**WEEK 1 HANDS ON**

Submitted by:Simran Raghav(6362006)

**DESIGN PATTERNS AND PRINCIPLES**

**Exercise 1: Implementing the Singleton Pattern**

CODE:

using System;

namespace SingletonPatternExample

{

    public class Logger

    {

        private static Logger? instance;

        private Logger()

        {

            Console.WriteLine("Logger initialized.");

        }

        public static Logger GetInstance()

        {

            if (instance == null)

            {

                instance = new Logger();

            }

            return instance;

        }

        public void Log(string message)

        {

            Console.WriteLine("[LOG] " + message);

        }

    }

    class Program

    {

        static void Main()

        {

            Logger logger1 = Logger.GetInstance();

            logger1.Log("Message from logger1");

            Logger logger2 = Logger.GetInstance();

            logger2.Log("Message from logger2");

            if (logger1 == logger2)

                Console.WriteLine("✅ Singleton confirmed: logger1 and logger2 are the same instance.");

            else

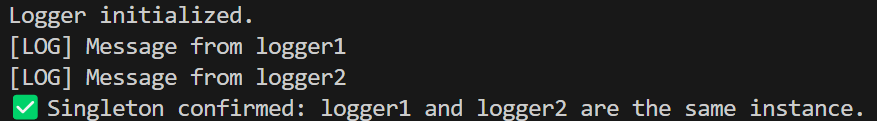
                Console.WriteLine("❌ Singleton failed: logger1 and logger2 are different.");

        }

    }

}

OUTPUT:



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

CODE:

using System;

namespace FactoryMethodPatternExample

{

    public abstract class Document

    {

        public abstract string Type { get; }

        public void Open()

        {

            Console.WriteLine($"Opening {Type} document...");

        }

        public void Save()

        {

            Console.WriteLine($"Saving {Type} document...");

        }

    }

    public class WordDocument : Document

    {

        public override string Type => "Word";

    }

    public class PdfDocument : Document

    {

        public override string Type => "PDF";

    }

    public class ExcelDocument : Document

    {

        public override string Type => "Excel";

    }

    public abstract class DocumentFactory

    {

        public abstract Document CreateDocument();

    }

    public class WordFactory : DocumentFactory

    {

        public override Document CreateDocument()

        {

            return new WordDocument();

        }

    }

    public class PdfFactory : DocumentFactory

    {

        public override Document CreateDocument()

        {

            return new PdfDocument();

        }

    }

    public class ExcelFactory : DocumentFactory

    {

        public override Document CreateDocument()

        {

            return new ExcelDocument();

        }

    }

    class Program

    {

        static void Main()

        {

            DocumentFactory[] factories = new DocumentFactory[]

            {

                new WordFactory(),

                new PdfFactory(),

                new ExcelFactory()

            };

            foreach (var factory in factories)

            {

                Document doc = factory.CreateDocument();

                Console.WriteLine($"Created a {doc.Type} document using {factory.GetType().Name}");

                doc.Open();

                doc.Save();

                Console.WriteLine();

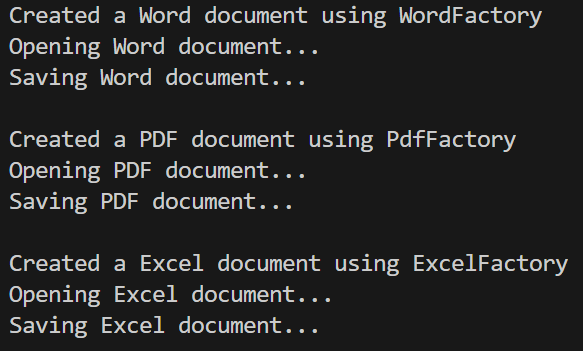
            }

        }

    }

}

OUTPUT:



**DATA STRUCTURES AND ALGORITHMS**

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

CODE:

using System;

using System.Collections.Generic;

public class Product

{

    public int ProductId { get; set; }

    public string ProductName { get; set; }

    public string Category { get; set; }

    public Product(int productId, string productName, string category)

    {

        ProductId = productId;

        ProductName = productName;

        Category = category;

    }

    public override string ToString()

    {

        return $"{ProductId} | {ProductName} | {Category}";

    }

}

class Program

{

    public static Product? LinearSearch(List<Product> products, int targetId)

    {

        foreach (var product in products)

        {

            if (product.ProductId == targetId)

                return product;

        }

        return null;

    }

    public static Product? BinarySearch(List<Product> products, int targetId)

    {

        int low = 0, high = products.Count - 1;

        while (low <= high)

        {

            int mid = low + (high - low) / 2;

            int midId = products[mid].ProductId;

            if (midId == targetId)

                return products[mid];

            else if (targetId < midId)

                high = mid - 1;

            else

                low = mid + 1;

        }

        return null;

    }

    static void Main()

    {

        var catalog = new List<Product>

        {

            new Product(17, "Running Shoes", "Sports"),

            new Product(5, "Headphones", "Electronics"),

            new Product(12, "Wallet", "Accessories"),

            new Product(33, "Laptop Stand", "Office"),

            new Product(1, "Yoga Mat", "Fitness")

        };

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

        var linearResult = LinearSearch(catalog, 12);

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

        Console.WriteLine("\n=== Binary Search ===");

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

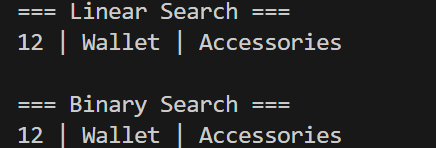
        var binaryResult = BinarySearch(catalog, 12);

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

    }

}

OUTPUT:



**Exercise 7: Financial Forecasting**

CODE:

using System;

namespace FinancialForecasting

{

    public class Program

    {

        static void Main()

        {

            double[] pastValues = { 100, 110, 121, 133.1 };

            int futurePeriods = 3;

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

            Console.WriteLine("Past Values: " + string.Join(", ", pastValues));

            Console.WriteLine($"Predicting value after {futurePeriods} future periods...\n");

            double predictedValue = Predict(pastValues, futurePeriods);

            Console.WriteLine($"Predicted future value after {futurePeriods} periods: {predictedValue:F4}");

        }

        public static double Predict(double[] pastValues, int periods)

        {

            if (pastValues.Length < 2)

                throw new ArgumentException("Need at least two past data points.");

            double sumRatios = 0;

            for (int i = 1; i < pastValues.Length; i++)

                sumRatios += pastValues[i] / pastValues[i - 1];

          double avgGrowth = sumRatios / (pastValues.Length - 1);

            return ProjectRecursively(pastValues[^1], avgGrowth, periods);

        }

        private static double ProjectRecursively(double lastValue, double growth, int t)

        {

            if (t == 0)

                return lastValue;

          return ProjectRecursively(lastValue, growth, t - 1) \* growth;

        }

    }

}

OUTPUT:

