**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 LogManager {

private static LogManager uniqueInstance;

private LogManager() {

}

public static LogManager getInstance() {

if (uniqueInstance == null) {

uniqueInstance = new LogManager();

}

return uniqueInstance;

}

public void record(String message) {

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

}

}

public class SingletonDemo1 {

public static void main(String[] args) {

LogManager manager1 = LogManager.getInstance();

LogManager manager2 = LogManager.getInstance();

manager1.record("This is the first log message.");

manager2.record("This is the second log message.");

if (manager1 == manager2) {

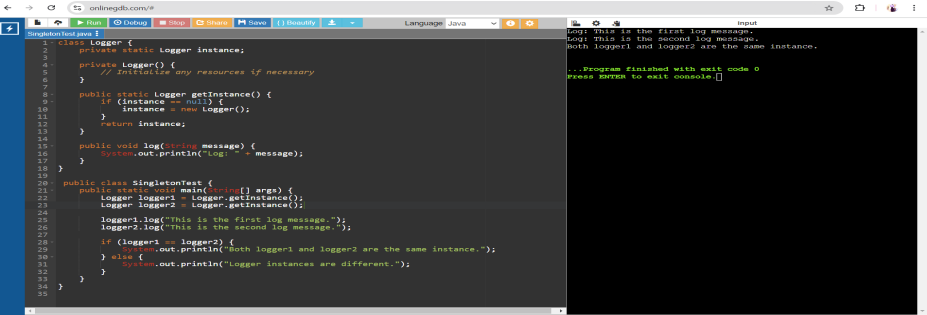
System.out.println("Both manager1 and manager2 are the same instance.");

} else {

System.out.println("Manager 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 WordDocument implements Document {

public void open() {

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

}

public void close() {

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

}

}

class PdfDocument implements Document {

public void open() {

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

}

public void close() {

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

}

}

class ExcelDocument implements Document {

public void open() {

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

}

public void close() {

System.out.println("Closing 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 FactoryMethodPatternExample {

public static void main(String[] args) {

DocumentFactory wordFactory = new WordDocumentFactory();

Document wordDocument = wordFactory.createDocument();

wordDocument.open();

wordDocument.close();

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdfDocument = pdfFactory.createDocument();

pdfDocument.open();

pdfDocument.close();

DocumentFactory excelFactory = new ExcelDocumentFactory();

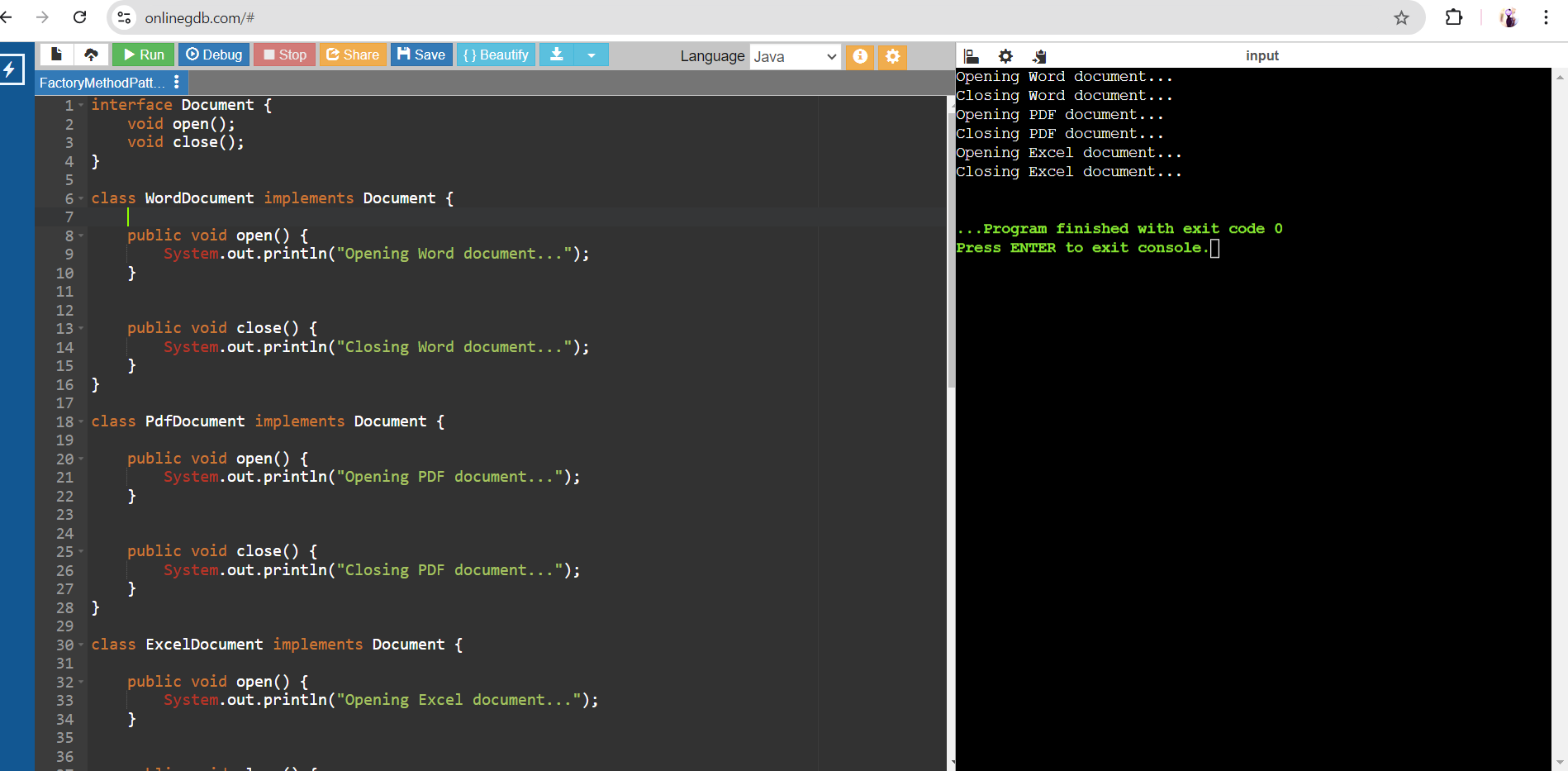
Document excelDocument = excelFactory.createDocument();

excelDocument.open();

excelDocument.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 BuilderPatternExample1 {

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")

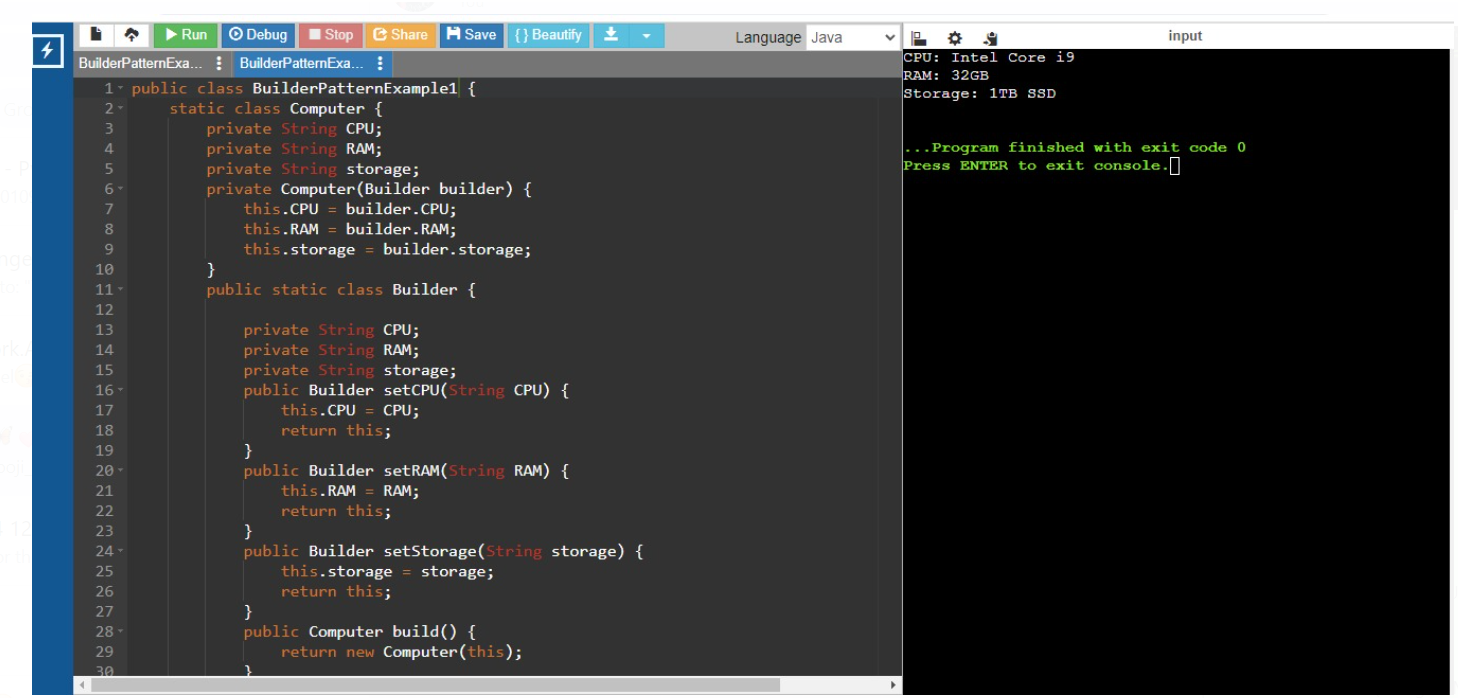
.build();

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

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

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

}



**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 PaymentProcessor {

void processPayment(double amount);

}

class PayWithGoogle {

public void makePayment(double amount) {

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

}

}

class TransactionSquare {

public void executePayment(double amount) {

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

}

}

class PayWithApple {

public void conductTransaction(double amount) {

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

}

}

class PayWithGoogleAdapter implements PaymentProcessor {

private PayWithGoogle payWithGoogle;

public PayWithGoogleAdapter(PayWithGoogle payWithGoogle) {

this.payWithGoogle = payWithGoogle;

}

public void processPayment(double amount) {

payWithGoogle.makePayment(amount);

}

}

class TransactionSquareAdapter implements PaymentProcessor {

private TransactionSquare transactionSquare;

public TransactionSquareAdapter(TransactionSquare transactionSquare) {

this.transactionSquare = transactionSquare;

}

public void processPayment(double amount) {

transactionSquare.executePayment(amount);

}

}

class PayWithAppleAdapter implements PaymentProcessor {

private PayWithApple payWithApple;

public PayWithAppleAdapter(PayWithApple payWithApple) {

this.payWithApple = payWithApple;

}

public void processPayment(double amount) {

payWithApple.conductTransaction(amount);

}

}

public class AdapterPatternExample {

public static void main(String[] args) {

PayWithGoogle payWithGoogle = new PayWithGoogle();

TransactionSquare transactionSquare = new TransactionSquare();

PayWithApple payWithApple = new PayWithApple();

PaymentProcessor payWithGoogleAdapter = new PayWithGoogleAdapter(payWithGoogle);

PaymentProcessor transactionSquareAdapter = new TransactionSquareAdapter(transactionSquare);

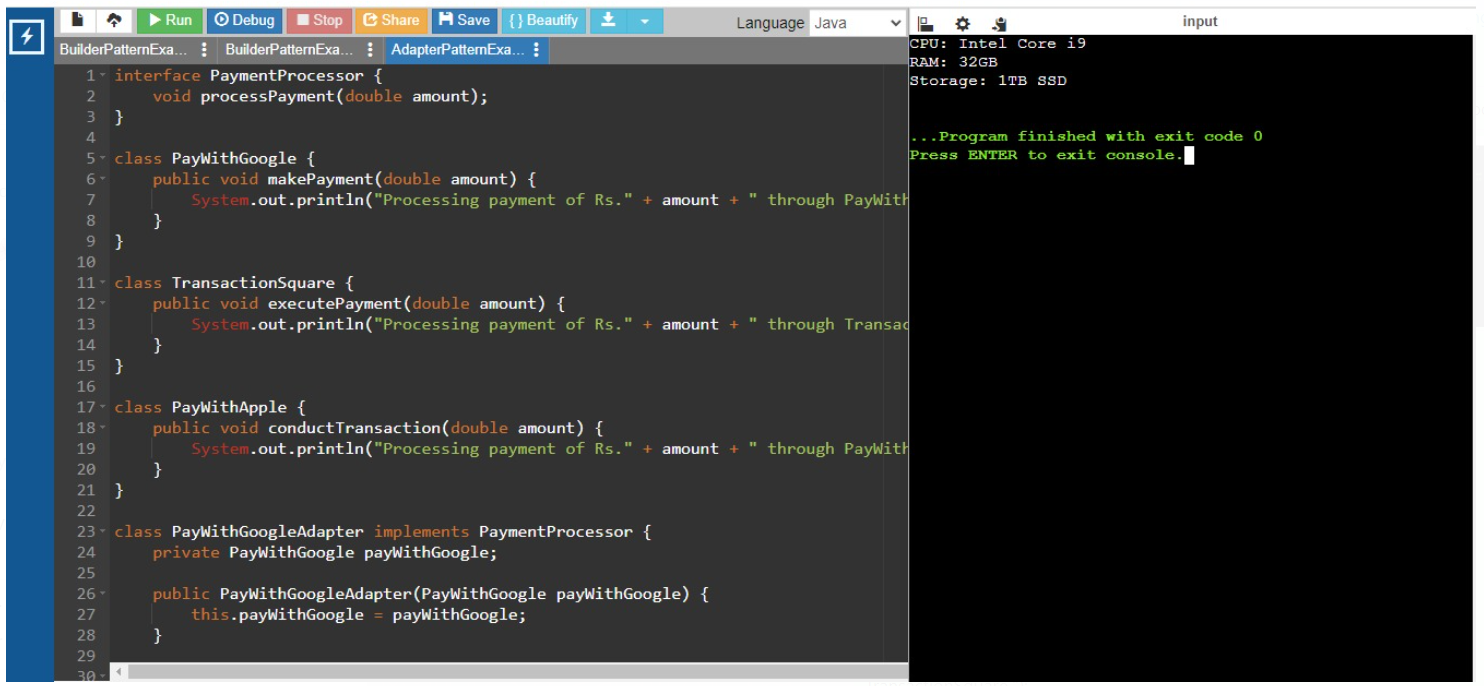
PaymentProcessor payWithAppleAdapter = new PayWithAppleAdapter(payWithApple);

payWithGoogleAdapter.processPayment(100.00);

transactionSquareAdapter.processPayment(200.00);

payWithAppleAdapter.processPayment(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 deliver(String message);

}

class EmailMessenger implements Messenger {

public void deliver(String message) {

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

}

}

abstract class MessengerDecorator implements Messenger {

protected Messenger messenger;

public MessengerDecorator(Messenger messenger) {

this.messenger = messenger;

}

public void deliver(String message) {

messenger.deliver(message);

}

}

class SMSDecorator extends MessengerDecorator {

public SMSDecorator(Messenger messenger) {

super(messenger);

}

public void deliver(String message) {

messenger.deliver(message);

sendSMS(message);

}

private void sendSMS(String message) {

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

}

}

class SlackDecorator extends MessengerDecorator {

public SlackDecorator(Messenger messenger) {

super(messenger);

}

public void deliver(String message) {

messenger.deliver(message);

sendSlack(message);

}

private void sendSlack(String message) {

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

}

}

public class DecoratorPatternExample {

public static void main(String[] args) {

Messenger emailMessenger = new EmailMessenger();

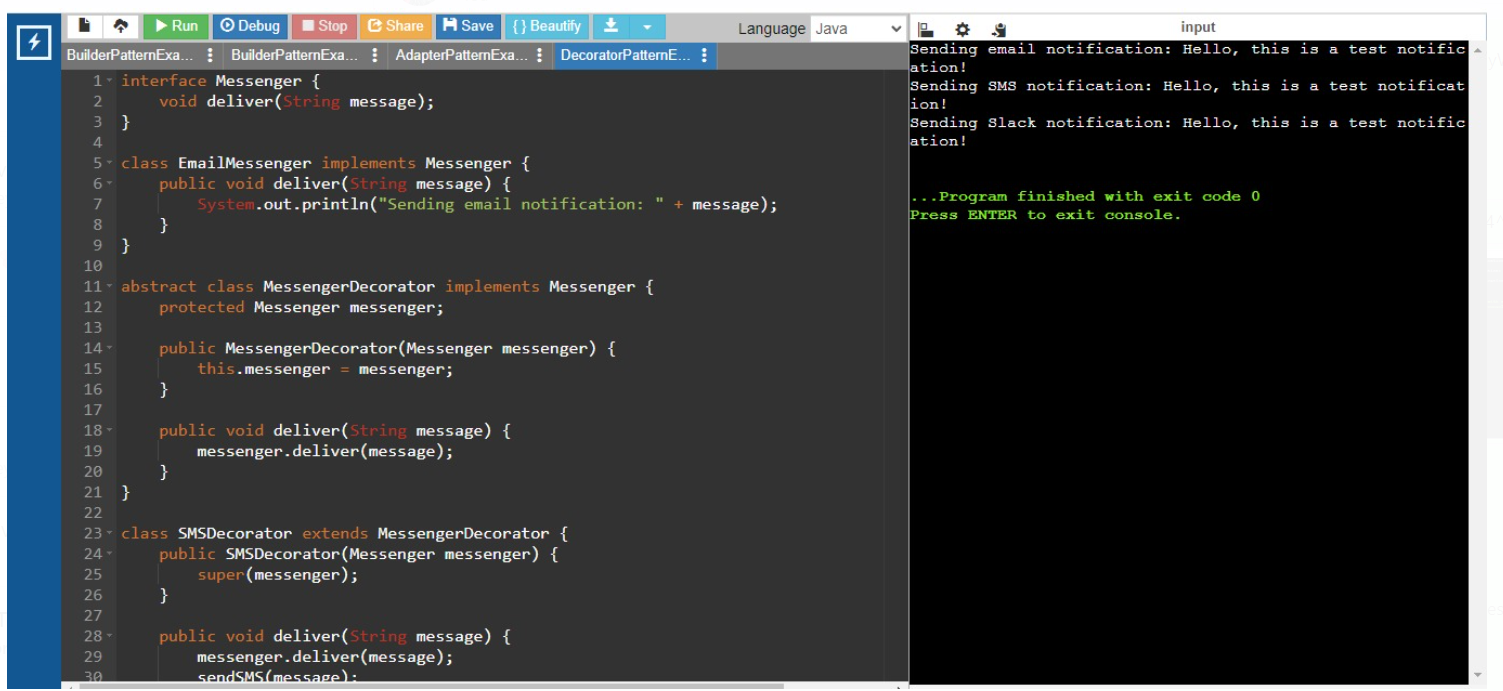
Messenger smsDecorator = new SMSDecorator(emailMessenger);

Messenger slackDecorator = new SlackDecorator(smsDecorator);

slackDecorator.deliver("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;

loadFromDisk();

}

private void loadFromDisk() {

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 Main {

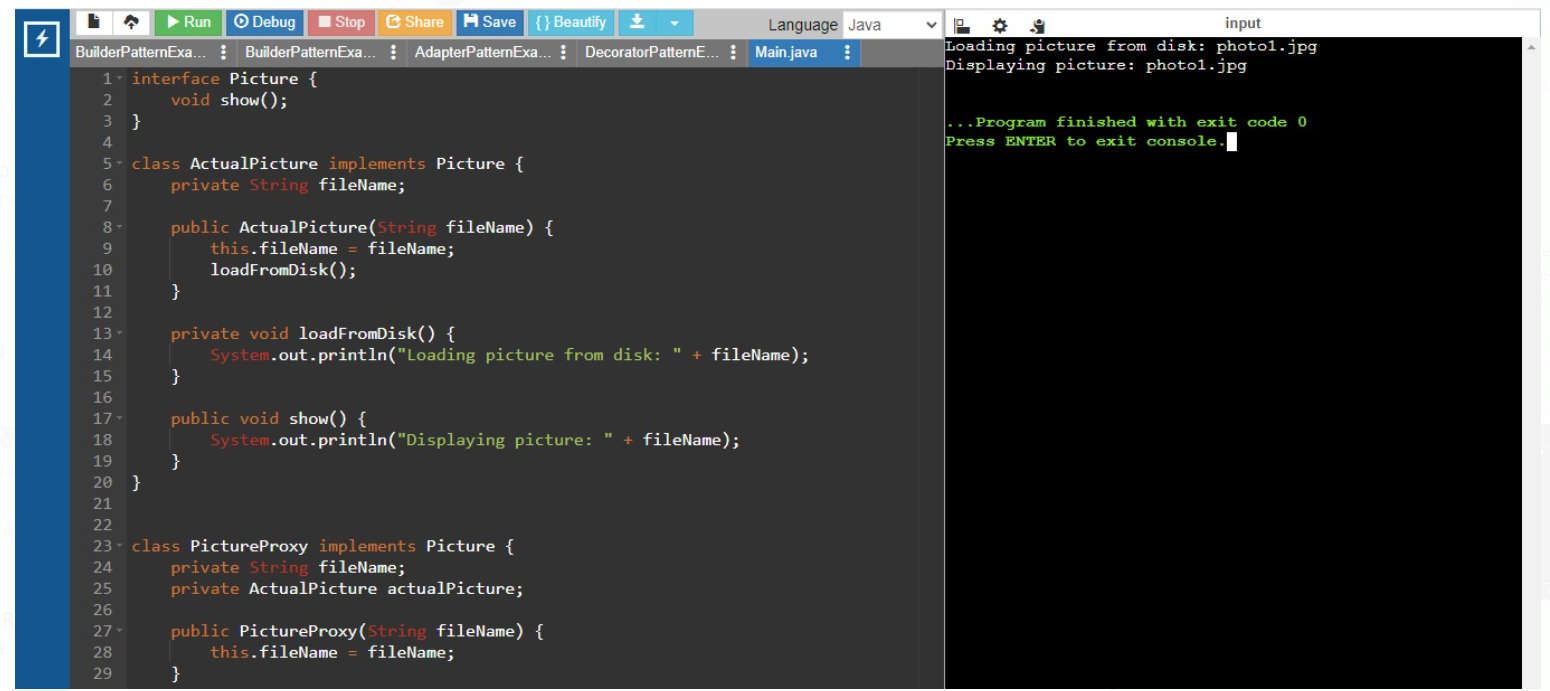
public static void main(String[] args) {

Picture pic1 = new PictureProxy("photo1.jpg");

Picture pic2 = new PictureProxy("photo2.jpg");

pic1.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 addObserver(Observer o);

void removeObserver(Observer o);

void notifyObservers();

}

class StockExchange implements Market {

private List<Observer> observers;

private double price;

public StockExchange() {

this.observers = new ArrayList<>();

}

public void addObserver(Observer o) {

observers.add(o);

}

public void removeObserver(Observer o) {

observers.remove(o);

}

public void notifyObservers() {

for (Observer o : observers) {

o.update(price);

}

}

public void setPrice(double price) {

this.price = price;

notifyObservers();

}

}

interface Observer {

void update(double price);

}

class PhoneApp implements Observer {

private String name;

public PhoneApp(String name) {

this.name = name;

}

public void update(double price) {

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

}

}

class BrowserApp implements Observer {

private String name;

BrowserApp(String name) {

this.name = name;

}

public void update(double price) {

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

}

}

public class Main {

public static void main(String[] args) {

StockExchange stockExchange = new StockExchange();

Observer phoneApp = new PhoneApp("PhoneApp");

Observer browserApp = new BrowserApp("BrowserApp");

stockExchange.addObserver(phoneApp);

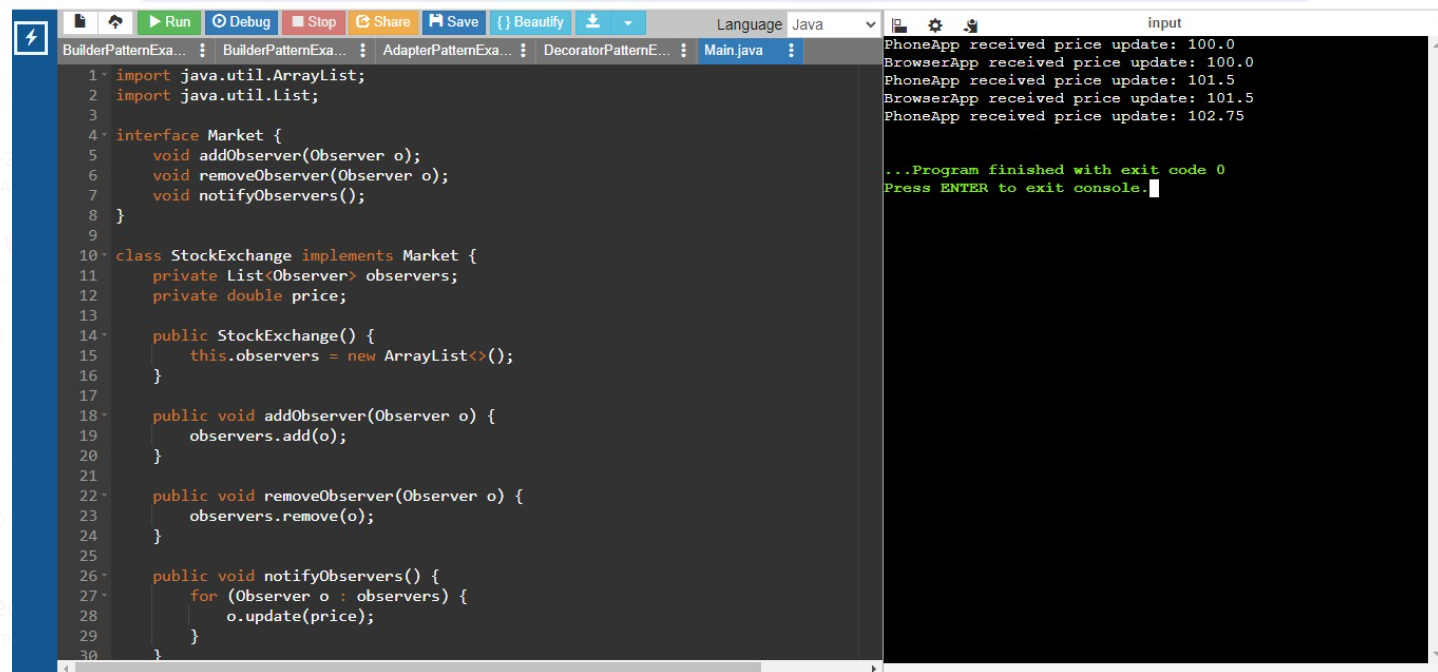
stockExchange.addObserver(browserApp);

stockExchange.setPrice(100.00);

stockExchange.setPrice(101.50);

stockExchange.removeObserver(browserApp);

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 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 processPayment(double amount) {

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

}

}

class EWalletPayment implements PaymentMethod {

private String accountEmail;

private String accountPassword;  
 public EWalletPayment(String accountEmail, String accountPassword) {

this.accountEmail = accountEmail;

this.accountPassword = accountPassword;

}

public void processPayment(double amount) {

System.out.println("Paid " + amount + " using E-Wallet.");

}

}

class PaymentContext {

private PaymentMethod paymentMethod;

public void setPaymentMethod(PaymentMethod paymentMethod) {

this.paymentMethod = paymentMethod;

}  
public void executePayment(double amount) {

paymentMethod.processPayment(amount);

}

}

public class StrategyPatternExample {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

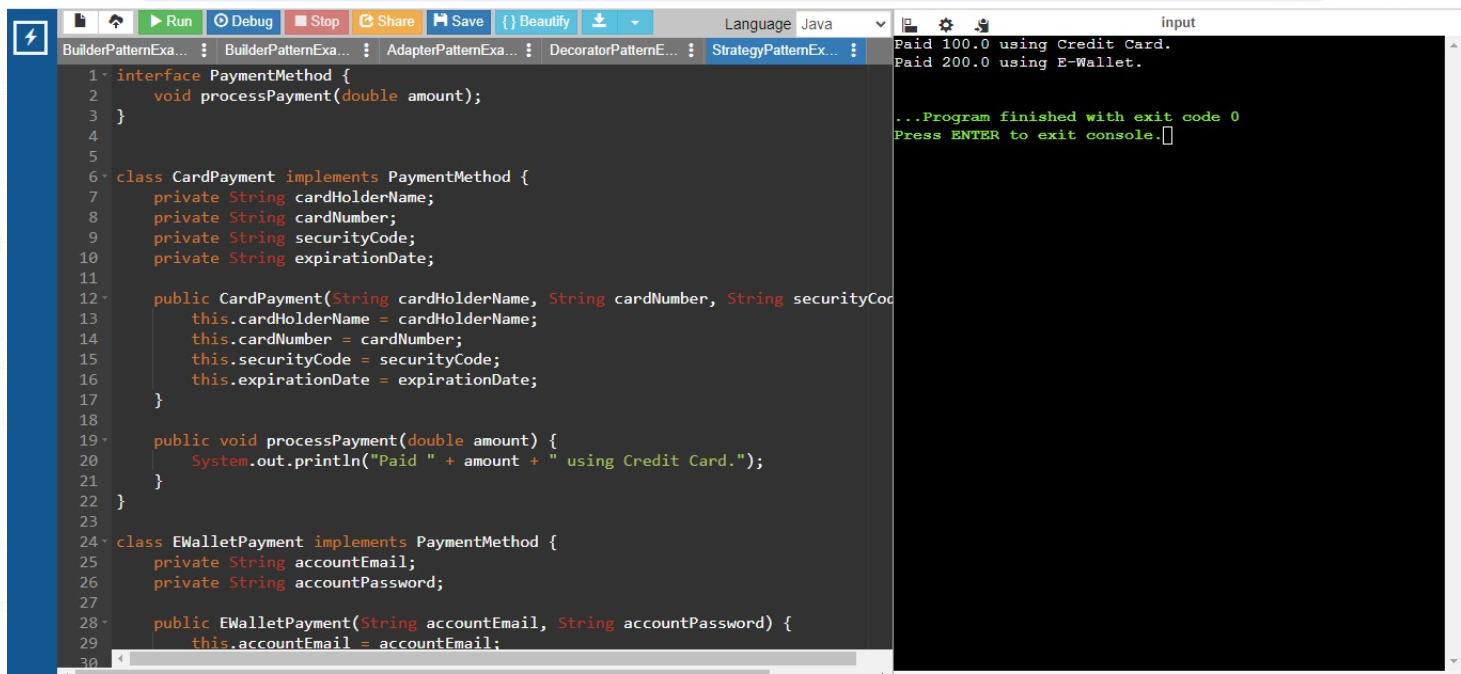
context.setPaymentMethod(new CardPayment("John Doe", "1234567890123456", "123", "12/23"));

context.executePayment(100.0);

context.setPaymentMethod(new EWalletPayment("john.doe@example.com", "password123"));

context.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.

public class CommandPattern {

public static void main(String[] args) {

Light livingRoomLight = new Light();

// Create command objects

Command lightOnCommand = new LightOnCommand(livingRoomLight);

Command lightOffCommand = new LightOffCommand(livingRoomLight);

// Create a remote control instance

RemoteControl remoteControl = new RemoteControl();

// Set command to turn on the light and press the button

remoteControl.setCommand(lightOnCommand);

remoteControl.pressButton();

// Set command to turn off the light and press the button

remoteControl.setCommand(lightOffCommand);

remoteControl.pressButton();

}

}

// Command interface

interface Command {

void execute();

}

// Command to turn on the light

class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOn();

}

}

// Command to turn off the light

class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOff();

}

}

// Receiver class

class Light {

public void turnOn() {

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

}

public void turnOff() {

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

}

}

// Invoker class

class RemoteControl {

private Command command;

// Set command to be executed

public void setCommand(Command command) {

this.command = command;

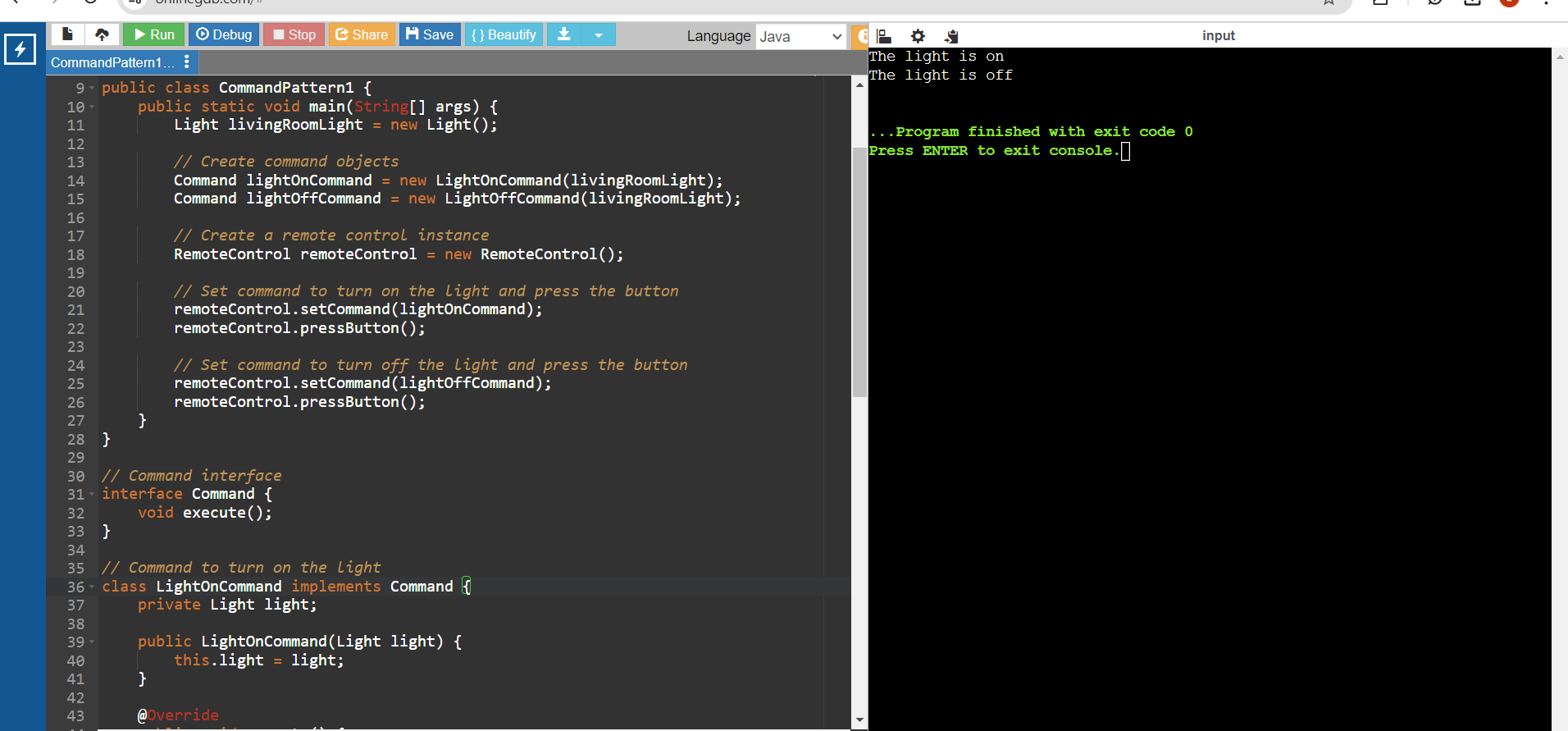
}

// Press button to execute the command

public void pressButton() {

command.execute();

}

}  


**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

public class MVCPExample {

public static void main(String[] args) {

Student studentModel = new Student("1", "John Doe", "A");

StudentView studentView = new StudentView();

StudentController studentController = new StudentController(studentModel, studentView);

studentController.updateView();

studentController.setStudentName("Jane Doe");

studentController.setStudentGrade("B");

studentController.updateView();

}

}

class Student {

private String studentId;

private String studentName;

private String studentGrade;

public Student(String studentId, String studentName, String studentGrade) {

this.studentId = studentId;

this.studentName = studentName;

this.studentGrade = studentGrade;

}

public String getStudentId() {

return studentId;

}

public void setStudentId(String studentId) {

this.studentId = studentId;

}

public String getStudentName() {

return studentName;

}

public void setStudentName(String studentName) {

this.studentName = studentName;

}

public String getStudentGrade() {

return studentGrade;

}

public void setStudentGrade(String studentGrade) {

this.studentGrade = studentGrade;

}

}

class StudentView {

public void displayStudentInfo(String studentName, String studentId, String studentGrade) {

System.out.println("Student Information:");

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

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

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

}

}

class StudentController {

private Student studentModel;

private StudentView studentView;

public StudentController(Student studentModel, StudentView studentView) {

this.studentModel = studentModel;

this.studentView = studentView;

}

public void setStudentName(String studentName) {

studentModel.setStudentName(studentName);

}

public String getStudentName() {

return studentModel.getStudentName();

}

public void setStudentId(String studentId) {

studentModel.setStudentId(studentId);

}

public String getStudentId() {

return studentModel.getStudentId();

}

public void setStudentGrade(String studentGrade) {

studentModel.setStudentGrade(studentGrade);

}

public String getStudentGrade() {

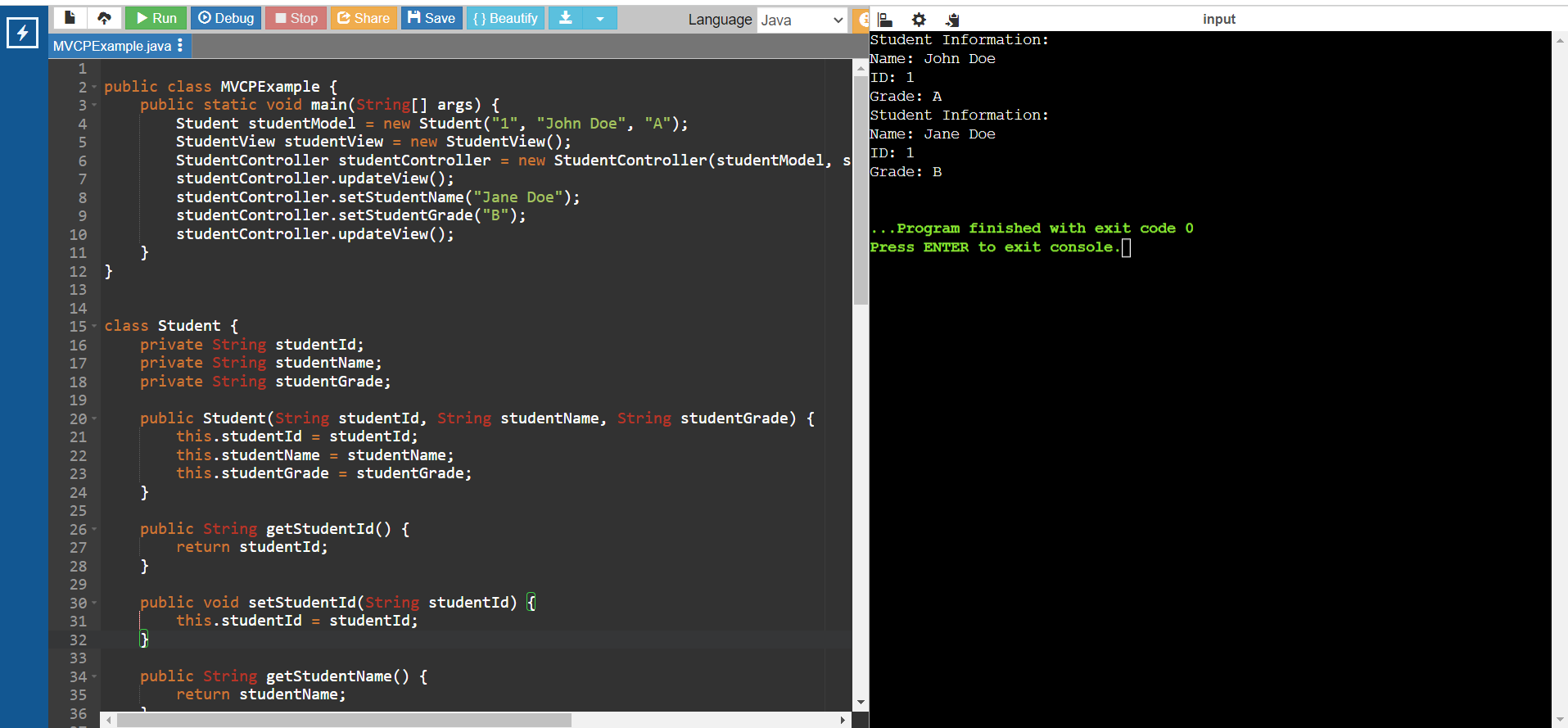
return studentModel.getStudentGrade();

}

public void updateView() {

studentView.displayStudentInfo(studentModel.getStudentName(), studentModel.getStudentId(), studentModel.getStudentGrade());

}

}  


**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 CustomerRepository {

String findCustomerById(String id);

}

// Implement Concrete Repository

class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(String id) {

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

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

return "John Doe";

} else {

return "Customer not found";

}}}

// Define Service Class

class CustomerService {

private CustomerRepository customerRepository;

// Implement Dependency Injection

public CustomerService(CustomerRepository customerRepository) {

this.customerRepository = customerRepository;

}

public String getCustomerDetails(String id) {

return customerRepository.findCustomerById(id);

}}

// Test the Dependency Injection Implementation

public class DIExample1 {

public static void main(String[] args) {

// Create a CustomerRepository instance

CustomerRepository customerRepository = new CustomerRepositoryImpl();

// Inject the repository into the service

CustomerService customerService = new CustomerService(customerRepository);

// Use the service to find customer details

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

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

}

}

