**Design Principles & Patterns**

**Exercise 1: Implementing the Singleton Pattern**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SingletonPatternExample

{

public sealed class Logger

{

private static Logger instance = null;

private static readonly object padlock = new object();

private Logger()

{

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

}

public static Logger Instance

{

get

{

if (instance == null)

{

lock (padlock)

{

if (instance == null)

{

instance = new Logger();

}

}

}

return instance;

}

}

public void Log(string message)

{

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

}

}

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Testing Singleton Logger...\n");

Logger logger1 = Logger.Instance;

logger1.Log("First message");

Logger logger2 = Logger.Instance;

logger2.Log("Second message");

if (ReferenceEquals(logger1, logger2))

{

Console.WriteLine("\nLogger is a Singleton. Both instances are the same.");

}

else

{

Console.WriteLine("\nLogger is NOT a Singleton. Instances are different.");

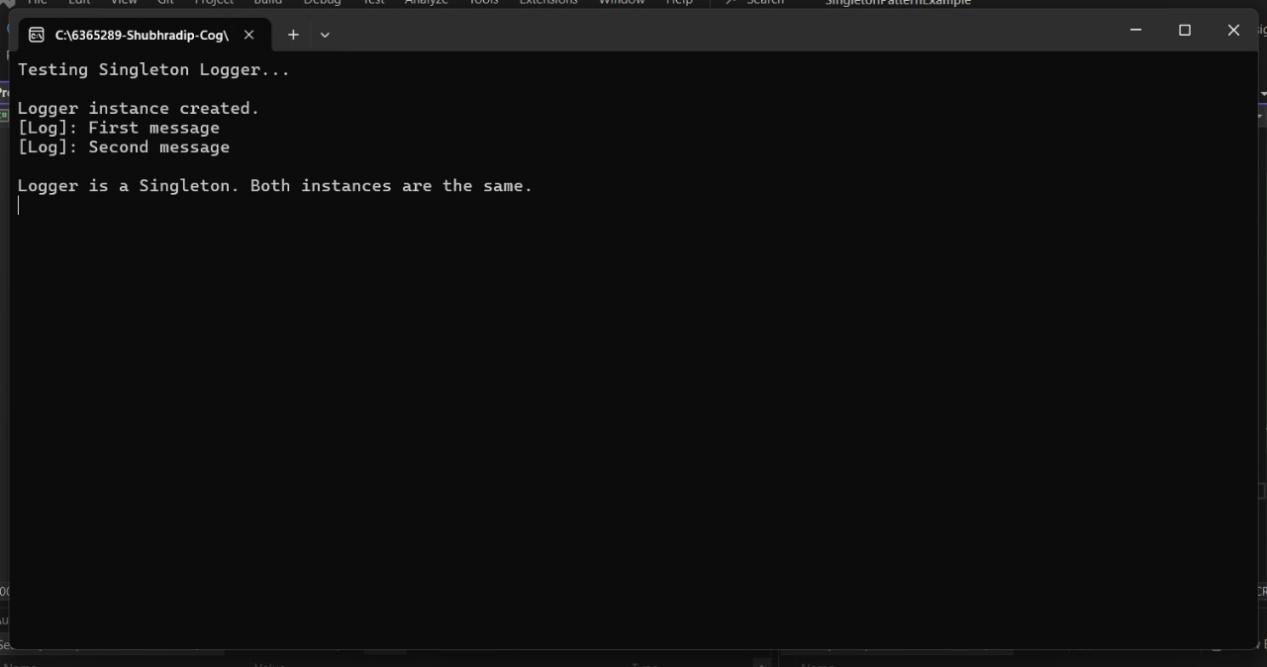
}

Console.ReadLine();

}

}

}



**Exercise 2: Implementing the Factory Method Pattern**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace FactoryMethodPatternExample

{

public interface IDocument

{

void Open();

}

public class WordDocument : IDocument

{

public void Open()

{

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

}

}

public class PdfDocument : IDocument

{

public void Open()

{

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

}

}

public class ExcelDocument : IDocument

{

public void Open()

{

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

}

}

public abstract class DocumentFactory

{

public abstract IDocument CreateDocument();

}

public class WordFactory : DocumentFactory

{

public override IDocument CreateDocument()

{

return new WordDocument();

}

}

public class PdfFactory : DocumentFactory

{

public override IDocument CreateDocument()

{

return new PdfDocument();

}

}

public class ExcelFactory : DocumentFactory

{

public override IDocument CreateDocument()

{

return new ExcelDocument();

}

}

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Factory Method Pattern Demo\n");

DocumentFactory wordFactory = new WordFactory();

IDocument wordDoc = wordFactory.CreateDocument();

wordDoc.Open();

DocumentFactory pdfFactory = new PdfFactory();

IDocument pdfDoc = pdfFactory.CreateDocument();

pdfDoc.Open();

DocumentFactory excelFactory = new ExcelFactory();

IDocument excelDoc = excelFactory.CreateDocument();

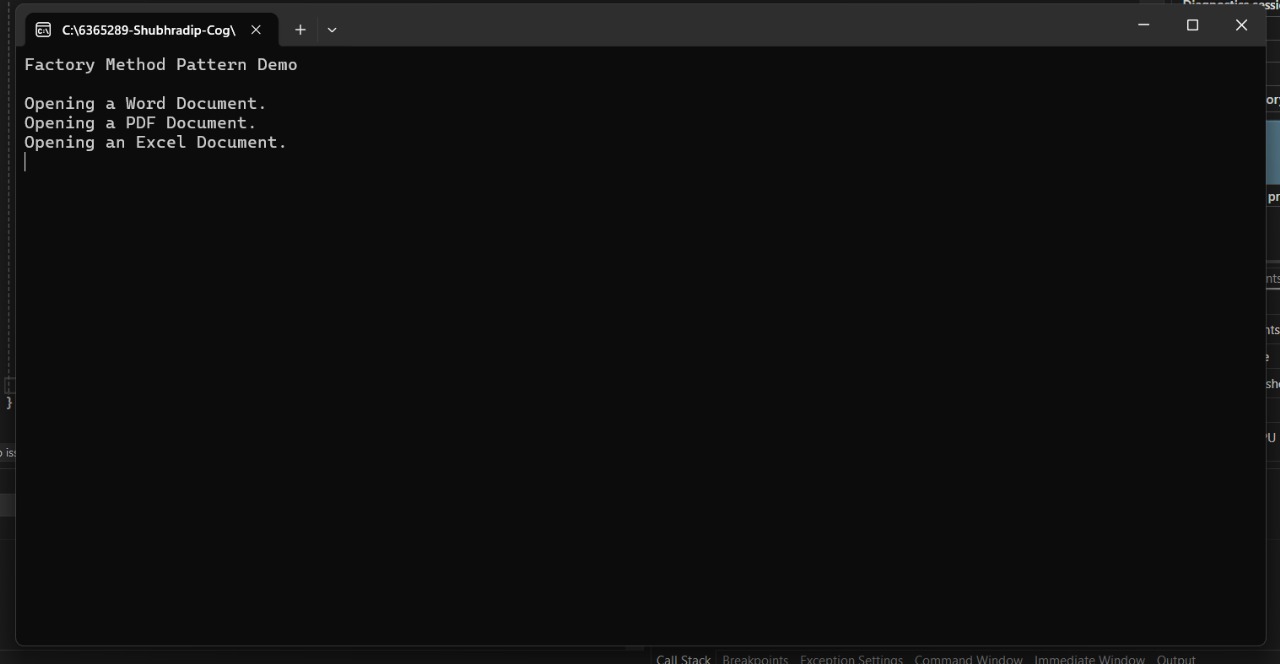
excelDoc.Open();

Console.ReadLine();

}

}

}



**Data Structures & Algorithm**

**Exercise 2: E-commerce Platform Search Function**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace E\_Commerce\_Platform

{

// Step 1: Explain Asymptotic Notation

/\*

\* Big O notation describes the performance (time or space) of an algorithm as the input size grows.

\* Big O notation is very important in analyzing the program .

\* - Analysis of Linear Search: O(n)

\* Best case: O(1) (first element match)

\* Average case: O(n/2)

\* Worst case: O(n) (last element or not found)

\*

\* - Analysis of Binary Search: O(log n)

\* Requires sorted array

\* Best case: O(1) (middle match)

\* Average/Worst case: O(log n)

\*/

public class Product

{

public int ProductId { get; set; }

public string ProductName { get; set; }

public string Category { get; set; }

public Product(int id, string name, string category)

{

ProductId = id;

ProductName = name;

Category = category;

}

public override string ToString()

{

return $"ID: {ProductId}, Name: {ProductName}, Category: {Category}";

}

}

class Program

{

static Product LinearSearch(Product[] products, int productId)

{

foreach (var product in products)

{

if (product.ProductId == productId)

return product;

}

return null;

}

static Product BinarySearch(Product[] products, int productId)

{

int left = 0;

int right = products.Length - 1;

while (left <= right)

{

int mid = left + (right - left) / 2;

if (products[mid].ProductId == productId)

return products[mid];

else if (products[mid].ProductId < productId)

left = mid + 1;

else

right = mid - 1;

}

return null;

}

static void Main(string[] args)

{

Product[] products = new Product[]

{

new Product(104, "Laptop", "Electronics"),

new Product(101, "Shirt", "Clothing"),

new Product(103, "Smartphone", "Electronics"),

new Product(102, "Book", "Education"),

new Product(105, "Shoes", "Footwear")

};

Console.WriteLine("Linear Search for Product ID 103:");

Product result1 = LinearSearch(products, 103);

Console.WriteLine(result1 != null ? result1.ToString() : "Product not found");

Array.Sort(products, (a, b) => a.ProductId.CompareTo(b.ProductId));

Console.WriteLine("\nBinary Search for Product ID 103:");

Product result2 = BinarySearch(products, 103);

Console.WriteLine(result2 != null ? result2.ToString() : "Product not found");

Console.WriteLine("\n--- Time Complexity Analysis ---");

Console.WriteLine("Linear Search: O(n) - Suitable for small/unsorted datasets.");

Console.WriteLine("Binary Search: O(log n) - Much faster, but requires sorted data.");

Console.WriteLine("Conclusion: Binary Search is more suitable for large-scale platforms like e-commerce, where fast lookup is critical.");

Console.ReadLine();

}

}

}



**Exercise 7: Financial Forecasting**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Financial\_Forecasting

{

class Program

{

/\* Recursion Explanation:

\* Recursion is a programming technique where a method calls itself to solve smaller subproblems of the problem.

\* It's useful for problems that can be broken down into similar subproblems, like calculating growth over time.

\* FutureValue(years) = PresentValue \* (1 + GrowthRate) ^ years \*/

static double CalculateFutureValue(double presentValue, double growthRate, int years)

{

if (years == 0)

return presentValue;

return (1 + growthRate) \* CalculateFutureValue(presentValue, growthRate, years - 1);

}

static double CalculateFutureValueMemo(double presentValue, double growthRate, int years, double[] memo)

{

if (years == 0)

return presentValue;

if (memo[years] != 0)

return memo[years];

memo[years] = (1 + growthRate) \* CalculateFutureValueMemo(presentValue, growthRate, years - 1, memo);

return memo[years];

}

static void Main(string[] args)

{

double presentValue = 10000;

double growthRate = 0.10;

int years = 5;

Console.WriteLine("📈 Financial Forecasting Tool Using Recursion\n");

double futureValue = CalculateFutureValue(presentValue, growthRate, years);

Console.WriteLine($"Future Value (Year {years}) using Recursion: ₹{futureValue:F2}");

double[] memo = new double[years + 1];

double futureValueOptimized = CalculateFutureValueMemo(presentValue, growthRate, years, memo);

Console.WriteLine($"Future Value (Year {years}) using Memoization: ₹{futureValueOptimized:F2}");

Console.WriteLine("\n🔍 Analysis:");

Console.WriteLine("- Recursive Time Complexity: O(n)");

Console.WriteLine("- With Memoization: Still O(n), but avoids repeated calculations.");

Console.WriteLine("- For better performance in production, iteration or closed formulas are preferred.");

Console.ReadLine();

}

}

}

