**WEEK-1**

**EXERCISE-1 :- IMPLEMENTING THE SINGLETONE PATTERN**

**PROGRAM :**

class Logger {

private static Logger instance;

private Logger() {

System.out.println("Logger Initialized");

}

public static Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

public void log(String message) {

System.out.println("[LOG]: " + message);

}

}

public class SingletonPatternExample {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

logger1.log("Application started");

Logger logger2 = Logger.getInstance();

logger2.log("Continuing application...");

if (logger1 == logger2) {

System.out.println("Both logger instances are the same (singleton confirmed).");

} else {

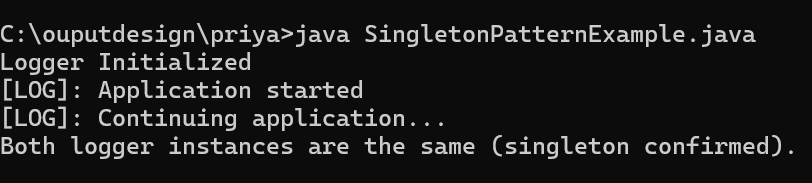
System.out.println("Different logger instances exist (singleton failed).");

}

}

}

**OUTPUT :**

****

**EXERCISE-2 :- IMPLEMENTING THE FACTORY METHOD**

**PATTERN**

**PROGRAM :**

interface Document {

void open();

void close();

void save();

}

class WordDocument implements Document {

@Override

public void open() {

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

}

@Override

public void close() {

System.out.println("Closing Word Document");

}

@Override

public void save() {

System.out.println("Saving Word Document");

}

}

class PdfDocument implements Document {

@Override

public void open() {

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

}

@Override

public void close() {

System.out.println("Closing PDF Document");

}

@Override

public void save() {

System.out.println("Saving PDF Document");

}

}

class ExcelDocument implements Document {

@Override

public void open() {

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

}

@Override

public void close() {

System.out.println("Closing Excel Document");

}

@Override

public void save() {

System.out.println("Saving Excel Document");

}

}

abstract class DocumentFactory {

public abstract Document createDocument();

}

class WordDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new WordDocument();

}

}

class PdfDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new PdfDocument();

}

}

class ExcelDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new ExcelDocument();

}

}

public class FactoryMethodPatternExample {

public static void main(String[] args) {

DocumentFactory wordFactory = new WordDocumentFactory();

Document wordDoc = wordFactory.createDocument();

wordDoc.open();

wordDoc.save();

wordDoc.close();

System.out.println();

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open();

pdfDoc.save();

pdfDoc.close();

System.out.println();

DocumentFactory excelFactory = new ExcelDocumentFactory();

Document excelDoc = excelFactory.createDocument();

excelDoc.open();

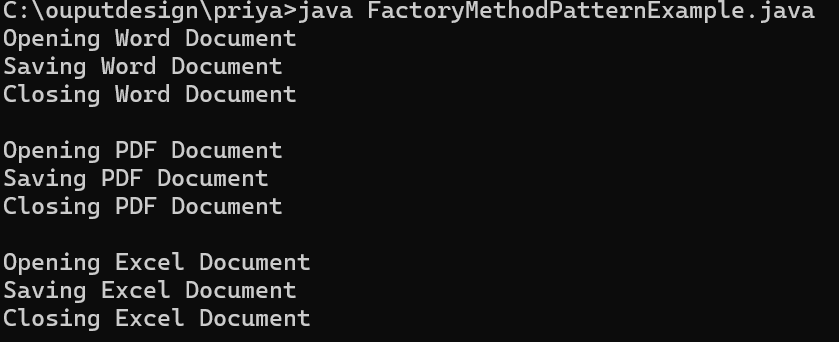
excelDoc.save();

excelDoc.close();

}

}

**OUTPUT:**



**EXERCISE-3 :- COMMERCE PLATFORM SEARCH FUNCTION**

**PROGRAM :**

import java.util.Arrays;

import java.util.Comparator;

public class EcommerceSearch {

static class Product {

int productId;

String productName;

String category;

public Product(int productId, String productName, String category) {

this.productId = productId;

this.productName = productName;

this.category = category;

}

@Override

public String toString() {

return "Product[ID=" + productId + ", Name=" + productName + ", Category=" + category + "]";

}

}

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

for (Product product : products) {

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

return product;

}

}

return null;

}

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

int left = 0;

int right = sortedProducts.length - 1;

while (left <= right) {

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

int comparison = sortedProducts[mid].productName.compareToIgnoreCase(targetName);

if (comparison == 0) {

return sortedProducts[mid];

} else if (comparison < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

public static void sortProductsByName(Product[] products) {

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

}

public static void main(String[] args) {

Product[] products = {

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

new Product(2, "Smartphone", "Electronics"),

new Product(3, "Shoes", "Footwear"),

new Product(4, "Book", "Stationery"),

new Product(5, "T-Shirt", "Clothing")

};

System.out.println("=== Linear Search ===");

String searchName = "Shoes";

Product foundProduct = linearSearch(products, searchName);

if (foundProduct != null) {

System.out.println("Product found: " + foundProduct);

} else {

System.out.println("Product not found.");

}

sortProductsByName(products);

System.out.println("\n=== Binary Search ===");

searchName = "Book";

foundProduct = binarySearch(products, searchName);

if (foundProduct != null) {

System.out.println("Product found: " + foundProduct);

} else {

System.out.println("Product not found.");

}

System.out.println("\n=== Analysis ===");

System.out.println("Big O Notation describes the worst-case time complexity of algorithms.");

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

System.out.println(" - Best Case: O(1) (first element matches)");

System.out.println(" - Average/Worst Case: O(n) (needs to check many or all elements)");

System.out.println("Binary Search:");

System.out.println(" - Requires sorted array.");

System.out.println(" - Best Case: O(1) (middle element matches)");

System.out.println(" - Average/Worst Case: O(log n) (divides search space by half repeatedly)");

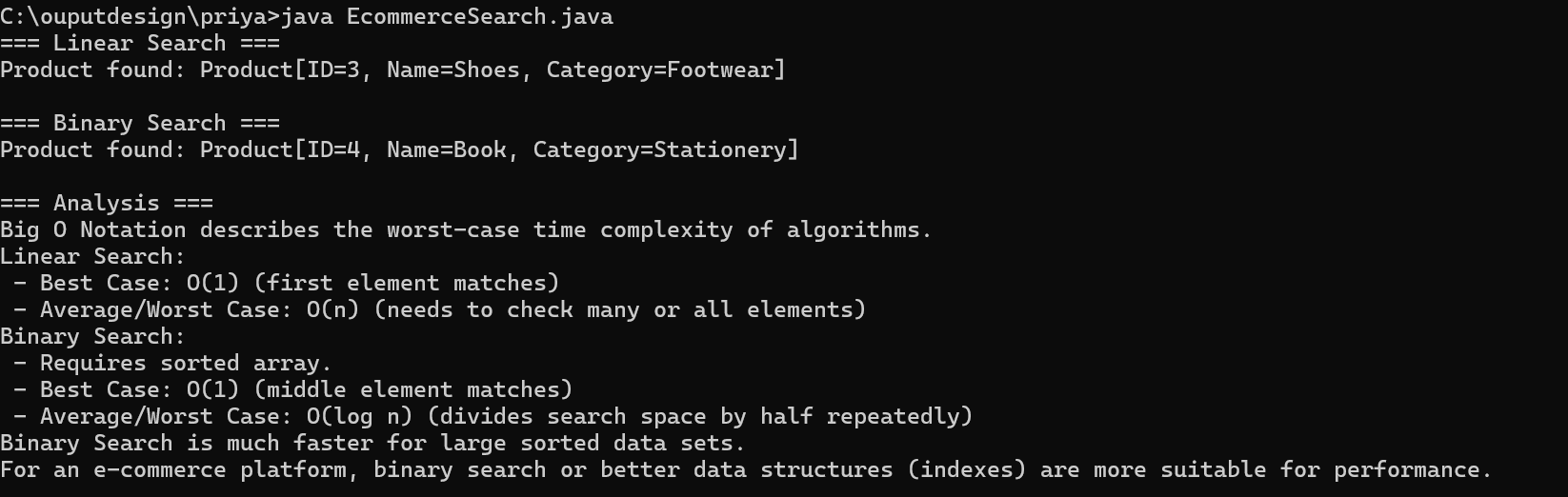
System.out.println("Binary Search is much faster for large sorted data sets.");

System.out.println("For an e-commerce platform, binary search or better data structures (indexes) are more suitable for performance.");

}

}

**OUTPUT :**



**EXERCISE-4 :- FINANCIAL FORECASTING**

**PROGRAM :**

public class FinancialForecasting {

/\*\*

\* Recursive method to calculate future value after 'n' periods

\* with a given initial principal and growth rate per period.

\*

\* Formula: FV(n) = FV(n-1) \* (1 + rate)

\* Base case: FV(0) = principal

\*

\* @param principal initial amount

\* @param rate growth rate per period (e.g., 0.05 for 5%)

\* @param periods number of periods into the future

\* @return future value after 'periods'

\*/

public static double futureValueRecursive(double principal, double rate, int periods) {

if (periods == 0) {

return principal;

}

return futureValueRecursive(principal, rate, periods - 1) \* (1 + rate);

}

/\*\*

\* Optimized recursive method using memoization (caching results)

\* to avoid redundant calculations.

\*/

public static double futureValueMemo(double principal, double rate, int periods, double[] memo) {

if (periods == 0) {

memo[0] = principal;

return principal;

}

if (memo[periods] != 0) {

return memo[periods]; // return cached result

}

memo[periods] = futureValueMemo(principal, rate, periods - 1, memo) \* (1 + rate);

return memo[periods];

}

public static void main(String[] args) {

double principal = 1000;

double rate = 0.05;

int periods = 10;

double futureValue = futureValueRecursive(principal, rate, periods);

System.out.printf("Future Value (recursive): %.2f%n", futureValue);

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

double futureValueOptimized = futureValueMemo(principal, rate, periods, memo);

System.out.printf("Future Value (optimized recursion): %.2f%n", futureValueOptimized);

System.out.println("\n=== Analysis ===");

System.out.println("Time Complexity of recursive solution without memoization: O(n)");

System.out.println("Each recursive call reduces the problem by 1 period until base case.");

System.out.println("Memoization optimizes by caching intermediate results, avoiding redundant calls.");

}

}

**OUTPUT :**

