Prajwalhonashetti143@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.

```
import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;


public class DBConnectionManager {

   private static volatile DBConnectionManager dbInstance;

   private Connection dbConnection;


   private DBConnectionManager() {

      try {

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

dbConnection=DriverManager.getConnection("jdbc:mysql://localhost:3306/yourdatabase
", "username", "password");

      } catch (ClassNotFoundException | SQLException e) {

         e.printStackTrace();

      }

   }


   public static DBConnectionManager getInstance() {

      if (dbInstance == null) {

         synchronized (DBConnectionManager.class) {

            if (dbInstance == null) {
```

```java
                dbInstance = new DBConnectionManager();
            }
        }
    }
    return dbInstance;
  }

  public Connection getConnection() {
    return dbConnection;
  }
}
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;

public class Main {
  public static void main(String[] args) {
    DBConnectionManager dbManager = DBConnectionManager.getInstance();
    Connection connection = dbManager.getConnection();

    if (connection != null) {
      System.out.println("Database connection established successfully!");

      try (Statement stmt = connection.createStatement()) {
        String sql = "SELECT * FROM your_table";
        stmt.executeQuery(sql);
      } catch (SQLException e) {
        e.printStackTrace();
      } finally {
```

```
                try {
                    connection.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
        } else {
            System.out.println("Failed to establish database connection.");
        }
    }
}
```

Explanation

Get Singleton Instance:

- DBConnectionManager dbManager = DBConnectionManager.getInstance();
- This line gets the singleton instance of DBConnectionManager.

Get Database Connection:

- Connection connection = dbManager.getConnection();
- This line retrieves the database connection from the singleton instance.

Check Connection:

- The program checks if the connection is not null to ensure that the connection was successfully established.

Database Operations:

- The program demonstrates executing a SQL query using the connection. You should replace "SELECT * FROM your_table" with an actual SQL query relevant to your database.

Exception Handling:

The program includes exception handling to manage SQL exceptions that may occur during the database operations.

Closing the Connection:

Although the singleton pattern is used to manage the connection, it is a good practice to close the connection when done with database operations to free resources.

O**utput:**

```
Database connection established successfully!
```

Task 2: Factory Method

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

```
public interface Drawable {
   void draw();
}
public class CircleShape implements Drawable {
   @Override
   public void draw() {
      System.out.println("Drawing a Circle");
   }
}
public class SquareShape implements Drawable {
   @Override
   public void draw() {
      System.out.println("Drawing a Square");
   }
}
public class RectangleShape implements Drawable {
    @Override
    public void draw() {
        System.out.println("Drawing a Rectangle");
    }
}
public class DrawableFactory {
    public Drawable getDrawable(String shapeType) {
        if (shapeType == null) {
            return null;
        }
        if (shapeType.equalsIgnoreCase("CIRCLE")) {
            return new CircleShape();
        } else if (shapeType.equalsIgnoreCase("SQUARE")) {
            return new SquareShape();
        } else if (shapeType.equalsIgnoreCase("RECTANGLE")) {
            return new RectangleShape();
        }
        return null;
    }
}
public class DrawableDemo {
    public static void main(String[] args) {
        DrawableFactory drawableFactory = new DrawableFactory();
        Drawable shape1 = drawableFactory.getDrawable("CIRCLE");
        shape1.draw();
        Drawable shape2 = drawableFactory.getDrawable("SQUARE");
        shape2.draw();
        Drawable shape3 = drawableFactory.getDrawable("RECTANGLE");
        shape3.draw();
    }
}
```

Explanation

- **Drawable** Interface: Defines a common method draw() that all shapes must implement.
- Concrete Shape Classes: Each shape class (Circle, Square, Rectangle) implements the Shape interface and provides its own implementation of the draw() method.
- ShapeFactory Class: Contains a method getShape() which takes a string input (shapeType) and returns an instance of the corresponding shape. It uses simple if statements to decide which shape to instantiate.
- Main Class: Demonstrates how to use the ShapeFactory to create different shape objects and call their draw() methods.

Output:

```
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.

```
public class SensitiveObject {
    private String secretKey;

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

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

```
}
public class SensitiveObjectProxy implements SecretProvider {
    private SensitiveObject sensitiveObject;
    private String correctPassword;

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

    @Override
    public String getSecretKey(String password) {
        if (password.equals(correctPassword)) {
            return sensitiveObject.getSecretKey();
        } else {
            return "Access Denied: Incorrect Password!";
        }
    }
}
public class ProxyDemo {
    public static void main(String[] args) {
        // Initialize the proxy with the secret key and correct password
        String secretKey = "SuperSecretKey123";
        String correctPassword = "password123";
        SensitiveObjectProxy proxy = new SensitiveObjectProxy(secretKey, correctPassword);

System.out.println("Attempt with correct password: " +
proxy.getSecretKey("password123"));
System.out.println("Attempt with incorrect password: " +
proxy.getSecretKey("wrongpassword"));
```

```
    }
}
```

Explanation

SensitiveObject Class:

- Holds the sensitive data (secretKey).
- Provides a method to access the secret key.

SecretProvider Interface:

- Defines the contract for accessing the secret key through the getSecretKey(String password) method.

SensitiveObjectProxy Class:

- Implements the SecretProvider interface.
- Contains a reference to a SensitiveObject instance.
- Stores the correct password required to access the secret key.
- Checks the provided password and returns the secret key if the password is correct; otherwise, returns an access denied message.

ProxyDemo Class:

- Demonstrates how to use the proxy class to control access to the secret key.
- Shows attempts to access the secret key with both correct and incorrect passwords.

Output:

```
Attempt with correct password: SuperSecretKey123

Attempt with incorrect password: Access Denied: Incorrect Password!
```

Task 4: Strategy

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

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

```
public class BubbleSortStrategy implements SortingStrategy {
    @Override
    public void sort(int[] numbers) {
        int n = numbers.length;
        for (int i = 0; i < n-1; i++) {
            for (int j = 0; j < n-i-1; j++) {
                if (numbers[j] > numbers[j+1]) {
                    // swap numbers[j+1] and numbers[j]
                    int temp = numbers[j];
                    numbers[j] = numbers[j+1];
                    numbers[j+1] = temp;
                }
            }
        }
    }
}
public class MergeSortStrategy implements SortingStrategy {
    @Override
    public void sort(int[] numbers) {
        mergeSort(numbers, 0, numbers.length - 1);
    }

    private void mergeSort(int[] numbers, int left, int right) {
        if (left < right) {
            int middle = (left + right) / 2;
            mergeSort(numbers, left, middle);
            mergeSort(numbers, middle + 1, right);
            merge(numbers, left, middle, right);
        }
```

```
}

private void merge(int[] numbers, int left, int middle, int right) {
    int n1 = middle - left + 1;
    int n2 = right - middle;

    int[] leftArray = new int[n1];
    int[] rightArray = new int[n2];

    for (int i = 0; i < n1; i++) {
        leftArray[i] = numbers[left + i];
    }
    for (int j = 0; j < n2; j++) {
        rightArray[j] = numbers[middle + 1 + j];
    }

    int i = 0, j = 0;
    int k = left;

    while (i < n1 && j < n2) {
        if (leftArray[i] <= rightArray[j]) {
            numbers[k] = leftArray[i];
            i++;
        } else {
            numbers[k] = rightArray[j];
            j++;
        }
        k++;
    }
```

```
        while (i < n1) {
            numbers[k] = leftArray[i];
            i++;
            k++;
        }

        while (j < n2) {
            numbers[k] = rightArray[j];
            j++;
            k++;
        }
    }
} public class SortingContext {
    private SortingStrategy strategy;

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

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


import java.util.Arrays;

public class SortingDemo {
    public static void main(String[] args) {
        SortingContext context = new SortingContext();
        int[] numbers = { 5, 1, 7, 3, 9, 2 };



        SortingStrategy bubbleSort = new BubbleSortStrategy();
        context.setStrategy(bubbleSort);
        context.performSort(numbers.clone());
        System.out.println("Bubble Sort Result: " + Arrays.toString(numbers));

        SortingStrategy mergeSort = new MergeSortStrategy();
        context.setStrategy(mergeSort);
        context.performSort(numbers.clone());
        System.out.println("Merge Sort Result: " + Arrays.toString(numbers));
    }
}
```

Explanation

Sorting Strategy Interface (SortingStrategy):

Defines the contract (sort() method) that all sorting strategies must implement.

Sorting Strategy Implementations (BubbleSortStrategy, MergeSortStrategy):

Each class implements the SortingStrategy interface with its own sorting algorithm.

Sorting Context (SortingContext):

Contains a reference to a SortingStrategy instance.

Provides methods (setStrategy() and performSort()) to set a strategy and perform sorting.

Sorting Demo (SortingDemo):

Demonstrates how to use the SortingContext to switch between different sorting strategies (BubbleSortStrategy and MergeSortStrategy).

Clones the array before sorting to ensure the original array remains unchanged for demonstration purposes. Expected Output

the output will be:

Bubble Sort Result: [1, 2, 3, 5, 7, 9]

Merge Sort Result: [1, 2, 3, 5, 7, 9]