Exercise : Implementing the Singleton Pattern

CODE :

```csharp
public class Logger
{
    private static Logger? _instance;  // static = shared across all uses

    private Logger()                   // private constructor = no one else can create
    {
        Console.WriteLine("Logger created.");
    }

    public static Logger GetInstance()
    {
        if (_instance == null)
        {
            _instance = new Logger();  // create only if not already created
        }
        return _instance;
    }

    public void Log(string message)
    {
        Console.WriteLine("Log: " + message);
    }
}
```

```csharp
class Program
{
    static void Main()
    {
        var logger1 = Logger.GetInstance();
        logger1.Log("Hello from logger 1");

        var logger2 = Logger.GetInstance();
        logger2.Log("Hello from logger 2");

        Console.WriteLine(Object.ReferenceEquals(logger1, logger2)
            ? "Same instance"
            : "Different instances");
    }
}
```

OUTPUT :

```
PS C:\Users\KIIT\CognizantAssignments\week1\SourceCode\SingletonPatternExample> dotnet run
Logger created.
Log: Hello from logger 1
Log: Hello from logger 2
Same instance
```

Exercise : Implementing the Factory Method Pattern

CODE :

```
public interface IDocument
{
    void Open();
}
```

```
public class WordDocument : IDocument
{
    public void Open()
    {
        Console.WriteLine("Opening Word Document.");
    }
}
```

```
public class PdfDocument : IDocument
{
    public void Open()
    {
        Console.WriteLine("Opening PDF Document.");
    }
}
```

```
public abstract class DocumentFactory
{
    public abstract IDocument CreateDocument();
}
```

```
public class WordFactory : DocumentFactory
{
    0 references
    public override IDocument CreateDocument()
    {
        return new WordDocument();
    }
}

```

```
public class PdfFactory : DocumentFactory
{
    0 references
    public override IDocument CreateDocument()
    {
        return new PdfDocument();
    }
}

```

```
public class ExcelFactory : DocumentFactory
{
    0 references
    public override IDocument CreateDocument()
    {
        return new ExcelDocument();
    }
}

```

```
class Program
{
    0 references
    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();
    }
}

```

OUTPUT :

```
PS C:\Users\KIIT\CognizantAssignments\week1\SourceCode\FactoryMethodPatternExample> dotnet run
Opening Word Document.
Opening PDF Document.
Opening Excel Document.
```

Exercise : E-commerce Platform Search Function

Understanding Asymptotic Notation (Big O) :

Big O notation describes how an algorithm’s runtime or space requirement grows as input size increases.

For Search Operations:

Case	Linear Search	Binary Search
Best	O(1) (first item)	O(1) (middle item)
Average	O(n/2) = O(n)	O(log n)
Worst	O(n)	O(log n)

Binary search is much faster than linear search — but it only works on sorted data.

Implementation :

CODE :

```
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;
    }
}
```

```csharp
using System.Diagnostics;
class Program
{
    static void Main()
    {
        Product[] products = new Product[]
        {
            new Product(101, "Laptop", "Electronics"),
            new Product(102, "Shirt", "Clothing"),
            new Product(103, "Book", "Stationery"),
            new Product(104, "Mouse", "Electronics"),
            new Product(105, "Notebook", "Stationery")
        };

        // 🔍 Linear Search
        Console.WriteLine("Linear Search for 'Book':");
        var stopwatch1 = Stopwatch.StartNew();
        var result1 = LinearSearch(products, "Book");
        stopwatch1.Stop();
        Console.WriteLine(result1?.ProductName ?? "Not Found");
        Console.WriteLine($"Linear search time: {stopwatch1.Elapsed.TotalMilliseconds} ms\n");

        // 🔍 Binary Search (requires sorted array)
        Array.Sort(products, (a, b) => a.ProductName.CompareTo(b.ProductName));

        Console.WriteLine("Binary Search for 'Book':");
        var stopwatch2 = Stopwatch.StartNew();
        var result2 = BinarySearch(products, "Book");
        stopwatch2.Stop();
        Console.WriteLine(result2?.ProductName ?? "Not Found");
        Console.WriteLine($"Binary search time: {stopwatch2.Elapsed.TotalMilliseconds} ms\n");
    }

```

```
    static Product? LinearSearch(Product[] products, string name)
    {
        foreach (var p in products)
        {
            if (p.ProductName == name)
                return p;
        }
        return null;
    }

    1 reference
    static Product? BinarySearch(Product[] products, string name)
    {
        int left = 0, right = products.Length - 1;

        while (left <= right)
        {
            int mid = (left + right) / 2;
            int cmp = string.Compare(products[mid].ProductName, name);
            if (cmp == 0) return products[mid];
            else if (cmp < 0) left = mid + 1;
            else right = mid - 1;
        }

        return null;
    }
}
```

OUTPUT :

```
PS C:\Users\KIIT\CognizantAssignments\week1\SourceCode\EcommerceSearch> dotnet run
Linear Search for 'Book':
Book
Linear search time: 0.1735 ms

Binary Search for 'Book':
Book
Binary search time: 0.1412 ms
```

Analysis :

Time Complexities:

Linear Search: O(n)

Binary Search: O(log n) — but only works if data is sorted.

Best Choice:

Use binary search for speed. We can store product data sorted by name or ID

Exercise : Financial Forecasting

Understanding Recursion :

Recursion is when a method calls itself to break a problem into smaller parts.

Example Use Case : predicting next value based on previous ones.

Implementation :

CODE :

```
class Program
{
    0 references
    static void Main()
    {
        double initialValue = 1000; // starting money
        double growthRate = 0.10;   // 10% growth rate
        int years = 5;
        Console.WriteLine($"Value after {years} years: {FutureValue(initialValue, growthRate, years):C2}");
    }
    // Recursive: FV = PV * (1 + r)^n
    2 references
    static double FutureValue(double amount, double rate, int years)
    {
        if (years == 0) return amount;
        return FutureValue(amount, rate, years - 1) * (1 + rate);
    }
}
```

OUTPUT :

```
PS C:\Users\KIIT\CognizantAssignments\week1\SourceCode\FinancialForecast> dotnet run
Value after 5 years: ₹ 1,610.51
```

Analysis :

Time Complexity:

Method	Time Complexity
Basic Recursion	O(n)
Memoized	O(n) (but faster in practice due to caching)

In this example, recursion is OK, but for large n, we should either use iteration if possible (loop) or use memoization to cache previous results.