Lab Report - 01 :

Introduction : Java is an object-oriented programming language that supports abstraction using abstract classes and interfaces. since java doesn't allow multiple inheritance using classes, abstract classes and interfaces are used to achieve abstraction. This lab focuses on comparing abstract classes and interfaces, especially in terms of multiple inheritance.

Objective :

- To understand abstract classes and interfaces.
- To compare them in terms of multiple inheritance
- To learn when to use an abstract class and when to use an interface.

Theory:

Abstract class: An abstract class is a class that can't be instantiated and may contain both abstract and concrete methods. A class can extend only one abstract class, so multiple inheritance is not supported.

Example (Abstract class):

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

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

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

Here Dog can extend only one abstract class (Animal).

Interface: An interface contains only abstract methods and constants. A class can implement multiple interfaces, which supports multiple inheritance.

Example (Interface - Multiple Inheritance):

```
interface Flyable {
    void fly();
}
interface Swimmable {
    void swim();
}
class Bird implements Flyable, Swimmable {
    public void fly() {
        System.out.println("Bird can fly");
    }
    public void swim() {
        System.out.println("Bird can swim");
    }
}
```

Here, Bird implements two interfaces, achieving multiple inheritance

Comparison Table:

Feature	Abstract class	Interface
Multiple inheritance	Not supported	Supported
Method implementation	Allowed	not allowed
Variables	Any type	Public, static, final
constructors	Allowed	Not allowed
Access Modifier	Any	Public Only

When to use Abstract class:

- When classes are closely related
- When code reuse is needed
- When partial implementation is required

When to use Interface:

- When multiple inheritance is required
- When classes are unrelated
- when complete abstraction is needed.

conclusion: Abstract classes are suitable for sharing common code among related classes but don't support multiple inheritance. Interfaces support multiple inheritance.

Lab Report - 02

Introduction: Encapsulation is one of the core principles of OOP. In 'java' encapsulation is achieved by making data members private and providing public methods to access or modify them. This helps protect data from unauthorized access and ensures that only valid data is stored in an object.

Object:

- To understand encapsulation
- To learn how encapsulation ensures data security and integrity
- to implement encapsulation using a BankAccount class.

Theory: Encapsulation ensures data security and integrity by:

- Hiding data using private variables
- Restricting direct class access from outside
- validating data before updating values.

Java Implementation :

```
class BankAccount {
    // private variables (data hiding)
    private string accountNumber;
    private double balance;

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

    // set initial balance with validation
    public void setInitialBalance(double amount) {
        if (amount < 0) {
            System.out.println("Balance can't be
                                negative");
        }
        else {
            balance = amount;
        }
```

```
// Display account details
    public void displayAccount () {
        System.out.println ("Account Number: " +
                                    accountNumber);
        System.out.println ("Balance: " + balance);
    }
}
```

How Encapsulation Ensures Security and Integrity:

- Direct access to accountNumber and balance is not allowed
- Invalid inputs like null, empty strings or negative values are rejected.
- Only validated data is stored in the object

Conclusion: Encapsulation ensures data security by hiding sensitive information and prevents data corruption by validating inputs before modification. Using private variables and controlled setter methods, a BankAccount class maintains safe and reliable data handling.

Lab Report - 03

Introduction: Multithreading in java allows multiple tasks to run concurrently, improving efficiency and responsiveness. In real-life systems like car parking management, multiple cars may arrive at the same time and multiple agents may handle parking simultaneously. this lab demonstrates how multithreading and synchronization can be used to simulate such a system safely.

Objective:

- To understand multithreading in Java
- To implement thread synchronization using a shared resource.
- To simulate a car parking management system.

System Description :

The system consists of 4 classes :

① RegistarParking – represents a parking request made by a car.

② ParkingPool – acts as a shared synchronized queue.

③ ParkingAgent – A thread that parks cars from the queue.

④ MainClass – Simulates multiple cars requesting parking concurrently.

Java Implementation

① RegistrarParking (car Request)

```
class RegistrarParking {
    String carNumber;

    RegistrarParking (String carNumber) {
        this.carNumber = carNumber;
        System.out.println ("car " + carNumber +
                            " requested parking.");
    }
}
```

② ParkingPool (shared Queue):

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

class ParkingPool {
    Queue<RegistreanParking> queue = new LinkedList
                                        <>();

    // Add car to queue
    synchronized void addcar(RegistreanParking
                                        car) {
        queue.add(car);
        notify(); //notify agents
    }
    //remove cair from queue
    synchronized RegistreanParking getcar()
        throws InterruptedException {
    while (queue.isEmpty()) {
        wait();
    }
    return queue.poll();
}
}
```

③ ParkingAgent (Thread)

```
class ParkingAgent extends Thread {

    ParkingPool pool;
    String agentName;

    ParkingAgent (ParkingPool pool, String name) {
        this.pool = pool;
        this.agentName = name;
    }

    public void run() {
        try {
            while (true) {
                RegisteredParking car = pool.getCar();
                System.out.println(agentName +
                        " parked car " + car.carNumber
                                + " - ");
                Thread.sleep(500); // simulated
                                   parking time
            }
        } catch (InterruptedException e) {
            System.out.println(agentName +
                " stopped.");
        }
    }
}
```

(4) Main class (simulation)

```
public class Mainclass {
    public static void main (String[] args) {
        ParkingPool pool = new ParkingPool ();
        parkingAgent agent1 = new ParkingAgent
                              (pool, "Agent 1");
        ParkingAgent agent2 = new ParkingAgent
                              (pool, "Agent 2");
        agent1. start ();
        agent2. start ();
        //simulating cars arriving concurrently
        String[] cars = {"ABC123", "XYZ456",
                         "LMN789", "DEF321"};
        for (String carNo : cars) {
            pool. addCar (new RegistrarParking
                          (carNo));
        }
    }
}
```

Sample Output :

Car ABC123 requested parking.

Car XYZ456 requested parking.

Agent 1 parked car ABC123.

Agent 2 parked car XYZ456.

Explanation :

- Multiple cars request parking simultaneously
- ParkingPool ensures thread-safe access using synchronized
- wait() and notify() coordinate between cars and agents
- Mutiple agents park cars concurrently without conflict

Conclusion :

this lab demonstrates how multithreading and synchronization can be used to build a real-world simulation. the car parking management system efficiently handles concurrent requests while ensuring data consistency using a shared synchronized queue.

Lab Report - 04:

Introduction: JDBC (Java Database connectivity) is an API that enables Java applications to communicate with relational database such as MySQL, Oracle, or PostgreSQL. It provides a standard interface to connect, execute SQL queries, retrieve results and handle database errors efficiently.

Objective:

- To understand how JDBC manages communication with a database
- To learn steps involved in executing a SELECT query
- To implement error handling using try-catch-finally.

How JDBC Manages communication: JDBC acts as a bridge between a java application and a database. The JDBC driver translates Java method calls into database-specific SQL commands and returns the results back to the java application

Steps to Execute a SELECT Query using JDBC:

1. Load JDBC driver
2. Establish database connection
3. Create SQL statement
4. Execute SELECT query
5. Process the ResultSet
6. close resources

Java code Example (SELECT Query with Error Handling)

```
import java.sql.*;

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

        Connection = null;

        try {
            // load driver
            class.forName("com.mysql.cj.jdbc.Driver");

            // create connection
            con = DriverManager.getConnection(
              "jdbc:mysql://localhost:3306/mydb",
              "root",
              password");
```

```
//create statement
Statement st = con.createStatement();
// Execute SELECT query
ResultSet rs = st.executeQuery("SELECT *
        FROM students");
// fetch data
while (rs.next()){
    system.out.println(rs.getInt("id")+" "+
        rs.getString("name"));
}
rs.close();
st.close();
}
catch (Exception e){
    system.out.println("Error occured");
}
finally{
    try{
        if (con!=null)
            con.close();
    }
    catch (Exception e){
        system.out.println("connection close
                error");
    }
}
}
```

Explanation:

- try block handles database operations
- catch block handles SQL and runtime errors
- finally block ensures resources are closed to prevent memory leaks.

Conclusion:

JDBC provides a reliable way for java applications to interact with relational databases. By following a structured process and using proper exception handling, applications can safely execute SQL queries and retrieve results efficiently.

Lab Report - 05 :

Introduction : In Java EE application, the Model - View - Controller (MVC) architecture is commonly used to separate business logic, user interface, and control-flow. The Servlet acts as the controller, managing communication between the model and the view (JSP). This approach improves code organization, maintainability and scalability.

Objective :

- To understand the role of a servlet controller in MVC
- To learn how a servlet forwards data to a JSP
- To demonstrate rendering a response using JSP

How a Servlet Controller Manages Flow

1. Receives a request from the client.
2. Interacts with the model
3. Stores data using request attributes
4. Forwards the request to a JSP
5. JSP displays the data as the response

Example Implementation

Servlet controller(controller)

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

public class welcomeServlet extends HttpServlet {
    protected void doGet (HttpServletRequest
            request, HttpServletResponse response)
    throws ServletException, IOException {

        // Model data
        String username = "Student";
        //send data to view
        request.setAttribute ("name', username);
        //forward to JSP
        RequestDispatcher rd = request.getRequestDis-
                patcher("welcome.jsp");
        rd.forward (request, response);
    }
}
```

JSP page (view) - welcome.jsp

```
<html>
<head>
    <title>welcome page</title>
</head>
<body>
    <h2>welcome, ${name}! </h2>
</body>
</html>
```

Explanation:

- The servlet acts as the controller
- Data is passed using setAttribute()
- RequestDispatcher forwards the request to JSP
- JSP uses Expression Language (${name}) to display data.

Conclusion: In a Java EE application, the servlet controller manages the flow by handling requests, interacting with the model, and forwarding data to the JSP view. This seperation of responsibilities follows the MVC pattern and results in cleaner, more maintainable

Lab Repond - 06 :

Introduction : In JDBC, SQL queries can be executed using Statement or PreparedStatement. While Statement executes SQL queries directly, PreparedStatement provides better performance, security and maintainability. This lab explains how PreparedStatement improves JDBC operations and demonstrates inserting a record into a MySQL table.

Objective :

- To understand the limitations of Statement
- To learn how PreparedStatement improves performance and security
- To insert data into a MySQL table using PreparedStatement

Why PreparedStatement is better than Statement

Performance Improvement

- SQL query is compiled only once
- can be executed multiple times with different val
- Reduces database parsing and execution time

Security Improvement :

- Prevents SQL Injection attacks
- user input is treated as data, not executable SQL
- safer than string-based queries

Java Code Example (Insert Using Prepared statement)

```
import java.sql.*;

public class InsertExample {
    public static void main (String[] args) {
        try {
            // load driver
            class.forName ("com.mysql.cj.jdbc.
                                Driver");
            // create connection
            connection con = DriverManager.getConn-
                ection ("jdbc:mysql://localhost:3306/
                    mydb", "root", password");

            // prepare SQL query
            String sql = "INSERT INTO students (id, name)
                values (?, ?)";
            Prepared statement ps = con.preparedstatement
                                    (sql);
```

```
        //set values
        ps.setInt(1, 101);
        ps.setString(2, "Ramey");

        // Execute query
        ps.executeUpdate();
        System.out.println("Record inserted successfully");

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

Explanation:

- ? placeholders prevent SQL injection
- Query is precompiled for better performance
- setInt() and setString() safely insert values

Conclusion: PreparedStatement improves JDBC performance by reusing compiled SQL queries and enhances security by preventing SQL injection attacks. It is the preferred choice for executing parameterized queries in real-world Java applications.

Lab Report - 07 :

Introduction : In JDBC, a ResultSet is an object that holds the data returned from a SQL SELECT query. It allows a Java Program to move through database records row by row and retrieve column values in different data types.

Objective :

- To understand what a ResultSet is
- To learn how data is retrieved from a MySQL database
- To explain the use of next(), getString(), and getInt() methods

What is ResultSet ?

A ResultSet represents a table of data generated by executing a SELECT statement. Initially the cursor is positioned before the first row and the next() method is used to move to the next row.

Important ResultSet Methods :

- next() → moves the cursor to the next row
- getString() → retrieves a column value as a string
- getInt() → Retrieves a column value as an int

Java Code Example:

```
import java.sql.*;
public class ResultsetExample{
    public static void main (String[] args){
    try{
        // load driver
        class.forName("com.mysql.cj.jdbc.Driver");
        //create connection
        connection con = DriverManager.getConnection
        ("jdbc:mysql://localhost:3306/mydb",
        "root", "password");
        // create statement
        statement st = con.createStatement();
```

```
// Execute SELECT query
ResultSet rs = st.executeQuery("SELECT id,
    name from students");

// Retrieve data
while (rs.next()){
    int id = rs.getInt("id");
    String name = rs.getString("name");
    System.out.println(id+" "+name);
    }
rs.close();
st.close();
con.close();
}
catch (Exception e){
    system.out.println("Error occured");
    }
}
}
```

Explanation:

- rs.next() moves to each row one by one
- getInt("id") reads integer data
- getString("name") reads text data

Lab Report - 08 :

Introduction: CRUD operations (create, Read, Update, Delete) are the basic functions of any database-driven application. Spring Boot, combined with Spring Data JPA, makes it easy to build CRUD applications by reducing boilerplate code. This lab demonstrates a simple student Management system using Spring Boot and MySQL, where CRUD operations are handled through a repository interface.

Objective:

- To design a simple CRUD application using Spring Boot.
- To understand CRUD operations using Spring Data JPA
- To manage student records stored in a MySQL database

System Overview:

The application consists of:

- Entity class → Represents student table
- Repository interface → Handles CRUD operations
- Controller class → Exposes CRUD functionality

Student Entity (Model)

```
import jakarta.persistence.*;

@Entity
public class student{

    @ID
    @GeneratedValue(Strategy=GenerationType.IDENTITY)

      private int id;
      private String name;
      private int age;

      // getters and setters
}
```

Student Repository Interface

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

}
```

* JpaRepository automatically provides CRUD methods.

CRUD Operations Using Repository

1. Create (Insert Student)

```
studentRepository.save(student);
```

2. Read (Get Student)

```
studentRepository.findAll();
studentRepository.findById(id);
```

3. Update (Update Student)

```
studentRepository.save(updateStudent);
```

4. Delete (Remove Student)

```
studentRepository.deleteById(id);
```

Controller Example;

```
import org.springframework.web.bind.annotation.*;
import java.util.List;

@RestController
@RequestMapping("/student")
public class StudentController {
    private final StudentRepository repo;
```

```
    public StudentController(StudentRepository repo){
        this.repo = repo;
    }

    @GetMapping
    public List<Student> getAll(){
        return repo.findAll();
    }

    @PostMapping
    public Student add(@RequestBody Student s){
        return repo.save(s);
    }

    @DeleteMapping("/{id}")
    public void delete(@PathVariable int id){
        repo.deleteById(id);
    }
}
```

Explanation:

- Spring Boot handles configuration automatically
- Repository interface eliminates manual SQL queries
- CRUD operations are performed using built-in JPA methods.
- MySQL stores student records permanently.

Example : Student REST Controller

```
import org.springframework.web.bind.annotation.*;
import java.util.ArrayList;
import java.util.List;

@RestController
@RequestMapping("/students")
public class StudentController {

    private List<Student> students = new ArrayList<>();

    //Get all students
    @GetMapping
    public List<Student> getAllStudents() {
        return students;
    }

    //post a new student
    @PostMapping
    public Student addStudent(@RequestBody Student s) {
        students.add(s);
        return s;
    }
}
```

Student class (Model)

```
public class Student {
    private int id;
    private String name;

    // constructors, getters, setters

    public Student() {}
    public Student(int id, String name) {this.id =
        id; this.name = name;}
    public int getId() { return id; }
    public void setId(int id) {this.id = id;}
    public String getName() {return name;}
    public void setName(String name)
        {this.name = name;}
}
```

How it works:

1. GET /students → Returns JSON list of students
2. POST /students → Accepts JSON data, converts to student object, adds to list
3. Spring Boot handles JSON serialization/deserialization automatically.

Sample JSON for POST :

```
{ "id" : 101,
  "name" : "Lamega"
}
```

Conclusion : Spring Boot simplifies RESTful service development by providing annotations that handle routing, request/response mapping and JSON serialization automatically

Lab Report - 10 :

Introduction : This lab demonstrates the project developed in previous exercises. the project is a Student Management System that allows users to Add, view, Update and Delete student records. the application uses Java Swing for GUI and Mysql for database storage.

Objective :

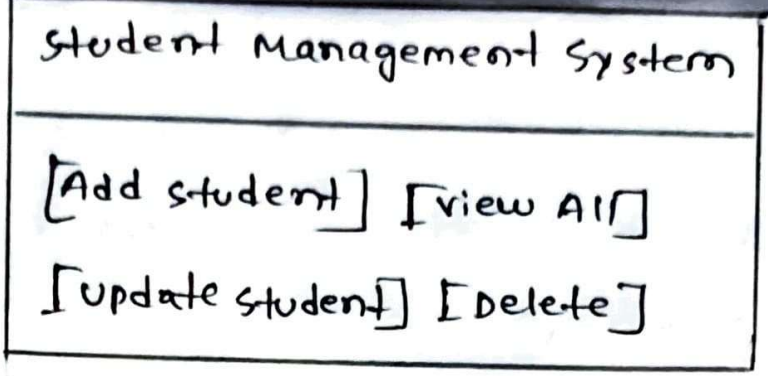
- To demonstrate the working project
- To show important codes for core functionalities
- To present the Graphical User Interface

Project features :

1. Add a new student
2. View all students
3. Update student details
4. Delete student records

Graphical User Interface (GUI)

- Main Window: Displays buttons for Add, View, Update, Delete
- Add Student form: ID, Name, Age, Submit button
- View Students: JTable displays all student records
- Update / Delete: Search by ID and modify or remove



1. Add Student Button Action

```
addButton.addActionListener(e -> {
try {
    String name = nameField.getText();
    int age = Integer.parseInt(ageField.getText());

String sql = "Insert into students(name, age)
  values (?, ?)";
  PreparedStatement ps = con.preparedStatement
                                    (sql);
  ps.setString(1, name);
  ps.setInt(2, age);
  ps.executeUpdate();

  JOptionPane.showMessageDialog(null, "Student
    added successfully!");
  }
  catch (Exception ex) {
  JOptionPane.showMessageDialog(null, "Error!" +
      ex.getMessage());
});
```

2. View All students

```
String sql = "SELECT * from students";
Statement st = con.createStatement();
ResultSet rs = st.executeQuery(sql);

while(rs.next()){
  tableModel.addRow(new Object[]{
      rs.getInt("id"),
      rs.getString("name"),
       rs.getInt("age")});
}
```

Explanations:

- GUI components handle user interaction
- Database operations use Prepared Statement and ResultSet
- User can perform CRUD operations using simple forms.

Conclusion: The project successfully demonstrates a GUI-based student management system. Users can interact with the application to manage student data and the database ensures persistent storage.