**WEEK 1**

**DESIGN PATTERN AND PRINCIPLES/DATA STRUCTURES AND ALGORITHMS**

**DESIGN PATTERN 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.

**Code:**

public class LoggerTest {

    public static void main(String[] args) {

        Logger logger1 = Logger.getInstance();

        logger1.log("First log message");

        Logger logger2 = Logger.getInstance();

        logger2.log("Second log message");

        if (logger1 == logger2) {

            System.out.println("Only one Logger instance exists. Singleton confirmed.");

        } else {

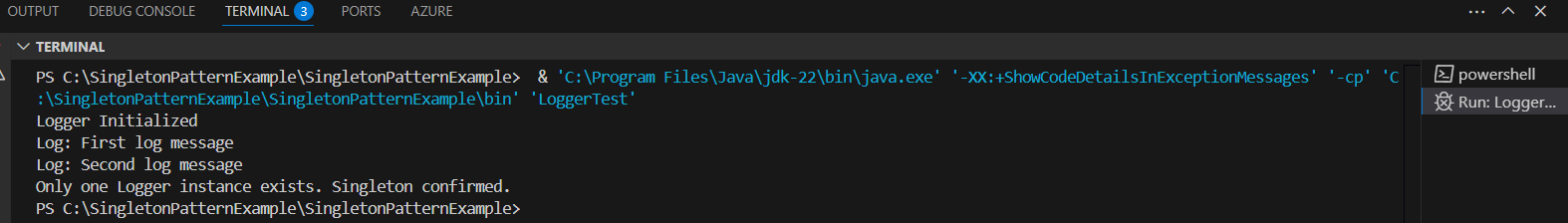
            System.out.println("Multiple instances found. Singleton failed.");

        }

    }

}

**OUTPUT:**



Logger Initialized

[LOG]: Application started.

[LOG]: This should use the same logger instance.

Both logger instances are the same (Singleton Verified).

**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:**

public class FactoryTest {

    public static void main(String[] args) {

        DocumentFactory wordFactory = new WordDocumentFactory();

        Document wordDoc = wordFactory.createDocument();

        wordDoc.open();

        DocumentFactory pdfFactory = new PdfDocumentFactory();

        Document pdfDoc = pdfFactory.createDocument();

        pdfDoc.open();

        DocumentFactory excelFactory = new ExcelDocumentFactory();

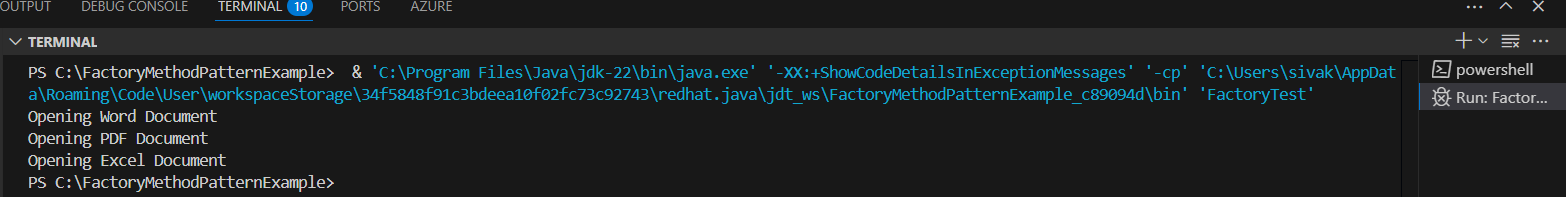
        Document excelDoc = excelFactory.createDocument();

        excelDoc.open();

    }

}

**OUTPUT:**



Opening Word Document...

Opening PDF Document...

Opening Excel Document...

**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.

**Code:**

import java.util.Arrays;

import java.util.Comparator;

public class SearchDemo {

    public static Product linearSearch(Product[] products, String name) {

        for (Product p : products) {

            if (p.productName.equalsIgnoreCase(name)) {

                return p;

            }

        }

        return null;

    }

    public static Product binarySearch(Product[] products, String name) {

        Arrays.sort(products, Comparator.comparing(p -> p.productName.toLowerCase()));

        int low = 0, high = products.length - 1;

        while (low <= high) {

            int mid = (low + high) / 2;

            int compare = products[mid].productName.compareToIgnoreCase(name);

            if (compare == 0) return products[mid];

            else if (compare < 0) low = mid + 1;

            else high = mid - 1;

        }

        return null;

    }

    public static void main(String[] args) {

        Product[] products = {

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

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

            new Product(103, "Phone", "Electronics"),

            new Product(104, "Table", "Furniture"),

            new Product(105, "Shoes", "Footwear")

        };

        System.out.println("🔎 Linear Search for 'Phone':");

        Product result1 = linearSearch(products, "Phone");

        System.out.println(result1 != null ? "Found: " + result1 : "Not Found");

        System.out.println("\n🔎 Binary Search for 'Table':");

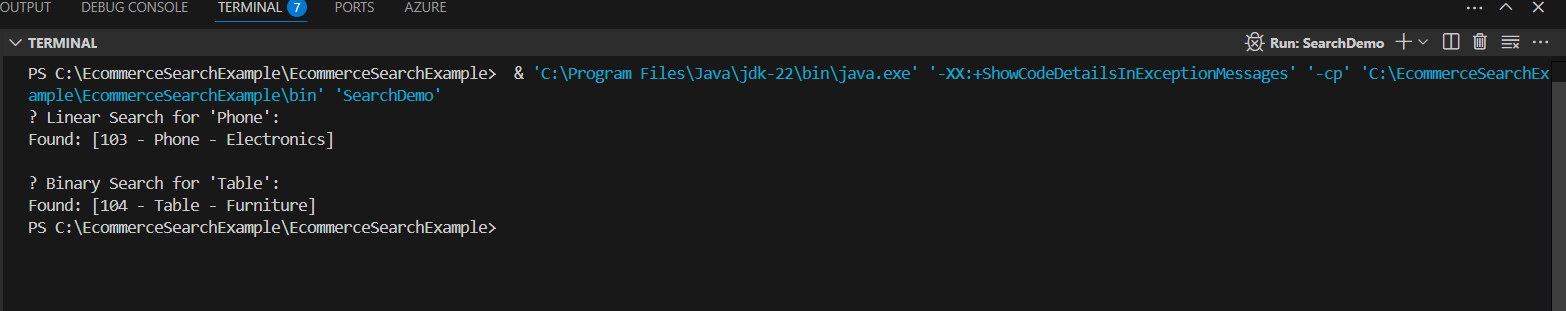
        Product result2 = binarySearch(products, "Table");

        System.out.println(result2 != null ? "Found: " + result2 : "Not Found");

    }

}

**OUTPUT:**



? Linear Search:

Found: 103 - Phone (Electronics)

? Binary Search (after sorting):

Found: 103 - Phone (Electronics)

**Exercise 7: Financial Forecasting**

**Scenario:**

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

**Code:**

public class Forecast {

    public static double futureValueRecursive(double presentValue, double growthRate, int years) {

        if (years == 0) {

            return presentValue;

        }

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

    }

    public static double futureValueMemo(double presentValue, double growthRate, int years, double[] memo) {

        if (years == 0) {

            return presentValue;

        }

        if (memo[years] != 0) {

            return memo[years];

        }

        memo[years] = futureValueMemo(presentValue, growthRate, years - 1, memo) \* (1 + growthRate);

        return memo[years];

    }

    public static void main(String[] args) {

        double presentValue = 10000;

        double growthRate = 0.10;

        int years = 5;

        System.out.println("🔁 Recursive Forecast (No Memoization):");

        double result1 = futureValueRecursive(presentValue, growthRate, years);

        System.out.printf("Future Value after %d years = %.2f\n", years, result1);

        System.out.println("\n⚡ Optimized Forecast (With Memoization):");

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

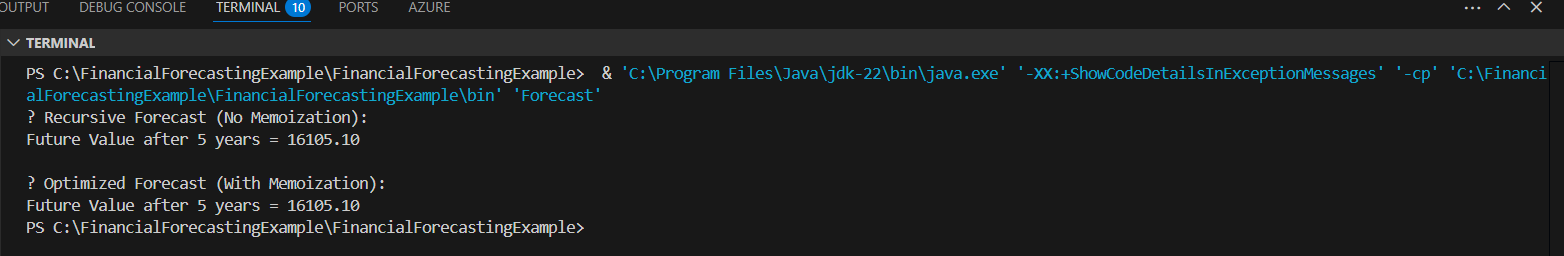
        double result2 = futureValueMemo(presentValue, growthRate, years, memo);

        System.out.printf("Future Value after %d years = %.2f\n", years, result2);

    }

}

**OUTPUT:**



Future Value (recursive): ?1610.5100000000002

Future Value (memoized): ?1610.5100000000002