Lab 1: Abstract classes vs Interfaces

objective: understand the difference between abstract class and interfaces, particularly in terms of multiple inheritance, and know when to use.

Abstract class:
- when i want to share common code among related classes.
- when i want to instance variables on class.

Interfaces:
- when i want unrelated classes follow the same contract.
- when i want multiple inheritance of type

Code:

```
abstract class vehicle {
        string brand;

    vehicle (String brand) {
        this.brand = brand;
    }
abstract void stunt ();
```

```
Void  showBrand (){
        System.out.Println("Brand : "+ brand);
    }
interface  Electric{
        void  change();
    }
interface  GPS{
        void navigate();
    }
class  Tesla extends vehicle implements
        Electric, GPS {
        Tesla (String brand){
            Supera (brand);
    }
@override
    void start (){
        system.out.Println ("Tesla Started
        silently.");  }
@ Public  void change(){
    system.out.Println("Navigator using GPS")
    }
}
```

```
Public class Lab1{
    Public static void main(String[] args){
Tesla myCar = new Tesla("Tesla model S");
myCar.ShowBrand();
myCar.start();
myCar.charge();
mycar.navigate();
}
}
```

Conclusion: Abstract classes are best for related classes sharing common code. Interfaces are ideal for multiple inheritance and defining contracts for unrelated classes.

Lab 2: Encapsulation for Data security and Integrity.

objective: understand how encapsulation protects data security and ensures data integrity by restricting direct access to class variables and providing controlled access through validated methods.

Encapsulation is the process of hiding the internal state of an object and providing public method to access and modify it.

BankAccount class:

```
Public class BankAccount {
        Private String accountNumber;
        Private double balance;

Public void setAccountNumber (String
accountNumber) {
if (accountNumber = null || accountNumber.
        isEmpty()) {
```

```
system.out. Println ('Invalid account numbers.
Cannot be null ons empty "); }
else { this. accountNumbers = accountNumbers;
System .out. Println ("Account numbers set
Successfully. ");
}}
Public void setInbtialBalanee (double balance){
if ( balance <0) {
system .out. println("Invalid balance");}
else {
this .balanee=balanee;
System.out. Println (" Initial
balanee set successfully ."};
}
}
Public String getAccountNumbers () {
return accountNumbers;
}
Public class getBanance () {
return balance;
}
```

```
Public void deposit (double amount){
    if(amount ≤ 0){
      system.out. Println("Deposit amount
must be Positve"); }
    else{ balance += amount;
      system.out.println("Deposited: " + amount +
"; New balance: " + balance);
}}
Public void withdraw(double amount) {
    if(amount <= 0){
     System.out. Println(" withdraw/ amount
must be Positive."); }
else if(amount > balance){
      System.out.Println("Insufficient balance");
} else{
    balance -= amount;
    }
}
```

```
Public class Lab2 {
    Public class Static void main(String [] args)
  { BankAccount.account =new BankAccount();
    account.setAccountNumber("12345");
    account.setInitialBalance(1000);

    account.deposit(500);
    account.withdraw(200);
    account.deposit(-50);
    account.setAccountNumber("");
    account.setFhitialBalance(-100);

      System.out.Println("Final Account
        Number:"+account.getAccountNumber);
   System.out.Println("final Balance:"+
          account.getBalance());

}
}
```

conclusion: Encapsulation Ensures security and integrity. Prevent direct modification of sensitive data. Reject null, empty, or negative values to maintain data integrity.

Lab3 Multithreading Cars parking management System.

Objective: To understand and apply multithreading in java. To simulate a cars parking management system. Using threads and a shared synchronized queue.

Code:

```
import java.util.*;
import java.util.concurrent.*;

class RegisterParking {
        String carNumber;
    RegisterParking(String carNumber){
    this.carNumber = carNumber;
    }
}

class ParkingPool{
    queue<RegistranParking> queue= new
        LinkedList<>();
Public synchronized void requestParking
    (RegistarParking car){
    queue.add(car);
```

```
System.out.println("Car" + car.carNumber +
        "requested parking.");
        notifyAll();
    }
Public synchronized RegistarParking getParking()
    throws InterruptedException{
        while (queue.isEmpty()){
            wait();
        }return queue.poll();
}}
class ParkingAgent extends Thread{
        ParkingPool Pool;
    ParkingAgent(parking Pool, String){
        super(name);
        this.Pool = Pool;
    }
Public void run(){
        try{
            while(true){
    RegistrarParking car = Pool.getParking.
    Request();
System.out.Println(getName() + " Parked car"
+ car.carNumber + ".");
```

```
Thread.sleep(500);
}
} catch (Interrupted Exception e)}
        System.out.Println(getName() +
                "stopped.");
}
}
} public class Lab3{
    public class void main(String[] args){
    ParkingPool Pool = new ParkingPool();
new ParkingAgent (Pool, "Agent 1").start();
new ParkingAgent (Pool, "Agent 2").start();
String[] cars = {"ABC123", "XYZ456", "LMN789"
for(String carNum : cars){
    Pool.requestParking(new RegistrParking
    (carNum));
try{thread.sleep(200);} catch(
Interrupted Exception e){}
}}
}
```

Conclusion: The lab demonstrated how multiple threads can work concurrently to process requests from a shared queue. Synchronized methods ensured that parking requests were handled safely and in order.

Lab 4: JDBC Communication with Relational Database.

Objective: To understand how JDBC enables communication between a java application and a relational database. To execute a SELECT operation using JDBC with Proper error handling.

Communication: users JDBC Drivers to translate java calles into database-specific Command Provide Standard interfaces for interation.

Code:
```
try {
    Connection con=DriverManager.getConnection
        (url, user, pass);
    Statement st = con.createStatement();
    Resultset rs = st.executeQuery("SELECT
    *FROM student);
while (rs.next()) {
        System.out.Println(rs.getString
        ("name"));
    } }
```

```
catch (SQLException e){
    System.out.println(e.getMessage());
        }
finally {
    }
```

Conclusion: JDBC provides a reliable and secure way to connect Java application with databases. Proper use of try-catch-finally ensures robust error handling and resource management.

Lab 5: Servlet Controller Flow

Controller:
- Receives HTTP request
- Interacts with model
- Sends data to view using
- Forwards request using

View:
- Reads data from request
- Renders HTML response

Code:
```
import java.io.IOException;
import jakarta.servlet.*;
import jakarta.servlet.http.*;

Public class StudentServlet extends HttpServle
{ Protected void doGet{HttpServletRequest
request, Http servletResponce responce)
throws ServletEnception, IOEnception{
String name = "Nazmul";
int marks = 85;
request.setAttribute("StudentName", name);
request.SetAttribute("Student mark," marks);
RequestDisPatcher rd = request.getRequest DisPatcher
("result.jsp"); rd.forward(request
```

JSP Code:

```
<html>
 <head><title> Result </title></head>

<body>
   <h2> student  Result </h2>

   Name : ${studentName} <br>

   Marks : ${studentmarks}

</body>
</html>
```

Conclusion: servlet controls application flow. Data passed using request.setAttribute. JSP displays data using Expression Language.

Lab 6: PreparedStatement vs Statement in JDBC

Prepared statement is better Because of Security which prevents SQL Injection. User input is treated as data, not SQL code. it also gives Readability and Performance.

Code: import java.sql.*;

Public class InsertData{

Public static void main (String[] arg

{ String url="jdbc:mysql://localhost:3306/testdb

String user="root";

String pass="1234";

try{

connection con=DriverManager.getConnection

(url), user, pass);

String sql="Insert into student (id,name,

marks) values (?,?,?)";

PreparedStatement PS=con.PrepareStatement

(sql);

Ps.setInt(1,1);

Ps.setString(2,"Nazmul");

```
        ps.setInt(3,90);

        ps.executeupdate();
        System.out.Println("Record inserted
                    successfully");

        ps.close();
        con.close();
    } catch(SQLException e) {
            System.out.println(e.getmessage());
    }
}
```

Conclusion: Prepaedstatement improves security, performance, and code quality Recommended for all dayhamic SQL operation in JDBC

Lab 7: Simple CRUD Application using Spring Boot & MySQL

Objective: To design a simple Boot CRUD application to manage student records using Spring Data JSP Repository.

Code:

```
import jakarta.Persistence.*;
@ public class Student{
   @
   @ GeneratedValue(Strategy= GenerationType.
        IDENTITY)
   Private string name;
   Private int marks;

}
```

Repository Interface:

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

1. Create

```
Student s = new Student();
s.setName("Nazmul");
s.setMarks(85);
StudentRepository.save(s);
```

2. Read

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

Student s = studentRepository.findById(1L)
            .orElse(null);
```

3. Update

```
Student s = studentRepository.findById
            (1L).orElse(null);

if(s != null) {
    s.setMarks(90);
    StudentRepository.save(s);
}
```

4. StudentRepository.deleteById(1L);

```
import org.springframework.web.bind.
    annotation.*;
```

```
@RestController
@RequestMapping("/students")
  public class studentController{
      private final studentRepository repo;

      public studentController(StudentRepository
      repo){
      this.repo = repo;

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

}
}
```

Conclusion: Spring boot + JPA Repository Simplifies CRUD operations. NO SQL required. Clean, fast and scalable architecture

Lab 8: Spring Boot RESTful services

objective: Spring Boot Simplifies REST Develop-ment. No XML configuration needed. Built-in Tomcat, no external deployment. Easy dependency management. less boilerplate, cleaner code.

Entity: Public class Student {

Private Long id;

Private String name;

Private int marks;

}

REST Controller:

import org.springframework.web.bind.annotation.*;

import java.util;

@RestController

@RequestMapping("/api/students")

Public class StudentRestController{

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

@GetMapping

Public List<Student) getAllstudents() {

return students;

```
@PostMapping
    Public student add student (@RequestBody
        student  student) {
        students.add(student);
        return student;

    }
}
```

JSON
```
{ "id":1,
  "name": "Nazmul",
  "marks":80
}
```

Conclusion:

Spring Boot enables fast REST API development. minimal configuration, automatic JSON handling.