**Exercise 1: Implementing the Singleton Pattern**

**Logger.java**

public class Logger {

    private static Logger instance;

    private Logger() {

    }

    public static Logger getInstance(String userName) {

        if (instance == null) {

            instance = new Logger();

        }

        return instance;

    }

    public void logDetails(String msg) {

        System.out.println("Log: " + msg);

    }

}

**Test.java**

public class Test {

    public static void main(String[] args) {

        Logger log1 = Logger.getInstance("Alice");

        Logger log2 = Logger.getInstance("Bob");

        if (log1.hashCode() == log2.hashCode()) {

            log1.logDetails("Hello");

            log2.logDetails("World");

        }

    }

}

**Output:**



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

**Document.java**

package FactoryMethodPattern;

public interface Document {

    void open();

    void close();

}

**ExcelDocument.java**

package FactoryMethodPattern;

public class ExcelDocument implements Document {

    @Override

    public void open() {

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

    }

    @Override

    public void close() {

        System.out.println("Closing Excel Document");

    }

}

**PdfDocument.java**

package FactoryMethodPattern;

public class PdfDocument implements Document {

    @Override

    public void open() {

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

    }

    @Override

    public void close() {

        System.out.println("Closing PDF Document");

    }

}

**WordDocument.java**

package FactoryMethodPattern;

public class WordDocument implements Document {

    @Override

    public void open() {

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

    }

    @Override

    public void close() {

        System.out.println("Closing Word Document");

    }

}

**DocumentFactory.java**

package FactoryMethodPattern;

public abstract class DocumentFactory {

    public abstract Document createDocument();

}

**ExcelFactory.java**

package FactoryMethodPattern;

public class ExcelFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new ExcelDocument();

    }

}

**PdfFactory.java**

package FactoryMethodPattern;

public class PdfFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new PdfDocument();

    }

}

**WordFactory.java**

package FactoryMethodPattern;

public class WordFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new WordDocument();

    }

}

**Main.java**

import java.util.Scanner;

import FactoryMethod.\*;

public class Main {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.println("Enter the type of document to create (Word/Excel/Pdf): ");

        String docType = sc.nextLine();

        DocumentFactory factory;

        switch (docType.toLowerCase()) {

            case "word":

                factory = new WordFactory();

                break;

            case "excel":

                factory = new ExcelFactory();

                break;

            case "pdf":

                factory = new PdfFactory();

                break;

            default:

                System.out.println("Invalid document type.");

                return;

        }

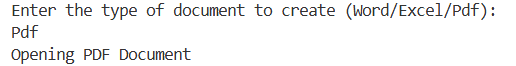
        Document document = factory.createDocument();

        document.open();

    }

}

**Output**

****

**Exercise 3: Implementing the Builder Pattern**

**Computer.java**

public class Computer{

    private final String cpu;

    private final String ram;

    private final String storage;

    private final String gpu;

    public Computer(Builder builder){

        this.cpu = builder.cpu;

        this.ram = builder.ram;

        this.storage = builder.storage;

        this.gpu = builder.gpu;

    }

    public static class Builder{

        private String cpu;

        private String ram;

        private String storage;

        private String gpu;

        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 Computer build(){

            return new Computer(this);

        }

    }

    public String getCpu() {

        return cpu;

    }

    public String getRam() {

        return ram;

    }

    public String getStorage() {

        return storage;

    }

    public String getGpu() {

        return gpu;

    }

}

**Test.java**

public class Test {

    public static void main(String[] args) {

        Computer computer = new Computer.Builder()

                .setCpu("Intel Core i9")

                .setRam("32GB")

                .setStorage("1TB SSD")

                .setGpu("NVIDIA RTX 3080")

                .build();

        System.out.println("Computer built with the following specifications:");

        System.out.println("CPU: " + computer.getCpu());

        System.out.println("RAM: " + computer.getRam());

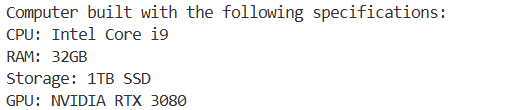
        System.out.println("Storage: " + computer.getStorage());

        System.out.println("GPU: " + computer.getGpu());

    }

}

**Output.java**

****

**Exercise 4: Implementing the Adapter Pattern**

**PaymentProcessor.java**

public interface PaymentProcessor {

    void processPayment(double amount);

    String getPaymentGateway();

}

**Gpay.java**

public class GPay {

    public void makePayment(double amount) {

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

    }

}

**Paypal.java**

public class PayPal {

    public void makePayment(double amount) {

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

    }

}

**GPayAdapter.java**

public class GpayAdapter implements PaymentProcessor {

    private final GPay gPay;

    public GpayAdapter(GPay gPay) {

        this.gPay = gPay;

    }

    @Override

    public void processPayment(double amount) {

        gPay.makePayment(amount);

    }

    @Override

    public String getPaymentGateway() {

        return "GPay";

    }

}

**PayPalAdapter.java**

public class PayPalAdapter implements PaymentProcessor {

    private final PayPal payPal;

    public PayPalAdapter(PayPal payPal) {

        this.payPal = payPal;

    }

    public void processPayment(double amount) {

        payPal.makePayment(amount);

    }

    public String getPaymentGateway() {

        return "PayPal";

    }

}

**Test.java**

public class Test {

    public static void main(String[] args) {

        PaymentProcessor paymentProcessor = new GpayAdapter(new GPay());

        paymentProcessor.processPayment(100.0);

        System.out.println("Using payment gateway: " + paymentProcessor.getPaymentGateway());

        PaymentProcessor paymentProcessor2 = new PayPalAdapter(new PayPal());

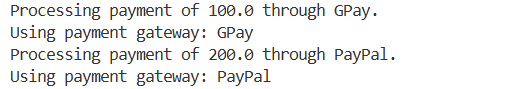
        paymentProcessor2.processPayment(200.0);

        System.out.println("Using payment gateway: " + paymentProcessor2.getPaymentGateway());

    }

}

**Output**

****

**Exercise 5: Implementing the Decorator Pattern**

**Notifier.java**

public interface Notifier {

    void send(String message);

}

**NotifierDecorator.java**

public 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**

public class SMSNotifierDecorator extends NotifierDecorator {

    public SMSNotifierDecorator(Notifier notifier) {

        super(notifier);

    }

    public void send(String message) {

        super.send(message);

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

    }

}

**SlackNotifier.java**

public 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);

    }

}

**Test.java**

public class Test {

        public static void main(String[] args) {

        Notifier base = new EmailNotifier();

        Notifier sms = new SMSNotifierDecorator(base);

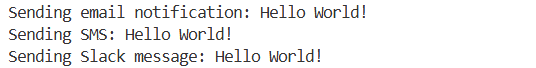
        Notifier slack = new SlackNotifierDecorator(sms);

        slack.send("Hello World!");

    }

}

**Output**

****

**Exercise 6: Implementing the Proxy Pattern**

**Image.java**

package com.mycompany.proxypattern;

public interface Image {

    void display();

}

**ProxyImage.java**

package com.mycompany.proxypattern;

public class ProxyImage implements Image {

    private String filename;

    private RealImage realImage;

    public ProxyImage(String filename) {

        this.filename = filename;

    }

    @Override

    public void display() {

        if (realImage == null) {

            realImage = new RealImage(filename);

        } else {

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

        }

        realImage.display();

    }

}

**RealImage.java**

package com.mycompany.proxypattern;

public 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);

        try {

            Thread.sleep(2000);

        } catch (InterruptedException e) {

            e.printStackTrace();

        }

    }

    @Override

    public void display() {

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

    }

}

**ProxyPattern.java**

package com.mycompany.proxypattern;

public class ProxyPattern {

    public static void main(String[] args) {

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

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

        img1.display();

        System.out.println();

        img1.display();

        System.out.println();

        img2.display();

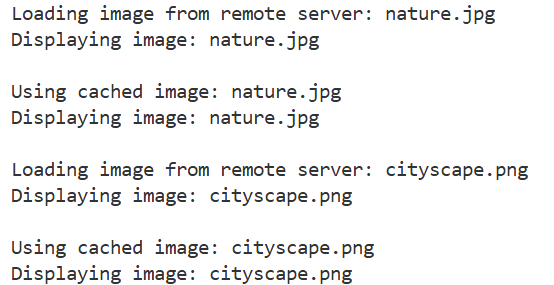
        System.out.println();

        img2.display();

    }

}

**Output**



**Exercise 7: Implementing the Observer Pattern**

**Observer.java**

package com.mycompany.observerpattern;

public interface Observer {

    void update(String stockName, double stockPrice);

}

**MobileApp.java**

package com.mycompany.observerpattern;

public class MobileApp implements Observer {

    private String user;

    public MobileApp(String user) {

        this.user = user;

    }

    @Override

    public void update(String stockName, double stockPrice) {

        System.out.println(" MobileApp [" + user + "] - " + stockName + " updated to ₹" + stockPrice);

    }

}

**Stock.java**

package com.mycompany.observerpattern;

public interface Stock {

    void registerObserver(String stockName, Observer o);

    void removeObserver(String stockName, Observer o);

    void notifyObservers(String stockName, double price);

}

**StockMarket.java**

package com.mycompany.observerpattern;

import java.util.\*;

public class StockMarket implements Stock {

    private Map<String, Double> stockPrices = new HashMap<>();

    private Map<String, List<Observer>> stockObservers = new HashMap<>();

    public void setStockPrice(String stockName, double price) {

        stockPrices.put(stockName, price);

        System.out.println("\n[Update] " + stockName + " = ₹" + price);

        notifyObservers(stockName, price);

    }

    @Override

    public void registerObserver(String stockName, Observer o) {

        stockObservers.putIfAbsent(stockName, new ArrayList<>());

        stockObservers.get(stockName).add(o);

    }

    @Override

    public void removeObserver(String stockName, Observer o) {

        if (stockObservers.containsKey(stockName)) {

            stockObservers.get(stockName).remove(o);

        }

    }

    @Override

    public void notifyObservers(String stockName, double price) {

        if (stockObservers.containsKey(stockName)) {

            for (Observer o : stockObservers.get(stockName)) {

                o.update(stockName, price);

            }

        }

    }

}

**WebApp.java**

package com.mycompany.observerpattern;

public class WebApp implements Observer {

    private String user;

    public WebApp(String user) {

        this.user = user;

    }

    @Override

    public void update(String stockName, double stockPrice) {

        System.out.println(" WebApp [" + user + "] - " + stockName + " updated to ₹" + stockPrice);

    }

}

**ObserverPattern.java**

package com.mycompany.observerpattern;

public class ObserverPattern {

    public static void main(String[] args) {

        StockMarket market = new StockMarket();

        Observer mobileAlice = new MobileApp("Alice");

        Observer webBoris = new WebApp("Boris");

        Observer mobileCyrus = new MobileApp("Cyrus");

        market.registerObserver("TCS", mobileAlice);

        market.registerObserver("TCS", webBoris);

        market.registerObserver("INFY", mobileCyrus);

        market.registerObserver("WIPRO", webBoris);

        market.registerObserver("WIPRO", mobileAlice);

        market.setStockPrice("TCS", 3500);

        market.setStockPrice("INFY", 1480.5);

        market.setStockPrice("WIPRO", 410.75);

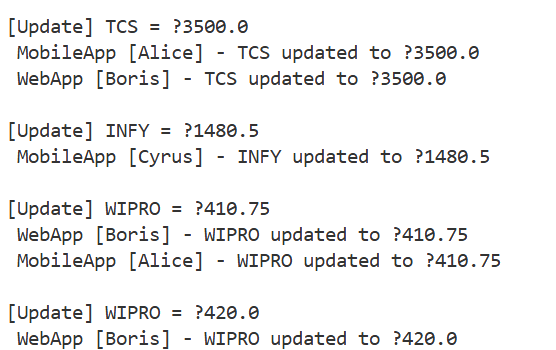
        market.removeObserver("WIPRO", mobileAlice);

        market.setStockPrice("WIPRO", 420.00);

    }

}

**Output**



**Exercise 8: Implementing the Strategy Pattern**

**PaymentStrategy.java**

package com.mycompany.strategypattern;

public interface PaymentStrategy {

    void pay(double amount);

}

**CreditCardPayment.java**

package com.mycompany.strategypattern;

public class CreditCardPayment implements PaymentStrategy {

    private String cardNumber;

    public CreditCardPayment(String cardNumber) {

        this.cardNumber = cardNumber;

    }

    @Override

    public void pay(double amount) {

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

    }

}

**PayPalPayment.java**

package com.mycompany.strategypattern;

public class PayPalPayment implements PaymentStrategy {

    private String email;

    public PayPalPayment(String email) {

        this.email = email;

    }

    @Override

    public void pay(double amount) {

        System.out.println("Paid Rs. " + amount + " using PayPal account: " + email);

    }

}

**UpiPayment.java**

package com.mycompany.strategypattern;

public class UpiPayment implements PaymentStrategy {

    private String upiId;

    public UpiPayment(String upiId) {

        this.upiId = upiId;

    }

    @Override

    public void pay(double amount) {

        System.out.println("Paid Rs. " + amount + " using UPI: " + upiId);

    }

}

**PaymentContext.java**

package com.mycompany.strategypattern;

public class PaymentContext {

    private PaymentStrategy strategy;

    public void setStrategy(PaymentStrategy strategy) {

        this.strategy = strategy;

    }

    public void executePayment(double amount) {

        if (strategy == null) {

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

        } else {

            strategy.pay(amount);

        }

    }

}

**StrategyPattern.java**

package com.mycompany.strategypattern;

import java.util.Scanner;

public class StrategyPattern {

 public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        PaymentContext context = new PaymentContext();

        while (true) {

            System.out.println("\n--- Select Payment Method ---");

            System.out.println("1. Credit Card");

            System.out.println("2. PayPal");

            System.out.println("3. UPI");

            System.out.println("4. Exit");

            System.out.print("Enter choice: ");

            int choice = scanner.nextInt();

            if (choice == 4) break;

            System.out.print("Enter amount: ₹");

            double amount = scanner.nextDouble();

            switch (choice) {

                case 1:

                    System.out.print("Enter card number: ");

                    String card = scanner.next();

                    context.setStrategy(new CreditCardPayment(card));

                    break;

                case 2:

                    System.out.print("Enter PayPal email: ");

                    String email = scanner.next();

                    context.setStrategy(new PayPalPayment(email));

                    break;

                case 3:

                    System.out.print("Enter UPI ID: ");

                    String upi = scanner.next();

                    context.setStrategy(new UpiPayment(upi));

                    break;

                default:

                    System.out.println("Invalid choice.");

                    continue;

            }

            context.executePayment(amount);

        }

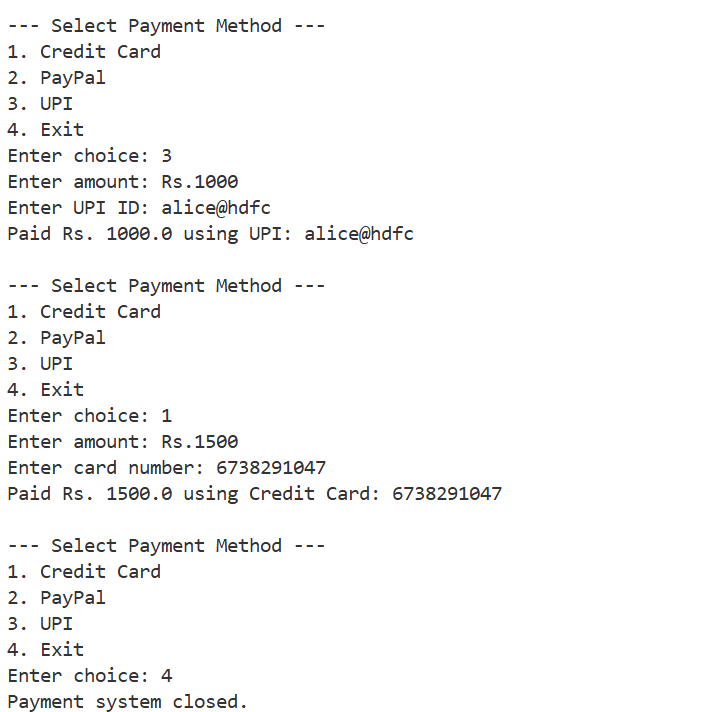
        scanner.close();

        System.out.println("Payment system closed.");

    }

}

**Output**



**Exercise 9: Implementing the Command Pattern**

**Command.java**

package com.mycompany.commandpattern;

public interface Command {

void execute();

}

**Fan.java**

package com.mycompany.commandpattern;

public class Fan {

public void turnOn() {

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

}

public void turnOff() {

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

}

}

**FanOffCommand.java**

package com.mycompany.commandpattern;

public class FanOffCommand implements Command {

private Fan fan;

public FanOffCommand(Fan fan) {

this.fan = fan;

}

public void execute() {

fan.turnOff();

}

}

**FanOnCommand.java**

package com.mycompany.commandpattern;

public class FanOnCommand implements Command {

private Fan fan;

public FanOnCommand(Fan fan) {

this.fan = fan;

}

public void execute() {

fan.turnOn();

}

}

**Light.java**

package com.mycompany.commandpattern;

public class Light {

public void turnOn() {

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

}

public void turnOff() {

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

}

}

**LightOffCommand.java**

package com.mycompany.commandpattern;

public class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOff();

}

}

**LightOnCommand.java**

package com.mycompany.commandpattern;

public class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOn();

}

}

**RemoteControl.java**

package com.mycompany.commandpattern;

public class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

command.execute();

}

}

**CommandPattern.java**

package com.mycompany.commandpattern;

import java.util.Scanner;

public class CommandPattern {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

RemoteControl remote = new RemoteControl();

Light light = new Light();

Fan fan = new Fan();

Command lightOn = new LightOnCommand(light);

Command lightOff = new LightOffCommand(light);

Command fanOn = new FanOnCommand(fan);

Command fanOff = new FanOffCommand(fan);

while (true) {

System.out.println("\n--- Home Automation ---");

System.out.println("1. Turn ON Light");

System.out.println("2. Turn OFF Light");

System.out.println("3. Turn ON Fan");

System.out.println("4. Turn OFF Fan");

System.out.println("5. Exit");

System.out.print("Choose: ");

int choice = sc.nextInt();

switch (choice) {

case 1:

remote.setCommand(lightOn);

break;

case 2:

remote.setCommand(lightOff);

break;

case 3:

remote.setCommand(fanOn);

break;

case 4:

remote.setCommand(fanOff);

break;

case 5:

System.out.println("System Shut Down.");

sc.close();

return;

default:

System.out.println("Invalid choice.");

continue;

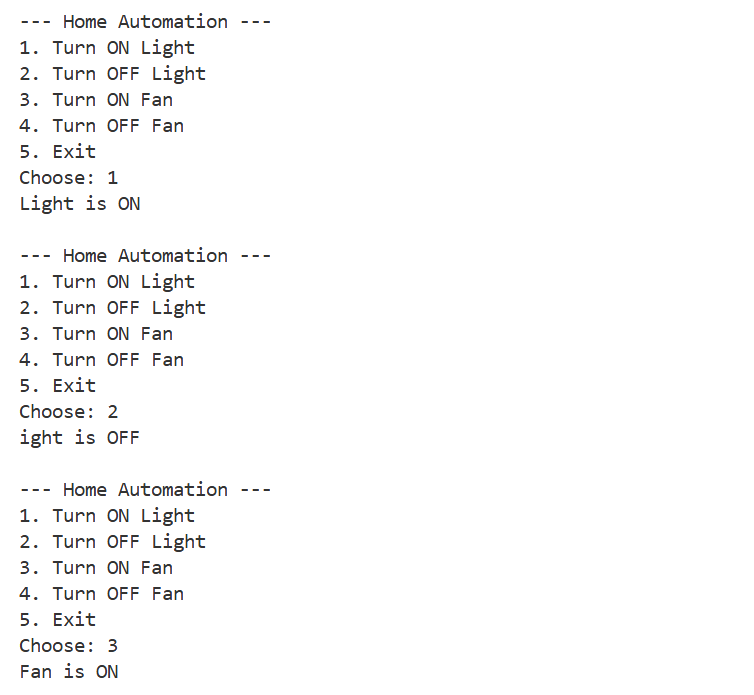
}

remote.pressButton();

}

}

}

**Output**

**Exercise 10: Implementing the MVC Pattern**

**Student.java**

package com.mycompany.mvcpattern;

public 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 String getId() { return id; }

    public String getGrade() { return grade; }

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

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

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

}

**StudentController.java**

package com.mycompany.mvcpattern;

public 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.getName(), model.getId(), model.getGrade());

    }

}

**StudentView.java**

package com.mycompany.mvcpattern;

public class StudentView {

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

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

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

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

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

    }

}

**MVCPattern.java**

package com.mycompany.mvcpattern;

import java.util.Scanner;

public class MVCPattern {

   public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.print("Enter student name: ");

        String name = sc.nextLine();

        System.out.print("Enter student ID: ");

        String id = sc.nextLine();

        System.out.print("Enter student grade: ");

        String grade = sc.nextLine();

        Student student = new Student(name, id, grade);

        StudentView view = new StudentView();

        StudentController controller = new StudentController(student, view);

        controller.updateView();

        System.out.println("\nUpdate student info? (yes/no): ");

        String choice = sc.nextLine().toLowerCase();

        if (choice.equals("yes")) {

            System.out.print("New name: ");

            controller.setStudentName(sc.nextLine());

            System.out.print("New ID: ");

            controller.setStudentId(sc.nextLine());

            System.out.print("New grade: ");

            controller.setStudentGrade(sc.nextLine());

            System.out.println("\nUpdated student details:");

            controller.updateView();

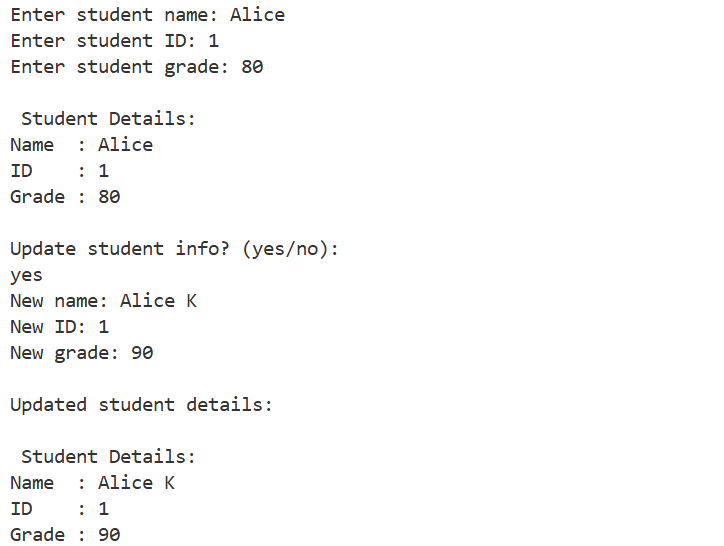
        }

        sc.close();

    }

}

**Output**

****

**Exercise 11: Implementing Dependency Injection**

**CustomerRepository.java**

package com.mycompany.dependencyinjection;

public interface CustomerRepository {

    String findCustomerById(int id);

}

**CustomerRepositoryImpl**.**java**

package com.mycompany.dependencyinjection;

import java.util.HashMap;

import java.util.Map;

import java.util.Scanner;

public class CustomerRepositoryImpl implements CustomerRepository {

    private Map<Integer, String> customerData = new HashMap<>();

    public void loadCustomers() {

        Scanner sc = new Scanner(System.in);

        System.out.print("Enter number of customers: ");

        int n = Integer.parseInt(sc.nextLine());

        for (int i = 0; i < n; i++) {

            System.out.print("Enter Customer ID: ");

            int id = Integer.parseInt(sc.nextLine());

            System.out.print("Enter Customer Name: ");

            String name = sc.nextLine();

            customerData.put(id, name);

        }

    }

    @Override

    public String findCustomerById(int id) {

        if (customerData.containsKey(id)) {

            return "Customer: " + customerData.get(id);

        }

        return "Customer not found.";

    }

}

**CustomerService.java**

package com.mycompany.dependencyinjection;

public class CustomerService {

    private CustomerRepository customerRepository;

    public CustomerService(CustomerRepository customerRepository) {

        this.customerRepository = customerRepository;

    }

    public void displayCustomerById(int id) {

    System.out.println(customerRepository.findCustomerById(id));

    }

}

**DependencyInjection.java**

package com.mycompany.dependencyinjection;

import java.util.Scanner;

public class DependencyInjection {

public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        CustomerRepositoryImpl repository = new CustomerRepositoryImpl();

        repository.loadCustomers();

        CustomerService service = new CustomerService(repository);

        while (true) {

            System.out.print("\nEnter Customer ID (0 to exit): ");

            int id = sc.nextInt();

            if (id == 0) {

                System.out.println("Exiting");

                break;

            }

            service.displayCustomerById(id);

        }

        sc.close();

    }

}

**Output**

