1.

IT23055

Lab No: 01

~~a. com~~ In OOP multiple inheritance means that a class can inherit from more than one parent class or multiple types.

Abstract class: A class can extend only one abstract class. Does not support multiple inheritance of classes.

Java code:

~~interface~~

```
abstract class A{
    abstract void show();
}

abstract class B{
    abstract void display();
}

class C extends A,B{}
```

Interface: A class can implemented multiple interfaces. supports multiple inheritance.

```
interface A {
        void show();
    }
interface B{
        void display();
}
class c implements A,B{
    public void show(){
      system.out.println("show");
   }
public void display(){
        system.out.println("Display.");
    }
}
```

Differences between Abstract class and interface:

Feature	Abstract class	Interface
Multiple Inheritance	Not supported	supported
Methods	Abstract + concrete	Abstract methods
Variables	Instances variable allowed	only public static final constants
constructor	Yes	No

~~Conclusion:~~

Use an abstract class when:

1. We want share common code.
2. We need to maintain state.
3. classes are closely related.
4. We don't need multiple inheritance.

IT-23055

Use an interface when:

1. We ~~m~~ need multiple inheritance.
2. Classes are unrelated but share behaviour
3. We want loose coupling.
4. We want to support future extensions.

conclusion: In conclusion, interfaces are preferred when multiple inheritance of type is needed and when defining a common ~~contrit~~ contraint across unrelated classes.

IT-23055

Lab No: 02

Encapsulation means wrapping data variables and method together and restricting direct access to data using access modifiers like private. This ensure

1. Data security: No one can directly modify ~~provide~~ private variables.

2. Data integrity: Input can be validated before changing the data. Invalid or harmful data is restricted.

Encapsulation ensure data security and Integrity

1. Hiding internal data (using private access modifier)
2. controlling access through public methods.
3. Validating inputs before modifying internal state.
4. Maintaing consistency by preventing invalid states.

Java code:

```
public class BankAccount{
        private String AccountNumber;
// private variables (can not be access directly)
        private double balance;

//constructor (optional)
public BankAccount (String accountNumber, double
                                        initialBalance)
  {
    . setAccountNumber (~~this~~ accountNumber);
      setInitialBalance (initialBalance);
  }
public void SetAccountNumber ( String accountNumber)
  if (accountNumber = null || accountNumber.trim().
                                        isEmpty()){
   System.out.println ("Invalid account Number.");
}
~~else~~else {
        this.accountNumber = accountNumber;
    }
}
```

```
//setter for initial Balance with validation
public void setInitialBalance(double balance){
  if (balance<0){
      system.out.println("Balance can't be negative.");
    }
  else {
      this.balance = balance;
    }
}
//getter methods
  public String getAccountNumber(){
        return accountNumber;
    }
  public double getBalance(){
      return balance;
//Example method to deposit money
public void deposit(double ammount){
    if (amount>0){
```

```
        balance = balance + amount;
    }
    else {
        system.out.println("Deposit amount should be
                                                positive;");
    }
}
//example method to withdraw money
public void withdraw (double ammount){
 if(ammount>0 && amount<= balance){
        balance -= amount;
    }
    else{
        system.out.println("Invalid withdrawal amount");
    }
}
public class Main{
public static void main (string[] args){
BankAccount account = new BankAccount (" ",-100);
account.SetAccountName ("12345");
```

```
account.setInitialBalance(500);
account.deposite(200);
account.withdraw(800);
System.out.println("Account Number:"+account.
                    getaccountNumber());
System.out.println("Balance:"+account.getBalance());
    }
}
```

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

Lab No: 03

Lab Name: Developing a multithreading based project to simulate a car parking management systems.

Introduction: To develop a multithreading based project with named car Parking mangement. The goal is to demonstrate thread sychronization and resource management.

Java code:

~~class Responsibilities~~

1. RegisterParking.Java

```
class RegisterParking {
    private string carNumber;
    public RegisterParking(String carNumber){
        this.carNumber = CarNumber;
    }
    public string get carNumber(){
```

```
        return carNumber;
    }
}
```

2. ParkingApp.java

```
import java.util.LinkedList;
import java.util.Queue;
class ParkingApp{
    private Queue<RegisterParking> queue = new
                                LinkedList<>();
    public synchonized void addCar (RegisterParkin
                                    Car){

      queue.add(car);
    System.out.println("Car "+car.getCarNumber()+"
                        requested parking.");
     notifyAll();
   }
public synchronized RegisterParking getCar(){
      while (queue.isEmpty()){
```

```
try {
    wait();
}
catch(InterruptedException e){
        e.printStackTrace();
    }
}
return queue.poll();
}
}
```

3. ParkingAgent.java

```
Class parkingAgent extends Thread{
    private parkingpool pool;
    private int agentID;
public ParkingAgent(parkingPool pool, int agentID){
        this.pool = pool;
        this.agentID= agentId;
}
@Override
    public void run(){
      while (true){
```

```
RegistarParking car = pool.getCar();
System.out.println("Agent "+agentID + " parkedcar"+
                        car.getCarNumber()+" . ");

try {
    Thread.sleep(1000);}
catch (InterruptExeception e){
    e.printStackTrace();}
 }
}
```

4. MainClass.java

```
public class mainClass{
   public static void main (String[] args){
     parkingPool = new ParkingPool();

   ParkingAgent agent1 =new ParkingAgent(pool,1);
   ParkingAgent agent2 = new ParkingAgent(pool,2);
   agent1.start();
   agent2.start();
String[] cars ={"ABC123," 'XYZ456","LMN789", DEF456:; "GHI999"};
for( String carNumber :cars){
```

```
RegisterParking car = new RegisterParking(carNumber)
        pool.addCar(car);

try {
    Thread.sleep(500);
  }
   catch (InterruptedException e) {
    e.printStackTrace();
  }
 }
}
}
```

Output:

```
Car ABC123 requested parking.
Car XYZ456 requested parking.
Agent1 parked car ABC123.
Agent2 parked car XYZ456.
Car LMN786 requested parking
Agent1 parked car LMN786
Car DEF456 requested parking
Agent 2 parked car DEF456.
```

conclusion: The project successfully demonstrates the use of multithreading to manage a real-time car parking system. This experiments highlights how thread coordination and shared resource management can optimize high traffic systems ensuring that Parking Agents process requests in a reliable, orderly fashion.

Lab No: 04

Lab Title: Database connectivity and select query execution using JDBC.

Introduction: The lab focuss on how JDBC connects a java application with relational databases.

Theory: JDBC acts as bridges between Java and the database ~~things~~. using:

1. DriverManager - loads the JDBC driver.
2. Connection - establish the session
3. statement - send sql commands.
4. ResultSet - Holds the data return from query.

Heres the consise 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.

Javacode:

```
import java.sql.*;
class SelectExample{
    public static void main(String[] args){
        Connection con = null;
    try{
        con = DriverManager.getConnection(
        "JDBC://localhost:3306/testdb",
        "root", "password");
      Student  st = con.createStatement();
    ResultSet rs = st.executequery("select *from
                                    student");
```

```
        while(rs.next()){
            System.out.println(rs.getInt(1)+" "+rs.getString(1));
        }
    }
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 resultsets.

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 MVC pattern.

Theory: In Java EE application a servlet controller manage ~~how~~ the flow between the Models and the view by:

1. Receiving request from the client
2. calling model classes to process data on fetch 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 student{

```
    private string name;
    private int age; }

public student (string name, int age){
    This.name = name;
    This.age = age; }

public string getName();
{
  return name;
 }
public $int getAge(){
    return age;
  }
```

2. JSP (view):

```
<%@ page import * package.student" %>
<% student student = (student) request.getAttribute("student
                                                    Data");
 %>
<html>
<head> <title> studentInfo</title> </head>
<body>
  <h2> student.details </h2>
</body>
</html>
```

3. controller (Servlet):

```
@webservlet("/student")
public class studentServlet extends HttpServlet{
protected void doGet(HttpServlet request requested,
                     Http servlet response response)
throws servlet Exception [Exception {
        student students = new student ("Sumi", 22);
        request.setAttribute("StudentData" student);
RequestDispatcher dispatcher request.setrequest
                     Dispatcher ('student.jsp");
dispatcher.forward request, response);
    }
}
```

conclusion: In conclusion the servlet controller manages application flow by seperations business logic from presentation.

Lab 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 performances and security.

Theory: Prepare Statement is used to execute sql queries safety in JDBC. It improves performance because the query ~~is~~ is precompiled by the database. The same query can be used many times with different values. Prepared-Statement is faster than statement for repeated queries. It also improves ~~seq~~ security. It ~~is~~ use placeholder(?) for input values. So user ~~data~~ input is handled by data not as code. This protects the application from SQL injection attacks.

MySQL table:

```
Create table student (
    id INT,
    name varchar(50));
```

Java code:

```
import.java.sql.*;
public class InsertEmample {
    public static void main (String[] args){
String url = "Jdbc:mysql://localhost:3306/studentdb;
String sql user = "root";
String password = "password"
String sql ="Insert Into Student (id, Name)
            VALUES (?, ?)";
try {
   //Load Driver
class.forName ("com.mysql.cj.jdbc.Driver");
Connection con = DriverManager.getConnection (Url, User,
                                              password);
PreparedStatement ps = con.PreparedStatement (sql);
ps.SetINT(1,101);
ps.setString(2,"sumi");
```

```
//Execute
ps.executeUpdate();
System.out.println("Record Inserted Successfully");
ps.close();
con.close();
}
catch (Exception e) {
    e.printStackTrace();
  }
}
}
```

conclusion: PreparedStatement is safer and more efficient than statement because it supports parameterized queries and reuses compiled SQL.

Lab Report No:07

Lab Title: Retreiving 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 execution 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. Retrieve column values
6. Process the retrieve data
7. Close resources

Use of common methods with example:

Next(): Moves the cursor to the next row, Return false when there are no more rows.

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

getInt(): Retrieves the value of the specified column as an INT.

Example Code:

```
String query = "Select id, name, marks from stuid;"
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 ~~m~~ negative query results row-by-row using next() to move through records and getter methods like getString() and getINT() allows structured and type-safe retrieval.