**WEEK-1**

**Design Patterns and Principles | Data Structures and Algorithms**

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.

**Code-**

class Logger {

private static Logger instance; //static variable

private Logger() { //constructor

System.out.println("Logger has been initialized.");

}

public static Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

public void log(String message) {

System.out.println( message);

}

}

public class Main {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

logger1.log("Hello,Welcome.App has started.");

Logger logger2 = Logger.getInstance();

logger2.log("User clicked login.");

if (logger1 == logger2) {

System.out.println("Both loggers are the same.");

} else {

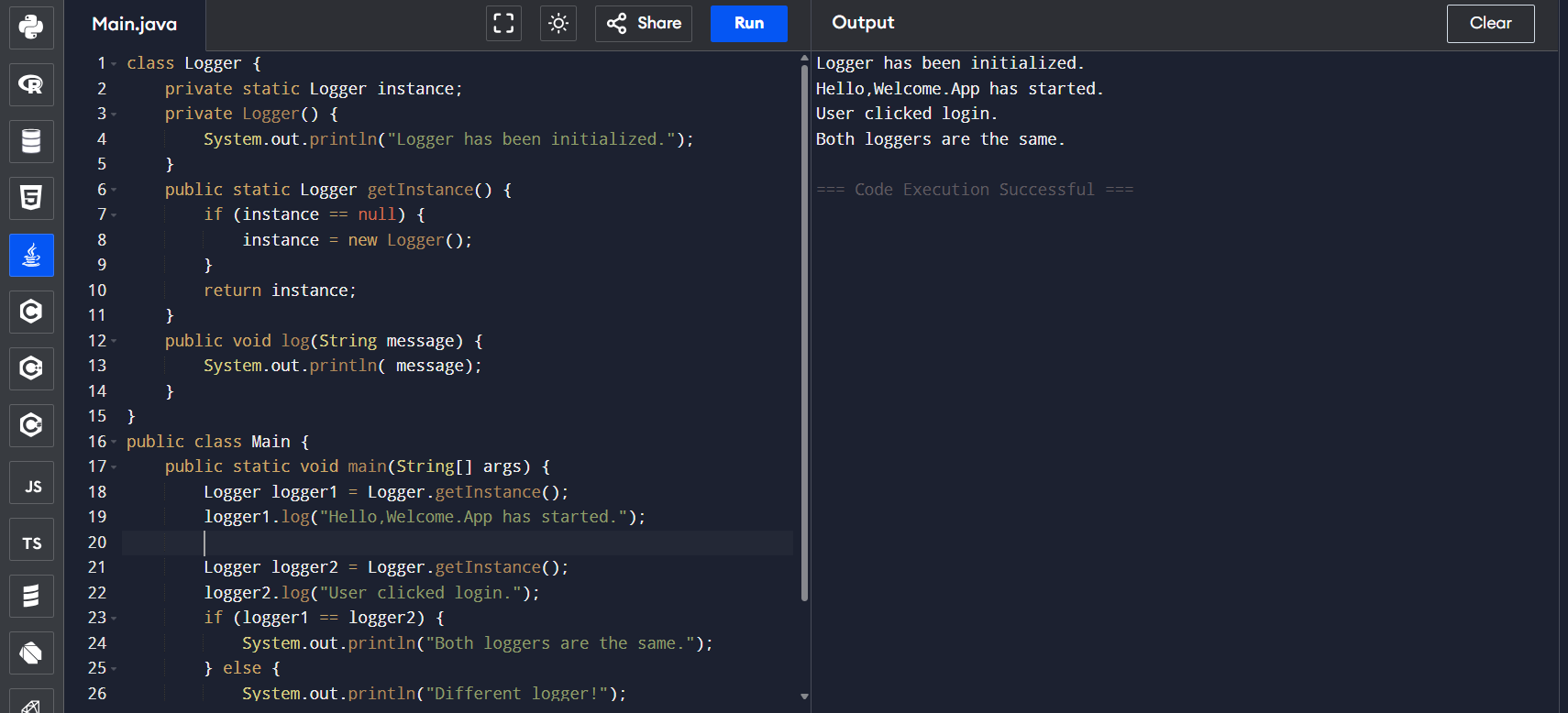
System.out.println("Different logger!");

}

}

}

**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.

**Code-**

interface Document {

void open();

}

class WordDocument implements Document {

public void open() {

System.out.println("It is a Word document...");

}

}

class PdfDocument implements Document {

public void open() {

System.out.println("It is a PDF document...");

}

}

class ExcelDocument implements Document {

public void open() {

System.out.println("It is an Excel document...");

}

}

abstract class DocumentFactory {

public abstract Document createDocument();

}

class WordFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

class PdfFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

class ExcelFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

public class Main {

public static void main(String[] args) {

DocumentFactory wordFactory = new WordFactory();

Document word = wordFactory.createDocument();

word.open();

DocumentFactory pdfFactory = new PdfFactory();

Document pdf = pdfFactory.createDocument();

pdf.open();

DocumentFactory excelFactory = new ExcelFactory();

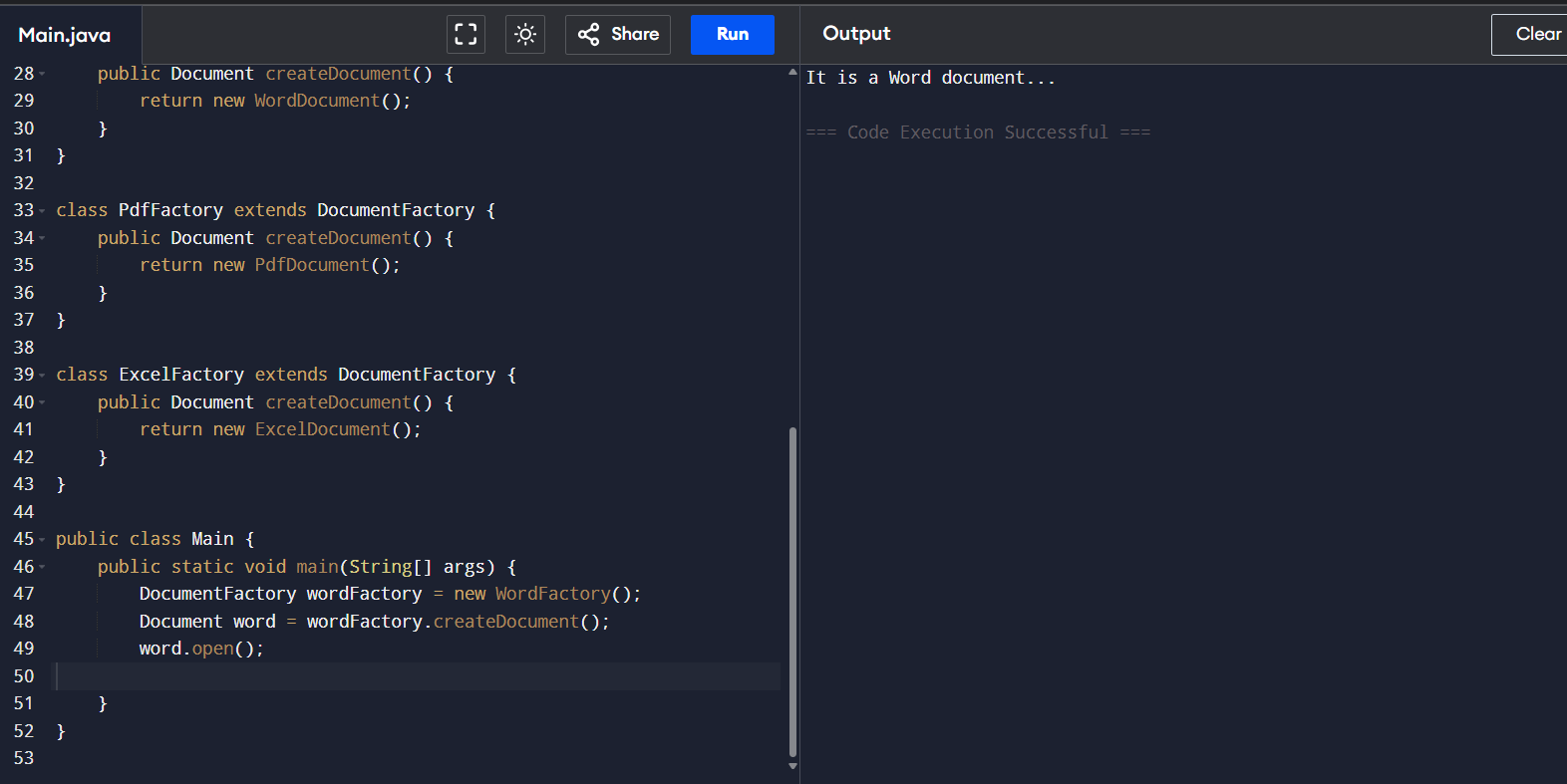
Document excel = excelFactory.createDocument();

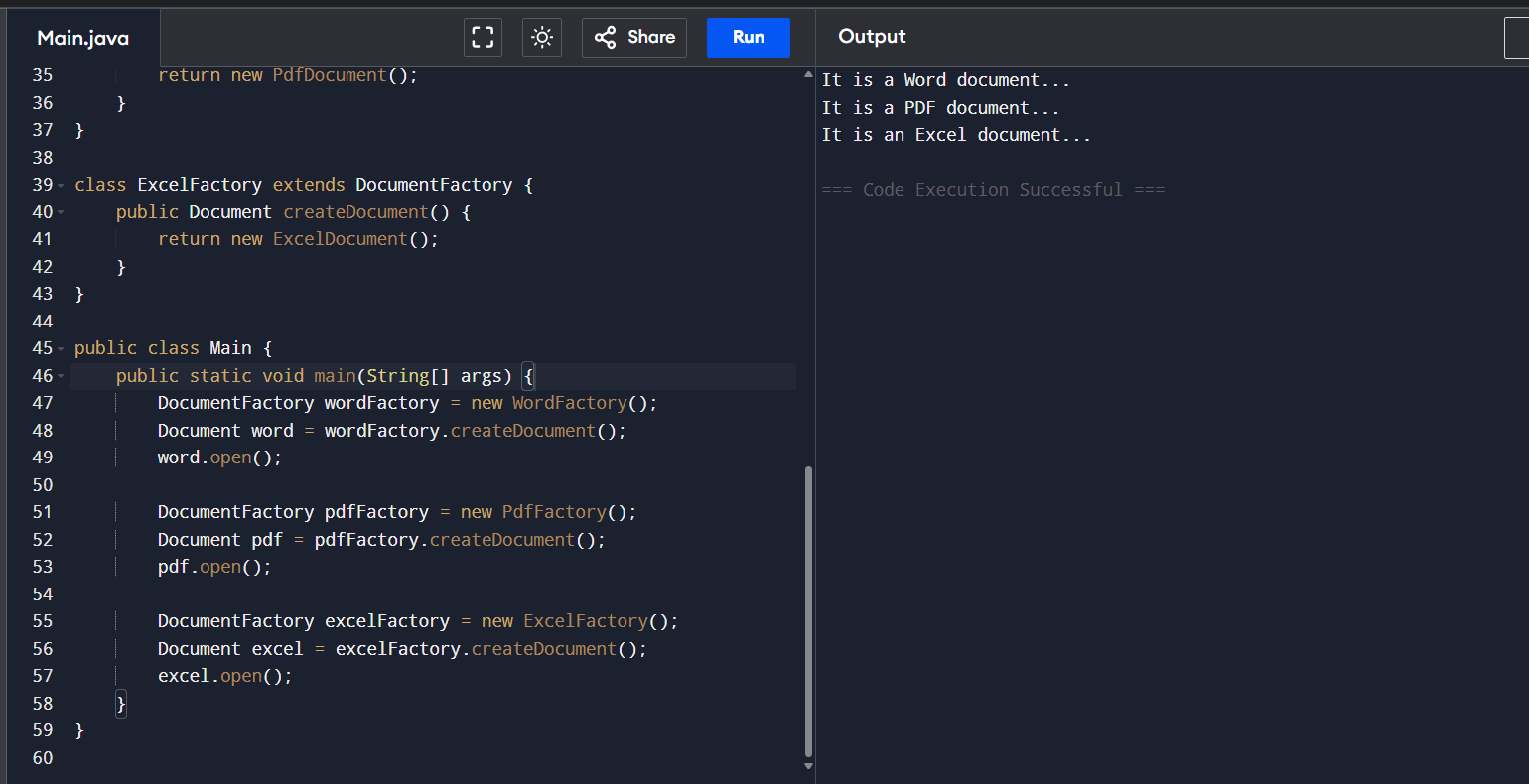
excel.open();

}

}

**Output-**

****

****

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:**

1. **Understand Asymptotic Notation:**

**Big O Notation** is used to express the time complexity of algorithms. It describes how the runtime or space requirement grows as input size increases.

**Common Cases:**

* **Best Case**: The fastest scenario (element found at first position).
* **Average Case**: Expected performance for random data.
* **Worst Case**: The slowest scenario (element not present).

|  |  |  |  |
| --- | --- | --- | --- |
| **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) |

**2. Code-**

import java.util.\*;

class Product {

int productId;

String productName;

String category;

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

this.productId = productId;

this.productName = productName;

this.category = category; }

public String toString() {

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

}

}

public class Main {

// Linear Search

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

for (Product p : products) {

if (p.productName.equalsIgnoreCase(targetName)) {

return p;

}

}

return null;

}

// Binary Search

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

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

while (left <= right) {

int mid = (left + right) / 2;

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

if (cmp == 0) return products[mid];

else if (cmp < 0) left = mid + 1;

else right = mid - 1;

}

return null;

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

Product[] products = {

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

new Product(102, "Shirt", "Clothing"),

new Product(104, "Mobile", "Electronics"),

new Product(105, "Shoes", "Footwear")

};

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

System.out.print("Enter product name You want to search: ");

String target = sc.nextLine();

Product result1 = linearSearch(products, target);

System.out.println("Linear Search Result: " + (result1 != null ? result1 : "Item Not Found"));

Product result2 = binarySearch(products, target);

System.out.println("Binary Search Result: " + (result2 != null ? result2 : "Item Not Found"));

sc.close();

}}

**Output-**

****

**3. Analysis**

* Binary Search is much faster for large datasets, but only works if the product list is sorted.So binary search is best for E-commerce Website.

**Exercise 7: Financial Forecasting**

**Scenario:**

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

**Steps:**

1. **Understand Recursive Algorithms:**

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

Example :

To compute future value for n years with a constant growth rate r, we can use:

FV(n) = FV(n-1) \* (1 + r)

1. **Code:**

import java.util.Scanner;

public class FinancialForecasting {

// Recursive method to calculate future value

public static double forecastFutureValue(double currentValue, double growthRate, int years) {

if (years == 0) return currentValue;

return forecastFutureValue(currentValue, growthRate, years - 1) \* (1 + growthRate);

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.print("Enter current value : ");

double currentValue = sc.nextDouble();

System.out.print("Enter annual growth rate (as %): ");

double ratePercent = sc.nextDouble();

double growthRate = ratePercent / 100;

System.out.print("Enter number of years to forecast: ");

int years = sc.nextInt();

double futureValue = forecastFutureValue(currentValue, growthRate, years);

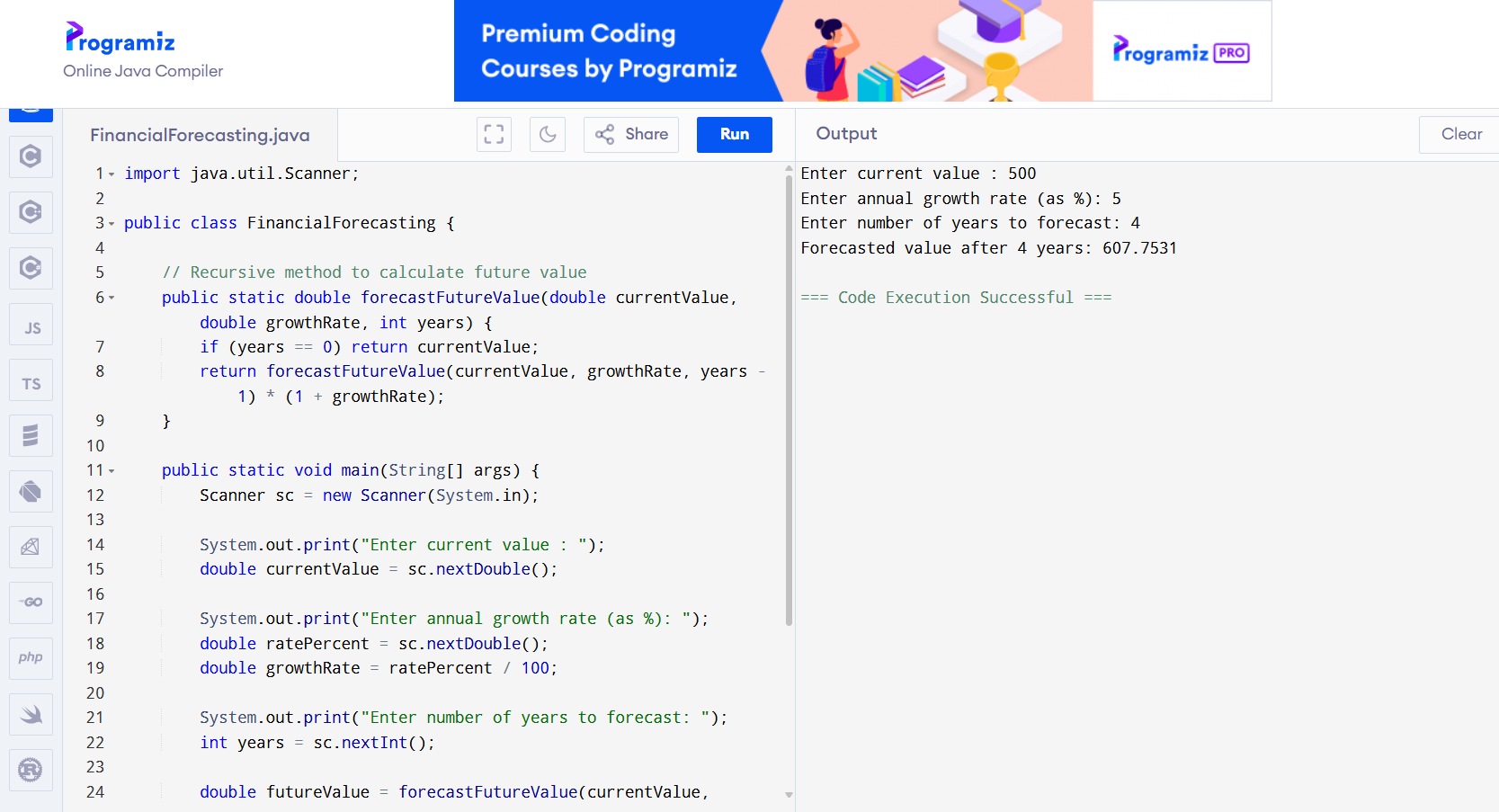
System.out.printf("Forecasted value after %d years: %.4f%n", years, futureValue);

sc.close();

}

}

1. **Output-**

****

**4.Analysis-**

**Time Complexity:**

* The time complexity of this recursive solution is **O(n)**, where n is the number of years. For each year, one function call is made.

**Optimization Suggestion:**

* If we want to optimize further, use memoization (storing results), but here it's unnecessary because each year is computed once and doesn't repeat.

Alternatively, for large n, an iterative or direct formula is better:

**futureValue = currentValue \* Math.pow(1 + growthRate, years);**