**Exercise 1: Implementing the Singleton Pattern**

**Scenario:**

You need to ensure that a logging utility class in your application has only one instance throughout the application lifecycle to ensure consistent logging.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **SingletonPatternExample**.
2. **Define a Singleton Class:**
   * Create a class named Logger that has a private static instance of itself.
   * Ensure the constructor of Logger is private.
   * Provide a public static method to get the instance of the Logger class.
3. **Implement the Singleton Pattern:**
   * Write code to ensure that the Logger class follows the Singleton design pattern.
4. **Test the Singleton Implementation:**
   * Create a test class to verify that only one instance of Logger is created and used across the application.

**Logger.java:**

package Singleton;

public class Logger {

private static Logger *instance*;

private Logger()

{

//private constructor

}

public static Logger getInstance()

{

if(*instance*==null)

{

*instance*=new Logger();

}

return *instance*;

}

public void log(String message)

{

System.*out*.println("Logging message: "+message);

}

}

**Test.java:**

package Singleton;

public class Test {

public static void main(String[] args) {

Logger log1 = Logger.*getInstance*();

Logger log2 = Logger.*getInstance*();

log1.log("Logging log1");

log2.log("Logging log2");

if(log1==log2)

{

System.*out*.println("Singleton Pattern");

}

else

{

System.*out*.println("Not a Singleton Pattern");

}

}

}

A screenshot of a computer

AI-generated content may be incorrect.

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

**Scenario:**

You are developing a document management system that needs to create different types of documents (e.g., Word, PDF, Excel). Use the Factory Method Pattern to achieve this.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **FactoryMethodPatternExample**.
2. **Define Document Classes:**
   * Create interfaces or abstract classes for different document types such as **WordDocument**, **PdfDocument**, and **ExcelDocument**.
3. **Create Concrete Document Classes:**
   * Implement concrete classes for each document type that implements or extends the above interfaces or abstract classes.
4. **Implement the Factory Method:**
   * Create an abstract class **DocumentFactory** with a method **createDocument()**.
   * Create concrete factory classes for each document type that extends DocumentFactory and implements the **createDocument()** method.
5. **Test the Factory Method Implementation:**
   * Create a test class to demonstrate the creation of different document types using the factory method.

**Document.java:**

package FactoryMethod;

public interface Document {

void open();

}

**DocumentFactory.java:**

package FactoryMethod;

public abstract class DocumentFactory {

public abstract Document createDocument();

}

**ExcelDocument.java:**

package FactoryMethod;

public class ExcelDocument implements Document {

public void open() {

System.*out*.println("EXCEL Documnet");

}

}

**ExcelDocumnetFactory.java:**

package FactoryMethod;

public class ExcelDocumnetFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

**PdfDocument.java:**

package FactoryMethod;

public class PdfDocument implements Document {

public void open() {

System.*out*.println("PDF Documnet");

}

}

**PdfDocumnetFactory.java:**

package FactoryMethod;

public class PdfDocumnetFactory extends DocumentFactory{

public Document createDocument() {

return new PdfDocument();

}

}

**WordDocument.java:**

package FactoryMethod;

public class WordDocument implements Document {

public void open() {

System.*out*.println("WORD Documnet");

}

}

**WordDocumentFactory.java:**

package FactoryMethod;

public class WordDocumentFactory extends DocumentFactory{

public Document createDocument() {

return new WordDocument();

}

}

**FactoryMethodTest.java:**

package FactoryMethod;

public class FactoryMethodTest {

public static void main(String[] arg) {

//Word

DocumentFactory word = new WordDocumentFactory();

Document wordDOC = word.createDocument();

wordDOC.open();

//PDF

DocumentFactory pdf = new PdfDocumnetFactory();

Document pdfDOC = pdf.createDocument();

pdfDOC.open();

//EXCEL

DocumentFactory excel = new ExcelDocumnetFactory();

Document excelDOC = excel.createDocument();

excelDOC.open();

}

}

A screenshot of a computer

AI-generated content may be incorrect.

**Exercise 3: Implementing the Builder Pattern**

**Scenario:**

You are developing a system to create complex objects such as a Computer with multiple optional parts. Use the Builder Pattern to manage the construction process.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **BuilderPatternExample**.
2. **Define a Product Class:**
   * Create a class **Computer** with attributes like **CPU**, **RAM**, **Storage**, etc.
3. **Implement the Builder Class:**
   * Create a static nested Builder class inside Computer with methods to set each attribute.
   * Provide a **build()** method in the Builder class that returns an instance of Computer.
4. **Implement the Builder Pattern:**
   * Ensure that the **Computer** class has a private constructor that takes the **Builder** as a parameter.
5. **Test the Builder Implementation:**
   * Create a test class to demonstrate the creation of different configurations of Computer using the Builder pattern.

**Computer.java:**

package BuilderPattern;

import java.security.KeyStore.Builder;

public class Computer {

private String cpu;

private String ram;

private String storage;

private Computer(Builder builder) {

this.cpu = builder.cpu;

this.ram = builder.ram;

this.storage = builder.storage;

}

public static class Builder {

private String cpu;

private String ram;

private String storage;

public Builder(String cpu, String ram, String storage) {

this.cpu = cpu;

this.ram = ram;

this.storage = storage;

}

public Computer build() {

return new Computer(this);

}

}

public void Display() {

System.*out*.println("CPU: " + cpu + " "+"RAM: "+ram+" "+"Storage: "+storage);

}

}

**BuilderPatternTest.java:**

package BuilderPattern;

public class BuilderPatternTest {

public static void main(String[] args) {

//com

Computer com = new Computer.Builder("Foxconn", "12GB", "256GB").build();

com.Display();

//com1

Computer com1 = new Computer.Builder("Intel", "32GB", "1TB").build();

com1.Display();

}

}

A screenshot of a computer

AI-generated content may be incorrect.

**Exercise 4: Implementing the Adapter Pattern**

**Scenario:**

You are developing a payment processing system that needs to integrate with multiple third-party payment gateways with different interfaces. Use the Adapter Pattern to achieve this.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **AdapterPatternExample**.
2. **Define Target Interface:**
   * Create an interface **PaymentProcessor** with methods like **processPayment()**.
3. **Implement Adaptee Classes:**
   * Create classes for different payment gateways with their own methods.
4. **Implement the Adapter Class:**
   * Create an adapter class for each payment gateway that implements PaymentProcessor and translates the calls to the gateway-specific methods.
5. **Test the Adapter Implementation:**
   * Create a test class to demonstrate the use of different payment gateways through the adapter.

**PaymentProcessor.java:**

package AdapterPattern;

public interface PaymentProcessor {

void processPayment(double amount);

}

**HostedPayment.java:**

package AdapterPattern;

public class HostedPayment {

public void makePayment(double amount)

{

System.*out*.println("Hosted Pay: Payment of $" + amount);

}

}

**HostedPaymentAdapter.java:**

package AdapterPattern;

public class HostedPaymentAdapter implements PaymentProcessor {

private HostedPayment hosted;

public HostedPaymentAdapter(HostedPayment hosted)

{

this.hosted = hosted;

}

public void processPayment(double amount)

{

hosted.makePayment(amount);

}

}

**LocalBankPayment.java:**

package AdapterPattern;

public class LocalBankPayment {

public void depositPayment(double amount) {

System.*out*.println("Local Bank Deposit: Deposit of $" + amount);

}

}

**LocalBankPaymentAdapter.java:**

package AdapterPattern;

public class LocalBankPaymentAdapter implements PaymentProcessor{

private LocalBankPayment local;

public LocalBankPaymentAdapter(LocalBankPayment local)

{

this.local = local;

}

public void processPayment(double amount)

{

local.depositPayment(amount);

}

}

**AdapterPatternTest.java:**

package AdapterPattern;

public class AdapterPatternTest {

public static void main(String[] args) {

//Hosted Payment

HostedPayment hosted = new HostedPayment();

PaymentProcessor hostedProcessor = new HostedPaymentAdapter(hosted);

hostedProcessor.processPayment(250.0);

//Local Bank Deposit

LocalBankPayment local = new LocalBankPayment();

PaymentProcessor localProcessor = new LocalBankPaymentAdapter(local);

localProcessor.processPayment(25000.0);

}

}

A screenshot of a computer

AI-generated content may be incorrect.

**Exercise 5: Implementing the Decorator Pattern**

**Scenario:**

You are developing a notification system where notifications can be sent via multiple channels (e.g., Email, SMS). Use the Decorator Pattern to add functionalities dynamically.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **DecoratorPatternExample**.
2. **Define Component Interface:**
   * Create an interface **Notifier** with a method **send()**.
3. **Implement Concrete Component:**
   * Create a class **EmailNotifier** that implements Notifier.
4. **Implement Decorator Classes:**
   * Create abstract decorator class **NotifierDecorator** that implements **Notifier** and holds a reference to a **Notifier** object.
   * Create concrete decorator classes like **SMSNotifierDecorator**, **SlackNotifierDecorator** that extend **NotifierDecorator**.
5. **Test the Decorator Implementation:**
   * Create a test class to demonstrate sending notifications via multiple channels using decorators.

**Notifier.java:**

package DecoratorPattern;

public interface Notifier {

void send(String message);

}

**EmailNotifier.java:**

package DecoratorPattern;

public class EmailNotifier implements Notifier {

public void send(String message) {

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

}

}

**NotifierDecorator.java:**

package DecoratorPattern;

public abstract class NotifierDecorator implements Notifier {

protected Notifier wrapNotifier;

public NotifierDecorator(Notifier notifier) {

this.wrapNotifier = notifier;

}

public void send(String message) {

wrapNotifier.send(message);

}

}

**SMSNotifierDecorator.java:**

package DecoratorPattern;

public 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("SMS: " + message);

}

}

**SlackNotifierDecorator.java:**

package DecoratorPattern;

public 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("Slack message: " + message);

}

}

**DecoratorPatternTest.java:**

package DecoratorPattern;

public class DecoratorPatternTest {

public static void main(String[] args) {

Notifier notifier = new EmailNotifier();

Notifier smsNotifier = new SMSNotifierDecorator(notifier);

Notifier slackAndSMSNotifier = new SlackNotifierDecorator(smsNotifier);

slackAndSMSNotifier.send("Sending Message!");

}

}

A screenshot of a computer

AI-generated content may be incorrect.

**Exercise 6: Implementing the Proxy Pattern**

**Scenario:**

You are developing an image viewer application that loads images from a remote server. Use the Proxy Pattern to add lazy initialization and caching.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **ProxyPatternExample**.
2. **Define Subject Interface:**
   * Create an interface Image with a method **display()**.
3. **Implement Real Subject Class:**
   * Create a class **RealImage** that implements Image and loads an image from a remote server.
4. **Implement Proxy Class:**
   * Create a class **ProxyImage** that implements Image and holds a reference to RealImage.
   * Implement lazy initialization and caching in **ProxyImage**.
5. **Test the Proxy Implementation:**
   * Create a test class to demonstrate the use of **ProxyImage** to load and display images.

**Image.java:**

package ProxyPattern;

public interface Image {

void display();

}

**RealImage.java**

package ProxyPattern;

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);

}

public void display() {

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

}

}

**ProxyImage.java:**

package ProxyPattern;

import java.util.HashMap;

import java.util.Map;

public class ProxyImage implements Image{

private String filename;

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

public ProxyImage(String filename) {

this.filename = filename;

}

public void display() {

RealImage realImage = cache.get(filename);

if (realImage == null) {

System.out.println("Image not in cache.Loading!");

realImage = new RealImage(filename);

cache.put(filename, realImage);

} else {

System.out.println("Image loaded from cache.");

}

realImage.display();

}

}

**ProxyPatternTest.java:**

package ProxyPattern;

public class ProxyPatternTest {

public static void main(String[] args) {

Image image1 = new ProxyImage("photo1.jpg");

Image image2 = new ProxyImage("photo2.jpg");

System.*out*.println("Display photo1.jpg:");

image1.display();

System.*out*.println("Display photo2.jpg:");

image2.display();

}

}

A screenshot of a computer

AI-generated content may be incorrect.

**Exercise 7: Implementing the Observer Pattern**

**Scenario:**

You are developing a stock market monitoring application where multiple clients need to be notified whenever stock prices change. Use the Observer Pattern to achieve this.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **ObserverPatternExample**.
2. **Define Subject Interface:**
   * Create an interface **Stock** with methods to **register**, **deregister**, and **notify** observers.
3. **Implement Concrete Subject:**
   * Create a class **StockMarket** that implements **Stock** and maintains a list of observers.
4. **Define Observer Interface:**
   * Create an interface Observer with a method **update().**
5. **Implement Concrete Observers:**
   * Create classes **MobileApp**, **WebApp** that implement Observer.
6. **Test the Observer Implementation:**
   * Create a test class to demonstrate the registration and notification of observers.

**Stock.java:**

package ObserverPattern;

public interface Stock {

void addObserver(Observer o);

void removeObserver(Observer o);

void notifyAllObservers();

}

**StockMarket.java:**

package ObserverPattern;

import java.util.ArrayList;

import java.util.List;

public class StockMarket implements Stock {

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

private String stockName;

private double price;

public void addObserver(Observer o) {

observers.add(o);

}

public void removeObserver(Observer o) {

observers.remove(o);

}

public void notifyAllObservers() {

for (Observer o : observers) {

o.update(stockName, price);

}

}

public void setStock(String stockName, double price) {

this.stockName = stockName;

this.price = price;

notifyAllObservers();

}

}

**Observer.java:**

package ObserverPattern;

public interface Observer {

void update(String stockName, double stockPrice);

}

**MobileApp.java:**

package ObserverPattern;

public class MobileApp implements Observer{

public void update(String stockName, double price)

{

System.*out*.println("Mobile App: " + stockName + " price is now $" + price);

}

}

**WebApp.java:**

package ObserverPattern;

public class WebApp implements Observer{

public void update(String stockName, double price) {

System.*out*.println("Web App: " + stockName + " price is now $" + price);

}

}

**ObserverPatternTest.java:**

package ObserverPattern;

public class ObserverPatternTest {

public static void main(String[] args) {

StockMarket stockMarket = new StockMarket();

MobileApp mobileApp = new MobileApp();

WebApp webApp = new WebApp();

stockMarket.addObserver(mobileApp);

stockMarket.addObserver(webApp);

stockMarket.setStock("Hyundai", 3400.50);

stockMarket.setStock("Toyota", 1500.75);

}

}

A screenshot of a computer

AI-generated content may be incorrect.

**Exercise 8: Implementing the Strategy Pattern**

**Scenario:**

You are developing a payment system where different payment methods (e.g., Credit Card, PayPal) can be selected at runtime. Use the Strategy Pattern to achieve this.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **StrategyPatternExample**.
2. **Define Strategy Interface:**
   * Create an interface PaymentStrategy with a method **pay()**.
3. **Implement Concrete Strategies:**
   * Create classes **CreditCardPayment**, **PayPalPayment** that implement **PaymentStrategy**.
4. **Implement Context Class:**
   * Create a class **PaymentContext** that holds a reference to **PaymentStrategy** and a method to execute the strategy.
5. **Test the Strategy Implementation:**
   * Create a test class to demonstrate selecting and using different payment strategies.

**PaymentStrategy.java**

package StrategyPattern;

public interface PaymentStrategy {

void pay(double amount);

}

**CreditCardPayment.java:**

package StrategyPattern;

public class CreditCardPayment implements PaymentStrategy{

private String cardNumber;

public CreditCardPayment(String cardNumber)

{

this.cardNumber = cardNumber;

}

public void pay(double amount)

{

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

}

}

**PayPalPayment.java:**

package StrategyPattern;

public 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);

}

}

**PaymentContext.java:**

package StrategyPattern;

public class PaymentContext {

private PaymentStrategy paymentStrategy;

public void setPaymentStrategy(PaymentStrategy paymentStrategy) {

this.paymentStrategy = paymentStrategy;

}

public void payAmount(double amount) {

if (paymentStrategy == null) {

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

} else {

paymentStrategy.pay(amount);

}

}

}

**StrategyPatternTest.java:**

package StrategyPattern;

public class StrategyPatternTest {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

//Credit Card Payment

context.setPaymentStrategy(new CreditCardPayment("7864-8573"));

context.payAmount(5000.00);

//Paypal Payment

context.setPaymentStrategy(new PayPalPayment("abc@gmail.com"));

context.payAmount(250.00);

}

}

A screenshot of a computer

AI-generated content may be incorrect.

**Exercise 9: Implementing the Command Pattern**

**Scenario:** You are developing a home automation system where commands can be issued to turn devices on or off. Use the Command Pattern to achieve this.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **CommandPatternExample**.
2. **Define Command Interface:**
   * Create an interface Command with a method **execute()**.
3. **Implement Concrete Commands:**
   * Create classes **LightOnCommand**, **LightOffCommand** that implement Command.
4. **Implement Invoker Class:**
   * Create a class **RemoteControl** that holds a reference to a Command and a method to execute the command.
5. **Implement Receiver Class:**
   * Create a class **Light** with methods to turn on and off.
6. **Test the Command Implementation:**
   * Create a test class to demonstrate issuing commands using the **RemoteControl**.

**Command.java:**

package CommandPattern;

public interface Command {

void execute();

}

**LightOnCommand.java:**

package CommandPattern;

public class LightOnCommand implements Command{

private Light light;

public LightOnCommand(Light light)

{

this.light = light;

}

public void execute()

{

light.turnOn();

}

}

**LightOffCommand.java:**

package CommandPattern;

public class LightOffCommand implements Command{

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOff();

}

}

**Light.java:**

package CommandPattern;

public class Light {

public void turnOn() {

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

}

public void turnOff() {

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

}

}

**RemoteControl.java:**

package CommandPattern;

public class RemoteControl {

private Command command;

public void setCommand(Command command)

{

this.command = command;

}

public void pressButton()

{

command.execute();

}

}

**CommandPatternTest.java:**

package CommandPattern;

public class CommandPatternTest {

public static void main(String[] args) {

Light light = new Light();

Command lightOn = new LightOnCommand(light);

Command lightOff = new LightOffCommand(light);

RemoteControl remote = new RemoteControl();

//Turning ON the Light

remote.setCommand(lightOn);

remote.pressButton();

//Turning OFF the Light

remote.setCommand(lightOff);

remote.pressButton();

}

}

A screenshot of a computer

AI-generated content may be incorrect.

**Exercise 10: Implementing the MVC Pattern**

**Scenario:**

You are developing a simple web application for managing student records using the MVC pattern.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **MVCPatternExample**.
2. **Define Model Class:**
   * Create a class **Student** with attributes like **name, id, and grade**.
3. **Define View Class:**
   * Create a class **StudentView** with a method **displayStudentDetails()**.
4. **Define Controller Class:**
   * Create a class **StudentController** that handles the communication between the model and the view.
5. **Test the MVC Implementation:**
   * Create a main class to demonstrate creating a **Student**, updating its details using **StudentController**, and displaying them using **StudentView**.

**Student.java:**

package MVCPattern;

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;

}

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;

}

}

**StudentView.java:**

package MVCPattern;

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:**

package MVCPattern;

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 setStudentId(String id) {

model.setId(id);

}

public void setStudentGrade(String grade) {

model.setGrade(grade);

}

public String getStudentName() {

return model.getName();

}

public String getStudentId() {

return model.getId();

}

public String getStudentGrade() {

return model.getGrade();

}

public void updateView() {

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

}

}

**MVCPatternTest.java:**

package MVCPattern;

public class MVCPatternTest {

public static void main(String[] args) {

Student student = new Student("Sai", "101", "A");

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

controller.updateView();

controller.setStudentName("Jane Smith");

controller.setStudentGrade("A+");

System.*out*.println("After updating details:");

controller.updateView();

}

}

A screenshot of a computer

AI-generated content may be incorrect.

**Exercise 11: Implementing Dependency Injection**

**Scenario:**

You are developing a customer management application where the service class depends on a repository class. Use Dependency Injection to manage these dependencies.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **DependencyInjectionExample**.
2. **Define Repository Interface:**
   * Create an interface **CustomerRepository** with methods like **findCustomerById()**.
3. **Implement Concrete Repository:**
   * Create a class **CustomerRepositoryImpl** that implements **CustomerRepository**.
4. **Define Service Class:**
   * Create a class **CustomerService** that depends on **CustomerRepository**.
5. **Implement Dependency Injection:**
   * Use constructor injection to inject **CustomerRepository** into **CustomerService**.
6. **Test the Dependency Injection Implementation:**
   * Create a main class to demonstrate creating a **CustomerService** with **CustomerRepositoryImpl** and using it to find a customer.

**CustomerRepository.java:**

package DependencyInjection;

public interface CustomerRepository {

Customer findCustomerById(int id);

}

**CustomerRepositoryImpl.java:**

package DependencyInjection;

public class CustomerRepositoryImpl implements CustomerRepository{

public Customer findCustomerById(int id) {

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

}

}

**CustomerService.java:**

package DependencyInjection;

public class CustomerService {

private CustomerRepository customerRepository;

public CustomerService(CustomerRepository customerRepository) {

this.customerRepository = customerRepository;

}

public void displayCustomer(int id) {

Customer customer = customerRepository.findCustomerById(id);

System.*out*.println("Customer ID: " + customer.getId());

System.*out*.println("Customer Name: " + customer.getName());

}

}

**Customer.java:**

package DependencyInjection;

public class Customer {

private int id;

private String name;

public Customer(int id, String name) {

this.id = id;

this.name = name;

}

public int getId() {

return id;

}

public String getName() {

return name;

}

}

**DependencyInjectionTest.java:**

package DependencyInjection;

public class DependencyInjectionTest {

public static void main(String args[]) {

CustomerRepository repository = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repository);

service.displayCustomer(101);

}

}

A screenshot of a computer

AI-generated content may be incorrect.