# **Week-1: Design Pattern and Principles**

### Exercise-1: Implementing the Singleton Pattern:

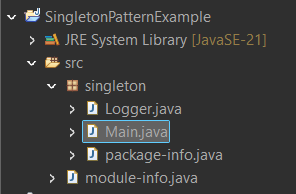
**Scenario:** Ensure a logging utility class in your application has **only one instance** throughout its lifecycle to guarantee consistent logging.

**Steps:**

1. **Create Java Project:**
   1. Create a new Java project named SingletonPatternExample.
2. **Define Logger Class:**
   1. Inside the src folder, create a package named singleton.
   2. Inside the singleton package, create a class named Logger.
   3. Declare a private static Logger instance = new Logger();
   4. Make the constructor private Logger().
   5. Provide a public static Logger getInstance() { return instance; }
3. **Implement Pattern:**
   1. The combination of the private constructor and the getInstance() method ensures only one Logger object can be created and used.
4. **Test in Main.java:**
   1. Inside the singleton package, create a class named Main.
   2. In the main method, fetch two Logger instances: Logger logger1 = Logger.getInstance(); Logger logger2 = Logger.getInstance();
   3. Compare them to verify the singleton behavior: if (logger1 == logger2) { System.out.println("Same Logger instance used. Singleton works."); } else { System.out.println("Different Logger instances. Singleton failed."); }

### Implementation:

**Folder Structure:**



**Code:**

**Logger.java**

package singleton;

public class Logger {

// Private static instance of Logger, initialized upon class loading

private static Logger instance = new Logger();

// Private constructor to prevent direct instantiation from outside

private Logger() {

System.out.println("Logger Initialized.");

}

// Public static method to get the single instance of the Logger class

public static Logger getInstance() {

return instance;

}}

**Main.java**

package singleton;

public class Main {

public static void main(String[] args) {

// Fetching Logger instances

// Both logger1 and logger2 should refer to the same instance due to the Singleton pattern

Logger logger1 = Logger.getInstance();

Logger logger2 = Logger.getInstance();

// Verifying Singleton behavior

// If logger1 and logger2 are the same object, the Singleton pattern works

if (logger1 == logger2) {

System.out.println("Same Logger instance used. Singleton works.");

} else {

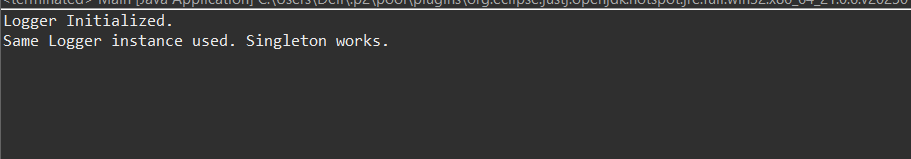
System.out.println("Different Logger instances. Singleton failed.");

}

}

}

Output:



### Exercise-2: Implementing the Factory Method Pattern:

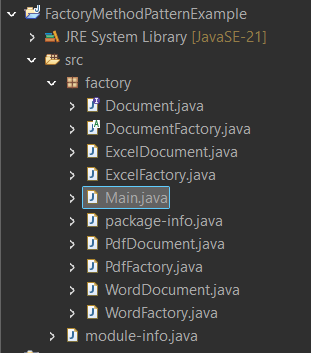
**Scenario:** Develop a document management system to create various document types (Word, PDF, Excel). Design for easy addition of new document formats **without modifying existing client code**, decoupling creation logic from usage.

**Steps:**

1. **Create Java Project:** Name it FactoryMethodPatternExample.
2. **Define Document Interface:**
   1. Create package factory, then interface Document with an open() method.
3. **Create Concrete Documents:**
   1. Create WordDocument, PdfDocument, ExcelDocument classes, each implementing Document and its open() method.
4. **Implement Abstract Factory:**
   1. Create abstract class DocumentFactory with public abstract Document createDocument();.
5. **Create Concrete Factories:**
   1. Create WordFactory, PdfFactory, ExcelFactory classes, each extending DocumentFactory and implementing createDocument() to return their respective document types.
6. **Test in Main.java:**
   1. In factory package, create Main class.
   2. In main, create instances of concrete factories (e.g., new WordFactory()).
   3. Use factory.createDocument() to get Document objects.
   4. Call document.open() on each.

### Implementation:

**Folder Structure:**



**Code:**

**Document.java**

package factory;

public interface Document {

void open();

}

**DocumentFactory.java**

package factory;

public abstract class DocumentFactory {

public abstract Document createDocument();

}

**WordDocument.java**

package factory;

public class WordDocument implements Document {

@Override

public void open() {

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

}

}

**PdfDocument.java**

package factory;

public class PdfDocument implements Document {

@Override

public void open() {

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

}

}

**ExcelDocument.java**

package factory;

public class ExcelDocument implements Document {

@Override

public void open() {

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

}

}

**WordFactory.java**

package factory;

public class WordFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new WordDocument();

}

}

**PdfFactory.java**

package factory;

public class PdfFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new PdfDocument();

}

}

**ExcelFactory.java**

package factory;

public class ExcelFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new ExcelDocument();

}

}

**Main.java**

package factory;

public class Main {

public static void main(String[] args) {

// Create a Word document using the WordFactory

DocumentFactory wordFactory = new WordFactory();

Document wordDoc = wordFactory.createDocument();

wordDoc.open();

// Create a PDF document using the PdfFactory

DocumentFactory pdfFactory = new PdfFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open();

// Create an Excel document using the ExcelFactory

DocumentFactory excelFactory = new ExcelFactory();

Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

Output:

