**Digital Nurture 4.0 .NET FSE (Week-1 Report)**

By-Jayati Pal

**Module-1: Design Pattern and Principles**

Exercise 1: Implementing the Singleton Pattern

Code for Logger.cs:

using System;

public class Logger

{

private static Logger instance;

private Logger() { }

public static Logger GetInstance()

{

if (instance == null)

{

instance = new Logger();

}

return instance;

}

public void Log(string message)

{

Console.WriteLine("[LOG]: " + message);

}

}

Code for Program.cs:

using System;

class Program

{

static void Main(string[] args)

{

Logger logger1 = Logger.GetInstance();

Logger logger2 = Logger.GetInstance();

logger1.Log("First log message");

logger2.Log("Second log message");

if (logger1 == logger2)

{

Console.WriteLine("Both logger1 and logger2 are the same instance.");

}

else

{

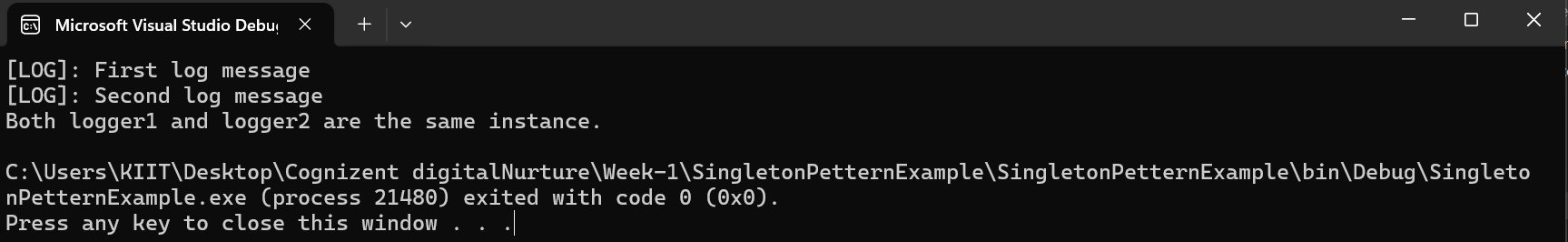
Console.WriteLine("Different instances exist!");

}

}

}

Output:



Exercise 2: Implementing the Factory Method Pattern

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();

}

}

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();

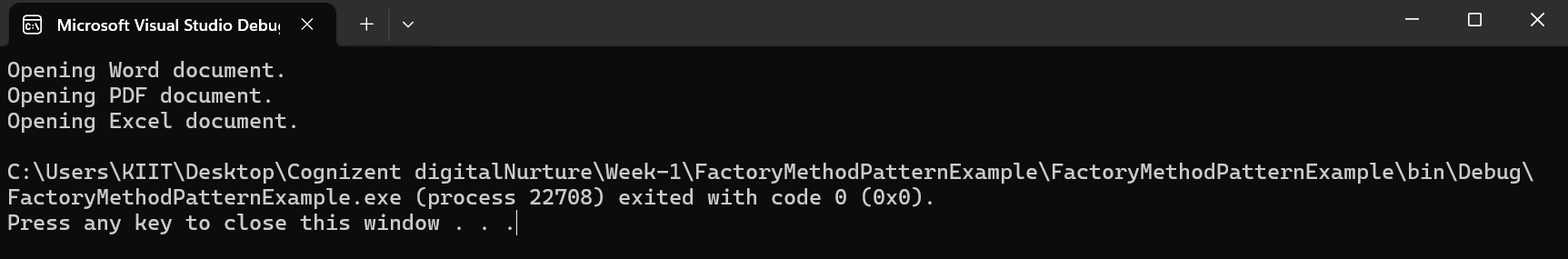
IDocument excelDoc = excelFactory.CreateDocument();

excelDoc.Open();

}

}

Output:



**Module-2: Data Structures and Algorithm**

Exercise 2: E-commerce Platform Search Function

Code:

using System;

namespace ECommerceProductSearch

{

internal class Program

{

public static Product LinearSearch(Product[] products, string targetName)

{

foreach (var product in products)

{

if (product.ProductName.Equals(targetName, StringComparison.OrdinalIgnoreCase))

{

return product;

}

}

return null;

}

public static Product BinarySearch(Product[] products, string targetName)

{

int left = 0;

int right = products.Length - 1;

while (left <= right)

{

int mid = left + (right - left) / 2;

int comparison = string.Compare(products[mid].ProductName, targetName, true);

if (comparison == 0)

return products[mid];

else if (comparison < 0)

left = mid + 1;

else

right = mid - 1;

}

return null;

}

static void Main(string[] args)

{

Product[] products = new Product[]

{

new Product(1, "Laptop", "Electronics"),

new Product(2, "Shoes", "Fashion"),

new Product(3, "Watch", "Accessories"),

new Product(4, "Headphones", "Electronics"),

new Product(5, "T-shirt", "Fashion")

};

// Linear Search

Console.WriteLine("Enter product name to search (Linear):");

string searchName = Console.ReadLine();

var resultLinear = LinearSearch(products, searchName);

Console.WriteLine(resultLinear != null ? $"Found: {resultLinear}" : "Product not found (Linear Search)");

// Binary Search requires sorted array

Array.Sort(products, (p1, p2) => string.Compare(p1.ProductName, p2.ProductName, true));

Console.WriteLine("Enter product name to search (Binary):");

string searchNameBinary = Console.ReadLine();

var resultBinary = BinarySearch(products, searchNameBinary);

Console.WriteLine(resultBinary != null ? $"Found: {resultBinary}" : "Product not found (Binary Search)");

Console.WriteLine("Press any key to exit.");

Console.ReadKey();

}

}

}

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 override string ToString()

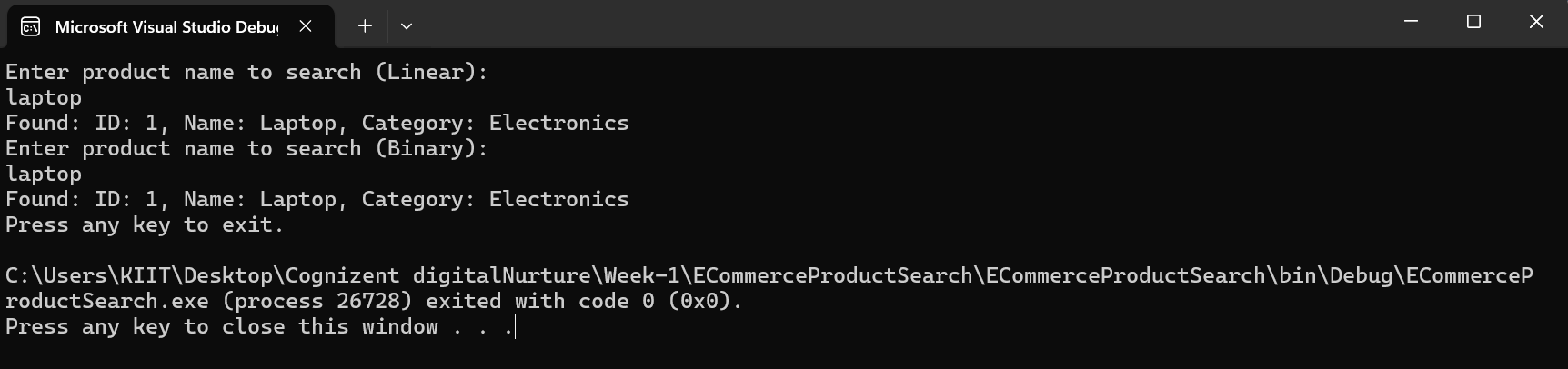
{

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

}

}

Output:



**Understanding and Analysis**

* Big O notation is a way to measure how an algorithm’s runtime or memory usage grows as input size increases
* Best case is when the target element is found immediately like the first element of an array, average case is when any random midway element is found, and worst case is when the last or missing element is made to find.
* Best case for both linear and binary search is O(1) while the average and worst case for them are O(n) and O(log n) respectively
* Binary Search is better suited for my platform as it does the computation is lesser time for large-scale e-commerce provided the products must be sorted by search key.

Exercise 7: Financial Forecasting

Code:

using System;

namespace FinancialForecasting

{

class Program

{

static double FutureValue(double initialValue, double growthRate, int periods)

{

if (periods == 0)

return initialValue;

else

return FutureValue(initialValue, growthRate, periods - 1) \* (1 + growthRate);

}

static void Main(string[] args)

{

Console.WriteLine("Enter initial value:");

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

Console.WriteLine("Enter growth rate (e.g., 0.05 for 5%):");

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

Console.WriteLine("Enter number of periods:");

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

double forecast = FutureValue(initial, rate, periods);

Console.WriteLine($"Forecasted value after {periods} periods: {forecast:F2}");

Console.WriteLine("Press any key to exit.");

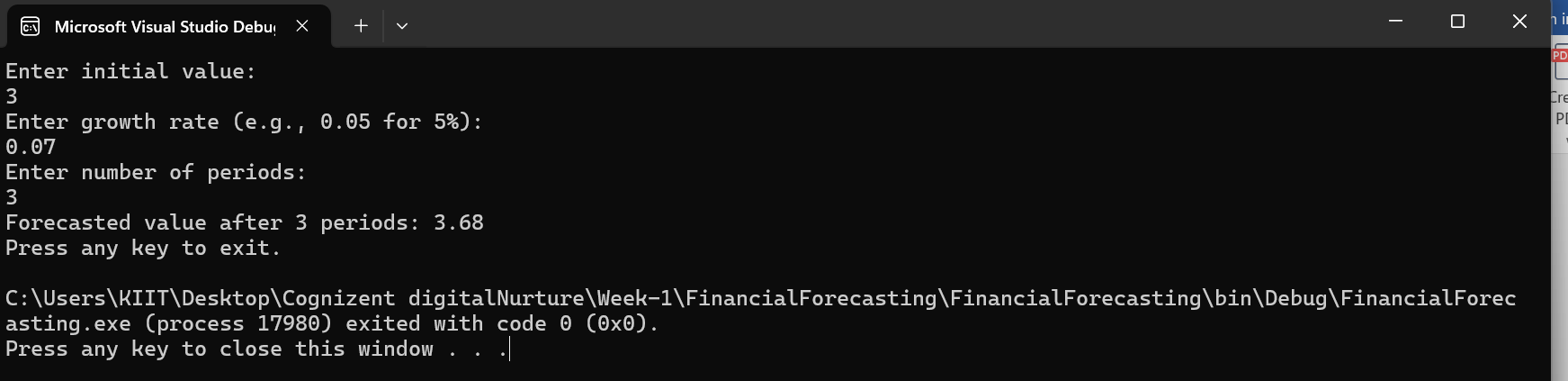
Console.ReadKey();

}

}

}

Output:



**Understanding and Analysis**

* Recursion is a programming technique where a function calls itself until it reaches a base condition, it can simplify certain problems by breaking down complex tasks into simpler, repeatable steps, making the code easier to understand.
* This recursive function has linear time complexity(n), because it makes one recursive call per period
* We can optimize recursion by memorization, by converting it to iteration or by tail recursion.