# 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.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **SingletonPatternExample**.
2. **Define a Singleton Class:**
   * Create a class named Logger that has a private static instance of itself.
   * Ensure the constructor of Logger is private.
   * Provide a public static method to get the instance of the Logger class.
3. **Implement the Singleton Pattern:**
   * Write code to ensure that the Logger class follows the Singleton design pattern.
4. **Test the Singleton Implementation:**
   * Create a test class to verify that only one instance of Logger is created and used across the application.

## Code :

using System;

public sealed class Logger

{

private static Logger \_instance = null;

private static readonly object \_lock = new object();

private Logger()

{

Console.WriteLine("Logger instance created");

}

public static Logger GetInstance()

{

if (\_instance == null)

{

lock (\_lock)

{

if (\_instance == null)

{

\_instance = new Logger();

}

}

}

return \_instance;

}

public void Log(string message)

{

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

}

}

public class Program

{

public static void Main()

{

Logger logger1 = Logger.GetInstance();

logger1.Log("This is the first log message.");

Logger logger2 = Logger.GetInstance();

logger2.Log("This is the second log message.");

if (Object.ReferenceEquals(logger1, logger2))

{

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

}

else

{

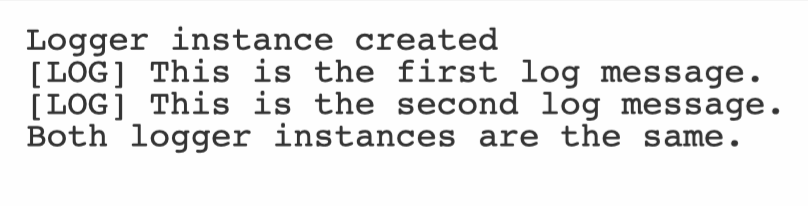
Console.WriteLine("Different instances exist! Singleton pattern failed.");

}

}

}

## 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.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **FactoryMethodPatternExample**.
2. **Define Document Classes:**
   * Create interfaces or abstract classes for different document types such as **WordDocument**, **PdfDocument**, and **ExcelDocument**.
3. **Create Concrete Document Classes:**
   * Implement concrete classes for each document type that implements or extends the above interfaces or abstract classes.
4. **Implement the Factory Method:**
   * Create an abstract class **DocumentFactory** with a method **createDocument()**.
   * Create concrete factory classes for each document type that extends DocumentFactory and implements the **createDocument()** method.
5. **Test the Factory Method Implementation:**
   * Create a test class to demonstrate the creation of different document types using the factory method.

## Code :

using System;

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 class ExcelDocument : IDocument

{

public void Open()

{

Console.WriteLine("Opening 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();

}

}

public class Program

{

public static void Main()

{

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 and Algorithms

**Exercise 2: E-commerce Platform Search Function**

**Scenario:**

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

**Steps:**

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.
2. **Setup:**
   * Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
3. **Implementation:**
   * Implement linear search and binary search algorithms.
   * Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
   * Compare the time complexity of linear and binary search algorithms.
   * Discuss which algorithm is more suitable for your platform and why.

## Code :

using System;

using System.Collections.Generic;

// Step 1: Define the Product class

public class Product : IComparable<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 int CompareTo(Product other)

{

return this.ProductId.CompareTo(other.ProductId);

}

public override string ToString()

{

return $"ID: {ProductId}, Name: {ProductName}, Category: {Category}";

}

}

public class Program

{

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;

int 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;

}

public static void Main()

{

Product[] productList = new Product[]

{

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

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

new Product(150, "Coffee Maker", "Appliances"),

new Product(102, "T-Shirt", "Clothing"),

new Product(175, "Phone", "Electronics")

};

Console.WriteLine("=== Linear Search ===");

var result1 = LinearSearch(productList, 150);

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

Array.Sort(productList);

Console.WriteLine("\n=== Binary Search ===");

var result2 = BinarySearch(productList, 150);

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

Console.WriteLine("\n=== Time Complexity Analysis ===");

Console.WriteLine("Linear Search: O(n) - checks each product one-by-one");

Console.WriteLine("Binary Search: O(log n) - halves the search space each step (needs sorted array)");

Console.WriteLine("Binary search is better for performance with large datasets");

}

}

## Output :

**Exercise 7: Financial Forecasting**

**Scenario:**

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

**Steps:**

1. **Understand Recursive Algorithms:**
   * Explain the concept of recursion and how it can simplify certain problems.
2. **Setup:**
   * Create a method to calculate the future value using a recursive approach.
3. **Implementation:**
   * Implement a recursive algorithm to predict future values based on past growth rates.
4. **Analysis:**
   * Discuss the time complexity of your recursive algorithm.
   * Explain how to optimize the recursive solution to avoid excessive computation.

## Code :

using System;

public class FinancialForecast

{

public static double ForecastValue(double initialValue, double growthRate, int years)

{

if (years == 0)

return initialValue;

return ForecastValue(initialValue, growthRate, years - 1) \* (1 + growthRate);

}

public static double ForecastValueMemo(double initialValue, double growthRate, int years, double[] memo)

{

if (years == 0)

return initialValue;

if (memo[years] != 0)

return memo[years];

memo[years] = ForecastValueMemo(initialValue, growthRate, years - 1, memo) \* (1 + growthRate);

return memo[years];

}

public static void Main()

{

double initial = 1000;

double rate = 0.05;

int years = 5;

Console.WriteLine("=== Recursive Forecast ===");

double futureValue = ForecastValue(initial, rate, years);

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

Console.WriteLine("\n=== Optimized (Memoized) Forecast ===");

double[] memo = new double[years + 1];

double optimizedValue = ForecastValueMemo(initial, rate, years, memo);

Console.WriteLine($"Future value after {years} years: {optimizedValue:F2}");

}

}

## Output :

