**Exercise 3: Implementing the Builder Pattern**

Code:

class Computer {

private String cpu;

private String ram;

private String storage;

private String gpu;

private boolean hasWifi;

private boolean hasBluetooth;

private Computer(Builder builder) {

this.cpu = builder.cpu;

this.ram = builder.ram;

this.storage = builder.storage;

this.gpu = builder.gpu;

this.hasWifi = builder.hasWifi;

this.hasBluetooth = builder.hasBluetooth;

}

public String toString() {

return "Computer [CPU=" + cpu + ", RAM=" + ram + ", Storage=" + storage +

", GPU=" + gpu + ", WiFi=" + hasWifi + ", Bluetooth=" + hasBluetooth + "]";

}

public static class Builder {

private String cpu;

private String ram;

private String storage;

private String gpu;

private boolean hasWifi;

private boolean hasBluetooth;

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 setGPU(String gpu) {

this.gpu = gpu;

return this;

}

public Builder setWiFi(boolean hasWifi) {

this.hasWifi = hasWifi;

return this;

}

public Builder setBluetooth(boolean hasBluetooth) {

this.hasBluetooth = hasBluetooth;

return this;

}

public Computer build() {

return new Computer(this);

}

}}

public class Main {

public static void main(String[] args) {

Computer gamingRig = new Computer.Builder()

.setCPU("Intel Core i9")

.setRAM("32GB")

.setStorage("1TB SSD")

.setGPU("NVIDIA RTX 4080")

.setWiFi(true)

.setBluetooth(true)

.build();

Computer officePC = new Computer.Builder()

.setCPU("Intel Core i5")

.setRAM("16GB")

.setStorage("512GB SSD")

.setWiFi(true)

.setBluetooth(false)

.build();

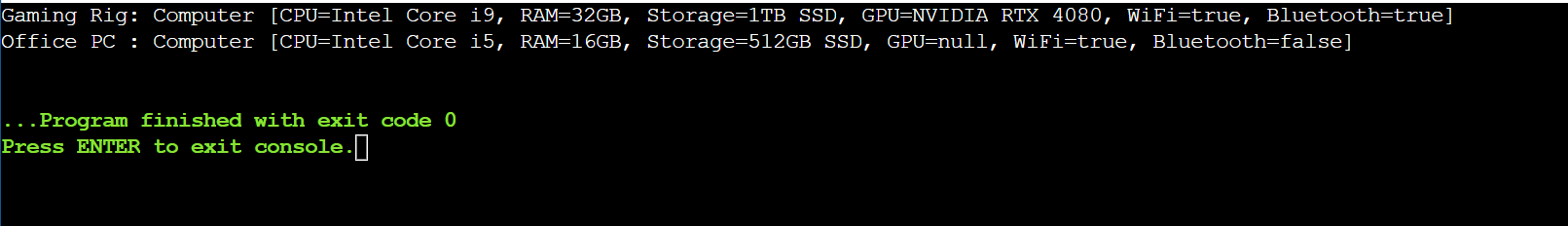
System.out.println("Gaming Rig: " + gamingRig);

System.out.println("Office PC : " + officePC);

}

}

Output:



**Exercise 4: Implementing the Adapter Pattern**

Code:

interface PaymentProcessor {

void processPayment(double amount);

}

class PayPalGateway {

public void makePayPalPayment(double amount) {

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

}

}

class StripeGateway {

public void makeStripeCharge(double amountInINR) {

System.out.println("Charged ₹" + amountInINR + " using Stripe.");

}

}

class RazorpayGateway {

public void doRazorpayTransaction(double amount) {

System.out.println("Razorpay processed ₹" + amount);

}

}

class PayPalAdapter implements PaymentProcessor {

private PayPalGateway payPal;

public PayPalAdapter(PayPalGateway payPal) {

this.payPal = payPal;

}

public void processPayment(double amount) {

payPal.makePayPalPayment(amount);

}

}

class StripeAdapter implements PaymentProcessor {

private StripeGateway stripe;

public StripeAdapter(StripeGateway stripe) {

this.stripe = stripe;

}

public void processPayment(double amount) {

stripe.makeStripeCharge(amount);

}

}

class RazorpayAdapter implements PaymentProcessor {

private RazorpayGateway razorpay;

public RazorpayAdapter(RazorpayGateway razorpay) {

this.razorpay = razorpay;

}

public void processPayment(double amount) {

razorpay.doRazorpayTransaction(amount);

}

}

public class Main {

public static void main(String[] args) {

PaymentProcessor paypalProcessor = new PayPalAdapter(new PayPalGateway());

PaymentProcessor stripeProcessor = new StripeAdapter(new StripeGateway());

PaymentProcessor razorpayProcessor = new RazorpayAdapter(new RazorpayGateway());

paypalProcessor.processPayment(1500.00);

stripeProcessor.processPayment(2300.00);

razorpayProcessor.processPayment(500.00);

}

}

Output:



**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 wrappedNotifier;

public NotifierDecorator(Notifier notifier) {

this.wrappedNotifier = notifier;

}

public void send(String message) {

wrappedNotifier.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: " + message);

}

}

public class Main {

public static void main(String[] args) {

Notifier baseNotifier = new EmailNotifier();

Notifier multiChannelNotifier = new SMSNotifierDecorator(

new SlackNotifierDecorator(

baseNotifier

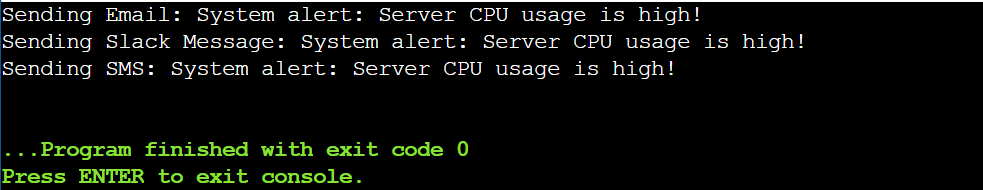
));

multiChannelNotifier.send("System alert: Server CPU usage is high!");

}

}

Output:



**Exercise 6: Implementing the Proxy Pattern**

Code:

interface Image {

void display();

}

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);

}

}

class ProxyImage implements Image {

private String filename;

private RealImage realImage;

public ProxyImage(String filename) {

this.filename = filename;

}

public void display() {

if (realImage == null) {

realImage = new RealImage(filename);

} else {

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

}

realImage.display();

}

}

public class Main {

public static void main(String[] args) {

Image img1 = new ProxyImage("nature\_photo\_01.jpg");

Image img2 = new ProxyImage("cityscape\_2025.png");

img1.display();

img1.display();

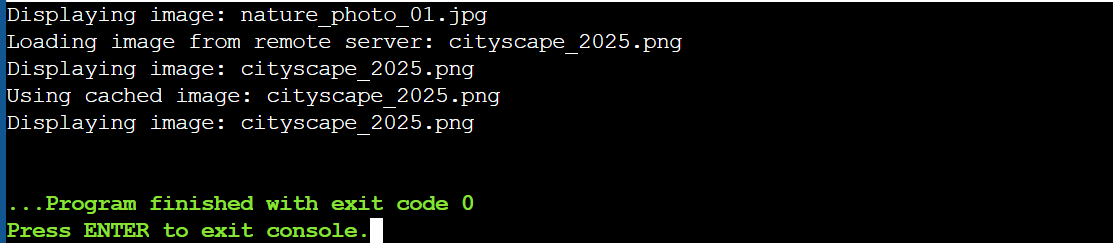
img2.display();

img2.display();

}

}

Output:



**Exercise 7: Implementing the Observer Pattern**

Code:

import java.util.ArrayList;

import java.util.List;

interface Stock {

void registerObserver(Observer o);

void removeObserver(Observer o);

void notifyObservers(double price);}

interface Observer {

void update(double price);

}

class StockMarket implements Stock {

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

private double stockPrice;

public void setStockPrice(double newPrice) {

this.stockPrice = newPrice;

notifyObservers(newPrice);

}

public void registerObserver(Observer o) {

observers.add(o);

}

public void removeObserver(Observer o) {

observers.remove(o);

}

public void notifyObservers(double price) {

for (Observer o : observers) {

o.update(price);

}

}

}

class MobileApp implements Observer {

private String user;

public MobileApp(String user) {

this.user = user;

}

public void update(double price) {

System.out.println("MobileApp [" + user + "]: Stock price updated to ₹" + price);

}

}

class WebApp implements Observer {

private String user;

public WebApp(String user) {

this.user = user;

}

public void update(double price) {

System.out.println("WebApp [" + user + "]: Stock price updated to ₹" + price);

}

}

public class Main {

public static void main(String[] args) {

StockMarket stockMarket = new StockMarket();

Observer mobileUser = new MobileApp("User1");

Observer webUser = new WebApp("User2");

stockMarket.registerObserver(mobileUser);

stockMarket.registerObserver(webUser);

stockMarket.setStockPrice(1450.75);

stockMarket.setStockPrice(1520.30);

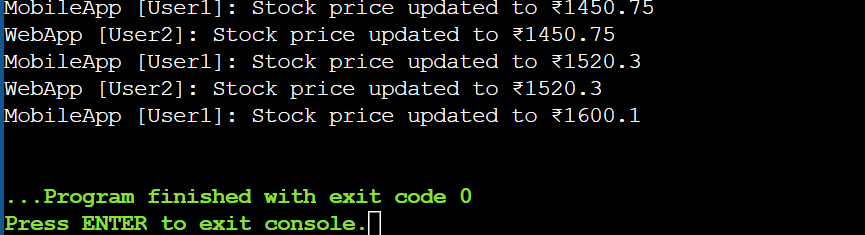
stockMarket.removeObserver(webUser);

stockMarket.setStockPrice(1600.10);

}

}

Output:



**Exercise 8: Implementing the Strategy Pattern**

Code:

interface PaymentStrategy {

void pay(double amount);

}

class CreditCardPayment implements PaymentStrategy {

public void pay(double amount) {

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

}

}

class PayPalPayment implements PaymentStrategy {

public void pay(double amount) {

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

}

}

class UpiPayment implements PaymentStrategy {

public void pay(double amount) {

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

}

}

class PaymentContext {

private PaymentStrategy paymentStrategy;

public void setPaymentStrategy(PaymentStrategy paymentStrategy) {

this.paymentStrategy = paymentStrategy;

}

public void processPayment(double amount) {

if (paymentStrategy == null) {

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

} else {

paymentStrategy.pay(amount);

}

}

}

public class Main {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

context.setPaymentStrategy(new CreditCardPayment());

context.processPayment(1000.00);

context.setPaymentStrategy(new PayPalPayment());

context.processPayment(750.50);

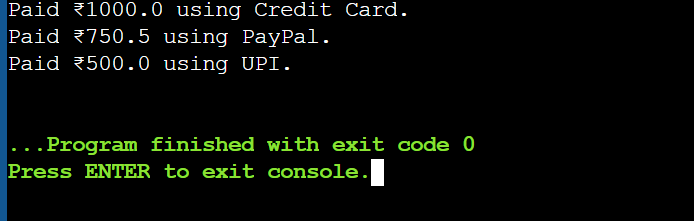
context.setPaymentStrategy(new UpiPayment());

context.processPayment(500.00);

}

}

Output:



**Exercise 9: Implementing the Command Pattern**

Code:

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() {

if (command != null) {

command.execute();

} else {

System.out.println("No command set.");

}

}

}

public class Main {

public static void main(String[] args) {

Light livingRoomLight = new Light();

Command lightsOn = new LightOnCommand(livingRoomLight);

Command lightsOff = new LightOffCommand(livingRoomLight);

RemoteControl remote = new RemoteControl();

remote.setCommand(lightsOn);

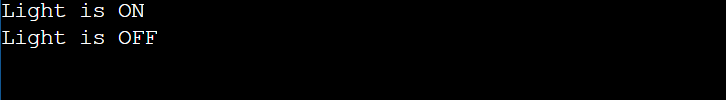
remote.pressButton();

remote.setCommand(lightsOff);

remote.pressButton();

}

}

Output: 

**Exercise 10: Implementing the MVC Pattern**

Code:

class Student {

private String name;

private String id;

private 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; }

}

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);

}

}

class StudentController {

private Student student;

private StudentView view;

public StudentController(Student student, StudentView view) {

this.student = student;

this.view = view;

}

public void setStudentName(String name) {

student.setName(name);

}

public String getStudentName() {

return student.getName();

}

public void setStudentId(String id) {

student.setId(id);

}

public String getStudentId() {

return student.getId();

} public void setStudentGrade(String grade) {

student.setGrade(grade);

}

public String getStudentGrade() {

return student.getGrade();

}

public void updateView() {

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

}

}

public class Main {

public static void main(String[] args) {

Student student = new Student("Priya", "S102", "A");

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

controller.updateView();

controller.setStudentName("user1");

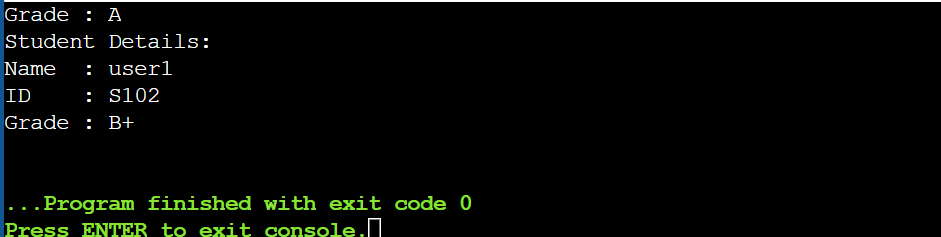
controller.setStudentGrade("B+");

controller.updateView();

}

}

Output:



**Exercise 11: Implementing Dependency Injection**

Code:

interface CustomerRepository {

String findCustomerById(String id);

}class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(String id) {

return "Customer ID: " + id + ", Name: user1, Status: Active";

}

}

class CustomerService {

private CustomerRepository repository;

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void getCustomerDetails(String id) {

String customer = repository.findCustomerById(id);

System.out.println(customer);

}

}

public class Main {

public static void main(String[] args) {

CustomerRepository repo = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repo);

service.getCustomerDetails("C1");

}

}

Output:

