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

Introduction : Multiple inheritance means a class can inherit features from more than one parent. Languages like Java do not support multiple inheritance using class directly to avoid ambiguity problems. Instead, Java uses abstract ~~class~~ classes and interfaces to achieve similar behavior. This report compares abstract classes and interfaces with respect to multiple inheritance.

Comparison:

Feature	Abstract Class	Interface
Multiple inheritance	not supported	supported
Method implementation	can have	mostly abstract
Variables	can have instance variables	only constants
Constructors	Allowed	not allowed
Access modifiers	Any	methods are public by default

When to use an Abstract Class:

i. Classes are closely related.
ii. Want to share code (method implementation)
iii. Need constructors or instance variables
iv. Want to provide base functionality

Example - Vehicle → Car, Bike

When to use an Interface

i. Need multiple inheritance
ii. Classes are not closely related
iii. Want to define a common behavior
iv. Want to support loose coupling

Example - Runnable, Serializable comparable.

Conclusion: Abstract classes and interfaces both help achieve abstraction is Java. However, multiple inheritance is only possible through interfaces.

Lab on : How does encapsulation ensure data security and integrity ? Show with a BankAccount class using private variables and validated methods such as setAccountNumber(String), setInitialBalance(double) that rejects null, negative or empty values.

Answer:

Encapsulation is an oop concept that wraps data and methods into a sigle unit and restricts direct access to the data. It ensures data security and data integrity by allowing controlled access through validate methods.

How Encapsulation esures Data Security and Integrity

i. Data hiding : Variables are declared private.

ii. Controlled Access : public methods control how data is read or updated

iii. Validation of Data : Setter methods check for invalid values and prevent inconsistent object states.

iv. Maintains integrity : Object data always remains valid. Business rules are enforced inside the class.

Example :

BankAccount.class

public class BankAccount {

```
private string accountNumber;
private double balance;

public setAccount Number(String accountNumber){
    if (account Number == null || accountNumber.
trim().isEmpty()){
System.out.println("Invalid!");}
else{
    this.accountNumber = accountNumber;}
    }

public void setInitial Balance (double balance){
    if (balance<0)
        System.out.println("Initial balance can not
    be negative!");
    else
        this.balance = balance;
    }
```

```
public string get_Account Number() {
        return accountNumber; }
public $ double get Balance() {
        return balance; }
}
```

accountNumber and balance are private, so they can not be accessed directly. Only valid data is stored.

Lab on: Developing a multithreading based project to simulate a car parking management system with classes namely RegistrarParking- Represents a parking request made by a car ; ParkingPool- Acts as a shared synchronized queue ; ParkingAgent Represents a thread that continuously checks the pool and parks cars from the queue and a MainClass- Simulates N cars arriving concurrently to request paring.

1. RegistrarParking

```
public class RegistrarParking {
    private String carNumber;
    public RegistrarParking (String carNumber) {
        this.carNumber = carNumber; }
    public String getCarNumber() {
```

```
    return carNumber; }
}
```

2. ParkingPool

```
import java.util.LinkedList;
import java.util.Queue;

public class ParkingPool {

    private Queue<RegistrarParking> queue =
        new LinkedList<>();

    public synchronized void requestParking
        (RegistrarParking car) {
        queue.add(car);
        System.out.println("Car " + car.getCarNumber()
        + " requested parking.");
        notifyAll(); }
```

```
public synchronized RegistrarParking parkCar(){
    while (queue.isEmpty()) {
        try {
            wait(); }
        Catch (InterruptedException e) {
            e.printStackTrace();
        }
    return queue.poll(); }
}
```

3. ParkingAgent

```
public class ParkingAgent extends Thread {
    private ParkingPool pool;
    private int agentId;
```

4. Main Class

```
public class MainClass {

    ParkingPool pool = new ParkingPool();
    ParkingAgent agent1 = new ParkingAgent(pool, 1);
    ParkingAgent agent2 = new ParkingAgent(pool, 2);
    agent1.start();
    agent2.start();

    for (String carNumber : cars){

        new Thread(() -> {

        RegistranParking car = new RegistranParking
        (carNumber)};

        pool.requestParking(car);
         }).start();

        }
    }
}
```

Lab on: Describe how JDBC manages communication between a Java application and a relational database. Outline the steps involved in executing a select query and fetching results. Include error handling with try-catch and finally blocks.

Introduction: JDBC (Java Database Connectivity) is an API that allows a Java application to communicate with a relational database.

How JDBC manages Communication

1. JDBC Driver: Converts Java JDBC calls into database specific commands.

2. Connection : Establishes a session between Java application and database.

3. Statement : Sends SQL queries to the database.

4. Result set : Recieves and stores the data returned by select queries.

5. Exception Handling : Handles SQL and run time errors safely.

Steps :

1. Load the JDBC Driver

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

2. Establish Connection

```
Connection con = DriverManager.getConnection(
```

"jdbc:mysql://localhost:3306/mydb",
"root", "password");

3. Create Statement

Statement stmt = con.createStatement();

4. Execute 'Select' Query

ResultSet rs = stmt.executeQuery("select id, name from student");

5. Fetch Results

6. Close Resources

JDBC Example with Error Handling:

```
import java.sql.*;
public class JDBCSelect{
    public static void main(String[] args){
```

```
try {
    // step 1
    Class.forName("com.mysql.cj.jdbc.Driver");

    // step 2

    // step 3

    // step 4

    while (rs.next()){
        System.out.println(rs.getString("name"));}
    }
catch (ClassNotFoundException e){

    }
catch (SQLException e){

    }
finally {
    try {
        if (rs != null) rs.close();
        if (stmt != null) stmt.close();
```

```
        if (con != null) con.close();
        }
      catch (SQL Exception e) {
        System.out.println ("Error closing
        resources");
        }
      }
    }
  }
```

Conclusion: JDBC provides a standard way for Java applications to communicate with relational database.

Lab on : 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 servlet to a JSP and rendering a response.

How a Servlet Controller Manages flow

1. Client sends request to the servelet.
2. Servlet processes the request.
3. It interacts with the Model.
4. Stores data in request or session scope
5. Forwards the request to a JSP
6. JSP renders the final response to the client.

Example :

```
import java.io.IOException;
import javax.servlet.*;
import javax.servlet.http.*;
public class StudentServlet extends HttpServlet {
    protected void doGet (HttpServletRequest
  request, HttpServletRespond response) throws
      ServletException, IOException {
    String studentName = "Rahim";
    int marks = 85;
     request.setAttribute ("name", studentName);
     request.setAttribute ("mariks", mariks);
    RequestDispatcher rd = request.getRequest
  Dispatcher ("result.jsp");
     rd.forwand (request, response);
    }
}
```

Lab on: How does Prepared Statement improve performance and security over Statement in JDBC? Write a short example to insert a record into a MySQL table using PreparedStatement.

Improvement of Performance:

1. Precompiled SQL: SQL query is compiled only once by the database.

2. Faster Execution: Reduces parsing and compilation overhead. Improves performance for repeated queries.

Improvement of security:

1. Prevents SQL injection: Uses parameterized queries. User input is treated as data.

2. Automatic Data Handling : Safely handles special characters. No need for manual string concatenation.

Example:

```
import java.sql.*;

public class InsertExample {

    public static void main(String[] args) {

    String sql = "insert int student (id, name)
     values (?,?)";

    try {
        Class.forName("com.mysql.cj.jdbc.Driver");

        ps.setINT(1,101);

        ps.setString(2,"Rahim");

        ps.setDouble(3,88.5
```

```
            ps.executeUpdate();
            ps.close();
            con.close();
        }
        catch (Exception e) {
            System.out.println("Error: " + e);
        }
    }
}
```

Conclusion: PreparedStatement improves performance by reusing precompiled SQL and improves security by preventing SQL injection.

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

ResultSet: It is JDBC object that stores data returned from a 'select' query executed on database. It represents a table of rows and columns and allows the Java program to move through each row and read column values one by one.

ResultSet in Retrieve Data:

1. Execute a 'select' query using statement.
2. Store the returned data in a ResultSet

object.

3. Use cursor movement and getter methods to read values.

Use of next():

i. moves the cursor to the next row
ii. Returns true if row exists otherwise false
iii. Must be called before reading data.

Use of getString():

i. Retrieves string-type column data
ii. Can use column name or column index

getInt():

i. Retrieves integer type column data
ii. Can use column name or index

Example:

```
import java.sql.*;

public class ResultSet {

    public static void main (String[] args) {

      try {
            //load driver
            // create statement
            // execute SELECT

          while (rs.next()) {
              int id = rs.getInt("id");
              String name = rs.getString("name");

    System.out.println (id + "  " + name);

   rs.close();
   stmt.close();
   con.close(); }

   catch (Exception e) {

      }

   }

}
```

Lab on: Design a simple CRUD application using spring Boot and Mysql to manage student records. Describe how each operation would be implemented.

1. student Entity

```
import jakarta.persistence.*;
@Entity
public class student{
    private long id;
    private String name;
    private int marks;
```

2. StudentRepository Interface

```
import. org.springframework.ddta-jpa.
repository.JPaRepository;
```

```
public interface StudentRepository extends
    JpaRepository<Student, Long> {

}
```

CRUD operations:

1. Create

```
Student student = new Student();
  student.setName("R");
student.setmarks(85);
studentRepository.save(student);
```

2. Read

```
List<Student> students = studentRepository.findAll();
```

Lab on: How does spring Boot simplify the development of RESTful services? Describe how to implement a REST controller using @RESTController, @GetMapping and @PostMapping, including JSON data handling

Simplification of REST:

1. Auto Configuration: Automatically configures beans based on dependencies.

2. Embedded Server: No need to deploy WAR files. Application runs as a standalone JAR.

3. REST Annotations: Simple annotations like @RestController, @getMapping, @Post Mapping.

4. JSON Handling : Automatically converts Java objects ⟷ JSON using Jackson.

5. Minimal Configuration : Fewer XML and configuration files.

Implementation of REST Controller

1. @RestController : Combines @Controller + @ResponseBody. Returns data directly in JSON format.

2. Student Class

```
public class student{
        private int id;
        private string name;}
```

3. Rest Controller

```
import org.springframework.web.bind.
  annotation.*;
import java.util.*;
public class StudentController{

    private List<Student> students = new
        ArrayList<>();
public List<Student> getStudents(){
      return students;}
public student addStudent(Student student){
        students.add(student);
        return student;
    }
}
```

Lab Name: Demonstrate the project you developed with the important codes and GUI.

1- Student Entity

```
public class student{
    private Long id;
    private String name;
    private int marks; }
```

2. Student Repository

```
import org.springframework.data.jpa.repository.
JpaRepository;
public interface StudentRepository extends
    JpaRepository<student, Long>{
}
```

3. Student Controller

```
public class StudentController{
    private final StudentRepository repo;
public StudentController(StudentRepository
  repo) {
      this.repo = repo; }
public Student addStudent(Student student){
    return repo.save(Student); }
public List<Student> getAllStudents() {
    return repo.findAll();
  }
public Student updateStudent(Long id, Student s)
{  s.setId(id);
   return repo.save(s); }
```

```
public void deleteStudent(Long id) {
        repo.deleteById(id);
        }
    }
```

4. GUI Code

```
<!DOCTYPE html>
<html>
<head>
    <title>Student Management</title>
</head>
<body>
<h2>Student Management System</h2>
<form method="post" action="/students">
Name:<input type="text" name="name">
<br><br>
```

```
Marks: <input type="number" name=
"~~name~~ marks" >
<br> <br>
<button type="submit" > Add Student
</button>
</form>
</body>
</html>
```