**Exercise 1: Implementing the Singleton Pattern**

class Logger {

private static Logger instance = new Logger();

private Logger() {

}

public static Logger getInstance() {

return instance;

}

}

class LoggerTest {

public static void main(String[] args) {

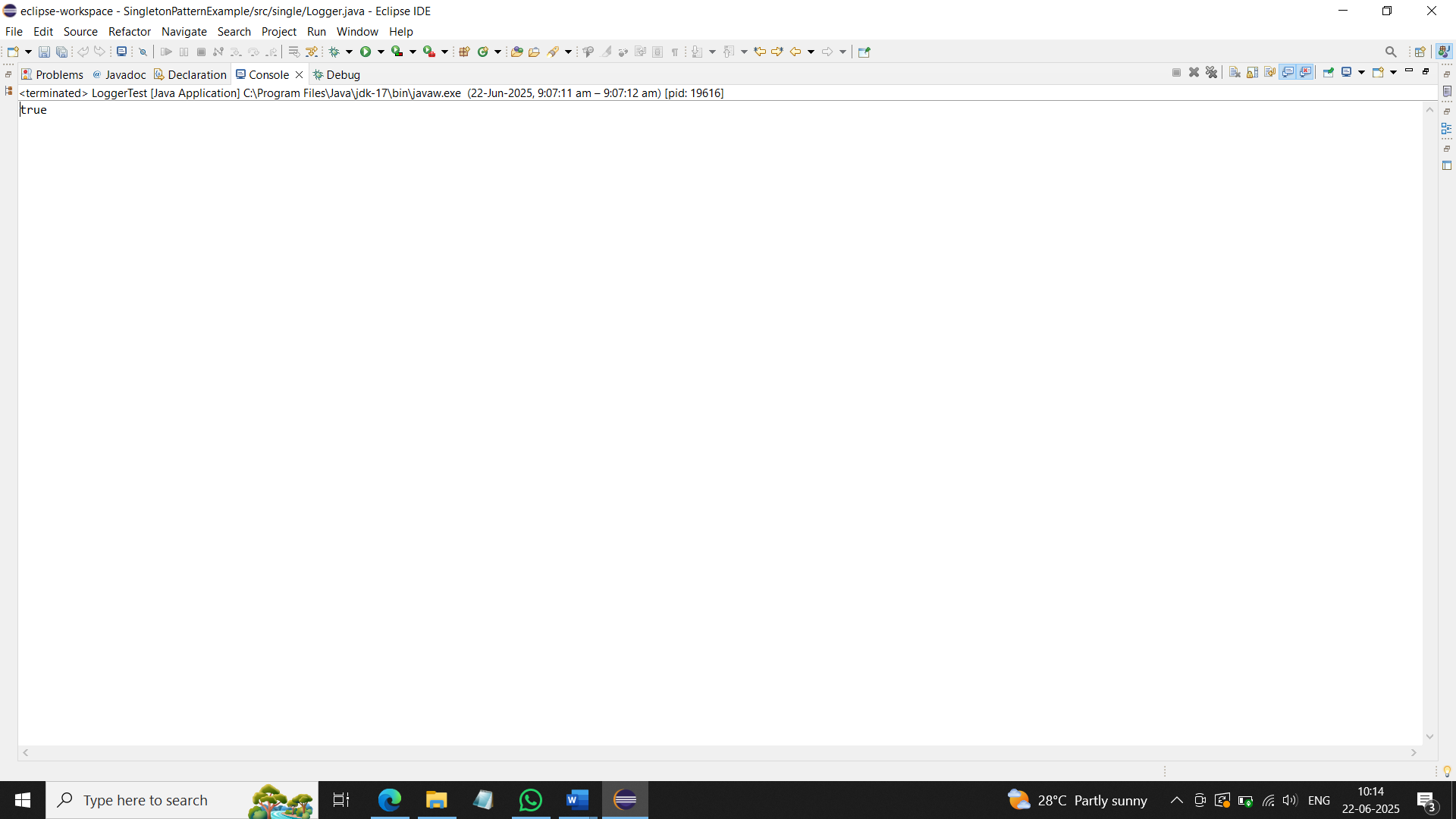
Logger l1 = Logger.*getInstance*();

Logger l2 = Logger.*getInstance*();

System.*out*.println(l1 == l2);

}

}



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

interface Document {

void open();

}

class WordDocument implements Document {

public void open() {

System.*out*.println("Word opened");

}

}

class PdfDocument implements Document {

public void open() {

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

}

}

class ExcelDocument implements Document {

public void open() {

System.*out*.println("Excel opened");

}

}

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 FactoryExample {

public static void main(String[] args) {

DocumentFactory f1 = new WordFactory();

Document d1 = f1.createDocument();

d1.open();

DocumentFactory f2 = new PdfFactory();

Document d2 = f2.createDocument();

d2.open();

}

}

A screenshot of a computer

AI-generated content may be incorrect.

**Exercise 3: Implementing the Builder Pattern**

class Computer {

private String cpu;

private String ram;

private String storage;

private Computer(Builder b) {

this.cpu = b.cpu;

this.ram = b.ram;

this.storage = b.storage;

}

public void display() {

System.*out*.println(cpu + " " + ram + " " + 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 BuilderExample {

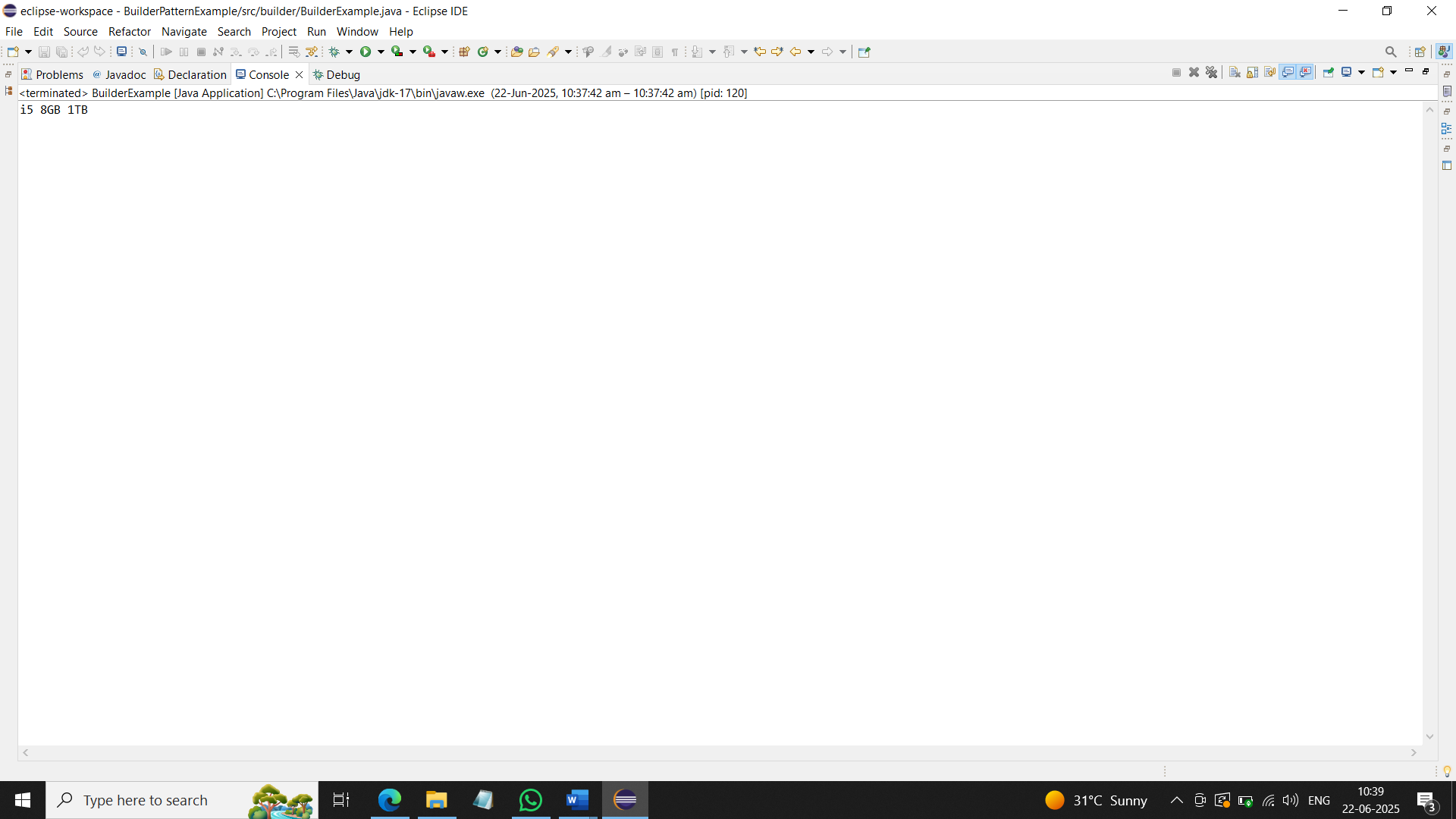
public static void main(String[] args) {

Computer c1 = new Computer.Builder().setCpu("i5").setRam("8GB").setStorage("1TB").build();

c1.display();

}

}



**Exercise 4: Implementing the Adapter Pattern**

interface PaymentProcessor {

void processPayment();

}

class Razorpay {

void makePayment() {

System.*out*.println("Payment made via Razorpay");

}

}

class RazorpayAdapter implements PaymentProcessor {

private Razorpay r;

public RazorpayAdapter(Razorpay r) {

this.r = r;

}

public void processPayment() {

r.makePayment();

}

}

public class AdapterExample {

public static void main(String[] args) {

Razorpay r = new Razorpay();

PaymentProcessor p = new RazorpayAdapter(r);

p.processPayment();

}

}

A screenshot of a computer

AI-generated content may be incorrect.

**Exercise 5: Implementing the 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;

}

}

class SMSNotifier extends NotifierDecorator {

public SMSNotifier(Notifier notifier) {

super(notifier);

}

public void send() {

notifier.send();

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

}

}

class SlackNotifier extends NotifierDecorator {

public SlackNotifier(Notifier notifier) {

super(notifier);

}

public void send() {

notifier.send();

System.*out*.println("Sending Slack Message");

}

}

public class DecoratorExample {

public static void main(String[] args) {

Notifier n = new SlackNotifier(new SMSNotifier(new EmailNotifier()));

n.send();

}

}

A screenshot of a computer

AI-generated content may be incorrect.

**Exercise 6: Implementing the Proxy Pattern**

interface Image {

void display();

}

class RealImage implements Image {

private String file;

public RealImage(String file) {

this.file = file;

load();

}

private void load() {

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

}

public void display() {

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

}

}

class ProxyImage implements Image {

private RealImage realImage;

private String file;

public ProxyImage(String file) {

this.file = file;

}

public void display() {

if (realImage == null) {

realImage = new RealImage(file);

}

realImage.display();

}

}

public class ProxyExample {

public static void main(String[] args) {

Image img = new ProxyImage("photo.jpg");

img.display();

img.display();

}

}

A screenshot of a computer

AI-generated content may be incorrect.

**Exercise 7: Implementing the Observer Pattern**

import java.util.\*;

interface Observer {

void update(String stock);

}

interface Stock {

void register(Observer o);

void deregister(Observer o);

void notifyObservers();

}

class StockMarket implements Stock {

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

private String stock;

public void register(Observer o) {

list.add(o);

}

public void deregister(Observer o) {

list.remove(o);

}

public void notifyObservers() {

for (Observer o : list) {

o.update(stock);

}

}

public void changeStock(String stock) {

this.stock = stock;

notifyObservers();

}

}

class MobileApp implements Observer {

public void update(String stock) {

System.*out*.println("Mobile received: " + stock);

}

}

class WebApp implements Observer {

public void update(String stock) {

System.*out*.println("Web received: " + stock);

}

}

public class ObserverExample {

public static void main(String[] args) {

StockMarket sm = new StockMarket();

Observer m = new MobileApp();

Observer w = new WebApp();

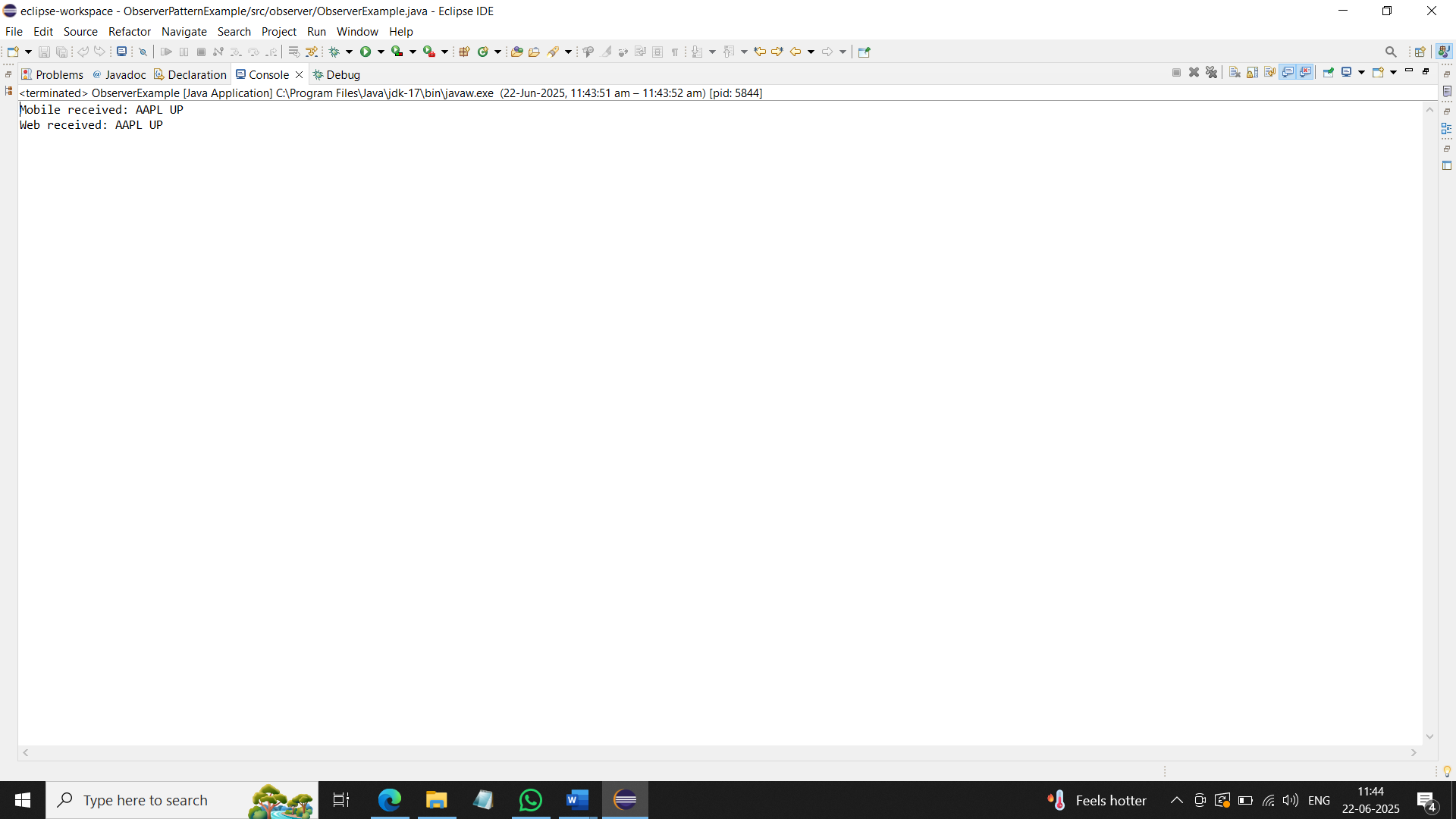
sm.register(m);

sm.register(w);

sm.changeStock("AAPL UP");

}

}



**Exercise 8: Implementing the Strategy Pattern**

interface PaymentStrategy {

void pay();

}

class CreditCardPayment implements PaymentStrategy {

public void pay() {

System.*out*.println("Paid with Credit Card");

}

}

class PayPalPayment implements PaymentStrategy {

public void pay() {

System.*out*.println("Paid with PayPal");

}

}

class PaymentContext {

private PaymentStrategy strategy;

public void setStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void pay() {

strategy.pay();

}

}

public class StrategyExample {

public static void main(String[] args) {

PaymentContext p = new PaymentContext();

p.setStrategy(new CreditCardPayment());

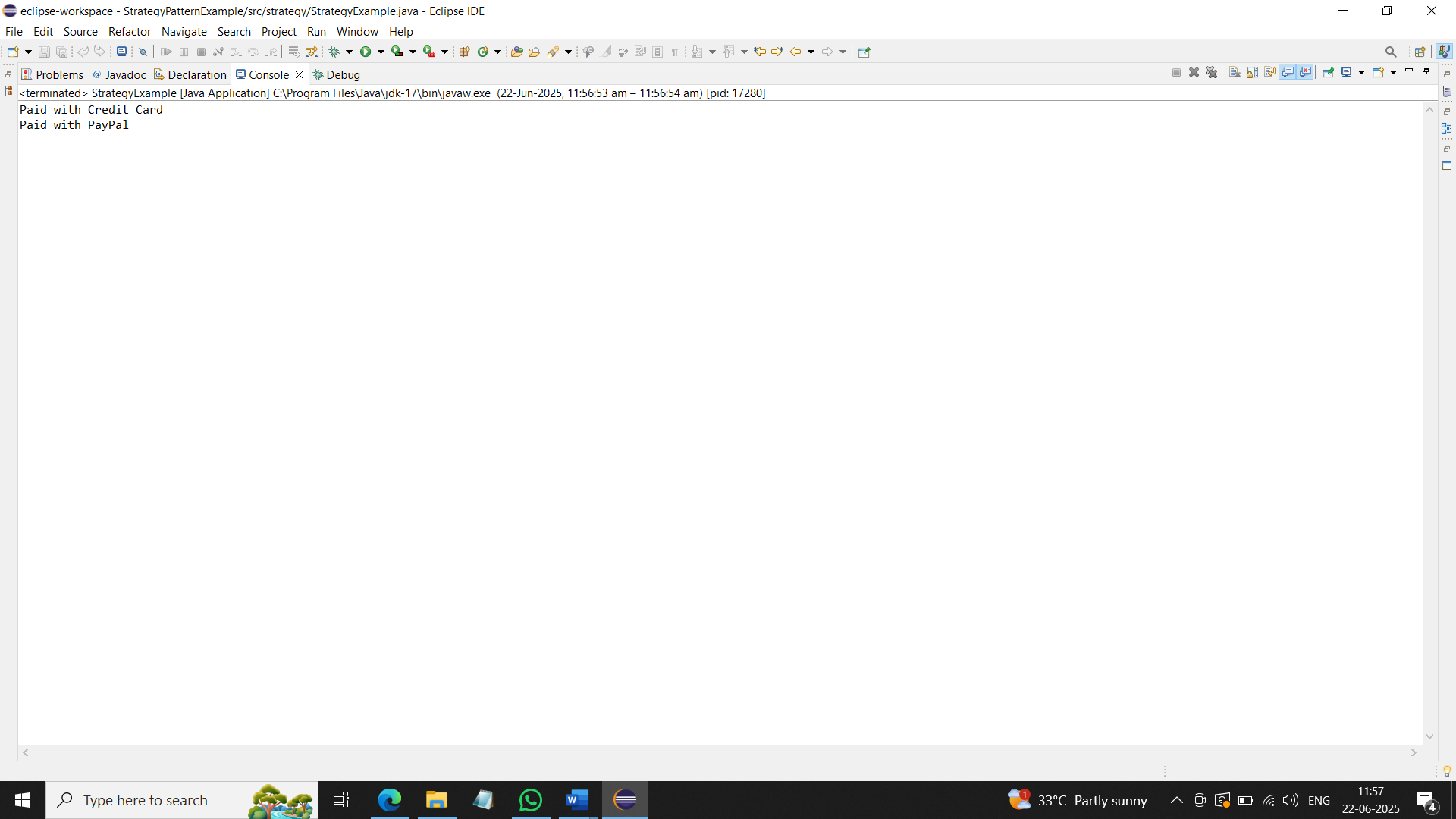
p.pay();

p.setStrategy(new PayPalPayment());

p.pay();

}

}



**Exercise 9: Implementing the Command Pattern**

interface Command {

void execute();

}

class Light {

void on() {

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

}

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 press() {

command.execute();

}

}

public class CommandExample {

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);

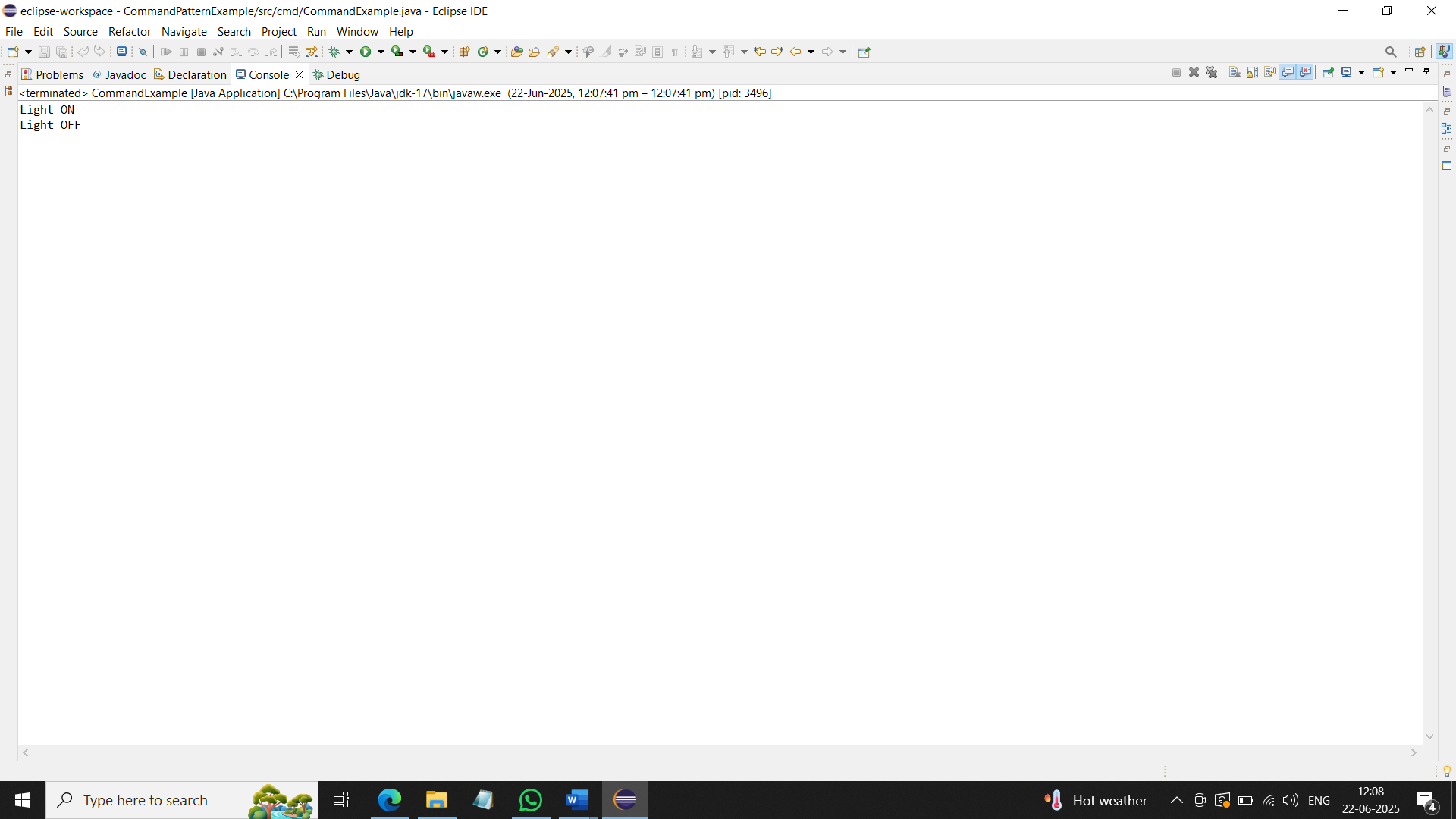
remote.press();

remote.setCommand(off);

remote.press();

}

}



**Exercise 10: Implementing the MVC Pattern**

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 setGrade(String grade) {

this.grade = grade;

}

}

class StudentView {

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

System.out.println(name + " " + id + " " + grade);

}

}

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

}

public void setStudentName(String name) {

model.setName(name);

}

public void setStudentGrade(String grade) {

model.setGrade(grade);

}

}

public class MVCExample {

public static void main(String[] args) {

Student s = new Student("Mithra", 101, "A");

StudentView v = new StudentView();

StudentController c = new StudentController(s, v);

c.updateView();

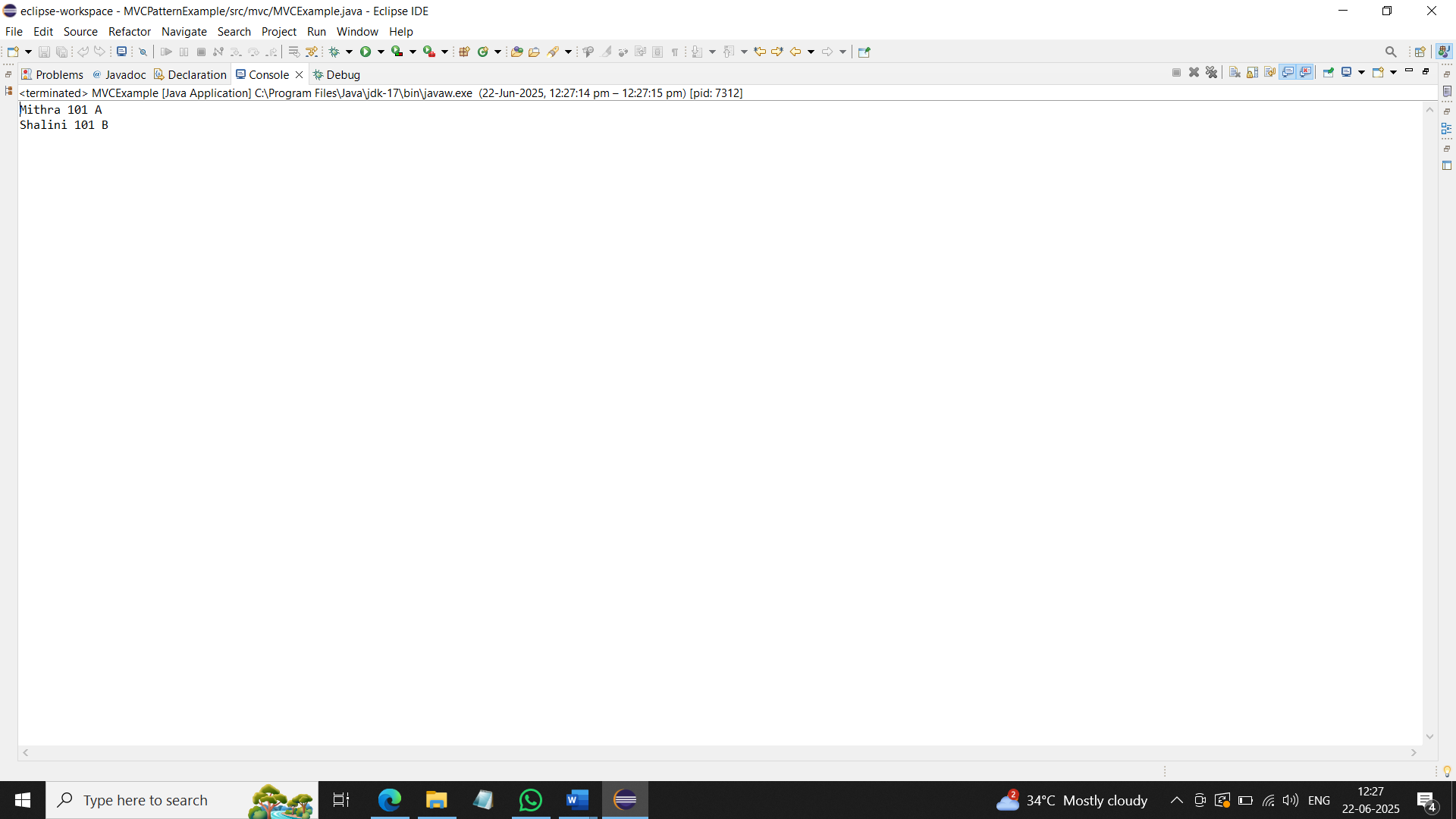
c.setStudentName("Shalini");

c.setStudentGrade("B");

c.updateView();

}

}



**Exercise 11: Implementing Dependency Injection**

interface CustomerRepository {

String findCustomerById(int id);

}

class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(int id) {

return "Customer " + id;

}

}

class CustomerService {

private CustomerRepository repo;

public CustomerService(CustomerRepository repo) {

this.repo = repo;

}

public void showCustomer(int id) {

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

}

}

public class DIExample {

public static void main(String[] args) {

CustomerRepository repo = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repo);

service.showCustomer(10);

}

}

