**Week-1: Design Principles And Patterns**

**Exercise 1: Implementing the Singleton Pattern**

**Program.cs**

**using System;**

**class Program**

**{**

**static void Main()**

**{**

**Logger logger1 = Logger.GetInstance();**

**logger1.Log("First log message");**

**Logger logger2 = Logger.GetInstance();**

**logger2.Log("Second log message");**

**Console.WriteLine(object.ReferenceEquals(logger1, logger2)**

**? "Both logger instances are the same (Singleton working)"**

**: "Different instances (Singleton failed)");**

**}**

**}**

**Logger.cs**

**using System;**

**public class Logger**

**{**

**private static Logger \_instance;**

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

**private Logger()**

**{**

**Console.WriteLine("Logger Initialized");**

**}**

**public static Logger GetInstance()**

**{**

**if (\_instance == null)**

**{**

**lock (\_lock)**

**{**

**if (\_instance == null)**

**{**

**\_instance = new Logger();**

**}**

**}**

**}**

**return \_instance;**

**}**

**public void Log(string message)**

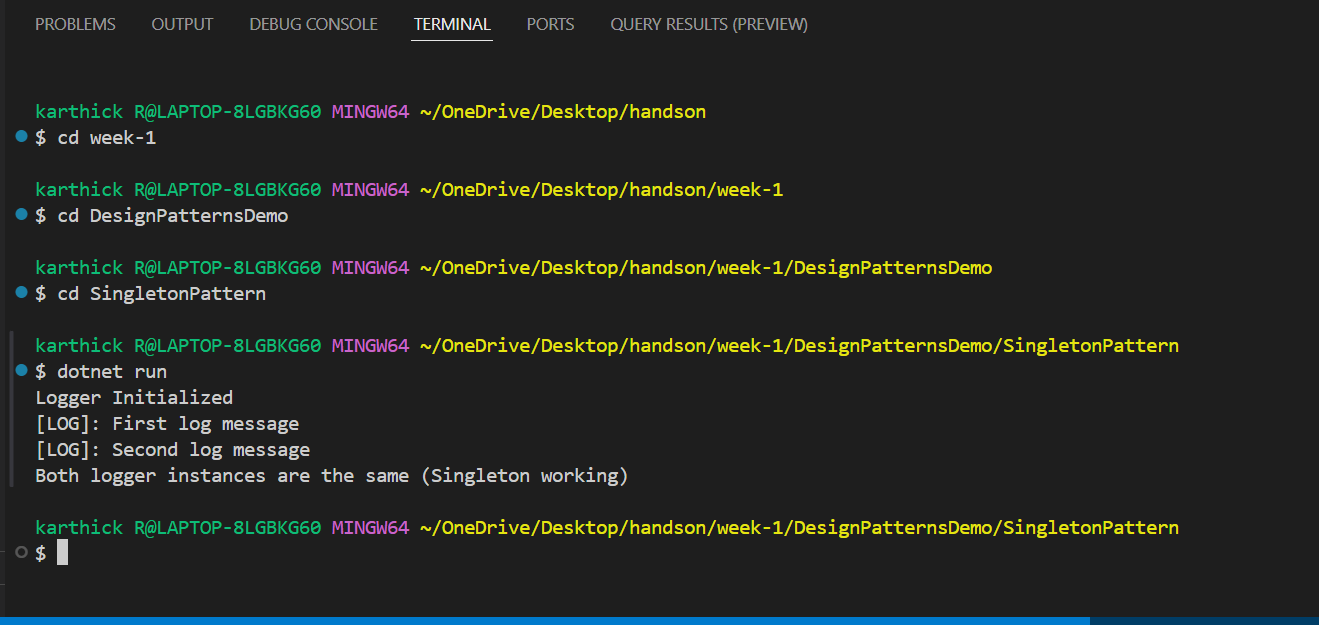
**{**

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

**}**

**}**

**OUTPUT:**

****

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

**Program.cs:**

**using System;**

**class Program**

**{**

**static void Main()**

**{**

**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();**

**}**

**}**

**IDocument.cs**

**public interface IDocument**

**{**

**void Open();**

**}**

**WordDocument.cs**

**using System;**

**public class WordDocument : IDocument**

**{**

**public void Open()**

**{**

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

**}**

**}**

**PdfDocument.cs**

**using System;**

**public class PdfDocument : IDocument**

**{**

**public void Open()**

**{**

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

**}**

**}**

**ExcelDocument.cs**

**using System;**

**public class ExcelDocument : IDocument**

**{**

**public void Open()**

**{**

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

**}**

**}**

**DocumentFactory.cs**

**public abstract class DocumentFactory**

**{**

**public abstract IDocument CreateDocument();**

**}**

**WordFactory.cs**

**public class WordFactory : DocumentFactory**

**{**

**public override IDocument CreateDocument()**

**{**

**return new WordDocument();**

**}**

**}**

**PdfFactory.cs**

**public class PdfFactory : DocumentFactory**

**{**

**public override IDocument CreateDocument()**

**{**

**return new PdfDocument();**

**}**

**}**

**ExcelFactory.cs**

**public class ExcelFactory : DocumentFactory**

**{**

**public override IDocument CreateDocument()**

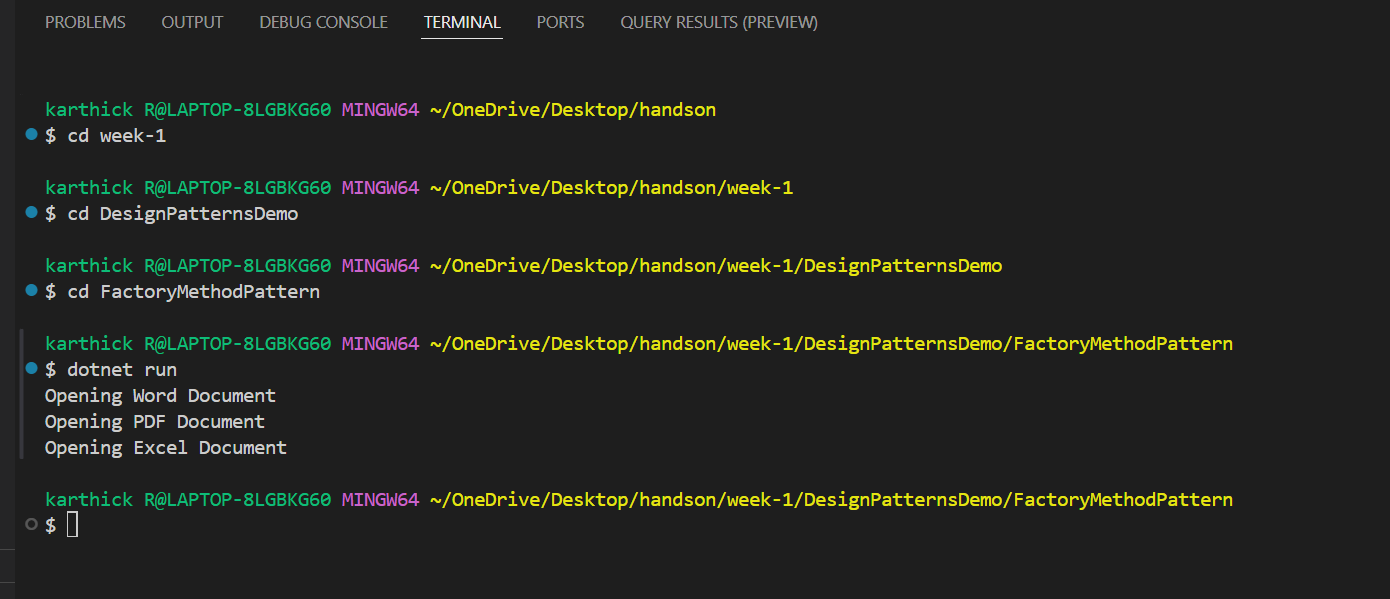
**{**

**return new ExcelDocument();**

**}**

**}**

**OUTPUT:**

****

# **Algorithms and Data Structures**

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

**using System;**

**using System.Linq;**

**class EcommerceSearchFunction**

**{**

**static void Main()**

**{**

**Product[] products = {**

**new Product(1, "Laptop", "Electronics"),**

**new Product(2, "Shoes", "Footwear"),**

**new Product(3, "Watch", "Accessories"),**

**new Product(4, "Camera", "Electronics"),**

**new Product(5, "Mobile", "Electronics")**

**};**

**Product[] sortedProducts = products.OrderBy(p => p.ProductName).ToArray();**

**string searchName = "Watch";**

**Console.WriteLine("Linear Search:");**

**Product result1 = LinearSearch(products, searchName);**

**DisplayResult(result1);**

**Console.WriteLine("\nBinary Search:");**

**Product result2 = BinarySearch(sortedProducts, searchName);**

**DisplayResult(result2);**

**Console.WriteLine("\nAnalysis:");**

**Console.WriteLine("Linear Search Time Complexity: O(n)");**

**Console.WriteLine("Binary Search Time Complexity: O(log n)");**

**Console.WriteLine("Binary Search is faster for large sorted data.");**

**}**

**static Product LinearSearch(Product[] products, string name)**

**{**

**foreach (Product product in products)**

**{**

**if (product.ProductName.Equals(name, StringComparison.OrdinalIgnoreCase))**

**{**

**return product;**

**}**

**}**

**return null;**

**}**

**static Product BinarySearch(Product[] products, string name)**

**{**

**int low = 0, high = products.Length - 1;**

**while (low <= high)**

**{**

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

**int compare = string.Compare(products[mid].ProductName, name, StringComparison.OrdinalIgnoreCase);**

**if (compare == 0)**

**return products[mid];**

**else if (compare < 0)**

**low = mid + 1;**

**else**

**high = mid - 1;**

**}**

**return null;**

**}**

**static void DisplayResult(Product product)**

**{**

**if (product != null)**

**{**

**Console.WriteLine($"Product Found: {product.ProductName}, Category: {product.Category}");**

**}**

**else**

**{**

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

**}**

**}**

**}**

**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;**

**}**

**}**

**OUTPUT:**

****

**Exercise 7: Financial Forecasting**

**using System;**

**class FinancialForecasting**

**{**

**static void Main()**

**{**

**double initialValue = 10000; // ₹10,000 investment**

**double growthRate = 0.08;    // 8% annual growth**

**int years = 5;               // Forecast for 5 years**

**double futureValue = ForecastRecursive(initialValue, growthRate, years);**

**Console.WriteLine($"Predicted Value after {years} years: ₹{futureValue:F2}");**

**}**

**static double ForecastRecursive(double currentValue, double rate, int yearsLeft)**

**{**

**if (yearsLeft == 0)**

**return currentValue; // Base case**

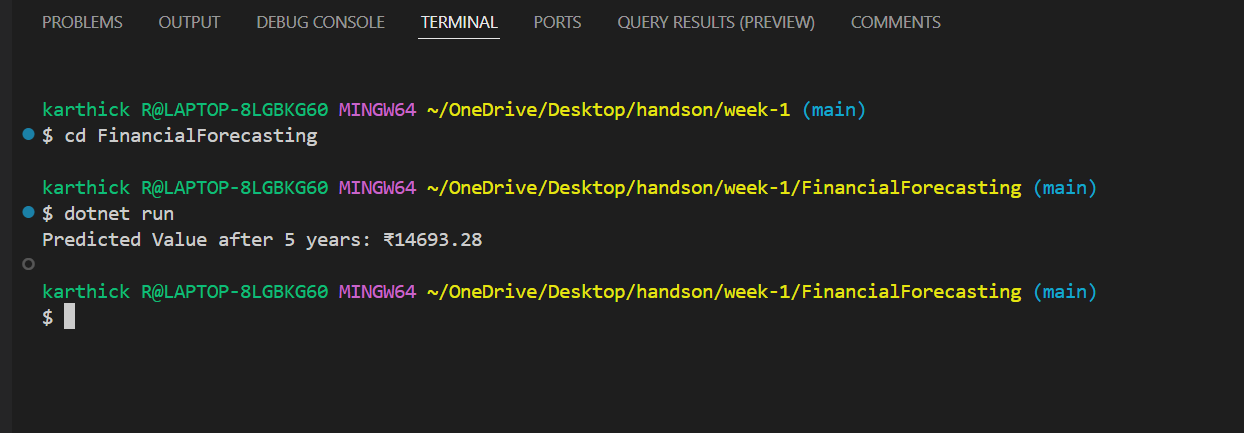
**// Recursive step**

**return ForecastRecursive(currentValue \* (1 + rate), rate, yearsLeft - 1);**

**}**

**}**

**OUTPUT:**

****

**How to Optimize Recursion?**

**Recursion is simple, but can be inefficient if:**

* **It repeats the same calculations (like Fibonacci).**
* **It goes too deep → stack overflow risk**

**In this case, it's already tail-recursive, and doesn’t recompute anything, so it’s efficient. But here are two ways to optimize if needed:**

**Use Iteration Instead (optional optimization):**

**static double ForecastIterative(double currentValue, double rate, int years)**

**{**

**for (int i = 0; i < years; i++)**

**{**

**currentValue \*= (1 + rate);**

**}**

**return currentValue;**

**}**

**This avoids recursion entirely and is faster for very large inputs.**