# **WEEK 1: Hands-On – Design Patterns and Principles & Algorithms\_Data Structures**

## **Exercise 1: Implementing the Singleton Pattern**

### **Code:**

using System;

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 SingletonPattern

{

    static void Main(string[] args)

    {

        Logger logger1=Logger.GetInstance();

        Logger logger2=Logger.GetInstance();

        logger1.Log("User logged in.");

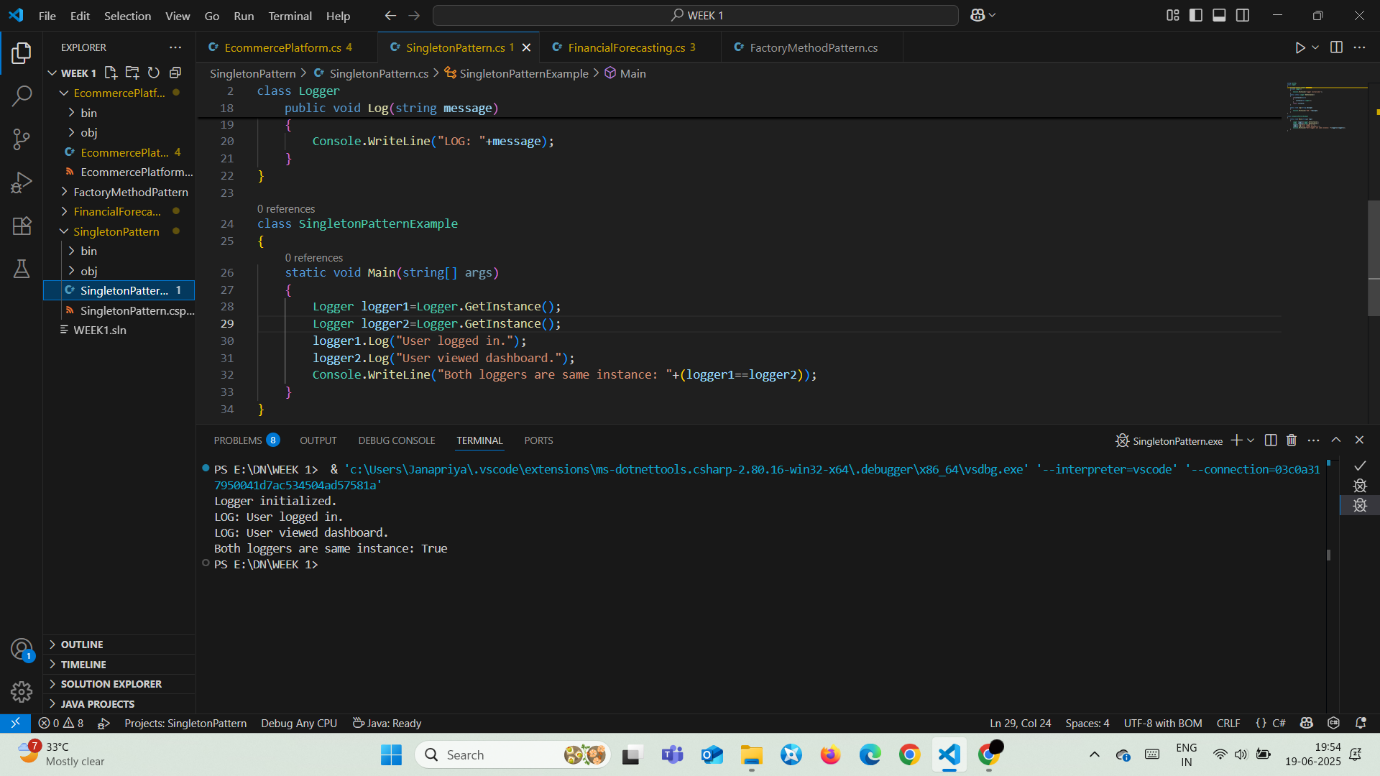
        logger2.Log("User viewed dashboard.");

        Console.WriteLine("Both loggers are same instance: "+(logger1==logger2));

    }

}

### **Output:**



### **Explanation:**

The singleton pattern ensures only one instance of the Logger class exists using a private constructor and a static method, maintaining consistent logging throughout the app.  
It prevents multiple object creation, saving memory and ensuring all parts of the system use the same logger without conflict.

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

### **Code:**

using System;

interface IDoc

{

    void Open();

}

class WordDoc:IDoc

{

    public void Open()

    {

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

    }

}

class PdfDoc:IDoc

{

    public void Open()

    {

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

    }

}

class ExcelDoc:IDoc

{

    public void Open()

    {

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

    }

}

abstract class DocumentFactory

{

    public abstract IDoc CreateDocument();

}

class WordFactory:DocumentFactory

{

    public override IDoc CreateDocument()

    {

        return new WordDoc();

    }

}

class PdfFactory:DocumentFactory

{

    public override IDoc CreateDocument()

    {

        return new PdfDoc();

    }

}

class ExcelFactory:DocumentFactory

{

    public override IDoc CreateDocument()

    {

        return new ExcelDoc();

    }

}

class FactoryMethodPattern

{

    static void Main(string[] args)

    {

        DocumentFactory word = new WordFactory();

        IDoc wordDoc = word.CreateDocument();

        wordDoc.Open();

        DocumentFactory pdf = new PdfFactory();

        IDoc pdfDoc = pdf.CreateDocument();

        pdfDoc.Open();

        DocumentFactory excel = new ExcelFactory();

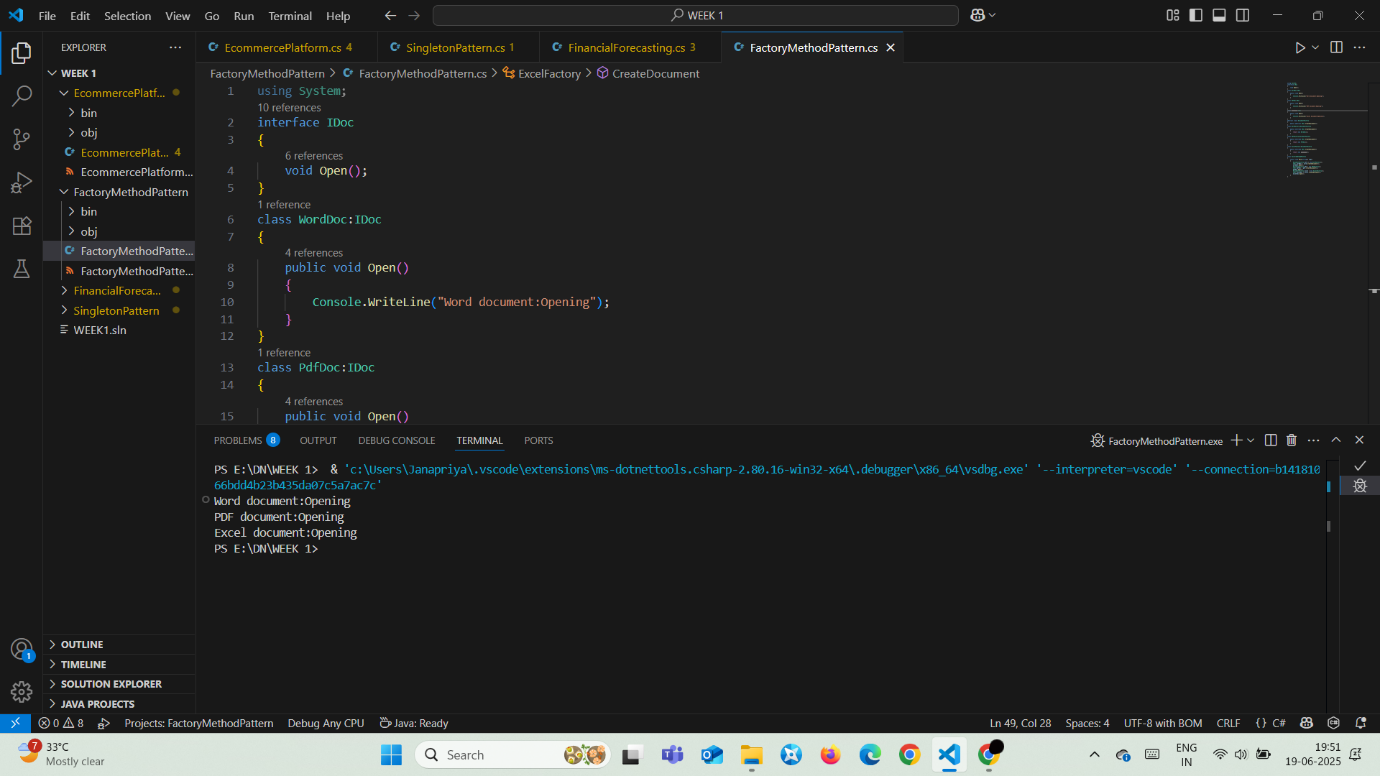
        IDoc excelDoc = excel.CreateDocument();

        excelDoc.Open();

    }

}

### **Output:**



### **Explanation:**

The factory method pattern is used to create different document types (Word, PDF, Excel) using abstract and concrete factories, promoting flexibility and scalability.  
This decouples object creation from the client code, allowing easy addition of new document types without modifying existing logic.

## **E-commerce Platform Search Function**

### **Code:**

using System;

class Product

{

    public int ProductId;

    public string ProductName;

    public string Category;

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

    {

        ProductId = productId;

        ProductName = productName;

        Category = category;

    }

    public string ConvertString()

    {

        return $"Product {ProductId}\nProduct Name: {ProductName}\nCategory: {Category}";

    }

}

class Search

{

    public static Product LinearSearch(Product[] products, string prod)

    {

        foreach (var product in products)

        {

            if (product.ProductName.Equals(prod, StringComparison.OrdinalIgnoreCase))

            {

                return product;

            }

        }

        return null;

    }

    public static Product BinarySearch(Product[] products, string prod)

    {

        int l = 0, r = products.Length - 1;

        while (l <= r)

        {

            int mid = l + (r - l) / 2;

            int check = string.Compare(products[mid].ProductName, prod, StringComparison.OrdinalIgnoreCase);

            if (check == 0) return products[mid];

            else if (check < 0) l = mid + 1;

            else r = mid - 1;

        }

        return null;

    }

    static void Main(string[] args)

    {

        Product[] products =

        {

            new Product(1, "Pen", "Stationery"),

            new Product(2, "Notebook", "Stationery"),

            new Product(3, "Mobile Phone", "Electronics"),

            new Product(4, "TV", "Electronics"),

            new Product(5, "Chair", "Furniture"),

            new Product(6, "Sofa", "Furniture"),

            new Product(7, "Shirt", "Clothing"),

            new Product(8, "Skirt", "Clothing")

        };

        Console.WriteLine("Enter product name to search:");

        string productName = Console.ReadLine();

        Product search = LinearSearch(products, productName);

        if (search != null)

        {

            Console.WriteLine("Linear Search:");

            Console.WriteLine(search.ConvertString());

        }

        else

        {

            Console.WriteLine("Product not found using Linear Search.");

        }

        Console.WriteLine();

        Array.Sort(products, (a, b) => string.Compare(a.ProductName, b.ProductName, StringComparison.OrdinalIgnoreCase));

        search = BinarySearch(products, productName);

        if (search != null)

        {

            Console.WriteLine("Binary Search:");

            Console.WriteLine(search.ConvertString());

        }

        else

        {

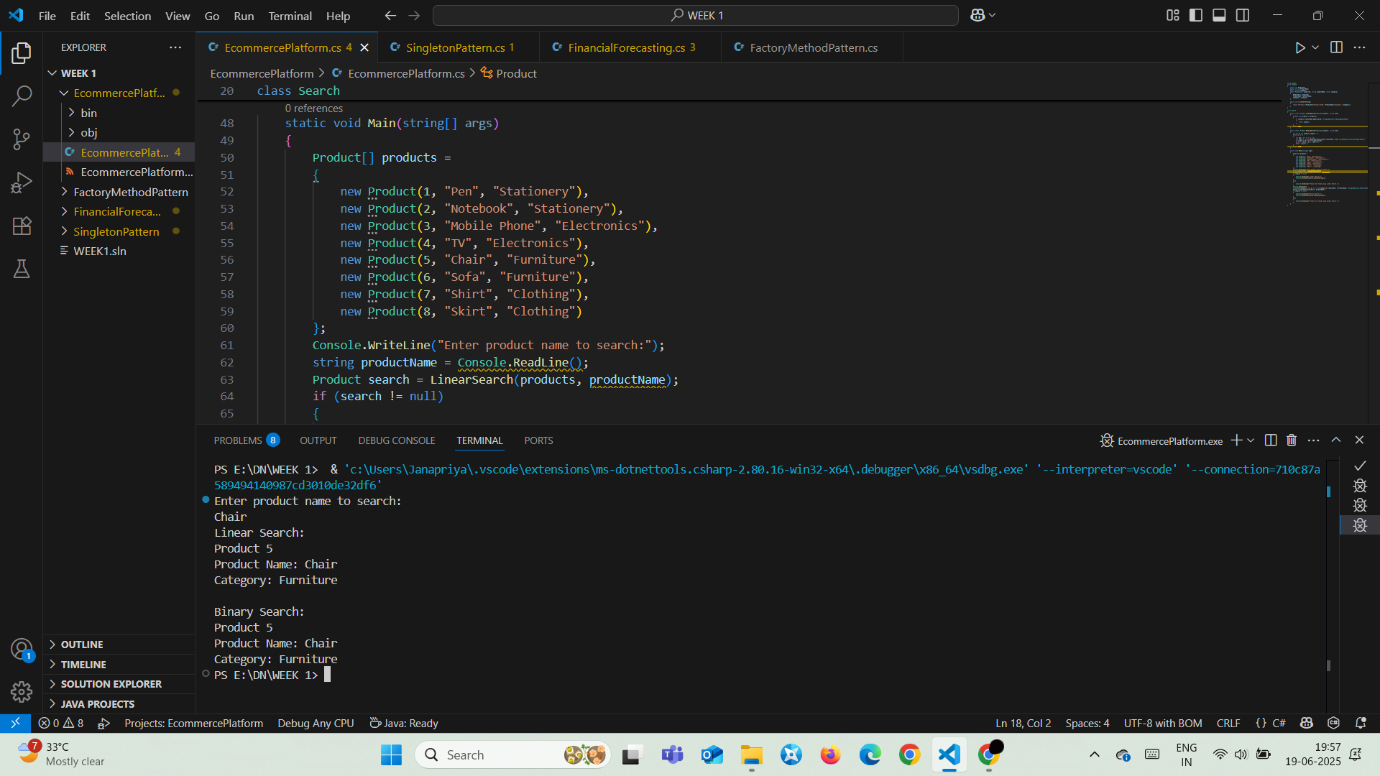
            Console.WriteLine("Product not found using Binary Search.");

        }

    }

}

### **Output:**



### **Explanation:**

The product search program uses both linear and binary search to find a product by name, comparing performance between unsorted and sorted arrays.  
Linear search is simple but slow (O(n)), while binary search is faster (O(log n)) after sorting; binary is ideal for large, stable datasets.

## **Financial Forecasting**

### **Code:**

using System;

class FinancialForecasting

{

    public static double CalculateFutureValue(double currentVal, double rate, int years)

    {

        if (years==0)

            return currentVal;

        return CalculateFutureValue(currentVal\*(1+rate), rate, years-1);

    }

    static void Main(string[] args)

    {

        Console.Write("Enter the starting amount: ");

        double startingAmount=double.Parse(Console.ReadLine());

        Console.Write("Enter the annual growth rate: ");

        double growthRate=double.Parse(Console.ReadLine())/100;

        Console.Write("Enter the number of years to forecast: ");

        int forecastYears=int.Parse(Console.ReadLine());

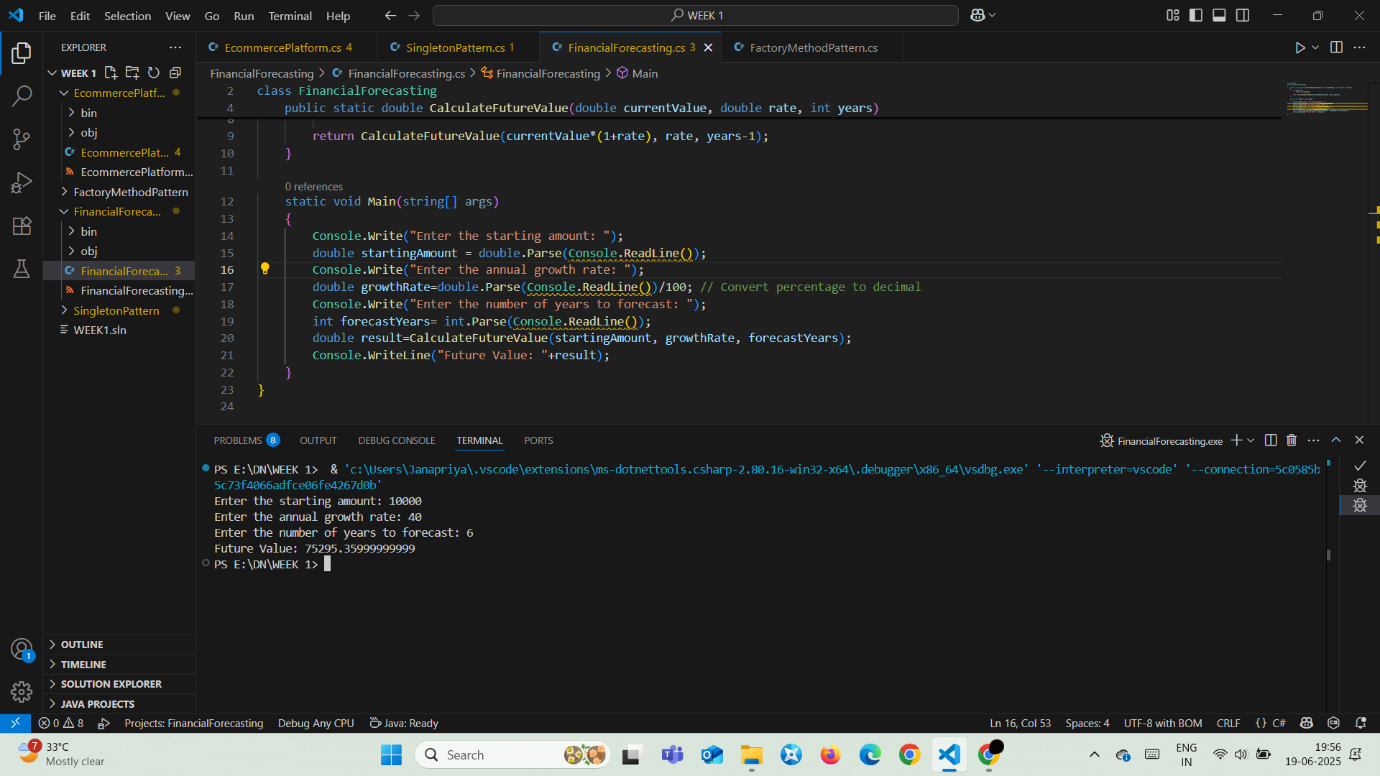
        double result=CalculateFutureValue(startingAmount, growthRate, forecastYears);

        Console.WriteLine("Future Value: "+result);

    }

}

### **Output:**



### **Explanation:**

The recursive method predicts future financial value by multiplying the current amount with (1 + rate) over a number of years.  
Though recursion expresses the logic elegantly, it uses more memory (O(n)); the iterative version is preferred for efficiency and large inputs.