**DESIGN PATTERNS AND PRINCIPLES**

**Ex 1: Implementing the Singleton Pattern**

**Scenario:**

You need to ensure that a logging utility class in your application has only one instance throughout the application lifecycle to ensure consistent logging.

**Code:**

**class** Log {

**private** **static** **Log** instance;

**private** Log() {

        System.out.println("Logger initialized");

    }

**public** **static** **Log** getInstance() {

**if** (instance **==** **null**) {

            instance **=** **new** Log();

        }

**return** instance;

    }

**public** **void** log(**String** message) {

        System.out.println("Log: " **+** message);

    }

}

**public** **class** SIngletonPattern {

**public** **static** **void** main(**String**[] args) {

**Log** logger1 **=** Log.getInstance();

**Log** logger2 **=** Log.getInstance();

        logger1.log("This is the first log message.");

        logger2.log("This is the second log message.");

**if** (logger1 **==** logger2) {

            System.out.println("Both logger instances are the same.");

        } **else** {

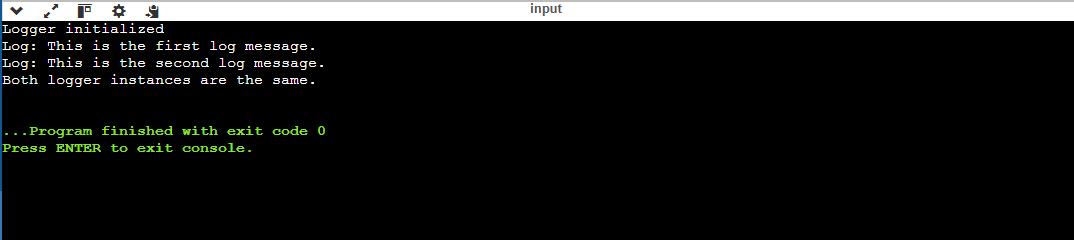
            System.out.println("Different logger instances exist (should not happen).");

        }

    }

}

**Output:**

****

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

**Scenario:**

You are developing a document management system that needs to create different types of documents (e.g., Word, PDF, Excel). Use the Factory Method Pattern to achieve this.

**Code:**

**interface** Document {

**void** open();

}

**class** WordDocument **implements** Document {

**public** **void** open() {

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

    }

}

**class** PdfDocument **implements** Document {

**public** **void** open() {

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

    }

}

**class** ExcelDocument **implements** Document {

**public** **void** open() {

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

    }

}

**class** DocumentFactory {

**public** **static** **Document** createDocument(**String** type) {

**switch** (type.toLowerCase()) {

**case** "word"**:** **return** **new** WordDocument();

**case** "pdf"**:** **return** **new** PdfDocument();

**case** "excel"**:** **return** **new** ExcelDocument();

**default:** **return** **null**;

        }

    }

}

**public** **class** FactoryPattern {

**public** **static** **void** main(**String**[] args) {

**Document** doc1 **=** DocumentFactory.createDocument("word");

**Document** doc2 **=** DocumentFactory.createDocument("pdf");

**Document** doc3 **=** DocumentFactory.createDocument("excel");

**if** (doc1 **!=** **null**) doc1.open();

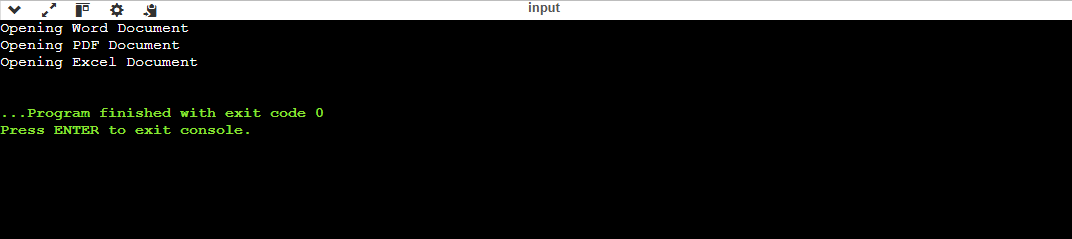
**if** (doc2 **!=** **null**) doc2.open();

**if** (doc3 **!=** **null**) doc3.open();

    }

}

**Output:**

****

**Ex 3: Implementing the Builder Pattern**

**Scenario:**

You are developing a system to create complex objects such as a Computer with multiple optional parts. Use the Builder Pattern to manage the construction process.

**Code:**

**class** Computer {

**private** **String** CPU;

**private** **String** RAM;

**private** **String** storage;

**private** Computer(**Builder** builder) {

        this.CPU **=** builder.CPU;

        this.RAM **=** builder.RAM;

        this.storage **=** builder.storage;

    }

**public** **void** showSpecs() {

        System.out.println("CPU: " **+** CPU **+** ", RAM: " **+** RAM **+** ", Storage: " **+** storage);

    }

**static** **class** Builder {

**private** **String** CPU;

**private** **String** RAM;

**private** **String** storage;

**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** **Computer** build() {

**return** **new** Computer(this);

        }

    }

}

**public** **class** Builderpattern {

**public** **static** **void** main(**String**[] args) {

**Computer** gamingPC **=** **new** Computer.Builder()

                .setCPU("Intel i9")

                .setRAM("32GB")

                .setStorage("1TB SSD")

                .build();

**Computer** officePC **=** **new** Computer.Builder()

                .setCPU("Intel i5")

                .setRAM("16GB")

                .setStorage("512GB SSD")

                .build();

        System.out.println("Gaming PC Specs:");

        gamingPC.showSpecs();

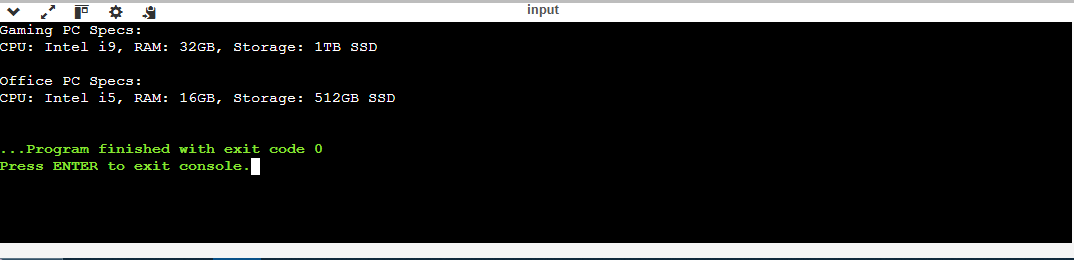
        System.out.println("\nOffice PC Specs:");

        officePC.showSpecs();

    }

}

**Output:**

****

**Ex 4: Implementing the Adapter Pattern**

**Scenario:**

You are developing a payment processing system that needs to integrate with multiple third-party payment gateways with different interfaces. Use the Adapter Pattern to achieve this.

**Code:**

**interface** PaymentProcessor {

**void** processPayment(**double** amount);

}

**class** PayPalGateway {

**public** **void** sendPayment(**double** amount) {

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

    }

}

**class** CreditCardGateway {

**public** **void** makeTransaction(**double** amount) {

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

    }

}

**class** PayPalAdapter **implements** PaymentProcessor {

**private** **PayPalGateway** gateway **=** **new** PayPalGateway();

**public** **void** processPayment(**double** amount) {

        gateway.sendPayment(amount);

    }

}

**class** CreditCardAdapter **implements** PaymentProcessor {

**private** **CreditCardGateway** gateway **=** **new** CreditCardGateway();

**public** **void** processPayment(**double** amount) {

        gateway.makeTransaction(amount);

    }

}

**public** **class** AdapterPattern {

**public** **static** **void** main(**String**[] args) {

**PaymentProcessor** paypal **=** **new** PayPalAdapter();

**PaymentProcessor** creditCard **=** **new** CreditCardAdapter();

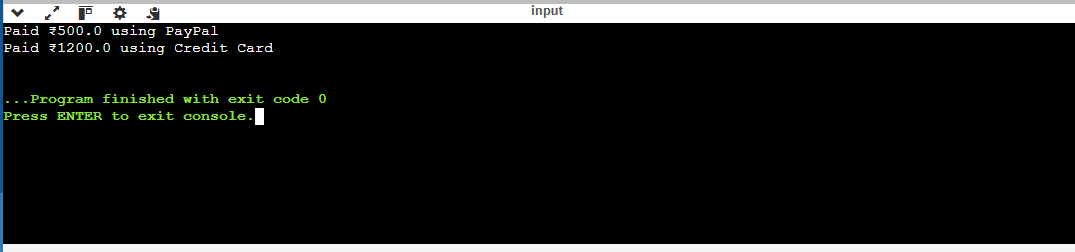
        paypal.processPayment(500.00);

        creditCard.processPayment(1200.00);

    }

}

**Output:**

****

**Ex 5: Implementing the Decorator Pattern**

**Scenario:**

You are developing a notification system where notifications can be sent via multiple channels (e.g., Email, SMS). Use the Decorator Pattern to add functionalities dynamically.

**Code:**

**interface** Notifier {

**void** send(**String** message);

}

**class** EmailNotifier **implements** Notifier {

**public** **void** send(**String** message) {

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

    }

}

**abstract** **class** NotifierDecorator **implements** Notifier {

**protected** **Notifier** notifier;

**public** NotifierDecorator(**Notifier** notifier) {

        this.notifier **=** notifier;

    }

}

**class** SMSNotifierDecorator **extends** NotifierDecorator {

**public** SMSNotifierDecorator(**Notifier** notifier) {

        super(notifier);

    }

**public** **void** send(**String** message) {

        notifier.send(message);

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

    }

}

**class** SlackNotifierDecorator **extends** NotifierDecorator {

**public** SlackNotifierDecorator(**Notifier** notifier) {

        super(notifier);

    }

**public** **void** send(**String** message) {

        notifier.send(message);

        System.out.println("Slack message sent: " **+** message);

    }

}

**public** **class** DecoratorPattern {

**public** **static** **void** main(**String**[] args) {

**Notifier** baseNotifier **=** **new** EmailNotifier();

**Notifier** smsNotifier **=** **new** SMSNotifierDecorator(baseNotifier);

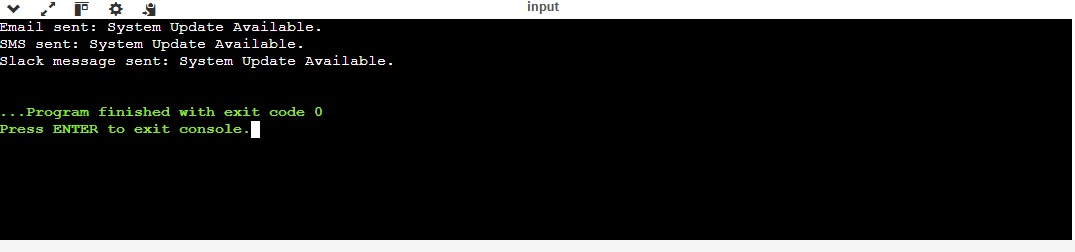
**Notifier** slackNotifier **=** **new** SlackNotifierDecorator(smsNotifier);

        slackNotifier.send("System Update Available.");

    }

}

**Output:**

****

**Ex 6: Implementing the Proxy Pattern**

**Scenario:**

You are developing an image viewer application that loads images from a remote server. Use the Proxy Pattern to add lazy initialization and caching.

**Code:**

**interface** Image {

**void** display();

}

**class** RealImage **implements** Image {

**private** **String** filename;

**public** RealImage(**String** filename) {

        this.filename **=** filename;

        loadFromDisk();

    }

**private** **void** loadFromDisk() {

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

    }

**public** **void** display() {

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

    }

}

**class** ProxyImage **implements** Image {

**private** **RealImage** realImage;

**private** **String** filename;

**public** ProxyImage(**String** filename) {

        this.filename **=** filename;

    }

**public** **void** display() {

**if** (realImage **==** **null**) {

            realImage **=** **new** RealImage(filename);

        }

        realImage.display();

    }

}

**public** **class** ProxyPattern {

**public** **static** **void** main(**String**[] args) {

**Image** image1 **=** **new** ProxyImage("nature.jpg");

**Image** image2 **=** **new** ProxyImage("car.png");

        image1.display(); *// loads and displays*

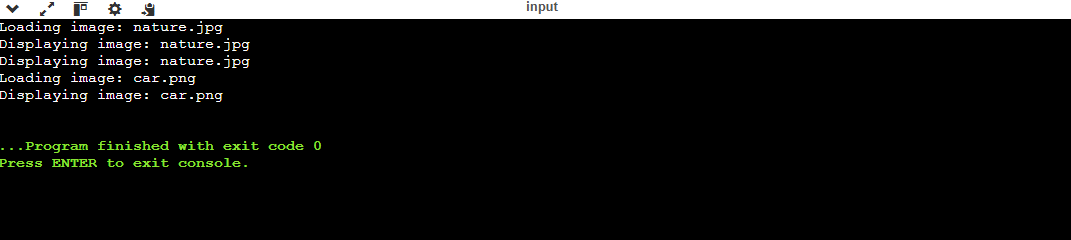
        image1.display(); *// only displays*

        image2.display();

    }

}

**Output:**



**Ex 7: Implementing the Observer Pattern**

**Scenario:**

You are developing a stock market monitoring application where multiple clients need to be notified whenever stock prices change. Use the Observer Pattern to achieve this.

**Code:**

**import** **java.util.\***;

**interface** Observer {

**void** update(**String** stockName, **double** price);

}

**class** MobileApp **implements** Observer {

**public** **void** update(**String** stockName, **double** price) {

        System.out.println("Mobile App - " **+** stockName **+** " is now ₹" **+** price);

    }

}

**class** WebApp **implements** Observer {

**public** **void** update(**String** stockName, **double** price) {

        System.out.println("Web App - " **+** stockName **+** " is now ₹" **+** price);

    }

}

**class** StockMarket {

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

**private** **String** stockName;

**private** **double** price;

**public** StockMarket(**String** stockName, **double** price) {

        this.stockName **=** stockName;

        this.price **=** price;

    }

**public** **void** registerObserver(**Observer** o) {

        observers.add(o);

    }

**public** **void** removeObserver(**Observer** o) {

        observers.remove(o);

    }

**public** **void** setPrice(**double** newPrice) {

        this.price **=** newPrice;

        notifyObservers();

    }

**private** **void** notifyObservers() {

**for** (**Observer** o **:** observers) {

            o.update(stockName, price);

        }

    }

}

**public** **class** ObserverPattern {

**public** **static** **void** main(**String**[] args) {

**StockMarket** stock **=** **new** StockMarket("TCS", 3500);

**Observer** mobile **=** **new** MobileApp();

**Observer** web **=** **new** WebApp();

        stock.registerObserver(mobile);

        stock.registerObserver(web);

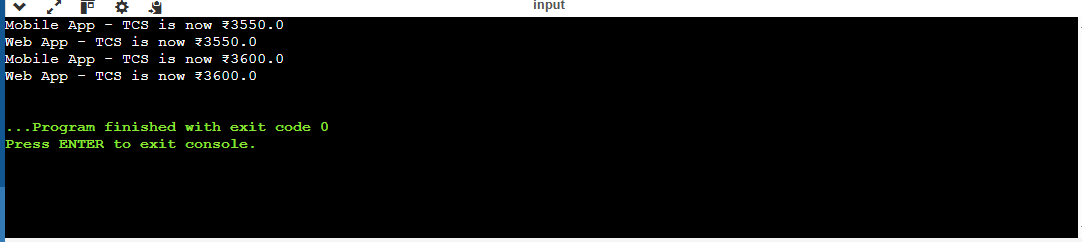
        stock.setPrice(3550);

        stock.setPrice(3600);

    }

}

**Output:**

****

**Ex 8: Implementing the Strategy Pattern**

**Scenario:**

You are developing a payment system where different payment methods (e.g., Credit Card, PayPal) can be selected at runtime. Use the Strategy Pattern to achieve this.

**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** PaymentContext {

**private** **PaymentStrategy** strategy;

**public** **void** setStrategy(**PaymentStrategy** strategy) {

        this.strategy **=** strategy;

    }

**public** **void** payAmount(**double** amount) {

**if** (strategy **!=** **null**) {

            strategy.pay(amount);

        } **else** {

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

        }

    }

}

**public** **class** StrategyPattern {

**public** **static** **void** main(**String**[] args) {

**PaymentContext** context **=** **new** PaymentContext();

        context.setStrategy(**new** CreditCardPayment());

        context.payAmount(1500);

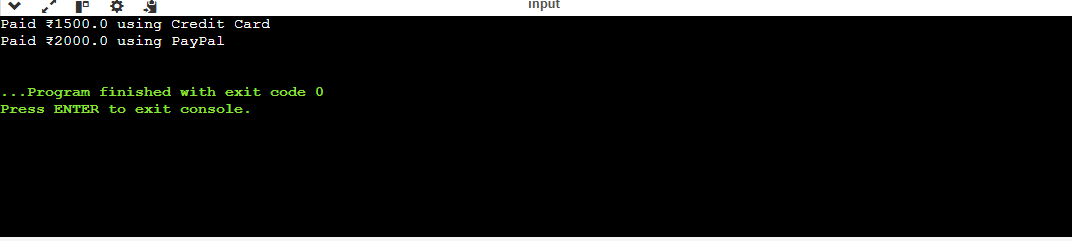
        context.setStrategy(**new** PayPalPayment());

        context.payAmount(2000);

    }

}

**Output:**

****

**Ex 9: Implementing the Command Pattern**

**Scenario:** You are developing a home automation system where commands can be issued to turn devices on or off. Use the Command Pattern to achieve this.

**Code:**

**interface** CommandPattern {

**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** CommandPattern {

**private** **Light** light;

**public** LightOnCommand(**Light** light) {

        this.light **=** light;

    }

**public** **void** execute() {

        light.turnOn();

    }

}

**class** LightOffCommand **implements** CommandPattern {

**private** **Light** light;

**public** LightOffCommand(**Light** light) {

        this.light **=** light;

    }

**public** **void** execute() {

        light.turnOff();

    }

}

**class** RemoteControl {

**private** **CommandPattern** command;

**public** **void** setCommand(**CommandPattern** command) {

        this.command **=** command;

    }

**public** **void** pressButton() {

**if** (command **!=** **null**) {

            command.execute();

        }

    }

}

**public** **class** CommandPat {

**public** **static** **void** main(**String**[] args) {

**Light** livingRoomLight **=** **new** Light();

**CommandPattern** onCommand **=** **new** LightOnCommand(livingRoomLight);

**CommandPattern** offCommand **=** **new** LightOffCommand(livingRoomLight);

**RemoteControl** remote **=** **new** RemoteControl();

        remote.setCommand(onCommand);

        remote.pressButton();

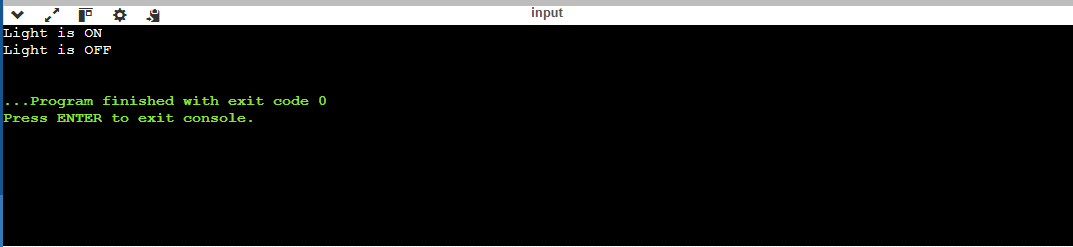
        remote.setCommand(offCommand);

        remote.pressButton();

    }

}

**Output:**



**Ex 10: Implementing the MVC Pattern**

**Scenario:**

You are developing a simple web application for managing student records using the MVC pattern.

**Code:**

**class** Student {

**private** **String** name;

**private** **int** id;

**private** **String** grade;

**public** Student(**String** name, **int** 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** **int** getId() { **return** id; }

**public** **String** getGrade() { **return** grade; }

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

}

**class** StudentView {

**public** **void** displayStudentDetails(**Student** student) {

        System.out.println("Student ID: " **+** student.getId());

        System.out.println("Name: " **+** student.getName());

        System.out.println("Grade: " **+** student.getGrade());

    }

}

**class** StudentController {

**private** **Student** student;

**private** **StudentView** view;

**public** StudentController(**Student** student, **StudentView** view) {

        this.student **=** student;

        this.view **=** view;

    }

**public** **void** updateView() {

        view.displayStudentDetails(student);

    }

**public** **void** setStudentName(**String** name) {

        student.setName(name);

    }

**public** **void** setStudentGrade(**String** grade) {

        student.setGrade(grade);

    }

}

**public** **class** MVCPattern {

**public** **static** **void** main(**String**[] args) {

**Student** student **=** **new** Student("Ravi", 101, "A");

**StudentView** view **=** **new** StudentView();

**StudentController** controller **=** **new** StudentController(student, view);

        controller.updateView();

        controller.setStudentName("Rakesh");

        controller.setStudentGrade("A+");

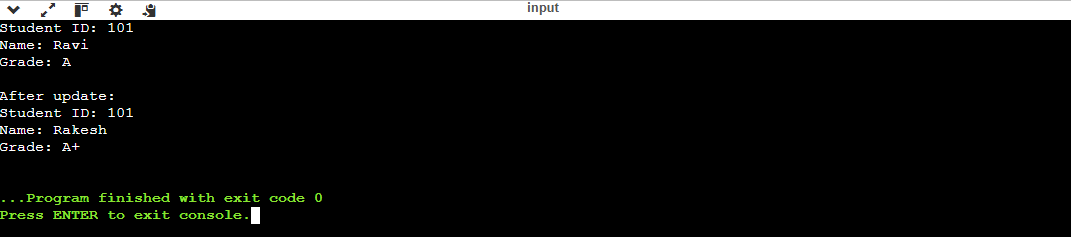
        System.out.println("\nAfter update:");

        controller.updateView();

    }

}

**Output:**



**Ex 11: Implementing Dependency Injection**

**Scenario:**

You are developing a customer management application where the service class depends on a repository class. Use Dependency Injection to manage these dependencies.

**Code:**

**interface** Customer {

**String** findCustomerById(**int** id);

}

**class** CustomerRepositoryImpl **implements** Customer {

**public** **String** findCustomerById(**int** id) {

**return** "Customer ID: " **+** id **+** " - Name: Ravi";

    }

}

**class** CustomerService {

**private** **Customer** repository;

**public** CustomerService(**Customer** repository) {

        this.repository **=** repository;

    }

**public** **void** showCustomer(**int** id) {

**String** result **=** repository.findCustomerById(id);

        System.out.println(result);

    }

}

**public** **class** DependencyInjection {

**public** **static** **void** main(**String**[] args) {

**Customer** repo **=** **new** CustomerRepositoryImpl();

**CustomerService** service **=** **new** CustomerService(repo);

        service.showCustomer(101);

    }

}

**Output:**

****