**Exercise 1: Implementing the Singleton Pattern**

**Program:**

public class SingletonPatternExample {

static class Logger {

private static Logger instance;

private Logger() {

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

}

public static Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

public void log(String message) {

System.out.println("[LOG]: " + message);

}

}

static class Invoice {

private String item;

private int quantity;

private int one;

public Invoice(String item, int quantity, int one) {

this.item = item;

this.quantity = quantity;

this.one = one;

}

public String getItem() {

return item;

}

public int getQuantity() {

return quantity;

}

public int getOne() {

return one;

}

public int getPrice() {

return quantity \* one;

}

}

static class InvoicePrinter {

public void printReport(Invoice inv) {

System.out.println("BILL:");

System.out.println("Item: " + inv.getItem());

System.out.println("Quantity: " + inv.getQuantity());

System.out.println("Price per item: Rs." + inv.getOne());

System.out.println("Total: ₹" + inv.getPrice());

Logger logger = Logger.getInstance();

logger.log("Invoice printed for item: " + inv.getItem());

}

}

public static void main(String[] args) {

Invoice invoice = new Invoice("Pen", 5, 10);

InvoicePrinter printer = new InvoicePrinter();

printer.printReport(invoice);

Logger logger1 = Logger.getInstance();

Logger logger2 = Logger.getInstance();

if (logger1 == logger2) {

System.out.println("Same Logger instance confirmed.");

} else {

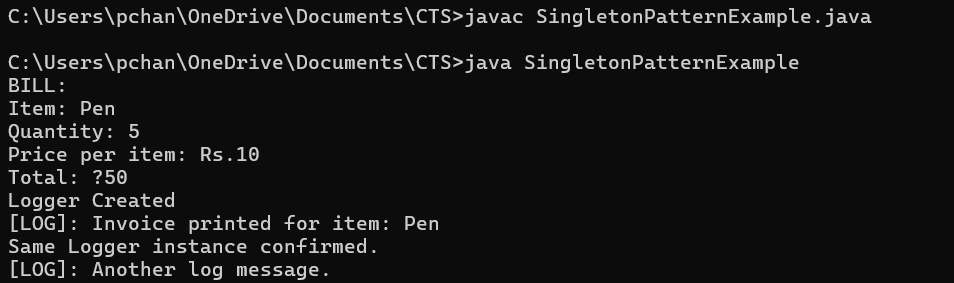
System.out.println("Different Logger instances found!");

}

logger2.log("Another log message.");

}

}



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

**Program:**

public class FactoryMethodPatternExample {

interface Document {

void create();

}

static class WordDocument implements Document {

public void create() {

System.out.println("WordDocument is created");

}

}

static class PdfDocument implements Document {

public void create() {

System.out.println("PdfDocument is created");

}

}

static class ExcelDocument implements Document {

public void create() {

System.out.println("ExcelDocument is created");

}

}

abstract static class DocumentFactory {

public abstract Document createDocument();

}

static class WordFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

static class PdfFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

static class ExcelFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

public static void main(String[] args) {

DocumentFactory wordFactory = new WordFactory();

Document word = wordFactory.createDocument();

word.create();

DocumentFactory pdfFactory = new PdfFactory();

Document pdf = pdfFactory.createDocument();

pdf.create();

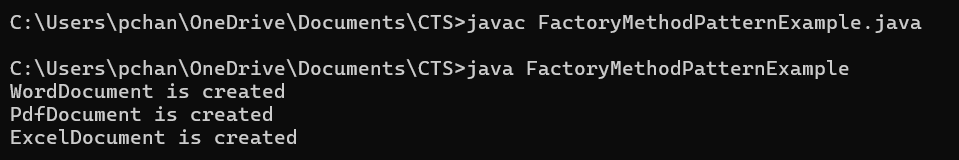
DocumentFactory excelFactory = new ExcelFactory();

Document excel = excelFactory.createDocument();

excel.create();

}

}



**Exercise 3: Implementing the Builder Pattern**

**Program:**

class Computer{

private String CPU;

private String RAM;

private String storage;

private Computer(ComputerBuilder builder){

this.CPU = builder.CPU;

this.RAM = builder.RAM;

this.storage = builder.storage;

}

public void ComputerDetails(){

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

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

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

}

public static class ComputerBuilder{

private String CPU;

private String RAM;

private String storage;

public ComputerBuilder setCPU(String CPU){

this.CPU=CPU;

return this;

}

public ComputerBuilder setRAM(String RAM){

this.RAM=RAM;

return this;

}

public ComputerBuilder setStorage(String storage){

this.storage=storage;

return this;

}

public Computer build(){

return new Computer(this);

}}}

public class BuilderPatternExample{

public static void main(String[] args){

Computer c = new Computer.ComputerBuilder()

.setCPU("intel i5")

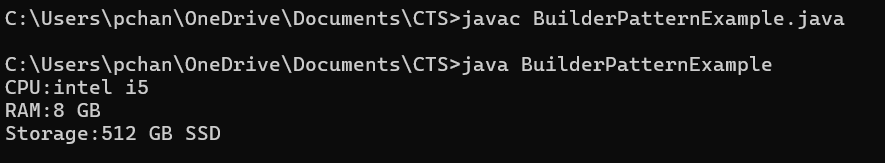
.setRAM("8 GB")

.setStorage("512 GB SSD")

.build();

c.ComputerDetails();

}}

****

**Exercise 4: Implementing the Adapter Pattern**

**Program:**

interface PaymentProcessor{

void processPayment(double amount);

}

class PayPalGateway{

public void sendPay(double amount){

System.out.println("Amount of Rs."+amount+" is processed through paypal gateway");

}}

class StripeGateway{

public void doPay(double amount){

System.out.println("Amount of Rs."+amount+" is processed through stripe gateway");

}}

class RazorPayGateway{

public void tryPay(double amount){

System.out.println("Amount of Rs."+amount+" is processed through Razor gateway");

}}

class PayPalAdapter implements PaymentProcessor{

private PayPalGateway paypal = new PayPalGateway();

public void processPayment(double amount){

paypal.sendPay(amount);

}}

class StripeAdapter implements PaymentProcessor{

private StripeGateway stripe = new StripeGateway();

public void processPayment(double amount){

stripe.doPay(amount);

}}

class RazorAdapter implements PaymentProcessor{

private RazorPayGateway razor = new RazorPayGateway();

public void processPayment(double amount){

razor.tryPay(amount);

}}

public class AdapterPatternExample{

public static void main(String[] args){

PaymentProcessor paypalprocessor = new PayPalAdapter();

paypalprocessor.processPayment(1500.00);

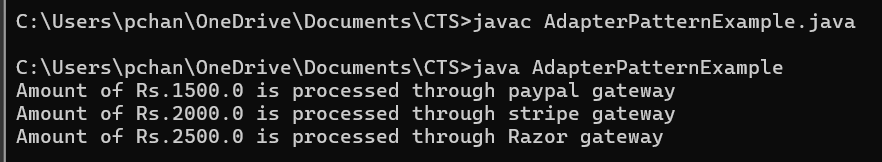
PaymentProcessor stripeprocessor = new StripeAdapter();

stripeprocessor.processPayment(2000.00);

PaymentProcessor razorprocessor = new RazorAdapter();

razorprocessor.processPayment(2500.00);

}}



**Exercise 5: Implementing the Decorator Pattern**

**Program:**

interface Notifier{

void send(String msg);

}

class EmailNotifier implements Notifier{

public void send(String message){

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

}}

abstract class NotifierDecorator implements Notifier{

protected Notifier notifier;

public NotifierDecorator(Notifier notifier){

this.notifier = notifier;

}

public void send(String message){

notifier.send(message);

}}

class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier){

super(notifier);

}

public void send(String message){

super.send(message);

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

}}

class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier){

super(notifier);

}

public void send(String message){

super.send(message);

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

}}

public class DecoratorPatternExample{

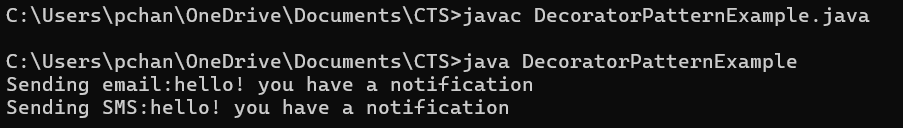
public static void main(String[] args){

Notifier notifier = new EmailNotifier();

notifier = new SMSNotifierDecorator(notifier);

notifier.send("hello! you have a notification");

}}



**Exercise 6: Implementing the Proxy Pattern**

**Program:**

public class ProxyPatternExample {

interface Image {

void display();

}

static class RealImage implements Image {

private String fileName;

public RealImage(String fileName) {

this.fileName = fileName;

loadFromRemoteServer();

}

private void loadFromRemoteServer() {

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

}

public void display() {

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

}

}

static class ProxyImage implements Image {

private RealImage realImage;

private String fileName;

public ProxyImage(String fileName) {

this.fileName = fileName;

}

public void display() {

if (realImage == null) {

realImage = new RealImage(fileName);

}

realImage.display();

}

}

public static void main(String[] args) {

Image image1 = new ProxyImage("scenery1.jpg");

Image image2 = new ProxyImage("scenery2.jpg");

image1.display();

System.out.println();

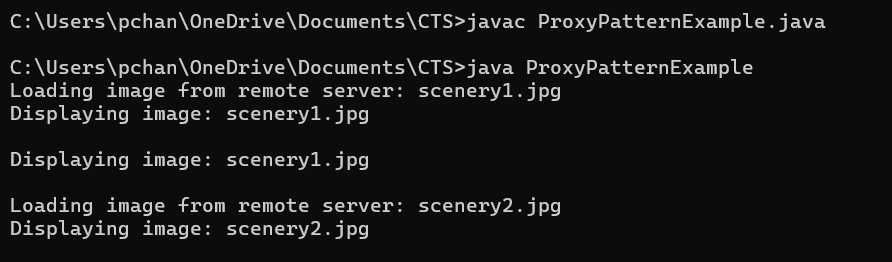
image1.display();

System.out.println();

image2.display();

}

}

****

**Exercise 7: Implementing the Observer Pattern**

**Program:**

import java.util.\*;

interface Stock {

void register(Observer o);

void deregister(Observer o);

void notifyObservers();

}

interface Observer {

void update(int StockPrice);

}

class StockMarket implements Stock {

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

private int StockPrice;

public void setStockPrice(int price){

this.StockPrice = price;

notifyObservers();

}

public void register(Observer o){

observers.add(o);

}

public void deregister(Observer o){

observers.remove(o);

}

public void notifyObservers(){

for(Observer o:observers){

o.update(StockPrice);

}}}

class MobileApp implements Observer {

public void update(int StockPrice){

System.out.println("MobileApp : Current stock price is :"+StockPrice);

}}

class WebApp implements Observer {

public void update(int StockPrice){

System.out.println("WebApp : Current stock price is :"+StockPrice);

}}

public class ObserverPatternExample{

public static void main(String[] args){

StockMarket market = new StockMarket();

Observer mobile = new MobileApp();

Observer web = new WebApp();

market.register(mobile);

market.register(web);

System.out.println(">> Stock price changed to Rs.150");

market.setStockPrice(150);

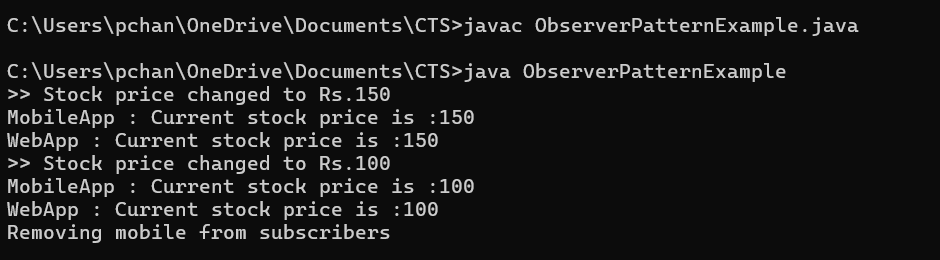
System.out.println(">> Stock price changed to Rs.100");

market.setStockPrice(100);

System.out.println("Removing mobile from subscribers");

market.deregister(mobile);

}}

****

**Exercise 8: Implementing the Strategy Pattern**

**Program:**

interface PaymentStrategy{

void pay();

}

class CreditCardPayment implements PaymentStrategy {

public void pay(){

System.out.println("Paying using credit card");

}}

class PayPalPayment implements PaymentStrategy {

public void pay(){

System.out.println("Paying using paypal");

}}

class PaymentContext{

private PaymentStrategy strategy;

public void setStrategy(PaymentStrategy strategy){

this.strategy = strategy;

}

public void startPayment(){

strategy.pay();

}}

public class StrategyPatternExample{

public static void main(String[] args){

PaymentContext context = new PaymentContext();

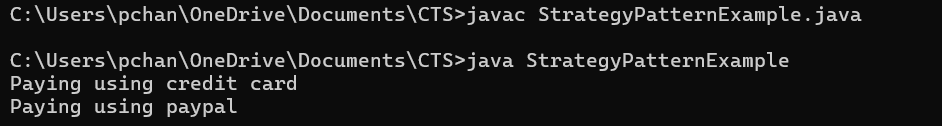
context.setStrategy(new CreditCardPayment());

context.startPayment();

context.setStrategy(new PayPalPayment());

context.startPayment();

}}



**Exercise 9: Implementing the Command Pattern**

**Program:**

interface Command {

void execute();

}

class Light {

public void turnOn() {

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

}

public void turnOff() {

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

}

}

class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOn();

}

}

class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOff();

}

}

class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

command.execute();

}

}

public class CommandPatternExample {

public static void main(String[] args) {

Light light = new Light();

Command onCommand = new LightOnCommand(light);

Command offCommand = new LightOffCommand(light);

RemoteControl remote = new RemoteControl();

remote.setCommand(onCommand);

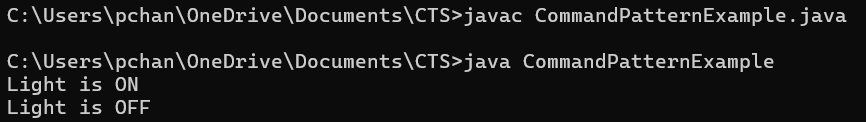
remote.pressButton();

remote.setCommand(offCommand);

remote.pressButton();

}

}



**Exercise 10: Implementing the MVC Pattern**

**Program:**

public class MVCPatternExample {

static class Student {

private String name;

private int id;

private char 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 char getGrade() {

return grade;

}

public void setGrade(char grade) {

this.grade = grade;

}

}

static class StudentView {

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

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

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

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

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

}

}

static class StudentController {

private Student model;

private StudentView view;

public StudentController(Student model, StudentView view) {

this.model = model;

this.view = view;

}

public void setStudentName(String name) {

model.setName(name);

}

public void setStudentId(int id) {

model.setId(id);

}

public void setGrade(char grade) {

model.setGrade(grade);

}

public void updateView() {

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

}

}

public static void main(String[] args) {

Student student = new Student();

student.setName("Amit");

student.setId(1);

student.setGrade('A');

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

controller.updateView();

controller.setStudentName("Raj");

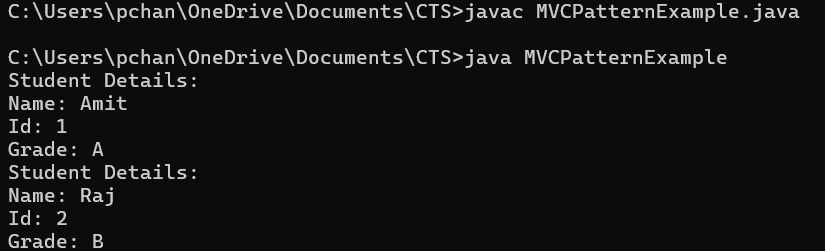
controller.setStudentId(2);

controller.setGrade('B');

controller.updateView();

}

}



**Exercise 11: Implementing Dependency Injection**

**Program:**

interface CustomerRepository {

void findCustomerById(int id);

}

class CustomerRepositoryImpl implements CustomerRepository {

public void findCustomerById(int id) {

System.out.println("Customer with ID " + id + " is found in the database.");

}

}

class CustomerService {

private CustomerRepository customerRepository;

public CustomerService(CustomerRepository customerRepository) {

this.customerRepository = customerRepository;

}

public void getCustomer(int id) {

customerRepository.findCustomerById(id);

}

}

public class DependencyInjectionExample {

public static void main(String[] args) {

CustomerRepository repository = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repository);

service.getCustomer(11);

}

}

