Week 1 Assignment

a) Design Pattern and Principal

1. Implementing the Singleton Pattern

```
using System;

public sealed class Logger
{
 private static readonly Lazy<Logger> _instance =
  new Lazy<Logger>(() => new Logger());

 private static int _instanceCount = 0;

 private Logger()
 {
  _instanceCount++;
  Console.WriteLine("Logger instance created");
 }

 public static Logger Instance => _instance.Value;

 public void Log(string message)
 {
  Console.WriteLine($"[LOG] {DateTime.Now}: {message}");
 }

 public static int GetInstanceCount()
 {
     return _instanceCount;
 }
}
class Program
{
```

```
 static void Main(string[] args)
 {
  Logger logger1 = Logger.Instance;
  Logger logger2 = Logger.Instance;

  logger1.Log("First log message");
  logger2.Log("Second log message");

  Console.WriteLine($"Same instance?
{ReferenceEquals(logger1, logger2)}");
  Console.WriteLine($"Instance count:
{Logger.GetInstanceCount()}");
 }
}
```

output

```
PS C:\Users\KIIT\Desktop\Cognizant Exercise Codes\dpp exercise 1> dotnet run SingletonPattern.cs
Logger instance created
[LOG] 22-06-2025 22:42:48: First log message
[LOG] 22-06-2025 22:42:48: Second log message
Same instance? True
Instance count: 1
PS C:\Users\KIIT\Desktop\Cognizant Exercise Codes\dpp exercise 1>
```

2. Implementing the Factory Method Pattern

```
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 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();
 }
}
}
```

output

```
PS C:\Users\KIIT\Desktop\Cognizant Exercise Codes\dpp exercise 2> dotnet run FactoryMethodPatter.cs
Opening a Word document.
Opening a PDF document.
Opening an Excel document.
PS C:\Users\KIIT\Desktop\Cognizant Exercise Codes\dpp exercise 2>
```

b) Algorithm Data Structure

1. Inventory Management System

```
using System;



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 interface IProgram

{

    static abstract Product BinarySearch(Product[] products,
global::System.Int32 targetId);

    static abstract Product LinearSearch(Product[] products,
global::System.Int32 targetId);

    static abstract void Main();

}

public class Program : IProgram

{

    public static Product? LinearSearch(Product[] products, int
targetId)
```

```
    {
        foreach (var product in products)
        {
            if (product.ProductId == targetId)
                return product;
        }
        return null;
    }
    public static Product? BinarySearch(Product[] products, int
targetId)
    {
        int left = 0, right = products.Length - 1;
        while (left <= right)
        {
            int mid = left + (right - left) / 2;
            if (products[mid].ProductId == targetId)
                return products[mid];
            else if (products[mid].ProductId < targetId)
                left = mid + 1;
            else
                right = mid - 1;
        }
        return null;
```

```
    }

    public static void Main()

    {

        Product[] products = {

            new Product(3, "Laptop", "Electronics"),

            new Product(1, "Shirt", "Apparel"),

            new Product(2, "Coffee Mug", "Kitchen")

        };

        Array.Sort(products, (a, b) =>
a.ProductId.CompareTo(b.ProductId));

        Product foundLinear = LinearSearch(products, 2);

        Console.WriteLine(foundLinear?.ProductName);

        Product foundBinary = BinarySearch(products, 2);

        Console.WriteLine(foundBinary?.ProductName);

    }

}
```

output

```
PS C:\Users\KIIT\Desktop\Cognizant Exercise Codes> dotnet run Program.cs
Coffee Mug
Coffee Mug
```

7. Financial Forecasting

```
using System;

public class FinancialForecast
```

```
{
    // Recursive calculation of future value
    public static double FutureValue(double presentValue,
double rate, int periods)
    {
        // Base case: no more periods
        if (periods == 0)
            return presentValue;
        // Recursive case: grow for one period, then solve for
remaining periods
        return FutureValue(presentValue * (1 + rate), rate,
periods - 1);
    }

    // Optimized recursive (using exponentiation, less depth)
    public static double FutureValueOptimized(double
presentValue, double rate, int periods)
    {
        // Base case
        if (periods == 0)
            return presentValue;
        if (periods % 2 == 0)
        {
            double half = FutureValueOptimized(presentValue,
rate, periods / 2);
            return FutureValueOptimized(half, rate, periods /
2);
        }
        else
        {
            return FutureValueOptimized(presentValue * (1 +
rate), rate, periods - 1);
        }
    }

    public static void Main()
    {
        double PV = 1000.0;
```

```
        double rate = 0.05; // 5% growth
        int n = 5; // 5 periods

        double result = FutureValue(PV, rate, n);
        Console.WriteLine($"Future Value (recursive):
{result:F2}");

        double optimized = FutureValueOptimized(PV, rate, n);
        Console.WriteLine($"Future Value (optimized recursion):
{optimized:F2}");

        // For comparison, using direct formula:
        double formula = PV * Math.Pow(1 + rate, n);
        Console.WriteLine($"Future Value (formula):
{formula:F2}");
    }
}
```

output

```
PS C:\Users\KIIT\Desktop\Cognizant Exercise Codes\exercise 7> dotnet run Program.cs
Future Value (recursive): 1276.28
Future Value (optimized recursion): 1276.28
Future Value (formula): 1276.28
```