**Name:Anisha M**

**Superset ID:6371860**

**WEEK-1 HANDS-ON SOLUTION**

**Design patterns and principles**

**Exercise 1: Implementing the Singleton Pattern**

**Code:**

class Logger {

private static Logger instance;

private Logger() {

System.out.println("Logger instance created");

}

public static Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

public void log(String message) {

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

}

}

public class Main {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

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

Logger logger2 = Logger.getInstance();

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

if (logger1 == logger2) {

System.out.println("Only one Logger instance exists.");

} else {

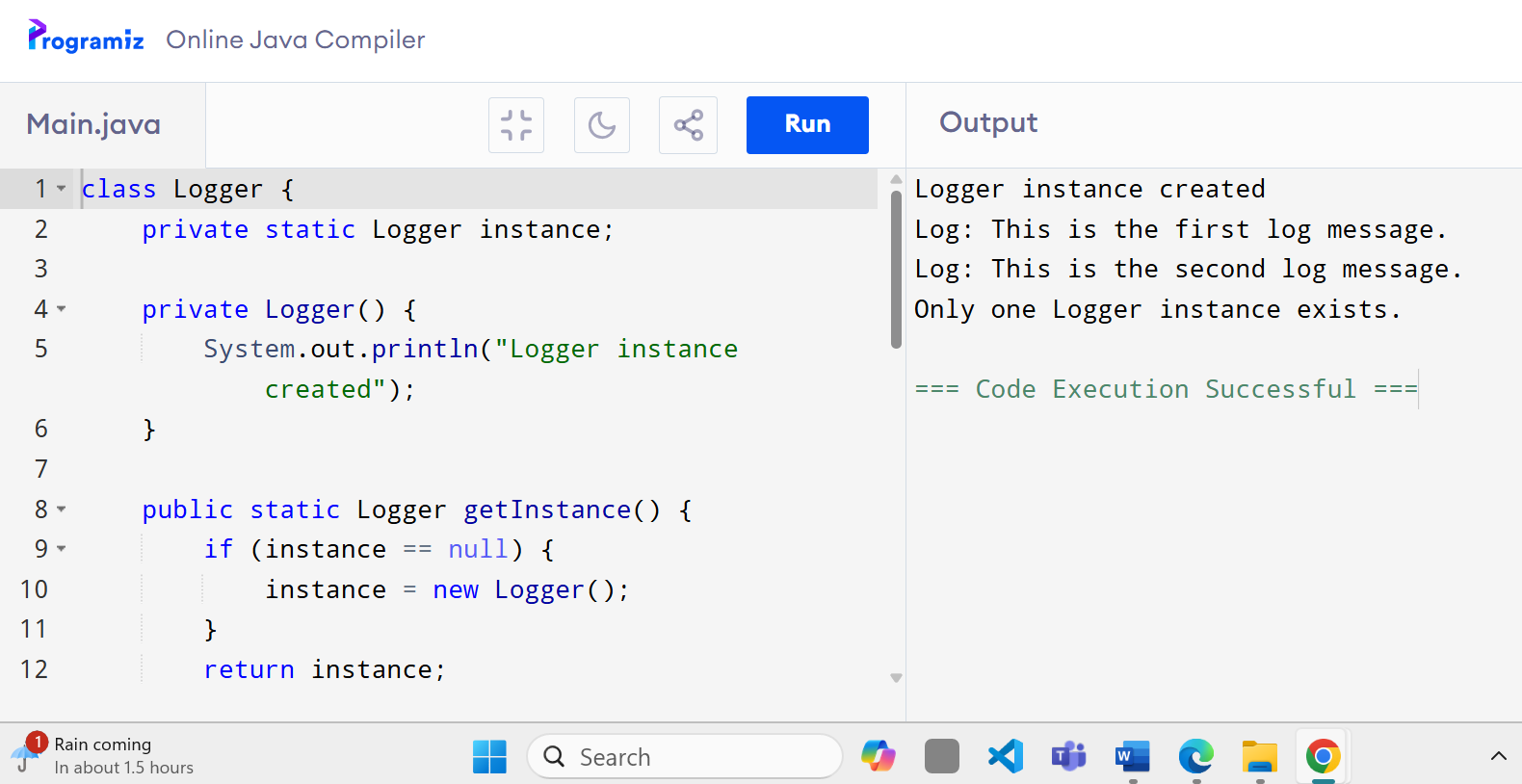
System.out.println("Different Logger instances exist.");

}

}

}

**Output:**

****

**Exercise 2: Implementing the Factory Method Pattern**

**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 WordDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

class PdfDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

class ExcelDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

public class Main {

public static void main(String[] args) {

DocumentFactory wordFactory = new WordDocumentFactory();

Document wordDoc = wordFactory.createDocument();

wordDoc.open();

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open();

DocumentFactory excelFactory = new ExcelDocumentFactory();

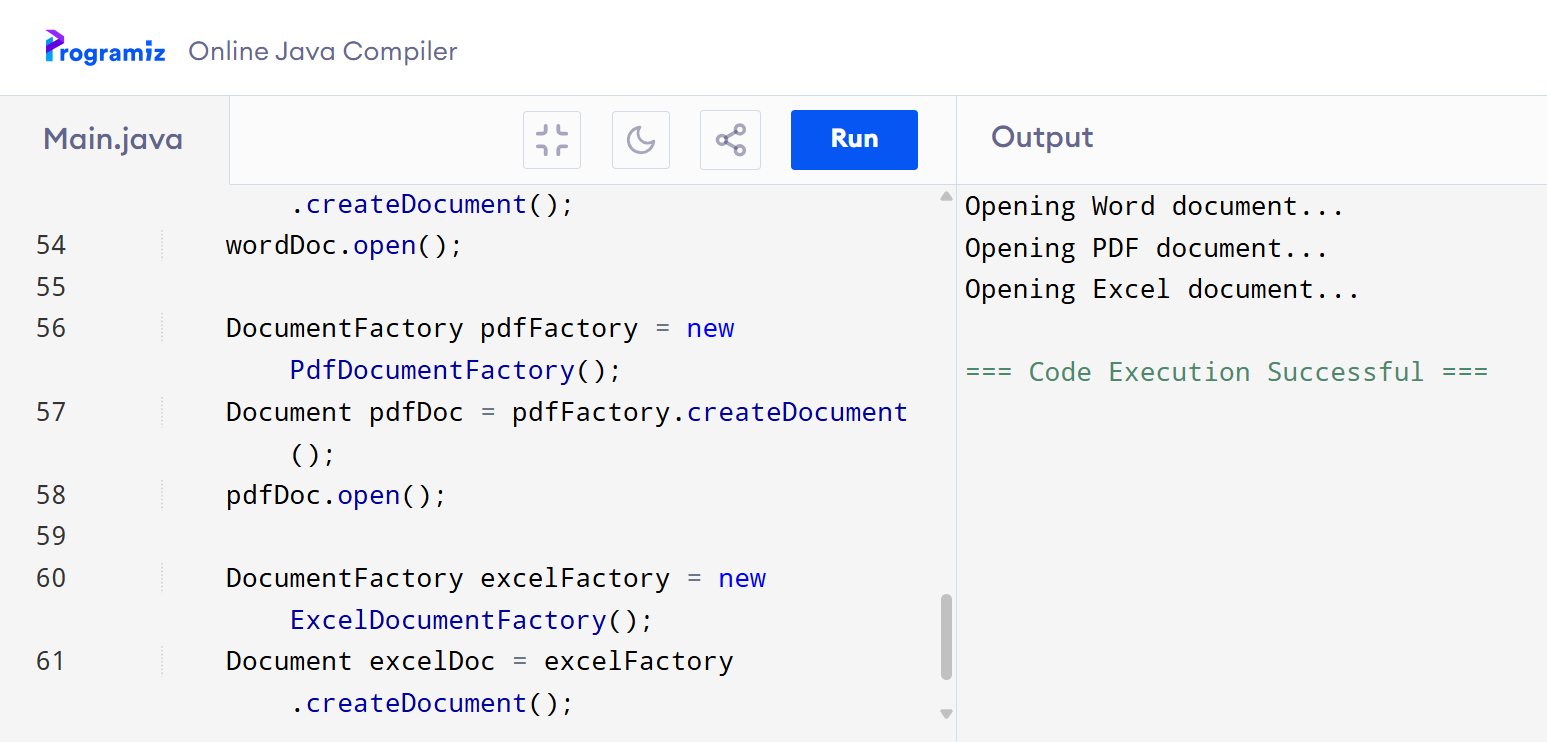
Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

**Output:**



**Exercise 3: Implementing the Builder Pattern**

**Code:**

static class Computer {

private String CPU;

private String RAM;

private String storage;

private Computer(Builder builder) {

this.CPU = builder.CPU;

this.RAM = builder.RAM;

this.storage = builder.storage;

}

static class Builder {

private String CPU;

private String RAM;

private String storage;

public Builder setCPU(String CPU) {

this.CPU = CPU;

return this;

}

public Builder setRAM(String RAM) {

this.RAM = RAM;

return this;

}

public Builder setStorage(String storage) {

this.storage = storage;

return this;

}

public Computer build() {

return new Computer(this);

}

}

public void showConfig() {

System.out.println("Computer Configuration:");

System.out.println("CPU: " + CPU);

System.out.println("RAM: " + RAM);

System.out.println("Storage: " + storage);

}

}

public static void main(String[] args) {

Computer gamingPC = new Computer.Builder()

.setCPU("Intel i9")

.setRAM("32GB")

.setStorage("1TB SSD")

.build();

Computer officePC = new Computer.Builder()

.setCPU("Intel i5")

.setRAM("8GB")

.setStorage("512GB SSD")

.build();

gamingPC.showConfig();

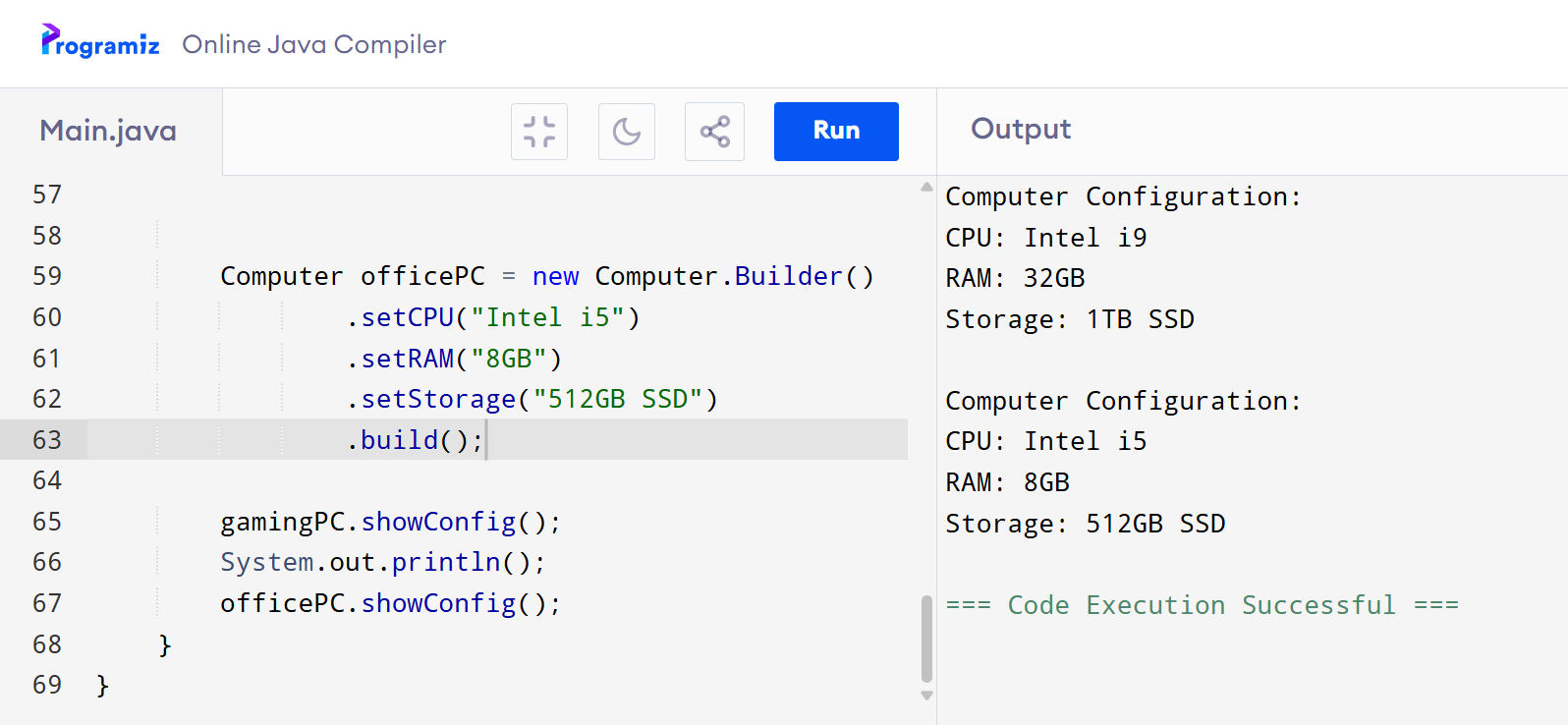
System.out.println();

officePC.showConfig();

}

}

**Output:**

****

**Exercise 4: Implementing the Adapter Pattern**

**Code:**

public class Main {

interface PaymentProcessor {

void processPayment(double amount);

}

static class PayPalGateway {

public void makePayment(double amount) {

System.out.println("Processing payment through PayPal: ₹" + amount);

}

}

static class StripeGateway {

public void doPayment(double amount) {

System.out.println("Processing payment through Stripe: ₹" + amount);

}

}

static class PayPalAdapter implements PaymentProcessor {

private PayPalGateway paypal;

public PayPalAdapter(PayPalGateway paypal) {

this.paypal = paypal;

}

public void processPayment(double amount) {

paypal.makePayment(amount);

}

}

static class StripeAdapter implements PaymentProcessor {

private StripeGateway stripe;

public StripeAdapter(StripeGateway stripe) {

this.stripe = stripe;

}

public void processPayment(double amount) {

stripe.doPayment(amount);

}

}

public static void main(String[] args) {

PaymentProcessor paypalProcessor = new PayPalAdapter(new PayPalGateway());

paypalProcessor.processPayment(2500);

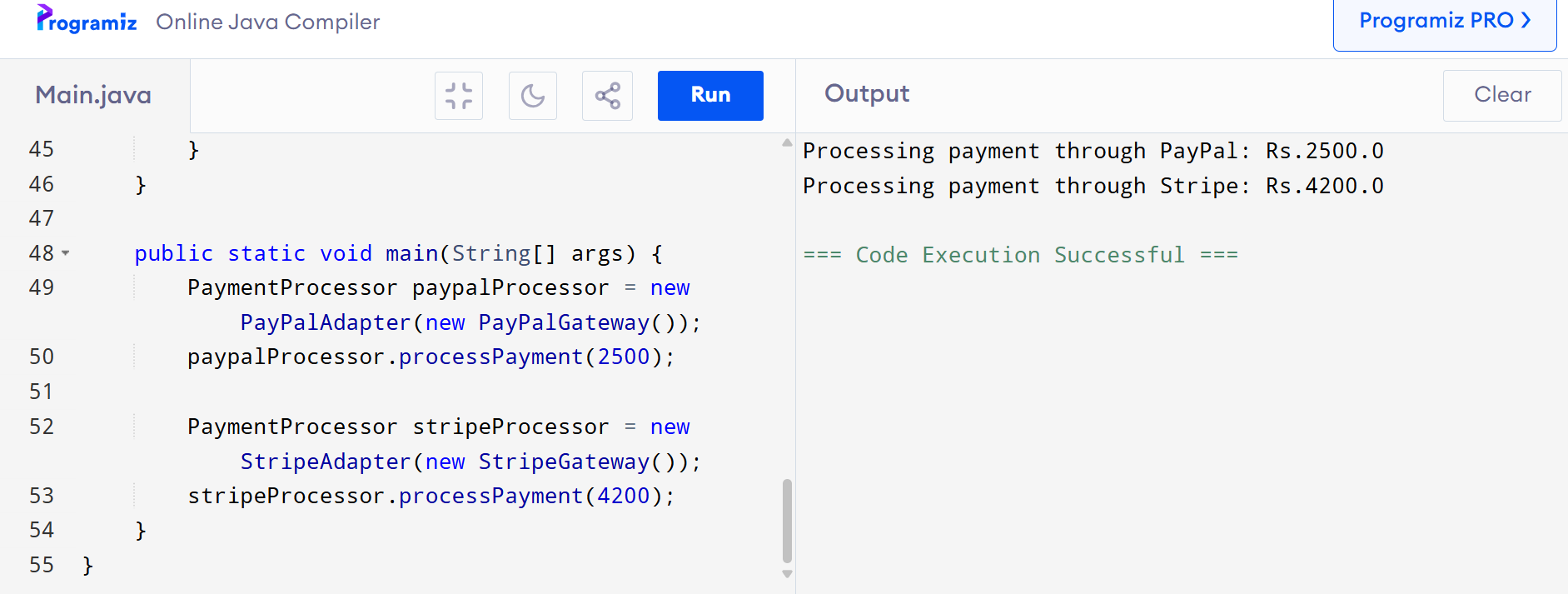
PaymentProcessor stripeProcessor = new StripeAdapter(new StripeGateway());

stripeProcessor.processPayment(4200);

}

}

**Output:**



**Exercise 5: Implementing the Decorator Pattern**

**Code:**

interface Notifier {

void send();

}

class EmailNotifier implements Notifier {

public void send() {

System.out.println("Sending Email Notification");

}

}

abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

public void send() {

notifier.send();

}

}

class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send() {

super.send();

System.out.println("Sending SMS Notification");

}

}

class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send() {

super.send();

System.out.println("Sending Slack Notification");

}

}

public class Main {

public static void main(String[] args) {

Notifier notifier = new EmailNotifier();

notifier = new SMSNotifierDecorator(notifier);

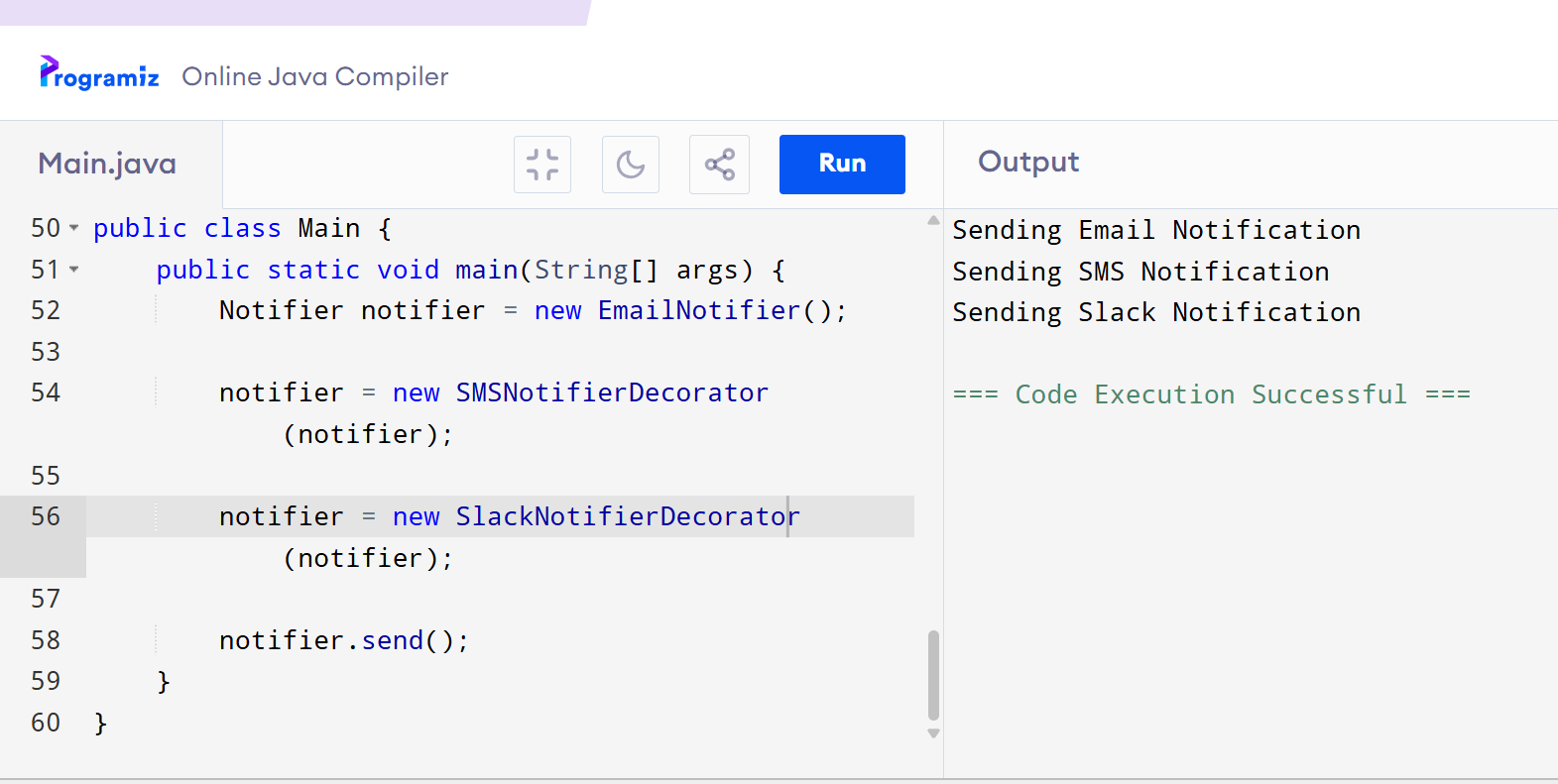
notifier = new SlackNotifierDecorator(notifier);

notifier.send();

}

}

**Output:**

****

**Exercise 6: Implementing the Proxy Pattern**

**Code:**

interface Image {

void display();

}

class RealImage implements Image {

private String filename;

public RealImage(String filename) {

this.filename = filename;

loadFromDisk();

}

private void loadFromDisk() {

System.out.println("Loading " + filename + " from disk...");

}

public void display() {

System.out.println("Displaying " + filename);

}

}

class ProxyImage implements Image {

private RealImage realImage;

private String filename;

public ProxyImage(String filename) {

this.filename = filename;

}

public void display() {

if (realImage == null) {

realImage = new RealImage(filename);

}

realImage.display();

}

}

public class Main {

public static void main(String[] args) {

Image image1 = new ProxyImage("image1.jpg");

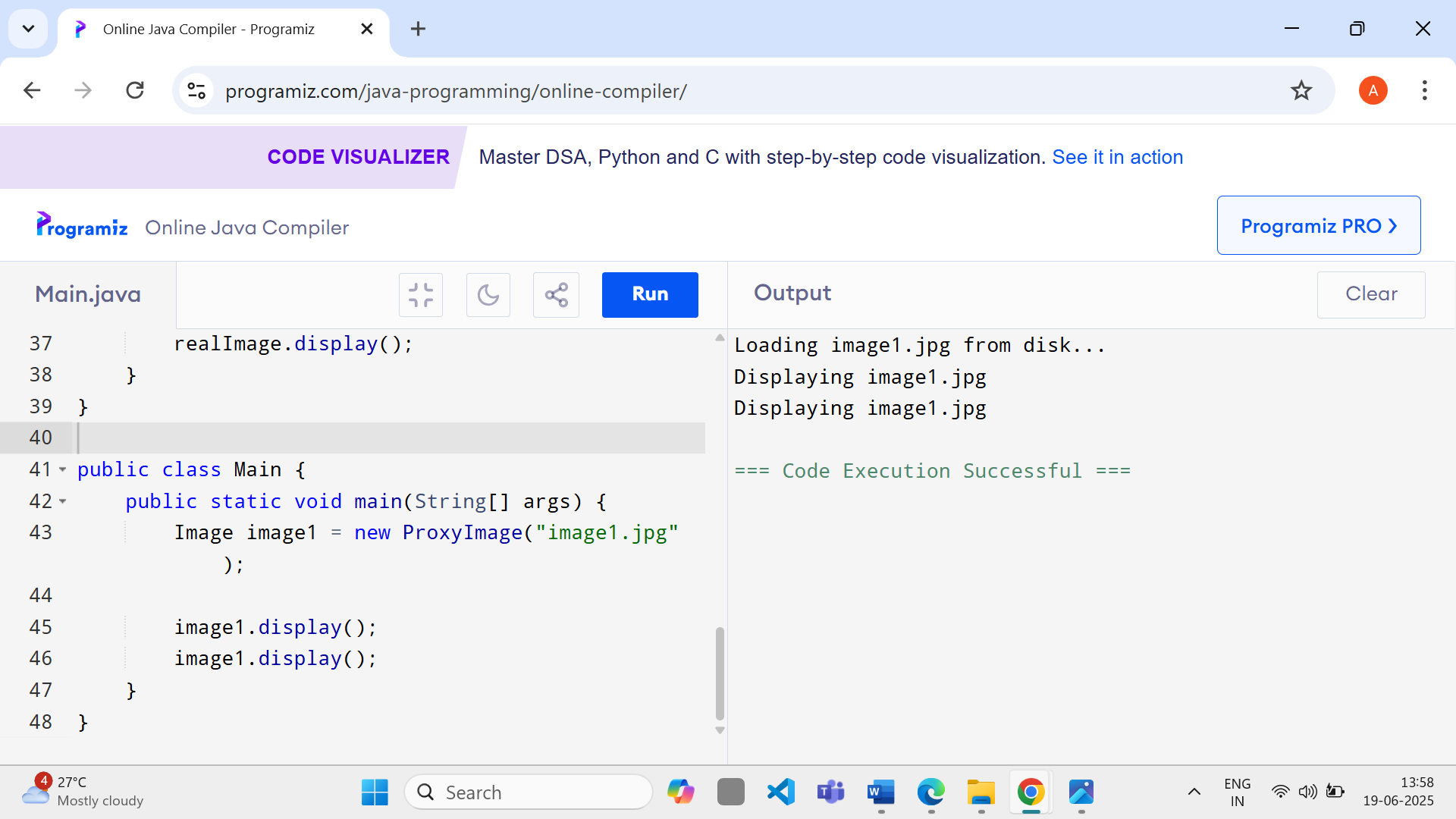
image1.display();

image1.display();

}

}

**Output:**

****

**Exercise 7: Implementing the Observer Pattern**

**Code:**

interface Stock {

void register(Observer o);

void deregister(Observer o);

void notifyObservers();

}

interface Observer {

void update(float price);

}

import java.util.ArrayList;

import java.util.List;

class StockMarket implements Stock {

private List<Observer> observers = new ArrayList<>();

private float stockPrice;

public void setPrice(float price) {

this.stockPrice = price;

notifyObservers();

}

public void register(Observer o) {

observers.add(o);

}

public void deregister(Observer o) {

observers.remove(o);

}

public void notifyObservers() {

for (Observer o : observers) {

o.update(stockPrice);

}

}

}

class MobileApp implements Observer {

public void update(float price) {

System.out.println("Mobile App - Stock Price Updated: Rs" + price);

}

}

class WebApp implements Observer {

public void update(float price) {

System.out.println("Web App - Stock Price Updated: Rs" + price);

}

}

public class Main {

public static void main(String[] args) {

StockMarket market = new StockMarket();

Observer mobile = new MobileApp();

Observer web = new WebApp();

market.register(mobile);

market.register(web);

market.setPrice(99.5f);

market.setPrice(102.75f);

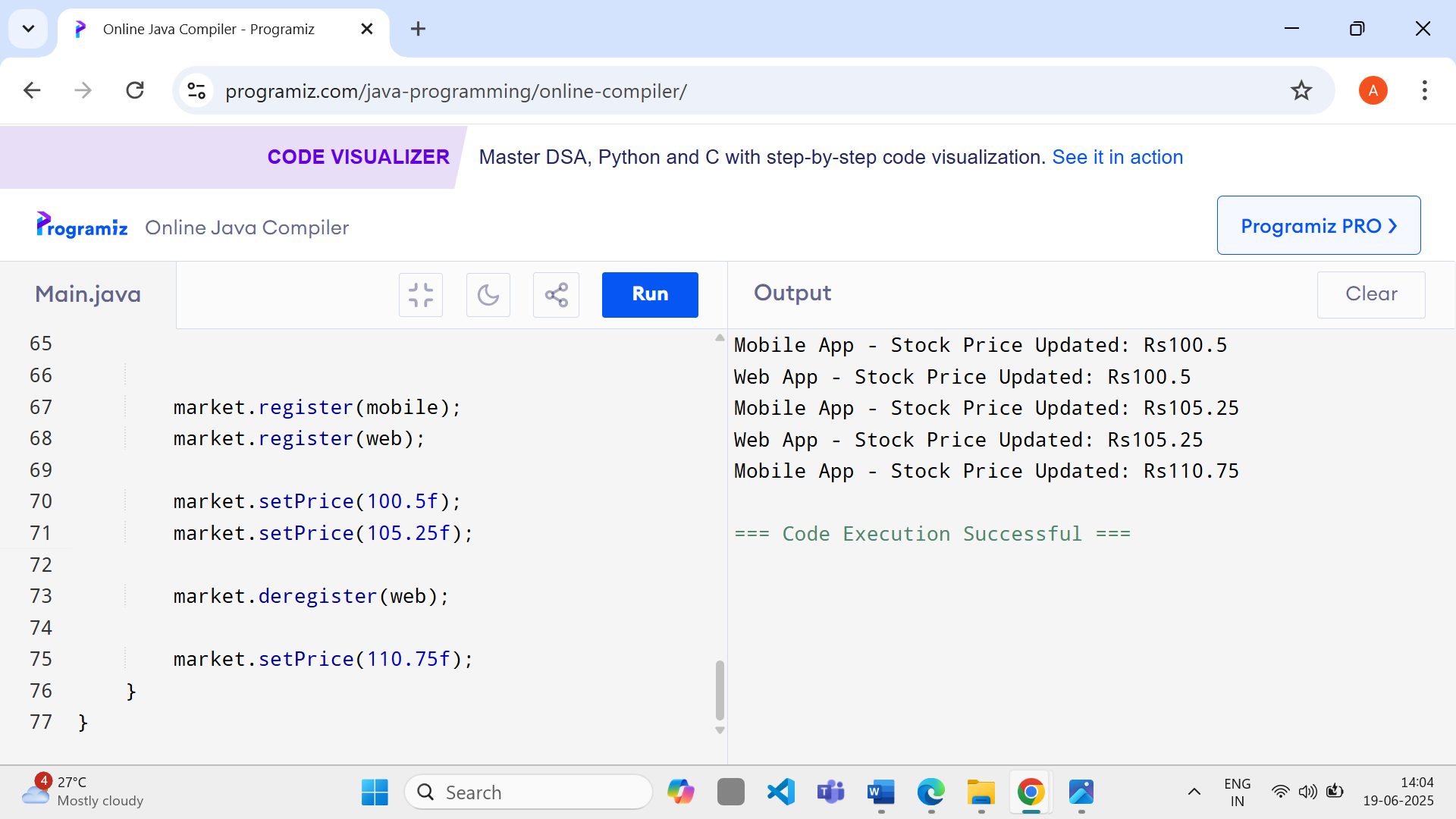
market.deregister(web);

market.setPrice(105.0f);

}

}

**Output:**

****

**Exercise 8: Implementing the Strategy Pattern**

**Code:**

interface PaymentStrategy {

void pay(double amount);

}

class CreditCardPayment implements PaymentStrategy {

public void pay(double amount) {

System.out.println("Paid " + amount + " using Credit Card.");

}

}

class PayPalPayment implements PaymentStrategy {

public void pay(double amount) {

System.out.println("Paid " + amount + " using PayPal.");

}

}

class PaymentContext {

private PaymentStrategy strategy;

public void setPaymentStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void pay(double amount) {

if (strategy == null) {

System.out.println("Payment strategy not set.");

return;

}

strategy.pay(amount);

}

}

public class Main {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

context.setPaymentStrategy(new CreditCardPayment());

context.pay($100.0);

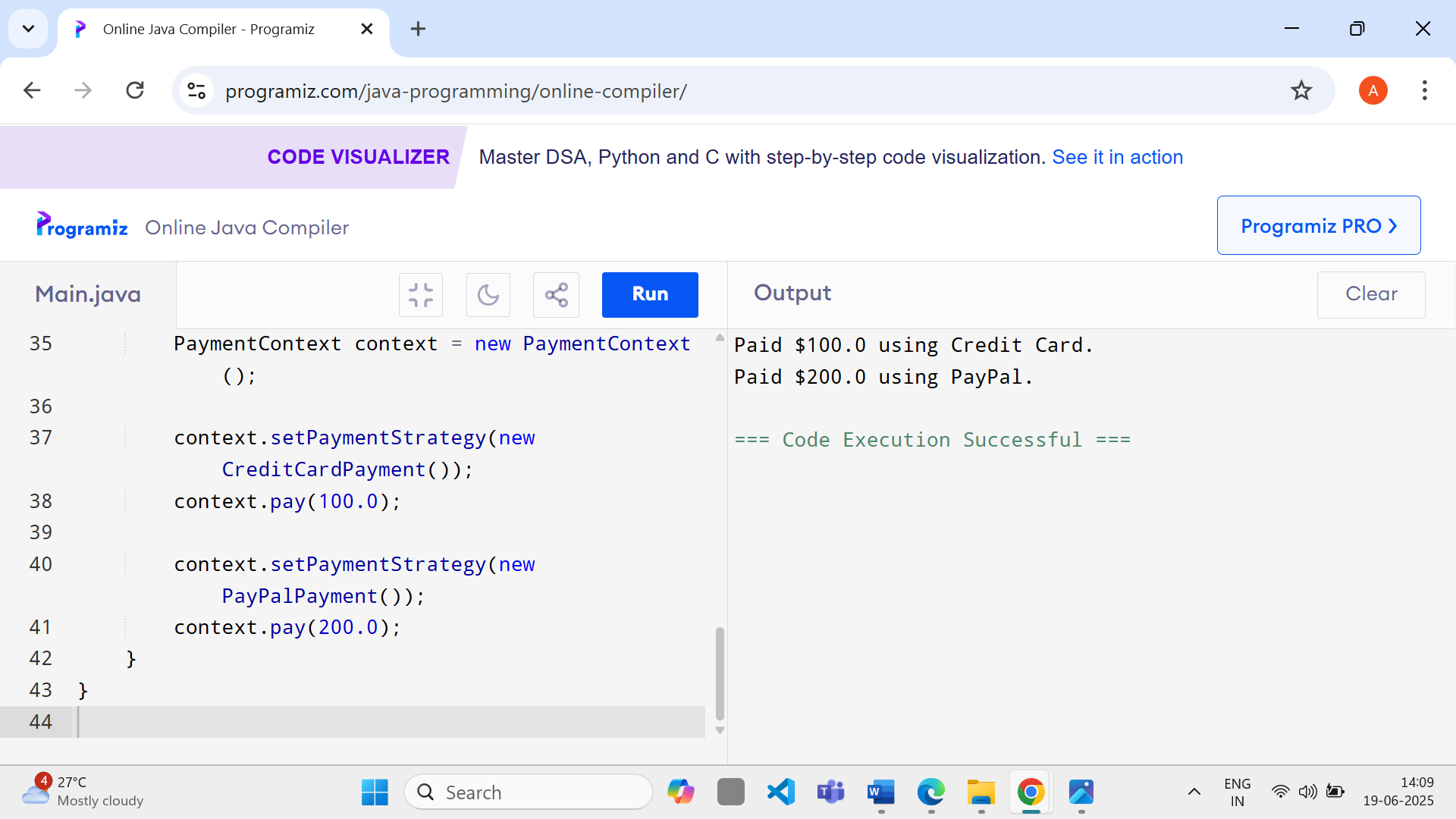
context.setPaymentStrategy(new PayPalPayment());

context.pay($200.0);

}

}

**Output:**

****

**Exercise: Implementing the Command Pattern**

**Code:**

interface Command {

void execute();

}

class Light {

public void turnOn() {

System.out.println("The light is ON.");

}

public void turnOff() {

System.out.println("The light is OFF.");

}

}

class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOn();

}

}

class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOff();

}

}

class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

if (command != null) {

command.execute();

}

}

}

public class Main {

public static void main(String[] args) {

Light light = new Light();

Command onCommand = new LightOnCommand(light);

Command offCommand = new LightOffCommand(light);

RemoteControl remote = new RemoteControl();

remote.setCommand(onCommand);

remote.pressButton();

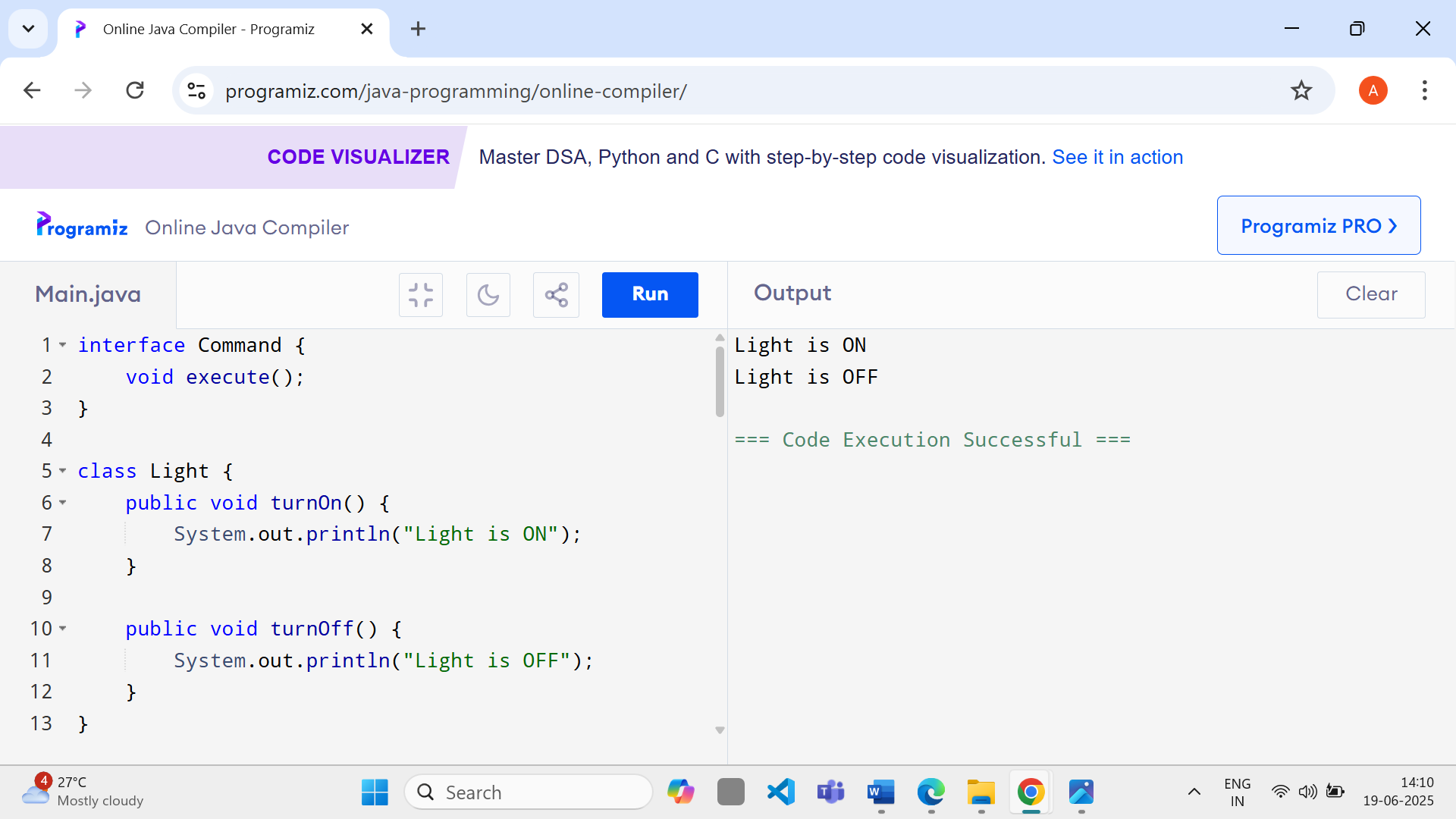
remote.setCommand(offCommand);

remote.pressButton();

}

}

**Output:**

****

**Exercise 10: Implementing the MVC Pattern**

**Code:**

class Student {

private String name;

private String id;

private String grade;

public Student(String name, String id, String grade) {

this.name = name;

this.id = id;

this.grade = grade;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getId() {

return id;

}

public String getGrade() {

return grade;

}

public void setGrade(String grade) {

this.grade = grade;

}

}

class StudentView {

public void displayStudentDetails(String name, String id, String grade) {

System.out.println("Student Details:");

System.out.println("Name: " + name);

System.out.println("ID: " + id);

System.out.println("Grade: " + grade);

}

}

class StudentController {

private Student model;

private StudentView view;

public StudentController(Student model, StudentView view) {

this.model = model;

this.view = view;

}

public void setStudentName(String name) {

model.setName(name);

}

public void setStudentGrade(String grade) {

model.setGrade(grade);

}

public void updateView() {

view.displayStudentDetails(model.getName(), model.getId(), model.getGrade());

}

}

public class Main {

public static void main(String[] args) {

Student model = new Student("John Doe", "101", "A");

StudentView view = new StudentView();

StudentController controller = new StudentController(model, view);

controller.updateView();

controller.setStudentName("Jane Smith");

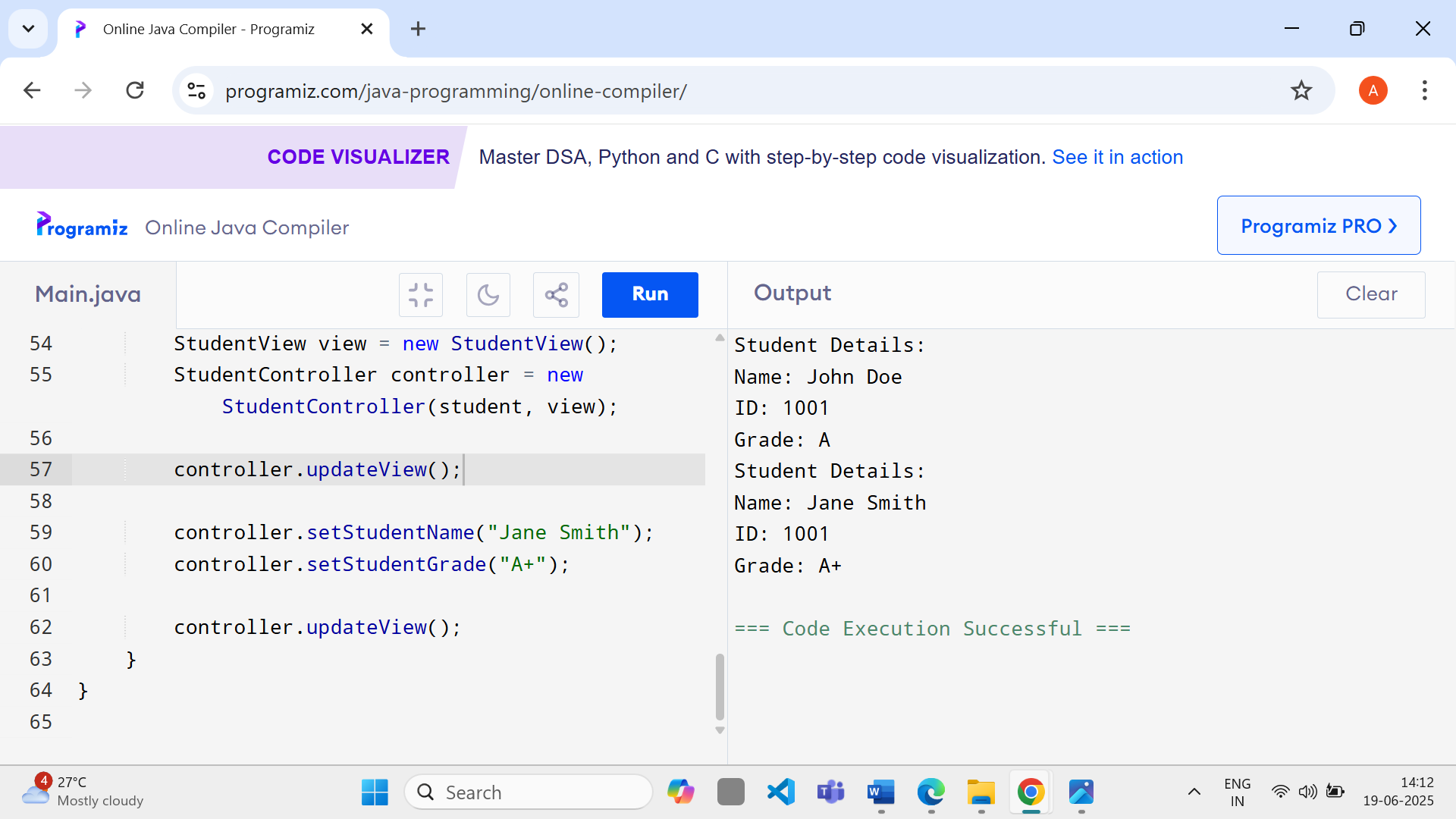
controller.setStudentGrade("A+");

controller.updateView();

}

}

**Output:**



**Exercise 11: Implementing Dependency Injection**

**Code:**

interface CustomerRepository {

Customer findCustomerById(String id);

}

class Customer {

private String id;

private String name;

public Customer(String id, String name) {

this.id = id;

this.name = name;

}

public String toString() {

return "Customer[ID=" + id + ", Name=" + name + "]";

}

}

class CustomerRepositoryImpl implements CustomerRepository {

public Customer findCustomerById(String id) {

return new Customer(id, "John Doe");

}

}

class CustomerService {

private CustomerRepository repository;

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void getCustomerById(String id) {

Customer customer = repository.findCustomerById(id);

System.out.println("Found: " + customer.toString());

}

}

public class Main {

public static void main(String[] args) {

CustomerRepository repository = new CustomerRepositoryImpl();

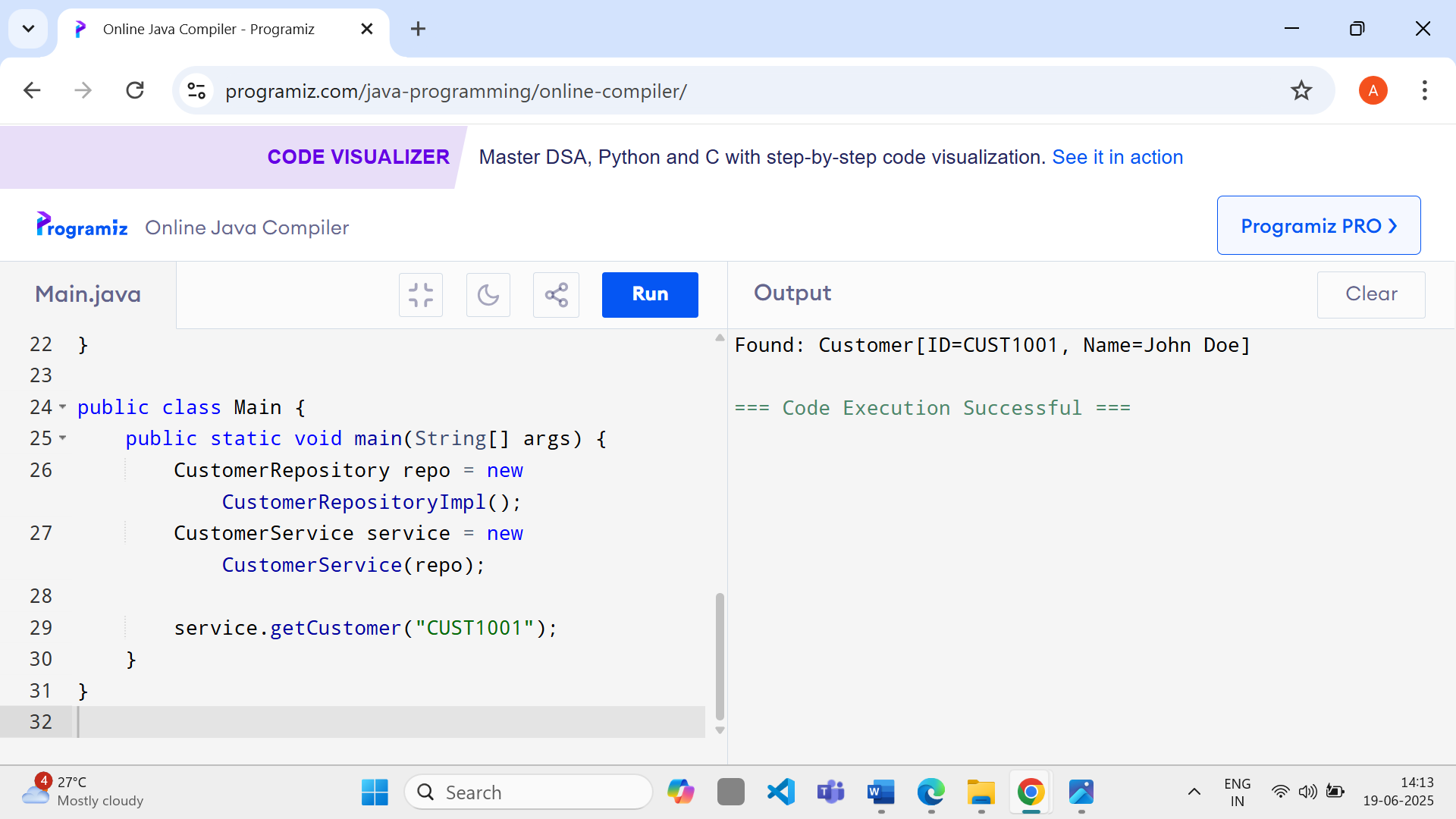
CustomerService service = new CustomerService(repository);

service.getCustomerById("CUST1001");

}

}

**Output:**

****

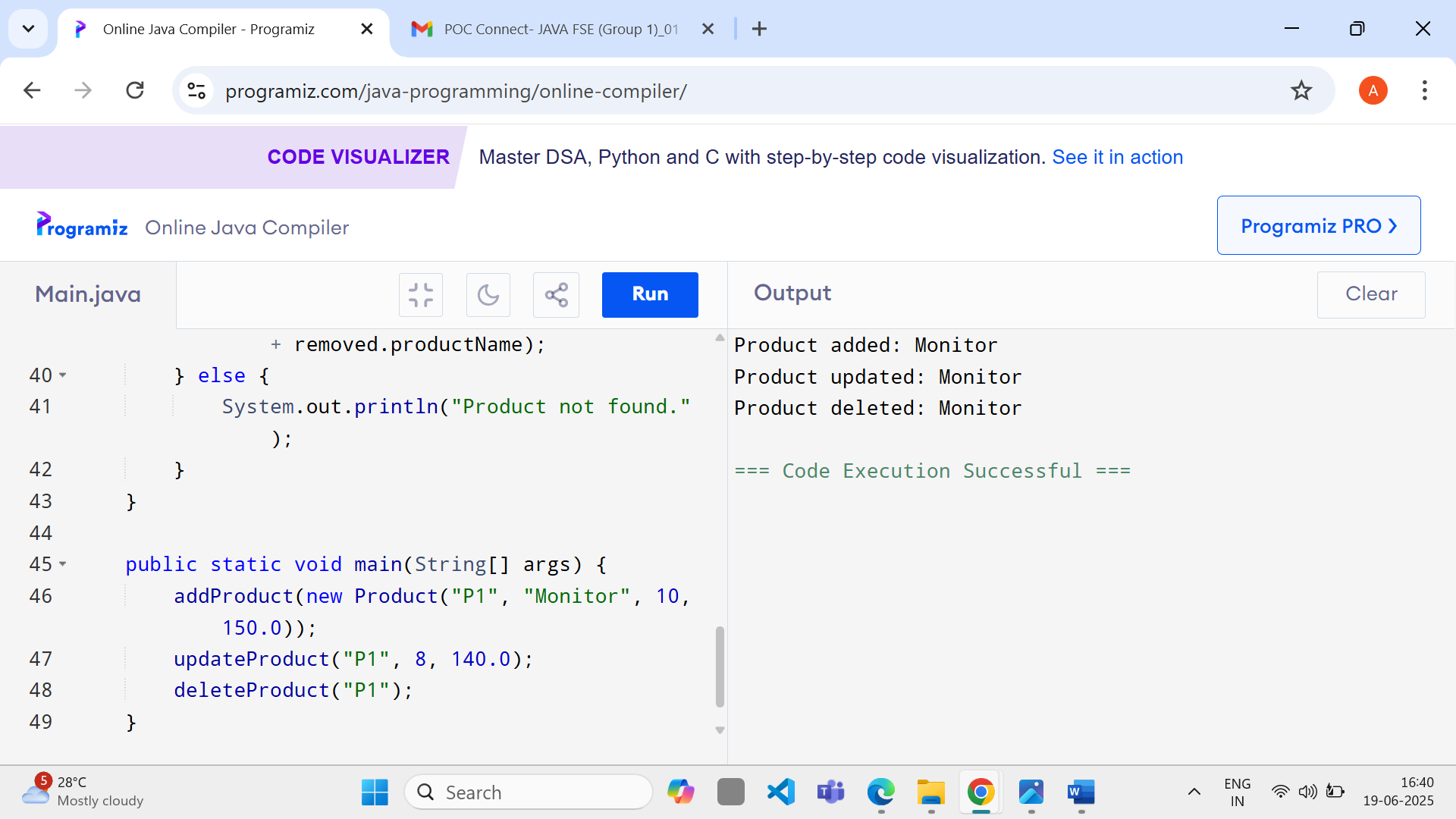
**Algorithms\_Data Structures**

**Exercise 1: Inventory Management System**

**Code:**

import java.util.\*;  
  
 class Product {  
 String productId, productName;  
 int quantity;  
 double price;  
  
 Product(String id, String name, int qty, double price) {  
 this.productId = id;  
 this.productName = name;  
 this.quantity = qty;  
 this.price = price;  
 }  
 }  
  
 public class Main {  
 static Map<String, Product> inventory = new HashMap<>();  
  
 public static void addProduct(Product p) {  
 inventory.put(p.productId, p);  
 System.out.println("Product added: " + p.productName);  
 }  
  
 public static void updateProduct(String id, int qty, double price) {  
 Product p = inventory.get(id);  
 if (p != null) {  
 p.quantity = qty;  
 p.price = price;  
 System.out.println("Product updated: " + p.productName);  
 }  
 }  
  
 public static void deleteProduct(String id) {  
 Product removed = inventory.remove(id);  
 if (removed != null) {  
 System.out.println("Product deleted: " + removed.productName);  
 } else {  
 System.out.println("Product not found.");  
 }  
 }  
  
 public static void main(String[] args) {  
 addProduct(new Product("P1", "Monitor", 10, 150.0));  
 updateProduct("P1", 8, 140.0);  
 deleteProduct("P1");  
 }  
 }

**Output :**



## Exercise 2: E-commerce Platform Search Function

**Code:**

import java.util.Arrays;

class Product {

String productId, productName, category;

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

this.productId = id;

this.productName = name;

this.category = cat;

}

}

public class Main {

static Product[] products = {

new Product("P1", "Mobile", "Electronics"),

new Product("P2", "Laptop", "Electronics"),

new Product("P3", "Tablet", "Electronics")

};

public static int linearSearch(String name) {

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

if (products[i].productName.equalsIgnoreCase(name)) return i;

}

return -1;

}

public static int binarySearch(String name) {

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

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

while (left <= right) {

int mid = (left + right) / 2;

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

if (cmp == 0) return mid;

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

else right = mid - 1;

}

return -1;

}

public static void main(String[] args) {

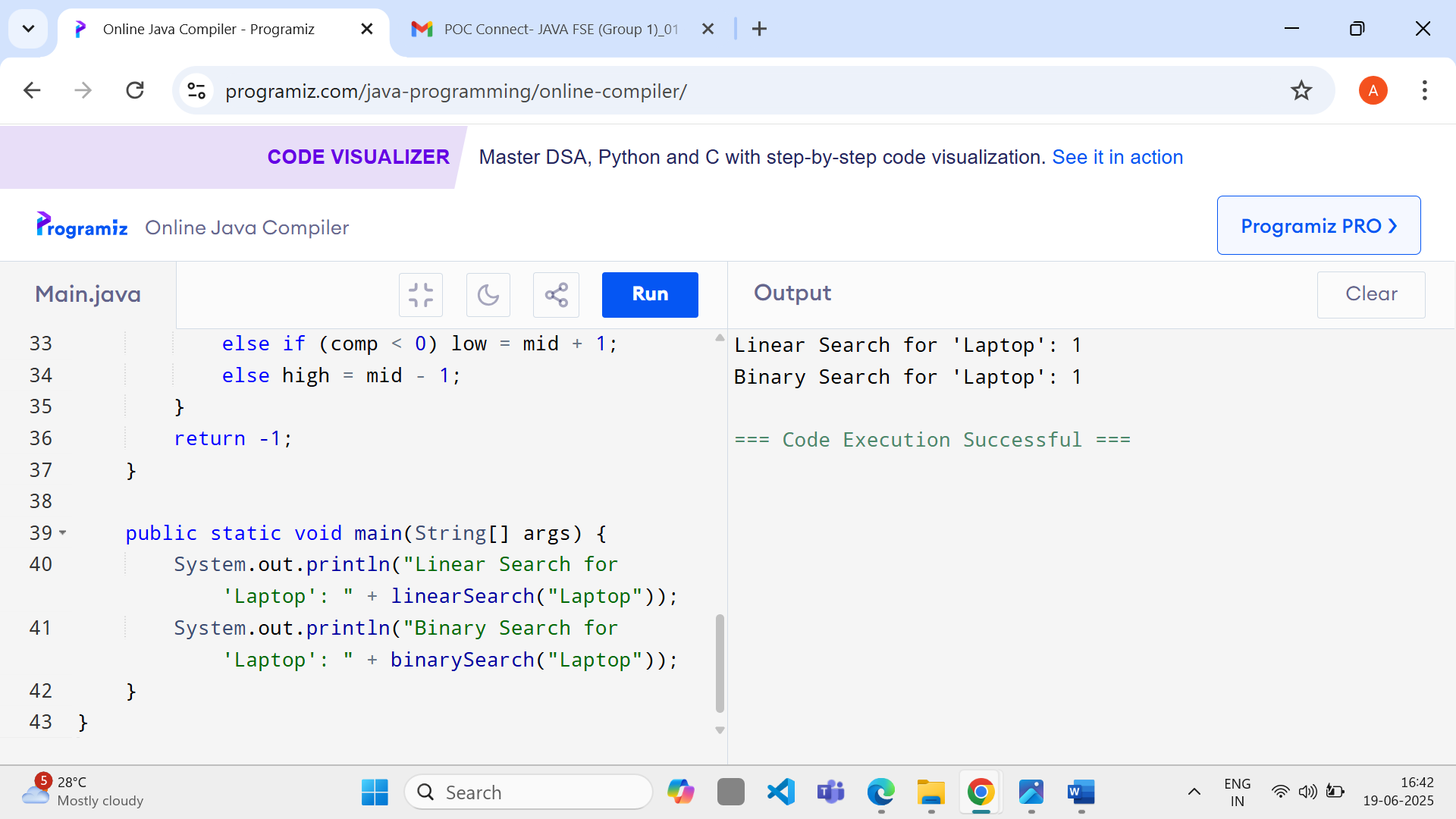
System.out.println("Linear Search for 'Laptop': " + linearSearch("Laptop"));

System.out.println("Binary Search for 'Laptop': " + binarySearch("Laptop"));

}

}

**Output:**

****

## Exercise 3: Sorting Customer Orders

**Code:**

class Order {

String customerName;

double totalPrice;

Order(String name, double price) {

this.customerName = name;

this.totalPrice = price;

}

}

public class Main {

static Order[] orders = {

new Order("Alice", 120.0),

new Order("Bob", 80.0),

new Order("Charlie", 150.0)

};

public static void bubbleSort() {

for (int i = 0; i < orders.length - 1; i++) {

for (int j = 0; j < orders.length - i - 1; j++) {

if (orders[j].totalPrice > orders[j + 1].totalPrice) {

Order temp = orders[j];

orders[j] = orders[j + 1];

orders[j + 1] = temp;

}

}

}

}

public static void printOrders() {

for (Order o : orders) {

System.out.println(o.customerName + ": $" + o.totalPrice);

}

}

public static void main(String[] args) {

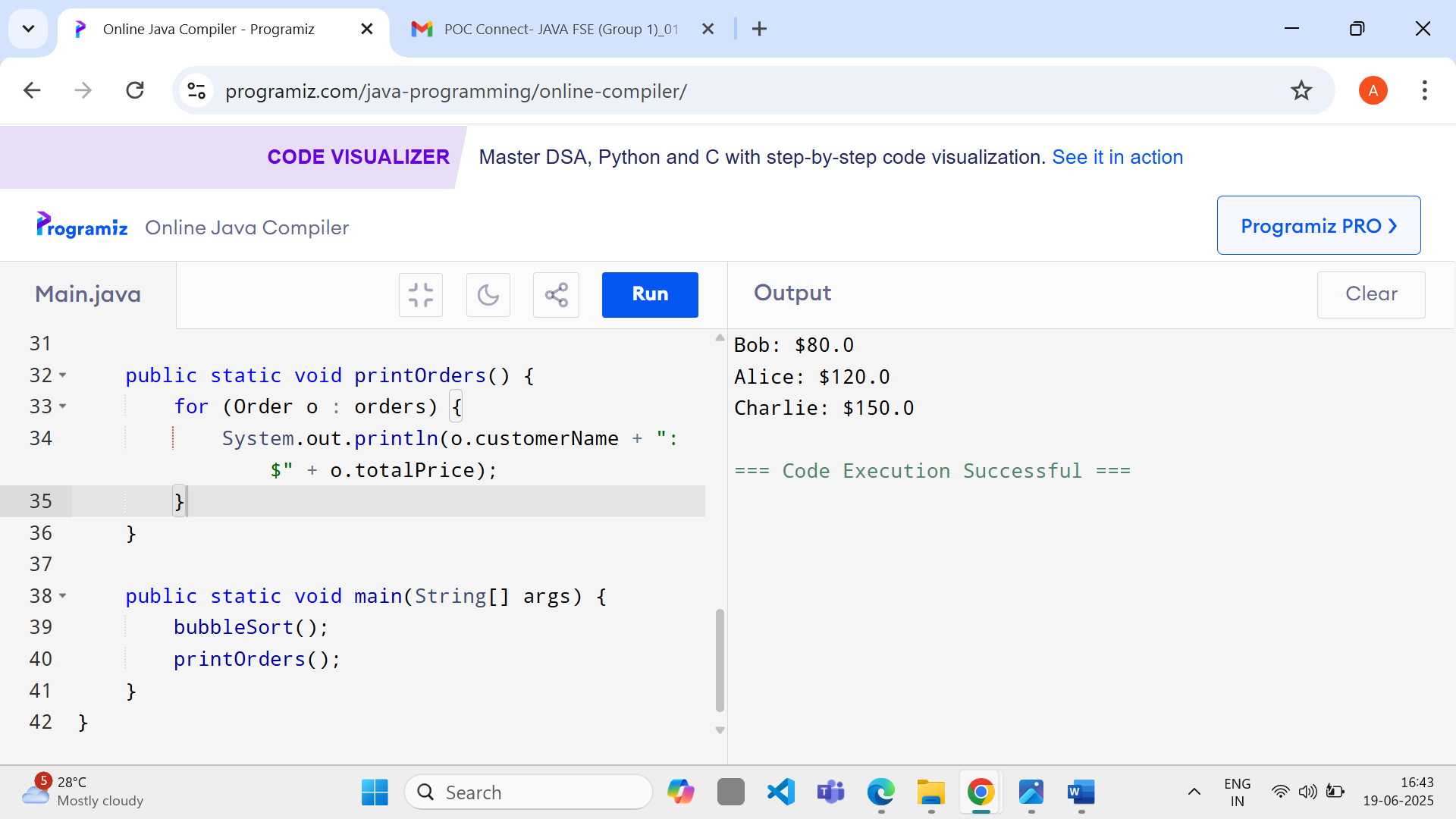
bubbleSort();

printOrders();

}

}

**Output:**

****

## 

## Exercise 4: Employee Management System

**Code:**

class Employee {

int employeeId;

String name, position;

double salary;

Employee(int id, String name, String pos, double sal) {

this.employeeId = id;

this.name = name;

this.position = pos;

this.salary = sal;

}

}

public class Main {

static Employee[] employees = new Employee[10];

static int count = 0;

public static void addEmployee(Employee e) {

employees[count++] = e;

System.out.println("Added: " + e.name);

}

public static void deleteEmployee(int id) {

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

if (employees[i].employeeId == id) {

System.out.println("Deleted Employee ID: " + id);

for (int j = i; j < count - 1; j++) {

employees[j] = employees[j + 1];

}

employees[--count] = null;

return;

}

}

System.out.println("Employee not found.");

}

public static void main(String[] args) {

addEmployee(new Employee(1, "Alice", "Manager", 5000));

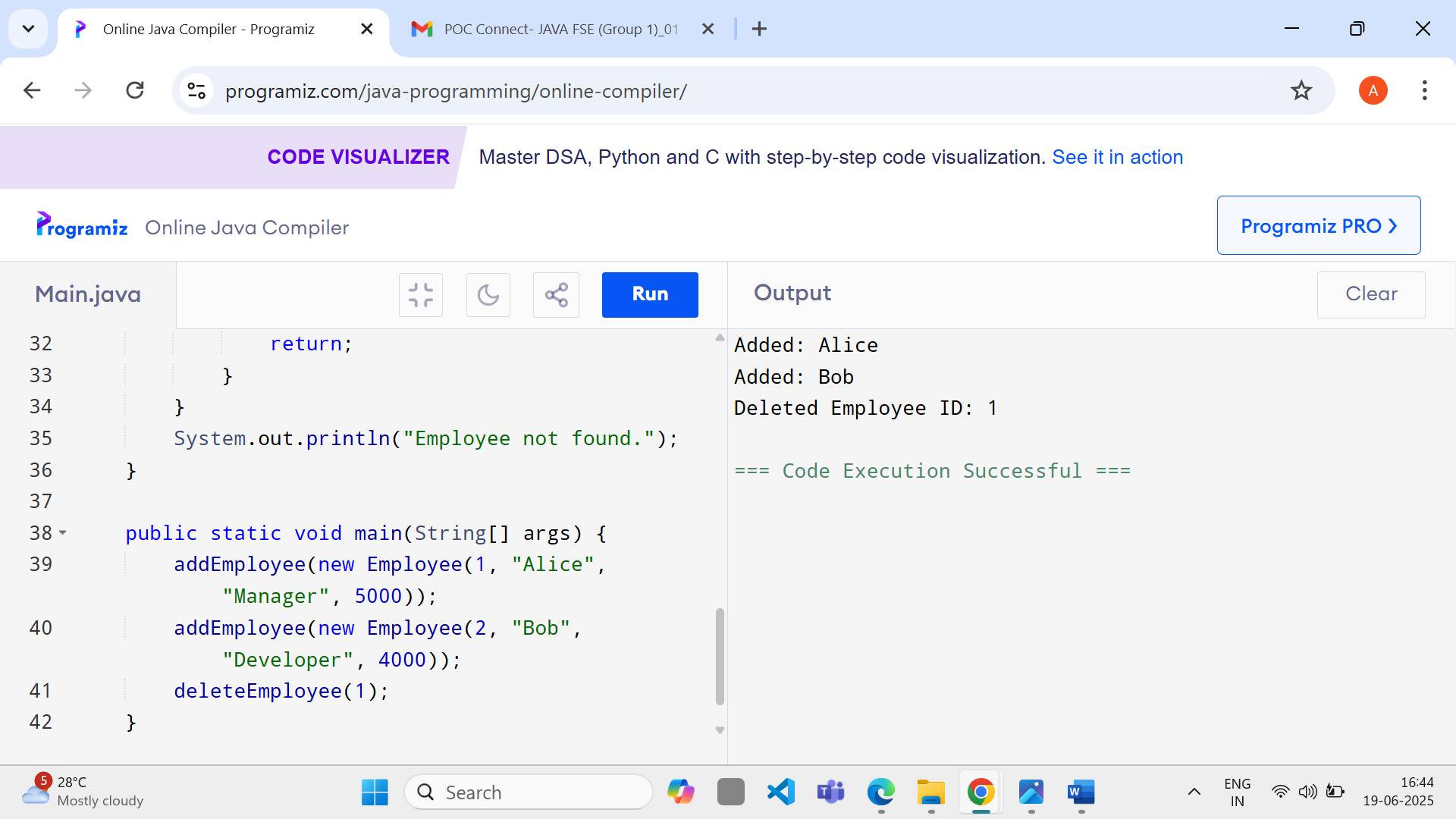
addEmployee(new Employee(2, "Bob", "Developer", 4000));

deleteEmployee(1);

}

}

**Output:**

****

## Exercise 5: Task Management System

**Code:**

class Task {

int taskId;

String taskName, status;

Task next;

Task(int id, String name, String status) {

this.taskId = id;

this.taskName = name;

this.status = status;

this.next = null;

}

}

public class Main {

static Task head = null;

public static void addTask(int id, String name, String status) {

Task newTask = new Task(id, name, status);

if (head == null) {

head = newTask;

} else {

Task current = head;

while (current.next != null) {

current = current.next;

}

current.next = newTask;

}

}

public static void printTasks() {

Task current = head;

while (current != null) {

System.out.println(current.taskId + ": " + current.taskName + " - " + current.status);

current = current.next;

}

}

public static void main(String[] args) {

addTask(1, "Design UI", "Pending");

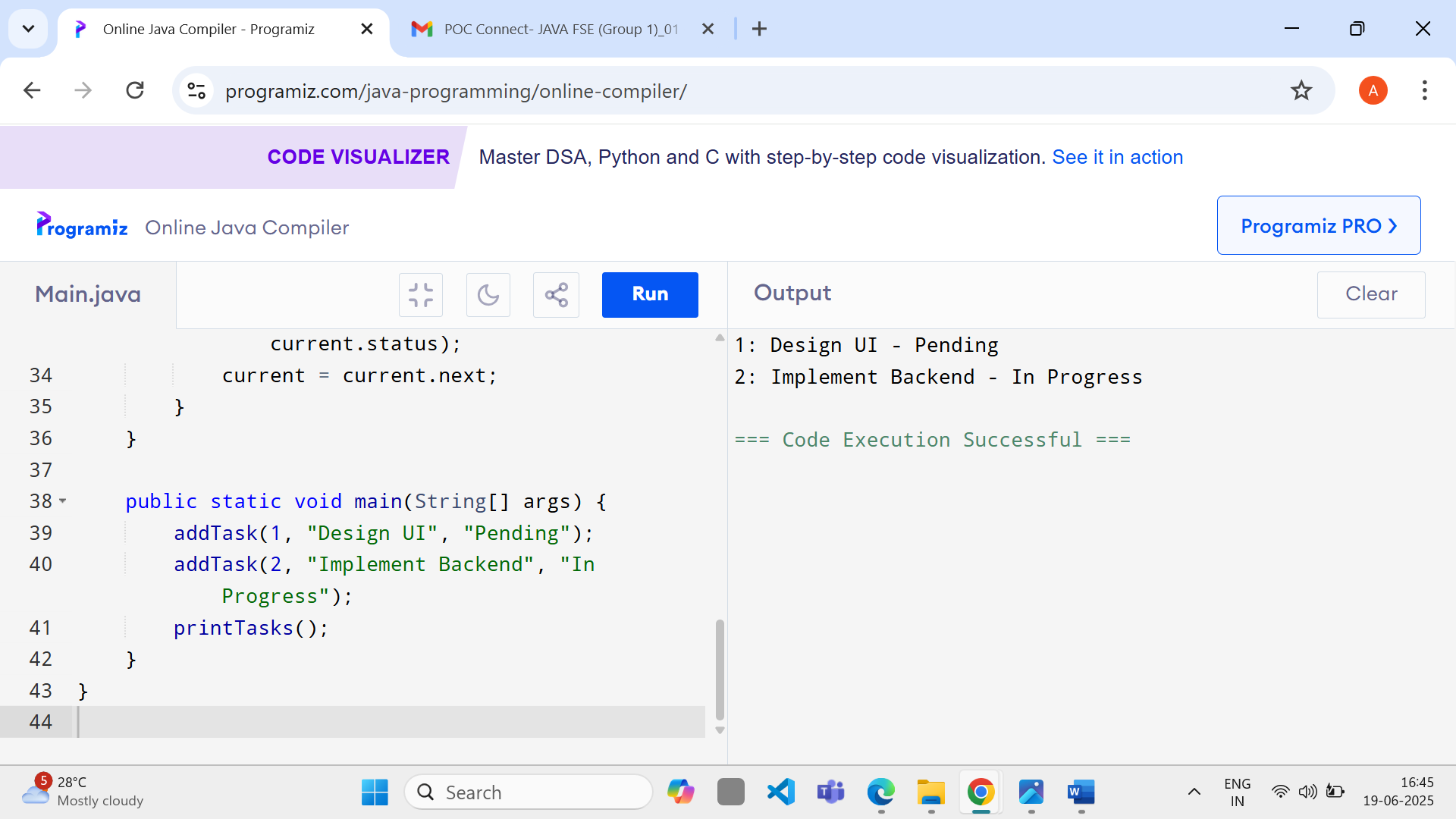
addTask(2, "Implement Backend", "In Progress");

printTasks();

}

}

**Output**

****

**Exercise 6: Library Management System**

**Code:**

class Book {

String bookId, title, author;

Book(String id, String title, String author) {

this.bookId = id;

this.title = title;

this.author = author;

}

}

public class Main {

static Book[] books = {

new Book("B1", "Java", "John"),

new Book("B2", "Python", "Alice"),

new Book("B3", "C++", "Bob")

};

public static int linearSearch(String title) {

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

if (books[i].title.equalsIgnoreCase(title))

return i;

}

return -1;

}

public static void main(String[] args) {

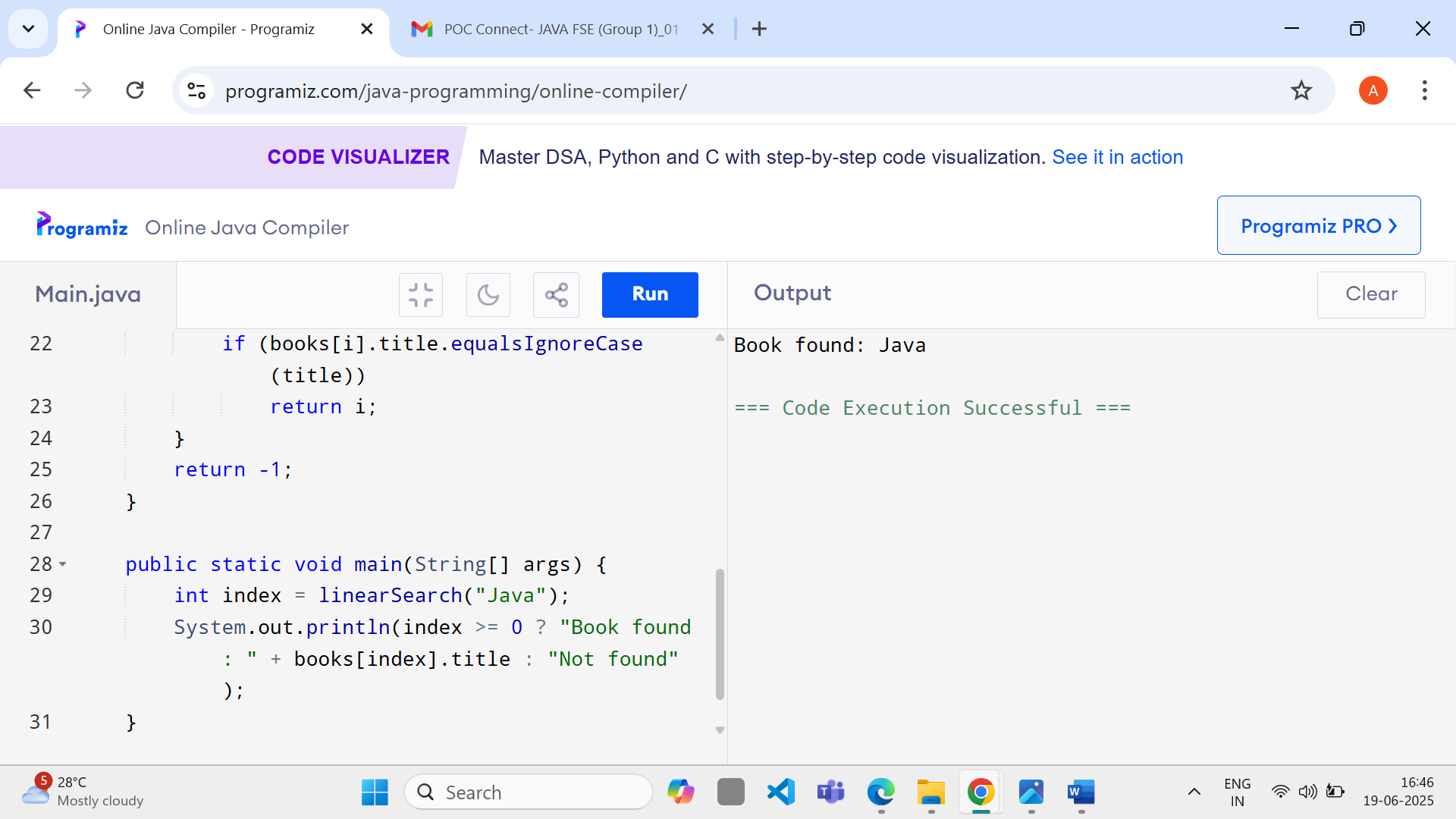
int index = linearSearch("Java");

System.out.println(index >= 0 ? "Book found: " + books[index].title : "Not found");

}

}

**Output**

****

## 

## Exercise 7: Financial Forecasting

**Code:**

public class Main {

public static double futureValue(double value, double rate, int years) {

if (years == 0) return value;

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

}

public static void main(String[] args) {

double initialValue = 1000.0;

double growthRate = 0.05; // 5%

int years = 5;

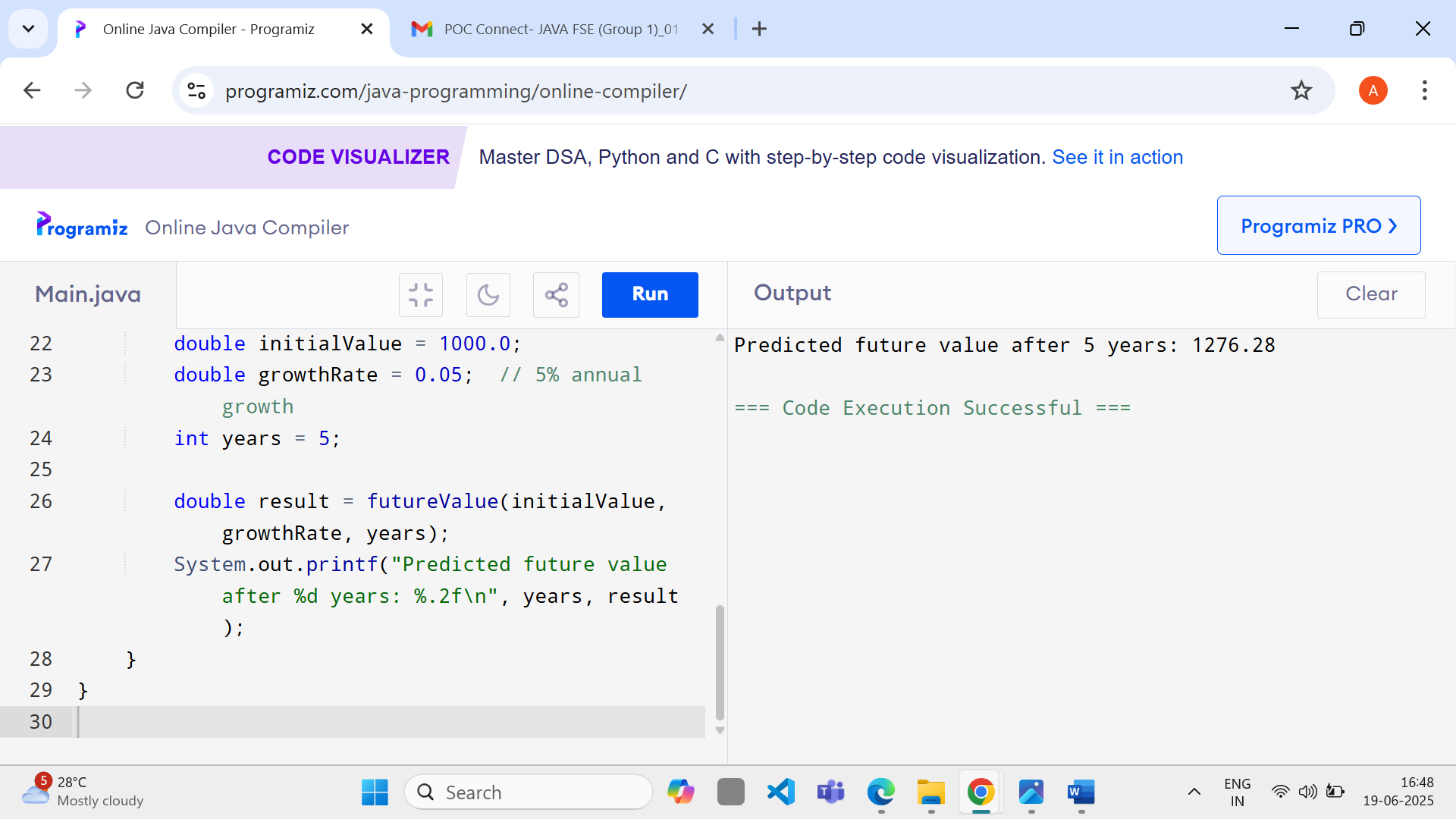
double result = futureValue(initialValue, growthRate, years);

System.out.printf("Predicted future value after %d years: %.2f\n", years, result);

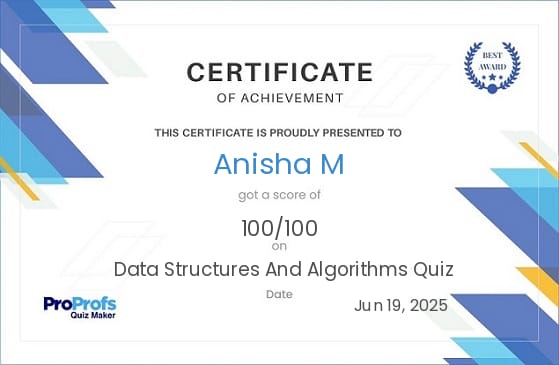
}

}

**Output:**

****

**Certificate of quiz**

****