**Exercise 1: Implementing the Singleton Pattern (Mandatory)**

class Logger {

private static Logger ins;

private Logger() {

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

}

public static Logger getInstance() {

if (ins == null) {

ins = new Logger();

}

return ins;

}

public void log(String s) {

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

}

}

public class SingletonPatternExample {

public static void main(String[] args) {

Logger l1 = Logger.getInstance();

l1.log("First Message");

Logger l2 = Logger.getInstance();

l2.log("Second Message");

if (l1 == l2) {

System.out.println("Same Instance");

} else {

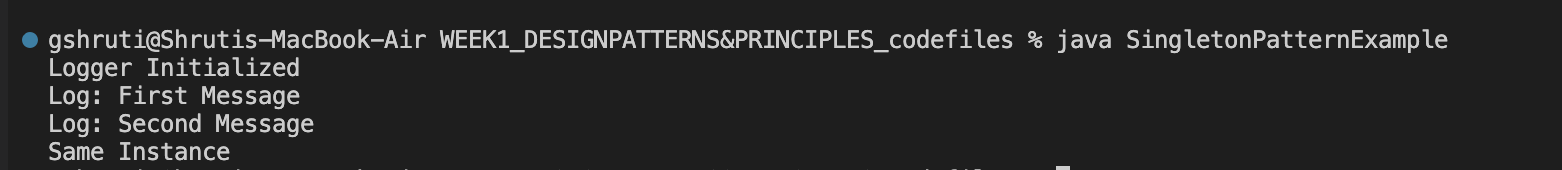
System.out.println("Not Same Instance");

}

}

}

Output:



**Exercise 2: Implementing the Factory Method Pattern (Mandatory)**

interface Document {

void open();

}

class WordDoc implements Document {

public void open() {

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

}

}

class PdfDoc implements Document {

public void open() {

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

}

}

class ExcelDoc implements Document {

public void open() {

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

}

}

abstract class DocumentFactory {

public abstract Document createDocument();

}

class WordFactory extends DocumentFactory {

public Document createDocument() {

return new WordDoc();

}

}

class PdfFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDoc();

}

}

class ExcelFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDoc();

}

}

public class FactoryMethodPatternExample {

public static void main(String[] args) {

DocumentFactory wordFactory = new WordFactory();

Document wordDoc = wordFactory.createDocument();

wordDoc.open();

DocumentFactory pdfFactory = new PdfFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open();

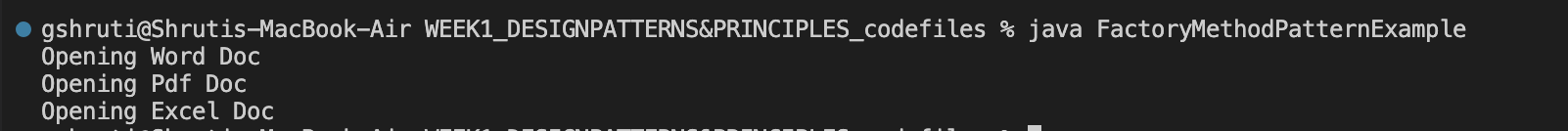
DocumentFactory excelFactory = new ExcelFactory();

Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

Output:   


**Exercise 3: Implementing the Builder Pattern**

public class BuilderPatternExample {

static class Computer {

private String cpu;

private String ram;

private String storage;

private String gpu;

private Computer(Builder builder) {

this.cpu = builder.cpu;

this.ram = builder.ram;

this.storage = builder.storage;

this.gpu = builder.gpu;

}

public void displayConfig() {

System.out.println("Computer Config:");

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

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

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

System.out.println("GPU: " + gpu);

System.out.println();

}

public static class Builder {

private String cpu;

private String ram;

private String storage;

private String gpu;

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 Computer build() {

return new Computer(this);

}

}

}

public static void main(String[] args) {

Computer basicPC = new Computer.Builder()

.setCpu("Intel i3")

.setRam("8GB")

.setStorage("256GB SSD")

.build();

Computer gamingPC = new Computer.Builder()

.setCpu("Intel i9")

.setRam("32GB")

.setStorage("1TB SSD")

.setGpu("NVIDIA RTX 4090")

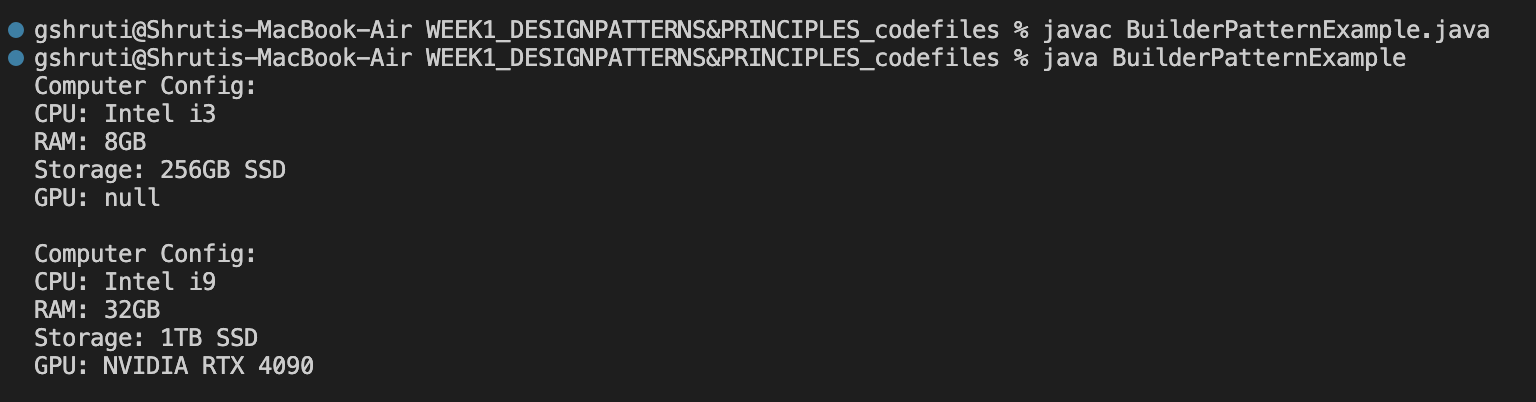
.build();

basicPC.displayConfig();

gamingPC.displayConfig();

}

}

Output:   


**Exercise 4: Implementing the Adapter Pattern**

interface PaymentProcessor {

void processPayment(double amt);

}

class GatewayA {

void printTransaction(double amt) {

System.out.println("Processing ₹" + amt + " through Gateway A");

}

}

class GatewayB {

void printpay(double amt) {

System.out.println("Processing ₹" + amt + " through Gateway B");

}

}

class GatewayAAdapter implements PaymentProcessor {

private GatewayA a = new GatewayA();

public void processPayment(double amt) {

ga.printTransaction(amt);

}

}

class GatewayBAdapter implements PaymentProcessor {

private GatewayB gb = new GatewayB();

public void processPayment(double amt) {

gb.printpay(amt);

}

}

public class AdapterPatternExample {

public static void main(String[] args) {

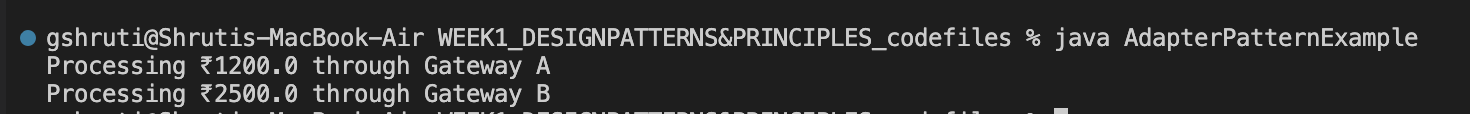
PaymentProcessor processorA = new GatewayAAdapter();

processorA.processPayment(1200.00);

PaymentProcessor processorB = new GatewayBAdapter();

processorB.processPayment(2500.00);

}

Output:   


**Exercise 5: Implementing the Decorator Pattern**

interface Notifier {

void send(String s);

}

class EmailNotifier implements Notifier {

public void send(String s) {

System.out.println("Email: " + s);

}

}

abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

public NotifierDecorator(Notifier n) {

this.notifier = n;

}

public void send(String s) {

notifier.send(s);

}

}

class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier n) {

super(n);

}

public void send(String s) {

super.send(s);

System.out.println("SMS: " + s);

}

}

class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier n) {

super(n);

}

public void send(String s) {

super.send(s);

System.out.println("Slack: " + s);

}

}

public class DecoratorPatternExample {

public static void main(String[] args) {

Notifier email = new EmailNotifier();

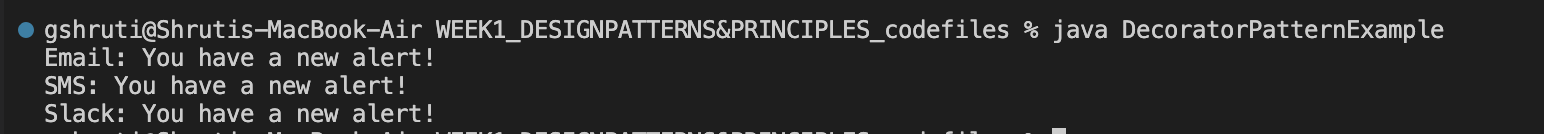
Notifier sae = new SMSNotifierDecorator(email);

Notifier ac = new SlackNotifierDecorator(sae);

ac.send("You have a new alert!");

}

}

Output:   


**Exercise 6: Implementing the Proxy Pattern**

interface Image {

void display();

}

class RealImage implements Image {

private String s;

public RealImage(String s) {

this.s = s;

loadImageFromServer();

}

private void loadImageFromServer() {

System.out.println("Loading " + s + " from remote server...");

}

public void display() {

System.out.println("Displaying " + s);

}

}

class ProxyImage implements Image {

private RealImage realImage;

private String s;

public ProxyImage(String s) {

this.s = s;

}

public void display() {

if (realImage == null) {

realImage = new RealImage(s);

}

realImage.display();

}

}

public class ProxyPatternExample {

public static void main(String[] args) {

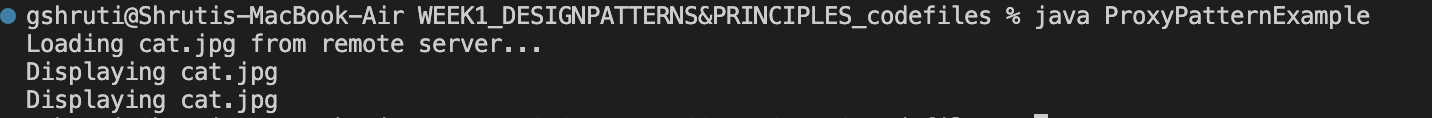
Image image1 = new ProxyImage("cat.jpg");

image1.display();

image1.display();

}

}

Output:   


**Exercise 7: Implementing the Observer Pattern**

import java.util.\*;

interface Observer {

void update(String s, double p);

}

interface Stock {

void register(Observer o);

void deregister(Observer o);

void notifyObservers();

}

class StockMarket implements Stock {

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

private String s;

private double p;

public void setStockPrice(String s, double p) {

this.s = s;

this.p = p;

notifyObservers();

}

public void register(Observer o) {

obs.add(o);

}

public void deregister(Observer o) {

obs.remove(o);

}

public void notifyObservers() {

for (Observer o : obs) {

o.update(s, p);

}

}

}

class MobileApp implements Observer {

public void update(String s, double p) {

System.out.println("Mobile App - " + s + ": ₹" + p);

}

}

class WebApp implements Observer {

public void update(String s, double p) {

System.out.println("Web App - " + s + ": ₹" + p);

}

}

public class ObserverPatternExample {

public static void main(String[] args) {

StockMarket m = new StockMarket();

Observer mob = new MobileApp();

Observer web = new WebApp();

m.register(mob);

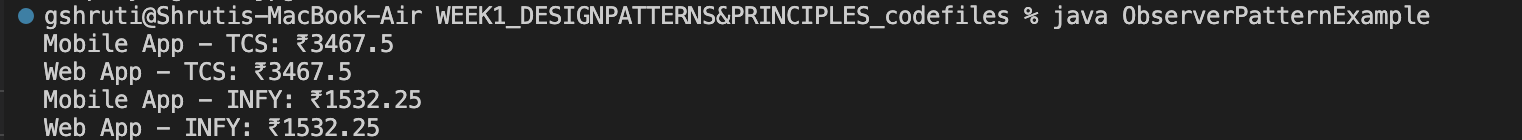
m.register(web);

m.setStockPrice("TCS", 3467.50);

m.setStockPrice("INFY", 1532.25);

}

}

Output:   


**Exercise 8: Implementing the Strategy Pattern**

interface PaymentStrategy {

void pay(int amt);

}

class CreditCardPayment implements PaymentStrategy {

public void pay(int amt) {

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

}

}

class UpiPayment implements PaymentStrategy {

public void pay(int amt) {

System.out.println("Paid ₹" + amt + " using UPI");

}

}

class PaymentContext {

private PaymentStrategy ps;

public PaymentContext(PaymentStrategy ps) {

this.ps = ps;

}

public void makePayment(int amt) {

ps.pay(amt);

}

}

public class StrategyPatternExample {

public static void main(String[] args) {

PaymentStrategy ccd = new CreditCardPayment();

PaymentContext c1 = new PaymentContext(ccd);

c1.makePayment(500);

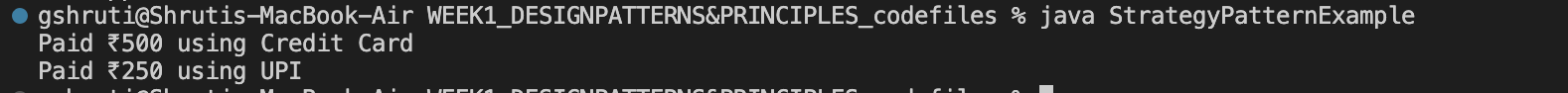
PaymentStrategy upi = new UpiPayment();

PaymentContext c2 = new PaymentContext(upi);

c2.makePayment(250);

}

}

Output:   


**Exercise 9: Implementing the Command Pattern**

interface Command {

void execute();

}

class Light {

public void on() {

System.out.println("Light is ON");

}

public void off() {

System.out.println("Light is OFF");

}

}

class LightOnCommand implements Command {

Light lgt;

LightOnCommand(Light lgt) {

this.lgt = lgt;

}

public void execute() {

lgt.on();

}

}

class LightOffCommand implements Command {

Light lgt;

LightOffCommand(Light lgt) {

this.lgt = lgt;

}

public void execute() {

lgt.off();

}

}

class RemoteControl {

Command cmd;

void setCommand(Command cmd) {

this.cmd = cmd;

}

void pressButton() {

cmd.execute();

}

}

public class CommandPatternExample {

public static void main(String[] args) {

Light lgt = new Light();

Command onCmd = new LightOnCommand(lgt);

Command offCmd = new LightOffCommand(lgt);

RemoteControl rmt = new RemoteControl();

rmt.setCommand(onCmd);

rmt.pressButton();

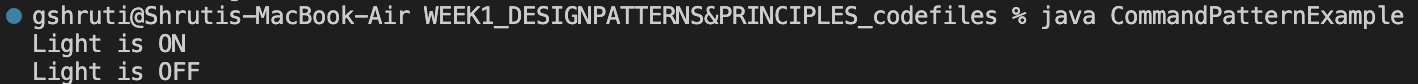
rmt.setCommand(offCmd);

rmt.pressButton();

}

}

Output :



**Exercise 10: Implementing the MVC Pattern**

class Student {

private String name;

private String id;

private String 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;

}

}

class StudentView {

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

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

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

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

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

}

}

class StudentController {

private Student std;

private StudentView vw;

public StudentController(Student std, StudentView vw) {

this.std = std;

this.vw = vw;

}

public void setStudentName(String name) {

std.setName(name);

}

public String getStudentName() {

return std.getName();

}

public void setStudentId(String id) {

std.setId(id);

}

public String getStudentId() {

return std.getId();

}

public void setStudentGrade(String grade) {

std.setGrade(grade);

}

public String getStudentGrade() {

return std.getGrade();

}

public void updateView() {

vw.displayStudentDetails(std.getName(), std.getId(), std.getGrade());

}

}

public class MVCPatternExample {

public static void main(String[] args) {

Student std = new Student();

std.setName("Anu");

std.setId("S101");

std.setGrade("A");

StudentView vw = new StudentView();

StudentController ctr = new StudentController(std, vw);

ctr.updateView();

ctr.setStudentName("Rahul");

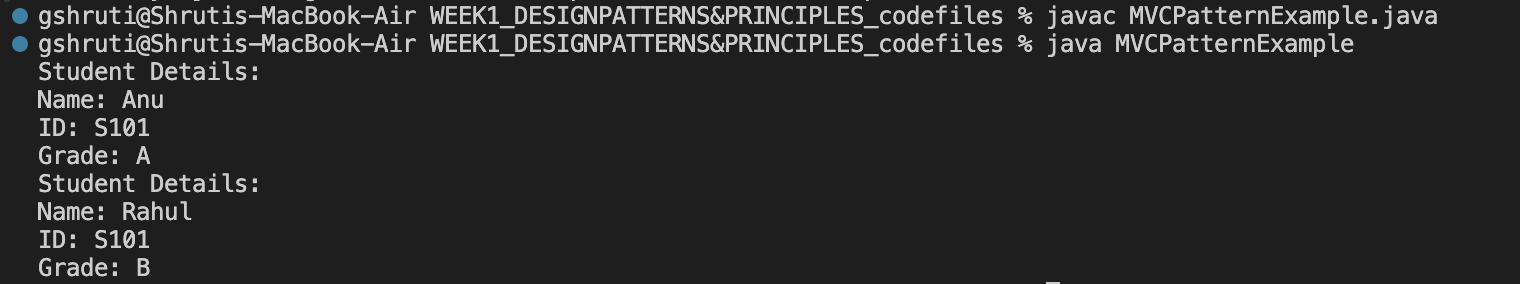
ctr.setStudentGrade("B");

ctr.updateView();

}

}

Output:



**Exercise 11: Implementing Dependency Injection**

interface CustomerRepository {

String findCustomerById(String id);

}

class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(String id) {

return "Customer Name for ID " + id + ": Priya";

}

}

class CustomerService {

private CustomerRepository rep;

public CustomerService(CustomerRepository rep) {

this.rep = rep;

}

public void displayCustomer(String id) {

String res = rep.findCustomerById(id);

System.out.println(res);

}

}

public class DependencyInjectionExample {

public static void main(String[] args) {

CustomerRepository rep = new CustomerRepositoryImpl();

CustomerService ser = new CustomerService(rep);

ser.displayCustomer("C101");

}

}

Output:  
