Design Patterns and Principles

----------------------------------------------------------------------------------------------

Exercise 1: Implementing the Singleton Pattern

------------------------------------------------------------------------------------

package singleton;

public class Logger {

private static Logger *instance*;

private Logger() {

}

public static Logger getInstance() {

if (*instance* == null) {

*instance* = new Logger();

}

return *instance*;

}

public void log(String message) {

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

}

}

package singleton;

public class Main {

public static void main(String[] args) {

Logger logger1 = Logger.*getInstance*();

Logger logger2 = Logger.*getInstance*();

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

logger2.log("Second log message.");

if (logger1 == logger2) {

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

} else {

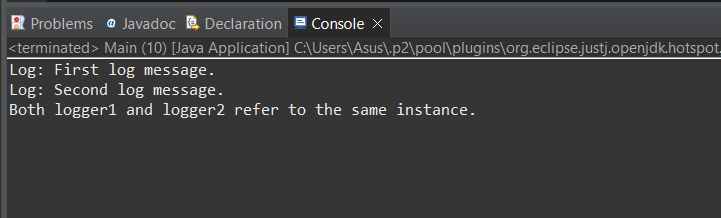
System.***out***.println("Different instances exist!");

}

}

}

OUTPUT:



------------------------------------------------------------------------------------------------------------------

Exercise 2: Implementing the Factory Method Pattern

------------------------------------------------------------------------------------------------------------------

package factorymethod;

public abstract class Document {

public abstract void open();

}

package factorymethod;

public abstract class DocumentFactory {

public abstract Document createDocument();

}

package factorymethod;

public class ExcelDocument extends Document {

*@Override*

public void open() {

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

}

}

package factorymethod;

public class ExcelDocumentFactory extends DocumentFactory {

*@Override*

public Document createDocument() {

return new ExcelDocument();

}

}

package factorymethod;

public class PdfDocument extends Document {

*@Override*

public void open() {

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

}

}

package factorymethod;

public class PdfDocumentFactory extends DocumentFactory {

*@Override*

public Document createDocument() {

return new PdfDocument();

}

}

package factorymethod;

public class WordDocument extends Document {

*@Override*

public void open() {

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

}

}

package factorymethod;

public class WordDocumentFactory extends DocumentFactory {

*@Override*

public Document createDocument() {

return new WordDocument();

}

}

package factorymethod;

public class Main {

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();

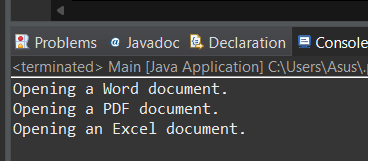
Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

OUTPUT:



Algorithms\_Data Structures

Exercise 2: E-commerce Platform Search Function

-----------------------------------------------------------------------------------------------

package E\_commerce;

public class Product {

private int productId;

private String productName;

private String category;

public Product(int productId, String productName, String category) {

this.productId = productId;

this.productName = productName;

this.category = category;

}

public int getProductId() {

return productId;

}

public String getProductName() {

return productName;

}

public String getCategory() {

return category;

}

*@Override*

public String toString() {

return productName + " (" + category + ") - ID: " + productId;

}

}

package E\_commerce;

import java.util.Arrays;

import java.util.Comparator;

public class SearchEngine {

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

for (Product product : products) {

if (product.getProductName().equalsIgnoreCase(targetName)) {

return product;

}

}

return null;

}

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

Arrays.sort(products, Comparator.comparing(Product::getProductName));

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

while (left <= right) {

int mid = (left + right) / 2;

int cmp = products[mid].getProductName().compareToIgnoreCase(targetName);

if (cmp == 0) {

return products[mid];

} else if (cmp < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

}

OUTPUT:

A screenshot of a computer

AI-generated content may be incorrect.

Exercise 7: Financial Forecasting

package Financial;

public class FinancialForecaster {

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

if (years == 0) {

return presentValue;

}

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

}

public static double futureValueMemo(double presentValue, double growthRate, int years, double[] memo) {

if (years == 0) return presentValue;

if (memo[years] != 0) return memo[years];

memo[years] = (1 + growthRate) \* *futureValueMemo*(presentValue, growthRate, years - 1, memo);

return memo[years];

}

}

package Financial;

public class Main {

public static void main(String[] args) {

double presentValue = 1000.0;

double growthRate = 0.08;

int years = 10;

System.***out***.println("📈 Recursive Forecasting");

double result = FinancialForecaster.*futureValue*(presentValue, growthRate, years);

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

System.***out***.println("\n⚡ Optimized with Memoization");

double[] memo = new double[years + 1];

double resultMemo = FinancialForecaster.*futureValueMemo*(presentValue, growthRate, years, memo);

System.***out***.printf("Future Value after %d years (memoized): $%.2f\n", years, resultMemo);

}

}

OUTPUT:

A screenshot of a computer program

AI-generated content may be incorrect.