MANDATORY HANDS-ON QUESTIONS

**Design Pattern and Principles**

Exercise 1: Implementing the Singleton Pattern

Code: Logger.cs file

using System;

namespace SingletonPatternExample;

public class Logger

{

private static Logger \_instance;

// Optional: Make it thread-safe

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

// Private constructor to prevent external instantiation

private Logger()

{

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

}

// Public method to get the singleton instance

public static Logger GetInstance()

{

if (\_instance == null)

{

lock (\_lock) // Ensure thread-safety

{

if (\_instance == null)

{

\_instance = new Logger();

}

}

}

return \_instance;

}

// Example log method

public void Log(string message)

{

Console.WriteLine($"Log: {message}");

}

}

Program.cs file

using System;

namespace SingletonPatternExample

{

class Program

{

static void Main(string[] args)

{

// Get singleton instances

Logger logger1 = Logger.GetInstance();

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

Logger logger2 = Logger.GetInstance();

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

// Verify both references point to the same instance

if (logger1 == logger2)

{

Console.WriteLine("Both logger instances are the same.");

}

else

{

Console.WriteLine("Logger instances are different.");

}

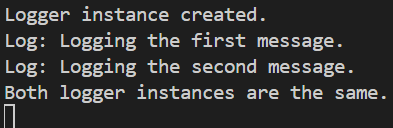
Console.ReadLine(); // Keep console open (optional)

}

}

}

Output:



Exercise 2: Implementing the Factory Method Pattern

Code: **Interface IDocument.cs**

namespace FactoryMethodPatternExample

{

public interface IDocument

{

void Open();

}

}

**Concrete document classes**

using System;

namespace FactoryMethodPatternExample

{

public class WordDocument : IDocument

{

public void Open()

{

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

}

}

}

using System;

namespace FactoryMethodPatternExample

{

public class PdfDocument : IDocument

{

public void Open()

{

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

}

}

}

using System;

namespace FactoryMethodPatternExample

{

public class ExcelDocument : IDocument

{

public void Open()

{

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

}

}

}

**Factory classes**

namespace FactoryMethodPatternExample

{

public abstract class DocumentFactory

{

public abstract IDocument CreateDocument();

}

}

namespace FactoryMethodPatternExample

{

public class WordDocumentFactory : DocumentFactory

{

public override IDocument CreateDocument()

{

return new WordDocument();

}

}

}

namespace FactoryMethodPatternExample

{

public class PdfDocumentFactory : DocumentFactory

{

public override IDocument CreateDocument()

{

return new PdfDocument();

}

}

}

namespace FactoryMethodPatternExample

{

public class ExcelDocumentFactory : DocumentFactory

{

public override IDocument CreateDocument()

{

return new ExcelDocument();

}

}

}

**Main**

using System;

namespace FactoryMethodPatternExample

{

class Program

{

static void Main(string[] args)

{

DocumentFactory wordFactory = new WordDocumentFactory();

IDocument word = wordFactory.CreateDocument();

word.Open();

DocumentFactory pdfFactory = new PdfDocumentFactory();

IDocument pdf = pdfFactory.CreateDocument();

pdf.Open();

DocumentFactory excelFactory = new ExcelDocumentFactory();

IDocument excel = excelFactory.CreateDocument();

excel.Open();

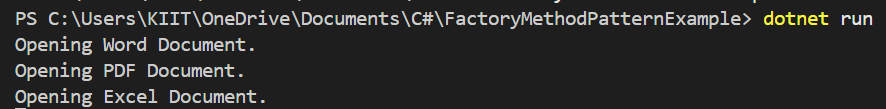
Console.ReadLine(); // Optional, to keep console open

}

}

}

**Output**



**Data Structures and Algorithms**

Exercise 2: E-commerce Platform Search Function

Code: Product.cs(Product class)

namespace EcommerceSearchApp

{

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;

}

public override string ToString()

{

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

}

}

}

SearchAlgorithms.cs (For implementing search algorithms)

using System;

namespace EcommerceSearchApp

{

public static class SearchAlgorithms

{

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

{

foreach (var product in products)

{

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

return product;

}

return null;

}

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

{

int left = 0;

int right = products.Length - 1;

while (left <= right)

{

int mid = (left + right) / 2;

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

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

else if (cmp < 0) left = mid + 1;

else right = mid - 1;

}

return null;

}

}

}

Program.cs

using System;

using System.Linq;

namespace EcommerceSearchApp

{

class Program

{

static void Main(string[] args)

{

Product[] products = new Product[]

{

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

new Product(2, "Sneakers", "Footwear"),

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

new Product(4, "Keyboard", "Accessories"),

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

};

Console.WriteLine("=== Linear Search ===");

var result1 = SearchAlgorithms.LinearSearch(products, "Shirt");

Console.WriteLine(result1 != null ? $"Found: {result1}" : "Product not found");

Console.WriteLine("\n=== Binary Search ===");

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

var result2 = SearchAlgorithms.BinarySearch(sortedProducts, "Shirt");

Console.WriteLine(result2 != null ? $"Found: {result2}" : "Product not found");

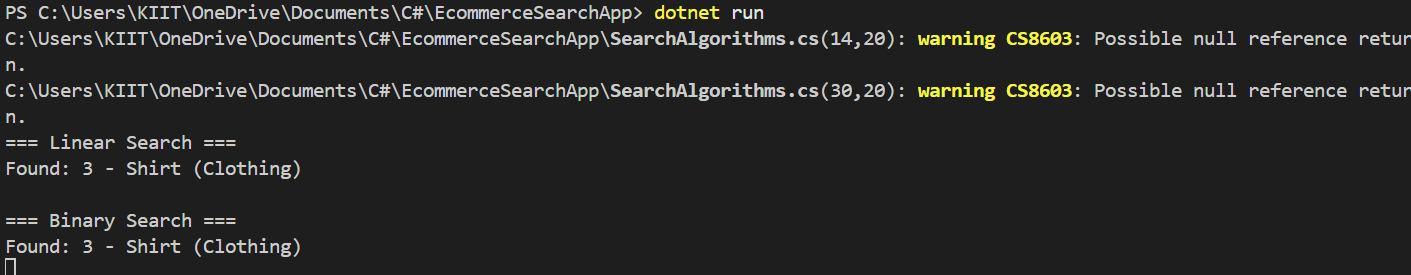
Console.ReadLine();

}

}

}

Output:



ANALYSIS OF THE TIME COMPLEXITY

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Time Complexity** | **Best Case** | **Worst Case** |
| Linear Search | O(n) | O(1) | O(n) |
| Binary Search | O(log n) | O(1) | O(log n) |

For E-Commerce platform binary Search is better overall because:

1. **Search speed matters a lot in e-commerce**
   1. Customers expect fast search responses.
   2. Binary Search gives O(log n) performance - scalable for 1000s of products.
2. **You can store products sorted by name, ID, or category**
   1. Sorting once enables fast repeated searches.
   2. Sorting can be done initially or periodically when products are added.
3. **Real platforms often use indexed databases**
   1. They operate similarly to binary search under the hood (e.g., B-trees).

Data Structures and Algorithm

Exercise 7: Financial Forecasting

Code: Forecast.cs (Recursive forecasting method)

using System;

namespace FinancialForecastingApp

{

public static class Forecast

{

// Recursive method to calculate future value

public static double PredictFutureValue(double currentValue, double growthRate, int years)

{

if (years == 0)

return currentValue;

return PredictFutureValue(currentValue, growthRate, years - 1) \* (1 + growthRate);

}

}

}

Program.cs

using System;

namespace FinancialForecastingApp

{

class Program

{

static void Main(string[] args)

{

double initialValue = 1000.0;

double annualGrowthRate = 0.10; // 10%

int years = 5;

double futureValue = Forecast.PredictFutureValue(initialValue, annualGrowthRate, years);

Console.WriteLine($"Initial Value: {initialValue}");

Console.WriteLine($"Growth Rate: {annualGrowthRate \* 100}%");

Console.WriteLine($"Years: {years}");

Console.WriteLine($"Predicted Future Value: {futureValue:F2}");

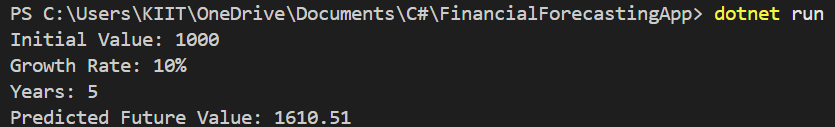
Console.ReadLine();

}

}

}

Output:



ANALYSIS: **Time Complexity of Recursive Method:**

* **Time Complexity:** O(n) - because it makes 1 call per year.
* **Space Complexity:** O(n) - due to recursive call stack.

Optimization process to avoid excessive computation

1. To use the iterative version

public static double PredictFutureValueIterative(double currentValue, double growthRate, int years)

{

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

{

currentValue \*= (1 + growthRate);

}

return currentValue;

}

1. Use memoization

Memoization is storing previously computed values to avoid repeated calculations. For this simple case, it's not necessary but essential in recursive algorithms like Fibonacci or tree traversal.