Lab 1 : Compare abstract class and interface in terms of multiple inheritance. When would you prefer to use an abstract class and when an interface ?

Abstract class VS Interface (Multiple inheritance)

1. Multiple Inheritance :-

Aspect	Abstract class	Interface
Multiple inheritance	Not supported (a class can extend only one abstract class)	supported (a class can implement multiple interfaces)
Reason	Avoids ambiguity in implementation	supports multiple behaviours without ambiguity.

2. Key Difference

Feature	Abstract class	Interface
Methods	can have abstract + concrete methods	method are abstract by default.
variables	can have instance variable	only puplic static final constants
Constructor	yes	No
Access modifier	Any (Private, Protected)	Methods are public by default

3. When to use abstract class

- You want to share common code / logic among related classes.
- You need instance variable or constructors.
- There is a clear "is a" relationship.
- only single inheritance is sufficient.

Example :-

```
abstract class Vehicle {
  int speed ;
  abstract void move() ;
  void start() {
    System.out.println ("vehicle started");
  }
}
```

4. When to use Interface

- You need multiple inheritance.
- You want to define a contract.
- Class are unrelated but share behaviour.
- You want to loose coupling and better flexibility.

```
interface Flyable {
  void fly();
}
Class Bird implements Flyable, serializeable {
 Public void fly() {
 System.out.println ("Birds in flying");
 }
}
```

Lab 2 :- How does encapsulation ensures data security and integrity ? Show with a Bank account class using Private variables and validated methods such as setAccount Number (String), setInitialBalance (double) that reject null, negative on empty values.

Encapsulation ensures data security and integrity by hiding data members using the Private access modifier and allowing access only through Public methods with validation. This Prevents unauthorized on invalid modification of data.

Example :- Bank account class

```
Public class BankAccount {
Private string accountNumber ;
Private double balance ;
Public Void setAccountNumber (string accountNumber){
if (accountNumber == = null || account number.trim()
    isEmpty () ) {
System.out.Println ("Invalid account Number " ) ;
}
```

```
this.accountNumber = accountNumber;
}
Public void setInitialBalance (double balance){
if (balance<0){
System.out.Println ("Balance cannot be negative");
return;
}
this.balance = balance;
}
Public String getAccountNumber (){
return accountNUmber;
}
Public double getBalance (){
return balance;
}
}
```

By restricting direct access to variables and enforcing validation through setter methods, encapsulation Protects data (security) and maintains correct and consistent values (integrity) in a Program.

Lab 3: Developing a multithreading based Project to simulate a car Parking management system with classes namely - RegistrarParking - represent a Parking Pool - Acts as a Shared Synchronized queue; Parking agent Represents a thread that continuosly checks the Pool and Parks cars from the queue and a Main Class - simulates N cars arriving Concurrently to request Parking.

Car ABC123 requested Parking.
Car XYZ456 requested Parking.
Agent 1 Parked car ABC123.
Agent 2 Parked car XYZ456.

RegistrarParking :-

```
Class RegistrarParking {
String CarNO;
RegistrarParking (String CarNO){
this.CarNO = CarNO;
system.out.Println (" Car" + CarNO + "requested
                                    Parking");
}
}
```

ParkingPool (Synchronized Queue):

```
import java.util.*;
class ParkingPool {
Queue<RegistrarParking> q = new LinkedList<>();
synchronized void addCar (RegistrarParking c){
q.add(c); notify;
}
synchronized RegistrarParking getCar() throws
Exception {
while (q.isEmpty()) wait();
return q.poll();
}
}
```

ParkingAgent (Thread)

```
class ParkingAgent extends Thread {
ParkingPool Pool;
ParkingAgent (ParkingPool p, String n){
super(n); Pool = p; }
Public void run(){
try {
while (true)
System.out.println(getName()+ " Parked Car"
+ Pool.getCar().carNo)
```

```
}catch (Exception e){ }
}
}
```

Mainclass:-

```
Public class Mainclass{
Public Static void main (String[] args){
ParkingPool P = new ParkingPool();
new ParkingAgent (P, "Agent1").start();
new ParkingAgent (P, "Agent2").start();
P.addCar (new RegistrarParking ("ABC123"));
P.addCar (new RegistrarParking ("XYZ456"));
}
}
```

Car request Parking concurrently, a synchronized queue stores requests, and multiple agen threads park car using the wait - notify mechanism.

Lab 4:- Describe how JDBC manages communication between java application and relational database. Outline the step involved in executing a select query and fetching result, include error handling with try-catch and finally blocks.

= JDBC (Java Database connectivity) is an API used to connect a java application with a relational database. It allows java Programs to send SQL queries to the database and receive results.

To execute SELECT query, JDBC loads the drivers, creates a connection, creates a statement, execute the queries and retrives data from the result set. Errors are handles using try catch finally blocks, where try execute database operations, catch handle exceptions and finally closes resources.

Example code (with try catch finally)

```
import java.sql.*;
class JdbcDemo{
Public static void main (String[] args){
Connection con =null;
Statement st =null;
Resultset rs = null;
try {
class.forName ("com.mysql.cj.jdbc.Driver");
con = DriverManager.getConnection ("jdbc:
mysql://localhost:3306/testdb", "root", "Password");
st = con.CreateStatement();
rs = st.executequery ("SELECT *from student);
while (rs.next()){
System.out.println (rs.getInt(1) +" " +rs.get
                                    String(2));}
catch (Exception e) {
e.PrintStackTrace();
} finally {
try {
    if (rs!=null) rs.close;
    if (st!=null) st.close;
    if (con!=null con.close;
}catch (SQLException e){
    e.PrintstackTrace(); } } } }
```

Lab 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.

Servlet controller in Java EE (Model - View - Controller Flow)

In a Java EE MVC architecture, a servlet acts as the controller.

1. Receives HTTP requests from the client.
2. Interacts with the Model (business Logic, database, JavaBeans).
3. Stores data in request / session scope
4. Forward the request to a JSP (view) for Presentation.

The servlet doesnot generate HTML directly it controls the flow.

Flow Diagram (Textual)

Client → Servlet (controller) → Model → servlet → JSP (view) → Client.

Basic Example

Model (Java Bean)

```
Public class Student {
  Private String name ;
 Public Student (String name) {
   this.name = name ;
 }
Public String getName () {
   return name ;
 }
}
```

2. Servlet controller .

```
import java.io IOException ;
import jakarta.servlet.* ;
import jakarta.servlet.http.* ;
Public class StudentServlet extends HttpServlet {
Protected void doGet (HttpServletRequest request,
 HttpServletResponse response)
throws ServletException , IOException {
 Student student = new Student ("Ranjan");
 request.setAttribute ("student", student);
RequestDispatcher rd = request.getResultDispatcher
rd.forward (request, response); } ("student.JSP");
```

3. JSP View (student.jsp)

```
<%@ Page language = "java" %>
<html>
<body>
    <h2> Student Information </h2>
    <p> Name : ${student.name} </p>
</body>
</html>
```

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.

1. How PreparedStatement improves Performance

• Precompiled SQL:

PreparedStatement is compiled once by database and reused, reducing execution time.

• Faster for repeated queries :-

Some SQL with different values executes efficiently

• Less parsing & optimization overhead compared to statement.

2. How Prepared statement improves security

• Prevents SQL injection:
User input is treated as data, not executable SQL.

• Self parameter binding using ? placeholders.

• Automatically handles special charecters (', -- etc).

3. Comparison table

Feature	Statement	PreparedStatement
SQL compilation	Everytime	Once (Precompiled)
SQL injection	Vulnerable	Safe
Parameter support	No	Yes
Performance	Slower	Faster
Readability	Lower	Higher

Q. Short Example : Insert Record Using Prepared Statement (MYSQL)

```
import java.sql.*;
Public class InsertStudent {
Public static void main (String[] args) {
try {
class.forName ("com.mysql.cj.jdbc.Driver");
Connection Con = DriverManager.getConnection(
"jdbc:mysql://localhost:3306/school",
"root",
"Password"
);

String sql = INSERT INTO Student (id,name,age)
.values (?, ?, ?)";

PreparedStatement Ps = Con.PreparedStatement (sql);
Ps.setInt (1, 101);
Ps.setString (2, "Farjana");
Ps.setInt (3, 21)

Ps.executeupdate ();
System.out.Println ("Record inserted successfully");
Ps Close ();
Con Close ();
```

```
} catch (Exception e) {
e.printStackTrace();
}
}
```

(7) What is Resultset in JDBC and how is it used to retrieve data from a MYSQL database? Briefly explain the use of @Entity, @Id, and @GeneratedValue, and other annotations. Discuss the advantages of using JPA over raw JDBC.

What is Resultset ?

A resultset in JDBC is an object that stores the data returned from a SQL SELECT query.

It represents a table of data where each row is fetched one at a time from the database.

- Returned by : executeQuery()
- Cursor initially points before the first row.
- used to read data row by row.

2. Important Resultset Methods

Method	Purpose
next()	moves cursor to the next row and returns true if data exists.
getString()	Retrives string type data from a column.
getInt()	Retrives int type data from a column.

3. Brief Explanation of Methods

- next()

moves the cursor forward by one row. must be called before reading data.

- getString (columnName / index)

used for VARCHAR, CHAR, TEXT columns.

- getInt (columnName / index)

Used for INT columns.

4. Example: Retrieve Data from MySQL using ResultSet

```
import java.util.*;
Public class FetchStudent{
Public Static void main (String[] args){
try{
 Class.forName("com.mysql.cj.Jdbc.Driver");
Connection con = DriverManager.getConnection(
"Jdbc:mysql://localhost:3306/school",
"root",
"Password".
);
String sql = "SELECT id, name, age FROM student";
String stmp = con.createStatement();
Result rs = stmp.executequery(sql);
while (rs.next()){
int id = rs.getInt("id");
String name = rs.getstring("name");
int age = rs.getInt("age");
System.out.Println(id + " " +name+ " " +age);
}
rs.close(); Stmp.close(); con.close();
} Catch{Exception e){
e.printstackTrace();
}
}
```