**WEEK-1**

**DESIGN 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.

**Solution:**

**// Logger.java**

**public class Logger {**

**private static Logger instance;**

**private Logger() {**

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

**}**

**public static Logger getInstance() {**

**if (instance == null) {**

**instance = new Logger(); // lazy initialization**

**}**

**return instance;**

**}**

**// Logger functionality**

**public void log(String message) {**

**System.out.println("Log: " + message);**

**}**

**}**

**// Main.java**

**public class Main {**

**public static void main(String[] args) {**

**Logger lg1 = Logger.getInstance();**

**Logger lg2 = Logger.getInstance();**

**lg1.log("This is the first log message.");**

**lg2.log("This is the second log message.");**

**// Verify if both instances are the same**

**if (lg1 == lg2) {**

**System.out.println("Both instances are the same. Singleton verified.");**

**} else {**

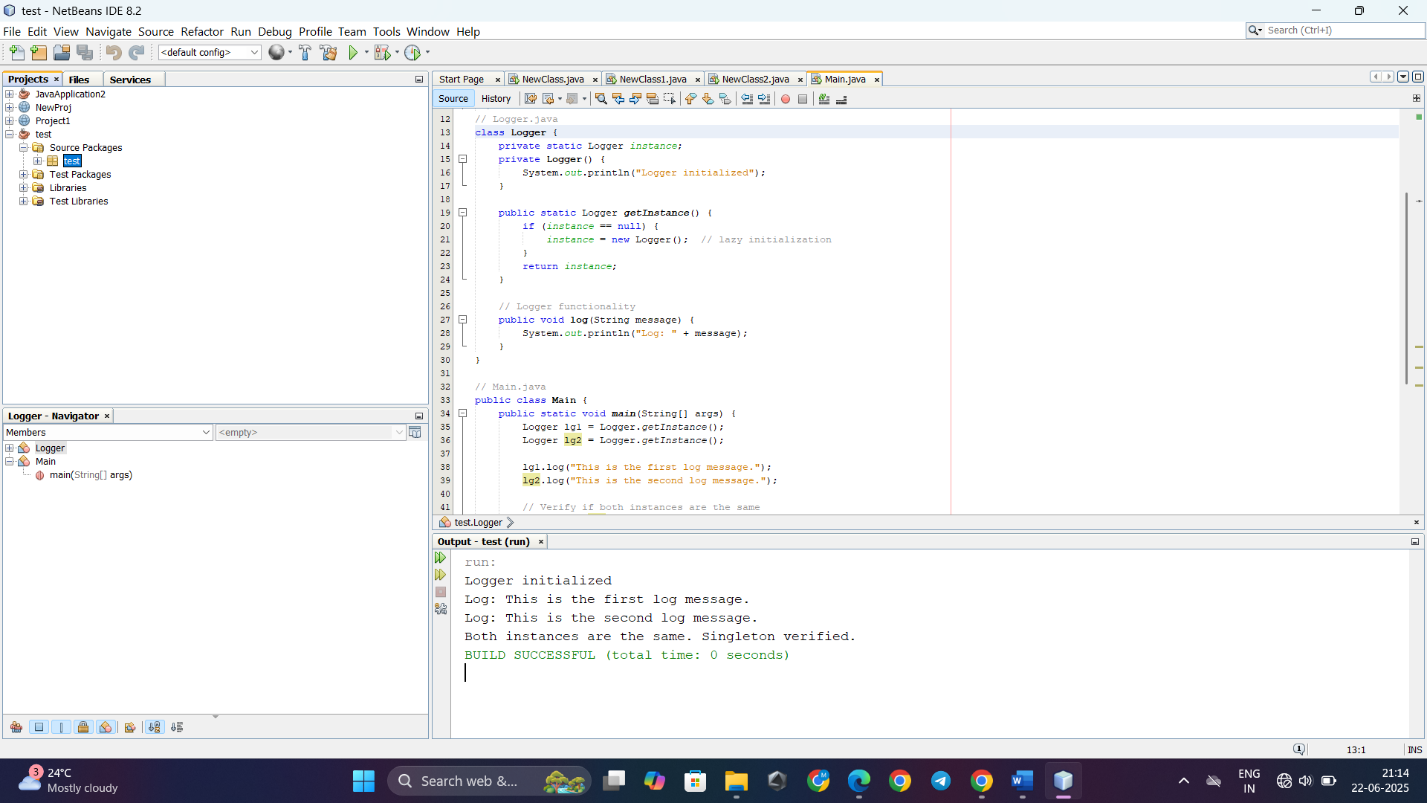
**System.out.println("Different logger instances. Singleton failed.");**

**}**

**}**

**}**

**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.

**Solution:**

**public class Main {**

**public static void main(String[] args) {**

**// Word document**

**DocumentFactory wordFactory = new WordDocumentFactory();**

**Document wordDoc = wordFactory.createDocument();**

**wordDoc.open();**

**// PDF document**

**DocumentFactory pdfFactory = new PdfDocumentFactory();**

**Document pdfDoc = pdfFactory.createDocument();**

**pdfDoc.open();**

**// Excel document**

**DocumentFactory excelFactory = new ExcelDocumentFactory();**

**Document excelDoc = excelFactory.createDocument();**

**excelDoc.open();**

**}**

**}**

**public interface Document {**

**void open();**

**}**

**public class WordDocument implements Document {**

**public void open() {**

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

**}**

**}**

**public class PdfDocument implements Document {**

**public void open() {**

**System.out.println("Opening PDF Document");**

**}**

**}**

**public class ExcelDocument implements Document {**

**public void open() {**

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

**}**

**}**

**public abstract class DocumentFactory {**

**public abstract Document createDocument();**

**}**

**public class WordDocumentFactory extends DocumentFactory {**

**public Document createDocument() {**

**return new WordDocument();**

**}**

**}**

**public class PdfDocumentFactory extends DocumentFactory {**

**public Document createDocument() {**

**return new PdfDocument();**

**}**

**}**

**public class ExcelDocumentFactory extends DocumentFactory {**

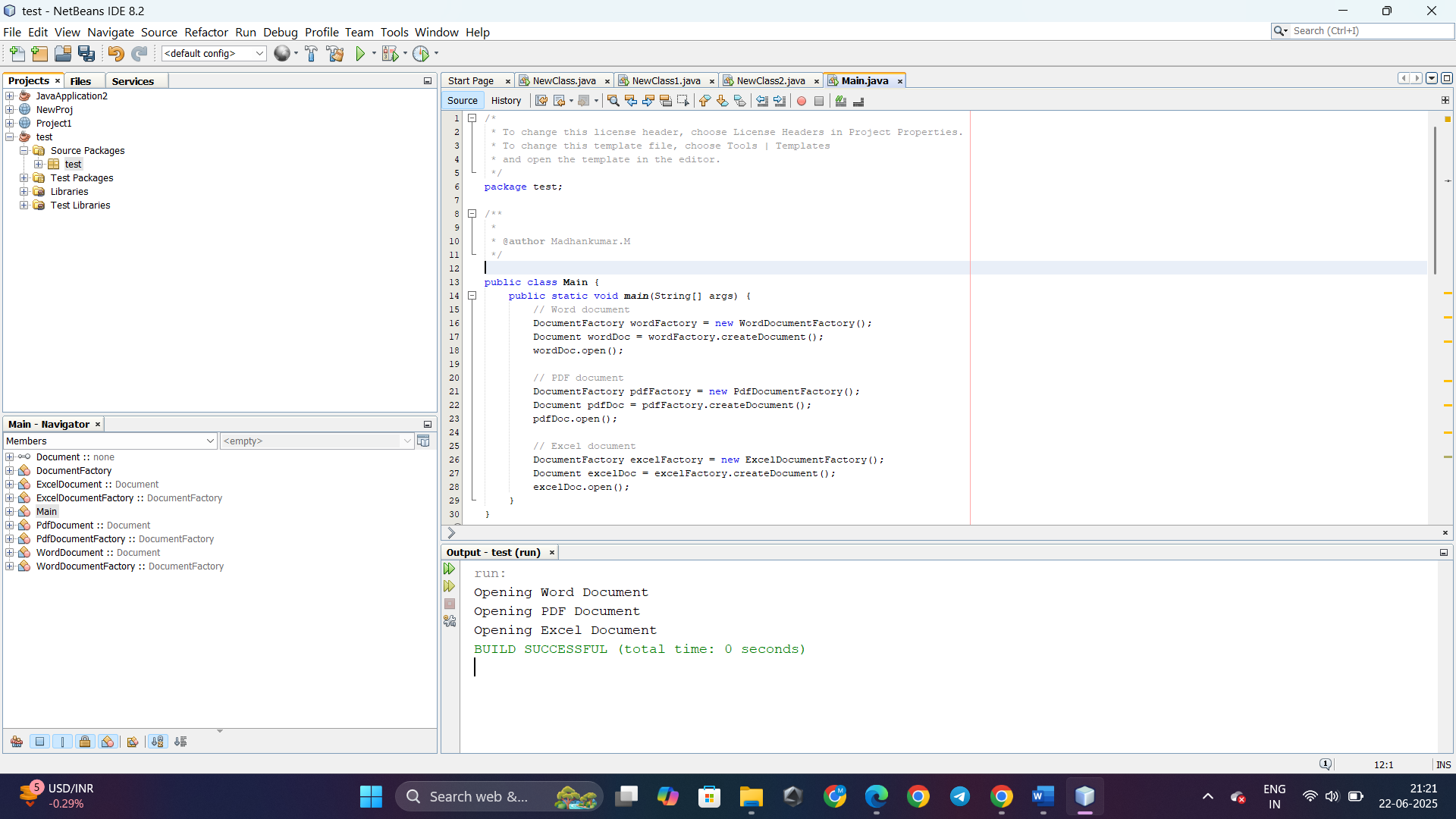
**public Document createDocument() {**

**return new ExcelDocument();**

**}**

**}**

**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.

**Solution:**

**Understanding Asymptotic Notation:**

**Big O Notation**

* Describes how the runtime or space requirement of an algorithm grows with input size.
* Focuses on the upper bound (worst-case) performance as input grows.
* O(1) - Constant time
* O(n) - Linear time
* O(log n) - Logarithmic time
* O(n log n) – Linearithmic
* O(n²) - Quadratic

**import java.util.\*;**

**public class SearchDemo {**

**public static void main(String[] args) {**

**Product[] products = {**

**new Product(1, "Laptop", "Electronics"),**

**new Product(2, "Chair", "Furniture"),**

**new Product(3, "Phone", "Electronics"),**

**new Product(4, "Pen", "Stationery"),**

**new Product(5, "Notebook", "Stationery")**

**};**

**// Linear Search**

**Product result1 = linearSearch(products, "Phone");**

**System.out.println("Linear Search: " + result1);**

**// Binary Search**

**Product result2 = binarySearch(products, "Phone");**

**System.out.println("Binary Search: " + result2);**

**}**

**}**

**public 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 productId + " - " + productName + " [" + 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) {**

**// Binary Search needs sorted data**

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

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

**while (left <= right) {**

**int mid = left + (right - left) / 2;**

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

**if (comparison == 0) {**

**return products[mid];**

**} else if (comparison < 0) {**

**left = mid + 1;**

**} else {**

**right = mid - 1;**

**}**

**}**

**return null;**

**}**

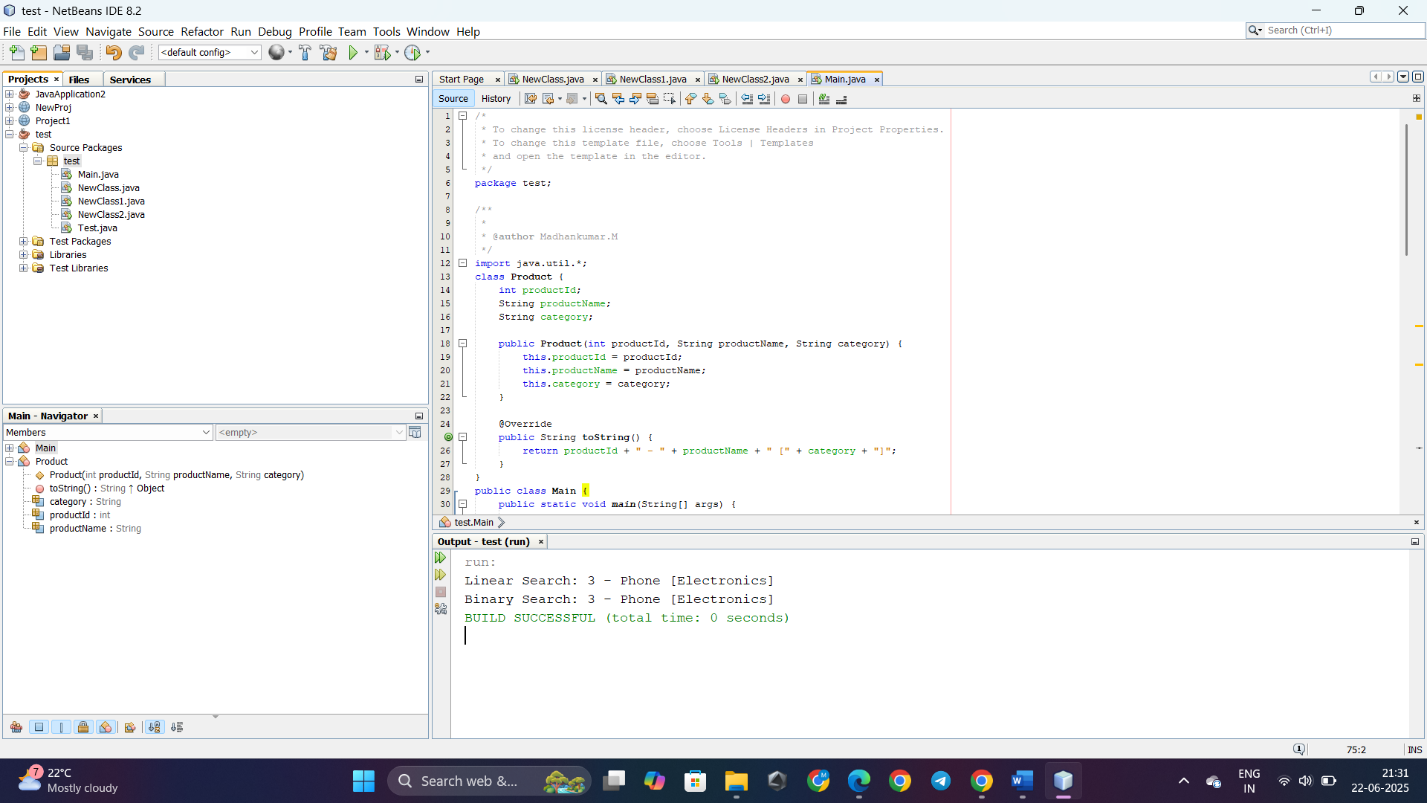
**Analysis: Linear vs Binary Search :**

* Linear Search - O(n) Slower for large datasets
* Binary Search - O(log n) Much faster for large n

**Which is better for E-commerce search?**

* Binary Search is more efficient for large datasets, especially if:
  + The product list is pre-sorted.
  + You are searching by a comparable field (e.g., productName).
* Linear Search is better if:
  + Data is unsorted and small.
  + Search is infrequent, or fields are complex (e.g., partial matches, fuzzy search).

**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.

**Solution:**

**What is Recursion?**

Recursion is a technique where a function calls itself to solve a smaller subproblem of the original.

**Why Use Recursion?**

* Simplifies problems that have **repetitive substructure** (like financial growth per year).
* More **readable** and **closer to mathematical definitions**.

**public class FinancialForecast {**

**// Recursive method to compute future value**

**public static double futureValue(double presentValue, double rate, int years) {**

**if (years == 0) {**

**return presentValue;**

**}**

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

**}**

**public static void main(String[] args) {**

**double presentValue = 10000;**

**double annualRate = 0.05; // 5%**

**int years = 10;**

**double result = futureValue(presentValue, annualRate, years);**

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

**}**

**}**

**Analysis & Optimization**

**Time Complexity:**

* The function calls itself once per year.
* So, Time Complexity = O(n) (linear time).

**Optimization (Memoization):**

In this example, since each year depends only on the previous year's value, memoization isn’t critical. But if:

* You have multiple forecasts with overlapping years,
* Or you need to forecast with variable rates each year,

Then memoization or dynamic programming will avoid recalculating the same values repeatedly.

**Output:**

