Design Patterns and Principles

Exercise 1: Implementing the Singleton Pattern

Code :

```
using System;

namespace SingletonPatternExample{
   public sealed class Logger{
      private static readonly Logger _instance = new Logger();
      private Logger() {
         Console.WriteLine("Logger instance created.");
      }
      public static Logger Instance{
         get{
            return _instance;
         }
      }
      public void Log(string message){
         Console.WriteLine($"[LOG] {message}");
      }
   }
    class Program{
      static void Main(string[] args){
         Logger logger1 = Logger.Instance;
         Logger logger2 = Logger.Instance;
         logger1.Log("This is the first log message.");
         logger2.Log("This is the second log message.");
         Console.WriteLine($"Are both loggers the same instance? {ReferenceEquals(logger1,
logger2)}");
      }
   }
}
```

Output :

Logger instance created.

[LOG] This is the first log message.

[LOG] This is the second log message.

Are both loggers the same instance? True

Exercise 2: Implementing the Factory Method Pattern

Code :

```
using System;
namespace FactoryMethodPatternExample{
  public interface IDocument {
    void Open();
  }
  public class WordDocument : IDocument{
    public void Open(){
      Console.WriteLine("Opening a Word document.");
    }
  }
  public class PdfDocument : IDocument{
    public void Open(){
      Console.WriteLine("Opening a PDF document.");
    }
  }
  public class ExcelDocument : IDocument{
    public void Open(){
      Console.WriteLine("Opening an Excel document.");
    }
  }
```

```
public abstract class DocumentFactory{
   public abstract IDocument CreateDocument();
}
public class WordDocumentFactory : DocumentFactory{
   public override IDocument CreateDocument(){
      return new WordDocument();
   }
}
public class PdfDocumentFactory : DocumentFactory{
   public override IDocument CreateDocument(){
      return new PdfDocument();
   }
}
public class ExcelDocumentFactory : DocumentFactory{
   public override IDocument CreateDocument(){
      return new ExcelDocument();
   }
}
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(); }
```

}

}

Output :

Opening a Word document.

Opening a PDF document.

Opening an Excel document.

Algorithms_Data Structures

Exercise 2: E-commerce Platform Search Function

Code :

1. **Understand Asymptotic Notation:**
    - Explain Big O notation and how it helps in analyzing algorithms.
    - Describe the best, average, and worst-case scenarios for search operations.

Sol : **Big O Notation** describes the **upper bound of time or space complexity** of an algorithm in terms of input size n. It helps evaluate **scalability and performance**.

Notation	Meaning	Example Algorithm
O(1)	Constant time	Accessing array element
O(n)	Linear time	Linear search
O(log n)	Logarithmic time	Binary search
O(n²)	Quadratic time	Nested loops

Best, Average, and Worst Cases for Search

Algorithm	Best Case	Average Case	Worst Case
Linear Search	O(1)	O(n/2) ≈ O(n)	O(n)
Binary Search	O(1)	O(log n)	O(log n)

Code :

```
using System;
using System.Linq;

namespace ECommerceSearchExample{
  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 $"ID: {ProductId}, Name: {ProductName}, Category: {Category}";
    }
  }
  class Program{
    static void Main(string[] args){
      Product[] products = new Product[]{
        new Product(3, "Laptop", "Electronics"),
        new Product(1, "Shoes", "Apparel"),
        new Product(2, "Book", "Books"),
        new Product(4, "Headphones", "Electronics")
      };
      Console.WriteLine("=== Linear Search (unsorted) ===");
      var linearResult = LinearSearch(products, 2);
      Console.WriteLine(linearResult != null ? linearResult.ToString() : "Product not found.");
      Console.WriteLine("\n=== Binary Search (sorted) ===");
      var sortedProducts = products.OrderBy(p => p.ProductId).ToArray();
      var binaryResult = BinarySearch(sortedProducts, 2);
```

```
            Console.WriteLine(binaryResult != null ? binaryResult.ToString() : "Product not found.");
        }
        public static Product LinearSearch(Product[] products, int productId){
            foreach (var product in products){
                if (product.ProductId == productId)
                    return product;
            }
            return null;
        }
        public static Product BinarySearch(Product[] products, int productId){
            int left = 0;
            int right = products.Length - 1;
            while (left <= right){
                int mid = (left + right) / 2;
                if (products[mid].ProductId == productId)
                    return products[mid];
                else if (products[mid].ProductId < productId)
                    left = mid + 1;
                else
                    right = mid - 1;
            }
            return null;
        }
    }
}
```

Output :

=== Linear Search (unsorted) ===

ID: 2, Name: Book, Category: Books

=== Binary Search (sorted) ===

ID: 2, Name: Book, Category: Books

Time Complexity Comparison

Algorithm	Sorted Required	Time Complexity	Suitable For Large Data?
Linear Search	No	O(n)	No
Binary Search	Yes	O(log n)	Yes

Recommendation :

- Use **Binary Search** for performance-critical search operations **after sorting** or indexing products (especially by `ProductId` or `ProductName`).
- For **small or unsorted datasets**, Linear Search is simpler and sufficient.
- For **real-time applications**, consider additional indexing or database-backed search engines (e.g., ElasticSearch).

Exercise 7: Financial Forecasting

1. **Understand Recursive Algorithms:**
    - Explain the concept of recursion and how it can simplify certain problems.

Sol :

Recursion is a technique where a function calls itself to solve smaller sub-problems of the original problem.

> Example use cases: factorial, Fibonacci, file directory traversal, etc.

It can **simplify complex problems** like repeated calculations, series, or tree traversal, by avoiding the need for explicit loops.

Code :

```
using System;

namespace FinancialForecasting{

  class Program{

    static void Main(string[] args){
```

```
            double initialValue = 1000; // Starting value
            double annualGrowthRate = 0.05; // 5% growth
            int years = 10;

            Console.WriteLine("=== Recursive Forecast ===");
            double futureValue = ForecastFutureValue(initialValue, annualGrowthRate, years);
            Console.WriteLine($"Future value after {years} years: {futureValue:C2}");

            Console.WriteLine("\n=== Optimized (Memoized) Forecast ===");
            double memoizedValue = ForecastFutureValueMemo(initialValue, annualGrowthRate,
years, new double?[years + 1]);
            Console.WriteLine($"Future value after {years} years: {memoizedValue:C2}");
        }
        public static double ForecastFutureValue(double currentValue, double rate, int years){
            if (years == 0)
                return currentValue;
            return ForecastFutureValue(currentValue, rate, years - 1) * (1 + rate);
        }
        public static double ForecastFutureValueMemo(double currentValue, double rate, int
years, double?[] memo){
            if (years == 0)
                return currentValue;
            if (memo[years] != null)
                return memo[years].Value;

            memo[years] = ForecastFutureValueMemo(currentValue, rate, years - 1, memo) * (1 +
rate);
            return memo[years].Value;
        }
    }
}
```

Output :

=== Recursive Forecast ===

Future value after 10 years: $1,628.89

=== Optimized (Memoized) Forecast ===

Future value after 10 years: $1,628.89

Time Complexity :

Basic Recursive Version:

- Time Complexity: `O(n)` (calls function once per year)
- Space Complexity: `O(n)` due to call stack

Memoized Version:

- Time Complexity: `O(n)` (no redundant calculations)
- Space Complexity: `O(n)` (uses memo array)

Optimization Strategy :

- Use **memoization** to cache results and avoid recalculating values.
- For larger `n`, consider **iterative** or **dynamic programming** for better performance and stack safety.