**Design Patterns and Principles**

**Exercise 1**: Implementing the Singleton Pattern

In software development, ensuring that a class has only one instance across the entire application is a common design requirement, particularly for utility classes like loggers. This report outlines the implementation of the **Singleton Design Pattern** in Java to maintain a consistent logging mechanism throughout an application.

**Scenario:**

We need to develop a logging utility class (Logger) that must have only a single instance during the application’s lifecycle. This is essential for consistent logging output and avoiding issues like duplicate log entries or unnecessary resource usage.

**Program:**

Logger.java

package singleton;

public class Logger {

private static Logger *instance*;

private Logger() {

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

}

public static synchronized Logger getInstance() {

if(*instance*==null) {

*instance*=new Logger();

}

return *instance*;

}

public void log(String message) {

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

}

}

SingletonTest.java

package singleton;

public class SingletonTest {

public static void main(String args[]) {

Logger logger1=Logger.*getInstance*();

Logger logger2=Logger.*getInstance*();

logger1.log("First message");

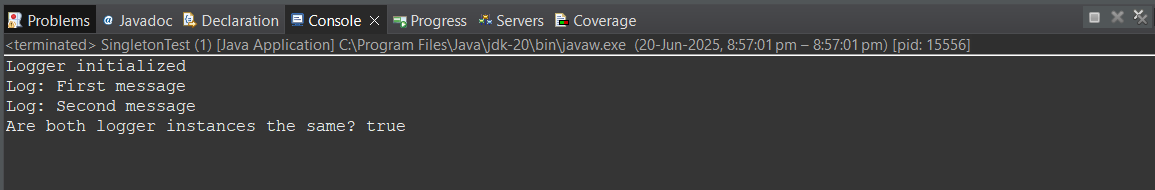
logger2.log("Second message");

System.***out***.println("Are both logger instances the same? " + (logger1 == logger2));

}

}

Output:



Exercise 2 – Implementing the Factory Method Pattern

The **Factory Method Pattern** is a creational design pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created. This pattern is particularly useful in applications requiring multiple variants of a similar type, such as different types of documents in a document management system.

**Scenario:**

The goal is to design a document management system that can create different types of documents such as Word, PDF, and Excel using the Factory Method Pattern. This approach enables flexible and scalable creation of documents without modifying existing code.

**Program:**

Document.java

package factoryMethod;

public interface Document {

void open();

}

//concrete classes

class WordDocument implements Document {

public void open() {

System.***out***.println("Word Document Created");

}

}

class PdfDocument implements Document{

public void open() {

System.***out***.println("Pdf Document Created");

}

}

class ExcelDocument implements Document{

public void open() {

System.***out***.println("Excel Document Created");

}

}

DocumentFactory.java

package factoryMethod;

abstract class DocumentFactory{

abstract Document createDocument();

}

//concrete classes

class WordDocumentFactory extends DocumentFactory {

Document createDocument() {

return new WordDocument();

}

}

class PdfDocumentFactory extends DocumentFactory{

Document createDocument() {

return new PdfDocument();

}

}

class ExcelDocumentFactory extends DocumentFactory{

Document createDocument() {

return new ExcelDocument();

}

}

DocumentFactoryTest.java

package factoryMethod;

public class DocumentFactoryTest {

public static void main(String args[]) {

DocumentFactory docW=new WordDocumentFactory();

docW.createDocument().open();

DocumentFactory docP=new PdfDocumentFactory();

docP.createDocument().open();

DocumentFactory docE=new ExcelDocumentFactory();

docE.createDocument().open();

}

}

Output:

