**Exercise 1: Implementing the Singleton Pattern**

**This is my code:**

**package** com.example.singleton;

**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 mes)

{

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

}

}

**package** com.example.singleton;

**public** **class** test {

**public** **static** **void** main(String[] args) {

Logger log1=Logger.*getInstance*();

log1.log("first log");

Logger log2=Logger.*getInstance*();

log2.log("second log");

**if**(log1==log2)

{

System.***out***.println("both are same single instances");

}**else** {

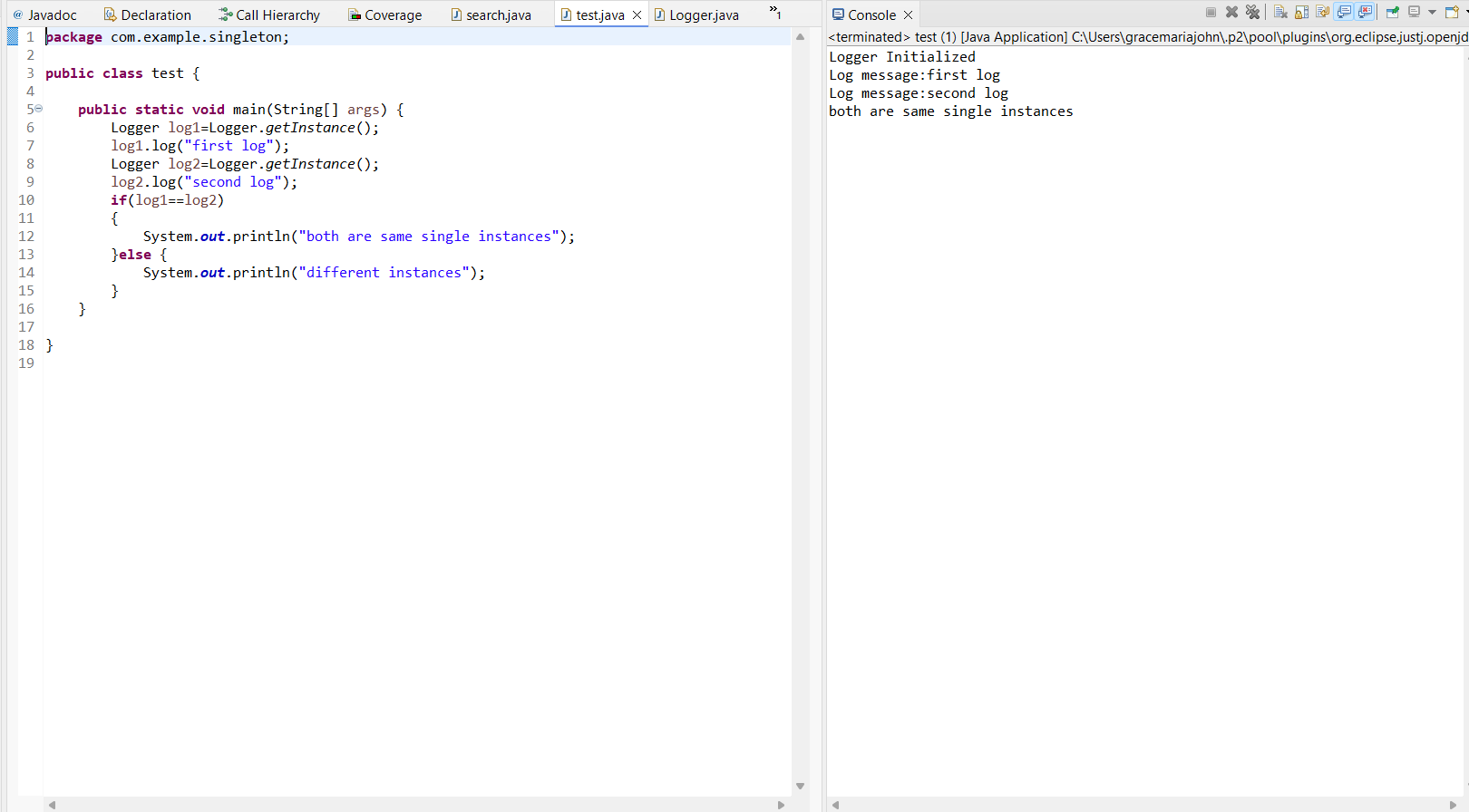
System.***out***.println("different instances");

}

}

}

**This is my Output:**



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

This is my code:

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 {

public Document createDocument() {

return new WordDocument();

}

}

class PdfFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

class ExcelFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

public class FactoryTest {

public static void main(String[] args) {

DocumentFactory factory = new PdfFactory();

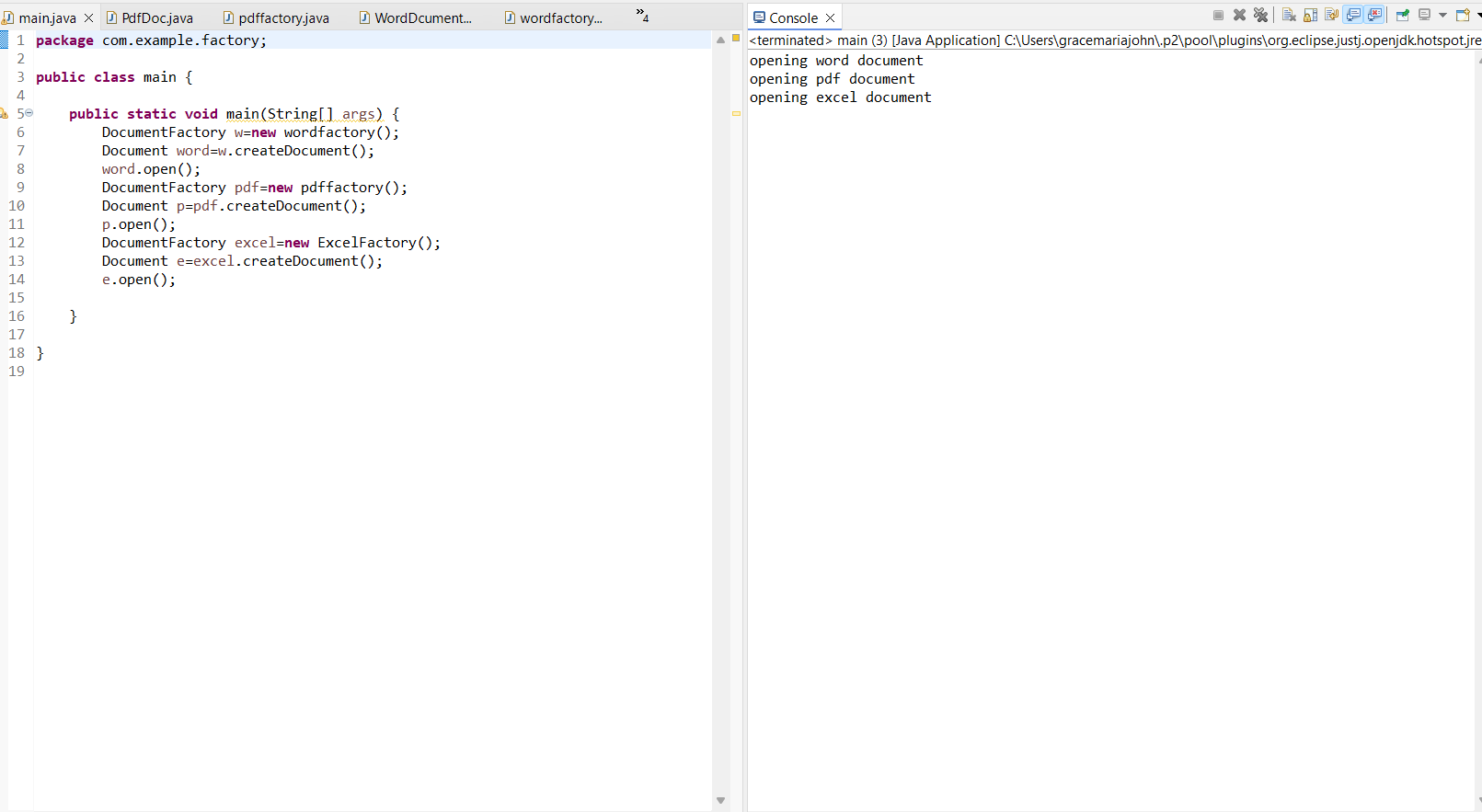
Document doc = factory.createDocument();

doc.open();

}

}

This is my Output:



**Exercise 3: Implementing the Builder Pattern**

This is my code:

package com.example.builder;

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(String CPU, String RAM) {

this.CPU = CPU;

this.RAM = RAM;

}

public Builder setStorage(String storage) {

this.storage = storage;

return this;

}

public Computer build() {

return new Computer(this);

}

}

}

package com.example.builder;

public class main {

public static void main(String[] args) {

Computer c=new Computer.Builder("intel i7","16GB").setStorage("512 GB").build();

c.showSpecs();

System.*out*.println("\nnext computer");

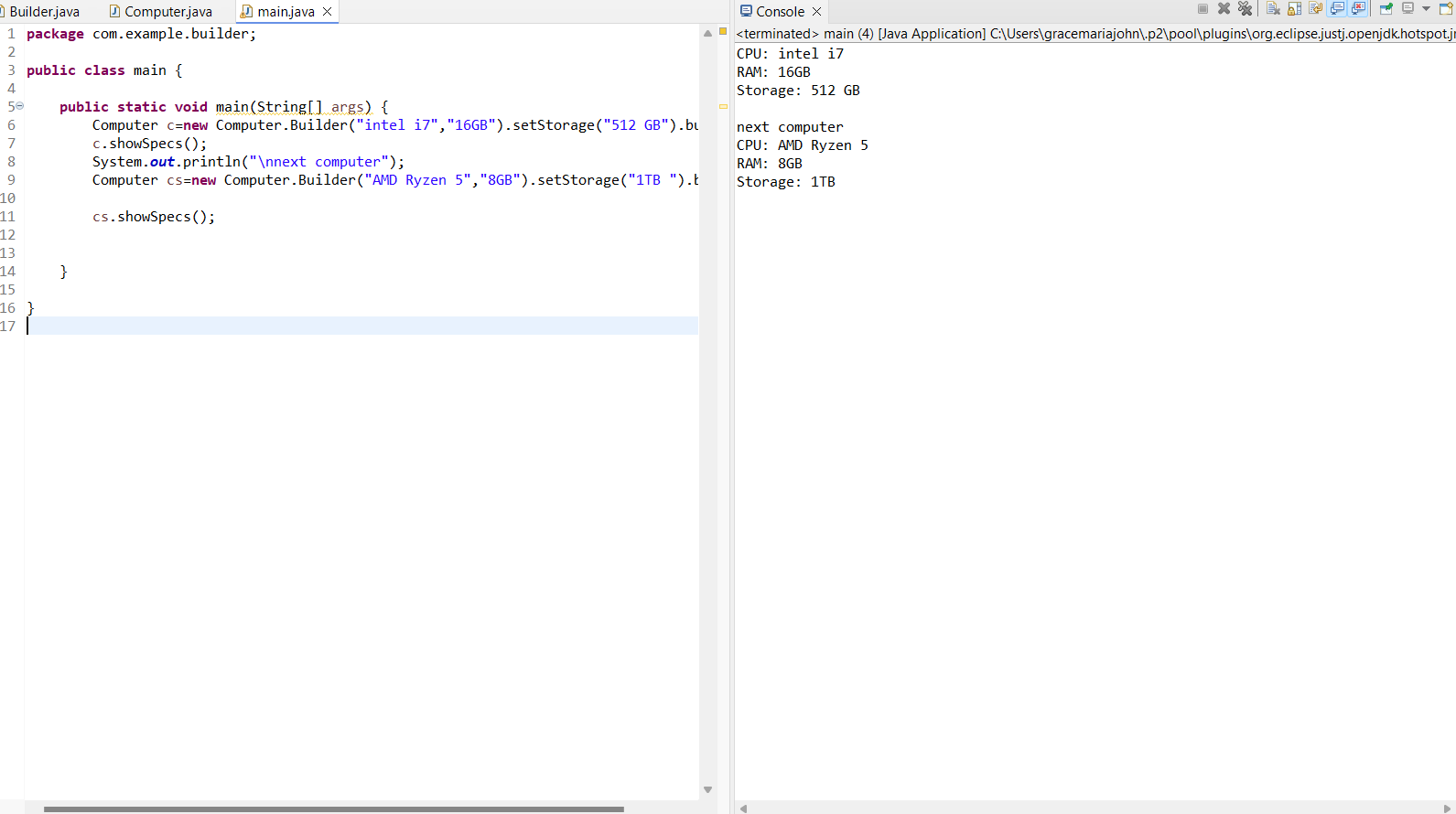
Computer cs=new Computer.Builder("AMD Ryzen 5","8GB").setStorage("1TB ").build();

cs.showSpecs();

}

}

This is my Output:



**Exercise 4: Implementing the Adapter Pattern**

This is my code:

package com.example.adapter;

public interface PaymentProcessor {

void processPayment(double amount);

}

package com.example.adapter;

public class Paypalgatway {

public void makePayment(double amount) {

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

}

}

package com.example.adapter;

public class Stripeadapter implements PaymentProcessor{

private Stripegatway pay;

public Stripeadapter(Stripegatway pay)

{

this.pay=pay;

}

@Override

public void processPayment(double amount)

{

pay.sendpayment(amount);

}

}

package com.example.adapter;

public class Main {

public static void main(String[] args) {

PaymentProcessor p=new Paypaladapter(new Paypalgatway());

p.processPayment(10000.0);

System.*out*.println();

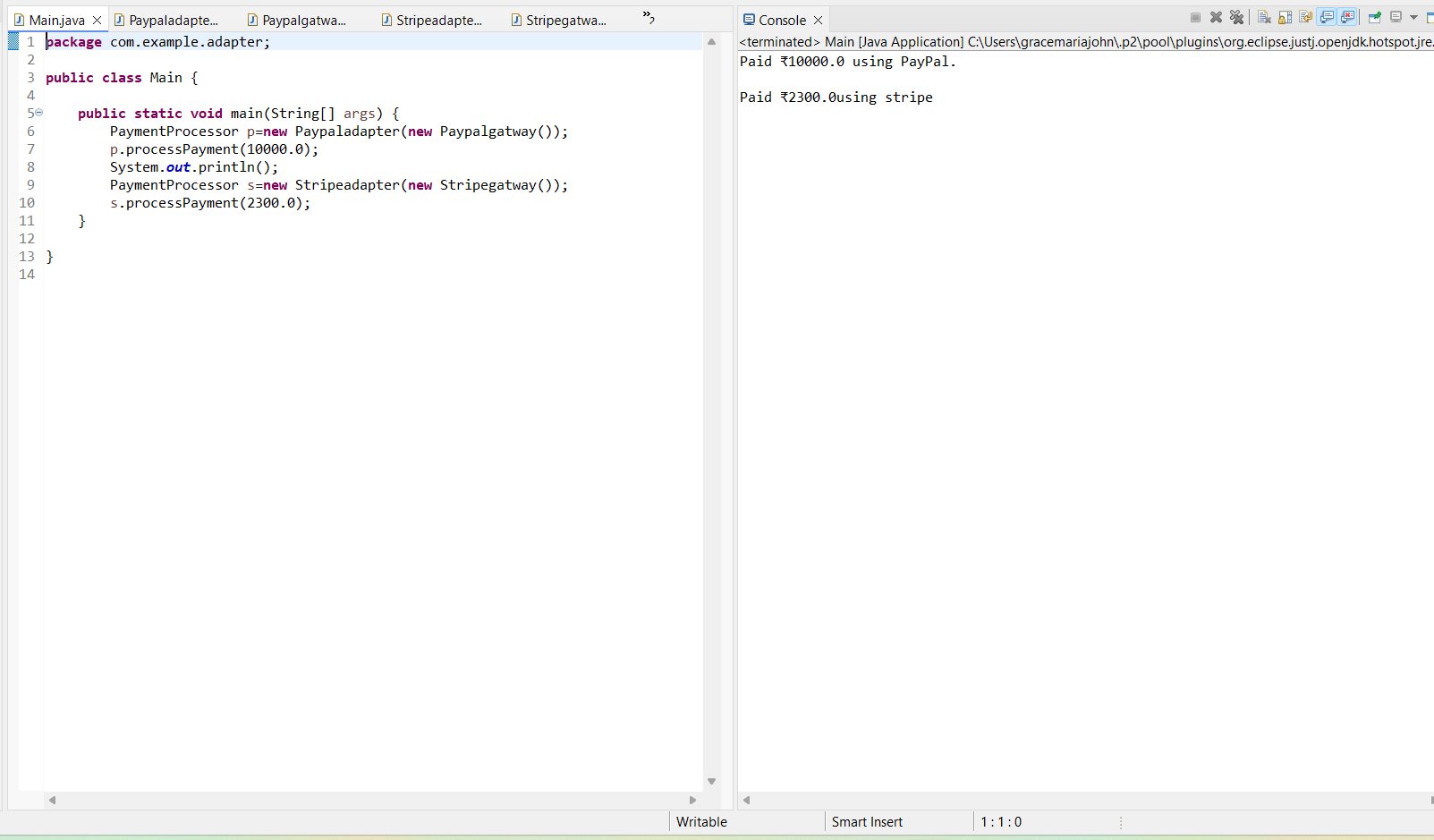
PaymentProcessor s=new Stripeadapter(new Stripegatway());

s.processPayment(2300.0);

}

}

This is my Output:



**Exercise 5: Implementing the Decorator Pattern**

This is my code:

package com.example.decorator;

public interface Notifier {

void send(String msg);

}

package com.example.decorator;

public class EmailNotifier implements Notifier {

@Override

public void send(String msg)

{

System.*out*.println("sending email:" +msg);

}

}

package com.example.decorator;

public class SlackNotifierDecorator extends NotifierDecorator{

public SlackNotifierDecorator(Notifier notifier)

{

super(notifier);

}

@Override

public void send(String msg)

{

super.send(msg);

System.*out*.println("sending slack:"+msg);

}

}

package com.example.decorator;

public class SMSNotifierDecorator extends NotifierDecorator{

public SMSNotifierDecorator(Notifier notifier)

{

super(notifier);

}

@Override

public void send(String msg) {

super.send(msg);

System.*out*.println("sending sms:"+msg);

}

}

package com.example.decorator;

public class main {

public static void main(String[] args) {

Notifier basicNotifier = new EmailNotifier(); // Only email

System.*out*.println("Basic Email Notification:");

basicNotifier.send("Hello Grace!");

System.*out*.println("\nEmail + SMS:");

Notifier smsNotifier = new SMSNotifierDecorator(new EmailNotifier());

smsNotifier.send("Welcome Grace!");

System.*out*.println("\nEmail + SMS + Slack:");

Notifier fullNotifier = new SlackNotifierDecorator(

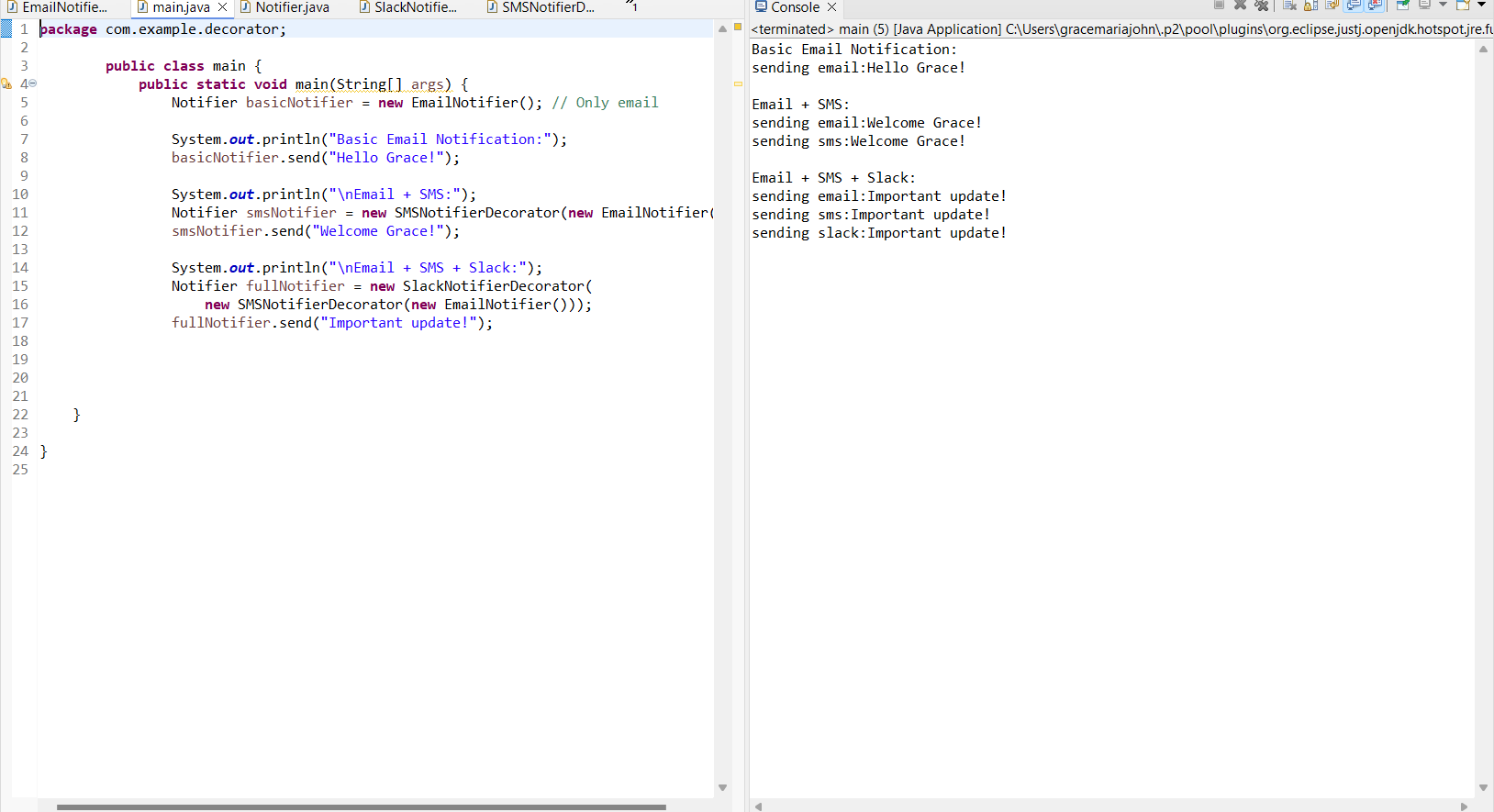
new SMSNotifierDecorator(new EmailNotifier()));

fullNotifier.send("Important update!");

}

}

This is my Output:



**Exercise 6: Implementing the Proxy Pattern**

This is my code:

package com.example.proxy;

public interface Image {

void display();

}

package com.example.proxy;

public class ProxyImage implements Image{

private RealImage realimage;

private String fname;

public ProxyImage(String fname)

{

this.fname=fname;

}

@Override

public void display()

{

if(realimage ==null)

{

realimage=new RealImage(fname);

}

realimage.display();

}

}

package com.example.proxy;

public class RealImage implements Image{

private String fname;

public RealImage(String fname)

{

this.fname=fname;

loadFromDisk();

}

private void loadFromDisk()

{

System.*out*.println("loading image from server");

}

@Override

public void display() {

System.*out*.println("Displaying image"+fname);

}

}

package com.example.proxy;

public class Main {

public static void main(String[] args) {

Image i1=new ProxyImage("img1.jpg");

Image i2=new ProxyImage("imag2.jpg");

System.*out*.println("first image access:");

i1.display();

System.*out*.println("accessing image");

i1.display();

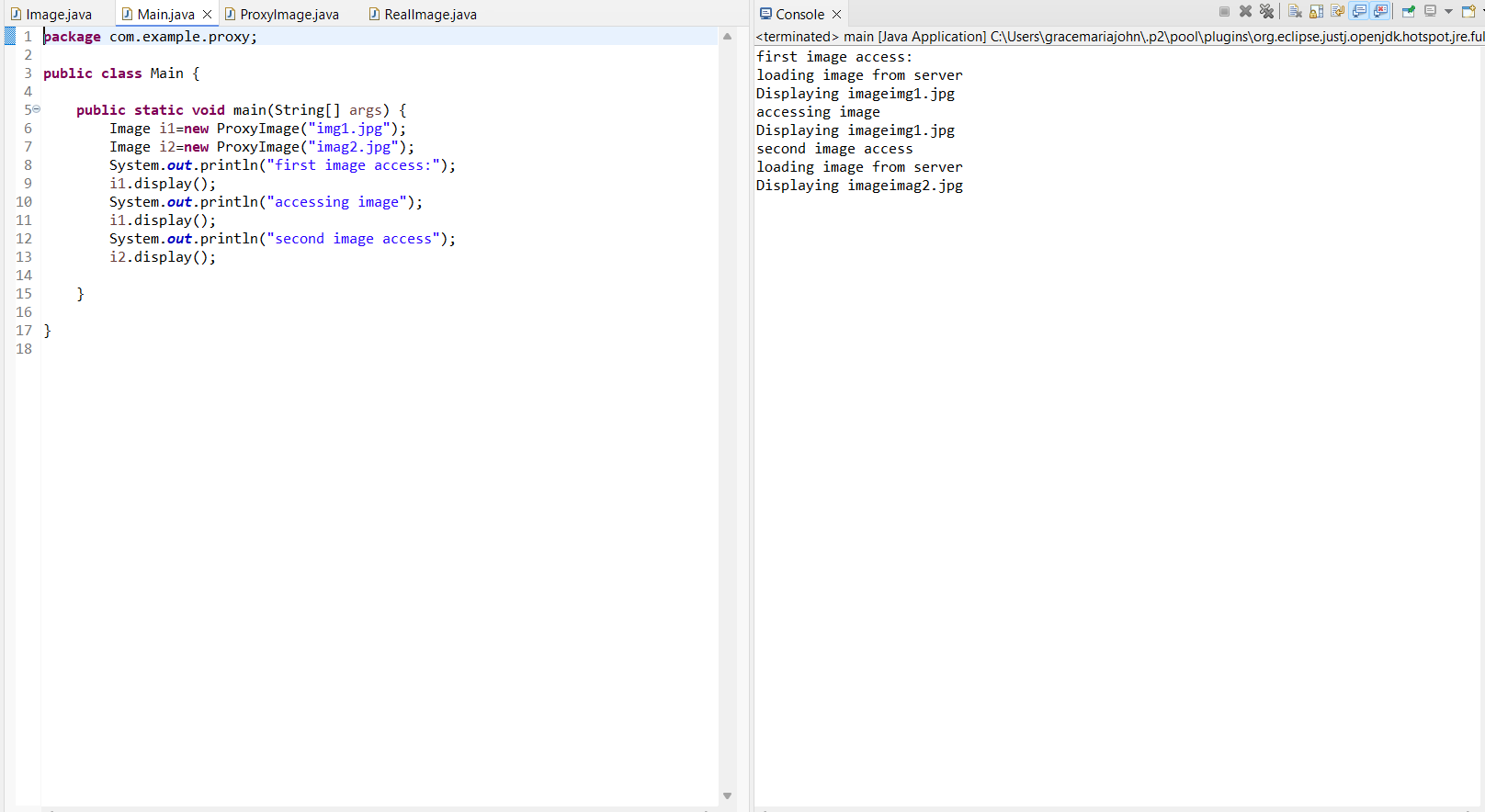
System.*out*.println("second image access");

i2.display();

}

}

This is my Output:



**Exercise 7: Implementing the Observer Pattern**

This is my code:

package com.example.observer;

public interface Observer {

void update(String stockname,double price);

}

package com.example.observer;

public interface Stock {

void register(Observer o);

void deregister (Observer o);

void notifyobserver();

}

package com.example.observer;

import java.util.ArrayList;

import java.util.List;

public class Stockmarket implements Stock{

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

private String stockname;

private double sprice;

public Stockmarket(String stockname,double iprice)

{

this.stockname=stockname;

this.sprice=iprice;

}

public void set(double nprice)

{

System.out.println("\nstock price updated for"+stockname+": ₹"+nprice);

this.sprice=nprice;

notifyobserver();

}

@Override

public void register(Observer o)

{

ob.add(o);

}

@Override

public void deregister(Observer o)

{

ob.remove(o);

}

@Override

public void notifyobserver()

{

for(Observer o:ob)

{

o.update(stockname, sprice);

}

}

} package com.example.observer;

public class Mobileapp implements Observer{

private String user;

public Mobileapp(String user)

{

this.user=user;

}

@Override

public void update(String stockname,double nprice)

{

System.*out*.println("MobileApp -"+user+ "alert :"+stockname+"is now ₹"+nprice);

}

}

package com.example.observer;

public class WebApp implements Observer{

private String user;

public WebApp(String user) {

this.user = user;

}

@Override

public void update(String stockname, double nprice) {

System.*out*.println(" WebApp - " + user + " Update: " + stockname + " is now ₹" + nprice);

}

}

package com.example.observer;

public class Main {

public static void main(String[]args)

{

Stockmarket reliance=new Stockmarket("Reliance",25000.0);

Observer u1=new Mobileapp("jiohotstar");

Observer u2=new WebApp("jiosaavn");

reliance.register(u1);

reliance.register(u2);

reliance.set(25002.0);

reliance.set(27000.0);

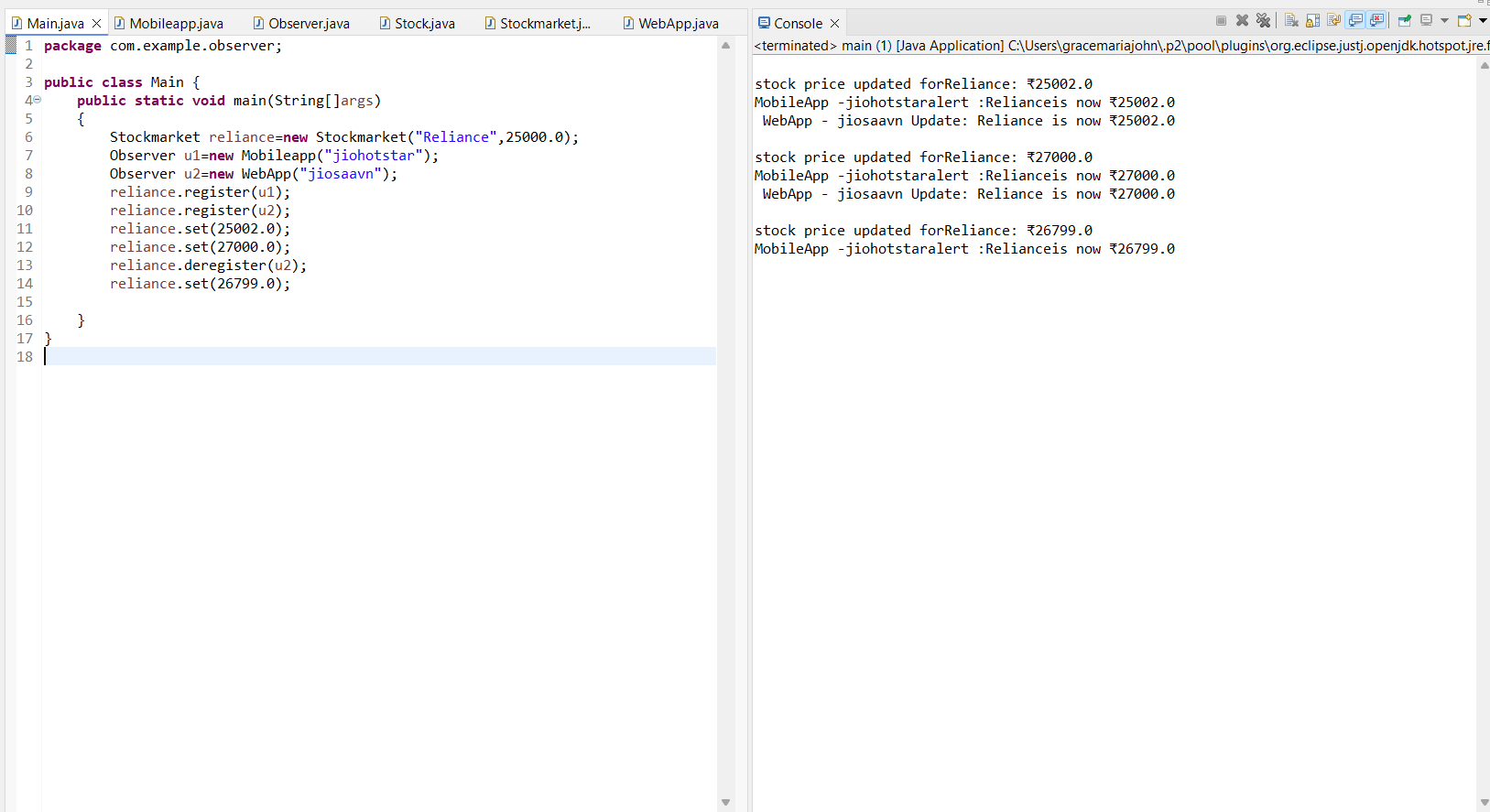
reliance.deregister(u2);

reliance.set(26799.0);

}

}

This is my Output:



**Exercise 8: Implementing the Strategy Pattern**

This is my code:

package com.example.command;

public interface Paymentstrategy {

void pay(double amount);

}

package com.example.command;

public class PaymentContext {

private Paymentstrategy strategy;

public void setPaymentStrategy(Paymentstrategy strategy) {

this.strategy = strategy;

}

public void makePayment(double amount) {

if (strategy == null) {

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

return;

}

strategy.pay(amount);

}

}

package com.example.command;

public class PayPalPayment implements Paymentstrategy {

private String email;

public PayPalPayment(String email) {

this.email = email;

}

@Override

public void pay(double amount) {

System.*out*.println("Paid ₹" + amount + " using PayPal account: " + email);

}

}

package com.example.command;

public class CreditCardPayment implements Paymentstrategy{

private String cardNumber;

public CreditCardPayment(String cardNumber) {

this.cardNumber = cardNumber;

}

@Override

public void pay(double amount) {

System.*out*.println("Paid ₹" + amount + " using Credit Card ending with " + cardNumber.substring(cardNumber.length() - 4));

}

}

package com.example.command;

public class Main {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

System.*out*.println("User chose Credit Card:");

context.setPaymentStrategy(new CreditCardPayment("1234567812345678"));

context.makePayment(1500.0);

System.*out*.println("\nUser chose PayPal:");

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

context.makePayment(2500.0);

System.*out*.println("\nTrying to pay without selecting a method:");

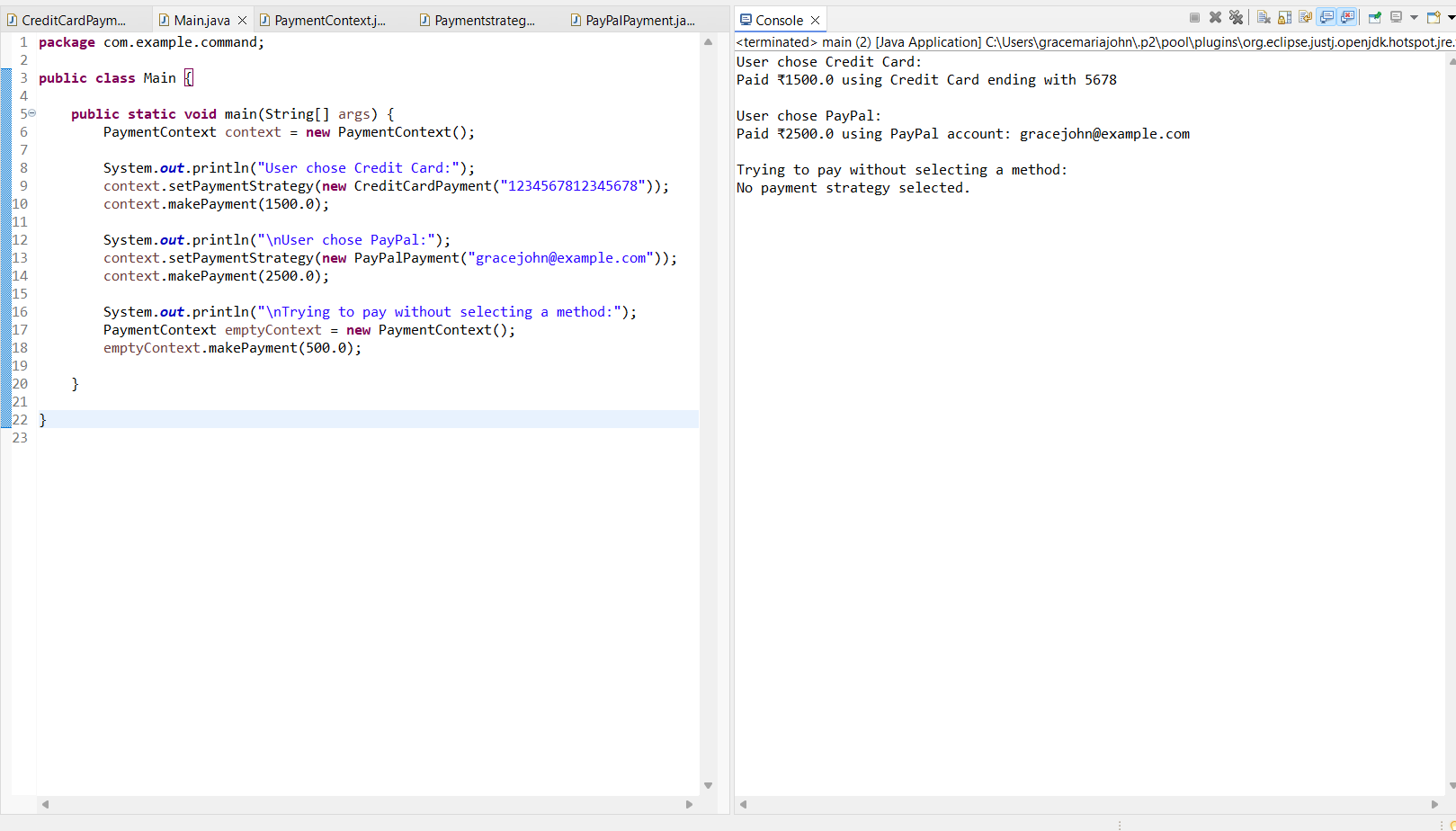
PaymentContext emptyContext = new PaymentContext();

emptyContext.makePayment(500.0);

}

}

This is my Output:



**Exercise 9: Implementing the Command Pattern**

This is my code:

package com.example.cpattern;

public class Light {

public void turnOn() {

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

}

public void turnOff() {

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

}

}

package com.example.cpattern;

public interface Command {

void execute();

}

package com.example.cpattern;

public class LightOnCommand implements Command{

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOn();

}

}

package com.example.cpattern;

public class LightOffCommand implements Command{

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOff();

}

}

package com.example.cpattern;

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();

System.*out*.println("Turning ON the light:");

remote.setCommand(lightOn);

remote.pressButton();

System.*out*.println("\nTurning OFF the light:");

remote.setCommand(lightOff);

remote.pressButton();

System.*out*.println("\nPressing button with no command:");

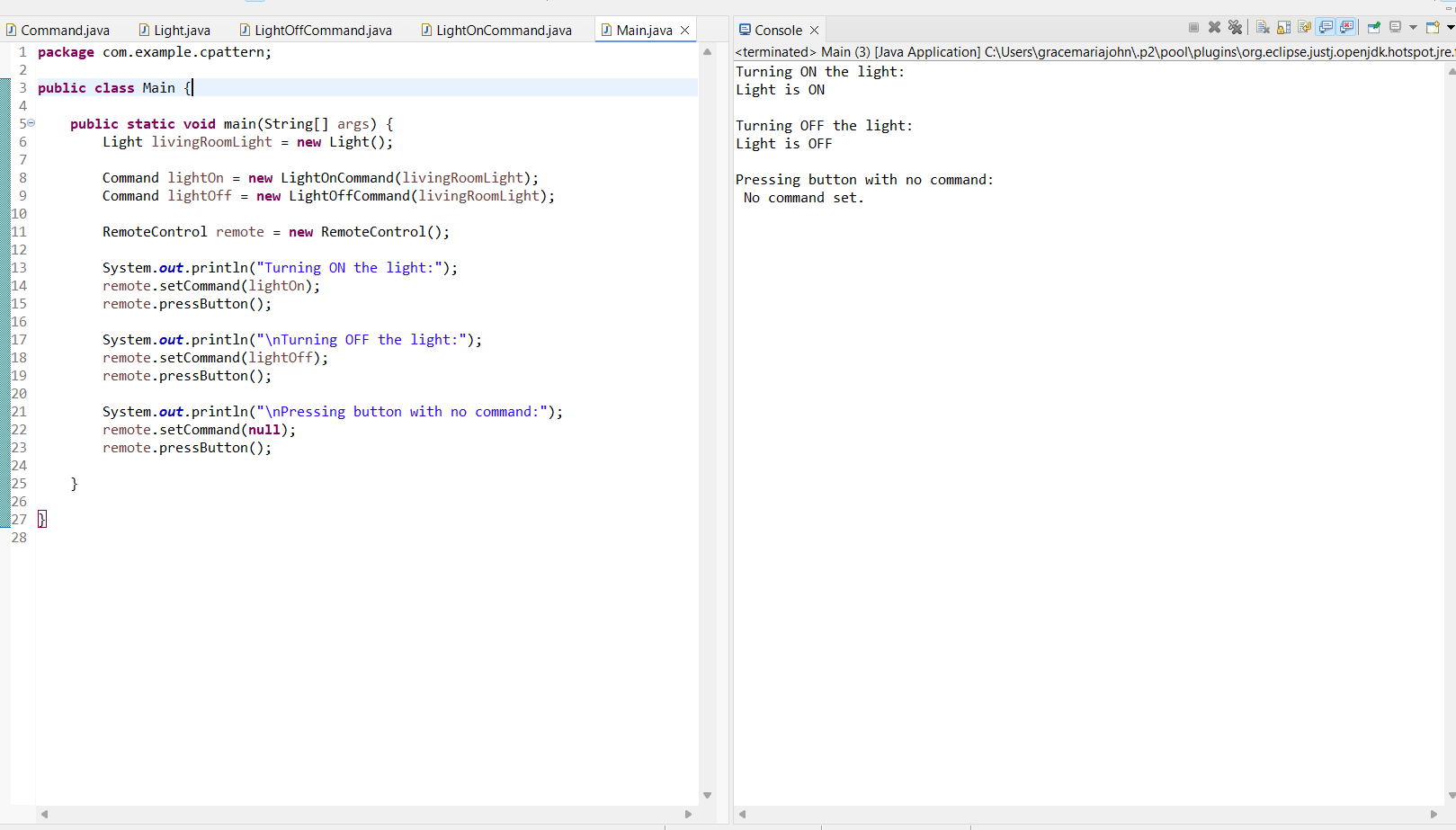
remote.setCommand(null);

remote.pressButton();

}

}

**This is my Output:**



**Exercise 10: Implementing the MVC Pattern**

**This is my code:**

package com.example.mvc;

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

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

}

package com.example.mvc;

public 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);

}

}

package com.example.mvc;

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 setStudentGrade(String grade) {

model.setGrade(grade);

}

public void updateView() {

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

}

}

package com.example.mvc;

public class Main {

public static void main(String[] args) {

Student student = new Student("Grace John", "S123", "C");

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

controller.updateView();

System.out.println("\nUpdating student info...");

controller.setStudentName("Grace J.");

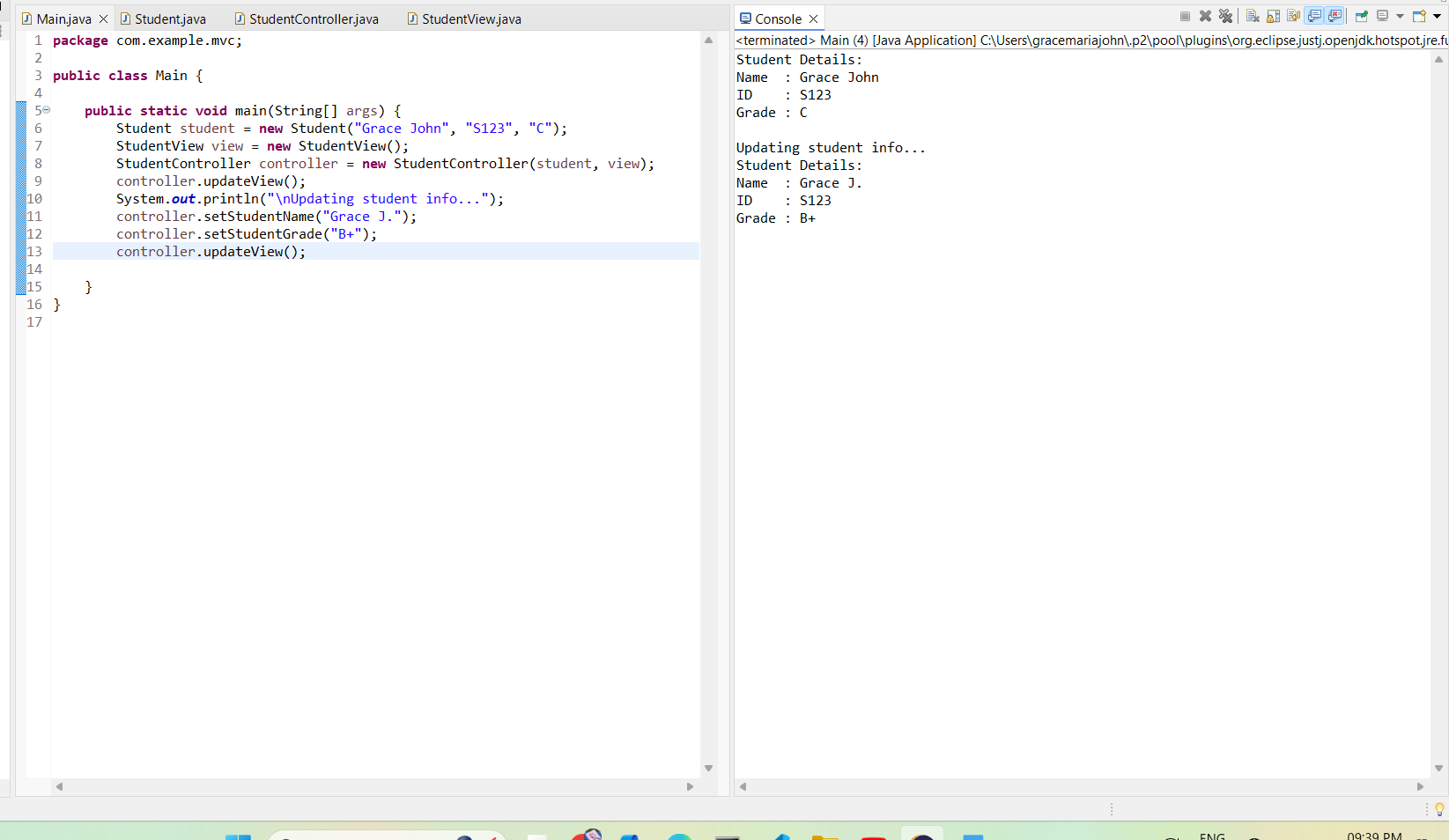
controller.setStudentGrade("B+");

controller.updateView();z

}

}

**This is my Output:**

****

**Exercise 11: Implementing Dependency Injection**

**This is my code:**

package com.example.dependency;

public interface CustomerRepository {

String findCustomerById(String id);

}

package com.example.dependency;

public class CustomerRepositoryImpl implements CustomerRepository{

@Override

public String findCustomerById(String id) {

return "Customer[" + id + "] = Grace John";

}

}

package com.example.dependency;

public class CustomerService {

private CustomerRepository repository;

public CustomerService(CustomerRepository repository) {

this.repository = repository;

} public void displayCustomer(String id) {

String customer = repository.findCustomerById(id);

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

}

}

package com.example.dependency;

public class Main {

public static void main(String[] args) {

CustomerRepository repo = new CustomerRepositoryImpl();

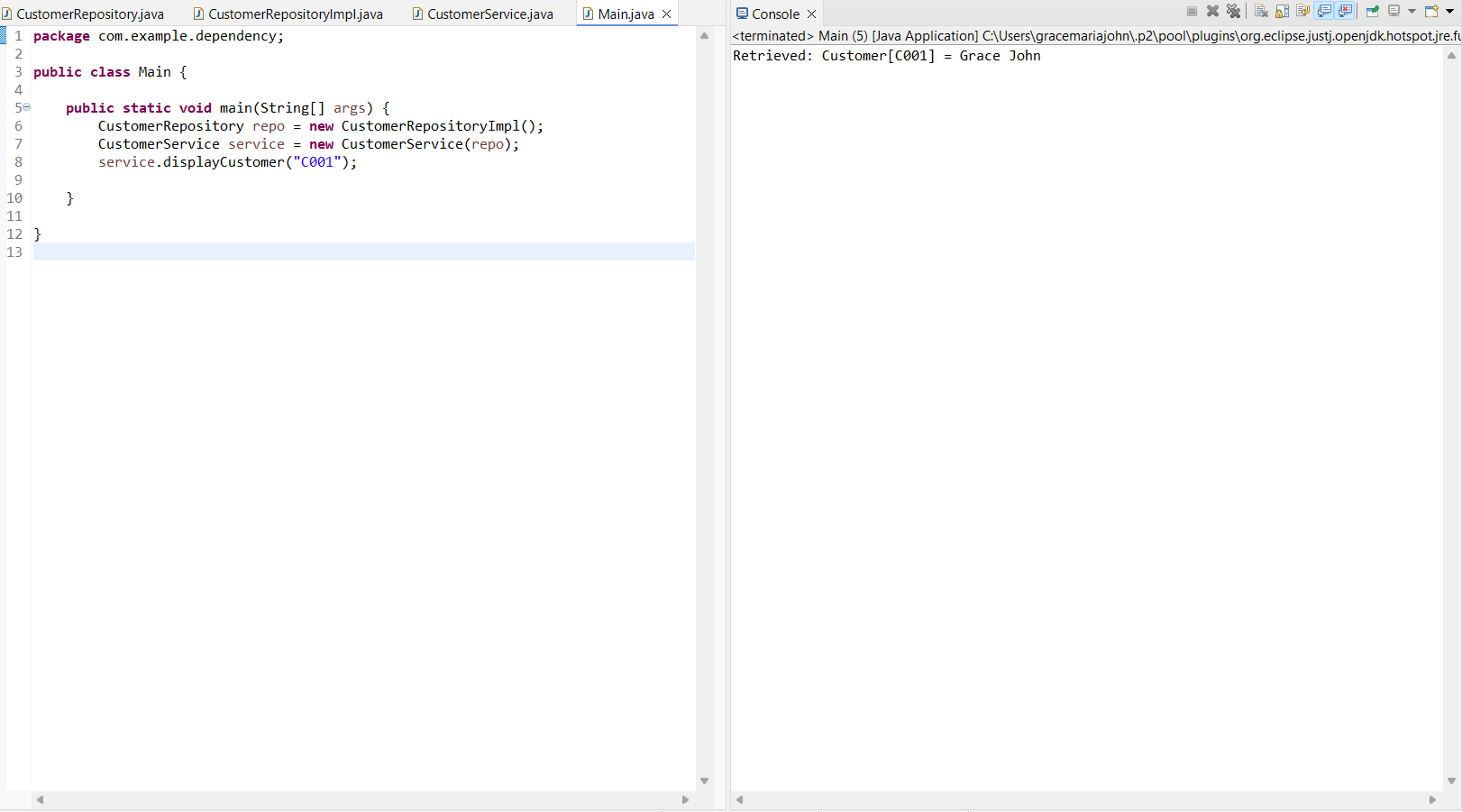
CustomerService service = new CustomerService(repo);

service.displayCustomer("C001");

}

}

**This is my Output:**

****