**Week 1**

**1.Singleton Pattern:**

**Code:**

package SingletonPatternExample;

public class Logger {

private static Logger *instance*;

private Logger() {

System.*out*.println("Logger instance is 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("Application started.");

logger2.log("This is another log message.");

if (logger1 == logger2) {

System.*out*.println("Only one instance of Logger is used.");

} else {

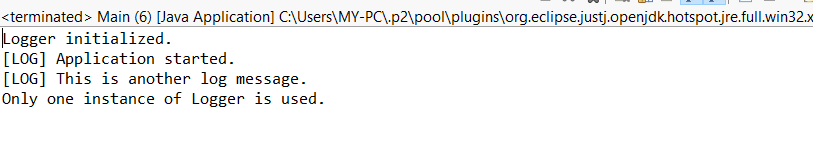
System.*out*.println("Different instances found (Error).");

}

}

}

**Output:**



**2.Factory Method Pattern:**

**Code:**

package com.FactorymethodExample;

interface Document {

void open();

}

class WordDocument implements Document {

public void open() {

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

}

}

class PdfDocument implements Document {

public void open() {

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

}

}

class ExcelDocument implements Document {

public void open() {

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

}

}

abstract class DocumentFactory {

public abstract Document createDocument();

}

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();

}

}

public class FactoryMethodExample {

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();

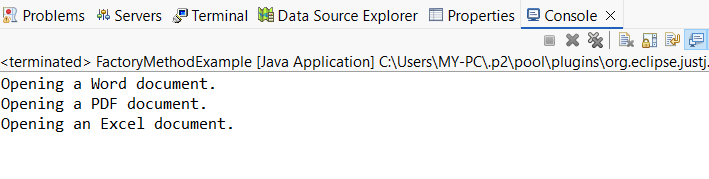
Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

**OUTPUT:**



**3.Ecommerce Platform Search Function**

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

}

}

public class ECommerceSearchExample {

// Linear Search by product name

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

for (Product product : products) {

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

return product;

}

}

return null;

}

// Binary Search by product name (array must be sorted by name)

public static Product 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 products[mid];

else if (cmp < 0)

low = mid + 1;

else

high = mid - 1;

}

return null;

}

public static void main(String[] args) {

// Sample product list

Product[] products = {

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

new Product(102, "Smartphone", "Electronics"),

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

new Product(104, "Watch", "Accessories"),

new Product(105, "Tablet", "Electronics")

};

// Linear Search Test

System.out.println(" Linear Search for 'Watch':");

Product linearResult = linearSearch(products, "Watch");

System.out.println(linearResult != null ? "Found: " + linearResult : " Not found");

// Sort array for Binary Search

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

// Binary Search Test

System.out.println("\nBinary Search for 'Watch':");

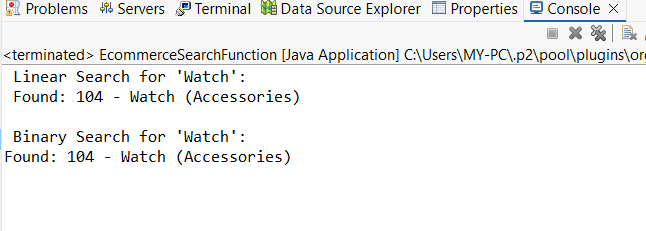
Product binaryResult = binarySearch(products, "Watch");

System.out.println(binaryResult != null ? " Found: " + binaryResult : " Not found");

}

}

**Output:**

****

3.1.Compare the time complexity of linear and binary search algorithms.

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

3.2.Discuss which algorithm is more suitable for your platform and why**.**

Binary Search is more suitable for an e-commerce platform,especially when:

* large datasets (thousands/millions of products).
* we can maintain sorted data (by product name, ID, etc.).
* we need fast and efficient search performance for a better user experience.

**4.FUTURE FORECASTING**

**CODE:**

package com**.**FinancialForecasting;

public class FinancialForecasting {

// Recursive method to calculate future value

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

if (years == 0) {

return presentValue;

} else {

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

}

}

// Optimized version using Iteration (to avoid recursion overhead)

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

double futureValue = presentValue;

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

futureValue \*= (1 + growthRate);

}

return futureValue;

}

public static void main(String[] args) {

double presentValue = 10000.0;

double growthRate = 0.08; // 8% growth

int years = 5;

double recursiveResult = *forecastValueRecursive*(presentValue, growthRate, years);

double iterativeResult = *forecastValueIterative*(presentValue, growthRate, years);

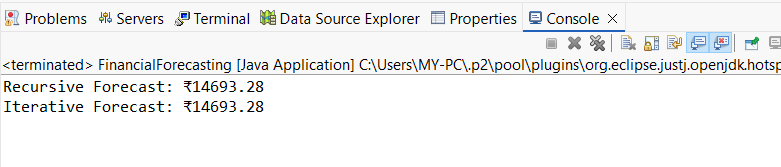
System.*out*.printf("Recursive Forecast: ₹%.2f%n", recursiveResult);

System.*out*.printf("Iterative Forecast: ₹%.2f%n", iterativeResult);

}

}

**OUTPUT:**

****4.1.Discuss the time complexity of your recursive algorithm.

Time Complexity

* Recursive Approach:
  + Time: O(n)
  + Space: O(n) (due to call stack for each recursive call)
* Iterative Approach:
  + Time: O(n)
  + Space: O(1) (no extra call stack)

4.2.Explain how to optimize the recursive solution to avoid excessive computation.

Recursive solutions can be inefficient if they repeatedly calculate the same values or go too deep (causing stack overflow). In financial forecasting or similar problems, you can optimize recursion in two main ways:

1.**Using Memoization**:Memoization stores already-computed values in a cache (like a map or array) so they aren't recomputed again.

2. **Convert to Iteration :**convert recursive logic to a loop-based solution which is faster and more memory efficient.