**DESIGN PATTERNS AND PRINCIPLES**

**Exercise 1: Implementing the Singleton Pattern**

Logger.java

package DesignPatternsAndPrinciples;

public class Logger {

private static Logger *instance*;

private Logger() {

System.*out*.println("logger instance created");

}

public static Logger getInstance() {

if(*instance*==null) {

*instance*=new Logger();

}

return *instance*;

}

public void log(String msg) {

System.*out*.println("Log: "+msg);

}

}

SingletonPatternExample.java

package DesignPatternsAndPrinciples;

public class SingletonPatternExample {

public static void main(String[] args) {

Logger l1=Logger.*getInstance*();

Logger l2=Logger.*getInstance*();

l1.log("first log");

l2.log("second log");

if(l1==l2) {

System.*out*.println("Same instance");

}

else {

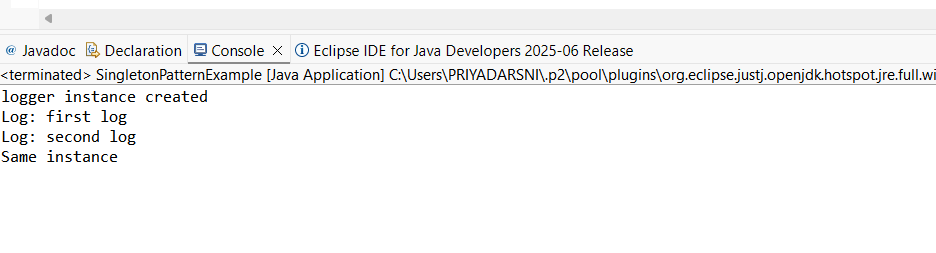
System.*out*.println("different instance");

}

}

}

OUTPUT:



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

Code:

Document.java

public interface Document {

void open();

void close();

}

WordDocument.java

public class WordDocument implements Document {

public void open() {

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

}

public void close() {

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

}

}

PdfDocument.java

public class PdfDocument implements Document {

public void open() {

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

}

public void close() {

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

}

}

ExcelDocument.java

public class ExcelDocument implements Document {

public void open() {

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

}

public void close() {

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

}

}

DocumentFactory.java

public abstract class DocumentFactory {

public abstract Document createDocument();

}

WordDocumentFactory.java

public class WordDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

PdfDocumentFactory.java

public class PdfDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

ExcelDocumentFactory.java

public class ExcelDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}}

DocumentManagementSystem.java

public class DocumentManagementSystem {

public static void main(String[] args) {

DocumentFactory wordFactory = new WordDocumentFactory();

Document wordDoc = wordFactory.createDocument();

wordDoc.open();

wordDoc.close();

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open();

pdfDoc.close();

DocumentFactory excelFactory = new ExcelDocumentFactory();

Document excelDoc = excelFactory.createDocument();

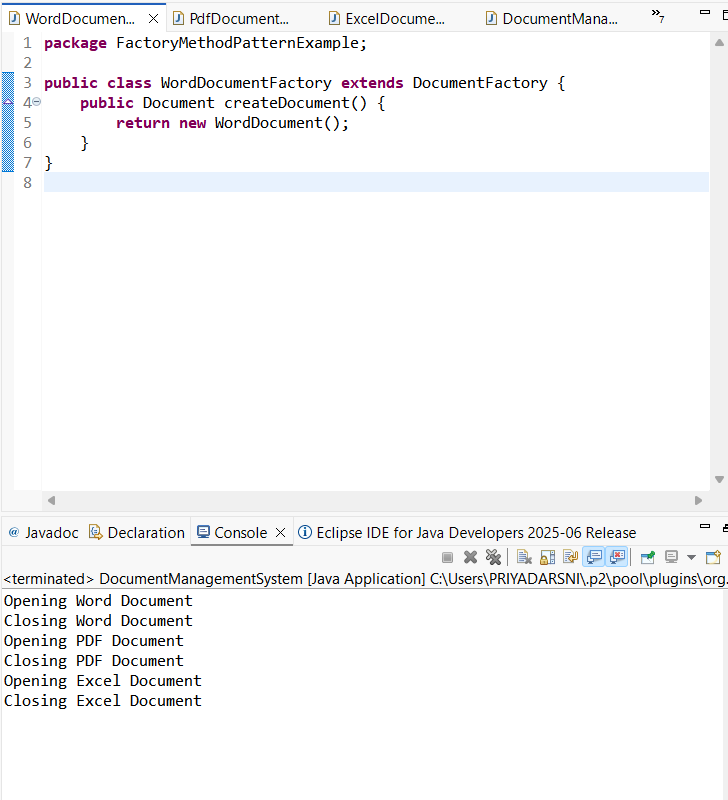
excelDoc.open();

excelDoc.close();

}

}

OUTPUT:



**Exercise 3: Implementing the Builder Pattern**

Computer.java

public class Computer {

private String CPU;

private String RAM;

private String storage;

private String GPU;

private String powerSupply;

private Computer(Builder builder) {

this.CPU = builder.CPU;

this.RAM = builder.RAM;

this.storage = builder.storage;

this.GPU = builder.GPU;

this.powerSupply = builder.powerSupply;

}

public String toString() {

return "Computer{" + "CPU='" + CPU + '\'' + ", RAM='" + RAM + '\'' + ", storage='" + storage + '\'' + ", GPU='" + GPU + '\'' + ", powerSupply='" + powerSupply + '\'' + '}';

}

public static class Builder {

private String CPU;

private String RAM;

private String storage;

private String GPU;

private String powerSupply;

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 Builder setGPU(String GPU) {

this.GPU = GPU;

return this;

}

public Builder setPowerSupply(String powerSupply) {

this.powerSupply = powerSupply;

return this;

}

public Computer build() {

return new Computer(this);

}

}

}

BuilderPatternTest.java

public class BuilderPatternTest {

public static void main(String[] args) {

Computer basicComputer = new Computer.Builder()

.setCPU("Intel i5")

.setRAM("8GB")

.setStorage("256GB SSD")

.build();

Computer gamingComputer = new Computer.Builder()

.setCPU("AMD Ryzen 7")

.setRAM("16GB")

.setStorage("1TB SSD")

.setGPU("NVIDIA RTX 3060")

.setPowerSupply("750W")

.build();

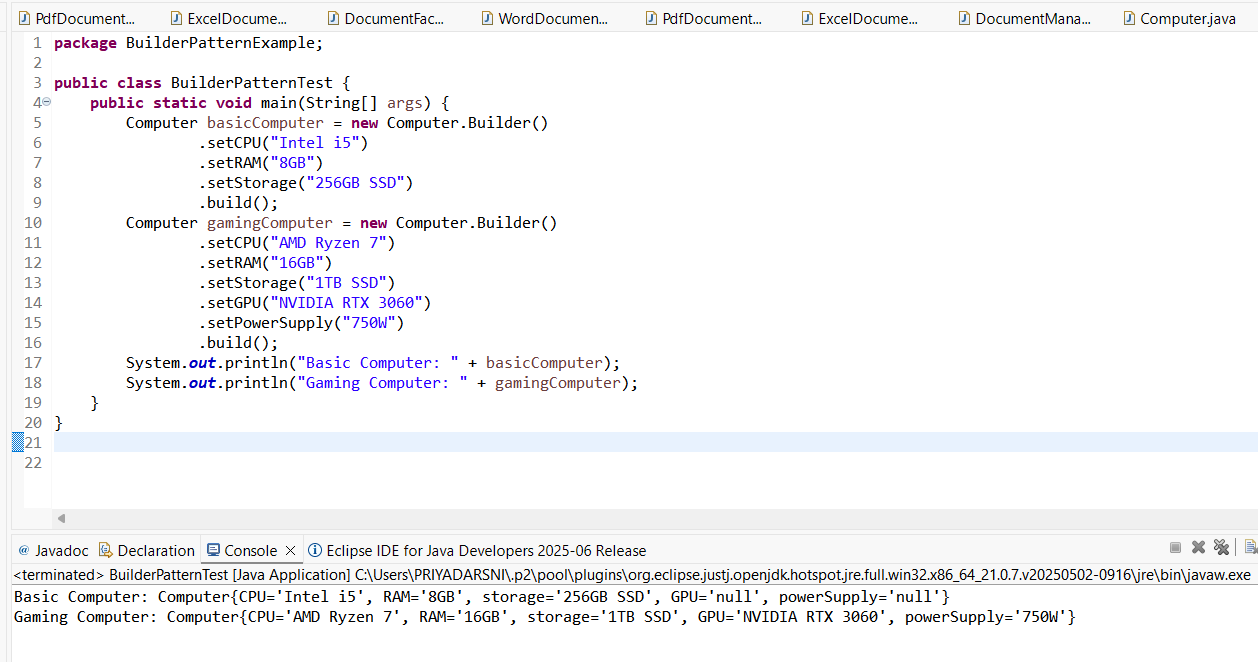
System.out.println("Basic Computer: " + basicComputer);

System.out.println("Gaming Computer: " + gamingComputer);

}

}

OUTPUT:



**Exercise 4: Implementing the Adapter Pattern**

Code:

PaymentProcessor.java

public interface PaymentProcessor {

void processPayment(double amount);

}

PayPal.java

public class PayPal {

public void makePayment(double amount) {

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

}

}

Stripe.java

public class Stripe {

public void charge(double amount) {

System.out.println("Charging $" + amount + " using Stripe.");

}

}

PayPalAdapter.java

public class PayPalAdapter implements PaymentProcessor {

private PayPal payPal;

public PayPalAdapter() {

this.payPal = new PayPal();

}

public void processPayment(double amount) {

payPal.makePayment(amount);

}

}

StripeAdapter.java

public class StripeAdapter implements PaymentProcessor {

private Stripe stripe;

public StripeAdapter() {

this.stripe = new Stripe();

}

public void processPayment(double amount) {

stripe.charge(amount);

}

}

PaymentProcessorTest.java

public class PaymentProcessorTest {

public static void main(String[] args) {

PaymentProcessor payPalProcessor = new PayPalAdapter();

payPalProcessor.processPayment(10000.00);

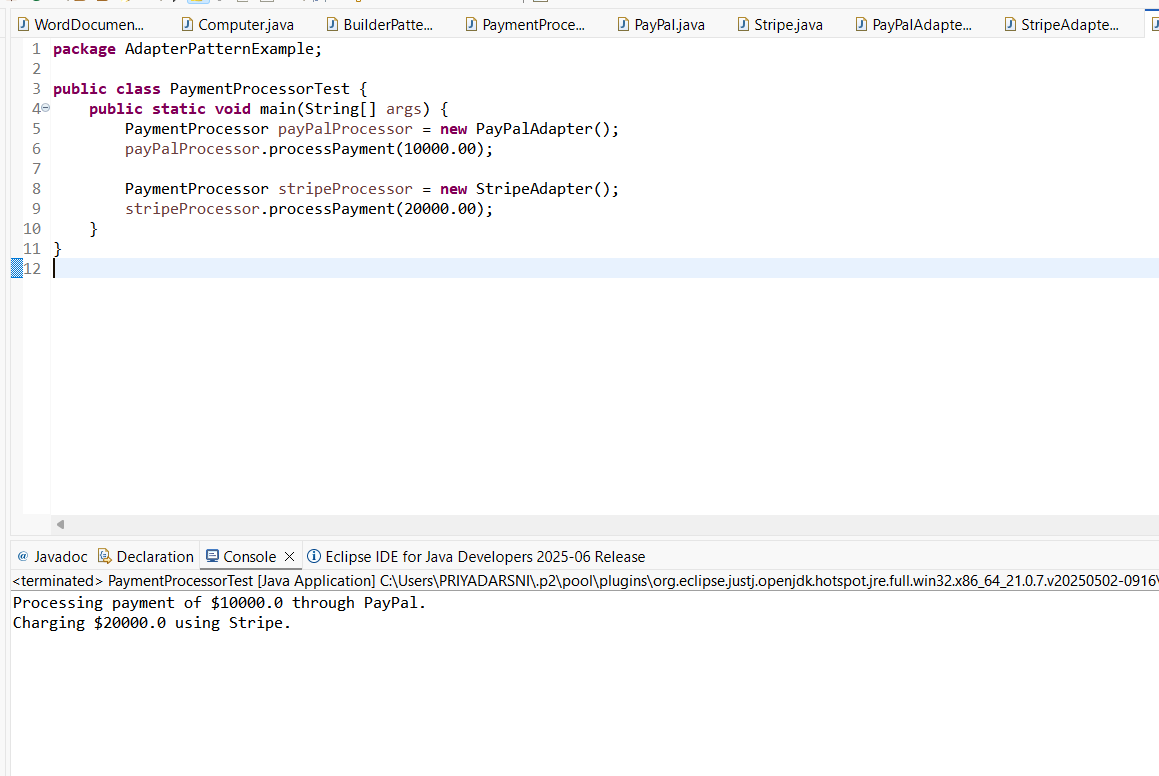
PaymentProcessor stripeProcessor = new StripeAdapter();

stripeProcessor.processPayment(20000.00);

}

}

OUTPUT:



**Exercise 5: Implementing the Decorator Pattern**

Code:

Notifier.java

public interface Notifier {

void send(String message);

}

EmailNotifier.java

public class EmailNotifier implements Notifier {

public void send(String message) {

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

}

}

NotifierDecorator.java

public abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

public void send(String message) {

notifier.send(message);

}

}

SMSNotifierDecorator.java

public 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("Sending SMS: " + message);

}

}

SlackNotifierDecorator.java

public 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("Sending Slack message: " + message);

}

}

NotificationTest.java

public class NotificationTest {

public static void main(String[] args) {

Notifier emailNotifier = new EmailNotifier();

emailNotifier.send("Hello, this is a test email!");

Notifier smsNotifier = new SMSNotifierDecorator(emailNotifier);

smsNotifier.send("Hello, this is a test email with SMS!");

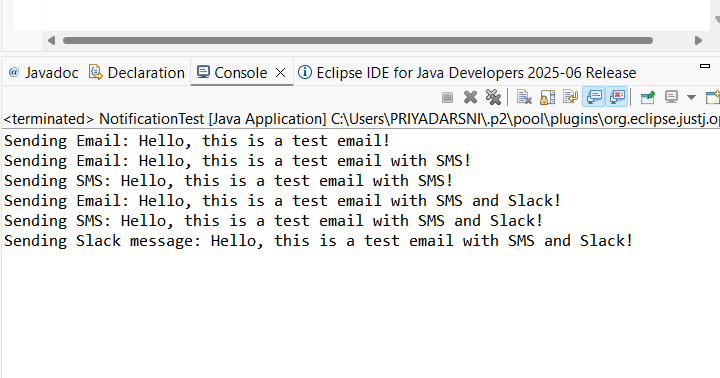
Notifier slackNotifier = new SlackNotifierDecorator(smsNotifier);

slackNotifier.send("Hello, this is a test email with SMS and Slack!");

}

}

OUTPUT:



**Exercise 6: Implementing the Proxy Pattern**

Code:

Image.java

public interface Image {

void display();

}

RealImage.java

public class RealImage implements Image {

private String imagePath;

public RealImage(String imagePath) {

this.imagePath = imagePath;

loadImageFromServer();

}

private void loadImageFromServer() {

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

}

public void display() {

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

}

}

ProxyImage.java

public class ProxyImage implements Image {

private RealImage realImage;

private String imagePath;

public ProxyImage(String imagePath) {

this.imagePath = imagePath;

}

public void display() {

if (realImage == null) {

realImage = new RealImage(imagePath);

}

realImage.display();

}

}

ProxyPatternTest.java

public class ProxyPatternTest {

public static void main(String[] args) {

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

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

image1.display();

image1.display();

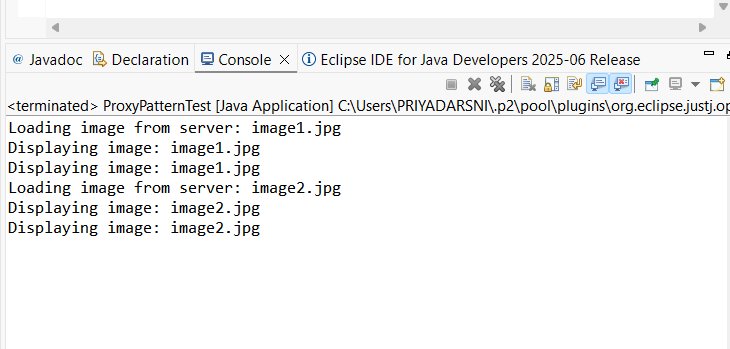
image2.display();

image2.display();

}

}

OUTPUT:



**Exercise 7: Implementing the Observer Pattern**

Stock.java

public interface Stock {

void registerObserver(Observer observer);

void deregisterObserver(Observer observer);

void notifyObservers();

}

StockMarket.java

public class StockMarket implements Stock {

private List<Observer> observers;

private double stockPrice;

public StockMarket() {

observers = new ArrayList<>();

}

public void setStockPrice(double stockPrice) {

this.stockPrice = stockPrice;

notifyObservers();

}

public double getStockPrice() {

return stockPrice;

}

public void registerObserver(Observer observer) {

observers.add(observer);

}

public void deregisterObserver(Observer observer) {

observers.remove(observer);

}

public void notifyObservers() {

for (Observer observer : observers) {

observer.update(stockPrice);

}

}

}

Observer.java

public interface Observer {

void update(double stockPrice);

}

MobileApp.java

public class MobileApp implements Observer {

public void update(double stockPrice) {

System.out.println("Mobile App: Stock price updated to $" + stockPrice);

}

}

WebApp.java

public class WebApp implements Observer {

public void update(double stockPrice) {

System.out.println("Web App: Stock price updated to $" + stockPrice);

}

}

ObserverPatternTest.java

public class ObserverPatternTest {

public static void main(String[] args) {

StockMarket stockMarket = new StockMarket();

MobileApp mobileApp = new MobileApp();

WebApp webApp = new WebApp();

stockMarket.registerObserver(mobileApp);

stockMarket.registerObserver(webApp);

stockMarket.setStockPrice(100.50);

stockMarket.setStockPrice(102.75);

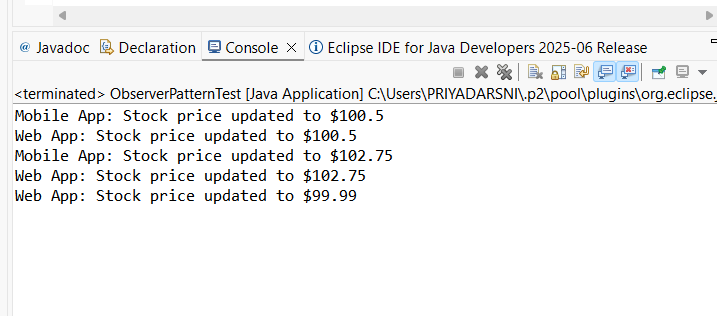
stockMarket.deregisterObserver(mobileApp);

stockMarket.setStockPrice(99.99);

}

}

OUTPUT:



**Exercise 8: Implementing the Strategy Pattern**

Code:

PaymentStrategy.java

public interface PaymentStrategy {

void pay(double amount);

}

CreditCardPayment.java

public 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);

}

}

PayPalPayment.java

public 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);

}

}

PaymentContext.java

public class PaymentContext {

private PaymentStrategy paymentStrategy;

public void setPaymentStrategy(PaymentStrategy paymentStrategy) {

this.paymentStrategy = paymentStrategy;

}

public void executePayment(double amount) {

if (paymentStrategy != null) {

paymentStrategy.pay(amount);

} else {

System.out.println("Payment strategy not set.");

}

}

}

StrategyPatternTest.java

public class StrategyPatternTest {

public static void main(String[] args) {

PaymentContext paymentContext = new PaymentContext();

PaymentStrategy creditCardPayment = new CreditCardPayment("1234-5678-9876-5432");

paymentContext.setPaymentStrategy(creditCardPayment);

paymentContext.executePayment(15000.00);

PaymentStrategy payPalPayment = new PayPalPayment("helouser@123.com");

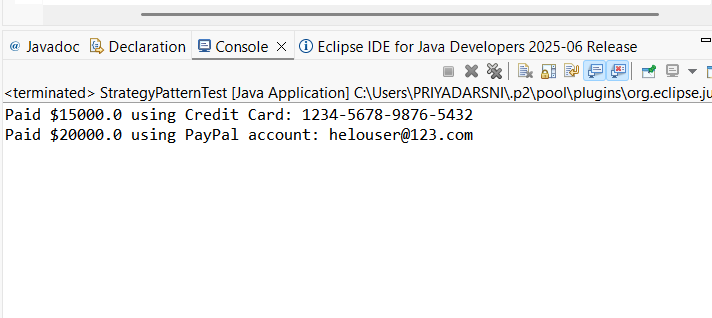
paymentContext.setPaymentStrategy(payPalPayment);

paymentContext.executePayment(20000.00);

}

}

OUTPUT:



**Exercise 9: Implementing the Command Pattern**

Code:

Command.java

public interface Command {

void execute();

}

LightOnCommand.java

public class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOn();

}

}

LightOffCommand.java

public class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOff();

}

}

RemoteControl.java

public 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.");

}

}

}

Light.java

public class Light {

public void turnOn() {

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

}

public void turnOff() {

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

}

}

CommandPatternTest.java

public class CommandPatternTest {

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**

Code:

Student.java

public class Student {

private String name;

private int id;

private String grade;

public Student(String name, int 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 int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getGrade() {

return grade;

}

public void setGrade(String grade) {

this.grade = grade;

}

}

StudentView.java

public class StudentView {

public void displayStudentDetails(String studentName, int studentId, String studentGrade) {

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

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

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

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

}

}

StudentController.java

public class StudentController {

private Student student;

private StudentView studentView;

public StudentController(Student student, StudentView studentView) {

this.student = student;

this.studentView = studentView;

}

public void setStudentName(String name) {

student.setName(name);

}

public String getStudentName() {

return student.getName();

}

public void setStudentId(int id) {

student.setId(id);

}

public int getStudentId() {

return student.getId();

}

public void setStudentGrade(String grade) {

student.setGrade(grade);

}

public String getStudentGrade() {

return student.getGrade();

}

public void updateView() {

studentView.displayStudentDetails(student.getName(), student.getId(), student.getGrade());

}

}

MVCPatternTest.java

public class MVCPatternTest {

public static void main(String[] args) {

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

StudentView studentView = new StudentView();

StudentController studentController = new StudentController(student, studentView);

studentController.updateView();

studentController.setStudentName("Jane Doe");

studentController.setStudentId(2);

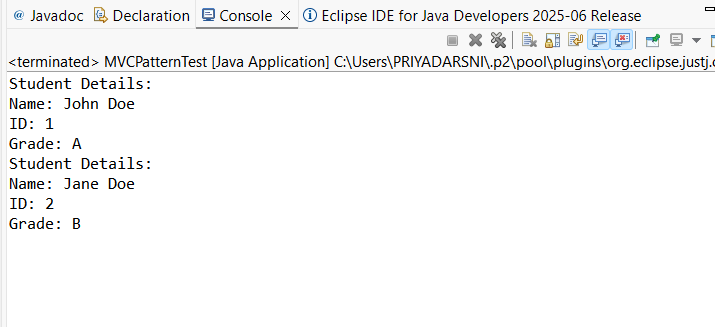
studentController.setStudentGrade("B");

studentController.updateView();

}

}

OUTPUT:



**Exercise 11: Implementing Dependency Injection**

Code:

CustomerRepository.java

public interface CustomerRepository {

String findCustomerById(int id);

}

CustomerRepositoryImpl.java

public class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(int id) {

return "Customer with ID: " + id;

}

}

CustomerService.java

public class CustomerService {

private CustomerRepository customerRepository;

public CustomerService(CustomerRepository customerRepository) {

this.customerRepository = customerRepository;

}

public String getCustomer(int id) {

return customerRepository.findCustomerById(id);

}

}

DependencyInjectionTest.java

public class DependencyInjectionTest {

public static void main(String[] args) {

CustomerRepository customerRepository = new CustomerRepositoryImpl();

CustomerService customerService = new CustomerService(customerRepository);

String customer = customerService.getCustomer(1);

System.out.println(customer);

}

}

OUTPUT:

