# WEEK-1

# Design Patterns and Principles

**Exercise 1: Implementing the Singleton Pattern**

**Code**

**Java Project (Package) : SingletonPatternExample**

**Logger.java:**

public class Logger {

private static Logger uniqueInstance;

private Logger() {

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

}

public static Logger getInstance() {

if (uniqueInstance == null) {

uniqueInstance = new Logger();

}

return uniqueInstance = new Logger();

}

public void log(String message) {

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

}

}

**Test.java:**

public class LoggerTest {

public static void main(String[] args) {

Logger loggerOne = Logger.getInstance();

Logger loggerTwo = Logger.getInstance();

loggerOne.log("Starting the application.");

loggerTwo.log("Application is running.");

if (loggerOne == loggerTwo) {

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

} else {

System.out.println("Different instances exist. Singleton failed.");

}

}

}

**Output:**

A black screen with white text

AI-generated content may be incorrect.

A computer screen with white text

AI-generated content may be incorrect.

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

**Code:**

**Java Project (Package)** : FactoryPatternExample

**Document.java:**

public interface Document {

void open();

}

**Pdf.java:**

public class Pdf implements Document {

@Override

public void open() {

System.out.println("\nLaunching PDF viewer and loading your document...");

}

}

**Word.java:**

public class Word implements Document {

@Override

public void open() {

System.out.println("\nOpening Word processor with your document...");

}

}

**Execl.java:**

public class Excel implements Document {

public void open() {

System.out.println("\nStarting Excel and preparing your spreadsheet...");

}

}

**DocumentFactory.java:**

public class DocumentFactory {

public Document createDocument(String type) {

switch (type.toLowerCase()) {

case "word":

return new Word();

case "pdf":

return new Pdf();

case "excel":

return new Excel();

default:

throw new IllegalArgumentException("\nUnsupported document type: " + type);

}

}

}

**Output:**

**A screen shot of a computer

AI-generated content may be incorrect.**

**Exercise 3: Implementing the Builder Pattern**

**Code:**

**Java Project (Package) : BuilderPatternExample**

**Computer.java:**

public class Computer {

private final String processor;

private final String memory;

private final String disk;

private final String gpu;

private final String os;

private Computer(ComputerBuilder builder) {

this.processor = builder.processor;

this.memory = builder.memory;

this.disk = builder.disk;

this.gpu = builder.gpu;

this.os = builder.os;

}

public static class ComputerBuilder {

private String processor;

private String memory;

private String disk;

private String gpu;

private String os;

public ComputerBuilder withProcessor(String processor) {

this.processor = processor;

return this;

}

public ComputerBuilder withMemory(String memory) {

this.memory = memory;

return this;

}

public ComputerBuilder withDisk(String disk) {

this.disk = disk;

return this;

}

public ComputerBuilder withGraphicsCard(String gpu) {

this.gpu = gpu;

return this;

}

public ComputerBuilder withOperatingSystem(String os) {

this.os = os;

return this;

}

public Computer assemble() {

return new Computer(this);

}

}

public void showConfiguration() {

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

System.out.println("- CPU : " + processor);

System.out.println("- RAM : " + memory);

System.out.println("- Storage : " + disk);

System.out.println("- GPU : " + (gpu != null ? gpu : "None"));

System.out.println("- OS : " + (os != null ? os : "Not installed"));

}

}

**Test.java:**

public class Test {

public static void main(String[] args) {

Computer developerLaptop = new Computer.ComputerBuilder()

.withProcessor("AMD Ryzen 7")

.withMemory("16GB DDR4")

.withDisk("1TB NVMe SSD")

.withOperatingSystem("Ubuntu 22.04 LTS")

.assemble();

Computer budgetSystem = new Computer.ComputerBuilder()

.withProcessor("Intel Pentium Gold")

.withMemory("4GB")

.withDisk("500GB HDD")

.assemble();

developerLaptop.showConfiguration();

System.out.println();

budgetSystem.showConfiguration();

}

}

**A screen shot of a computer

AI-generated content may be incorrect.**

**Exercise 4: Implementing the Adapter Pattern**

**Code:**

**Java Project (Package)** : AdapterPatternExample

**PaymentProcessor.java:**

public interface PaymentProcessor {

void processPayment(double amount);

}

**NFCBasepay.java:**

public class NFCBasedPay {

public void tapToPay(double tapAmount) {

System.out.println("NFC payment completed: $" + tapAmount);

}

}

**Banktransferpay.java:**

public class BankTransferPay {

public void transferToBank(double money) {

System.out.println("Amount transferred via bank: $" + money);

}

}

**Walletpay.java:**

public class WalletPay {

public void payWithWallet(double value) {

System.out.println("Wallet payment successful: $" + value);

}

}

**PaymentAdapter.java:**

public class PaymentAdapter implements PaymentProcessor {

private Object gateway;

public PaymentAdapter(Object gateway) {

this.gateway = gateway;

}

@Override

public void processPayment(double amount) {

if (gateway instanceof WalletPay) {

((WalletPay) gateway).payWithWallet(amount);

} else if (gateway instanceof BankTransferPay) {

((BankTransferPay) gateway).transferToBank(amount);

} else if (gateway instanceof NFCBasedPay) {

((NFCBasedPay) gateway).tapToPay(amount);

} else {

System.out.println("Unsupported payment method.");

}

}

}

**PaymentTest.java:**

public class PaymentTest {

public static void main(String[] args) {

PaymentProcessor walletPayment = new PaymentAdapter(new WalletPay());

PaymentProcessor bankPayment = new PaymentAdapter(new BankTransferPay());

PaymentProcessor nfcPayment = new PaymentAdapter(new NFCBasedPay());

walletPayment.processPayment(299.99);

bankPayment.processPayment(1800.50);

nfcPayment.processPayment(75.25);

}

}

A screenshot of a computer

AI-generated content may be incorrect.

**Exercise 5: Implementing the Decorator Pattern**

**Code:**

**Java Project (Package)** : DecoratorPatternExample

**Notifier.java:**

public interface Notifier {

void send(String message);

}

**EmailNotifier.java:**

public class EmailNotifier implements Notifier {

@Override

public void send(String message) {

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

}

}

**NotifierDecorator.java:**

public abstract class NotifierDecorator implements Notifier {

protected Notifier wrappee;

public NotifierDecorator(Notifier notifier) {

this.wrappee = notifier;

}

public void send(String message) {

wrappee.send(message);

}

}

**SlackNotifierDecorator.java:**

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);

}

}

**SMSNotifierDecorator.java:**

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);

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

// Base Notifier

Notifier notifier = new EmailNotifier();

// Add SMS and Slack using decorators

notifier = new SMSNotifierDecorator(notifier);

notifier = new SlackNotifierDecorator(notifier);

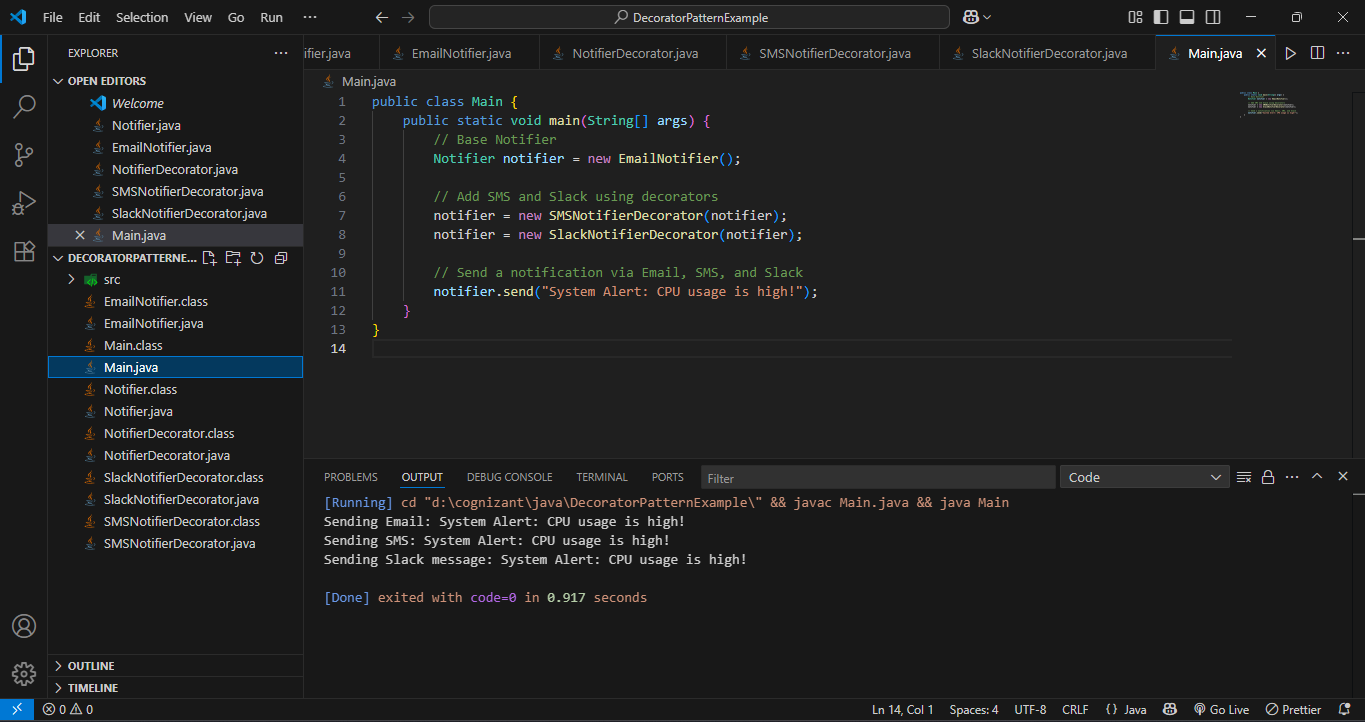
// Send a notification via Email, SMS, and Slack

notifier.send("System Alert: CPU usage is high!");

}

}

**Output:**



**Exercise 6: Implementing the Proxy Pattern**

**Code:**

**Java Project (Package)** :ProxyPatternExample

**Image.java:**

public interface Image {

void display();

}

**ProxyImage.java:**

public class ProxyImage implements Image {

private RealImage realImage;

private String filename;

public ProxyImage(String filename) {

this.filename = filename;

}

@Override

public void display() {

if (realImage == null) {

realImage = new RealImage(filename); // lazy loading

} else {

System.out.println("Using cached image.");

}

realImage.display();

}

}

**RealImage.java:**

public class RealImage implements Image {

private 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);

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

Image img1 = new ProxyImage("logo.png");

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

// First time: image is loaded from remote server

img1.display();

System.out.println();

// Second time: uses cached RealImage

img1.display();

System.out.println();

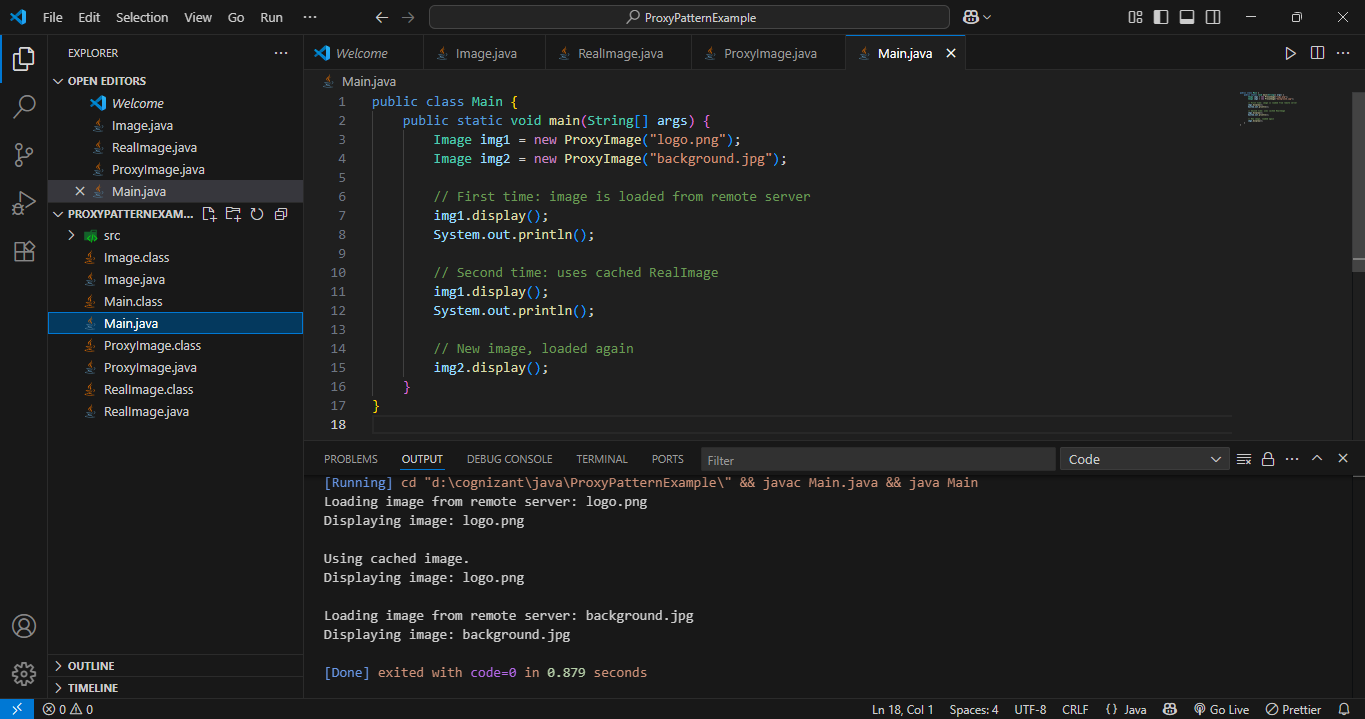
// New image, loaded again

img2.display();

}

}

**Output:**



**Exercise 7: Implementing the Observer Pattern**

**Code:**

**Java Project (Package)** :ObserverPatternExample

**Observer.java:**

public interface Observer {

void update(double stockPrice);

}

**MobileApp.java:**

public class MobileApp implements Observer {

private String appName;

public MobileApp(String appName) {

this.appName = appName;

}

@Override

public void update(double stockPrice) {

System.out.println(appName + " received stock update: $" + stockPrice);

}

}

**Stock.java:**

public interface Stock {

void registerObserver(Observer o);

void removeObserver(Observer o);

void notifyObservers();

}

**StockMarket.java:**

import java.util.ArrayList;

import java.util.List;

public class StockMarket implements Stock {

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

private double stockPrice;

public void setStockPrice(double price) {

this.stockPrice = price;

notifyObservers(); // notify on price change

}

public double 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 o : observers) {

o.update(stockPrice);

}

}

}

**WebApp.java:**

public class WebApp implements Observer {

private String appName;

public WebApp(String appName) {

this.appName = appName;

}

@Override

public void update(double stockPrice) {

System.out.println(appName + " shows stock update: $" + stockPrice);

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

StockMarket stockMarket = new StockMarket();

Observer mobile = new MobileApp("MobileApp-1");

Observer web = new WebApp("WebApp-1");

stockMarket.registerObserver(mobile);

stockMarket.registerObserver(web);

System.out.println("Setting stock price to $150.0");

stockMarket.setStockPrice(150.0);

System.out.println("\nSetting stock price to $175.0");

stockMarket.setStockPrice(175.0);

// Remove one observer

stockMarket.removeObserver(web);

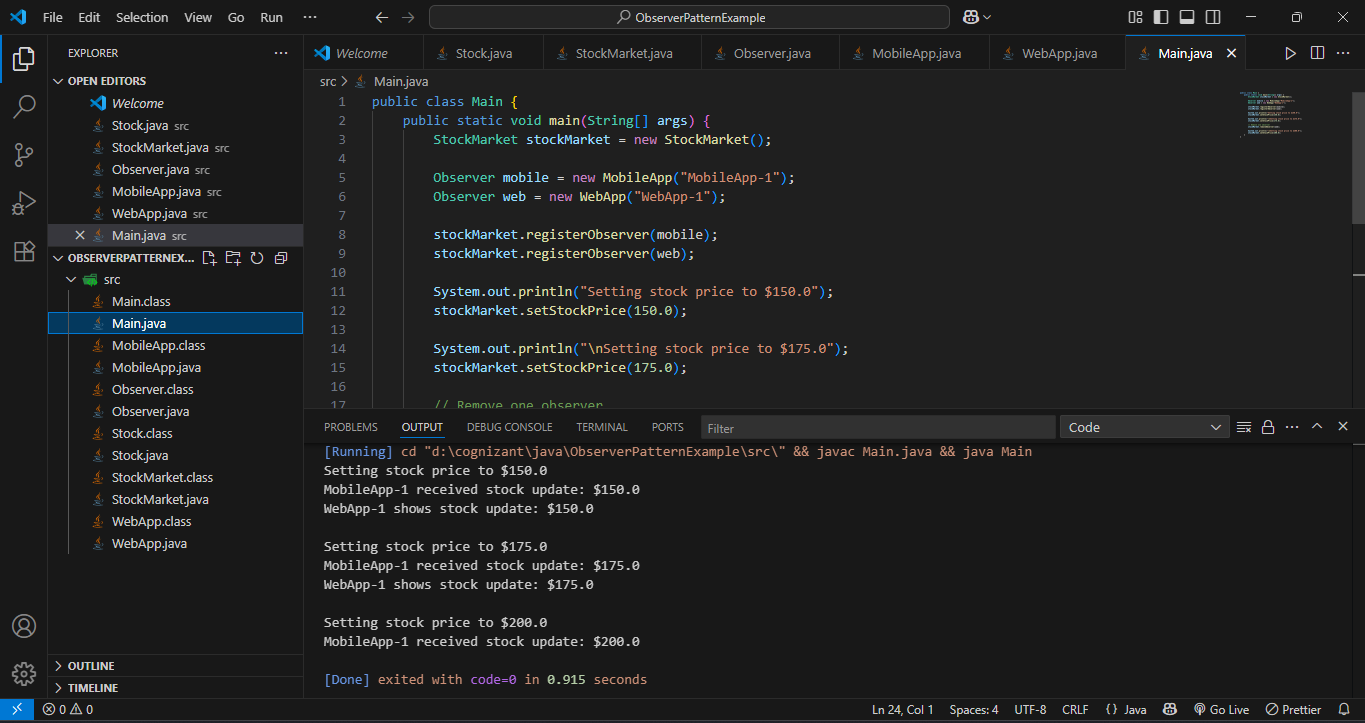
System.out.println("\nSetting stock price to $200.0");

stockMarket.setStockPrice(200.0);

}

}

**Output:**



**Exercise 8: Implementing the Strategy Pattern**

**Code:**

**Java Project (Package) :StrategyPatternExample**

**PaymentStrategy.java:**

public interface PaymentStrategy {

void pay(double 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: " + email);

}

}

**PaymentContext.java:**

public class PaymentContext {

private PaymentStrategy strategy;

public void setPaymentStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void makePayment(double amount) {

if (strategy == null) {

System.out.println("Please select a payment method.");

} else {

strategy.pay(amount);

}

}

}

**Main.java:**

public class Main {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

// Using Credit Card

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

context.makePayment(2500.00);

// Using PayPal

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

context.makePayment(3200.50);

}

}

**Output:**

A screenshot of a computer program

AI-generated content may be incorrect.

**Exercise 9: Implementing the Command Pattern**

**Code:**

**Java Project (Package) :CommandPatternExample**

**Command.java:**

public interface Command {

void execute();

}

**Light.java:**

public class Light {

public void turnOn() {

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

}

public void turnOff() {

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

}

}

**LightOnCommand.java:**

public class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOn();

}

}

**LightOffCommand.java:**

public class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOff();

}

}

**RemoteControl.java:**

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!");

}

}

}

**Main.java:**

public class Main {

public static void main(String[] args) {

Light livingRoomLight = new Light();

Command lightOn = new LightOnCommand(livingRoomLight);

Command lightOff = new LightOffCommand(livingRoomLight);

RemoteControl remote = new RemoteControl();

// Turn ON the light

remote.setCommand(lightOn);

remote.pressButton();

// Turn OFF the light

remote.setCommand(lightOff);

remote.pressButton();

}

}

**Output:**

A screenshot of a computer program

AI-generated content may be incorrect.

**Exercise 10: Implementing the MVC Pattern**

**Code:**

**Java Project (Package)** :MVCPatternExample

**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; }

}

**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);

}

}

**StudentController.java:**

public class StudentController {

private Student model;

private StudentView view;

public StudentController(Student model, StudentView view) {

this.model = model;

this.view = view;

}

public void setStudentName(String name) {

model.setName(name);

}

public void setStudentGrade(String grade) {

model.setGrade(grade);

}

public void updateView() {

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

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

// Create model and view

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

StudentView view = new StudentView();

// Controller manages the model and view

StudentController controller = new StudentController(student, view);

// Initial Display

controller.updateView();

// Update data via controller

controller.setStudentName("Bob");

controller.setStudentGrade("B+");

// Display after update

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

controller.updateView();

}

}

**Output**:

A screenshot of a computer program

AI-generated content may be incorrect.

**Exercise 11: Implementing Dependency Injection**

**Code:**

**Java Project (Package)** :DependencyInjectionPatternExample

**Customer.java:**

public class Customer {

private int id;

private String name;

public Customer(int id, String name) {

this.id = id;

this.name = name;

}

public String toString() {

return "Customer[ID=" + id + ", Name=" + name + "]";

}

}

**CustomerRepository.java:**

public interface CustomerRepository {

Customer findCustomerById(int id);

}

**CustomerRepositoryImpl.java:**

public class CustomerRepositoryImpl implements CustomerRepository {

@Override

public Customer findCustomerById(int id) {

// Return dummy data

return new Customer(id, "John Doe");

}

}

**CustomerService.java:**

public class CustomerService {

private CustomerRepository customerRepository;

// Constructor Injection

public CustomerService(CustomerRepository customerRepository) {

this.customerRepository = customerRepository;

}

public void displayCustomer(int id) {

Customer customer = customerRepository.findCustomerById(id);

System.out.println(customer);

}

}

**Main.java:**

public class Main {

public static void main(String[] args) {

// Create repository

CustomerRepository repository = new CustomerRepositoryImpl();

// Inject into service

CustomerService service = new CustomerService(repository);

// Call service method

service.displayCustomer(101);

}

}

Output:

A screenshot of a computer program

AI-generated content may be incorrect.