Q. Write a Java program that uses polymorphism by defining an interface called Shape

with methods to calculate the area and perimeter of a shape. Then create classes

that implement the Shape interface for different types of shapes, such as circles and

triangles.

Certainly! Here's an example Java program that demonstrates polymorphism using an interface called Shape with methods to calculate the area and perimeter. The program includes classes for circles and triangles that implement the Shape interface:

// Shape interface

interface Shape {

double calculateArea();

double calculatePerimeter();

}

// Circle class implementing Shape interface

class Circle implements Shape {

private double radius;

public Circle(double radius) {

this.radius = radius;

}

@Override

public double calculateArea() {

return Math.PI \* radius \* radius;

}

@Override

public double calculatePerimeter() {

return 2 \* Math.PI \* radius;

}

}

// Triangle class implementing Shape interface

class Triangle implements Shape {

private double side1;

private double side2;

private double side3;

public Triangle(double side1, double side2, double side3) {

this.side1 = side1;

this.side2 = side2;

this.side3 = side3;

}

@Override

public double calculateArea() {

// Using Heron's formula to calculate the area of a triangle

double s = (side1 + side2 + side3) / 2;

return Math.sqrt(s \* (s - side1) \* (s - side2) \* (s - side3));

}

@Override

public double calculatePerimeter() {

return side1 + side2 + side3;

}

}

// Main class to test the polymorphism

public class ShapeTest {

public static void main(String[] args) {

// Creating objects of Circle and Triangle

Shape circle = new Circle(5);

Shape triangle = new Triangle(3, 4, 5);

// Using polymorphism to calculate area and perimeter

System.out.println("Circle Area: " + circle.calculateArea());

System.out.println("Circle Perimeter: " + circle.calculatePerimeter());

System.out.println("Triangle Area: " + triangle.calculateArea());

System.out.println("Triangle Perimeter: " + triangle.calculatePerimeter());

}

}

In this example, the **Shape** interface declares two methods (**calculateArea** and **calculatePerimeter**). The **Circle** and **Triangle** classes implement this interface and provide their own implementations of the methods. In the **ShapeTest** class, we demonstrate polymorphism by creating objects of type **Shape** that can reference instances of both **Circle** and **Triangle**. This allows us to call the methods defined in the **Shape** interface on these objects, regardless of their actual types.

Q. Write a Java program to invoke parent class constructor from a child class. Create

Child class object and parent class constructor must be invoked. Demonstrate by

writing a program. Also explain key points about Constructor.

Ans. In Java, a child class can invoke the constructor of its parent class using the **super()** keyword. Here's an example program demonstrating how to invoke a parent class constructor from a child class:

class Parent {

public Parent() {

System.out.println("Parent class constructor invoked");

}

}

class Child extends Parent {

public Child() {

super(); // Invoking parent class constructor

System.out.println("Child class constructor invoked");

}

}

public class ConstructorInvocationExample {

public static void main(String[] args) {

Child childObj = new Child();

}

}

In this example, we have a **Parent** class with a parameterless constructor, and a **Child** class that extends **Parent**. The **Child** class has its own constructor, which uses **super()** to invoke the constructor of the parent class. When you create an object of the **Child** class (**Child childObj = new Child();**), both the parent and child class constructors are invoked.

Key points about Constructors in Java:

1. **Constructor Naming**: The constructor name must be the same as the class name.
2. **Default Constructor**: If a class does not explicitly define any constructors, Java provides a default constructor with no parameters.
3. **Explicit Constructor Invocation**: A constructor can explicitly call another constructor in the same class using **this()** or call a constructor in the parent class using **super()**.
4. **Constructor Overloading**: Like methods, constructors can be overloaded by providing different sets of parameters.
5. **Initialization**: Constructors are used to initialize the object's state. Instance variables can be assigned values, and necessary setup can be performed within the constructor.
6. **No Return Type**: Constructors do not have a return type, not even **void**.
7. **Inheritance and Constructors**: When a child class is instantiated, the constructor of its immediate parent class is called first (using **super()**), followed by its own constructor.
8. **Chaining Constructors**: Constructors can call other constructors in the same class or the parent class, creating a chain of constructor calls.

Remember that if you don't explicitly provide a call to **super()** in the child class constructor, Java automatically inserts a call to the default (parameterless) constructor of the parent class. If the parent class does not have a default constructor and you want to call a specific constructor, you need to use **super()** with appropriate argument.

**Write a Java programme that takes an integer from the user and throws an exception**

**if it is negative.Demonstrate Exception handling of same program as solution.**

Certainly! Below is a simple Java program that takes an integer from the user and throws an exception if it is negative. It demonstrates exception handling to handle the negative input:

import java.util.Scanner;

public class NegativeNumberExceptionDemo {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

try {

System.out.print("Enter a positive integer: ");

int userInput = scanner.nextInt();

if (userInput < 0) {

throw new NegativeNumberException("Negative numbers are not allowed");

}

System.out.println("Entered number is: " + userInput);

} catch (NegativeNumberException e) {

System.err.println("Exception: " + e.getMessage());

} catch (Exception e) {

System.err.println("Invalid input. Please enter a valid integer.");

} finally {

scanner.close();

}

}

}

class NegativeNumberException extends Exception {

public NegativeNumberException(String message) {

super(message);

}

}

This program uses a custom exception **NegativeNumberException** that extends the **Exception** class. If the user enters a negative number, it throws this custom exception, and the catch block for **NegativeNumberException** handles the exception, displaying an error message. The program also includes a general catch block to handle other types of exceptions, such as **InputMismatchException** in case the user enters a non-integer value. The **finally** block ensures that the **Scanner** is closed, regardless of whether an exception occurred or not.

Create a Java program that simulates a bank account. The program should allow

users to deposit and withdraw money, check their balance.

Certainly! Below is a simple Java program that simulates a bank account. It allows users to deposit and withdraw money, as well as check their balance:

import java.util.Scanner;

public class BankAccountSimulation {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

// Creating a BankAccount object

BankAccount account = new BankAccount();

int choice;

do {

System.out.println("1. Deposit");

System.out.println("2. Withdraw");

System.out.println("3. Check Balance");

System.out.println("4. Exit");

System.out.print("Enter your choice: ");

choice = scanner.nextInt();

switch (choice) {

case 1:

System.out.print("Enter the amount to deposit: ");

double depositAmount = scanner.nextDouble();

account.deposit(depositAmount);

break;

case 2:

System.out.print("Enter the amount to withdraw: ");

double withdrawAmount = scanner.nextDouble();

account.withdraw(withdrawAmount);

break;

case 3:

System.out.println("Current Balance: " + account.getBalance());

break;

case 4:

System.out.println("Exiting the program. Thank you!");

break;

default:

System.out.println("Invalid choice. Please enter a valid option.");

}

} while (choice != 4);

scanner.close();

}

}

class BankAccount {

private double balance;

public void deposit(double amount) {

if (amount > 0) {

balance += amount;

System.out.println("Deposit successful. Current Balance: " + balance);

} else {

System.out.println("Invalid deposit amount. Please enter a positive value.");

}

}

public void withdraw(double amount) {

if (amount > 0 && amount <= balance) {

balance -= amount;

System.out.println("Withdrawal successful. Current Balance: " + balance);

} else {

System.out.println("Invalid withdrawal amount or insufficient funds.");

}

}

public double getBalance() {

return balance;

}

}

This program uses a **BankAccount** class to represent a bank account. Users can interact with the account by choosing options for deposit, withdrawal, and checking the balance. The program uses a loop to repeatedly prompt the user for input until they choose to exit.

Demonstrate the difference between abstract class and interface by writing program as well as in keypoints.

Certainly! Let's first discuss the key differences between abstract classes and interfaces, and then I'll provide examples in Java to illustrate these differences.

### Key Differences:

**1. Definition:**

* **Abstract Class:** It is a class that can have both abstract (unimplemented) and concrete (implemented) methods. It may also have instance variables. You can use the **abstract** keyword to declare an abstract class.
* **Interface:** It is a collection of abstract methods. In Java, all methods declared in an interface are implicitly abstract. Interfaces can't have instance variables, and all variables are implicitly public, static, and final.

**2. Multiple Inheritance:**

* **Abstract Class:** Supports single inheritance in Java. A class can extend only one abstract class.
* **Interface:** Supports multiple inheritance. A class can implement multiple interfaces.

**3. Constructor:**

* **Abstract Class:** Can have constructors.
* **Interface:** Cannot have constructors.

**4. Access Modifiers:**

* **Abstract Class:** Can have access modifiers (public, private, protected) for methods and variables.
* **Interface:** All methods are implicitly public. Variables are public, static, and final.

**5. Fields (Variables):**

* **Abstract Class:** Can have instance variables (fields).
* **Interface:** Cannot have instance variables. Only constant variables are allowed.

### Examples in Java:

#### **Abstract Class Example:**

javaCopy code

abstract class Shape

{

String color; Shape(String color)

{

this.color = color;

}

abstract double calculateArea();

}

class Circle extends Shape

{

double radius; Circle(String color, double radius)

{

super(color); this.radius = radius;

}

@Override double calculateArea()

{

return Math.PI \* radius \* radius;

}

}

public class AbstractClassExample

{

public static void main(String[] args) {

Circle circle = new Circle("Red", 5.0);

System.out.println("Area of Circle: " + circle.calculateArea()); } }

#### **Interface Example:**

javaCopy code

interface Shape {

String color = "Red";

// Implicitly public, static, and final double calculateArea();

// Implicitly abstract

}

class Circle implements Shape {

double radius; Circle(double radius) {

this.radius = radius;

}

@Override public double calculateArea() {

return Math.PI \* radius \* radius;

}

}

public class InterfaceExample

{

public static void main(String[] args) {

Circle circle = new Circle(5.0);

System.out.println("Area of Circle: " + circle.calculateArea());

}

}

These examples demonstrate the key differences between abstract classes and interfaces in terms of structure, inheritance, and usage.

Write a Java program that uses stream api to perform operations on a large data set, such as sorting or filtering the data.

javaCopy code

import java.util.Arrays;

import java.util.List;

class Employee {

private String name;

private int age;

private double salary;

public Employee(String name, int age, double salary) {

this.name = name;

this.age = age;

this.salary = salary;

}

public String getName() {

return name;

}

public int getAge() {

return age;

}

public double getSalary() {

return salary;

}

@Override

public String toString() {

return "Employee{" +

"name='" + name + '\'' +

", age=" + age +

", salary=" + salary +

'}';

}

}

public class StreamApiExample {

public static void main(String[] args) {

// Create a list of employees

List<Employee> employees = Arrays.asList(

new Employee("John", 30, 50000.0),

new Employee("Alice", 25, 60000.0),

new Employee("Bob", 35, 75000.0),

new Employee("Eva", 28, 55000.0),

new Employee("Mike", 40, 80000.0)

);

// Use Stream API to perform operations on the data set

// Filtering employees younger than 30

System.out.println("Employees younger than 30:");

employees.stream()

.filter(e -> e.getAge() < 30)

.forEach(System.out::println);

// Sorting employees by salary in ascending order

System.out.println("\nEmployees sorted by salary (ascending):");

employees.stream()

.sorted((e1, e2) -> Double.compare(e1.getSalary(), e2.getSalary()))

.forEach(System.out::println);

// Mapping employee names to uppercase

System.out.println("\nEmployee names in uppercase:");

employees.stream()

.map(Employee::getName)

.map(String::toUpperCase)

.forEach(System.out::println);

}

}

Create a Java program that implements a binary search algorithm. The program should accept user input for the target value and search for it in a sorted array. The program should return the index of the target value if found or a message if not found.

import java.util.Arrays;

import java.util.Scanner;

public class BinarySearchExample {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

// Sorted array

int[] sortedArray = {1, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20};

// Prompt user for the target value

System.out.print("Enter the target value to search: ");

int target = scanner.nextInt();

// Perform binary search

int result = binarySearch(sortedArray, target);

// Display the result

if (result != -1) {

System.out.println("Target value found at index: " + result);

} else {

System.out.println("Target value not found in the array.");

}

scanner.close();

}

// Binary search algorithm

private static int binarySearch(int[] array, int target) {

int low = 0;

int high = array.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

int midValue = array[mid];

if (midValue == target) {

return mid; // Target found

} else if (midValue < target) {

low = mid + 1; // Discard the left half

} else {

high = mid - 1; // Discard the right half

}

}

return -1; // Target not found

}

}

Write a Java program that creates two threads. The first thread should print even numbers between 1 and 10, and the second thread should print odd numbers between 1 and 10.

import java.util.Arrays;

import java.util.Scanner;

public class BinarySearchExample {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

// Sorted array

int[] sortedArray = {1, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20};

// Prompt user for the target value

System.out.print("Enter the target value to search: ");

int target = scanner.nextInt();

// Perform binary search

int result = binarySearch(sortedArray, target);

// Display the result

if (result != -1) {

System.out.println("Target value found at index: " + result);

} else {

System.out.println("Target value not found in the array.");

}

scanner.close();

}

// Binary search algorithm

private static int binarySearch(int[] array, int target) {

int low = 0;

int high = array.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

int midValue = array[mid];

if (midValue == target) {

return mid; // Target found

} else if (midValue < target) {

low = mid + 1; // Discard the left half

} else {

high = mid - 1; // Discard the right half

}

}

return -1; // Target not found

}

}

Write a Java program that implements a producer-consumer model using multithreading. The program should have a producer thread that generates random numbers and adds them to a queue, and a consumer thread that reads numbers from the queue and calculates their sum. The program should use synchronization to ensure that the queue is accessed by only one thread at a time.

import java.util.Arrays;

import java.util.Scanner;

public class BinarySearchExample {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

// Sorted array

int[] sortedArray = {1, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20};

// Prompt user for the target value

System.out.print("Enter the target value to search: ");

int target = scanner.nextInt();

// Perform binary search

int result = binarySearch(sortedArray, target);

// Display the result

if (result != -1) {

System.out.println("Target value found at index: " + result);

} else {

System.out.println("Target value not found in the array.");

}

scanner.close();

}

// Binary search algorithm

private static int binarySearch(int[] array, int target) {

int low = 0;

int high = array.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

int midValue = array[mid];

if (midValue == target) {

return mid; // Target found

} else if (midValue < target) {

low = mid + 1; // Discard the left half

} else {

high = mid - 1; // Discard the right half

}

}

return -1; // Target not found

}

}

Write a Java program that reads a set of integers from the user and stores them in a List. The program should then find the second largest and second smallest elements in the List.

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

import java.util.Scanner;

public class SecondLargestSmallestExample {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

// Read a set of integers from the user and store them in a List

List<Integer> numbers = new ArrayList<>();

System.out.print("Enter the number of elements: ");

int n = scanner.nextInt();

System.out.println("Enter the elements:");

for (int i = 0; i < n; i++) {

int num = scanner.nextInt();

numbers.add(num);

}

// Find the second largest and second smallest elements

if (numbers.size() < 2) {

System.out.println("List should have at least two elements to find second largest and second smallest.");

} else {

int secondLargest = findSecondLargest(numbers);

int secondSmallest = findSecondSmallest(numbers);

System.out.println("Second Largest Element: " + secondLargest);

System.out.println("Second Smallest Element: " + secondSmallest);

}

scanner.close();

}

// Method to find the second largest element

private static int findSecondLargest(List<Integer> numbers) {

Collections.sort(numbers, Collections.reverse

Write a Java program that connects to a MySQL database using JDBC. The program should read data from a table and display the results in the console.

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

public class MySQLJDBCExample {

// JDBC URL, username, and password of MySQL server

private static final String JDBC\_URL = "jdbc:mysql://localhost:3306/your\_database\_name";

private static final String USERNAME = "your\_username";

private static final String PASSWORD = "your\_password";

public static void main(String[] args) {

try {

// Load the MySQL JDBC driver

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

// Establish a connection to the MySQL database

try (Connection connection = DriverManager.getConnection(JDBC\_URL, USERNAME, PASSWORD)) {

System.out.println("Connected to the database");

// Create a statement object for executing SQL queries

try (Statement statement = connection.createStatement()) {

// Define the SQL query to retrieve data from a table

String query = "SELECT \* FROM your\_table\_name";

// Execute the query and get the result set

try (ResultSet resultSet = statement.executeQuery(query)) {

// Process the result set and display the data in the console

while (resultSet.next()) {

int id = resultSet.getInt("id");

String name = resultSet.getString("name");

// Add more columns as needed

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

// Print other columns as needed

}

}

}

}

} catch (ClassNotFoundException | SQLException e) {

e.printStackTrace();

}

}

}

12. Write a Java program that uses JDBC to implement a simple CRUD (create, read,update, delete) application. The program should allow users to add, view, update,and delete records in a MySQL database table.

import java.sql.\*;

import java.util.Scanner;

public class CRUDApplication {

private static final String JDBC\_URL = "jdbc:mysql://localhost:3306/your\_database\_name";

private static final String USERNAME = "your\_username";

private static final String PASSWORD = "your\_password";

public static void main(String[] args) {

try {

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

} catch (ClassNotFoundException e) {

e.printStackTrace();

return;

}

try (Connection connection = DriverManager.getConnection(JDBC\_URL, USERNAME, PASSWORD)) {

System.out.println("Connected to the database");

while (true) {

System.out.println("1. Add Record");

System.out.println("2. View Records");

System.out.println("3. Update Record");

System.out.println("4. Delete Record");

System.out.println("5. Exit");

System.out.print("Enter your choice: ");

Scanner scanner = new Scanner(System.in);

int choice = scanner.nextInt();

switch (choice) {

case 1:

addRecord(connection);

break;

case 2:

viewRecords(connection);

break;

case 3:

updateRecord(connection);

break;

case 4:

deleteRecord(connection);

break;

case 5:

System.out.println("Exiting the program");

return;

default:

System.out.println("Invalid choice. Please enter a number between 1 and 5.");

}

}

} catch (SQLException e) {

e.printStackTrace();

}

}

private static void addRecord(Connection connection) throws SQLException {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter name: ");

String name = scanner.nextLine();

System.out.print("Enter age: ");

int age = scanner.nextInt();

System.out.print("Enter salary: ");

double salary = scanner.nextDouble();

String insertQuery = "INSERT INTO employees (name, age, salary) VALUES (?, ?, ?)";

try (PreparedStatement preparedStatement = connection.prepareStatement(insertQuery)) {

preparedStatement.setString(1, name);

preparedStatement.setInt(2, age);

preparedStatement.setDouble(3, salary);

int affectedRows = preparedStatement.executeUpdate();

if (affectedRows > 0) {

System.out.println("Record added successfully");

} else {

System.out.println("Failed to add record");

}

}

}

private static void viewRecords(Connection connection) throws SQLException {

String selectQuery = "SELECT \* FROM employees";

try (Statement statement = connection.createStatement();

ResultSet resultSet = statement.executeQuery(selectQuery)) {

while (resultSet.next()) {

int id = resultSet.getInt("id");

String name = resultSet.getString("name");

int age = resultSet.getInt("age");

double salary = resultSet.getDouble("salary");

System.out.println("ID: " + id + ", Name: " + name + ", Age: " + age + ", Salary: " + salary);

}

}

}

private static void updateRecord(Connection connection) throws SQLException {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter the ID of the record to update: ");

int id = scanner.nextInt();

scanner.nextLine(); // consume the newline character

System.out.print("Enter new name: ");

String newName = scanner.nextLine();

System.out.print("Enter new age: ");

int newAge = scanner.nextInt();

System.out.print("Enter new salary: ");

double newSalary = scanner.nextDouble();

String updateQuery = "UPDATE employees SET name = ?, age = ?, salary = ? WHERE id = ?";

try (PreparedStatement preparedStatement = connection.prepareStatement(updateQuery)) {

preparedStatement.setString(1, newName);

preparedStatement.setInt(2, newAge);

preparedStatement.setDouble(3, newSalary);

preparedStatement.setInt(4, id);

int affectedRows = preparedStatement.executeUpdate();

if (affectedRows > 0) {

System.out.println("Record updated successfully");

} else {

System.out.println("No record found with the given ID");

}

}

}

private static void deleteRecord(Connection connection) throws SQLException {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter the ID of the record to delete: ");

int id = scanner.nextInt();

String deleteQuery = "DELETE FROM employees WHERE id = ?";

try (PreparedStatement preparedStatement = connection.prepareStatement(deleteQuery)) {

preparedStatement.setInt(1, id);

int affectedRows = preparedStatement.executeUpdate();

if (affectedRows > 0) {

System.out.println("Record deleted successfully");

} else {

System.out.println("No record found with the given ID");

}

}

}

}

Create a Java program that connects to a PostgreSQL database and executes a batch update. The program should read the input data from a file and insert it into the database using JDBC batch updates.

import java.io.BufferedReader;

import java.io.FileReader;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.SQLException;

public class PostgreSQLBatchUpdate {

private static final String JDBC\_URL = "jdbc:postgresql://localhost:5432/your\_database\_name";

private static final String USERNAME = "your\_username";

private static final String PASSWORD = "your\_password";

public static void main(String[] args) {

try {

Class.forName("org.postgresql.Driver");

} catch (ClassNotFoundException e) {

e.printStackTrace();

return;

}

try (Connection connection = DriverManager.getConnection(JDBC\_URL, USERNAME, PASSWORD)) {

System.out.println("Connected to the database");

// Specify the path to your input data file

String inputFile = "path/to/your/inputfile.txt";

batchInsertData(connection, inputFile);

} catch (SQLException e) {

e.printStackTrace();

}

}

private static void batchInsertData(Connection connection, String inputFile) throws SQLException {

String insertQuery = "INSERT INTO your\_table\_name (column1, column2, column3) VALUES (?, ?, ?)";

try (PreparedStatement preparedStatement = connection.prepareStatement(insertQuery)) {

// Read data from the file and add to the batch

try (BufferedReader reader = new BufferedReader(new FileReader(inputFile))) {

String line;

while ((line = reader.readLine()) != null) {

String[] data = line.split(","); // assuming data is comma-separated

preparedStatement.setString(1, data[0]); // replace with the appropriate column index and type

preparedStatement.setString(2, data[1]); // replace with the appropriate column index and type

preparedStatement.setString(3, data[2]); // replace with the appropriate column index and type

preparedStatement.addBatch();

}

}

// Execute the batch update

int[] affectedRows = preparedStatement.executeBatch();

System.out.println("Batch update completed. Total affected rows: " + getTotalAffectedRows(affectedRows));

}

}

private static int getTotalAffectedRows(int[] affectedRows) {

int total = 0;

for (int rows : affectedRows) {

total += rows;

}

return total;

}

}

Create a Java servlet that reads the name of the user from a form and displays a welcome message on the web page. The servlet should use the GET method to read the input data from the user.

import java.io.IOException;

import java.io.PrintWriter;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

@WebServlet("/WelcomeServlet")

public class WelcomeServlet extends HttpServlet {

private static final long serialVersionUID = 1L;

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException {

// Set the content type of the response

response.setContentType("text/html");

// Get the user's name from the request parameters

String userName = request.getParameter("name");

// Create a PrintWriter to write HTML response

PrintWriter out = response.getWriter();

// Generate HTML response

out.println("<html>");

out.println("<head>");

out.println("<title>Welcome Page</title>");

out.println("</head>");

out.println("<body>");

// Display the welcome message

out.println("<h2>Welcome, " + userName + "!</h2>");

out.println("</body>");

out.println("</html>");

}

}

Write a Java servlet that reads the data from a MySQL database table and displays it in an HTML table on the web page. The servlet should use JDBC to connect to the database and retrieve the data.

import java.io.IOException;

import java.io.PrintWriter;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

@WebServlet("/WelcomeServlet")

public class WelcomeServlet extends HttpServlet {

private static final long serialVersionUID = 1L;

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException {

// Set the content type of the response

response.setContentType("text/html");

// Get the user's name from the request parameters

String userName = request.getParameter("name");

// Create a PrintWriter to write HTML response

PrintWriter out = response.getWriter();

// Generate HTML response

out.println("<html>");

out.println("<head>");

out.println("<title>Welcome Page</title>");

out.println("</head>");

out.println("<body>");

// Display the welcome message

out.println("<h2>Welcome, " + userName + "!</h2>");

out.println("</body>");

out.println("</html>");

}

}

Create a Java servlet that uses session management to maintain the state of the user across multiple requests. The servlet should store the user's name in a session object and display it on multiple pages of the web application.

import java.io.IOException;

import java.io.PrintWriter;

import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import javax.servlet.http.HttpSession;

@WebServlet("/UserSessionServlet")

public class UserSessionServlet extends HttpServlet {

private static final long serialVersionUID = 1L;

protected void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

// Set the content type of the response

response.setContentType("text/html");

// Get the user's name from the request parameter

String userName = request.getParameter("name");

// Create a session or retrieve an existing session

HttpSession session = request.getSession(true);

// Store the user's name in the session

session.setAttribute("userName", userName);

// Create a PrintWriter to write HTML response

PrintWriter out = response.getWriter();

// Generate HTML response

out.println("<html>");

out.println("<head>");

out.println("<title>User Session Page</title>");

out.println("</head>");

out.println("<body>");

out.println("<h2>Welcome, " + userName + "!</h2>");

out.println("<p>Your name is stored in the session.</p>");

out.println("<a href='/your-web-app-context/NextPageServlet'>Next Page</a>");

out.println("</body>");

out.println("</html>");

}

}

Create a web application that lets users create and view blog posts. The web application should use the MVC pattern, with servlets as controllers, JSPs as views, and a database as the model. Users should be able to create new blog posts by filling out a form that includes a title, description, and content. The web application should use a servlet to store the blog post data in the database. Users should also be able to view all the blog posts on a separate page, and the web application should use a servlet to retrieve the blog post data from the database and display it in a formatted way.

### Project Structure

### /blog-app

### |-- src

### | |-- main

### | |-- java

### | |-- com

### | |-- yourcompany

### | |-- controller

### | |-- BlogController.java

### | |-- model

### | |-- BlogPost.java

### | |-- dao

### | |-- BlogPostDAO.java

### | |-- util

### | |-- DBUtil.java

### | |-- webapp

### | |-- WEB-INF

### | |-- pages

### | |-- create.jsp

### | |-- view.jsp

### | |-- web.xml

### BlogPost.java (Model)

### package com.yourcompany.model;

### public class BlogPost {

### private int id;

### private String title;

### private String description;

### private String content;

### // Getters and setters

### }

### 3. DBUtil.java (Database Utility)

### package com.yourcompany.util;

### import java.sql.Connection;

### import java.sql.DriverManager;

### import java.sql.SQLException;

### public class DBUtil {

### private static final String URL = "jdbc:mysql://localhost:3306/blog\_db";

### private static final String USER = "your\_username";

### private static final String PASSWORD = "your\_password";

### public static Connection getConnection() throws SQLException {

### return DriverManager.getConnection(URL, USER, PASSWORD);

### }

### }

### 4. BlogPostDAO.java (Data Access Object)

package com.yourcompany.dao;

import com.yourcompany.model.BlogPost;

import java.sql.\*;

import java.util.ArrayList;

import java.util.List;

public class BlogPostDAO {

private static final String INSERT\_POST = "INSERT INTO blog\_posts (title, description, content) VALUES (?, ?, ?)";

private static final String SELECT\_ALL\_POSTS = "SELECT \* FROM blog\_posts";

public void createBlogPost(BlogPost post) {

try (Connection connection = DBUtil.getConnection();

PreparedStatement preparedStatement = connection.prepareStatement(INSERT\_POST)) {

preparedStatement.setString(1, post.getTitle());

preparedStatement.setString(2, post.getDescription());

preparedStatement.setString(3, post.getContent());

preparedStatement.executeUpdate();

} catch (SQLException e) {

e.printStackTrace();

}

}

public List<BlogPost> getAllBlogPosts() {

List<BlogPost> posts = new ArrayList<>();

try (Connection connection = DBUtil.getConnection();

Statement statement = connection.createStatement();

ResultSet resultSet = statement.executeQuery(SELECT\_ALL\_POSTS)) {

while (resultSet.next()) {

BlogPost post = new BlogPost();

post.setId(resultSet.getInt("id"));

post.setTitle(resultSet.getString("title"));

post.setDescription(resultSet.getString("description"));

post.setContent(resultSet.getString("content"));

posts.add(post);

}

} catch (SQLException e) {

e.printStackTrace();

}

return posts;

}

}

### 5. BlogController.java (Servlet Controller)

package com.yourcompany.controller;

import com.yourcompany.dao.BlogPostDAO;

import com.yourcompany.model.BlogPost;

import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import java.io.IOException;

import java.util.List;

@WebServlet("/blog")

public class BlogController extends HttpServlet {

private BlogPostDAO blogPostDAO;

@Override

public void init() throws ServletException {

super.init();

blogPostDAO = new BlogPostDAO();

}

protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

String title = request.getParameter("title");

String description = request.getParameter("description");

String content = request.getParameter("content");

BlogPost post = new BlogPost();

post.setTitle(title);

post.setDescription(description);

post.setContent(content);

blogPostDAO.createBlogPost(post);

response.sendRedirect(request.getContextPath() + "/blog");

}

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

List<BlogPost> posts = blogPostDAO.getAllBlogPosts();

request.setAttribute("posts", posts);

request.getRequestDispatcher("/WEB-INF/pages/view.jsp").forward(request, response);

}

}

### 6. create.jsp (View - Form for creating a blog post)

<%@ page contentType="text/html;charset=UTF-8" language="java" %>

<html>

<head>

<title>Create Blog Post</title>

</head>

<body>

<h2>Create Blog Post</h2>

<form action="${pageContext.request.contextPath}/blog" method="post">

Title: <input type="text" name="title" required><br>

Description: <input type="text" name="description" required><br>

Content: <textarea name="content" rows="5" required></textarea><br>

<input type="submit" value="Create Post">

</form>

</body>

</html>

### 7. view.jsp (View - Display all blog posts)

<%@ page contentType="text/html;charset=UTF-8" language="java" %>

<html>

<head>

<title>View Blog Posts</title>

</head>

<body>

<h2>All Blog Posts</h2>

<ul>

<c:forEach var="post" items="${posts}">

<li>

<h3>${post.title}</h3>

<p>${post.description}</p>

<p>${post.content}</p>

</li>

</c:forEach>

</ul>

<a href="${pageContext.request.contextPath}/create.jsp">Create a new post</a>

</body>

</html>

### . Deployment

Deploy the project on a servlet container such as Apache Tomcat. Make sure to configure your database settings and create the necessary database and table (e.g., **blog\_db** and **blog\_posts**).

This is a basic example, and in a real

Create a Java program that uses Hibernate to connect to a MySQL database and

retrieve data from a table. The program should use Hibernate to map the table to a

To create a Java program that uses Hibernate to connect to a MySQL database, retrieve data from a table, and display it on the console, you need to follow these steps:

### 1. Set Up Your Project

Make sure you have the required dependencies in your project. For this example, you'll need Hibernate and a MySQL JDBC driver. You can use tools like Maven or Gradle to manage dependencies.

### 2. Create Hibernate Configuration File

Create a **hibernate.cfg.xml** file to configure Hibernate. Place it in the **src** directory of your project.

xmlCopy code

<!-- src/hibernate.cfg.xml --> <!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN" "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd"> <hibernate-configuration> <session-factory> <!-- JDBC Database connection settings --> <property name="hibernate.connection.driver\_class">com.mysql.cj.jdbc.Driver</property> <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/your\_database</property> <property name="hibernate.connection.username">your\_username</property> <property name="hibernate.connection.password">your\_password</property> <!-- JDBC connection pool settings --> <property name="hibernate.c3p0.min\_size">5</property> <property name="hibernate.c3p0.max\_size">20</property> <property name="hibernate.c3p0.timeout">300</property> <property name="hibernate.c3p0.max\_statements">50</property> <property name="hibernate.c3p0.idle\_test\_period">3000</property> <!-- Specify dialect --> <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property> <!-- Echo all executed SQL to stdout --> <property name="hibernate.show\_sql">true</property> <!-- Drop and re-create the database schema on startup --> <property name="hibernate.hbm2ddl.auto">update</property> <!-- Mention annotated class --> <mapping class="com.yourcompany.model.YourEntityClass"/> </session-factory> </hibernate-configuration>

### 3. Create Entity Class

Create a Java class representing your entity and annotate it with Hibernate annotations.

javaCopy code

// src/com/yourcompany/model/YourEntityClass.java package com.yourcompany.model; import javax.persistence.\*; @Entity @Table(name = "your\_table\_name") public class YourEntityClass { @Id @GeneratedValue(strategy = GenerationType.IDENTITY) @Column(name = "id") private int id; @Column(name = "column\_name1") private String columnName1; @Column(name = "column\_name2") private String columnName2; // Getters and setters @Override public String toString() { return "YourEntityClass{" + "id=" + id + ", columnName1='" + columnName1 + '\'' + ", columnName2='" + columnName2 + '\'' + '}'; } }

### 4. Create Hibernate Utility Class

Create a utility class to manage Hibernate sessions and transactions.

javaCopy code

// src/com/yourcompany/util/HibernateUtil.java package com.yourcompany.util; import org.hibernate.SessionFactory; import org.hibernate.cfg.Configuration; public class HibernateUtil { private static final SessionFactory sessionFactory = buildSessionFactory(); private static SessionFactory buildSessionFactory() { try { // Create the SessionFactory from hibernate.cfg.xml return new Configuration().configure("hibernate.cfg.xml").buildSessionFactory(); } catch (Throwable ex) { System.err.println("Initial SessionFactory creation failed." + ex); throw new ExceptionInInitializerError(ex); } } public static SessionFactory getSessionFactory() { return sessionFactory; } }

### 5. Create Main Program

Create a Java program that uses Hibernate to connect to the MySQL database, retrieve data, and display it on the console.

javaCopy code

// src/com/yourcompany/MainApp.java package com.yourcompany; import com.yourcompany.model.YourEntityClass; import com.yourcompany.util.HibernateUtil; import org.hibernate.Session; import org.hibernate.SessionFactory; public class MainApp { public static void main(String[] args) { // Create a session factory SessionFactory sessionFactory = HibernateUtil.getSessionFactory(); // Open a session try (Session session = sessionFactory.openSession()) { // Begin a transaction session.beginTransaction(); // Retrieve data from the database YourEntityClass result = session.get(YourEntityClass.class, 1); // Display data on the console if (result != null) { System.out.println("Entity Retrieved: " + result); } else { System.out.println("Entity not found!"); } // Commit the transaction session.getTransaction().commit(); } catch (Exception e) { e.printStackTrace(); } finally { // Close the session factory sessionFactory.close(); } } }

### 6. Run the Program

Run the **MainApp** class. Ensure that the necessary MySQL database and table are created, and adjust the configuration in **hibernate.cfg.xml** accordingly.

This example assumes you have a basic understanding of Hibernate, MySQL, and Java programming. Adjust the code according to your specific use case and database schema.

Create a Java program that uses Hibernate to insert data into a MySQL database table. The program should use Hibernate to map the table to a Java object and then insert the data into the table. After inserting the data, the program should retrieve it from the database and display it on the console.

To achieve this, you can follow these steps to create a Java program that uses Hibernate to insert data into a MySQL database table, retrieve the data, and display it on the console:

### 1. Set Up Your Project

Make sure you have the required dependencies in your project. You'll need Hibernate and a MySQL JDBC driver. You can use tools like Maven or Gradle to manage dependencies.

### 2. Create Hibernate Configuration File

Create a **hibernate.cfg.xml** file to configure Hibernate. Place it in the **src** directory of your project. Ensure that the **hibernate.hbm2ddl.auto** property is set to **update** or **create** if you want Hibernate to create or update the database schema.

xmlCopy code

<!-- src/hibernate.cfg.xml --> <!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN" "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd"> <hibernate-configuration> <session-factory> <!-- JDBC Database connection settings --> <property name="hibernate.connection.driver\_class">com.mysql.cj.jdbc.Driver</property> <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/your\_database</property> <property name="hibernate.connection.username">your\_username</property> <property name="hibernate.connection.password">your\_password</property> <!-- JDBC connection pool settings --> <property name="hibernate.c3p0.min\_size">5</property> <property name="hibernate.c3p0.max\_size">20</property> <property name="hibernate.c3p0.timeout">300</property> <property name="hibernate.c3p0.max\_statements">50</property> <property name="hibernate.c3p0.idle\_test\_period">3000</property> <!-- Specify dialect --> <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property> <!-- Echo all executed SQL to stdout --> <property name="hibernate.show\_sql">true</property> <!-- Drop and re-create the database schema on startup --> <property name="hibernate.hbm2ddl.auto">update</property> <!-- Mention annotated class --> <mapping class="com.yourcompany.model.YourEntityClass"/> </session-factory> </hibernate-configuration>

### 3. Create Entity Class

Create a Java class representing your entity and annotate it with Hibernate annotations.

javaCopy code

// src/com/yourcompany/model/YourEntityClass.java package com.yourcompany.model; import javax.persistence.\*; @Entity @Table(name = "your\_table\_name") public class YourEntityClass { @Id @GeneratedValue(strategy = GenerationType.IDENTITY) @Column(name = "id") private int id; @Column(name = "column\_name1") private String columnName1; @Column(name = "column\_name2") private String columnName2; // Getters and setters @Override public String toString() { return "YourEntityClass{" + "id=" + id + ", columnName1='" + columnName1 + '\'' + ", columnName2='" + columnName2 + '\'' + '}'; } }

### 4. Create Hibernate Utility Class

Create a utility class to manage Hibernate sessions and transactions.

javaCopy code

// src/com/yourcompany/util/HibernateUtil.java package com.yourcompany.util; import org.hibernate.SessionFactory; import org.hibernate.cfg.Configuration; public class HibernateUtil { private static final SessionFactory sessionFactory = buildSessionFactory(); private static SessionFactory buildSessionFactory() { try { // Create the SessionFactory from hibernate.cfg.xml return new Configuration().configure("hibernate.cfg.xml").buildSessionFactory(); } catch (Throwable ex) { System.err.println("Initial SessionFactory creation failed." + ex); throw new ExceptionInInitializerError(ex); } } public static SessionFactory getSessionFactory() { return sessionFactory; } }

### 5. Create Main Program

Create a Java program that uses Hibernate to insert data into the MySQL database table, retrieve the data, and display it on the console.

javaCopy code

// src/com/yourcompany/MainApp.java package com.yourcompany; import com.yourcompany.model.YourEntityClass; import com.yourcompany.util.HibernateUtil; import org.hibernate.Session; import org.hibernate.SessionFactory; public class MainApp { public static void main(String[] args) { // Create a session factory SessionFactory sessionFactory = HibernateUtil.getSessionFactory(); // Open a session try (Session session = sessionFactory.openSession()) { // Begin a transaction session.beginTransaction(); // Create an instance of YourEntityClass and set values YourEntityClass entity = new YourEntityClass(); entity.setColumnName1("Value1"); entity.setColumnName2("Value2"); // Save the entity to the database session.save(entity); // Commit the transaction session.getTransaction().commit(); // Display the saved entity System.out.println("Entity Saved: " + entity); } catch (Exception e) { e.printStackTrace(); } finally { // Close the session factory sessionFactory.close(); } } }

### 6. Run the Program

Run the **MainApp** class. Ensure that the necessary MySQL database and table are created, and adjust the configuration in **hibernate.cfg.xml** accordingly.

This example assumes you have a basic understanding of Hibernate, MySQL, and Java programming. Adjust the code according to your specific use case and database schema.

Top of Form

Top of Form

The program should use Hibernate to map the table to a Java object and then update

the data in the table. After updating the data, the program should retrieve it from the

database and display it on the console.

To update data using Hibernate, you can follow a similar structure as the previous example but with modifications to demonstrate the update operation. Below are the steps:

### 1. Update YourEntityClass

Add necessary methods in **YourEntityClass** to support update operations.

javaCopy code

// src/com/yourcompany/model/YourEntityClass.java package com.yourcompany.model; import javax.persistence.\*; @Entity @Table(name = "your\_table\_name") public class YourEntityClass { @Id @GeneratedValue(strategy = GenerationType.IDENTITY) @Column(name = "id") private int id; @Column(name = "column\_name1") private String columnName1; @Column(name = "column\_name2") private String columnName2; // Getters and setters @Override public String toString() { return "YourEntityClass{" + "id=" + id + ", columnName1='" + columnName1 + '\'' + ", columnName2='" + columnName2 + '\'' + '}'; } // Additional methods for update public void setColumnName1(String columnName1) { this.columnName1 = columnName1; } public void setColumnName2(String columnName2) { this.columnName2 = columnName2; } }

### 2. Update Main Program

Modify the **MainApp** program to perform an update operation.

javaCopy code

// src/com/yourcompany/MainApp.java package com.yourcompany; import com.yourcompany.model.YourEntityClass; import com.yourcompany.util.HibernateUtil; import org.hibernate.Session; import org.hibernate.SessionFactory; public class MainApp { public static void main(String[] args) { // Create a session factory SessionFactory sessionFactory = HibernateUtil.getSessionFactory(); // Open a session try (Session session = sessionFactory.openSession()) { // Begin a transaction session.beginTransaction(); // Retrieve an entity by ID for update YourEntityClass entityToUpdate = session.get(YourEntityClass.class, 1); // Update the entity if (entityToUpdate != null) { entityToUpdate.setColumnName1("UpdatedValue1"); entityToUpdate.setColumnName2("UpdatedValue2"); // Save the updated entity to the database session.update(entityToUpdate); // Commit the transaction session.getTransaction().commit(); // Display the updated entity System.out.println("Entity Updated: " + entityToUpdate); } else { System.out.println("Entity not found for update!"); } } catch (Exception e) { e.printStackTrace(); } finally { // Close the session factory sessionFactory.close(); } } }

### 3. Run the Updated Program

Run the **MainApp** class. Ensure that the necessary MySQL database and table are created, and adjust the configuration in **hibernate.cfg.xml** accordingly.

This modified example demonstrates how to use Hibernate to update data in a MySQL database table and then retrieve and display the updated data on the console. Adjust the code according to your specific use case and database schema.

Java object and then display the data on the console.Top of Form