**Design Pattern and Principles**

Q1. Logger.cs

using System;

namespace SingletonPatternExample

{

public class Logger

{

private static Logger instance;

private Logger()

{

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

}

public static Logger GetInstance()

{

if (instance == null)

{

instance = new Logger();

}

return instance;

}

public void Log(string message)

{

Console.WriteLine($"[LOG]: {message}");

}

}

}

Program.cs

using System;

namespace SingletonPatternExample

{

class Program

{

static void Main(string[] args)

{

Logger logger1 = Logger.GetInstance();

Logger logger2 = Logger.GetInstance();

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

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

if (object.ReferenceEquals(logger1, logger2))

{

Console.WriteLine("logger1 and logger2 refer to the same instance.");

}

else

{

Console.WriteLine("Singleton pattern not working correctly.");

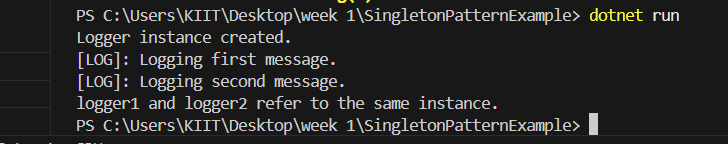
}

}

}

}

Output



Q2. Document.cs

namespace FactoryMethodPatternExample

{

public interface IDocument

{

void Open();

}

}

WordDocument.cs

using System;

namespace FactoryMethodPatternExample

{

public class WordDocument : IDocument

{

public void Open()

{

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

}

}

}

PdfDocument.cs

using System;

namespace FactoryMethodPatternExample

{

public class PdfDocument : IDocument

{

public void Open()

{

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

}

}

}

ExcelDocument.cs

using System;

namespace FactoryMethodPatternExample

{

public class ExcelDocument : IDocument

{

public void Open()

{

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

}

}

}

DocumentFactory.cs

namespace FactoryMethodPatternExample

{

public abstract class DocumentFactory

{

public abstract IDocument CreateDocument();

}

}

WordDocumentFactory.cs

namespace FactoryMethodPatternExample

{

public class WordDocumentFactory : DocumentFactory

{

public override IDocument CreateDocument()

{

return new WordDocument();

}

}

}

PdfDocumentFactory.cs

namespace FactoryMethodPatternExample

{

public class PdfDocumentFactory : DocumentFactory

{

public override IDocument CreateDocument()

{

return new PdfDocument();

}

}

}

ExcelDocumentFactory.cs

namespace FactoryMethodPatternExample

{

public class ExcelDocumentFactory : DocumentFactory

{

public override IDocument CreateDocument()

{

return new ExcelDocument();

}

}

}

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();

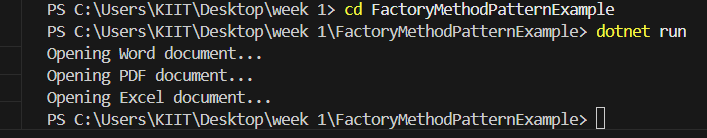
excelDoc.Open();

}

}

}

Output



Q3. 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 id, string name, string category)

{

ProductId = id;

ProductName = name;

Category = category;

}

public override string ToString()

{

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

}

}

}

Program.cs

using System;

using System.Linq;

namespace ECommerceSearch

{

class Program

{

static void Main(string[] args)

{

// Sample product data

Product[] products = {

new Product(101, "Laptop", "Electronics"),

new Product(203, "Shoes", "Footwear"),

new Product(150, "Smartphone", "Electronics"),

new Product(404, "T-shirt", "Clothing"),

new Product(301, "Watch", "Accessories")

};

Console.WriteLine("\n🔍 Linear Search for Product ID 150:");

var result1 = LinearSearch(products, 150);

Console.WriteLine(result1 != null ? result1.ToString() : "Product not found");

// Sort the array for binary search

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

Console.WriteLine("\n🔍 Binary Search for Product ID 150:");

var result2 = BinarySearch(sortedProducts, 150);

Console.WriteLine(result2 != null ? result2.ToString() : "Product not found");

}

// Linear search implementation

static Product LinearSearch(Product[] products, int targetId)

{

foreach (var product in products)

{

if (product.ProductId == targetId)

return product;

}

return null;

}

// Binary search implementation

static Product BinarySearch(Product[] products, int targetId)

{

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

while (left <= right)

{

int mid = (left + right) / 2;

if (products[mid].ProductId == targetId)

return products[mid];

else if (products[mid].ProductId < targetId)

left = mid + 1;

else

right = mid - 1;

}

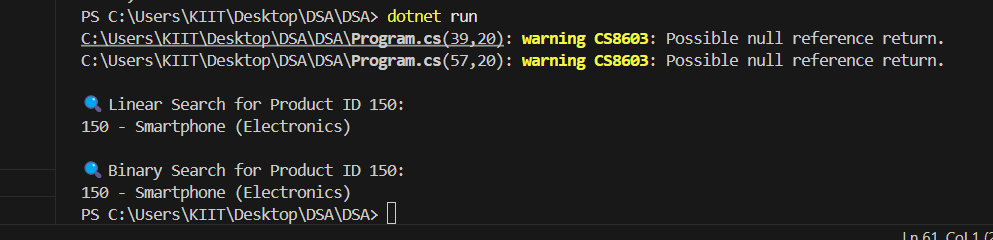
return null;

}

}

}

Output



Q7. Program.cs

using System;

namespace FinancialForecasting

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("📊 Financial Forecasting Tool\n");

// Input: Present value, annual growth rate, and number of years

Console.Write("Enter present value (e.g. 10000): ");

double presentValue = Convert.ToDouble(Console.ReadLine());

Console.Write("Enter annual growth rate (as %): ");

double growthRatePercent = Convert.ToDouble(Console.ReadLine());

double growthRate = growthRatePercent / 100.0;

Console.Write("Enter number of years: ");

int years = Convert.ToInt32(Console.ReadLine());

// Recursive forecast

double futureValueRecursive = ForecastRecursive(presentValue, growthRate, years);

Console.WriteLine($"\n🔁 Recursive Forecast: ₹{futureValueRecursive:F2}");

// Iterative forecast

double futureValueIterative = ForecastIterative(presentValue, growthRate, years);

Console.WriteLine($"🔄 Iterative Forecast: ₹{futureValueIterative:F2}");

}

// Recursive method

static double ForecastRecursive(double value, double rate, int years)

{

if (years == 0)

return value;

return ForecastRecursive(value \* (1 + rate), rate, years - 1);

}

// Iterative method (recommended for large years)

static double ForecastIterative(double value, double rate, int years)

{

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

{

value \*= (1 + rate);

}

return value;

}

}

}

Product.cs

namespace FinancialForecasting

{

    public class Product

    {

        public string ProductName { get; set; }

        public double InitialInvestment { get; set; }

        public double AnnualGrowthRate { get; set; }  // In decimal, e.g., 0.08 for 8%

        public Product(string productName, double initialInvestment, double annualGrowthRate)

        {

            ProductName = productName;

            InitialInvestment = initialInvestment;

            AnnualGrowthRate = annualGrowthRate;

        }

        public override string ToString()

        {

            return $"{ProductName} - ₹{InitialInvestment} @ {AnnualGrowthRate \* 100}% annual growth";

        }

    }

}

namespace FinancialForecasting

{

public class Product

{

public string ProductName { get; set; }

public double InitialInvestment { get; set; }

public double AnnualGrowthRate { get; set; } // In decimal, e.g., 0.08 for 8%

public Product(string productName, double initialInvestment, double annualGrowthRate)

{

ProductName = productName;

InitialInvestment = initialInvestment;

AnnualGrowthRate = annualGrowthRate;

}

public override string ToString()

{

return $"{ProductName} - ₹{InitialInvestment} @ {AnnualGrowthRate \* 100}% annual growth";

}

}

}

Output

