**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:**

**class Test**

**{**

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

**{**

**Logger l1=Logger.getinstance();**

**l1.getlogger("1st Logger");**

**Logger l2=Logger.getinstance();**

**l2.getlogger("2nd logger");**

**if(l1==l2)**

**System.out.println("l1 and l2 are same instance");**

**else**

**System.out.println("l1 and l2 are not same instance");**

**}**

**}**

**class Logger {**

**private static Logger instance;**

**private Logger()**

**{**

**System.out.println("Instance created!!!");**

**}**

**public static Logger getinstance()**

**{**

**if(instance==null)**

**instance=new Logger();**

**return instance;**

**}**

**public void getlogger(String msg)**

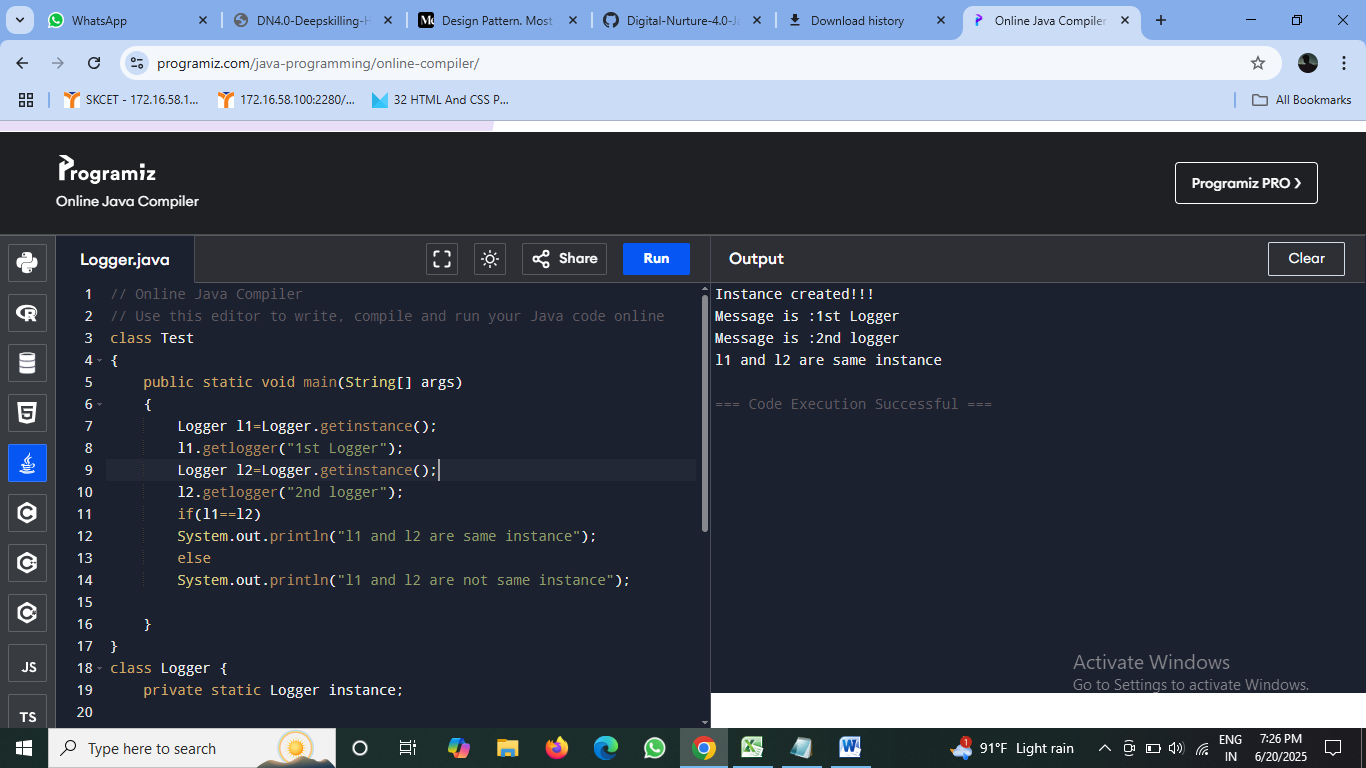
**{**

**System.out.println("Message is :"+msg);**

**}**

**}**

**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:**

**class Test {**

**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();**

**}**

**}**

**public abstract class Document {**

**public abstract void open();**

**}**

**public class WordDocument extends Document {**

**public void open() {**

**System.out.println("Word doc opens!!!");**

**}**

**}**

**public class PdfDocument extends Document {**

**public void open() {**

**System.out.println("PDF opens!!!");**

**}**

**}**

**public class ExcelDocument extends Document {**

**public void open() {**

**System.out.println("Excel doc opens!!!");**

**}**

**}**

**public abstract class DocumentFactory {**

**public abstract Document createDocument();**

**}**

**public class WordFactory extends DocumentFactory {**

**public Document createDocument() {**

**return new WordDocument();**

**}**

**}**

**public class PdfFactory extends DocumentFactory {**

**public Document createDocument() {**

**return new PdfDocument();**

**}**

**}**

**public class ExcelFactory extends DocumentFactory {**

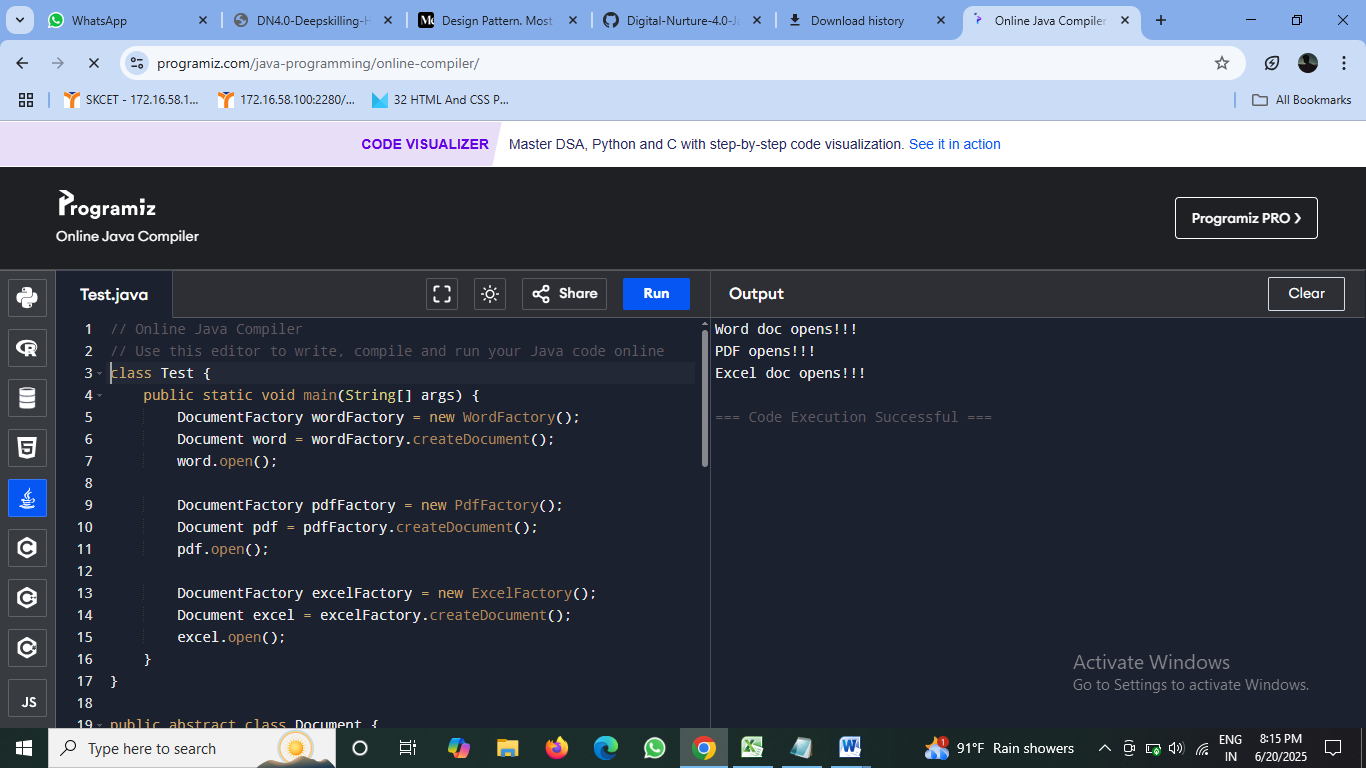
**public Document createDocument() {**

**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:**

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

**public class Search {**

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

**Product[] products = {**

**new Product(101, "Tablet", "Gadgets"),**

**new Product(102, "Sneakers", "Footwear"),**

**new Product(103, "Backpack", "Accessories"),**

**new Product(104, "Notebook", "Stationery"),**

**new Product(105, "Smartwatch", "Gadgets")**

**};**

**Product foundLinear = LinearSearch.searchByName(products, "smartwatch");**

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

**Product foundBinary = BinarySearch.searchByName(products, "Tablet");**

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

**}**

**}**

**public class Product {**

**private int productId;**

**private String productName;**

**private String category;**

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

**this.productId = productId;**

**this.productName = productName;**

**this.category = category;**

**}**

**public int getProductId() {**

**return productId;**

**}**

**public String getProductName() {**

**return productName;**

**}**

**public String getCategory() {**

**return category;**

**}**

**public String toString() {**

**return**

**"productId=" + productId +**

**", productName='" + productName + '\'' +**

**", category='" + category+"'";**

**}**

**}**

**public class LinearSearch {**

**public static Product searchByName(Product[] products, String name) {**

**for (Product product : products) {**

**if (product.getProductName().equalsIgnoreCase(name)) {**

**return product;**

**}**

**}**

**return null;**

**}**

**}**

**public class BinarySearch {**

**public static Product searchByName(Product[] products, String name) {**

**Arrays.sort(products, (a, b) -> a.getProductName().compareToIgnoreCase(b.getProductName()));**

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

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

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

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

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

**return products[mid];**

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

**left = mid + 1;**

**} else {**

**right = mid - 1;**

**}**

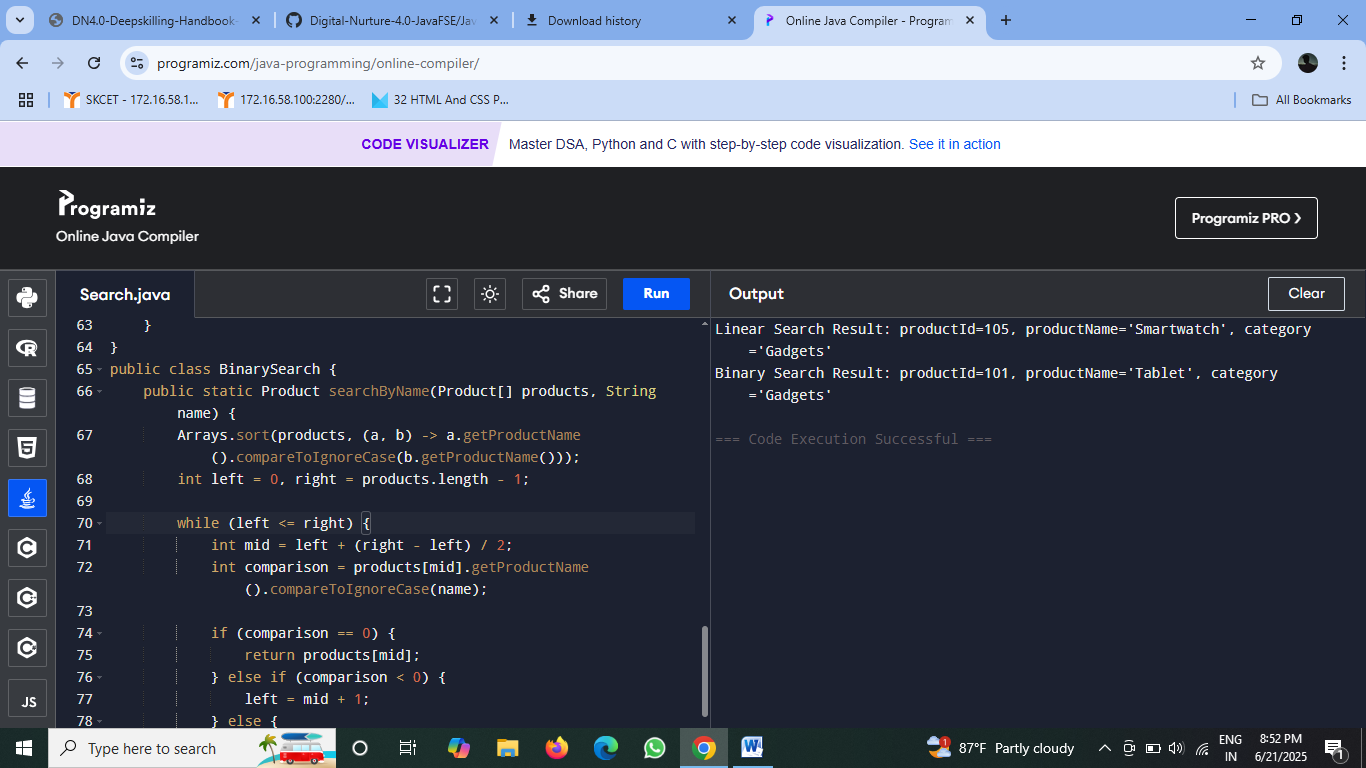
**}**

**return null;**

**}**

**}**

**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:**

**public class Financialprojection {**

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

**double initialAmount = 100;**

**double annualRate = 0.05;**

**int years = 5;**

**double projectedValue = futureValue(initialAmount, annualRate, years);**

**System.out.printf("Projected Value after %d years: Rs.%.2f%n", years, projectedValue);**

**}**

**public static double futureValue(double amount, double rate, int periods) {**

**if (periods == 0) {**

**return amount;**

**}**

**return (1 + rate) \* futureValue(amount, rate, periods - 1);**

**}**

**}**

**Output:**

