns like logging and security?**

Ans →

→ AOP is a **programming paradigm** that allows you to **separate common code (cross-cutting concerns)** from your core business logic.

→ In simple terms:
It lets you run some common code (like logging, security checks, transactions) before, after, or around your actual business methods.

Cross-Cutting Concerns

→ These are parts of a program that affect **multiple layers or modules** of your application, like:

→ Logging
→ Security checks
→ Transaction management
→ Exception handling
→ Performance monitoring

→ Without AOP, you'd need to **repeat the same code** in every class/method — leading to code duplication and tight coupling.

Working

→ Spring uses **proxies** behind the scenes to inject the cross-cutting logic.
→ You define **aspects**, and Spring AOP automatically **weaves that logic** into the target methods at runtime.

LePic

Term	Meaning
Aspect	Common code that you want to apply across different parts (e.g., logging logic)
Advice	Action taken at a specific point (e.g., before, after, around a method)
JoinPoint	A point during execution (like method call) where advice can be applied
Pointcut	A condition that selects which JoinPoints to apply the advice to
Weaving	Connecting the aspect (logic) with actual code (done at runtime in Spring)

→ **Without AOP:**

You'd write logging code manually in every method:

```
public void transfer() {
    System.out.println("Start log");
    // transfer logic
    System.out.println("End log");
}
```


→ **With AOP:**

You write the logging logic in one **aspect class**, and Spring automatically applies it to all required methods:

```java
@Aspect
@Component
public class LoggingAspect {

    @Before("execution(* com.bank.service.*.*(..))")
    public void logBefore() {
        System.out.println("Logging before method...");
    }
}
```

Ques →	**Discuss the key features of the Spring Framework.state the suitable reasons of how spring boot is better than spring framework**

Ans →

Features of Spring Framework

→ **1. Lightweight and Modular**
→ Spring is lightweight and can be used in any Java application. You can use only what you need — it is modular.

→ **2. Dependency Injection (DI)**
→ Spring promotes loose coupling by managing object creation and dependency injection automatically.

→ **3. Aspect-Oriented Programming (AOP)**
→ Allows separation of cross-cutting concerns like logging, security, and transactions from business logic.

→ **4. MVC Web Framework**
→ Spring provides a powerful MVC architecture for building web applications easily.

→ **5. Transaction Management**
→ Provides a consistent programming model for transaction management, across different transaction APIs.

→ **6. Integration with Other Frameworks**
→ Easily integrates with Hibernate, JPA, JDBC, JMS, and other enterprise technologies.

→ **7. Security**
→ Spring Security module helps in managing authentication, authorization, and protection against common attacks.

→ **8. Testing Support**
→ Offers built-in support for unit and integration testing using JUnit and mock objects.

→ **9. Centralized Configuration**
→ XML and annotation-based configuration allow centralized management of beans and app logic.

Why Spring Boot Is Better Than Spring Framework

Reason	Spring Framework	Spring Boot
Setup	Manual configuration of dependencies, XML files, etc.	Zero-configuration – auto-setup using spring-boot-starter dependencies
Web Server	Need to deploy WAR on external server like Tomcat	Comes with embedded Tomcat, no need for external deployment
Boilerplate Code	Requires a lot of XML or Java config	Auto-configures most components, less code
Project Structure	More complex setup needed	Provides a default structure, easier to get started
Dependency Management	Need to manage all versions manually	Uses Spring Boot Starters which include correct dependencies automatically
Production-Readiness	No built-in tools for monitoring	Comes with Actuator for health check, metrics, etc.
Command-line Execution	Cannot run directly	Can be run with one command: java -jar app.jar

Ques →	How does Spring Boot differ from Spring MVC in terms of configuration and rapid development support?

Ans→	Spring MVC	Spring Boot
	Requires manual configuration for beans, view resolvers, dispatcher servlet, etc.	Provides automatic configuration using @SpringBootApplication and auto-setup
	Needs external servers like Tomcat or Jetty to deploy WAR files	Comes with embedded servers (Tomcat, Jetty) — runs as a standalone JAR
	Slower development due to boilerplate and XML/Java config	Fast development using Spring Initializr, starters, and minimal setup
	No built-in monitoring — you must add tools manually	Has built-in monitoring via Spring Boot Actuator
	Limited convention support — explicit configuration needed	Follows convention over configuration — less code, more defaults
	Complex project startup and folder structure	Pre-defined structure and simplified startup process

Ques →	Define REST architecture. How does Spring Boot support RESTful web services using annotations like @RestController and @RequestMapping?

Ans →

REST Architecture

→ REST stands for Representational State Transfer.
→ It is an architectural style for designing networked applications, especially web services.

Principles of REST

→ **1. Stateless**
→ Each client request must contain all information — the server does not store anything about previous requests.

→ **2. Client-Server Separation**
→ Client and server are separate — they can evolve independently.

→ **3. Uniform Interface**
→ All interactions are done through standard HTTP methods:

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

→ **4. Resource-Based URLs**
→ Everything is treated as a resource and identified by a URI (e.g., /students/101).

→ **5. Use of Standard HTTP**
→ Uses standard HTTP response codes (200 OK, 404 Not Found, etc.)

How Spring Boot Supports RESTful Web Services

Spring Boot makes it **easy to create REST APIs** using **annotations** to define endpoints and behaviors.

Annotations in Spring Boot for REST

→ **@RestController**

→ A specialized version of @Controller that automatically **returns data as JSON or XML**, instead of a view.
→ Combines @Controller + @ResponseBody.

```
@RestController
public class StudentController {
    // your methods
}
```

→ **@RequestMapping**

→ Maps HTTP requests to specific **URL paths** and **handler methods**.
→ Can be used at class level and method level.

```
@RequestMapping("/students")
public String getStudents() {
    return "List of Students";
}
```

Ques →	**Explain the structure of a typical Spring Boot application. What are the roles of @SpringBootApplication and SpringApplication.run()?**

Ans →

Structure of a Typical Spring Boot Application

A Spring Boot project generally has the following structure:

```
src/
├── main/
│   ├── java/
│   │   └── com/example/project/
│   │       ├── ProjectApplication.java        → Main class
│   │       ├── controller/                    → Handles HTTP requests
│   │       ├── service/                       → Business logic
│   │       └── repository/                    → Data access layer (JPA, JDBC, etc.)
│   └── resources/
│       ├── application.properties (or .yml)   → Configuration
│       └── static/, templates/                → Web resources (if any)
```

Role of @SpringBootApplication

→ This is a **meta-annotation** that combines:

- @Configuration → Marks the class as a source of bean definitions
- @EnableAutoConfiguration → Enables Spring Boot's auto-configuration feature
- @ComponentScan → Automatically scans the package and sub-packages for Spring components (@Component, @Service, etc.)

→ It tells Spring Boot to:

- Scan the package
- Load configuration
- Start auto-configured beans

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

Role of SpringApplication.run()

→ This method **starts the entire Spring Boot application**.

→ It performs these tasks:

→ Loads the application context

→ Starts the embedded web server (e.g., Tomcat)

→ Initializes all beans and auto-configured components

→ Bootstraps the application and listens for requests (if it's a web app)

Internal Functioning

1. SpringApplication.run() is called
2. Spring Boot auto-scans all components in the package

3. Beans are created and dependencies injected
4. If it's a web application, the embedded server (Tomcat/Jetty) starts
5. The application is ready to serve requests

ThankYou For watching!!

for notes visit → http://lepic.mzelo.com

Join Our Telegram Channel → lepic official