1. **Implementing Singleton Pattern**

**Logger.java**

public class Logger {

    private static Logger instance;

    private Logger() {

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

    }

    public static Logger getInstance() {

        if(instance == null) {

            instance =  new Logger();

            return instance;

        }

        else {

            return instance;

        }

    }

    public void log(String msg) {

        System.out.println(msg);

    }

}

**Example.java**

public class Example {

    Logger instance = Logger.getInstance();

    public Example() {

        instance.log("I am Example class");

        System.out.println(instance.hashCode());

    }

}

**App.java**

public class App {

    public static void main(String[] args) throws Exception {

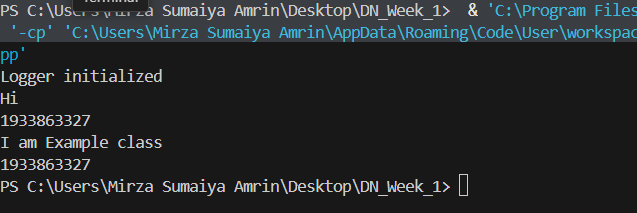
        Logger instance = Logger.getInstance();

        instance.log("Hi");

        System.out.println(instance.hashCode());

        Example ex = new Example();

    }

****}

**Output:**

**2.Implementing the Factory Method Pattern**

**Document.java**

public interface Document {

    public void read();

    public void write();

}

**DocumentFactory.java**

public class DocumentFactory {

    public Document getDocument(String documentType) {

        if (documentType == null) return null;

        if (documentType.equalsIgnoreCase(".DOC")) {

            return (Document) new WordDocument();

        }

        if(documentType.equalsIgnoreCase(".PDF")) {

            return (Document) new PdfDocument();

        }

        if(documentType.equalsIgnoreCase(".CSV")) {

            return (Document) new ExcelDocument();

        }

        // Add more types here as needed

        return null;

    }

}

**ExcelDocument.java**

public class ExcelDocument implements Document{

    @Override

    public void read() {

        System.out.println("Reading Excel file");

    }

    @Override

    public void write() {

        System.out.println("Writing to an Excel file");

    }

}

**PdfDocument.java**

public class PdfDocument implements Document{

    @Override

    public void read() {

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

    }

    @Override

    public void write() {

        System.out.println("Cannot write to a PDF ");

    }

}

**WordDocument.java**

public class WordDocument implements Document{

    @Override

    public void read() {

        System.out.println("reading word document");

    }

    @Override

    public void write() {

        System.out.println("Writing word document");

    }

}

**App.java**

public class App {

    public static void main(String[] args) throws Exception {

        DocumentFactory df = new DocumentFactory();

        Document excel = (Document)df.getDocument(".CSV");

        excel.read();

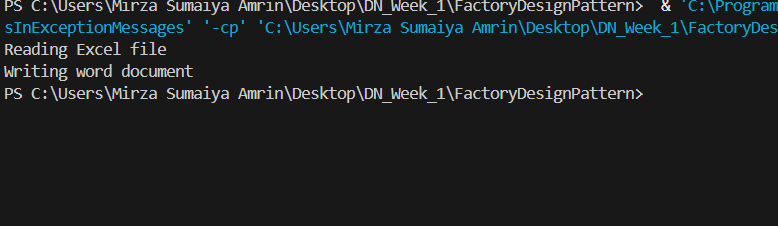
        Document word = (Document) df.getDocument(".DOC");

        word.write();

    }

}

**Output:**



1. **Implementing the Builder Pattern**

**Computer.java**

public class Computer {

    private String CPU;

    private int storage;

    private String graficCard;

    private int RAM;

    public Computer(Builder builder) {

        this.CPU=builder.CPU;

        this.storage=builder.storage;

        this.graficCard=builder.graficCard;

        this.RAM=builder.RAM;

    }

    public String getCPU() {

        return CPU;

    }

    public void setCPU(String CPU) {

        this.CPU = CPU;

    }

    public int getStorage() {

        return storage;

    }

    public void setStorage(int storage) {

        this.storage = storage;

    }

    public String getGraficCard() {

        return graficCard;

    }

    public void setGraficCard(String graficCard) {

        this.graficCard = graficCard;

    }

    public int getRAM() {

        return RAM;

    }

    public void setRAM(int RAM) {

        this.RAM = RAM;

    }

    public static class Builder {

        private String CPU;

        private int storage;

        private String graficCard;

        private int RAM;

        public Builder(String CPU, int RAM) {

            this.CPU=CPU;

            this.RAM=RAM;

        }

        public Builder storage(int storage) {

            this.storage=storage;

            return this;

        }

        public Builder graficCard(String graficCard) {

            this.graficCard=graficCard;

            return this;

        }

        public Computer build() {

            return new Computer(this);

        }

    }

}

**App.java**

public class App {

    public static void main(String[] args) throws Exception {

        //Builder design pattern

        Computer pc = new Computer.Builder("i9",32).storage(512).build();

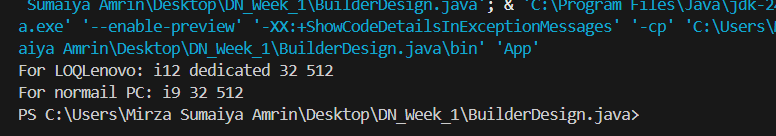
        Computer LOQLenovo = new Computer.Builder("i12",32).graficCard("dedicated").storage(512).build();

        System.out.println("For LOQLenovo: "+LOQLenovo.getCPU()+" "+LOQLenovo.getGraficCard()+" "+LOQLenovo.getRAM()+" "+LOQLenovo.getStorage());

        System.out.println("For normail PC: "+ pc.getCPU()+" "+pc.getRAM()+" "+pc.getStorage());

    }

**Output:**

****

1. **Implementing the Adapter Pattern**

**PaymentProcessor.java**

public interface PaymentProcessor {

    void pay(String PaymentType,int amount);

}

**AdvancedPaymentProcess.java**

public interface AdvancedPaymentProcess {

    void payUPI(int amount);

    void payBank(int amount);

}

**PaymentAdapter.java**

public class PaymentAdapter{

    AdvancedPaymentProcess aMP;

    public PaymentAdapter(String paymentType,int amount) {

        if(paymentType.equalsIgnoreCase("UPI")) {

            aMP = new UPIPayment();

            aMP.payUPI(amount);

        }

        else if(paymentType.equalsIgnoreCase("Bank")) {

            aMP = new BankPayment();

            aMP.payBank(amount);

        }

    }

}

**UPIPayment.java**

public class UPIPayment implements AdvancedPaymentProcess{

    @Override

    public void payUPI(int amount) {

        System.out.println("Paying through upi");

    }

    @Override

    public void payBank(int amount) {

        //do nothing

    }

}

**BankPayment.java**

public class BankPayment implements AdvancedPaymentProcess{

    @Override

    public void payUPI(int amount) {

    }

    @Override

    public void payBank(int amount) {

       System.out.println("Paying through bank");

    }

}

**App.java**

public class App {

    public static void main(String[] args) throws Exception {

        Payment payment1 = new Payment();

        payment1.pay("UPI",1000);

        Payment payment2 = new Payment();

        payment2.pay("Bank",25000);

    }

**Output:**

A computer screen with white text

AI-generated content may be incorrect.

1. **Implementing the Decorator Pattern**

**Notifier.java**

public interface Notifier {

    public void send(String msg);

}

**EmailNotifier.java**

public class EmailNotifier implements Notifier{

    @Override

    public void send(String msg) {

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

    }

}

**SlackNotifier.java**

public class SlackNotifier extends NotifierDecorator{

    public SlackNotifier(Notifier notifier) {

        super(notifier);

    }

    @Override

    public void send(String msg) {

        super.send(msg);

        sendSlackMsg(msg);

    }

    public void sendSlackMsg(String msg) {

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

    }

}

**SMSNotifierDecorator.java**

public class SMSNotifierDecorator extends NotifierDecorator{

    public SMSNotifierDecorator(Notifier notifier) {

        super(notifier);

    }

    @Override

    public void send(String msg) {

       super.send(msg);

       sendSMS(msg);

    }

    public void sendSMS(String msg) {

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

    }

}

**App.java**

public class App {

    public static void main(String[] args) throws Exception {

        Notifier notifier = new EmailNotifier();

        notifier.send("This is an email msg");

        // notifier = new SMSNotifierDecorator(notifier);

        // notifier.send("This will be sent by email and SMS");

        notifier = new SlackNotifier(notifier);

        notifier.send("This is an Email and Slack Msg");

    }

}

**Output:**

A screen shot of a computer

AI-generated content may be incorrect.

1. **Implementing the Proxy Pattern**

**Image.java**

public interface Image {

    void display();

}

**ProxyImage.java**

import java.util.HashMap;

import java.util.Map;

public class ProxyImage implements Image {

    private static Map<String, RealImage> imageCache = new HashMap<>();

    private String filename;

    public ProxyImage(String filename) {

        this.filename = filename;

    }

    @Override

    public void display() {

        RealImage realImage = imageCache.get(filename);

        if (realImage == null) {

            System.out.println("Creating and caching image: " + filename);

            realImage = new RealImage(filename);

            imageCache.put(filename, realImage);

        } else {

            System.out.println("Retrieving from cache: " + filename);

        }

        realImage.display();

    }

}

**RealImage.java**

public class RealImage implements Image {

    private String filename;

    public RealImage(String filename) {

        this.filename = filename;

        loadFromServer();

    }

    private void loadFromServer() {

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

        // Simulate loading time

        try {

            Thread.sleep(1000); // 1 second delay

        } catch (InterruptedException e) {

            Thread.currentThread().interrupt();

        }

    }

    @Override

    public void display() {

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

    }

}

**App.java**

public class App {

    public static void main(String[] args) throws Exception {

        Image img1 = new ProxyImage("photo1.jpg");

        Image img2 = new ProxyImage("photo2.jpg");

        Image img3 = new ProxyImage("photo1.jpg"); // Same as img1 to test caching

        // First-time load (triggers RealImage creation)

        img1.display();

        img2.display();

        // Cached display (no loading)

        img3.display();

    }

}

**Output:**

A screen shot of a computer

AI-generated content may be incorrect.

1. **Implementing the Observer Pattern**

**Stock.java**

public interface Stock {

    public void register(Observer o);

    public void deregister(Observer o);

    public void notifyObservers();

}

**StockMarket.java**

import java.util.ArrayList;

import java.util.List;

public class StockMarket implements Stock{

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

    private double stockPrice;

    public double getStockPrice() {

        return stockPrice;

    }

    public void setStockPrice(double stockPrice) {

        this.stockPrice = stockPrice;

        System.out.println("Stock price updated to: "+stockPrice);

        notifyObservers();

    }

    @Override

    public void register(Observer o) {

        obs.add(o);

    }

    @Override

    public void deregister(Observer o) {

        obs.remove(o);

    }

    @Override

    public void notifyObservers() {

        for(Observer o: obs) {

            o.update(stockPrice);

        }

    }

}

**Observer.java**

public interface Observer {

    public void update(double price);

}

**MobileApp.java**

public class MobileApp implements Observer{

    private String name;

    public MobileApp(String name) {

        this.name = name;

    }

    public void update(double stockPrice) {

        System.out.println("Mobile "+name+" recieved update "+stockPrice);

    }

}

**WebApp.java**

public class WebApp implements Observer{

    private String name;

    public WebApp(String name) {

        this.name = name;

    }

    public void update(double stockPrice) {

        System.out.println("Web user: "+name+"updated on "+stockPrice);

    }

}

**App.java**

public class App {

    public static void main(String[] args) throws Exception {

        StockMarket sm = new StockMarket();

        Observer o1 = new MobileApp("investor1");

        Observer o2 = new WebApp("investor2");

        sm.register(o1);

        sm.register(o2);

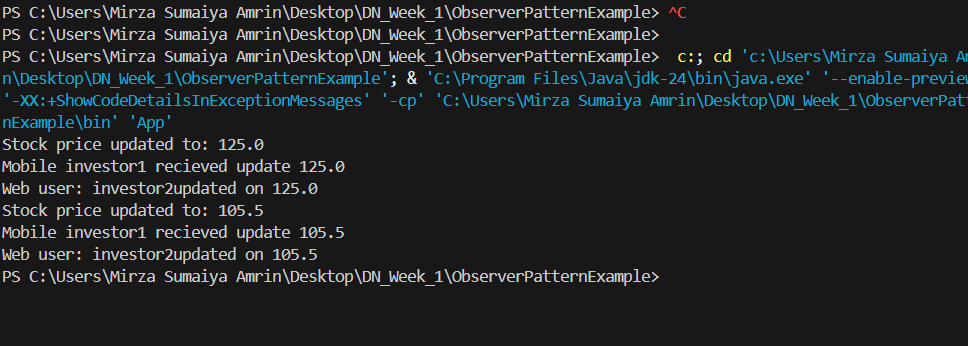
        sm.setStockPrice(125.0);

        sm.setStockPrice(105.5);

    }

}

**Output:**



1. **Implementing the Strategy Pattern**

**PaymentStrategy.java**

public interface PaymentStrategy {

    void pay(double amount);

}

**PaymentContext.java**

public class PaymentContext {

    private PaymentStrategy paymentStrategy;

    // Set strategy at runtime

    public void setPaymentStrategy(PaymentStrategy paymentStrategy) {

        this.paymentStrategy = paymentStrategy;

    }

    // Execute strategy

    public void pay(double amount) {

        if (paymentStrategy == null) {

            System.out.println("Payment strategy not set!");

            return;

        }

        paymentStrategy.pay(amount);

    }

}

**CreditCardPayment.java**

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 ₹" + amount + " using Credit Card: " + cardNumber);

    }

}

**PayPalPayment.java**

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 ₹" + amount + " using PayPal account: " + email);

    }

}

**App.java**

public class App {

    public static void main(String[] args) {

        PaymentContext context = new PaymentContext();

        // Use Credit Card Payment

        context.setPaymentStrategy(new CreditCardPayment("1234-5678-9876-5432"));

        context.pay(2500.00);

        // Switch to PayPal Payment

        context.setPaymentStrategy(new PayPalPayment("user@example.com"));

        context.pay(1800.00);

    }

}

**Output:** A screen shot of a computer

AI-generated content may be incorrect.

1. **Implementing the Command Pattern**

**Command.java**

public interface Command {

    void execute();

}

**Light.java**

public class Light {

    public void turnOn() {

        System.out.println("The light is ON");

    }

    public void turnOff() {

        System.out.println("The light is OFF");

    }

}

**LightOffCommand.java**

public class LightOffCommand implements Command {

    private Light light;

    public LightOffCommand(Light light) {

        this.light = light;

    }

    @Override

    public void execute() {

        light.turnOff();

    }

}

**LightOnCommand.java**

public class LightOnCommand implements Command {

    private Light light;

    public LightOnCommand(Light light) {

        this.light = light;

    }

    @Override

    public void execute() {

        light.turnOn();

    }

}

**RemoteControl.java**

public class RemoteControl {

    private Command command;

    // Set the command dynamically

    public void setCommand(Command command) {

        this.command = command;

    }

    // Execute the command

    public void pressButton() {

        if (command != null) {

            command.execute();

        } else {

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

        }

    }

}

**App.java**

public class App {

    public static void main(String[] args) {

        // Receiver

        Light livingRoomLight = new Light();

        // Commands

        Command lightOn = new LightOnCommand(livingRoomLight);

        Command lightOff = new LightOffCommand(livingRoomLight);

        // Invoker

        RemoteControl remote = new RemoteControl();

        // Turn ON light

        remote.setCommand(lightOn);

        remote.pressButton();

        // Turn OFF light

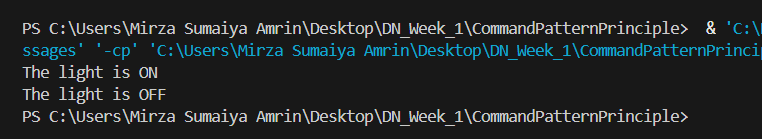
        remote.setCommand(lightOff);

        remote.pressButton();

    }

}

**Output:**



**10.Implementing the MVC Pattern**

**Student.java**

public class Student {

    private String name;

    private String id;

    private String grade;

    // Constructor

    public Student(String name, String id, String grade) {

        this.name = name;

        this.id = id;

        this.grade = grade;

    }

    // Getters and Setters

    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; }

}

**StudentController.java**

public class StudentController {

    private Student model;

    private StudentView view;

    public StudentController(Student model, StudentView view) {

        this.model = model;

        this.view = view;

    }

    // Controller methods to update model

    public void setStudentName(String name) {

        model.setName(name);

    }

    public void setStudentId(String id) {

        model.setId(id);

    }

    public void setStudentGrade(String grade) {

        model.setGrade(grade);

    }

    // Controller method to update view

    public void updateView() {

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

    }

}

**StudentView.java**

public 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);

    }

}

**App.java**

public class App {

    public static void main(String[] args) {

        // Create model

        Student student = new Student("Alice", "S123", "A");

        // Create view

        StudentView view = new StudentView();

        // Create controller

        StudentController controller = new StudentController(student, view);

        // Display initial details

        controller.updateView();

        // Update model via controller

        controller.setStudentName("Bob");

        controller.setStudentGrade("B+");

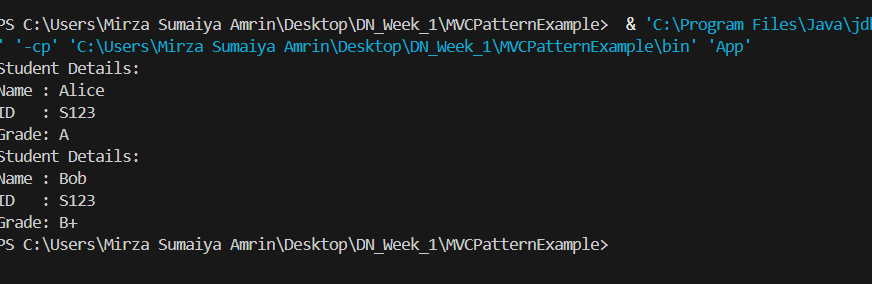
        // Display updated details

        controller.updateView();

    }

}

**Output:**



**11.Implementing Dependency Injection**

**Customer.java**

public class Customer {

    private int id;

    private String name;

    private String email;

    public Customer(int id, String name, String email) {

        this.id = id;

        this.name = name;

        this.email = email;

    }

    public void displayInfo() {

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

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

        System.out.println("Email     : " + email);

    }

}

**CustomerRepository.java**

public interface CustomerRepository {

    Customer findCustomerById(int id);

}

CustomerReplositoryImpl.java

public class CustomerRepositoryImpl implements CustomerRepository {

    @Override

    public Customer findCustomerById(int id) {

        // Simulate fetching customer data

        return new Customer(id, "Customer #" + id, "customer" + id + "@example.com");

    }

}

**CustomerService.java**

public class CustomerService {

    private final CustomerRepository customerRepository;

    // Constructor Injection

    public CustomerService(CustomerRepository customerRepository) {

        this.customerRepository = customerRepository;

    }

    public void displayCustomerById(int id) {

        Customer customer = customerRepository.findCustomerById(id);

        customer.displayInfo();

    }

}

**App.java**

public class App {

    public static void main(String[] args) {

        // Create repository instance

        CustomerRepository repository = new CustomerRepositoryImpl();

        // Inject repository into service

        CustomerService service = new CustomerService(repository);

        // Use the service

        service.displayCustomerById(101);

    }

}

**Output:**

A screen shot of a computer program

AI-generated content may be incorrect.