**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 Logger {

private static Logger instance;

private Logger() {

// Initialization

}

public static Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

public void log(String message) {

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

}}

class SingletonTest {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

Logger logger2 = Logger.getInstance();

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

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

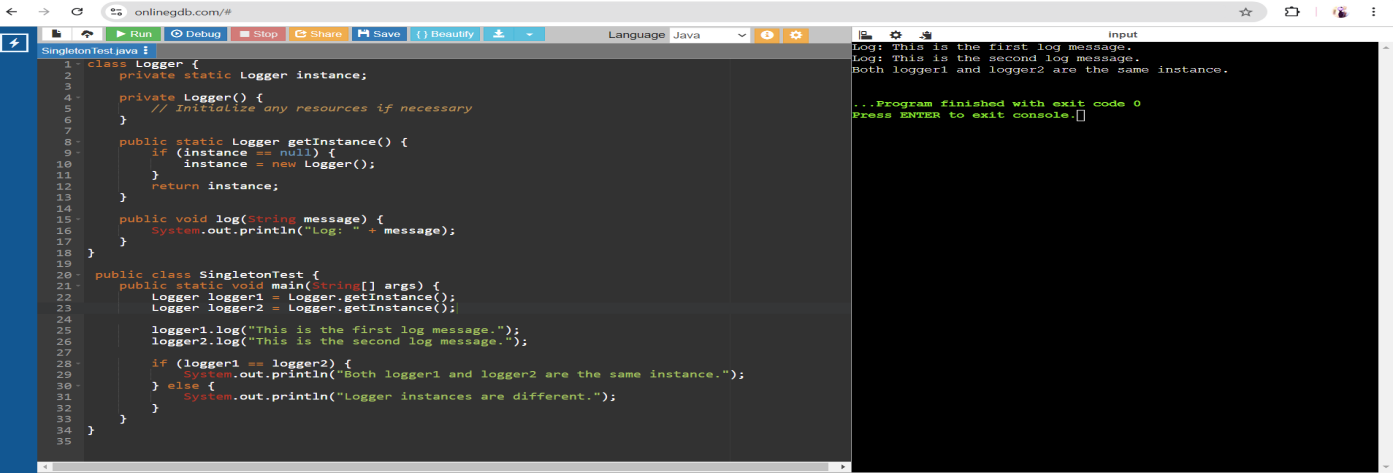
if (logger1 == logger2) {

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 DocumentManager {

public static void main(String[] args) {

DocumentFactory textFactory = new TextDocumentFactory();

Document textDoc = textFactory.createDocument();

textDoc.open();

textDoc.close();

DocumentFactory pdfFactory = new PortableDocumentFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open();

pdfDoc.close();

DocumentFactory excelFactory = new SpreadsheetDocumentFactory();

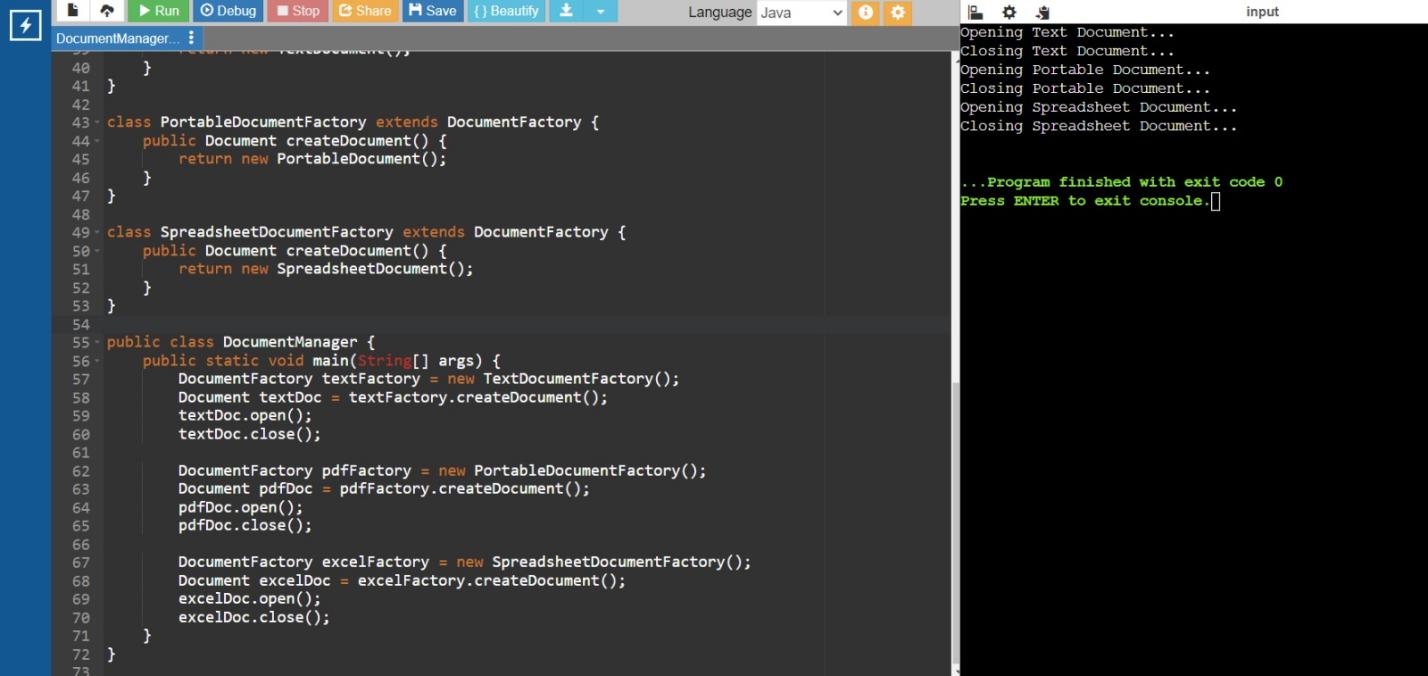
Document excelDoc = excelFactory.createDocument();

excelDoc.open();

excelDoc.close();

}

}



**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 CustomBuilderExample {

static class Device {

private String processor;

private String memory;

private String diskSpace;

private Device(Creator creator) {

this.processor = creator.processor;

this.memory = creator.memory;

this.diskSpace = creator.diskSpace;

}

public static class Creator {

private String processor;

private String memory;

private String diskSpace;

public Creator setProcessor(String processor) {

this.processor = processor;

return this;

}

public Creator setMemory(String memory) {

this.memory = memory;

return this;

}

public Creator setDiskSpace(String diskSpace) {

this.diskSpace = diskSpace;

return this;

}

public Device build() {

return new Device(this);

}

}

}

public static void main(String[] args) {

Device highEndDevice = new Device.Creator()

.setProcessor("Intel Core i9")

.setMemory("32GB")

.setDiskSpace("1TB SSD")

.build();

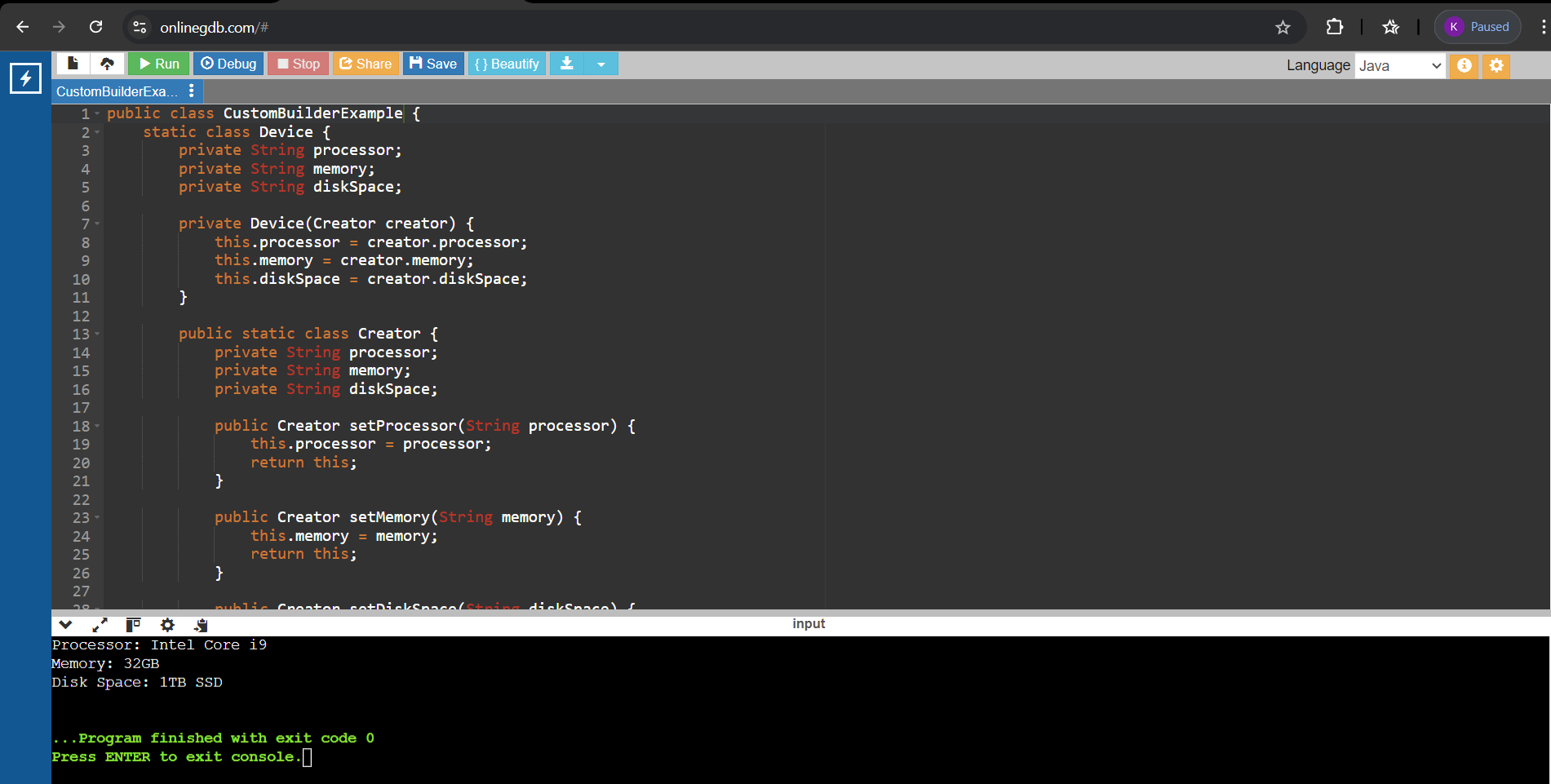
System.out.println("Processor: " + highEndDevice.processor);

System.out.println("Memory: " + highEndDevice.memory);

System.out.println("Disk Space: " + highEndDevice.diskSpace);

}

}



**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 PaymentHandler {

void handlePayment(double amount);

}

class Paytm {

public void executePayment(double amount) {

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

}

}

class Razorpay {

public void process(double amount) {

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

}

}

class GooglePay {

public void executeTransaction(double amount) {

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

}

}

class PaytmAdapter implements PaymentHandler {

private Paytm paytm;

public PaytmAdapter(Paytm paytm) {

this.paytm = paytm;

}

public void handlePayment(double amount) {

paytm.executePayment(amount);

}

}

class RazorpayAdapter implements PaymentHandler {

private Razorpay razorpay;

public RazorpayAdapter(Razorpay razorpay) {

this.razorpay = razorpay;

}

public void handlePayment(double amount) {

razorpay.process(amount);

}

}

class GooglePayAdapter implements PaymentHandler {

private GooglePay googlePay;

public GooglePayAdapter(GooglePay googlePay) {

this.googlePay = googlePay;

}

public void handlePayment(double amount) {

googlePay.executeTransaction(amount);

}

}

public class AdapterPatternDemo {

public static void main(String[] args) {

Paytm paytm = new Paytm();

Razorpay razorpay = new Razorpay();

GooglePay googlePay = new GooglePay();

PaymentHandler paytmAdapter = new PaytmAdapter(paytm);

PaymentHandler razorpayAdapter = new RazorpayAdapter(razorpay);

PaymentHandler googlePayAdapter = new GooglePayAdapter(googlePay);

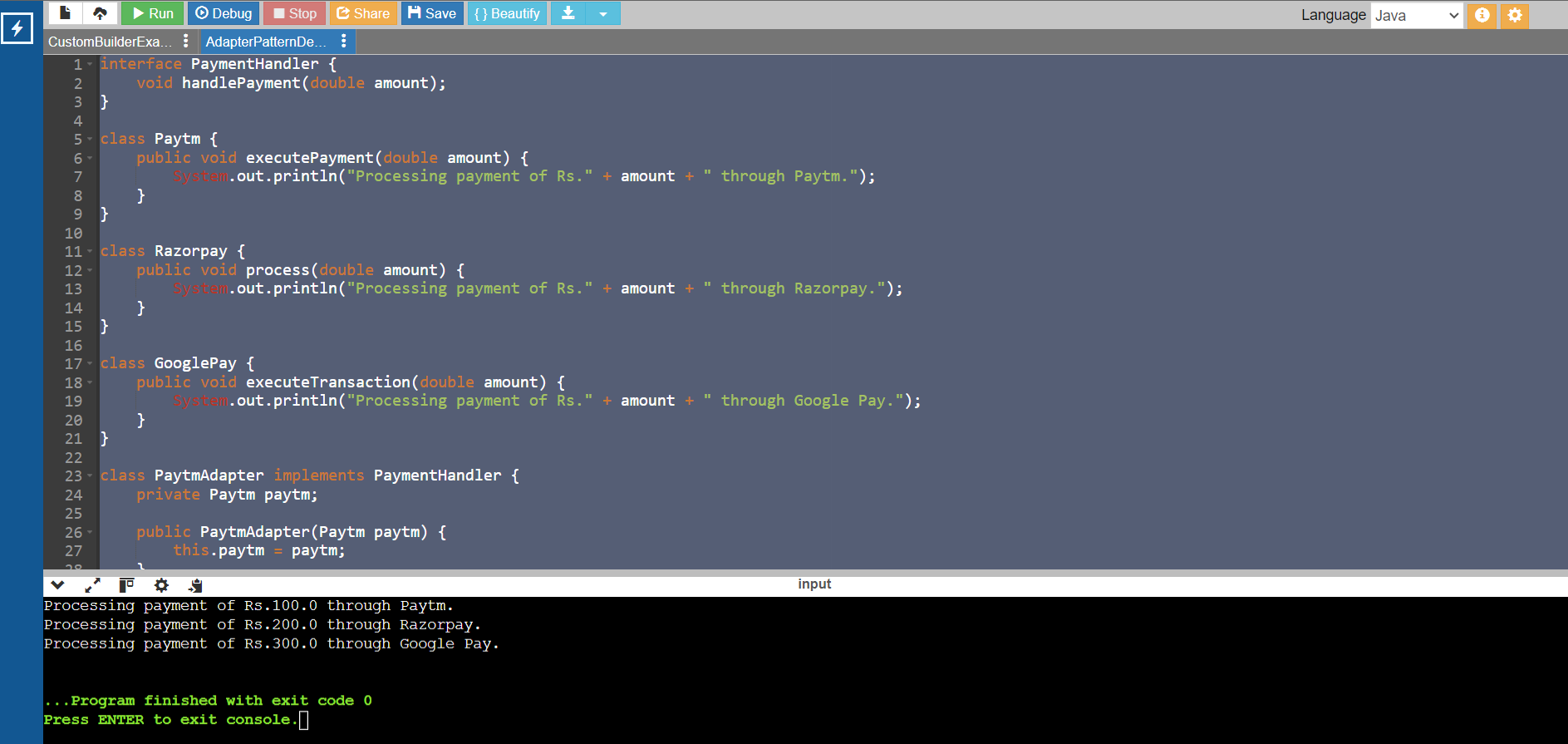
paytmAdapter.handlePayment(100.00);

razorpayAdapter.handlePayment(200.00);

googlePayAdapter.handlePayment(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 Messenger {

void notify(String content);

}

class EmailMessenger implements Messenger {

public void notify(String content) {

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

}

}

abstract class MessengerDecorator implements Messenger {

protected Messenger messenger;

public MessengerDecorator(Messenger messenger) {

this.messenger = messenger;

}

public void notify(String content) {

messenger.notify(content);

}

}

class SMSMessengerDecorator extends MessengerDecorator {

public SMSMessengerDecorator(Messenger messenger) {

super(messenger);

}

public void notify(String content) {

messenger.notify(content);

notifyViaSMS(content);

}

private void notifyViaSMS(String content) {

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

}

}

class SlackMessengerDecorator extends MessengerDecorator {

public SlackMessengerDecorator(Messenger messenger) {

super(messenger);

}

public void notify(String content) {

messenger.notify(content);

notifyViaSlack(content);

}

private void notifyViaSlack(String content) {

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

}

}

public class DecoratorPatternExample {

public static void main(String[] args) {

Messenger emailMessenger = new EmailMessenger();

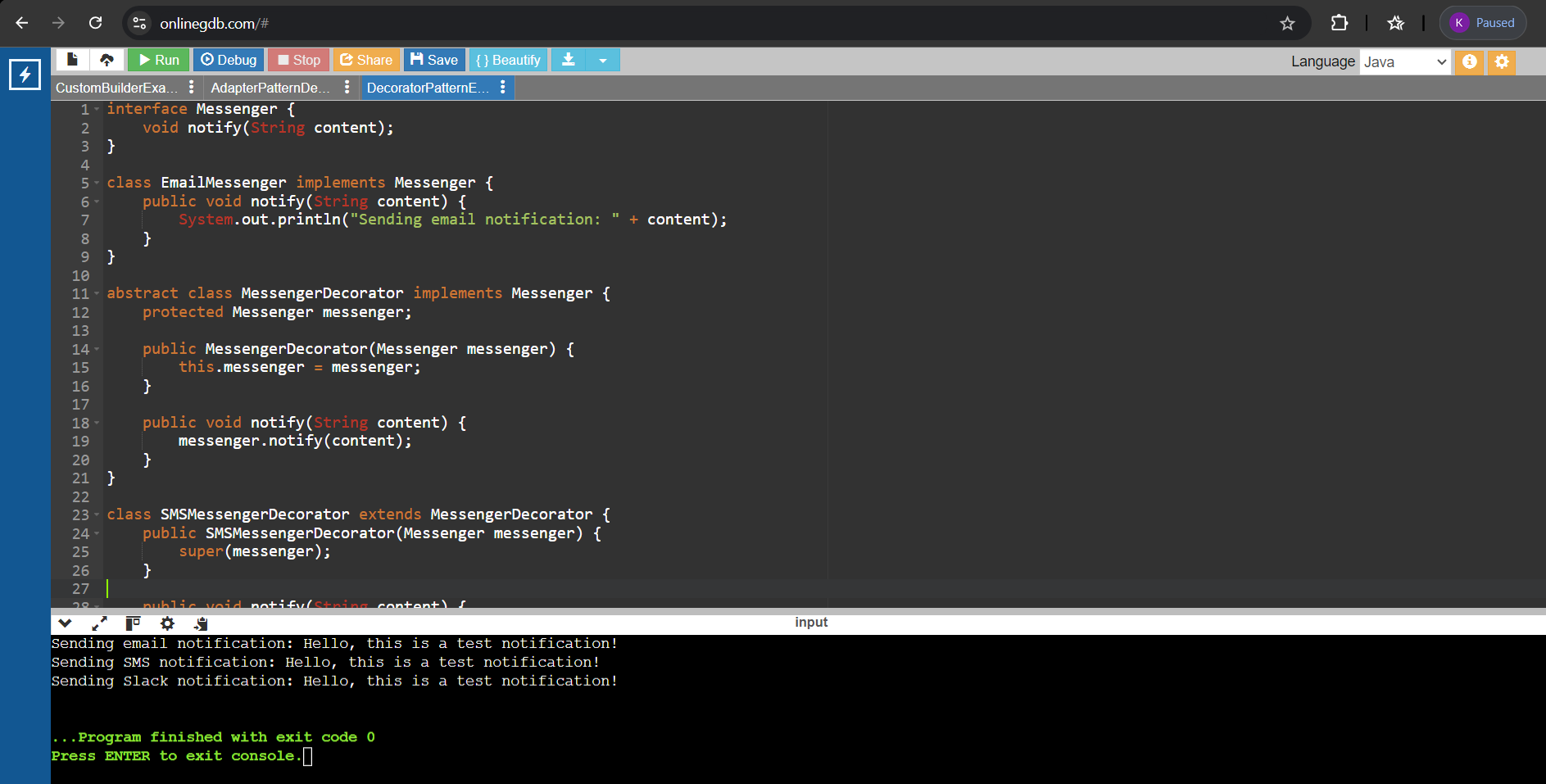
Messenger smsMessenger = new SMSMessengerDecorator(emailMessenger);

Messenger slackMessenger = new SlackMessengerDecorator(smsMessenger);

slackMessenger.notify("Hello, this is a test notification!");

}

}



**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 ActualPicture implements Picture {

private String fileName;

public ActualPicture(String fileName) {

this.fileName = fileName;

loadPictureFromDisk();

}

private void loadPictureFromDisk() {

System.out.println("Loading picture from disk: " + fileName);

}

public void show() {

System.out.println("Displaying picture: " + fileName);

}

}

class PictureProxy implements Picture {

private String fileName;

private ActualPicture actualPicture;

public PictureProxy(String fileName) {

this.fileName = fileName;

}

public void show() {

if (actualPicture == null) {

actualPicture = new ActualPicture(fileName);

}

actualPicture.show();

}

}

public class ProxyPatternExample {

public static void main(String[] args) {

Picture picture1 = new PictureProxy("picture1.jpg");

Picture picture2 = new PictureProxy("picture2.jpg");

picture1.show();

System.out.println("");

picture1.show();

System.out.println("");

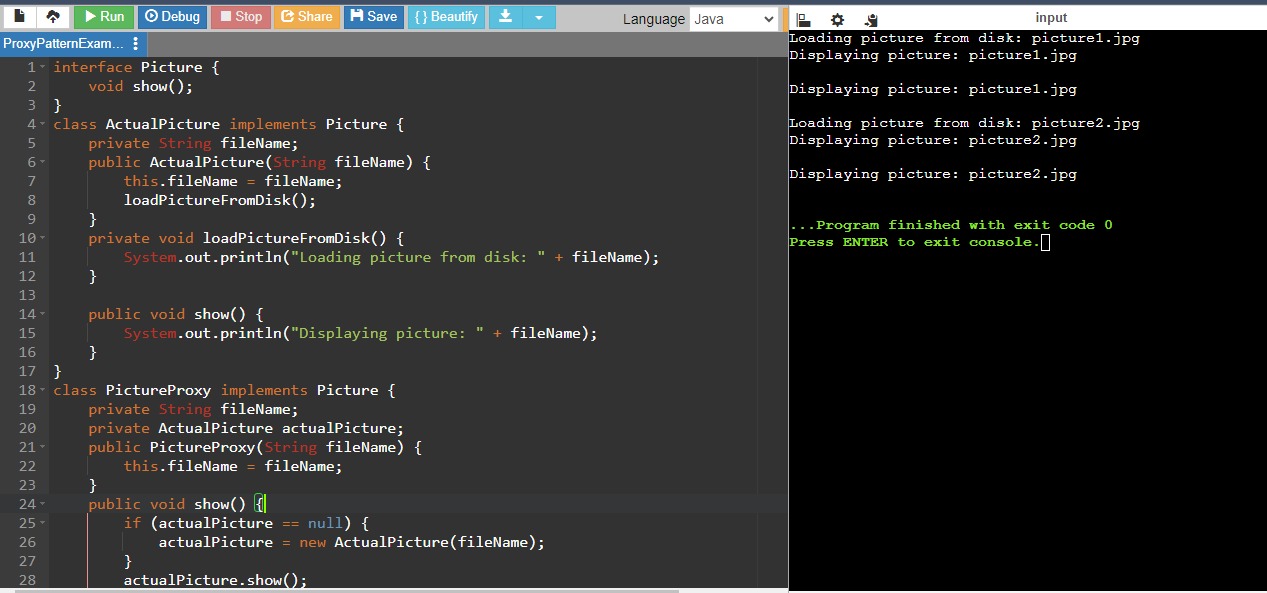
picture2.show();

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.

**Code:**

import java.util.ArrayList;

import java.util.List;

interface Market {

void addObserver(Listener l);

void removeObserver(Listener l);

void informObservers();

}

class MarketPlace implements Market {

private List<Listener> listeners;

private double marketPrice;

public MarketPlace() {

this.listeners = new ArrayList<>();

}

public void addObserver(Listener l) {

listeners.add(l);

}

public void removeObserver(Listener l) {

listeners.remove(l);

}

public void informObservers() {

for (Listener l : listeners) {

l.refresh(marketPrice);

}

}

public void setMarketPrice(double marketPrice) {

this.marketPrice = marketPrice;

informObservers();

}

}

interface Listener {

void refresh(double marketPrice);

}

class PhoneApp implements Listener {

private String appName;

public PhoneApp(String appName) {

this.appName = appName;

}

public void refresh(double marketPrice) {

System.out.println(appName + " received market price update: " + marketPrice);

}

}

class DesktopApp implements Listener {

private String appName;

public DesktopApp(String appName) {

this.appName = appName;

}

public void refresh(double marketPrice) {

System.out.println(appName + " received market price update: " + marketPrice);

}

}

public class ObserverPatternExample {

public static void main(String[] args) {

MarketPlace marketPlace = new MarketPlace();

Listener phoneApp = new PhoneApp("PhoneApp");

Listener desktopApp = new DesktopApp("DesktopApp");

marketPlace.addObserver(phoneApp);

marketPlace.addObserver(desktopApp);

marketPlace.setMarketPrice(100.00);

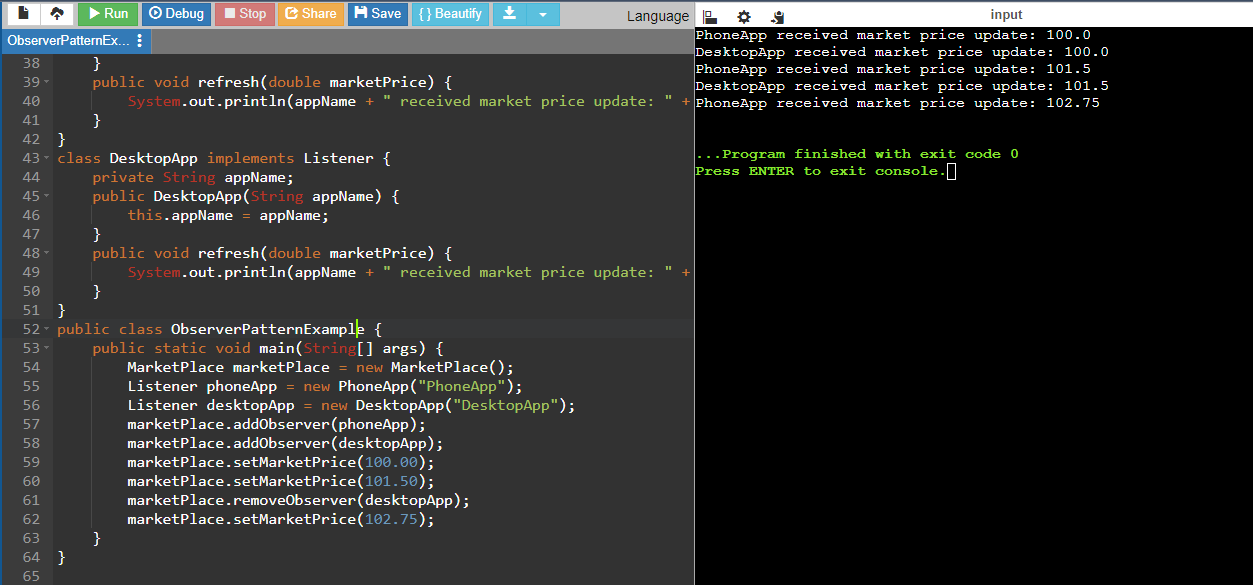
marketPlace.setMarketPrice(101.50);

marketPlace.removeObserver(desktopApp);

marketPlace.setMarketPrice(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.

**Code:**

interface PaymentMethod {

void makePayment(double amount);

}

class CardPayment implements PaymentMethod {

private String cardHolderName;

private String cardNumber;

private String securityCode;

private String expirationDate;

public CardPayment(String cardHolderName, String cardNumber, String securityCode, String expirationDate) {

this.cardHolderName = cardHolderName;

this.cardNumber = cardNumber;

this.securityCode = securityCode;

this.expirationDate = expirationDate;

}

public void makePayment(double amount) {

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

}

}

class EmailPayment implements PaymentMethod {

private String emailAddress;

private String userPassword;

public EmailPayment(String emailAddress, String userPassword) {

this.emailAddress = emailAddress;

this.userPassword = userPassword;

}

public void makePayment(double amount) {

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

}

}

class PaymentProcessor {

private PaymentMethod paymentMethod;

public void setPaymentMethod(PaymentMethod paymentMethod) {

this.paymentMethod = paymentMethod;

}

public void processPayment(double amount) {

paymentMethod.makePayment(amount);

}

}

public class StrategyPatternExample {

public static void main(String[] args) {

PaymentProcessor processor = new PaymentProcessor();

processor.setPaymentMethod(new CardPayment("Jane Doe", "9876543210987654", "321", "11/24"));

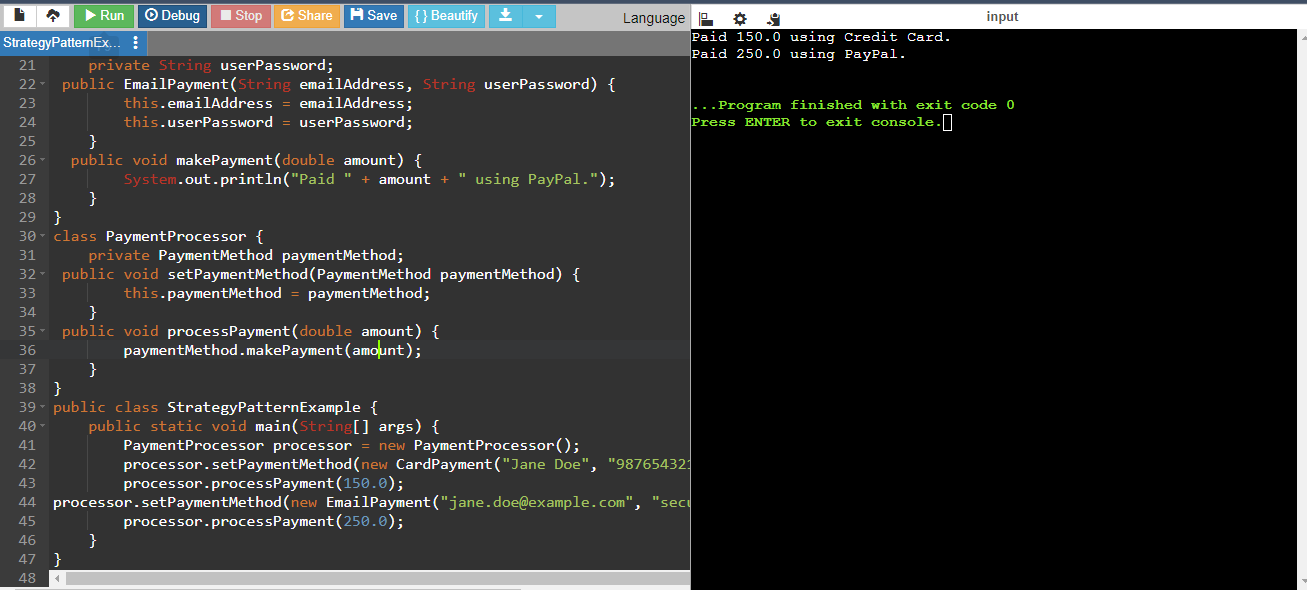
processor.processPayment(150.0);

processor.setPaymentMethod(new EmailPayment("jane.doe@example.com", "securepassword"));

processor.processPayment(250.0);

}

}



**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.

**Code:**

interface Action {

void perform();

}

class TurnOnLight implements Action {

private Lamp lamp;

public TurnOnLight(Lamp lamp) {

this.lamp = lamp;

}

@Override

public void perform() {

lamp.switchOn();

}

}

class TurnOffLight implements Action {

private Lamp lamp;

public TurnOffLight(Lamp lamp) {

this.lamp = lamp;

}

@Override

public void perform() {

lamp.switchOff();

}

}

class Lamp {

public void switchOn() {

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

}

public void switchOff() {

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

}

}

class Controller {

private Action action;

public void setAction(Action action) {

this.action = action;

}

public void pressButton() {

action.perform();

}

}

public class CommandPatternExample {

public static void main(String[] args) {

Lamp bedroomLamp = new Lamp();

Action turnOn = new TurnOnLight(bedroomLamp);

Action turnOff = new TurnOffLight(bedroomLamp);

Controller remote = new Controller();

remote.setAction(turnOn);

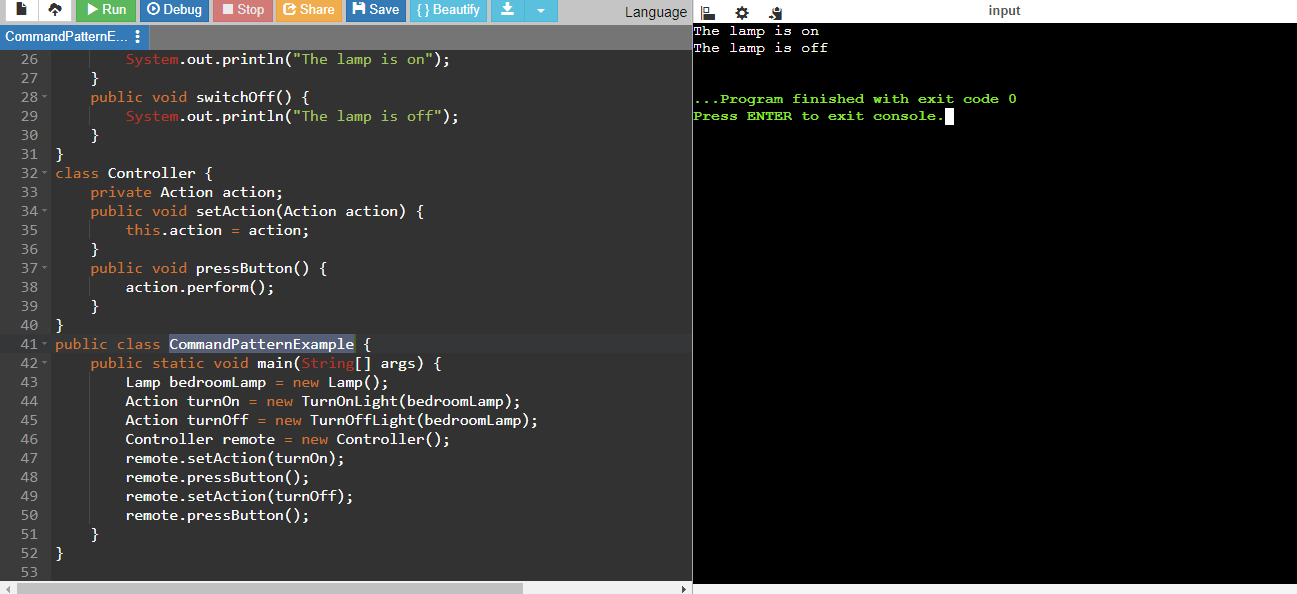
remote.pressButton();

remote.setAction(turnOff);

remote.pressButton();

}

}



**Exercise 10: Implementing the MVC Pattern**

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

**Code:**

class Pupil {

private String pupilId;

private String pupilName;

private String pupilGrade;

public Pupil(String pupilId, String pupilName, String pupilGrade) {

this.pupilId = pupilId;

this.pupilName = pupilName;

this.pupilGrade = pupilGrade;

}

public String getPupilId() {

return pupilId;

}

public void setPupilId(String pupilId) {

this.pupilId = pupilId;

}

public String getPupilName() {

return pupilName;

}

public void setPupilName(String pupilName) {

this.pupilName = pupilName;

}

public String getPupilGrade() {

return pupilGrade;

}

public void setPupilGrade(String pupilGrade) {

this.pupilGrade = pupilGrade;

}

}

class PupilView {

public void displayPupilDetails(String pupilName, String pupilId, String pupilGrade) {

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

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

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

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

}

}

class PupilController {

private Pupil model;

private PupilView view;

public PupilController(Pupil model, PupilView view) {

this.model = model;

this.view = view;

}

public void setPupilName(String name) {

model.setPupilName(name);

}

public String getPupilName() {

return model.getPupilName();

}

public void setPupilId(String id) {

model.setPupilId(id);

}

public String getPupilId() {

return model.getPupilId();

}

public void setPupilGrade(String grade) {

model.setPupilGrade(grade);

}

public String getPupilGrade() {

return model.getPupilGrade();

}

public void updateView() {

view.displayPupilDetails(model.getPupilName(), model.getPupilId(), model.getPupilGrade());

}

}

public class MVCPatternExample {

public static void main(String[] args) {

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

PupilView view = new PupilView();

PupilController controller = new PupilController(model, view);

controller.updateView();

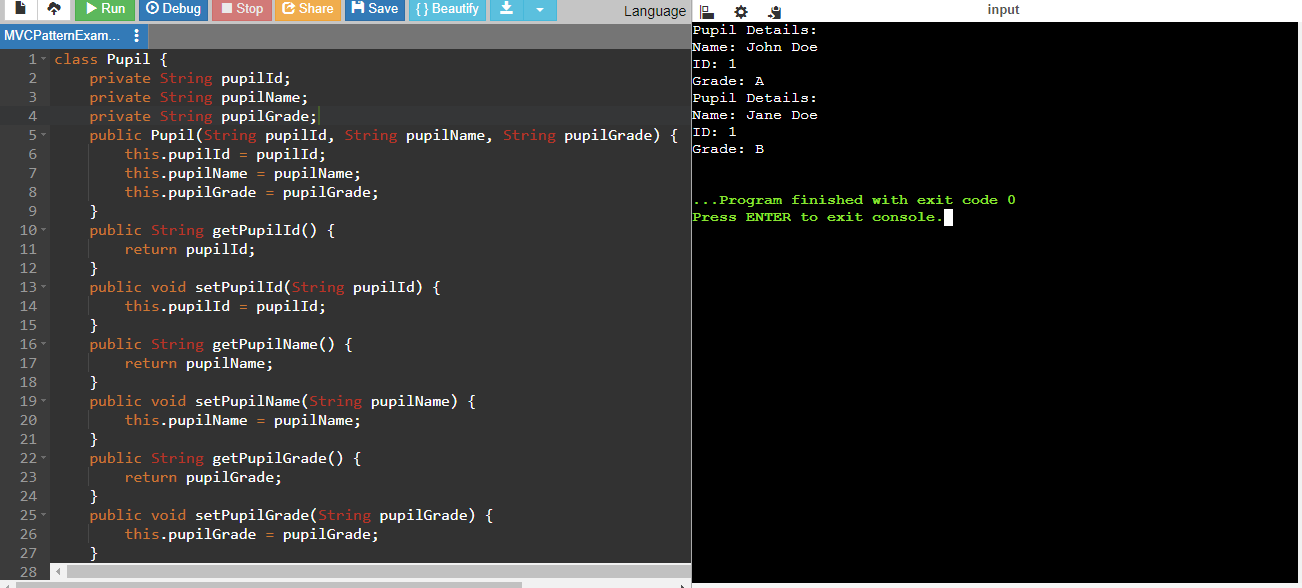
controller.setPupilName("Jane Doe");

controller.setPupilGrade("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.

**Code:**

interface ClientRepository {

String findClientById(String id);

}

class ClientRepositoryImpl implements ClientRepository {

public String findClientById(String id) {

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

return "Jane Smith";

} 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 DependencyInjectionExample {

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);

}

}

