**Design & Patterns:**

**Exercise 1: Implementing the Singleton Pattern**

**Code:**

public 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: " + message);

}

}

public class Main {

public static void main(String[] args) {

Logger logger1 = Logger.*getInstance*();

logger1.log("First log");

Logger logger2 = Logger.*getInstance*();

logger2.log("Second log");

if (logger1 == logger2) {

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

} else {

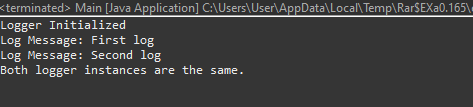
System.***out***.println("Different logger instances found!");

}

}

}

**Output:**



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

**Code:**

public interface Document {

void open();

}

public class WordDocument implements Document {

public void open() {

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

}

}

public class PdfDocument implements Document {

public void open() {

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

}

}

public class ExcelDocument implements Document {

public void open() {

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

}

}

public abstract class DocumentFactory {

public abstract Document createDocument();

}

public class PdfFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

public class ExcelFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

public class Main {

public static void main(String[] args) {

DocumentFactory wordFactory = new WordFactory();

Document wordDoc = wordFactory.createDocument();

wordDoc.open();

DocumentFactory pdfFactory = new PdfFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open();

DocumentFactory excelFactory = new ExcelFactory();

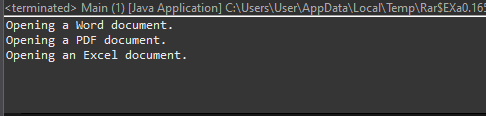
Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

**Output:**



**Data Structure & Algorithms:**

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

**Step1: Understand Asymptotic Notation:**

**Big O Notation:**

Big O notation describes the upper bound of an algorithm's running time or space usage in terms of input size n. It helps developers understand how algorithms scale and compare them independently of hardware or implementation details.

* **Example:** An algorithm with time complexity O(n) will take linear time with respect to input size, meaning if the input size doubles, the time taken also roughly doubles.

**Search Operation Cases:**

* **Best Case:**
  + **Linear Search:** The desired product is found at the first position --- **O(1)**
  + **Binary Search:** The desired product is found at the middle of the array on the first try --- **O(1)**
* **Average Case:**
  + **Linear Search:** The product is somewhere in the middle → **O(n/2)**, simplified to **O(n)**
  + **Binary Search:** Logarithmic splitting reduces time → **O(log n)**
* **Worst Case:**
  + **Linear Search:** The product is at the last position or not found → **O(n)**
  + **Binary Search:** The product is not found after all divisions → **O(log n)**

**Step2,3:**

**Code:**

package Ecommerce;

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 productId + " - " + productName + " (" + category + ")";

}

}

package Ecommerce;

import java.util.Arrays;

import java.util.Comparator;

public class SearchFunction {

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

for (int i = 0; i < products.length; i++) {

if (products[i].productName.equalsIgnoreCase(name)) {

return i;

}

}

return -1;

}

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

int low = 0, high = products.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

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

if (cmp == 0) return mid;

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

else high = mid - 1;

}

return -1;

}

public static void main(String[] args) {

Product[] products = {

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

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

new Product(103, "Shoes", "Footwear"),

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

new Product(105, "Book", "Education")

};

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

int index1 = *linearSearch*(products, "Camera");

if (index1 != -1) System.***out***.println("Found: " + products[index1]);

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

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

System.***out***.println("\nBinary Search Result:");

int index2 = *binarySearch*(products, "Camera");

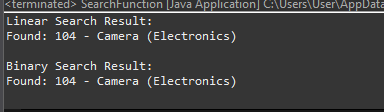
if (index2 != -1) System.***out***.println("Found: " + products[index2]);

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

}

}

**Output:**



**Step 4: Analysis**

**Time Complexity Comparison:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** |
| Linear Search | O(1) | O(n) | O(n) |
| Binary Search | O(1) | O(log n) | O(log n) |

**Suitability for Ecommerce Platform:**

* **Linear Search** is simple but inefficient for large product catalogs. Its time complexity grows linearly, which becomes slow as the number of products increases.
* **Binary Search** is much faster for large, sorted datasets, as it reduces the search.

**Exercise 7: Financial Forecasting**

**Solution:**

**Step 1: Understand Recursive Algorithms**

#### ****What is Recursion?****

Recursion is a programming technique where a function **calls itself** to solve smaller instances of the same problem until it reaches a **base case**.

#### ****How it Simplifies Problems:****

* Recursion is ideal for problems that can be broken down into similar subproblems.
* Instead of using loops or manual stack management, recursion naturally models problems like:
  + Computing future value based on repeated interest application.
  + Traversing trees or graphs.
  + Performing divide-and-conquer operations like in Merge Sort or Binary Search.

**Code:**

package FinancialForecasting;

import java.util.Scanner;

public class ForecastCalculator {

public static double futureValue(double initialValue, double rate, int years) {

if (years == 0) {

return initialValue;

}

return *futureValue*(initialValue, rate, years - 1) \* (1 + rate);

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.***in***);

System.***out***.print("Enter the initial investment amount (₹): ");

double initialValue = scanner.nextDouble();

System.***out***.print("Enter the annual growth rate (in %): ");

double ratePercent = scanner.nextDouble();

double rate = ratePercent / 100;

System.***out***.print("Enter the number of years to forecast: ");

int years = scanner.nextInt();

double result = *futureValue*(initialValue, rate, years);

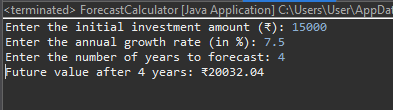
System.***out***.printf("Future value after %d years: ₹%.2f\n", years, result);

scanner.close();

}

}

**Output:**



### ****Step 4: Analysis****

#### ****Time Complexity of Recursive Algorithm:****

#### Recursive calls: O(n) (because 1 call for each year).

#### ****Optimization to Avoid Excessive Computation:****

1. **Memoization (for complex recursive relations):** Store already-computed values in a cache (array/map) to avoid repeated calculations.
2. **Convert to Iterative:** For simple calculations like future value, use a **loop-based** solution for
3. **Tail Recursion (in languages that support tail call optimization):** Refactor recursive functions to tail-recursive form, though Java does not optimize tail recursion.