**WEEK-1**

**EXERCISE-1- Implementing the singleton pattern:**

**Skill:Design principles & Patterns**

**Program:**

public class SingletonPatternExample {

// Singleton Logger class

static class Logger {

// Private static instance

private static Logger instance;

// Private constructor

private Logger() {

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

}

// Public method to get the single instance

public static Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

// Logger method

public void log(String message) {

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

}

}

// Main method to test Singleton

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("Both logger1 and logger2 point to the same instance.");

} else {

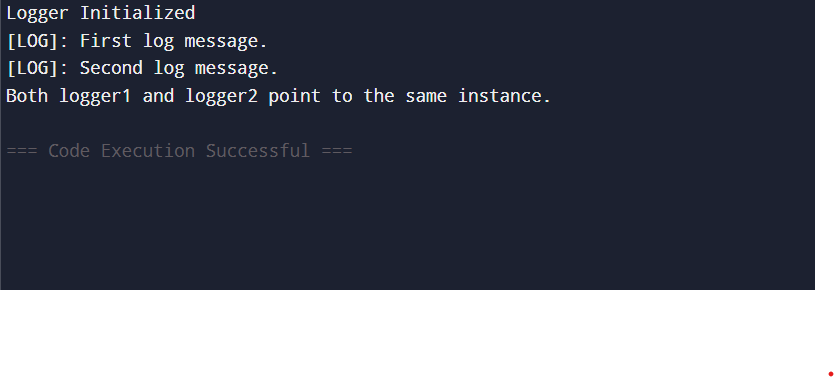
System.out.println("Different instances exist. Singleton failed.");

}

}

}

**Output:**



**EXERCISE-2- Implementing the Factory Method Pattern:**

**Skill:Design principles & Patterns**

**Program:**

public class FactoryMethodPatternExample {

// Step 2: Document interface

interface Document {

void open();

}

// Step 3: 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 4: Abstract factory class

abstract static class DocumentFactory {

public abstract Document createDocument();

}

// Step 4: Concrete factory classes

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 pattern

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();

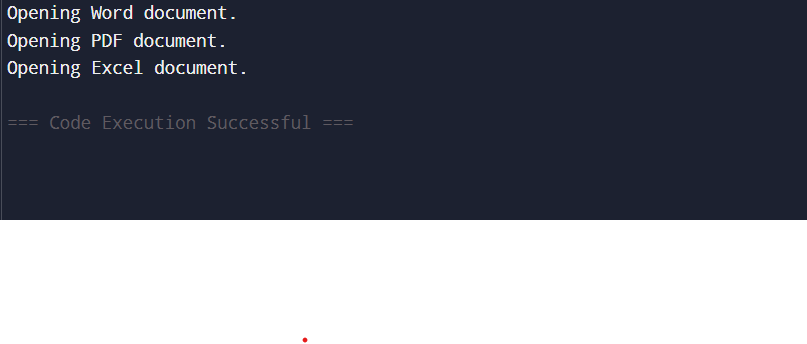
Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

**Output:**



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

**Skill:Data structures and Algorithms**

**Program:**

import java.util.Arrays;

import java.util.Comparator;

public class EcommerceSearch {

// Step 1: Understanding Asymptotic Notation

/\*

\* Big O notation describes the upper bound of time or space complexity.

\* It's used to understand how performance scales with input size.

\*

\* Linear Search:

\* Best Case: O(1) - first item matches

\* Average: O(n/2)

\* Worst: O(n)

\*

\* Binary Search:

\* Best Case: O(1)

\* Average: O(log n)

\* Worst: O(log n)

\* Note: requires sorted data

\*/

// Step 2: Product class

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 productId + " - " + productName + " - " + category;

}

}

// Step 3: Linear Search by product name

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

for (Product product : products) {

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

return product;

}

}

return null;

}

// Step 3: Binary Search by product name (requires sorted array)

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

int low = 0;

int high = products.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

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

if (compare == 0) {

return products[mid];

} else if (compare < 0) {

low = mid + 1;

} else {

high = mid - 1;

}

}

return null;

}

// Step 4: Display and test both search methods

public static void main(String[] args) {

Product[] products = {

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

new Product(102, "Phone", "Electronics"),

new Product(103, "Shirt", "Fashion"),

new Product(104, "Book", "Education"),

new Product(105, "Headphones", "Electronics")

};

// Linear Search

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

Product resultLinear = linearSearch(products, "Shirt");

System.out.println(resultLinear != null ? resultLinear : "Product not found");

// Binary Search

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

System.out.println("\nBinary Search for 'Shirt':");

Product resultBinary = binarySearch(products, "Shirt");

System.out.println(resultBinary != null ? resultBinary : "Product not found");

// Time Complexity Comparison

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

System.out.println("Linear Search: O(n) - scans each item until match.");

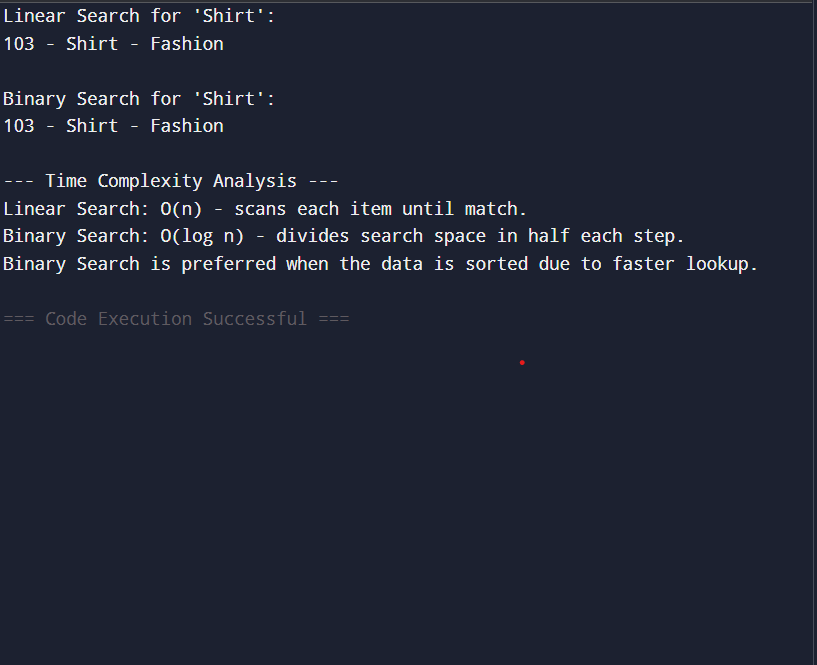
System.out.println("Binary Search: O(log n) - divides search space in half each step.");

System.out.println("Binary Search is preferred when the data is sorted due to faster lookup.");

}

}

**Output:**



**EXERCISE-7- Financial Forecasting:**

**Skill:Data structures and Algorithms**

**Program:**

import java.util.HashMap;

public class FinancialForecasting {

// Step 1: Recursion Explanation

/\*

\* Recursion is a programming technique where a method calls itself

\* to solve smaller instances of a problem.

\* It is useful for problems like factorial, Fibonacci, or tree traversal.

\*/

// Step 2 & 3: Recursive method to calculate future value

// Formula: FutureValue(n) = FutureValue(n-1) \* (1 + growthRate)

public static double predictFutureValue(int year, double baseValue, double growthRate) {

if (year == 0) {

return baseValue;

}

return predictFutureValue(year - 1, baseValue, growthRate) \* (1 + growthRate);

}

// Step 3 (Optimized): Memoized recursive method

static HashMap<Integer, Double> memo = new HashMap<>();

public static double predictFutureValueMemo(int year, double baseValue, double growthRate) {

if (year == 0) {

return baseValue;

}

if (memo.containsKey(year)) {

return memo.get(year);

}

double result = predictFutureValueMemo(year - 1, baseValue, growthRate) \* (1 + growthRate);

memo.put(year, result);

return result;

}

public static void main(String[] args) {

int forecastYears = 5;

double initialValue = 10000.0; // ₹10,000

double annualGrowthRate = 0.10; // 10%

System.out.println("Recursive Forecast (Without Optimization):");

for (int i = 0; i <= forecastYears; i++) {

double value = predictFutureValue(i, initialValue, annualGrowthRate);

System.out.printf("Year %d: ₹%.2f\n", i, value);

}

System.out.println("\nRecursive Forecast (With Memoization):");

for (int i = 0; i <= forecastYears; i++) {

double value = predictFutureValueMemo(i, initialValue, annualGrowthRate);

System.out.printf("Year %d: ₹%.2f\n", i, value);

}

// Step 4: Analysis

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

System.out.println("Naive Recursive Time Complexity: O(n) - due to one call per year.");

System.out.println("Memoized Recursive Time Complexity: O(n) - avoids recomputation.");

System.out.println("Memoization is essential for optimizing recursive algorithms and avoiding stack overflow.");

}

}

**Output:**

