Lab Report : 01

Lab Title : Comparison Between Abstract classes and Interfaces in Terms of Multiple Inheritance in Java

Theory :

Multiple Inheritance : Multiple inheritance refers to a feature where a class can inherit properties and behaviors from more than one parent class. Java does not support multiple inheritance using classes to avoid ambiguity. but the supports multiple inheritance using interfaces.

Abstract class :

An abstract class is a class that cannot be instantiated and may contain:

- Abstract methods (without body)
- Non-abstract methods (with body)
- Instance variables
- Constructors

Java code:

```
abstract class vehicle {
  abstract void start();
  void fuelType(){
     system.out.printIn("uses fuel");
  }
}
```

Interface:

An interface is blueprint of a class that contains

- Abstract methods
- constants (public static final)
- Default and static methods

Java code:

```
interface Electric{
  void change();
}
Interface Automomous {
  void autoDrive();
}
class Tesla implements Electric, Autonomous {
   public void charg(){
   system.out.printIn ("charging"};
  }
```

```
    public void autoDrive(){
        system.out.printIn ("Auto driving ");
    }
}
```

Comparison Between Abstract class and Interface:

Feature	→	Abstract class	—	Interface
1. Multiple Inheritance	→	Not supported	—	supported
2. Method Implementation	→	Allowed	—	Not allowed
3. variables	→	Instance variables allowed	—	only constants
4. Constructors	→	Allowed	—	Not allowed
5. Access Modifiers	→	Any	—	public only
6. Inheritance keywork	→	extends	—	implements

Lab Report - 2

Encapsolation is an oop concept that hides data and protect them from direct access. It ensures data sequrity by making variable private inside ~~ae~~ class. Outside ~~access~~ class 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 keep data consistent and correct. If invalid ~~data~~ data is given, the method reject it. In bank system, this is a very important for safety.
Therefore, encapsulation ensures both data and sequrity

Java code:

```
class Account{
   private String accountNumber;
   private double balance;
   public void setAccountNumber(String accNO){
        if (accNO!= null && !accNo.isempty()){
            accountNumber = accNO;
        }
        else{
        system.out.printIn("Invalid account number");
       }
     }
   public void setInitialBalance(double amount){
   if (amount >=0){
   balance amount;
   }
  else{
   System.out.printIn("Balance cannot be negative");
  }
 }
}
```

Lab Report - 03

Lab title: simulation of a Multithreading-Based car Parking management system using Java

Multithreading allows a program to execute multiple tasks concurrently. In this experiment the producer consumer problem is implemented:

- Producers: Car requesting parking
- Consumers: Parking agents parking cars
- Shared Resource: Parking pool (queue)

To avoid race conditions and data inconsistency. Java's "syncronized" keyword along with wait() and modify() methods are used.

Algorithm of the project:

1. Initialize a shared ParkingPool
2. Create and start multiple ParkingAgent threads
3. simulate multiple cars requesting parking concurrently.
4. Each car adds its request to the parking pool.

5. Parking agents retrive requests from the pool.
6. If the pool is empty, agents wait.
7. When a new car arrives, agents are notified.
8. The process continues concurrently.

Source code :

RegistratParking.java

```
public class RegistrarParking{
     private string carNumber;
     public RegistrarParking(string carNumber){
          this.carNumber = carNumber;
     }
     public string getCarNumber(){
          return carNumber;
     }
  }
```

ParkingPool.java

```
import java.util.LinkedList;
import java.util.Queue;
public class ParkingPool {
     private Queue<RegistrarParking> queue = new
                                        linkedList<>();
public synchronized void addcarr (RegisterarrParking
                                          carr) {
 queue.add (carr);
 System.out.printIn("carr" + carr.getcarrNumberr( ) +
                               "rrequested parking!");
 notify();
}
public synchronized RegistrarrParking getcarr()
                         throws Interrrupted Exception{
 while(queue.isEmpty()){
    wait();
  }
  rreturrn queue.poll ();
 }
}
```

Parking Agent.java

```
public class ParkingAgent extends Thread{
   private ParkingPool pool;
   private int agentId;
 public ParkingAgent (parkingPool, int agentId){
     this.pool = pool;
     this.agentId = agentId;
 }
 public void rrun (){
  try{
```

```
while (true){
  RegistercarParking car = pool.getcar();
  System.out.printIn("Agent"+ agentId + "parked car"
                        + car.getCarNumber()+".");

   Thread.sleep(1000);
   }
  } catch (InterruptedException e){
    System.out.printIn("Agent"+agentId +"stopped.");
  }
 }
}
```

MainClass.java

```
public class MainClass{
    public static void main (string[] args){
        ParkingPool pool= new ParkingPool();
        ParkingAgent agent1 = new ParkingAgent (pool,1);
        ParkingAgent agent2 =new ParkingAgent (pool,2);
        agent1.start();
        agent2.start();
        string[]cars = {"ABC123","XYZ456", "DEF789"};
for (String car : cars) {
    new Thread (() -> {
     · pool.addcar (new RegistrarParcking (car));
      }).start ();

    }
  }
}
```

Sample output:

```
Car ABC123 requested parcking.

Car XYZ456 requested parcking.

Agent 1 parcked car ABC123.

Agent 2 parcked car XYZ456.
```

Lab Report : 04

JDBC stands for Java Database Connectivity. It is used for connect a ~~data~~ Java program with a relational database. JDBC work as a bridge between Java application and database. The java program sent SQL queries using JDBC. JDBC driver receaves the request and talks to the database. The database process the quary and sends to the result back. JDBC driver receaves/returns the result to the Java program. JDBC allow data to be inserted, updated, deleted and read. It also handles connection and error management. Thus JDBC manages smooth connection between Java and Database. To execute a SELECT query, JDBC follows some fixed steps. First a connection is created with the database.

Then a statement prepared statement object is created. After that, a ~~se~~ SELECT SQL query is written. The query is executed using executequery method. This method return a ResultSet. try block is used to handle database operation safely. catch block handles SQL or runtime errors. Finally block is used to close connection and resources.

Java code:

```
import Java.sql* ;
class selectExample{
   public static void main (string[] args){
        collection con = null;
   try{
        con = DriverManager get.connection (
  "Jdbc:mysql://localhost:3306/testdb", "root", "Password");
        statement st = con.createstatement ();
   Resultset rs = st.executeQuery ("SELECT * FROM
                                        Student");
while(rs.next()){
System.out.printIn (rs.getInt(1)+ "  "+rs.getstring(2));

}
}
  catch (Exception e) {
     system.out.printIn (" Ennop occuned");

   } finally {
     try {
       if (con ! = null)
        con.close ();
       }
      catch (Exception e){
     system. out. printIn ("connection not closed");

    }
   }
  }
 }
```

Lab report : 5

In a Java EE application , a servlet works as a controller. The controller manages the flow between model and view . The model contains between business data and logic . The servlate gets data from the model . Then it sends this data to the view. JSP is used as the view to show data to the user. The servlat forward the request to JSP using request attributes. JSP reads the data and displays ~~if~~ it. Thus , the servlet controller controls the application flow .

Java code:

```
import Java.io * ;
import Javax servlet.*;
import Javax.http.* ;
public class Helloservlet extends Httpservlet {
protected void doGet (Httpservletrequest req,
    Httpservletresponse res)
throws servletException.IOException {
    String name = "student" ;
   req.setAttribute("msg" ;name);
  RequestDispatcher rd = req.getRequestDispatcher(
                              "hello.jsp");
        rd.forward (req.res);
     }
   }
```

<u>JSP code for view:</u>

```
  <html>
  <body>
    <h2> Hello, ${msg} <h?
  </body>
  </html>
```

Lab report :- 068

Prepaidd statement is used to execute SQL queries safely in JDBC. It improves performance because the query is precompiled by the database. The same query can be used many time with different values. Prepiared statement is faster than statement for repeaired queries. It also improves security by preventing SQL injection attacks. user input is treated as data, not as SQL code. statement direclty executes SQL and is less secure . preparedstatement uses placeholder(?) for values. These values are set using setter methods. So, prepared-statement is better for the performance and security.

Java code (Insert using preparedstatement):

```
import java.sql*;
class insertExecute{
    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 =com.preparedstatement(sql);

   ps.setInt(1,1);
   ps.setString(2,"Tajul");
   ps.executeupdate();
   system.out.printIn("Record inserted");
   con.close();
   }
    catch (Exception e){
    system.out.printIn("Error occurred");
   }
   }
  }
```

Lab report - 07:

Resultset is an object in JDBC that store data returned from a database query. It is mainly used with select querries. Resultset works like a table with rows and columns. The next( ) method moves the cursor to the next row. It returns true if data is available. The setString( ) method is used to read string type data. Data is read column by column from Resultset. ResultSet helps Java programs fetch database records easily. Thus it is very important for retriving data in JDBC.

Java code(resultset usage)

```
import java.sql*;
class resultsetExample{
public static void main (string[] args){
 try{
```

```
Connection con = DriverManager.getConnection(
  "Jdbc:mysql://localhost:3306/testdb", "root", "password");
  Statement st = con.createstatement();
  Resultset rs = st.executequary ("SELECT id, name
                   from student");
  while (rs.next){
     int id = rs.getInt ("int");
     String name = rs.getstring (" Name");
     System.out.printIn (id +"  " + name);
    }
   con.close();
  } catch (Exception e){
     System.out.printIn (" Error occerred");
  }
 }
}
```

Lab Report : 08

Lab Title : Development of RESTful Web Services using Spring Boot

Objective: The objective of this lab is to understand how spring boot simplifies the development of RESTful web services and to implement a REST Controller using @RestController, @GetMapping and @postMapping annotations with JSON data handling.

Tool and Technologies used:

1. Java (JDK)
2. Spring boot Framework
3. Spring web Dependency
4. Embedded Apache Tomcat server
5. JSON (JavaScript object Notation)
6. IDE (IntelliJ IDEA)

Theory: Spring boot greatly simplifies the development of Restful services using

i) Auto configuration : Automatically configures web servers and JSON converters with minimal setup

ii) Embedded server : No need to deploy WARs to and external server. Just run your app like a java application.

iii) Spring web starter : Includes all required dependencies for building REST API's.

iv) Reduced Boilerplate : Annotations like @RESTControlle @GetMapping @postMapping etc. simplify request handling.

Description of Annotations :

@RESTController

i). Used to create RESTful web services

ii). Combines @controller and @ResponseBody

iii) Automatically returns data in JSON format

@GetMapping

i) Handles HTTP Get requests

ii) Used retrives data from the server.

@PostMapping
i) Handles HTTP post requests
ii) used to sent data to the server

<u>Implementation:</u>

<u>step 1: Create model class</u>

```
public class student {
    private int id;
    private string name;

   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 ;
   }
}
```

Step 2: Create 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<>();

    @GetMapping
    public List<student>getAllStudents (){
        return students;
    }
    @PostMapping
    public student addStudent (@RequestBody Student student){
        students.add (student);
        return student;
    }
}
```

Sample JSON Requests and Responses

Post Request (Add student)

```
{
 "id": 1,
 "name"; "Rahim"
}
```

Get Response (List of Students)

```
[
  {
   "id": 1,
   "name": "Rahim"
  }
 ]
```

Result: The RESTful web service was successfully developed using Spring Boot. The application handled GET and POST requests correctly and JSON data was automatically converted to and from Java objects.

Conclusion: Spring Boot simplifies RESTful web service development through auto-configuration embedded servers and annotation-based programming. Using @RestController, @GetMapping and @PostMapping. REST APIs can be created efficiently with automatic JSON handling.

Lab Report - 09

I developed a Hotel Booking Management system using java swing for the Grcaphical user interface and MySQL for database management.

Graphical User Interface (GUI) Description:

Main window

```
public class HotelBooking extends JFrame
```

Input section (Top panel)

```
idField = new JTextField ();
nameField = new JTextField();
nightsField = new JTextField ();
roomTypeBox = new JcomboBox<><new string[]{
    "single($500)","Double($1000)","Deluxe ($2000)"
   });
```

Button Section (Middle Panel)

```
JButton b1 = new JButton ("Book");
JButton b2 = new JButton ("update");
JButton b3 = new JButton ("cancel");
```

Display section (Botton panel)

```
displayArea = new JTextArea ();
```

Importent code explanation :

Database connection (JDBC)

```
Connection c = DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/hotel_db",
     "root",
    "Tajul@51"
  );
```

Booking,

INSERT, UPDATE and DELETE operation

```
b1.addActionListener(e->
      runSQL("INSERT INTO booking (name, room_type, nights
                                        VALUES(?,?,?)")
      );
b2.addActionListener(e->
      runSQL("UPDATE booking SET name = ?, room_type=
                            nights = ? WHERE id = ? ")
      );
b3.addActionListener(e->
      runSQL("DELETE FROM booking WHERE id = ? ")
      );
```

PreparedStatement Usage

```
PreparedStatement ps = c.prepareStatement(sql);
```

Room cost calculation

```
int p= roomType.equals("single") ? 500
        : roomType.equals ("Double.")? 1000 : 2000;
```

Total fee = Room Price x Number of Nights

Displaying Records

```
displayArea.append(
    id +"\t" +name + "\t" + type+ "\t" +nights+"\t"+
                          total + "\n"
    );
```

Error Handing

```
JOptionPane.showMessageDialog(this, "Error message");
```

SampleDemonstraction (Input → output)

Input

~~Napnet~~ : Tajul
Room type: single
Nights : 3

output:

```
ID   Name   Type    Nights  Fee
1    Tajul  single  3       1500
```