**WEEK 1 : DESIGN PATTERNS AND PRINCIPLES**

**Exercise 1: Implementing the Singleton Pattern**

**Logger.java**

public class Logger {

private static Logger logger = null;

private Logger() {

System.out.println("Logger Class Initialized");

}

public static Logger getInstance() {

if (logger == null) {

logger = new Logger();

}

return logger;

}

}

**Main.java**

public class Main{

public static void main(String[] args) {

Logger obj1 = Logger.getInstance();

Logger obj2 = Logger.getInstance();

if (obj1 == obj2) {

System.out.println("Single Instance");

} else {

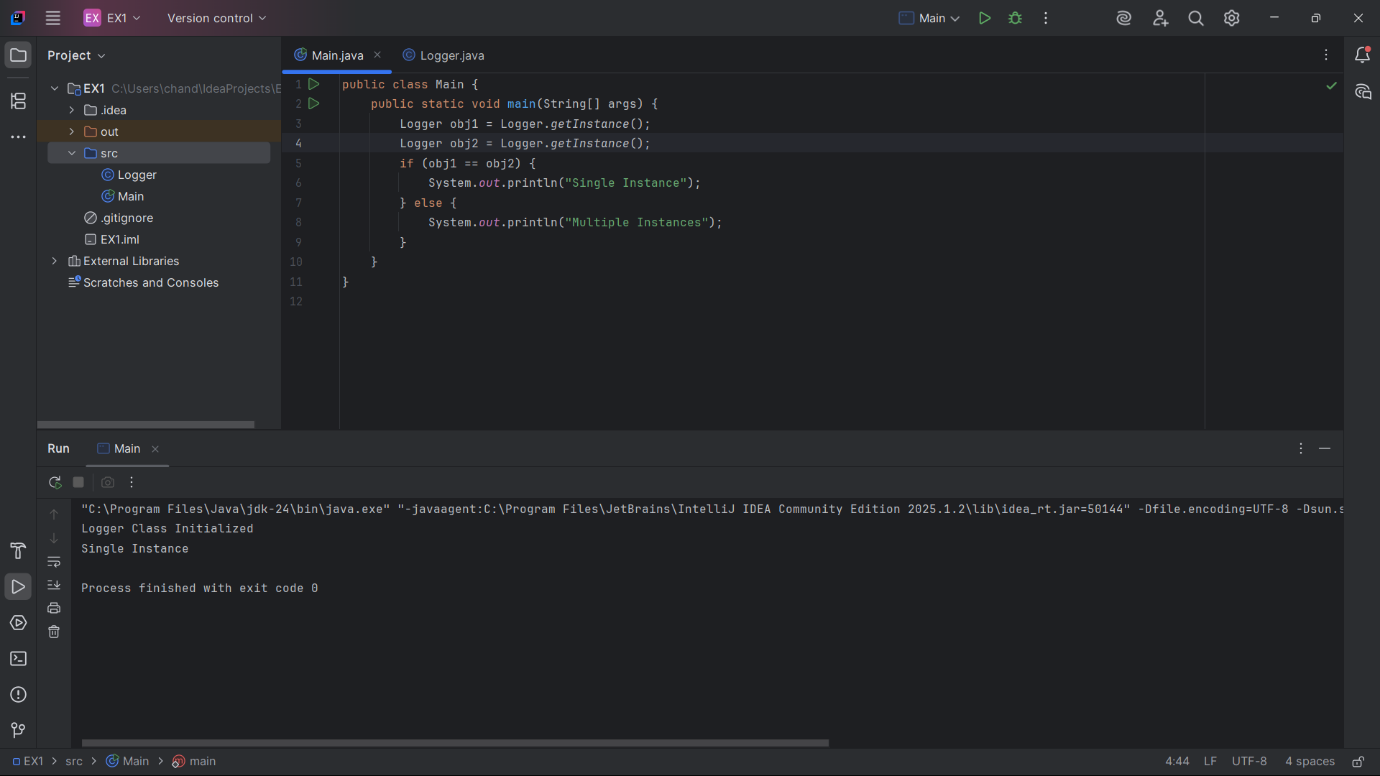
System.out.println("Multiple Instances");

}

}

}

**OUTPUT:**

****

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

**Document.java**

public interface Document {

void open();

}

**WordDocument.java**

public class WordDocument implements Document {

@Override

public void open() {

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

}

}

**PdfDocument.java**

public class PdfDocument implements Document {

@Override

public void open() {

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

}

}

**ExcelDocument.java**

public class ExcelDocument implements Document {

@Override

public void open() {

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

}

}

**DocumentFactory.java**

public abstract class DocumentFactory {

public abstract Document createDocument();

}

**WordDocumentFactory.java**

public class WordDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new WordDocument();

}

}

**PdfDocumentFactory.java**

public class PdfDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new PdfDocument();

}

}

**ExcelDocumentFactory.java**

public class ExcelDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new ExcelDocument();

}

}

**Main.java**

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();

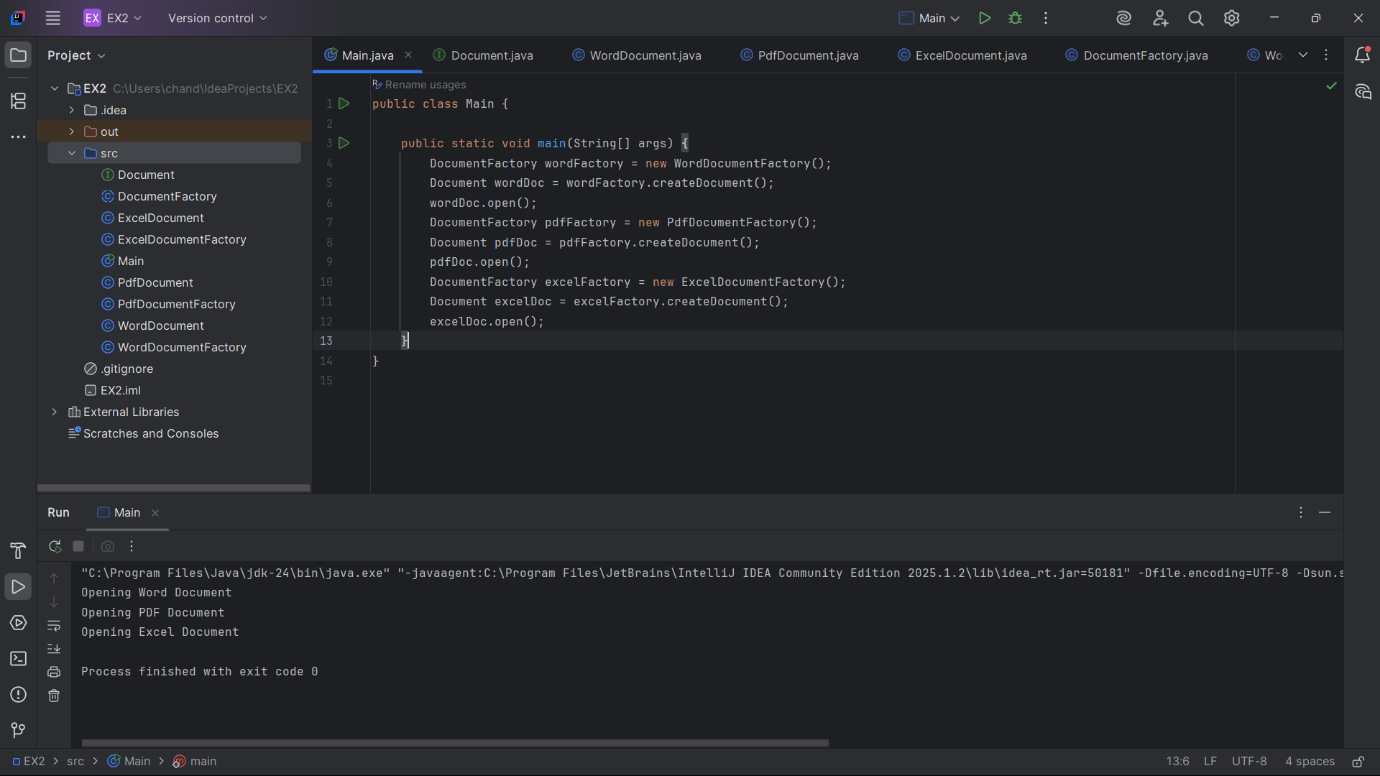
Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

**OUTPUT:**



**Exercise 3: Implementing the Builder Pattern**

**Computer.java**

public class Computer {

private String cpu;

private String ram;

private String storage;

private String graphicsCard;

private String operatingSystem;

private Computer(Builder builder) {

this.cpu = builder.cpu;

this.ram = builder.ram;

this.storage = builder.storage;

this.graphicsCard = builder.graphicsCard;

this.operatingSystem = builder.operatingSystem;

}

public void displayConfig() {

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

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

System.out.println("Storage: " + (storage != null ? storage : "Not Specified"));

System.out.println("Graphics Card: " + (graphicsCard != null ? graphicsCard : "Not Specified"));

System.out.println("Operating System: " + (operatingSystem != null ? operatingSystem : "Not Specified"));

System.out.println("--------------------------------------------------");

}

public static class Builder {

private String cpu;

private String ram;

private String storage;

private String graphicsCard;

private String operatingSystem;

public Builder(String cpu, String ram) {

this.cpu = cpu;

this.ram = ram;

}

public Builder setStorage(String storage) {

this.storage = storage;

return this;

}

public Builder setGraphicsCard(String graphicsCard) {

this.graphicsCard = graphicsCard;

return this;

}

public Builder setOperatingSystem(String operatingSystem) {

this.operatingSystem = operatingSystem;

return this;

}

public Computer build() {

return new Computer(this);

}

}

}

**Main.java**

public class Main{

public static void main(String[] args) {

Computer basicPC = new Computer.Builder("Intel i3", "4GB")

.build();

Computer gamingPC = new Computer.Builder("Intel i9", "32GB")

.setGraphicsCard("NVIDIA RTX 4090")

.setStorage("2TB SSD")

.setOperatingSystem("Windows 11 Pro")

.build();

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

.setStorage("512GB SSD")

.setOperatingSystem("Ubuntu Linux")

.build();

System.out.println("Basic PC Configuration:");

basicPC.displayConfig();

System.out.println("Gaming PC Configuration:");

gamingPC.displayConfig();

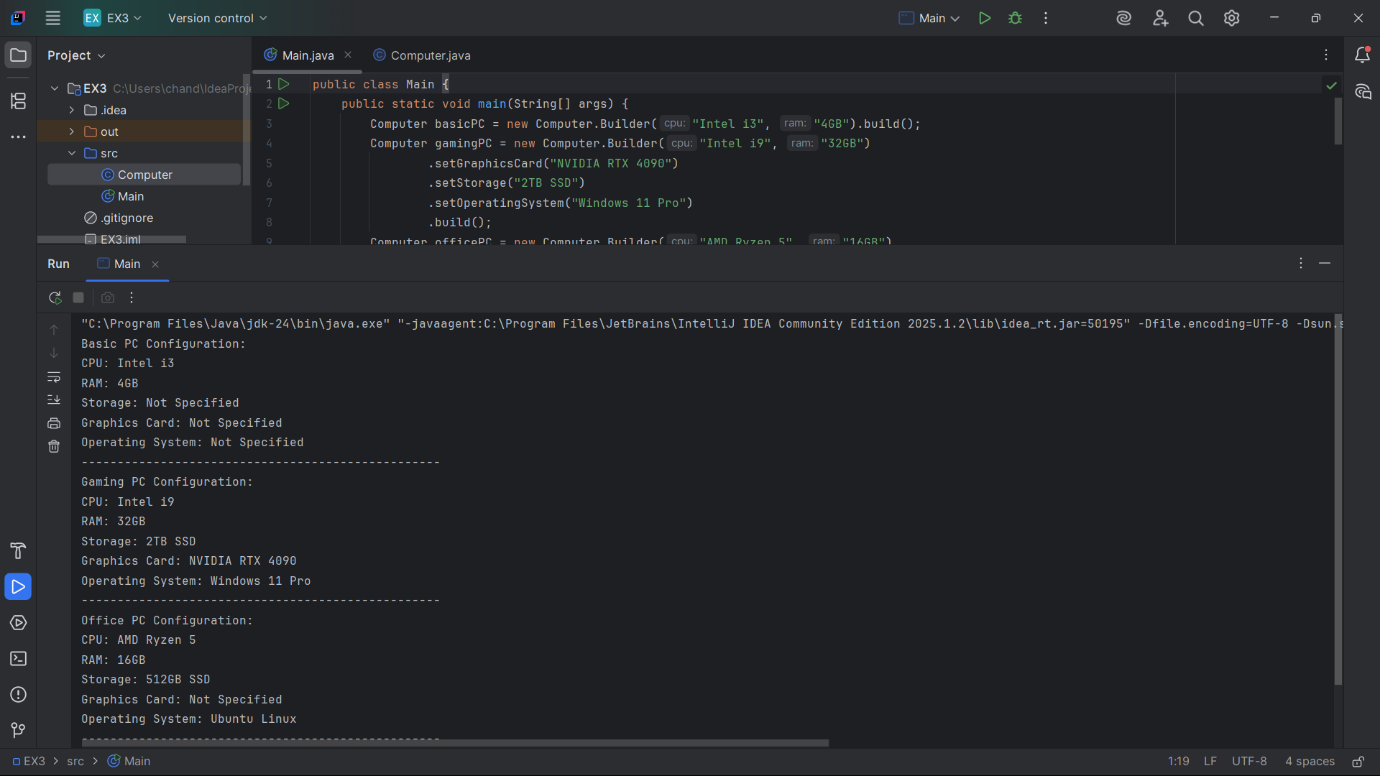
System.out.println("Office PC Configuration:");

officePC.displayConfig();

}

}

**OUTPUT:**

****

**Exercise 4: Implementing the Adapter Pattern**

**PaymentProcessor.java**

public interface PaymentProcessor {

void processPayment(double amount);

}

**PayPalGateway.java**

public class PayPalGateway {

public void sendPayment(double amount) {

System.out.println("Processing payment via PayPal: $" + amount);

}

}

**StripeGateway.java**

public class StripeGateway {

public void makePayment(double amount) {

System.out.println("Processing payment via Stripe: $" + amount);

}

}

**PayPalAdapter.java**

public class PayPalAdapter implements PaymentProcessor {

private PayPalGateway payPal;

public PayPalAdapter(PayPalGateway payPal) {

this.payPal = payPal;

}

@Override

public void processPayment(double amount) {

payPal.sendPayment(amount);

}

}

**StripeAdapter.java**

public class StripeAdapter implements PaymentProcessor {

private StripeGateway stripe;

public StripeAdapter(StripeGateway stripe) {

this.stripe = stripe;

}

@Override

public void processPayment(double amount) {

stripe.makePayment(amount);

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

PayPalGateway paypal = new PayPalGateway();

StripeGateway stripe = new StripeGateway();

PaymentProcessor paypalProcessor = new PayPalAdapter(paypal);

PaymentProcessor stripeProcessor = new StripeAdapter(stripe);

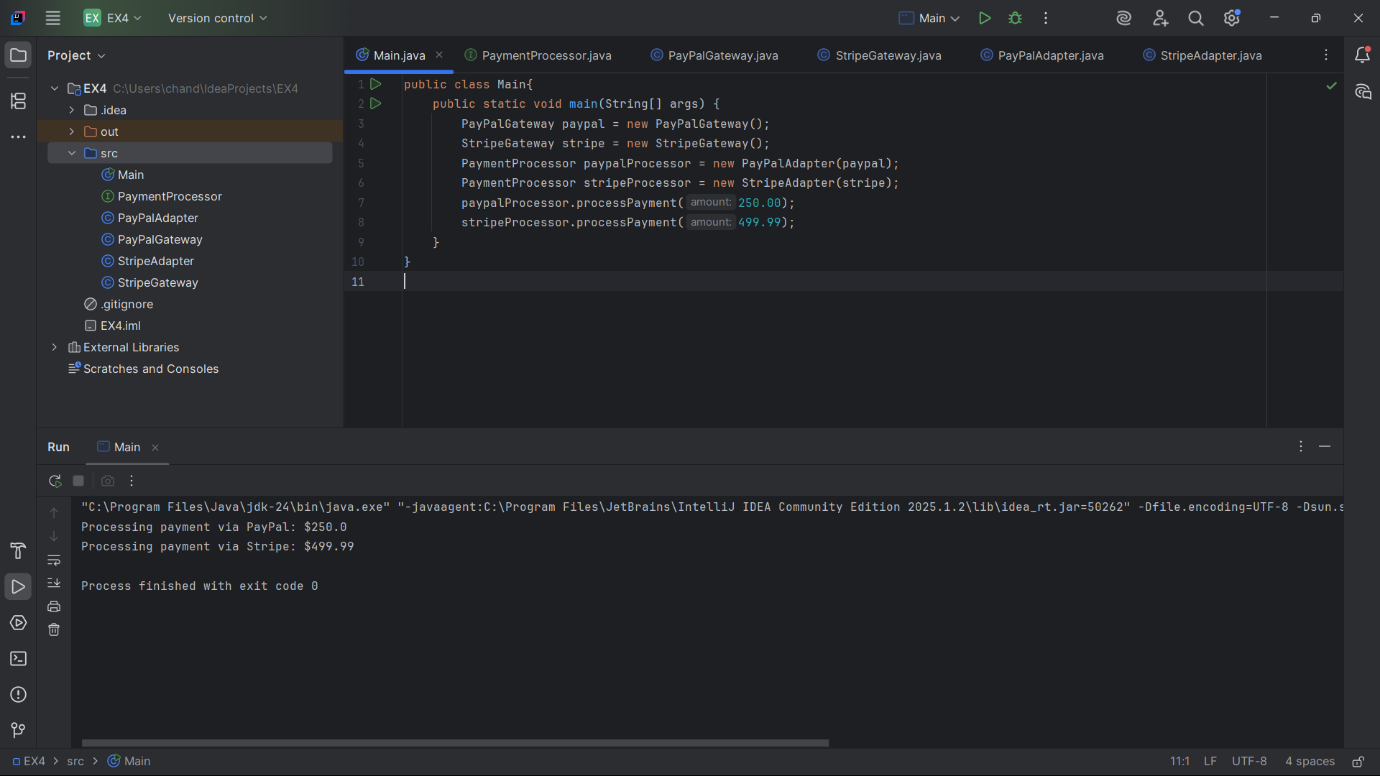
paypalProcessor.processPayment(250.00);

stripeProcessor.processPayment(499.99);

}

}

**OUTPUT:**

****

**Exercise 5: Implementing the Decorator Pattern**

**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 wrappedNotifier;

public NotifierDecorator(Notifier notifier) {

this.wrappedNotifier = notifier;

}

@Override

public void send(String message) {

wrappedNotifier.send(message);

}

}

**SMSNotifierDecorator.java**

public class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

@Override

public void send(String message) {

super.send(message);

sendSMS(message);

}

private void sendSMS(String message) {

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

}

}

**SlackNotifierDecorator.java**

public class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

@Override

public void send(String message) {

super.send(message);

sendSlack(message);

}

private void sendSlack(String message) {

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

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

Notifier email = new EmailNotifier();

Notifier emailAndSMS = new SMSNotifierDecorator(email);

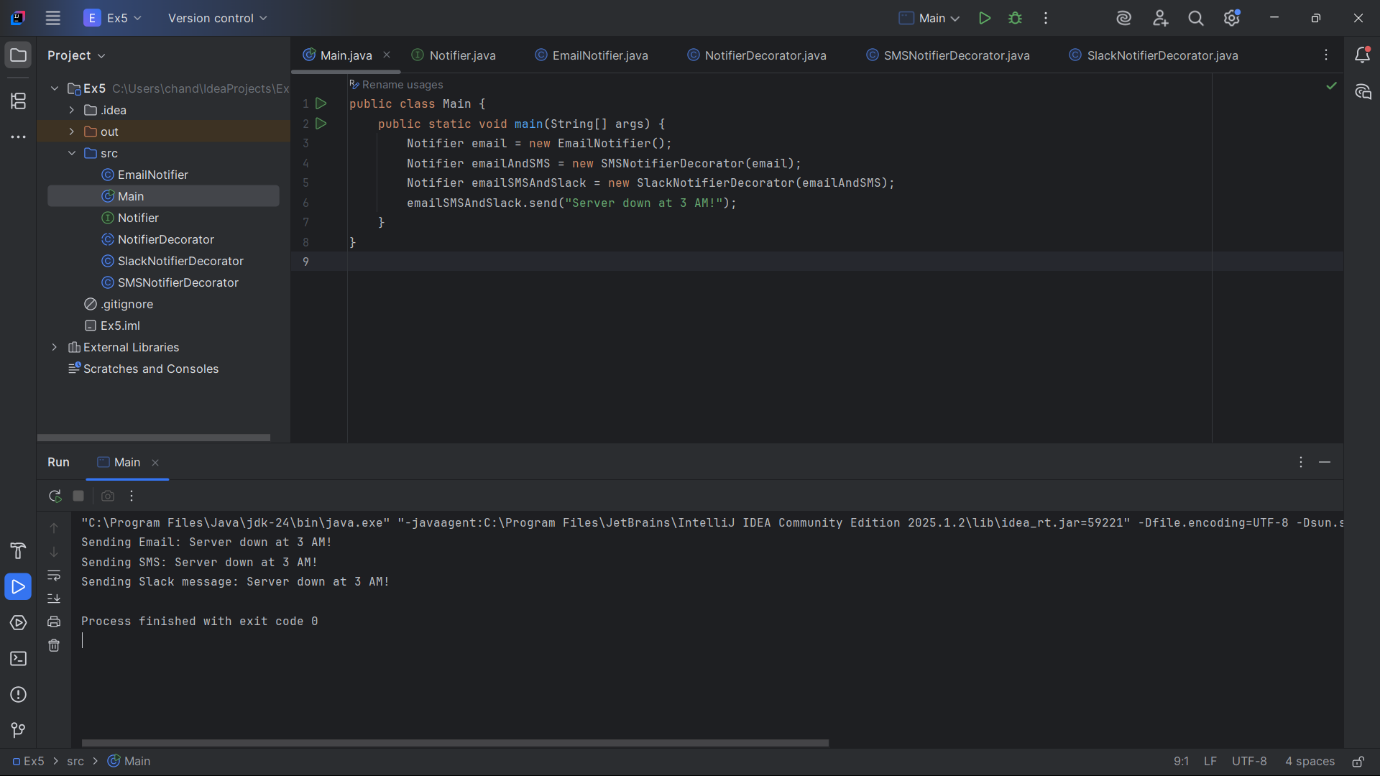
Notifier emailSMSAndSlack = new SlackNotifierDecorator(emailAndSMS);

emailSMSAndSlack.send("Server down at 3 AM!");

}

}

**OUTPUT:**

****

**Exercise 6: Implementing the Proxy Pattern**

**Image.java**

public interface Image {

void 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);

}

}

**ProxyImage.java**

public class ProxyImage implements Image {

private String filename;

private RealImage realImage;

public ProxyImage(String filename) {

this.filename = filename;

}

@Override

public void display() {

if (realImage == null) {

realImage = new RealImage(filename);

}

realImage.display();

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

Image img1 = new ProxyImage("nature.jpg");

Image img2 = new ProxyImage("sunset.png");

img1.display();

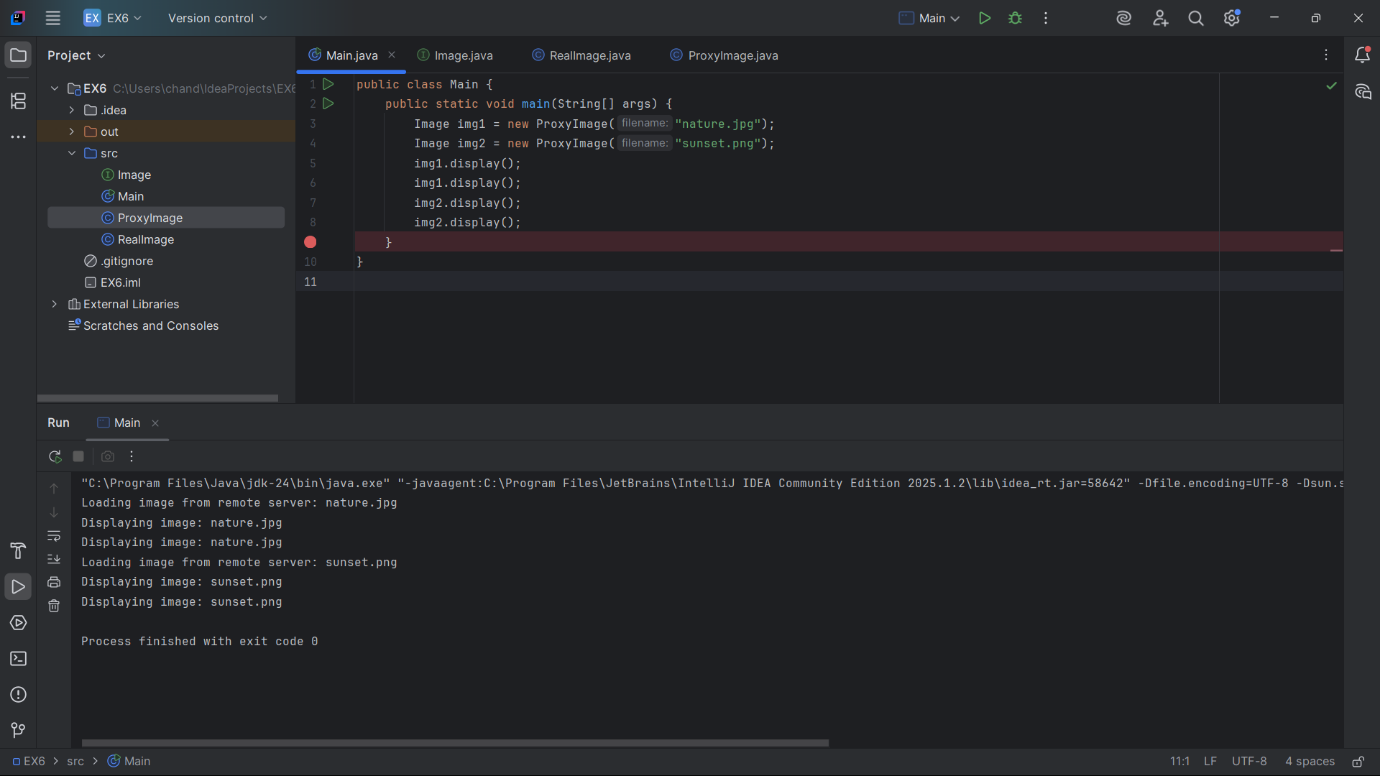
img1.display();

img2.display();

img2.display();

}

}

**OUTPUT:  
**

**Exercise 7: Implementing the Observer Pattern**

**Stock.java**

public interface Stock {

void registerObserver(Observer observer);

void removeObserver(Observer observer);

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;

@Override

public void registerObserver(Observer observer) {

observers.add(observer);

}

@Override

public void removeObserver(Observer observer) {

observers.remove(observer);

}

@Override

public void notifyObservers() {

for (Observer obs : observers) {

obs.update(stockPrice);

}

}

public void setStockPrice(double newPrice) {

this.stockPrice = newPrice;

System.out.println("\nStock Price Updated: $" + stockPrice);

notifyObservers();

}

}

**Observer.java**

public interface Observer {

void update(double price);

}

**MobileApp.java**

public class MobileApp implements Observer {

String name;

public MobileApp(String name) {

this.name = name;

}

@Override

public void update(double price) {

System.out.println("MobileApp [" + name + "] received update: $" + price);

}

}

**WebApp.java**

public class WebApp implements Observer {

String name;

public WebApp(String name) {

this.name = name;

}

@Override

public void update(double price) {

System.out.println("WebApp [" + name + "] received update: $" + price);

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

StockMarket market = new StockMarket();

Observer mobile1 = new MobileApp("MobileTrader1");

Observer mobile2 = new MobileApp("MobileTrader2");

Observer web1 = new WebApp("WebTerminal");

market.registerObserver(mobile1);

market.registerObserver(mobile2);

market.registerObserver(web1);

market.setStockPrice(150.50);

market.setStockPrice(155.75);

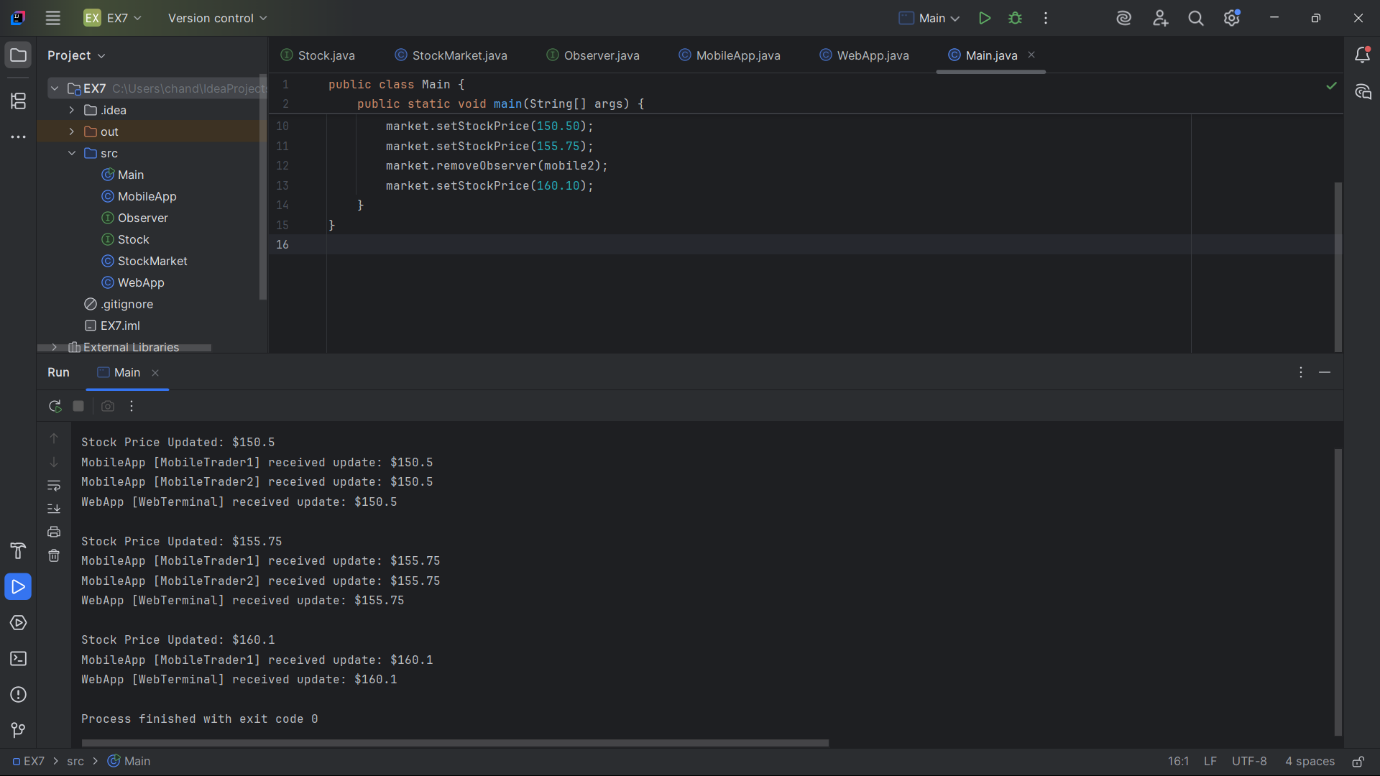
market.removeObserver(mobile2);

market.setStockPrice(160.10);

}

}

**OUTPUT:**

****

**Exercise 8: Implementing the Strategy Pattern**

**PaymentStrategy.java**

public interface PaymentStrategy {

void pay(double amount);

}

**CreditCardPayment.java**

public class CreditCardPayment implements PaymentStrategy {

String cardNumber;

String name;

public CreditCardPayment(String cardNumber, String name) {

this.cardNumber = cardNumber;

this.name = name;

}

@Override

public void pay(double amount) {

System.out.println("Paid ₹" + amount + " using Credit Card [" + cardNumber + "] - Holder: " + name);

}

}

**PayPalPayment.java**

public class PayPalPayment implements PaymentStrategy {

String email;

public PayPalPayment(String email) {

this.email = email;

}

@Override

public void pay(double amount) {

System.out.println("Paid ₹" + amount + " via PayPal - Account: " + email);

}

}

**PaymentContext.java**

public class PaymentContext {

PaymentStrategy strategy;

public void setPaymentStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void pay(double amount) {

if (strategy == null) {

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

} else {

strategy.pay(amount);

}

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

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

context.pay(1200.50);

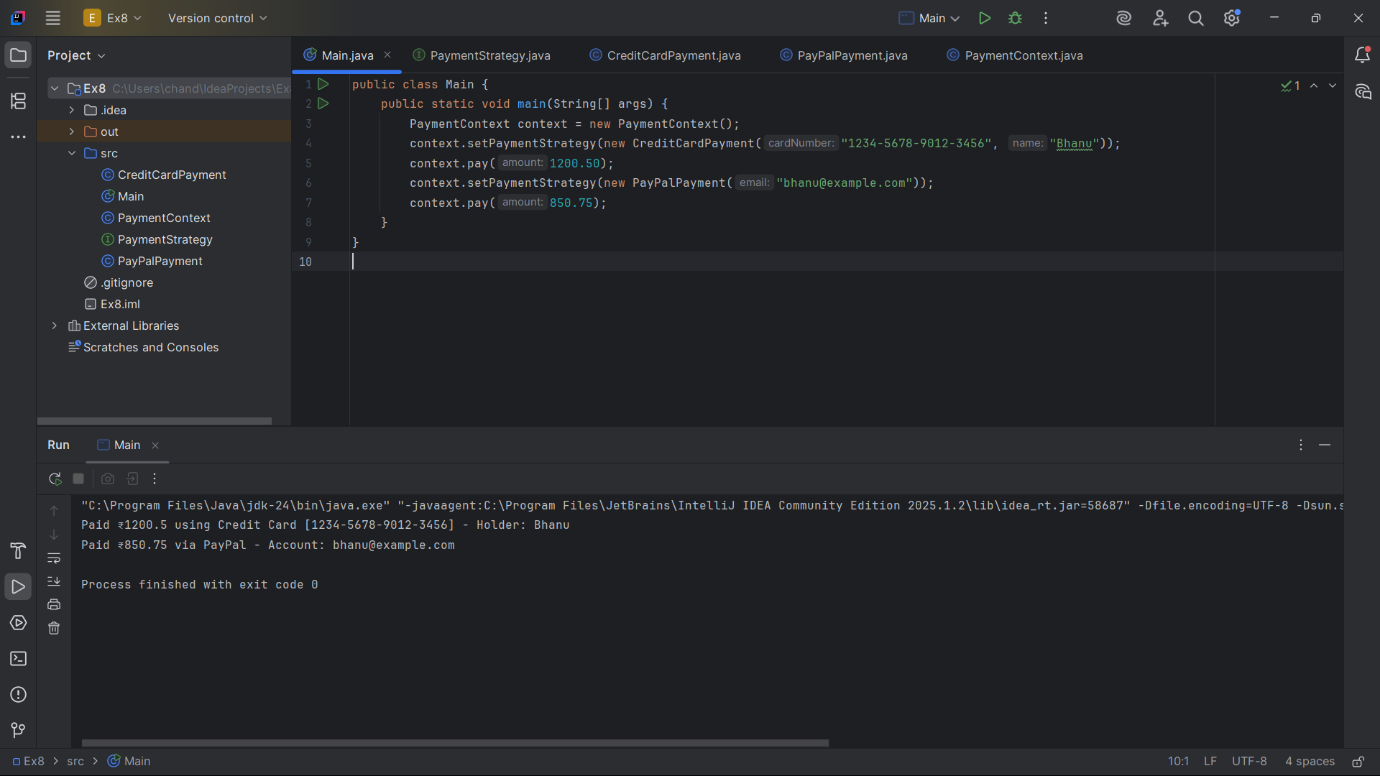
context.setPaymentStrategy(new PayPalPayment("bhanu@example.com"));

context.pay(850.75);

}

}

**OUTPUT:**

****

**Exercise 9: Implementing the Command Pattern**

**Command.java**

public interface Command {

void execute();

}

**Light.java**

public class Light {

void turnOn() {

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

}

void turnOff() {

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

}

}

**LightOnCommand.java**

public class LightOnCommand implements Command {

Light light;

LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOn();

}

}

**LightOffCommand.java**

public class LightOffCommand implements Command {

Light light;

LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOff();

}

}

**RemoteControl.java**

public class RemoteControl {

Command command;

void setCommand(Command command) {

this.command = command;

}

void pressButton() {

if (command != null) {

command.execute();

} else {

System.out.println("No command assigned.");

}

}

}

**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();

remote.setCommand(lightOn);

remote.pressButton();

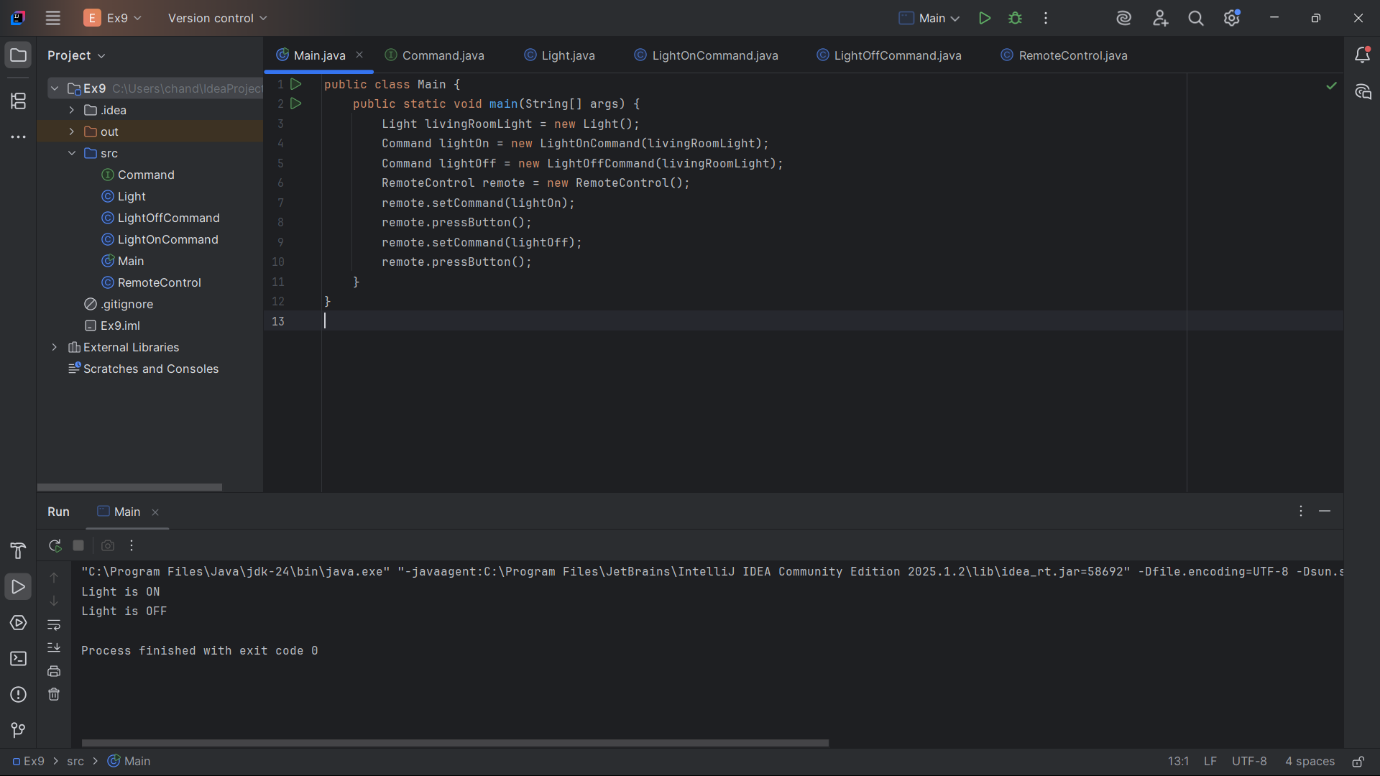
remote.setCommand(lightOff);

remote.pressButton();

}

}

**OUTPUT:**

****

**Exercise 10: Implementing the MVC Pattern**

**Student.java**

public class Student {

String name;

String id;

String grade;

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

this.name = name;

this.id = id;

this.grade = grade;

}

String getName() {

return name;

}

void setName(String name) {

this.name = name;

}

String getId() {

return id;

}

void setId(String id) {

this.id = id;

}

String getGrade() {

return grade;

}

void setGrade(String grade) {

this.grade = grade;

}

}

**StudentView.java**

public class StudentView {

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 {

Student student;

StudentView view;

StudentController(Student student, StudentView view) {

this.student = student;

this.view = view;

}

void setStudentName(String name) {

student.setName(name);

}

void setStudentId(String id) {

student.setId(id);

}

void setStudentGrade(String grade) {

student.setGrade(grade);

}

String getStudentName() {

return student.getName();

}

String getStudentId() {

return student.getId();

}

String getStudentGrade() {

return student.getGrade();

}

void updateView() {

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

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

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

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

controller.updateView();

controller.setStudentName("Bhanu Prakash");

controller.setStudentGrade("A+");

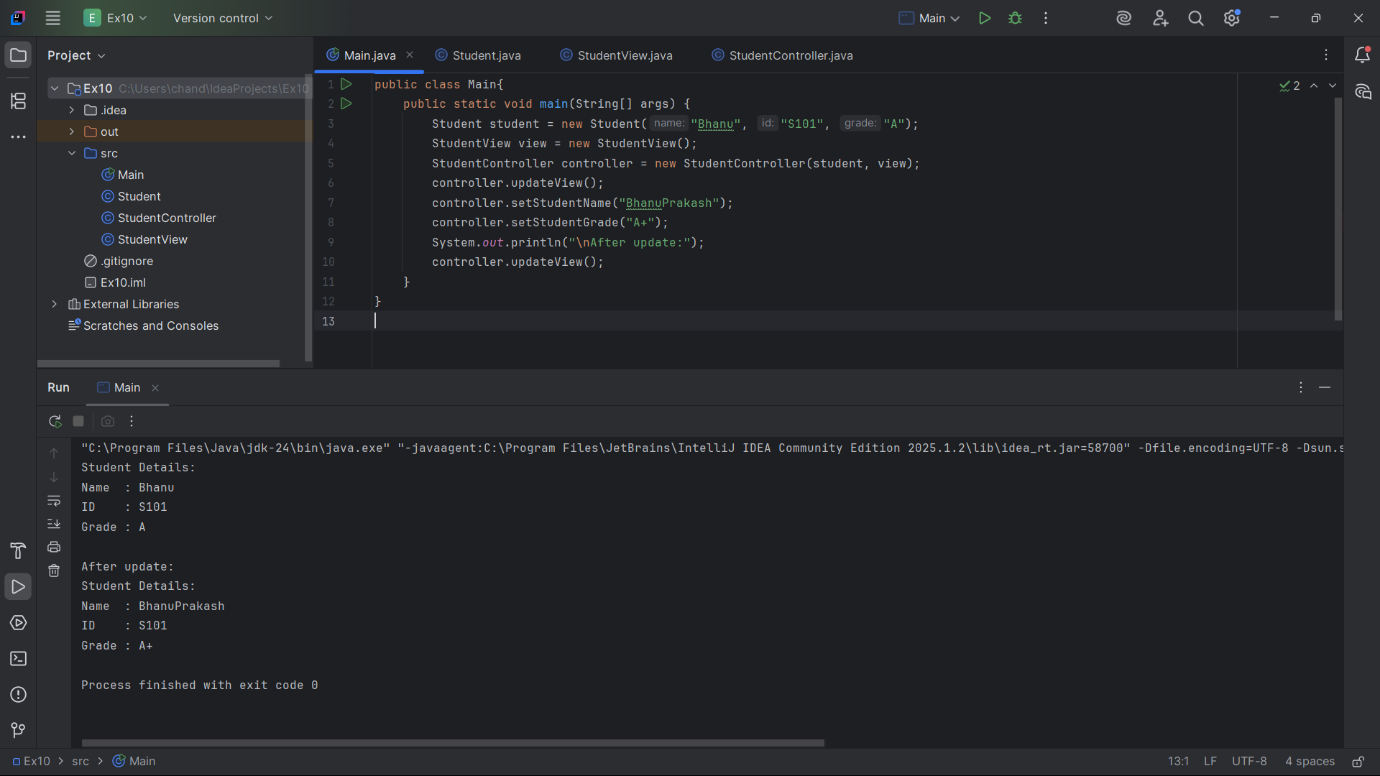
System.out.println("\nAfter update:");

controller.updateView();

}

}

**OUTPUT:**

****

**Exercise 11: Dependency Injection**

**CustomerRepository.java**

public interface CustomerRepository {

String findCustomerById(String id);

}

**CustomerRepositoryImpl.java**

public class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(String id) {

return "Customer[ID: " + id + ", Name: Aditi Kumar]";

}

}

**CustomerService.java**

public class CustomerService {

CustomerRepository repository;

CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void displayCustomer(String id) {

String customer = repository.findCustomerById(id);

System.out.println(customer);

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

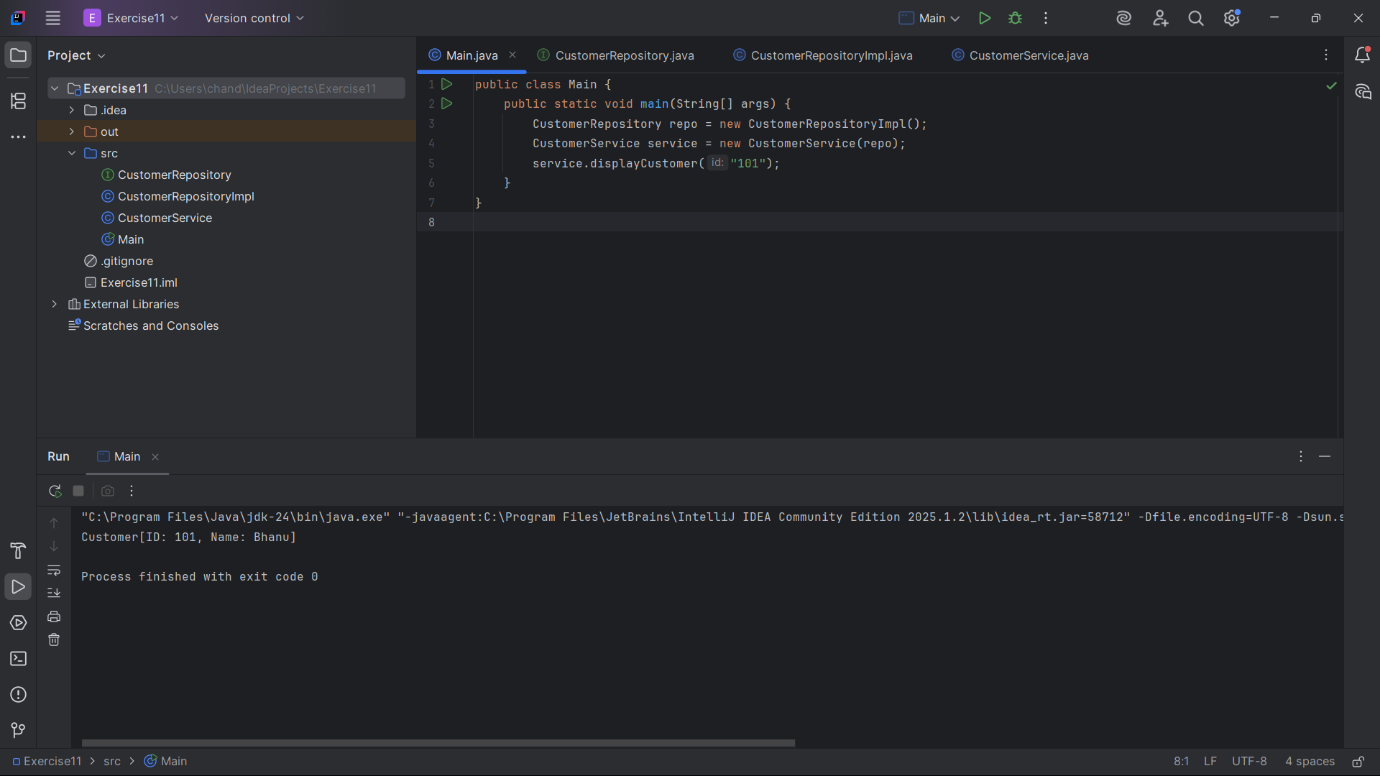
CustomerRepository repo = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repo);

service.displayCustomer("101");

}

}

**OUTPUT:  
**