**EXERCISE1: 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:
   * Create a new Java project named SingletonPatternExample.
2. Define a Singleton Class:
   * Create a class named Logger that has a private static instance of itself.
   * Ensure the constructor of Logger is private.
   * Provide a public static method to get the instance of the Logger class.
3. Implement the Singleton Pattern:
   * Write code to ensure that the Logger class follows the Singleton design pattern.
4. Test the Singleton Implementation:
   * Create a test class to verify that only one instance of Logger is created and used across the application.

# Singleton Pattern

The Singleton Pattern ensures that a class has only one instance and provides a global point of access to it. This pattern is commonly used for resources like loggers, configuration managers, and database connections.  
  
Benefits:  
- Ensures a single instance across the application  
- Reduces memory usage  
- Useful for shared resources  
  
Common Pitfalls:  
- Thread safety (consider synchronized or eager initialization for multithreaded applications)

**Steps:**

# 1. Create a New Java Project

Create a new Java project named `SingletonPattern`.

# 2. Define a Singleton Class

Create a class named `Logger`:  
- It should have a private static instance of itself.  
- The constructor should be private to prevent external instantiation.  
- Provide a public static method `getInstance()` to access the single instance.

# 3. Implement the Singleton Pattern

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

# 4. Test the Singleton Implementation

Create a test class to verify that only one instance of Logger is created:  
  
public class TestLogger {  
 public static void main(String[] args) {  
 Logger logger1 = Logger.getInstance();  
 Logger logger2 = Logger.getInstance();  
  
 logger1.log("This is the first log message.");  
 logger2.log("This is the second log message.");  
  
 System.out.println("logger1 and logger2 are the same instance: " + (logger1 == logger2));  
 }  
}

# Sample Output

# 

**EXERCISE 2: IMPLEMENTATION OF 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:
   * Create a new Java project named FactoryMethodPatternExample.
2. Define Document Classes:
   * Create interfaces or abstract classes for different document types such as WordDocument, PdfDocument, and ExcelDocument.
3. Create Concrete Document Classes:
   * Implement concrete classes for each document type that implements or extends the above interfaces or abstract classes.
4. Implement the Factory Method:
   * Create an abstract class DocumentFactory with a method createDocument().
   * Create concrete factory classes for each document type that extends DocumentFactory and implements the createDocument() method.
5. Test the Factory Method Implementation:
   * Create a test class to demonstrate the creation of different document types using the factory method.

# Theory: Factory Method Pattern

The Factory Method Pattern defines an interface for creating an object but allows subclasses to alter the type of objects that will be created.  
  
Benefits:  
- Promotes loose coupling between client code and object creation logic.  
- Easier to extend the system with new types.  
  
Usage:  
- When the client code needs to work with abstract types and object creation needs to be delegated to subclasses.

**Steps:**

# 1. Create a New Java Project

Create a new Java project named `FactoryMethodPatternExample`.

# 2. Define Document Classes

Create an interface `Document` to represent the base type of all documents.  
  
public interface Document {  
 void open();  
}

# 3. Create Concrete Document Classes

Implement concrete classes for each document type:  
  
public class WordDocument implements Document {  
 public void open() {  
 System.out.println("Opening Word document");  
 }  
}  
  
public class PdfDocument implements Document {  
 public void open() {  
 System.out.println("Opening PDF document");  
 }  
}  
  
public class ExcelDocument implements Document {  
 public void open() {  
 System.out.println("Opening Excel document");  
 }  
}

# 4. Implement the Factory Method

Create an abstract factory class:  
  
public abstract class DocumentFactory {  
 public abstract Document createDocument();  
}  
  
Create concrete factories for each type:  
  
public class WordFactory extends DocumentFactory {  
 public Document createDocument() {  
 return new WordDocument();  
 }  
}  
  
public class PdfFactory extends DocumentFactory {  
 public Document createDocument() {  
 return new PdfDocument();  
 }  
}  
  
public class ExcelFactory extends DocumentFactory {  
 public Document createDocument() {  
 return new ExcelDocument();  
 }  
}

# 5. Test the Factory Method Implementation

Test class:  
  
public class FactoryTest {  
 public static void main(String[] args) {  
 DocumentFactory wordFactory = new WordFactory();  
 Document wordDoc = wordFactory.createDocument();  
 wordDoc.open();  
  
 DocumentFactory pdfFactory = new PdfFactory();  
 Document pdfDoc = pdfFactory.createDocument();  
 pdfDoc.open();  
  
 DocumentFactory excelFactory = new ExcelFactory();  
 Document excelDoc = excelFactory.createDocument();  
 excelDoc.open();  
 }  
}

# Output:

# 