**Exercise 1: Implementing the Singleton Pattern**

**Code:**

**//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);

}

}

**//LoggerTest.java**

public class LoggerTest {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

Logger logger2 = Logger.getInstance();

logger1.log("This is the first log message.");

logger2.log("This is the second log message.");

if(logger1 == logger2) {

System.out.println("Both logger1 and logger2 are the same instance.");

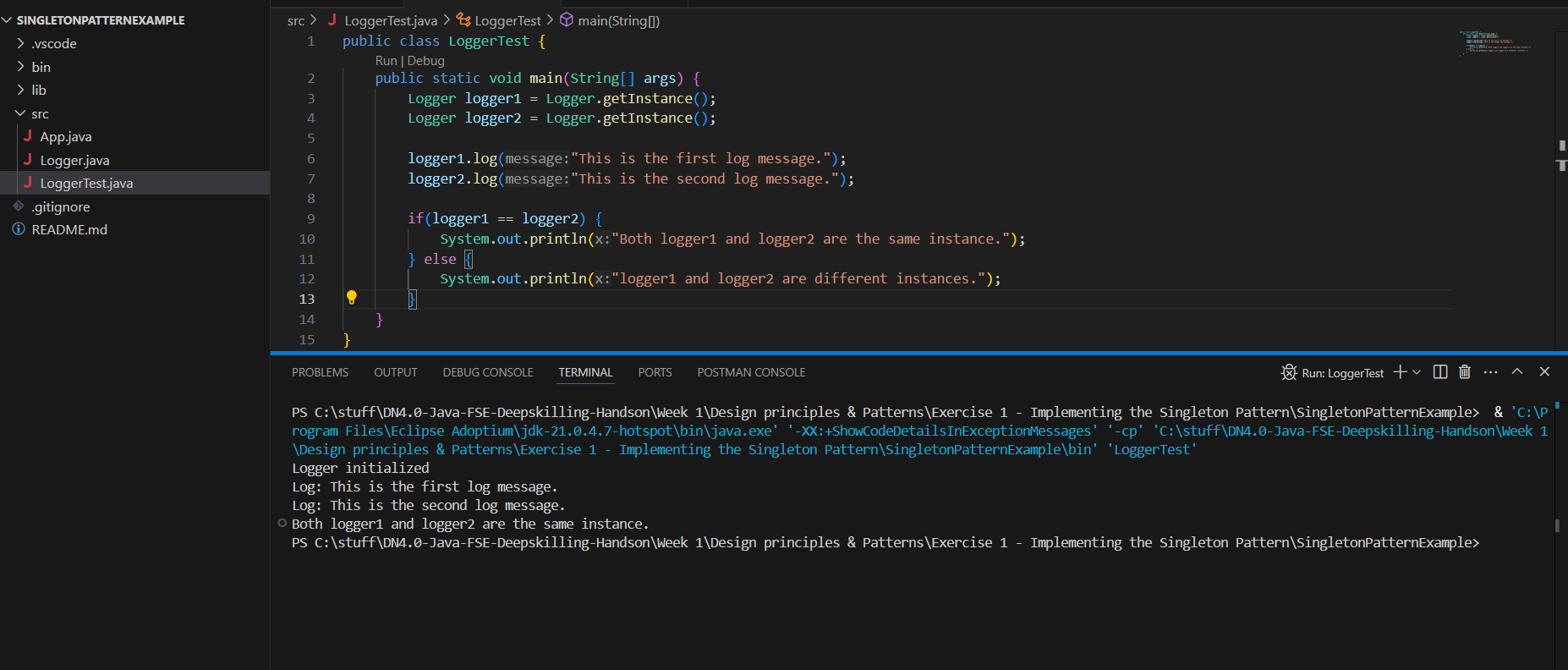
} else {

System.out.println("logger1 and logger2 are different instances.");

}

}

}

**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 a Word Document");

}

}

**// PdfDocument.java**

public class PdfDocument implements Document {

public void open() {

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

}

}

**//ExcelDocument.java**

public class ExcelDocument implements Document {

public void open() {

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

}

}

**//DocumentFactory.java**

public abstract class DocumentFactory {

public abstract Document createDocument();

}

**//WordDocumentFactory.java**

public class WordDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

**//PdfDocumentFactory.java**

public class PdfDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

**//ExcelDocumentFactory.java**

public class ExcelDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

**//App.java**

public class App {

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();

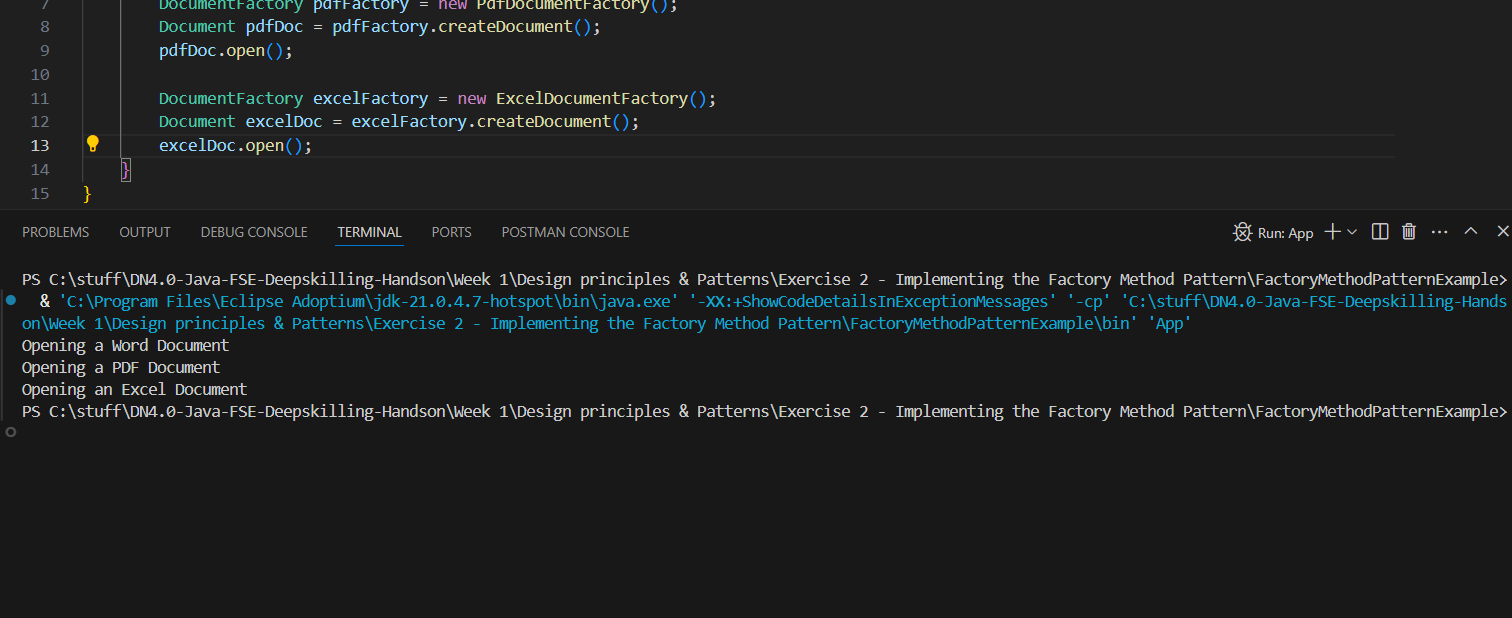
DocumentFactory excelFactory = new ExcelDocumentFactory();

Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

**Output:**

**Exercise 3: Implementing the Builder Pattern**

**Code:**

**//Computer.java**

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;

}

@Override

public String toString() {

return "Computer{" +

"CPU='" + CPU + '\'' +

", RAM='" + RAM + '\'' +

", storage='" + storage + '\'' +

'}';

}

public static class Builder{

private String CPU;

private String RAM;

private String storage;

public Builder(String CPU, String RAM, String storage) {

this.CPU = CPU;

this.RAM = RAM;

this.storage = 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() {

if (this.CPU == null || this.RAM == null || this.storage == null) {

throw new IllegalStateException("CPU, RAM, and Storage are required.");

}

return new Computer(this);

}

}

}

**//App.java**

public class App {

public static void main(String[] args) {

Computer basicComputer = new Computer.Builder("Intel i5", "8GB", "256GB SSD")

.build();

Computer gamingComputer = new Computer.Builder("AMD Ryzen 7", "16GB", "1TB SSD")

.setCPU("AMD Ryzen 9")

.setRAM("32GB")

.build();

Computer workstationComputer = new Computer.Builder("Intel Xeon", "64GB", "2TB SSD")

.setStorage("4TB SSD")

.build();

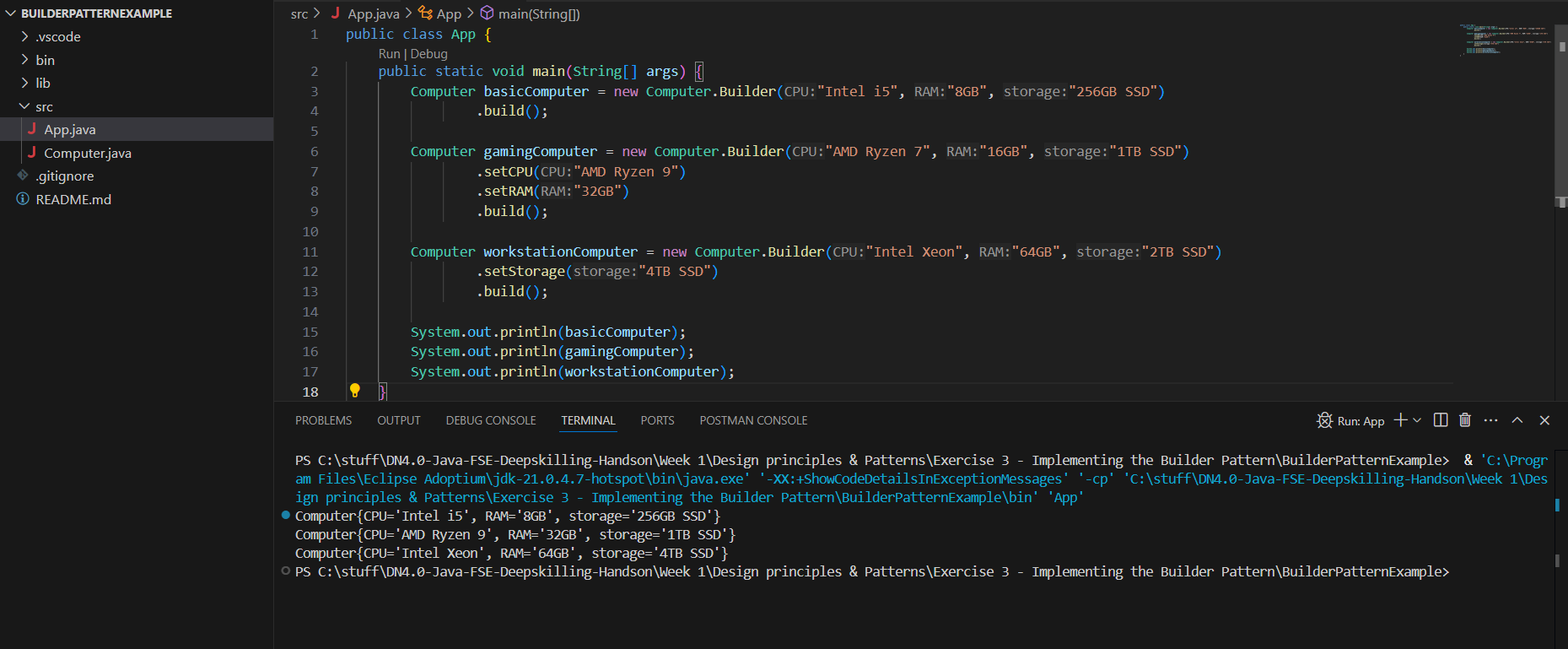
System.out.println(basicComputer);

System.out.println(gamingComputer);

System.out.println(workstationComputer);

}

}

**Output:**

**Exercise 4: Implementing the Adapter Pattern**

**Code:**

**//** **PaymentProcessor.java**

public interface PaymentProcessor {

void processPayment(double amount);

}

**//** **PayPalGateway.java**

public class PayPalGateway {

public void makePayment(double amount) {

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

}

}

**//** **RazorpayGateway.java**

public class RazorpayGateway {

public void makePayment(double amount) {

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

}

}

**//PayPalAdapter.java**

public class PayPalAdapter implements PaymentProcessor {

private PayPalGateway paypal;

public PayPalAdapter(PayPalGateway paypal) {

this.paypal = paypal;

}

@Override

public void processPayment(double amount) {

paypal.makePayment(amount);

}

}

**//RazorpayAdapter.java**

public class RazorpayAdapter implements PaymentProcessor {

private RazorpayGateway razorpayGateway;

public RazorpayAdapter(RazorpayGateway razorpayGateway) {

this.razorpayGateway = razorpayGateway;

}

@Override

public void processPayment(double amount) {

razorpayGateway.makePayment(amount);

}

}

**//App.java**

public class App {

public static void main(String[] args) throws Exception {

PayPalGateway paypal = new PayPalGateway();

PaymentProcessor paypalProcessor = new PayPalAdapter(paypal);

paypalProcessor.processPayment(150.00);

RazorpayGateway razorpay = new RazorpayGateway();

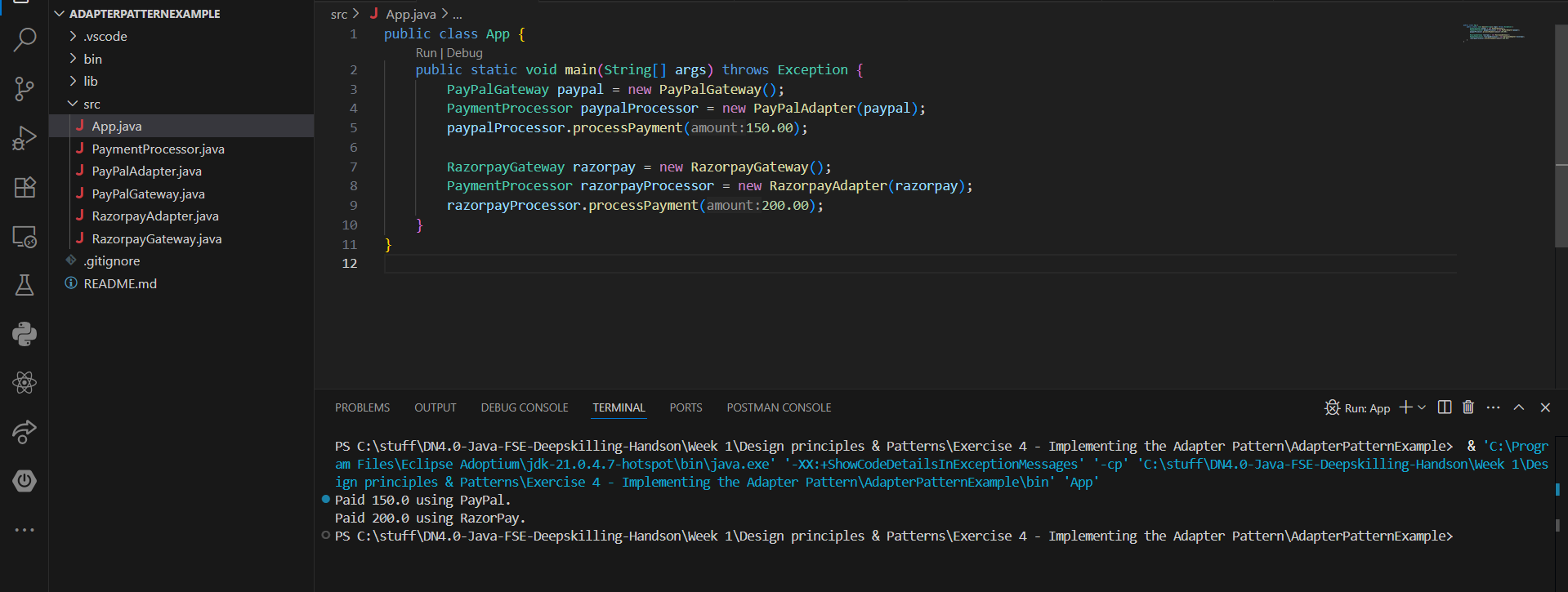
PaymentProcessor razorpayProcessor = new RazorpayAdapter(razorpay);

razorpayProcessor.processPayment(200.00);

}

}

**Output:**

****

**Exercise 5: Implementing the Decorator Pattern**

**Code:**

**//** **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 with message: " + message);

}

}

**//NotifierDecorator.java**

public abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

@Override

public void send(String message) {

notifier.send(message);

}

}

**//SMSNotifier.java**

public class SMSNotifier extends NotifierDecorator {

public SMSNotifier(Notifier notifier) {

super(notifier);

}

@Override

public void send(String message) {

super.send(message);

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

}

}

**//SlackNotifier.java**

public class SlackNotifier extends NotifierDecorator {

public SlackNotifier(Notifier notifier) {

super(notifier);

}

@Override

public void send(String message) {

super.send(message);

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

}

}

**//App.java**

public class App {

public static void main(String[] args) {

Notifier notifier = new EmailNotifier();

notifier = new SMSNotifier(notifier);

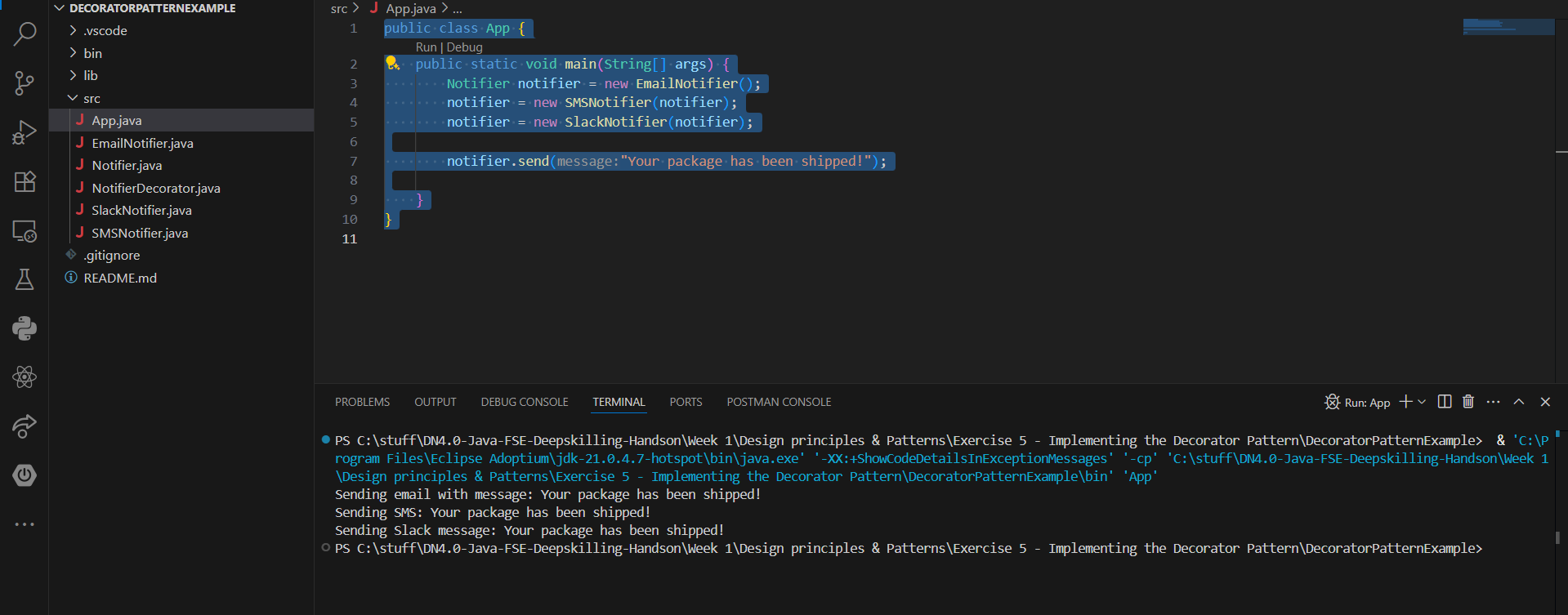
notifier = new SlackNotifier(notifier);

notifier.send("Your package has been shipped!");

}

}

**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;

loadImageFromRemoteServer();

}

private void loadImageFromRemoteServer() {

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

}

@Override

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;

}

@Override

public void display() {

if (realImage == null) {

realImage = new RealImage(fileName);

}

realImage.display();

}

}

**//App.java**

public class App {

public static void main(String[] args) {

Image image1 = new ProxyImage("image1.jpg");

Image image2 = new ProxyImage("image2.jpg");

image1.display();

System.out.println("");

image1.display();

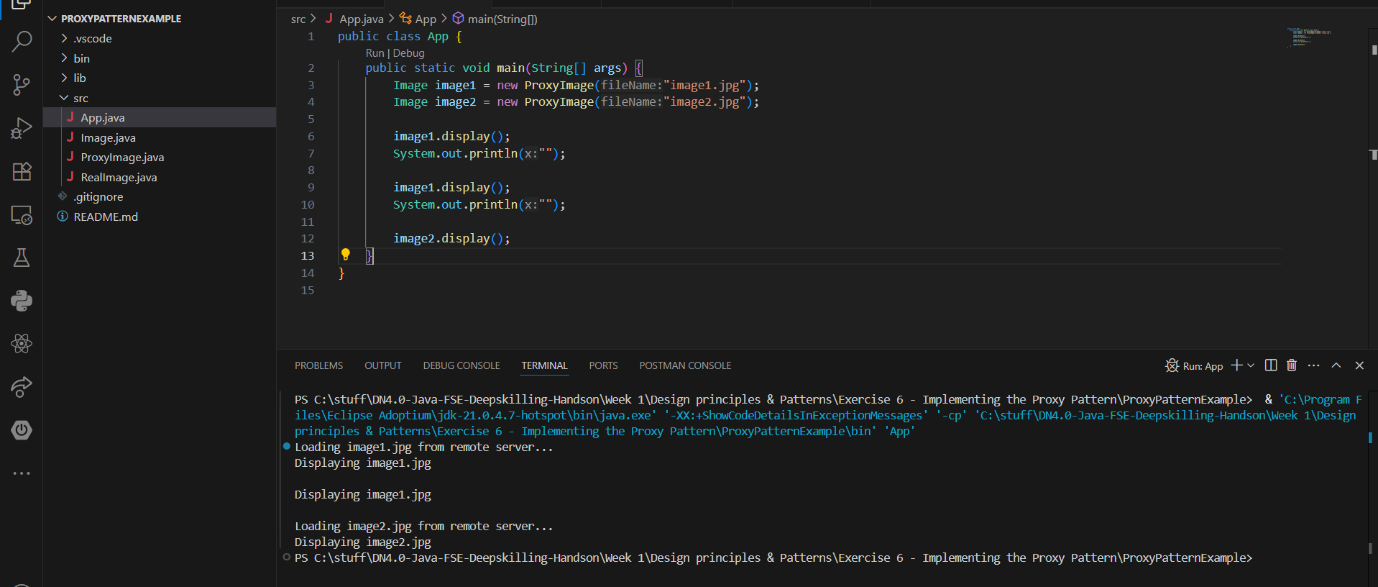
System.out.println("");

image2.display();

}

}

**Output:**



**Exercise 7: Implementing the Observer Pattern**

**Code:**

**//Observer.java**

public interface Observer {

void update(double price);

}

**//Stock.java**

public interface Stock {

void register(Observer observer);

void deregister(Observer observer);

void notifyObserver();

}

**//MobileApp.java**

public class MobileApp implements Observer {

private String name;

public MobileApp(String name) {

this.name = name;

}

@Override

public void update(double price) {

System.out.println("Mobile App " + name + " received stock price update: $" + price);

}

}

**// WebApp.java**

public class WebApp implements Observer {

private String name;

public WebApp(String name) {

this.name = name;

}

@Override

public void update(double price) {

System.out.println("Web App " + name + " received stock price update: $" + price);

}

}

**//StockMarket.java**

import java.util.\*;

public class StockMarket implements Stock {

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

private double price;

@Override

public void register(Observer observer) {

observers.add(observer);

}

@Override

public void deregister(Observer observer) {

observers.remove(observer);

}

@Override

public void notifyObserver() {

for (Observer observer : observers) {

observer.update(price);

}

}

public void setPrice(double price) {

this.price = price;

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

notifyObserver();

}

}

**//App.java**

public class App {

public static void main(String[] args){

StockMarket stockMarket = new StockMarket();

Observer mobileApp = new MobileApp("MobileApp1");

Observer webApp = new WebApp("WebApp1");

stockMarket.register(mobileApp);

stockMarket.register(webApp);

stockMarket.setPrice(100.0);

stockMarket.setPrice(105.5);

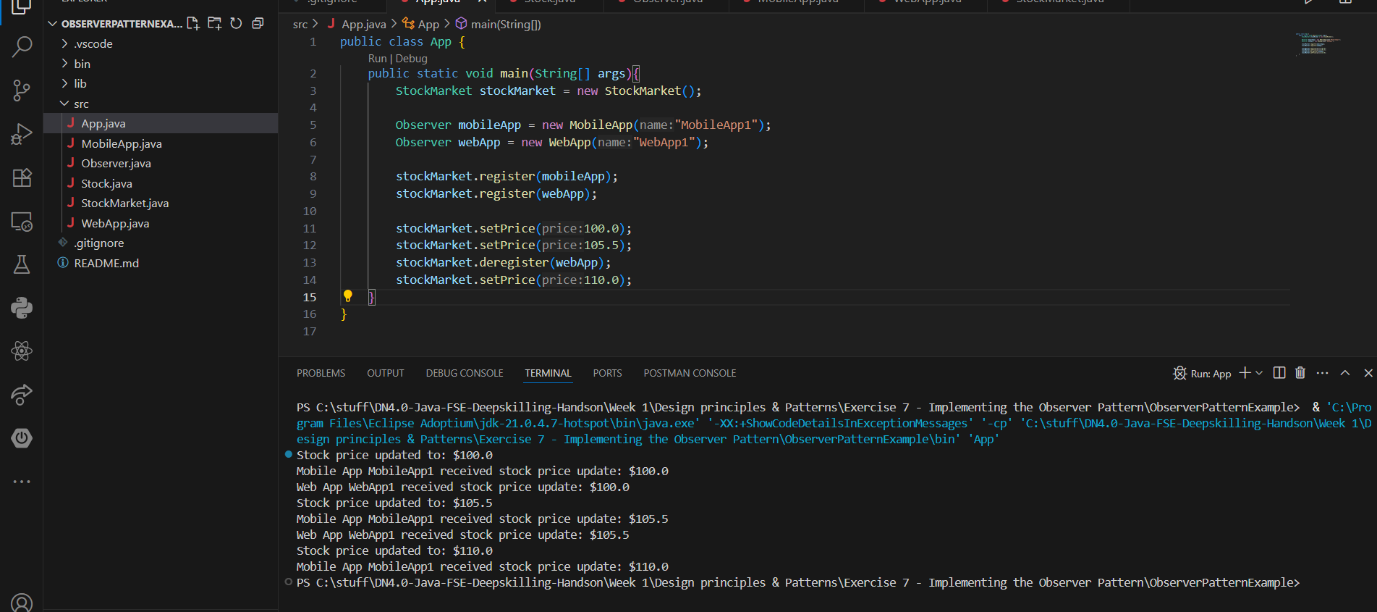
stockMarket.deregister(webApp);

stockMarket.setPrice(110.0);

}

}

**Output:**



**Exercise 8: Implementing the Strategy Pattern**

**Code:**

**//PaymentStrategy.java**

public interface PaymentStrategy {

void pay(double amount);

}

**//CreditCardPayment.java**

public class CreditCardPayment implements PaymentStrategy {

private String cardNo;

public CreditCardPayment(String cardNo) {

this.cardNo = cardNo;

}

@Override

public void pay(double amount) {

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

}

}

**//PayPalPayment.java**

public class PayPalPayment implements PaymentStrategy {

private String id;

public PayPalPayment(String id) {

this.id = id;

}

@Override

public void pay(double amount) {

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

}

}

**//PaymentContext.java**

public class PaymentContext {

private PaymentStrategy paymentStrategy;

public void setPaymentStrategy(PaymentStrategy paymentStrategy) {

this.paymentStrategy = paymentStrategy;

}

public void pay(double amount) {

if (paymentStrategy == null) {

System.out.println("Payment strategy not set");

}

else

paymentStrategy.pay(amount);

}

}

**//App.java**

public class App {

public static void main(String[] args) {

PaymentContext paymentContext = new PaymentContext();

paymentContext.setPaymentStrategy(new PayPalPayment("31231saff"));

paymentContext.pay(100.0);

paymentContext.setPaymentStrategy(new CreditCardPayment("1234-5678-9876-5432"));

paymentContext.pay(200.0);

}

}