# 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 named SingletonPatternExample.

2. Create a class named Logger that has a private static instance of itself.

3. Ensure the constructor of Logger is private.

4. Provide a public static method to get the instance of the Logger class.

5. Write code to ensure that the Logger class follows the Singleton design pattern.

6. Create a test class to verify that only one instance of Logger is created and used across the application.

## Code Implementation

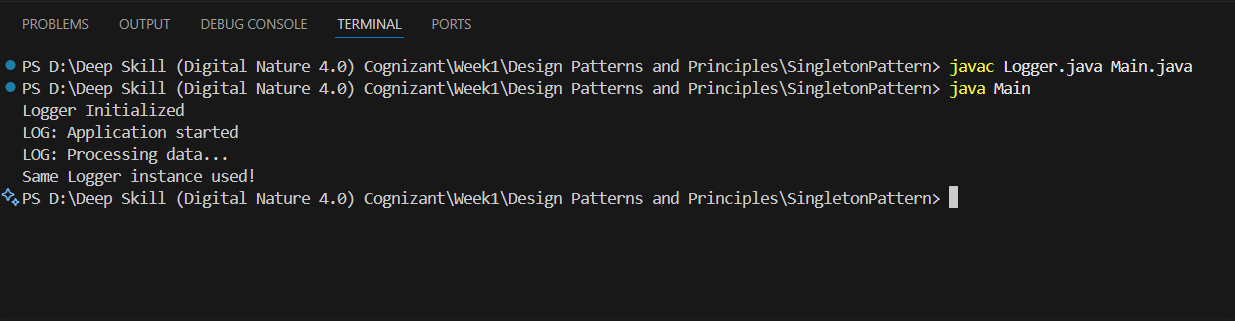
### Logger.java

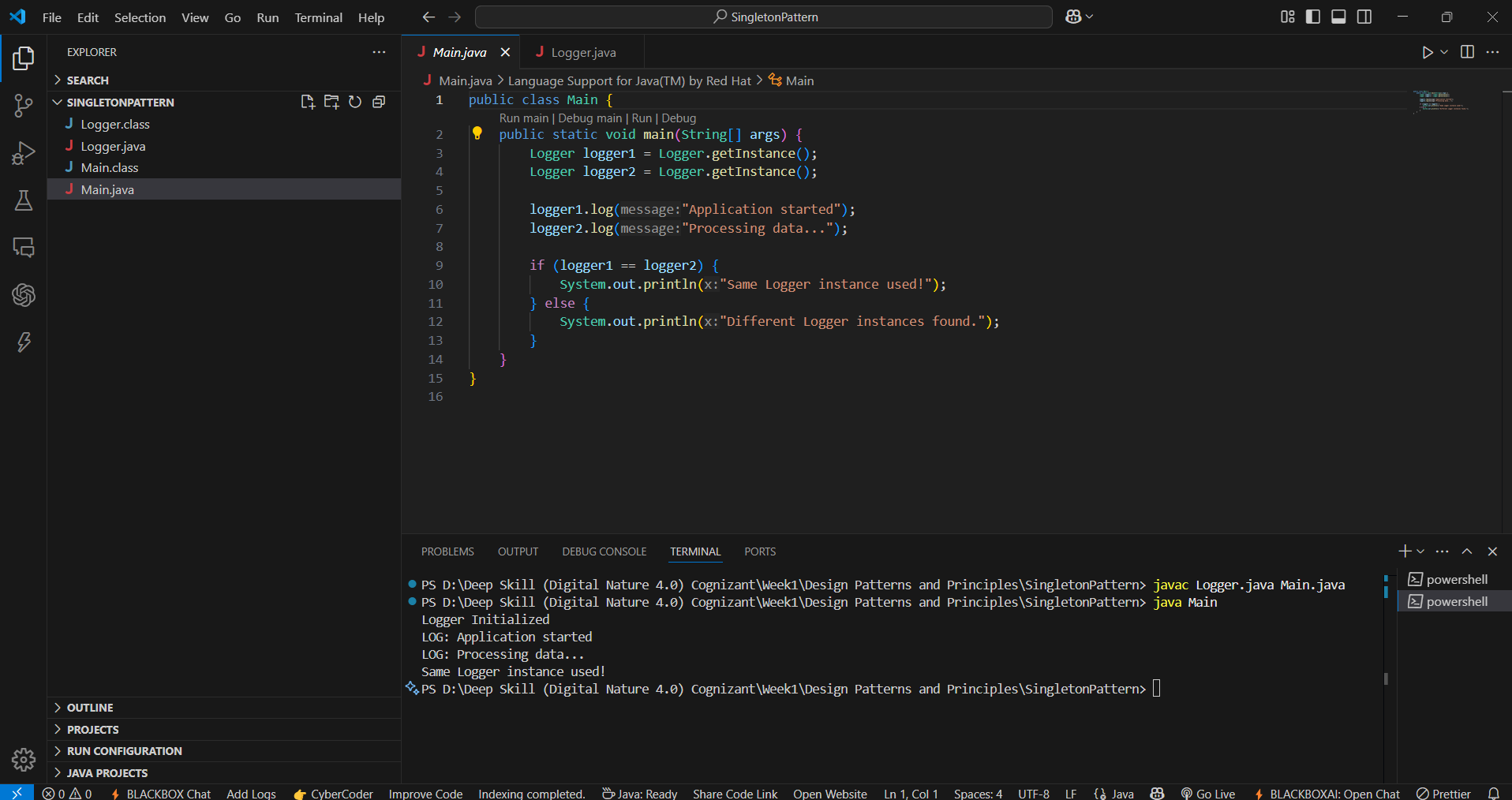
public class Logger {  
 private static Logger instance;  
  
 private Logger() {  
 System.out.println("Logger Initialized");  
 }  
  
 public static Logger getInstance() {  
 if (instance == null) {  
 instance = new Logger();  
 }  
 return instance;  
 }  
  
 public void log(String message) {  
 System.out.println("LOG: " + message);  
 }  
}

### Main.java

public class Main {  
 public static void main(String[] args) {  
 Logger logger1 = Logger.getInstance();  
 Logger logger2 = Logger.getInstance();  
  
 logger1.log("Application started");  
 logger2.log("Processing data...");  
  
 if (logger1 == logger2) {  
 System.out.println("Same Logger instance used!");  
 } else {  
 System.out.println("Different Logger instances found.");  
 }  
 }  
}

## 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 named FactoryMethodPatternExample.

2. Define interfaces or abstract classes for different document types such as WordDocument, PdfDocument, and ExcelDocument.

3. Implement concrete classes for each document type that implements or extends the above interfaces or abstract classes.

4. Create an abstract class DocumentFactory with a method createDocument().

5. Create concrete factory classes for each document type that extends DocumentFactory and implements the createDocument() method.

6. Create a test class to demonstrate the creation of different document types using the factory method.

## Code Implementation

### Document.java

public interface Document {  
 void open();  
}

### WordDocument.java

public class WordDocument implements Document {  
 public void open() {  
 System.out.println("Opening Word Document");  
 }  
}

### PdfDocument.java

public class PdfDocument implements Document {  
 public void open() {  
 System.out.println("Opening PDF Document");  
 }  
}

### ExcelDocument.java

public class ExcelDocument implements Document {  
 public void open() {  
 System.out.println("Opening Excel Document");  
 }  
}

### DocumentFactory.java

public abstract class DocumentFactory {  
 public abstract Document createDocument();  
}

### WordDocumentFactory.java

public class WordDocumentFactory extends DocumentFactory {  
 public Document createDocument() {  
 return new WordDocument();  
 }  
}

### PdfDocumentFactory.java

public class PdfDocumentFactory extends DocumentFactory {  
 public Document createDocument() {  
 return new PdfDocument();  
 }  
}

### ExcelDocumentFactory.java

public class ExcelDocumentFactory extends DocumentFactory {  
 public Document createDocument() {  
 return new ExcelDocument();  
 }  
}

### Main.java

public class Main {  
 public static void main(String[] args) {  
 DocumentFactory wordFactory = new WordDocumentFactory();  
 DocumentFactory pdfFactory = new PdfDocumentFactory();  
 DocumentFactory excelFactory = new ExcelDocumentFactory();  
  
 Document word = wordFactory.createDocument();  
 Document pdf = pdfFactory.createDocument();  
 Document excel = excelFactory.createDocument();  
  
 word.open();  
 pdf.open();  
 excel.open();  
 }  
}

### Output

