**Superset ID: 6416838**

**Exercise 1: Implementing the Singleton Pattern**

**PROGRAM**

**LOGGER.JAVA**

package singleton;

public class Logger {

private static Logger instance = new Logger();

private Logger() {

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

}

public static Logger getInstance() {

return instance;

}

public void log(String message) {

System.out.println("[LOG] " + message);

}

}

**MAIN.JAVA**

package singleton;

public class Main {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

Logger logger2 = Logger.getInstance();

logger1.log("Application started.");

logger2.log("User logged in.");

if (logger1 == logger2) {

System.out.println("Only one Logger instance is used across the application.");

} else {

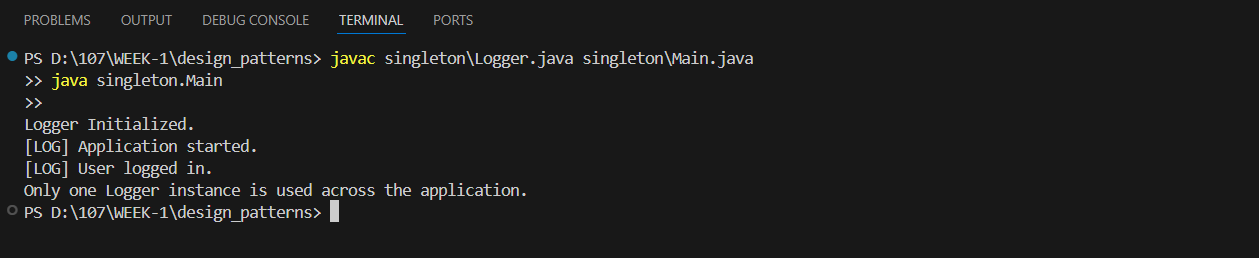
System.out.println(" Multiple Logger instances detected.");

}

}

}

**OUTPUT:**

****

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

**PROGRAM:**

public class FactoryMethodPatternExample {

    interface Document {

        void open();

    }

    static class WordDocument implements Document {

        public void open() {

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

        }

    }

    static class PdfDocument implements Document {

        public void open() {

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

        }

    }

    static class ExcelDocument implements Document {

        public void open() {

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

        }

    }

    abstract static class DocumentFactory {

        public abstract Document createDocument();

    }

    static class WordDocumentFactory extends DocumentFactory {

        public Document createDocument() {

            return new WordDocument();

        }

    }

    static class PdfDocumentFactory extends DocumentFactory {

        public Document createDocument() {

            return new PdfDocument();

        }

    }

    static class ExcelDocumentFactory extends DocumentFactory {

        public Document createDocument() {

            return new ExcelDocument();

        }

    }

    public static void main(String[] args) {

        DocumentFactory wordFactory = new WordDocumentFactory();

        Document word = wordFactory.createDocument();

        word.open();

        DocumentFactory pdfFactory = new PdfDocumentFactory();

        Document pdf = pdfFactory.createDocument();

        pdf.open();

        DocumentFactory excelFactory = new ExcelDocumentFactory();

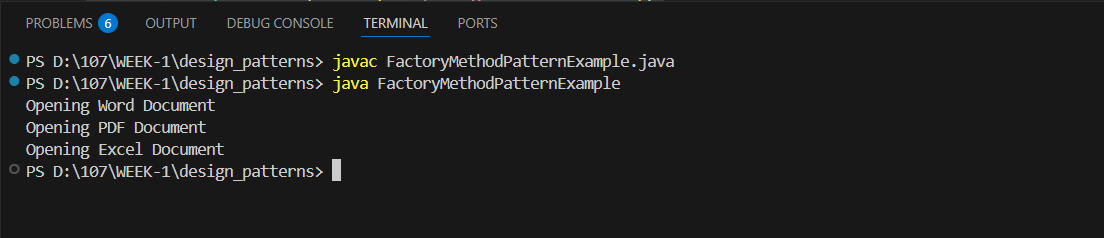
        Document excel = excelFactory.createDocument();

        excel.open();

    }

}

**OUTPUT:**

****

**Exercise 3: Implementing the Builder Pattern**

**PROGRAM:**

public class BuilderPatternExample {

    static class Computer {

        private final String cpu;

        private final String ram;

        private final String storage;

        private final String graphicsCard;

        private Computer(Builder builder) {

            this.cpu = builder.cpu;

            this.ram = builder.ram;

            this.storage = builder.storage;

            this.graphicsCard = builder.graphicsCard;

        }

        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);

            }

        }

        public void showConfig() {

            System.out.println("Computer Configuration:");

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

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

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

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

            System.out.println();

        }

    }

    public static void main(String[] args) {

        Computer basicComputer = new Computer.Builder()

                .setCPU("Intel i3")

                .setRAM("8GB")

                .setStorage("256GB SSD")

                .build();

        Computer gamingComputer = new Computer.Builder()

                .setCPU("Intel i7")

                .setRAM("16GB")

                .setStorage("1TB SSD")

                .setGraphicsCard("NVIDIA RTX 3060")

                .build();

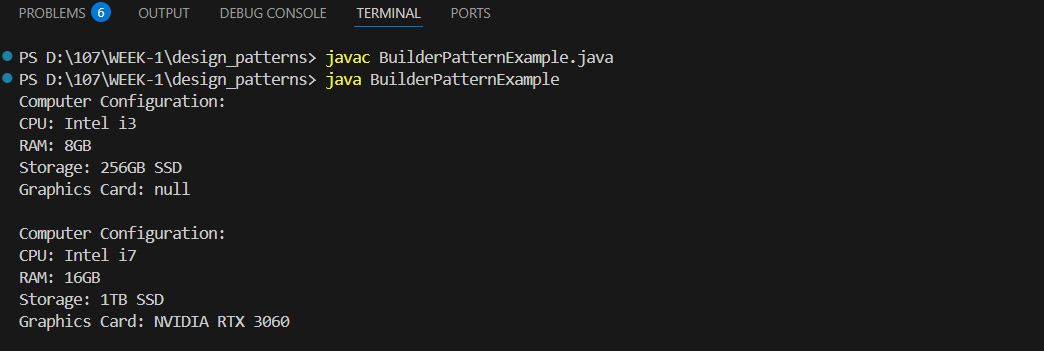
        basicComputer.showConfig();

        gamingComputer.showConfig();

    }

}

OUTPUT:



**Exercise 4: Implementing the Adapter Pattern**

PROGRAM:

public class AdapterPatternExample {

    interface PaymentProcessor {

        void processPayment(double amount);

    }

    static class PayPalGateway {

        public void sendPayment(double amountInDollars) {

            System.out.println("Processing payment via PayPal: $" + amountInDollars);

        }

    }

    static class StripeGateway {

        public void makePayment(double value) {

            System.out.println("Processing payment via Stripe: $" + value);

        }

    }

    static class RazorpayGateway {

        public void pay(double amount) {

            System.out.println("Processing payment via Razorpay: $" + amount);

        }

    }

    static class PayPalAdapter implements PaymentProcessor {

        private final PayPalGateway payPalGateway;

        public PayPalAdapter(PayPalGateway payPalGateway) {

            this.payPalGateway = payPalGateway;

        }

        public void processPayment(double amount) {

            payPalGateway.sendPayment(amount);

        }

    }

    static class StripeAdapter implements PaymentProcessor {

        private final StripeGateway stripeGateway;

        public StripeAdapter(StripeGateway stripeGateway) {

            this.stripeGateway = stripeGateway;

        }

        public void processPayment(double amount) {

            stripeGateway.makePayment(amount);

        }

    }

    static class RazorpayAdapter implements PaymentProcessor {

        private final RazorpayGateway razorpayGateway;

        public RazorpayAdapter(RazorpayGateway razorpayGateway) {

            this.razorpayGateway = razorpayGateway;

        }

        public void processPayment(double amount) {

            razorpayGateway.pay(amount);

        }

    }

    public static void main(String[] args) {

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

        paypalProcessor.processPayment(150.0);

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

        stripeProcessor.processPayment(250.0);

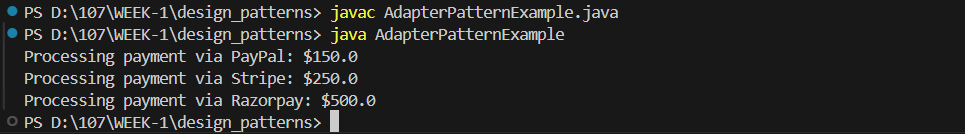
        PaymentProcessor razorpayProcessor = new RazorpayAdapter(new RazorpayGateway());

        razorpayProcessor.processPayment(500.0);

    }

}

**OUTPUT:**



**Exercise 5: Implementing the Decorator Pattern**

**PROGRAM:**

public class DecoratorPatternExample {

    interface Notifier {

        void send(String message);

    }

    static class EmailNotifier implements Notifier {

        public void send(String message) {

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

        }

    }

    static abstract class NotifierDecorator implements Notifier {

        protected Notifier wrappee;

        public NotifierDecorator(Notifier notifier) {

            this.wrappee = notifier;

        }

        public void send(String message) {

            wrappee.send(message);

        }

    }

    static class SMSNotifierDecorator extends NotifierDecorator {

        public SMSNotifierDecorator(Notifier notifier) {

            super(notifier);

        }

        public void send(String message) {

            super.send(message);

            sendSMS(message);

        }

        private void sendSMS(String message) {

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

        }

    }

    static class SlackNotifierDecorator extends NotifierDecorator {

        public SlackNotifierDecorator(Notifier notifier) {

            super(notifier);

        }

        public void send(String message) {

            super.send(message);

            sendSlack(message);

        }

        private void sendSlack(String message) {

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

        }

    }

    public static void main(String[] args) {

        Notifier notifier = new EmailNotifier();

        notifier = new SMSNotifierDecorator(notifier);

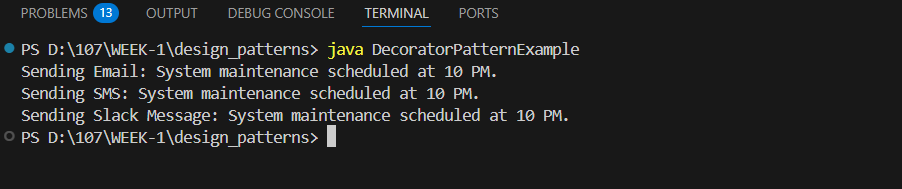
        notifier = new SlackNotifierDecorator(notifier);

        notifier.send("System maintenance scheduled at 10 PM.");

    }

}

**OUTPUT:**

****

**Exercise 6: Implementing the Proxy Pattern**

**PROGRAM:**

public class ProxyPatternExample {

    interface Image {

        void display();

    }

    static class RealImage implements Image {

        private final 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);

        }

    }

    static class ProxyImage implements Image {

        private final 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("Image already loaded. Using cached image: " + filename);

            }

            realImage.display();

        }

    }

        public static void main(String[] args) {

        System.out.println("Creating ProxyImage objects...");

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

        Image image2 = new ProxyImage("image2.jpg");

        System.out.println("\nDisplaying images...");

        image1.display();

        image2.display();

        System.out.println("\nDisplaying images again (should use cache)...");

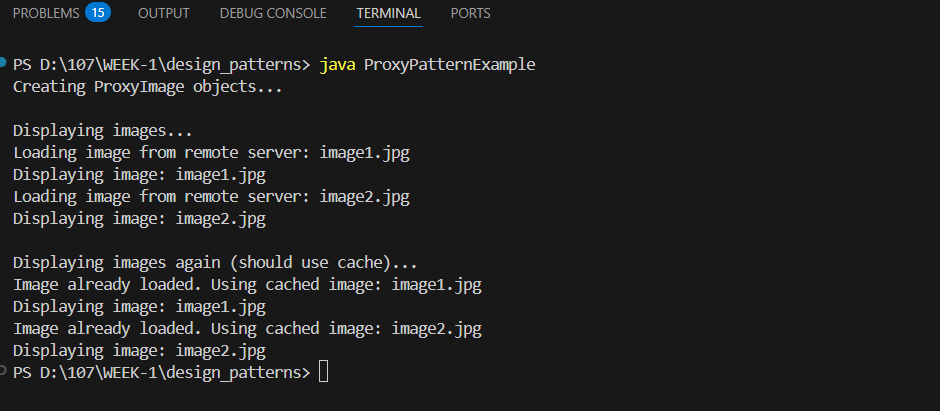
        image1.display();

        image2.display();

    }

}

**OUTPUT:**

****

**Exercise 7: Implementing the Observer Pattern**

**PROGRAM:**

import java.util.ArrayList;

import java.util.List;

import java.util.Scanner;

interface Stock {

    void registerObserver(Observer observer);

    void removeObserver(Observer observer);

    void notifyObservers();

}

interface Observer {

    void update(String stockName, double stockPrice);

}

class StockMarket implements Stock {

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

    private String stockName;

    private double stockPrice;

    public void setStock(String stockName, double stockPrice) {

        this.stockName = stockName;

        this.stockPrice = stockPrice;

        notifyObservers();

    }

    public void registerObserver(Observer observer) {

        observers.add(observer);

        System.out.println(observer.getClass().getSimpleName() + " registered.");

    }

    public void removeObserver(Observer observer) {

        observers.remove(observer);

        System.out.println(observer.getClass().getSimpleName() + " removed.");

    }

    public void notifyObservers() {

        for (Observer observer : observers) {

            observer.update(stockName, stockPrice);

        }

    }

}

class MobileApp implements Observer {

    public void update(String stockName, double stockPrice) {

        System.out.println("MobileApp: " + stockName + " updated to $" + stockPrice);

    }

}

class WebApp implements Observer {

    public void update(String stockName, double stockPrice) {

        System.out.println("WebApp: " + stockName + " updated to $" + stockPrice);

    }

}

public class ObserverPatternExample {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        StockMarket stockMarket = new StockMarket();

        Observer mobileApp = new MobileApp();

        Observer webApp = new WebApp();

        stockMarket.registerObserver(mobileApp);

        stockMarket.registerObserver(webApp);

        System.out.println("\nEnter stock updates (type 'exit' to stop):");

        while (true) {

            System.out.print("\nEnter stock name: ");

            String name = scanner.nextLine();

            if (name.equalsIgnoreCase("exit")) {

                break;

            }

            System.out.print("Enter stock price: ");

            double price;

            try {

                price = Double.parseDouble(scanner.nextLine());

            } catch (NumberFormatException e) {

                System.out.println("Invalid price. Try again.");

                continue;

            }

            stockMarket.setStock(name, price);

        }

        scanner.close();

        System.out.println("\nProgram terminated.");

    }

}

**OUTPUT:**

****

**Exercise 8: Implementing the Strategy Pattern**

**PROGRAM:**

import java.util.Scanner;

interface PaymentStrategy {

    void pay(double amount);

}

class CreditCardPayment implements PaymentStrategy {

    private String cardNumber;

    private String cardHolder;

    public CreditCardPayment(String cardNumber, String cardHolder) {

        this.cardNumber = cardNumber;

        this.cardHolder = cardHolder;

    }

    public void pay(double amount) {

        System.out.println("Paid $" + amount + " using Credit Card [" + cardHolder + "]");

    }

}

class PayPalPayment implements PaymentStrategy {

    private String email;

    public PayPalPayment(String email) {

        this.email = email;

    }

    public void pay(double amount) {

        System.out.println("Paid $" + amount + " using PayPal account [" + email + "]");

    }

}

class PaymentContext {

    private PaymentStrategy paymentStrategy;

    public void setPaymentStrategy(PaymentStrategy strategy) {

        this.paymentStrategy = strategy;

    }

    public void processPayment(double amount) {

        if (paymentStrategy == null) {

            System.out.println("Payment method not set.");

        } else {

            paymentStrategy.pay(amount);

        }

    }

}

public class StrategyPatternExample {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        PaymentContext context = new PaymentContext();

        System.out.println("Select Payment Method:");

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

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

        System.out.print("Enter choice (1 or 2): ");

        int choice = scanner.nextInt();

        scanner.nextLine();

        switch (choice) {

            case 1:

                System.out.print("Enter card holder name: ");

                String name = scanner.nextLine();

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

                String number = scanner.nextLine();

                context.setPaymentStrategy(new CreditCardPayment(number, name));

                break;

            case 2:

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

                String email = scanner.nextLine();

                context.setPaymentStrategy(new PayPalPayment(email));

                break;

            default:

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

                scanner.close();

                return;

        }

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

        double amount = scanner.nextDouble();

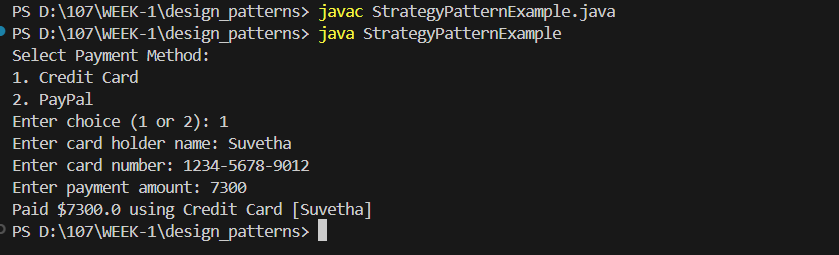
        context.processPayment(amount);

        scanner.close();

    }

}

**OUTPUT:**



**Exercise 9: Implementing the Command Pattern**

**PROGRAM:**

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 CommandPatternExample {

    public static void main(String[] args) {

        Light livingRoomLight = new Light();

        Command lightOn = new LightOnCommand(livingRoomLight);

        Command lightOff = new LightOffCommand(livingRoomLight);

        RemoteControl remote = new RemoteControl();

        System.out.println("Turning on the light:");

        remote.setCommand(lightOn);

        remote.pressButton();

        System.out.println("Turning off the light:");

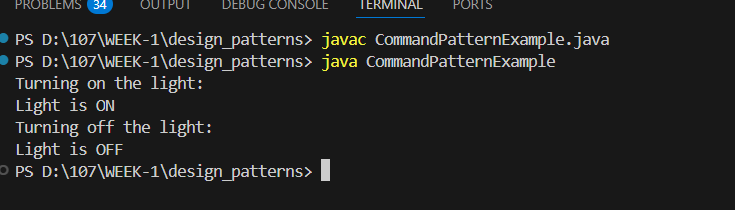
        remote.setCommand(lightOff);

        remote.pressButton();

    }

}

**OUTPUT:**

****

**Exercise 10: Implementing the MVC Pattern**

**PROGRAM:**

public class MVCPatternExample {

    static 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; }

    }

    static 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();

        }

    }

    static 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 void setStudentId(String id) {

            student.setId(id);

        }

        public void setStudentGrade(String grade) {

            student.setGrade(grade);

        }

        public void updateView() {

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

        }

    }

    public static void main(String[] args) {

        Student student = new Student("Suve", "S101", "A");

        StudentView view = new StudentView();

        StudentController controller = new StudentController(student, view);

        System.out.println("Initial Student Info:");

        controller.updateView();

        controller.setStudentName("Suvetha");

        controller.setStudentGrade("A+");

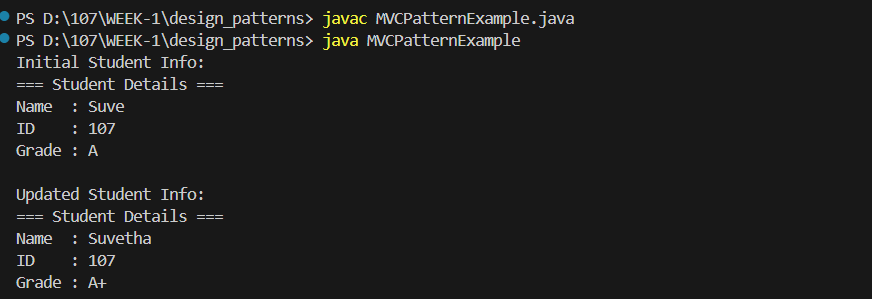
        System.out.println("Updated Student Info:");

        controller.updateView();

    }

}

**OUTPUT:**

****

**Exercise 11: Implementing Dependency Injection**

**PROGRAM:**

public class DependencyInjectionExample {

    interface CustomerRepository {

        String findCustomerById(String id);

    }

    static class CustomerRepositoryImpl implements CustomerRepository {

        @Override

        public String findCustomerById(String id) {

            if (id.equals("C001")) {

                return "Customer[ID=C001, Name=Suvetha, Email=suvetha@example.com]";

            } else if (id.equals("C002")) {

                return "Customer[ID=C002, Name=Bob, Email=bob@example.com]";

            } else {

                return "Customer not found for ID: " + id;

            }

        }

    }

    static class CustomerService {

        private final CustomerRepository repository;

        public CustomerService(CustomerRepository repository) {

            this.repository = repository;

        }

        public void displayCustomerById(String id) {

            String customerInfo = repository.findCustomerById(id);

            System.out.println(customerInfo);

        }

    }

    public static void main(String[] args) {

        CustomerRepository repo = new CustomerRepositoryImpl();

        CustomerService service = new CustomerService(repo);

        System.out.println("Fetching C001:");

        service.displayCustomerById("C001");

        System.out.println("\nFetching C002:");

        service.displayCustomerById("C002");

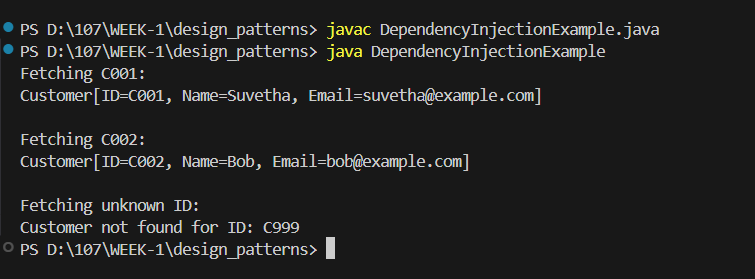
        System.out.println("\nFetching unknown ID:");

        service.displayCustomerById("C999");

    }

}

**OUTPUT:**

****