**Day 10:**

**Task 1: Singleton**
**Implement a Singleton class that manages database connections. Ensure the class adheres strictly to the singleton pattern principles.**

We have created Singleton DatabaseConnectionManager class with a different class name, such as App, to demonstrate its usage:

DatabaseConnectionManager

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DatabaseConnectionManager {
    // Static variable to hold the single instance of the class
    private static DatabaseConnectionManager instance;

    // Database connection
    private Connection connection;

    // Database credentials
    private static final String URL = "jdbc:oracle:thin:@localhost:9501/XE";
    private static final String USER = "system";
    private static final String PASSWORD = "rps@123";
    private static final String DRIVER_CLASS = "oracle.jdbc.OracleDriver";

    // Private constructor to prevent instantiation
    private DatabaseConnectionManager() {
        try {
            // Load the database driver
            Class.forName(DRIVER_CLASS);
            // Establish the connection
```

```
            this.connection = DriverManager.getConnection(URL, USER,
PASSWORD);
        } catch (ClassNotFoundException | SQLException e) {
            e.printStackTrace();
            throw new RuntimeException("Failed to connect to the database",
e);
        }
    }

    // Public method to provide access to the instance
    public static DatabaseConnectionManager getInstance() {
        if (instance == null) {
            synchronized (DatabaseConnectionManager.class) {
                if (instance == null) {
                    instance = new DatabaseConnectionManager();
                }
            }
        }
        return instance;
    }

    // Method to get the database connection
    public Connection getConnection() {
        return connection;
    }
}
```

App

```
import java.sql.Connection;
import java.sql.SQLException;

public class App {
    public static void main(String[] args) {
        // Get the singleton instance of the DatabaseConnectionManager
        DatabaseConnectionManager dbManager =
DatabaseConnectionManager.getInstance();

        // Get the database connection
        Connection connection = dbManager.getConnection();

        // Use the connection as needed (for example, to perform a query)
        // For demonstration, we'll just print a message
        System.out.println("Successfully connected to the database.");

        // Remember to close the connection when done
        try {
            if (connection != null && !connection.isClosed()) {
                connection.close();
                System.out.println("Connection closed successfully.");
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

When we run the App class, it will get the single instance of DatabaseConnectionManager, obtain a connection to the database, and then print messages to confirm the connection and its closure. This demonstrates the proper usage of the Singleton pattern for managing database connections.

```
Console ×  Problems  @ Javadoc  Declaration
<terminated> App [Java Application] /snap/eclipse/87/plugins/org.eclip
Successfully connected to the database.
Connection closed successfully.
```

**Task 2: Factory Method**
**Create a ShapeFactory class that encapsulates the object creation logic of different Shape objects like Circle, Square, and Rectangle.**

1. Shape Interface: Defines the common draw method that all shapes will implement.

```
public interface Shape {
    void draw();
}
```

2. Concrete Shape Classes: Circle, Square, and Rectangle implement the Shape interface and provide their specific implementation of the draw method.

```
public class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a Circle");
    }
}
```

```
public class Square implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a Square");
    }
}
```

```
public class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a Rectangle");
    }
}
```

3. ShapeFactory Class: Contains the createShape method that takes a shapeType string as input and returns an instance of the corresponding shape. The method uses if-else statements to determine which shape to create.

```
public class ShapeFactory {

    // Factory method to create shapes
    public Shape createShape(String shapeType) {
        if (shapeType == null) {
            return null;
        }
        if (shapeType.equalsIgnoreCase("CIRCLE")) {
            return new Circle();
        } else if (shapeType.equalsIgnoreCase("SQUARE")) {
            return new Square();
        } else if (shapeType.equalsIgnoreCase("RECTANGLE")) {
            return new Rectangle();
        }
        return null;
```

```
    }
}
```

4. MainApp Class: Demonstrates how to use the ShapeFactory to create different shape objects and call their draw methods.

```
public class MainApp {
    public static void main(String[] args) {
        ShapeFactory shapeFactory = new ShapeFactory();

        // Create a Circle and call its draw method
        Shape shape1 = shapeFactory.createShape("CIRCLE");
        shape1.draw();

        // Create a Square and call its draw method
        Shape shape2 = shapeFactory.createShape("SQUARE");
        shape2.draw();

        // Create a Rectangle and call its draw method
        Shape shape3 = shapeFactory.createShape("RECTANGLE");
        shape3.draw();
    }
}
```

When you run the MainApp class, it will create instances of Circle, Square, and Rectangle using the ShapeFactory and call their draw methods, printing the corresponding messages to the console. This demonstrates the Factory Method pattern in action.

```
Console ×   Problems   @ Javadoc   Declaration
<terminated> MainApp [Java Application] /snap/eclipse/87/plugins/o
Drawing a Circle
Drawing a Square
Drawing a Rectangle
```

**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.**

*Class SensitiveObject containing the secret key*

```
class SensitiveObject {
    private String secretKey;

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

    public String getSecretKey() {
        return secretKey;
    }
}
```

*SensitiveObjectProxyclass that controls access to the SensitiveObject*

```
class SensitiveObjectProxy {
    private SensitiveObject sensitiveObject;
```

```
    private String correctPassword;

    public SensitiveObjectProxy(String secretKey, String correctPassword)
{
        this.sensitiveObject = new SensitiveObject(secretKey);
        this.correctPassword = correctPassword;
    }

    public String getSecretKey(String password) {
        if (authenticate(password)) {
            return sensitiveObject.getSecretKey();
        } else {
            throw new SecurityException("Invalid password. Access
denied.");
        }
    }

    private boolean authenticate(String password) {
        return correctPassword.equals(password);
    }
}
```

```
public class SensitiveMain {
    public static void main(String[] args) {
        String secretKey = "superSecretKey123";
        String correctPassword = "password123";

        SensitiveObjectProxy proxy = new SensitiveObjectProxy(secretKey,
correctPassword);

        try {
```

```
            // Attempt to access the secret key with the correct password
            System.out.println("Accessing with correct password: " +
proxy.getSecretKey("password123"));
        } catch (SecurityException e) {
            System.out.println(e.getMessage());
        }

        try {
            // Attempt to access the secret key with an incorrect password
            System.out.println("Accessing with incorrect password: " +
proxy.getSecretKey("wrongPassword"));
        } catch (SecurityException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Console × Problems @ Javadoc Declaration

```
<terminated> SensitiveMain [Java Application] /snap/eclipse/87/plugi
Accessing with correct password: superSecretKey123
Invalid password. Access denied.
```

**Task 4: Strategy**
**Develop a Context class that can use different SortingStrategy algorithms interchangeably to sort a collection of numbers**

*SortingStrategy interface*

```
public interface SortingStrategy {
    void sort(int[] arr);
}
```

*BubbleSort implementation*

```
public class BubbleSort implements SortingStrategy {
    @Override
    public void sort(int[] arr) {
        int n = arr.length;
        for (int i = 0; i < n - 1; i++) {
            for (int j = 0; j < n - i - 1; j++) {
                if (arr[j] > arr[j + 1]) {
                    // Swap arr[j] and arr[j+1]
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                }
            }
        }
    }
}
```

*MergeSort implementation*

```
public class MergeSort implements SortingStrategy {
    @Override
    public void sort(int[] arr) {
```

```
        if (arr.length > 1) {
            // Merge sort the first half
            int[] firstHalf = new int[arr.length / 2];
            System.arraycopy(arr, 0, firstHalf, 0, arr.length / 2);
            sort(firstHalf);

            // Merge sort the second half
            int secondHalfLength = arr.length - arr.length / 2;
            int[] secondHalf = new int[secondHalfLength];
            System.arraycopy(arr, arr.length / 2, secondHalf, 0,
secondHalfLength);
            sort(secondHalf);

            // Merge firstHalf and secondHalf into arr
            merge(firstHalf, secondHalf, arr);
        }
    }

    private void merge(int[] firstHalf, int[] secondHalf, int[] arr) {
        int i = 0, j = 0, k = 0;
        while (i < firstHalf.length && j < secondHalf.length) {
            if (firstHalf[i] < secondHalf[j]) {
                arr[k++] = firstHalf[i++];
            } else {
                arr[k++] = secondHalf[j++];
            }
        }

        while (i < firstHalf.length) {
            arr[k++] = firstHalf[i++];
        }

        while (j < secondHalf.length) {
            arr[k++] = secondHalf[j++];
```

```
        }
    }
}
```

*Context Class*

```
public class Context {
    private SortingStrategy strategy;

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

    public void sort(int[] arr) {
        strategy.sort(arr);
    }
}
```

*ContextMain Class*

```
public class Main {
    public static void main(String[] args) {
        int[] arr = {5, 2, 8, 1, 9};
        System.out.println("Original array: " + Arrays.toString(arr));

        // Sort using BubbleSort
        Context context = new Context();
        context.setStrategy(new BubbleSort());
        context.sort(arr);
        System.out.println("Array sorted using BubbleSort: " +
Arrays.toString(arr));

        // Sort using MergeSort
```

```
        int[] arr2 = {5, 2, 8, 1, 9};
        context.setStrategy(new MergeSort());
        context.sort(arr2);
        System.out.println("Array sorted using MergeSort: " +
Arrays.toString(arr2));
    }
}
```

**Output**

```
Console ×   Problems   @ Javadoc   Declaration
<terminated> ContextMain [Java Application] /snap/eclipse/87/plugins/
Original array: [5, 2, 8, 1, 9]
Array sorted using BubbleSort: [1, 2, 5, 8, 9]
Array sorted using MergeSort: [1, 2, 5, 8, 9]
```