Name: Vaishali Kale

Email id: kalevaishalir16@gmail.com

Day 23:

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

Solution:::

Explanation:

- Private Static Instance: The instance variable holds the single instance of the DatabaseConnection class.
- Private Constructor: The constructor is private to prevent instantiation from outside the class.
- Double-Checked Locking: The getInstance method checks if the instance is null before synchronizing. After entering the synchronized block, it checks again if the instance is still null before creating a new instance. This ensures that the instance is created only once in a thread-safe manner.
- Connection Handling: The constructor initializes the database connection using the DriverManager. Make sure to replace URL, USER, and PASSWORD with your actual database credentials.
- Public Method to Get Connection: The getConnection method returns the Connection object.
- This implementation ensures that only one instance of DatabaseConnection is created and provides a global point of access to the database connection.

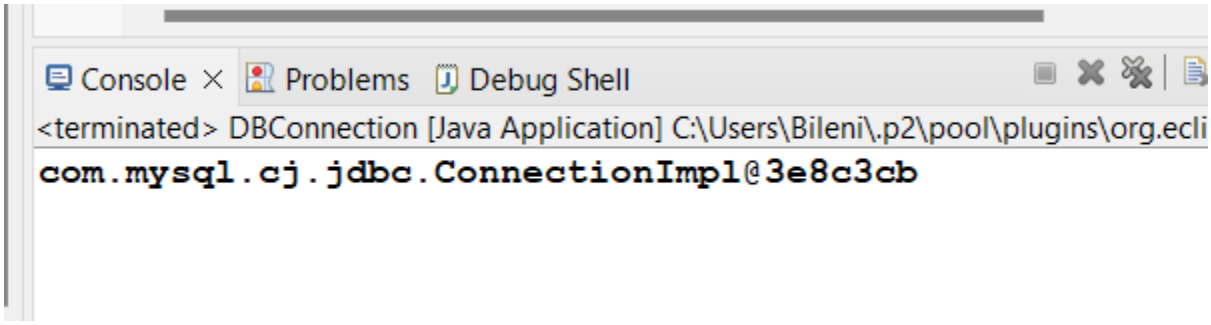
CODE:::

```
package com.wipro.assignments;


import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;


public class DatabaseConnection {

   private static volatile DatabaseConnection instance;

   private Connection connection;
```

```
    private DatabaseConnection() {
        try {
            // Load the JDBC driver
            Class.forName("com.mysql.cj.jdbc.Driver");
            // Establish the connection
            this.connection =
DriverManager.getConnection("jdbc:mysql://localhost:3306/jdbc1", "root",
"Pass@1234");
        } catch (ClassNotFoundException | SQLException e) {
            e.printStackTrace();
            // Handle exceptions
        }
    }

    public static DatabaseConnection getInstance() {
        if (instance == null) {
            synchronized (DatabaseConnection.class) {
                if (instance == null) {
                    instance = new DatabaseConnection();
                }
            }
        }
        return instance;
    }

    public Connection getConnection() {
        return connection;
    }
```

}

OUTPUT::::



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

Solution:::

Explanation:

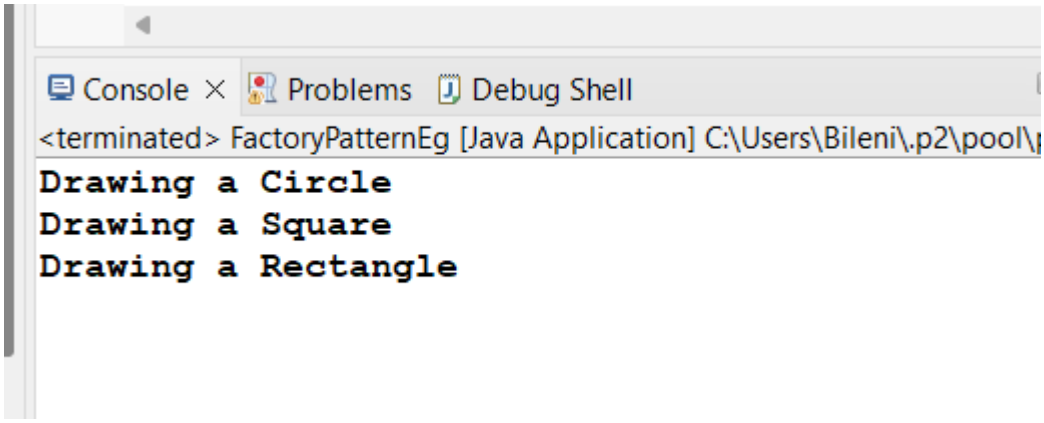
- Shape Interface: Defined as Shape with a method draw().
- Circle, Square, Rectangle Classes: Each implements Shape and defines draw().
- ShapeFactory Class: Contains getShape(String shapeType) method that returns the appropriate Shape object based on the input string.
- FactoryPatternDemo Class: The main method demonstrates how to use ShapeFactory to get different Shape objects and call their draw() methods.
-

CODE:::

```
package com.wipro.assignments;

//Shape.java
interface Shape {
 void draw();
}

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

//Square.java
class Square implements Shape {
```

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

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

//ShapeFactory.java
class ShapeFactory {
 // Use getShape method to get object of type Shape
 public Shape getShape(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;
 }
}

//FactoryPatternDemo.java
public class FactoryPatternEg {

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

      // Get an object of Circle and call its draw method
      Shape shape1 = sf.getShape("CIRCLE");
      shape1.draw();

      // Get an object of Square and call its draw method
      Shape shape2 = sf.getShape("SQUARE");
      shape2.draw();

      // Get an object of Rectangle and call its draw method
      Shape shape3 = sf.getShape("RECTANGLE");
      shape3.draw();
 }
}
```

OUTPUT::::



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.

Solution:::

Explanation:

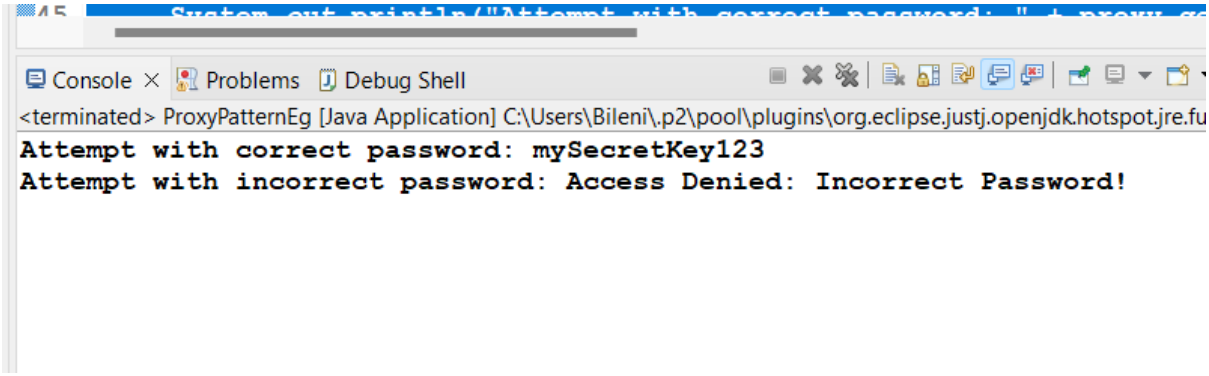
- SensitiveObject Class: This class holds the secret key and has a method to retrieve it.
- SensitiveObjectProxy Class: This proxy class holds a SensitiveObject instance and a password. It checks the provided password before returning the secret key.
- ProxyPatternDemo Class: This class demonstrates how to use the SensitiveObjectProxy to control access to the secret key. It shows access attempts with both correct and incorrect passwords.
- This code block contains the complete implementation of the Proxy pattern, including classes for the sensitive object, its proxy, and a demo class to test the functionality.

CODE:::

```
package com.wipro.assignments;

//SensitiveObject.java
class SensitiveObject {
 private String secretKey;
```

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

 public String getSecretKey() {
     return secretKey;
 }
}

//SensitiveObjectProxy.java
class SensitiveObjectProxy {
 private SensitiveObject senObj;
 private String password;

 public SensitiveObjectProxy(String secretKey, String password) {
     this.senObj = new SensitiveObject(secretKey);
     this.password = password;
 }

 public String getSecretKey(String password) {
     if (this.password.equals(password)) {
         return senObj.getSecretKey();
     } else {
         return "Access Denied: Incorrect Password!";
     }
 }
}

//ProxyPatternDemo.java
public class ProxyPatternEg {
 public static void main(String[] args) {
     String secretKey = "mySecretKey123";
     String correctPassword = "securePassword";
     String incorrectPassword = "wrongPassword";

     SensitiveObjectProxy proxy = new
SensitiveObjectProxy(secretKey, correctPassword);

     // Attempt to access with correct password
     System.out.println("Attempt with correct password: " +
proxy.getSecretKey(correctPassword)); // Output: mySecretKey123

     // Attempt to access with incorrect password
     System.out.println("Attempt with incorrect password: " +
proxy.getSecretKey(incorrectPassword)); // Output: Access Denied:
Incorrect Password!
 }
}
```

OUTPUT::::



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

Solution:::

Explanation:

- SortingStrategy Interface: Defines the sort method that all sorting algorithms must implement.
- BubbleSortStrategy Class: Implements the SortingStrategy interface using the bubble sort algorithm.
- QuickSortStrategy Class: Implements the SortingStrategy interface using the quicksort algorithm.
- Context Class: Holds a reference to a SortingStrategy and uses it to sort an array of numbers.
- StrategyPatternDemo Class: Demonstrates how to use the Context class with different sorting strategies.

CODE:::

```
package com.wipro.assignments;

import java.util.Arrays;

// Define the SortingStrategy interface
interface SortingStrategy {
    void sort(int[] array);
}

// Implement BubbleSortStrategy
class BubbleSortStrategy implements SortingStrategy {
    @Override
    public void sort(int[] array) {
        int n = array.length;
```

```
        for (int i = 0; i < n - 1; i++) {
            for (int j = 0; j < n - i - 1; j++) {
                if (array[j] > array[j + 1]) {
                    int temp = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = temp;
                }
            }
        }
    }
}

// Implement SelectionSortStrategy
class SelectionSortStrategy implements SortingStrategy {
    @Override
    public void sort(int[] array) {
        int n = array.length;
        for (int i = 0; i < n - 1; i++) {
            int min = i;
            for (int j = i + 1; j < n; j++) {
                if (array[j] < array[min])
                    min = j;
            }
            int temp = array[min];
            array[min] = array[i];
            array[i] = temp;
        }
    }
}

// Context class
class Sorter {
    private SortingStrategy strategy;

    public Sorter(SortingStrategy strategy) {
        this.strategy = strategy;
    }

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

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

// Main application class
public class StrategyPatternEg {
    public static void main(String[] args) {
        int[] array = {5, 3, 8, 4, 2, 1};

        // Bubble sort
        SortingStrategy bubbleSortStrategy = new
BubbleSortStrategy();
        Sorter sorter = new Sorter(bubbleSortStrategy);
```

```
        sorter.sort(array);
        System.out.println("Bubble sort done: " +
Arrays.toString(array));

        // Reset array
        array = new int[]{5, 3, 8, 4, 2, 1};

        // Selection sort
        SortingStrategy selectionSortStrategy = new
SelectionSortStrategy();
        sorter.setStrategy(selectionSortStrategy);
        sorter.sort(array);
        System.out.println("Selection sort done: " +
Arrays.toString(array));
    }
}
```

OUTPUT::::