Design Patterns and Principles

# 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(); logger1.log("First log message");

Logger logger2 = Logger.getInstance(); logger2.log("Second log message");

if (logger1 == logger2) {

System.out.println("Both logger1 and logger2 refer to the same instance.");

} else {

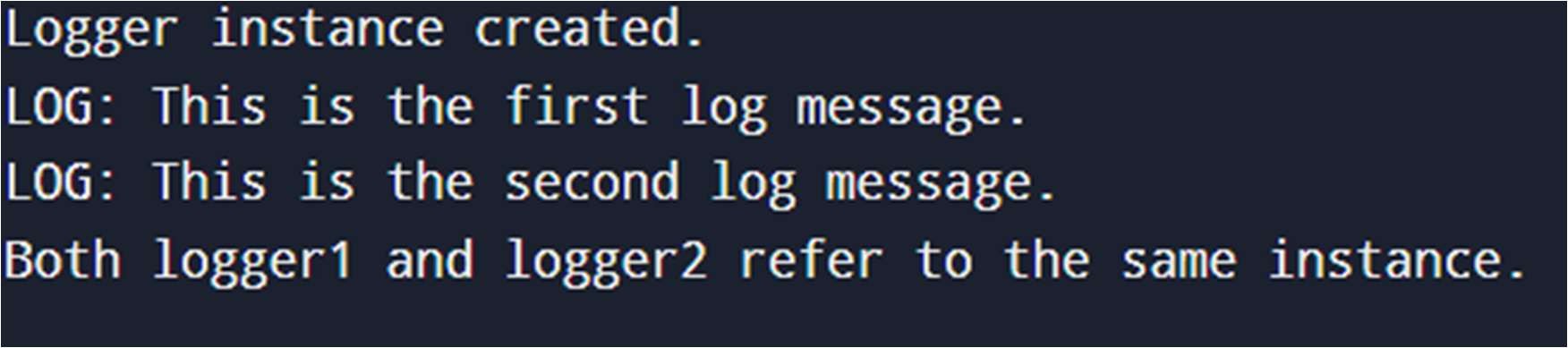
System.out.println("Different instances found! Singleton pattern failed.");

}

}

}

**Output**

****

# Exercise 2: Implementing the Factory Method Pattern Code

// Interface for Document

interface Document { void open();

}

// Concrete Document Types

class WordDocument implements Document { public void open() {

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

}

}

class PdfDocument implements Document { public void open() {

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

}

}

class ExcelDocument implements Document { public void open() {

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

}

}

// Abstract Factory

abstract class DocumentFactory { abstract Document createDocument();

}

// Concrete Factories

class WordDocumentFactory extends DocumentFactory { public Document createDocument() {

return new WordDocument();

}

}

class PdfDocumentFactory extends DocumentFactory { public Document createDocument() {

return new PdfDocument();

}

}

class ExcelDocumentFactory extends DocumentFactory { public Document createDocument() {

return new ExcelDocument();

}

}

// Test Class public class Main {

public static void main(String[] args) {

DocumentFactory wordFactory = new WordDocumentFactory(); Document word = wordFactory.createDocument(); word.open();

DocumentFactory pdfFactory = new PdfDocumentFactory(); Document pdf = pdfFactory.createDocument();

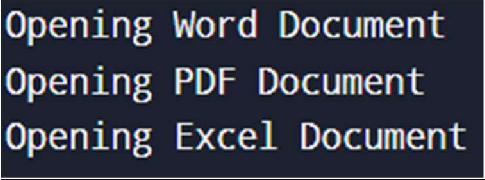
pdf.open();

DocumentFactory excelFactory = new ExcelDocumentFactory(); Document excel = excelFactory.createDocument(); excel.open();

}

}

**Output**

****

**Data Structure G Algorithms**

# Exercise 2: E-commerce Platform Search Function

Let us Proceed step by step.

Step 1 - **Understanding Asymptotic Notation**

**Big O notation** is used to describe the efficiency of algorithms based on the input size. It helps compare different algorithms without depending on hardware or actual running time.

* + **Best Case:** The scenario where the search item is found in the first attempt.
  + **Average Case:** The item is found somewhere in the middle.
  + **Worst Case:** The item is not found at all or is found at the very end.

# Linear Search

* **Best Case**:

The element is found at the first position.

→ Time Complexity: **O(1)**

# Average Case:

The element is located somewhere in the middle of the array.

→ Time Complexity: **O(n)**

# Worst Case:

The element is not found or is located at the last position.

→ Time Complexity: **O(n)**

**Binary Search *(on sorted arrays)***

# Best Case:

The element is found at the middle index.

→ Time Complexity: **O(1)**

# Average Case:

The element is found after repeatedly halving the search space.

→ Time Complexity: **O(log n)**

# Worst Case:

The element is not present, and all levels of the search tree are traversed.

→ Time Complexity: **O(log n)**

|  |  |  |  |
| --- | --- | --- | --- |
| **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) |

Step 2 – **Setup**

We define a Product class with attributes that are commonly used in search functionalities.

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 + ")";

}

}

Step 3 – **Implementation**

We use:

* **Linear Search** on an unsorted array
* **Binary Search** on a sorted array (sorted by product name) import java.util.Arrays;

import java.util.Comparator;

public class Main {

public static int linearSearch(Product[] products, String targetName) { for (int i = 0; i < products.length; i++) {

if (products[i].productName.equalsIgnoreCase(targetName)) { return i;

}

}

return -1;

}

public static int binarySearch(Product[] products, String targetName) { int left = 0, right = products.length - 1;

while (left <= right) {

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

products[mid].productName.compareToIgnoreCase(targetName);

if (cmp == 0) return mid;

else if (cmp < 0) left = mid + 1; else right = mid - 1;

}

return -1;

}

public static void main(String[] args) { Product[] products = {

new Product(101, "Laptop", "Electronics"), new Product(102, "Shoes", "Fashion"),

new Product(103, "Book", "Education"), new Product(104, "Mobile", "Electronics"), new Product(105, "Watch", "Accessories")

};

String searchTerm = "Book";

int linearIndex = linearSearch(products, searchTerm); if (linearIndex != -1)

System.out.println("Linear Search: Found → " + products[linearIndex]);

else

System.out.println("Linear Search: Product not found");

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

int binaryIndex = binarySearch(products, searchTerm); if (binaryIndex != -1)

System.out.println("Binary Search: Found → " + products[binaryIndex]);

else

System.out.println("Binary Search: Product not found");

}

}

Step 4 – **Analysis**

# TIME COMPLEXITY ANALYSIS

|  |  |  |  |
| --- | --- | --- | --- |
| **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) |

* **Linear Search**:
  + In the best case (when the first element matches), it takes constant time O(1).
  + On average, it checks half the list → O(n/2) = O(n).
  + In the worst case, it checks every element → O(n).
  + No sorting required, but not efficient for large datasets.

# Binary Search:

* + Requires the array to be **sorted** first.
  + In all scenarios except the best case, it reduces the search space by half in each step → O(log n).
  + Much faster than linear search for large datasets.

# Which Algorithm is More Suitable and Why?

For an **e-commerce platform**, the number of products can range from thousands to millions. In such a scenario:

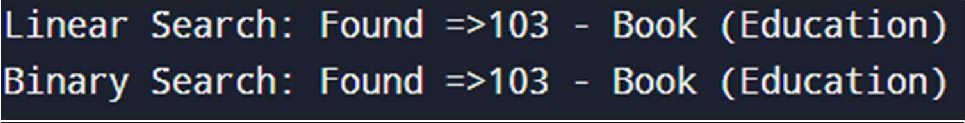
* **Linear search becomes inefficient** because its time complexity increases linearly with the number of products. A single search may have to scan every product in the worst case.
* **Binary search**, on the other hand, is much more efficient due to its logarithmic time complexity. Although it requires the data to be sorted, this can be handled in the background or done once during data upload.

# Therefore, binary search is more suitable for an e-commerce platform where:

* Speed and performance are critical,
* Product data can be indexed or pre-sorted (which is a standard practice).

In conclusion, **binary search offers a better long-term solution** for performance-driven applications like e-commerce, especially when the volume of data is large and search operations are frequent.

**OUTPUT**

****

# Exercise 7: Financial Forecasting

Let us proceed step by step.

1. Step 1 - **Understanding Recursive Algorithms**

Recursion is a problem-solving technique where a function calls itself to solve smaller parts of the same problem. Each recursive call should bring the solution closer to a base case, which stops the

recursion. This approach helps in reducing complex problems into simpler, manageable sub-problems.

In the context of **financial forecasting**, recursion can be used to simulate the process of compounding values over a number of periods (e.g., years). Rather than using loops, recursion allows us to express the logic naturally, especially when modeling mathematical formulas such as:

# Future Value = Present Value × (1 + Rate)^n

Instead of computing this using loops or built-in functions, recursion can represent each year's growth as a separate step, making the logic easy to trace and understand — especially useful in academic or learning scenarios.

Step 2 – **Setup**

To begin implementing the solution, we will:

* Define a method named calculateFutureValue
* Accept the following inputs:
  1. presentValue: the initial amount of money/investment
  2. growthRate: the percentage by which the value increases annually
  3. years: the number of years over which we are predicting the future value

This recursive method will simulate how the value grows year by year. The growth rate is applied recursively to the result of the previous year until the base case (0 years remaining) is reached.

Step 3 – **Implementation** import java.util.Scanner; public class FinancialForecast {

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

if (years == 0) {

return presentValue;

1);

}

}

double updatedValue = presentValue \* (1 + growthRate / 100); return calculateFutureValue(updatedValue, growthRate, years -

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter the initial investment amount: "); double initialInvestment = scanner.nextDouble();

System.out.print("Enter the annual growth rate (%): "); double annualGrowthRate = scanner.nextDouble();

System.out.print("Enter the number of years for forecasting: "); int forecastYears = scanner.nextInt();

double futureValue = calculateFutureValue(initialInvestment, annualGrowthRate, forecastYears);

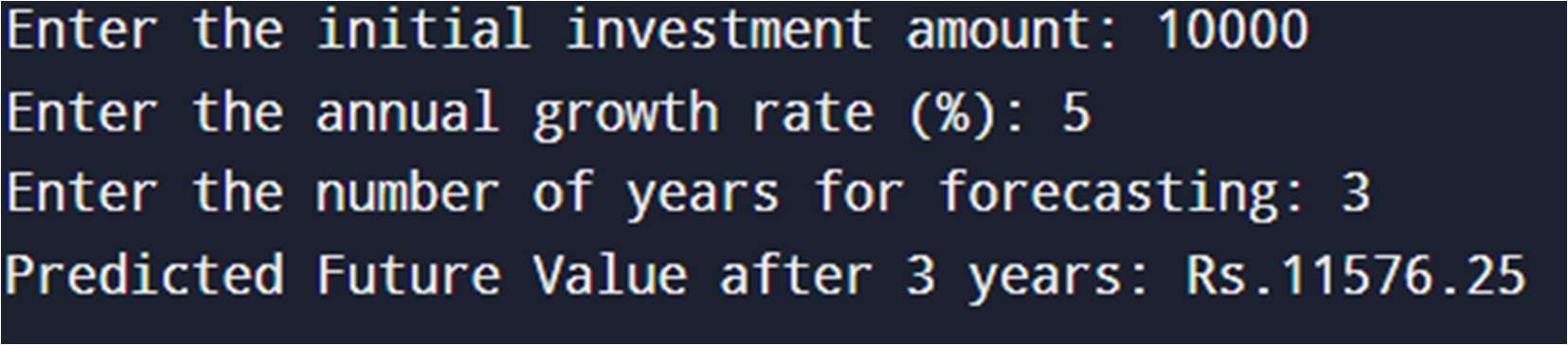
System.out.printf("Predicted Future Value after %d years: Rs.%.2f\n", forecastYears, futureValue);

scanner.close();

}

}

**Output**

****

Step 4: **Analysis**

# Time Complexity of the Recursive Algorithm

The recursive method calculateFutureValue(presentValue, growthRate, years) makes **one recursive call for each year**. In each call, it performs constant-time arithmetic operations.

**Number of Recursive Calls:** Equal to the number of years (n). **Operations per Call:** One multiplication and one subtraction — both are O(1).

# Therefore, the time complexity is:

**O(n)** — where n is the number of years.

This means the execution time grows linearly with the number of forecast years.

# Optimizing the Recursive Solution

While recursion simplifies the logic, it introduces performance issues for larger inputs due to:

**Stack Overflow Risk:** Each recursive call consumes stack memory. For large values of n, the stack depth may exceed the system limit.

**Call Overhead:** Each call involves memory allocation and function invocation, which can be slower than iteration.

# Optimization Strategy: Use an Iterative Approach

You can **optimize** the solution by replacing recursion with a simple for loop. This eliminates function call overhead and avoids stack memory usage.

# Optimized Iterative Version (Same Time Complexity, Better Space)

public static double calculateFutureValueIterative(double

presentValue, double growthRate, int years) { for (int i = 0; i < years; i++) {

presentValue \*= (1 + growthRate / 100);

}

return presentValue;

}

# Time Complexity: O(n)

**Space Complexity:** O(1) (constant memory)