**Exercise 1: Singleton Pattern**

**Code:**

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);

}

}

public class Main {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

Logger logger2 = Logger.getInstance();

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

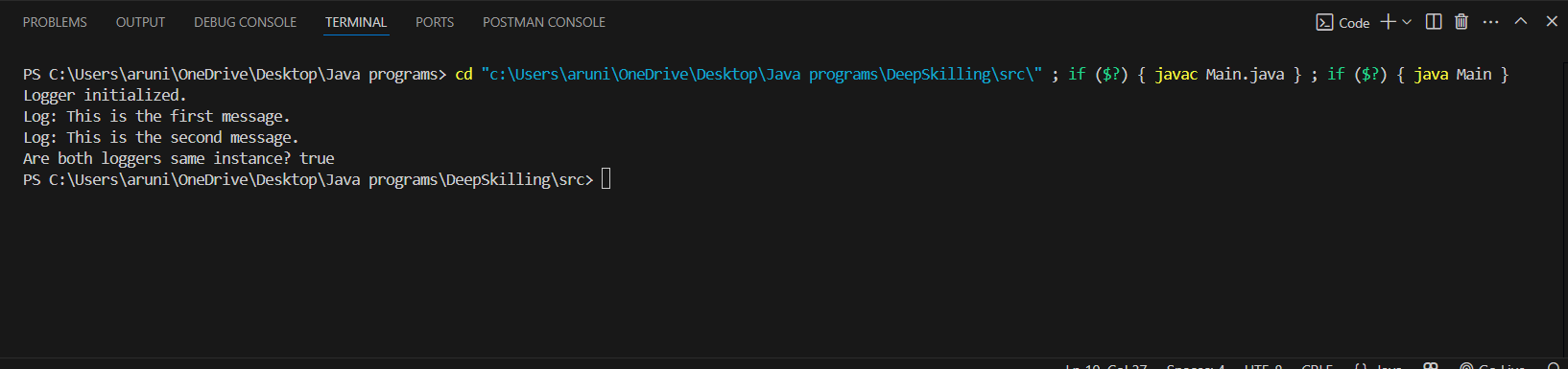
logger2.log("This is the second message.");

System.out.println("Are both loggers same instance? " + (logger1 == logger2));

}

}

**Output:**

****

**Exercise 2: Factory Method Pattern**

**Code:**

**Document.java**

package Factory;

public interface Document {

void open();

}

**WordDocument.java**

package Factory;

public class WordDocument implements Document {

public void open() {

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

}

}

**PdfDocument.java**

package Factory;

public class PdfDocument implements Document {

public void open() {

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

}

}

**ExcelDocument.java**

package Factory;

public class ExcelDocument implements Document {

public void open() {

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

}

}

**DocumentFactory.java**

package Factory;

public abstract class DocumentFactory {

public abstract Document createDocument();

}

**WordFactory.java**

package Factory;

public class WordFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

**PdfFactory.java**

package Factory;

public class PdfFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

**ExcelFactory.java**

package Factory;

public class ExcelFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

**FactoryMain.java**

package Factory;

public class FactoryMain {

public static void main(String[] args) {

DocumentFactory wordFactory = new WordFactory();

Document word = wordFactory.createDocument();

word.open();

DocumentFactory pdfFactory = new PdfFactory();

Document pdf = pdfFactory.createDocument();

pdf.open();

DocumentFactory excelFactory = new ExcelFactory();

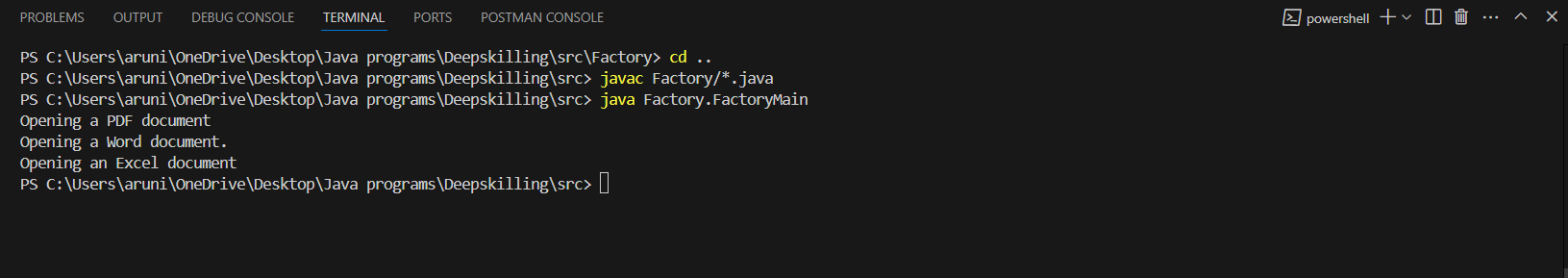
Document excel = excelFactory.createDocument();

excel.open();

}

}

**Output:**

****

**Exercise 3: Builder Pattern**

**Code:**

**Computer.java**

package builder;

public class Computer {

private String CPU;

private String RAM;

private String storage;

private String GPU;

private Computer(Builder builder) {

this.CPU = builder.CPU;

this.RAM = builder.RAM;

this.storage = builder.storage;

this.GPU = builder.GPU;

}

public void specs() {

System.out.println("Computer Specs:");

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

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

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

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

}

// Static nested Builder class

public static class Builder {

private String CPU;

private String RAM;

private String storage;

private String GPU;

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 setGPU(String GPU) {

this.GPU = GPU;

return this;

}

public Computer build() {

return new Computer(this);

}

}

}

**BuilderMain.java**

package builder;

public class BuilderMain {

public static void main(String[] args) {

Computer gamingRig = new Computer.Builder()

.setCPU("Intel i9")

.setRAM("32GB")

.setStorage("2TB SSD")

.setGPU("NVIDIA RTX 4090")

.build();

Computer budgetBuild = new Computer.Builder()

.setCPU("Intel i3")

.setRAM("8GB")

.setStorage("512GB HDD")

.build();

System.out.println("Gaming Rig:");

gamingRig.specs();

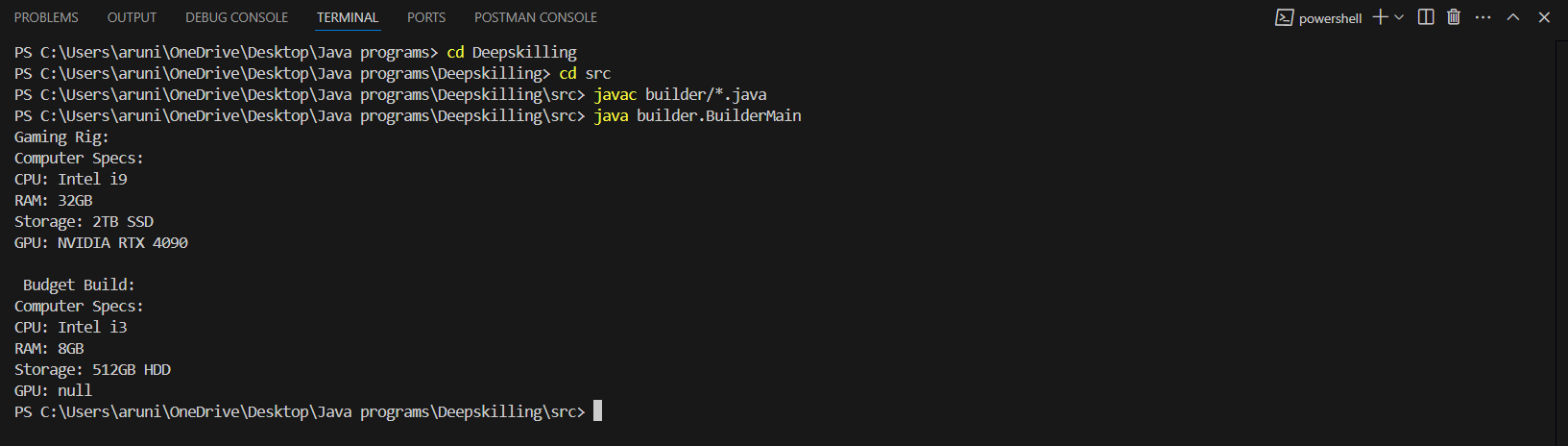
System.out.println("\n Budget Build:");

budgetBuild.specs();

}

}

**Output:**

****

**Exercise 4: Builder Pattern**

**Code:**

**PaymentProcessor.java**

package adapter;

public interface PaymentProcessor {

void processPayment(double amount);

}

**PayPal.java**

package adapter;

public class PayPal {

public void sendMoney(double amount) {

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

}

}

**Stripe.java**

package adapter;

public class Stripe {

public void makePayment(double amount) {

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

}

}

**PayPalAdapter.java**

package adapter;

public class PayPalAdapter implements PaymentProcessor {

private PayPal paypal;

public PayPalAdapter(PayPal paypal) {

this.paypal = paypal;

}

public void processPayment(double amount) {

paypal.sendMoney(amount);

}

}

**StripeAdapter.java**

package adapter;

public class StripeAdapter implements PaymentProcessor {

private Stripe stripe;

public StripeAdapter(Stripe stripe) {

this.stripe = stripe;

}

public void processPayment(double amount) {

stripe.makePayment(amount);

}

}

**AdapterMain.java**

package adapter;

public class AdapterMain {

public static void main(String[] args) {

PaymentProcessor paypalProcessor = new PayPalAdapter(new PayPal());

PaymentProcessor stripeProcessor = new StripeAdapter(new Stripe());

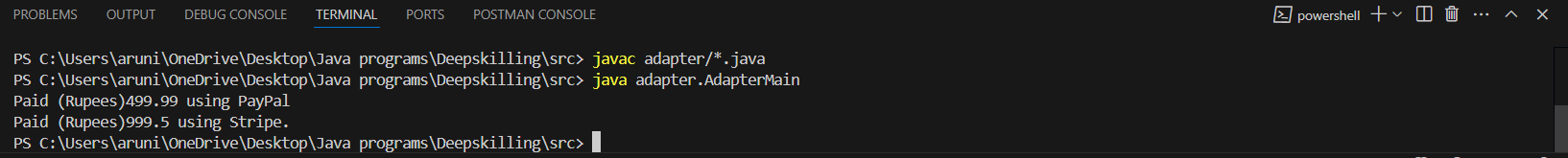
paypalProcessor.processPayment(499.99);

stripeProcessor.processPayment(999.50);

}

}

**Output:**

****

**Exercise 5: Decorator Pattern**

**Code:**

**Notifier.java**

package decorator;

public interface Notifier {

void send(String message);

}

**EmailNotifier.java**

package decorator;

public class EmailNotifier implements Notifier {

public void send(String message) {

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

}

}

**NotifierDecorator.java**

package decorator;

public abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

public void send(String message) {

notifier.send(message);

}

}

**SMSNotifier.java**

package decorator;

public class SMSNotifier extends NotifierDecorator {

public SMSNotifier(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

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

}

}

**SlackNotifier.java**

package decorator;

public class SlackNotifier extends NotifierDecorator {

public SlackNotifier(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

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

}

}

**DecoratorMain.java**

package decorator;

public class DecoratorMain {

public static void main(String[] args) {

Notifier baseNotifier = new EmailNotifier();

// Add SMS and Slack functionality

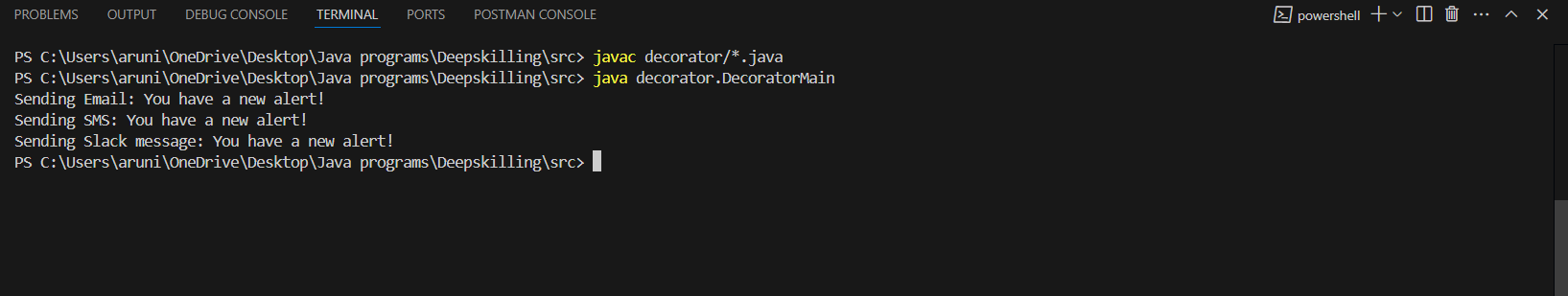
Notifier fullNotifier = new SlackNotifier(new SMSNotifier(baseNotifier));

fullNotifier.send("You have a new alert!");

}

}

**Output:**

****

**Exercise 6: Proxy Pattern**

**Code:**

**Image.java**

package proxy;

public interface Image {

void display();

}

**RealImage.java**

package proxy;

public class RealImage implements Image {

private String filename;

public RealImage(String filename) {

this.filename = filename;

loadFromDisk();

}

private void loadFromDisk() {

System.out.println("Loading image from disk: " + filename);

}

public void display() {

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

}

}

**ProxyImage.java**

package proxy;

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); // Lazy loading

}

realImage.display();

}

}

**ProxyMain.java**

package proxy;

public class ProxyMain {

public static void main(String[] args) {

Image image1 = new ProxyImage("picture.png");

// image not loaded yet

System.out.println("Image created but not displayed yet...");

// now viewing image

image1.display();

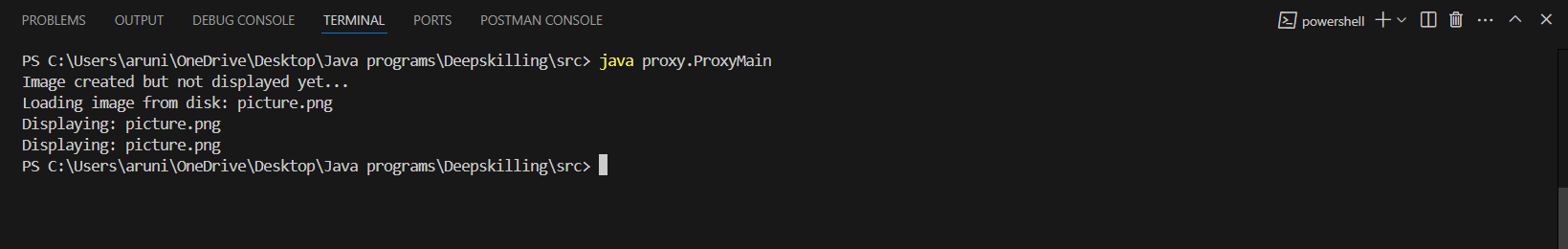
// viewing again, no loading this time

image1.display();

}

}

**Output:**

****

**Exercise 7: Observer Pattern**

**Code:**

**Observer.java**

package observer;

public interface Observer {

void update(String stockName, double newPrice);

}

**Stock.java**

package observer;

public interface Stock {

void registerObserver(Observer o);

void removeObserver(Observer o);

void notifyObservers();

}

**StockMarket.java**

package observer;

import java.util.ArrayList;

import java.util.List;

public class StockMarket implements Stock {

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

private String stockName;

private double stockPrice;

public StockMarket(String stockName, double initialPrice) {

this.stockName = stockName;

this.stockPrice = initialPrice;

}

public void setPrice(double newPrice) {

System.out.println("\nStock price updated to (Rupees)" + newPrice);

this.stockPrice = newPrice;

notifyObservers();

}

public void registerObserver(Observer o) {

observers.add(o);

}

public void removeObserver(Observer o) {

observers.remove(o);

}

public void notifyObservers() {

for (Observer o : observers) {

o.update(stockName, stockPrice);

}

}

}

**MobileApp.java**

package observer;

public class MobileApp implements Observer {

public void update(String stockName, double newPrice) {

System.out.println("Mobile App - " + stockName + ": (Rupees)" + newPrice);

}

}

**WebApp.java**

package observer;

public class WebApp implements Observer {

public void update(String stockName, double newPrice) {

System.out.println("Web App - " + stockName + ": (Rupees)" + newPrice);

}

}

**ObserverMain.java**

package observer;

public class ObserverMain {

public static void main(String[] args) {

StockMarket companyStock = new StockMarket("Company", 3500);

Observer mobile = new MobileApp();

Observer web = new WebApp();

companyStock.registerObserver(mobile);

companyStock.registerObserver(web);

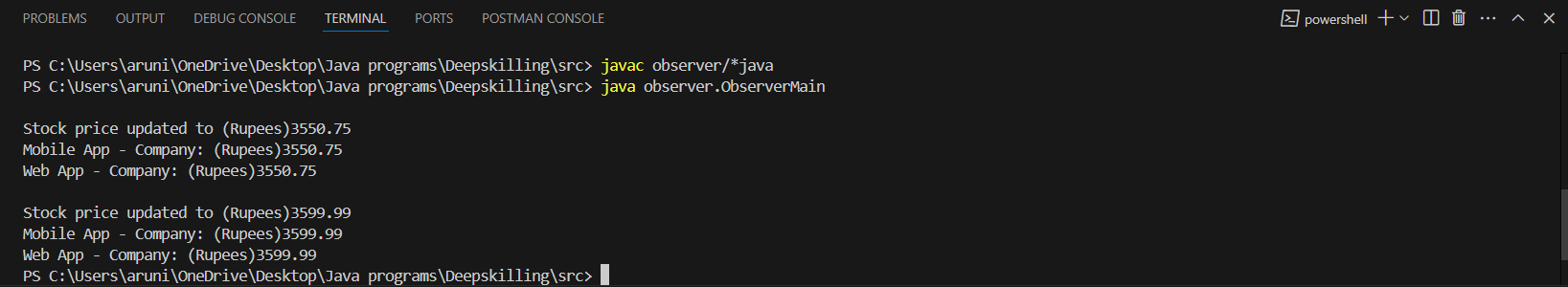
companyStock.setPrice(3550.75);

companyStock.setPrice(3599.99);

}

}

**Output:**

****