**Exercise 1: Implementing the Singleton Pattern**

**CODE:**

**Logger.java**

public class Logger {

    private Logger() {

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

    }

    private static Logger log;

    public static Logger glog() {

        if (log == null) {

            log = new Logger();

        }

        return log;

    }

}

**LoggerTest.java**

public class LoggerTest {

    public static void main(String[] args) {

        Logger log1= Logger.glog();

        System.out.println(log1.hashCode());

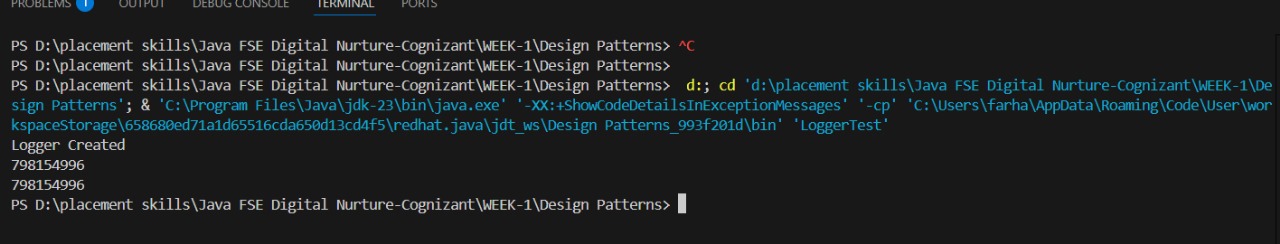
        Logger log2= Logger.glog();

        System.out.println(log2.hashCode());

    }

}

**OUTPUT:**



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

**CODE:**

**Document.java**

public interface Document{

    void open();

}

**ExcelDocument.java**

public class ExcelDocument implements Document {

    public void open() {

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

    }

}

**WordDocument.java**

public class WordDocument implements Document{

    public void open() {

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

    }

}

**PdfDocument.java**

public class PdfDocument implements Document {

      public void open() {

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

    }

}

**DocumentFactory.java**

public abstract class DocumentFactory {

    public abstract Document createDocument();

}

**WordFactory.java**

public class WordFactory extends DocumentFactory {

    public Document createDocument() {

        return new WordDocument();

    }

}

**ExcelFactory.java**

public class ExcelFactory extends DocumentFactory {

     public Document createDocument() {

        return new ExcelDocument();

    }

}

**PdfFactory.java**

public class PdfFactory extends DocumentFactory {

     public Document createDocument() {

        return new PdfDocument();

    }

}

**DocumentTest.java**

public class DocumentTest {

      public static void main(String[] args) {

        DocumentFactory wordFactory = new WordFactory();

        Document word = wordFactory.createDocument();

        word.open();

        DocumentFactory pdfFactory = new PdfFactory();

        Document pdf = pdfFactory.createDocument();

        pdf.open();

        DocumentFactory excelFactory = new ExcelFactory();

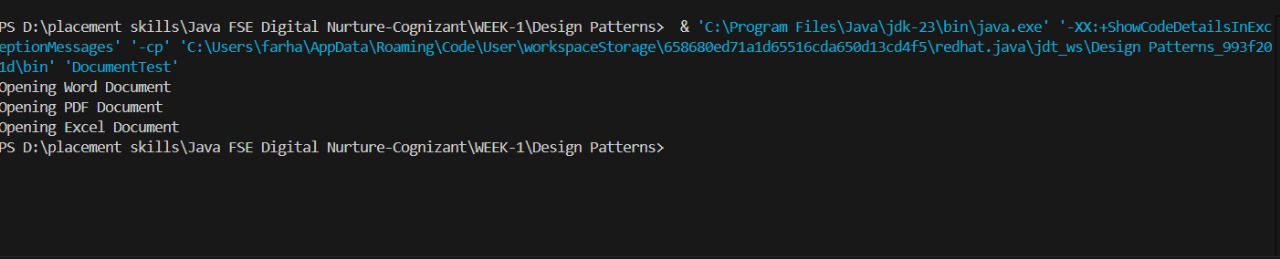
        Document excel = excelFactory.createDocument();

        excel.open();

    }

}

**OUTPUT:**

****

**Exercise 3: Implementing the Builder Pattern**

**Computer.java**

public class Computer {

    // Required and optional parts

    private String cpu;

    private String ram;

    private String storage;

    private String graphicsCard;

    // Private constructor

    private Computer(Builder builder) {

        this.cpu = builder.cpu;

        this.ram = builder.ram;

        this.storage = builder.storage;

        this.graphicsCard = builder.graphicsCard;

    }

    // Print computer configuration

    public void displayConfig() {

        System.out.println("CPU: " + cpu);

        System.out.println("RAM: " + ram);

        System.out.println("Storage: " + storage);

        System.out.println("Graphics Card: " + graphicsCard);

    }

    // Static nested Builder class

    public static class Builder {

        private String cpu;

        private String ram;

        private String storage;

        private String graphicsCard;

        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 setGraphicsCard(String graphicsCard) {

            this.graphicsCard = graphicsCard;

            return this;

        }

        public Computer build() {

            return new Computer(this);

        }

    }

}

**ComputerTest.java**

public class ComputerTest {

    public static void main(String[] args) {

        // Basic build

        Computer officePC = new Computer.Builder()

                .setCpu("Intel i5")

                .setRam("8GB")

                .setStorage("256GB SSD")

                .build();

        System.out.println("Office PC Configuration:");

        officePC.displayConfig();

        // Advanced build

        Computer gamingPC = new Computer.Builder()

                .setCpu("Intel i9")

                .setRam("32GB")

                .setStorage("1TB SSD")

                .setGraphicsCard("NVIDIA RTX 4080")

                .build();

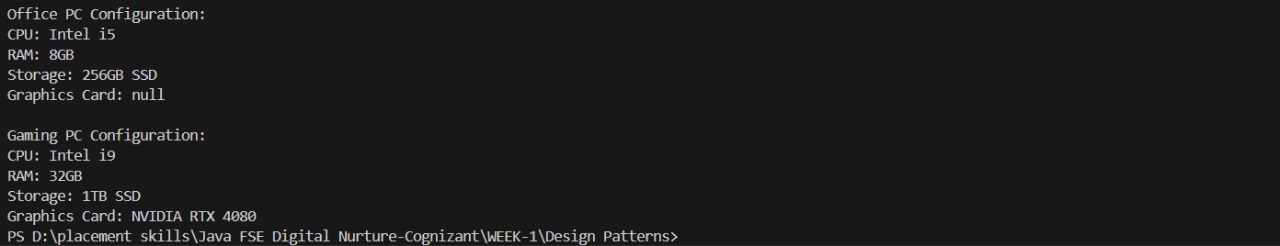
        System.out.println("\nGaming PC Configuration:");

        gamingPC.displayConfig();

    }

}

**OUTPUT:**



**Exercise 4: Implementing the Adapter Pattern**

**PaymentProcessor.java**

public interface PaymentProcessor {

    void processPayment(int amount);

}

**Paytm.java**

public class Paytm {

    public void sendmoney(int money){

        System.out.println("Money has been transfered using Paytm. Amount:"+money);

    }

}

**PhonePe.java**

public class PhonePe {

    public void transfermoney(int cash){

        System.out.println("Money has been transfered using PhonePe. Amount:"+cash);

    }

}

**PaytmAdapter.java**

public class PaytmAdapter implements PaymentProcessor{

    private Paytm paytm=new Paytm();

    public void processPayment(int amount){

        paytm.sendmoney(amount);

    }

}

**PhonePeAdapter.java**

public class PhonePeAdapter implements PaymentProcessor {

    private PhonePe phonepe=new PhonePe();

    public void processPayment(int amount){

        phonepe.transfermoney(amount);

    }

}

**PaymentTest.java**

public class PaymentTest {

    public static void main(String[] args) {

        PaymentProcessor processor1 = new PhonePeAdapter();

        processor1.processPayment(500);

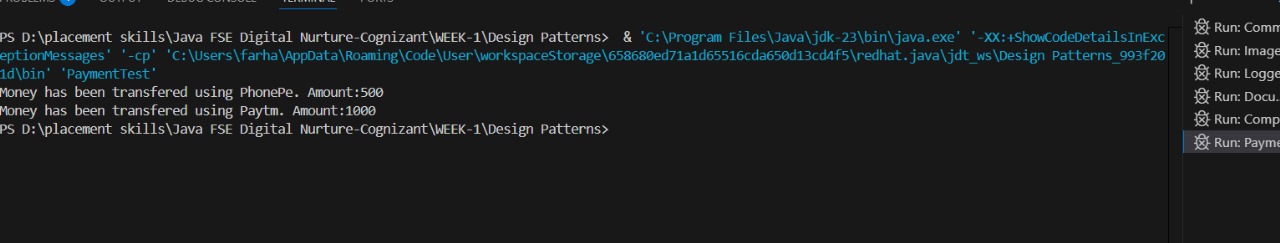
        PaymentProcessor processor2 = new PaytmAdapter();

        processor2.processPayment(1000);

    }

}

**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 wrappedNotifier;

    public NotifierDecorator(Notifier notifier) {

        this.wrappedNotifier = notifier;

    }

    public void send(String message) {

        wrappedNotifier.send(message);

    }

}

**EmailNotifier.java**

public class EmailNotifier implements Notifier {

     public void send(String message) {

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

    }

}

**SlackNotifier.java**

public class SlackNotifier extends NotifierDecorator {

    public SlackNotifier(Notifier notifier) {

        super(notifier);

    }

    public void send(String message) {

        super.send(message);

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

    }

}

**SMSNotifier.java**

public class SMSNotifier extends NotifierDecorator {

    public SMSNotifier(Notifier notifier) {

        super(notifier);

    }

    public void send(String message) {

        super.send(message);

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

    }

}

**NotificationTest.java**

public class NotificationTest {

    public static void main(String[] args) {

        Notifier baseNotifier = new EmailNotifier();

        Notifier smsNotifier = new SMSNotifier(baseNotifier);

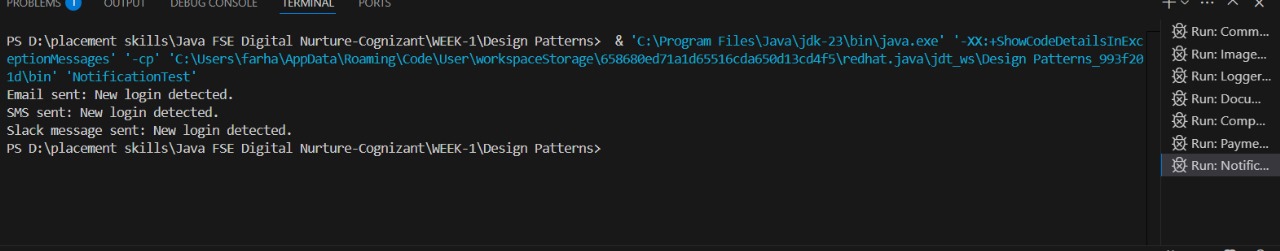
        Notifier fullNotifier = new SlackNotifier(smsNotifier);

        fullNotifier.send("New login detected.");

    }

}

**OUTPUT:**



**Exercise 6: Implementing the Proxy Pattern**

**Image.java**

public interface Image {

    void display();

}

**RealImage.java**

public class RealImage implements Image{

    private String filename;

    public RealImage(String filename) {

        this.filename = filename;

        loadFromDisk();  // expensive operation

    }

    private void loadFromDisk() {

        System.out.println("Loading " + filename + " from disk...");

    }

    public void display() {

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

    }

}

**ProxyImage.java**

public 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); // lazy load

        }

        realImage.display();

    }

}

**ImageViewer.java**

public class ImageViewer {

     public static void main(String[] args) {

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

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

        image1.display();

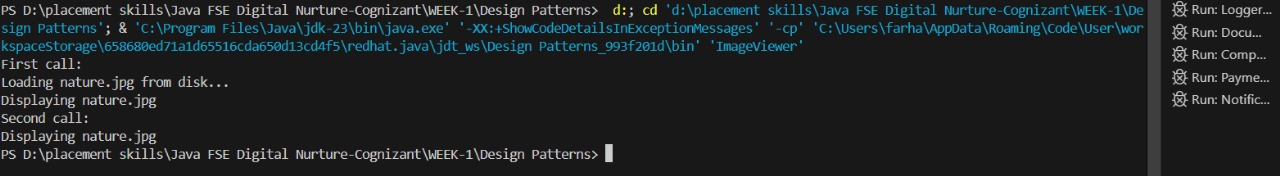
        System.out.println("Second call:");

        image1.display();

    }

}

**OUTPUT:**

****

**Exercise 7: Implementing the Observer Pattern**

**Stock.java**

public interface Stock {

    void registerObserver(Observer o);

    void removeObserver(Observer o);

    void notifyObservers();

}

**Observer.java**

public interface Observer {

    void update(int price);

}

**MobileApp.java**

public class MobileApp implements Observer {

    private String name;

    public MobileApp(String name) {

        this.name = name;

    }

    public void update(int price) {

        System.out.println("[" + name + " - Mobile] Stock Price Updated: " + price);

    }

}

**WebApp.java**

public class WebApp implements Observer {

    private String name;

    public WebApp(String name) {

        this.name = name;

    }

    public void update(int price) {

        System.out.println("[" + name + " - Web] Stock Price Updated: " + price);

    }

}

**StockMarket.java**

import java.util.\*;

public class StockMarket implements Stock {

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

    private int stockPrice;

    public void setStockPrice(int price) {

        this.stockPrice = price;

        notifyObservers();

    }

    public int getStockPrice() {

        return stockPrice;

    }

    @Override

    public void registerObserver(Observer o) {

        observers.add(o);

    }

    @Override

    public void removeObserver(Observer o) {

        observers.remove(o);

    }

    @Override

    public void notifyObservers() {

        for (Observer observer : observers) {

            observer.update(stockPrice);

        }

    }

}

**ObserverPatternTest.java**

public class ObserverPatternTest {

    public static void main(String[] args) {

        StockMarket stockMarket = new StockMarket();

        Observer mobileClient = new MobileApp("Client A");

        Observer webClient = new WebApp("Client B");

        stockMarket.registerObserver(mobileClient);

        stockMarket.registerObserver(webClient);

        System.out.println("Setting stock price to 120...");

        stockMarket.setStockPrice(120);

        System.out.println("\nSetting stock price to 150...");

        stockMarket.setStockPrice(150);

        // Remove one observer

        stockMarket.removeObserver(webClient);

        System.out.println("\nSetting stock price to 200...");

        stockMarket.setStockPrice(200);

    }

}

**OUTPUT: **

**Exercise 8: Implementing the Strategy Pattern**

**PaymentStrategy.java**

public interface PaymentStrategy {

    void pay(int amount);

}

**BankTransferPayment.java**

public class BankTransferPayment implements PaymentStrategy {

    public void pay(int amount) {

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

    }

}

**PaypalPayment.java**

public class PayPalPayment implements PaymentStrategy {

    public void pay(int amount) {

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

    }

}

**CreditCardPayment.java**

public class CreditCardPayment implements PaymentStrategy {

    public void pay(int amount) {

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

    }

}

**ShoppingCart.java**

public class ShoppingCart {

    private PaymentStrategy paymentStrategy;

    // Set strategy at runtime

    public void setPaymentStrategy(PaymentStrategy strategy) {

        this.paymentStrategy = strategy;

    }

    public void checkout(int amount) {

        if (paymentStrategy == null) {

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

            return;

        }

        paymentStrategy.pay(amount);

    }

}

**StrategyTest.java**

public class StrategyTest {

    public static void main(String[] args) {

        ShoppingCart cart = new ShoppingCart();

        cart.setPaymentStrategy(new CreditCardPayment());

        cart.checkout(250);  // Paid 250 using Credit Card.

        cart.setPaymentStrategy(new PayPalPayment());

        cart.checkout(400);  // Paid 400 using PayPal.

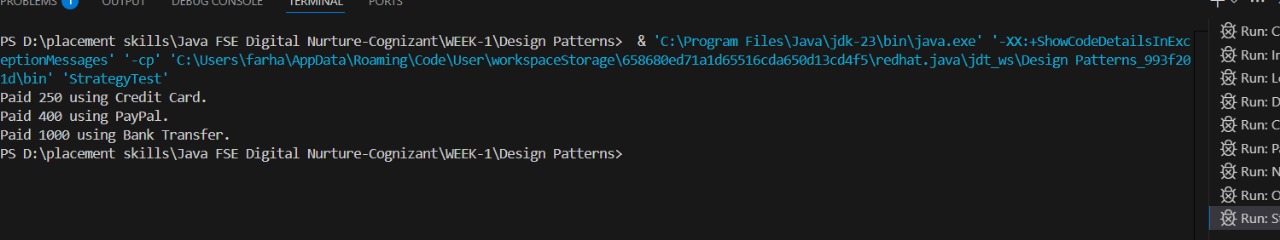
        cart.setPaymentStrategy(new BankTransferPayment());

        cart.checkout(1000); // Paid 1000 using Bank Transfer.

    }

}

**OUTPUT**

****

**Exercise 9: Implementing the Command Pattern**

**Light.java**

public class Light {

    public void turnOn() {

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

    }

    public void turnOff() {

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

    }

}

**Command.java**

public interface Command {

    void execute();

}

**RemoteControl.java**

public class RemoteControl {

    private Command command;

    // Set 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 assigned.");

        }

    }

}

**LightOnCommand.java**

public class LightOnCommand implements Command {

    private Light light;

    public LightOnCommand(Light light) {

        this.light = light;

    }

    public void execute() {

        light.turnOn();

    }

}

**LightOffCommand.java**

public class LightOffCommand implements Command {

    private Light light;

    public LightOffCommand(Light light) {

        this.light = light;

    }

    public void execute() {

        light.turnOff();

    }

}

**CommandPatternTest.java**

public class CommandPatternTest {

    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 the light

        remote.setCommand(lightOn);

        remote.pressButton();  // Output: Light is ON

        // Turn OFF the light

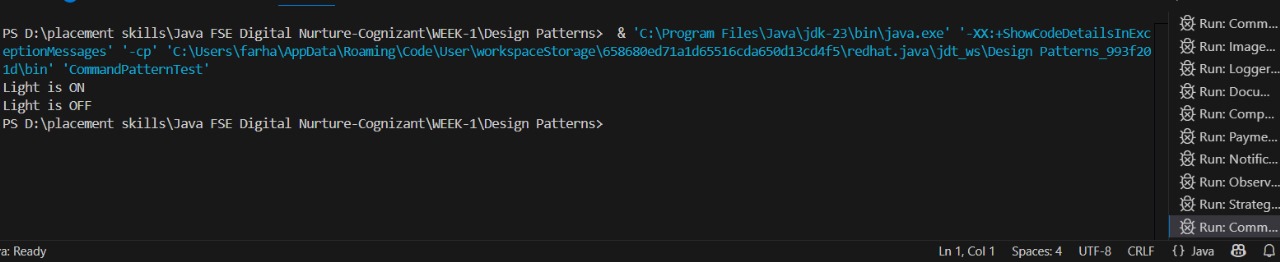
        remote.setCommand(lightOff);

        remote.pressButton();  // Output: Light is OFF

    }

}

**OUTPUT:**

****

**Exercise 10: Implementing the MVC Pattern**

**Student.java**

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;

    }

    // 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 student;

    private StudentView view;

    public StudentController(Student student, StudentView view) {

        this.student = student;

        this.view = view;

    }

    // Setters

    public void setStudentName(String name) {

        student.setName(name);

    }

    public void setStudentId(String id) {

        student.setId(id);

    }

    public void setStudentGrade(String grade) {

        student.setGrade(grade);

    }

    // Getters

    public String getStudentName() {

        return student.getName();

    }

    public String getStudentId() {

        return student.getId();

    }

    public String getStudentGrade() {

        return student.getGrade();

    }

    // Display

    public void updateView() {

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

    }

}

**StudenView.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);

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

    }

}

**MVCTest.java**

public class MVCTest {

    public static void main(String[] args) {

        // Model

        Student student = new Student("Alice", "S001", "A");

        // View

        StudentView view = new StudentView();

        // Controller

        StudentController controller = new StudentController(student, view);

        // Display original

        controller.updateView();

        // Update via controller

        controller.setStudentName("Bob");

        controller.setStudentGrade("B+");

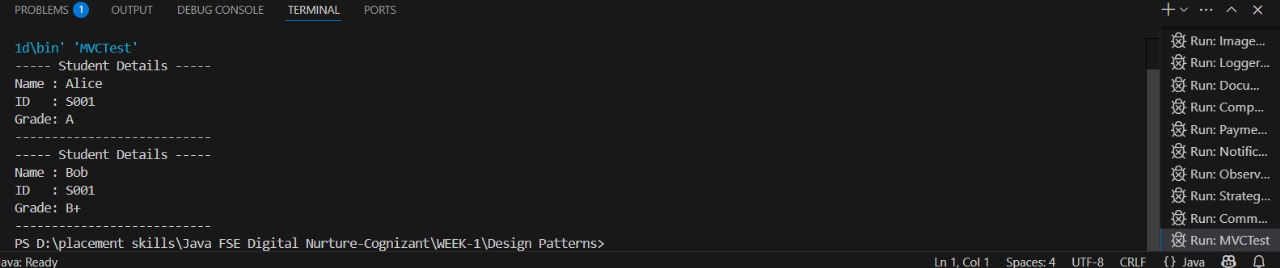
        // Display updated

        controller.updateView();

    }

}

**OUTPUT:**



**Exercise 11: Implementing Dependency Injection**

**CustomerRepository.java**

public interface CustomerRepository {

  String findCustomerById(String id);

}

**CustomerRepositoryImpl.java**

public class CustomerRepositoryImpl implements CustomerRepository {

    @Override

    public String findCustomerById(String id) {

        // In real apps, this would query a DB

        return "Customer{id='" + id + "', name='Alice'}";

    }

}

**CustomerService.java**

public class CustomerService {

    private CustomerRepository customerRepository;

    // Constructor Injection

    public CustomerService(CustomerRepository customerRepository) {

        this.customerRepository = customerRepository;

    }

    public void getCustomerDetails(String id) {

        String customer = customerRepository.findCustomerById(id);

        System.out.println("Fetched Customer: " + customer);

    }

}

**DITest.java**

public class DITest {

    public static void main(String[] args) {

        // Create repository (dependency)

        CustomerRepository repo = new CustomerRepositoryImpl();

        // Inject into service

        CustomerService service = new CustomerService(repo);

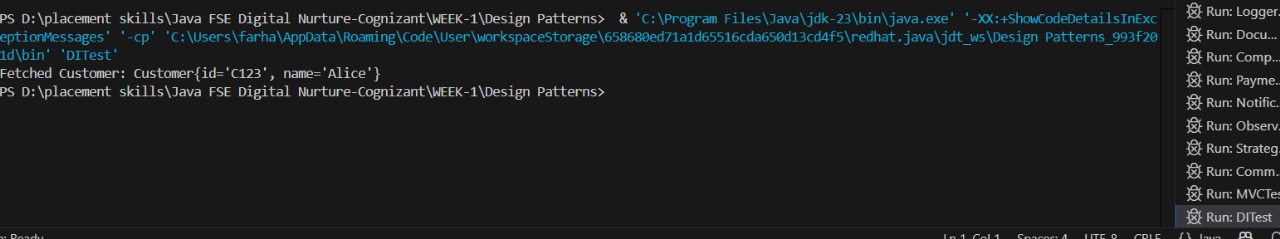
        // Use service

        service.getCustomerDetails("C123");

    }

}

**OUTPUT:**

****