**Exercise 1: Implementing the Singleton Pattern**

**Logger.java**public class Logger {

private static Logger instance;

private Logger() {

System.out.println("Logger initialized.");

}

public static Logger getInstance() {

if(instance == null) {

instance = new Logger();

}

return instance;

}

public void log(String message) {

System.out.println("[LOG]: " + message);

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

Logger l1 = Logger.getInstance();

l1.log("This is the first log.");

Logger l2 = Logger.getInstance();

l2.log("This is the second log.");

if(l1 == l2)

System.out.println("Only one instance used.");

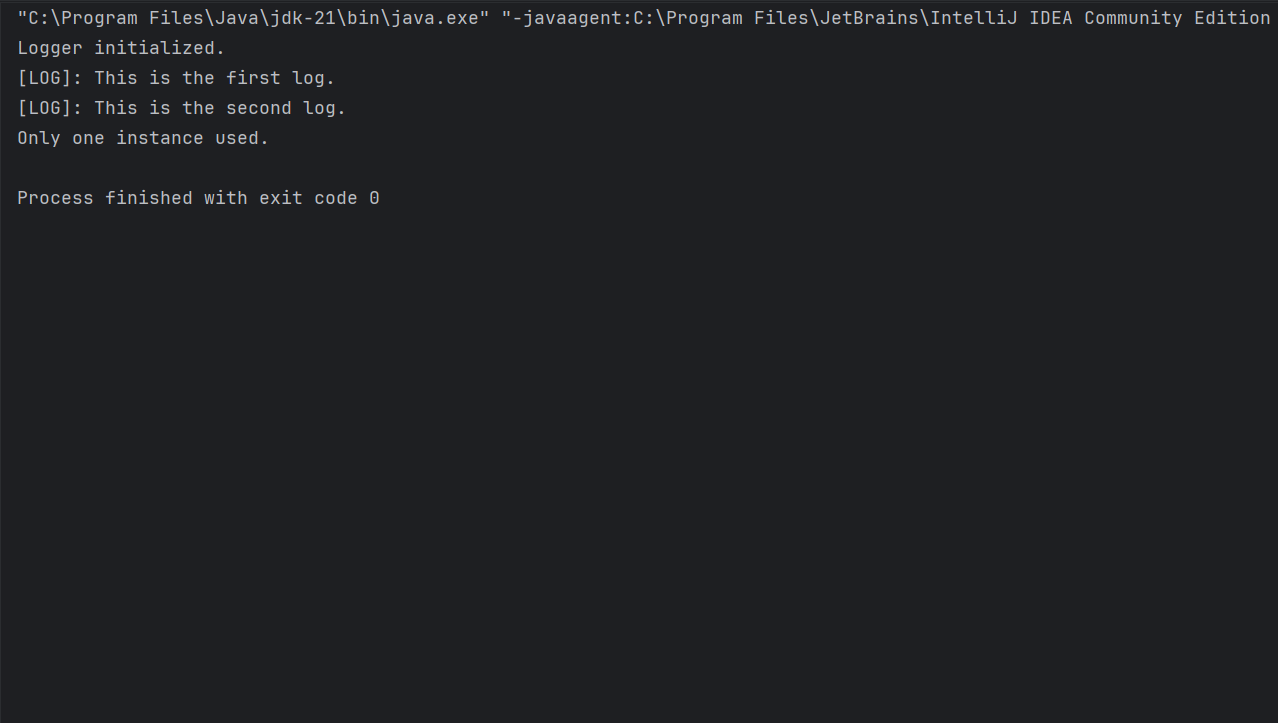
else

System.out.println("Multiple instances exist.");

}

}

**Output:**



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

**Code:**

**Document.java**

public interface Document {

void open();

}

**WordDocument.java**

public class WordDocument implements Document {

public void open() {

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

}

}

**PdfDocument.java**

public class PdfDocument implements Document {

public void open() {

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

}

}

**ExcelDocument.java**

public class ExcelDocument implements Document {

public void open() {

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

}

}

**DocumentFactory.java**

public abstract class DocumentFactory {

public abstract Document createDocument();

}

**WordFactory.java**

public class WordFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

**PdfFactory.java**

public class PdfFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

**ExcelFactory.java**

public class ExcelFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

DocumentFactory wordFactory = new WordFactory();

DocumentFactory pdfFactory = new PdfFactory();

DocumentFactory excelFactory = new ExcelFactory();

Document word = wordFactory.createDocument();

Document pdf = pdfFactory.createDocument();

Document excel = excelFactory.createDocument();

word.open();

pdf.open();

excel.open();

}

}

**Output:**

****

**Exercise 3: Implementing the Builder Pattern**

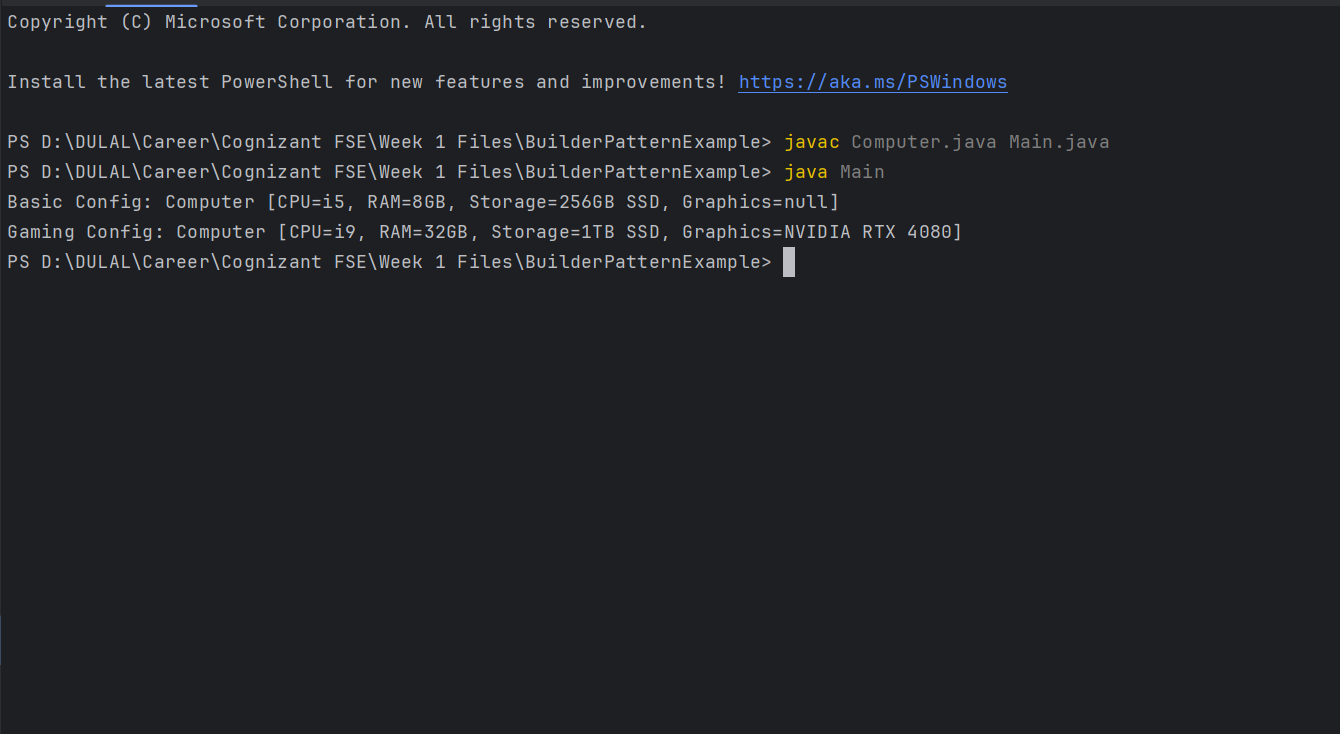
**Code:**

**Main.java**

public class Main {  
 public static void main(String[] args) {  
 Computer basic = new Computer.Builder()  
 .setCPU("i5")  
 .setRAM("8GB")  
 .setStorage("256GB SSD")  
 .build();  
  
 Computer gaming = new Computer.Builder()  
 .setCPU("i9")  
 .setRAM("32GB")  
 .setStorage("1TB SSD")  
 .setGraphicsCard("NVIDIA RTX 4080")  
 .build();  
  
 System.out.println("Basic Config: " + basic);  
 System.out.println("Gaming Config: " + gaming);  
 }  
}

**Computer.java**public class Computer {  
 private String CPU;  
 private String RAM;  
 private String storage;  
 private String graphicsCard;  
  
 private Computer(Builder builder) {  
 this.CPU = builder.CPU;  
 this.RAM = builder.RAM;  
 this.storage = builder.storage;  
 this.graphicsCard = builder.graphicsCard;  
 }  
  
 public String toString() {  
 return "Computer [CPU=" + CPU + ", RAM=" + RAM + ", Storage=" + storage + ", Graphics=" + graphicsCard + "]";  
 }  
  
 public static class Builder {  
 private String CPU;  
 private String RAM;  
 private String storage;  
 private String graphicsCard;  
  
 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 setGraphicsCard(String graphicsCard) {  
 this.graphicsCard = graphicsCard;  
 return this;  
 }  
  
 public Computer build() {  
 return new Computer(this);  
 }  
 }  
}

**Output:**

****

**Exercise 4: Implementing the Adapter Pattern**

**Code:**

**Main.java**

public class Main {

public static void main(String[] args) {

PaymentProcessor paypal = new PayPalAdapter();

PaymentProcessor stripe = new StripeAdapter();

paypal.processPayment(1500);

stripe.processPayment(2500);

}

}

**StripeAdapter.java**

public class StripeAdapter implements PaymentProcessor {

private StripeGateway gateway = new StripeGateway();

public void processPayment(double amount) {

gateway.sendPayment(amount);

}

}

**PayPalAdapter.java**

public class PayPalAdapter implements PaymentProcessor {

private PayPalGateway gateway = new PayPalGateway();

public void processPayment(double amount) {

gateway.makePayment(amount);

}

}

**StripeGateway.java**

public class StripeGateway {

public void sendPayment(double amount) {

System.out.println("Stripe processed ₹" + amount);

}

}

**PayPalGateway.java**

public class PayPalGateway {

public void makePayment(double amount) {

System.out.println("PayPal processed ₹" + amount);

}

}

**PaymentProcessor.java**

public interface PaymentProcessor {

void processPayment(double amount);

}

**Output:**

****

**Exercise 5: Implementing the Decorator Pattern**

**Code:**

**Main.java**

public class Main {

public static void main(String[] args) {

Notifier base = new EmailNotifier();

Notifier smsDecorator = new SMSNotifierDecorator(base);

Notifier fullDecorator = new SlackNotifierDecorator(smsDecorator);

fullDecorator.send("Server is down!");

}

}

**SlackNotifierDecorator.java**

public class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier wrappee) {

super(wrappee);

}

public void send(String message) {

super.send(message);

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

}

}

**SMSNotifierDecorator.java**

public class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier wrappee) {

super(wrappee);

}

public void send(String message) {

super.send(message);

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

}

}

**NotifierDecorator.java**

public abstract class NotifierDecorator implements Notifier {

protected Notifier wrappee;

public NotifierDecorator(Notifier wrappee) {

this.wrappee = wrappee;

}

public void send(String message) {

wrappee.send(message);

}

}

**EmailNotifier.java**

public class EmailNotifier implements Notifier {

public void send(String message) {

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

}

}

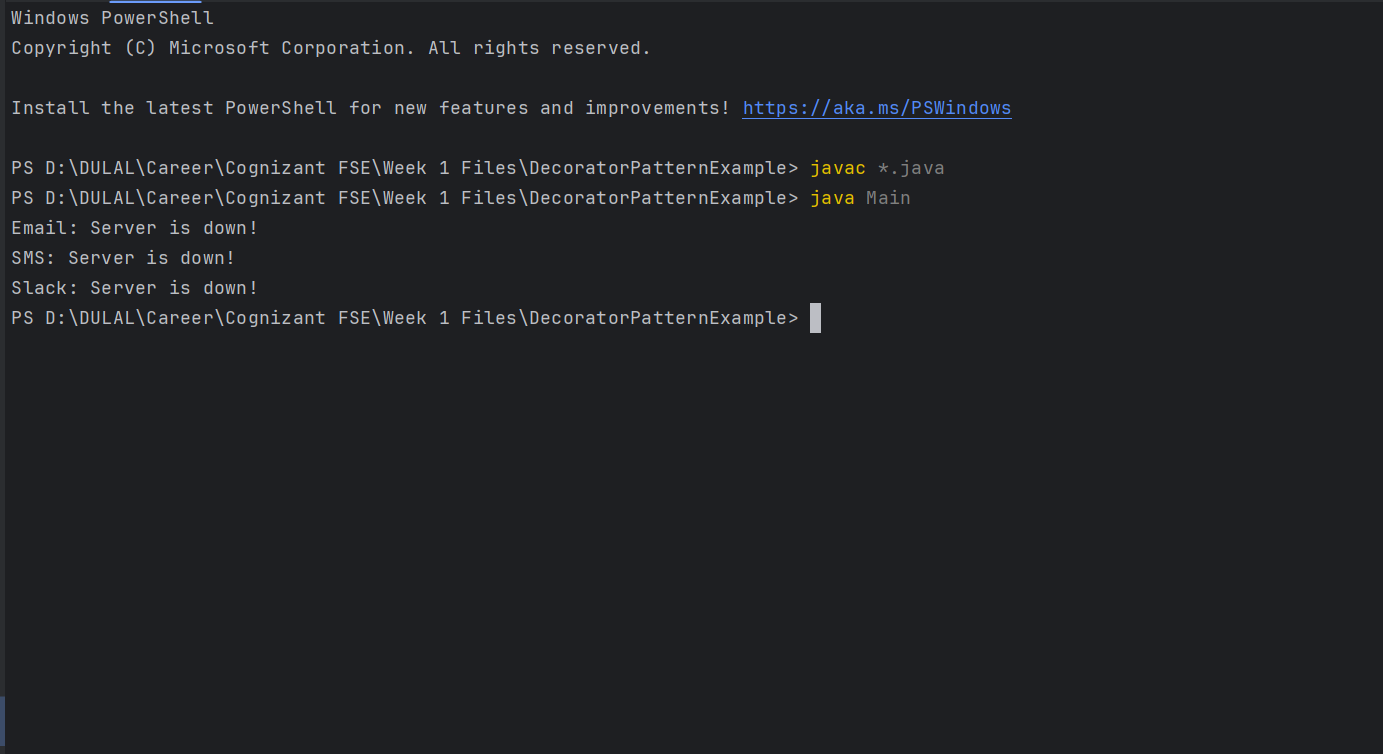
**Notifier.java**

public interface Notifier {

void send(String message);

}

**Output:**

****

**Exercise 6: Implementing the Proxy Pattern**

**Code:**

**Image.java**

public interface Image {

void display();

}

**RealImage.java**

public class RealImage implements Image {

private String filename;

public RealImage(String filename) {

this.filename = filename;

loadFromRemote();

}

private void loadFromRemote() {

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

try { Thread.sleep(1000); } catch(Exception e) {}

}

public void display() {

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

}

}

**ProxyImage.java**

public 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();

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

Image img = new ProxyImage("pic1.jpg");

System.out.println("First call:");

img.display();

System.out.println("\nSecond call:");

img.display();

}

}

**Output:**

****

**Exercise 7: Implementing the Observer Pattern**

**Code:**

**Stock.java**

public interface Stock {

void register(Observer o);

void deregister(Observer o);

void notifyObservers(String stock, double price);

}

**Observer.java**

public interface Observer {

void update(String stock, double price);

}

**StockMarket.java**

import java.util.\*;

public class StockMarket implements Stock {

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

public void register(Observer o) {

observers.add(o);

}

public void deregister(Observer o) {

observers.remove(o);

}

public void notifyObservers(String stock, double price) {

for(Observer o : observers) {

o.update(stock, price);

}

}

public void changeStockPrice(String stock, double price) {

System.out.println("📈 " + stock + " updated to ₹" + price);

notifyObservers(stock, price);

}

}

**MobileApp.java**

public class MobileApp implements Observer {

public void update(String stock, double price) {

System.out.println("MobileApp: " + stock + " -> ₹" + price);

}

}

**WebApp.java**

public class WebApp implements Observer {

public void update(String stock, double price) {

System.out.println("WebApp: " + stock + " -> ₹" + price);

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

StockMarket market = new StockMarket();

Observer mobile = new MobileApp();

Observer web = new WebApp();

market.register(mobile);

market.register(web);

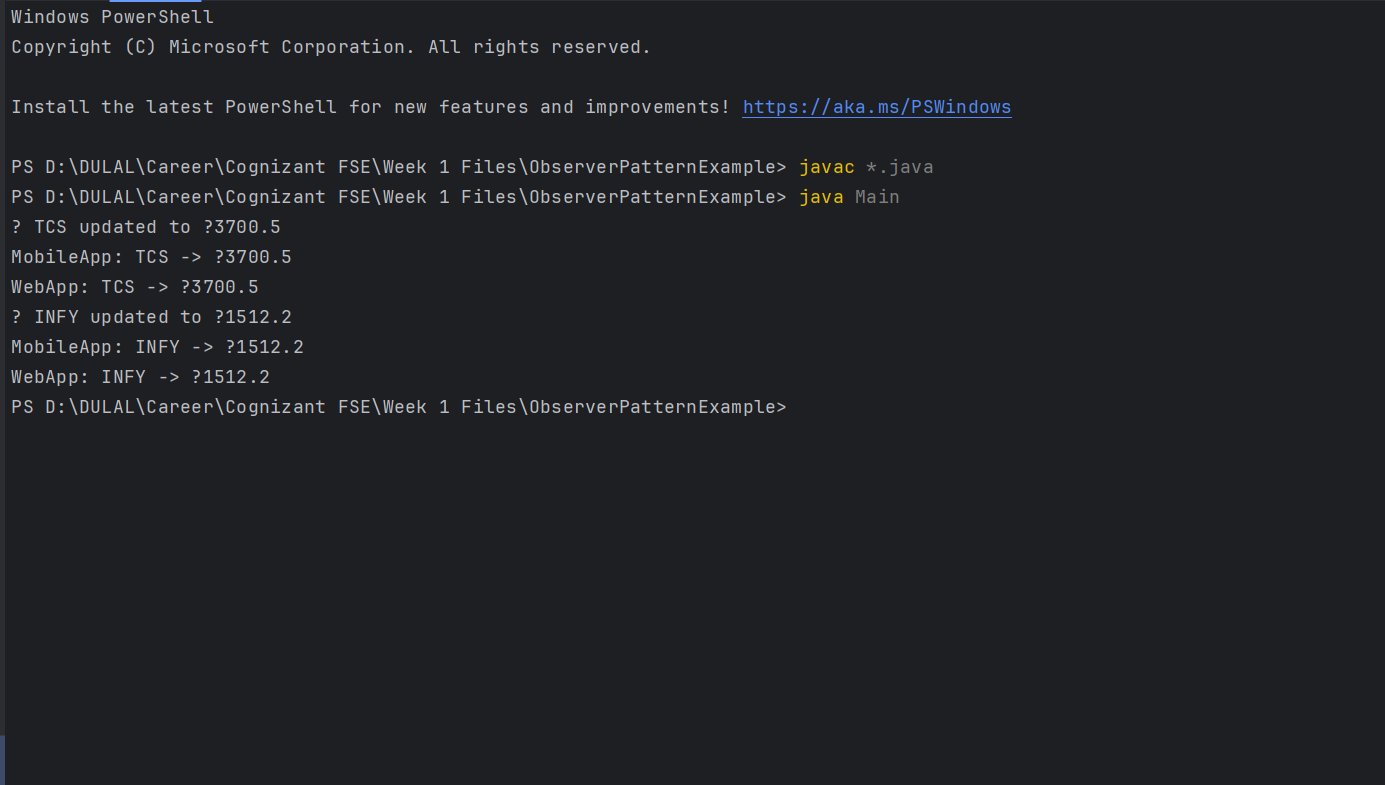
market.changeStockPrice("TCS", 3700.50);

market.changeStockPrice("INFY", 1512.20);

}

}

**Output:**

****

**Exercise 9: Implementing the Command Pattern**

**Code:**

**Main.java**

public class Main {

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();

}

}

**RemoteControl.java**

public class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

command.execute();

}

}

**LightOffCommand.java**

public class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOff();

}

}

**LightOnCommand.java**

public class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOn();

}

}

**Light.java**

public class Light {

public void turnOn() {

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

}

public void turnOff() {

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

}

}

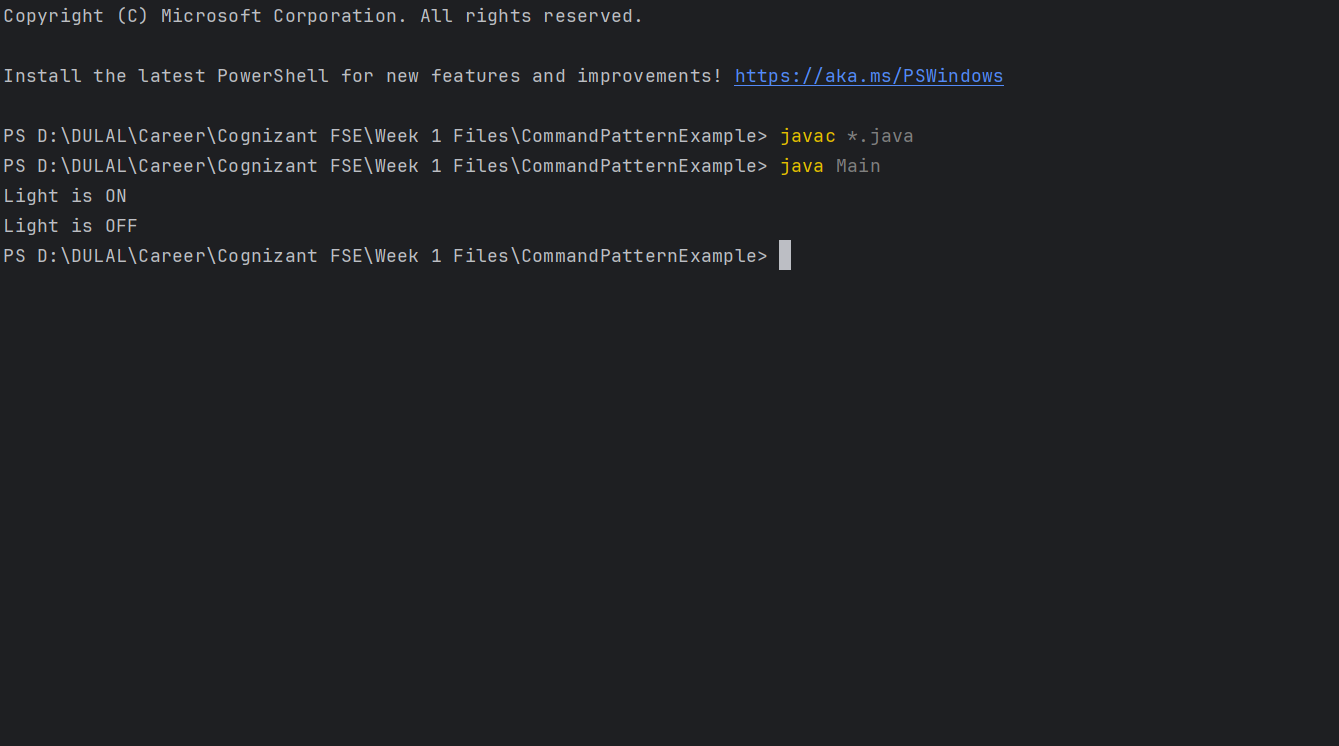
**Command.java**

public interface Command {

void execute();

}

**Output:**

****

**Exercise 10: Implementing the MVC Pattern**

**Code:**

**Student.java**

public class Student {

private String name;

private String id;

private String grade;

public String getName() { return name; }

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

public String getId() { return id; }

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

public String getGrade() { return grade; }

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

}

**StudentView.java**

public class StudentView {

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

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

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

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

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

}

}

**StudentController.java**

public class StudentController {

private Student model;

private StudentView view;

public StudentController(Student model, StudentView view) {

this.model = model;

this.view = view;

}

public void setStudentName(String name) { model.setName(name); }

public void setStudentId(String id) { model.setId(id); }

public void setStudentGrade(String grade) { model.setGrade(grade); }

public void updateView() {

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

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

Student student = new Student();

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

controller.setStudentName("Dulal Roy");

controller.setStudentId("STU123");

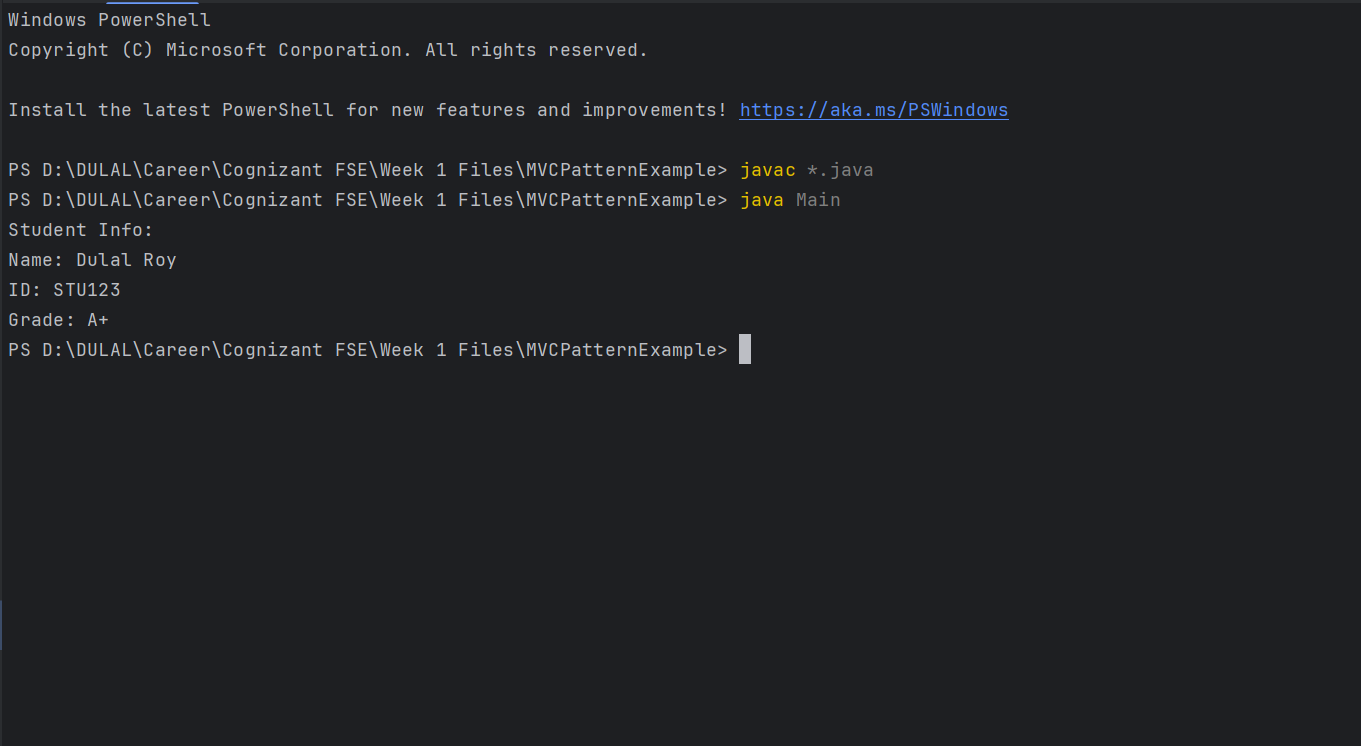
controller.setStudentGrade("A+");

controller.updateView();

}

}

**Output:**

****

**Exercise 11: Implementing Dependency Injection**

**Code:**

**CustomerRepository.java**

public interface CustomerRepository {

String findCustomerById(String id);

}

**CustomerRepositoryImpl.java**

public class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(String id) {

return " Customer: " + id + " – Dulal Roy";

}

}

**CustomerService.java**

public class CustomerService {

private CustomerRepository repository;

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void getCustomerDetails(String id) {

String result = repository.findCustomerById(id);

System.out.println(result);

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

CustomerRepository repo = new CustomerRepositoryImpl();

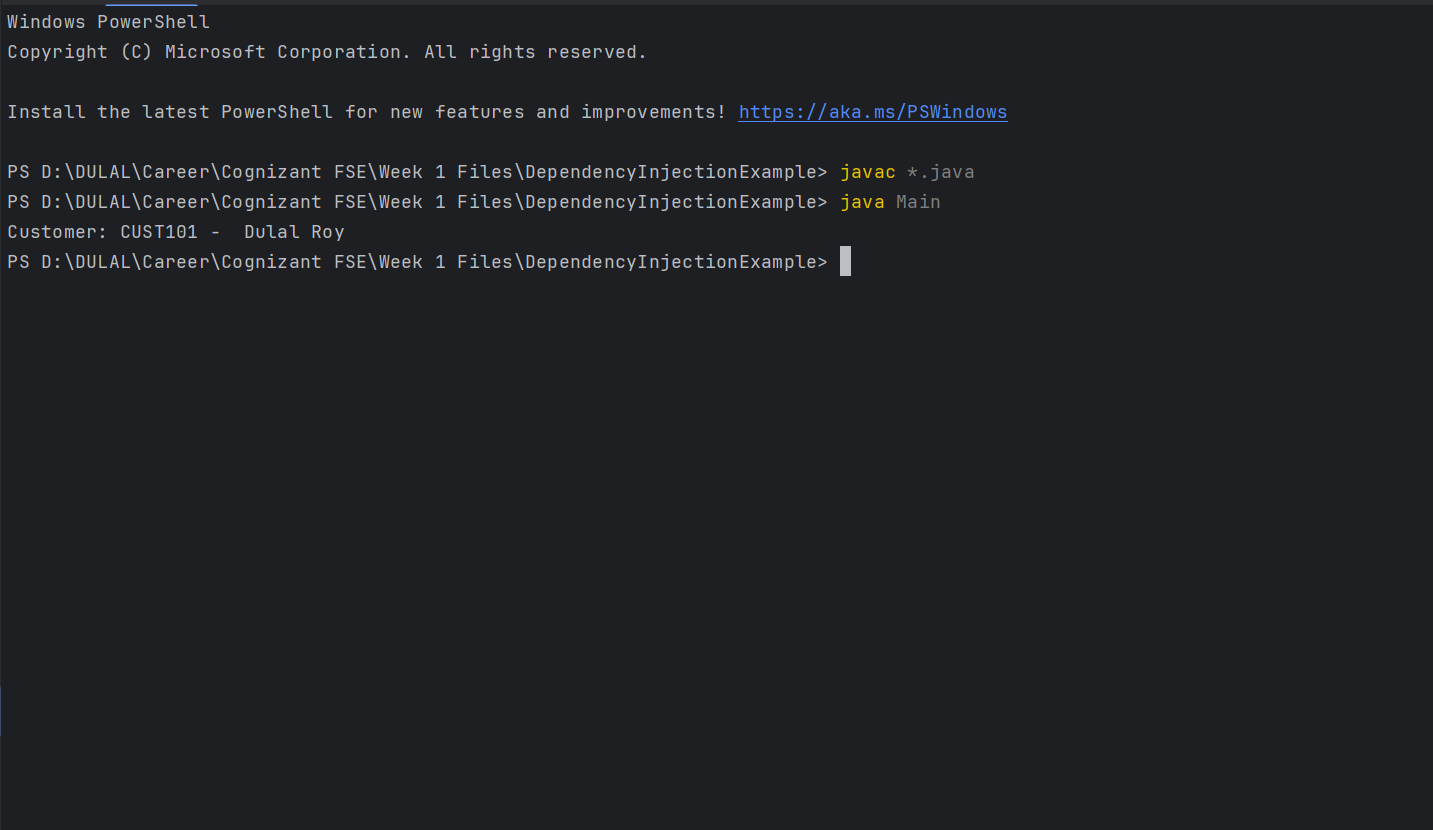
CustomerService service = new CustomerService(repo);

service.getCustomerDetails("CUST101");

}

}

**Output:**

****