**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 SingletonLogger {

private static SingletonLogger uniqueInstance;

private SingletonLogger() {

// Initialization

}

public static SingletonLogger getUniqueInstance() {

if (uniqueInstance == null) {

uniqueInstance = new SingletonLogger();

}

return uniqueInstance;

}

public void logEntry(String message) {

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

}

}

class EntryPoint {

public static void main(String[] args) {

SingletonLogger loggerInstance1 = SingletonLogger.getUniqueInstance();

SingletonLogger loggerInstance2 = SingletonLogger.getUniqueInstance();

loggerInstance1.logEntry("This is the first log message.");

loggerInstance2.logEntry("This is the second log message.");

if (loggerInstance1 == loggerInstance2) {

System.out.println("Both loggerInstance1 and loggerInstance2 are the same instance.");

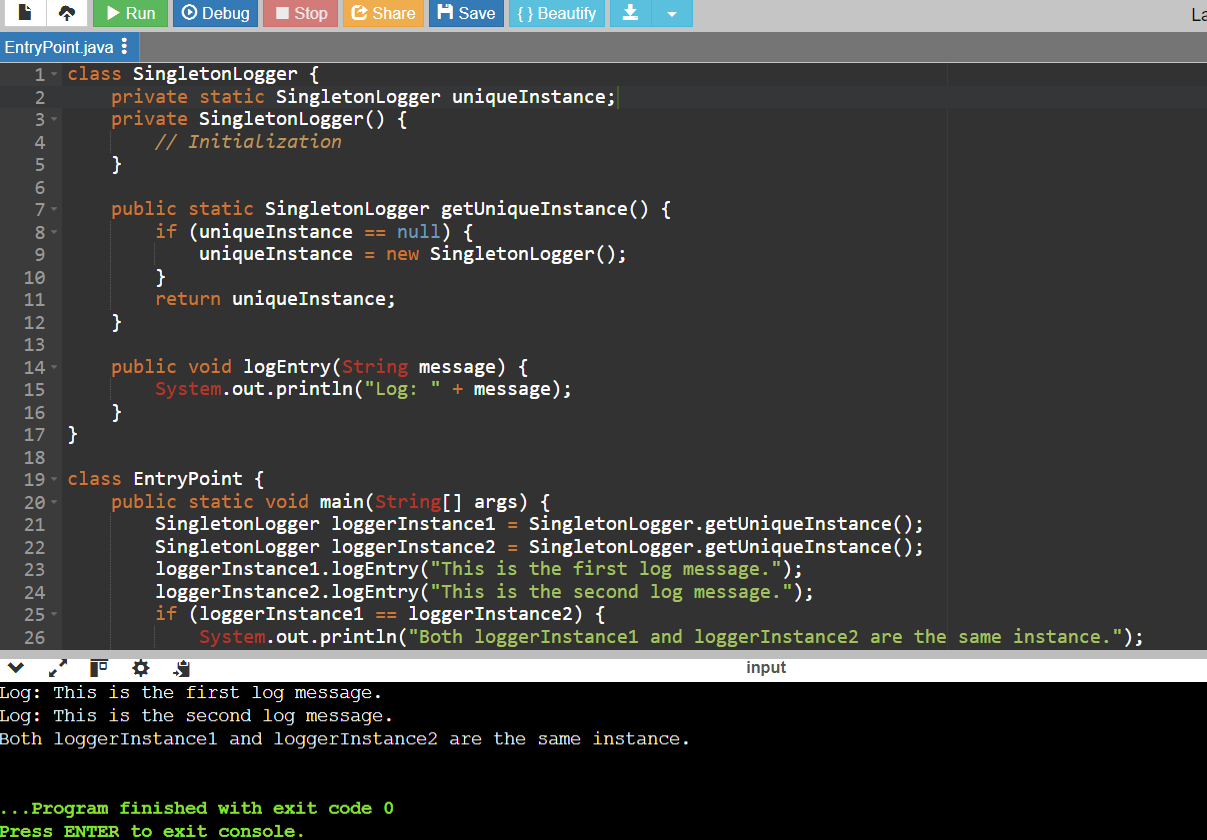
} else {

System.out.println("SingletonLogger 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 Doc {

void open();

void close();

}

class WordDoc implements Doc {

public void open() {

System.out.println("Opening Word Doc...");

}

public void close() {

System.out.println("Closing Word Doc...");

}

}

class PdfDoc implements Doc {

public void open() {

System.out.println("Opening PDF Doc...");

}

public void close() {

System.out.println("Closing PDF Doc...");

}

}

class ExcelDoc implements Doc {

public void open() {

System.out.println("Opening Excel Doc...");

}

public void close() {

System.out.println("Closing Excel Doc...");

}

}

abstract class DocFactory {

public abstract Doc createDoc();

}

class WordDocFactory extends DocFactory {

public Doc createDoc() {

return new WordDoc();

}

}

class PdfDocFactory extends DocFactory {

public Doc createDoc() {

return new PdfDoc();

}

}

class ExcelDocFactory extends DocFactory {

public Doc createDoc() {

return new ExcelDoc();

}

}

public class Example {

public static void main(String[] args) {

DocFactory wordFactory = new WordDocFactory();

Doc wordDoc = wordFactory.createDoc();

wordDoc.open();

wordDoc.close();

DocFactory pdfFactory = new PdfDocFactory();

Doc pdfDoc = pdfFactory.createDoc();

pdfDoc.open();

pdfDoc.close();

DocFactory excelFactory = new ExcelDocFactory();

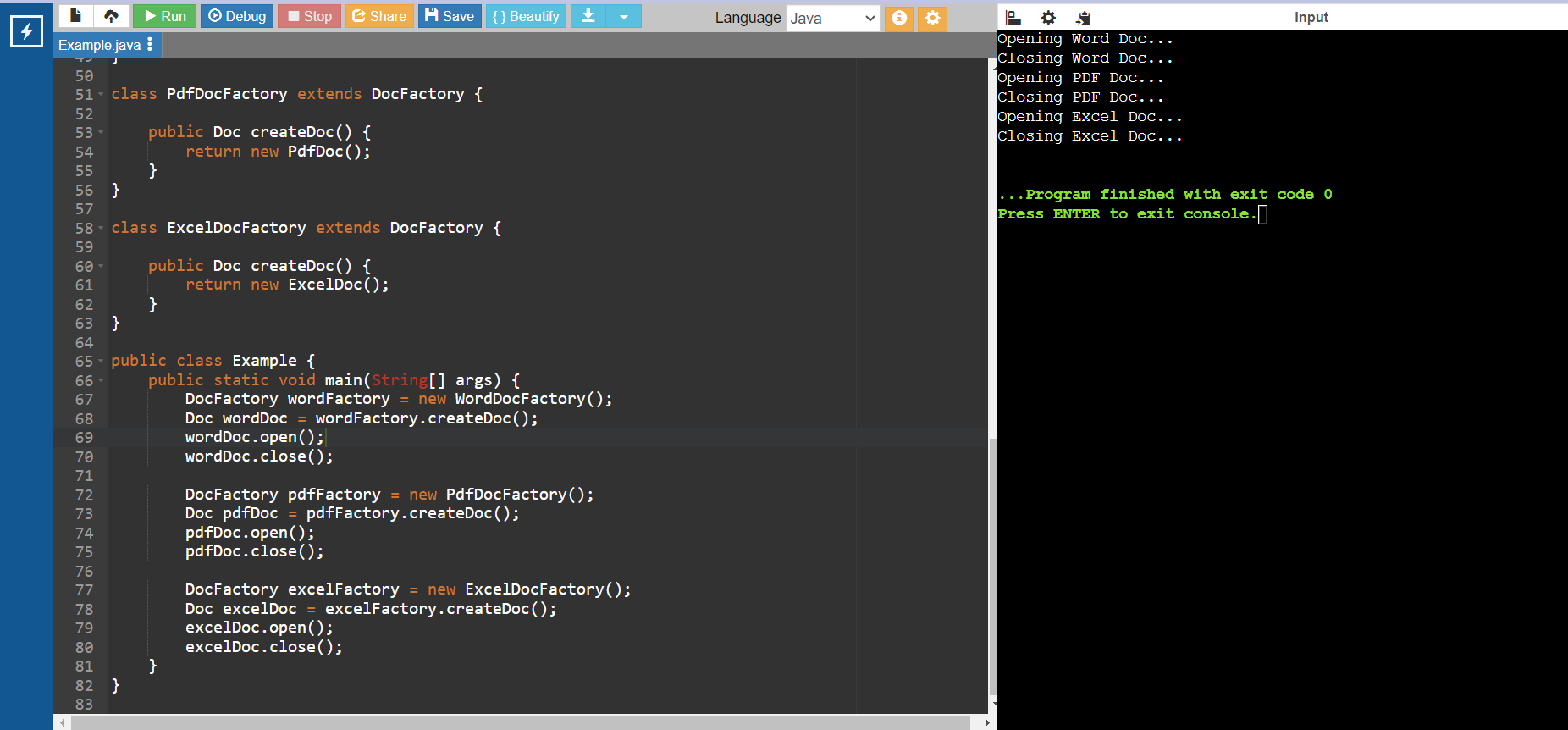
Doc excelDoc = excelFactory.createDoc();

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 Pattern {

static class Comp {

private String CPU;

private String RAM;

private String storage;

private Comp(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 Comp build() {

return new Comp(this);

}

}

}

public static void main(String[] args) {

Comp gamingPC = new Comp.Builder()

.setCPU("Intel Core i9")

.setRAM("32GB")

.setStorage("1TB SSD")

.build();

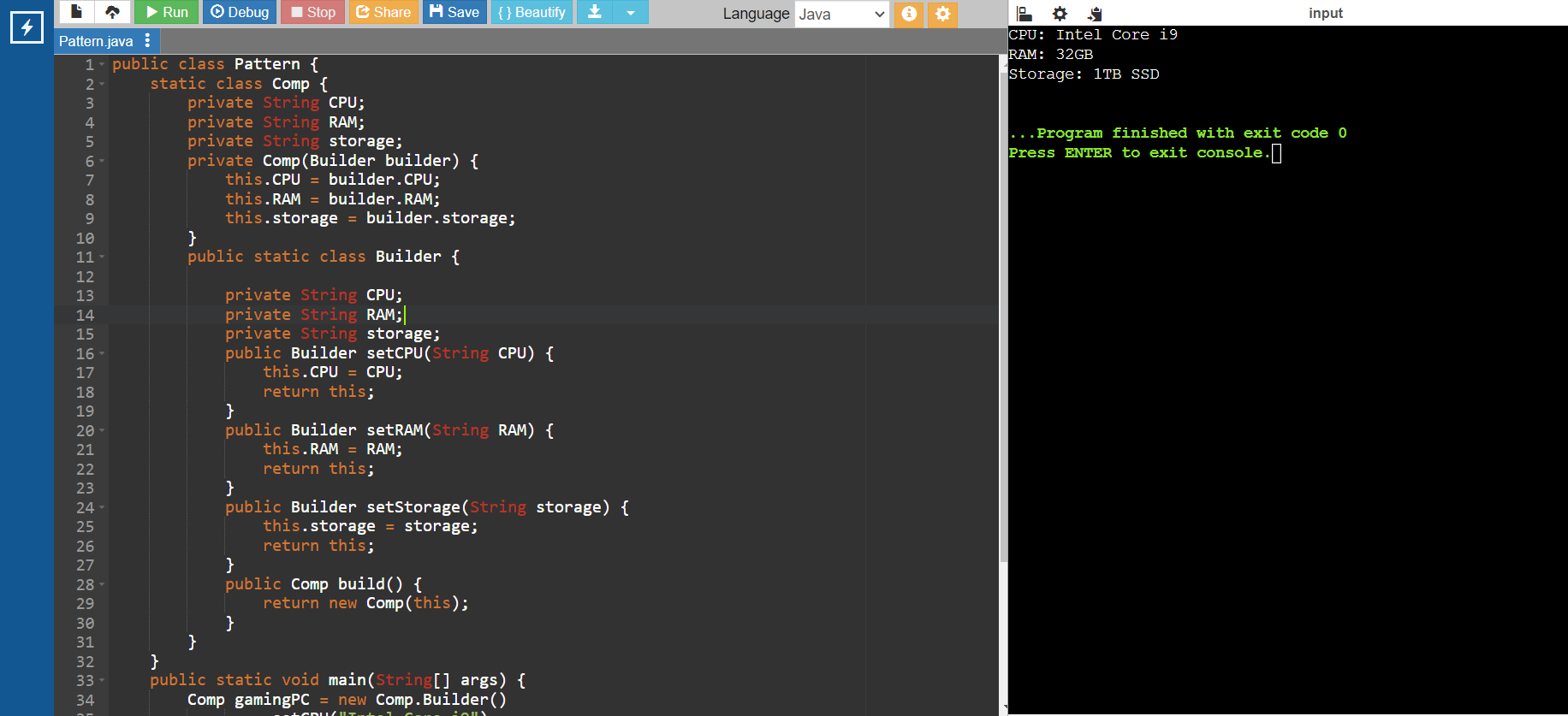
System.out.println("CPU: " + gamingPC.CPU);

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

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

}

}



**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 OnlinePaymentService {

public void executePayment(double amount) {

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

}

}

class CreditCardService {

public void authorizePayment(double amount) {

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

}

}

class WalletService {

public void completeTransaction(double amount) {

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

}

}

class OnlinePaymentAdapter implements PaymentHandler {

private OnlinePaymentService onlinePaymentService;

public OnlinePaymentAdapter(OnlinePaymentService onlinePaymentService) {

this.onlinePaymentService = onlinePaymentService;

}

public void handlePayment(double amount) {

onlinePaymentService.executePayment(amount);

}

}

class CreditCardAdapter implements PaymentHandler {

private CreditCardService creditCardService;

public CreditCardAdapter(CreditCardService creditCardService) {

this.creditCardService = creditCardService;

}

public void handlePayment(double amount) {

creditCardService.authorizePayment(amount);

}

}

class WalletAdapter implements PaymentHandler {

private WalletService walletService;

public WalletAdapter(WalletService walletService) {

this.walletService = walletService;

}

public void handlePayment(double amount) {

walletService.completeTransaction(amount);

}

}

public class PaymentSystem {

public static void main(String[] args) {

OnlinePaymentService onlinePaymentService = new OnlinePaymentService();

CreditCardService creditCardService = new CreditCardService();

WalletService walletService = new WalletService();

PaymentHandler onlinePaymentAdapter = new OnlinePaymentAdapter(onlinePaymentService);

PaymentHandler creditCardAdapter = new CreditCardAdapter(creditCardService);

PaymentHandler walletAdapter = new WalletAdapter(walletService);

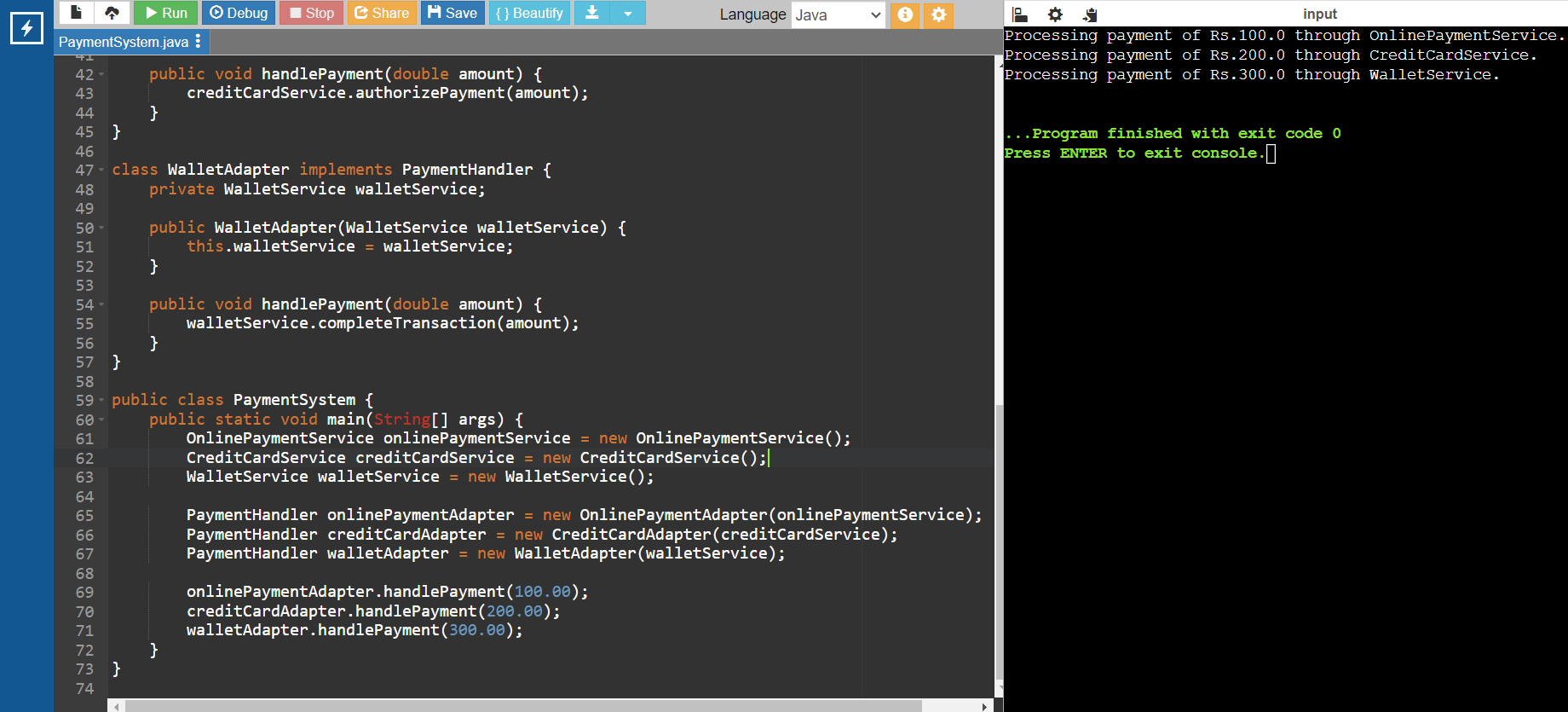
onlinePaymentAdapter.handlePayment(100.00);

creditCardAdapter.handlePayment(200.00);

walletAdapter.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 Notify {

void send(String message);

}

class EmailNotify implements Notify {

public void send(String message) {

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

}

}

abstract class NotifyDecor implements Notify {

protected Notify Notify;

public NotifyDecor(Notify Notify) {

this.Notify = Notify;

}

public void send(String message) {

Notify.send(message);

}

}

class SMSNotifyDecor extends NotifyDecor {

public SMSNotifyDecor(Notify Notify) {

super(Notify);

}

public void send(String message) {

Notify.send(message);

sendSMS(message);

}

private void sendSMS(String message) {

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

}}

class SlackNotifyDecor extends NotifyDecor {

public SlackNotifyDecor(Notify Notify) {

super(Notify);

}

public void send(String message) {

Notify.send(message);

sendSlack(message);

}

private void sendSlack(String message) {

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

}}

public class Decor {

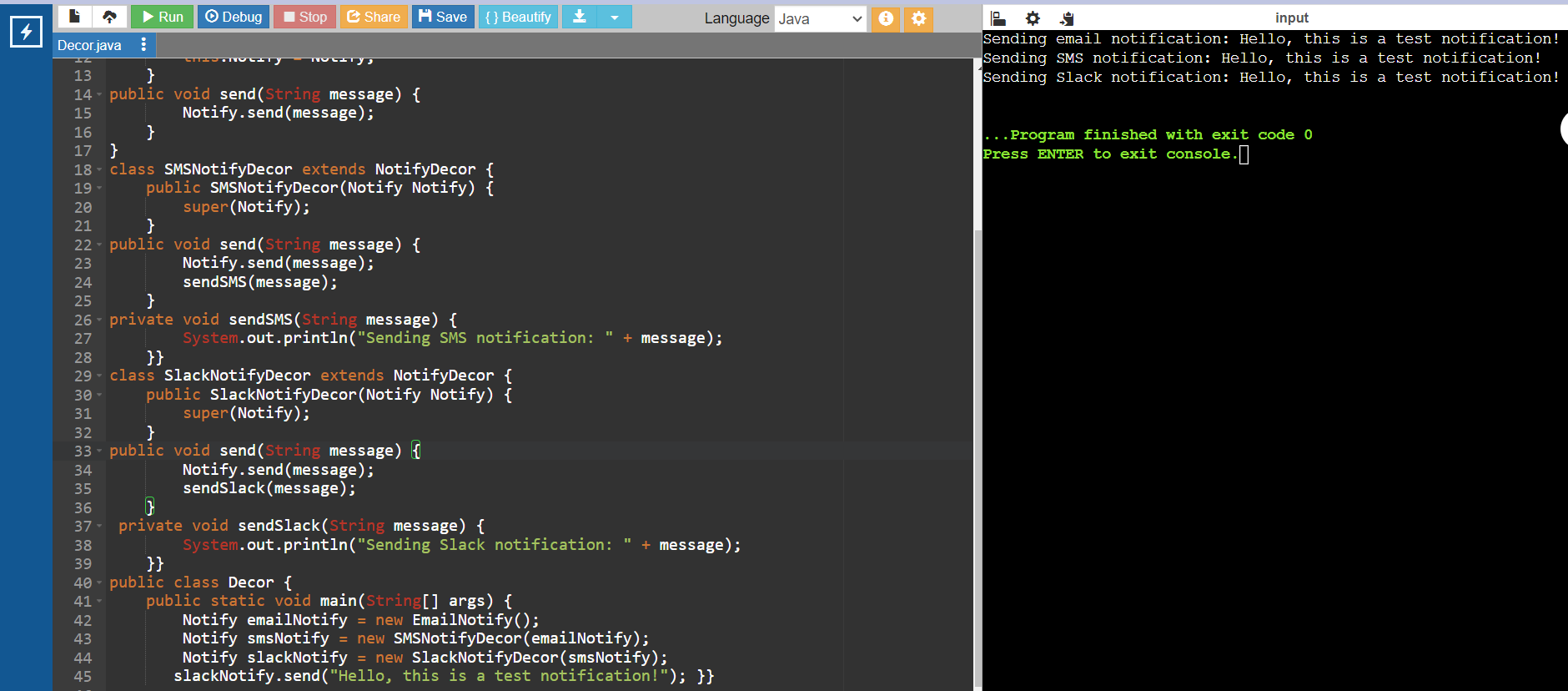
public static void main(String[] args) {

Notify emailNotify = new EmailNotify();

Notify smsNotify = new SMSNotifyDecor(emailNotify);

Notify slackNotify = new SlackNotifyDecor(smsNotify);

slackNotify.send("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 Img {

void display();

}

//Implement Real Subject Class

class R\_i implements Img {

private String img\_name;

public R\_i(String img\_name) {

this.img\_name = img\_name;

loadImgFromDisk();

}

private void loadImgFromDisk() {

System.out.println("Loading Image from disk: " + img\_name);

}

public void display() {

System.out.println("Displaying Image: " + img\_name);

}

}

// Implement Proxy Class

class ProxyImg implements Img {

private String img\_name;

private R\_i R\_i;

public ProxyImg(String img\_name) {

this.img\_name = img\_name;

}

public void display() {

if (R\_i == null) {

R\_i = new R\_i(img\_name);

}

R\_i.display();

}

}

// Test the Proxy Implementation

public class Proxy {

public static void main(String[] args) {

Img Img1 = new ProxyImg("Img1.jpg");

Img Img2 = new ProxyImg("Img2.jpg");

// Img will be loaded from disk

Img1.display();

System.out.println("");

// Img will not be loaded from disk as it is already loaded

Img1.display();

System.out.println("");

// Img will be loaded from disk

Img2.display();

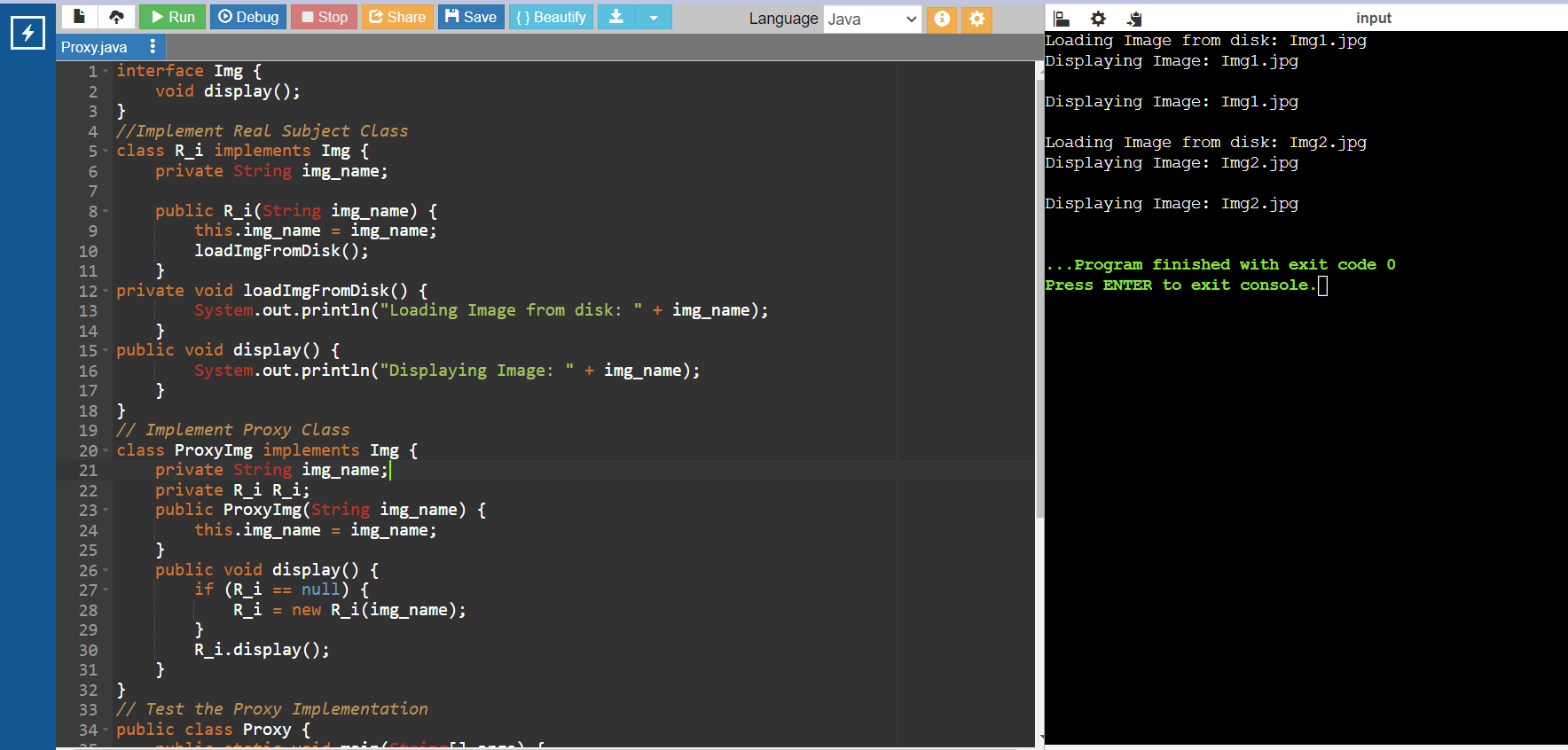
System.out.println("");

// Img will not be loaded from disk as it is already loaded

Img2.display();

}

}



**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 addObserver(Subscriber s);

void removeObserver(Subscriber s);

void alertObservers();

}

class TradingPlatform implements Market {

private List<Subscriber> subscribers;

private double price;

public TradingPlatform() {

this.subscribers = new ArrayList<>();

}

public void addObserver(Subscriber s) {

subscribers.add(s);

}

public void removeObserver(Subscriber s) {

subscribers.remove(s);

}

public void alertObservers() {

for (Subscriber s : subscribers) {

s.update(price);

}

}

public void setPrice(double price) {

this.price = price;

alertObservers();

}

}

interface Subscriber {

void update(double price);

}

class SmartphoneApp implements Subscriber {

private String appIdentifier;

public SmartphoneApp(String appIdentifier) {

this.appIdentifier = appIdentifier;

}

public void update(double price) {

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

}

}

class BrowserApp implements Subscriber {

private String appIdentifier;

BrowserApp(String appIdentifier) {

this.appIdentifier = appIdentifier;

}

public void update(double price) {

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

}

}

public class NotificationSystem {

public static void main(String[] args) {

TradingPlatform tradingPlatform = new TradingPlatform();

Subscriber smartphoneApp = new SmartphoneApp("SmartphoneApp");

Subscriber browserApp = new BrowserApp("BrowserApp");

tradingPlatform.addObserver(smartphoneApp);

tradingPlatform.addObserver(browserApp);

tradingPlatform.setPrice(100.00);

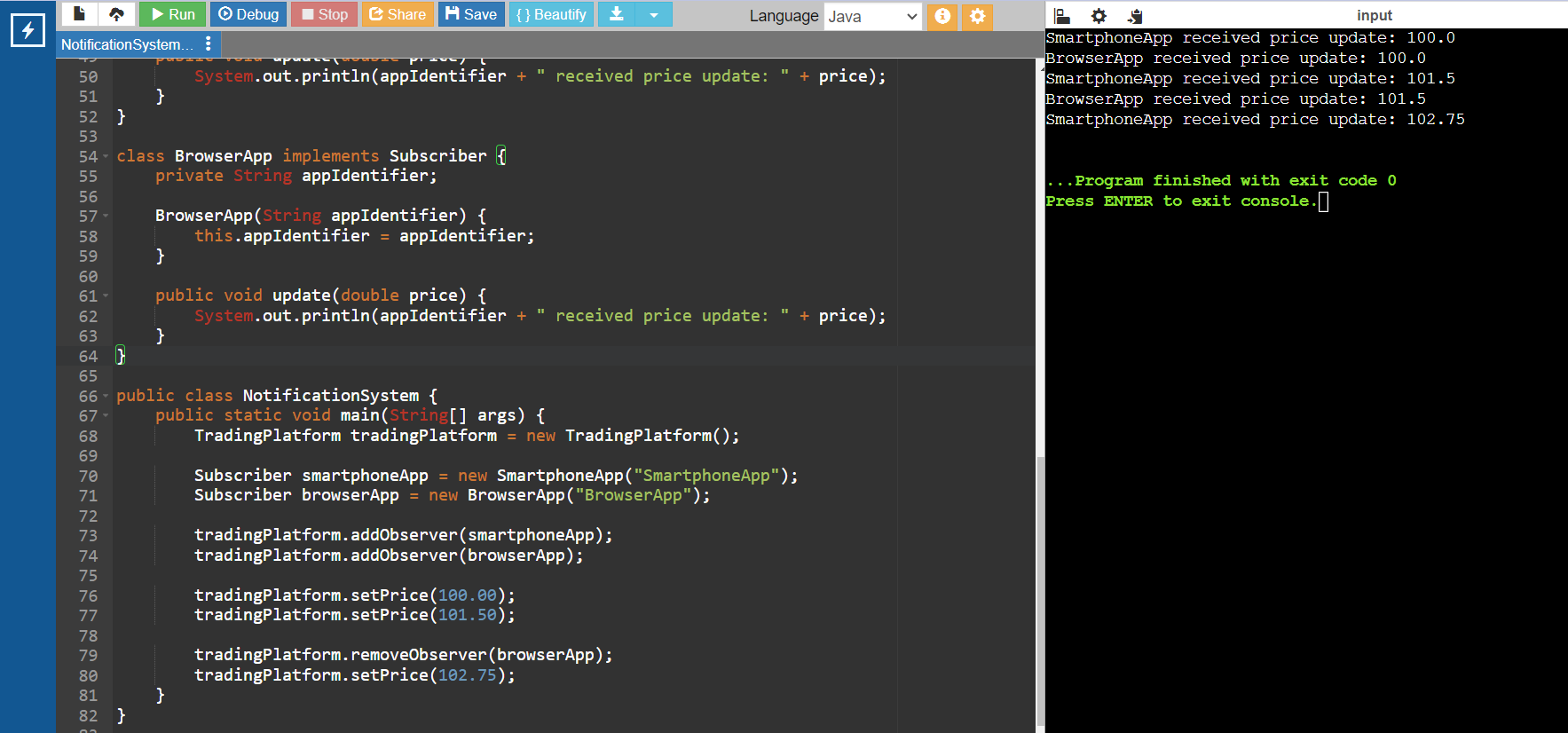
tradingPlatform.setPrice(101.50);

tradingPlatform.removeObserver(browserApp);

tradingPlatform.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 Strategy {

void pay(double amount);

}

// Step 3: Implement Concrete Strategies

class CreditCardPayment implements Strategy {

private String name;

private String card\_number;

private String cvv;

private String expiry\_date;

public CreditCardPayment(String name, String card\_number, String cvv, String expiry\_date) {

this.name = name;

this.card\_number = card\_number;

this.cvv = cvv;

this.expiry\_date = expiry\_date;

}

public void pay(double amount) {

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

}}

class PhonepePayment implements Strategy {

private String email;

private String password;

public PhonepePayment(String email, String password) {

this.email = email;

this.password = password;

}

public void pay(double amount) {

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

}

}

// Step 4: Implement Context Class

class PaymentContext {

private Strategy Strategy;

public void setStrategy(Strategy Strategy) {

this.Strategy = Strategy;

}

public void executePayment(double amount) {

Strategy.pay(amount);

}

}

// Step 5: Test the Strategy Implementation

public class StrategyPatternExample {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

// Pay using Credit Card

context.setStrategy(new CreditCardPayment("jag ann", "1234567890123456", "annc", "11/29"));

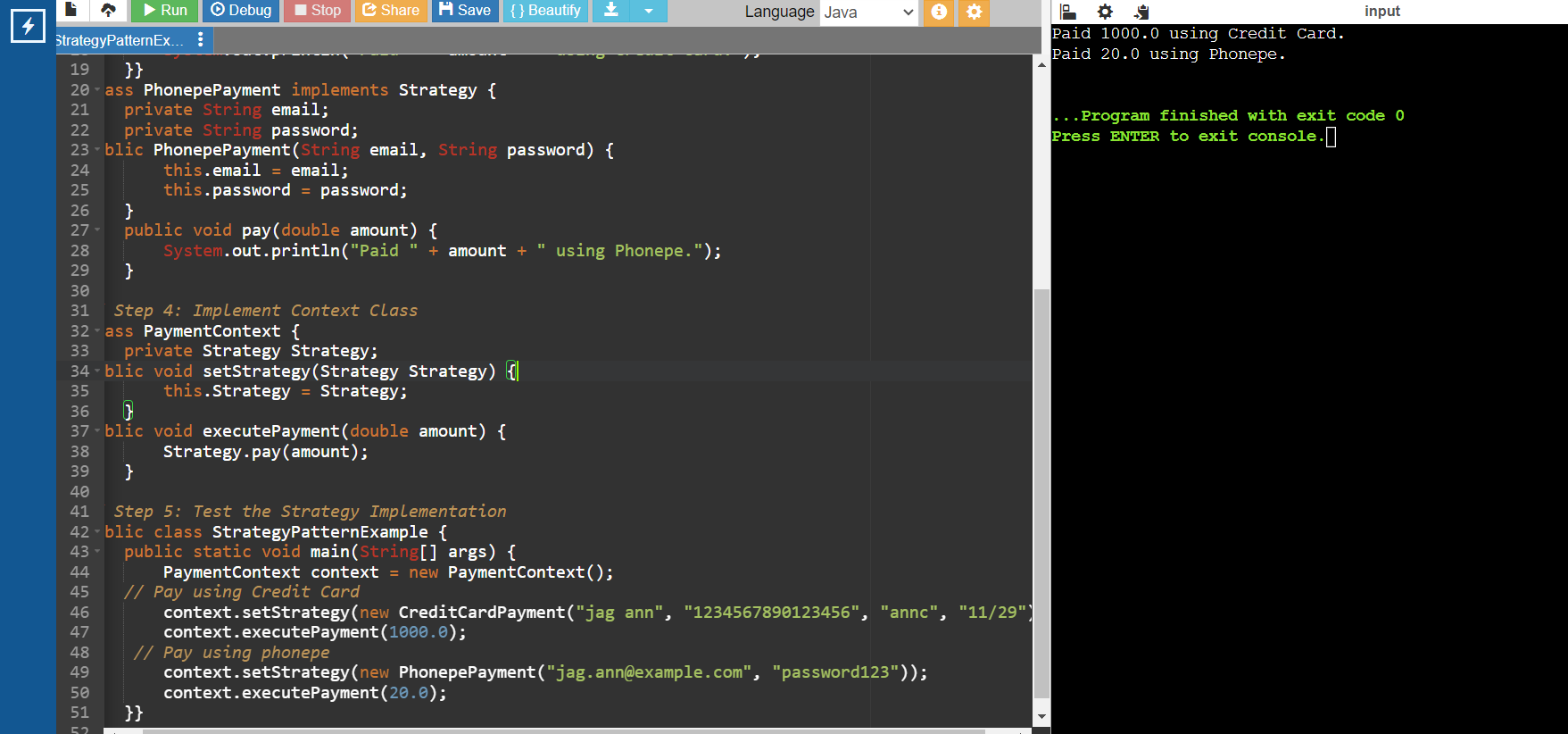
context.executePayment(1000.0);

// Pay using phonepe

context.setStrategy(new PhonepePayment("jag.ann@example.com", "password123"));

context.executePayment(20.0);

}}



**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 Command {

void exe();

}

// Implement Concrete Commands

class LightOn implements Command {

private Light light;

public LightOn(Light light) {

this.light = light;

}

@Override

public void exe() {

light.turnOn();

}

}

class LightOff implements Command {

private Light light;

public LightOff(Light light) {

this.light = light;

}

@Override

public void exe() {

light.turnOff();

}

}

// Implement Receiver Class

class Light {

public void turnOn() {

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

}

public void turnOff() {

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

}

}

// Implement Invoker Class

class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

command.exe();

}

}

// Test the Command Implementation

public class CommandPattern {

public static void main(String[] args) {

Light livingRoomLight = new Light();

Command lightOn = new LightOn(livingRoomLight);

Command lightOff = new LightOff(livingRoomLight);

RemoteControl remote = new RemoteControl();

// Turn the light on

remote.setCommand(lightOn);

remote.pressButton();

// Turn the light off

remote.setCommand(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.

// Define Model Class

class Std {

private String id;

private String name;

private String grade;

public Std(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;

}

}

// Define View Class

class StdView {

public void displayStdDetails(String StdName, String StdId, String StdGrade) {

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

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

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

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

}

}

// Define Controller Class

class StdController {

private Std model;

private StdView view;

public StdController(Std model, StdView view) {

this.model = model;

this.view = view;

}

public void setStdName(String name) {

model.setName(name);

}

public String getStdName() {

return model.getName();

}

public void setStdId(String id) {

model.setId(id);

}

public String getStdId() {

return model.getId();

}

public void setStdGrade(String grade) {

model.setGrade(grade);

}

public String getStdGrade() {

return model.getGrade();

}

public void updateView() {

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

}}

// Test the MVC Implementation

public class MVCPatternExample {

public static void main(String[] args) {

// Create a Std model

Std model = new Std("8", "A ram", "A+");

// Create a Std view

StdView view = new StdView();

// Create a Std controller

StdController controller = new StdController(model, view);

// Display initial Std details

controller.updateView();

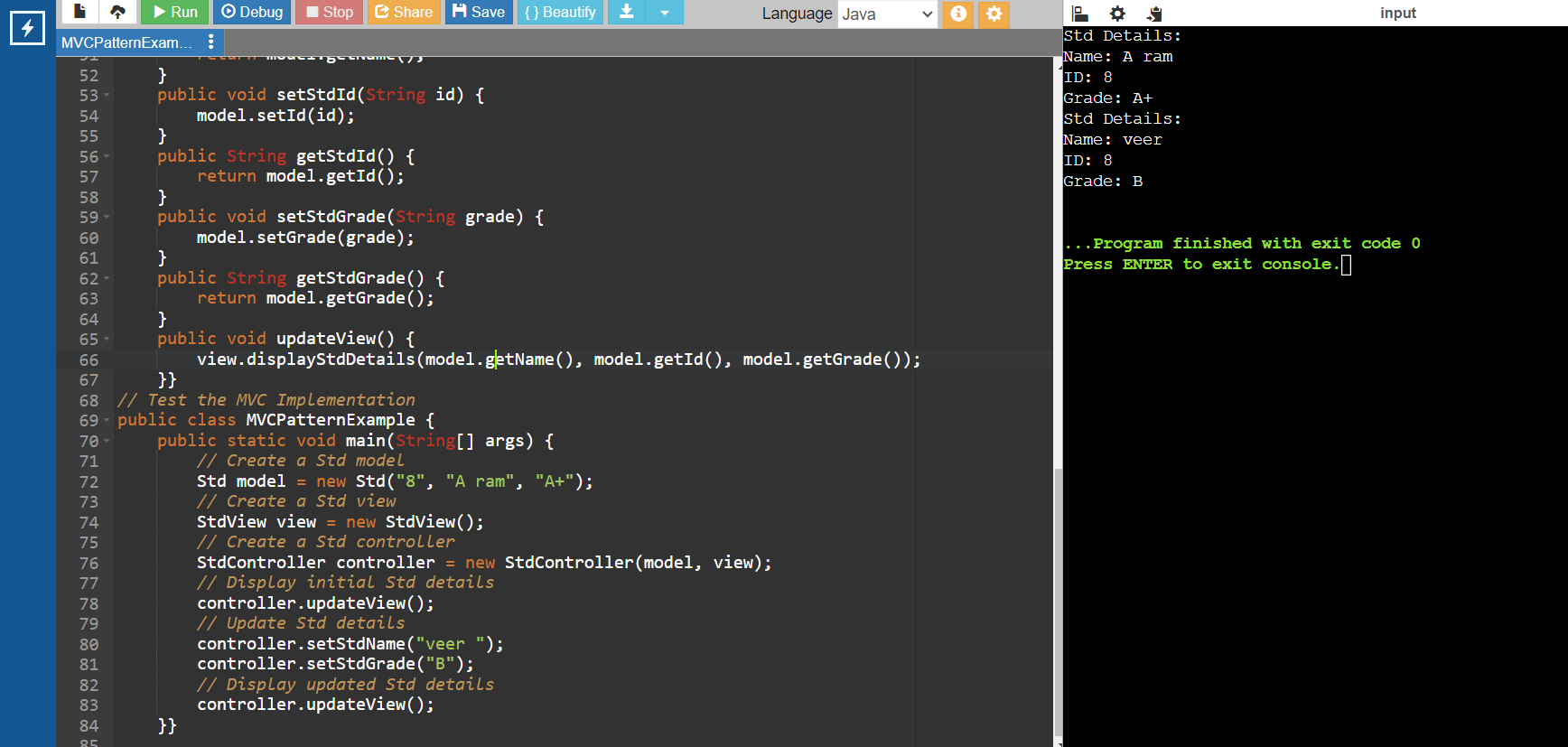
// Update Std details

controller.setStdName("veer ");

controller.setStdGrade("B");

// Display updated Std details

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 Cstr\_Repo {

String findCustomerById(String id);

}

// Implement Concrete Repository

class Cstr\_RepoImpl implements Cstr\_Repo {

public String findCustomerById(String id) {

// Mock implementation, in real scenario, it would interact with a database

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

return "Raj";

} else {

return "Customer not found";

}}}

// Define Service Class

class CustomerService {

private Cstr\_Repo Cstr\_Repo;

// Implement Dependency Injection

public CustomerService(Cstr\_Repo Cstr\_Repo) {

this.Cstr\_Repo = Cstr\_Repo;

}

public String getCustomerDetails(String id) {

return Cstr\_Repo.findCustomerById(id);

}}

// Test the Dependency Injection Implementation

public class DependencyInjection {

public static void main(String[] args) {

// Create a Cstr\_Repo instance

Cstr\_Repo Cstr\_Repo = new Cstr\_RepoImpl();

// Inject the repository into the service

CustomerService customerService = new CustomerService(Cstr\_Repo);

// Use the service to find customer details

String customerDetails = customerService.getCustomerDetails("1");

System.out.println("Customer Details: " + customerDetails);

}

} 