**Exercise 1: Implementation of Singleton Pattern**

Program:

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();

logger1.log("First log message.");

Logger logger2 = Logger.getInstance();

logger2.log("Second log message.");

if (logger1 == logger2) {

System.out.println("Both logger instances are the same (singleton confirmed).");

} else {

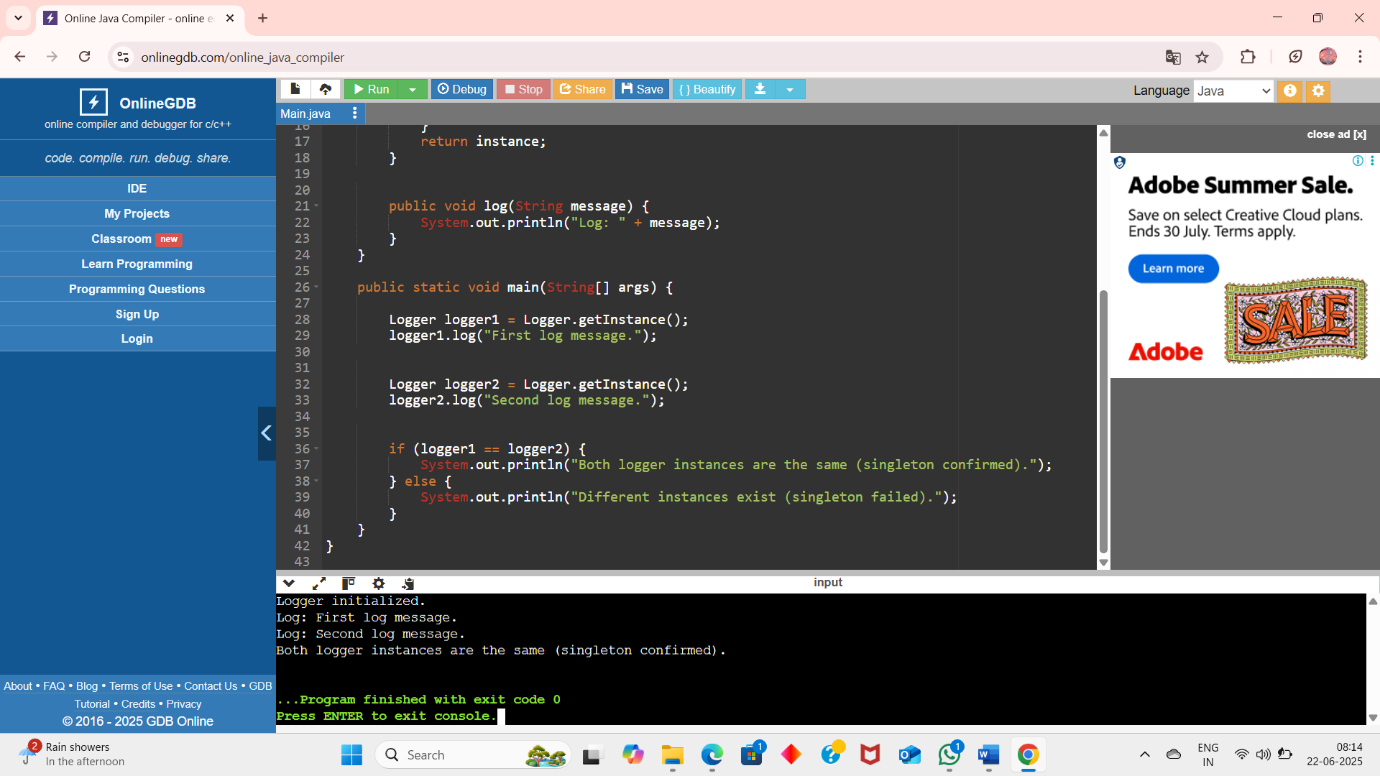
System.out.println("Different instances exist (singleton failed).");

}

}

}

Output:



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

Program:

interface Document {

void open();

}

class WordDocument implements Document {

public void open() {

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

}

}

class PdfDocument implements Document {

public void open() {

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

}

}

class ExcelDocument implements Document {

public void open() {

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

word.open();

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdf = pdfFactory.createDocument();

pdf.open();

DocumentFactory excelFactory = new ExcelDocumentFactory();

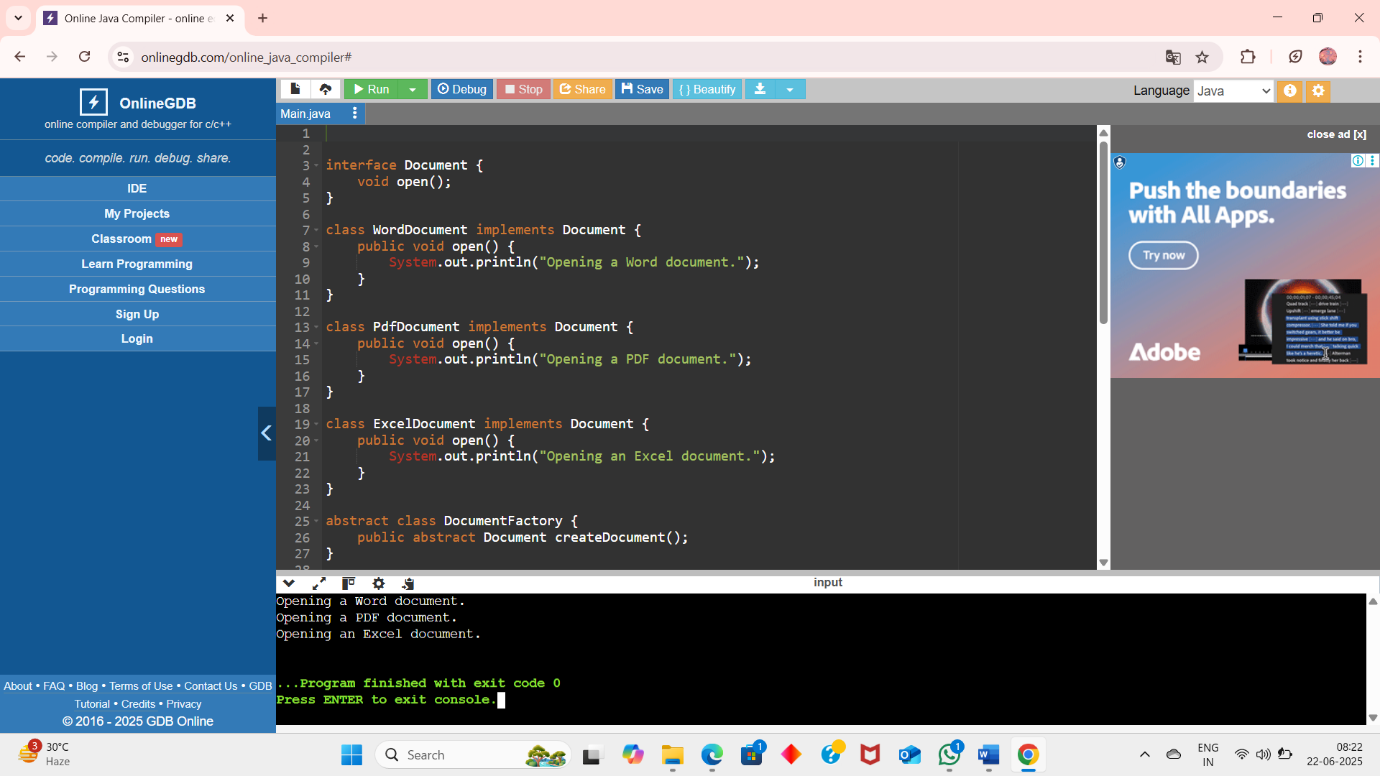
Document excel = excelFactory.createDocument();

excel.open();

}

}

Output:



**Exercise 3: Implementing the Builder Pattern**

Program:

class Computer {

private String CPU;

private String RAM;

private String storage;

private String graphicsCard;

private boolean isBluetoothEnabled;

private boolean isWifiEnabled;

private Computer(Builder builder) {

this.CPU = builder.CPU;

this.RAM = builder.RAM;

this.storage = builder.storage;

this.graphicsCard = builder.graphicsCard;

this.isBluetoothEnabled = builder.isBluetoothEnabled;

this.isWifiEnabled = builder.isWifiEnabled;

}

public static class Builder {

private String CPU;

private String RAM;

private String storage;

private String graphicsCard;

private boolean isBluetoothEnabled;

private boolean isWifiEnabled;

public Builder(String CPU, String RAM) {

this.CPU = CPU;

this.RAM = RAM;

}

public Builder setStorage(String storage) {

this.storage = storage;

return this;

}

public Builder setGraphicsCard(String graphicsCard) {

this.graphicsCard = graphicsCard;

return this;

}

public Builder setBluetoothEnabled(boolean isBluetoothEnabled) {

this.isBluetoothEnabled = isBluetoothEnabled;

return this;

}

public Builder setWifiEnabled(boolean isWifiEnabled) {

this.isWifiEnabled = isWifiEnabled;

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);

System.out.println("Graphics Card: " + graphicsCard);

System.out.println("Bluetooth Enabled: " + isBluetoothEnabled);

System.out.println("WiFi Enabled: " + isWifiEnabled);

System.out.println("--------------------------------");

}

}

public class Main {

public static void main(String[] args) {

Computer basicComputer = new Computer.Builder("Intel i5", "8GB")

.setStorage("256GB SSD")

.build();

Computer gamingComputer = new Computer.Builder("AMD Ryzen 9", "32GB")

.setStorage("1TB SSD")

.setGraphicsCard("NVIDIA RTX 4080")

.setBluetoothEnabled(true)

.setWifiEnabled(true)

.build();

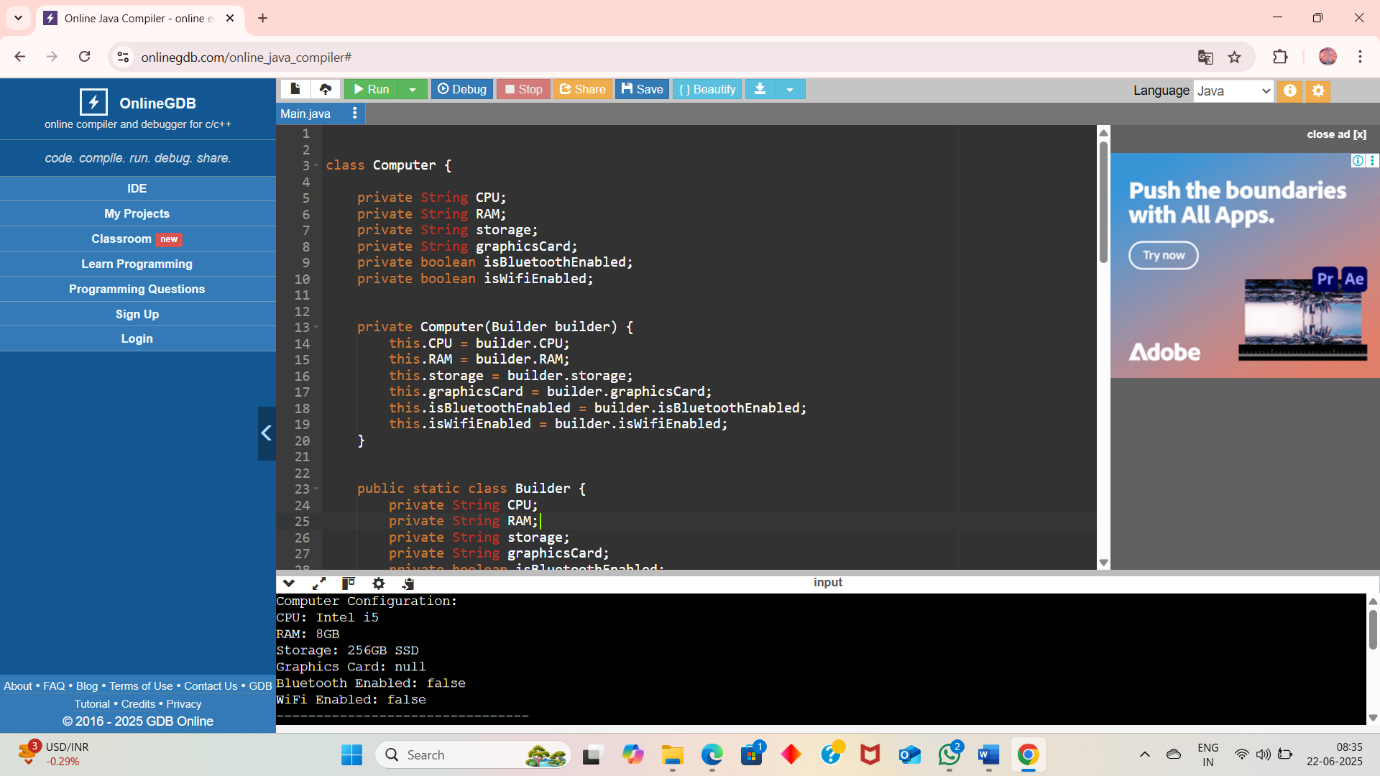
basicComputer.showConfig();

gamingComputer.showConfig();

}

}

Ouput:



**Exercise 4 : Implementing the Adapter Pattern**

Program:

interface PaymentProcessor {

void processPayment(double amount);

}

class PayPalGateway {

public void makePayment(double amount) {

System.out.println("Payment of $" + amount + " processed using PayPal.");

}

}

class StripeGateway {

public void executeTransaction(double amount) {

System.out.println("Payment of $" + amount + " processed using Stripe.");

}

}

class SquareGateway {

public void performPayment(double amount) {

System.out.println("Payment of $" + amount + " processed using Square.");

}

}

class PayPalAdapter implements PaymentProcessor {

private PayPalGateway payPal;

public PayPalAdapter(PayPalGateway payPal) {

this.payPal = payPal;

}

public void processPayment(double amount) {

payPal.makePayment(amount);

}

}

class StripeAdapter implements PaymentProcessor {

private StripeGateway stripe;

public StripeAdapter(StripeGateway stripe) {

this.stripe = stripe;

}

public void processPayment(double amount) {

stripe.executeTransaction(amount);

}

}

class SquareAdapter implements PaymentProcessor {

private SquareGateway square;

public SquareAdapter(SquareGateway square) {

this.square = square;

}

public void processPayment(double amount) {

square.performPayment(amount);

}

}

public class Main {

public static void main(String[] args) {

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

paypalProcessor.processPayment(100.50);

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

stripeProcessor.processPayment(250.75);

PaymentProcessor squareProcessor = new SquareAdapter(new SquareGateway());

squareProcessor.processPayment(399.99);

}

}

Output:



**Exercise 5 : Implementing the Decorator pattern**

Program :

interface Notifier {

void send(String message);

}

class EmailNotifier implements Notifier {

public void send(String message) {

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

}

}

abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

public void send(String message) {

notifier.send(message);

}

}

class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

sendSMS(message);

}

private void sendSMS(String message) {

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

}

}

class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

sendSlack(message);

}

private void sendSlack(String message) {

System.out.println("Slack message sent: " + message);

}

}

public class Main {

public static void main(String[] args) {

Notifier notifier = new EmailNotifier();

notifier = new SMSNotifierDecorator(notifier);

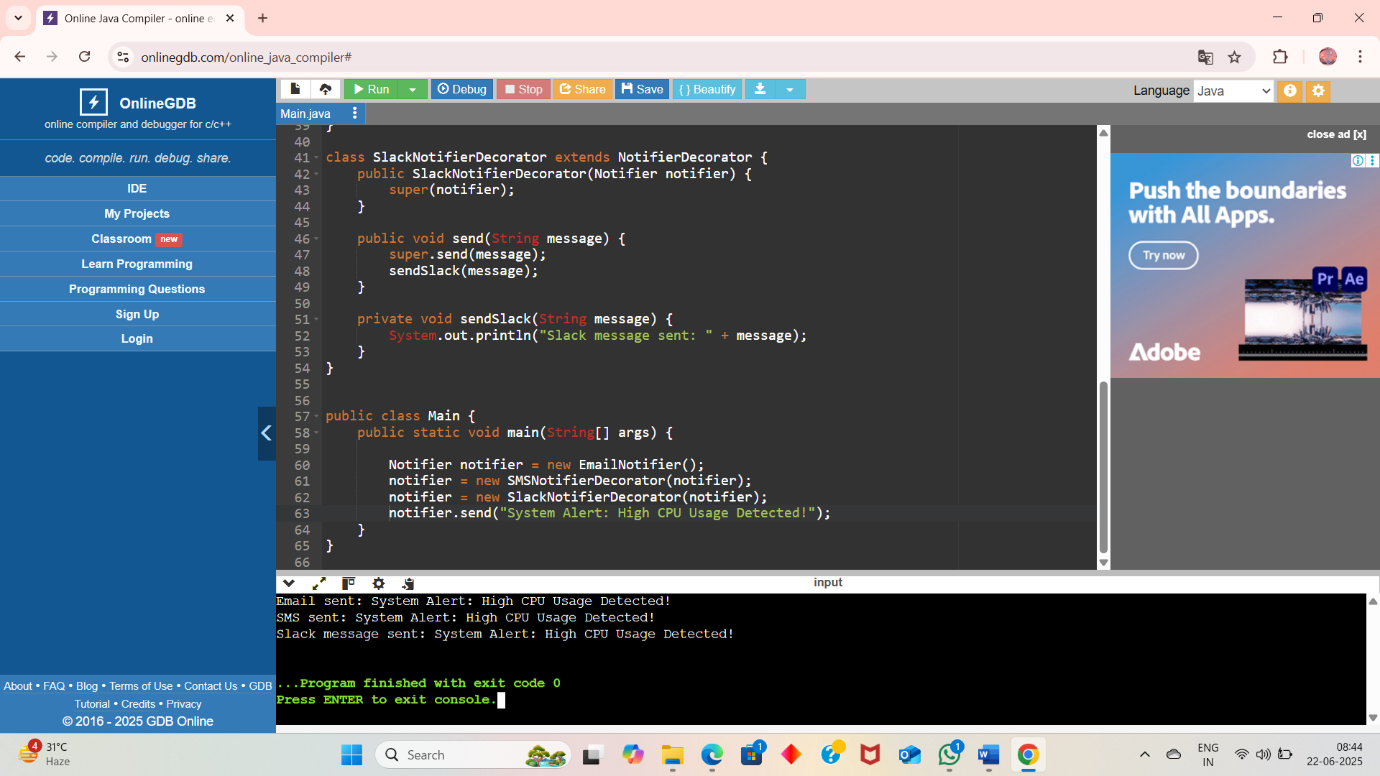
notifier = new SlackNotifierDecorator(notifier);

notifier.send("System Alert: High CPU Usage Detected!");

}

}

Output:



**Exercise 6 : Implementing the Proxy Pattern**

Program:

interface Image {

void display();

}

class RealImage implements Image {

private String filename;

public RealImage(String filename) {

this.filename = filename;

loadFromRemoteServer();

}

private void loadFromRemoteServer() {

System.out.println("Loading image from remote server: " + filename);

}

public void display() {

System.out.println("Displaying image: " + 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); // Lazy initialization

} else {

System.out.println("Using cached image: " + filename);

}

realImage.display();

}

}

public class Main {

public static void main(String[] args) {

Image image1 = new ProxyImage("photo1.jpg");

Image image2 = new ProxyImage("photo2.jpg");

image1.display();

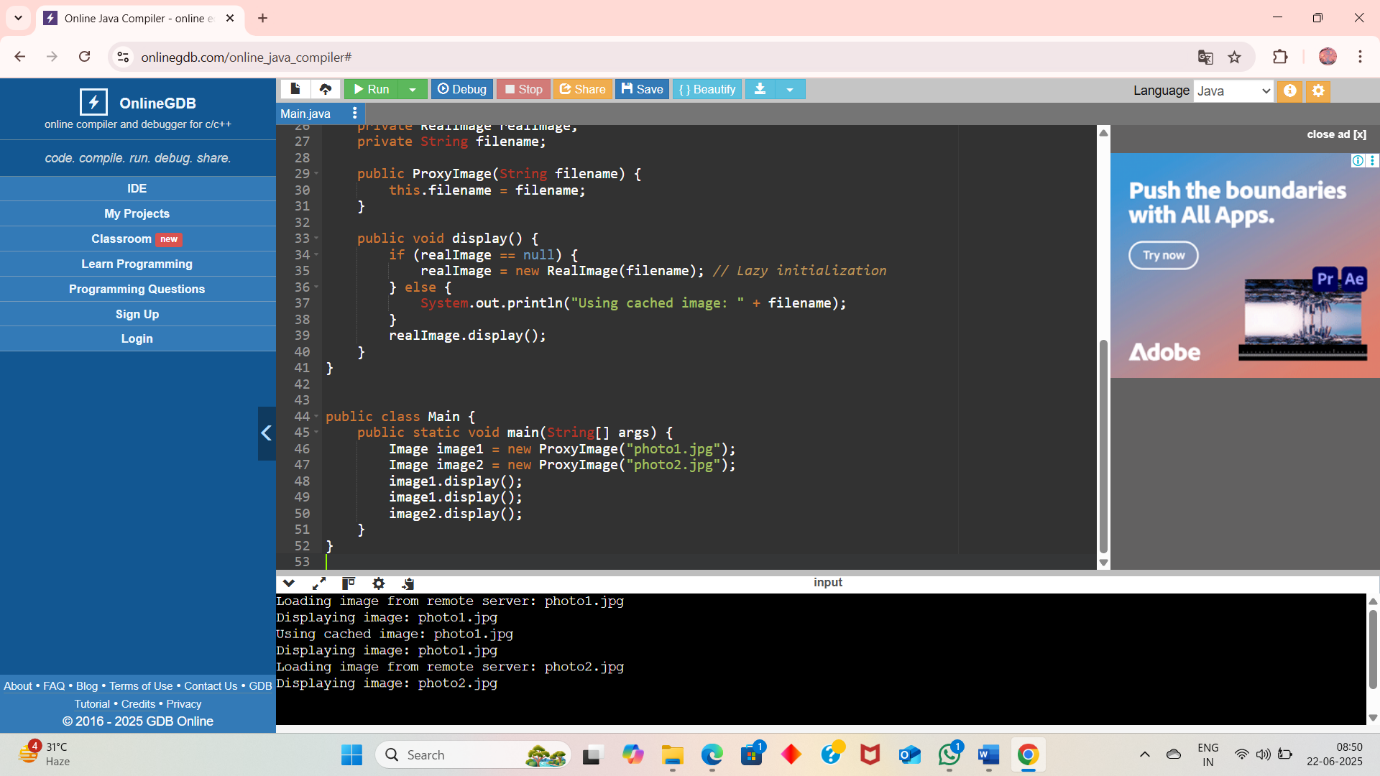
image1.display();

image2.display();

}

}

Output:



**Exercise 7 : Implementing the Observer Pattern**

Program:

import java.util.ArrayList;

import java.util.List;

interface Stock {

void registerObserver(Observer o);

void removeObserver(Observer o);

void notifyObservers();

}

interface Observer {

void update(String stockName, double price);

}

class StockMarket implements Stock {

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

private String stockName;

private double price;

public void setStockPrice(String stockName, double price) {

this.stockName = stockName;

this.price = price;

notifyObservers();

}

public void registerObserver(Observer o) {

observers.add(o);

}

public void removeObserver(Observer o) {

observers.remove(o);

}

public void notifyObservers() {

for (Observer o : observers) {

o.update(stockName, price);

}

}

}

class MobileApp implements Observer {

private String user;

public MobileApp(String user) {

this.user = user;

}

public void update(String stockName, double price) {

System.out.println("MobileApp [" + user + "] - " + stockName + " price updated to $" + price);

}

}

class WebApp implements Observer {

private String user;

public WebApp(String user) {

this.user = user;

}

public void update(String stockName, double price) {

System.out.println("WebApp [" + user + "] - " + stockName + " price updated to $" + price);

}

}

public class Main {

public static void main(String[] args) {

StockMarket stockMarket = new StockMarket();

Observer mobileUser1 = new MobileApp("Alice");

Observer webUser1 = new WebApp("Bob");

stockMarket.registerObserver(mobileUser1);

stockMarket.registerObserver(webUser1);

stockMarket.setStockPrice("AAPL", 189.25);

stockMarket.setStockPrice("GOOGL", 2873.50);

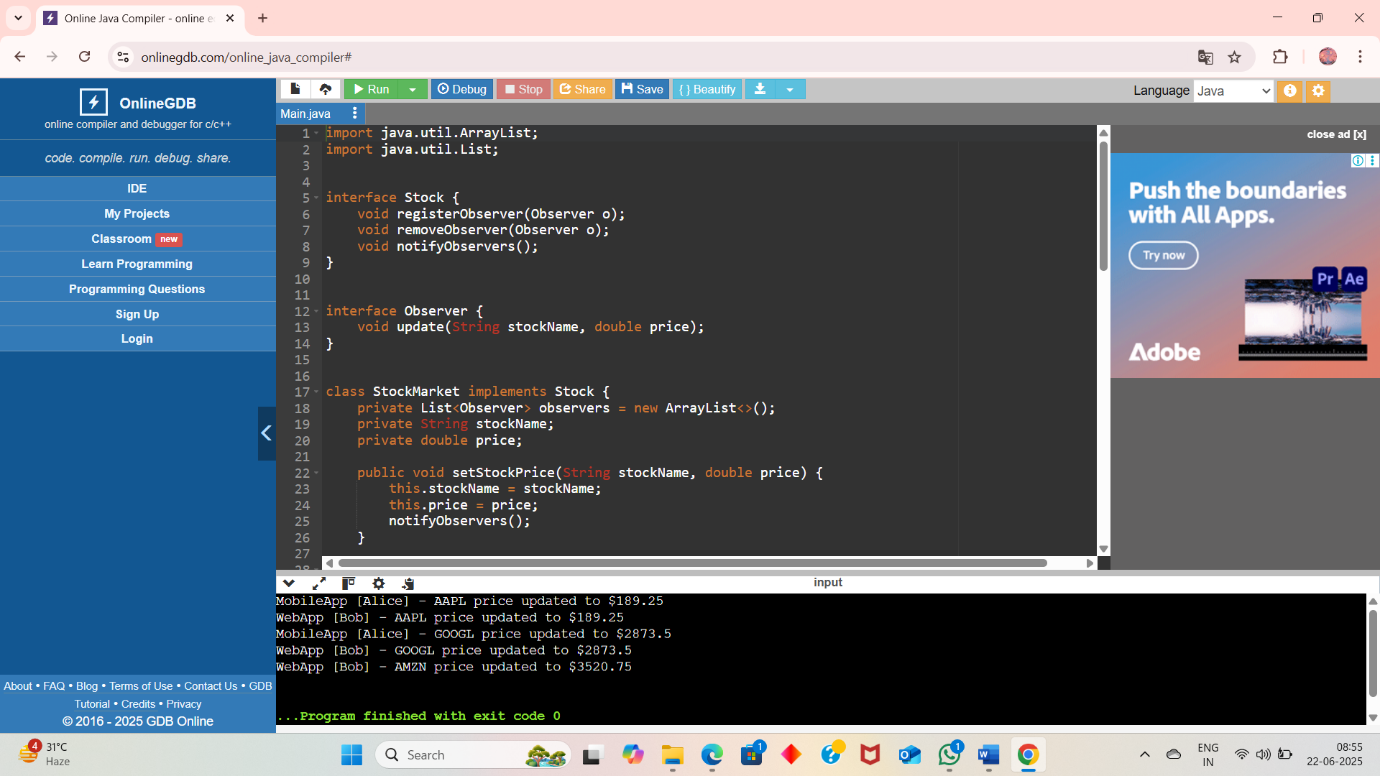
stockMarket.removeObserver(mobileUser1);

stockMarket.setStockPrice("AMZN", 3520.75);

}

}

Output:



**Exercise 8 : Implementing the Strategy Pattern**

Program:

interface PaymentStrategy {

void pay(double amount);

}

class CreditCardPayment implements PaymentStrategy {

private String cardNumber;

public CreditCardPayment(String cardNumber) {

this.cardNumber = cardNumber;

}

public void pay(double amount) {

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

}

}

class PayPalPayment implements PaymentStrategy {

private String email;

public PayPalPayment(String email) {

this.email = email;

}

public void pay(double amount) {

System.out.println("Paid $" + amount + " using PayPal account: " + email);

}

}

class PaymentContext {

private PaymentStrategy strategy;

public void setPaymentStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void processPayment(double amount) {

if (strategy != null) {

strategy.pay(amount);

} else {

System.out.println("No payment strategy selected.");

}

}

}

public class Main {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

context.setPaymentStrategy(new CreditCardPayment("1234-5678-9012-3456"));

context.processPayment(150.00);

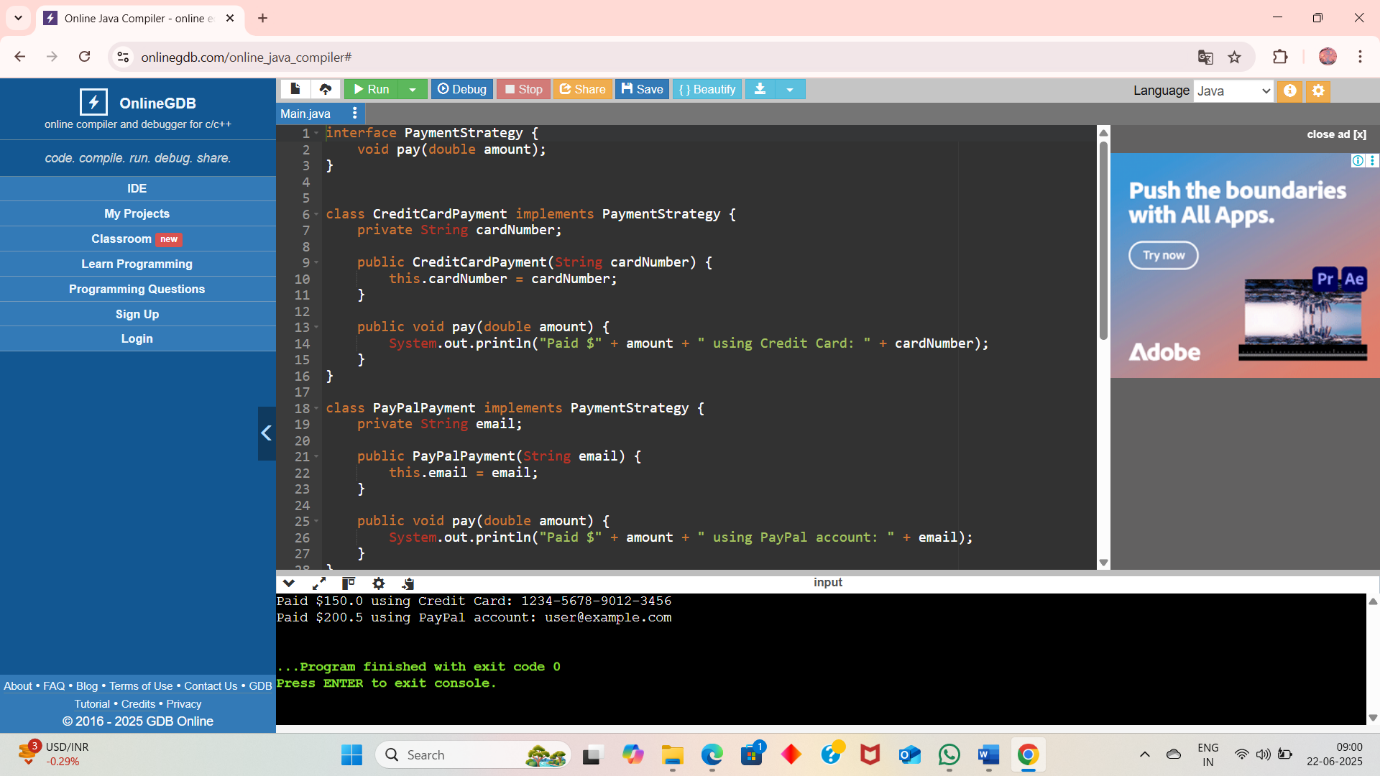
context.setPaymentStrategy(new PayPalPayment("user@example.com"));

context.processPayment(200.50);

}

}

Output:



**Exercise 9 : Implementing the Command Pattern**

Program:

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();

} else {

System.out.println("No command set.");

}

}

}

public class Main {

public static void main(String[] args) {

Light livingRoomLight = new Light();

Command lightOn = new LightOnCommand(livingRoomLight);

Command lightOff = new LightOffCommand(livingRoomLight);

RemoteControl remote = new RemoteControl();

remote.setCommand(lightOn);

remote.pressButton();

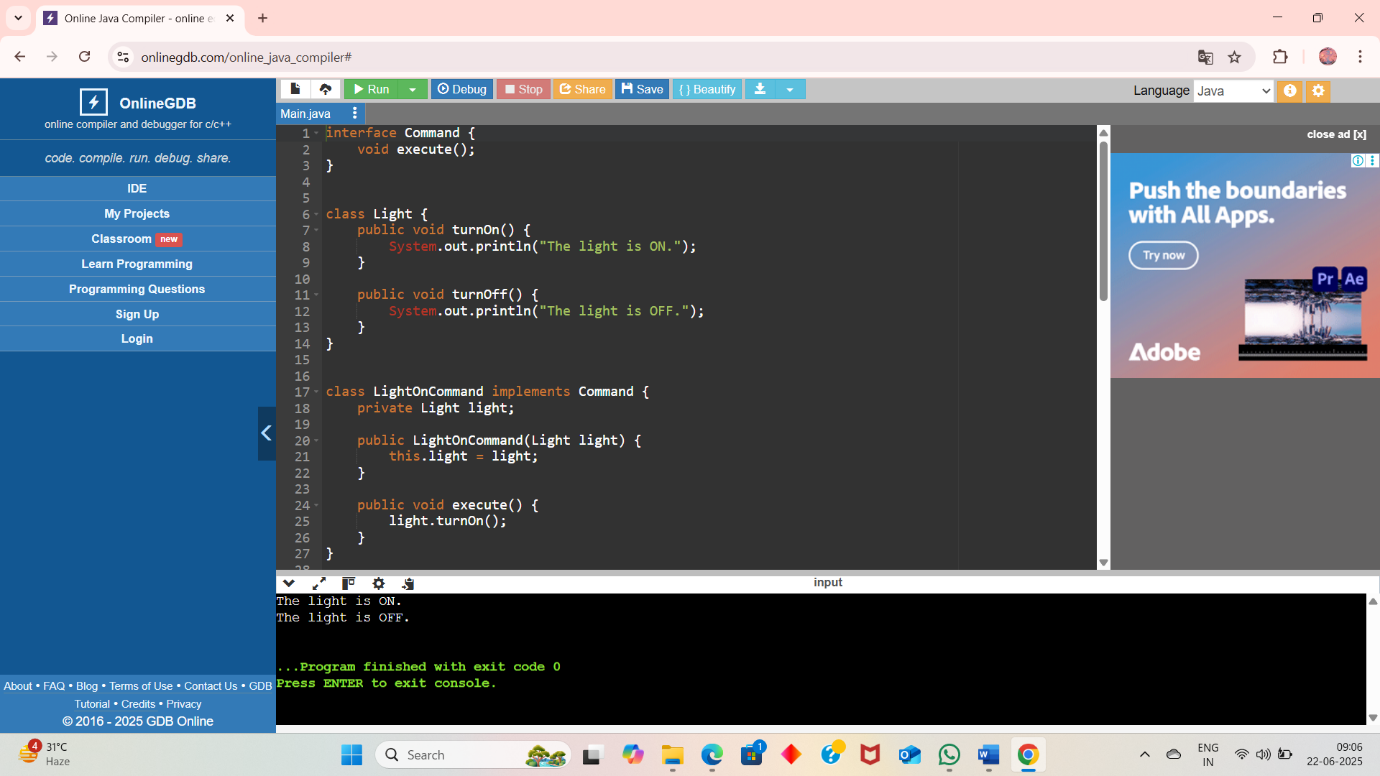
remote.setCommand(lightOff);

remote.pressButton();

}

}

Output:



**Exercise 10 : Implementing the MVC Pattern**

Program:

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 void setId(String id) {

this.id = 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);

System.out.println("-------------------------");

}

}

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 setStudentId(String id) {

model.setId(id);

}

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 student = new Student("Alice", "S101", "A");

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

controller.updateView();

controller.setStudentName("Bob");

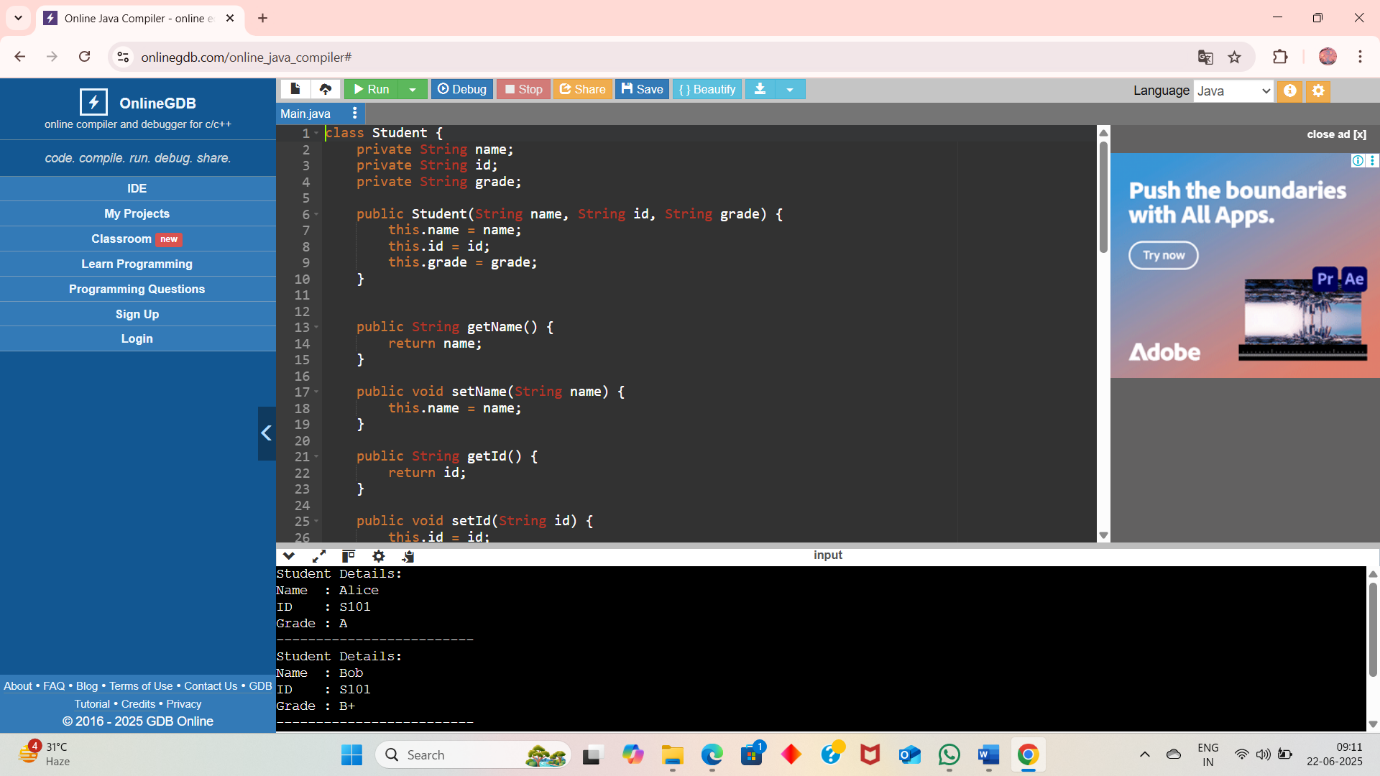
controller.setStudentGrade("B+");

controller.updateView();

}

}

Ouput:



**Exercise 11 : Implementing Dependency Injection**

Program:

interface CustomerRepository {

Customer findCustomerById(int id);

}

class Customer {

private int id;

private String name;

public Customer(int id, String name) {

this.id = id;

this.name = name;

}

public int getId() { return id; }

public String getName() { return name; }

}

class CustomerRepositoryImpl implements CustomerRepository {

private Customer[] customers = {

new Customer(1, "Alice"),

new Customer(2, "Bob"),

new Customer(3, "Charlie")

};

public Customer findCustomerById(int id) {

for (Customer c : customers) {

if (c.getId() == id) return c;

}

return null;

}

}

class CustomerService {

private CustomerRepository repository;

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void printCustomerDetails(int id) {

Customer c = repository.findCustomerById(id);

if (c != null) {

System.out.println("Customer found: ID=" + c.getId() + ", Name=" + c.getName());

} else {

System.out.println("Customer with ID " + id + " not found.");

}

}

}

public class Main {

public static void main(String[] args) {

CustomerRepository repository = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repository);

service.printCustomerDetails(2);

service.printCustomerDetails(5);

}

}

Ouput:

