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

I create a logging utility that only one instance of the **“Logger”** class is used across the entire application. This will help maintain consistency in logging throughout your program.

#### **Step 1:- Create a New Project**

I created a Console Application named “**SingletonPatternExample”.**

#### **Step 2:- Create the Logger Class**

This class will be our singleton. It must prevent other classes from creating new instances, and it should provide a global access point.

**>>>>**[**loggers.cs**](http://loggers.cs)

**using System;**

**public class Logger**

**{**

**private static Logger instance;**

**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}");**

**}**

**}**

#### **Step 3:- Write the Test Code**

Now, I’ll verify that the same instance is used every time we access the Logger.

**>>>**[**Program.cs**](http://program.cs)

**using System;**

**class Program**

**{**

**static void Main(string[] args)**

**{**

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

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

**logger1.Log("Logging the first message.");**

**logger2.Log("Logging the second message.");**

**if (logger1 == logger2)**

**{**

**Console.WriteLine("Success: Both references point to the same Logger instance.");**

**}**

**else**

**{**

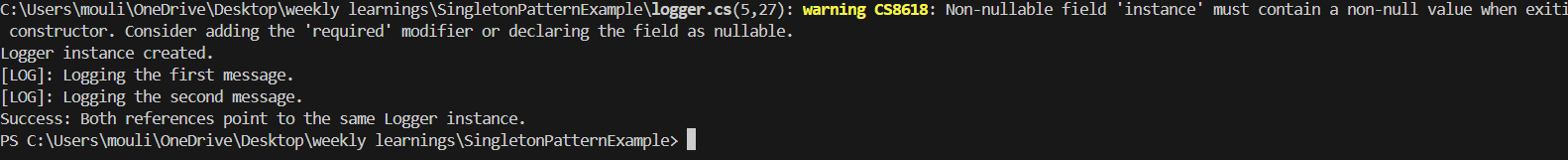
**Console.WriteLine("Failure: Logger instances are different.");**

**}**

**}**

**}**

**The output of the code is:**



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

**Step:- 1**

**Binary Search splits the list in half each time (like guessing a number), so it’s much faster—but only works on sorted data.**

Big O notation describes the performance (time or space) of an algorithm.

It helps to compare the efficiency of the algorithms.

O(1) is a Constant time

O(n) is a Linear time

O(log n) is a Logarithmic time

O(n log n), O(n²) is a More complex patterns

| **Search Type** | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- |
| **Linear Search** | Super fast if the match is first | Checks about half the list | Checks the whole list or not found |
| **Binary Search** | Super fast if it's in the middle | Very quick (logarithmic) | Still very quick (logarithmic) |

**Step:- 2**

Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.

I created a total of 3 files as the class names separately.

Class to represent the products in the ecommerce site

// Product.cs

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 to write both Linear and Binary Search logic.

### **Step:-3 Implementing the Search Methods**

I created a helper class to write both Linear and Binary Search logic.

using System;

public class SearchService{

public static Product LinearSearch(Product[] products, string keyword){

foreach (var item in products){

if (item.ProductName.Equals(keyword, StringComparison.OrdinalIgnoreCase)){

return item;

}

}

return null;

}

public static Product BinarySearch(Product[] sortedProducts, string keyword){

int left = 0;

int right = sortedProducts.Length - 1;

while (left <= right)

{

int middle = (left + right) / 2;

int compareResult = string.Compare(sortedProducts[middle].ProductName, keyword, StringComparison.OrdinalIgnoreCase);

if (compareResult == 0)

return sortedProducts[middle];

else if (compareResult < 0)

left = middle + 1;

else

right = middle - 1;

}

return null;

}

}

### **Step 4: main program**

using System;

using System.Linq;

class Program

{

static void Main()

{

Product[] items = new Product[]

{

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

new Product(2, "Keyboard", "Electronics"),

new Product(3, "Shirt", "Clothing"),

new Product(4, "Shoes", "Footwear"),

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

};

string searchTerm = "Shirt";

// Run linear search

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

var foundLinear = SearchService.LinearSearch(items, searchTerm);

if (foundLinear != null)

Console.WriteLine($"{foundLinear.ProductName} found in {foundLinear.Category}.");

else

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

// Run binary search (after sorting)

var sortedItems = items.OrderBy(p => p.ProductName).ToArray();

Console.WriteLine("\n Binary Search Result:");

var foundBinary = SearchService.BinarySearch(sortedItems, searchTerm);

if (foundBinary != null)

Console.WriteLine($"{foundBinary.ProductName} found in {foundBinary.Category}.");

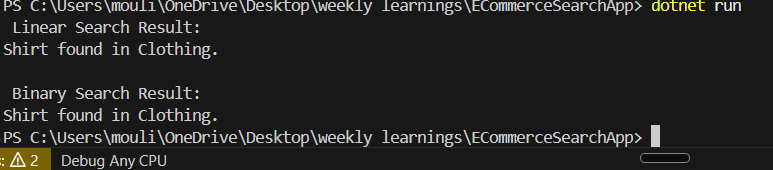
else

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

}

}

**Output:-**



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

**Scenario:**

You are developing a document management system that needs to create different types of documents (e.g., Word, PDF, Excel). Use the Factory Method Pattern to achieve this.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **FactoryMethodPatternExample**.
2. **Define Document Classes:**
   * Create interfaces or abstract classes for different document types such as **WordDocument**, **PdfDocument**, and **ExcelDocument**.

**Program.cs**

using System;

class Program

{

static void Main(string[] args)

{

DocumentFactory wordFactory = new WordDocumentFactory();

Document wordDoc = wordFactory.CreateDocument();

wordDoc.Open();

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdfDoc = pdfFactory.CreateDocument();

pdfDoc.Open();

DocumentFactory excelFactory = new ExcelDocumentFactory();

Document excelDoc = excelFactory.CreateDocument();

excelDoc.Open();

}

}

**Document.cs**

public interface Document

{

void Open();

}

[**Documentfactory.cs**](http://documentfactory.cs)

public abstract class DocumentFactory

{

public abstract Document CreateDocument();

}

[Exceldocument.cs](http://exceldocument.cs)

using System;

public class ExcelDocument : Document

{

public void Open()

{

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

}

}

[exceldocumentfactory.cs](http://exceldocumentfactory.cs)

public class ExcelDocumentFactory : DocumentFactory

{

public override Document CreateDocument()

{

return new ExcelDocument();

}

}

[pdfdocument.cs](http://pdfdocument.cs)

using System;

public class PdfDocument : Document

{

public void Open()

{

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

}

}

[pdfdocumentfactory.cs](http://pdfdocumentfactory.cs)

public class PdfDocumentFactory : DocumentFactory

{

public override Document CreateDocument()

{

return new PdfDocument();

}

}

[worddocument.cs](http://worddocument.cs)

using System;

public class WordDocument : Document

{

public void Open()

{

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

}

}

[worddocumentfactory.cs](http://worddocumentfactory.cs)

public class WordDocumentFactory : DocumentFactory

{

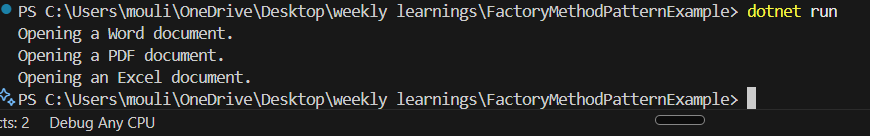
public override Document CreateDocument()

{

return new WordDocument();

}

}

**Output:-**

**Exercise 7: Financial Forecasting**

**Scenario:**

You are developing a financial forecasting tool that predicts future values based on past data.

**Steps 1:- Understand Recursive Algorithms:**

Explain the concept of recursion and how it can simplify certain problems.

**Recursion** is when a method calls itself to break a problem into smaller pieces. It’s useful when problems follow a repeatable pattern, ex:- calculating compound values, factorials, or Fibonacci numbers.

Recursive methods typically:

1. Have a **base case** (when to stop).
2. Call themselves with **reduced input**.

### **Step 2:- Setup a Recursive Method**

* Starting value
* Annual growth rate
* Number of years to project

**Step 3: Implement the Recursive Forecasting Algorithm**

**// ForecastService.cs**

**using System;**

**public class ForecastService**

**{**

**public static double ForecastFutureValue(double initialAmount, double growthRate, int years)**

**{**

**if (years == 0)**

**return initialAmount;**

**double updatedAmount = initialAmount \* (1 + growthRate);**

**return ForecastFutureValue(updatedAmount, growthRate, years - 1);**

**}**

**}**

**Time Complexity :**

**Time Complexity**: O(n) (where n = number of years)

**Step 4: Test the Forecast**

using System;

class Program

{

static void Main()

{

double startingAmount = 10000; // Initial invest

double annualGrowthRate = 0.08; // 8% growth rate

int forecastYears = 5; // Forecast for 5y

double result = ForecastService.ForecastFutureValue(startingAmount, annualGrowthRate, forecastYears);

Console.WriteLine($"Projected value after {forecastYears} years: ₹{result:F2}");

}

}

**Output:**

