1. Compare abstract class and interface in terms of multiple inheritence. When would you prefer to use an abstract class and when an interface.

Ans: In oop multiple inheritence means that a class can inherit from more than one parent class or multiple types.

Comparision between abstract class and interface in terms of multiple inheritence.

Feature	Abstract class	Interface
Inheritance type	Single inheritance only	Multiple inheritence supported.
keyword	abstract class	inteface
Method	can have both abstract and concrete methods	can have abstract methods, default, static and private methods
constructor	can have constructor	can't have constructor
Access Modifier	Methods can be public, private, protected or default	All methods are implicitly public abstract
Fields	can have instance variable and constants	only public static final constants

When to use abstract class:

① You want to share code (common implementation methods)

② You want to control access modifier.

③ You need constructor

④ When the classes are closely related.

When to use interface:

① You need multiple inheritence.

② You're defining capabilities.

③ You are using Functional programming pattern.

④ When the class can be unrelated.

2. How does encapsulation ensure data security and integrity? Show with a bank account class using private variable and validate methods such as setAccountNumber(String), setInitialBalance(double) that reject null, negative empty values.

⇒ Encaptulation a core principle of object oriented programming enhances data security and integrity by bundling data and methods that operate on that data within a single unit and restrict direct access to the data.

It protect data from

① Direct unauthorized access.
② Invalid or harmful input.
③ Inconsistent object states.

Work process:

```
class BankAccount {
    private String accountNumber;
    private double balance;

public void setAccountNumber(String accNo){
    if (accNo == null || accNo.trim().isEmpty()) {
        System.out.println("Invalid account number");
    }
    else{
        accountNumber = accNo;
    }
}

public void setInitialBalance(double amount){
    if (amount<0){
        System.out.println{"Invalid balance amount"};
    }
    else {
        balance = amount;
    }
}

public String getAccountNumber(){
    return accountNumber; }
public double getBalance() {
    return balance; }
}
```

5

```
public class BankApp {
    public static void main(String[] args) {
        BankAccount acc = new BankAccount();
        acc.setAccountNumber("Acc2304");
        acc.setInitialBalance(1000.00);
        System.out.println("Ac No:" + acc.getAccountNumber());
        System.out.println("Ac Bal:" + acc.getAccountBalance());
    }
```

* accountNumber and balance are private, so they cannot be modified directly.
* Setter methods validate inputs before assignment.
* Invalid values like null, empty string, or negative balance are rejected.
* This ensure safe, consistent and secure data handling.

Conclusion: Encapsulation protects data by restricting direct access and ensure integrity by allowing only validate data through controlled methods.

6

Describe how JDBC manages communication between a java application and a relational data base. Outline the steps involved in executing a SELECT query and fetching results. Include error handling with try catch and finally blocks.

JDBC is an API that allows Java application with to interact with relational database like MySQL

How JDBC manages communication:

JDBC act as a bridge between Java and the database using:

① DriverManager → loads the JDBC driver.

② Connection → establishes the session.

③ Statement → sends SQL commands.

④ Resultset → holds the data returned from queries.

Here's a concise outline of the steps to execute a SELECT query using JDBC with error handling.

① Load the JDBC Driver,

② Establish a connector.

③ Prepare SQL query.

4. Set Parameter
5. Execute the query
6. Process the result
7. Handle Exceptions
8. Closure resource in finally block.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class JdbcSelectDemo{
    public static void main(String[] args){

        Connection con = null;
        Statement stmt = null;
        ResultSet rs = null;

    try {
            class.forname("com.mysql.cj.jdbc.Driver");   // Load JDBC driver

    // establish connection
        con = DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/testdb", "root", "password");

        stmt = con.createStatement(); --> create statement
    rs = stmt.executeQuery("SELECT * FROM student");
```

```
// Resultset
    System.out.println("ID \t NAME");
    System.out.println("-------");
    while(rs.next()){
        int id = rs.getInt("id");
        String name = rs.getString("name");
    System.out.println(id +"\t" + name);
    }}
catch (ClassNotfoundException e){
    System.out.println("JDBC Driver not found:"+
                e.getMessage()); }
catch( SQLExeception e){
    System.out.println("Database error:" + e.getMessage());
    }
finally{ try{ if(rs!=null) rs.close();
            if(stmt!=null) stmt.close();
            if(con!=null) con.close(); }
    catch(SQLException e){
        System.out.println("Error closing resource\n" +
                e.getMessage());
}}}
```

5

In a Java EE application, how does a servlet controller manage the flow between the model and the view? Provide a brief example that demonstrates forwarding data from a servlet to a JSP and rendering a response.

Example: Forwarding Data from Servlet to JSP

servlet (Controller)

```
import java.io.IOException;
import javax.servlet.*;
import javax.servlet.http.*;

public class StudentServlet extends HttpServlet {
  protected void doGet(HttpServletRequest request,
                       HttpServletResponse response)
    throws ServletException, IOException {
      String studentName = "Rahim";   → model data can be fetch from DB
      request.setAttribute("Name", studentName);
                                      → set data as request attribute
RequestDispatcher rd = request.getRequestDispatcher
                       ("student.jsp");
  rd.forward(request, response);      → forward request to JSP view
}}
```

JSP view - student.jsp

```
<html>
<body>
    <h2> Student Name : ${name} </h2>
</body>
</ html>
```

Key Points:

① Servlet (Controller) handle request & business logic

② Data is passed to JSP (view) using request.setAttribute();

③ JSP renders data using Expression Language ${}

④ MVC separate model, view, controller for better maintainability.

Conclusion:

A servlet controller in Java EE manages the flow by handling requests, interacting with the model and forwarding data to a JSP view for rendering the response.

Lab 6:

How does PreparedStatement improve performance and security over Statement in JDBC? write a short example to insert a record into MYSQL table using preparedStatement.

Prepared Statement over Statement:

Feature	Statement	PreparedStatement
Performance	Execute SQL every time	Precompile SQL → faster for repeated queries
Security	Vulnerable to SQL injection	Use placeholders (?) prevent SQL injection.
Parameter Handling	Concatenation required	Set parameters using setString(); setInt(), etc.

※ Insert record using prepare statement.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.PreparedStatement;
import java.sql.SQLException;
```

```
public class InsertPrepared {
    public static void main(String[] args){
        String url = "Jdbc:mysql://localhost:3306/Testdb";
        String user: "root";
        string password: "tanjilsql";
        String sql :"INSERT INTO students(name, age)
                    VALUES (?,?)";
try (Connection con = DrivenManagen.getConnection(url,
        user, password); PreparedStatement
        ps : con. prepareStatement (sql)) {
        ps. setString (1, "Rahim");
        ps. setInt (2, 22);
        ps. executeUpdate ();
System. out. println ("Record inserted succesfully!");
} catch (SQLException e){
        System.out.println("Database Error:"+ e.getMessage());
}}}
```

Lab 7

What is a ResultSet in JDBC and how is it uned to retrieve data from a MySQL database? Briefly explain the use of next(), getString() and getInt() methods with an example:

Resultset in JDBC:

* It is an object that holds data retrieved from database after executing a SELECT query in JDBC.
* It acts like a cursor pointing to one row at a time.

next() → move cursor to the next row; return false if no more rows.

getString() → Retrieves String values from the current row.

getInt() → Retrieves Int values from the current row.

```
impont java.sql.*;
  public class ResultSetDemo{
    public static void main(String[] args){
String url = "jdbc:mysql://localhost:3306/testdb";
String user : "root"; String password = "tanjilsql";
```

```
try(Connection con = DriverManager(url, user, password);
    Statement stmt = con.createStatement();
  ResultSet rs = stmt.executeQuery("SELECT id,
        name, age FROM student ")){
  while (rs.next()){
   int id = rs.getInt("id");
  String name = rs.getString("name");
  int age = rs.getInt("age");
 System.out.println(id + "\t"+name + "\t"+age);
}}
catch(SQLException e){
   System.out.println("Database error:"+e.
                      getMessage());
}{}
```

Lab 3 Multithreading - Based Car Parking Mangement System:

Class Description:

RegisterParking: Represent a parking request made by a car.

Parking_pool: Shared synchronized queue that stores parking request.

Parking-Agent: A threads that parks car from the pool

MainClass: Simulates multiple cars anniving concurrently.

Code:

RegisterParking.java

```
class RegisterParking{
    String carNumber;
    RegisterParking(String carNumber){
        this.carNumber = carNumber;
    }
```

```
clam parkingPool {
 private Queue<RegistenParking> queue = new LinkedList<>();
 public synchnonized void requestParking (RegistenParking
                                          car) {
     queue.add (car);
  System.out.println("Car " + car.carNumber + " requested
                          parking.");
 }
public synchnonized RegistenParking ParkCar () {
     while(queue.isEmpty()){
      try { wait ();
          }
      catch (InterruptedException e) { }
      }
      return queue.poll();
 } }
```

ParkingPool.java

```
import java.util.LinkedList;
import java.util.Queue;
clas ParkingPool{
```

ParkingAgent.java

```
class ParkingAgent extends Thread{
    private ParkingPool pool;
ParkingAgent (String name, ParkingPool Pool) {
        supper(name);
        this.pool = pool;
    }
public void run() {
    while (true) {
    RegistenParking Carz = pool.parkCar();
    }
  }
}
```

MainClass.java

```
public class Mainclass{
    public static void main (String[] args) {
        Parking pool = new ParkingPool();

    new ParkingAgent("Agent 1", pool).start();
    new ParkingAgent("Agent2", pool).start();
pool.requestParking(new. RegistenParking("ABC 123"));
pool.requestParking(new. RegistenParking("XYZ123"));
}}
```

Lab 8 How does Spring Boot simplify the development of RESTful service? Describe how to implement a REST controller using @RESTController, @GetMapping and @PostMapping including JSON data handling.

SpringBoot greatly simplifies the development of Restful services using:

① Auto configuration:
Automatically configure web servers and JSON converter with minimal setup.

② Embeded Server:
No need to deploy WARS to an external server. Just run your app like a java application.

③ Spring web starter:
Includes all required dependencies for building REST API's.

④ Reduce Boilerplate
Annotation like @RESTController, @GetMapping, @PostMapping simply request handling.

Implementation of REST controller using annotation and JSON data handling:

① @RESTController Annotation:

- Marks the class as a REST API Controller.
- It combines @Controller and @ResponseBody

② @GetMapping

- @GetMapping is used to map HTTP Get requests
- Typically used to retrieve data from server

③ @PostMapping

- @PostMapping maps HTTP post requests
- used to send data to server
- Conversion here is handled using @RequestBody

④ JSON data handling

- The JSON is sent by the client
- Spring convert this JSON into a Java object using Jackson
- This is done by making the method parameter with @RequestBody.