**DEEP NURTURE 4.0----WEEK-1**

**Design Patterns and principles:**

**Excercise1:**

Logger.java:

package singleton;

public class Logger {

private static Logger *instance*;

private Logger() {

System.***out***.println("Logger instance created.");

}

public static Logger getInstance() {

if (*instance* == null) {

*instance* = new Logger(); // lazy initialization

}

return *instance*;

}

public void log(String message) {

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

}

}

Main.java:

package singleton;

public class Main {

public static void main(String[] args) {

Logger logger1 = Logger.*getInstance*();

logger1.log("This is the first log message.");

Logger logger2 = Logger.*getInstance*();

logger2.log("This is the second log message.");

if (logger1 == logger2) {

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

} else {

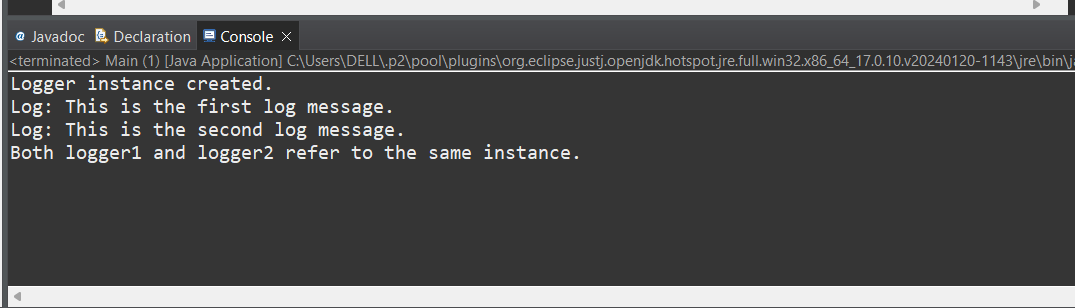
System.***out***.println("logger1 and logger2 are different instances.");

}

}

}

**OUTPUT:**

****

**Excercise2:**

Document.java:

package factory;

public interface Document {

void open();

}

DocumentFactory.java:

package factory;

public abstract class DocumentFactory {

public abstract Document createDocument();

}

ExcelDocument.java:

package factory;

public class ExcelDocument implements Document {

*@Override*

public void open() {

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

}

}

ExecelDocumentFactory.java:

package factory;

public class ExcelDocumentFactory extends DocumentFactory {

*@Override*

public Document createDocument() {

return new ExcelDocument();

}

}

PdfDocument.java:

package factory;

public class PdfDocument implements Document {

*@Override*

public void open() {

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

}

}

PdfDocumentFactory.java:

package factory;

public class PdfDocumentFactory extends DocumentFactory {

*@Override*

public Document createDocument() {

return new PdfDocument();

}

}

WordDocument.java:

package factory;

public class WordDocument implements Document {

*@Override*

public void open() {

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

}

}

WordDocumentFactory.java:

package factory;

public class WordDocumentFactory extends DocumentFactory {

*@Override*

public Document createDocument() {

return new WordDocument();

}

}

Main.java:

package factory;

public class Main {

public static void main(String[] args) {

DocumentFactory wordFactory = new WordDocumentFactory();

DocumentFactory pdfFactory = new PdfDocumentFactory();

DocumentFactory excelFactory = new ExcelDocumentFactory();

Document wordDoc = wordFactory.createDocument();

Document pdfDoc = pdfFactory.createDocument();

Document excelDoc = excelFactory.createDocument();

wordDoc.open();

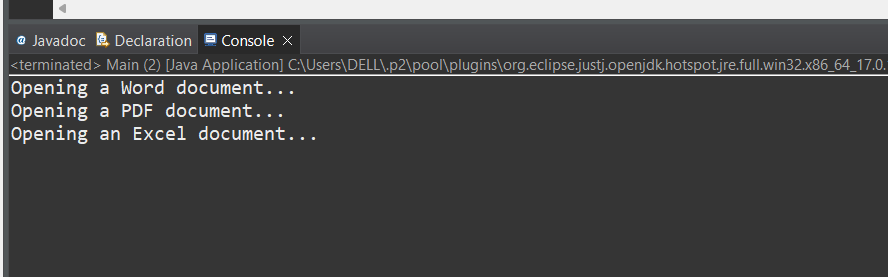
pdfDoc.open();

excelDoc.open();

}

}

**OUTPUT:**

****

**Algorithm Data Structures:**

**Excercise1:**

**Big O Notation:**

* Big O notation describes the worst-case time complexity of an algorithm, showing how the runtime scales with the size of the input.

| **Complexity** | **Description** |
| --- | --- |
| O(1) **--------------------->** | Constant time |
| O(log n) **--------------------->** | Logarithmic time |
| O(n) **--------------------->** | Linear time |
| O(n log n) **--------------------->** | Log-linear time |
| O(n²) **--------------------->** | Quadratic time |

**Search Scenarios:**

* **Best Case:** Found at first attempt (start of list).
* **Average Case:** Found after scanning half the list**.**
* **Worst Case:** Element not found or found at the end**.**

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

Product.java:

package ecommerce;

public class Product {

int productId;

String productName;

String category;

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

this.productId = productId;

this.productName = productName;

this.category = category;

}

*@Override*

public String toString() {

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

}

}

ECommercesearch.java:

package ecommerce;

import java.util.Arrays;

import java.util.Comparator;

public class ECommercesearch {

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

for (Product product : products) {

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

return product;

}

}

return null;

}

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

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

while (low <= high) {

int mid = (low + high) / 2;

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

if (comparison == 0)

return products[mid];

else if (comparison < 0)

low = mid + 1;

else

high = mid - 1;

}

return null;

}

public static void main(String[] args) {

Product[] products = {

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

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

new Product(103, "Shirt", "Clothing"),

new Product(104, "Shoes", "Footwear"),

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

};

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

Product result1 = *linearSearch*(products, "Shoes");

System.***out***.println(result1 != null ? result1 : "Product not found");

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

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

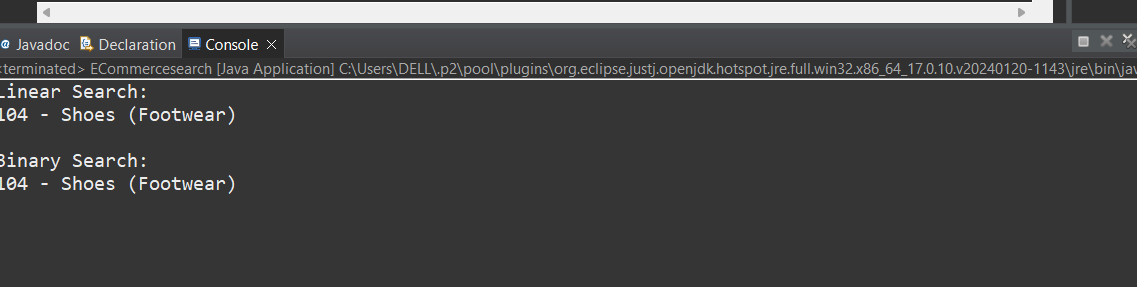
Product result2 = *binarySearch*(products, "Shoes");

System.***out***.println(result2 != null ? result2 : "Product not found");

}

}

**OUTPUT:**

****

**Excercise2:**

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

**Example:**  
To compute compound future value:

* FutureValue(n)=FutureValue(n−1)×(1+r)
* Base case:FutureValue(0)=initialValue

Benefits:

* Elegant and simplifies code.
* Especially useful in problems that follow a repeated pattern (e.g., financial growth over years).

FinancialForecast.java:

package ecommerce;

public class FinancialForecast {

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

if (years == 0) {

return initialValue; // base case

}

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

}

public static double futureValueMemo(double initialValue, double rate, int years, Double[] memo) {

if (years == 0) return initialValue;

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

memo[years] = *futureValueMemo*(initialValue, rate, years - 1, memo) \* (1 + rate);

return memo[years];

}

public static void main(String[] args) {

double initialInvestment = 10000;

double growthRate = 0.08;

int numYears = 10;

System.***out***.println("Recursive Future Value:");

double futureVal = *futureValue*(initialInvestment, growthRate, numYears);

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

Double[] memo = new Double[numYears + 1];

System.***out***.println("\nOptimized (Memoized) Future Value:");

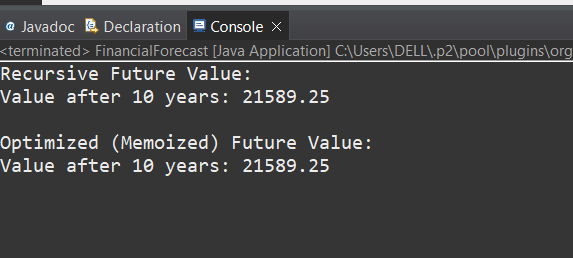
double optimizedVal = *futureValueMemo*(initialInvestment, growthRate, numYears, memo);

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

}

}

**OUTPUT:**

****