**DESIGN PATTERNS AND PRINCIPLES**

**EXERCISE-1: IMPLEMENTING THE SINGLETON PATTERN**

**Logger.java**

public class Logger {

private static Logger *instance*;

public Logger() {

// TODO Auto-generated constructor stub

System.*out*.println("Logger initialized");

}

public static Logger getInstance() {

if(*instance* == null) {

*instance* = new Logger();

}

return *instance*;

}

public void testLog(int number1, int number2) {

System.*out*.println("The two numbers are: "+ number1 + " " + number2);

}

}

**Test.java**

public class Test {

public static void main(String[] args) {

Logger logger1 = Logger.*getInstance*();

Logger logger2 = Logger.*getInstance*();

logger1.testLog(1, 2);

logger2.testLog(3, 4);

if (logger1 == logger2) {

System.*out*.println("Only one instance is used for the entire application");

} else {

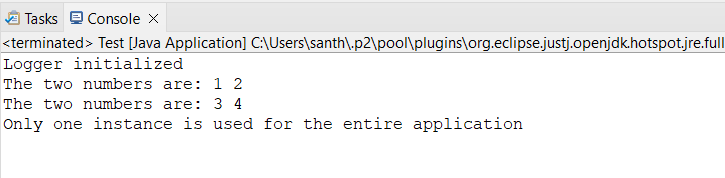
System.*out*.println("Different instances are used");

}

}

}

**OUTPUT:**



**EXERCISE-2: IMPLEMENTING THE FACTORY METHOD PATTERN**

**WordDocument.java**

package FactoryMethod;

public interface WordDocument {

void openDocuments();

}

PdfDocument.java

package FactoryMethod;

public interface PdfDocument {

void openDocuments();

}

**ExcelDocument.java**

package FactoryMethod;

public interface ExcelDocument {

void openDocuments();

}

**WordInterfaceImplement.java**

package FactoryMethod;

public class WordInterfaceImplement implements WordDocument {

@Override

public void openDocuments() {

System.*out*.println("Word is opened");

}

}

**PdfInterfaceImplements.java**

package FactoryMethod;

public class PdfInterfaceImplements implements PdfDocument{

@Override

public void openDocuments() {

System.*out*.println("Pdf is opened");

}

}

**ExcelInterfaceImplements.java**

package FactoryMethod;

public class ExcelInterfaceImplements implements ExcelDocument{

@Override

public void openDocuments() {

System.*out*.println("Excel is opened");

}

}

**DocumentFactory.java**

package FactoryMethod;

public abstract class DocumentFactory {

public abstract Object createDocument();

}

**WordFactory.java**

package FactoryMethod;

public class WordFactory extends DocumentFactory {

@Override

public WordDocument createDocument() {

return new WordInterfaceImplement();

}

}

**PdfFactory.java**

package FactoryMethod;

public class PdfFactory extends DocumentFactory {

@Override

public PdfDocument createDocument() {

return new PdfInterfaceImplements();

}

}

**ExcelFactory.java**

package FactoryMethod;

public class ExcelFactory extends DocumentFactory {

@Override

public ExcelDocument createDocument() {

return new ExcelInterfaceImplements();

}

}

**Test.java**

package FactoryMethod;

public class Test {

public static void main(String[] args) {

WordFactory wordFactory = new WordFactory();

WordDocument wordDoc = wordFactory.createDocument();

wordDoc.openDocuments();

PdfFactory pdfFactory = new PdfFactory();

PdfDocument pdfDoc = pdfFactory.createDocument();

pdfDoc.openDocuments();

ExcelFactory excelFactory = new ExcelFactory();

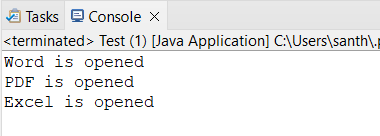
ExcelDocument excelDoc = excelFactory.createDocument();

excelDoc.openDocuments();

}

}

**OUTPUT:**



**DATA STRUCTURES AND ALGORITHMS**

**EXERCISE-2: E-COMMERCE PLATFORM SEARCH FUNCTION**

**Big-O notation:**

Big O Notation is a mathematical concept used in computer science to describe the efficiency of an algorithm in terms of time or space complexity as the input size grows.

It focuses on the upper bound of an algorithm's running time (worst-case scenario), ignoring constants and lower-order terms, to provide a general idea of performance scalability.

**Product.java:**

import java.util.\*;

import java.util.Comparator;

class Product {

int productId;

String productName;

String category;

public Product() {

}

public Product(int id, String name, String category) {

// TODO Auto-generated constructor stub

this.productId = id;

this.productName = name;

this.category = category;

}

public String linearSearch(Product[] products, int id) {

for(int i=0;i<products.length;i++) {

if(products[i].productId == id) {

return "Product is present in the application";

}

}

return "Product is not there";

}

public String binarySearch(Product[] products, int id) {

int low = 0;

int high = products.length;

while(low<high) {

int mid = (low+high)/2;

if(products[mid].productId == id) {

return "Product is present in the application";

}

else if(products[mid].productId < id) {

low = mid+1;

}

else {

high = mid-1;

}

}

return "Product is not present in products";

}

public static void main(String[] args) {

Scanner scn = new Scanner(System.*in*);

// TODO Auto-generated method stub

Product[] products = {new Product(1, "IPhone", "Smart phones"),

new Product(2, "Samsung", "Smart phones"),

new Product(3, "Notes", "Stationary"),

new Product(4, "Laptop", "Electronics"),

new Product(5, "Pen", "Stationary"),

new Product(6, "Redmi", "Smart phones"),

};

Product obj = new Product();

Product[] sortedProducts = products;

Arrays.*sort*(sortedProducts, Comparator.*comparingInt*(prod -> prod.productId));

System.*out*.print("Enter an prodct id to search: ");

int id = scn.nextInt();

System.*out*.println("Enter the type of search of your choice(Linear/Binary): ");

String searchType = scn.next();

String res;

if(searchType.toLowerCase() == "linear") {

res = obj.linearSearch(products, id);

}

else {

res = obj.binarySearch(sortedProducts, id);

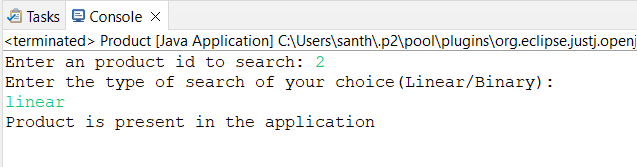
}

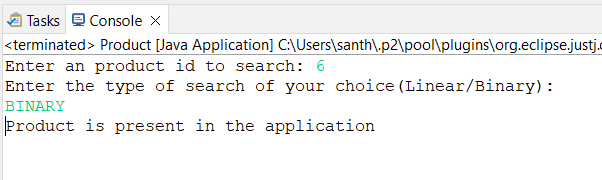
System.*out*.println(res);

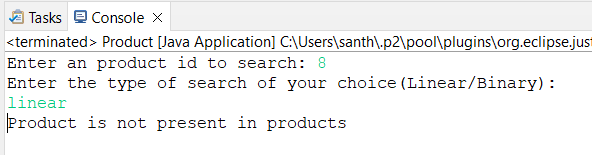
}

}

**OUTPUT:**







**Analysis:**

Time complexity for linear search:

1. Best case: O(1) – If element is present at the stating index.
2. Average case: O(n) – If element is present in the middle or end of the array.
3. Worst case: O(n) - If element is present in the end of the array.

Time complexity for binary search:

1. Best case: O(1) – If element is found as the first mid element.
2. Average case: O(logn) – Logarithmic complexity
3. Worst case: O(logn) - Logarithmic complexity

Which is more suitable?

Linear search:

1. Linear search is ideal when the search space is small.
2. It can be used when the data is unsorted.

Binary search:

1. Binary search is preferred when the data is so large.
2. It is faster on large datasets than linear search.
3. It requires sorted data.

**EXERCISE-7: FINANCIAL FORECASTING**

Recursion is a technique where a function calls itself to solve a problem. Each recursive call breaks the problem down into smaller sub-problems, eventually reaching a base case to stop recursion.

1.It simplifies problems that are naturally repetitive or hierarchical.

2.Commonly used in tasks like Fibonacci sequence, tree traversal, backtracking, etc.

3.For forecasting, if each value depends on previous ones (e.g., compound growth), recursion provides a clean way to model this.

**Forecasting.java**

public class Forecasting {

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

if (years == 0) {

return principal;

}

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

}

public static void main(String[] args) {

double initialInvestment = 1000;

double growthRate = 0.05;

int futureYears = 10;

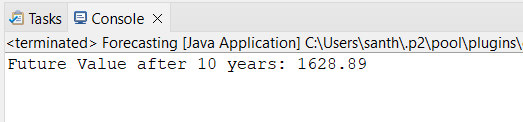
double result = *futureValue*(initialInvestment, growthRate, futureYears);

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

}

}

**OUTPUT:**



**Time Complexity:**

The function calls itself once per year, so:

Time Complexity: **O(n)** where n is the number of years.

**Space Complexity:**

Due to the recursive call stack: **O(n)** space is used.

**Optimization:**

**Use Iteration Instead of Recursion**

public static double futureValueIterative(double principal, double rate, int years) {

double result = principal;

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

result \*= (1 + rate);

}

return result;

}

Time Complexity: O(n)

Space Complexity: O(1)