**PAKHI SHARMA**

**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.

**CODE**

**// Logger.java**

**class Logger {**

**private static Logger instance;**

**private Logger() {**

**// private constructor**

**}**

**public static Logger getInstance() {**

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

**instance = new Logger();**

**}**

**return instance;**

**}**

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

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

**}**

**}**

**// Main.java**

**public class Main {**

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

**Logger logger1 = Logger.getInstance();**

**Logger logger2 = Logger.getInstance();**

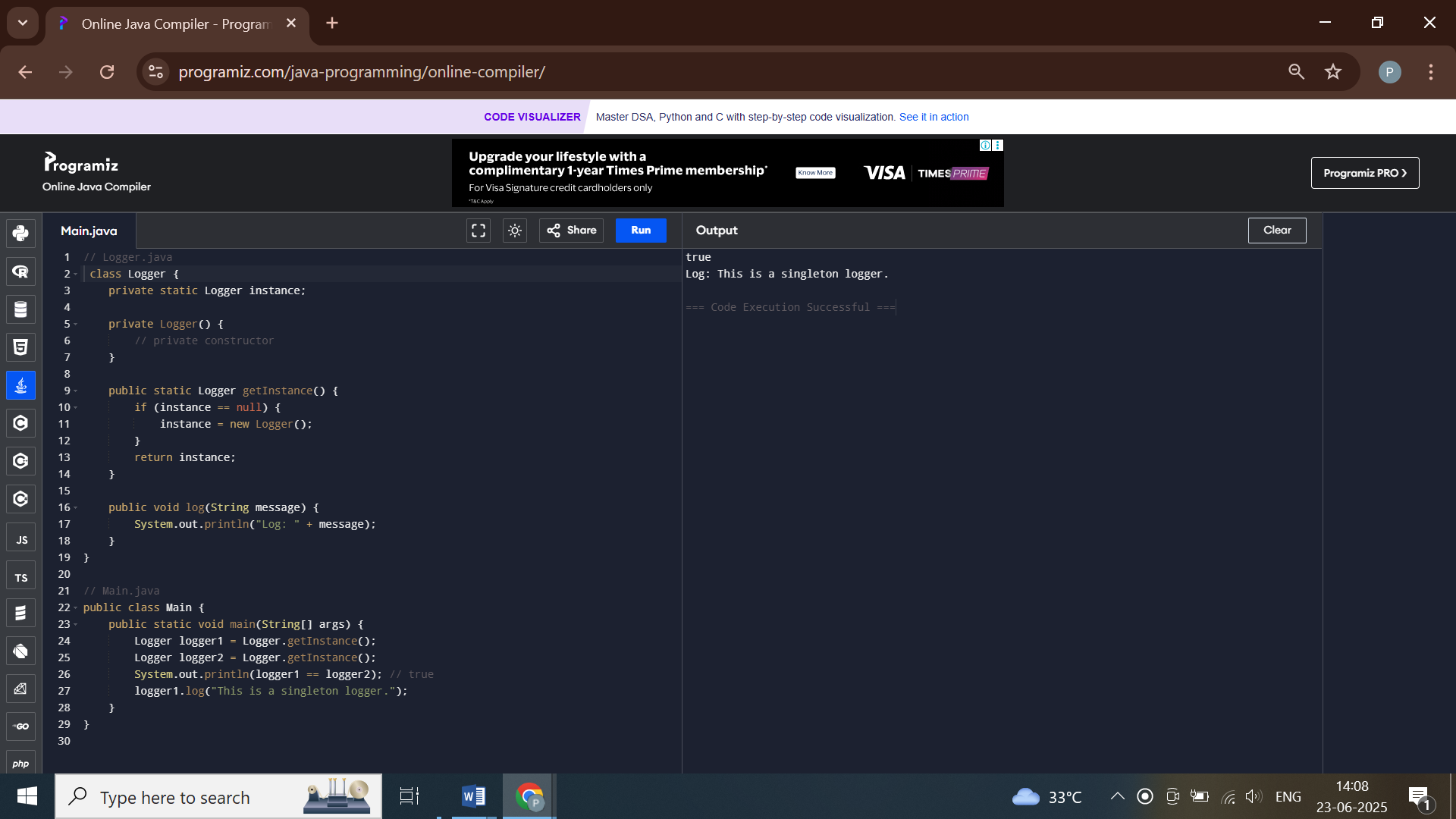
**System.out.println(logger1 == logger2); // true**

**logger1.log("This is a singleton logger.");**

**}**

**}**

**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.

**CODE**

**// Document.java**

**interface Document {**

**void open();**

**}**

**// WordDocument.java**

**class WordDocument implements Document {**

**public void open() {**

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

**}**

**}**

**// PdfDocument.java**

**class PdfDocument implements Document {**

**public void open() {**

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

**}**

**}**

**// ExcelDocument.java**

**class ExcelDocument implements Document {**

**public void open() {**

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

**}**

**}**

**// DocumentFactory.java**

**abstract class DocumentFactory {**

**public abstract Document createDocument();**

**}**

**// WordFactory.java**

**class WordFactory extends DocumentFactory {**

**public Document createDocument() {**

**return new WordDocument();**

**}**

**}**

**// Main.java**

**public class Main {**

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

**DocumentFactory factory = new WordFactory();**

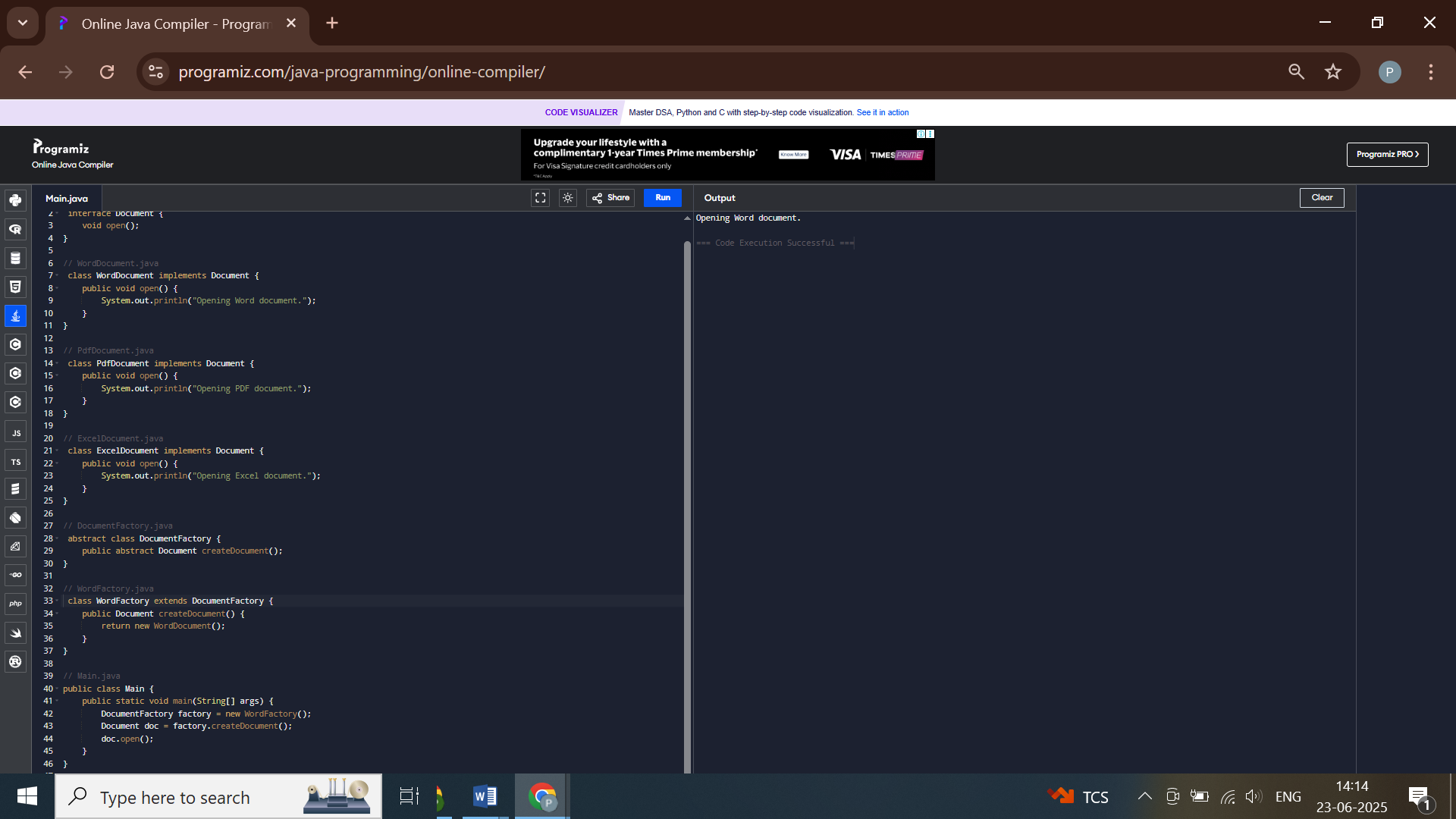
**Document doc = factory.createDocument();**

**doc.open();**

**}**

**}**

**OUTPUT**



**Exercise 3: Implementing the Builder Pattern**

**Scenario:**

You are developing a system to create complex objects such as a Computer with multiple optional parts. Use the Builder Pattern to manage the construction process.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **BuilderPatternExample**.
2. **Define a Product Class:**
   * Create a class **Computer** with attributes like **CPU**, **RAM**, **Storage**, etc.
3. **Implement the Builder Class:**
   * Create a static nested Builder class inside Computer with methods to set each attribute.
   * Provide a **build()** method in the Builder class that returns an instance of Computer.
4. **Implement the Builder Pattern:**
   * Ensure that the **Computer** class has a private constructor that takes the **Builder** as a parameter.
5. **Test the Builder Implementation:**
   * Create a test class to demonstrate the creation of different configurations of Computer using the Builder pattern.

**CODE**

**// Computer.java**

**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;**

**}**

**public 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 specs() {**

**System.out.println("CPU: " + CPU + ", RAM: " + RAM + ", Storage: " + storage);**

**}**

**}**

**// Main.java**

**public class Main {**

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

**Computer comp = new Computer.Builder()**

**.setCPU("i7")**

**.setRAM("16GB")**

**.setStorage("512GB SSD")**

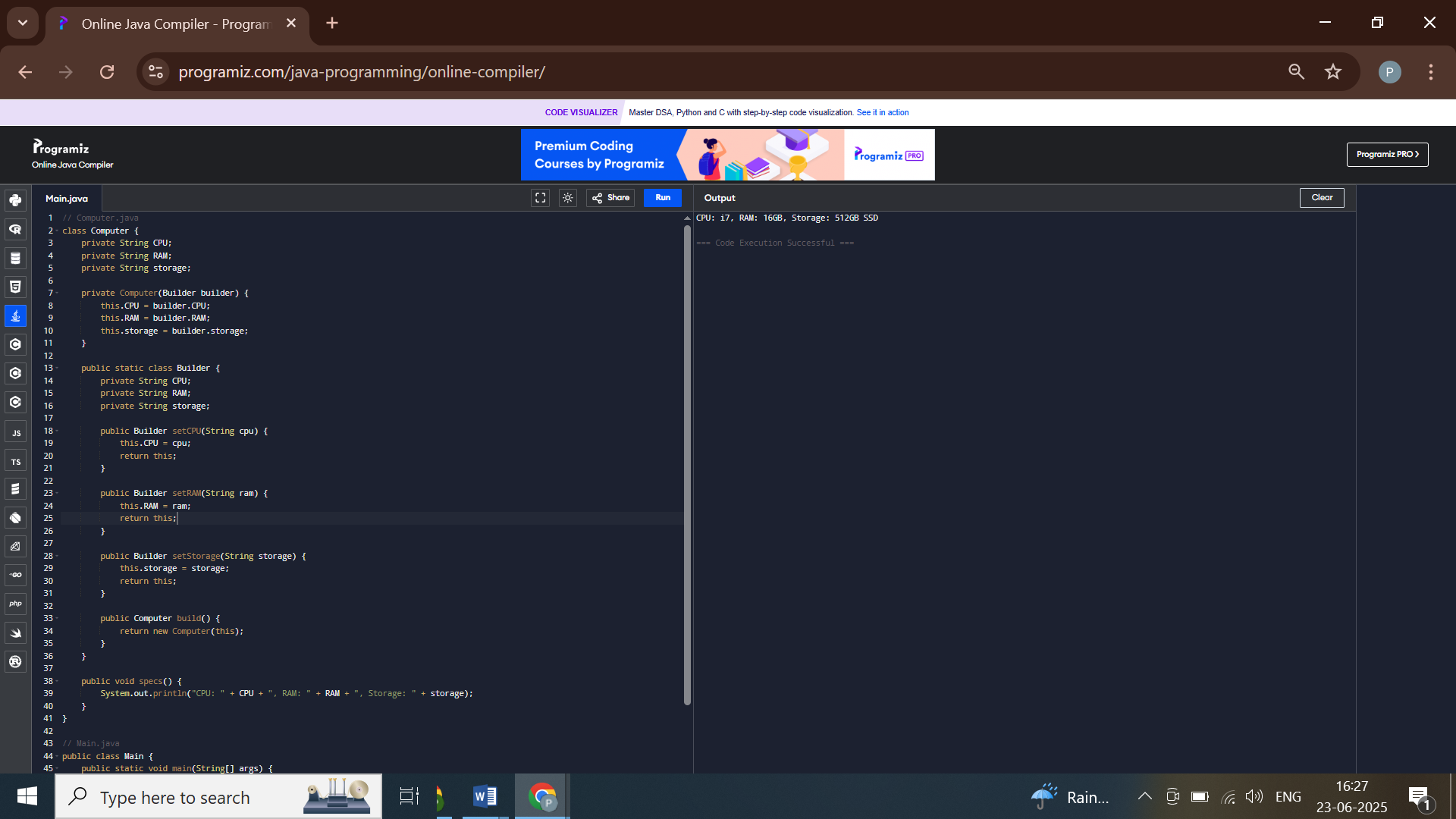
**.build();**

**comp.specs();**

**}**

**}**

**OUTPUT**



**Exercise 4: Implementing the Adapter Pattern**

**Scenario:**

You are developing a payment processing system that needs to integrate with multiple third-party payment gateways with different interfaces. Use the Adapter Pattern to achieve this.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **AdapterPatternExample**.
2. **Define Target Interface:**
   * Create an interface **PaymentProcessor** with methods like **processPayment()**.
3. **Implement Adaptee Classes:**
   * Create classes for different payment gateways with their own methods.
4. **Implement the Adapter Class:**
   * Create an adapter class for each payment gateway that implements PaymentProcessor and translates the calls to the gateway-specific methods.
5. **Test the Adapter Implementation:**
   * Create a test class to demonstrate the use of different payment gateways through the adapter.

**CODE**

**// PaymentProcessor.java**

**interface PaymentProcessor {**

**void processPayment(double amount);**

**}**

**// PayPalGateway.java**

**class PayPalGateway {**

**public void makePayment(double amount) {**

**System.out.println("Paid via PayPal: " + amount);**

**}**

**}**

**// PayPalAdapter.java**

**class PayPalAdapter implements PaymentProcessor {**

**private PayPalGateway paypal;**

**public PayPalAdapter() {**

**this.paypal = new PayPalGateway();**

**}**

**public void processPayment(double amount) {**

**paypal.makePayment(amount);**

**}**

**}**

**// Main.java**

**public class Main {**

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

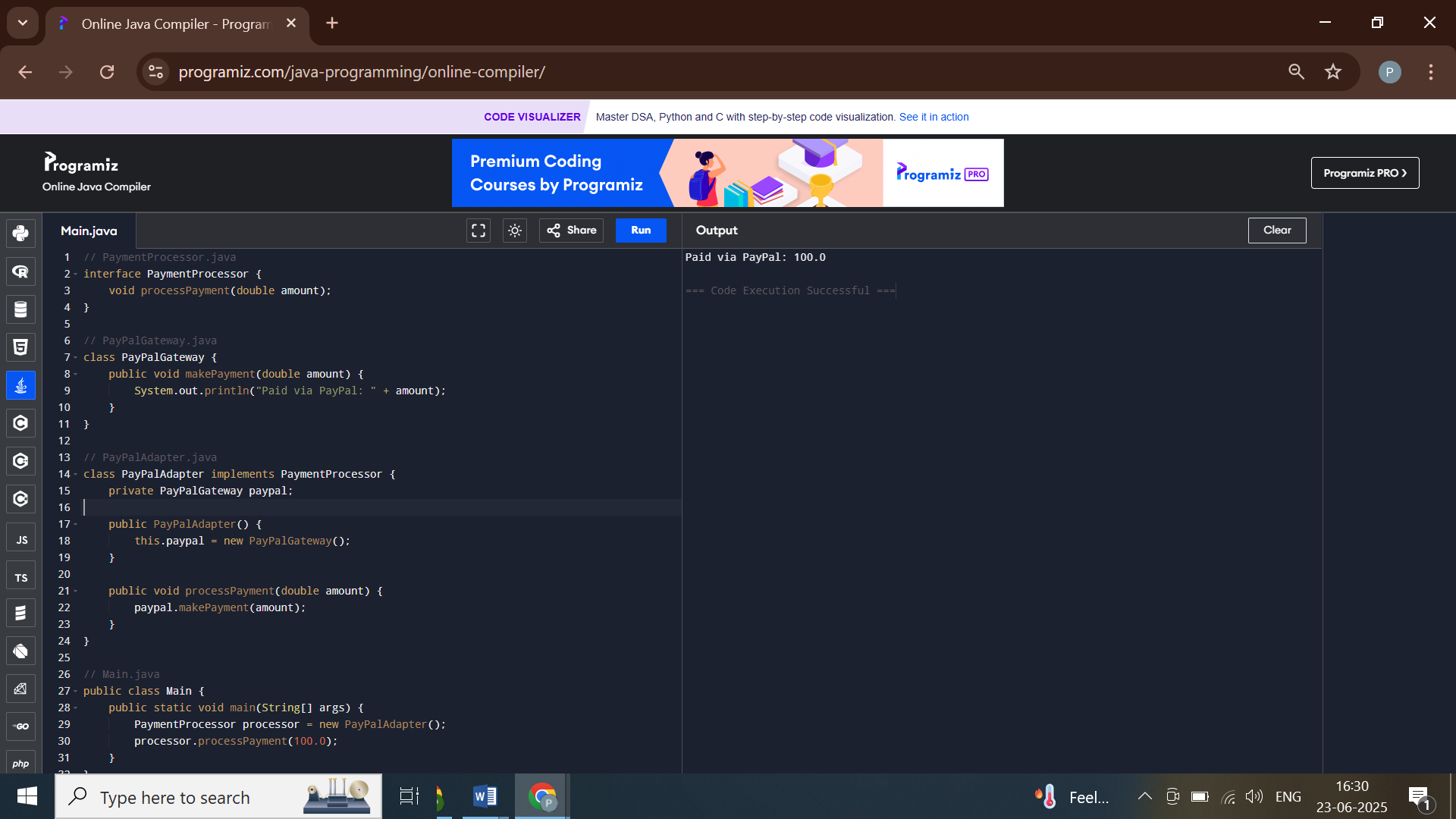
**PaymentProcessor processor = new PayPalAdapter();**

**processor.processPayment(100.0);**

**}**

**}**

**OUTPUT**



**Exercise 5: Implementing the Decorator Pattern**

**Scenario:**

You are developing a notification system where notifications can be sent via multiple channels (e.g., Email, SMS). Use the Decorator Pattern to add functionalities dynamically.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **DecoratorPatternExample**.
2. **Define Component Interface:**
   * Create an interface **Notifier** with a method **send()**.
3. **Implement Concrete Component:**
   * Create a class **EmailNotifier** that implements Notifier.
4. **Implement Decorator Classes:**
   * Create abstract decorator class **NotifierDecorator** that implements **Notifier** and holds a reference to a **Notifier** object.
   * Create concrete decorator classes like **SMSNotifierDecorator**, **SlackNotifierDecorator** that extend **NotifierDecorator**.
5. **Test the Decorator Implementation:**
   * Create a test class to demonstrate sending notifications via multiple channels using decorators.

**CODE**

**// Notifier.java**

**interface Notifier {**

**void send(String message);**

**}**

**// EmailNotifier.java**

**class EmailNotifier implements Notifier {**

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

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

**}**

**}**

**// NotifierDecorator.java**

**abstract class NotifierDecorator implements Notifier {**

**protected Notifier notifier;**

**public NotifierDecorator(Notifier notifier) {**

**this.notifier = notifier;**

**}**

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

**notifier.send(message);**

**}**

**}**

**// SMSNotifier.java**

**class SMSNotifier extends NotifierDecorator {**

**public SMSNotifier(Notifier notifier) {**

**super(notifier);**

**}**

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

**super.send(message);**

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

**}**

**}**

**// Main.java**

**public class Main {**

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

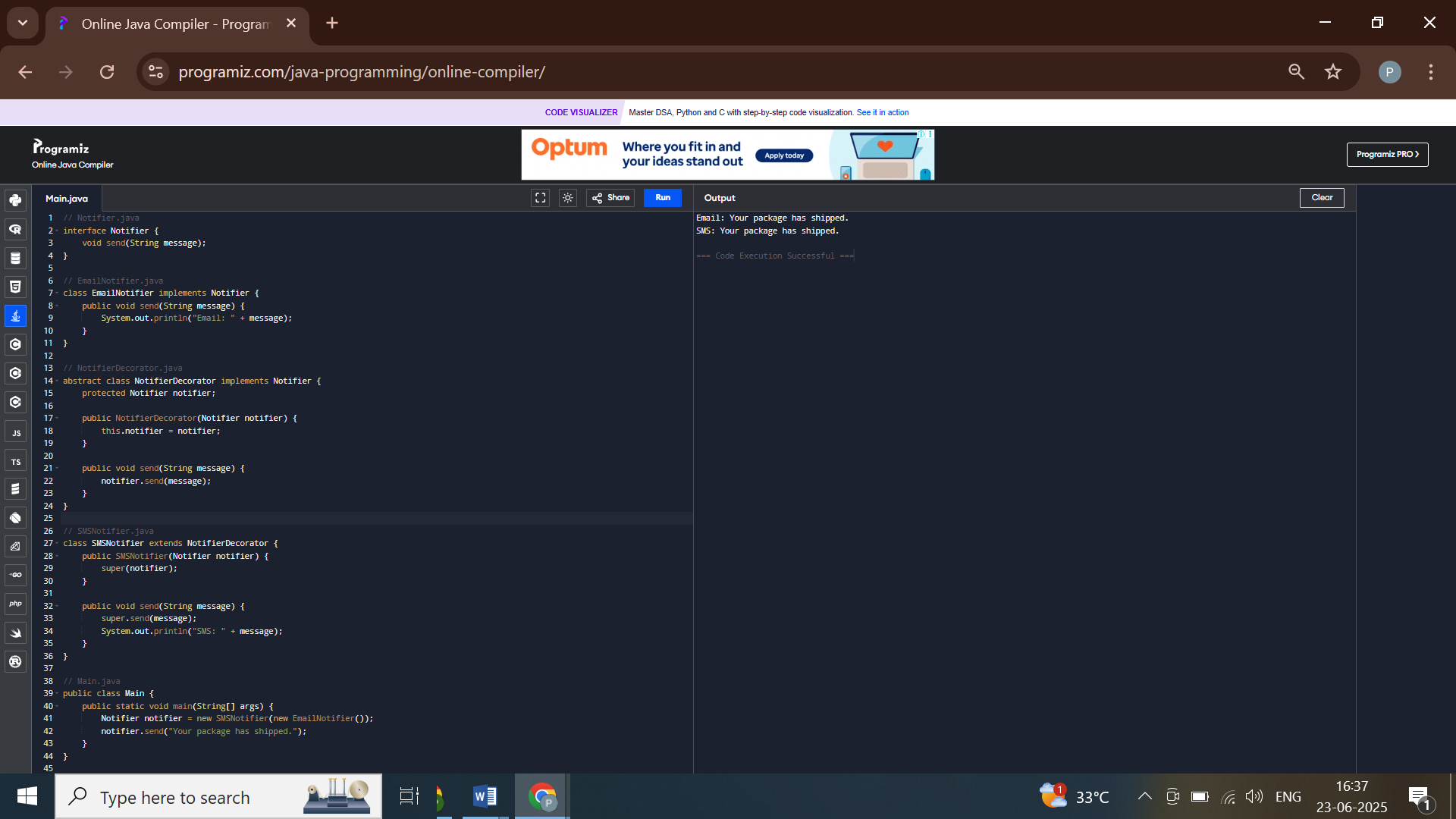
**Notifier notifier = new SMSNotifier(new EmailNotifier());**

**notifier.send("Your package has shipped.");**

**}**

**}**

**OUTPUT**



**Exercise 6: Implementing the Proxy Pattern**

**Scenario:**

You are developing an image viewer application that loads images from a remote server. Use the Proxy Pattern to add lazy initialization and caching.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **ProxyPatternExample**.
2. **Define Subject Interface:**
   * Create an interface Image with a method **display()**.
3. **Implement Real Subject Class:**
   * Create a class **RealImage** that implements Image and loads an image from a remote server.
4. **Implement Proxy Class:**
   * Create a class **ProxyImage** that implements Image and holds a reference to RealImage.
   * Implement lazy initialization and caching in **ProxyImage**.
5. **Test the Proxy Implementation:**
   * Create a test class to demonstrate the use of **ProxyImage** to load and display images.

**CODE**

**// Image.java**

**interface Image {**

**void display();**

**}**

**// RealImage.java**

**class RealImage implements Image {**

**private String fileName;**

**public RealImage(String fileName) {**

**this.fileName = fileName;**

**loadImageFromDisk();**

**}**

**private void loadImageFromDisk() {**

**System.out.println("Loading " + fileName);**

**}**

**public void display() {**

**System.out.println("Displaying " + fileName);**

**}**

**}**

**// ProxyImage.java**

**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();**

**}**

**}**

**// Main.java**

**public class Main {**

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

**Image img = new ProxyImage("photo.jpg");**

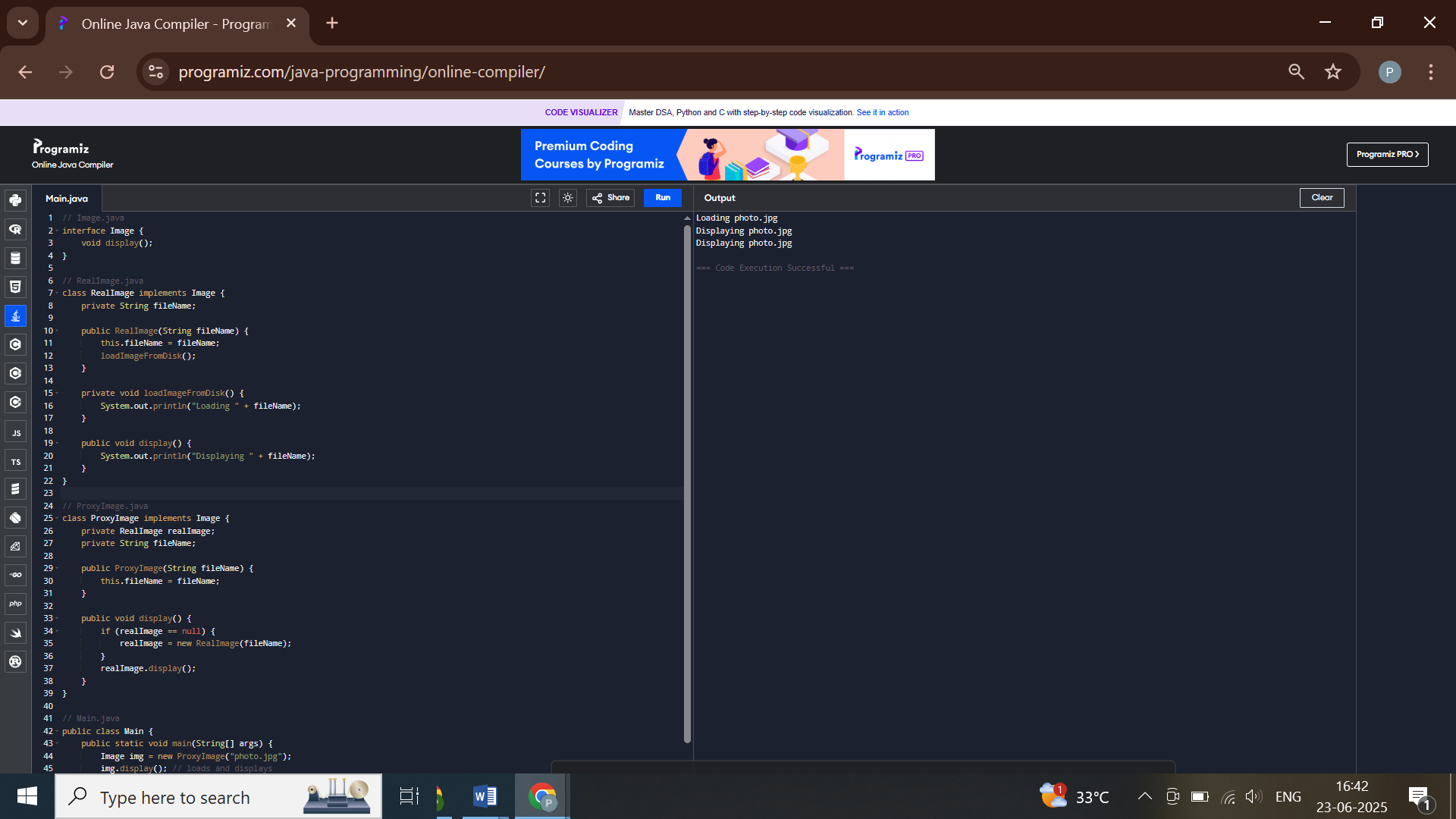
**img.display(); // loads and displays**

**img.display(); // only displays (cached)**

**}**

**}**

**OUTPUT**



**Exercise 7: Implementing the Observer Pattern**

**Scenario:**

You are developing a stock market monitoring application where multiple clients need to be notified whenever stock prices change. Use the Observer Pattern to achieve this.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **ObserverPatternExample**.
2. **Define Subject Interface:**
   * Create an interface **Stock** with methods to **register**, **deregister**, and **notify** observers.
3. **Implement Concrete Subject:**
   * Create a class **StockMarket** that implements **Stock** and maintains a list of observers.
4. **Define Observer Interface:**
   * Create an interface Observer with a method **update().**
5. **Implement Concrete Observers:**
   * Create classes **MobileApp**, **WebApp** that implement Observer.
6. **Test the Observer Implementation:**
   * Create a test class to demonstrate the registration and notification of observers.

**CODE**

**// Observer.java**

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

**interface Observer {**

**void update(float price);**

**}**

**// Stock.java**

**interface Stock {**

**void register(Observer o);**

**void deregister(Observer o);**

**void notifyObservers();**

**}**

**// StockMarket.java**

**class StockMarket implements Stock {**

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

**private float price;**

**public void setPrice(float price) {**

**this.price = 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(price);**

**}**

**}**

**}**

**// MobileApp.java**

**class MobileApp implements Observer {**

**public void update(float price) {**

**System.out.println("Mobile App - Stock price: " + price);**

**}**

**}**

**// Main.java**

**public class Main {**

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

**StockMarket market = new StockMarket();**

**Observer mobile = new MobileApp();**

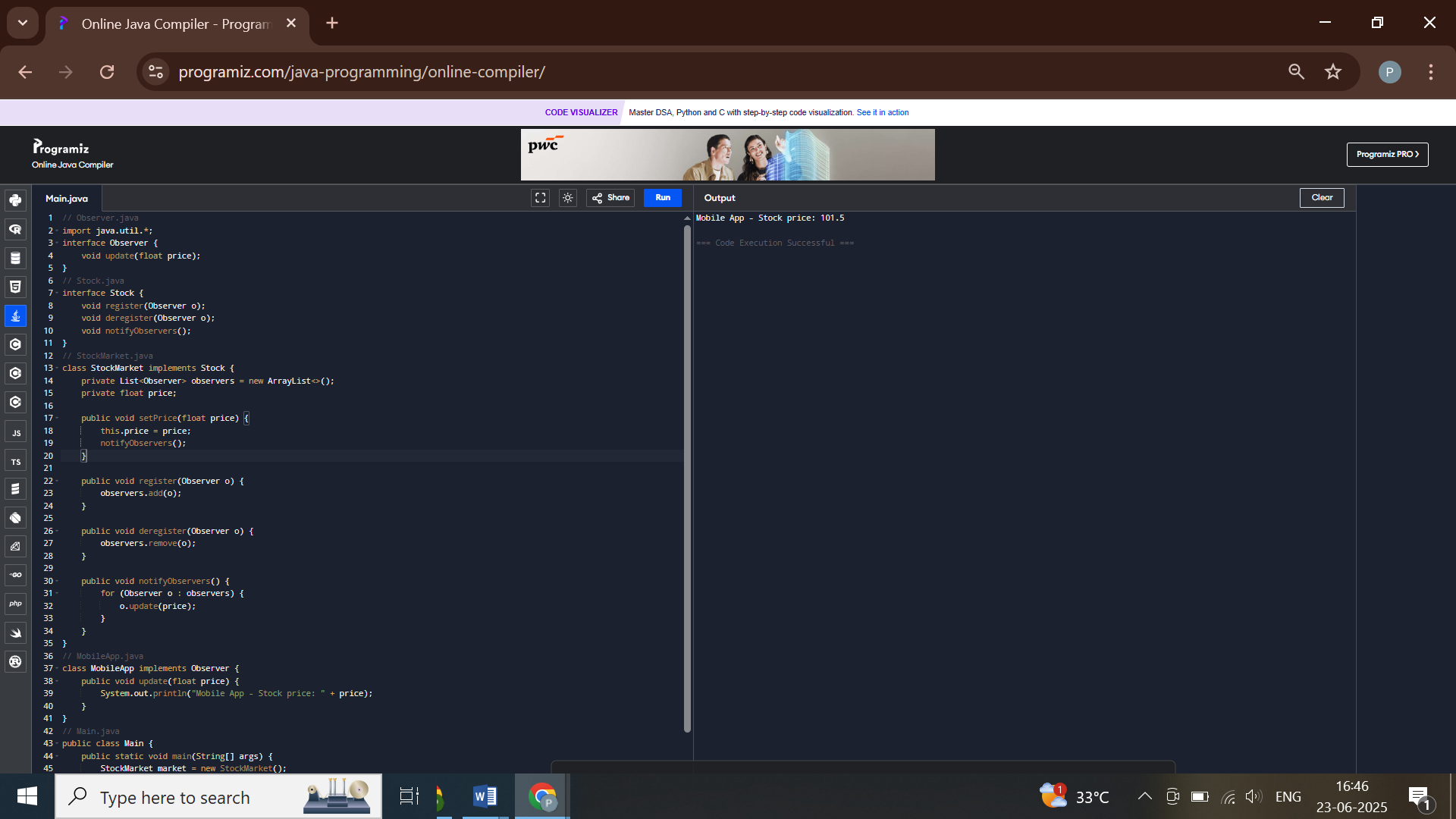
**market.register(mobile);**

**market.setPrice(101.5f);**

**}**

**}**

**OUTPUT**



**Exercise 8: Implementing the Strategy Pattern**

**Scenario:**

You are developing a payment system where different payment methods (e.g., Credit Card, PayPal) can be selected at runtime. Use the Strategy Pattern to achieve this.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **StrategyPatternExample**.
2. **Define Strategy Interface:**
   * Create an interface PaymentStrategy with a method **pay()**.
3. **Implement Concrete Strategies:**
   * Create classes **CreditCardPayment**, **PayPalPayment** that implement **PaymentStrategy**.
4. **Implement Context Class:**
   * Create a class **PaymentContext** that holds a reference to **PaymentStrategy** and a method to execute the strategy.
5. **Test the Strategy Implementation:**
   * Create a test class to demonstrate selecting and using different payment strategies.

**CODE**

**// PaymentStrategy.java**

**interface PaymentStrategy {**

**void pay(double amount);**

**}**

**// CreditCardPayment.java**

**class CreditCardPayment implements PaymentStrategy {**

**public void pay(double amount) {**

**System.out.println("Paid via Credit Card: " + amount);**

**}**

**}**

**// PayPalPayment.java**

**class PayPalPayment implements PaymentStrategy {**

**public void pay(double amount) {**

**System.out.println("Paid via PayPal: " + amount);**

**}**

**}**

**// PaymentContext.java**

**class PaymentContext {**

**private PaymentStrategy strategy;**

**public void setStrategy(PaymentStrategy strategy) {**

**this.strategy = strategy;**

**}**

**public void pay(double amount) {**

**strategy.pay(amount);**

**}**

**}**

**// Main.java**

**public class Main {**

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

**PaymentContext context = new PaymentContext();**

**context.setStrategy(new CreditCardPayment());**

**context.pay(500.0);**

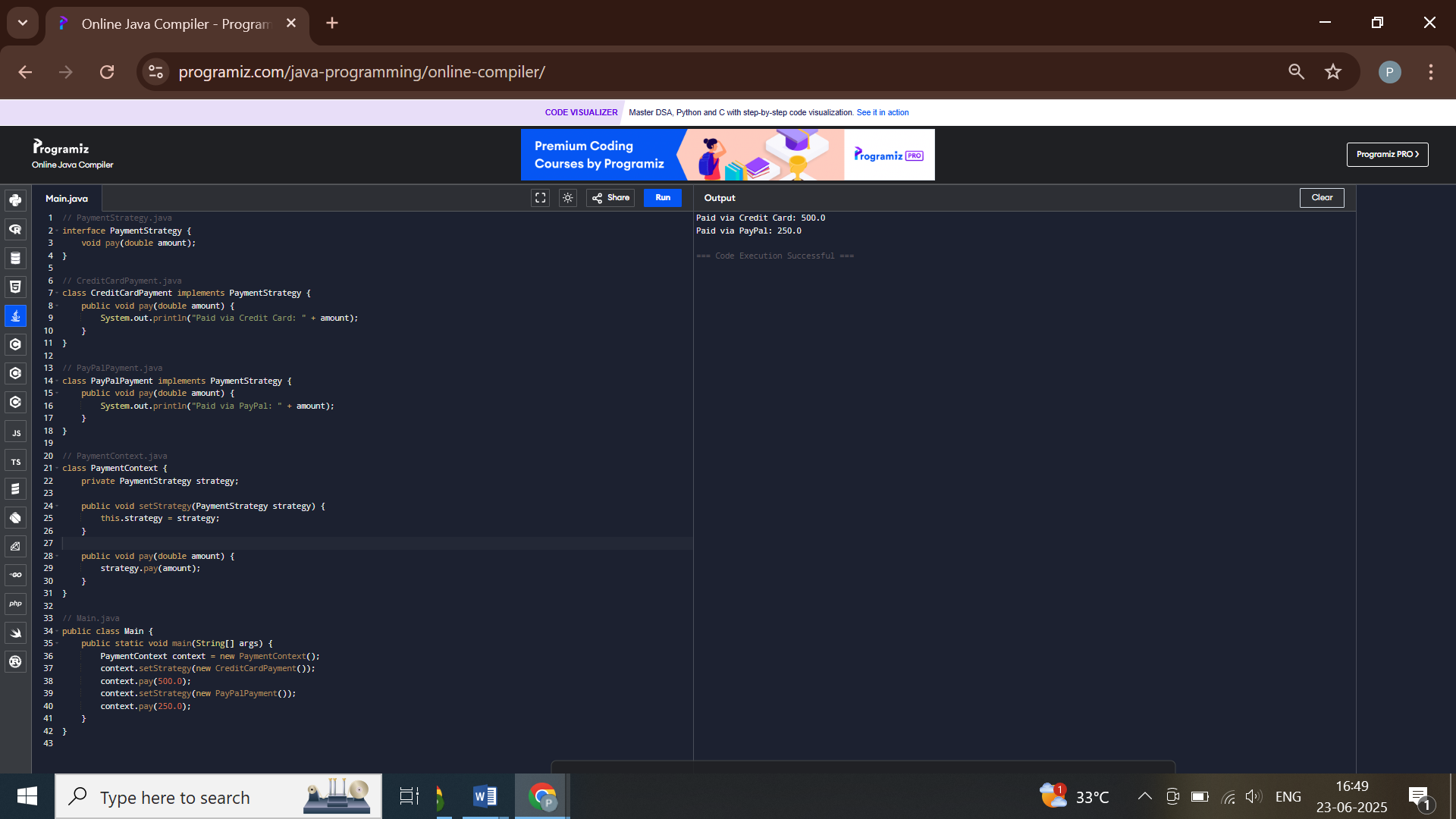
**context.setStrategy(new PayPalPayment());**

**context.pay(250.0);**

**}**

**}**

**OUTPUT**



**Exercise 9: Implementing the Command Pattern**

**Scenario:** You are developing a home automation system where commands can be issued to turn devices on or off. Use the Command Pattern to achieve this.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **CommandPatternExample**.
2. **Define Command Interface:**
   * Create an interface Command with a method **execute()**.
3. **Implement Concrete Commands:**
   * Create classes **LightOnCommand**, **LightOffCommand** that implement Command.
4. **Implement Invoker Class:**
   * Create a class **RemoteControl** that holds a reference to a Command and a method to execute the command.
5. **Implement Receiver Class:**
   * Create a class **Light** with methods to turn on and off.
6. **Test the Command Implementation:**
   * Create a test class to demonstrate issuing commands using the **RemoteControl**.

**CODE**

**// Command.java**

**interface Command {**

**void execute();**

**}**

**// Light.java**

**class Light {**

**public void turnOn() {**

**System.out.println("Light is ON");**

**}**

**public void turnOff() {**

**System.out.println("Light is OFF");**

**}**

**}**

**// LightOnCommand.java**

**class LightOnCommand implements Command {**

**private Light light;**

**public LightOnCommand(Light light) {**

**this.light = light;**

**}**

**public void execute() {**

**light.turnOn();**

**}**

**}**

**// LightOffCommand.java**

**class LightOffCommand implements Command {**

**private Light light;**

**public LightOffCommand(Light light) {**

**this.light = light;**

**}**

**public void execute() {**

**light.turnOff();**

**}**

**}**

**// RemoteControl.java**

**class RemoteControl {**

**private Command command;**

**public void setCommand(Command command) {**

**this.command = command;**

**}**

**public void pressButton() {**

**command.execute();**

**}**

**}**

**// Main.java**

**public class Main {**

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

**Light light = new Light();**

**RemoteControl remote = new RemoteControl();**

**remote.setCommand(new LightOnCommand(light));**

**remote.pressButton();**

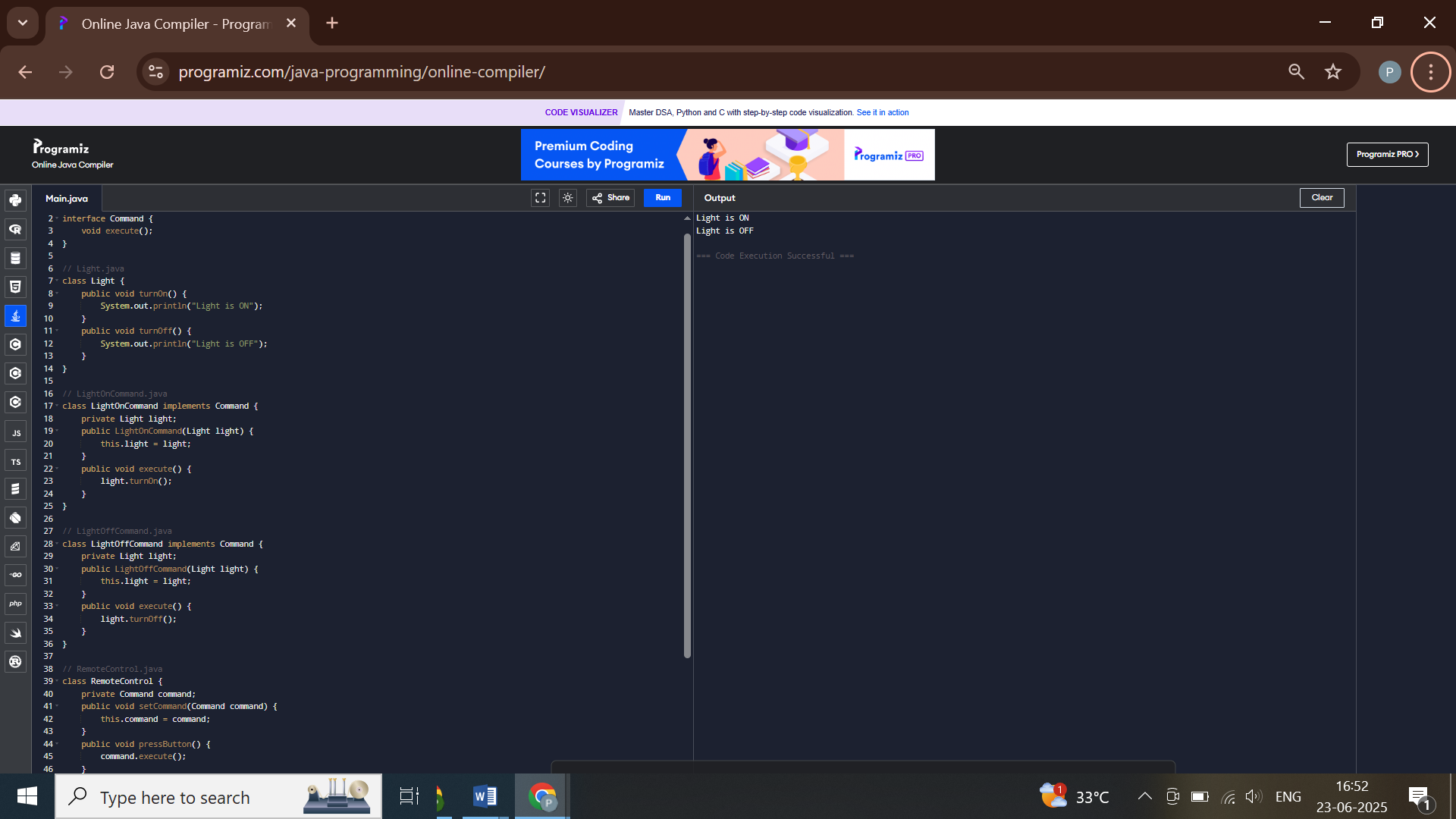
**remote.setCommand(new LightOffCommand(light));**

**remote.pressButton();**

**}**

**}**

**OUTPUT**



**Exercise 10: Implementing the MVC Pattern**

**Scenario:**

You are developing a simple web application for managing student records using the MVC pattern.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **MVCPatternExample**.
2. **Define Model Class:**
   * Create a class **Student** with attributes like **name, id, and grade**.
3. **Define View Class:**
   * Create a class **StudentView** with a method **displayStudentDetails()**.
4. **Define Controller Class:**
   * Create a class **StudentController** that handles the communication between the model and the view.
5. **Test the MVC Implementation:**
   * Create a main class to demonstrate creating a **Student**, updating its details using **StudentController**, and displaying them using **StudentView**.

**CODE**

**// Student.java**

**class Student {**

**private String name;**

**private int id;**

**private String grade;**

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

**this.name = name;**

**this.id = id;**

**this.grade = grade;**

**}**

**// Getters and setters**

**public String getName() { return name; }**

**public int getId() { return id; }**

**public String getGrade() { return grade; }**

**public void setGrade(String grade) { this.grade = grade; }**

**}**

**// StudentView.java**

**class StudentView {**

**public void displayStudentDetails(Student student) {**

**System.out.println("Name: " + student.getName() + ", ID: " + student.getId() + ", Grade: " + student.getGrade());**

**}**

**}**

**// StudentController.java**

**class StudentController {**

**private Student student;**

**private StudentView view;**

**public StudentController(Student student, StudentView view) {**

**this.student = student;**

**this.view = view;**

**}**

**public void updateGrade(String grade) {**

**student.setGrade(grade);**

**}**

**public void show() {**

**view.displayStudentDetails(student);**

**}**

**}**

**// Main.java**

**public class Main {**

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

**Student s = new Student("Pakhi", 1, "A");**

**StudentView view = new StudentView();**

**StudentController controller = new StudentController(s, view);**

**controller.show();**

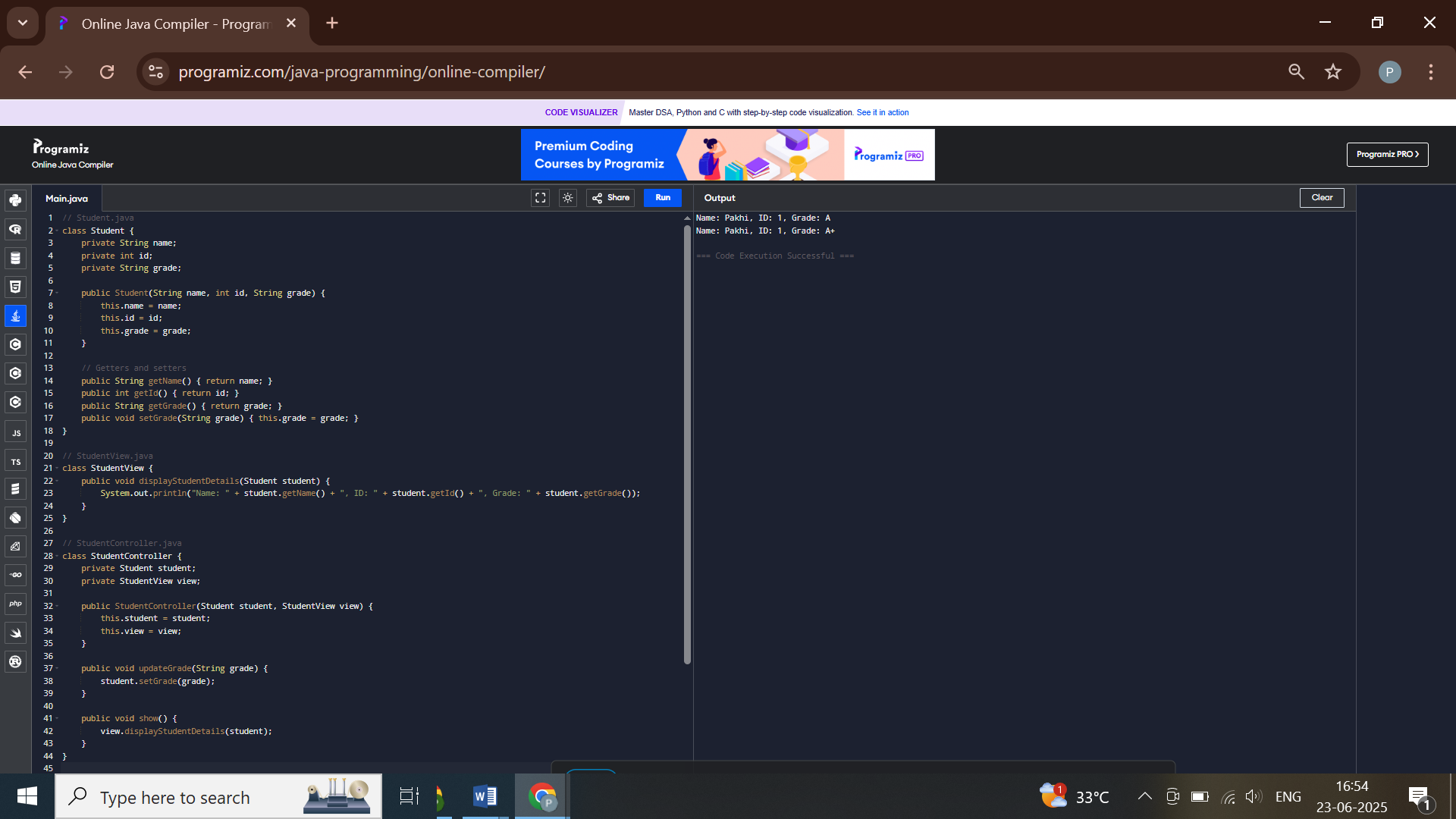
**controller.updateGrade("A+");**

**controller.show();**

**}**

**}**

**OUTPUT**



**Exercise 11: Implementing Dependency Injection**

**Scenario:**

You are developing a customer management application where the service class depends on a repository class. Use Dependency Injection to manage these dependencies.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **DependencyInjectionExample**.
2. **Define Repository Interface:**
   * Create an interface **CustomerRepository** with methods like **findCustomerById()**.
3. **Implement Concrete Repository:**
   * Create a class **CustomerRepositoryImpl** that implements **CustomerRepository**.
4. **Define Service Class:**
   * Create a class **CustomerService** that depends on **CustomerRepository**.
5. **Implement Dependency Injection:**
   * Use constructor injection to inject **CustomerRepository** into **CustomerService**.
6. **Test the Dependency Injection Implementation:**
   * Create a main class to demonstrate creating a **CustomerService** with **CustomerRepositoryImpl** and using it to find a customer.

**CODE**

**// CustomerRepository.java**

**interface CustomerRepository {**

**String findCustomerById(int id);**

**}**

**// CustomerRepositoryImpl.java**

**class CustomerRepositoryImpl implements CustomerRepository {**

**public String findCustomerById(int id) {**

**return "Customer #" + id;**

**}**

**}**

**// CustomerService.java**

**class CustomerService {**

**private CustomerRepository repository;**

**public CustomerService(CustomerRepository repository) {**

**this.repository = repository;**

**}**

**public void printCustomer(int id) {**

**System.out.println(repository.findCustomerById(id));**

**}**

**}**

**// Main.java**

**public class Main {**

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

**CustomerRepository repo = new CustomerRepositoryImpl();**

**CustomerService service = new CustomerService(repo);**

**service.printCustomer(101);**

**}**

**}**

**OUTPUT**

