**DESIGN PATTERNS AND PRINCIPLES (Additional Problem)**

**Exercise 3: Implementing the Builder Pattern**

**CODE:**

public class BuilderExample {

static class computer {

private String cpu;

private String ram;

private String storage;

private String graphicscard;

private computer(builder b) {

this.cpu = b.cpu;

this.ram = b.ram;

this.storage = b.storage;

this.graphicscard = b.graphicscard;

}

static class builder {

private String cpu;

private String ram;

private String storage;

private String graphicscard;

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 setgraphicscard(String graphicscard) {

this.graphicscard = graphicscard;

return this;

}

public computer build() {

return new computer(this);

}

}

public void showconfig() {

System.out.println("computer configuration:");

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

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

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

System.out.println("graphics card: " + (graphicscard != null ? graphicscard : "none"));

}

}

public static void main(String[] args) {

computer basiccomputer = new computer.builder()

.setcpu("intel i3")

.setram("4gb")

.setstorage("500gb hdd")

.build();

computer gamingcomputer = new computer.builder()

.setcpu("intel i9")

.setram("32gb")

.setstorage("1tb ssd")

.setgraphicscard("nvidia rtx 4090")

.build();

basiccomputer.showconfig();

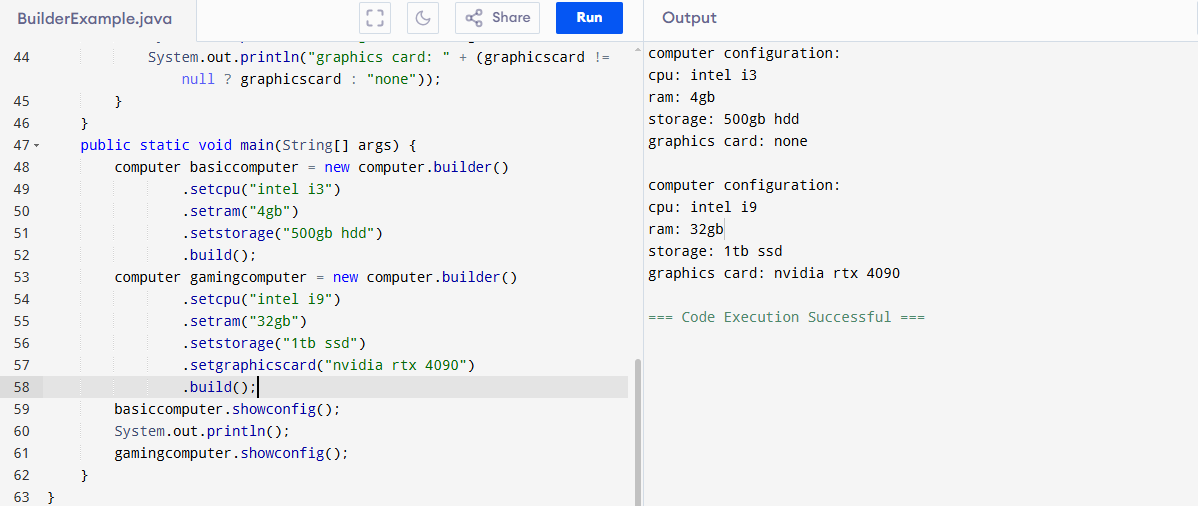
System.out.println();

gamingcomputer.showconfig();

}

}

**OUTPUT:**

****

**Exercise 4: Implementing the Adapter Pattern**

**CODE:**

public class AdapterPatternExample {

interface paymentprocessor {

void processpayment(double amount);

}

static class paypal {

public void sendpayment(double amount) {

System.out.println("Processed payment of " + amount + " via PayPal");

}

}

static class stripe {

public void makepayment(double amount) {

System.out.println("Processed payment of " + amount + " via Stripe");

}

}

static class paypaladapter implements paymentprocessor {

private paypal pp;

public paypaladapter(paypal pp) {

this.pp = pp;

}

public void processpayment(double amount) {

pp.sendpayment(amount);

}

}

static class stripeadapter implements paymentprocessor {

private stripe st;

public stripeadapter(stripe st) {

this.st = st;

}

public void processpayment(double amount) {

st.makepayment(amount);

}

}

public static void main(String[] args) {

paymentprocessor paypal= new paypaladapter(new paypal());

paypal.processpayment(500.0);

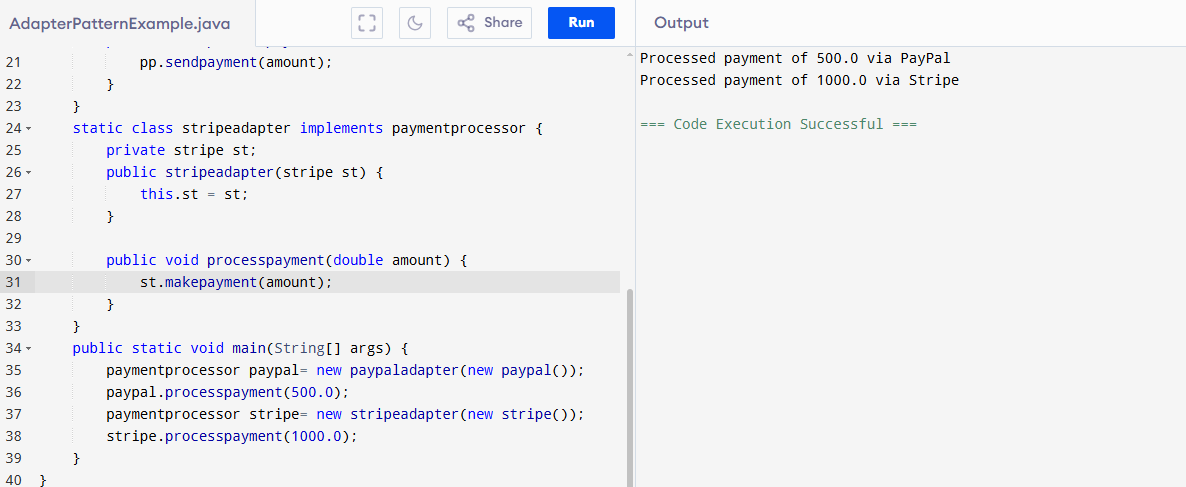
paymentprocessor stripe= new stripeadapter(new stripe());

stripe.processpayment(1000.0);

}

}

**OUTPUT:**

****

**Exercise 5: Implementing the Decorator Pattern**

**CODE:**

public class DecoratorPatternExample{

interface notifier {

void send(String message);

}

static class emailnotifier implements notifier {

public void send(String message) {

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

}

}

static abstract class notifierdecorator implements notifier {

protected notifier wrappee;

public notifierdecorator(notifier wrappee) {

this.wrappee = wrappee;

}

public void send(String message) {

wrappee.send(message);

}

}

static class smsnotifierdecorator extends notifierdecorator {

public smsnotifierdecorator(notifier wrappee) {

super(wrappee);

}

public void send(String message) {

super.send(message);

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

}

}

static class slacknotifierdecorator extends notifierdecorator {

public slacknotifierdecorator(notifier wrappee) {

super(wrappee);

}

public void send(String message) {

super.send(message);

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

}

}

public static void main(String[] args) {

notifier basic = new emailnotifier();

notifier sms = new smsnotifierdecorator(basic);

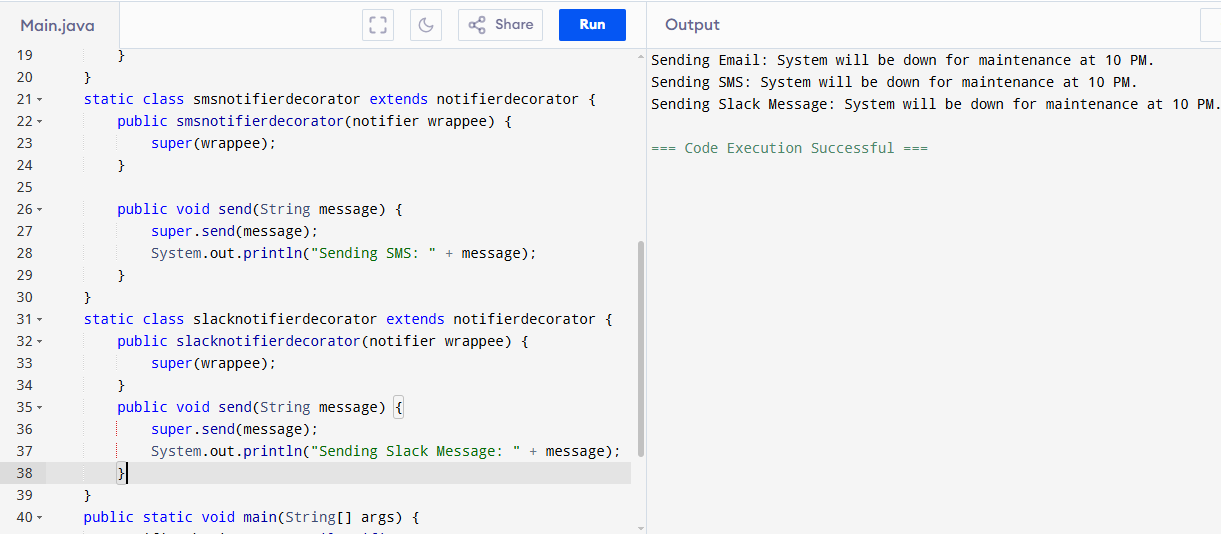
notifier multi = new slacknotifierdecorator(sms);

multi.send("System will be down for maintenance at 10 PM.");

}

}

**OUTPUT:**

****

**Exercise 6: Implementing the Proxy Pattern**

**CODE:**

public class ProxyPatternExample{

interface image {

void display();

}

static class realimage implements image {

private String filename;

public realimage(String filename) {

this.filename = filename;

loadfromserver();

}

private void loadfromserver() {

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

}

public void display() {

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

}

}

static class proxyimage implements image {

private String filename;

private realimage realimg;

public proxyimage(String filename) {

this.filename = filename;

}

public void display() {

if (realimg == null) {

realimg = new realimage(filename); // lazy loading

} else {

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

}

realimg.display();

}

}

public static void main(String[] args) {

image img1 = new proxyimage("nature.jpg");

img1.display();

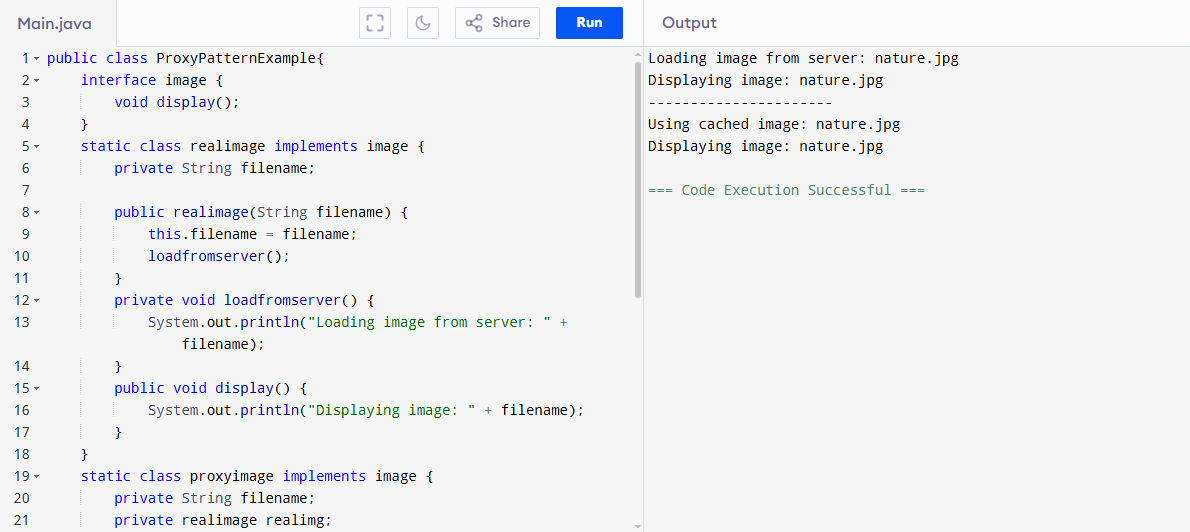
System.out.println("----------------------");

img1.display();

}

}

**OUTPUT:**

****

**Exercise 7: Implementing the Observer Pattern**

**CODE:**

import java.util.\*;

public class ObserverPatternExample{

interface observer {

void update(String stockName, double price);

}

interface stock {

void register(observer o);

void deregister(observer o);

void notifyobservers();

}

static class stockmarket implements stock {

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

private String stockName;

private double price;

public void setstock(String name, double price) {

this.stockName = name;

this.price = 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(stockName, price);

}

}

}

static class mobileapp implements observer {

public void update(String stockName, double price) {

System.out.println("Mobile App - " + stockName + " price updated to " + price);

}

}

static class webapp implements observer {

public void update(String stockName, double price) {

System.out.println("Web App - " + stockName + " price updated to " + price);

}

}

public static void main(String[] args) {

stockmarket market = new stockmarket();

observer mobile = new mobileapp();

observer web = new webapp();

market.register(mobile);

market.register(web);

market.setstock("CTS", 3545.75);

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

market.setstock("INFY", 1462.30);

System.out.println("------ Deregister mobile app ------");

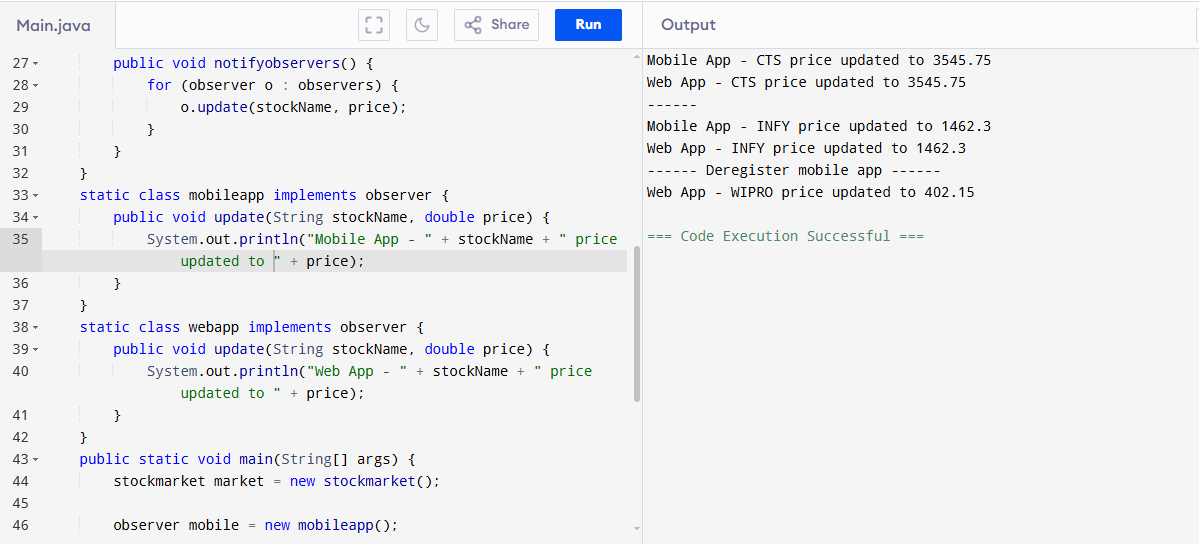
market.deregister(mobile);

market.setstock("WIPRO", 402.15);

}

}

**OUTPUT:**

****

**Exercise 8: Implementing the Strategy Pattern**

**CODE:**

import java.util.\*;

public class StrategyPatternExample{

interface paymentstrategy {

void pay(double amount);

}

static 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);

}

}

static 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);

}

}

static class paymentcontext {

private paymentstrategy strategy;

public void setstrategy(paymentstrategy strategy) {

this.strategy = strategy;

}

public void execute(double amount) {

if (strategy == null) {

System.out.println("Payment method not selected.");

} else {

strategy.pay(amount);}

}

}

public static void main(String[] args) {

paymentcontext con = new paymentcontext();

con.setstrategy(new creditcardpayment("1234-5678-9012-3456"));

con.execute(2500.00);

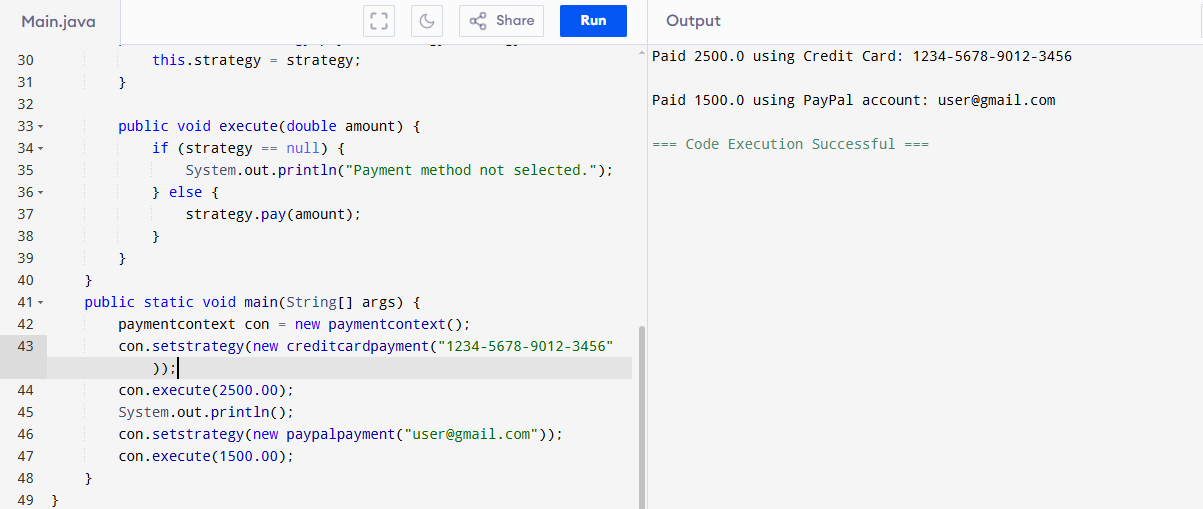
System.out.println();

con.setstrategy(new paypalpayment("user@gmail.com"));

con.execute(1500.00);

}}

**OUTPUT:**



**Exercise 9: Implementing the Command Pattern**

**CODE:**

public class CommandPatternExample{

interface command {

void execute();

}

static class light {

public void turnon() {

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

}

public void turnoff() {

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

}

}

static class lightoncommand implements command {

private light light;

public lightoncommand(light light) {

this.light = light;

}

public void execute() {

light.turnon();

}

}

static class lightoffcommand implements command {

private light light;

public lightoffcommand(light light) {

this.light = light;

}

public void execute() {

light.turnoff();

}

}

static 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.");

}

}

}

public static void main(String[] args) {

light l= new light();

command on = new lightoncommand(l);

command off = new lightoffcommand(l);

remotecontrol r= new remotecontrol();

r.setcommand(on);

r.pressbutton();

r.setcommand(off);

r.pressbutton();

}

}

**OUTPUT:**



**Exercise 10: Implementing the MVC Pattern**

**CODE:**

public class MVCPatternExample{

static class student {

String name;

String id;

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; }

}

static 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);

}

}

static class studentcontroller {

student model;

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 updatestudentview() {

view.displaystudentdetails(model.getname(), model.getid(), model.getgrade());

}

}

public static void main(String[] args) {

student s= new student("nithya", "S123", "A");

studentview view = new studentview();

studentcontroller c= new studentcontroller(s, view);

c.updatestudentview();

System.out.println("------ After Update ------");

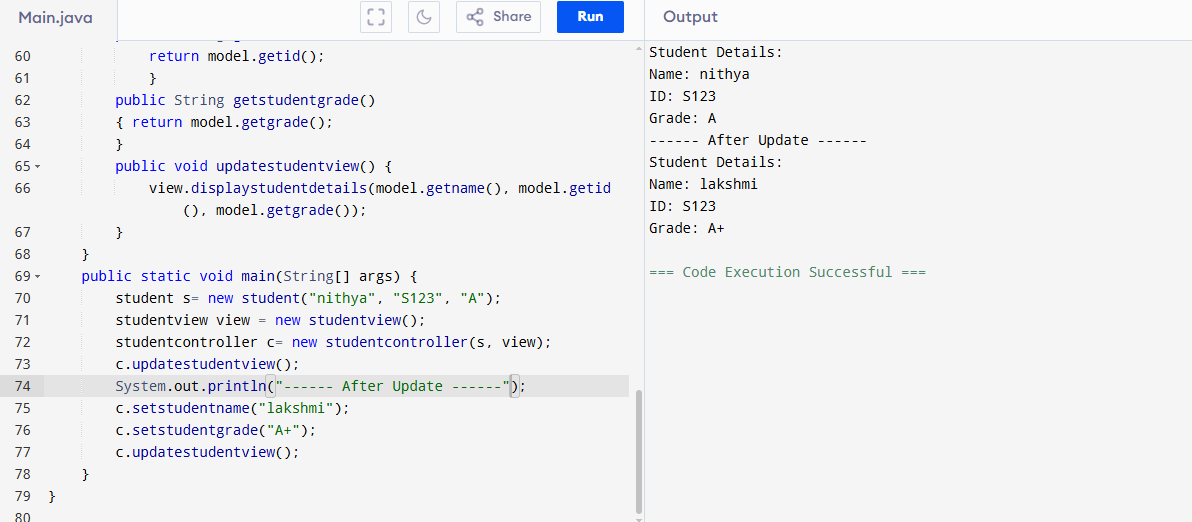
c.setstudentname("lakshmi");

c.setstudentgrade("A+");

c.updatestudentview();

}}

**OUTPUT:**

****

**Exercise 11: Implementing Dependency Injection**

**CODE:**

public class DependencyInjectionExample{

interface customerrepository {

String findcustomerbyid(String id);

}

static class customerrepositoryimpl implements customerrepository {

public String findcustomerbyid(String id) {

return "Customer{id='" + id + "', name='nithya lakshmi'}";

}

}

static class customerservice {

customerrepository repo;

public customerservice(customerrepository repo) {

this.repo= repo;

}

public void displaycustomer(String id) {

String customer = repo.findcustomerbyid(id);

System.out.println("Fetched: " + customer);

}

}

public static void main(String[] args) {

customerrepository r = new customerrepositoryimpl();

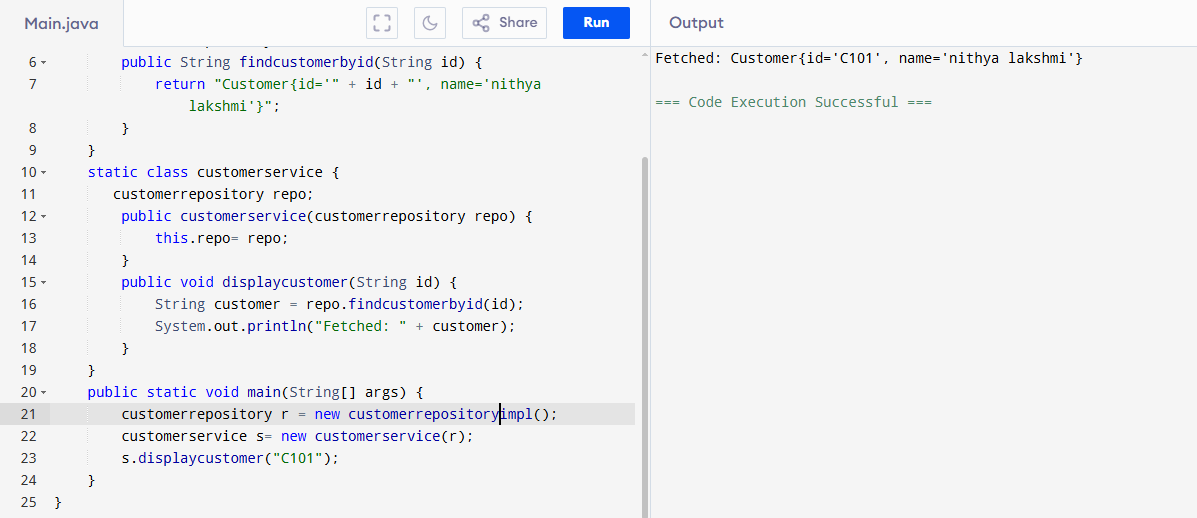
customerservice s= new customerservice(r);

s.displaycustomer("C101");

}

}

**OUTPUT:**

****