**Superset ID: 6394725**

**Design Patterns and Principles**

**Exercise 1: Implementing the Singleton Pattern**

**Scenario:**

You need to ensure that a logging utility class in your application has only one instance throughout the application lifecycle to ensure consistent logging.

**Steps:**

1. **Create a New Java Project:**
   1. Create a new Java project named **SingletonPatternExample**.
2. **Define a Singleton Class:**
   1. Create a class named Logger that has a private static instance of itself.
   2. Ensure the constructor of Logger is private.
   3. Provide a public static method to get the instance of the Logger class.
3. **Implement the Singleton Pattern:**
   1. Write code to ensure that the Logger class follows the Singleton design pattern.
4. **Test the Singleton Implementation:**
5. Create a test class to verify that only one instance of Logger is created and used across the application.

**Step-1 :** **Create a New Java Project**

Create a Java project named SingletonPatternExample.

Create two files:

* Logger.java → Singleton class
* LoggerTest.java → Test class

**Step-2 :** **Define a Singleton Class**

**Logger.java**

package SingletonPatternExample;

public class Logger {

// Step 1: Create a private static instance

private static Logger *instance*;

// Step 2: Make the constructor private

private Logger() {

System.***out***.println("Logger initialized...");

}

// Step 3: Public method to provide access to the instance

public static Logger getInstance() {

if (*instance* == null) {

*instance* = new Logger(); // Create instance only once

}

return *instance*;

}

// Method to simulate logging

public void log(String message) {

System.***out***.println("Log: " + message);

}

}

**Step-3&4: Implement & Test the Singleton Pattern**

**LoggerTest.java**

package SingletonPatternExample;

public class LoggerTest {

public static void main(String[] args) {

// Get Logger instance 1

Logger logger1 = Logger.*getInstance*();

logger1.log("First log message");

// Get Logger instance 2

Logger logger2 = Logger.*getInstance*();

logger2.log("Second log message");

// Verify both instances are the same

if (logger1 == logger2) {

System.***out***.println("Both logger instances are the same (singleton works!)");

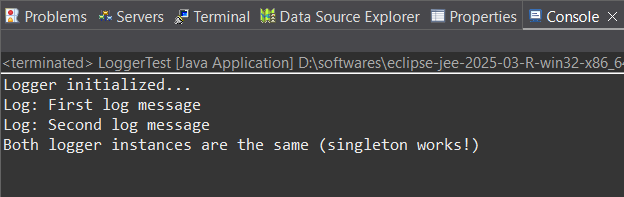
} else {

System.***out***.println("Logger instances are different (singleton failed!)");

}

}

}

**Output :** 

**Exercise 2: Implementing the Factory Method Pattern**

**Scenario:**

You are developing a document management system that needs to create different types of documents (e.g., Word, PDF, Excel). Use the Factory Method Pattern to achieve this.

**Steps:**

1. **Create a New Java Project:**
   1. Create a new Java project named **FactoryMethodPatternExample**.
2. **Define Document Classes:**
   1. Create interfaces or abstract classes for different document types such as **WordDocument**, **PdfDocument**, and **ExcelDocument**.
3. **Create Concrete Document Classes:**
   1. Implement concrete classes for each document type that implements or extends the above interfaces or abstract classes.
4. **Implement the Factory Method:**
   1. Create an abstract class **DocumentFactory** with a method **createDocument()**.
   2. Create concrete factory classes for each document type that extends DocumentFactory and implements the **createDocument()** method.
5. **Test the Factory Method Implementation:**
   1. Create a test class to demonstrate the creation of different document types using the factory method.

## **Step 1: Create a New Java Project**

**Project Name:** FactoryMethodPatternExample  
 Create the following files:

* Document.java (interface or abstract class)
* WordDocument.java, PdfDocument.java, ExcelDocument.java (concrete classes)
* DocumentFactory.java (abstract factory)
* WordFactory.java, PdfFactory.java, ExcelFactory.java (concrete factories)
* Main.java (test class)

## **Step 2: Define Document Interface**

**Document.java**

package FactoryMethodPatternExample;

public interface Document {

void open();

}

## **Step 3: Create Concrete Document Classes**

**WordDocument.java**

package FactoryMethodPatternExample;

public class WordDocument implements Document {

public void open() {

System.***out***.println("Opening Word document...");

}

}

**PdfDocument.java**

package FactoryMethodPatternExample;

public class PdfDocument implements Document {

public void open() {

System.***out***.println("Opening PDF document...");

}

}

**ExcelDocument.java**

package FactoryMethodPatternExample;

public class ExcelDocument implements Document {

public void open() {

System.***out***.println("Opening Excel document...");

}

}

## **Step 4: Implement the Factory Method**

### **Abstract Factory:**

package FactoryMethodPatternExample;

public abstract class DocumentFactory {

public abstract Document createDocument();

}

### **Concrete Factories:**

**WordDocumentFactory.java**

package FactoryMethodPatternExample;

public class WordDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

**PdfDocumentFactory.java**

package FactoryMethodPatternExample;

public class PdfDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

**ExcelDocumentFactory.java**

package FactoryMethodPatternExample;

public class ExcelDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

## **Step 5: Test the Factory Method Implementation**

package FactoryMethodPatternExample;

public class DocumentFactoryTest {

public static void main(String[] args) {

DocumentFactory wordFactory = new WordDocumentFactory();

Document wordDoc = wordFactory.createDocument();

wordDoc.open();

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open();

DocumentFactory excelFactory = new ExcelDocumentFactory();

Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

**Output:**

