**WEEK 1**

**MODULE 1 – DESIGN PATTERNS AND PRINCIPLES**

**EXERCISE 1**

**SINGLETON PATTERN:**

In this exercise, I created a Logger class that follows the Singleton Pattern. I used a private constructor to prevent external instantiation, and a static method to return the single instance. I tested it by calling the method twice and confirming both references point to the same object. This made sure only one logger instance exists throughout the application.

**CODE:**

public class Main {

static class Logger {

private static Logger instance;

private Logger() {

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

}

public static Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

public void log(String message) {

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

}

}

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

Logger logger2 = Logger.getInstance();

logger1.log("This is the first message.");

logger2.log("This is the second message.");

if (logger1 == logger2) {

System.out.println("Both logger1 and logger2 refer to the same instance.");

} else {

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

}

}

}

**OUTPUT:**

A screenshot of a computer

AI-generated content may be incorrect.

**EXERCISE 2:**

**FACTORY METHOD PATTERN:**

For this task, I used the Factory Method Pattern to create different types of documents. I defined a Document interface and created separate classes for Word, PDF, and Excel documents. Then I created factory classes for each type, each returning its respective document. In the main method, I used the factories to create and open the documents, showing how object creation is handled by the factory.

**CODE**

interface Document {

void open();

}

class WordDocument implements Document {

public void open() {

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

}

}

class PdfDocument implements Document {

public void open() {

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

}

}

class ExcelDocument implements Document {

public void open() {

System.out.println("Opening 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 wordDoc = wordFactory.createDocument();

wordDoc.open();

DocumentFactory pdfFactory = new PdfFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open();

DocumentFactory excelFactory = new ExcelFactory();

Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

**OUTPUT:**

A screen shot of a computer

AI-generated content may be incorrect.

**MODULE - 2**

**DATA STRUCTURES AND ALGORITHM**

**EXERCISE – 2**

**E-COMMERCE PLATFORM SEARCH FUNCTION:**

In this code, I created a Product class with basic details like ID, name, and category. I then implemented both linear and binary search methods to find a product by its name. Linear search checks each item one by one, while binary search is faster but works only on sorted data, so I sorted the product list before using it. Finally, I tested both methods with sample data and printed the results to compare their output.

**CODE :**

import java.util.\*;

class Product {

int productId;

String productName;

String category;

Product(int id, String name, String cat) {

productId = id;

productName = name;

category = cat;

}

public String toString() {

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

}

}

public class Main {

public static Product linearSearch(Product[] items, String name) {

for (int i = 0; i < items.length; i++) {

if (items[i].productName.equalsIgnoreCase(name)) {

return items[i];

}

}

return null;

}

public static Product binarySearch(Product[] items, String name) {

int low = 0, high = items.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

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

if (cmp == 0) {

return items[mid];

} else if (cmp < 0) {

low = mid + 1;

} else {

high = mid - 1;

}

}

return null;

}

public static void main(String[] args) {

Product[] products = {

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

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

new Product(103, "Laptop", "Electronics"),

new Product(104, "Pen", "Stationery"),

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

};

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

Product p1 = linearSearch(products, "Phone");

if (p1 != null) {

System.out.println(p1);

} else {

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

}

Arrays.sort(products, new Comparator<Product>() {

public int compare(Product a, Product b) {

return a.productName.compareToIgnoreCase(b.productName);

}

});

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

Product p2 = binarySearch(products, "Phone");

if (p2 != null) {

System.out.println(p2);

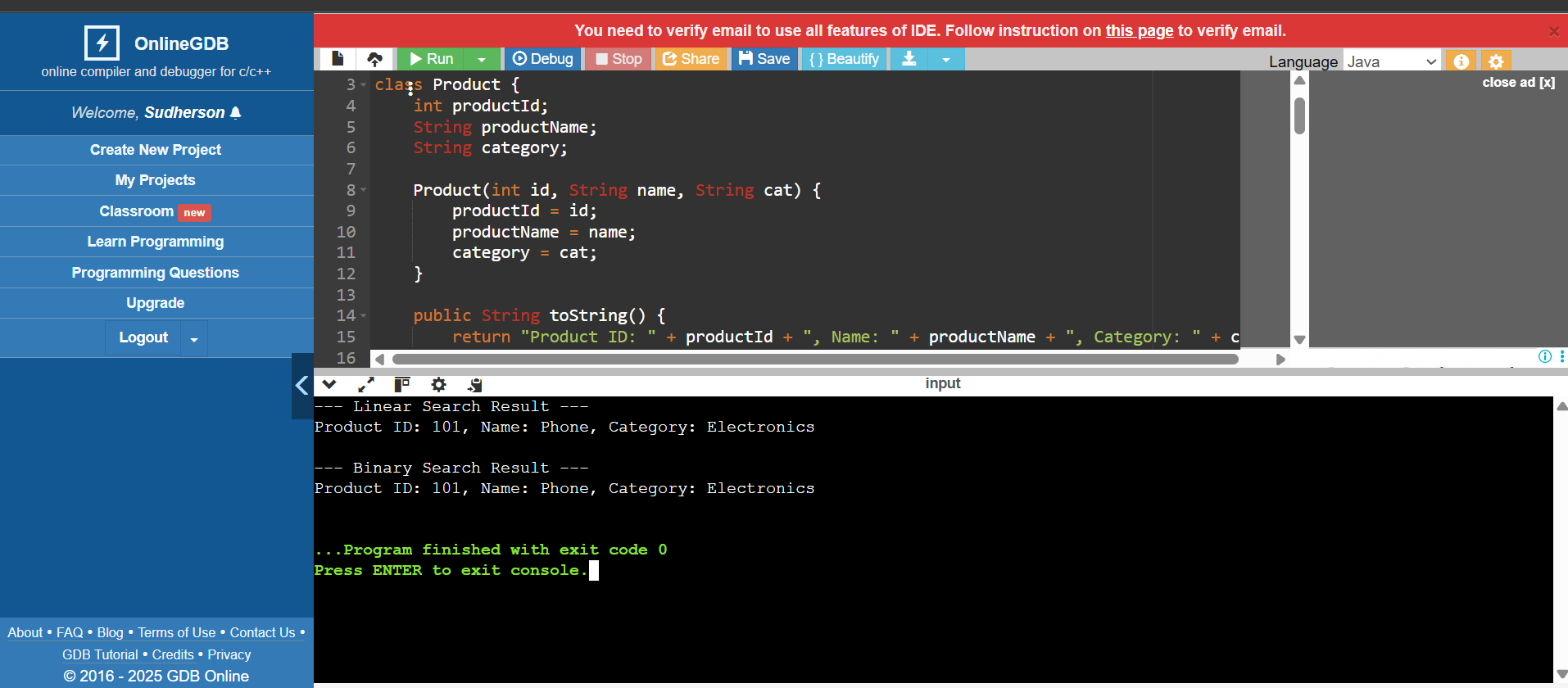
} else {

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

}

}

}

**OUTPUT:** 

**EXERCISE -7**

**FINANCIAL FORECASTING:**

In this code, I used recursion to predict the future value of an investment based on a fixed annual growth rate. I created a function that multiplies the current value by the growth rate every year and calls itself for each remaining year until it reaches zero. This approach shows how recursion can be used for repetitive calculations like forecasting.

**CODE:**

public class Main {

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

if (years == 0) {

return currentValue;

}

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

}

public static void main(String[] args) {

double initialValue = 1000;

double growthRate = 0.10;

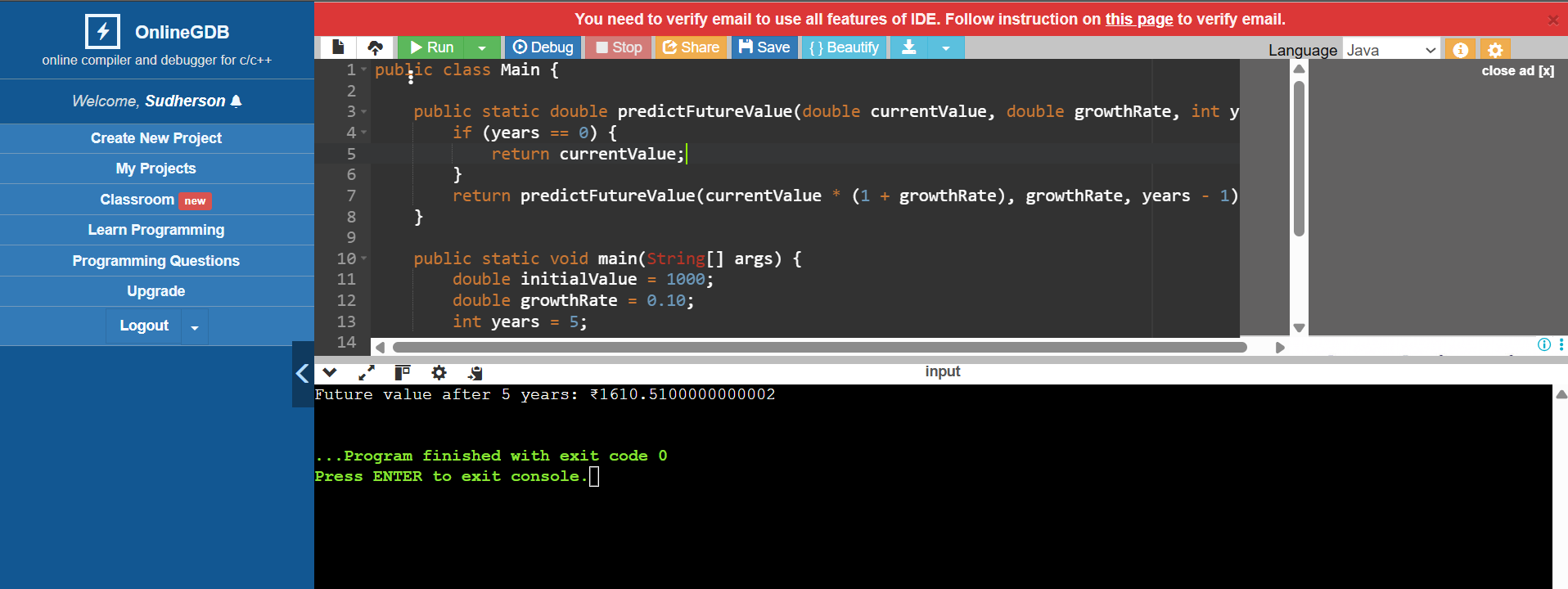
int years = 5;

double futureValue = predictFutureValue(initialValue, growthRate, years);

System.out.println("Future value after " + years + " years: ₹" + futureValue);

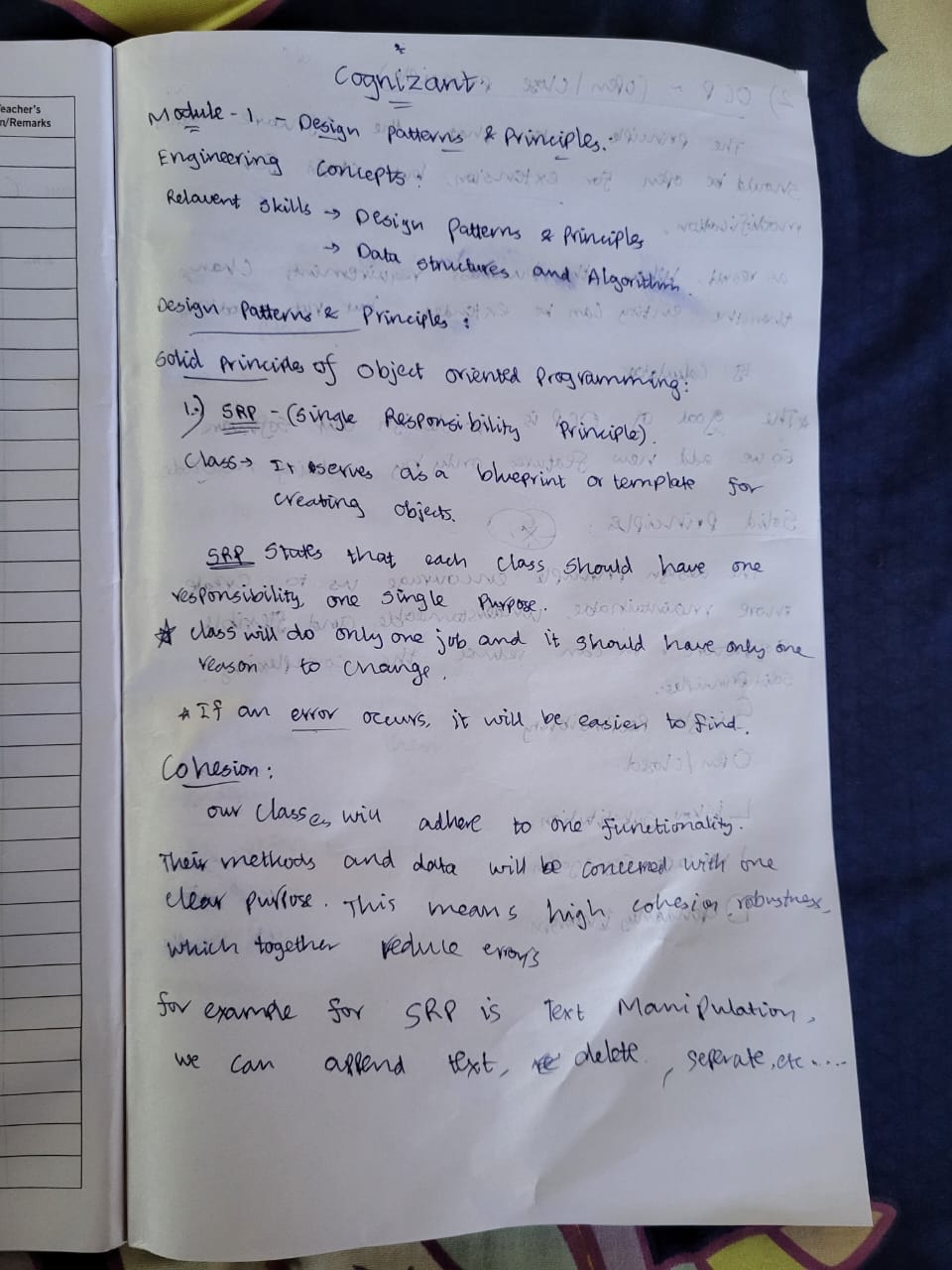
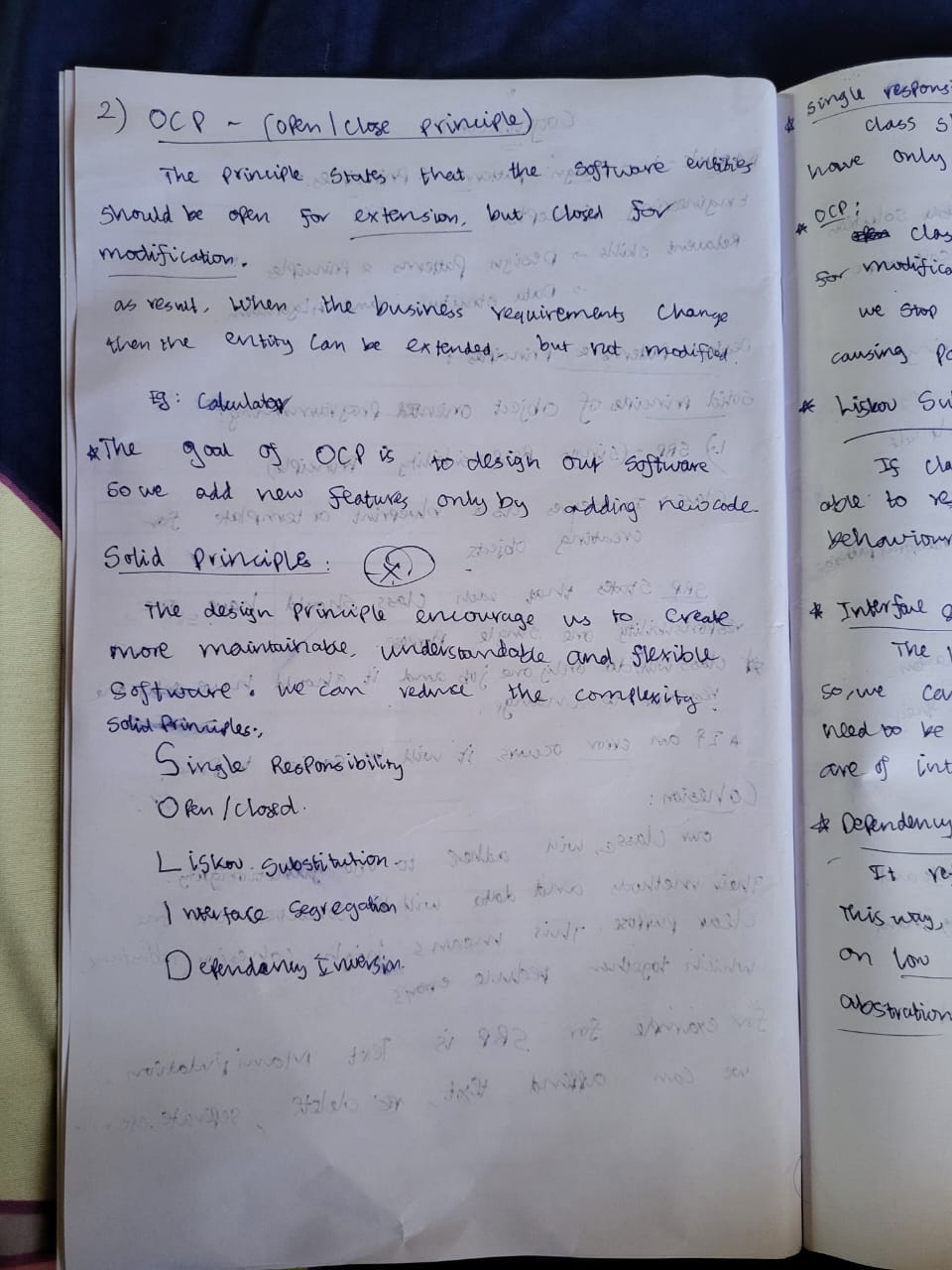
}

}

**OUTPUT:** ****

**WHAT I HAVE LEARNT:**

I have added my own notes summarizing the key points I understood**.**

  A notebook with writing on it

AI-generated content may be incorrect. A piece of paper with writing on it

AI-generated content may be incorrect. A notebook with writing on it

AI-generated content may be incorrect. A piece of paper with writing on it

AI-generated content may be incorrect. A notebook with writing on it

AI-generated content may be incorrect.