**Deep Skilling Week 1 Assignment**

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

**Code:**

**[Logger.cs](http://logger.cs)**

using System;

namespace SingletonPatternExample

{

internal class Logger

{

private static Logger \_instance;

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

private Logger()

{

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

}

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}");

}

}

}

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

using System;

namespace SingletonPatternExample

{

class Program

{

static void Main(string[] args)

{

Logger logger1 = Logger.GetInstance();

Logger logger2 = Logger.GetInstance();

logger1.Log("This is the first log message.");

logger2.Log("This is the second log message.");

if (logger1 == logger2)

{

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

}

else

{

Console.WriteLine("Singleton failed: Different instances found.");

}

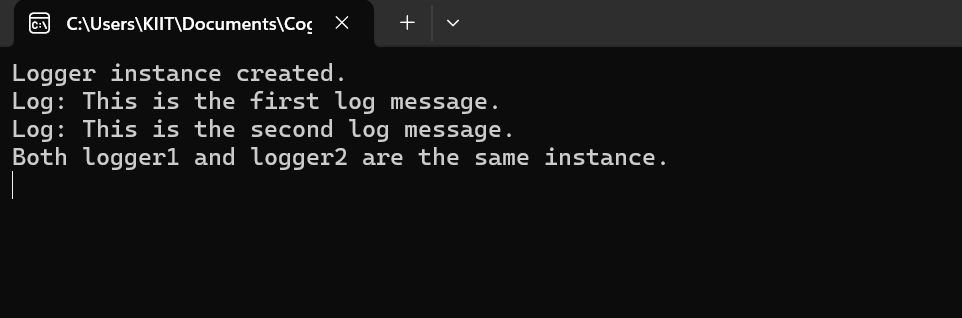
Console.ReadLine(); // Keeps the console open

}

}

}

**Output:**



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

**Code:**

**[IDocument.cs](http://idocument.cs)**

namespace FactoryMethodPatternExample

{

public interface IDocument

{

void Open();

}

}

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

namespace FactoryMethodPatternExample

{

public class WordDocument : IDocument

{

public void Open()

{

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

}

}

}

**[ExcelDocument.cs](http://exceldocument.cs)**

namespace FactoryMethodPatternExample

{

public class ExcelDocument : IDocument

{

public void Open()

{

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

}

}

}

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

namespace FactoryMethodPatternExample

{

public abstract class DocumentFactory

{

public abstract IDocument CreateDocument();

}

}

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

namespace FactoryMethodPatternExample

{

public class WordDocumentFactory : DocumentFactory

{

public override IDocument CreateDocument()

{

return new WordDocument();

}

}

}

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

namespace FactoryMethodPatternExample

{

public class PdfDocumentFactory : DocumentFactory

{

public override IDocument CreateDocument()

{

return new PdfDocument();

}

}

}

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

namespace FactoryMethodPatternExample

{

public class ExcelDocumentFactory : DocumentFactory

{

public override IDocument CreateDocument()

{

return new ExcelDocument();

}

}

}

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

using System;

namespace FactoryMethodPatternExample

{

class Program

{

static void Main(string[] args)

{

DocumentFactory wordFactory = new WordDocumentFactory();

IDocument wordDoc = wordFactory.CreateDocument();

wordDoc.Open();

DocumentFactory pdfFactory = new PdfDocumentFactory();

IDocument pdfDoc = pdfFactory.CreateDocument();

pdfDoc.Open();

DocumentFactory excelFactory = new ExcelDocumentFactory();

IDocument excelDoc = excelFactory.CreateDocument();

excelDoc.Open();

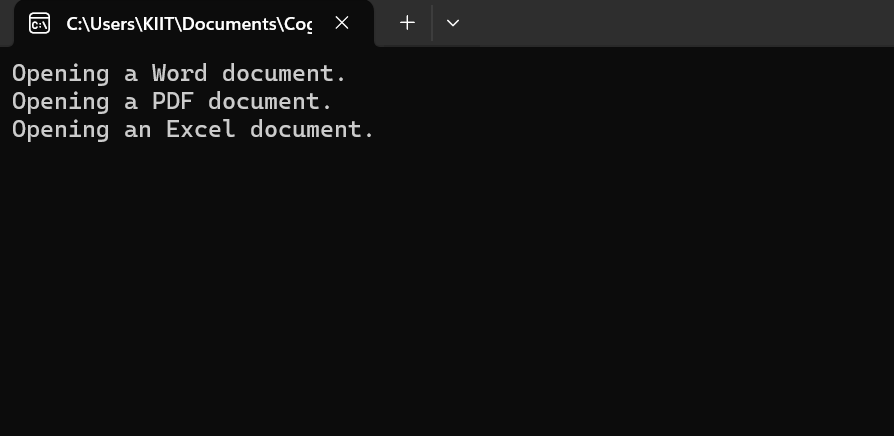
Console.ReadLine();

}

}

}

**Output:**



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

## **Step 1: Understand Asymptotic Notation**

### **What is Big O Notation?** Big O notation describes the upper bound of an algorithm’s runtime or space requirement in terms of input size n. **It helps you:** i).Compare different algorithms independently of hardware. ii).Predict how performance scales with larger datasets.

**Best, Average, and Worst-Case Scenarios:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Search Type** | **Best Case** | **Average Case** | **Worst Case** |
| **Linear Search** | O(1) | O(n/2) | O(n) |
| **Binary Search** | O(1) | O(log n) | O(log n) |

**Code:**

**[Product.cs](http://product.cs)**

namespace EcommerceSearch

{

public class 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;

}

public override string ToString()

{

return $"{ProductId} - {ProductName} ({Category})";

}

}

}

**[SearchUtility.cs](http://searchutility.cs)**

using System;

namespace EcommerceSearch

{

public static class SearchUtility

{

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

{

foreach (var product in products)

{

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

{

return product;

}

}

return null;

}

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

{

int left = 0, right = products.Length - 1;

while (left <= right)

{

int mid = (left + right) / 2;

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

if (comparison == 0)

return products[mid];

else if (comparison < 0)

left = mid + 1;

else

right = mid - 1;

}

return null;

}

}

}

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

using System;

using System.Linq;

namespace EcommerceSearch

{

class Program

{

static void Main(string[] args)

{

Product[] products = {

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

new Product(2, "Shoes", "Fashion"),

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

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

new Product(5, "T-shirt", "Fashion")

};

Console.WriteLine("Linear Search for 'Watch':");

var linearResult = SearchUtility.LinearSearch(products, "Watch");

Console.WriteLine(linearResult != null ? linearResult.ToString() : "Not Found");

Console.WriteLine("Binary Search for 'Watch':");

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

var binaryResult = SearchUtility.BinarySearch(sortedProducts, "Watch");

Console.WriteLine(binaryResult != null ? binaryResult.ToString() : "Not Found");

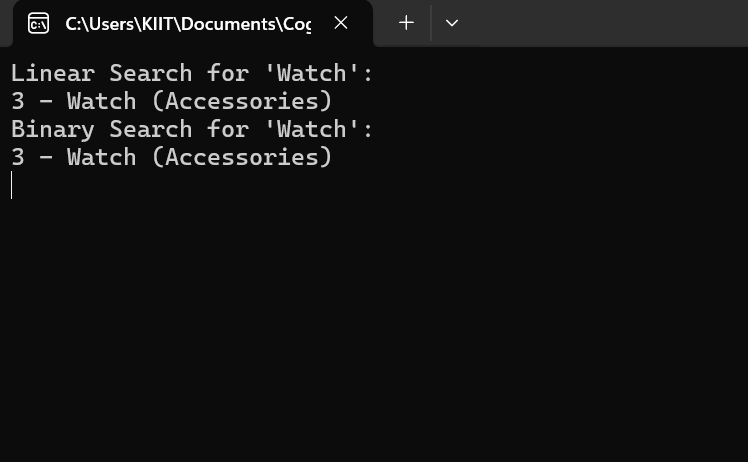
Console.ReadLine();

}

}

}

Output:



**Comparison of Time Complexity:**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Linear Search** | **Binary Search** |
| Time Complexity | O(n) | O(log n) |
| Best Case | O(1) (if element is first) | O(1) (if element is at middle) |
| Average Case | O(n/2) | O(log n) |
| Worst Case | O(n) (last element or not present) | O(log n) (element not found) |
|  |  |  |
|  |  |  |

## **Suitability for an E-commerce Platform**

### Binary Search is More Suitable When:

* Your product list is pre-sorted by name, ID, or other criteria.
* You need high-speed search over large product datasets (e.g., thousands of items).
* You can afford the initial sorting step or already store products in a sorted database/index.

### Linear Search is Acceptable When:

* The list is small (e.g., <100 items).
* Data is unsorted and can change rapidly (e.g., adding/removing items frequently).
* Simplicity matters more than speed.

### **E-commerce Platform Recommendation:**

Use Binary Search (or more advanced alternatives like hash maps, search trees, or full-text search engines) because:

1. Performance matters — customers expect instant results.
2. Product lists are often large and need fast access.
3. Sorted data is common (e.g., by name, category, rating, etc.).

However, in real-world production systems, you'd likely use:

* Databases with indexing (WHERE, LIKE, ORDER BY)
* Search engines like Elasticsearch for keyword search, partial matches, and ranking

**Exercise 7: Financial Forecasting**

**1. Understand Recursive Algorithms:**

#### Recursion is a programming technique where a function calls itself to solve smaller instances of the same problem. Every recursive function has:

#### A base case (the stopping condition)

#### A recursive case (the part where the function calls itself)

### **How Does Recursion Simplify Problems?**

#### **1. Breaks Big Problems into Smaller Ones**

Instead of solving a complex task all at once, recursion reduces it to solving a smaller subproblem, and combines the results.

#### **2. More Natural for Certain Problems**

Some problems are inherently recursive in structure, such as:

* Tree traversal
* Tower of Hanoi
* Fibonacci sequence
* Backtracking (e.g., solving mazes or Sudoku)
* Financial forecasting (e.g., future value over years)

**Code:**

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

using System;

namespace FinancialForecasting

{

class Program

{

static void Main(string[] args)

{

double initialInvestment = 10000;

double annualGrowthRate = 0.08;

int years = 10;

double forecast = FutureValueRecursive(initialInvestment, annualGrowthRate, years);

Console.WriteLine($"Forecasted value after {years} years: Rs {forecast:F2}");

Console.ReadLine();

}

public static double FutureValueRecursive(double initialValue, double growthRate, int years)

{

if (years == 0)

return initialValue;

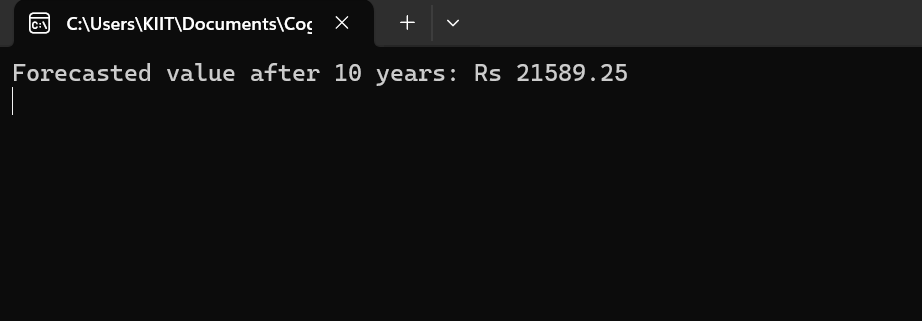
return FutureValueRecursive(initialValue, growthRate, years - 1) \* (1 + growthRate);

}

}

}

Output:



**Analysis:**

**Time Complexity: O(n)**

* The function makes one recursive call per year.
* So for n = years, it makes n calls.
* Each call does constant time work (just a multiplication), so:

**Space Complexity: O(n)**

* Each recursive call is placed on the call stack.
* So for n years, the stack depth is n.
* If n is large (e.g., 10,000), you may run into a stack overflow.

How to Optimize the Recursive Solution?

1. Replace Recursion with Iteration (Most Efficient)

* Time Complexity: O(n)
* Space Complexity: O(1)
* No risk of stack overflow
* More readable and efficient

2. Use Recursion with Memoization

* Prevents repeating calculations
* Keeps recursive structure if needed
* Good for recursive models like Fibonacci, DP problems, etc.