Lab 1: Abstract class vs Interface (multiple Inheritance)

Aspect	Abstract class	Interface
Multiple Inheritance	Not supported	supported
Reason	A class can extends only one abstract class	A class can implement multiple interfaces
Keyword used	extends	implements

When to use Abstract class:

- We need code reuse (common method implementations)
- classes are closely related
- you want to store instance variables/state
- Expect future subclasses to share base behaviour

When to use Interface:

- We need multiple inheritance
- Classes are unrelated but share common behavior
- want to define a contract
- want to support plug-and-play design.

BankAccount (Encapsulated class with validation):

```
Public class BankAccount {
  private String accountNumber;
  private double balance;
  public void setAccountNumber(String accountNumber){
  if(accountNumber == null || accountNumber.isEmpty()){
     system.out.println("Invalid account number!");
    }
  else {
     this.accountNumber = accountNumber;
    }
  }
public void setInitalBalance(double balance) {
    if (balance < 0){
    System.out.println("Balance cannot be negative!");
    }
   else {
   this.balance = balance;
  }
 }
```

```
public string getAccountNumber() {
    return accountNumber;
}
public double getBalance(){
    return balance;
}
}
public class Main {
 public static void main(string[] args){
 BankAccount acc = new BankAccount();
 acc.setAccountNumber(" ");
 acc. setInitialBalance(-500);
acc. setAccountNumber("ACC12345");
acc. setInitialBalance(1000);
system.out.println("Account Number: "+acc.get.
                        AccountNumber());
system.out.println("Balance:" +acc.getBalance());
}
}
```

Lab 2: How Encapsulation Ensure Data Security and Integrity....

Encapsulation: Encapsulation means wrapping data (variables) and methods together in a class and restricting direct access to the data using modifiers like private.

It ensures security and integrity by:

1. Hiding Data:
   - private variables cannot be accessed directly from outside the class.
2. Controlled access:
   - Data can be modified only through validated methods (setters).
3. preventing invalid data:
   - Setter reject null, negative or invalid values.
4. Maintaining Consisteney:
   - Objects all always remains in a valid state.

```
public synchronized void addcar(Registrarparking car){
    queue.add(car);
    notify();
}
public synchronized Registrarparking getcar(){
 while (queue.isEmpty()){
   try{
   wait();
  } catch(InterruptedException e){
      e.printstackTrace();
    }
   }
  return queue.poll();
  }
}
```

ParkingAgent (Thread):

```
public class parkingAgent extends Thread{
 private parkingpool pool;
 private string agentName;
```

```
public parkingAgent(parkingpool, string agentName){
  this.pool = pool;
  this.agentName = agentName;
}

public void run() {
    while (true) {
    Registrarparking car = pool.getcar();
  System.out.println(agentName + "parked car" + car.
                getCarNameber() + ". ");

   try {
     Thread.sleep (1020);
    }
    catch(InterruptedExcception e){
       e.printstackTrace();
    }
   }
```

Main class(simulation):

```
public class mainclass{

public static void main(string[], args){
```

Lab 3: Car parking Management System (Multithreading - java)

RegistrarParking (parking Request)

```
public class Registrarparking {
    private string carNumber;

public Registrarparking (string carNumber){
    this.carNumber = carNumber;
System.out.println("car"+carNumber+"requested
                                    parking");
}
public string getcarNumber(){
    return carNumber;
  }
}
```

parkingPool (shared synchronized queue)

```
import java.util.LinkedList;
import java.util.queues;
public class parkingpool {
Private queue<Registrarparking>queue=
new LinkedList<>();
```

Lab 4: Complete JDBC example with Error Handling:

```
import java.sql.*;
public class JdbcSelectExample {
  public static void main(String[] args) {
    Connection con = null;
    Statement stmt = null;
    ResultSet rs = null;
try {
    Class.forName("com.mysql.cj.jdbc.Driver");
    con = DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/testdb",
      "root",
      "password"
     );
   stmt = con.createStatement();
   rs = stmt.executeQuery("SELECT * FROM
                           student");
  while(rs.next()) {
  System.out.println(
      rs.getInt("id") + " " +
      rs.getString("name"));
```

```
}
}
catch(ClassNotFoundException e) {
  System.out.println("JDBC Drivers not found");
    e.printStackTrace();
  }
  catch(SQLException e){
  System.out.println("Database error occurred");
    e. printStackTrace();
  }
  finally {
     try {
     if(rs!=null) rs.close();
    if(stmt!=null) stmt.close();
    if(con!=null) con.close();
  }
   catch(SQLException e) {
     e. printStackTrace();
        }
      }
    }
  }
```

```
parkingpool pool = new parkingpool();
parkingAgent agent1 = new parkingAgent(pool, "Agent 1");
parkingAgent agent2 = new parkingAgent(pool, "Agent 2");
    agent1.start();
    agent2.start();
String[] cars = {"ABC123", "XYZ456", "DEF789", "LMN321"};
for (String car: cars) {
    new ~~(String~~
    new Thread(() -> {
    Registrarparking request = new Registrarparking
                                    (car);
      pool.addcar(request);
    }).start();
  }
 }
}
```

```
this.name = name;
 this.id = id;
}
public String getName() {
      return Name;
   }
 public int getID() {
       return id;
    }
  }
```

Servlet Controller

```
import jakarta.servlet.*;
import jakarta.servlet.http.*;
import java.io.IOException;
public class studentServlet extends HttpServlet {
 protected void doGet(HttpServletRequest request,
                      HttpServletResponse response)
     throws ServletException, IOException {
```

```
Student student = new Student("Rahim", 101);
request.setAttribute("studentData", student);
RequestDispatcher rd = request.getRequestDis-
patcher("student.JSP");
  rd.forward(request,response);
 }
}
```

JSP view (Student.JSP)

```
<%@ page contentType = "text/html" %>
  <html>
   <head>
      <title> Student info </title>
   </head>
  <body>
  <h2> student Details </h2>
<p> Name: ${studentData.name} </p>
<P> ID: ${studentData.id} </p>
</body>
</html>
```

Lab5: Role of a servlet Controller in JavaEE:

In java EE web Application, the servlet acts as the controller in the MVC (Model-View-Controller) pattern.

How the controller manages flow:

1. Receives request from client (Browser)
2. Interacts with the model
3. Stores data in request/session scope
4. Forwards request to a jsp (view)
5. JSP renders the response to the user

Client → servlet (controller) → Model → servlet → JSP (view) → client

Servlet Forwarding Data to JSP:

Model class (Bussiness logic)

```
public class student {
    private string name;
    private int id;
public student (string name, int id) {
```

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

 con = DriverManager.getConnection (
     "jdbc:mysql://localhost:3306/testdb", "root",
     "password");
String sql="INSERT INTO student (id, name) VALUES(?,?)";

PS = con. preparedStatement(sql);
PS=setInt(1, 101);
PS=setString(2,"Rahim");
PS.setInt
 PS. executeUpdate();
System.out.println("Record inserted successfully");
} catch(classNotfoundException e){
 System.out.println("Driver not found");
   e.printStackTrace();
} catch(sqlException e){
 System.out.println("Database error");
   e.printStackTrace();
}
```

```
finally {
 try {
 if (ps!=null) ps.close();
 if (con!=null) con.close();
 }
 catch (SQLException e) {
      e.printStackTrace();
      }
    }
  }
}
```

Lab 6: How PreparedStatement Improves performance and security....

Performance Improvement:

- SQL query is pre-complied onee by the database.
- Same query can be exeeuted multiple times with different parameters.
- Reduces query parsing and execution time.

Security Improvement:

- prevents SQL Injection attacks
- User input is treated as data, not exeeutable SQL.
- Special characters (', --, ;) are safely handled.

Insert Record Using preparedStatement (mysql):

```
import java.sql.*;
public class preparedInsertExample{
  public static void main(String[] args){
    Connection con = null;
    preparedStatement ps = null;
```

getInt (columnName/index)

- Retrives integer data from a et column
- Used for INT columns

Retrieving Data from mysql Using Resultset:

```
import java.sql.*;
public class ResultSetExample {
public static void main(string[] args) {
try{
class.forName("com.mysql.cj.jdbc.Driver");
Connectim con = DriverManager.getConnection(
"jdbc:mysql://localhost:3306/testdb", "root", "password");
statement stmt = Con.Createstatement();
Resultset rs = stmt.executeQuery("seleect id,
        name, from student");
while (rs.next()){
  int id = rs.getInt("id");
 String name = rs.getstring("name");
```

```
System.out.println("ID: "+id+", Name: "+name);
}
rs.close();
stmt.close();
con.close();
}catch(Exception e){
    e.printStackTrace();
  }
 }
}
```

Sample Output :

ID : 101, Name : Rahim

ID : 102, Name = AA

Lab 8: How Spring Boot Simplifies RestFul service Development:

1. Auto-Configuration: Spring Boot automatically configures embedded Tomcat, Jackson (JSON) and Spring MVC.
2. Embedded Server: No need to deploy to an external server - run with main() method.
3. Annotation-based Development: use @RestController, @GetMapping, @PostMapping to define endpoints.
4. JSON support: converts java objects to JSON automatically using Jackson.
5. Minimal ~~Bi~~ Boilerplate: No XML config, no explicit servlet setup, reduces repetitive code.

Rest Controller in Spring Boot:

A Rest Controller handles HTTP requests and returns JSON responses,

Key Annotations are below:

lab 7: What is a ResultSet in JDBC?

A ResultSet in an object in JDBC that stores the data returned by a SELECT query. It represents a table of data where each row corresponds to a database record.

How ResultSet is Used to Retrive Data:

1. Execute a SELECT query using statement or preparedstatement.
2. Store the result in a ResultSet
3. Move the cursor using next();
4. Retrive column values using getter methods.

Important ResultSet Methods:

1. Next()
   - Move cursor to the next row
   - Returns true if a row exists, otherwise false.
   - Must be called before reading data
2. getString(columnName/Index)
   - Retrives string data from a column
   - Used for varchar, char, text columns

Lab 9: Project Demonstration

Student Management System (Spring Boot + MySQL + GUI)

Project Overview:

objectives: To develop a CRUD based student management system where users can:

- Add students, view Student list, update student information, Delete students

Project Architecture:

Controller → Services → Repository → MySQL

↓

thymeleaf (GUI)

Student Entity (model)

```
@Entity
@Table(name = "students")
public class student {
@Id
@GeneratedValue( strategy = GenerationType.
                                Indentity)
private Long id;
private String name;
```

```
private int age;
}
```

Student controller:

```
@ controller
public class studentController {
    @Autowired
  private StudentRepository studentRepo;
  @GetMapping("/")
  public string viewHome(model model){
  model.addAttribute("students", studentRepo.
      return "index";                findAll());
  }
@GetMapping("/add")
public string addform(model model){
  model.AddAttribute("student", new student());
    return "add student");
}
@postMapping("/save")
public string savestudent (@ModelAttribute
```

1. @RestController : Marks a class as a REST controller (combines @Controller + @ResponseBody)
2. @GetMapping("/path"): Maps HTTP GET requests to a method.
3. @postMapping("/path"): Maps HTTP post requests to a method.
4. @RequestBody : Maps JSON from request body to java object
5. @ResponseBody: Maps Java object to JSON (automatic with @RestController)

```
<form action = "/save" method = "post">

    Name : <input type = "text" name = "name">

        <br> <br>

    Age : <input type = "number" name = "age"> <br>
                                        <br>

    <button type = "submit"> Save </button>

</form>
</body>
</html>
```

Conclusion: This project demonstrates a complete student management system using spring Boot and mysql with a graphical user interface. CRUD operations are implemented using Spring Data ~~JAP~~ JPA, and Thymeleaf is used to render dynamic web pages. The system allows users to add, view, and delete student records efficiency through a user-friendly interface.

```
        Student student) {
    studentRepo.save(student);
      return "redirect:/";
}
@GetMapping("delete/{id}")
public string deletestudent (@pathVariable.
                             long id){
    studentRepo.deleteById(id);
    return "redirect:/";
}
```

Gpa Graphical User Interface (GUI):

Student Form

```
<!DOCTYPE html>
<html xmlns:th="https://www.thymeleaf.org">
<head>
    <title>Add student</title>
</head>
<body>
<h2>Add student</h2>
```