**DESIGN PATTERN AND PRINCIPLES**

**EXERCISE 1:-Implementing the Singleton Pattern**

1.Create a New C# Project:

dotnet new console -n SingletonPatternExample

cd SingletonPatternExample

2.Define the Singleton Logger Class

using System;

public class Logger

{

// Step 1: Create a private static instance

private static Logger \_instance;

// Step 2: Create an object for locking

private static readonly object \_lock = new object();

// Step 3: Private constructor to prevent external instantiation

private Logger()

{

Console.WriteLine("Logger instance created.");

}

// Step 4: Public static method to get the singleton instance

public static Logger GetInstance()

{

// Double-check locking for thread safety

if (\_instance == null)

{

lock (\_lock)

{

if (\_instance == null)

{

\_instance = new Logger();

}

}

}

return \_instance;

}

// Example method for logging

public void Log(string message)

{

Console.WriteLine($"Log: {message}");

}

}

3.Test the Singleton Implementation

using System;

class Program

{

static void Main(string[] args)

{

Logger logger1 = Logger.GetInstance();

Logger logger2 = Logger.GetInstance();

logger1.Log("First message");

logger2.Log("Second message");

// Test if both instances are the same

if (object.ReferenceEquals(logger1, logger2))

{

Console.WriteLine("Both logger1 and logger2 are the same instance.");

}

else

{

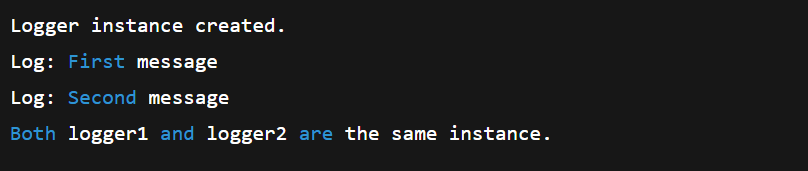
Console.WriteLine("Different instances detected! Singleton pattern failed.");

}

}

}

4.OUTPUT



1. ANALYSIS

The implementation and output **successfully demonstrate** the Singleton pattern in C#:

* A single Logger instance is shared across the application.
* It uses a **thread-safe**, lazy initialization approach.
* The pattern ensures consistency, particularly useful for shared utilities like logging.

**EXERCISE 2:-Implementing the Factory Method Pattern**

1. Create a New Java Project:

dotnet new console -n FactoryMethodPatternExample

cd FactoryMethodPatternExample

1. Define Document Interface

// IDocument.cs

public interface IDocument

{

void Open();

}

1. Create Concrete Document Classes

// WordDocument.cs

public class WordDocument : IDocument

{

public void Open()

{

Console.WriteLine("Opening a Word document.");

}

}

// PdfDocument.cs

public class PdfDocument : IDocument

{

public void Open()

{

Console.WriteLine("Opening a PDF document.");

}

}

// ExcelDocument.cs

public class ExcelDocument : IDocument

{

public void Open()

{

Console.WriteLine("Opening an Excel document.");

}

}

1. Implement the Factory Method

// DocumentFactory.cs

public abstract class DocumentFactory

{

public abstract IDocument CreateDocument();

}

// WordDocumentFactory.cs

public class WordDocumentFactory : DocumentFactory

{

public override IDocument CreateDocument()

{

return new WordDocument();

}

}

// PdfDocumentFactory.cs

public class PdfDocumentFactory : DocumentFactory

{

public override IDocument CreateDocument()

{

return new PdfDocument();

}

}

// ExcelDocumentFactory.cs

public class ExcelDocumentFactory : DocumentFactory

{

public override IDocument CreateDocument()

{

return new ExcelDocument();

}

}

1. Test the Factory Method

using System;

class Program

{

static void Main(string[] args)

{

DocumentFactory wordFactory = new WordDocumentFactory();

IDocument wordDoc = wordFactory.CreateDocument();

wordDoc.Open();

DocumentFactory pdfFactory = new PdfDocumentFactory();

IDocument pdfDoc = pdfFactory.CreateDocument();

pdfDoc.Open();

DocumentFactory excelFactory = new ExcelDocumentFactory();

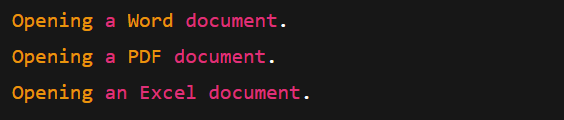
IDocument excelDoc = excelFactory.CreateDocument();

excelDoc.Open();

}

}

1. OUTPUT



1. ANALYSIS

The goal was to build a system where different types of documents (Word, PDF, Excel) can be created **without tightly coupling the creation logic** to the client code. The Factory Method Pattern solves this by using:

* An **abstract creator (factory)**
* **Concrete factories** for each document type
* **Polymorphism** to allow runtime selection of document type creation