**Exercise 1: Implementing the Singleton Pattern**

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);

}

}

public class Main {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

logger1.log("This is the first log message.");

Logger logger2 = Logger.getInstance();

logger2.log("This is the second log message.");

if (logger1 == logger2) {

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

} else {

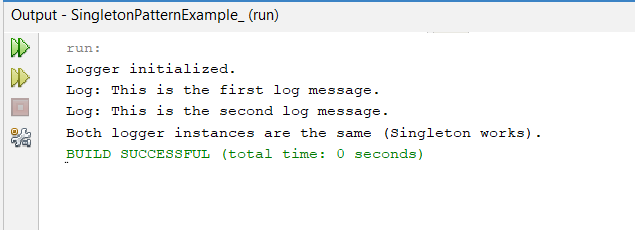
System.out.println("Different logger instances (Singleton failed).");

}

}

}

Output:



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

// Document.java

interface Document {

void open();

}

// WordDocument.java

class WordDocument implements Document {

public void open() {

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

}

}

// PdfDocument.java

class PdfDocument implements Document {

public void open() {

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

}

}

// ExcelDocument.java

class ExcelDocument implements Document {

public void open() {

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

}

}

// DocumentFactory.java

abstract class DocumentFactory {

public abstract Document createDocument();

}

// WordDocumentFactory.java

class WordDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

// PdfDocumentFactory.java

class PdfDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

// ExcelDocumentFactory.java

class ExcelDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

// FactoryMethodPatternExample.java

class FactoryMethodPatternExample {

public static void main(String[] args) {

DocumentFactory wordFactory = new WordDocumentFactory();

Document word = wordFactory.createDocument();

word.open();

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdf = pdfFactory.createDocument();

pdf.open();

DocumentFactory excelFactory = new ExcelDocumentFactory();

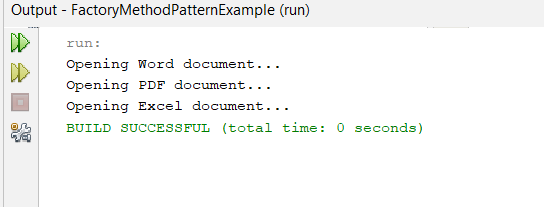
Document excel = excelFactory.createDocument();

excel.open();

}

}

Output:



**Exercise 3: Implementing the Builder Pattern**

// Computer.java

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);

}

}

@Override

public String toString() {

return "Computer [CPU=" + CPU + ", RAM=" + RAM + ", storage=" + storage + "]";

}

}

// BuilderPatternExample.java

class BuilderPatternExample {

public static void main(String[] args) {

Computer computer = new Computer.Builder()

.setCPU("Intel i7")

.setRAM("16GB")

.setStorage("512GB SSD")

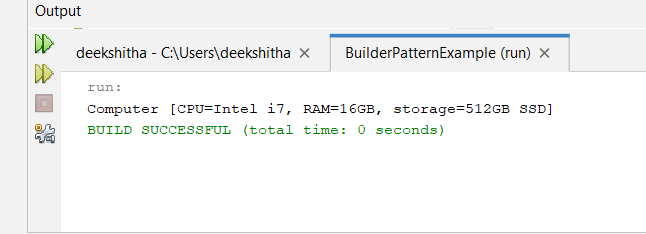
.build();

System.out.println(computer);

}

}

Output:



**Exercise 4: Implementing the Adapter Pattern**

// PaymentProcessor.java

interface PaymentProcessor {

void processPayment(double amount);

}

// PayPal.java

class PayPal {

public void sendPayment(double amount) {

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

}

}

// PayPalAdapter.java

class PayPalAdapter implements PaymentProcessor {

private PayPal payPal;

public PayPalAdapter(PayPal payPal) {

this.payPal = payPal;

}

public void processPayment(double amount) {

payPal.sendPayment(amount);

}

}

// AdapterPatternExample.java

class AdapterPatternExample {

public static void main(String[] args) {

PayPal payPal = new PayPal();

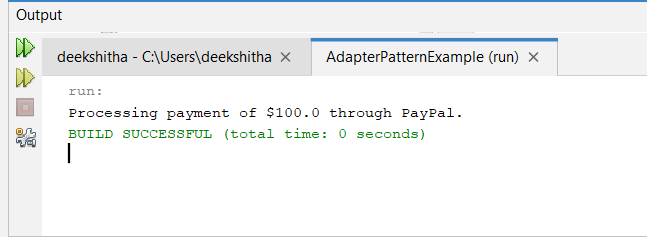
PaymentProcessor payPalAdapter = new PayPalAdapter(payPal);

payPalAdapter.processPayment(100.0);

}

}

Output:



**Exercise 5: Implementing the Decorator Pattern**

// Notifier.java

interface Notifier {

void send(String message);

}

// EmailNotifier.java

class EmailNotifier implements Notifier {

public void send(String message) {

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

}

}

// NotifierDecorator.java

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

class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

notifier.send(message);

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

}

}

// DecoratorPatternExample.java

class DecoratorPatternExample {

public static void main(String[] args) {

Notifier emailNotifier = new EmailNotifier();

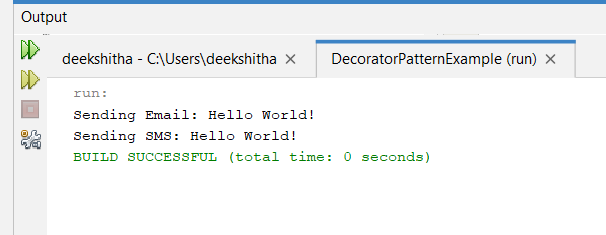
Notifier smsNotifier = new SMSNotifierDecorator(emailNotifier);

smsNotifier.send("Hello World!");

}

}

Output:



**Exercise 6: Implementing the Proxy Pattern**

// Image.java

interface Image {

void display();

}

// RealImage.java

class RealImage implements Image {

private String fileName;

public RealImage(String fileName) {

this.fileName = fileName;

loadFromDisk();

}

private void loadFromDisk() {

System.out.println("Loading " + fileName);

}

public void display() {

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

}

}

// ProxyImage.java

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();

}

}

// ProxyPatternExample.java

class ProxyPatternExample {

public static void main(String[] args) {

Image image = new ProxyImage("test\_image.jpg");

// Image will be loaded from disk

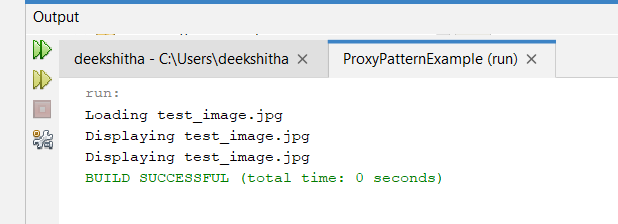
image.display();

// Image will not be loaded from disk

image.display();

}

}

Output:

**Exercise 7: Implementing the Observer Pattern**

import java.util.ArrayList;

import java.util.List;

// Observer.java

interface Observer {

void update(String stockPrice);

}

// Stock.java

interface Stock {

void register(Observer o);

void deregister(Observer o);

void notifyObservers();

}

// StockMarket.java

class StockMarket implements Stock {

private List<Observer> observers;

private String stockPrice;

public StockMarket() {

observers = new ArrayList<>();

}

public void setStockPrice(String stockPrice) {

this.stockPrice = stockPrice;

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);

}

}

}

// MobileApp.java

class MobileApp implements Observer {

private String stockPrice;

public void update(String stockPrice) {

this.stockPrice = stockPrice;

display();

}

public void display() {

System.out.println("Mobile App Stock Price: " + stockPrice);

}

}

// WebApp.java

class WebApp implements Observer {

private String stockPrice;

public void update(String stockPrice) {

this.stockPrice = stockPrice;

display();

}

public void display() {

System.out.println("Web App Stock Price: " + stockPrice);

}

}

// ObserverPatternExample.java

class ObserverPatternExample {

public static void main(String[] args) {

StockMarket stockMarket = new StockMarket();

Observer mobileApp = new MobileApp();

Observer webApp = new WebApp();

stockMarket.register(mobileApp);

stockMarket.register(webApp);

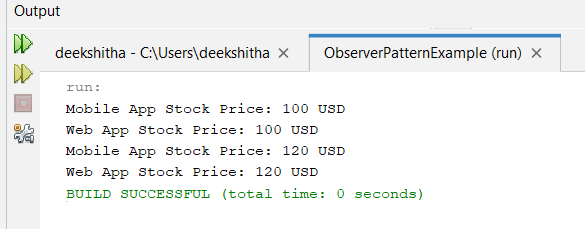
stockMarket.setStockPrice("100 USD");

stockMarket.setStockPrice("120 USD");

}

}

Output:



**Exercise 8: Implementing the Strategy Pattern**

// PaymentStrategy.java

interface PaymentStrategy {

void pay(double amount);

}

// CreditCardPayment.java

class CreditCardPayment implements PaymentStrategy {

public void pay(double amount) {

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

}

}

// PayPalPayment.java

class PayPalPayment implements PaymentStrategy {

public void pay(double amount) {

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

}

}

// PaymentContext.java

class PaymentContext {

private PaymentStrategy strategy;

public void setPaymentStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void pay(double amount) {

strategy.pay(amount);

}

}

// StrategyPatternExample.java

class StrategyPatternExample {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

context.setPaymentStrategy(new CreditCardPayment());

context.pay(100.0);

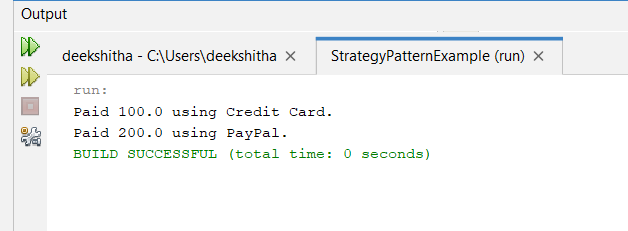
context.setPaymentStrategy(new PayPalPayment());

context.pay(200.0);

}

}

Output:



**Exercise 9: Implementing the Command Pattern**

// Command.java

interface Command {

void execute();

}

// Light.java

class Light {

public void turnOn() {

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

}

public void turnOff() {

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

}

}

// LightOnCommand.java

class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOn();

}

}

// LightOffCommand.java

class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOff();

}

}

// RemoteControl.java

class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

command.execute();

}

}

class CommandPattern {

public static void main(String[] args) {

Light light = new Light();

Command lightOn = new LightOnCommand(light);

Command lightOff = new LightOffCommand(light);

RemoteControl remote = new RemoteControl();

remote.setCommand(lightOn);

remote.pressButton();

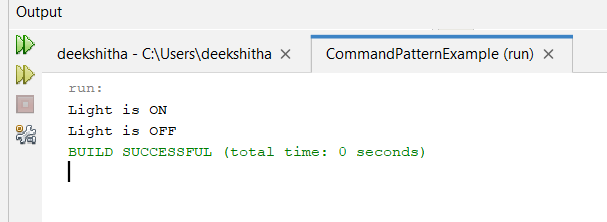
remote.setCommand(lightOff);

remote.pressButton();

}

}

Output:



**Exercise 10: Implementing the MVC Pattern**

class Student {

private String id;

private String name;

private String grade;

// Constructor

public Student(String id, String name, String grade) {

this.id = id;

this.name = name;

this.grade = grade;

}

// Getters and Setters

public String getId() { return id; }

public void setId(String id) { this.id = id; }

public String getName() { return name; }

public void setName(String name) { this.name = name; }

public String getGrade() { return grade; }

public void setGrade(String grade) { this.grade = grade; }

}

class StudentView {

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

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

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

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

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

}

}

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(String id) {

model.setId(id);

}

public void setStudentGrade(String grade) {

model.setGrade(grade);

}

public String getStudentName() {

return model.getName();

}

public String getStudentId() {

return model.getId();

}

public String getStudentGrade() {

return model.getGrade();

}

public void updateView() {

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

}

}

class MVCPatternDemo {

public static void main(String[] args) {

// Create model (student)

Student student = new Student("101", "Rohit", "A");

// Create view

StudentView view = new StudentView();

// Create controller

StudentController controller = new StudentController(student, view);

// Initial display

controller.updateView();

// Update student details using controller

controller.setStudentName("Jane");

controller.setStudentGrade("B+");

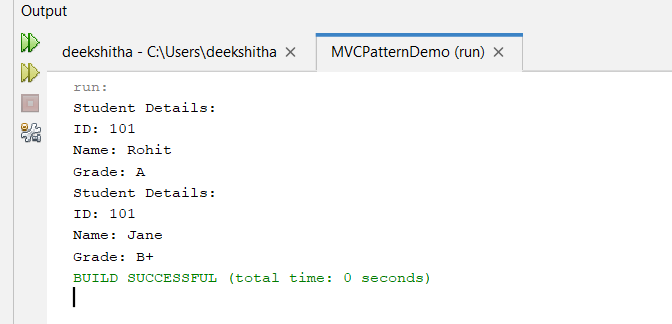
// Display updated details

controller.updateView();

}

}

Output:



**Exercise 11: Implementing Dependency Injection**

// ChatMediator.java

import java.util.ArrayList;

import java.util.List;

interface ChatMediator {

void sendMessage(String message, User user);

void addUser(User user);

}

// ChatMediatorImpl.java

class ChatMediatorImpl implements ChatMediator {

private List<User> users;

public ChatMediatorImpl() {

this.users = new ArrayList<>();

}

public void addUser(User user) {

this.users.add(user);

}

public void sendMessage(String message, User user) {

for (User u : this.users) {

// message should not be received by the user sending it

if (u != user) {

u.receive(message);

}

}

}

}

// User.java

abstract class User {

protected ChatMediator mediator;

protected String name;

public User(ChatMediator mediator, String name) {

this.mediator = mediator;

this.name = name;

}

public abstract void send(String message);

public abstract void receive(String message);

}

// UserImpl.java

class UserImpl extends User {

public UserImpl(ChatMediator mediator, String name) {

super(mediator, name);

}

public void send(String message) {

System.out.println(this.name + ": Sending Message=" + message);

mediator.sendMessage(message, this);

}

public void receive(String message) {

System.out.println(this.name + ": Received Message=" + message);

}

}

// MediatorPatternExample.java

class MediatorPatternExample {

public static void main(String[] args) {

ChatMediator mediator = new ChatMediatorImpl();

User user1 = new UserImpl(mediator, "Alice");

User user2 = new UserImpl(mediator, "Bob");

User user3 = new UserImpl(mediator, "Charlie");

User user4 = new UserImpl(mediator, "Dave");

mediator.addUser(user1);

mediator.addUser(user2);

mediator.addUser(user3);

mediator.addUser(user4);

user1.send("Hello, everyone!");

}

}

Output:

