**Design principles & Patterns – Hands on**

**Exercise 1: Implementing the Singleton Pattern**

**Code: JAVA**

Class SingletonPatternExample{

public class Logger{

private static Logger instance;

private Logger() {

System.out.println("Singleton is Instantiated.");

}

public static Logger getInstance(){

if (instance == null) {

instance = new Logger();

}

return instance;

}

public void log(String message) {

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

}

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

Logger logger2 = Logger.getInstance();

Logger logger3 = Logger.getInstance();

logger1.log("Message from logger1");

logger2.log("Message from logger2");

logger3.log("Message from logger3");

if (logger1 == logger2 && logger2 == logger3) {

System.out.println("Test Passed: All references point to the same Logger instance.");

}

else {

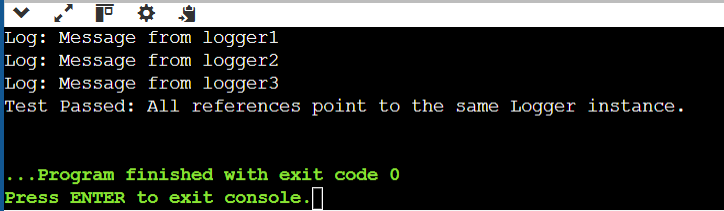
System.out.println("Test Failed: Different Logger instances were created.");

}

}

}

**Sample Output:**



**Exercise 2: Implementing the Factory Method Pattern**

**Code: JAVA**

public class FactoryMethodPatternExample {

public interface Document {

void open();

}

public static class WordDocument implements Document {

public void open() {

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

}

}

public static class PdfDocument implements Document {

public void open() {

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

}

}

public static class ExcelDocument implements Document {

public void open() {

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

}

}

// **Implementing the Factory Method**

public static abstract class DocumentFactory {

public abstract Document createDocument();

}

public static class WordDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

public static class PdfDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

public static class ExcelDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

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();

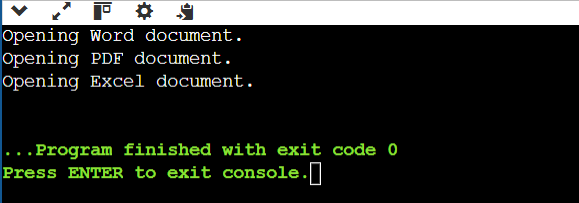
Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

**Sample Output:**



**Data structures and Algorithms – Hands on**

**Exercise 1: E-commerce Platform Search Function**

1. **Big O Notation :**

* A tool that helps us understand how fast or slow our code will run when the amount of data increases.
* Instead of telling us the exact time in seconds, it shows us how an algorithm's performance changes as we deal with more data like in an e-commerce platform.
* It helps evaluate algorithm efficiency and scalability.
* Example
* O(1) – Constant time (ideal)
* O(n) – linear time (grows with input size)
* O(log n) – logarithmic time (grows slowly with input)

1. **Best, average, and worst-case scenarios**

* Best Case Scenario:
* Performs **the minimum number of steps** to complete the task.
* Fastest algorithm.
* Example: Linear Search => Searching First element in list => immediately.
* Time Complexity => O(1).
* Average Case Scenario
* Performs based on Typical or random inputs.
* Gives realistic view of performance.
* Example: Linear Search => Searching some middle elements => Finds anywhere in list.
* Time Complexity => O(n).
* Worst Case Scenario
* Performs maximum number of steps.
* Slowest performance.
* Example : Linear Search => Last element or element not in the list.
* Time Complexity => O(n).

**Code : JAVA**

import java.util.\*;

public 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;

}

public String toString() {

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

}

// Linear Search

public static Product linearSearch(Product[] products, int targetId) {

for (Product product : products) {

if (product.productId == targetId) {

return product;

}

}

return null;

}

// Binary Search

public static Product binarySearch(Product[] products, int targetId) {

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

while (left <= right) {

int mid = (left + right) / 2;

if (products[mid].productId == targetId) {

return products[mid];

} else if (products[mid].productId < targetId) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter number of products: ");

int n = sc.nextInt();

sc.nextLine();

Product[] products = new Product[n];

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

System.out.println("Enter details for product " + (i + 1));

System.out.print("Product ID: ");

int id = sc.nextInt();

sc.nextLine();

System.out.print("Product Name: ");

String name = sc.nextLine();

System.out.print("Category: ");

String category = sc.nextLine();

products[i] = new Product(id, name, category);

}

System.out.print("Enter Product ID to search: ");

int targetId = sc.nextInt();

System.out.println("\n--- Linear Search Result ---");

Product linearResult = linearSearch(products, targetId);

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

//Sorting for binary search

Arrays.sort(products, Comparator.comparingInt(p -> p.productId));

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

Product binaryResult = binarySearch(products, targetId);

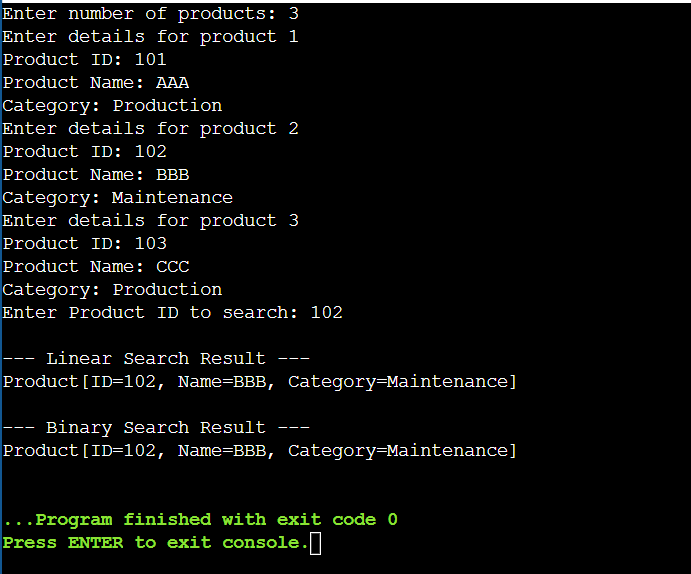
System.out.println(binaryResult != null ? binaryResult : "Product not found");

sc.close();

}

}

**Sample Output:**

****

**Time Complexity :**

* **Linear search** has a time complexity of **O(n)** in the worst and average cases.
* **Binary search** has a time complexity of **O(log n)** in the worst and average cases.
* In the **best case**, both algorithms take **O(1)** time if the target is found immediately.
* As the data size grows, **linear search becomes slower**, while **binary search remains efficient**.
* Binary search is faster, but only if the array is **sorted**, while linear search works regardless of order.

**Algorithm Used:**

* Binary search is more suitable for platforms that handle large and sorted datasets.
* Linear search would be too slow and inefficient for the large volume of products and users.
* The logarithmic time complexity of binary search ensures scalability as the number of products grows.
* Since product data is often stored in sorted databases or can be indexed, binary search efficiently handles frequent queries.
* Although binary search requires sorted data, e-commerce platforms typically maintain sorted or indexed data structures to optimize search.
* Therefore, binary search provides the best balance of speed and performance needed for a smooth user experience on e-commerce sites.

**Exercise 7: Financial Forecasting**

1. **Recursive Algorithms**

* Recursion is a technique where a function calls itself to solve smaller instances of a problem.
* Recursion helps simplify complex logic, especially in repetitive tasks like financial forecasting or tree traversal.
* A base case is essential to prevent infinite recursion and avoid memory overflows.
* While recursion makes code cleaner, it optimizes for large inputs to avoid performance issues.
* Recursive algorithms are often more readable and closer to the way humans think and it eliminates complex loops.

1. **Analysis:**

* The time complexity of the recursive algorithm is O(n)
* To optimize, we can convert the recursion into an iterative loop, which is more memory-efficient.
* Memoization can help store results and avoid repeated calculations.
* Helps to simplify complex Logic.

1. **Code: JAVA**

import java.util.Scanner;

public class FinancialForecasting {

// Recursive method to calculate future value

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

if (years == 0) {

return presentValue;

}

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

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter Present Value (₹): ");

double presentValue = scanner.nextDouble();

System.out.print("Enter Annual Growth Rate (in %): ");

double ratePercent = scanner.nextDouble();

double rate = ratePercent / 100;

System.out.print("Enter Number of Years: ");

int years = scanner.nextInt();

double futureValue = futureValueRecursive(presentValue, rate, years);

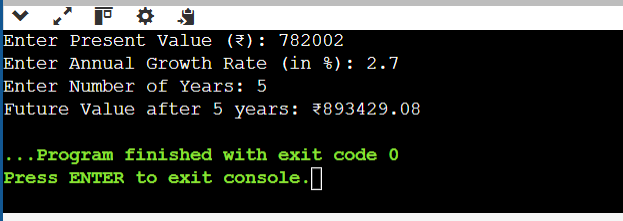
System.out.printf("Future Value after %d years: ₹%.2f", years, futureValue);

scanner.close();

}

}

**Sample Output:**

****