**Exercise 1: Implementing the Singleton Pattern**

**Logger.java**

package singleton;

public class Logger {

// Private static instance of Logger

private static Logger instance;

// Private constructor to prevent instantiation

private Logger() {

System.out.println("Logger instance created.");

}

// Public method to provide access to the instance

public static Logger getInstance() {

if (instance == null) {

instance = new Logger(); // Lazy initialization

}

return instance;

}

// Method for logging messages

public void log(String message) {

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

}

}

**TestLogger.java**

package singleton;

public class TestLogger {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

Logger logger2 = Logger.getInstance();

logger1.log("First message");

logger2.log("Second message");

if (logger1 == logger2) {

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

} else {

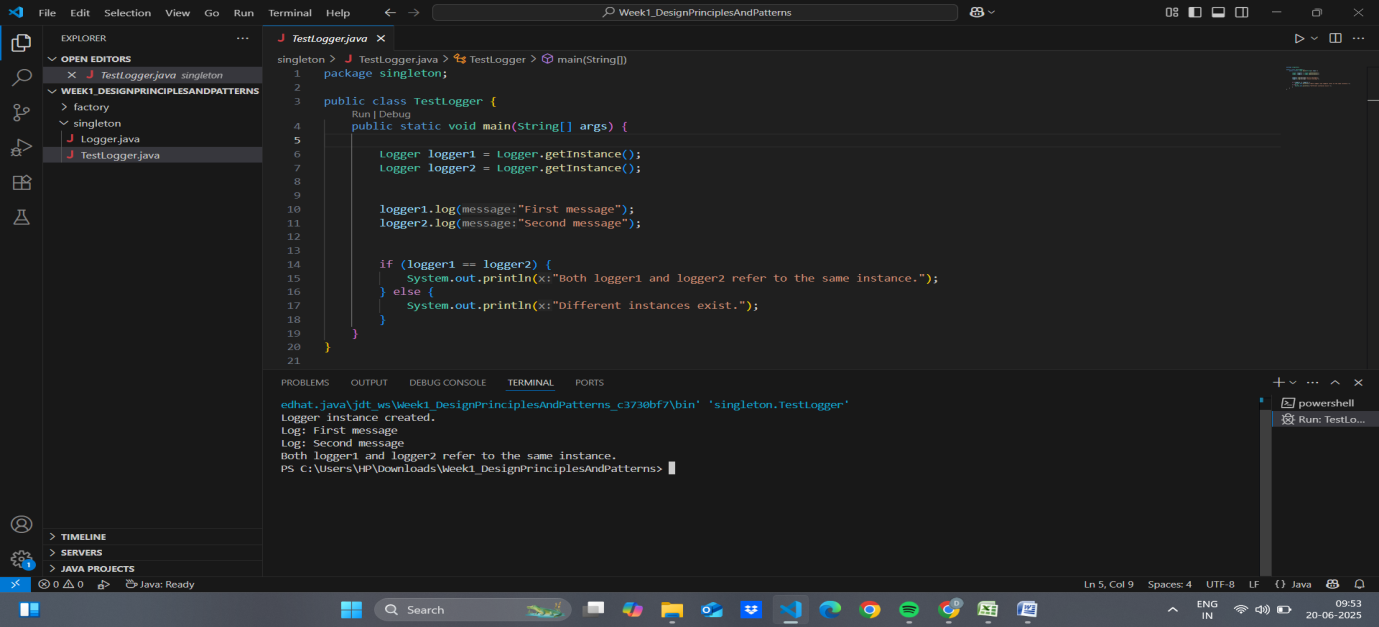
System.out.println("Different instances exist.");

}

}

}

**Output:**

****

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

**Document.java**

package factory;

public interface Document {

void open();

}

**DocumentFactory.java**

package factory;

public abstract class DocumentFactory {

public abstract Document createDocument();

}

**ExcelDocument.java**

package factory;

public class ExcelDocument implements Document {

public void open() {

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

}

}

**ExcelDocumentFactory.java**

package factory;

public class ExcelDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

**pdfDocument.java**

package factory;

public class PdfDocument implements Document {

public void open() {

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

}

}

**pdfDocumentFactory.java**

package factory;

public class PdfDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

**TestFactoryMethod.java**

package factory;

public class TestFactoryMethod {

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();

}

}

**WordDocument.java**

package factory;

public class WordDocument implements Document {

public void open() {

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

}

}

**WordDocumentfactory.java**

package factory;

public class WordDocumentFactory extends DocumentFactory {

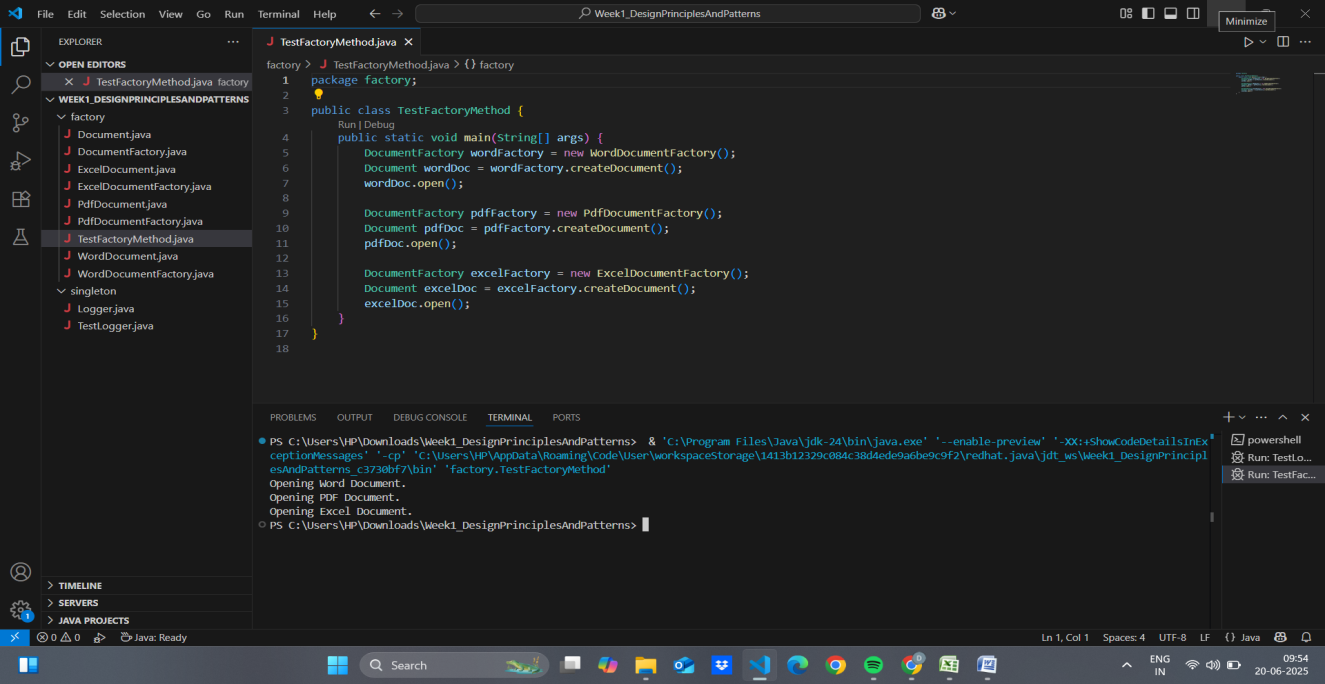
public Document createDocument() {

return new WordDocument();

}

}

**Output:**

****

**Exercise 3: Implementing the Builder Pattern**

**BuilderPatternTest.java**

public class BuilderPatternTest {

public static void main(String[] args) {

// Creating a basic computer

Computer basicComputer = new Computer.Builder()

.setCPU("Intel i3")

.setRAM("4GB")

.setStorage("256GB SSD")

.build();

// Creating a gaming computer

Computer gamingComputer = new Computer.Builder()

.setCPU("Intel i9")

.setRAM("32GB")

.setStorage("1TB SSD")

.setGraphicsCard("NVIDIA RTX 4080")

.setBluetoothEnabled(true)

.setWiFiEnabled(true)

.build();

// Creating an office computer

Computer officeComputer = new Computer.Builder()

.setCPU("Intel i5")

.setRAM("8GB")

.setStorage("512GB SSD")

.setWiFiEnabled(true)

.build();

// Display configurations

basicComputer.showSpecs();

gamingComputer.showSpecs();

officeComputer.showSpecs();

}

}

**Computer.java**

public class Computer {

// Required and optional attributes

private String CPU;

private String RAM;

private String storage;

private String graphicsCard;

private boolean isBluetoothEnabled;

private boolean isWiFiEnabled;

// Private constructor accessed via Builder

private Computer(Builder builder) {

this.CPU = builder.CPU;

this.RAM = builder.RAM;

this.storage = builder.storage;

this.graphicsCard = builder.graphicsCard;

this.isBluetoothEnabled = builder.isBluetoothEnabled;

this.isWiFiEnabled = builder.isWiFiEnabled;

}

// Static Nested Builder Class

public static class Builder {

private String CPU;

private String RAM;

private String storage;

private String graphicsCard;

private boolean isBluetoothEnabled;

private boolean isWiFiEnabled;

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 Builder setBluetoothEnabled(boolean isBluetoothEnabled) {

this.isBluetoothEnabled = isBluetoothEnabled;

return this;

}

public Builder setWiFiEnabled(boolean isWiFiEnabled) {

this.isWiFiEnabled = isWiFiEnabled;

return this;

}

public Computer build() {

return new Computer(this);

}

}

// Method to print computer configuration

public void showSpecs() {

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

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

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

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

System.out.println("Graphics Card: " + graphicsCard);

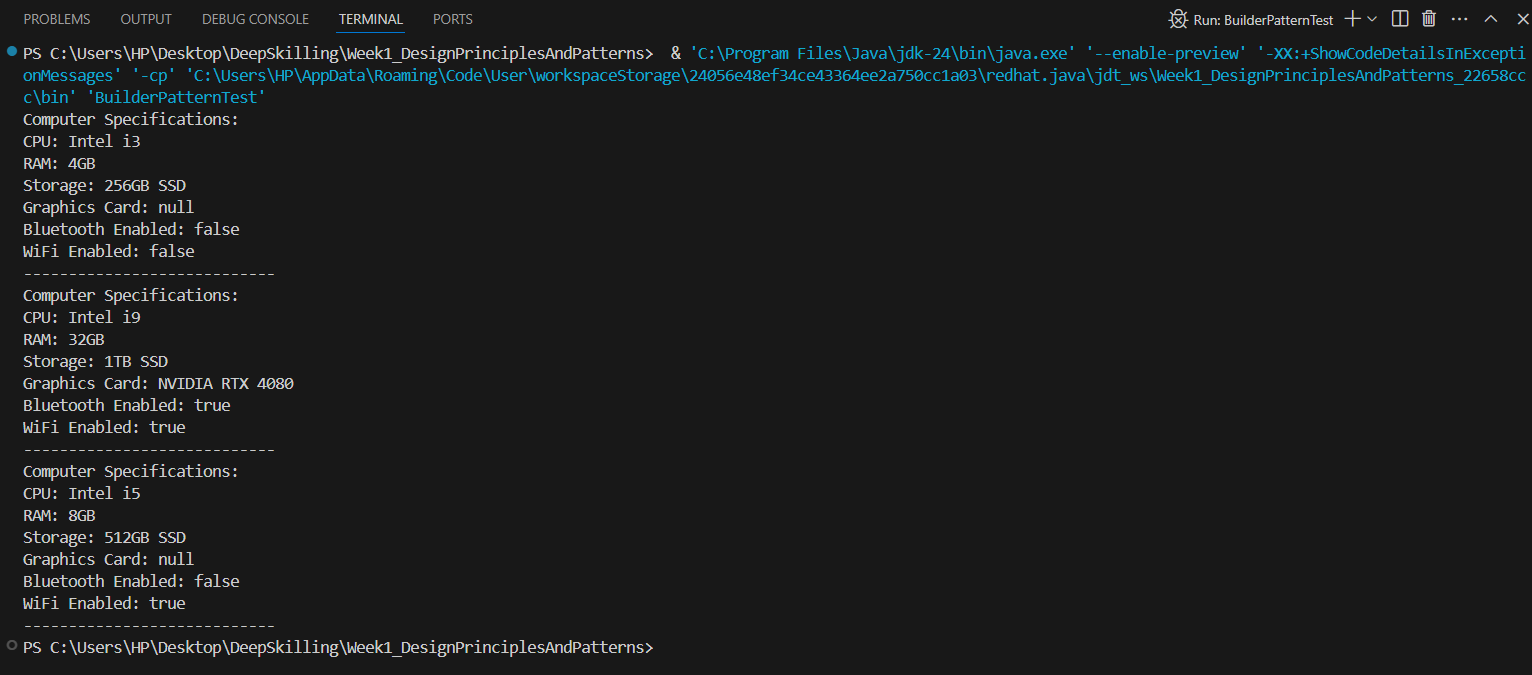
System.out.println("Bluetooth Enabled: " + isBluetoothEnabled);

System.out.println("WiFi Enabled: " + isWiFiEnabled);

System.out.println("----------------------------");

}

}

**Output:  
**

**Exercise 4: Implementing the Adapter Pattern**

**Main.java**

package src;

public class Main {

public static void main(String[] args) {

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

paypalProcessor.processPayment(150.75);

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

stripeProcessor.processPayment(200.50);

}

}

**PaymentProcessor.java**

package src;

public interface PaymentProcessor {

void processPayment(double amount);

}

**PayPalAdapter.java**

package src;

public class PayPalAdapter implements PaymentProcessor {

private PayPalGateway payPal;

public PayPalAdapter(PayPalGateway payPal) {

this.payPal = payPal;

}

public void processPayment(double amount) {

payPal.makePayment(amount);

}

}

**PayPalGateway.java**

package src;

public class PayPalGateway {

public void makePayment(double amount) {

System.out.println("Processing payment via PayPal: $" + amount);

}

}

**StripeAdapter.java**

package src;

public class StripeAdapter implements PaymentProcessor {

private StripeGateway stripe;

public StripeAdapter(StripeGateway stripe) {

this.stripe = stripe;

}

public void processPayment(double amount) {

stripe.pay(amount);

}

}

**StripeGateway.java**

package src;

public class StripeGateway {

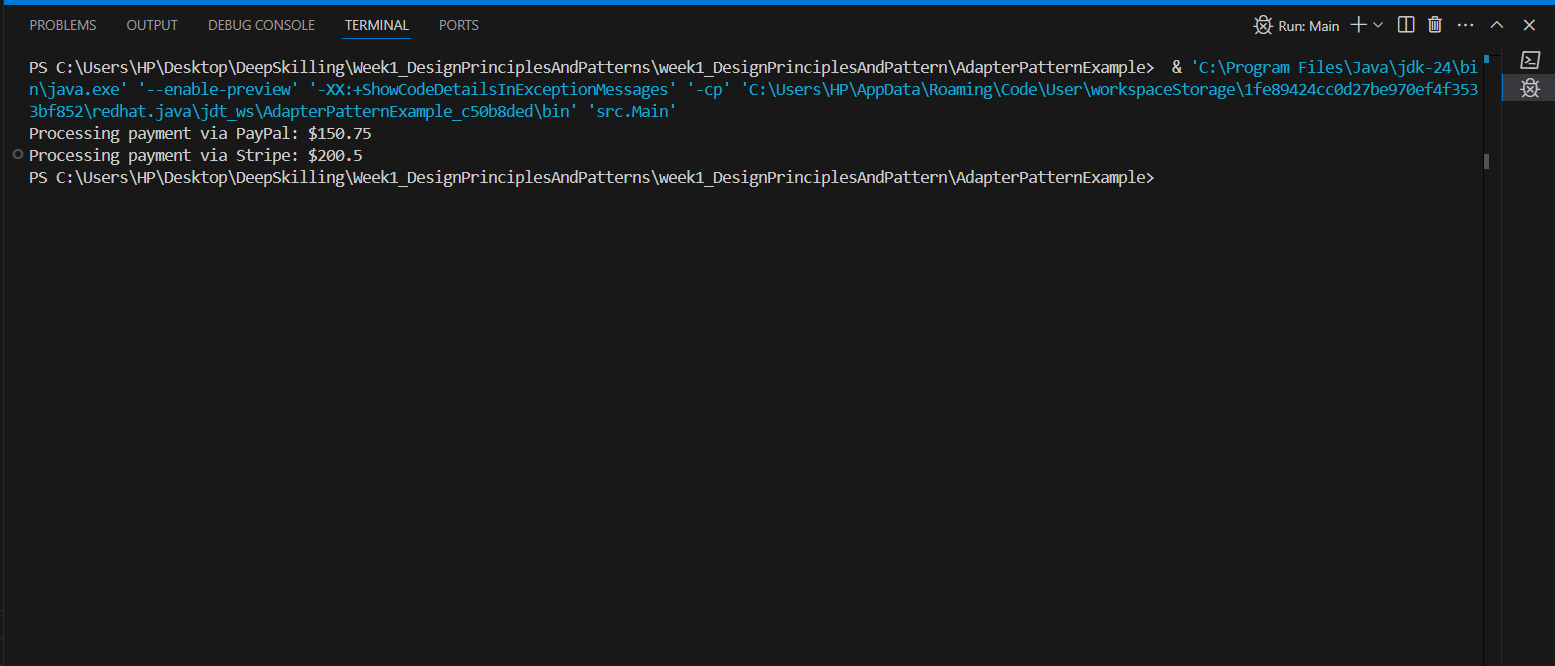
public void pay(double amount) {

System.out.println("Processing payment via Stripe: $" + amount);

}

}

**Output:**

****

**Exercise 5: Implementing the Decorator Pattern**

**EmailNotifier.java**

package src;

public class EmailNotifier implements Notifier {

public void send(String message) {

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

}

}

**Main.java**

package src;

public class Main {

public static void main(String[] args) {

Notifier notifier = new EmailNotifier();

notifier = new SMSNotifierDecorator(notifier);

notifier = new SlackNotifierDecorator(notifier);

notifier.send("System maintenance at 10 PM.");

}

}

**Notifier.java**

package src;

public interface Notifier {

void send(String message);

}

**NotifierDecorator.java**

package src;

public abstract class NotifierDecorator implements Notifier {

protected Notifier wrappee;

public NotifierDecorator(Notifier notifier) {

this.wrappee = notifier;

}

public void send(String message) {

wrappee.send(message);

}

}

**SlackNotifierDecorator.java**

package src;

public class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

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

}

}

**SMSNotifierDecorator.java**

package src;

public class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

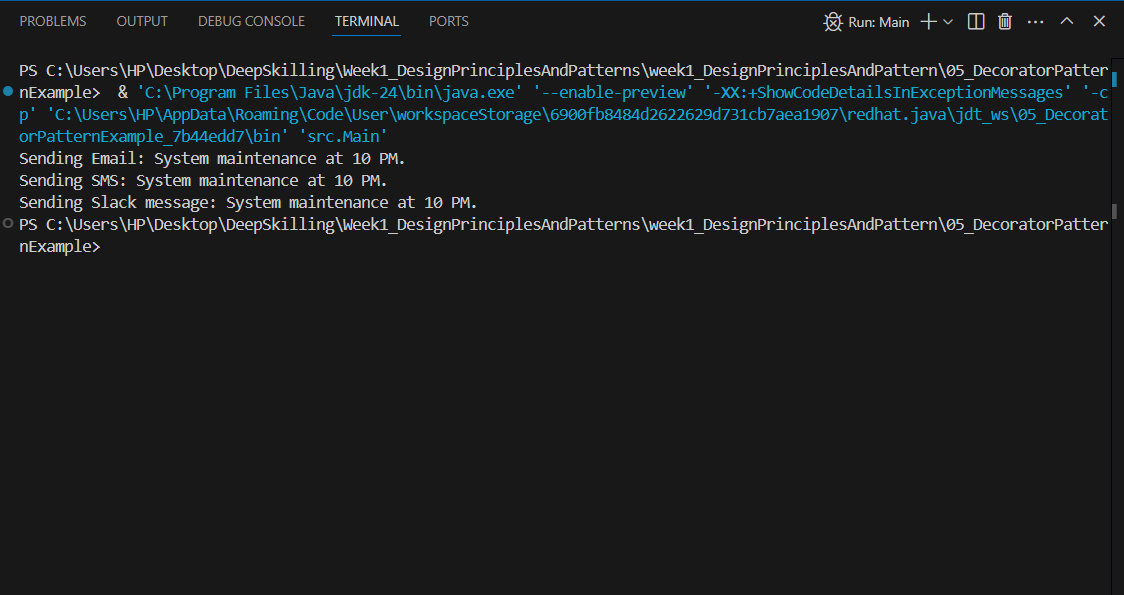
super.send(message);

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

}

}

**Output:**



**Exercise 6: Implementing the Proxy Pattern**

**Image.java**

package src;

public interface Image {

void display();

}

**Main.java**

package src;

public class Main {

public static void main(String[] args) {

Image image1 = new ProxyImage("nature1.jpg");

Image image2 = new ProxyImage("nature2.jpg");

// Image will be loaded from remote server only once

image1.display(); // Loads and displays

image1.display(); // Just displays from cache

image2.display(); // Loads and displays

}

}

**ProxyImage.java**

package src;

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();

}

}

**RealImage.java**

package src;

public class RealImage implements Image {

private String filename;

public RealImage(String filename) {

this.filename = filename;

loadFromRemoteServer();

}

private void loadFromRemoteServer() {

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

}

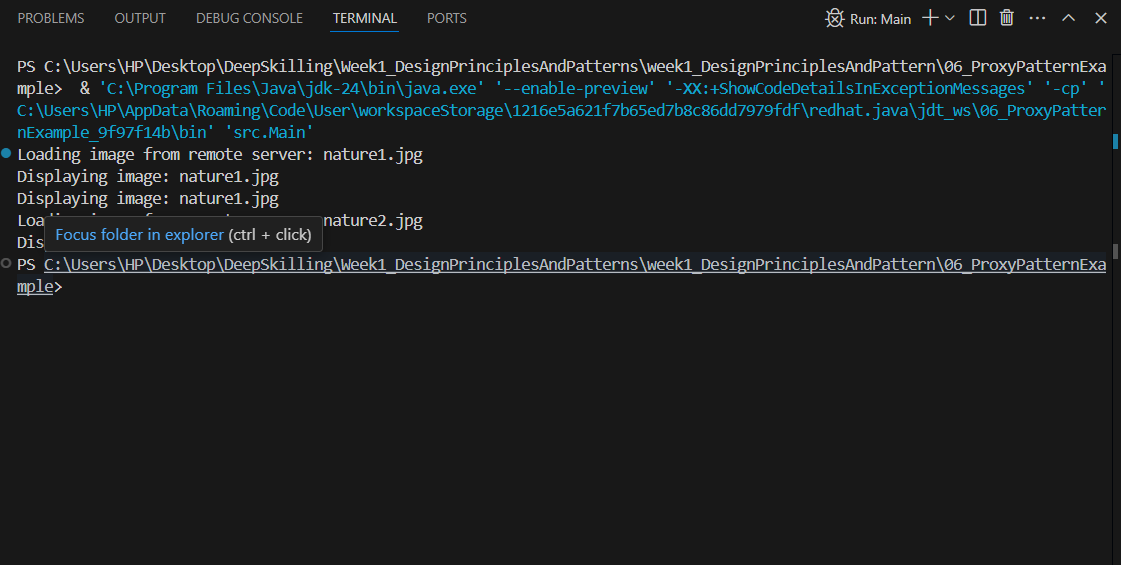
public void display() {

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

}

}

**Output:**

****

**Exercise 7: Implementing the Observer Pattern**

**Mobile.java**

package src;

public class MobileApp implements Observer {

private String appName;

public MobileApp(String appName) {

this.appName = appName;

}

@Override

public void update(String stockName, double stockPrice) {

System.out.println(appName + " - Stock Update: " + stockName + " is now $" + stockPrice);

}

}

**Observer.java**

package src;

public interface Observer {

void update(String stockName, double stockPrice);

}

**ObserverPattern.java**

package src;

public class ObserverPatternTest {

public static void main(String[] args) {

StockMarket stockMarket = new StockMarket();

Observer mobileApp = new MobileApp("PhoneNotifier");

Observer webApp = new WebApp("BrowserDashboard");

stockMarket.registerObserver(mobileApp);

stockMarket.registerObserver(webApp);

stockMarket.setStock("Apple", 190.50);

stockMarket.setStock("Google", 2850.75);

stockMarket.removeObserver(mobileApp);

stockMarket.setStock("Amazon", 3450.20);

}

}

**Stock.java**

package src;

public interface Stock {

void registerObserver(Observer o);

void removeObserver(Observer o);

void notifyObservers();

}

**StockMarket.java**

package src;

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 void setStock(String stockName, double stockPrice) {

this.stockName = stockName;

this.stockPrice = stockPrice;

notifyObservers();

}

@Override

public void registerObserver(Observer o) {

observers.add(o);

}

@Override

public void removeObserver(Observer o) {

observers.remove(o);

}

@Override

public void notifyObservers() {

for (Observer o : observers) {

o.update(stockName, stockPrice);

}

}

}

**WebApp.java**

package src;

public class WebApp implements Observer {

private String siteName;

public WebApp(String siteName) {

this.siteName = siteName;

}

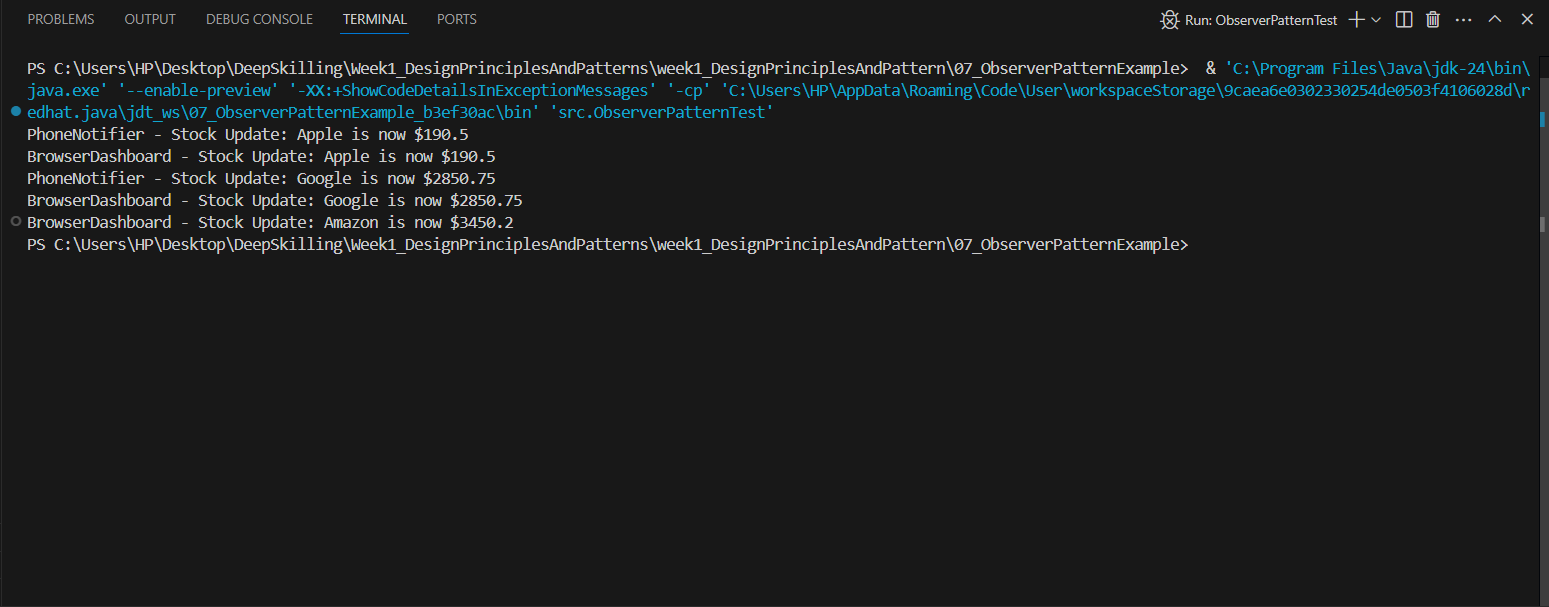
@Override

public void update(String stockName, double stockPrice) {

System.out.println(siteName + " - Stock Update: " + stockName + " is now $" + stockPrice);

}

}  
**Output:**

****

**Exercise 8: Implementing the Strategy Pattern**

**CreditCardPayment.java**

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: " + cardNumber);

}

}

**PaymentContext.java**

public class PaymentContext {

private PaymentStrategy strategy;

public void setPaymentStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void executePayment(double amount) {

strategy.pay(amount);

}

}

**PaymentStrategy.java**

public interface PaymentStrategy {

void pay(double amount);

}

**PayPalPayment.java**

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: " + email);

}

}

**StrategeyPatternTest.java**

public class StrategyPatternTest {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

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

context.executePayment(250.00);

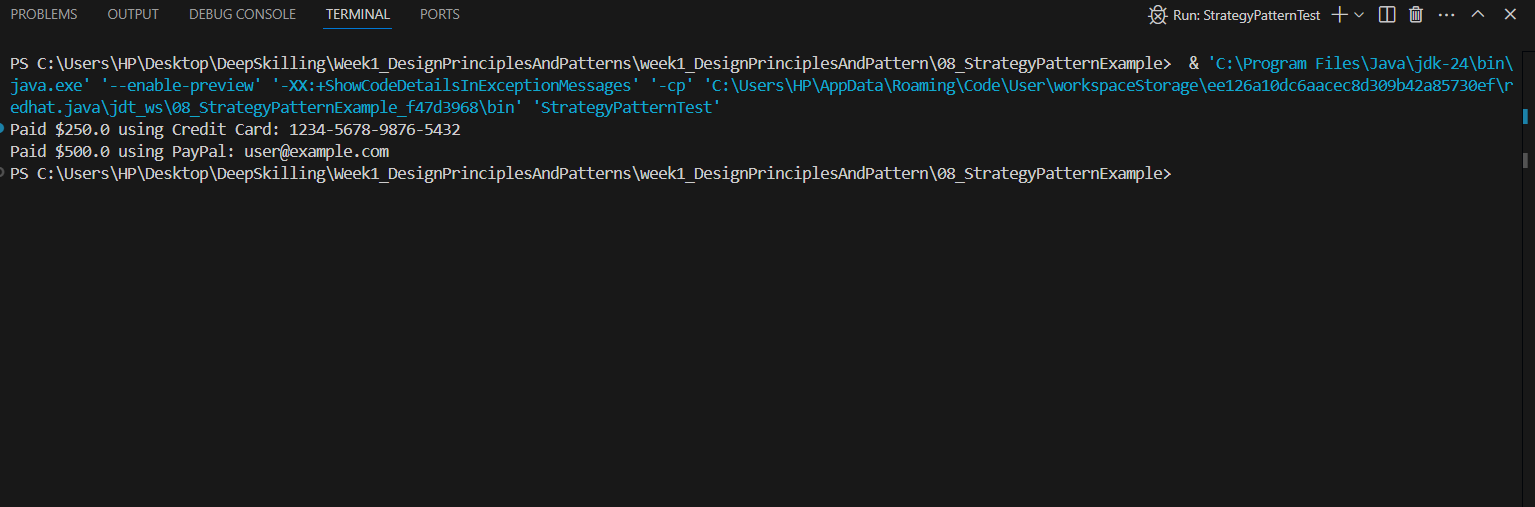
context.setPaymentStrategy(new PayPalPayment("user@example.com"));

context.executePayment(500.00);

}

}

**Output:**

****

**Exercise 9: Implementing the Command Pattern**

**Command.java**

public interface Command {

void execute();

}

**CommandPatternTest.java**

public class CommandPatternTest {

public static void main(String[] args) {

Light light = new Light();

Command lightOn = new LightOnCommand(light);

Command lightOff = new LightOffCommand(light);

RemoteControl remote = new RemoteControl();

remote.setCommand(lightOn);

remote.pressButton();

remote.setCommand(lightOff);

remote.pressButton();

}

}

**Light.java**

public class Light {

public void turnOn() {

System.out.println("The light is ON");

}

public void turnOff() {

System.out.println("The light is OFF");

}

}

**LightOffCommand.java**

public class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOff();

}

}

**LightOnCommand.java**

public class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOn();

}

}

**RemoteControl.java**

public class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

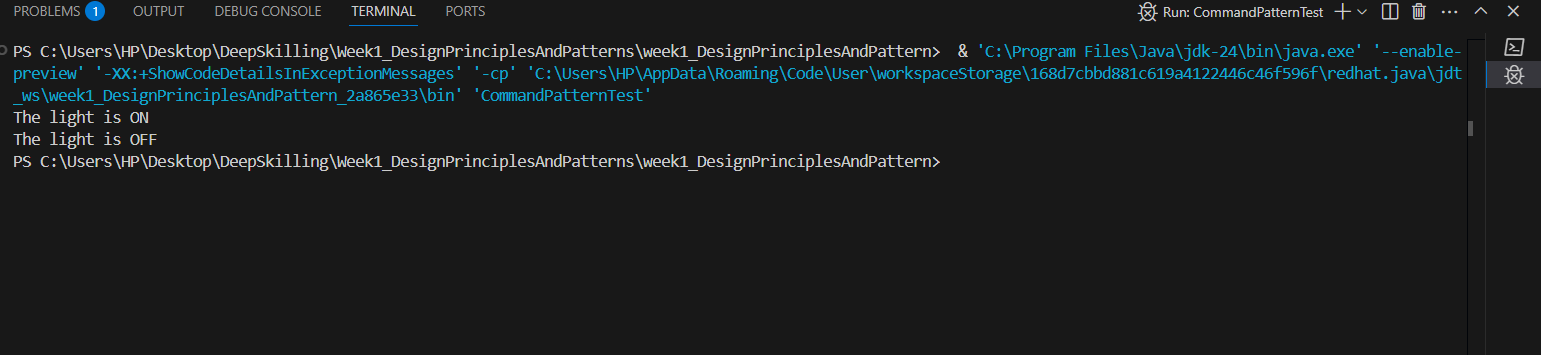
public void pressButton() {

command.execute();

}

}

**Output:**

****

**Exercise 10: Implementing the MVC Pattern**

**MVCTest.java**

public class MVCTest {

public static void main(String[] args) {

Student model = new Student();

model.setName("Alice");

model.setId("S123");

model.setGrade("A");

StudentView view = new StudentView();

StudentController controller = new StudentController(model, view);

controller.updateView();

controller.setStudentGrade("A+");

controller.updateView();

}

}

**Student.java**

public class Student {

private String id;

private String name;

private String grade;

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getGrade() {

return grade;

}

public void setGrade(String grade) {

this.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());

}

}

**StudentView.java**

public class StudentView {

public voiddisplayStudentDetails(String studentName, String studentId, String grade) {

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

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

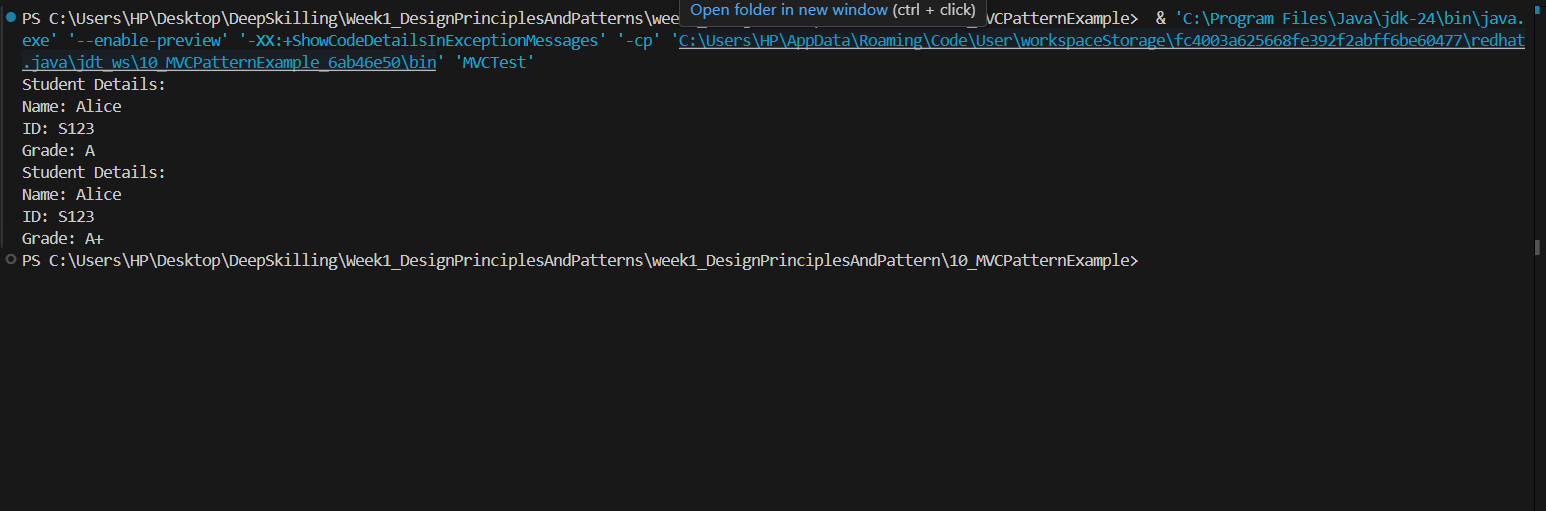
System.out.println("ID: " + studentId);

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

}

}

**Output:**

****

**Exercise 11: Implementing Dependency Injection**

**CustomerRepository.java**

public interface CustomerRepository {

String findCustomerById(String id);

}

CustomerRepositoryImpl.java

public class CustomerRepositoryImpl implements CustomerRepository {

@Override

public String findCustomerById(String id) {

return "Customer{id='" + id + "', name='John Doe'}";

}

}

**CustomerService.java**

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("Customer Found: " + customer);

}

}

**DIExample.java**

public class DIExampleTest {

public static void main(String[] args) {

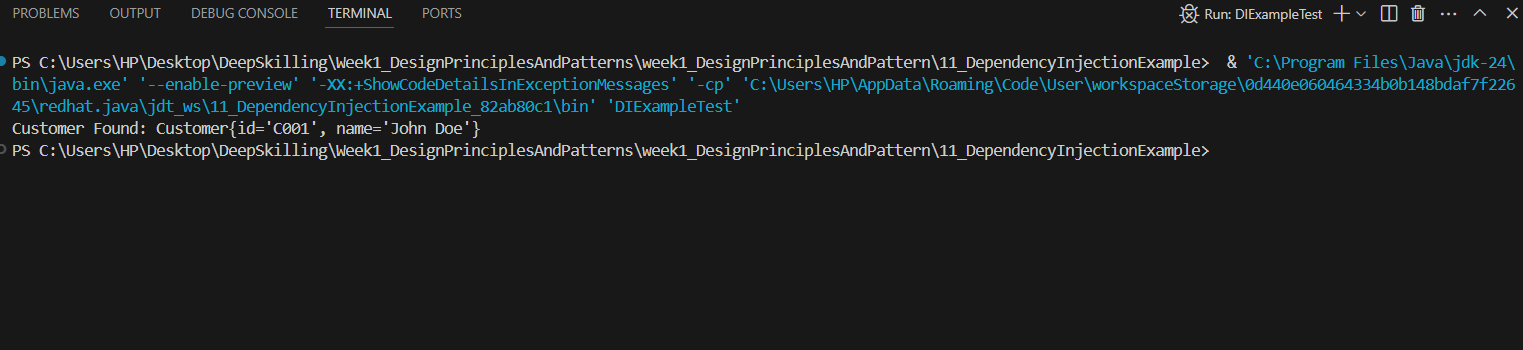
CustomerRepository repository = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repository);

service.displayCustomer("C001");

}

}

**Output:  
**