**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.

class SingletonPatternExample {

    // Singleton Logger class

    static class Logger {

        // Private static instance

        private static Logger instance;

        // Private constructor

        private Logger() {

            System.out.println("Logger instance created.");

        }

        // Public method to return the single instance

        public static Logger getInstance() {

            if (instance == null) {

                instance = new Logger();

            }

            return instance;

        }

        // Sample logging method

        public void log(String message) {

            System.out.println("Log message: " + message);

        }

    }

    // Main method to test Singleton

    public static void main(String[] args) {

        Logger logger1 = Logger.getInstance();

        Logger logger2 = Logger.getInstance();

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

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

        // Check if both instances are the same

        if (logger1 == logger2) {

            System.out.println("Both logger1 and logger2 are the same instance.");

        } else {

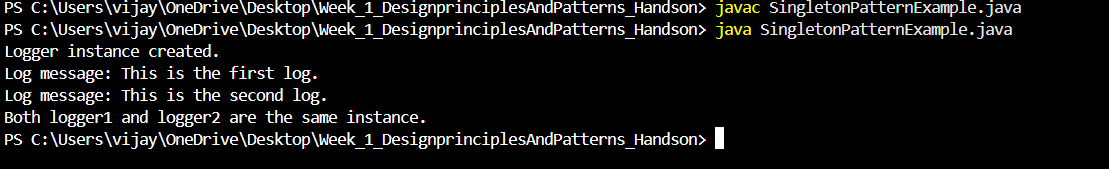
            System.out.println("Singleton failed: Different instances exist.");

        }

    }

}

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.

**PROGRAM CODE:**

public class FactoryMethodPatternExample {

    // Step 1: Document interface

    interface Document {

        void open();

    }

    // Step 2: Concrete Document Classes

    static class WordDocument implements Document {

        public void open() {

            System.out.println("Opening Word document.");

        }

    }

    static class PdfDocument implements Document {

        public void open() {

            System.out.println("Opening PDF document.");

        }

    }

    static class ExcelDocument implements Document {

        public void open() {

            System.out.println("Opening Excel document.");

        }

    }

    // Step 3: Abstract Factory

    static abstract class DocumentFactory {

        public abstract Document createDocument();

    }

    // Step 4: Concrete Factories

    static class WordDocumentFactory extends DocumentFactory {

        public Document createDocument() {

            return new WordDocument();

        }

    }

    static class PdfDocumentFactory extends DocumentFactory {

        public Document createDocument() {

            return new PdfDocument();

        }

    }

    static class ExcelDocumentFactory extends DocumentFactory {

        public Document createDocument() {

            return new ExcelDocument();

        }

    }

    // Step 5: Test the Factory Method

    public static void main(String[] args) {

        // Create Word document

        DocumentFactory wordFactory = new WordDocumentFactory();

        Document wordDoc = wordFactory.createDocument();

        wordDoc.open();

        // Create PDF document

        DocumentFactory pdfFactory = new PdfDocumentFactory();

        Document pdfDoc = pdfFactory.createDocument();

        pdfDoc.open();

        // Create Excel document

        DocumentFactory excelFactory = new ExcelDocumentFactory();

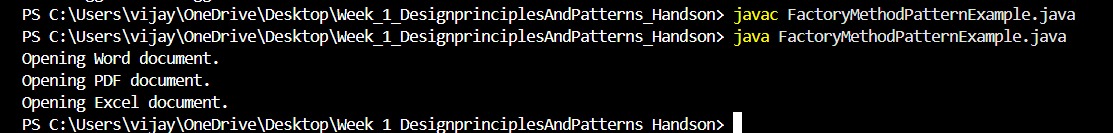
        Document excelDoc = excelFactory.createDocument();

        excelDoc.open();

    }

}

**OUTPUT:**



**ALOGORITHM\_DATA STRUCTURES**

**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.

**PROGRAM CODE:**

import java.util.Arrays;

import java.util.Comparator;

public class EcommerceSearchExample {

    // Product class with toString for display

    static class Product {

        int productId;

        String productName;

        String category;

        Product(int id, String name, String cat) {

            productId = id;

            productName = name;

            category = cat;

        }

        @Override

        public String toString() {

            return "[" + productId + ", " + productName + ", " + category + "]";

        }

    }

    // Linear Search by product name

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

        for (Product p : products) {

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

                return p;

            }

        }

        return null;

    }

    // Binary Search by product name (array must be sorted by name)

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

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

        while (left <= right) {

            int mid = (left + right) / 2;

            int cmp = products[mid].productName.compareToIgnoreCase(targetName);

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

            else if (cmp < 0) left = mid + 1;

            else right = mid - 1;

        }

        return null;

    }

    public static void main(String[] args) {

        // Array of products

        Product[] products = {

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

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

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

            new Product(104, "Shoes", "Footwear"),

            new Product(105, "Watch", "Accessories")

        };

        // Sort a copy of the array for binary search

        Product[] sortedProducts = Arrays.copyOf(products, products.length);

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

        // Search target

        String target = "Phone";

        // Perform Linear Search

        Product linearResult = linearSearch(products, target);

        System.out.println("Linear Search Result: " + (linearResult != null ? linearResult : "Not Found"));

        // Perform Binary Search

        Product binaryResult = binarySearch(sortedProducts, target);

        System.out.println("Binary Search Result: " + (binaryResult != null ? binaryResult : "Not Found"));

        // Time Complexity Summary

        System.out.println("\n--- Time Complexity Analysis ---");

        System.out.println("Linear Search: O(n)");

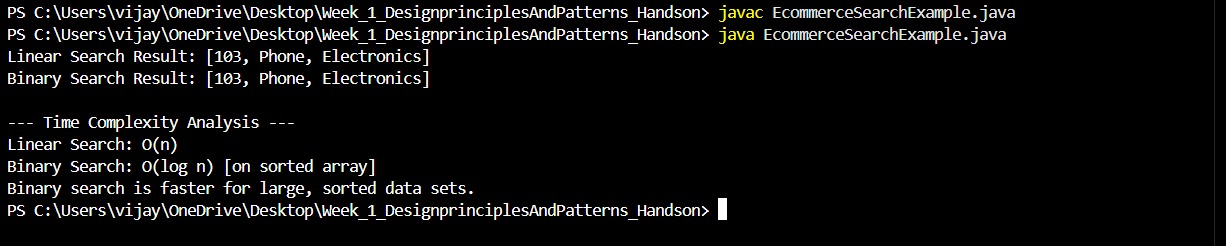
        System.out.println("Binary Search: O(log n) [on sorted array]");

        System.out.println("Binary search is faster for large, sorted data sets.");

    }

}

**OUTPUT:**



**Exercise 7: Financial Forecasting**

**Scenario:**

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

**PROGRAM CODE:**

public class FinancialForecasting {

    // Recursive method to calculate future value

    public static double forecastRecursive(double initialAmount, double growthRate, int years) {

        if (years == 0) {

            return initialAmount; // Base case

        } else {

            return forecastRecursive(initialAmount, growthRate, years - 1) \* (1 + growthRate);

        }

    }

    // Optimized method using dynamic programming (bottom-up approach)

    public static double forecastOptimized(double initialAmount, double growthRate, int years) {

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

        futureValues[0] = initialAmount;

        for (int i = 1; i <= years; i++) {

            futureValues[i] = futureValues[i - 1] \* (1 + growthRate);

        }

        return futureValues[years];

    }

    public static void main(String[] args) {

        // Initial data

        double initialAmount = 1000.0;   // starting investment

        double growthRate = 0.08;        // 8% annual growth

        int years = 5;                   // number of years to forecast

        // Forecast using recursive method

        double valueRecursive = forecastRecursive(initialAmount, growthRate, years);

        System.out.printf("Future Value (Recursive) after %d years: %.2f\n", years, valueRecursive);

        // Forecast using optimized method

        double valueOptimized = forecastOptimized(initialAmount, growthRate, years);

        System.out.printf("Future Value (Optimized) after %d years: %.2f\n", years, valueOptimized);

    }

}

**OUTPUT:**

