**Week 1 Hands-on exercises:**

Data Structures and Algorithms:

**Exercise 2:**

E-commerce Platform Search Function Scenario: You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance. Steps: Understand Asymptotic Notation: Explain Big O notation and how it helps in analyzing algorithms. Describe the best, average, and worst-case scenarios for search operations. Setup: Create a class Product with attributes for searching, such as productId, productName, and category. Implementation: Implement linear search and binary search algorithms. Store products in an array for linear search and a sorted array for binary search. Analysis: Compare the time complexity of linear and binary search algorithms. Discuss which algorithm is more suitable for your platform and why.

• As the search needs to be optimized for fast performance:

Big O Notation tells us how fast our search performs as the number of products grows.

O(1) → Constant time

O(n) → Linear time

O(log n) → Logarithmic time

O(n log n) → Linearithmic time

O(n²) → Quadratic time

• Linear Search

Best Case: O(1) → First element matches.

Average Case: O(n/2) ≈ O(n)

Worst Case: O(n) → Last or not found.

• Binary Search

Best Case: O(1) → Middle element matches.

Average/Worst Case: O(log n)

**Source Code :**

import java.util.Arrays;

import java.util.Comparator;

import java.util.Scanner;

public class ProductSearchWithUserInput {

static 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 "Product ID: " + productId + ", Name: " + productName + ", Category: " + category;

}

}

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) {

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

int left = 0;

int right = products.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

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

if (cmp == 0) {

return products[mid];

} else if (cmp < 0) {

right = mid - 1;

} else {

left = mid + 1;

}

}

return null;

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter number of products: ");

int n = Integer.parseInt(scanner.nextLine());

Product[] products = new Product[n];

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

System.out.println("\nEnter details for product " + (i + 1) + ":");

System.out.print("Product ID: ");

int id = Integer.parseInt(scanner.nextLine());

System.out.print("Product Name: ");

String name = scanner.nextLine();

System.out.print("Category: ");

String category = scanner.nextLine();

products[i] = new Product(id, name, category);

}

System.out.print("\nEnter product name to search: ");

String searchKey = scanner.nextLine();

Product foundLinear = linearSearch(products, searchKey);

System.out.println("\nLinear Search Result:");

if (foundLinear != null) {

System.out.println(foundLinear);

} else {

System.out.println("Product not found.");

}

Product foundBinary = binarySearch(products, searchKey);

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

if (foundBinary != null) {

System.out.println(foundBinary);

} else {

System.out.println("Product not found.");

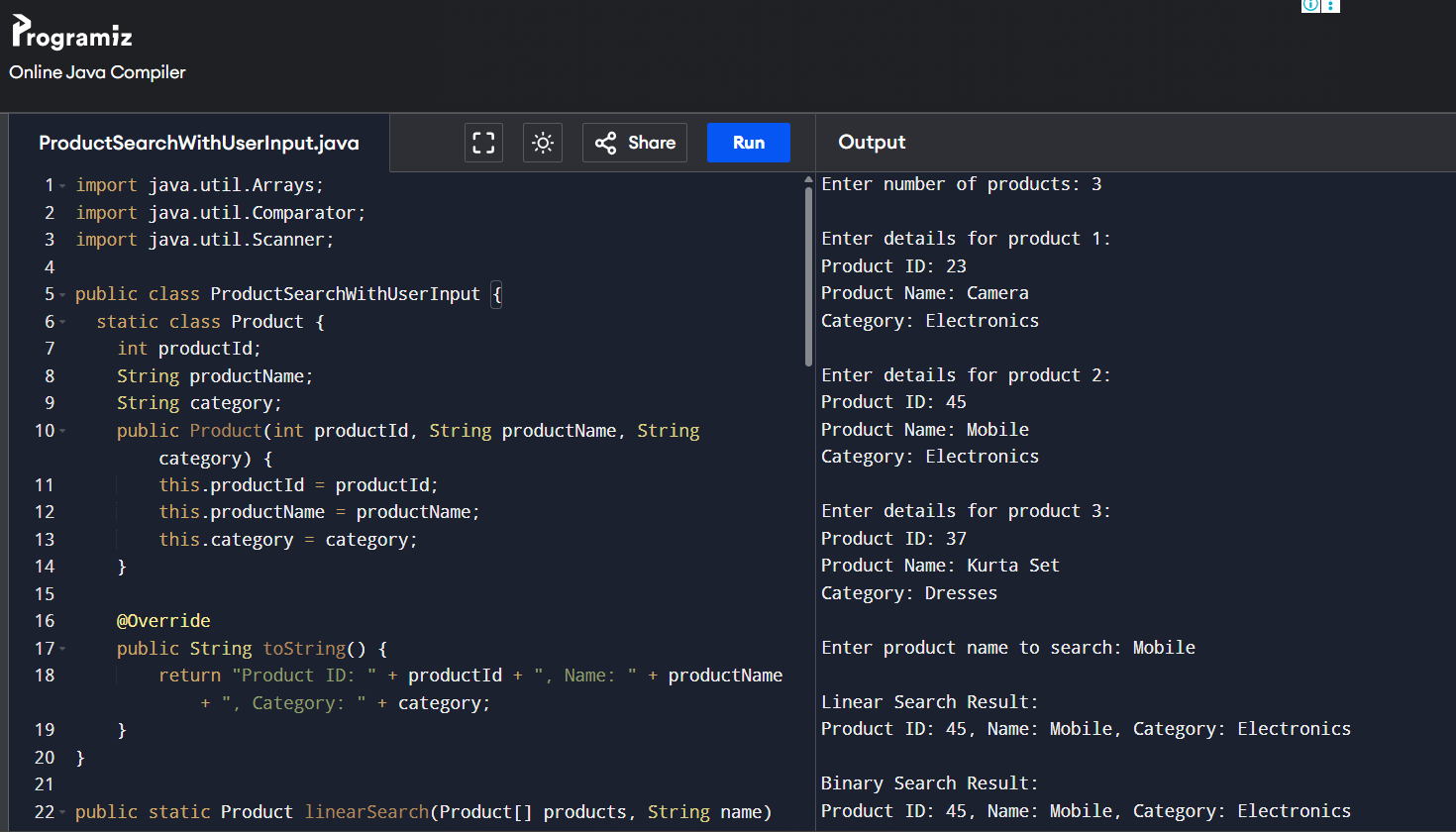
}

scanner.close();

}

}

**Output :**



In an e-commerce platform , product search happens very frequently.

So, Binary Search is more suitable for the core product search function.

**Exercise 7:**

FinancialForecasting - You are developing a financial forecasting tool that predicts future values based on past data.

**Steps:**

1. **Understand Recursive Algorithms:**
   * Explain the concept of recursion and how it can simplify certain problems.
2. **Setup:**
   * Create a method to calculate the future value using a recursive approach.
3. **Implementation:**
   * Implement a recursive algorithm to predict future values based on past growth rates.
4. **Analysis:**
   * Discuss the time complexity of your recursive algorithm.

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

* Recursion is a function calling itself to solve small instances.

**Source Code :**

import java.util.\*;

public class FinancialForecast {

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) {

Scanner sc=new Scanner(System.in);

double principal = sc.nextDouble();

double rate = sc.nextDouble();

int years = sc.nextInt();

double futureVal = futureValue(principal, rate, years);

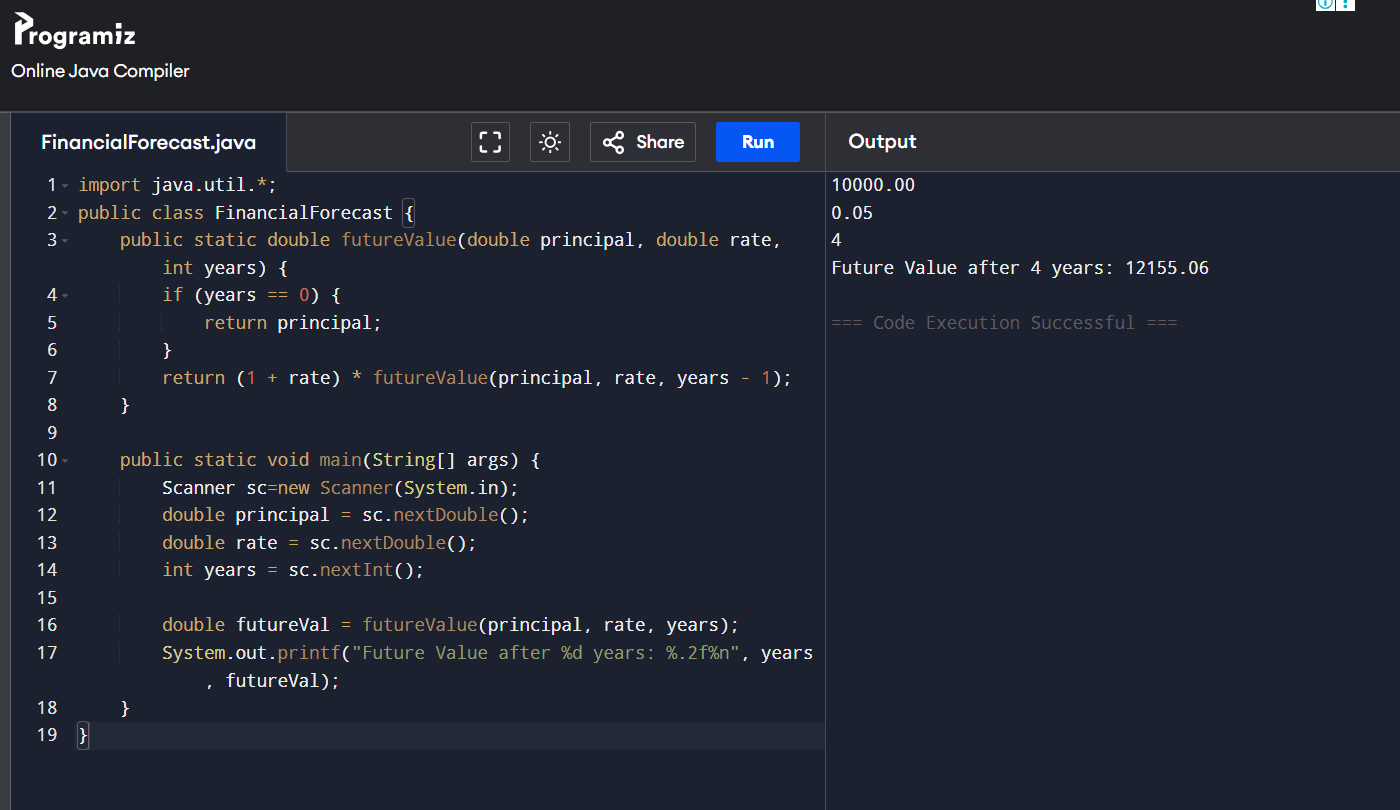
System.out.printf("Future Value after %d years: %.2f%n", years, futureVal);

}

}

***Time Complexity : O(n).***

**Output :**



Design Patterns and Principles :

**Exercise 1:**

Implementing the Singleton Pattern

Scenario:

You need to ensure that a logging utility class in your application has only one instance throughout the application lifecycle to ensure consistent logging.

Steps:

1. Create a New Java Project:
   * Create a new Java project named SingletonPatternExample.
2. Define a Singleton Class:
   * Create a class named Logger that has a private static instance of itself.
   * Ensure the constructor of Logger is private.
   * Provide a public static method to get the instance of the Logger class.
3. Implement the Singleton Pattern:
   * Write code to ensure that the Logger class follows the Singleton design pattern.
4. Test the Singleton Implementation:
   * Create a test class to verify that only one instance of Logger is created and used across the application.

**Source Code :**

**Logger.java :**

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();

}

return instance;

}

public void log(String message) {

System.out.println(message);

}

}

**TestLogger.java :**

public class TestLogger {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

logger1.log("Starting the application...");

Logger logger2 = Logger.getInstance();

logger2.log("Application is running.");

if (logger1 == logger2) {

System.out.println("Both logger instances are the same. Singleton works!");

} else {

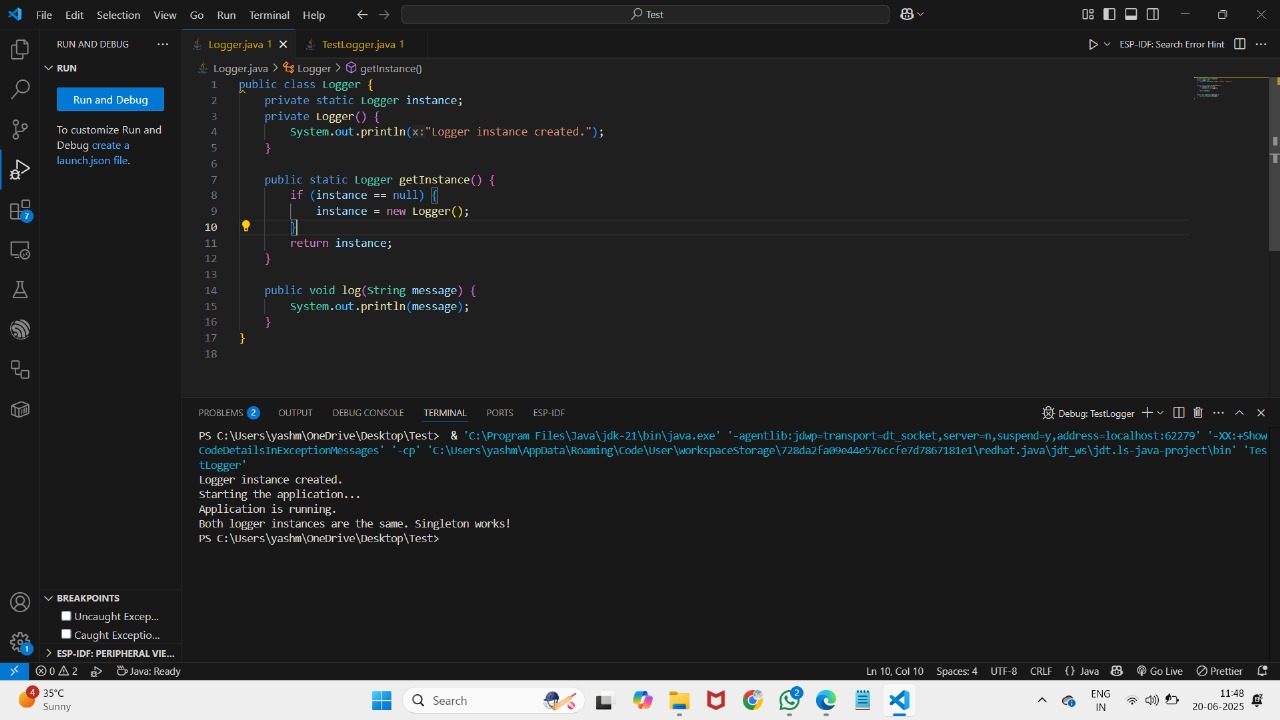
System.out.println("Different instances. Singleton failed.");

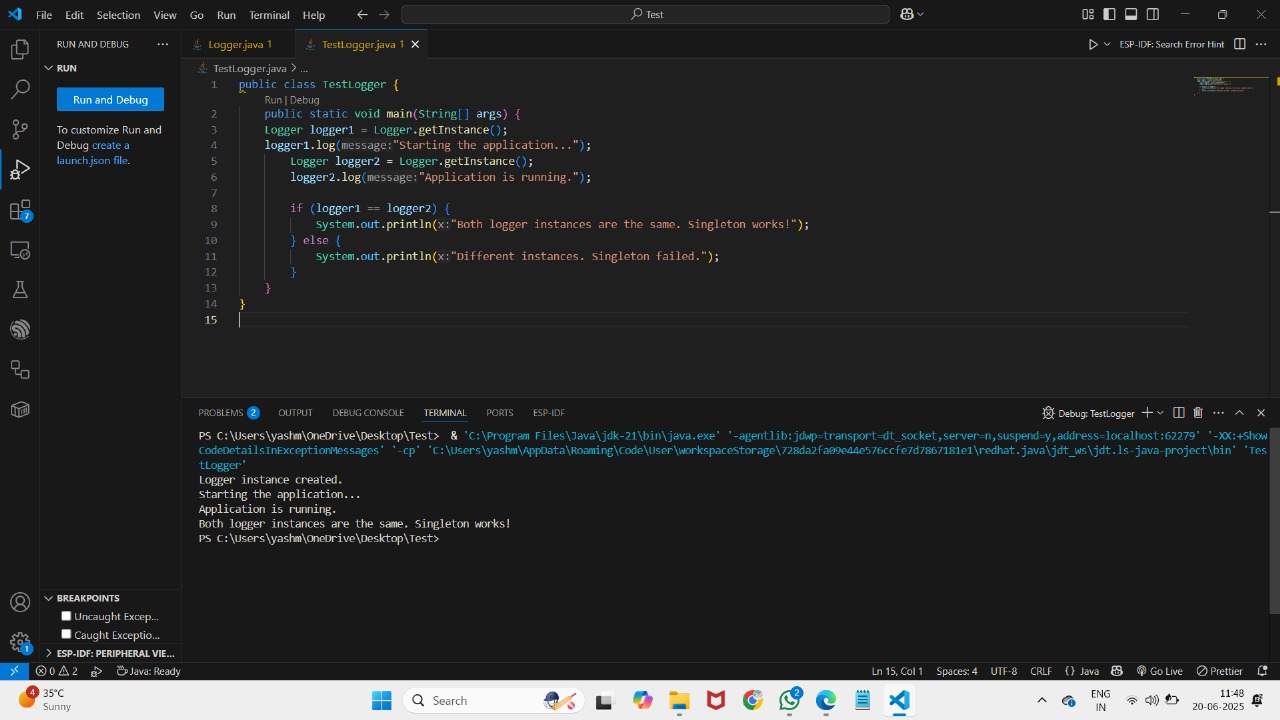
}

}

}

**Output :**





**Exercise 2:**

Implementing the Factory Method Pattern

Scenario:

You are developing a document management system that needs to create different types of documents (e.g., Word, PDF, Excel). Use the Factory Method Pattern to achieve this.

Steps:

1. Create a New Java Project:
   * Create a new Java project named FactoryMethodPatternExample.
2. Define Document Classes:
   * Create interfaces or abstract classes for different document types such as WordDocument, PdfDocument, and ExcelDocument.
3. Create Concrete Document Classes:
   * Implement concrete classes for each document type that implements or extends the above interfaces or abstract classes.
4. Implement the Factory Method:
   * Create an abstract class DocumentFactory with a method createDocument().
   * Create concrete factory classes for each document type that extends DocumentFactory and implements the createDocument() method.
5. Test the Factory Method Implementation:
   * Create a test class to demonstrate the creation of different document types using the factory method.

**Source Code :**

public class Main{

interface Document {

void open();

}

static class WordDocument implements Document {

@Override

public void open() {

System.out.println("Word document.");

}

}

static class PdfDocument implements Document {

@Override

public void open() {

System.out.println("PDF document.");

}

}

static class ExcelDocument implements Document {

@Override

public void open() {

System.out.println("Excel document.");

}

}

abstract static class DocumentFactory {

public abstract Document createDocument();

}

static class WordDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new WordDocument();

}

}

static class PdfDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new PdfDocument();

}

}

static class ExcelDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new ExcelDocument();

}

}

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();

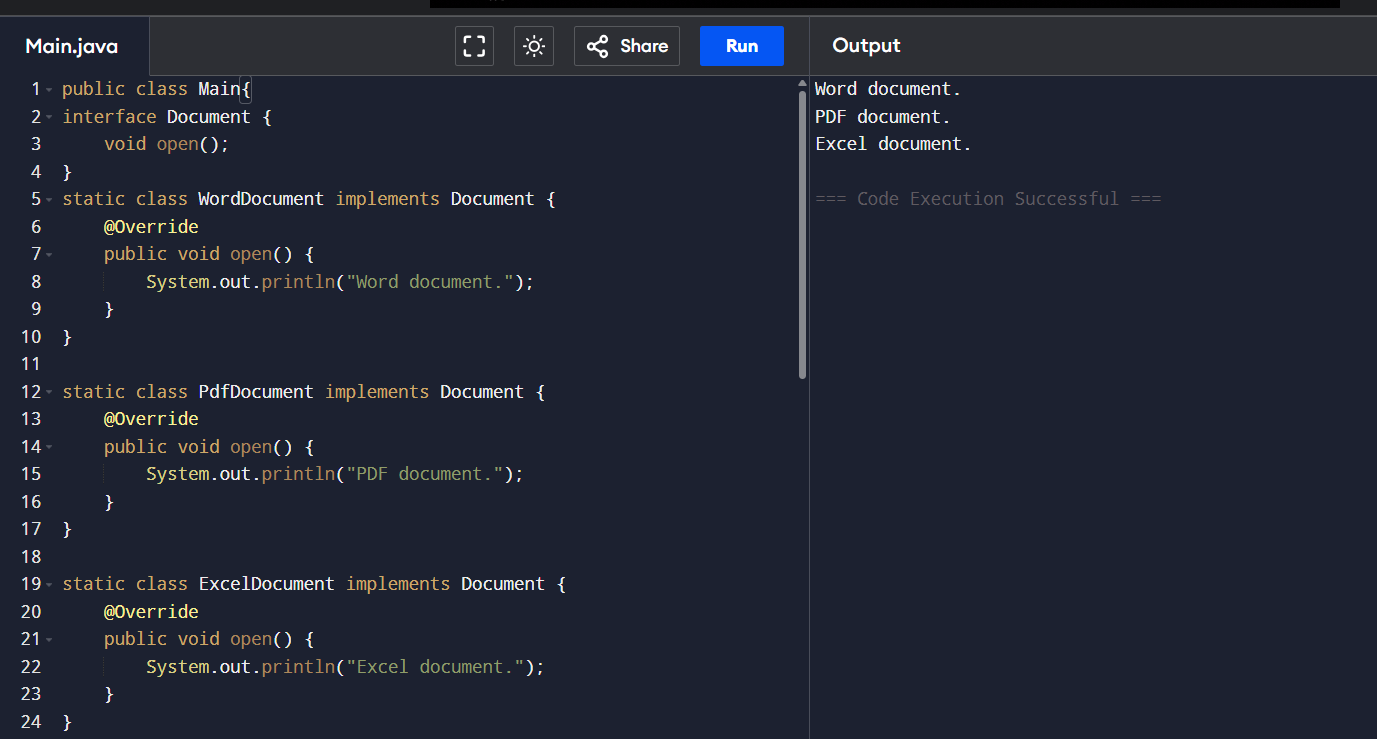
Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

**Output :**

****