**SKILL LEARNT: DESIGN PRINCIPLES AND PATTERN (WEEK 1)**

**EXERCISE 1: IMPLEMENTING THE SINGLETON PATTERN**

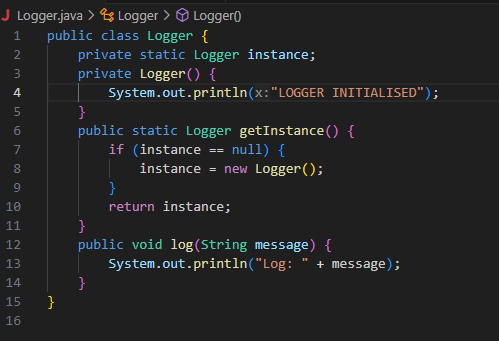
In this exercise, I used the Singleton Design Pattern to create a logging utility class that makes sure that only one instance of the Logger class exists throughout the application's lifecycle. This pattern is useful when just one object is needed to manage actions globally.

To do this, I set up a new Java project called SingletonPatternExample and created a Logger class. It has a private static instance, a private constructor, and a public static method called getInstance() that manages access to the single instance. This way, any part of the application that calls the Logger will get the same instance, ensuring consistent and centralized logging.

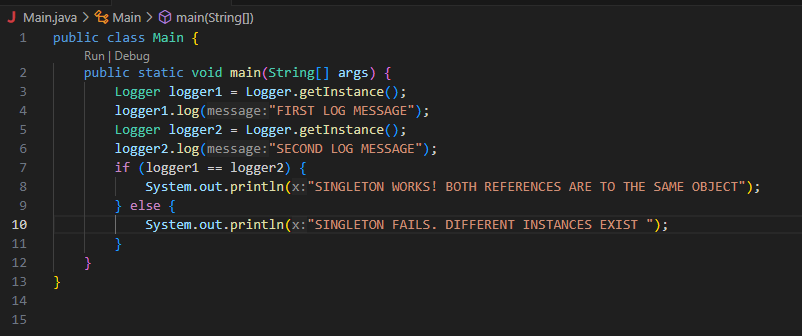
To test the implementation, I developed a separate Main class that accesses the Logger from different points. The output showed that the constructor was called only once, and both references pointed to the same Logger object. This successfully demonstrated the Singleton behavior.

The codes for the Logger.java and Main.java are shown below:

1. **LOGGER.JAVA**



1. **MAIN.JAVA**



**OUTPUT:**

1. **COMPILATION OF THE JAVA FILES**

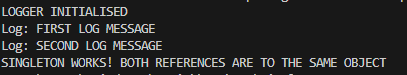


After compilation Logger.class and Main.class files are created.

1. **RUN**



1. **OUTPUT**



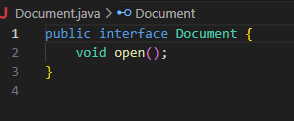
Through this exercise, I learned how to implement the Singleton Design Pattern in Java, ensuring that a class has only one instance throughout the application. I understood the importance of using a private constructor, a static instance, and a public accessor method to control object creation. I also learned how Singleton helps maintain consistent behavior, especially in scenarios like logging, where shared access to a single resource is crucial.

**EXERCISE 2: IMPLEMENTING THE FACTORY METHOD PATTERN**

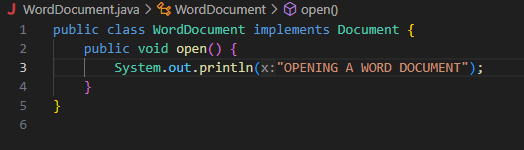
In this exercise, I used the Factory Method Pattern to design a document management system that can create different types of documents like Word, PDF, and Excel. First, I defined a common Document interface. Then, I created concrete classes for each document type. I also developed an abstract DocumentFactory with a factory method called createDocument(), and I implemented separate concrete factory classes for each document variant. Finally, I tested the system by dynamically creating and opening various document types. This showed how the factory method offers flexibility and scalability in creating objects.

There are 9 files under FactoryMethodPatternExample.

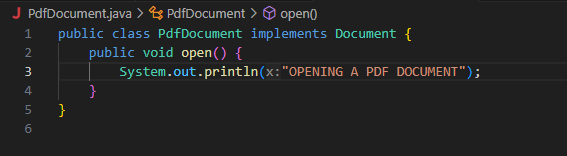
1. **DOCUMENT.JAVA**



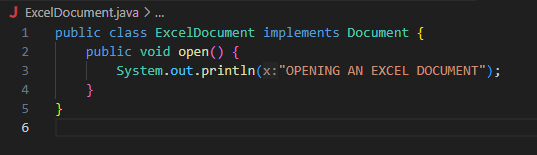
1. **WORDDOCUMENT.JAVA**



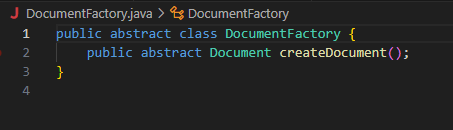
1. **PDFDOCUMENT.JAVA**



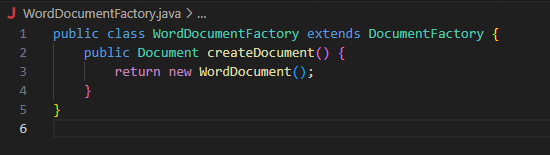
1. **EXCELDOCUMENT.JAVA**



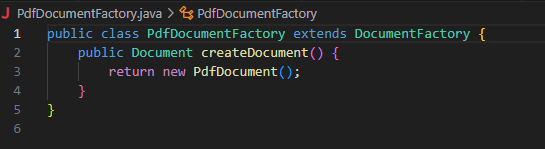
1. **DOCUMENTFACTORY.JAVA**



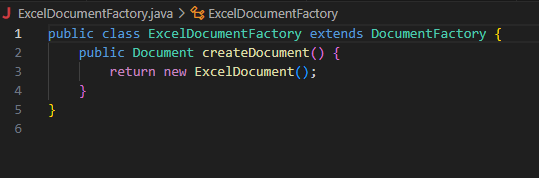
1. **WORDDOCUMENTFACTORY.JAVA**



1. **PDFDOCUMENTFACTORY.JAVA**



1. **EXCELDOCUMENTFACTORY.JAVA**



1. **MAIN.JAVA**



**OUTPUT**

1. **COMPILATION OF JAVA FILES**



1. **RUN**



1. **OUTPUT**



Through this exercise, I learned how to apply the Factory Method Pattern to create objects in a flexible and scalable way. It helped me understand the importance of decoupling object creation from the client code, making the system easier to maintain and extend when new document types are added.