**Core Java Concepts**

**1. OOPs Concepts:**

**i. Class and Object:**

* A class is a blueprint or template that defines the properties (variables) and behaviors (methods) of objects. It does not occupy memory until an object is created.
* An object is a real-world instance of a class that has its own state (data) and behavior (methods).It is created from a class using the new keyword.

**ii. Constructor and it's overloading:**

* A constructor is a special method in a class that is used to initialize objects.It has the same name as the class and does not have a return type.It is called automatically when an object is created.
* Constructor overloading means having more than one constructor in the same class with different parameter lists.It allows you to create objects in different ways (with or without initial values).

**iii. Static Members:**

* Static Variable: A static variable belongs to the class, not to any specific object.It is shared by all objects of the class.
* Static Method: A static method can be called without creating an object of the class. It can access only static variables directly.
* Static Block: A static block runs once when the class is loaded into memory. It is used to initialize static data.

**iv. Final Keyword:**

* Final Variable: A final variable value cannot be changed once assigned. **Example**: final int x = 10;
* Final Method: A final method cannot be overridden in a subclass.**Example:** class A {final void display() {System.out.println("Hello");}}
* Final Class: A final class cannot be inherited by any other class. **Example:** final class A { }class B extends A { }

**v. Access Modifier:**

Access modifiers define the visibility (where a member can be accessed) of classes, variables, and methods.

* public: Can be accessed from anywhere — inside or outside the class, even from other packages.
* private: Can be accessed only within the same class. It is not visible to other classes.
* protected: Can be accessed within the same package and also by subclasses (even in other packages)
* default (no modifier): If no modifier is used, it is accessible only within the same package.

**vi. Encapsulation:**

Encapsulation means binding data (variables) and methods (functions) together in a single unit (class). It is used to protect data by keeping it private and providing public getter and setter methods to access it.

**Example:**

class Student {

private int id;

private String name;

public void setId(int i) {id = i;}

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

public int getId() {return id;}

public String getName() {return name;}

}

class Test {

public static void main(String[] args) {

Student s = new Student();

s.setId(101);

s.setName("Dharshini");

System.out.println(s.getId() + " " + s.getName());

}

}

**vi. Inheritance:**

Inheritance is the process where one class (child) gets the properties and methods of another class (parent). It helps in code reusability and method overriding.

1. Single Inheritance: One class inherits from another single class.

class A {

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

}

class B extends A {

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

}

B inherits from A.

2. Multilevel Inheritance: A class inherits from another class, which itself is inherited from another.

class A { void showA() { } }

class B extends A { void showB() { } }

class C extends B { void showC() { } }

C inherits from B, and B inherits from A.

3. Hierarchical Inheritance: Multiple classes inherit from the same parent class.

class A { void showA() { } }

class B extends A { void showB() { } }

class C extends A { void showC() { } }

Both B and C inherit from A.

**vii. Polymorphism:**

Polymorphism means "many forms" — the same method or operation behaves differently based on the object or situation. It allows one name to have multiple behaviors.

1. Compile-time Polymorphism (Method Overloading): Happens when multiple methods in the same class have the same name but different parameters. The method is decided at compile time.

**Example:**

class Calculator {

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

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

}

The correct add() method is chosen during compilation.

2. Runtime Polymorphism (Method Overriding): Happens when a child class provides a new implementation for a method that is already defined in its parent class. The method is decided at runtime.

**Example:**

class Animal {

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

}

class Dog extends Animal {

void sound() {System.out.println("Bark");}

}

class Test {

public static void main(String[] args) {

Animal a = new Dog();

a.sound();

}

}

**viii. Abstraction:**

Abstraction means showing only essential features and hiding the internal details. It focuses on what an object does, not how it does it.

Abstraction can be achieved in two ways:

1. Using Abstract Classes
2. Using Interfaces

1. **Abstract Class**: A class declared with the keyword abstract. It can have abstract methods (without body) and non-abstract methods (with body). It cannot be instantiated (objects cannot be created directly). It is used for partial abstraction.

**Example:**

abstract class Animal {

abstract void sound();

void sleep() {System.out.println("Sleeping");}

}

class Dog extends Animal {

void sound() {System.out.println("Bark");}

}

**2. Interface:**

An interface is a blueprint of a class that contains abstract methods (methods without a body). It is used to achieve abstraction and multiple inheritance in Java. A class that uses an interface must implement all its methods.

* Declared using the keyword interface.
* All methods are public and abstract by default.
* A class uses the keyword implements to use an interface.
* One class can implement multiple interfaces.
* Interfaces help in loose coupling and code reusability.

**Example:**

import java.util.\*;

interface Calculator

{

double add(double a, double b);

double sub(double a, double b);

double mul(double a, double b);

double div(double a, double b);

}

interface Scientific {double power(double a, double b);}

class BasicCalculator implements Scientific,Calculator­­­

{­­­

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

public double sub(double a, double b) {return a-b;}­

public double mul(double a, double b) {return a\*b;}

public double div(double a, double b){return a/b;}

public double power(double a, double b){ return Math.pow(a,b);}

}

public class App {

public static void main(String[] args)

{

Scanner s=new Scanner(System.in);

BasicCalculator app=new BasicCalculator();

System.out.println("Enter first Value: ");

double a=s.nextDouble();

System.out.println("Enter Second Value: ");

double b=s.nextDouble();

System.out.println("Enter Any Symbol(+,-,\*,/,^): ");

char sys=s.next().charAt(0);

double ans;

switch(sys)

{

case '+':

ans=app.add(a,b);

System.out.println("Answer: "+ans);

break;

case '-':

ans=app.sub(a,b);

System.out.println("Answer: "+ans);

break;

case '\*':

ans=app.mul(a,b);

System.out.println("Answer: "+ans);

break;

case '/':

ans=app.div(a,b);

System.out.println("Answer: "+ans);

break;

case '^':

ans=app.power(a,b);

System.out.println("Answer: "+ans);

break;

}}}

**2. Enums:**

An enum is a special type in Java used to define a set of constant values (fixed values that do not change). It helps make the code more readable and type-safe.

* Declared using the keyword enum.
* Enum constants are public, static, and final by default.
* Used when you have a fixed set of related constants — like days, directions, colors, etc.
* You can also add methods and variables inside an enum.

**Example:**

enum Day {

SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY;

}

class TestEnum {

public static void main(String[] args) {

Day d = Day.MONDAY;

System.out.println(d);

}

}

**3. Annotations:**

Annotations in Java are special markers or metadata that provide information to the compiler or JVM about the code. They do not change the actual logic, but they help give instructions or add information. They start with the symbol @.

Commonly Used Annotations

**1. @Override**: Used when a method overrides a method from its parent class.

Helps the compiler check correctness.

**Example:**

class Parent {

void show() { }

}

class Child extends Parent {

@Override

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

}

**2. @Deprecated**: Marks a method or class as outdated (not recommended to use). The compiler gives a warning if it’s used.

**Example:**

@Deprecated

void oldMethod() {System.out.println("This method is deprecated");}

**3. @SuppressWarnings**: Used to ignore compiler warnings for specific code parts.

**Example:**

@SuppressWarnings("unchecked")

void example() {

List list = new ArrayList(); // unchecked warning suppressed

list.add("Hello");

}

**4. @SafeVarargs**: Used to suppress warnings when using varargs with generics (from Java 7+). It ensures that the method does not perform unsafe operations.

**Example:**

@SafeVarargs

private final void print(List<String>... lists) {

for (List<String> list : lists) {System.out.println(list);}

}

**5. @FunctionalInterface:** Used to mark an interface that has exactly one abstract method (used in Lambda Expressions).

**Example:**

@FunctionalInterface

interface MyFunction {void display();}

Meta-Annotations: These are annotations that apply to other annotations (used when creating custom annotations).

**6. @Target:** Specifies where the annotation can be used (e.g., on a method, class, field, etc.).

**7. @Retention**: Specifies how long the annotation should be retained:

SOURCE: discarded after compilation

CLASS: kept in class file but ignored by JVM

RUNTIME: available at runtime (via reflection)

**8. @Documented:** Indicates that the annotation should be included in Javadoc.

**9. @Inherited:** Allows a subclass to inherit an annotation from its superclass.

**10. Custom Annotation:** You can create your own annotation using @interface.

**4. Serialization:**

Serialization is the process of converting an object into a byte stream, so it can be saved to a file, sent over a network, or stored in a database.

Deserialization is the reverse process — converting the byte stream back into an object.

Normally, Java objects exist only in memory (RAM). If the program stops, the data is lost. Serialization lets us save the object’s data and reuse it later.

**Example**

import java.io.\*;

class Student implements Serializable {

int id;

String name;

Student(int id, String name) {

this.id = id;

this.name = name;

}

}

Serialization (Saving object to file)

Student s1 = new Student(101, "Dharshini");

FileOutputStream fos = new FileOutputStream("student.ser");

ObjectOutputStream oos = new ObjectOutputStream(fos);

oos.writeObject(s1);

oos.close();

fos.close();

System.out.println("Object serialized!");

Deserialization (Reading object from file)

FileInputStream fis = new FileInputStream("student.ser");

ObjectInputStream ois = new ObjectInputStream(fis);

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

ois.close();

fis.close();

System.out.println("ID: " + s.id);

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

**Output:**

ID: 101

Name: Dharshini

**transient Keyword:** If you don’t want a variable to be saved during serialization, mark it as transient:

transient String password;

(This field will be skipped and become null when deserialized.)

**serialVersionUID**: Used to maintain version control of a serialized class.

private static final long serialVersionUID = 1L;

(If the class changes later, this ID ensures deserialization still works.)

**5. Exception Handling:**

Exception Handling is a way to handle runtime errors in Java — so the program doesn’t crash and can continue executing smoothly. We use try–catch–finally blocks for this.

**Example:**

try {

int a = 10 / 0; // ❌ ArithmeticException

} catch (ArithmeticException e) {

System.out.println("Cannot divide by zero!");

} finally {

System.out.println("Finally block always runs!");

}

**Multiple Catch Blocks**: Used when different exceptions might occur.

try {

int[] arr = new int[3];

arr[5] = 10; // ArrayIndexOutOfBoundsException

} catch (ArithmeticException e) {

System.out.println("Math error");

} catch (ArrayIndexOutOfBoundsException e) {

System.out.println("Array index error");

} catch (Exception e) {

System.out.println("Other exception");

}

Always keep specific exceptions before the general Exception class.

**throw Keyword**: Used to manually throw an exception.

throw new ArithmeticException("Invalid operation");

**throws Keyword**: Used in method declaration to say that the method might throw an exception.

void readFile() throws IOException {

// code that may cause IOException

}

**Custom Exception:** You can create your own exception class by extending Exception or RuntimeException.

class MyException extends Exception {

MyException(String msg) {super(msg);}

}

public class Demo {

public static void main(String[] args) {

try {

throw new MyException("Custom error message!");

} catch (MyException e) {

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

}

}

}

Some of the common exceptions in Java are ArithmeticException (when dividing by zero), NullPointerException (when trying to access a null object), ArrayIndexOutOfBoundsException (when using an invalid array index), and NumberFormatException (when converting a string to a number fails). There are also checked exceptions like IOException (for input/output errors) and FileNotFoundException (when a file is missing).

**6. Threads**

A thread in Java is a lightweight subprocess — the smallest unit of execution. When you run a Java program, there is always at least one thread running called the main thread. Threads allow multiple parts of a program to run concurrently, which helps in performing multiple tasks at the same time.

For example, in a web browser, one thread can handle user input while another loads data in the background.

**Creating Threads**

There are two main ways to create a thread in Java:

1. By Extending the Thread Class: You can create a thread by extending the Thread class and overriding the run() method.

class MyThread extends Thread {

public void run() {

System.out.println("Thread is running...");

}

}

public class Demo {

public static void main(String[] args) {

MyThread t = new MyThread();

t.start(); // starts the thread and calls run()

}

}

start() creates a new thread and calls the run() method internally.

2. By Implementing the Runnable Interface: This is the preferred way because Java does not support multiple inheritance. Using Runnable, your class can extend another class and still define thread behavior.

class MyRunnable implements Runnable {

public void run() {

System.out.println("Runnable thread is running...");

}

}

public class Demo {

public static void main(String[] args) {

Thread t = new Thread(new MyRunnable());

t.start();

}

}

Here, Runnable is a functional interface, meaning it has only one abstract method (run()).

**7. Thread States:**

**1. New State**: When a thread is created but not yet started. The thread object exists, but no system resources are allocated yet.

Thread t = new Thread();

**2. Start / Runnable State**: When you call the start() method, the thread moves from new to runnable. It means the thread is ready to run and waiting for the CPU to schedule it. It may start running immediately or after some time depending on the CPU.

t.start();

**3. Running State:** When the CPU picks the thread, it starts executing the code inside the run() method.A thread can move back and forth between runnable and running based on CPU scheduling.

public void run() {

System.out.println("Thread is running...");

}

**4. Sleep / Waiting State**: A thread enters sleeping or waiting state when it temporarily stops execution. It can happen because:

You call Thread.sleep(time) (sleeping)

You call wait() (waiting for another thread)

It’s waiting to acquire a lock (blocked)

The thread goes back to runnable once the waiting time is over or another thread notifies it.

try {

Thread.sleep(1000); // TIMED\_WAITING

} catch (InterruptedException e) {

e.printStackTrace();

}

**5. Dead (Terminated) State:** When the run() method finishes, the thread’s life ends. Once a thread is dead, it cannot be restarted.

System.out.println("Thread finished!");

**Thread Lifecycle:**

NEW → RUNNABLE → RUNNING → WAITING/SLEEPING → DEAD

**Example Flow:**

Thread created → NEW

start() called → RUNNABLE

CPU executes → RUNNING

sleep() or wait() → WAITING

Task finished → DEAD

**8. Comparable vs Comparator**

**1. Comparable – “Default Way to Compare”**

**Definition:**

Comparable is an interface that allows a class to define its own natural way of ordering — how its objects should be compared.

**Syntax:**

public interface Comparable<T> {

int compareTo(T other);

}

**Meaning:**

If a class implements Comparable, it must define the logic in the compareTo() method that explains how one object is compared with another.

**Example:**

class Student implements Comparable<Student> {

int id;

String name;

Student(int id, String name) {

this.id = id;

this.name = name;

}

public int compareTo(Student other) {

return this.id - other.id;

}

}

Now when you do:

Collections.sort(studentList);

It automatically sorts by id, because that’s the natural order defined in the class.

**2. Comparator – “Custom Way to Compare”**

**Definition:**

Comparator is an interface used to write custom comparison logic outside the class — useful when you want different sorting options.

**Syntax:**

public interface Comparator<T> {

int compare(T o1, T o2);

}

**Meaning:**

You can create multiple comparators for different sorting logic — for example, by name, age, or marks.

**Example (Anonymous Class):**

Comparator<Student> nameComparator = new Comparator<Student>() {

public int compare(Student s1, Student s2) {

return s1.name.compareTo(s2.name);

}

};

**Example (Lambda Expression):**

Comparator<Student> nameComparator = (s1, s2) -> s1.name.compareTo(s2.name);

Then use:

Collections.sort(studentList, nameComparator);

This sorts the students by name, not by id.