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

**Code Snippet:**

class MyLogger {

// Only one instance of MyLogger should exist

private static MyLogger singletonLogger; // Yeah, just keeping it static here

// Private constructor so no one can randomly create this class elsewhere

private MyLogger() {

System.out.println(">>> Logger initialized (only once).");

}

// Public way to access the single logger instance

public static MyLogger getLogger() {

// Lazy instantiation – we only create it when someone asks for it

if (singletonLogger == null) {

singletonLogger = new MyLogger();

}

return singletonLogger;

}

// Method for dumping messages to console

public void writeLog(String msg) {

// I might fancy up the format later, but this works for now

System.out.println(">> LOG: " + msg);

}

}

public class SingletonPatternExample {

public static void main(String[] args) {

System.out.println("Starting Logger Singleton Demo");

// First grab

System.out.println("\n[Step 1] Fetching Logger instance");

MyLogger logOne = MyLogger.getLogger();

// Second grab – should be the same as the first

System.out.println("\n[Step 2] Fetching Logger again");

MyLogger logTwo = MyLogger.getLogger();

// Confirming singleton nature (a little redundant but reassuring)

System.out.println("\n[Step 3] Are both loggers the same object?");

System.out.println("logOne == logTwo? -> " + (logOne == logTwo));

// Try logging something through both

System.out.println("\n[Step 4] Writing logs from both references...");

logOne.writeLog("Message from Logger1");

logTwo.writeLog("Message from Logger2");

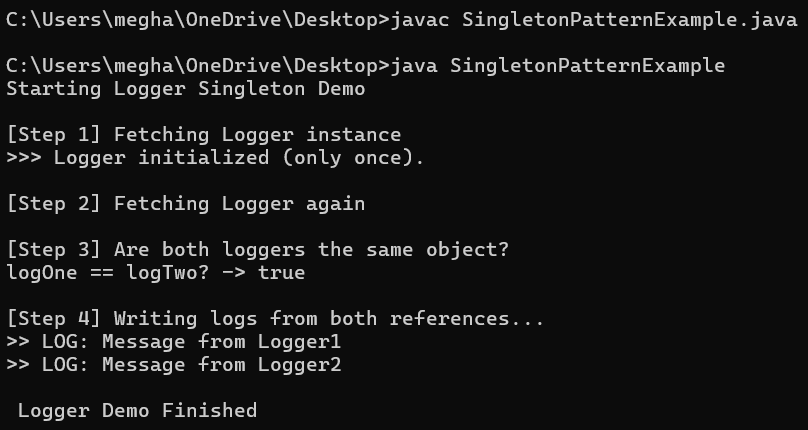
// End of demo

System.out.println("\nLogger Demo Finished");

}

}

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

**Code Snippet:**

abstract class BaseDoc {

public abstract void generate();

}

// Word doc implementation

class WordDoc extends BaseDoc {

@Override

public void generate() {

System.out.println("Word Document was created.");

}

}

// PDF doc implementation

class PdfDoc extends BaseDoc {

@Override

public void generate() {

System.out.println("PDF Document was created.");

}

}

// Excel doc implementation

class ExcelDoc extends BaseDoc {

@Override

public void generate() {

System.out.println("Excel Document was created.");

}

}

// Abstract factory

abstract class DocFactory {

public abstract BaseDoc makeDocument();

// Common method to run the document through some setup

public void handleDocument() {

BaseDoc doc = makeDocument();

doc.generate();

}

}

// Factory for Word docs

class WordDocFactory extends DocFactory {

@Override

public BaseDoc makeDocument() {

return new WordDoc(); // Just return the WordDoc instance

}

}

// Factory for PDF docs

class PdfDocFactory extends DocFactory {

@Override

public BaseDoc makeDocument() {

return new PdfDoc();

}

}

// Factory for Excel docs

class ExcelDocFactory extends DocFactory {

@Override

public BaseDoc makeDocument() {

return new ExcelDoc();

}

}

// Testing it

public class FactoryMethodPatternExample {

public static void main(String[] args) {

System.out.println("Factory Method Pattern Test ");

// Word

System.out.println("\n[1] Creating Word document ");

DocFactory wordFactory = new WordDocFactory();

BaseDoc word = wordFactory.makeDocument();

word.generate(); // Just directly calling it

// PDF

System.out.println("\n[2] Creating PDF document ");

DocFactory pdfFactory = new PdfDocFactory();

BaseDoc pdf = pdfFactory.makeDocument();

pdf.generate();

// Excel

System.out.println("\n[3] Creating Excel document ");

DocFactory excelFactory = new ExcelDocFactory();

BaseDoc excel = excelFactory.makeDocument();

excel.generate();

// Now using the built-in handler

System.out.println("\n[4] Using the factory's built-in process method:");

System.out.println("Processing Word doc ");

wordFactory.handleDocument();

System.out.println("Processing PDF doc ");

pdfFactory.handleDocument();

System.out.println("Processing Excel doc ");

excelFactory.handleDocument();

}

}

**Output:**

