**DESIGN PRINCIPLES**

**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.

Solution:

class Logger{

private static Logger instance;

private Logger(){

//private Logger constructor

}

public static Logger getInstance(){

if(instance==null) instance=new Logger();

return instance;

}

}

public class Main{

public static void main(String[] args){

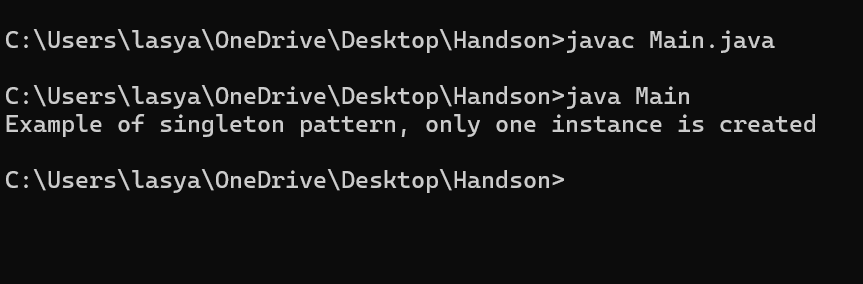
Logger l1 = Logger.getInstance();

Logger l2 = Logger.getInstance();

if(l1==l2) System.out.println("Example of singleton pattern, only one instance is created");

}

}

Output: 

**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.

Solution:

public class Main {

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();

}

}

interface Document {

void open();

}

class WordDocument implements Document {

public void open() {

System.out.println("Opening a Word document.");

}

}

class PdfDocument implements Document {

public void open() {

System.out.println("Opening a PDF document.");

}

}

class ExcelDocument implements Document {

public void open() {

System.out.println("Opening an Excel document.");

}

}

abstract class DocumentFactory {

public abstract Document createDocument();

}

class WordDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

class PdfDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

class ExcelDocumentFactory extends DocumentFactory {

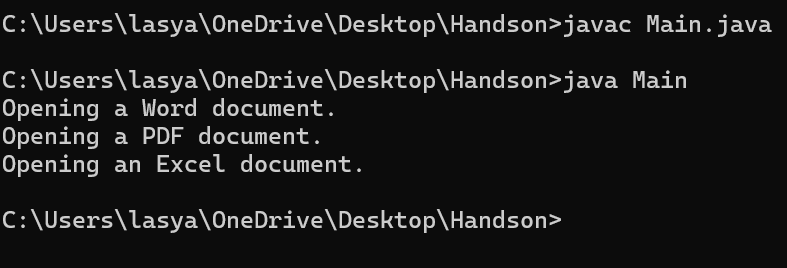
public Document createDocument() {

return new ExcelDocument();

}

}

Output:



**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.

Solution:

public class Main {

public static void main(String[] args) {

Computer gamingPC = new Computer.Builder()

.setCPU("Intel Core i9")

.setRAM("32GB")

.setStorage("1TB SSD")

.setGPU("NVIDIA RTX 4080")

.build();

gamingPC.displayConfig();

Computer officePC = new Computer.Builder()

.setCPU("Intel Core i5")

.setRAM("8GB")

.setStorage("512GB SSD")

.build();

officePC.displayConfig();

}

}

class Computer {

private final String CPU;

private final String RAM;

private final String Storage;

private final String GPU; // Optional

private Computer(Builder builder) {

this.CPU = builder.CPU;

this.RAM = builder.RAM;

this.Storage = builder.Storage;

this.GPU = builder.GPU;

}

public static class Builder {

private String CPU;

private String RAM;

private String Storage;

private String GPU;

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 setGPU(String GPU) {

this.GPU = GPU;

return this;

}

public Computer build() {

return new Computer(this);

}

}

public void displayConfig() {

System.out.println("\nComputer Configuration");

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

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

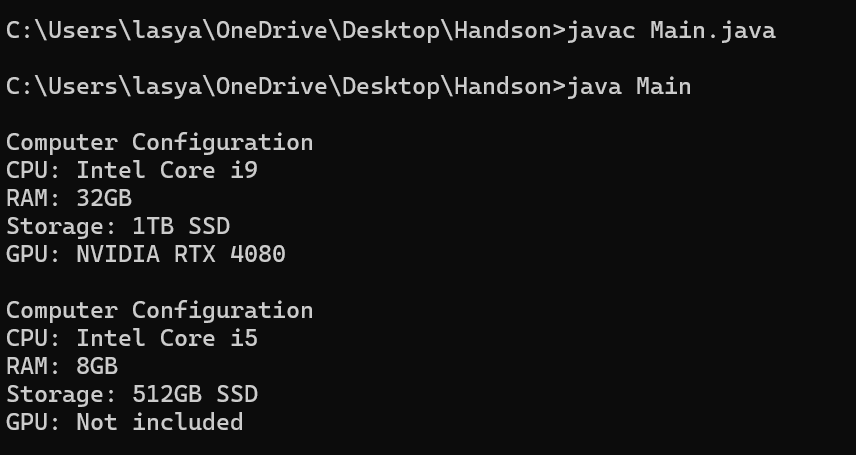
System.out.println("Storage: " + Storage);

System.out.println("GPU: " + (GPU != null ? GPU : "Not included"));

}

}

Output:



**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.

Solution:

interface PaymentProcessor {

void processPayment();

}

class OldGateway {

public void makePayment() {

System.out.println("Paid using OldGateway.");

}

}

class OldGatewayAdapter implements PaymentProcessor {

private OldGateway oldGateway;

public OldGatewayAdapter() {

this.oldGateway = new OldGateway();

}

@Override

public void processPayment() {

oldGateway.makePayment();

}

}

public class Main {

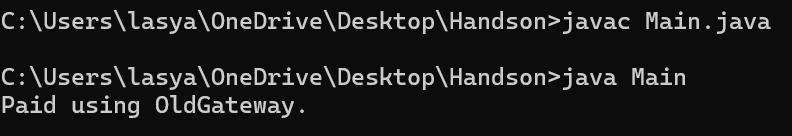
public static void main(String[] args) {

PaymentProcessor paymentProcessor = new OldGatewayAdapter();

paymentProcessor.processPayment(); }

}

Output:



**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.

Solution:

interface Notifier {

void send();

}

class EmailNotifier implements Notifier {

@Override

public void send() {

System.out.println("Sending Email notification.");

}

}

abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

@Override

public void send() {

notifier.send();

}

}

class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

@Override

public void send() {

super.send();

System.out.println("Sending SMS notification.");

}

}

class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

@Override

public void send() {

super.send();

System.out.println("Sending Slack notification.");

}

}

public class Main {

public static void main(String[] args) {

Notifier notifier = new EmailNotifier();

notifier = new SMSNotifierDecorator(notifier);

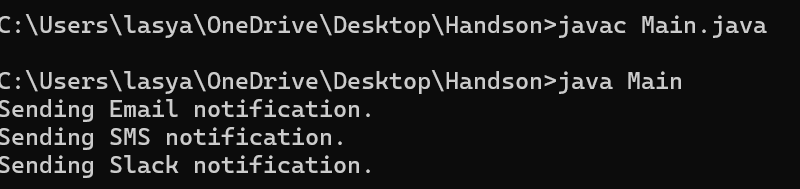
notifier = new SlackNotifierDecorator(notifier);

notifier.send();

}

}

Output:



**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.

Solution:

interface Image {

void display();

}

class RealImage implements Image {

private String filename;

public RealImage(String filename) {

this.filename = filename;

loadImageFromRemoteServer();

}

private void loadImageFromRemoteServer() {

System.out.println("Loading image from remote server " );

}

@Override

public void display() {

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

}

}

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);

}

realImage.display();

}

}

public class Main {

public static void main(String[] args) {

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

image1.display();

image1.display();

System.out.println();

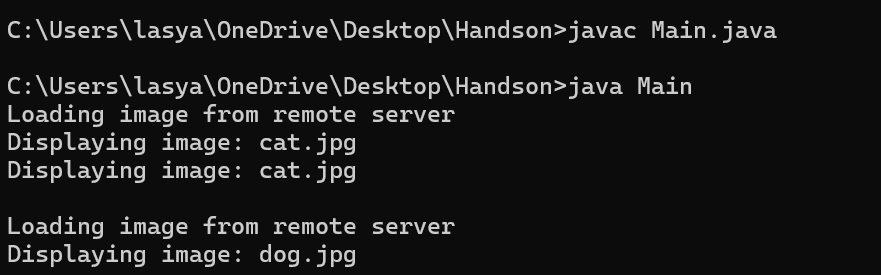
Image image2 = new ProxyImage("dog.jpg");

image2.display();

}

}

Output:



**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.

Solution:

import java.util.\*;

interface Observer {

void update(String stockName, double price);

}

interface Stock {

void register(Observer o);

void deregister(Observer o);

void notifyObservers();

}

class StockMarket implements Stock {

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

private String stockName;

private double stockPrice;

public void setStockData(String stockName, double stockPrice) {

this.stockName = stockName;

this.stockPrice = stockPrice;

notifyObservers();

}

@Override

public void register(Observer o) {

observers.add(o);

}

@Override

public void deregister(Observer o) {

observers.remove(o);

}

@Override

public void notifyObservers() {

for (Observer observer : observers) {

observer.update(stockName, stockPrice);

}

}

}

class MobileApp implements Observer {

@Override

public void update(String stockName, double price) {

System.out.println("MobileApp - " + stockName + ": $" + price);

}

}

class WebApp implements Observer {

@Override

public void update(String stockName, double price) {

System.out.println("WebApp - " + stockName + ": $" + price);

}

}

public class Main {

public static void main(String[] args) {

StockMarket stockMarket = new StockMarket();

Observer mobile = new MobileApp();

Observer web = new WebApp();

stockMarket.register(mobile);

stockMarket.register(web);

stockMarket.setStockData("AAPL", 189.25);

stockMarket.setStockData("GOOGL", 2800.50);

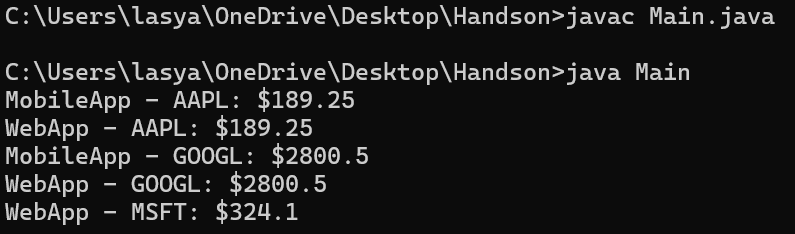
stockMarket.deregister(mobile);

stockMarket.setStockData("MSFT", 324.10);

}

}

Output:



**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.

Solution:

interface PaymentStrategy {

void pay();

}

class CreditCardPayment implements PaymentStrategy {

@Override

public void pay() {

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

}

}

class PayPalPayment implements PaymentStrategy {

@Override

public void pay() {

System.out.println("Paid $ using PayPal.");

}

}

class PaymentContext {

private PaymentStrategy strategy;

public PaymentContext(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void setPaymentStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void executePayment() {

strategy.pay();

}

}

public class Main {

public static void main(String[] args) {

PaymentContext context = new PaymentContext(new CreditCardPayment());

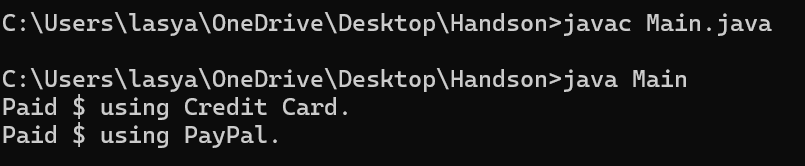
context.executePayment();

context.setPaymentStrategy(new PayPalPayment());

context.executePayment();

}

}



**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**.

Solution:

interface Command {

void execute();

}

class Light {

public void on() {

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

}

public void off() {

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

}

}

class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.on();

}

}

class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.off();

}

}

class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

command.execute();

}

}

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();

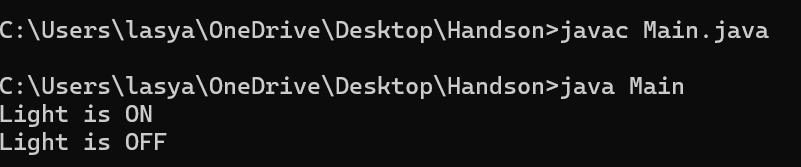
remote.setCommand(lightOn);

remote.pressButton();

remote.setCommand(lightOff);

remote.pressButton(); }

}



**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**.

Solution:

class Student {

private String name;

private int id;

private String grade;

public Student(String name, int id, String grade) {

this.name = name;

this.id = id;

this.grade = grade;

}

public String getName() { return name; }

public int getId() { return id; }

public String getGrade() { return grade; }

public void setName(String name) { this.name = name; }

public void setId(int id) { this.id = id; }

public void setGrade(String grade) { this.grade = grade; }

}

class StudentView {

public void displayStudentDetails(String name, int id, String grade) {

System.out.println("Student Details:");

System.out.println("Name: " + name);

System.out.println("ID: " + id);

System.out.println("Grade: " + grade);

}

}

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(int id) {

student.setId(id);

}

public void setStudentGrade(String grade) {

student.setGrade(grade);

}

public void updateView() {

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

}

}

public class Main {

public static void main(String[] args) {

Student student = new Student("Ayesha", 123, "A");

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

controller.updateView();

controller.setStudentName("lasya");

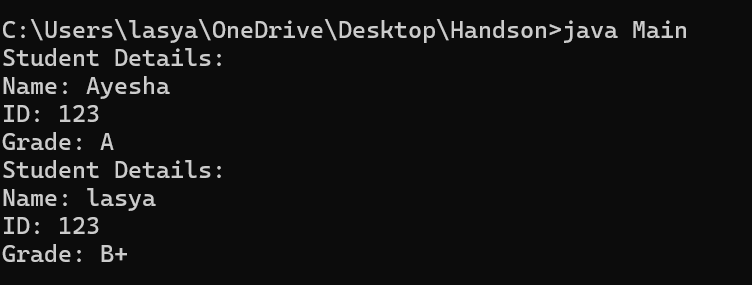
controller.setStudentGrade("B+");

controller.updateView();

}

}

Output:



**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.

Solution:

interface CustomerRepository {

String findCustomerById(int id);

}

class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(int id) {

return "Customer with ID: " + id;

}

}

class CustomerService {

private CustomerRepository repository;

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void findCustomer(int id) {

String customer = repository.findCustomerById(id);

System.out.println("Found: " + customer);

}

}

public class Main {

public static void main(String[] args) {

CustomerRepository repository = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repository);

service.findCustomer(1);

}

}

