1. DESIGN PATTERNS AND PRINCIPLES

# Exercise 1: Implementing the Singleton Pattern

Logger.java

package SingletonPatternExample;

public class Logger{

    private static Logger instance;

    private Logger(){

        System.out.println("Logger Initialized");

    }

    public static Logger getInstance(){

        if(instance==null){

            instance = new Logger();

        }

        return instance;

    }

    public void log(String message){

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

    }

}

Main.java

package SingletonPatternExample;

public class Main {

    public static void main(String args[]){

        Logger logger1=Logger.getInstance();

        System.out.println("This is the first log");

        Logger logger2=Logger.getInstance();

        System.out.println("This is the second log");

        if(logger1==logger2){

            System.out.println("Both logger instances are same. Singleton works!");

        }

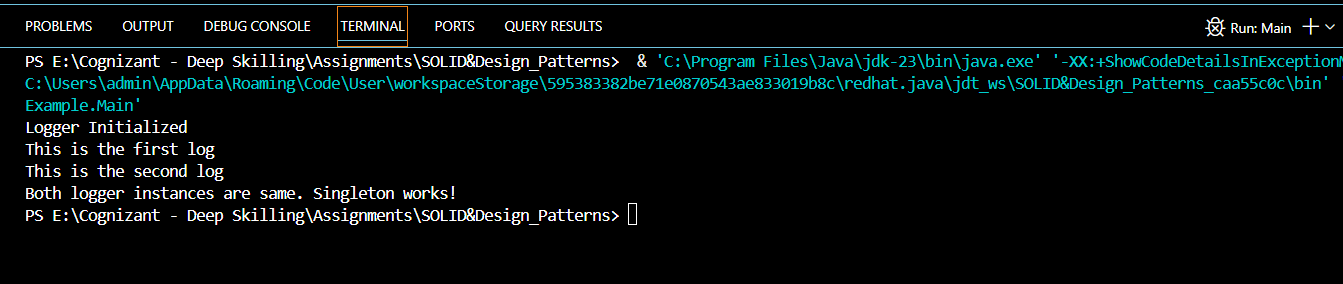
        else{

            System.out.println("Different instances detected. Singleton failed");

        }

    } }

Output:



# Exercise 2: Implementing the Factory Method Pattern

Document.java

package FactoryMethodPatternExample;

public interface Document {

    void open();

}

ExcelDocument.java

package FactoryMethodPatternExample;

public class ExcelDocument implements Document{

    @Override

    public void open(){

        System.out.println("This is Excel Document");

    }

}

PdfDocument.java

package FactoryMethodPatternExample;

public class PdfDocument implements Document{

    @Override

    public void open(){

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

    } }

WordDocument.java

package FactoryMethodPatternExample;

public class WordDocument implements Document{

    @Override

    public void open(){

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

    }

}

DocumentFactory.java

package FactoryMethodPatternExample;

public abstract class DocumentFactory {

    public abstract Document createDocument();

}

ExcelDocumentFactory.java

package FactoryMethodPatternExample;

public class ExcelDocumentFactory extends DocumentFactory {

    @Override

    public Document createDocument(){

        return new WordDocument();

    }

}

WordDocumentFactory.java

package FactoryMethodPatternExample;

public class WordDocumentFactory extends DocumentFactory {

    @Override

    public Document createDocument(){

        return new WordDocument();

    }

}

PdfDocumentFactory.java

package FactoryMethodPatternExample;

public class PdfDocumentFactory extends DocumentFactory{

    @Override

    public Document createDocument(){

        return new PdfDocument();

    }

}

Main.java

package FactoryMethodPatternExample;

public class Main {

    public static void main(String args[]){

        DocumentFactory wordFactory=new WordDocumentFactory();

        Document wordDoc=wordFactory.createDocument();

        wordDoc.open();

        DocumentFactory pdfFactory=new PdfDocumentFactory();

        Document pdfDoc = pdfFactory.createDocument();

        pdfDoc.open();

        DocumentFactory excelFactory=new ExcelDocumentFactory();

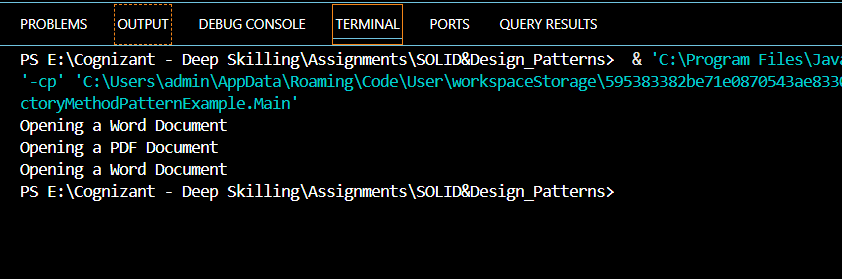
        Document excelDoc=excelFactory.createDocument();

        excelDoc.open();

    }

}

Output



# 3. Implementing a Builder Pattern

Computer.java

package BuilderPatternExample;

public class Computer {

    private final String cpu;

    private final String ram;

    private final String storage;

    private final String graphicscard;

    private final String wificard;

    private Computer(Builder builder){

        this.cpu=builder.cpu;

        this.ram=builder.ram;

        this.storage=builder.storage;

        this.graphicscard=builder.graphicscard;

        this.wificard=builder.wificard;

    }

    public String getCPU(){

        return cpu;

    }

    public String getRAM(){

        return ram;

    }

    public String getStorage(){

        return storage;

    }

    public String getGraphicsCard(){

        return graphicscard;

    }

    public String getwifiCard(){

        return wificard;

    }

    public void displayConfig(){

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

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

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

        System.out.println("Graphics Card : "+graphicscard);

        System.out.println("WifiCard : "+wificard);

    }

    public static class Builder{

        private final String cpu;

        private final String ram;

        private String storage;

        private String graphicscard;

        private String wificard;

        public Builder (String cpu,String ram){

            this.cpu=cpu;

            this.ram=ram;

        }

        public Builder setStorage(String storage){

            this.storage=storage;

            return this;

        }

        public Builder setgrapgicsCard(String graphicscard){

            this.graphicscard=graphicscard;

            return this;

        }

        public Builder setwifiCard(String wificard){

            this.wificard=wificard;

            return this;

        }

        public Computer build(){

            return new Computer(this);

        }

    }

}

Main.java

package BuilderPatternExample;

public class Main {

    public static void main(String args[]){

        Computer basiComputer=new Computer.Builder("Intel i3", "8GB").build();

        basiComputer.displayConfig();

        System.out.println("=================================");

        Computer gamingComputer = new Computer.Builder("Intel i9", "32GB")

                .setStorage("1TB SSD")

                .setgrapgicsCard("NVIDIA RTX 4080")

                .setwifiCard("Intel Wi-Fi 6")

                .build();

        gamingComputer.displayConfig();

         System.out.println("=================================");

        Computer officeComputer=new Computer.Builder("AMD Ryzen 5", "16GB")

                .setStorage("512GB SSD")

                .setwifiCard("Realtek Wi-Fi")

                .build();

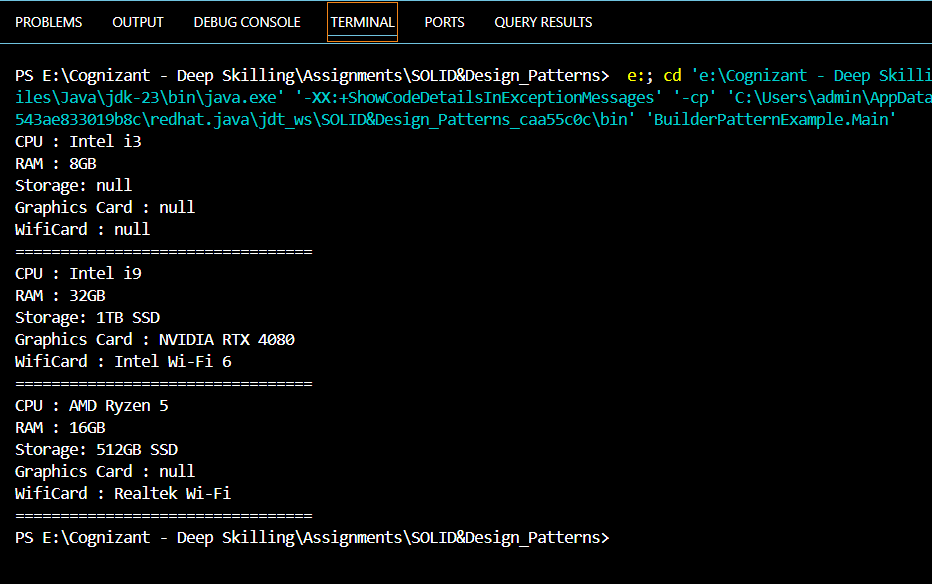
        officeComputer.displayConfig();

         System.out.println("=================================");

    }

}

Output:



# 4. Implementing the Adapter Pattern

PaymentProcessor.java

package AdapterPatternExample;

public interface PaymentProcessor {

    void processPayment(double count);

}

PayPal Adapter.java

package AdapterPatternExample;

public class PayPalAdapter implements PaymentProcessor {

    private PayPalGateway paypal;

    public PayPalAdapter(PayPalGateway paypal){

        this.paypal=paypal;

    }

    @Override

    public void processPayment(double amount){

        paypal.sendPaypalGateway(amount);

    }

}

StripeAdapter.java

package AdapterPatternExample;

public class StripeAdapter implements PaymentProcessor{

    private StripeGateway stripe;

    public StripeAdapter(StripeGateway stripe){

        this.stripe=stripe;

    }

    @Override

    public void processPayment(double amount){

        stripe.makeStripePayment(amount);

    }

}

PayPalGateway.java

package AdapterPatternExample;

public class PayPalGateway {

    public void sendPaypalGateway(double amount){

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

    }

}

StripeGateway.java

package AdapterPatternExample;

public class StripeGateway {

    public void makeStripePayment(double amount){

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

    }

}

Main.java

package AdapterPatternExample;

public class Main {

    public static void main(String args[]){

        StripeGateway stripeGateway=new StripeGateway();

        PaymentProcessor stripProcessor=new StripeAdapter(stripeGateway);

        stripProcessor.processPayment(2500.00);

        PayPalGateway payPalGateway = new PayPalGateway();

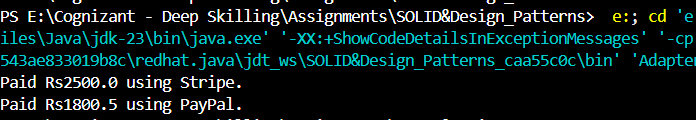
        PaymentProcessor paypalProcessor = new PayPalAdapter(payPalGateway);

        paypalProcessor.processPayment(1800.50);

    }

}

Output:



# 5. Implementing the Decorator Pattern

Notifier.java

package DecoratorPatternExample;

public interface Notifier {

    void send(String message);

}

EmailNotifier.java

package DecoratorPatternExample;

public class EmailNotifier implements Notifier{

    @Override

    public void send(String message){

        System.out.println("Send "+message);

    }

}

NotifierDecorator.java

package DecoratorPatternExample;

public abstract class NotifierDecorator implements Notifier{

    protected Notifier wrapperNotifier;

    public NotifierDecorator(Notifier notifier){

        this.wrapperNotifier=notifier;

    }

    @Override

    public void send(String message){

        wrapperNotifier.send(message);

    }

}

SMSNotifierDecorator.java

package DecoratorPatternExample;

public class SMSNotifierDecorator extends NotifierDecorator{

    public SMSNotifierDecorator(Notifier notifier){

        super(notifier);

    }

    @Override

    public void send(String message){

        super.send(message);

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

    }

}

SlackNotifierDecorator.java

package DecoratorPatternExample;

public class SlackNotifierDecorator extends NotifierDecorator{

    public SlackNotifierDecorator(Notifier notifier){

        super(notifier);

    }

    @Override

    public void send(String message){

        super.send(message);

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

    }

}

Main.java

package DecoratorPatternExample;

public class Main {

    public static void main(String[] args) {

        Notifier baseNotifier=new EmailNotifier();

        Notifier emailAndSms=new SMSNotifierDecorator(baseNotifier);

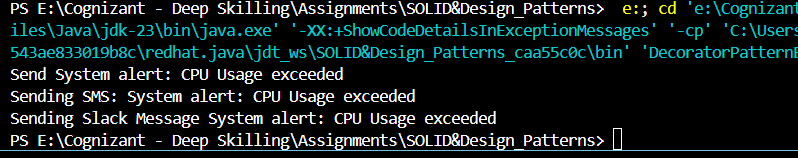
        Notifier fullNotifier=new SlackNotifierDecorator(emailAndSms);

        fullNotifier.send("System alert: CPU Usage exceeded");

    }

}

Output



# 6. Implementing Proxy Pattern

Image.java

package ProxyPatternExample;

public interface Image {

    void display();

}

RealImage.java

package ProxyPatternExample;

public class RealImage implements Image {

    private String filename;

    public RealImage(String filename){

        this.filename=filename;

        loadFromRemoteServer(filename);

    }

    public void loadFromRemoteServer(String filename){

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

    }

    @Override

    public void display(){

        System.out.println("Displaying Image: "+filename);

    }

}

ProxyImage.java

package ProxyPatternExample;

public class ProxyImage implements Image {

    private RealImage realImage;

    private String filename;

    public ProxyImage(String filename){

        this.filename=filename;

    }

    @Override

    public void display(){

        if(realImage==null){

            realImage=new RealImage(filename);

        }

        else{

            System.out.println("Using cached Image: "+filename);

        }

        realImage.display();

    }

}

Main.java

package ProxyPatternExample;

public class Main {

    public static void main(String args[]){

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

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

    image1.display();

    image1.display();

    System.out.println("==========================");

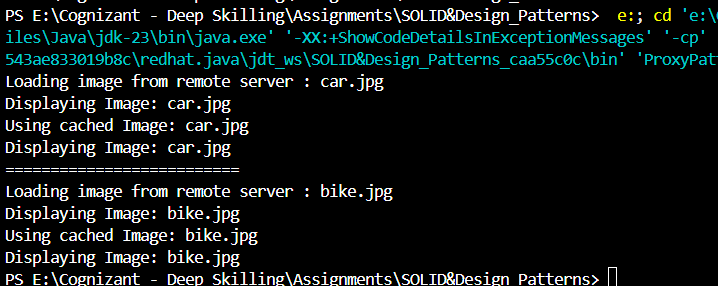
    image2.display();

    image2.display();

    }

}

Output:



# 7. Implementing the Observer Pattern

Stock.java

package ObserverPatternExample;

public interface Stock {

    void registerObserver(Observer o);

    void removeObserver(Observer o);

    void notifyObserver();

}

StockMarket.java

package ObserverPatternExample;

import java.util.ArrayList;

import java.util.List;

public class StockMarket implements Stock {

    private List<Observer> observers=new ArrayList<>();

    private double stockPrice;

    @Override

    public void registerObserver(Observer o){

        observers.add(o);

    }

    @Override

    public void removeObserver(Observer o){

        observers.remove(o);

    }

    @Override

    public void notifyObserver(){

        for(Observer o:observers){

            o.update(stockPrice);

        }

    }

    public void setStockPrice(double price) {

        this.stockPrice = price;

        System.out.println("\nStock price updated to: Rs " + price);

        notifyObserver();

    }

}

Observer.java

package ObserverPatternExample;

public interface Observer {

    void update(double stockPrice);

}

MobileApp.java

package ObserverPatternExample;

public class MobileApp implements Observer{

    private String user;

    public MobileApp(String user){

        this.user=user;

    }

    @Override

    public void update(double stockPrice){

        System.out.println("Mobile App [" + user + "] - New stock price: Rs " + stockPrice);

    }

}

WebApp.java

package ObserverPatternExample;

public class WebApp implements Observer{

    private String user;

    public WebApp(String user){

      this.user=user;

    }

    @Override

    public void update(double stockPrice){

        System.out.println("Web App [" + user + "] - New stock price: Rs " + stockPrice);

    }

}

Main.java

package ObserverPatternExample;

public class Main {

    public static void main(String args[]){

        StockMarket market=new StockMarket();

        Observer mobileUser=new MobileApp("Ayshu");

        Observer webUser=new WebApp("Dhivakar");

        market.registerObserver(mobileUser);

        market.registerObserver(webUser);

        market.setStockPrice(1025.75);

        market.setStockPrice(1043.20);

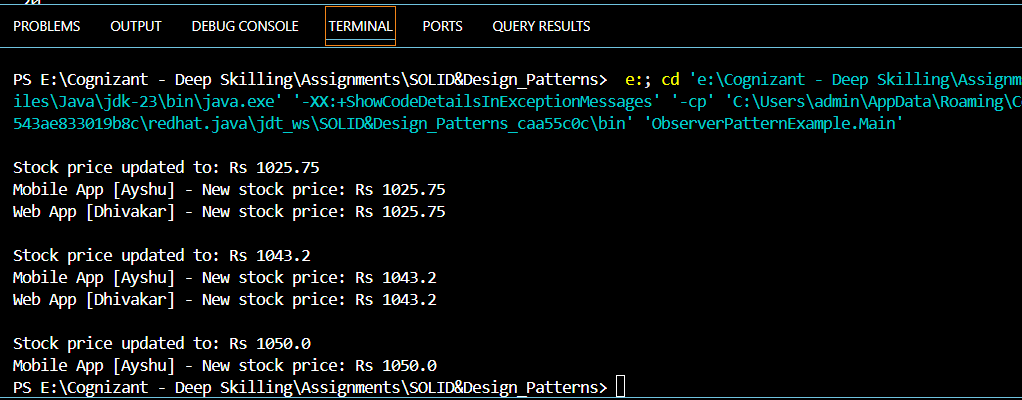
        market.removeObserver(webUser);

        market.setStockPrice(1050.00);

    }

}

**Output:**

****

# 8. Implementing Strategy Pattern

PaymentStrategy.java

package StrategyPatternExample;

public interface PaymentStrategy {

    void pay(double amount);

}

PaymentContext.java

package StrategyPatternExample;

public class PaymentContext {

    private PaymentStrategy paymentStrategy;

    public void setPaymentStrategy(PaymentStrategy strategy){

        this.paymentStrategy=strategy;

    }

    public void processPayment(double amount){

        if(paymentStrategy==null){

            System.out.println("No payment method selected");

        }

        else{

            paymentStrategy.pay(amount);

        }

    }

}

PayPalPayment.java

package StrategyPatternExample;

public class PayPalPayment implements PaymentStrategy{

    private String email;

    public PayPalPayment(String email){

        this.email=email;

    }

    @Override

    public void pay(double amount){

        System.out.println("Paid Rs " + amount + " using PayPal account [" + email + "]");

    }

}

CreditCardPayment.java

package StrategyPatternExample;

public class CreditCardPayment implements PaymentStrategy{

    @SuppressWarnings("unused")

    private String cardNumber;

    private String cardHolder;

    public CreditCardPayment(String cardNumber, String cardHolder) {

        this.cardNumber=cardNumber;

        this.cardHolder=cardHolder;

    }

    @Override

    public void pay(double amount) {

        System.out.println("Paid Rs " + amount + " using Credit Card [" + cardHolder + "]");

    }

}

Main.java

package StrategyPatternExample;

public class Main {

    public static void main(String[] args) {

        PaymentContext context = new PaymentContext();

        context.setPaymentStrategy(new CreditCardPayment("1234-5678-9012-3456", "Ayshu"));

        context.processPayment(2500.0);

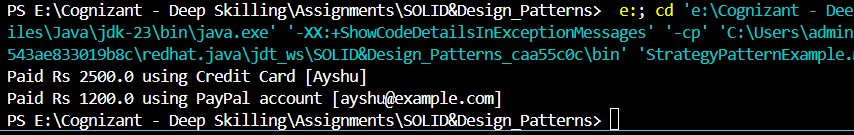
        context.setPaymentStrategy(new PayPalPayment("ayshu@example.com"));

        context.processPayment(1200.0);

    }

}

Output:



# 9. Implementing the Command Pattern

Command.java

package CommandPatternExample;

public interface Command {

    void execute();

}

Light.java

package CommandPatternExample;

public class Light {

    public void turnOn(){

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

    }

    public void turnOff(){

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

    }

}

LightOffCommand.java

package CommandPatternExample;

public class LightOffCommand implements Command{

    private Light light;

    public LightOffCommand(Light light){

        this.light=light;

    }

    @Override

    public void execute(){

        light.turnOff();

    }

}

**LightOnCommand.java**

package CommandPatternExample;

public class LightOnCommand implements Command{

    private Light light;

    LightOnCommand(Light light){

        this.light=light;

    }

    @Override

    public void execute(){

        light.turnOn();

    }

}

**RemoteControl.java**

package CommandPatternExample;

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 assigned to the button");

        }

    }

}

**Main.java**

package CommandPatternExample;

public class Main {

    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();

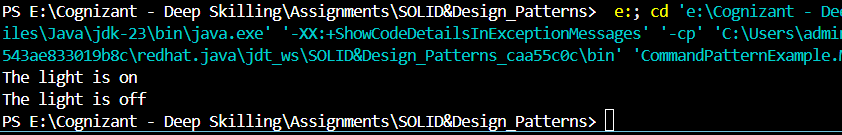
        remote.setCommand(lightOff);

        remote.pressButton();

    }

}

**Output:**

****

# 10. Implementing the MVC Pattern

Student.java

package MVCPatternExample;

public class Student {

    private String name;

    private String id;

    private String grade;

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

        return id;

    }

    public void setId(String id){

        this.id=id;

    }

    public String getGrade(){

        return grade;

    }

    public void setGrade(String grade){

        this.grade=grade;

    }

}

StudentController.java

package MVCPatternExample;

public class StudentController {

    private Student model;

    private StudentView view;

    public StudentController(Student model, StudentView view){

        this.model=model;

        this.view=view;

    }

    public void updateView(){

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

    }

     public void setStudentName(String name) {

        model.setName(name);

    }

    public void setStudentId(String id) {

        model.setId(id);

    }

    public void setStudentGrade(String grade) {

        model.setGrade(grade);

    }

}

StudentView.java

package MVCPatternExample;

public class StudentView {

    public void displayStudentDetails(String name, String id, String grade){

        System.out.println("Student Name: "+name);

        System.out.println("Student ID: "+id);

        System.out.println("Student Grade : "+grade);

        System.out.println("===========================");

    }

}

Main.java

package MVCPatternExample;

public class Main {

    public static void main(String[] args) {

        Student student = new Student("Ayshu", "S102", "A");

        StudentView view = new StudentView();

        StudentController controller = new StudentController(student, view);

        controller.updateView();

        controller.setStudentName("Abi Ayshwariya");

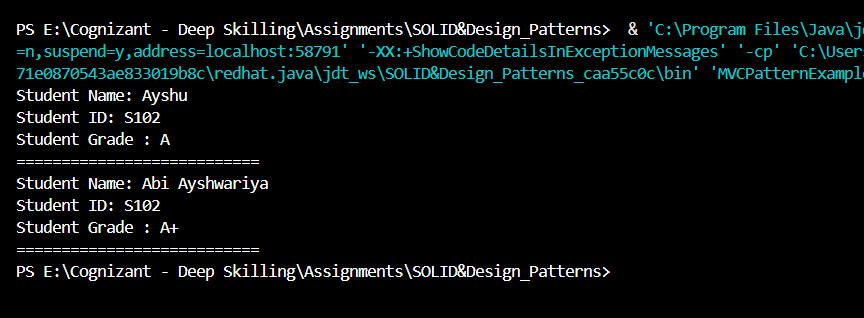
        controller.setStudentGrade("A+");

        controller.updateView();

    }

}

Output:



# 11. Implementing Dependency Injection

CustomerRepository.java

package DependencyInjectionExample;

public interface CustomerRepository {

    String findCustomerById(String customerID);

}

CustomerRepositoryImpl.java

package DependencyInjectionExample;

public class CustomerRepositoryImpl implements CustomerRepository {

    @Override

    public String findCustomerById(String customerId) {

        return "Customer [ID: " + customerId + ", Name: Ayshu]";

    }

}

CustomerService.java

package DependencyInjectionExample;

public class CustomerService {

    private CustomerRepository repository;

    public CustomerService(CustomerRepository repository){

        this.repository=repository;

    }

    public void displayCustomerDetails(String customerId) {

        String customer = repository.findCustomerById(customerId);

        System.out.println(customer);

    }

}

Main.java

package DependencyInjectionExample;

public class Main {

    public static void main(String[] args) {

        CustomerRepository repository = new CustomerRepositoryImpl();

        CustomerService service = new CustomerService(repository);

        service.displayCustomerDetails("C102");

        service.displayCustomerDetails("A105");

    }

}

Output:

