Design Patterns and Principles

**Exercise 1: Implementing the Singleton Pattern**

**Scenario:**

You need to ensure that a logging utility class in your application has only one instance throughout the application lifecycle to ensure consistent logging.

**# Source code:**

using System;

public class Logger

{

private static readonly Logger \_instance = new Logger();

private Logger()

{

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

}

public static Logger GetInstance()

{

return \_instance;

}

public void Log(string message)

{

Console.WriteLine("Log: " + message);

}

}

class Program

{

static void Main(string[] args)

{

Logger logger1 = Logger.GetInstance();

logger1.Log("First message");

Logger logger2 = Logger.GetInstance();

logger2.Log("Second message");

if (logger1 == logger2)

{

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

}

else

{

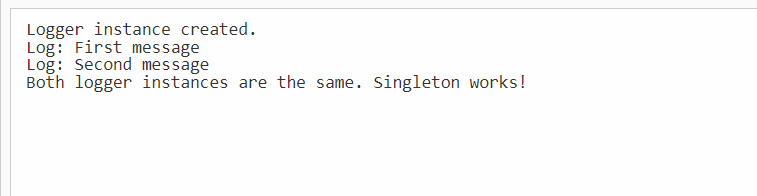
Console.WriteLine("Different instances exist. Singleton failed.");

}

}

}

**OUTPUT:**



**Exercise 2: Implementing the Factory Method Pattern**

**Scenario:**

You are developing a document management system that needs to create different types of documents (e.g., Word, PDF, Excel). Use the Factory Method Pattern to achieve this.

#**code:**

using System;

namespace FactoryMethodPatternExample

{

// Step 2: Define Document Interface

public interface IDocument

{

void Open();

}

// Step 3: Create Concrete Document Classes

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 class ExcelDocument : IDocument

{

public void Open()

{

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

}

}

// Step 4: Abstract Factory

public abstract class DocumentFactory

{

public abstract IDocument CreateDocument();

}

// Concrete Factories

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();

}

}

// Step 5: Test the Factory Method Implementation

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();

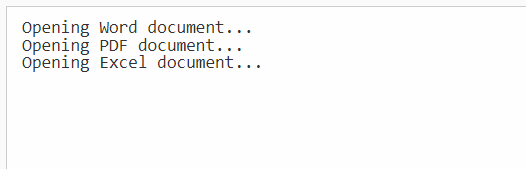
excelDoc.Open();

}

}

}

**OUTPUT: -**



**Data Structures & Algorithms :**

BIG O notaion: Big O notation is used to describe the **efficiency of algorithms** in terms of **time** or **space** as the input size grows.

**#CODE:**

using System;

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})";

}

}

class Program

{

static void Main(string[] args)

{

Product[] products = new Product[]

{

new Product(104, "Watch", "Accessories"),

new Product(102, "Shoes", "Fashion"),

new Product(105, "Phone", "Electronics"),

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

new Product(103, "Book", "Education")

};

Console.WriteLine("Linear Search for Product ID 104:");

Product result1 = LinearSearch(products, 104);

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

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

Console.WriteLine("\nBinary Search for Product ID 104:");

Product result2 = BinarySearch(products, 104);

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

}

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

{

foreach (var product in products)

{

if (product.ProductId == productId)

return product;

}

return null;

}

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;

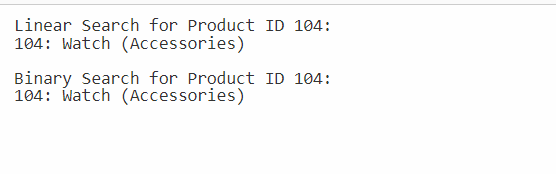
}

return null;

}

}

**OUTPUT:**



**Exercise 7: Financial Forecasting**

**Scenario:**

You are developing a financial forecasting tool that predicts future values based on past data.

**Recurison:** Recursion is when a method/function calls itself to solve a smaller part of a problem. It's especially useful for problems with repeated or nested subproblems.

**#CODE:**

using System;

class Program

{

// Recursive method to calculate future value

static double CalculateFutureValueRecursive(double presentValue, double growthRate, int years)

{

if (years == 0)

return presentValue;

return (1 + growthRate) \* CalculateFutureValueRecursive(presentValue, growthRate, years - 1);

}

static void Main(string[] args)

{

double presentValue = 10000; // starting money

double growthRate = 0.05; // 5% annual growth

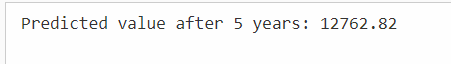
int years = 5;

double futureValue = CalculateFutureValueRecursive(presentValue, growthRate, years);

Console.WriteLine($"Predicted value after {years} years: {futureValue:F2}");

}

}



### Time Complexity:

Recursive calls : n

Therefore, Time Complexity : O(n)

**Optimization:** We can use memoization for bteer perfromance .

static double CalculateFutureValueIterative(double presentValue, double growthRate, int years)

{

double result = presentValue;

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

{

result \*= (1 + growthRate);

}

return result;

}