WEEK 1

Exercise 1

Singleton Pattern

using System;

namespace SingletonPattern

{

public class Logger

{

private static Logger instance = null;

private Logger()

{

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

}

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(string[] args)

{

var logger1 = Logger.GetInstance();

logger1.Log("First message");

var logger2 = Logger.GetInstance();

logger2.Log("Second message");

if (logger1 == logger2)

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

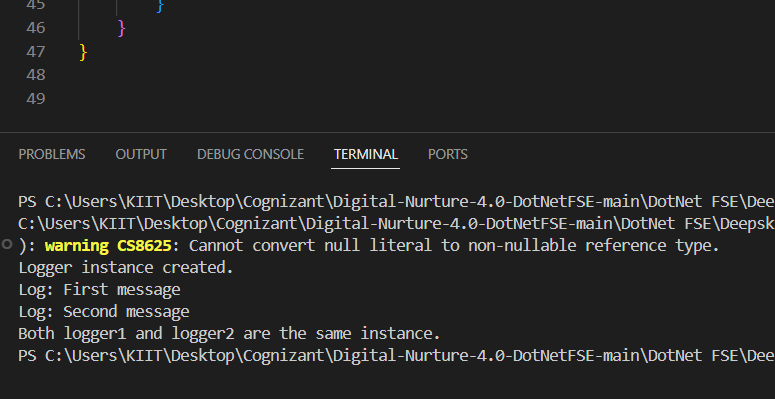
else

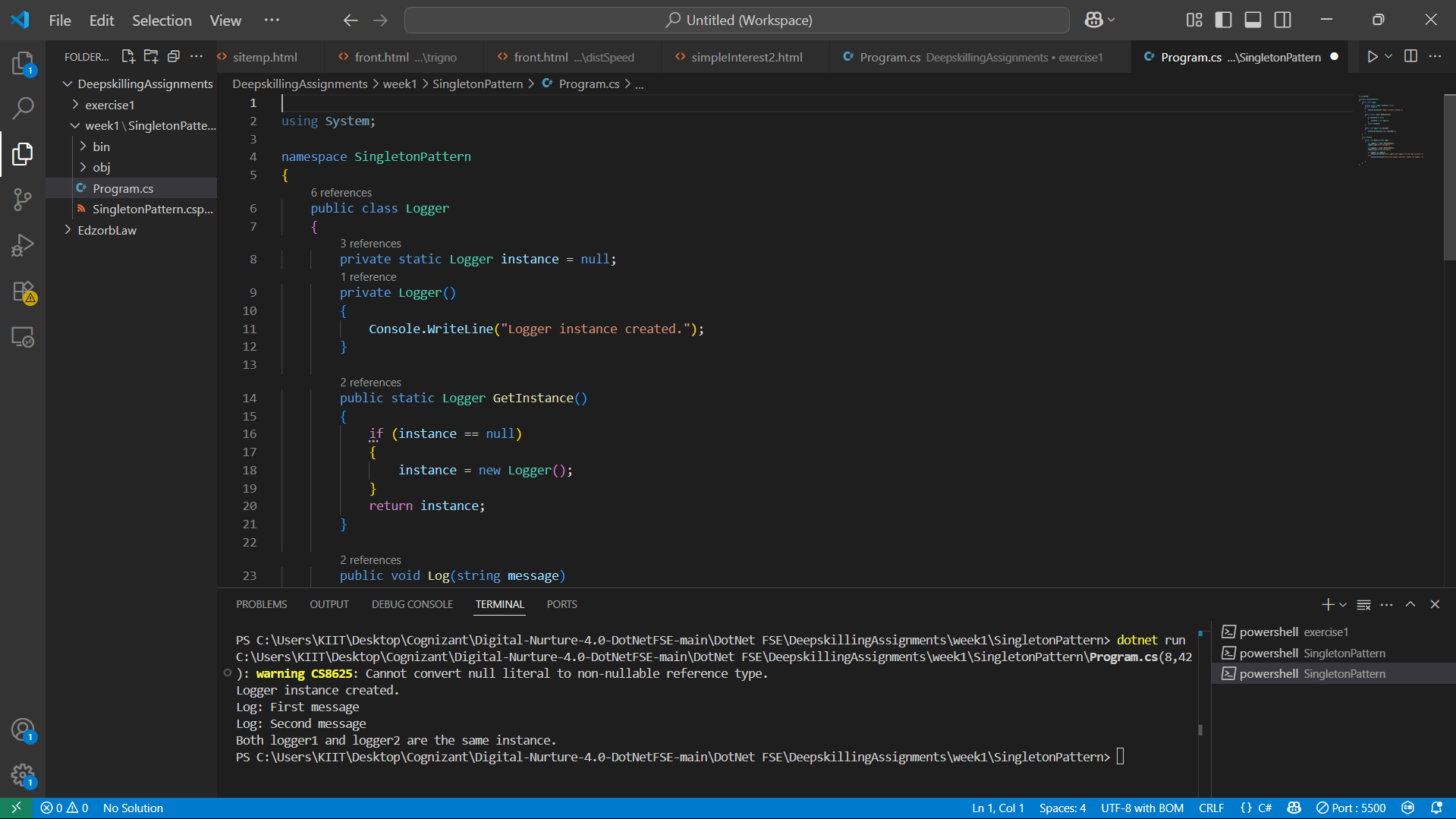
Console.WriteLine("Different logger instances (should not happen).");

}

}

}





Exercise 2

FactoryMethod

using System;

namespace FactoryMethodPattern

{

    // Document interface

    public interface IDocument

    {

        void Open();

    }

    // Concrete document types

    public class WordDocument : IDocument

    {

        public void Open()

        {

            Console.WriteLine("Word document opened.");

        }

    }

    public class PdfDocument : IDocument

    {

        public void Open()

        {

            Console.WriteLine("PDF document opened.");

        }

    }

    public class ExcelDocument : IDocument

    {

        public void Open()

        {

            Console.WriteLine("Excel document opened.");

        }

    }

    public abstract class DocumentFactory

    {

        public abstract IDocument CreateDocument();

    }

    // Concrete factories

    public class WordDocumentFactory : DocumentFactory

    {

        public override IDocument CreateDocument()

        {

            return new WordDocument();

        }

    }

    public class PdfDocumentFactory : DocumentFactory

    {

        public override IDocument CreateDocument()

        {

            return new PdfDocument();

        }

    }

    public class ExcelDocumentFactory : DocumentFactory

    {

        public override IDocument CreateDocument()

        {

            return new ExcelDocument();

        }

    }

TEST :

class Program

    {

        static void Main(string[] args)

        {

            DocumentFactory factory;

            // Create Word document

            factory = new WordDocumentFactory();

            IDocument wordDoc = factory.CreateDocument();

            wordDoc.Open();

            // Create PDF document

            factory = new PdfDocumentFactory();

            IDocument pdfDoc = factory.CreateDocument();

            pdfDoc.Open();

            // Create Excel document

            factory = new ExcelDocumentFactory();

            IDocument excelDoc = factory.CreateDocument();

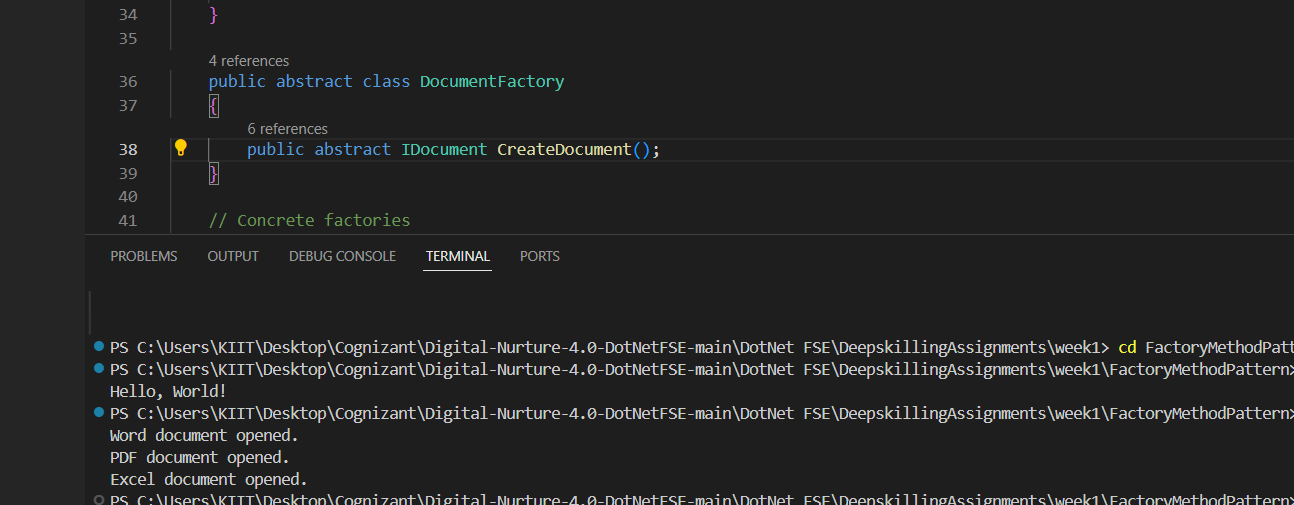
            excelDoc.Open();

        }

    }

}

**OUTPUT:**



Exercise 3

Ecommerce Product Search

using System;

using System.Linq;

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;

    }

}

class Program

{

    static void Main(string[] args)

    {

        Product[] products = new Product[]

        {

            new Product(101, "Laptop", "Electronics"),

            new Product(203, "Shoes", "Apparel"),

            new Product(150, "Watch", "Accessories"),

            new Product(120, "Phone", "Electronics")

        };

        Console.WriteLine(" Linear Search for ProductId 150:");

        var linearResult = LinearSearch(products, 150);

        if (linearResult != null)

            Console.WriteLine($"Found: {linearResult.ProductName} in {linearResult.Category}");

        else

            Console.WriteLine("Product not found.");

        var sortedProducts = products.OrderBy(p => p.ProductId).ToArray();

        Console.WriteLine("\n Binary Search for ProductId 150:");

        var binaryResult = BinarySearch(sortedProducts, 150);

        if (binaryResult != null)

            Console.WriteLine($"Found: {binaryResult.ProductName} in {binaryResult.Category}");

        else

            Console.WriteLine("Product not found.");

    }

    public static Product LinearSearch(Product[] products, int targetId)

    {

        foreach (var product in products)

        {

            if (product.ProductId == targetId)

            {

                return product;

            }

        }

        return null;

    }

    public static Product BinarySearch(Product[] products, int targetId)

    {

        int left = 0;

        int right = products.Length - 1;

        while (left <= right)

        {

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

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

            {

                return products[mid];

            }

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

            {

                left = mid + 1;

            }

            else

            {

                right = mid - 1;

            }

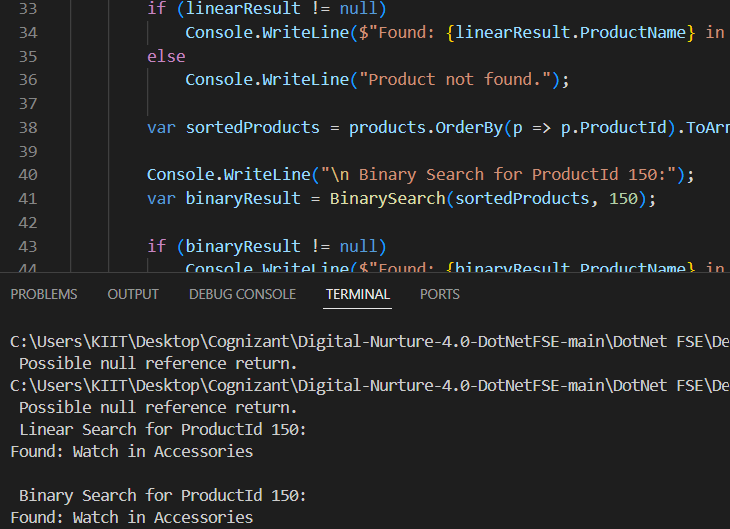
        }

        return null;

    }

}

**OUTPUT:**



Exercise 4

Financial Prediction

using System;

class Program

{

    static void Main(string[] args)

    {

        double initialValue = 10000;   // Example: Rs. 10000

        double growthRate = 2.5;     // 2.5% growth per period

        int periods = 5;              // Predict for 5 periods

        double futureValueBasic = ForecastRecursive(initialValue, growthRate, periods);

        Console.WriteLine($"Future Value (basic recursion) after {periods} periods: {futureValueBasic:F2}");

        double futureValueOptimized = ForecastOptimizedRecursive(initialValue, growthRate, periods);

        Console.WriteLine($"Future Value (optimized recursion) after {periods} periods: {futureValueOptimized:F2}");

    }

    // Basic recursive method

    static double ForecastRecursive(double initialValue, double growthRate, int periods)

    {

        if (periods == 0)

            return initialValue;

        return ForecastRecursive(initialValue, growthRate, periods - 1) \* (1 + growthRate);

    }

    //  Optimized recursive method

    static double ForecastOptimizedRecursive(double initialValue, double growthRate, int periods)

    {

        return initialValue \* PowerRecursive(1 + growthRate, periods);

    }

    static double PowerRecursive(double baseValue, int exponent)

    {

        if (exponent == 0)

            return 1;

        if (exponent % 2 == 0)

        {

            double halfPower = PowerRecursive(baseValue, exponent / 2);

            return halfPower \* halfPower;

        }

        else

        {

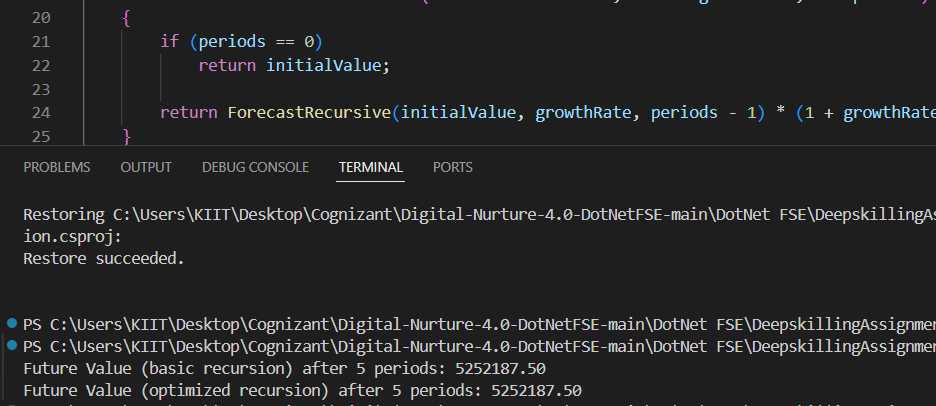
            return baseValue \* PowerRecursive(baseValue, exponent - 1);

        }

    }

}

**OUTPUT:**



**Explanation:**

We make the recursion better by using a method called **exponentiation by squaring**. Instead of multiplying the value for each period one after another, this method breaks the problem into smaller, smarter parts. If the number of periods is even, it finds the result for half the periods and then multiplies that by itself. If the number of periods is odd, it does one extra multiplication and then keeps going. This way, the program doesn’t need to repeat the same work so many times. It makes far fewer calls — not one for every period, but just a few — so it runs much faster, especially when we have many periods. In simple words, we help the computer do the job in bigger, faster steps instead of small, slow ones.