**Design pattern and Principles**

**TASK 1 :** **Implementing the Singleton Pattern**

**SOURCE CODE**

**Program.cs**

using System;

namespace SingletonExample

{

// Singleton class

public sealed class Singleton

{

// Private static variable to hold the single instance

private static Singleton? \_instance = null;

// Lock object for thread safety

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

// Private constructor so no one can instantiate from outside

private Singleton()

{

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

}

// Public static property to get the instance

public static Singleton Instance

{

get

{

// Double-checked locking to make it thread safe

if (\_instance == null)

{

lock (\_lock)

{

if (\_instance == null)

{

\_instance = new Singleton();

}

}

}

return \_instance;

}

}

// A sample method to demonstrate functionality

public void ShowMessage()

{

Console.WriteLine("Hello from the Singleton instance!");

}

}

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Accessing Singleton instance first time:");

Singleton s1 = Singleton.Instance;

s1.ShowMessage();

Console.WriteLine("\nAccessing Singleton instance second time:");

Singleton s2 = Singleton.Instance;

s2.ShowMessage();

Console.WriteLine("\nAre both instances same? " + (s1 == s2));

// Wait for user input before closing

Console.WriteLine("\nPress any key to exit...");

Console.ReadKey();

}

}

}

**Task1. csproj**

<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>

    <OutputType>Exe</OutputType>

    <TargetFramework>net9.0</TargetFramework>

    <RootNamespace>Task\_1</RootNamespace>

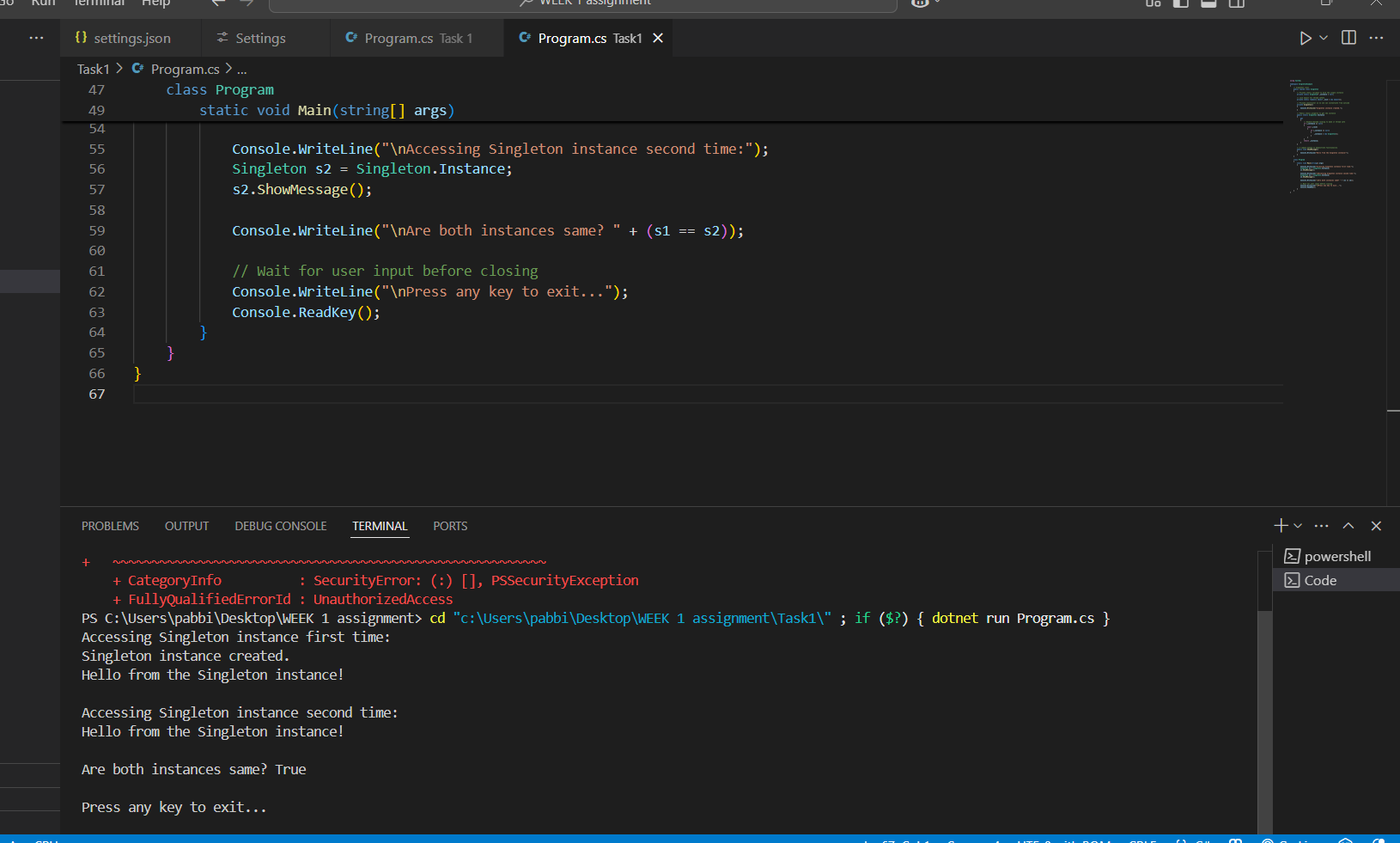
    <ImplicitUsings>enable</ImplicitUsings>

    <Nullable>enable</Nullable>

  </PropertyGroup>

</Project>

**Output**

****

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

**1. Document.cs (Abstract Product)**

// Document.cs

public abstract class Document

{

public abstract void Print();

}

**2. Report.cs (Concrete Product)**

// Report.cs

public class Report : Document

{

public override void Print()

{

Console.WriteLine("Printing Report document.");

}

}

**3. Invoice.cs (Concrete Product)**

// Invoice.cs

public class Invoice : Document

{

public override void Print()

{

Console.WriteLine("Printing Invoice document.");

}

}

**4. DocumentCreator.cs (Creator Abstract Class)**

// DocumentCreator.cs

public abstract class DocumentCreator

{

// Factory Method

public abstract Document CreateDocument();

// Business logic that uses the product

public void SomeOperation()

{

Document doc = CreateDocument();

doc.Print();

}

}

**5. ReportCreator.cs (Concrete Creator)**

// ReportCreator.cs

public class ReportCreator : DocumentCreator

{

public override Document CreateDocument()

{

return new Report();

}

}

**6. InvoiceCreator.cs (Concrete Creator)**

// InvoiceCreator.cs

public class InvoiceCreator : DocumentCreator

{

public override Document CreateDocument()

{

return new Invoice();

}

}

**7. Program.cs (Client)**

// Program.cs

using System;

class Program

{

static void Main(string[] args)

{

DocumentCreator creator;

creator = new ReportCreator();

creator.SomeOperation();

creator = new InvoiceCreator();

creator.SomeOperation();

Console.ReadLine();

}

}

**Project file** (**FactoryMethodPatternDemo.csproj**)

<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>

    <OutputType>Exe</OutputType>

    <TargetFramework>net9.0</TargetFramework>

    <RootNamespace>Task\_2</RootNamespace>

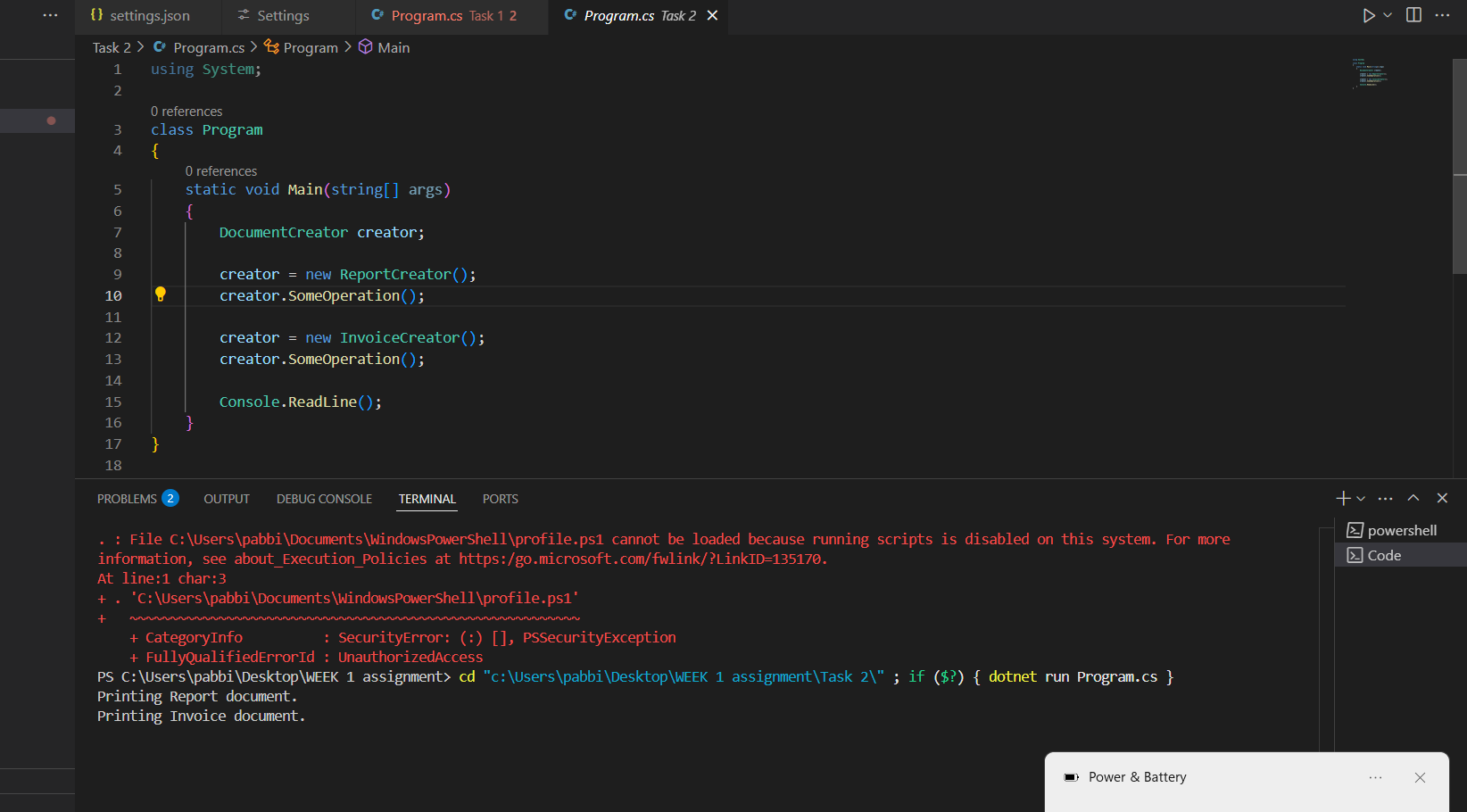
    <ImplicitUsings>enable</ImplicitUsings>

    <Nullable>enable</Nullable>

  </PropertyGroup>

</Project>

**Output**

****

**Data structures and Algorithms**

**Task 3: E-commerce Platform Search Function**

**Source code**

**Product.cs**

public class Product : IComparable<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;

    }

    // For sorting in binary search

    public int CompareTo(Product other)

    {

        if (other == null) return 1;

        return this.ProductId.CompareTo(other.ProductId);

    }

    public override string ToString()

    {

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

    }

}

**Program.cs**

using System;

class Program

{

    // Linear Search by productName

    static int LinearSearch(Product[] products, string targetName)

    {

        for (int i = 0; i < products.Length; i++)

        {

            if (products[i].ProductName.Equals(targetName, StringComparison.OrdinalIgnoreCase))

                return i;

        }

        return -1; // Not found

    }

    // Binary Search by productId (array must be sorted)

    static int BinarySearch(Product[] sortedProducts, int targetId)

    {

        int left = 0;

        int right = sortedProducts.Length - 1;

        while (left <= right)

        {

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

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

                return mid;

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

                left = mid + 1;

            else

                right = mid - 1;

        }

        return -1; // Not found

    }

    static void Main(string[] args)

    {

        // Unsorted products for linear search

        Product[] products = new Product[]

        {

            new Product(102, "Laptop", "Electronics"),

            new Product(205, "Shoes", "Apparel"),

            new Product(150, "Smartphone", "Electronics"),

            new Product(310, "Coffee Maker", "Home Appliances"),

            new Product(110, "T-Shirt", "Apparel")

        };

        // Linear Search example

        string searchName = "Smartphone";

        int linearIndex = LinearSearch(products, searchName);

        if (linearIndex != -1)

            Console.WriteLine($"Linear Search: Found '{searchName}' at index {linearIndex}. Details: {products[linearIndex]}");

        else

            Console.WriteLine($"Linear Search: '{searchName}' not found.");

        // Sort products by ProductId for binary search

        Array.Sort(products);

        // Binary Search example

        int searchId = 310;

        int binaryIndex = BinarySearch(products, searchId);

        if (binaryIndex != -1)

            Console.WriteLine($"Binary Search: Found product with ID {searchId} at index {binaryIndex}. Details: {products[binaryIndex]}");

        else

            Console.WriteLine($"Binary Search: Product with ID {searchId} not found.");

        Console.ReadLine();

    }

}

**Task3.csproj**

<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>

    <OutputType>Exe</OutputType>

    <TargetFramework>net9.0</TargetFramework>

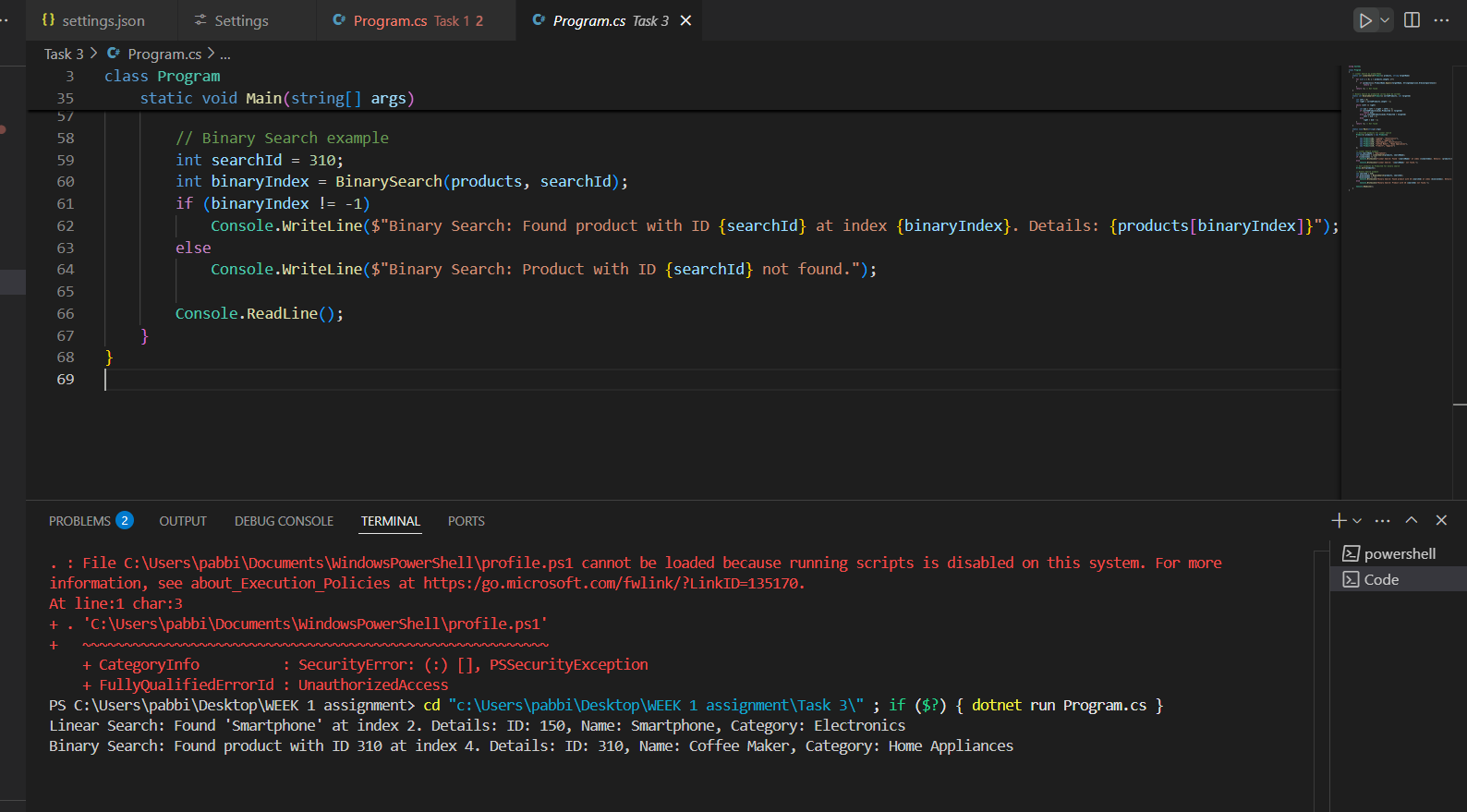
    <RootNamespace>Task\_3</RootNamespace>

    <ImplicitUsings>enable</ImplicitUsings>

    <Nullable>enable</Nullable>

  </PropertyGroup>

</Project>



**Task 4: Financial Forecasting**

**Source code**

**Program.cs**

using System;

class FinancialForecasting

{

    private double[] revenues;

    public FinancialForecasting(double[] historicalRevenues)

    {

        revenues = historicalRevenues;

    }

    private double CalculateAverageGrowthRate()

    {

        double totalGrowth = 0;

        int count = 0;

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

        {

            double growth = (revenues[i] - revenues[i - 1]) / revenues[i - 1];

            totalGrowth += growth;

            count++;

        }

        return totalGrowth / count;

    }

    public double[] Forecast(int quartersToPredict)

    {

        double[] forecasted = new double[quartersToPredict];

        double avgGrowth = CalculateAverageGrowthRate();

        double lastRevenue = revenues[revenues.Length - 1];

        ForecastRecursive(forecasted, 0, quartersToPredict, avgGrowth, lastRevenue);

        return forecasted;

    }

    private void ForecastRecursive(double[] forecasted, int currentQuarter, int totalQuarters, double growthRate, double previousRevenue)

    {

        if (currentQuarter == totalQuarters)

            return;

        double predictedRevenue = previousRevenue \* (1 + growthRate);

        forecasted[currentQuarter] = predictedRevenue;

        ForecastRecursive(forecasted, currentQuarter + 1, totalQuarters, growthRate, predictedRevenue);

    }

}

class Program

{

    static void Main()

    {

        double[] historicalRevenues = { 100, 110, 115, 120, 130, 140 };

        FinancialForecasting forecasting = new FinancialForecasting(historicalRevenues);

        int quartersToPredict = 4;

        double[] forecast = forecasting.Forecast(quartersToPredict);

        Console.WriteLine("Financial Forecast (next quarters):");

        for (int i = 0; i < forecast.Length; i++)

        {

            Console.WriteLine($"Quarter {i + 1}: {forecast[i]:F2} million");

        }

        Console.ReadLine();

    }

}

**Task4**.**csproj**

<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>

    <OutputType>Exe</OutputType>

    <TargetFramework>net9.0</TargetFramework>

    <RootNamespace>Task\_4</RootNamespace>

    <ImplicitUsings>enable</ImplicitUsings>

    <Nullable>enable</Nullable>

  </PropertyGroup>

</Project>

**Output**

