Lab Report No : 01

Lab Title : Comparison of Abstract classes and Interfaces in Java (multiple inheritance)

Introduction : This lab explores the differences between abstract classes and interfaces in Java, especially regarding multiple inheritance. It focus on how Java allow multiple inheritance through interfaces and when abstract classes are more suitable for sharing common behavior.

Theory : In Java, multiple inheritance means a class can get features from more than one parent. An abstract class does not support multiple inheritance. A class can extend only one abstract class. A interface supports multiple inheritance in Java.

A class can implimint more than one interface at the same time. Abstract classes can have both abstract and normal methods. Interfaces contain only abstract methods. We use an abstract class when classes are closely related. we use an interface when multiple inheritance is needed.

Java code example:

```
interface A {
        void show();
}

interface B {
        void print();
}

class Test implements A, B {
        public void show() {
                system.out.println("Interface A");
        }
```

```
public void print() {
    system.out.println("Interface B");
}
```

Conclusion: In conclusion, interfaces are preferred when multiple inheritance of type is needed and when defining a common contract across unrelated classes. Abstract classes are better when sharing common fields, constructors or partially implemented methods within closely related classes.

Lab Report No : 02

Lab Title : Implementing Encapsulation and Data validation in a Bank account class.

Introduction : This lab demonstrates encapsulation in java to ensure data security and integrity. A BankAccount class is designed using private variables and validated setter methods to prevent invalid inputs such as null, empty account numbers, and negative balances.

Theory : Encapsulation is an OOP concept that hides data and product theme from direct access. It ensure data seqvrity by making variable private inside a class.

Outside classes cannot change data directly, data can be changed only using public methods. These methods check the values

before saving them. This prevents wrong or harmful data from entering the system. Encapsulation also keeps data consistant and correct.

Java code :

```
class Account {
    private string accountNumber;
    private double balance;
public void setAccountNumber (String acc no) {
    if (accNo != null && !accNo.isempty ()) {
        accountNumber = accNo;
    }
    else {
        system.out.println ("Invalid account number");
    }
}
public void setInitialBalance (double amount) {
    if (amount >= 0) {
        balance = amount; }
    else {
        system.out.println ("Balance cannot be negative");
    }
}
```

Conclusion: In conclusion, encapsulation protects data by restricting direct class access and enforcing rules through validated methods. Using private fields and input validation helps ensure the BankAccount always remains in a consistent and secure state.

Lab Report No : 4

Lab Title : Database connectivity and select Query Execution using JDBC.

Introduction : The Lab focuses on how JDBC connects a java application with relational databases.

Theory : JDBC acts as bridge between Java and the database using :

(i) DriverManager - loads the JDBC driver
(ii) Connection - establish the session
(iii) Statement - sends SQL commands
(iv) 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 connection

(3) Prepare SQL query

(4) set parameters

(5) Execute the query

(6) Process the ResultSet

(7) Handle Exception

(8) Closure resource in finally block.

Java code :

```
import java.sql.*;
class selectExample {
    public static void main (String[] args) {
        connection con = null;
        try {
            con = DriverManager.getConnection(
            "Jdbc:mysql://localhost:3306/testdb",
            "root", "password");
        Statement st = con.createStatement();
    ResultSet rs = st.executeQuery("SELECT *
        from student");
```

```
while (rs.next()) {
    System.out.println(rs.getInt(1) + "  " +
    + rs.getString(2));
    }
}
catch (Exception e) {
    System.out.println("Error occured");
} finally {
    try {
        if (con != null)
            con.close();
        }
    catch (Exception e) {
        System.out.println("connection not closed");
}
}
```

Conclusion : In conclusion, JDBC provides a structured way to communicate with databases using connections, statements and result sets.

Lab Report No : 05

Lab Title : Servlet controller and MVC flow in Java

Introduction : this lab introduces the role of a servlet controller in Java EE applications using the MVC pattern.

Theory : In Java EE application, a servlet controller manage the flow between the model and the view by :

1. Receiving request from the client
2. Calling model classes to process data or feteh from the database.
3. setting attributes on the request scope.

Flow overview :

client → Servlet → model → JSP (view)

client ← (controller)

Example : student info

1. Model class :

```
public class student {
        privet string name ;
        privet int age ; }
    public student (String name , int age){
        this. name = name ;
        this. age = age ; }
    public string get Name () ;
    { return name;
        }
    public int getAge () {
            return age ;
        }
```

2. JSP (view) :

```
<%. @ Page import " your. package. student " %>
<% student student = (student) request.getAttribute ( " Student Data ");
%. >
<html>
<head> <title > student Info </title></head>
<body>
   <h2> student. details </h2>
</body>
</html>
```

3. Controller (Servlet):

```
@WebServlet("/student")
public class StudentServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        Student student = new Student("Ram", 22);
        request.setAttribute("studentData", student);
        RequestDispatcher dispatcher = request.getRequestDispatcher("student.jsp");
        dispatcher.forward(request, response);
    }
}
```

Conclusion: In conclusion, the servlet controller manages application flow by separating business logic from presentation.

Lab Report No : 06

Lab Title : Secure and Efficient Database Insert using Prepared Statement in JDBC

Introduction : This lab compares PreparedStatement and Statement in JDBC, focusing on performance and security.

Theory : prepare Statement is used to execute SQL queries safty in JDBC. It improves performance because the query is precompiled by the database. The same query can be used many time with different values. Prepared-statement is faster than Statement for repeated queries. It also improves security by preventing SQL injection attacks. Users input is treated as data, not a SQL code. Statement directly executs SQL and is less secure. PrepareStatement uses

placeholder (?) for values. These values are set using setter methods. So, prepared statement is better for fast performance and security.

Java code :

```
import java.sql.*;
class insertExample {
    public static void main(String[] args) {
        try {
        Connection con = DriverManager.getConnection(
        "jdbc:mysql://localhost:3306/testdb", "root",
        "password");
String sql = "insert into student (id, name)
        values (?,?);
PreparedStatement ps = con.prepareStatement(sql);
```

```
ps.setInt(1,1);
ps.setString(2,"Ram");
ps.executeUpdate();
System.out.println("Record inserted");
con.close();
} catch(Exception e) {
    System.out.println("Error occurred");
}
}
}
```

Conclusion: In conclusion, PreparedStatement is safer and more efficient than Statement because it supports parameterized queries and reuses compiled SQL.

Lab Report No : 07

Lab Title : Retrieving and processing Database Records Using Resultset in JDBC.

Introduction : The lab explain how a Resultset is used in JDBC to retrieve and process data returned from a database query.

Theory : In JDBC, a Resultset is an object that holds the data returned from executing a result query on a relational data such as : MYSQL.

How it retrives data is given below :

1. Establish a database connection
2. Create a statement
3. Execute the query
4. Iterate through the resultset
5. Retrive column values
6. Process the retrived data
7. Close Resources.

use of common methods with example :

Next() : moves the cursor to the nextrow, Return false when there are no more rows.

getString : Retrives the values of the specified column as a string.

getInt() : Retrives the value of the specified column as an int.

Example Code :

```
String query = "select id, name, marks from student";
ResultSet rs = stmt.executeQuery(query);
while(rs.next()){
    int id = rs.getInt("id");
    String name = rs.getString("name");
    int marks = rs.getInt("marks");
System.out.println(id + "|" + name + "|" + marks);
}
```

Conclusion : In conclusion, ResultSet provides an easy way to navigate query results row-by-row using next() to move through records and getter methods like getString() and getInt() allows structured and type-safe retrieval.