**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.

class EventLogger {

private static EventLogger instance;

private EventLogger() {

}

public static EventLogger getInstance() {

if (instance == null) {

instance = new EventLogger();

}

return instance;

}

public void logMessage(String message) {

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

}

}

public class Main {

public static void main(String[] args) {

EventLogger logger1 = EventLogger.getInstance();

EventLogger logger2 = EventLogger.getInstance();

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

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

if (logger1 == logger2) {

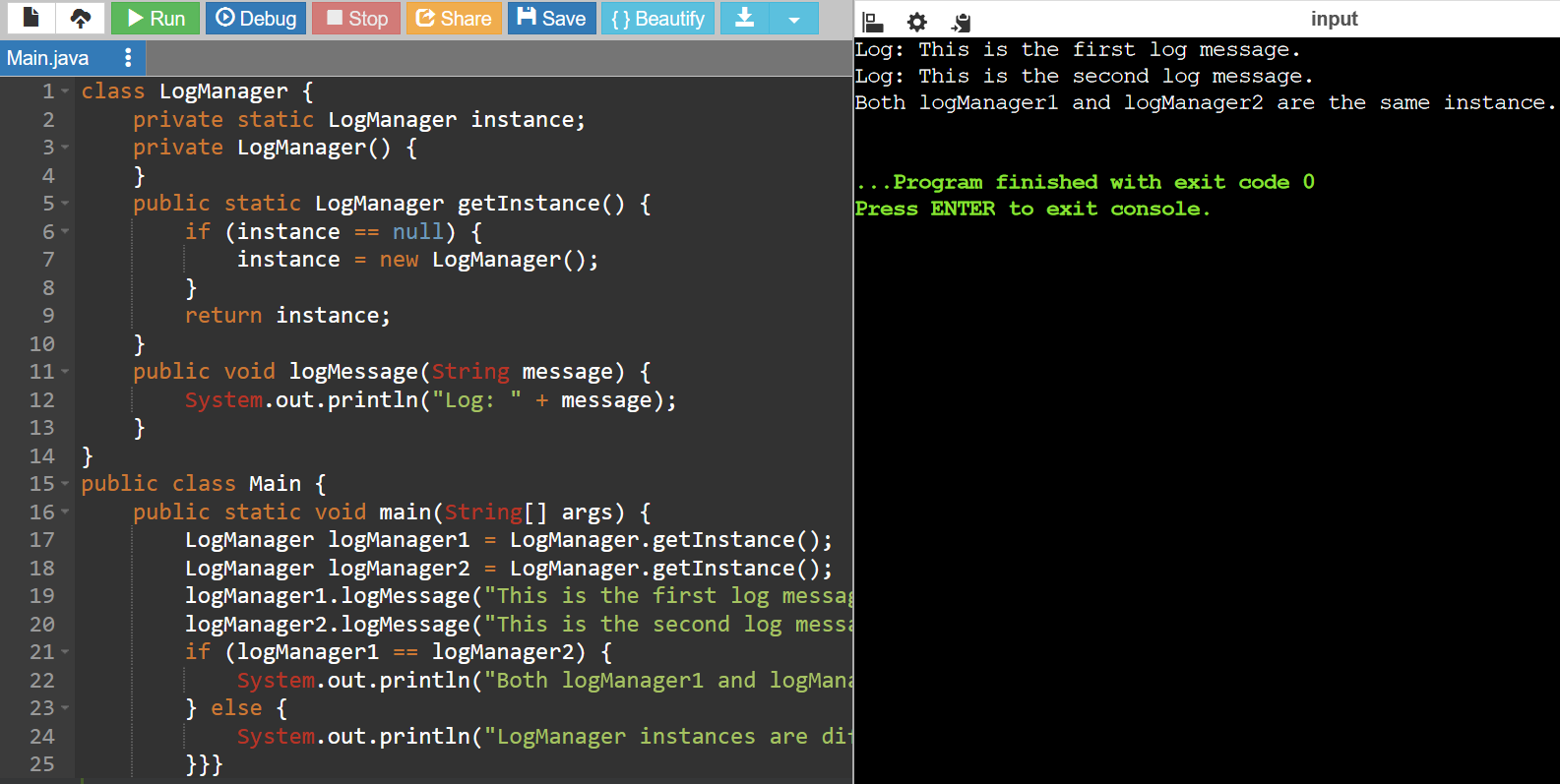
System.out.println("Both logger1 and logger2 are the same instance.");

} else {

System.out.println("Logger instances are different.");

}

}}



**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.

interface Document {

void open();

void close();

}

class TextDocument implements Document {

public void open() {

System.out.println("Opening Text document...");

}

public void close() {

System.out.println("Closing Text document...");

}

}

class PortableDocument implements Document {

public void open() {

System.out.println("Opening Portable document...");

}

public void close() {

System.out.println("Closing Portable document...");

}

}

class SpreadsheetDocument implements Document {

public void open() {

System.out.println("Opening Spreadsheet document...");

}

public void close() {

System.out.println("Closing Spreadsheet document...");

}

}

abstract class DocumentFactory {

public abstract Document createDocument();

}

class TextDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new TextDocument();

}

}

class PortableDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new PortableDocument();

}

}

class SpreadsheetDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new SpreadsheetDocument();

}

}

public class Main {

public static void main(String[] args) {

DocumentFactory textFactory = new TextDocumentFactory();

Document textDoc = textFactory.createDocument();

textDoc.open();

textDoc.close();

DocumentFactory portableFactory = new PortableDocumentFactory();

Document portableDoc = portableFactory.createDocument();

portableDoc.open();

portableDoc.close();

DocumentFactory spreadsheetFactory = new SpreadsheetDocumentFactory();

Document spreadsheetDoc = spreadsheetFactory.createDocument();

spreadsheetDoc.open();

spreadsheetDoc.close();

}}

A screenshot of a computer screen

Description automatically generated

**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.

public class Main {

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;

}

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 static void main(String[] args) {

Computer gamingPC = new Computer.Builder()

.setCPU("Intel Core i9")

.setRAM("32GB")

.setStorage("1TB SSD")

.build();

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

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

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

}}

A screenshot of a computer screen

Description automatically generated

**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.

interface PaymentGateway {

void executePayment(double amount);

}

class PayPalService {

public void initiatePayment(double amount) {

System.out.println("Processing payment of Rs." + amount + " through PayPal.");

}

}

class StripeService {

public void performPayment(double amount) {

System.out.println("Processing payment of Rs." + amount + " through Stripe.");

}

}

class AmazonPayService {

public void handlePayment(double amount) {

System.out.println("Processing payment of Rs." + amount + " through Amazon Pay.");

}

}

class PayPalAdapter implements PaymentGateway {

private PayPalService payPalService;

public PayPalAdapter(PayPalService payPalService) {

this.payPalService = payPalService;

}

public void executePayment(double amount) {

payPalService.initiatePayment(amount);

}

}

class StripeAdapter implements PaymentGateway {

private StripeService stripeService;

public StripeAdapter(StripeService stripeService) {

this.stripeService = stripeService;

}

public void executePayment(double amount) {

stripeService.performPayment(amount);

}

}

class AmazonPayAdapter implements PaymentGateway {

private AmazonPayService amazonPayService;

public AmazonPayAdapter(AmazonPayService amazonPayService) {

this.amazonPayService = amazonPayService;

}

public void executePayment(double amount) {

amazonPayService.handlePayment(amount);

}

}

public class Main {

public static void main(String[] args) {

PayPalService payPalService = new PayPalService();

StripeService stripeService = new StripeService();

AmazonPayService amazonPayService = new AmazonPayService();

PaymentGateway payPalAdapter = new PayPalAdapter(payPalService);

PaymentGateway stripeAdapter = new StripeAdapter(stripeService);

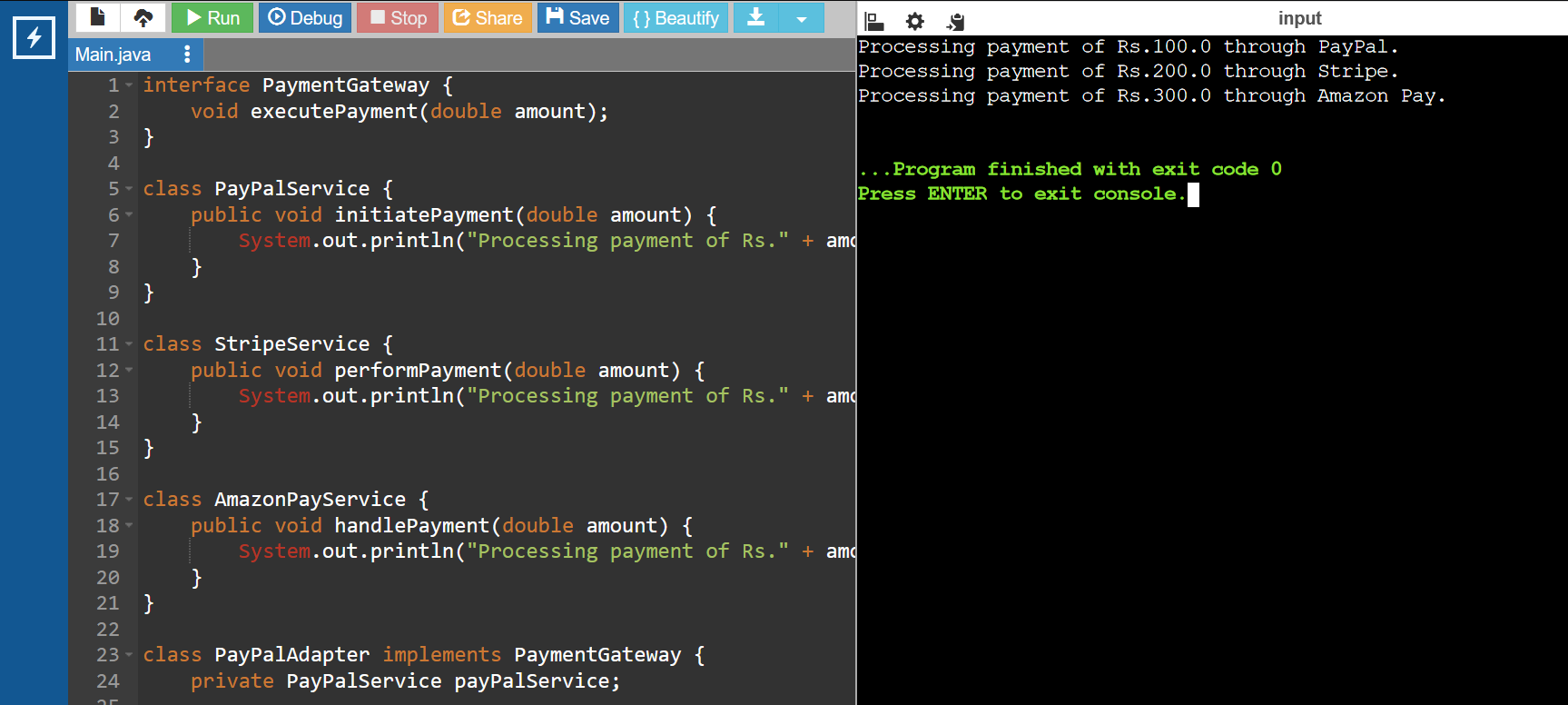
PaymentGateway amazonPayAdapter = new AmazonPayAdapter(amazonPayService);

payPalAdapter.executePayment(100.00);

stripeAdapter.executePayment(200.00);

amazonPayAdapter.executePayment(300.00);

}}



**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

interface Notifier {

void send(String message);

}

class EmailAlert implements Notifier {

public void send(String message) {

System.out.println("Sending email notification: " + message);

}

}

abstract class AlertDecorator implements Notifier {

protected Notifier notifier;

public AlertDecorator(Notifier notifier) {

this.notifier = notifier;

}

public void send(String message) {

notifier.send(message);

}

}

class SMSAlertDecorator extends AlertDecorator {

public SMSAlertDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

notifier.send(message);

sendSMS(message);

}

private void sendSMS(String message) {

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

}

}

class SlackAlertDecorator extends AlertDecorator {

public SlackAlertDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

notifier.send(message);

sendSlack(message);

}

private void sendSlack(String message) {

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

}

}

public class Main {

public static void main(String[] args) {

Notifier emailAlert = new EmailAlert();

Notifier smsAlert = new SMSAlertDecorator(emailAlert);

Notifier slackAlert = new SlackAlertDecorator(smsAlert);

slackAlert.send("Hello, this is a test notification!");

}}

A screenshot of a computer

Description automatically generated

**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

interface Picture {

void show();

}

class ActualImage implements Picture {

private String imageName;

public ActualImage(String imageName) {

this.imageName = imageName;

loadImageFromServer();

}

private void loadImageFromServer() {

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

}

public void show() {

System.out.println("Displaying image: " + imageName);

}

}

class ImageProxy implements Picture {

private String imageName;

private ActualImage actualImage;

public ImageProxy(String imageName) {

this.imageName = imageName;

}

public void show() {

if (actualImage == null) {

actualImage = new ActualImage(imageName);

}

actualImage.show();

}

}

public class Main {

public static void main(String[] args) {

Picture picture1 = new ImageProxy("picture1.png");

Picture picture2 = new ImageProxy("picture2.png");

picture1.show();

System.out.println("");

picture1.show();

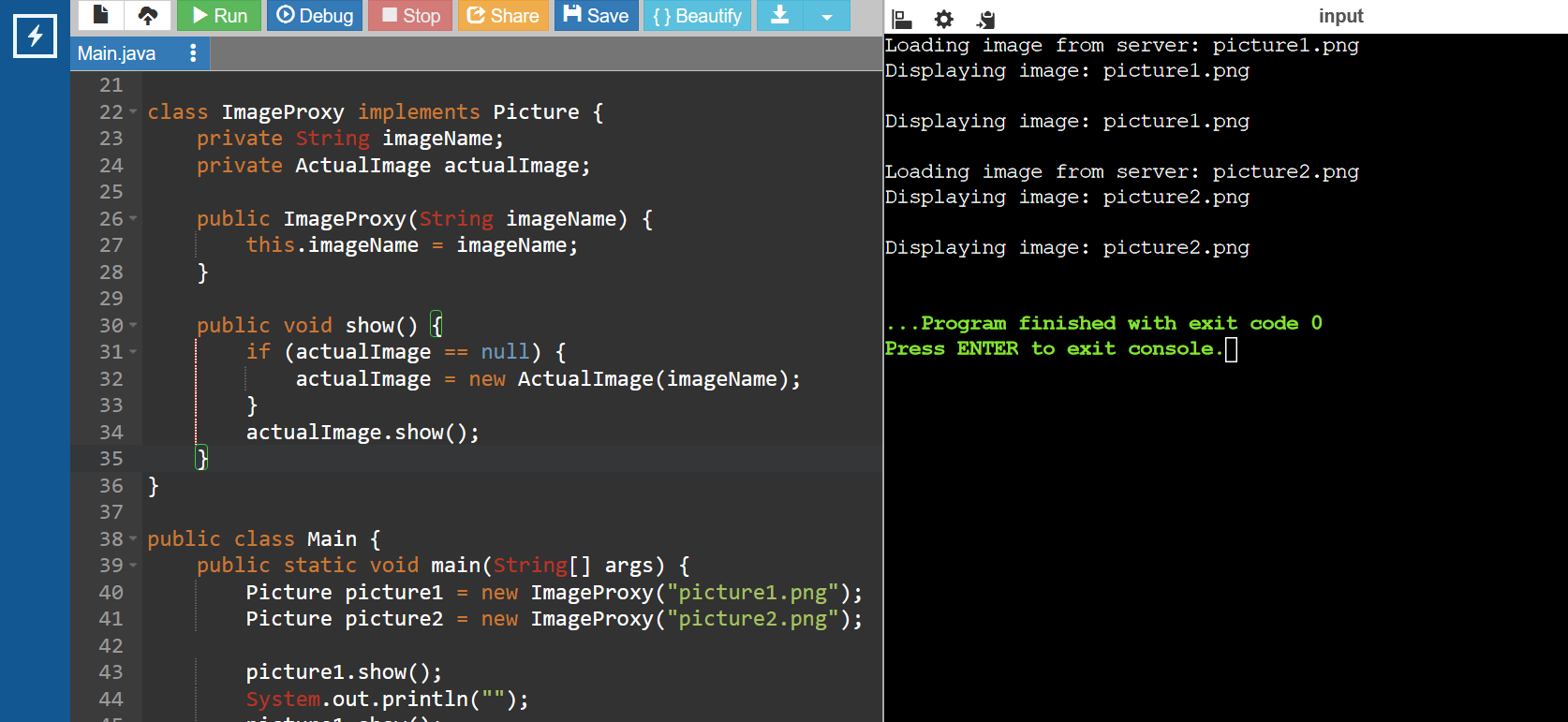
System.out.println("");

picture2.show();

System.out.println("");

picture2.show();

}}



**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.

import java.util.ArrayList;

import java.util.List;

interface Market {

void addSubscriber(Subscriber s);

void removeSubscriber(Subscriber s);

void notifySubscribers();

}

class StockExchange implements Market {

private List<Subscriber> subscribers;

private double price;

public StockExchange() {

this.subscribers = new ArrayList<>();

}

public void addSubscriber(Subscriber s) {

subscribers.add(s);

}

public void removeSubscriber(Subscriber s) {

subscribers.remove(s);

}

public void notifySubscribers() {

for (Subscriber s : subscribers) {

s.update(price);

}

}

public void setPrice(double price) {

this.price = price;

notifySubscribers();

}

}

interface Subscriber {

void update(double price);

}

class MobileClient implements Subscriber {

private String clientName;

public MobileClient(String clientName) {

this.clientName = clientName;

}

public void update(double price) {

System.out.println(clientName + " received price update: " + price);

}

}

class WebClient implements Subscriber {

private String clientName;

public WebClient(String clientName) {

this.clientName = clientName;

}

public void update(double price) {

System.out.println(clientName + " received price update: " + price);

}

}

public class Main {

public static void main(String[] args) {

StockExchange stockExchange = new StockExchange();

Subscriber mobileClient = new MobileClient("MobileClient");

Subscriber webClient = new WebClient("WebClient");

stockExchange.addSubscriber(mobileClient);

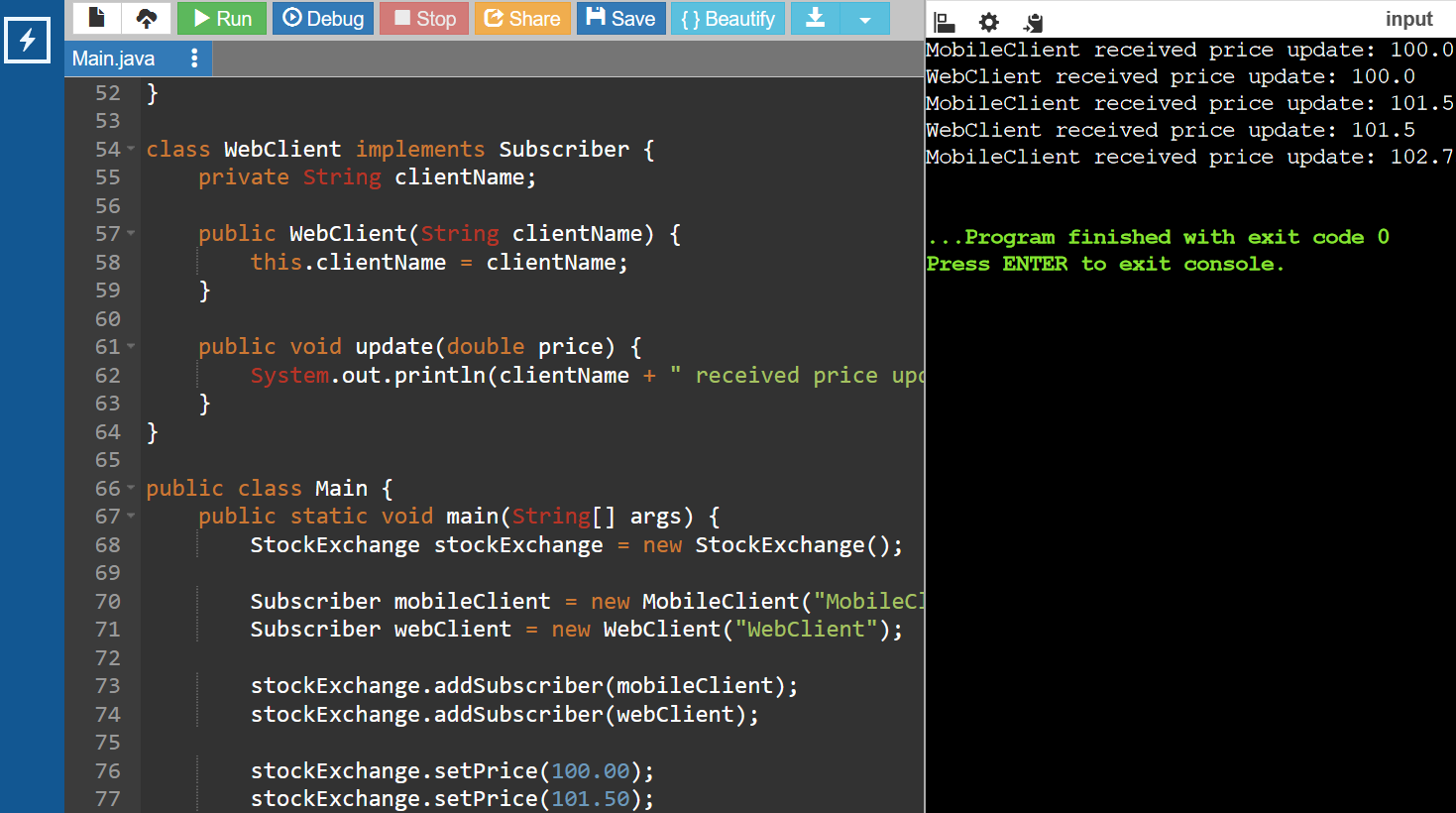
stockExchange.addSubscriber(webClient);

stockExchange.setPrice(100.00);

stockExchange.setPrice(101.50);

stockExchange.removeSubscriber(webClient);

stockExchange.setPrice(102.75);



**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.

// Step 2: Define Strategy Interface

interface PaymentMethod {

void processPayment(double amount);

}

class CardPayment implements PaymentMethod {

private String holderName;

private String cardNumber;

private String cvv;

private String expirationDate;

public CardPayment(String holderName, String cardNumber, String cvv, String expirationDate) {

this.holderName = holderName;

this.cardNumber = cardNumber;

this.cvv = cvv;

this.expirationDate = expirationDate;

}

public void processPayment(double amount) {

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

}

}

class PayPalProcessor implements PaymentMethod {

private String accountEmail;

private String accountPassword;

public PayPalProcessor(String accountEmail, String accountPassword) {

this.accountEmail = accountEmail;

this.accountPassword = accountPassword;

}

public void processPayment(double amount) {

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

}

}

class PaymentHandler {

private PaymentMethod paymentMethod;

public void setPaymentMethod(PaymentMethod paymentMethod) {

this.paymentMethod = paymentMethod;

}

public void executePayment(double amount) {

paymentMethod.processPayment(amount);

}

}

public class Main {

public static void main(String[] args) {

PaymentHandler handler = new PaymentHandler();

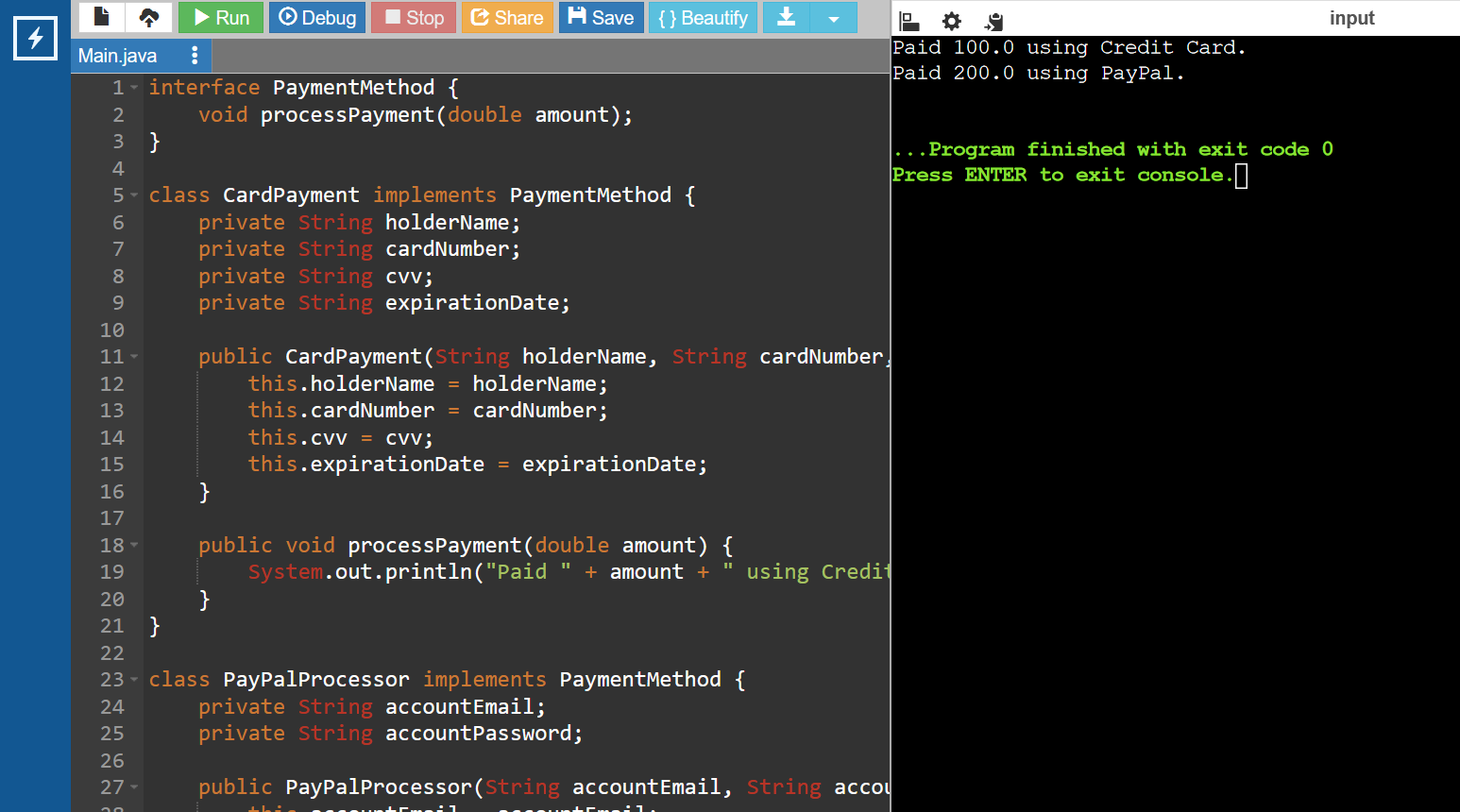
handler.setPaymentMethod(new CardPayment("John Doe", "1234567890123456", "123", "12/23"));

handler.executePayment(100.0);

handler.setPaymentMethod(new PayPalProcessor("john.doe@example.com", "password123"));

handler.executePayment(200.0);

}}



**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.

interface Action {

void execute();

}

class TurnOnLightAction implements Action {

private Illumination light;

public TurnOnLightAction(Illumination light) {

this.light = light;

}

@Override

public void execute() {

light.switchOn();

}

}

class TurnOffLightAction implements Action {

private Illumination light;

public TurnOffLightAction(Illumination light) {

this.light = light;

}

@Override

public void execute() {

light.switchOff();

}

}

class Illumination {

public void switchOn() {

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

}

public void switchOff() {

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

}

}

class Controller {

private Action action;

public void setAction(Action action) {

this.action = action;

}

public void pressButton() {

action.execute();

}

}

public class Main {

public static void main(String[] args) {

Illumination livingRoomLight = new Illumination();

Action lightOn = new TurnOnLightAction(livingRoomLight);

Action lightOff = new TurnOffLightAction(livingRoomLight);

Controller remote = new Controller();

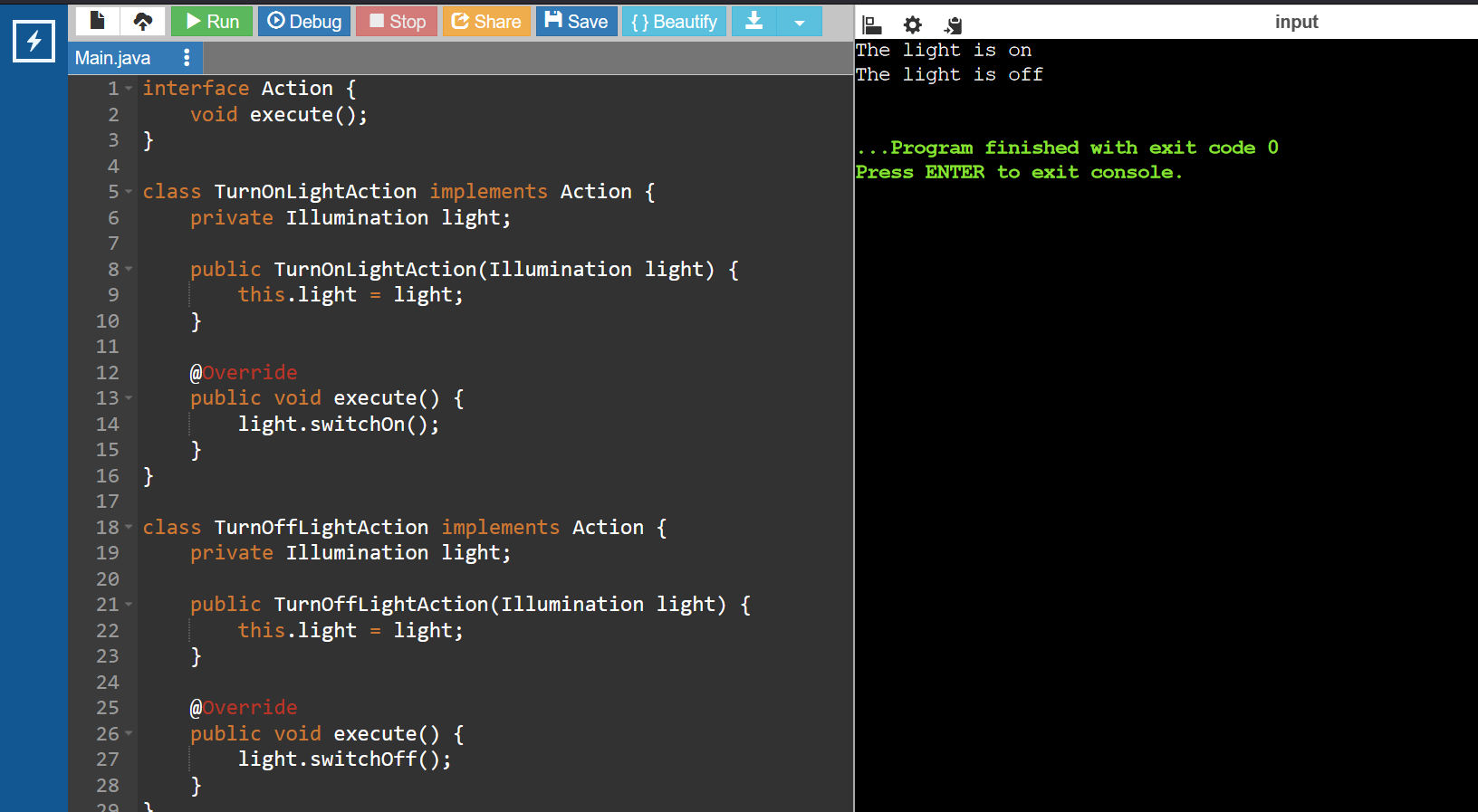
remote.setAction(lightOn);

remote.pressButton();

remote.setAction(lightOff);

remote.pressButton();

}}



**Exercise 10: Implementing the MVC Pattern**

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

class Scholar {

private String id;

private String name;

private String grade;

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

this.id = id;

this.name = name;

this.grade = grade;

}

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getGrade() {

return grade;

}

public void setGrade(String grade) {

this.grade = grade;

}

}

class ScholarView {

public void displayScholarDetails(String scholarName, String scholarId, String scholarGrade) {

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

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

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

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

}

}

class ScholarController {

private Scholar model;

private ScholarView view;

public ScholarController(Scholar model, ScholarView view) {

this.model = model;

this.view = view;

}

public void setScholarName(String name) {

model.setName(name);

}

public String getScholarName() {

return model.getName();

}

public void setScholarId(String id) {

model.setId(id);

}

public String getScholarId() {

return model.getId();

}

public void setScholarGrade(String grade) {

model.setGrade(grade);

}

public String getScholarGrade() {

return model.getGrade();

}

public void updateView() {

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

}

}

public class Main {

public static void main(String[] args) {

Scholar model = new Scholar("1", "John Doe", "A");

ScholarView view = new ScholarView();

ScholarController controller = new ScholarController(model, view);

controller.updateView();

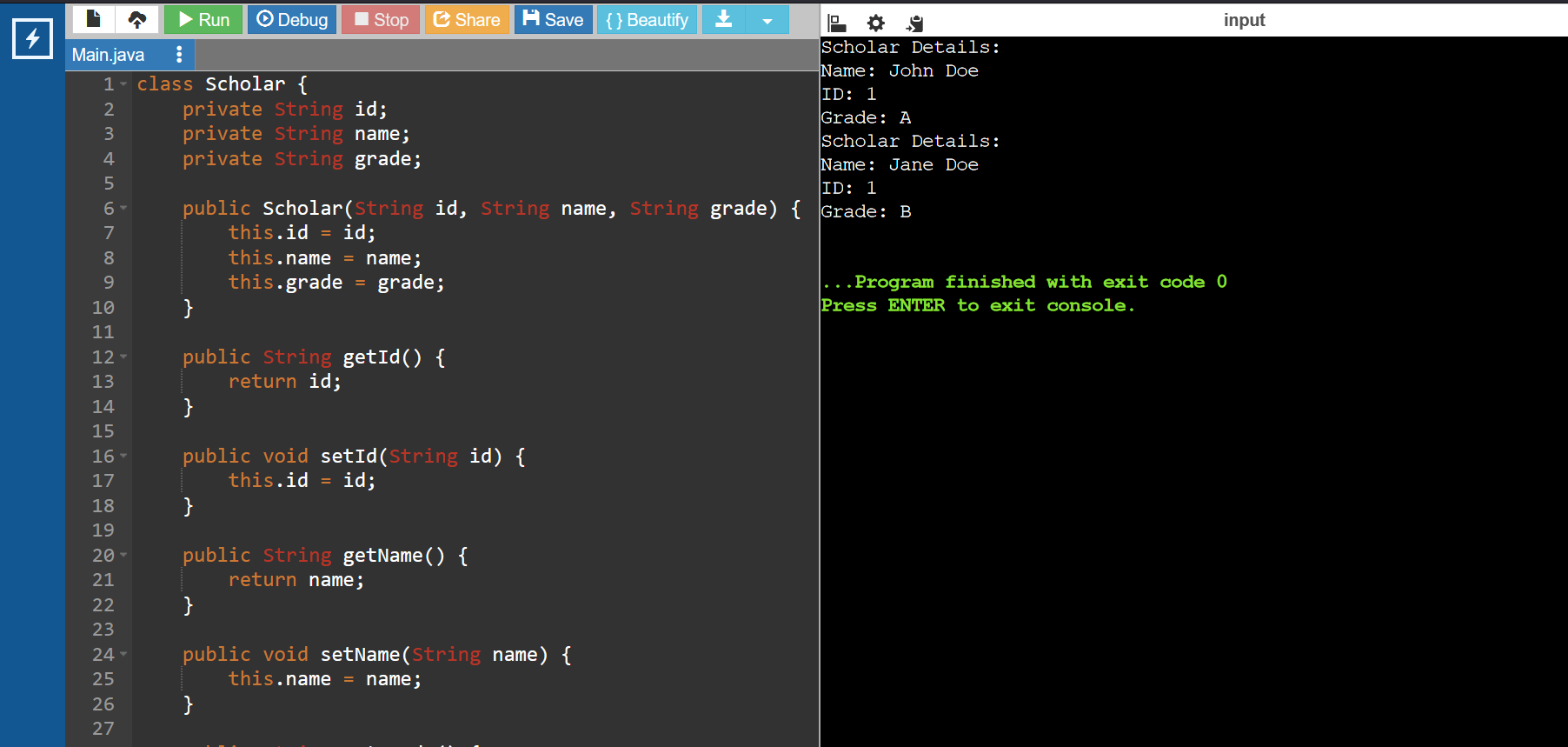
controller.setScholarName("Jane Doe");

controller.setScholarGrade("B");

controller.updateView();

}

}



**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.

interface ClientRepository {

String findClientById(String id);

}

class ClientRepositoryImpl implements ClientRepository {

public String findClientById(String id) {

if (id.equals("1")) {

return "John Doe";

} else {

return "Client not found";

}

}

}

class ClientService {

private ClientRepository clientRepository;

public ClientService(ClientRepository clientRepository) {

this.clientRepository = clientRepository;

}

public String getClientDetails(String id) {

return clientRepository.findClientById(id);

}

}

public class Main {

public static void main(String[] args) {

ClientRepository clientRepository = new ClientRepositoryImpl();

ClientService clientService = new ClientService(clientRepository);

String clientDetails = clientService.getClientDetails("1");

System.out.println("Client Details: " + clientDetails);

}}

