**DESIGN PATTERNS AND PRINCIPALS**

**Exercise 1: Implementing the Singleton Pattern**

**CODE:**

***File name: Logger.java***

package singleton;

public class Logger {

    // 1. private static volatile instance (guarantees visibility across threads)

    private static volatile Logger instance;

    // 2. private constructor so no other class can instantiate

    private Logger() {

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

    }

    // 3. public accessor with double‑checked locking

    public static Logger getInstance() {

        if (instance == null) {                     // First check (no locking)

            synchronized (Logger.class) {

                if (instance == null) {             // Second check (with lock)

                    instance = new Logger();

                }

            }

        }

        return instance;

    }

    // Example utility method

    public void log(String message) {

        System.out.printf("[%s] %s%n",

                java.time.LocalTime.now(), message);

    }

}

***File name: SingletonTest.java***

package singleton;

public class SingletonTest {

    public static void main(String[] args) {

        Logger log1 = Logger.getInstance();

        Logger log2 = Logger.getInstance();

        // Verify both references point to the same object

        System.out.println("Same instance? " + (log1 == log2));

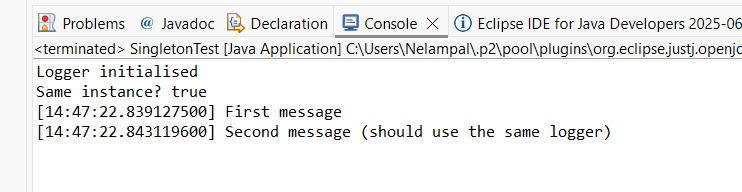
        log1.log("First message");

        log2.log("Second message (should use the same logger)");

    }

}

**OUTPUT**



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

**CODE:**

***File name: Document.java***

package factory;

public interface Document {

    void open();

}

***File name: DocumentFactory.java***

package factory;

public abstract class DocumentFactory {

    public abstract Document createDocument();

}

***File name: ExcelDocument.java***

package factory;

public class ExcelDocument implements Document {

    @Override

    public void open() {

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

    }

}

***File name: ExcelDocumentFactory.java***

package factory;

public class ExcelDocumentFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new ExcelDocument();

    }

}

***File name: pdfDocument.java***

package factory;

public class PdfDocument implements Document {

    @Override

    public void open() {

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

    }

}

***File name: pdfDocumentFactory.java***

package factory;

public class PdfDocumentFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new PdfDocument();

    }

}

***File name: WordDocument.java***

package factory;

public class WordDocument implements Document {

    @Override

    public void open() {

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

    }

}

***File name: WordDocumentFactory.java***

package factory;

public class WordDocumentFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new WordDocument();

    }

}

***File name: FactoryPatternTest.java***

package factory;

public class FactoryPatternTest {

    public static void main(String[] args) {

        DocumentFactory wordFactory = new WordDocumentFactory();

        Document wordDoc = wordFactory.createDocument();

        wordDoc.open();

        DocumentFactory pdfFactory = new PdfDocumentFactory();

        Document pdfDoc = pdfFactory.createDocument();

        pdfDoc.open();

        DocumentFactory excelFactory = new ExcelDocumentFactory();

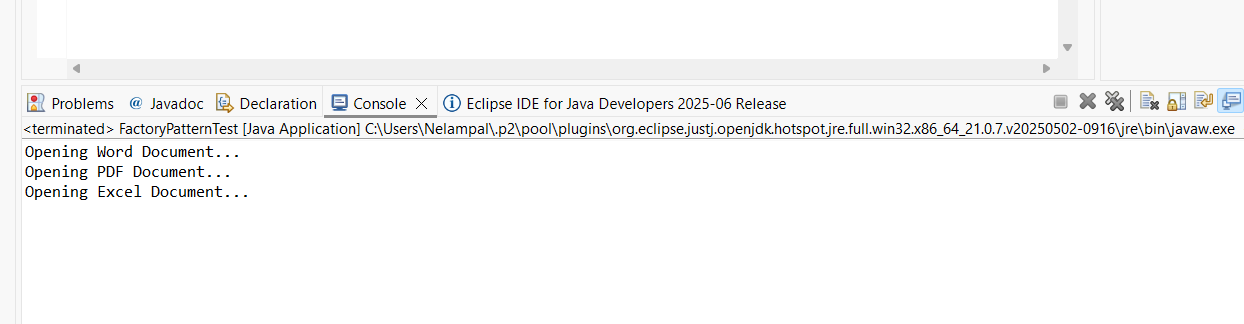
        Document excelDoc = excelFactory.createDocument();

        excelDoc.open();

    }

}

**OUTPUT**

****

**Exercise 3: Implementing the Builder Pattern**

**CODE**

***File name: Computer.java***

package builder;

public class Computer {

/\* ---------- required parts ---------- \*/

private final String cpu;

private final int ramGB;

/\* ---------- optional parts ---------- \*/

private final int ssdGB;

private final String gpu;

private final boolean wifi;

private final boolean bluetooth;

private final String operatingSystem;

/\*  private constructor takes Builder \*/

private Computer(Builder b) {

this.cpu = b.cpu;

this.ramGB = b.ramGB;

this.ssdGB = b.ssdGB;

this.gpu = b.gpu;

this.wifi = b.wifi;

this.bluetooth = b.bluetooth;

this.operatingSystem = b.operatingSystem;

}

/\* static nested Builder \*/

public static class Builder {

// required parameters

private final String cpu;

private final int ramGB;

// optional parameters – initialise with defaults

private int ssdGB = 0;

private String gpu = "Integrated";

private boolean wifi = false;

private boolean bluetooth = false;

private String operatingSystem = "No OS";

public Builder(String cpu, int ramGB) {

this.cpu = cpu;

this.ramGB = ramGB;

}

public Builder ssdGB(int ssdGB) { this.ssdGB = ssdGB; return this; }

public Builder gpu(String gpu) { this.gpu = gpu; return this; }

public Builder wifi(boolean wifi) { this.wifi = wifi; return this; }

public Builder bluetooth(boolean bluetooth) { this.bluetooth = bluetooth; return this; }

public Builder operatingSystem(String os) { this.operatingSystem = os; return this; }

/\*  factory method \*/

public Computer build() {

return new Computer(this);

}

}

@Override public String toString() {

return String.*format*(

"Computer [CPU=%s, RAM=%d GB, SSD=%d GB, GPU=%s, WiFi=%s, BT=%s, OS=%s]",

cpu, ramGB, ssdGB, gpu, wifi, bluetooth, operatingSystem);

}

}

***File name: BuilderDemo.java***

package builder;

public class BuilderDemo {

public static void main(String[] args) {

/\* Gaming rig \*/

Computer gamingPC = new Computer.Builder("Intel i9‑14900K", 32)

.ssdGB(2000)

.gpu("NVIDIA RTX 4090")

.wifi(true)

.bluetooth(true)

.operatingSystem("Windows 11 Pro")

.build();

/\* Budget office desktop \*/

Computer officePC = new Computer.Builder("AMD Ryzen 5 5600G", 16)

.ssdGB(512)

.wifi(true)

.operatingSystem("Ubuntu 24.04 LTS")

.build();

/\* Minimal bare‑metal server \*/

Computer bareMetal = new Computer.Builder("Intel Xeon E‑2378", 64)

.build();

System.*out*.println(gamingPC);

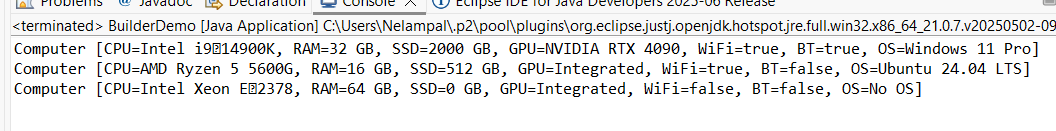
System.*out*.println(officePC);

System.*out*.println(bareMetal);

}

}

**OUTPUT**

****

**Exercise 4: Implementing the Adapter Pattern**

**CODE**

***File name: PaymentProcessor.java***

package adapter;

public interface PaymentProcessor {

void processPayment(double amount);

}

***File name: StripeGateway.java***

package adapter;

public class StripeGateway {

public void makeStripePayment(double amountInUSD) {

System.*out*.println("Payment processed through Stripe: $" + amountInUSD);

}

}

***File name: PayPalGateway.java***

package adapter;

public class PayPalGateway {

public void sendMoney(double amount) {

System.out.println("Payment sent via PayPal: $" + amount);

}

}

***File name: RazorpayGateway.java***

package adapter;

public class RazorpayGateway {

public void pay(double rupees) {

System.out.println("Payment done using Razorpay: ₹" + rupees);

}

}

***File name: StripeAdapter.java***

package adapter;

public class StripeAdapter implements PaymentProcessor {

private final StripeGateway stripe = new StripeGateway();

@Override

public void processPayment(double amount) {

stripe.makeStripePayment(amount);

}

}

***File name: PayPalAdapter.java***

package adapter;

public class PayPalAdapter implements PaymentProcessor {

private final PayPalGateway paypal = new PayPalGateway();

@Override

public void processPayment(double amount) {

paypal.sendMoney(amount);

}

}

***File name: RazorpayAdapter.java***

package adapter;

public class RazorpayAdapter implements PaymentProcessor {

private final RazorpayGateway razorpay = new RazorpayGateway();

@Override

public void processPayment(double amount) {

razorpay.pay(amount);

}

}

***File name: PaymentTest.java***

package adapter;

public class PaymentTest {

public static void main(String[] args) {

PaymentProcessor stripe = new StripeAdapter();

PaymentProcessor paypal = new PayPalAdapter();

PaymentProcessor razorpay = new RazorpayAdapter();

// All calls use the same unified interface

stripe.processPayment(100.0);

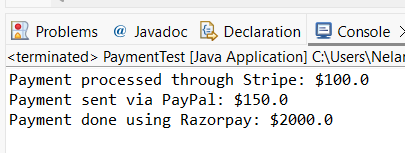
paypal.processPayment(150.0);

razorpay.processPayment(2000.0); // assume INR

}

}

**OUTPUT**



**Exercise 5: Implementing the Decorator Pattern**

**CODE**

***File name: Notifier.java***

package notifier;

public interface Notifier {

void send(String message);

}

***File name: EmailNotifier.java***

package notifier;

public class EmailNotifier implements Notifier {

@Override

public void send(String message) {

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

}

}

***File name: NotifierDecorator.java***

package notifier;

public abstract class NotifierDecorator implements Notifier {

protected Notifier wrappee;

public NotifierDecorator(Notifier notifier) {

this.wrappee = notifier;

}

@Override

public void send(String message) {

wrappee.send(message);

}

}

**File name: SMSNotifierDecorator.java**

package notifier;

public class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

@Override

public void send(String message) {

super.send(message);

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

}

}

***File name: SlackNotifierDecorator.java***

package notifier;

public class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

@Override

public void send(String message) {

super.send(message);

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

}

}

***File name: NotificationTest.java***

package notifier;

public class NotificationTest {

public static void main(String[] args) {

// Base notifier

Notifier base = new EmailNotifier();

// Add SMS

Notifier smsDecorator = new SMSNotifierDecorator(base);

// Add Slack

Notifier fullNotifier = new SlackNotifierDecorator(smsDecorator);

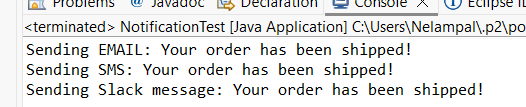
// Send using all channels

fullNotifier.send("Your order has been shipped!");

}

}

**OUTPUT**



**Exercise 6: Implementing the Proxy Pattern**

**CODE**

***File name: Image.java***

package proxy;

public interface Image {

void display();

}

***File name: RealImage.java***

package proxy;

public 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);

}

@Override

public void display() {

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

}

}

***File name: ProxyImage.java***

package proxy;

public class ProxyImage implements Image {

private final String fileName;

private RealImage realImage; // initially null

public ProxyImage(String fileName) {

this.fileName = fileName;

}

@Override

public void display() {

if (realImage == null) {

realImage = new RealImage(fileName); // lazy initialization

}

realImage.display();

}

}

***File name: ProxyPatternTest.java***

package proxy;

public class ProxyPatternTest {

public static void main(String[] args) {

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

System.out.println("Image created. Not loaded yet.");

System.out.println("\nFirst display call:");

image1.display(); // Loads and displays

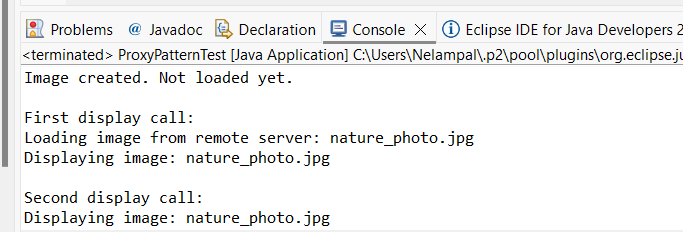
System.out.println("\nSecond display call:");

image1.display(); // Uses cached image

}

}

**OUTPUT**

****

**Exercise 7: Implementing the Observer Pattern**

**CODE**

***File name: Stock.java***

package observer;

public interface Stock {

void registerObserver(Observer observer);

void removeObserver(Observer observer);

void notifyObservers();

}

***File name: StockMarket.java***

package observer;

import java.util.ArrayList;

import java.util.List;

public class StockMarket implements Stock {

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

private double stockPrice;

public void setStockPrice(double newPrice) {

System.out.println("\nStock price updated to: ₹" + newPrice);

this.stockPrice = newPrice;

notifyObservers();

}

@Override

public void registerObserver(Observer observer) {

observers.add(observer);

System.out.println(observer.getName() + " registered.");

}

@Override

public void removeObserver(Observer observer) {

observers.remove(observer);

System.out.println(observer.getName() + " removed.");

}

@Override

public void notifyObservers() {

for (Observer observer : observers) {

observer.update(stockPrice);

}

}

}

***File name: Observer.java***

package observer;

public interface Observer {

void update(double stockPrice);

String getName(); // optional for demo/logging

}

***File name: MobileApp.java***

package observer;

public class MobileApp implements Observer {

private final String name;

public MobileApp(String name) {

this.name = name;

}

@Override

public void update(double stockPrice) {

System.out.println("[MobileApp - " + name + "] Stock price updated: ₹" + stockPrice);

}

@Override

public String getName() {

return "MobileApp - " + name;

}

}

***File name: WebApp.java***

package observer;

public class WebApp implements Observer {

private final String name;

public WebApp(String name) {

this.name = name;

}

@Override

public void update(double stockPrice) {

System.out.println("[WebApp - " + name + "] Stock price updated: ₹" + stockPrice);

}

@Override

public String getName() {

return "WebApp - " + name;

}

}

***File name: ObserverTest.java***

package observer;

public class ObserverTest {

public static void main(String[] args) {

StockMarket market = new StockMarket();

Observer mobile = new MobileApp("NSE Tracker");

Observer web = new WebApp("Finance Dashboard");

market.registerObserver(mobile);

market.registerObserver(web);

market.setStockPrice(1050.25);

market.setStockPrice(1085.50);

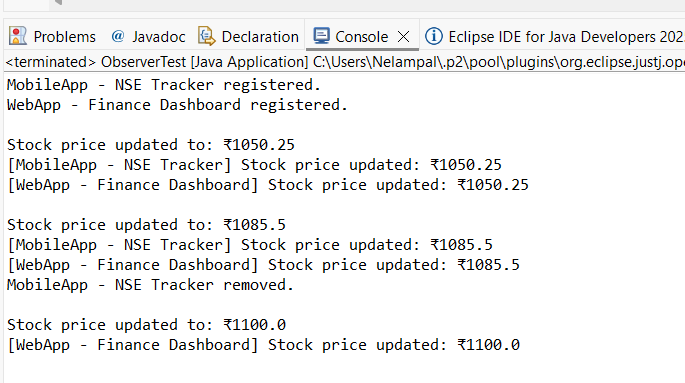
market.removeObserver(mobile);

market.setStockPrice(1100.00); // only WebApp will receive this

}

}

**OUTPUT**



**Exercise 8: Implementing the Strategy Pattern**

**CODE**

***File name: PaymentStrategy.java***

package strategy;

public interface PaymentStrategy {

void pay(double amount);

}

***File name: CreditCardPayment.java***

package strategy;

public class CreditCardPayment implements PaymentStrategy {

private final String cardNumber;

public CreditCardPayment(String cardNumber) {

this.cardNumber = cardNumber;

}

@Override

public void pay(double amount) {

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

}

}

***File name: PayPalPayment.java***

package strategy;

public class PayPalPayment implements PaymentStrategy {

private final String email;

public PayPalPayment(String email) {

this.email = email;

}

@Override

public void pay(double amount) {

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

}

}

***File name: PaymentContext.java***

package strategy;

public class PaymentContext {

private PaymentStrategy strategy;

public void setPaymentStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void processPayment(double amount) {

if (strategy == null) {

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

} else {

strategy.pay(amount);

}

}

}

***File name: StrategyPatternTest.java***

package strategy;

public class StrategyPatternTest {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

// Use Credit Card payment

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

context.processPayment(1500.00);

// Switch to PayPal payment

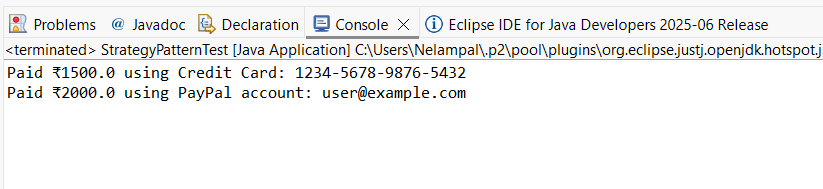
context.setPaymentStrategy(new PayPalPayment("user@example.com"));

context.processPayment(2000.00);

}

}

**OUTPUT**



**Exercise 9: Implementing the Command Pattern**

**CODE**

***File name: Command.java***

package command;

public interface Command {

void execute();

}

***File name: LightOnCommand.java***

package command;

public class LightOnCommand implements Command {

private final Light light;

public LightOnCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOn();

}

}

***File name: LightOffCommand.java***

package command;

public class LightOffCommand implements Command {

private final Light light;

public LightOffCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOff();

}

}

***File name: Light.java***

package command;

public class Light {

public void turnOn() {

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

}

public void turnOff() {

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

}

}

***File name: RemoteControl.java***

package command;

public 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.");

}

}

}

***File name: CommandPatternTest.java***

package command;

public class CommandPatternTest {

public static void main(String[] args) {

Light livingRoomLight = new Light();

Command lightOn = new LightOnCommand(livingRoomLight);

Command lightOff = new LightOffCommand(livingRoomLight);

RemoteControl remote = new RemoteControl();

remote.setCommand(lightOn);

remote.pressButton(); // Output: The light is ON

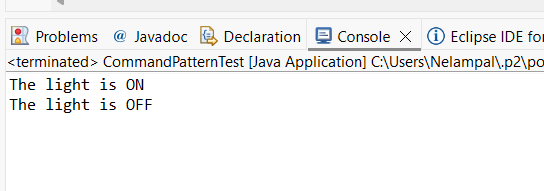
remote.setCommand(lightOff);

remote.pressButton(); // Output: The light is OFF

}

}

**OUTPUT**



**Exercise 10: Implementing the MVC Pattern**

***File name: Student.java***

package mvc;

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 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; }

}

***File name: StudentView.java***

package mvc;

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);

}

}

***File name: StudentController.java***

package mvc;

public class StudentController {

private Student model;

private StudentView view;

public StudentController(Student model, StudentView view) {

this.model = model;

this.view = view;

}

// Setters

public void setStudentName(String name) {

model.setName(name);

}

public void setStudentId(String id) {

model.setId(id);

}

public void setStudentGrade(String grade) {

model.setGrade(grade);

}

// Getters

public String getStudentName() {

return model.getName();

}

public String getStudentId() {

return model.getId();

}

public String getStudentGrade() {

return model.getGrade();

}

// Update view

public void updateView() {

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

}

}

***File name: MVCTest.java***

package mvc;

public class MVCTest {

public static void main(String[] args) {

// Step 1: Create Model

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

// Step 2: Create View

StudentView view = new StudentView();

// Step 3: Create Controller

StudentController controller = new StudentController(student, view);

// Step 4: Initial display

controller.updateView();

// Step 5: Modify data using controller

controller.setStudentName("Meena");

controller.setStudentGrade("A+");

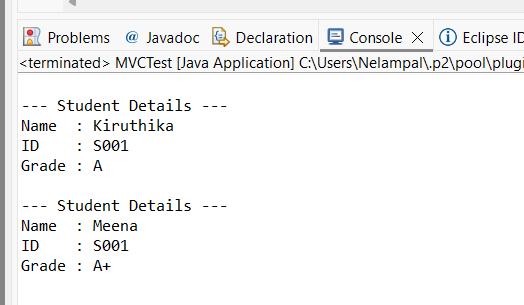
// Step 6: Updated display

controller.updateView();

}

}

**OUTPUT**



**Exercise 11: Implementing Dependency Injection**

**CODE**

***File name: Customer.java***

package di;

public record Customer(String id, String name) { }

***File name: CustomerRepository.java***

package di;

public interface CustomerRepository {

Customer findCustomerById(String id);

}

***File name: CustomerRepositoryImpl.java***

package di;

import java.util.Map;

public class CustomerRepositoryImpl implements CustomerRepository {

// Simulated persistence layer

private static final Map<String, Customer> DB = Map.of(

"C001", new Customer("C001", "Kiruthika"),

"C002", new Customer("C002", "Anand"),

"C003", new Customer("C003", "Meena"));

@Override

public Customer findCustomerById(String id) {

System.out.println("Fetching customer [" + id + "] from repository");

return DB.get(id); // null if not found

}

}

***File name: CustomerService.java***

package di;

public class CustomerService {

private final CustomerRepository repository; // dependency

// Constructor injection

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public Customer getCustomer(String id) {

return repository.findCustomerById(id);

}

}

***File name: DITest.java***

package di;

public class DITest {

public static void main(String[] args) {

// Assemble dependencies manually (would be done by a DI container in Spring/Guice)

CustomerRepository repo = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repo);

// Use the service

Customer customer = service.getCustomer("C002");

if (customer != null) {

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

} else {

System.out.println("Customer not found.");

}

}

}

**OUTPUT**

