**Week – 1**

**DESIGN PRINCIPLES AND PATTERNS**

**EXERCISE– 1:** Implementing the Singleton Pattern.

**SINGLETON PATTERN:**

The Singleton is a *creational design pattern* that ensures only one instance of a class is created throughout an application, and it provides a global point of access to that instance.

**CHARACTERISTICS:**

**1. Private constructor** — prevents instantiation from other classes.

**2.Static field** — holds the sole instance of the class.

**3.Public static access method** — typically getInstance(), to retrieve the singleton instance.

**WHY USE IT**?

* **Coordinate shared resources** like loggers, config managers, or thread pools.
* **Lazy initialization** helps save resources by delaying object creation until it's needed .

**SOURCE CODE:**

enum SingletonEnum {

INSTANCE;

int value;

public int getValue() { return value; }

public void setValue(int value) { this.value = value; }

}

public class Main {

public static void main(String[] args) {

SingletonEnum singleton = SingletonEnum.INSTANCE;

System.out.println(singleton.getValue()); // default value

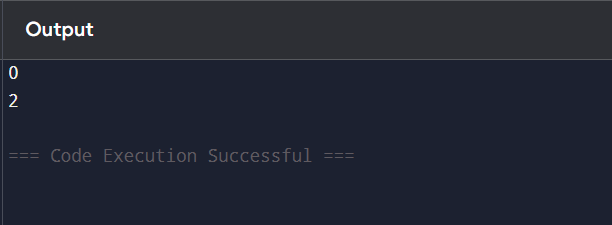
singleton.setValue(2);

System.out.println(singleton.getValue()); // after setting to 2

}

}

OUTPUT:



**EXERCISE-2:** Implementing the Factory Method Pattern

**PROBLEM STATEMENT:**

Build a logistics system with two types of transports: Truck (for road) and Ship (for sea). The system should use the Factory Method Pattern to instantiate these objects, not direct new calls in the client code.

**SOURCE CODE:**

public class Main {

// Product Interface

interface Transport {

String deliver();

}

// Concrete Products

static class Truck implements Transport {

public String deliver() {

return "Delivering by land in a truck.";

}

}

static class Ship implements Transport {

public String deliver() {

return "Delivering by sea in a ship.";

}

}

// Creator

static abstract class Logistics {

public abstract Transport createTransport();

public String planDelivery() {

Transport transport = createTransport();

return transport.deliver();

}

}

// Concrete Creators

static class RoadLogistics extends Logistics {

public Transport createTransport() {

return new Truck();

}

}

static class SeaLogistics extends Logistics {

public Transport createTransport() {

return new Ship();

}

}

// Main method (Client)

public static void main(String[] args) {

Logistics road = new RoadLogistics();

System.out.println("Road Logistics:");

System.out.println(road.planDelivery());

Logistics sea = new SeaLogistics();

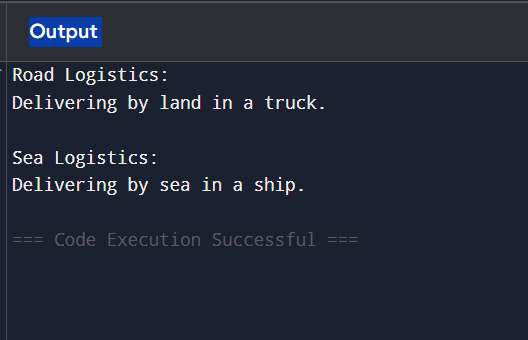
System.out.println("\nSea Logistics:");

System.out.println(sea.planDelivery());

}

}

OUTPUT:



**EXERCISE- 3:** Implementing the Builder Pattern.

**PROBLEM STATEMENT:**

Design and implement a Java program that uses the **Builder Design Pattern** to construct complex objects — specifically a Computer — with multiple optional parts (like RAM, CPU, storage, GPU, and operating system).

**REQUIREMENTS:**

**1. Create a Computer Class (Product):**

* Attributes:
  1. CPU (String)
  2. RAM (String)
  3. Storage (String)
  4. GPU (String)
  5. OS (String)
* The Computer class must:
  1. Have a **private constructor** that accepts a Builder object.
  2. Override the toString() method to display configuration details.

**2. Implement the Builder Class (Static Nested Class):**

* Create a static nested class named Builder inside Computer.
* Include setter-like methods for each attribute (e.g., setCPU(String cpu)).
* Each method should return the builder instance to allow **method chaining**.
* Provide a build() method that returns a fully constructed Computer object.

**3. Test the Implementation:**

* Write a test class (e.g., Main) with a main method.
* Use the Builder to create:
  1. A basic computer (with only CPU and RAM).
  2. A high-end gaming computer (all parts set).
* Print both computer configurations to verify output.

**SOURCE CODE:**

public class Main {

public static void main(String[] args) {

// Basic computer with CPU and RAM

Computer basicComputer = new Computer.Builder()

.setCPU("Intel i3")

.setRAM("4GB")

.build();

// Gaming computer with all parts

Computer gamingComputer = new Computer.Builder()

.setCPU("Intel i9")

.setRAM("32GB")

.setStorage("1TB SSD")

.setGPU("NVIDIA RTX 4080")

.setOS("Windows 11 Pro")

.build();

// Print computer configurations

System.out.println("Basic Computer: " + basicComputer);

System.out.println("Gaming Computer: " + gamingComputer);

}

}

// Product class with Builder pattern

class Computer {

private String CPU;

private String RAM;

private String storage;

private String GPU;

private String OS;

private Computer(Builder builder) {

this.CPU = builder.CPU;

this.RAM = builder.RAM;

this.storage = builder.storage;

this.GPU = builder.GPU;

this.OS = builder.OS;

}

public static class Builder {

private String CPU;

private String RAM;

private String storage;

private String GPU;

private String OS;

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 Builder setOS(String OS) {

this.OS = OS;

return this;

}

public Computer build() {

return new Computer(this);

}

}

@Override

public String toString() {

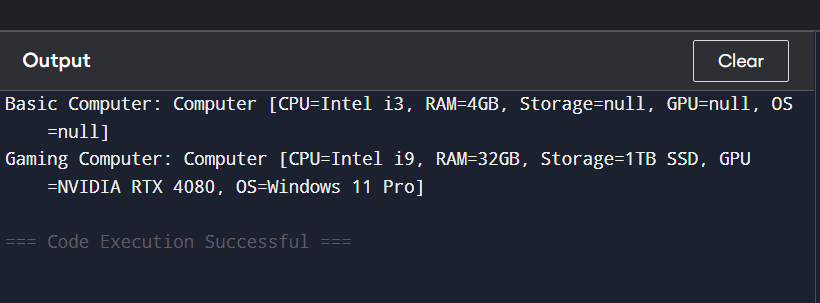
return "Computer [CPU=" + CPU + ", RAM=" + RAM + ", Storage=" + storage +

", GPU=" + GPU + ", OS=" + OS + "]";

}

}

**OUTPUT:**

****

**EXERCISE-4:** Implementing the Adapter Pattern.

**PROBLEM STATEMENT:**

You are building a payment processing system that integrates with multiple third-party payment gateways, each with a different API. Use the Adapter Pattern to create a common interface so your system can use all payment gateways interchangeably**.**

**REQUIREMENTS:**

1. **Define a Target Interface PaymentProcessor with method:**

void processPayment(double amount);

1. **Create Third-Party Payment Classes:**
   1. StripeGateway with makeStripePayment(double amount)
   2. PayPalGateway with sendPayPalPayment(double amount)
2. **Create Adapter Classes:**
   1. StripeAdapter and PayPalAdapter should implement PaymentProcessor and internally call the respective third-party methods.
3. **Test:**

In your main method, demonstrate payment through both gateways using the unified interface.

**SOURCE CODE:**

public class Main {

public static void main(String[] args) {

// Using Stripe via adapter

StripeGateway stripe = new StripeGateway();

PaymentProcessor stripeProcessor = new StripeAdapter(stripe);

stripeProcessor.processPayment(100.0);

// Using PayPal via adapter

PayPalGateway paypal = new PayPalGateway();

PaymentProcessor paypalProcessor = new PayPalAdapter(paypal);

paypalProcessor.processPayment(250.0);

}

}

interface PaymentProcessor {

void processPayment(double amount);

}

class StripeGateway {

public void makeStripePayment(double amount) {

System.out.println("Paid $" + amount + " using Stripe.");

}

}

class PayPalGateway {

public void sendPayPalPayment(double amount) {

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

}

}

class StripeAdapter implements PaymentProcessor {

private StripeGateway stripeGateway;

public StripeAdapter(StripeGateway stripeGateway) {

this.stripeGateway = stripeGateway;

}

@Override

public void processPayment(double amount) {

stripeGateway.makeStripePayment(amount);

}

}

class PayPalAdapter implements PaymentProcessor {

private PayPalGateway payPalGateway;

public PayPalAdapter(PayPalGateway payPalGateway) {

this.payPalGateway = payPalGateway;

}

@Override

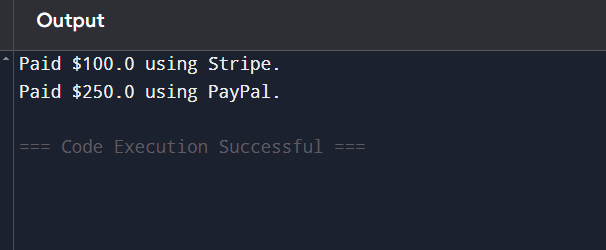
public void processPayment(double amount) {

payPalGateway.sendPayPalPayment(amount);

}

}

**OUTPUT:**



**EXERCISE-5:** Implementing the Decorator Pattern.

**PROBLEM STATEMENT:**

Develop a notification system where notifications can be sent via multiple channels (e.g., Email, SMS, Slack). Use the Decorator Pattern to dynamically add notification methods without altering the base Notifier implementation.

**REQUIREMENTS:**

1. Define a Notifier interface.
2. Create a base EmailNotifier class.
3. Create decorator classes:
   * SMSNotifierDecorator
   * SlackNotifierDecorator
4. Demonstrate sending a single message via all channels using decorators.

**SOURCE CODE:**

public class Main {

public static void main(String[] args) {

// Base notifier: Email only

Notifier emailNotifier = new EmailNotifier();

// Add SMS notification on top of Email

Notifier smsNotifier = new SMSNotifierDecorator(emailNotifier);

// Add Slack notification on top of Email + SMS

Notifier fullNotifier = new SlackNotifierDecorator(smsNotifier);

// Send the message via all channels

fullNotifier.send("Server is down! Please check immediately.");

}

}

// Step 1: Component Interface

interface Notifier {

void send(String message);

}

// Step 2: Concrete Component

class EmailNotifier implements Notifier {

public void send(String message) {

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

}

}

// Step 3: Abstract Decorator

abstract class NotifierDecorator implements Notifier {

protected Notifier wrappedNotifier;

public NotifierDecorator(Notifier notifier) {

this.wrappedNotifier = notifier;

}

public void send(String message) {

wrappedNotifier.send(message);

}

}

// Step 4: SMS Decorator

class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

sendSMS(message);

}

private void sendSMS(String message) {

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

}

}

// Step 5: Slack Decorator

class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

sendSlack(message);

}

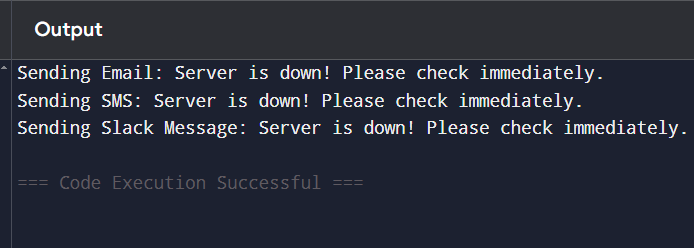
private void sendSlack(String message) {

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

}

}

**OUTPUT:**

****

**EXERCISE-6:** Implementing the Proxy Pattern.

**PROBLEM STATEMENT:**

Build an image viewer that simulates loading images from a remote server. Use the Proxy Pattern to:

* Delay image loading until it's actually needed (lazy initialization)
* Avoid reloading the same image more than once (caching)

**SOURCE CODE:**

public class Main {

public static void main(String[] args) {

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

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

// Image is loaded only when display() is called

image1.display(); // Loads and displays

image1.display(); // Just displays (no reloading)

image2.display(); // Loads and displays

image2.display(); // Just displays

}

}

// Step 1: Subject Interface

interface Image {

void display();

}

// Step 2: Real Subject

class RealImage implements Image {

private String filename;

public RealImage(String filename) {

this.filename = filename;

loadFromRemoteServer(); // Simulate loading from server

}

private void loadFromRemoteServer() {

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

}

public void display() {

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

}

}

// Step 3: Proxy Class

class ProxyImage implements Image {

private String filename;

private RealImage realImage;

public ProxyImage(String filename) {

this.filename = filename;

}

public void display() {

if (realImage == null) {

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

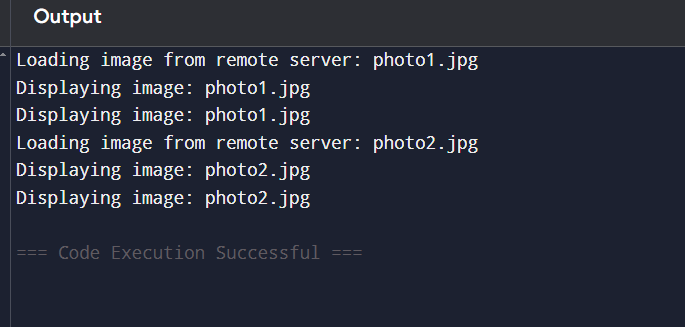
}

realImage.display(); // Always display

}

}

**OUTPUT:**

****

**EXERCISE- 7:** Implementing the Observer Pattern.

**PROBLEM STATEMENT:**

You are building a stock market monitoring app. When stock prices change, all registered clients (like mobile and web apps) must be automatically notified. Use the Observer Pattern to manage subscriptions and notifications.

**SOURCE CODE:**

import java.util.\*;

// Step 1: Observer Interface

interface Observer {

void update(String stockName, double newPrice);

}

// Step 2: Subject Interface

interface Stock {

void registerObserver(Observer o);

void removeObserver(Observer o);

void notifyObservers();

}

// Step 3: Concrete Subject

class StockMarket implements Stock {

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

private String stockName;

private double stockPrice;

public void setStock(String name, double price) {

this.stockName = name;

this.stockPrice = price;

notifyObservers();

}

@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(stockName, stockPrice);

}

}

}

// Step 4: Concrete Observer – Mobile App

class MobileApp implements Observer {

private String name;

public MobileApp(String name) {

this.name = name;

}

public void update(String stockName, double newPrice) {

System.out.println("[" + name + " Mobile] Stock " + stockName + " is now $" + newPrice);

}

}

// Step 5: Concrete Observer – Web App

class WebApp implements Observer {

private String name;

public WebApp(String name) {

this.name = name;

}

public void update(String stockName, double newPrice) {

System.out.println("[" + name + " Web] Stock " + stockName + " is now $" + newPrice);

}

}

// Step 6: Test Class

public class Main {

public static void main(String[] args) {

StockMarket stockMarket = new StockMarket();

Observer mobileApp = new MobileApp("InvestorPro");

Observer webApp = new WebApp("TradePortal");

stockMarket.registerObserver(mobileApp);

stockMarket.registerObserver(webApp);

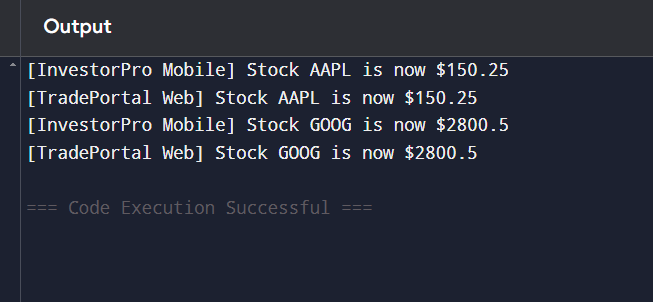
stockMarket.setStock("AAPL", 150.25);

stockMarket.setStock("GOOG", 2800.50);

}

}

**OUTPUT:**

****

**EXERCISE-8:** Implementing the Strategy Pattern.

**PROBLEM STATEMENT:**

Design a payment system that supports multiple payment methods (like Credit Card and PayPal) that can be selected at runtime. Use the Strategy Pattern to make payment algorithms interchangeable without modifying the context class.

**SOURCE CODE:**

// Step 1: Strategy Interface

interface PaymentStrategy {

void pay(double amount);

}

// Step 2: Concrete Strategy - Credit Card

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);

}

}

// Step 2: Concrete Strategy - PayPal

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);

}

}

// Step 3: Context Class

class PaymentContext {

private PaymentStrategy strategy;

public void setPaymentStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void processPayment(double amount) {

if (strategy == null) {

System.out.println("Payment method not selected!");

} else {

strategy.pay(amount);

}

}

}

// Step 4: Test Class

public class Main {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

// Pay using Credit Card

context.setPaymentStrategy(new CreditCardPayment("1234-5678-9012-3456"));

context.processPayment(100.00);

// Pay using PayPal

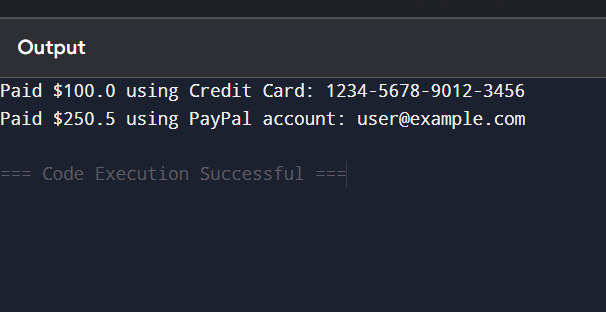
context.setPaymentStrategy(new PayPalPayment("user@example.com"));

context.processPayment(250.50);

}

}

**OUTPUT:**

****

**EXERCISE-9:** Implementing the Command Pattern.

**PROBLEM STATEMENT:**

Design a system where you can issue commands (like turning a light on or off) using a remote control. Use the Command Pattern to encapsulate each action as an object.

**SOURCE CODE:**

// Step 1: Command Interface

interface Command {

void execute();

}

// Step 2: Receiver Class

class Light {

public void turnOn() {

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

}

public void turnOff() {

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

}

}

// Step 3: Concrete Command - Turn Light ON

class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOn();

}

}

// Step 3: Concrete Command - Turn Light OFF

class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOff();

}

}

// Step 4: Invoker

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 assigned.");

}

}

}

// Step 5: Test Class

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(); // Output: The light is ON

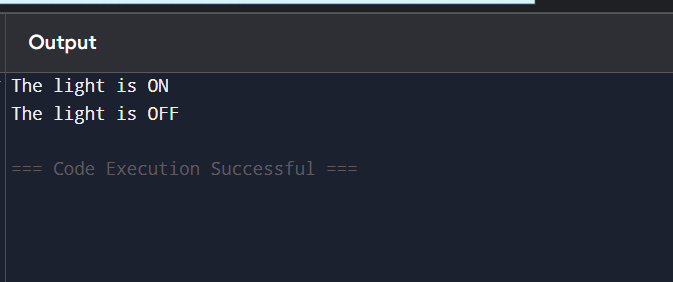
remote.setCommand(lightOff);

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

}

}

**OUTPUT:**

****

**EXERCISE-10:** Implementing the MVC Pattern.

**PROBLEM STATEMENT:**

Design a simple student record management system using the Model-View-Controller (MVC) design pattern. The system should allow viewing and updating student details through a controller.

**SOURCE CODE:**

// Model

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 setGrade(String grade) {

this.grade = grade;

}

}

// View

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);

}

}

// Controller

class StudentController {

private Student model;

private StudentView view;

public StudentController(Student model, StudentView view) {

this.model = model;

this.view = view;

}

// Controller Methods to update and retrieve data

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());

}

}

// Test class (Main)

public class Main {

public static void main(String[] args) {

// Create model

Student student = new Student("Alice", "S123", "A");

// Create view

StudentView view = new StudentView();

// Create controller

StudentController controller = new StudentController(student, view);

// Initial display

controller.updateView();

// Update student info

controller.setStudentName("Alice Johnson");

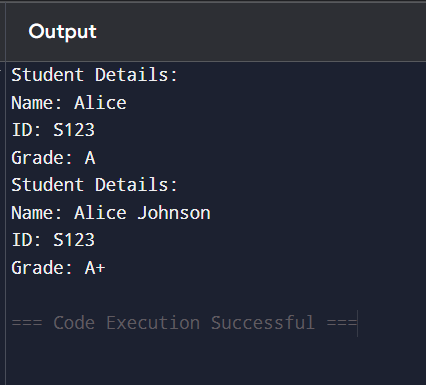
controller.setStudentGrade("A+");

controller.updateView();

}

}

**OUTPUT:**

****

**EXERCISE-11:** Implementing Dependency Injection.

**PROBLEM STATEMENT:**

Create a customer management system where the CustomerService depends on CustomerRepository. Use **constructor-based dependency injection** to provide CustomerRepositoryImpl to the service.

**SOURCE CODE:**

// Step 1: Repository Interface

interface CustomerRepository {

String findCustomerById(String id);

}

// Step 2: Concrete Repository

class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(String id) {

// Simulate database lookup

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

}

}

// Step 3: Service Class

class CustomerService {

private CustomerRepository customerRepository;

// Constructor-based Dependency Injection

public CustomerService(CustomerRepository customerRepository) {

this.customerRepository = customerRepository;

}

public void getCustomerDetails(String id) {

String customer = customerRepository.findCustomerById(id);

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

}

}

// Step 4: Test Class (Main)

public class Main {

public static void main(String[] args) {

// Injecting dependency manually

CustomerRepository repo = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repo);

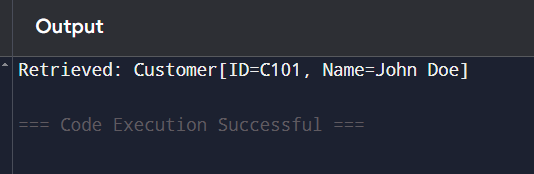
// Using the service

service.getCustomerDetails("C101");

}

}

**OUTPUT:**

****