**MANDATORY HANDS – ON WEEK 1**

**Part 1 : Design Patterns and Principles.**

Exercise 1: Implementing the Singleton Pattern

Code:

class Logger{

private static Logger obj;

private Logger() {

System.out.println("Class - 'Logger' is called");

}

public static Logger getInstance() {

return obj;

}

}

public class Main {

public static void main(String[] args) {

Logger obj1 = Logger.getInstance();

Logger obj2 = Logger.getInstance();

if (obj1 == obj2) {

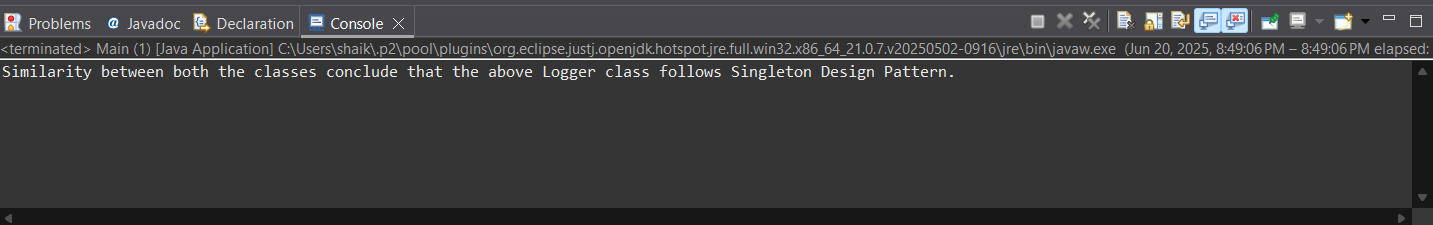
System.out.println("Similarity between both the classes conclude that the above Logger class follows Singleton Design Pattern.");

}

}

}

Output:



Exercise 2: Factory Method Pattern

Code:

// Document.java

public class Word implements Document{

public void open() {

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

}

}

// Word.java

public class Word implements Document{

public void open() {

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

}

}

// Pdf.java

public class Pdf implements Document{

public void open() {

System.***out***.println("Pdf Document");

}

}

// Excel.java

public class Excel implements Document{

public void open() {

System.***out***.println("Excel Document");

}

}

// Factory,java

public abstract class Factory {

public abstract Document createDocument();

}

// WordFactory.java

public class WordFactory extends Factory{

public Document createDocument() {

return new Word();

}

}

// PdfFactory.java

public class PdfFactory extends Factory{

public Document createDocument() {

return new Pdf();

}

}

// ExcelFactory.java

public class xcelFactory extends Factory{

public Document createDocument() {

return new Excel();

}

}

// Test.java

public class Test {

public static void main(String[] args) {

Factory a = new WordFactory();

Document WordDoc = a.createDocument();

WordDoc.open();

Factory b = new PdfFactory();

Document PdfDoc = b.createDocument();

PdfDoc.open();

Factory c = new ExcelFactory();

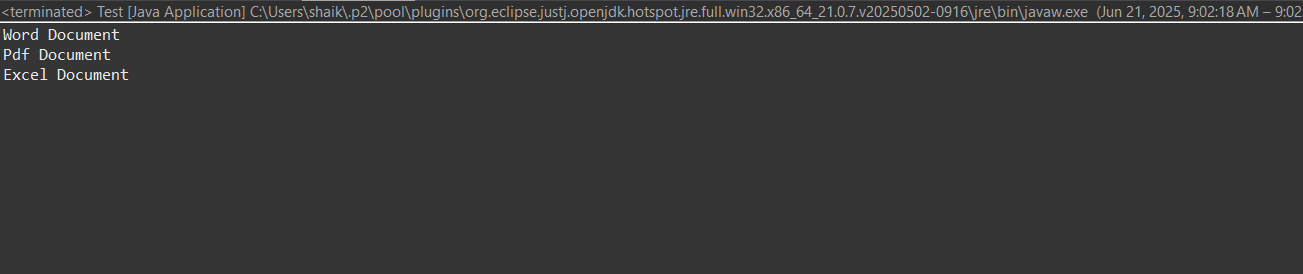
Document ExcelDoc = c.createDocument();

ExcelDoc.open();

}

}

Output:



**Part 2 : Data Structures and Algorithms.**

Exercise1: E-commerce Platform Search Function

Big-O Notation:

Two functions f(n) and g(n), we say that f(n) is O(g(n)) if there exist constants c > 0 and n0 >= 0 such that f(n) <= c\*g(n) for all n >= n0.

Performance:

|  |  |  |
| --- | --- | --- |
| Notation | Linear Search  (Time Complexity) | Binary Search  (Time Complexity) |
| Best Case | O(1) | O(1) |
| Average Case | O(n) | O(logn) |
| Worst Case | O(n) | O(logn) |

Code:

class Product {

public String productId;

public String productName;

public String category;

public Product(String id, String name, String Category) {

this.productId = id.toLowerCase();

this.productName = name.toLowerCase();

this.category = Category.toLowerCase();

}

public String[] getInfo() {

String a = this.productId;

String b = this.productName;

String c = this.category;

String info[] = {a,b,c};

return info;

}

}

public class Searches {

public static void main(String args[]) {

Product product1 = new Product("A", "Cricket Bat", "Sports");

Product product2 = new Product("B", "Smart Phone", "Electronics");

Product product3 = new Product("C", "Stove", "Kitchen");

String[][] products = {

product2.getInfo(),

product3.getInfo(),

product1.getInfo()

};

// Linear Search

int found = 0;

int position = 0;

String Search\_Id = "B";

Search\_Id = Search\_Id.toLowerCase();

for (int i = 0; i < 3; i ++) {

if (products[i][0].equals(Search\_Id.toLowerCase())) {

found = 1;

position = i;

break;

}

}

System.***out***.println("\*\*\*LINEAR SEARCH\*\*\*");

if (found == 1){

System.***out***.println("Found as Product Number:" + (position+1));

}

else {

System.***out***.println("No product with that Id is present");

}

// Sorting the arrys 'products' before binary search.

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

for(int j = 0; j < products.length-i-1; j++) {

if (products[j][0].compareTo(products[j+1][0]) > 0){

String temp[] = products[j];

products[j] = products[j+1];

products[j+1] = temp;

}

}

}

// Binary Search

found = 0;

System.***out***.println("\*\*\*BINARY SEARCH\*\*\*");

int l = 0;

int h = products.length-1;

while (l<=h) {

int m = (l+h)/2;

if (products[m][0].equals(Search\_Id.toLowerCase())){

System.***out***.println("Found as Product Number:" + (m+1));

found = 1;

break;

}

else if (products[m][0].compareTo(Search\_Id) < 0) {

l = m+1;

}

else {

h = m-1;

}

}

if (found == 0) {

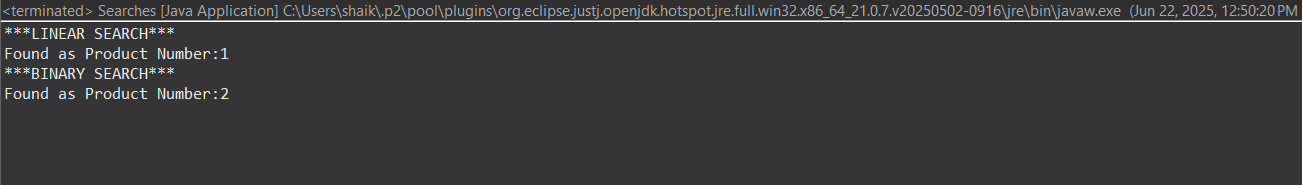
System.***out***.println("No Product with that Id is present");

}

}

}

Output:



Performance Analysis:

Binary Search better option than Linear search as it takes less time in all cases (i.e. to find an item ar any index).

Exercise 2: Financial Forecasting.

Recursion is a programming technique where a method calls itself to solve a problem. This technique is particularly useful for problems that can be broken down into smaller, similar subproblems.

Let us calculate the future value of an amount including compound interest (C.I) over the years.

**Formula:**

A=P×(1+(r/n))^(nt)

Where:

* A: final amount after time t
* P: principal amount (initial investment)
* r: annual interest rate (in decimal)
* n: number of times interest applied per year (compounded)
* t: number of years

But in our simplified recursive model, we’ll assume:

* Interest is compounded once per year i.e. n = 1
* So the formula simplifies to:

A=P×(1+r)^t

Code:

import java.util.Scanner;

public class Main {

public static double CompoundInterest(double amount, double rate, int years) {

if (years == 0) {

return amount;

} else {

return *CompoundInterest*(amount \* (1 + rate), rate, years - 1);

}

}

public static void main(String[] args) {

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

System.***out***.println("Enter Principle amount: ");

double principal = s.nextDouble();

System.***out***.println("Enter interest rate: ");

double interestRate = s.nextDouble();

System.***out***.println("Enter time period: ");

int duration = s.nextInt();

s.close();

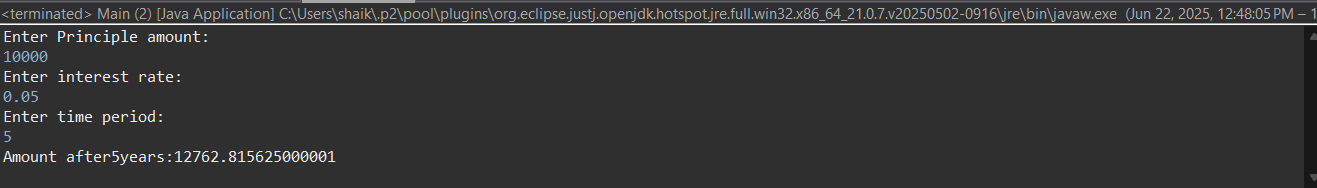
double FinalAmount = *CompoundInterest*(principal, interestRate, duration);

System.***out***.printf("Amount after" + duration +"years:" + FinalAmount);

}

}

Output:



Analysis:

Recursive approach takes O(n) time where n is the time period.