K DEEPAK ASSIGNMENT-(DAY-30)

Day 23: PATTERNS CORE JAVA

Task 1: Singleton

Implement a Singleton class that manages database connections. Ensure the class adheres strictly to the singleton pattern principles.

```
import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;


public class DatabaseManager {



    private static DatabaseManager instance;


    private DatabaseManager() {


        try {

            Class.forName("org.sqlite.JDBC");

            connection = DriverManager.getConnection("jdbc:sqlite:test.db");

        } catch (ClassNotFoundException | SQLException e) {

            e.printStackTrace();

        }

    }
```

```java
public static synchronized DatabaseManager getInstance() {
    if (instance == null) {
        instance = new DatabaseManager();
    }
    return instance;
}

public void executeQuery(String query) {
    try {

        connection.createStatement().execute(query);
    } catch (SQLException e) {
        e.printStackTrace();
    }
}


public void closeConnection() {
    try {
        if (connection != null) {
            connection.close();
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

```
  private Connection connection;


  public static void main(String[] args) {
    DatabaseManager dbManager1 = DatabaseManager.getInstance();
    DatabaseManager dbManager2 = DatabaseManager.getInstance();

    System.out.println(dbManager1 == dbManager2);
    dbManager1.executeQuery("SELECT * FROM users");


    dbManager1.closeConnection();
  }
}
```

Task 2: Factory Method

Create a ShapeFactory class that encapsulates the object creation logic of different Shape objects like Circle, Square, and Rectangle.

```
public class ShapeFactory {


  public Shape createCircle(double radius) {
    return new Circle(radius);
  }
```

```java
    public Shape createSquare(double sideLength) {
        return new Square(sideLength);
    }


    public Shape createRectangle(double width, double height) {
        return new Rectangle(width, height);
    }

    public static void main(String[] args) {
        ShapeFactory factory = new ShapeFactory();


        Shape circle = factory.createCircle(5.0);
        System.out.println("Circle Area: " + circle.area());


        Shape square = factory.createSquare(4.0);
        System.out.println("Square Area: " + square.area());


        Shape rectangle = factory.createRectangle(3.0, 6.0);
        System.out.println("Rectangle Area: " + rectangle.area());
    }
}
```

```
interface Shape {
    double area();
}

class Circle implements Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public double area() {
        return Math.PI * radius * radius;
    }
}

class Square implements Shape {
    private double sideLength;

    public Square(double sideLength) {
        this.sideLength = sideLength;
    }

    @Override
    public double area() {
        return sideLength * sideLength;
    }
}

class Rectangle implements Shape {
    private double width;
    private double height;

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    @Override
    public double area() {
        return width * height;
    }
}
```

Task 3: Proxy

Create a proxy class for accessing a sensitive object that contains a secret key. The proxy should only allow access to the secret key if a correct password is provided.

Code:

```
interface SensitiveObject {
    String getSecretKey();
}



class RealSensitiveObject implements SensitiveObject {
    private String secretKey;

    public RealSensitiveObject(String secretKey) {
        this.secretKey = secretKey;
    }

    @Override
    public String getSecretKey() {
        return secretKey;
    }
}



class SensitiveObjectProxy implements SensitiveObject {
    private RealSensitiveObject realObject;
    private String password;

    public SensitiveObjectProxy(String secretKey, String password) {
        this.realObject = new RealSensitiveObject(secretKey);
        this.password = password;
    }

    @Override
    public String getSecretKey() {
        if (authenticate()) {
            return realObject.getSecretKey();
        } else {
            throw new SecurityException("Access denied: Incorrect password");
        }
    }

    private boolean authenticate() {

        return "correctPassword".equals(password);
    }
}


public class ProxyPatternExample {
    public static void main(String[] args) {
        SensitiveObject proxy = new SensitiveObjectProxy("superSecretKey123", "correctPassword");



        try {
            String secretKey = proxy.getSecretKey();
            System.out.println("Secret Key: " + secretKey);
        } catch (SecurityException e) {
            System.out.println("Error: " + e.getMessage());
        }



        try {
            SensitiveObject proxyWrongPassword = new SensitiveObjectProxy("superSecretKey123", "wrongPassword");
            String secretKey = proxyWrongPassword.getSecretKey();
            System.out.println("Secret Key: " + secretKey);
        } catch (SecurityException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

Task 4: Strategy

Develop a Context class that can use different SortingStrategy algorithms interchangeably to sort a collection of numbers

Code:

```
interface SortingStrategy {
    void sort(int[] array);
}
class BubbleSort implements SortingStrategy {
    @Override
    public void sort(int[] array) {
        int n = array.length;
        boolean swapped;
        do {
            swapped = false;
            for (int i = 1; i < n; i++) {
                if (array[i - 1] > array[i]) {

                    int temp = array[i - 1];
                    array[i - 1] = array[i];
                    array[i] = temp;
                    swapped = true;
                }
            }
            n--;
        } while (swapped);
    }
}

class QuickSort implements SortingStrategy {
    @Override
```

```
public void sort(int[] array) {
    quickSort(array, 0, array.length - 1);
}

private void quickSort(int[] array, int low, int high) {
    if (low < high) {
        int pi = partition(array, low, high);

        quickSort(array, low, pi - 1);
        quickSort(array, pi + 1, high);
    }
}

private int partition(int[] array, int low, int high) {
    int pivot = array[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (array[j] < pivot) {
            i++;
            // Swap elements
            int temp = array[i];
            array[i] = array[j];
            array[j] = temp;
        }
    }
```

```
        int temp = array[i + 1];
        array[i + 1] = array[high];
        array[high] = temp;
        return i + 1;
    }
}

class Context {
    private SortingStrategy strategy;

    public void setStrategy(SortingStrategy strategy) {
        this.strategy = strategy;
    }

    public void sortArray(int[] array) {
        strategy.sort(array);
    }
}


public class StrategyPatternExample {
    public static void main(String[] args) {

        int[] numbers = {5, 2, 8, 1, 6};
        Context context = new Context();


        context.setStrategy(new BubbleSort());
        context.sortArray(numbers.clone());
        System.out.println("Sorted using Bubble Sort:");
        printArray(numbers);


        context.setStrategy(new QuickSort());
        context.sortArray(numbers.clone());
        System.out.println("Sorted using Quick Sort:");
        printArray(numbers);
    }

    private static void printArray(int[] array) {
        for (int num : array) {
            System.out.print(num + " ");
        }
        System.out.println();
    }
}
```