**DIGITAL NURTURE 4.0 JavaFSE**

**WEEK 1**

**DESIGN PATTERNS**

**Exercise 1:**

**Implementing the Singleton Pattern:**

**Code :**

class Logger {

private static Logger instance;

private Logger() {

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

}

public static Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

public void log(String message) {

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

}

}

public class Main {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

Logger logger2 = Logger.getInstance();

logger1.log("hi this is my first message");

logger2.log("this message follows the first one");

if (logger1 == logger2) {

System.out.println("same instance");

} else {

System.out.println("dissimilar instance");

}

}

}

**Output :**



**Exercise 2 :**

**Implementing the Factory Method Pattern**

**Code:**

public class Main {

interface Document {

void open();

}

static class WordDocument implements Document {

public void open() {

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

}

}

static class PdfDocument implements Document {

public void open() {

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

}

}

static class ExcelDocument implements Document {

public void open() {

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

}

}

abstract static class DocumentFactory {

public abstract Document createDocument();

}

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();

}

}

public static void main(String[] args) {

DocumentFactory wordFactory = new WordDocumentFactory();

Document word = wordFactory.createDocument();

word.open();

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdf = pdfFactory.createDocument();

pdf.open();

DocumentFactory excelFactory = new ExcelDocumentFactory();

Document excel = excelFactory.createDocument();

excel.open();

}

}

**Output :**



**ALGORITHMS DATA STRUCTURES :**

**Exercise 2:**

**E-commerce Platform Search Function:**

**Big O :**

Big O is a way to describe how fast or slow an algorithm is when the amount of input gets bigger. It usually talks about the **worst-case scenario**, so we know how the algorithm behaves when things are the hardest.

* **Best Case:** O(1) → The fastest it can be
* **Worst Case:** O(log n) → Slows down as input grows, but not too badly.
* **Average Case:** O(n) → Gets slower as input grows

**Code:**

import java.util.Arrays;

import java.util.Comparator;

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 class Main {

public static void main(String[] args) {

Product[] products = {

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

new Product(102, "Shampoo", "Personal Care"),

new Product(103, "Notebook", "Stationery"),

new Product(104, "Headphones", "Electronics"),

new Product(105, "Pencil", "Stationery")

};

String searchTerm = "Notebook";

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

Product result1 = linearSearch(products, searchTerm);

System.out.println(result1 != null ? result1 : "Product not found.");

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

Product result2 = binarySearch(products, searchTerm);

System.out.println(result2 != null ? result2 : "Product not found.");

}

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[] products, String targetName) {

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

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

while (left <= right) {

int mid = (left + right) / 2;

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

if (comparison == 0) {

return products[mid];

} else if (comparison < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

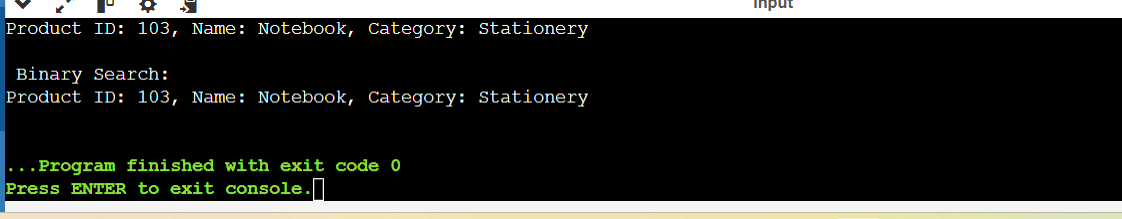
}

return null;

}

}

**Output :**



Comparison of time complexity and which is better:

**Linear Search:** O(n) – Checks each element one by one.

**Binary Search:** O(log n) – Repeatedly divides the sorted list in half.

Binary Search is more suitable for an e-commerce platform because it provides faster performance on large, sorted datasets, which is common in product catalogs.

**Exercise 7 :**

**Financial Forecasting :**

**Recursion:**

Recursion is a programming technique in which a function calls itself to solve a problem.  
It simplifies complex problems by breaking them down into smaller, similar sub-problems and continues this process until a base case is reached.

**Code :**

public class Main {

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

if (years == 0) {

return initialAmount;

}

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

}

public static void main(String[] args) {

double initialAmount = 10000.0;

double growthRate = 0.10;

int years = 5;

double futureValue = calculateFutureValue(initialAmount, growthRate, years);

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

}

}

**Time Complexity : O(n)**

**Output :**

