6 Lab 1

key differences for multiple Inheritance

Abstract classes:

- A class can only extend ONE abstract class
- Can have both abstract and concrete methods
- Can haave instance variables with any access modifier
- Can have constructors.
- Methods can have any access modifier (public, private, protected).

Interfaces:

- A class can implement MULTIPLE interfaces
- All methods are abstract by default
- Can only have constants
- Cannot have constructors

- All methods are implicitly public

When to Use Abstract Classes

Use an abstract class when:

1. When You need to share code among closely related classes: For example, if you have animal as an abstract class with a concrete method breathe(), all subclasses (Dog, Cat, Bird) can inherit and reuse with this.

2. You need non-public members - If you want protected or private fields/methods that subclasses can access.

3. You want to provide common functionality with some flexibility - when classes share a common base but need to implement specific behaviours differently.

Example: A 'vehicle' abstract class with concrete methods like 'startEngine()' and abstract methods like 'drive()'.

When to Use Interfaces

use an interface when:

1. You need multiple inheritance: A class can implement multiple interfaces. For example, a 'smartphone' class can implement both 'Camera' and 'MusicPlayer' interfaces.

2. Unrelated classes should implement the same methods – when you want completely different classes to guarantee they have certain behaviours. Example: Both 'Bird' and 'Airplance' can implement a 'Flyable' interface.

3. You want to specify a contract without implementation — when you just want to define what methods a class must have, without dictating how.

Example: Comparable, Sercializable, Runnable

— these can be implemented by any class regardless of its inheritance hierarchy.

Practical Example

Scenario : Animal Kingdom

Abstract Class Approch :

— Animal (adstract class) with method eat()

— Dog extends Animal

— Cat extends Animal

Problem : Dog cannot also extend Robot !

Interface approach:

- Walkable (interface) with method walk()
- Swimmable (interface) with method swim()
- Duck implements both Walkable AND swimmable
- Amphibious Robot implements both Walkable AND swimmable

This shows interfaces enable multiple inheritance!

Summary

Chose abstract classes when you have an "is-a" relationship and want to share code among related classes. Choose interfaces when you need multiple inheritance, want to define capabilities that unpack classes can share, or

need to achieve pt polymorchism across different class interitances.

Lab 2

How Encap$ulation Ensure Data Security and Intergrity —

Data Security:

- private varciable hide data from direct external access.
- Only the class ifeslf can directly modify if its data
- Prevents unauthorized or accidental changes from outside code.

Data Integrity:

- Validated setter methods ensure only valid data is stored.
- Business rules are enforced before data is changed.
- Invalid data is rejected, keeping the object in a valid state.

BankAccount Class Example:

```
class BankAccount {

    private string accountNumber;
    private double balance;
    public void setAccountNumber(String accNo)
    { if (accNo!=null && !accNo.isEmpty())}
        accountNumber=accNo;
    }
}
```

```
public void setInitialBalance(double amount){
    if(amount>=0){
        balance=amount;
    }
  }
public String getAccountNumber(){
    return accountNumber;
}
public double getBalance(){
    return balance;
  }
}
```

Key Points

1. Data Security Through Encapsulation:

- Private variables prevent direct access:

accountNumber="hack" wont work.

- External Code must use setter methods, which enforces rules
- This protects sensitive banking data from unauthorized modification.

2. Data Integrity Through Validation :

- Null check : Prevents NullpointerException and ensures data exits.
- Empty check : Ensures meaning data is Stored.
- Negative check : Enforces as business rule
- Format validation : Ensures account Number follows required pattern.

3. Benifits :

- Invalid data never enters the object
- object always remains in a valid state

- Changes are controlled and logged.
- Easy to add more validation rules in one place

4. Without Encapsulation:

- If variables were public, someone could do:

```
account.accountNumber = null;
account.initialBalance = -9999;
```

Conclusion:

Encapsulation ensures data security by hiding internal data (private variable) and ensures data integrity by validating all input through setter methods before allowing changing. This prevents invalid state and protects sensitive information in the BankAccount.

Lab 3

Aim

To design and implement a multithreaded car parking management system where multiple cars request parking concurrently and parking agents process requests using a shared synchronized queue.

Problem Description:

Multiple cars arrive simultaneously and request parking. Parking agents (threads) continuously monitor a shared parking pool and park cars in the order of Arrival, ensuring thread safe access.

Class Description

1. RegistrationParking

Represents a parking request made by a car

2. Parking Pool

A shared synchronized queue that stores parking requests.

3. Parking Agent

A thread that retrieves parking requests from the pool and parks cars.

4. MainClass

Simulates multiple cars arriving concurrently.

Algorithm:

1. Create a syncronized parking pool
2. Generate Multiple parking requests concurrently.
3. Start multiple parking agent threads.
4. Agents continuously fetch and process requests from the pool.
5. Display parking status.

Java Implementation:

```
import java.util.*;
class RegistrarParking {
    String carNO;
    RegistrarParking(String carNO){this.carNo=carNo;}
}
Class ParkingPool{
    Queue<RegistrarParking> queue=new list();
```

```
Synchronized void addRequest(RegistrarParking){
    queue.add(r);
    notify()
}
synchronized RegistrarParking getRequest()
                        throws Interrupted (){
    while(queue.isEmpty()) wait();
    return queue.poll();
}
}
class parkingAgent extends Thread {
    parkingPool pool;
    int agentId;
    parkingAgent(parkingPool pool, int id){
    this.pool = pool;
    this.agentId = id;
}
    public void run(){
    try{
```

```
        RgistrarParking r=pool.getRequest();
        system.out.println("Agent"+agentId + "parked
    car" + r.carNo + ".");
    } catch(Exception e){}
}
}
public. class MainClass{
    public static void main(String[] args){
        parking pool pool = new parkingPool();

    System.out.println("Car ABC123 requested
                        parking.");
    pool.add Requested parking.");
    System.out.println("Car XYZ126 requested
                        parking.");
    pool.addRequest(new ResistrarParking("
                        XYZ126"));
```

```
        new parkingAgent(pool, 1).start();
        new parkingAgent(pool, 2).start();
    }
}
```

Sample Output:

```
Car ABC123 requested parking.
Car XYZ456 requested parking.
Agent 1 parked car ABC123.
Agent 2 parked car XYZ456.
```

Conclusion:

The system successfully demonstrates multiplethreading and synchronization using shared resources. Parking agents safely process concurrent parking requests without conflict.

Lab 4

Aim

To study how JDBC manages communication between a Java application and a relational database and to execute a SELECT query with proper exception handling.

Steps to Execute a 'SELECT' query using JDBC:

1. Import JDBC packages.
2. Load and register the JDBC driver
3. Establish a database connection.
4. Create a statement or prepareStatement.
5. Execute the SELECT query.
6. Process the Resultset.
7. Close all resources.

Java Code :

```
import java.sql.*;

public class JDBCselect {
    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", "user",
            "password");
        stmt = con.createStatement();
        rs = stmt.executeQuery("SELECT id,
          name FROM student");
        while (rs.next()) {
            System.out.println(rs.getInt("id") +
```

```
            " " + rs.getString("name"));
        }
    }

    catch (Exception e){
      System.out.println("Error: "+e.getMessage());
    }
    finally{
      try{
          if (rs! = null) rs.close();
          if(stmt! =null) stmt.close();
          if(con! =null) con.close();
      }
      catch (SQLException e){
      system.out.println("Resource closing error");
      }
    }
  }
}
```

Error Handling Explanation:

- try block: Contains database connection and query execution logic.
- catch block: Handles SQL and runtime exceptions
- finally block: Ensures database resources are closed to prevent memory leaks.

Conclusion:

JPBC provides a reliable and standerdized method for java applications to communicate with relational databases. Proper exception handling ensures robustness and safe resource management.

Lab 5

Aim

To understan how a servlet controller manages the flow between the model and the view in a ~~Jasa~~ Java EE application and to demonstrate data forwarding from a servlet to a JSP

Servlet Controller Role:

In a Java EE application following the MVC architecture:

- Model : Contains business logic and data
- View : JSP renders the response to the client
- Controller (Servlet): Handles client requests, interacts with the model and forwards data to the view.

The servlet ast as the central controller, ensuring separation of concerns and controlled navigation.

Flow of Execution:

1. Client sends a request to the servlet.
2. Servlet ~~po~~ processes the request and interacts with the model.
3. Servlet stores data in request scope.
4. Request is forwarded to a JSP
5. JSP reads the & response using the forwarded data.

Example Code:

Servlet (controller) -

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class StudentServlet extends HttpServlet {
```

```
protected void doget(HttpServletRequest req,
HttpServletResponse res)
    throws ServletException, IOException {

String Name = "John Doe";
req.setAttribute("StudentName", name);

RequestDispatcher rd = req.getRequstDispatcher
("Student.jsp");
rd.forward(req, res);
}
}
```

JSP ~~to~~ view :

```
<html>
<body>
<h3> Student Name: </h3>
<P> ${ Student Name } </P>
</body>
</htm>
```

Explanation:

- The servlet receives the request and pe prepares the data.
- Data is stored using 'setAttribute()'
- RequestDispatcher.forward() sends the request the JSP.
- JSP accesses the data using Expression Langua and renders the output.

Conclusion:

The servlet controller efficiency manages request flow by coordinating between the model and the view, enabling clean separat of Logic and presentation.

Lab 6 ..

performance and security improvement using prepared Statement.

Introduction:

In JDBC, SQL Statements can be executed using either Statement or preparedStatement while Statement is used for simple SQL queries, preparedStatement is a more advanced and secure option that allow parameterized queries. It is widely used in real-world java application.

PreparedStatement vs Statement

performance Improvement

preparedStatement is pre-compiled by the database only once. When the same query is executed multiple times with different values. the database does not need to recompile the query, which improves performance.

Security Improvement

preparedStatement helps prevent SQL injection attacks because user input is treated as data not as part of the SQL command. This makes applications more secure compared using Statement.

Advantages of prepared Statement

- Faster execution for repeated queries.
- prevents SQL injection attacks
- Supports parameterized queries.
- Improves code readability and maintainability

Example

```
import java.sql.*;
public Class InsertExample{
    public static void main(String[]args){
Connection con = null;
preparedStatement ps = null;
try{
Class.ForName("com.mysql.cj.jdbc.Driver");
```

```
con = DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/student",
    "root"
    "password"
);

String sql = "Insert into info(id, name) Valuen(?,?)";

ps = con.prepareStatement (sql);

ps.setInt (1, 101);
ps.setString (2, "Akib");

ps.executeUpdate ();
system.out.println ("Record inserted Successfully");
}
catch (Exeption e){
    system.out.println ("Error occured: "+e);
}finaly{
    try{ if (ps != null) ps.close();
         if (con != null) con.close();
```

```
} catch (SQLException e){
    System.out.println("Error closing
                        resource");
}
}
}
```

Lab 7

What is a ResultSet in JDBC and how is it used to retrive data from a MySQL database?

In JDBC a ResultSet is essentially a table of data representing a database result set. It is generated by executing a statement that queries the database

Core Methods Explained.

. next() : The method moves the cursor forword one row from its current position. It return a boolean : "true" if there is a valid row to read and false if there are no more rows.

- getString(String ColumnLabel):

Retrieves the value of the designated Colom in the current row as a String. You can also use the Colum index

- getInt(String ColumnLabel):

Similar to getString, but it retrieve the value as an int. This is used for numeric data like IDs or ages.

Code Example

```
ResultSet rs = stmt.executeQuery("select id, name
                                  from student");

while(rs.next()){
    int id = rs.getInt("id");
    String name = rs.getString("name");

    System.out.println("ID: "+id+" Name: "+name);
```

Lab 8

How Spring Boot simplifies RESTful web service Development:

Springboot Simplifies the development of RESTful services by providing :

- Auto-configuration: which reduces manual setup and configuration.
- Embedded servers, eliminating the need for external deployment.
- Annotation-based programming, making REST APIs easier to develop and maintain.
- Automatic JSON conversion using the Jackson library.

Implementing a REST controller in Spring Boot :

A REST controller handles HTTP requests and returns data in JSON format. Spring Boot automatically converts Java objects to JSON and vice versa.

Example: (simple controller with JSON)

```
import org.springframework.web.bind.annotation.*;
@RestController
@RequestMapping("/student")
public class StudentController {

    @getMapping("/{id}")
    public Stdent getstudent(@PathVariable int id) {
    return new student(id, "Rahman", 21); }
```

```
    @PostMapping ("/add");
    public Student addStudent (@RequestBody
    Student student) {
        return student;
    }
}
public class Student {
    private int id;
    private String name;
    private int age;

    public Student (int id, String name,
    int age) {
        this.id= id;
        this.name = name;
        this.age = age;
    }
}
```

Lab 9

Title:

Development of a simple graphical Calculator using Python and Tkinter

Objective:

To design and implement a basic calculator application with a graphical user interface that performs addition, subtraction, multiplication and division.

Tools and Technologies Used:

- Programming Language: Python 3
- GUI Library: Tkinter
- Development Environment: Any python IDE or text editor

Project Description:

The application provides:

- A display field for input and output
- Numeric buttons (0-9) and decimal point
- Arithmatic operator buttons (+, −, ÷, ×)
- Equals button (=) to evaluate the expression
- clear button (C) to reset the display.

The expression is built as a string and evaluated using python's eval() function when the equals button is pressed.

Important Code Snippets :

1. Important window setup —

```
from tkinter import *
root = Tk()
root.title = ("simple Calculator")
root.geometry ("320 x 420")
root.resizable (False, False)
```

2. Display and Global Expression —

```
expression = " "
display = StringVar()
display_field = Entry (root, font=("arial", 18),
                       textvariable=display, bd=10,
                       insertwidth=4, width=14,
                       borderwidth=4, justify='right')
display_field.grid=(row=0, column=0, columnspan
                    =4, padx=10, pady=10)
```

3. Button Click Functions :-

```
def press(num):
    global expression
    expression += str(num)
    display.set(experssion)
def equal():
    global expression
    try:
        result=str(eval(expression))
        display.set(result)
        expression=result
    except:
        display.set("Error")
        expression = ""
def clear():
    global expression
    expression = ""
    display.set("")
```

4. GUI Button Layout —

```
#Row 1
Button(root, text='C', font=('arial', 14),
       commander=clear, width=5).grid(
       row=1, column=0, padx=5, pady=5)
Button(root, text='/', font('arial', 14),
       comman=lamba : press('/'), width=5).
       grid#(row=1, colemn=3, padx=5, pady=5)
#Row 2-4 (numbers and operators)
buttons=[
  ('7', 2, 0), ('8', 2, 1), ('9') 2, 2), ('*', 2, 3),
  ('4', 3, 0), ('5', 3, 1), ('6', 3, 2), ('-', 3, 3),
  ('1', 4, 0), ('2', 4, 1), ('3', 4, 2), ('+', 4, 3), ]
for (text, row, col) in buttons:
  Button(root, text=text, font=('arial', 14),
    command=lamba t=text: press(t); width=5)
```

```
.grid(row=row, column=col, padx=5,
    pady=5)

#Row 5
Button(root, text='0', font=('arcial', 14), Com
    command=lamba: press('0'), width=12)
    .grid(row=15, column=0, columnspan=2,
    padx=5, pady=5)
Button(root, text='.', font('arcial, 14),
    command=lamba: press('.'), width=5)
    .grid(row=5, column=2, padx=5, pady=5)
Button(root, text='=', font=('arcial', 14), command=
    equal, width=5, bg='lightgreen').grid(
    row=5, column=3, padx=5, pady=5)

root.main.loop()
```

Graphical User Interface Description

The Interface consists of:

- A single line entry widget at the top displaying the current expression and result.
- Four rows of buttons arranged in a grid layout (4 columns).
- Clear buttons (C) is red/orange shade.
- Equals button highlighted for visual emphasis.