**WEEK-1**

**Design principles & Patterns**

**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.  
  
**Code (SingletonPatternExample)**

//creating a class named Logger

class Logger {

//private static instance of logger class

private static Logger obj;

// Private constructor

private Logger () {

System.out.println("Object Created");

}

// Public static method to get the singleton instance

public static Logger getInstance() {

if (obj == null)

{

synchronized(Logger.class)

{

if(obj==null)

obj = new Logger();

}

}

return obj;

}

}

public class Main {

public static void main(String[] args) {

//implementing singleton pattern

Logger obj1 = Logger.getInstance();

Logger obj2 = Logger.getInstance();

//testing the singleton implementation

if (obj1 == obj2) {

System.out.println("Both the objects of logger class are the same.");

} else {

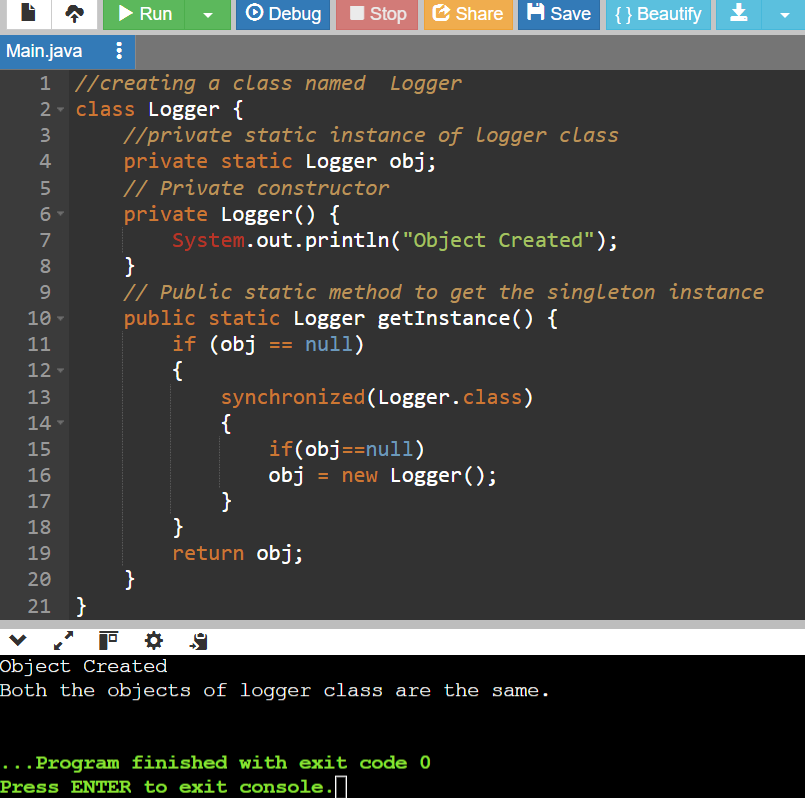
System.out.println("Two different objects of logger class exist.");

}

}

}

**Output:**



**Explanation:**

* Singleton class is a class that has only a single object or instance.
* A singleton class should always have a private static object, private constructor and a public static method to return object.
* Even though I tried to create object twice, only one object is created.
* My code implements singleton pattern implementation + double checking , to ensure software is thread safe and robust.
* There are other techniques of sleeping a thread, using synchronized methods , using enum. But this method is best performing and suitable for the requirement.

**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.

**Code (FactoryMethodPatternExample)**

// Document interface

interface Document {

void open();

}

// Concrete Document Classes

class WordDocument implements Document {

public void open() {

System.out.println("Opening a Word doc");

}

}

class PdfDocument implements Document {

public void open() {

System.out.println("Opening a PDF doc");

}

}

class ExcelDocument implements Document {

public void open() {

System.out.println("Opening an Excel doc");

}

}

//Creating abstract class DocumentFactory with a method createDocument().

abstract class DocumentFactory {

public abstract Document createDocument();

}

class WordDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

class PdfDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

class ExcelDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

// Main class to test the factory method

public class Main {

public static void main(String[] args) {

//Creating concrete factory classes & implementing createDocument() method.

DocumentFactory wordFactory = new WordDocumentFactory();

Document word = wordFactory.createDocument();

word.open();

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdf = pdfFactory.createDocument();

pdf.open();

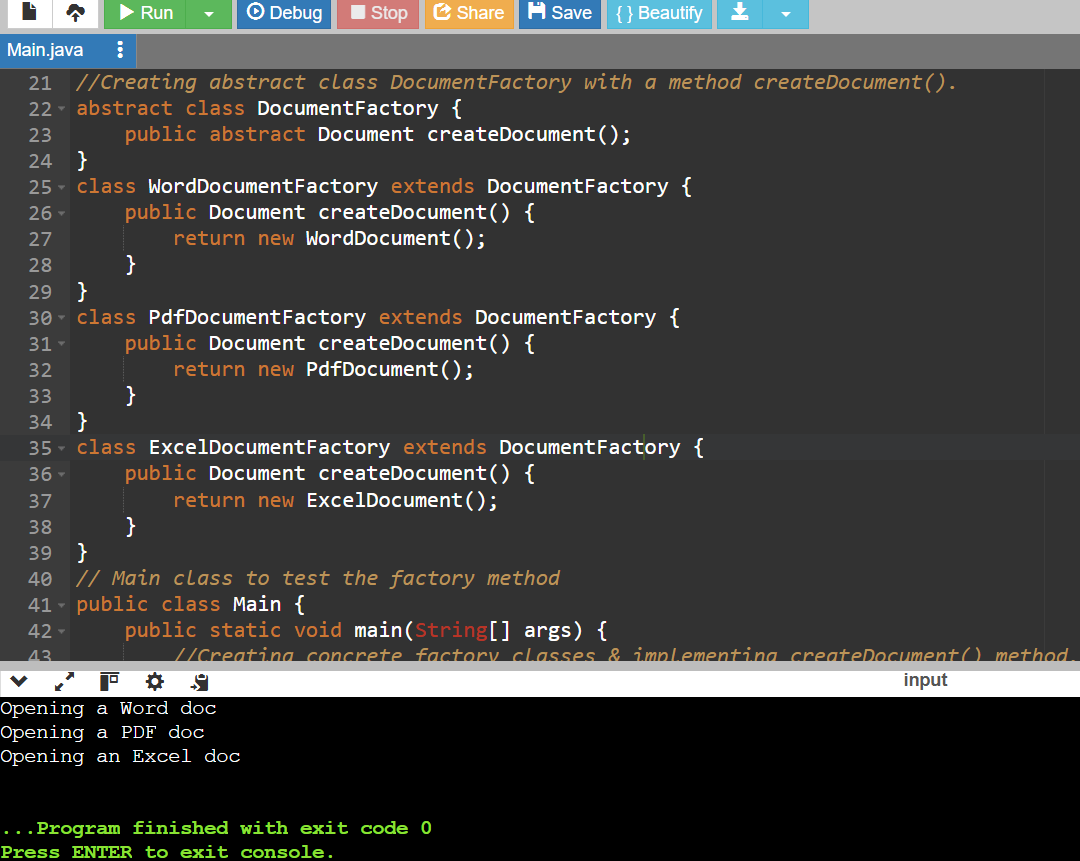
DocumentFactory excelFactory = new ExcelDocumentFactory();

Document excel = excelFactory.createDocument();

excel.open();

} }

**Output:**



**Explanation:**

* We use the **Factory Method Pattern** when we want to create objects without using the new keyword directly. Instead, a factory class takes care of creating the objects.
* In my code, I used this pattern to create different types of documents like Word, PDF, and Excel.
* I made a common interface called Document, and then created separate classes for each document type: WordDocument, PdfDocument, and ExcelDocument.
* Each document type has its own factory class that extends an abstract class and overrides the createDocument() method to return the correct object.
* This keeps the object creation separate from the main program and makes it easy to add new document types later without changing existing code.