## ****EXERCISE 1: Singleton Pattern Implementation****

### ****Objective:****

Demonstrate how to implement a thread-safe Singleton class in C#, ensuring only one instance of a class is used throughout the application.

### ****C# Implementation:****

using System;

namespace SingletonPattern

{

// A sealed Singleton class to restrict inheritance

public sealed class Logger

{

// Static readonly instance created at compile time

private static readonly Logger instance = new Logger();

// Private constructor to prevent direct instantiation

private Logger() { }

// Public accessor for the single instance

public static Logger Instance

{

get { return instance; }

}

// Logging method

public void Log(string message)

{

Console.WriteLine("LOG: " + message);

}

}

class Program

{

static void Main(string[] args)

{

Logger loggerA = Logger.Instance;

Logger loggerB = Logger.Instance;

loggerA.Log("This message is logged using the Singleton logger.");

if (object.ReferenceEquals(loggerA, loggerB))

{

Console.WriteLine("Confirmed: loggerA and loggerB refer to the same Singleton instance.");

}

else

{

Console.WriteLine("Error: Multiple instances detected. Singleton pattern broken.");

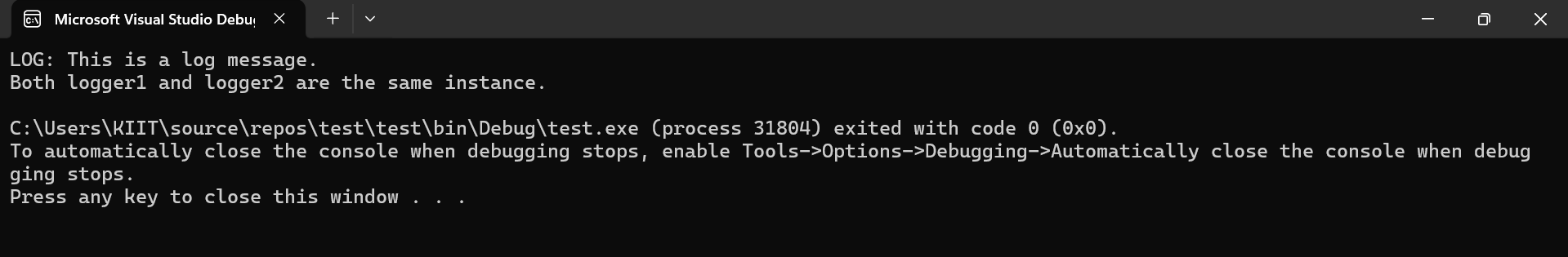
}

}

}

}

### ****Expected Output:****



### ****Concept Recap:****

* Singleton ensures a class has only **one instance globally**.
* Useful in scenarios like logging, configuration management, or caching.
* Implemented using a **private constructor** and a **static instance** reference.

## ****EXERCISE 2: Factory Method Pattern Implementation****

### ****Objective:****

Implement the Factory Method design pattern to instantiate different types of document classes (Word, PDF, Excel) without modifying the client code.

### ****C# Implementation:****

using System;

namespace FactoryMethodPatternExample

{

// Interface that defines document behavior

public interface IDocument

{

void Open();

void Close();

}

// Concrete implementations of documents

public class WordDocument : IDocument

{

public void Open() => Console.WriteLine("Opening a Word Document...");

public void Close() => Console.WriteLine("Closing the Word Document.");

}

public class PdfDocument : IDocument

{

public void Open() => Console.WriteLine("Opening a PDF Document...");

public void Close() => Console.WriteLine("Closing the PDF Document.");

}

public class ExcelDocument : IDocument

{

public void Open() => Console.WriteLine("Opening an Excel Document...");

public void Close() => Console.WriteLine("Closing the Excel Document.");

}

// Abstract Factory class

public abstract class DocumentFactory

{

public abstract IDocument CreateDocument();

}

// Specific factories for each document type

public class WordDocumentFactory : DocumentFactory

{

public override IDocument CreateDocument() => new WordDocument();

}

public class PdfDocumentFactory : DocumentFactory

{

public override IDocument CreateDocument() => new PdfDocument();

}

public class ExcelDocumentFactory : DocumentFactory

{

public override IDocument CreateDocument() => new ExcelDocument();

}

// Client application

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Document Creator System");

Console.WriteLine("========================");

// Factory instances

DocumentFactory wordFactory = new WordDocumentFactory();

DocumentFactory pdfFactory = new PdfDocumentFactory();

DocumentFactory excelFactory = new ExcelDocumentFactory();

// Using the factories to create and operate on documents

Console.WriteLine("\n[Word Document]");

IDocument word = wordFactory.CreateDocument();

word.Open();

word.Close();

Console.WriteLine("\n[PDF Document]");

IDocument pdf = pdfFactory.CreateDocument();

pdf.Open();

pdf.Close();

Console.WriteLine("\n[Excel Document]");

IDocument excel = excelFactory.CreateDocument();

excel.Open();

excel.Close();

Console.WriteLine("\nProgram execution complete. Press any key to exit.");

Console.ReadKey();

}

}

}

### ****Expected Output:****



### ****Concept Recap:****

* The Factory Method Pattern helps in **creating objects without exposing the instantiation logic**.
* Encourages **loose coupling** and follows the **Open-Closed Principle**.
* Makes the system easily extendable when new document types are introduced.