**Exercise 3: Implementing the Builder Pattern**

**Code:**

class Computer

{

private String CPU;

private String RAM;

private String storage;

private String os;

private Computer(Builder builder)

{

this.CPU=builder.CPU;

this.RAM=builder.RAM;

this.storage=builder.storage;

this.os=builder.os;

}

public void display()

{

System.out.println("CPU: "+CPU+" RAM: "+RAM+" Storage: "+storage+" Operating System: "+os);

}

public static class Builder

{

String CPU;

String RAM;

String storage;

String os;

public Builder(String CPU)

{

this.CPU=CPU;

}

public Builder setRAM(String RAM)

{

this.RAM=RAM;

return this;

}

public Builder setStorage(String storage)

{

this.storage=storage;

return this;

}

public Builder setOs(String os)

{

this.os=os;

return this;

}

public Computer build()

{

return new Computer(this);

}

}

}

public class BuilderPattern

{

public static void main(String[] args)

{

Computer a=new Computer.Builder("Intel i9")

.setRAM("32GB")

.setStorage("1TB SSD")

.setOs("Windows")

.build();

a.display();

Computer b=new Computer.Builder("Intel i3")

.setRAM("16GB")

.setStorage("128GB SSD")

.setOs("Mac")

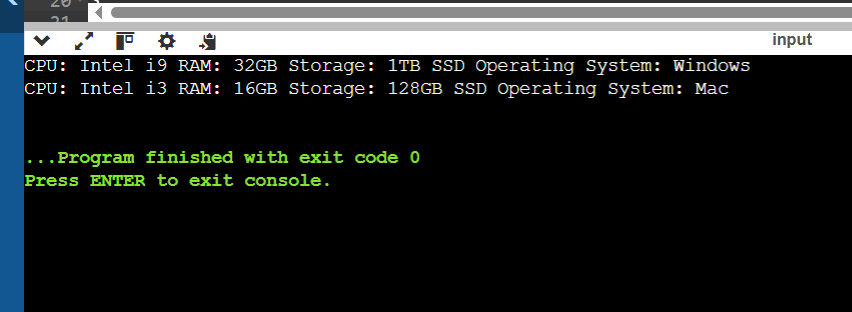
.build();

b.display();

}

}

**Output:**

****

**Exercise 4: Implementing the Adapter Pattern**

**Code:**

interface PaymentProcessor

{

void processPayment(String amount);

}

class PaypalGateway

{

public void sendamt(String amt)

{

System.out.println("Sent $"+amt+" using Paypal Gateway");

}

}

class RazorPay{

public void pay(String amt)

{

System.out.println("Paid $"+amt+" using RazorPay Gateway");

}

}

class PayPalAdapter implements PaymentProcessor{

private PaypalGateway pp;

public PayPalAdapter(PaypalGateway pp)

{

this.pp=pp;

}

public void processPayment(String amount)

{

pp.sendamt(amount);

}

}

class RazorPayAdaptor implements PaymentProcessor{

private RazorPay rp;

public RazorPayAdaptor(RazorPay rp)

{

this.rp=rp;

}

public void processPayment(String amount)

{

rp.pay(amount);

}

}

public class AdapterPattern{

public static void main(String[] args) {

PaymentProcessor p1=new PayPalAdapter(new PaypalGateway());

p1.processPayment("5000");

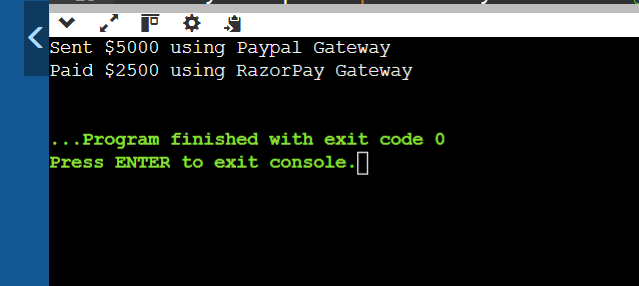
PaymentProcessor p2=new RazorPayAdaptor(new RazorPay());

p2.processPayment("2500");

}

}

**Output:**

****

**Exercise 5: Implementing the Decorator Pattern**

**Code:**

interface Notifier

{

void send(String msg);

}

class EmailNotifier implements Notifier{

public void send(String msg)

{

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

}

}

abstract class NotifierDecorator implements Notifier{

protected Notifier n;

public NotifierDecorator(Notifier n){

this.n=n;

}

public void send(String msg)

{

n.send(msg);

}

}

class SmsNotifierDecorator extends NotifierDecorator{

public SmsNotifierDecorator(Notifier n)

{

super(n);

}

public void send(String msg)

{

super.send(msg);

sms(msg);

}

public void sms(String msg)

{

System.out.println("Sending an sms: "+msg);

}

}

class SlackNotifierDecorator extends NotifierDecorator{

public SlackNotifierDecorator(Notifier n)

{

super(n);

}

public void send(String msg){

super.send(msg);

slack(msg);

}

public void slack(String msg)

{

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

}

}

public class DecoratorPattern

{

public static void main(String[] args) {

Notifier nt=new EmailNotifier();

System.out.println("EMAIL");

nt.send("Security Alert!!!");

Notifier sm=new SmsNotifierDecorator(nt);

System.out.println("SMS and EMail");

sm.send("Balance limit alert");

Notifier sl=new SlackNotifierDecorator(sm);

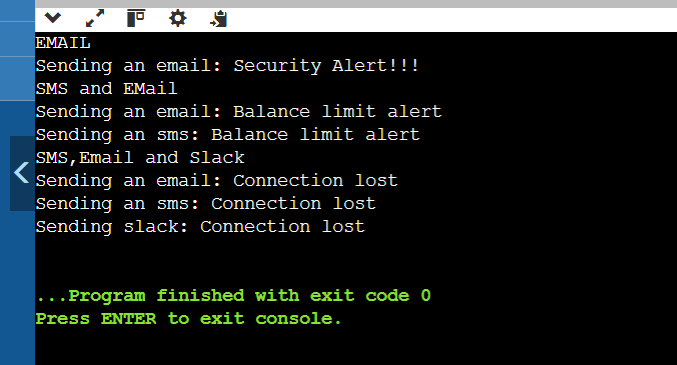
System.out.println("SMS,Email and Slack");

sl.send("Connection lost");

}

}

**Output:**

****

**Exercise 6: Implementing the Proxy Pattern**

**Code:**

interface Image{

void display();

}

class RealImage implements Image{

private String name;

public RealImage(String name)

{

this.name=name;

loadfromRemoteServer();

}

private void loadfromRemoteServer()

{

System.out.println("Loading an image from Remote Server: "+name);

}

public void display()

{

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

}

}

class ProxyImage implements Image{

private String name;

private RealImage img;

public ProxyImage(String name)

{

this.name=name;

}

public void display()

{

if(img!=null)

{

img.display();

}

else

{

img=new RealImage(name);

}

}

}

public class ProxyPattern

{

public static void main(String[] args) {

Image i1=new ProxyImage("Hyd.jpg");

Image i2=new ProxyImage("Beach.jpg");

System.out.println("\nInitial image1");

i1.display();

System.out.println("\nUpdated image1");

i1.display();

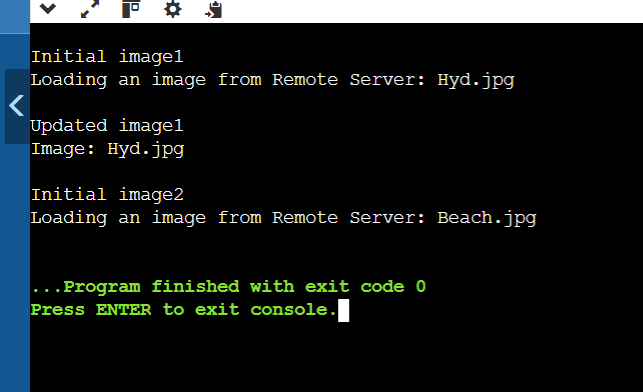
System.out.println("\nInitial image2");

i2.display();

}

}

**Output:**

****

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Exercise 7: Implementing the Observer Pattern**

**Code:**

import java.util.\*;

interface stock{

void register(Observer o);

void deregister(Observer o);

void notifyObservers();

void set(double amt);

}

interface Observer{

void update(double amt);

}

class Stockmarket implements stock{

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

private double price;

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(price);

}

}

public void set(double amt)

{

this.price=amt;

System.out.println("Price updated to $"+price);

notifyObservers();

}

}

class Mobileapp implements Observer{

private String name;

public Mobileapp(String name)

{

this.name=name;

}

public void update(double p)

{

System.out.println("Mobileapp: "+name+"price is now: "+p);

}

}

class Webapp implements Observer{

private String customer;

public Webapp(String customer)

{

this.customer=customer;

}

public void update(double p)

{

System.out.println("Webapp: "+customer+"price is now: "+p);

}

}

public class ObserverPattern

{

public static void main(String[] args) {

Stockmarket s=new Stockmarket();

Observer m1=new Mobileapp("Alice");

Observer m2=new Mobileapp("Bob");

Observer w1=new Webapp("Charlie");

s.register(m1);

s.register(m2);

s.register(w1);

s.set(200.0);

s.set(123.0);

s.deregister(m2);

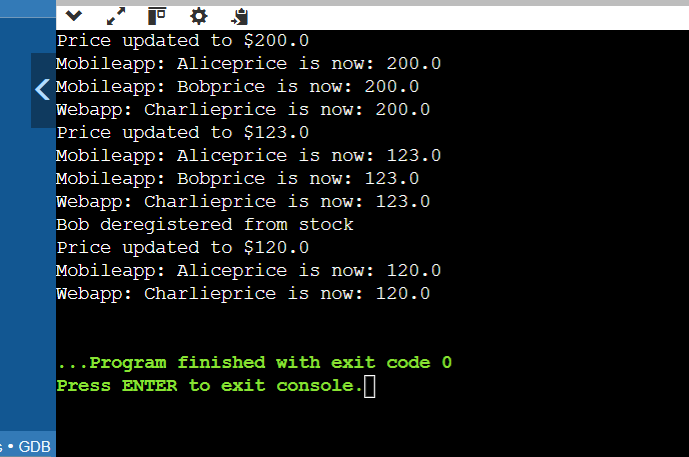
System.out.println("Bob deregistered from stock");

s.set(120.0);

}

}

**Output:**

****

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Exercise 8: Implementing the Strategy Pattern**

**Code:**

import java.util.\*;

interface Paymentstrategy{

void pay(double amt);

}

class CreditcardPayment implements Paymentstrategy{

private String id;

private String name;

public CreditcardPayment(String id,String name)

{

this.id=id;

this.name=name;

}

public void pay(double amt)

{

System.out.println("Name: "+name+" card number: "+id+" Paid: $"+amt+" using Creditcard");

}

}

class PayPalPayment implements Paymentstrategy{

private String name;

public PayPalPayment(String name)

{

this.name=name;

}

public void pay(double amt)

{

System.out.println("Name "+name+" Paid $"+amt+" using PayPal");

}

}

class Paymentcontext{

private Paymentstrategy st;

public void set(Paymentstrategy st)

{

this.st=st;

}

public void pay(double amt)

{

if(st==null)

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

else

st.pay(amt);

}

}

public class StrategyPattern

{

public static void main(String[] args) {

Paymentcontext c=new Paymentcontext();

Paymentstrategy p1=new CreditcardPayment("1234","Bob");

c.set(p1);

System.out.println("\nCredit Card");

c.pay(1500.0);

Paymentstrategy p2=new PayPalPayment("David");

c.set(p2);

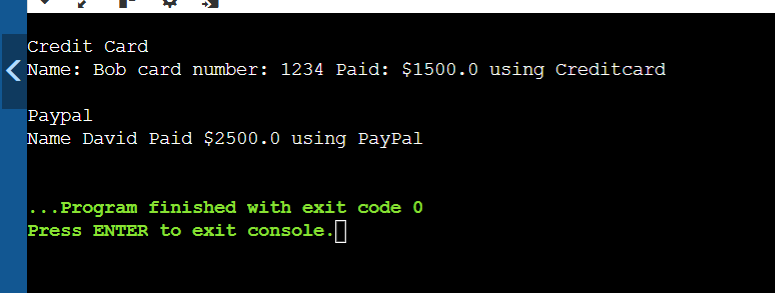
System.out.println("\nPaypal");

c.pay(2500.0);

}

}

**Output:**



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Exercise 9: Implementing the Command Pattern**

**Code:**

interface command{

void exec();

}

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 l;

public LightOnCommand(Light l)

{

this.l=l;

}

public void exec()

{

l.turnon();

}

}

class LightOffCommand implements command{

private Light l;

public LightOffCommand(Light l)

{

this.l=l;

}

public void exec()

{

l.turnoff();

}

}

class Remotecontrol

{

private command c;

public void set(command c)

{

this.c=c;

}

public void action(){

if(c!=null)

c.exec();

else

System.out.println("No command to perform");

}

}

public class CommandPattern

{

public static void main(String[] args) {

Light lt=new Light();

command on=new LightOnCommand(lt);

command off=new LightOffCommand(lt);

Remotecontrol rc=new Remotecontrol();

System.out.println("\nTurning ON the light");

rc.set(on);

rc.action();

System.out.println("\nTurning OFF the light");

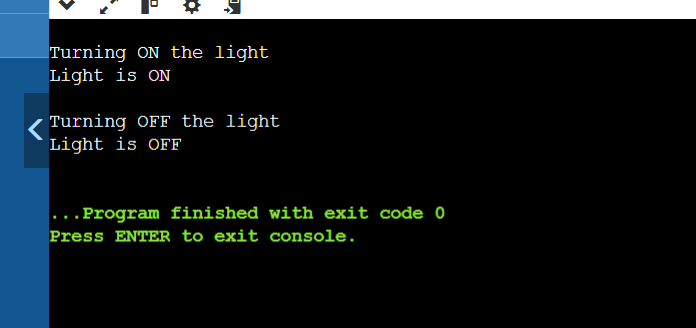
rc.set(off);

rc.action();

}

}

**Output:**

****

**Exercise 10: Implementing the MVC Pattern**

**Code:**

class student

{

private String name;

private String id;

private int marks;

public student(String name,String id,int marks)

{

this.name=name;

this.id=id;

this.marks=marks;

}

public String getname()

{

return name;

}

public String getid()

{

return id;

}

public int getmarks()

{

return marks;

}

public void setname(String name)

{

this.name=name;

}

public void setid(String id)

{

this.id=id;

}

public void setmarks(int marks){

this.marks=marks;

}

}

class studentView{

public void display(String name,String id,int marks)

{

System.out.println("Name: "+name+" ID: "+id+" Marks: "+marks);

}

}

class studentController{

private student model;

private studentView view;

public studentController(student model,studentView view)

{

this.model=model;

this.view=view;

}

public String getsname()

{

return model.getname();

}

public String getsid()

{

return model.getid();

}

public int getsmarks()

{

return model.getmarks();

}

public void setsname(String name)

{

model.setname(name);

}

public void setsid(String id)

{

model.setid(id);

}

public void setsmarks(int marks)

{

model.setmarks(marks);

}

public void updateview()

{

view.display(model.getname(),model.getid(),model.getmarks());

}

}

public class MVCPattern

{

public static void main(String[] args) {

student s1=new student("Charlie","12",98);

studentView sv=new studentView();

studentController sc=new studentController(s1,sv);

sc.updateview();

sc.setsname("Charlie Pope");

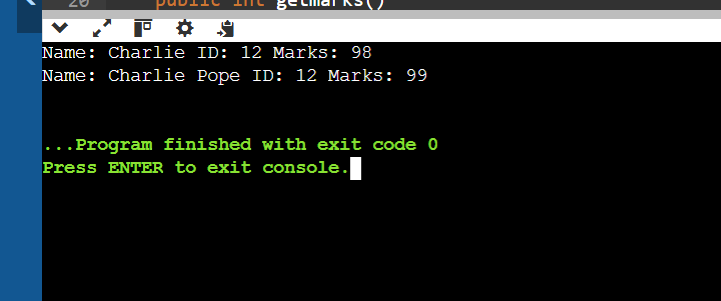
sc.setsmarks(99);

sc.updateview();

}

}

**Output:**

****

**Exercise 11: Implementing Dependency Injection**

**Code:**

class Customer{

private String id;

private String name;

private String phnum;

public Customer(String id,String name,String phnum)

{

this.id=id;

this.name=name;

this.phnum=phnum;

}

public String getid(){

return id;

}

public String getname()

{

return name;

}

public String getphnum()

{

return phnum;

}

public void display()

{

System.out.println("Name: "+name+" ID: "+id+" Phone number: "+phnum);

}

}

interface CustomerRepository

{

Customer findCustomerById(String id);

}

class CustomerRepositoryImpl implements CustomerRepository{

public Customer findCustomerById(String id)

{

Customer c=new Customer("Alice","12","1234567892");

return c;

}

}

class CustomerService{

private CustomerRepository cr;

public CustomerService(CustomerRepository cr)

{

this.cr=cr;

}

public void show(String id)

{

Customer ct=cr.findCustomerById(id);

ct.display();

}

}

public class DependencyInjection

{

public static void main(String[] args) {

CustomerRepository rep=new CustomerRepositoryImpl();

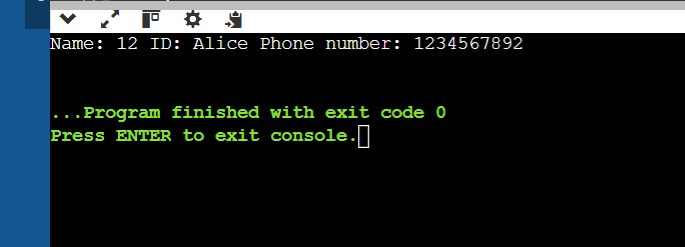
CustomerService serv=new CustomerService(rep);

serv.show("12");

}

}

**Output:**

****

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**