# Design Patterns and Principles

**Exercise-1: Implementing the Singleton Pattern**

**Code:**

public class Main{

public static void main(String[] args){

Singleton object1 =Singleton.getInstance();

object1.message(“first object”);

Singleton object2 =Singleton.getInstance();

object2.message("second object ");

if(object1==object2)

System.out.println("both refers to the same object");

else

System.out.println("different objects");

}

}

class Singleton{

private static Singleton obj1;

private Singleton(){

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

}

public static Singleton getInstance(){

if(obj1==null){

obj1=new Singleton();

}

return obj1;

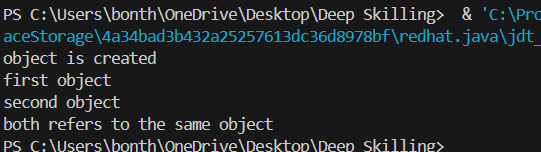
}

public static void message(String msg){

System.out.println(msg );

}

}



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

interface Document{

void method();

}

class pdf implements Document{

public void method(){

System.out.println("This is a pdf document instance");

}

}

class word implements Document{

public void method(){

System.out.println("This is a word document instance");

}

}

class excel implements Document{

public void method(){

System.out.println("This is a excel document instance");

}

}

abstract class DocumentFactory{

public abstract Document createDocument();

}

class pdfDocument extends DocumentFactory{

public Document createDocument(){

return new pdf();

}

}

class wordDocument extends DocumentFactory{

public Document createDocument(){

return new word();

}

}

class excelDocument extends DocumentFactory{

public Document createDocument(){

return new excel(); }

}

public class Main{

public static void main(String[] args){

DocumentFactory p1=new pdfDocument();

Document pdf1=p1.createDocument();

pdf1.method();

DocumentFactory w1=new wordDocument();

Document word1=w1.createDocument();

word1.method();

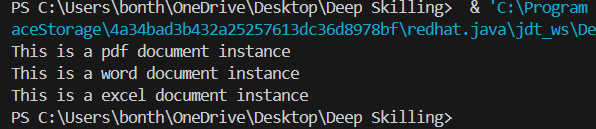
DocumentFactory e1=new excelDocument();

Document excel1=e1.createDocument();

excel1.method();

}

}



**Exercise 3: Implementing the Builder Pattern**

class Computer{

private int CPU,RAM,Storage;

private Computer(Builder b){

this.CPU=b.CPU;

this.RAM=b.RAM;

this.Storage=b.Storage;

System.out.println("instance of computer created with CPU:"+CPU+" RAM:"+RAM+" Storage:"+Storage);

}

static class Builder{

private int CPU;

private int RAM;

private int Storage;

public Computer build(){

return new Computer(this);

}

public Builder setCPU(int CPU){

this.CPU=CPU;

return this; }

public Builder setRAM(int RAM){

this.RAM=RAM;

return this;

}

public Builder setStorage(int Storage){

this.Storage=Storage;

return this;

}

}

}

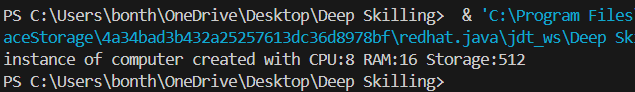
public class Main{

public static void main(String[] args){

Computer c1=new Computer.Builder().setCPU(8).setRAM(16).setStorage(512).build();

}

}



**Exercise 4: Implementing the Adapter Pattern**

interface PaymentProcessor {

void processPayment(double amount);

}

class PaymentGateway1 {

public void sendPayment(double amount) {

System.out.println("Paid " + amount + " using Gateway1.");

}

}

class PaymentGateway2 {

public void makePayment(double amount) {

System.out.println("Paid " + amount + " using Gateway2.");

}

}

class Gateway1 implements PaymentProcessor {

private PaymentGateway1 p1;

public Gateway1(PaymentGateway1 p1) {

this.p1 = p1;

}

public void processPayment(double amount) {

p1.sendPayment(amount);

}

}

class Gateway2 implements PaymentProcessor {

private PaymentGateway2 p2;

public Gateway2(PaymentGateway2 p2) {

this.p2 = p2;

}

public void processPayment(double amount) {

p2.makePayment(amount);

}

}

public class Main {

public static void main(String[] args) {

PaymentGateway1 gateway1 = new PaymentGateway1();

PaymentProcessor adapter1 = new Gateway1(gateway1);

adapter1.processPayment(100.0);

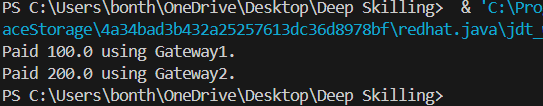
PaymentGateway2 gateway2 = new PaymentGateway2();

PaymentProcessor adapter2 = new Gateway2(gateway2);

adapter2.processPayment(200.0);

}

}



**Exercise 5: Implementing the Decorator Pattern**

interface Notifier{

void send(String message);}

class EmailNotifier implements Notifier{

public void send(String message) {

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

}

}

abstract class NotifierDecorator implements Notifier {

protected Notifier n1;

public NotifierDecorator(Notifier n1) {

this.n1 = n1;

}

public void send(String message) {

n1.send(message);

}

}

class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier n2) {

super(n2);

}

public void send(String message) {

super.send(message);

sendSMS(message);

}

private void sendSMS(String message) {

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

}

}

public class Main {

public static void main(String[] args) {

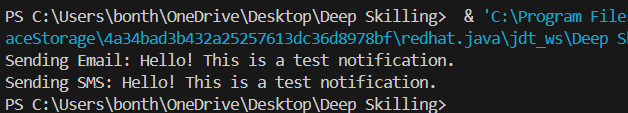
Notifier e1 = new EmailNotifier();

Notifier sms1= new SMSNotifierDecorator(e1);

sms1.send("Hello! This is a test notification.");

}

}



**Exercise 6: Implementing the Proxy Pattern**

interface Image {

void display();

}

class RealImage implements Image {

private String file1;

public RealImage(String file1) {

this.file1 = file1;

loadFromRemoteServer();

}

private void loadFromRemoteServer() {

System.out.println("Loading image from remote server: " + file1);

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

public void display() {

System.out.println("Displaying image: " + file1);

}

}

class ProxyImage implements Image {

private String file2;

private RealImage realImage;

public ProxyImage(String file2) {

this.file2 = file2;

}

public void display() {

if (realImage == null) {

realImage = new RealImage(file2);

}

realImage.display();

}

}

public class Main {

public static void main(String[] args) {

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

System.out.println("First call:");

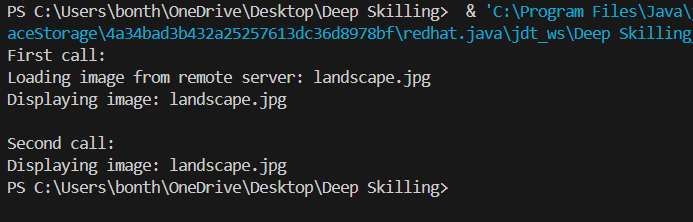
i1.display();

System.out.println("\nSecond call:");

i1.display();

}

}



**Exercise 7: Implementing the Observer Pattern**

import java.util.\*;

interface Stock {

void registerObserver(Observer o);

void removeObserver(Observer o);

void notifyObservers();

}

interface Observer {

void update(String stockName, double stockPrice);

}

class StockMarket implements Stock {

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

private String stockName;

private double stockPrice;

public void registerObserver(Observer o) {

observers.add(o);

} public void removeObserver(Observer o) {

observers.remove(o);

} public void notifyObservers() {

for (Observer o : observers) {

o.update(stockName, stockPrice);

}

}

public void setStockPrice(String stockName, double price) {

this.stockName = stockName;

this.stockPrice = price;

System.out.println("Stock price updated: " + stockName + " = $" + price);

notifyObservers();

}

}

class MobileApp implements Observer {

private String appName;

public MobileApp(String appName) {

this.appName = appName;

}

public void update(String stockName, double stockPrice) {

System.out.println(appName + " - Update: " + stockName + " is now $" + stockPrice);

}

}

class WebApp implements Observer {

private String siteName;

public WebApp(String siteName) {

this.siteName = siteName;

} public void update(String stockName, double stockPrice) {

System.out.println(siteName + " - Update: " + stockName + " is now $" + stockPrice);

}

}public class Main {

public static void main(String[] args) {

StockMarket market = new StockMarket();

Observer mobile = new MobileApp("StockMobileApp");

Observer web = new WebApp("StockWebApp");

market.registerObserver(mobile);

market.registerObserver(web);

market.setStockPrice("AAPL", 190.50);

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

market.setStockPrice("TSLA", 703.10);

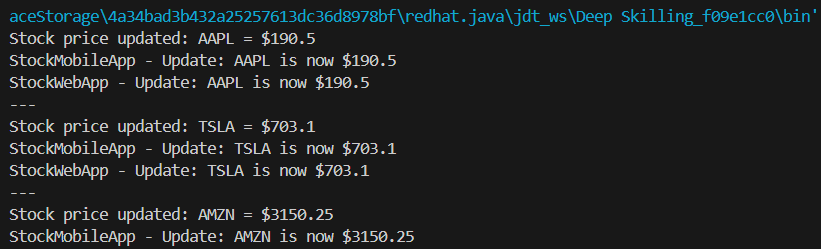
System.out.println("---");

market.removeObserver(web);

market.setStockPrice("AMZN", 3150.25);

}

}



**Exercise 8: Implementing the Strategy Pattern**

interface PaymentStrategy {

void pay(double amount);

}

class CreditCardPayment implements PaymentStrategy {

private String cardNumber;

private String name;

private String cvv;

public CreditCardPayment(String cardNumber, String name, String cvv) {

this.cardNumber = cardNumber;

this.name = name;

this.cvv = cvv;

}

public void pay(double amount) {

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

}

}

class PayPalPayment implements PaymentStrategy {

private String email;

private String password;

public PayPalPayment(String email, String password) {

this.email = email;

this.password = password;

}

public void pay(double amount) {

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

}

}

class PaymentContext {

private PaymentStrategy strategy;

public void setPaymentStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void executePayment(double amount) {

if (strategy == null) {

System.out.println("No payment strategy selected!");

} else {

strategy.pay(amount);

}

}

}

public class StrategyPatternExample {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

context.setPaymentStrategy(new CreditCardPayment("1234567890123456", "Alice", "123"));

context.executePayment(3000);

context.setPaymentStrategy(new PayPalPayment("alice@example.com", "securepass"));

context.executePayment(1500);

}

}

