**Exercise 1: Singleton Pattern**

public class Logger {

private static Logger instance;

private Logger() {}

public static Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

public void log(String message) {

System.out.println("Log: " + message);

}

}

class Main {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

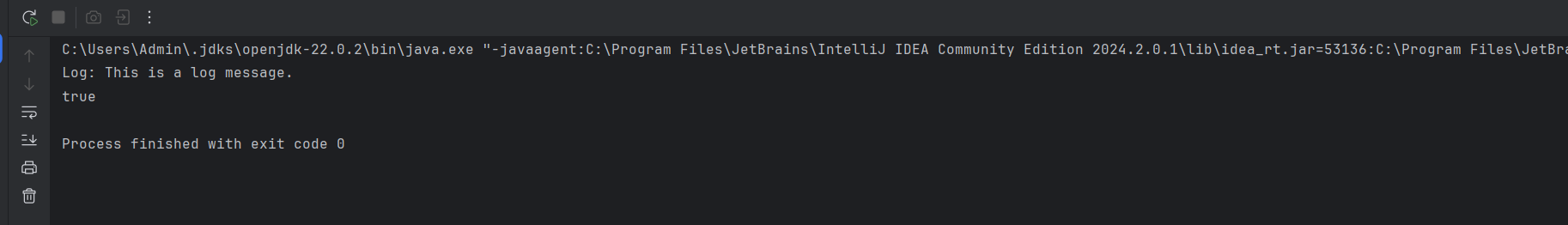
Logger logger2 = Logger.getInstance();

logger1.log("This is a log message.");

System.out.println(logger1 == logger2);

}

}



**Exercise 2: Factory Method Pattern**

interface Document {

void open();

}

class WordDocument implements Document {

public void open() {

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

}

}

class PdfDocument implements Document {

public void open() {

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

}

}

class ExcelDocument implements Document {

public void open() {

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

}

}

abstract class DocumentFactory {

abstract Document createDocument();

}

class WordFactory extends DocumentFactory {

Document createDocument() {

return new WordDocument();

}

}

class PdfFactory extends DocumentFactory {

Document createDocument() {

return new PdfDocument();

}

}

class ExcelFactory extends DocumentFactory {

Document createDocument() {

return new ExcelDocument();

}

}

public class Factory {

public static void main(String[] args) {

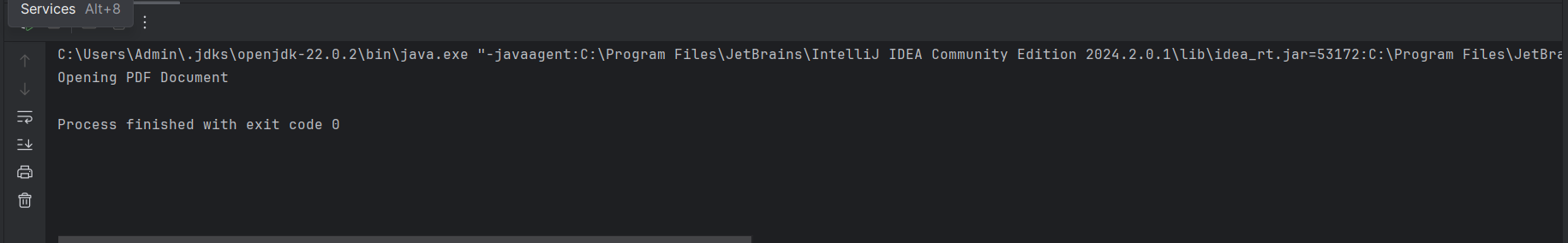
DocumentFactory factory = new PdfFactory();

Document doc = factory.createDocument();

doc.open();

}

}



**Exercise 3: Builder Pattern**

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 void showSpecs() {

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

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

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

}

public static class Builder {

private String CPU;

private String RAM;

private String storage;

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 Computer build() {

return new Computer(this);

}

}

}

public class Comp {

public static void main(String[] args) {

Computer comp = new Computer.Builder()

.setCPU("i7")

.setRAM("16GB")

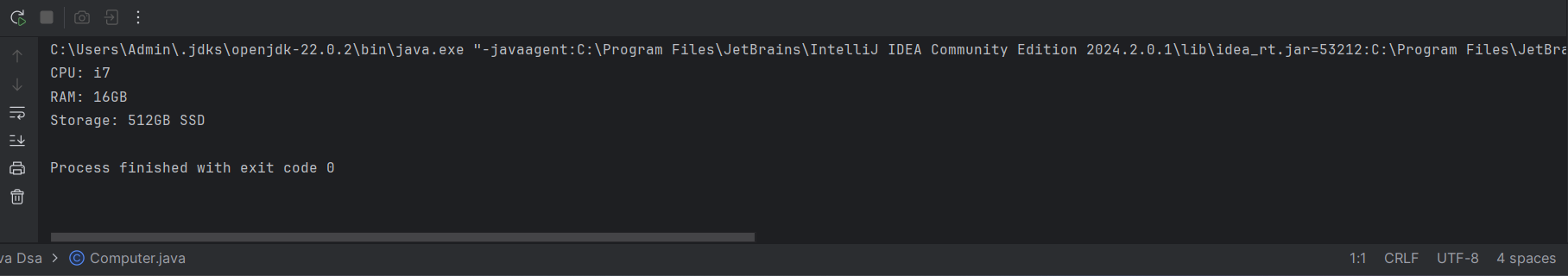
.setStorage("512GB SSD")

.build();

comp.showSpecs();

}

}



**Exercise 4: Adapter Pattern**

interface PaymentProcessor {

void processPayment();

}

class StripeGateway {

public void makePayment() {

System.out.println("Payment via Stripe");

}

}

class StripeAdapter implements PaymentProcessor {

private StripeGateway stripe;

public StripeAdapter(StripeGateway stripe) {

this.stripe = stripe;

}

public void processPayment() {

stripe.makePayment();

}

}

public class Adapter {

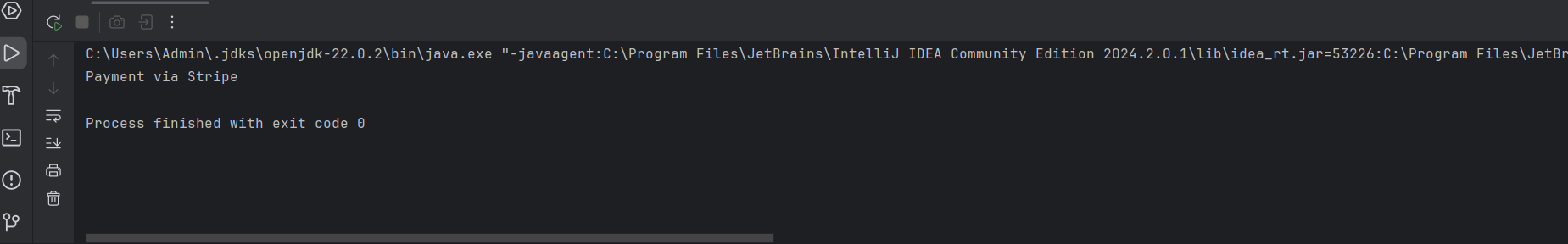
public static void main(String[] args) {

PaymentProcessor processor = new StripeAdapter(new StripeGateway());

processor.processPayment();

}

}



**Exercise 5: Decorator Pattern**

interface Notifier {

void send();

}

class EmailNotifier implements Notifier {

public void send() {

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

}

}

abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

public void send() {

notifier.send();

}

}

class SMSNotifier extends NotifierDecorator {

public SMSNotifier(Notifier notifier) {

super(notifier);

}

public void send() {

super.send();

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

}

}

public class Decorater {

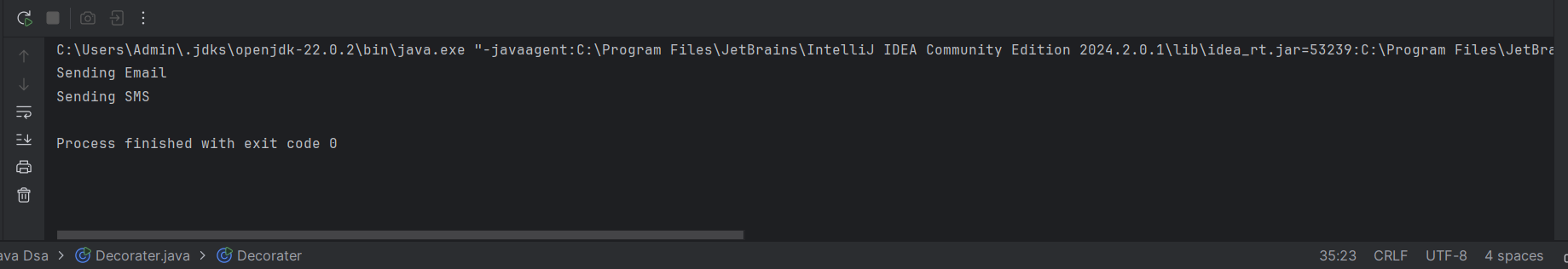
public static void main(String[] args) {

Notifier notifier = new SMSNotifier(new EmailNotifier());

notifier.send();

}

}



**Exercise 6: Proxy Pattern**

interface Image {

void display();

}

class RealImage implements Image {

private String filename;

public RealImage(String filename) {

this.filename = filename;

loadFromDisk();

}

private void loadFromDisk() {

System.out.println("Loading " + filename);

}

public void display() {

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

}

}

class ProxyImage implements Image {

private RealImage realImage;

private String filename;

public ProxyImage(String filename) {

this.filename = filename;

}

public void display() {

if (realImage == null) {

realImage = new RealImage(filename);

}

realImage.display();

}

}

public class Proxypattern {

public static void main(String[] args) {

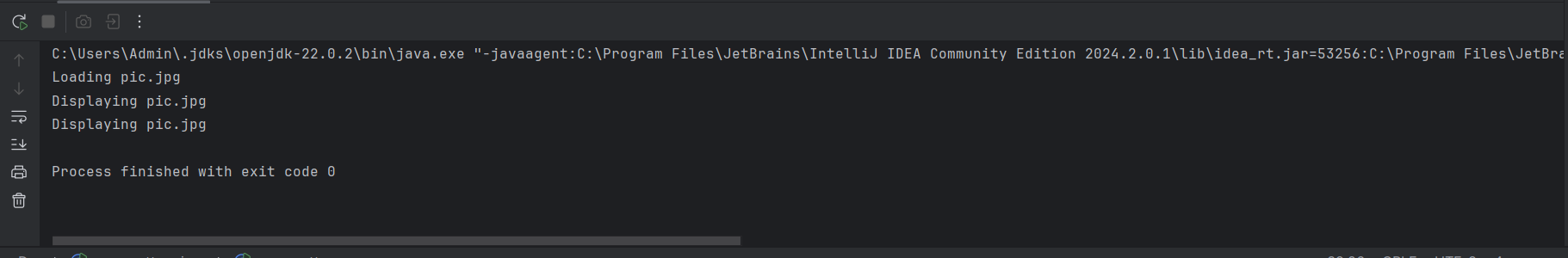
Image image = new ProxyImage("pic.jpg");

image.display();

image.display();

}

}



**Exercise 7: Observer Pattern**

import java.util.\*;

interface Observer {

void update(float price);

}

interface Stock {

void register(Observer o);

void deregister(Observer o);

void notifyObservers();

}

class StockMarket implements Stock {

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

private float price;

public void register(Observer o) {

observers.add(o);

}

public void deregister(Observer o) {

observers.remove(o);

}

public void notifyObservers() {

for (Observer o : observers) {

o.update(price);

}

}

public void setPrice(float price) {

this.price = price;

notifyObservers();

}

}

class MobileApp implements Observer {

public void update(float price) {

System.out.println("MobileApp: Stock price updated to " + price);

}

}

public class Observerpattern {

public static void main(String[] args) {

StockMarket stock = new StockMarket();

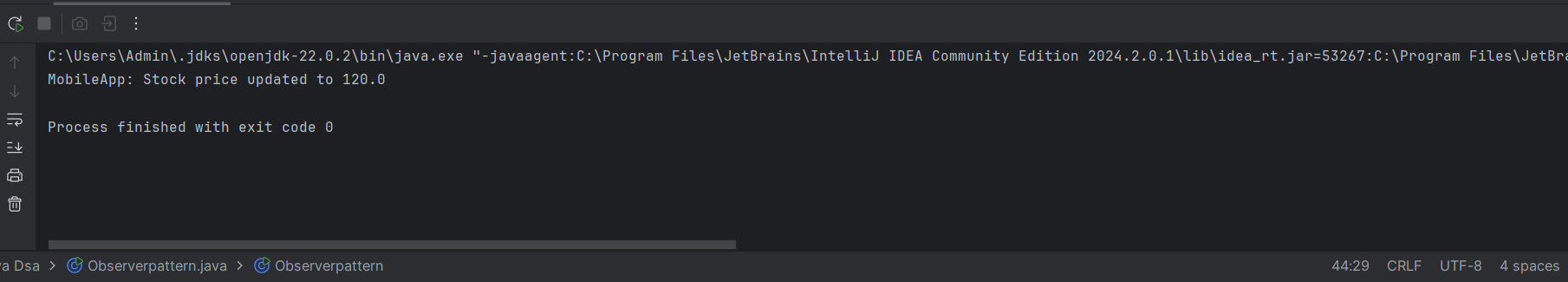
Observer mobile = new MobileApp();

stock.register(mobile);

stock.setPrice(120.0f);

}

}



**Exercise 8: Strategy Pattern**

interface PaymentStrategy {

void pay(int amount);

}

class CreditCardPayment implements PaymentStrategy {

public void pay(int amount) {

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

}

}

class PayPalPayment implements PaymentStrategy {

public void pay(int amount) {

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

}

}

class PaymentContext {

private PaymentStrategy strategy;

public PaymentContext(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void executePayment(int amount) {

strategy.pay(amount);

}

}

public class Strategypattern {

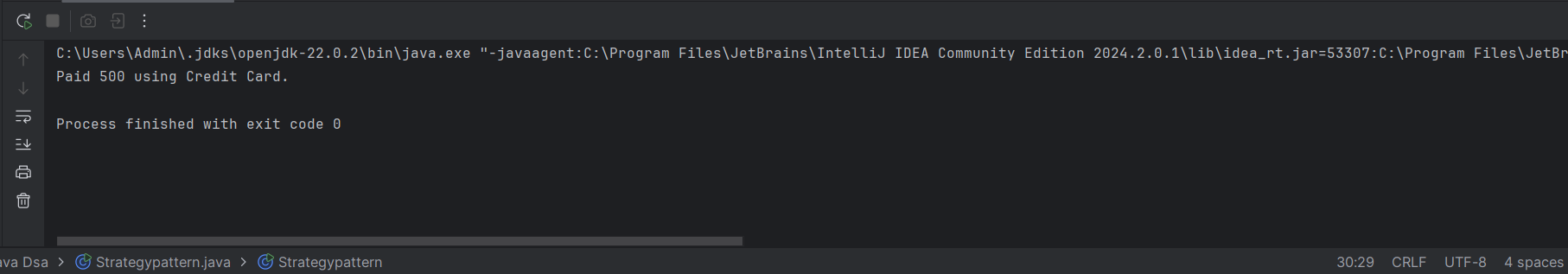
public static void main(String[] args) {

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

context.executePayment(500);

}

}



**Exercise 9: Command Pattern**

interface Command {

void execute();

}

class Light {

public void on() {

System.out.println("Light On");

}

public void off() {

System.out.println("Light Off");

}

}

class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.on();

}

}

class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

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 Commandpattern {

public static void main(String[] args) {

Light light = new Light();

Command on = new LightOnCommand(light);

Command off = new LightOffCommand(light);

RemoteControl remote = new RemoteControl();

remote.setCommand(on);

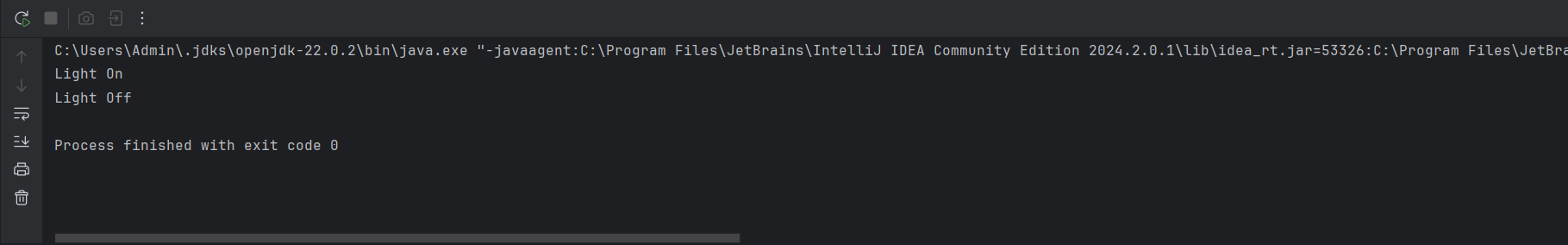
remote.pressButton();

remote.setCommand(off);

remote.pressButton();

}

}



**Exercise 10: MVC Pattern**

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

}

class StudentView {

public void displayStudentDetails(Student student) {

System.out.println("Student: " + student.getName() + ", ID: " + student.getId() + ", Grade: " + student.getGrade());

}

}

class StudentController {

private Student model;

private StudentView view;

public StudentController(Student model, StudentView view) {

this.model = model;

this.view = view;

}

public void updateView() {

view.displayStudentDetails(model);

}

public void setStudentName(String name) {

model.setName(name);

}

}

public class MVC {

public static void main(String[] args) {

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

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

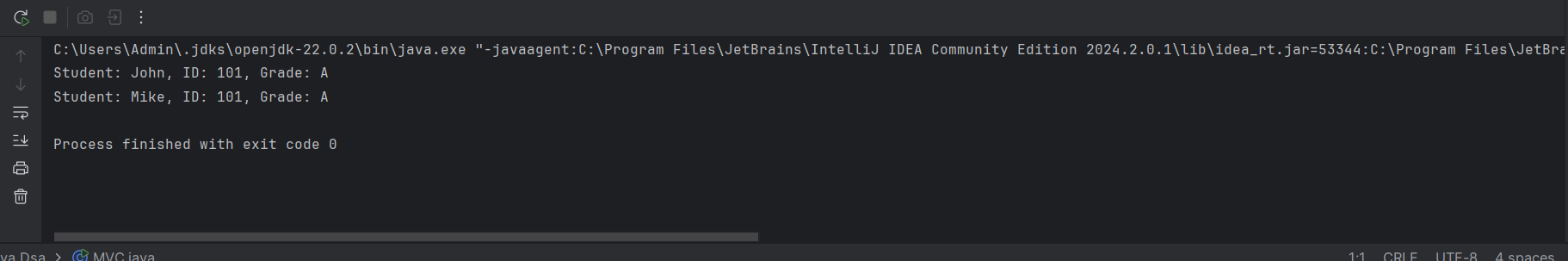
controller.updateView();

controller.setStudentName("Mike");

controller.updateView();

}

}



**Exercise 11: Dependency Injection**

interface CustomerRepository {

String findCustomerById(int id);

}

class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(int id) {

return "Customer#" + id;

}

}

class CustomerService {

private CustomerRepository repository;

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void displayCustomer(int id) {

System.out.println(repository.findCustomerById(id));

}

}

public class DpInjc{

public static void main(String[] args) {

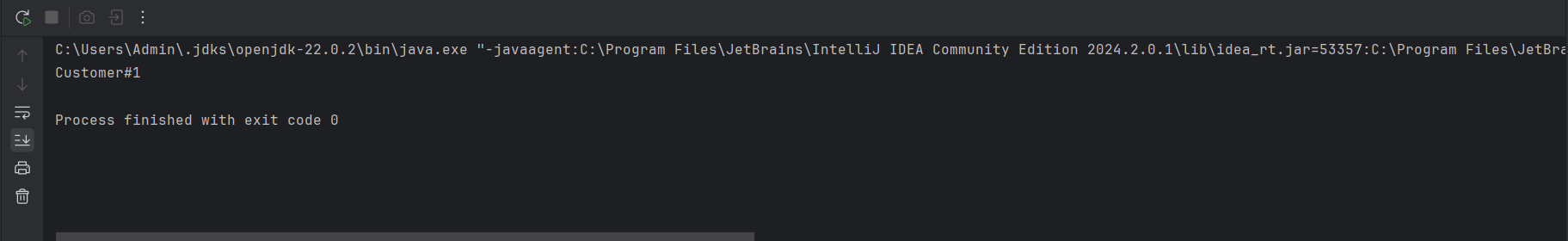
CustomerRepository repo = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repo);

service.displayCustomer(1);

}

}



Thank You