**DESIGN PRINCIPLES & PATTERNS**

**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.

ANSWER CODE:

public class SingletonPatternExample {

static class Logger {

private static Logger logger;

private Logger() {

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

}

public static Logger getLoggerInstance() {

if (logger == null) {

logger = new Logger();

}

return logger;

}

public void logMessage(String msg) {

System.out.println("LOG >> " + msg);

}

}

public static void main(String[] args) {

Logger log1 = Logger.getLoggerInstance();

Logger log2 = Logger.getLoggerInstance();

log1.logMessage("Starting the system...");

log2.logMessage("System is running fine.");

if (log1 == log2) {

System.out.println("Same logger instance is used everywhere.");

} else {

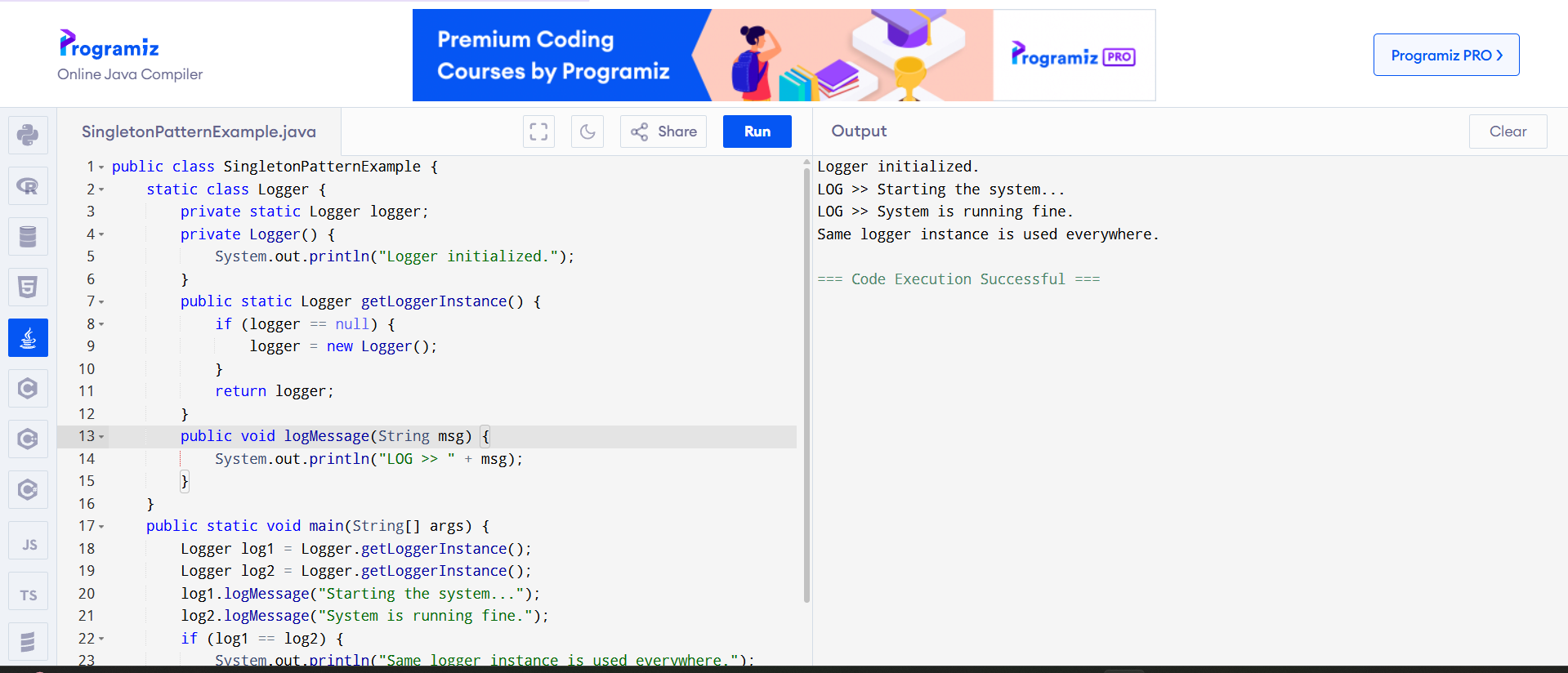
System.out.println("Different logger instances were created.");

}

}

}

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.

ANSWER CODE:

public class FactoryMethodPatternExample {

interface Document {

void displayContent();

}

static class WordDocument implements Document {

public void displayContent() {

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

}

}

static class PdfDocument implements Document {

public void displayContent() {

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

}

}

static class ExcelDocument implements Document {

public void displayContent() {

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

}

}

abstract static class DocumentFactory {

public abstract Document generateDocument();

}

static class WordFactory extends DocumentFactory {

public Document generateDocument() {

return new WordDocument();

}

}

static class PdfFactory extends DocumentFactory {

public Document generateDocument() {

return new PdfDocument();

}

}

static class ExcelFactory extends DocumentFactory {

public Document generateDocument() {

return new ExcelDocument();

}

}

public static void main(String[] args) {

DocumentFactory wordFactory = new WordFactory();

Document doc1 = wordFactory.generateDocument();

doc1.displayContent();

DocumentFactory pdfFactory = new PdfFactory();

Document doc2 = pdfFactory.generateDocument();

doc2.displayContent();

DocumentFactory excelFactory = new ExcelFactory();

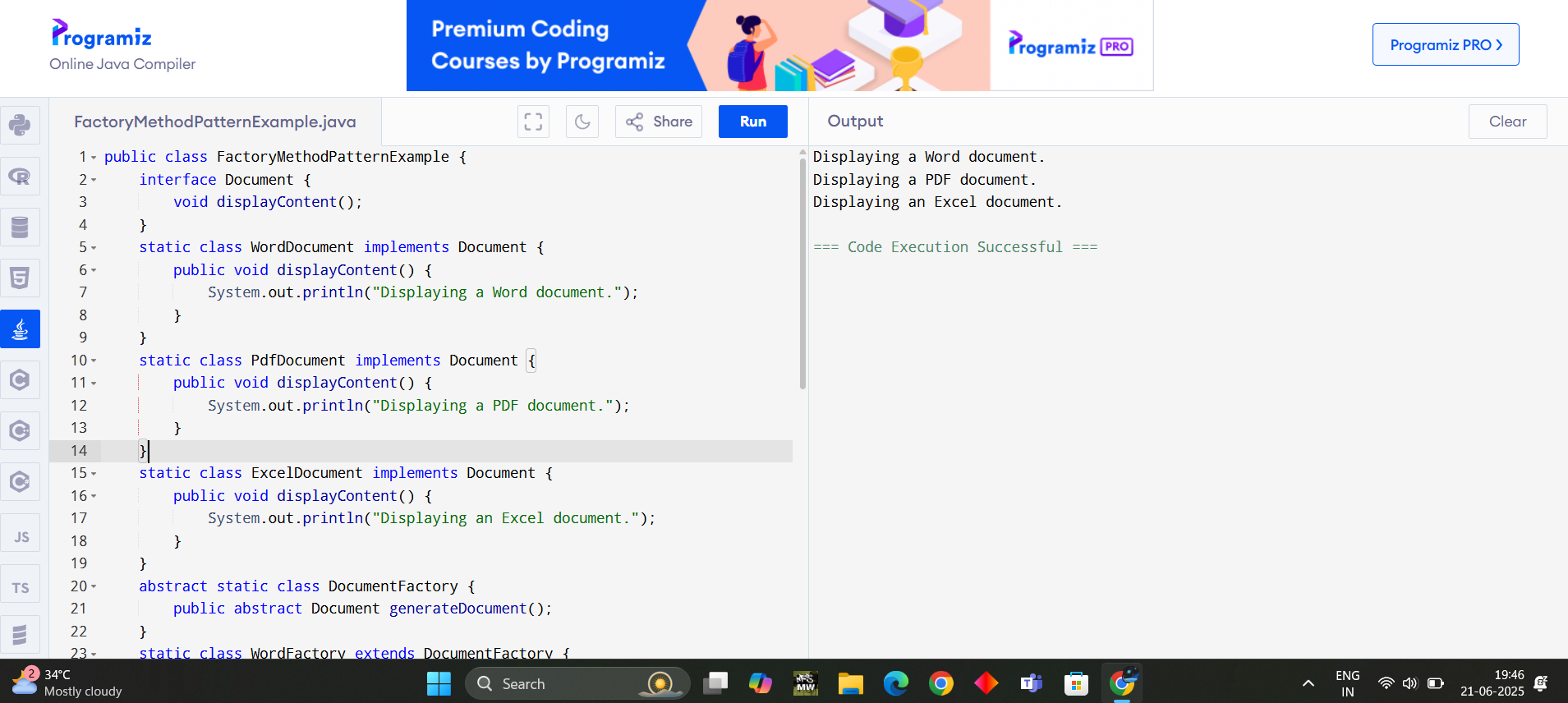
Document doc3 = excelFactory.generateDocument();

doc3.displayContent();

}

}

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:**
   * Explain Big O notation and how it helps in analyzing algorithms.
   * Describe the best, average, and worst-case scenarios for search operations.
2. **Setup:**
   * Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
3. **Implementation:**
   * Implement linear search and binary search algorithms.
   * Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
   * Compare the time complexity of linear and binary search algorithms.
   * Discuss which algorithm is more suitable for your platform and why.

ANSWER CODE:

import java.util.Arrays;

import java.util.Comparator;

public class EcommerceSearchSystem {

static class Product {

int id;

String name;

String category;

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

this.id = id;

this.name = name;

this.category = category;

}

@Override

public String toString() {

return "[" + id + "] " + name + " - Category: " + category;

}

}

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

for (Product product : inventory) {

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

return product;

}

}

return null;

}

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

int left = 0;

int right = sortedInventory.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

int comparison = sortedInventory[mid].name.compareToIgnoreCase(targetName);

if (comparison == 0) return sortedInventory[mid];

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

else right = mid - 1;

}

return null;

}

public static void main(String[] args) {

Product[] products = {

new Product(201, "Laptop", "Electronics"),

new Product(202, "Shoes", "Fashion"),

new Product(203, "Keyboard", "Electronics"),

new Product(204, "Notebook", "Stationery"),

new Product(205, "Smartphone", "Electronics")

};

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

Product resultLinear = linearSearch(products, "Keyboard");

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

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

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

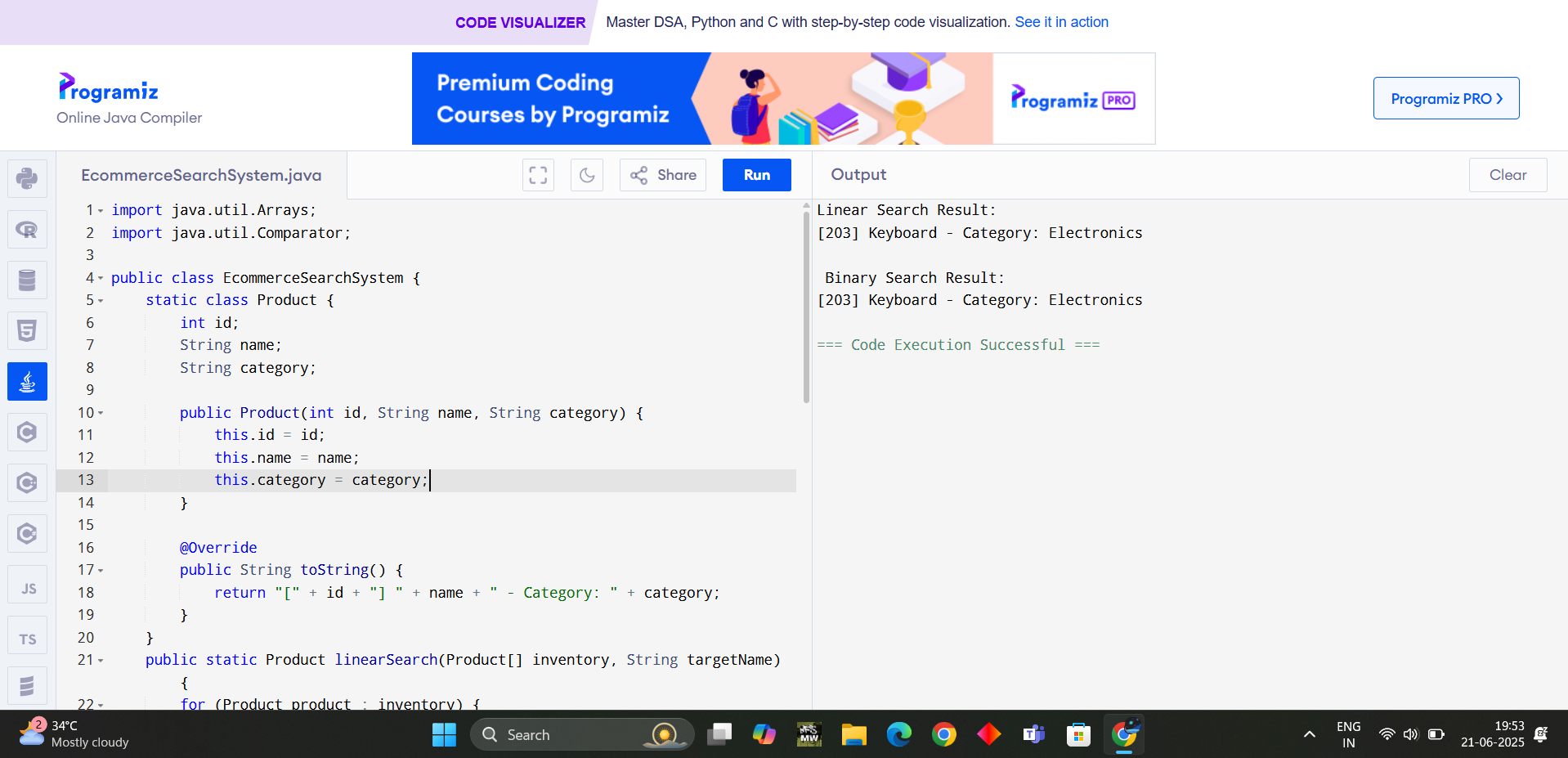
Product resultBinary = binarySearch(products, "Keyboard");

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

}

}

OUTPUT:



**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:**
   * 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.

ANSWER CODE:

public class FinancialForecastingTool {

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

if (years == 0) {

return currentValue;

}

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

}

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

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

currentValue \*= (1 + growthRate);

}

return currentValue;

}

public static void main(String[] args) {

double initialInvestment = 10000.0;

double annualGrowthRate = 0.08; // 8%

int forecastYears = 5;

System.out.println("Financial Forecasting Tool");

System.out.println("------------------------------------------------");

System.out.println("Initial Amount : $" + initialInvestment);

System.out.println("Annual Growth Rate : " + (annualGrowthRate \* 100) + "%");

System.out.println("Forecast Period : " + forecastYears + " years\n");

double futureRecursive = calculateFutureValueRecursive(initialInvestment, annualGrowthRate, forecastYears);

double futureIterative = calculateFutureValueIterative(initialInvestment, annualGrowthRate, forecastYears);

System.out.printf("Recursive Forecast Value: $%.2f\n", futureRecursive);

System.out.printf("Iterative Forecast Value: $%.2f\n", futureIterative);

System.out.println("\n Both methods yield the same result. Use iteration for efficiency in large datasets.");

}

}

OUTPUT:

