**Design principles & Patterns:**

**Exercise 1: Implementing the Singleton Pattern**

**CODE:**

**Looger.java**

package com.singleton.example;

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);

}

}

**LoggerTest.java**

package com.singleton.example;

public class LoggerTest {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

logger1.log("Starting the application");

Logger logger2 = Logger.getInstance();

logger2.log("Application running");

if (logger1 == logger2) {

System.out.println("Both loggers are the same instance ");

} else {

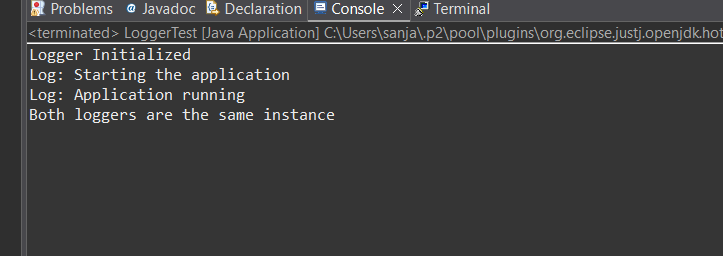
System.out.println("Different instances ");

}

}

}

**OUTPUT:**



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

**CODE:**

**Document.java**

package com.factorymethod.example;

public interface Document {

void open();

}

**WordDocument.java**

package com.factorymethod.example;

public class WordDocument implements Document {

public void open() {

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

}

}

**PdfDocument.java**

package com.factorymethod.example;

public class PdfDocument implements Document {

public void open() {

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

}

}

**ExcelDocument.java**

package com.factorymethod.example;

public class ExcelDocument implements Document {

public void open() {

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

}

}

**DocumentFactory.java (Abstract Factory)**

package com.factorymethod.example;

public abstract class DocumentFactory {

public abstract Document createDocument();

}

**WordDocumentFactory.java**

package com.factorymethod.example;

public class WordDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

**PdfDocumentFactory.java**

package com.factorymethod.example;

public class PdfDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

**ExcelDocumentFactory.java**

package com.factorymethod.example;

public class ExcelDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

**FactoryMethodTest.java**

package com.factorymethod.example;

public class FactoryMethodTest {

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();

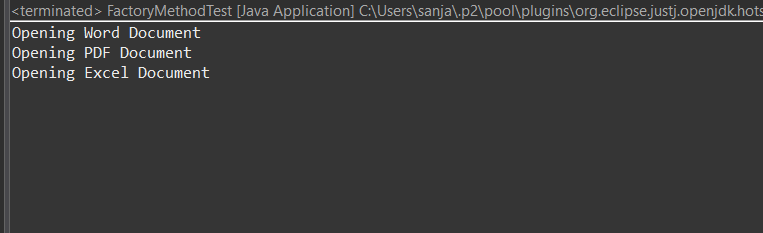
Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

**OUTPUT:**



**Data structures and Algorithms:**

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

**Understand Asymptotic Notation**

**Explain Big O notation and how it helps in analyzing algorithms.**

**Big O Notation describes the time or space complexity of an algorithm in terms of input size n. It shows how fast or slow an algorithm grows as data increases**

**It helps developers compare algorithms, predict performance, and choose the most efficient approach for large datasets.**

**Describe the best, average, and worst-case scenarios for search operations:**

**Linear search : best case O(1) , average case O(n) , worst case O(n)**

**Binary search : best case O(1),average case O(log n),worst case O(log n)**

**CODE:**

P**roduct.java**

package com.ecommerce.search;

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 "ID: " + productId + ", Name: " + productName + ", Category: " + category;

}

}

**SearchDemo.java**

import java.util.Arrays;

import java.util.Comparator;

public class SearchDemo {

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

for (Product p : products) {

if (p.productName.equalsIgnoreCase(name)) {

return p;

}

}

return null;

}

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

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

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 products[mid];

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

else high = mid - 1;

}

return null;

}

public static void main(String[] args) {

Product[] products = {

new Product(1, "Laptop", "Electronics"),

new Product(2, "Shoes", "Footwear"),

new Product(3, "Mobile", "Electronics"),

new Product(4, "Book", "Education"),

new Product(5, "T-shirt", "Apparel")

};

Product result1 = linearSearch(products, "Mobile");

System.out.println("Linear Search Result: " + (result1 != null ? result1 : "Not Found"));

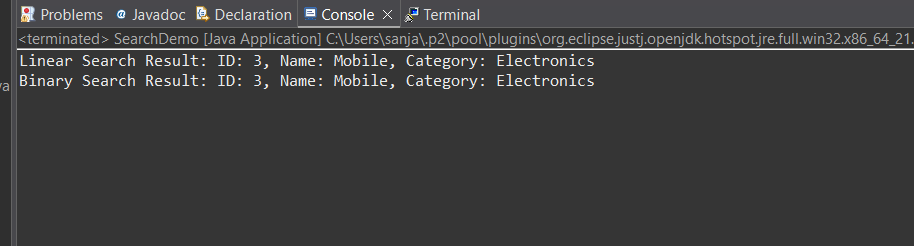
Product result2 = binarySearch(products, "Mobile");

System.out.println("Binary Search Result: " + (result2 != null ? result2 : "Not Found"));

}

}

OUTPUT:



4 **:Analysis**

Linear Search TC: O(n) WHY: Small lists, unsorted data

Binary Search TC:nO(log n) WHY: Large sorted datasets

**Conclusion: Which to Choose?**

**Use Linear Search**: when data is small or unsorted

**Use Binary Search**: when performance is critical and data is already sorted

For large e-commerce platforms: Binary seaech

**Exercise 7: Financial Forecasting**

**What is Recursion?**

**Recursion** is a method where a function **calls itself** to solve smaller instances of the same problem.

It simplifies problems like:

* Factorial
* Fibonacci
* Tree traversals
* Compound interest
* forecasting

CODE:

package com.financial.forecasting;

public class FinancialForecast {

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

if (years == 0) return presentValue;

return forecast(presentValue \* (1 + growthRate), growthRate, years - 1);

}

public static void main(String[] args) {

double presentValue = 10000;

double growthRate = 0.05;

int years = 5;

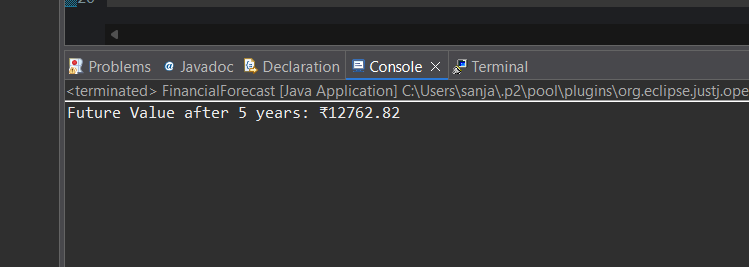
double futureValue = forecast(presentValue, growthRate, years);

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

}

}

OUTPUT:



T**ime Complexity = O(n)**

If we used **Math.pow()**, we can directly calculate:

public static double forecastOptimized(double pv, double rate, int years) {

return pv \* Math.pow(1 + rate, years);

}

This is more efficient → **O(1)** time complexity.