**Design Patterns Solutions**

**Exercise 1: Singleton Pattern**

**Logger.java**

public class Logger {

    private static Logger l;

    private Logger() {

    }

    public static Logger getInstance() {

        if (l == null) {

            l = new Logger();

        }

        return l;

    }

    public void log(String message) {

        System.out.println(message);

    }

}

**Loggertest.java**

public class LoggerTest {

    public static void main(String[] args) {

        Logger l1 = Logger.getInstance();

        Logger l2 = Logger.getInstance();

        if (l1 == l2) {

            System.out.println("Logger is a singleton. Both references point to the same instance.");

        } else {

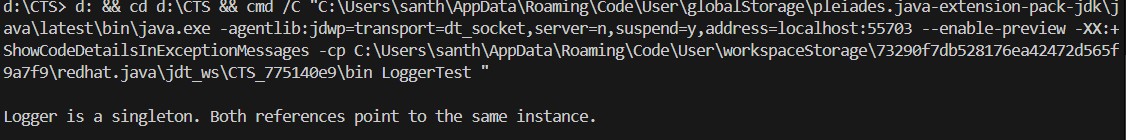
            System.out.println("Logger is not a singleton. Different instances were created.");

        }

    }

}

**OUTPUT:**



**SingletonPatternExample.java**

public class SingletonPatternExample {

    public static void main(String[] args) {

        Logger l1 = Logger.getInstance();

        Logger l2 = Logger.getInstance();

        System.out.println("Logger 1: " + l1);

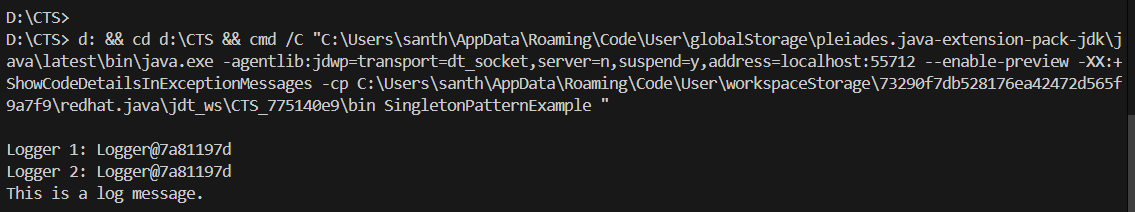
        System.out.println("Logger 2: " + l2);

        l1.log("This is a log message.");

    }

}

**OUTPUT:**



**Exercise 2: Factory Method Pattern**

**Document.java**

public interface Document {

    String getContent();

    String getType();

}

**ExcelDocument.java**

public class ExcelDocument implements Document {

    @Override

    public String getContent() {

        return "This is an Excel document.";

    }

    @Override

    public String getType() {

        return "Excel";

    }

}

**PdfDocument.java**

public class PdfDocument implements Document {

    @Override

    public String getContent() {

        return "This is the content of a PDF document.";

    }

    @Override

    public String getType() {

        return "PDF Document";

    }

}

**WordDocument.java**

public class WordDocument implements Document {

    @Override

    public String getContent() {

        return "This is the content of a Word document.";

    }

    @Override

    public String getType() {

        return "Word Document";

    }

}

**DocumentFactory.java**

public abstract class DocumentFactory {

    public abstract Document createDocument();

}

**ExcelDocumentFactory.java**

public class ExcelDocumentFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new ExcelDocument();

    }

}

**PdfDocumentFactory.java**

public class PdfDocumentFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new PdfDocument();

    }

}

**WordDocumentFactory.java**

public class WordDocumentFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new WordDocument();

    }

}

**FactoryMethodPatternExample.java**

public class FactoryMethodPatternExample {

    public static void main(String[] args) {

        DocumentFactory wordFactory = new WordDocumentFactory();

        Document wordDoc = wordFactory.createDocument();

        System.out.println("Created: " + wordDoc.getType() + " with content: " + wordDoc.getContent());

        DocumentFactory pdfFactory = new PdfDocumentFactory();

        Document pdfDoc = pdfFactory.createDocument();

        System.out.println("Created: " + pdfDoc.getType() + " with content: " + pdfDoc.getContent());

        DocumentFactory excelFactory = new ExcelDocumentFactory();

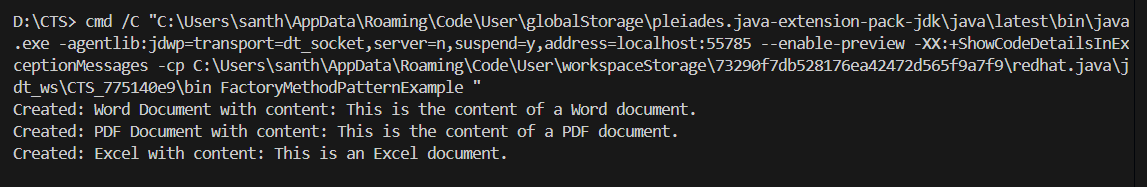
        Document excelDoc = excelFactory.createDocument();

        System.out.println("Created: " + excelDoc.getType() + " with content: " + excelDoc.getContent());

    }

}

**OUTPUT:**

****

**Exercise 3: Builder Pattern**

**Computer.java**

public class Computer {

    private String cpu;

    private String ram;

    private String storage;

    private Computer(Builder b) {

        this.cpu = b.cpu;

        this.ram = b.ram;

        this.storage = b.storage;

    }

    public static class Builder {

        private String cpu;

        private String ram;

        private String 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() {

            return new Computer(this);

        }

    }

    public void display() {

        System.out.println("CPU: " + cpu + ", RAM: " + ram + ", Storage: " + storage);

    }

}

**Test.java**

public class Test {

    public static void main(String[] args) {

        Computer c1 = new Computer.Builder()

                .setCpu("Intel i5")

                .setRam("8GB")

                .setStorage("256GB SSD")

                .build();

        Computer c2 = new Computer.Builder()

                .setCpu("AMD Ryzen")

                .setRam("16GB")

                .build();

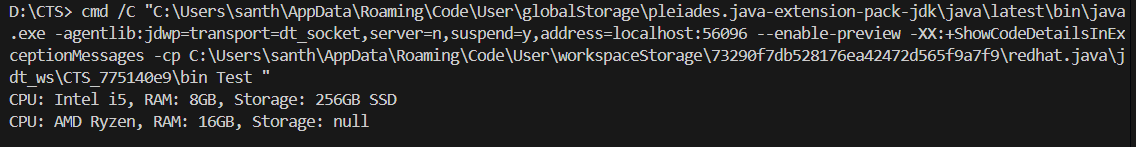
        c1.display();

        c2.display();

    }

}

**Output:**



**Exercise 4: Adapter Pattern**

**PaymentProcessor.java**

public interface PaymentProcessor {

    void processPayment(double amount);

}

**PayPalGateway.java**

public class PayPalGateway {

    public void payWithPayPal(double amount) {

        System.out.println("Processing payment of $" + amount + " through PayPal.");

    }

}

**StripeGateway.java**

public class StripeGateway {

    public void payWithStripe(double amount) {

        System.out.println("Processing payment of $" + amount + " through Stripe.");

    }

}

**PayPalAdapter.java**

public class PayPalAdapter implements PaymentProcessor {

    private PayPalGateway payPalGateway;

    public PayPalAdapter(PayPalGateway payPalGateway) {

        this.payPalGateway = payPalGateway;

    }

    @Override

    public void processPayment(double amount) {

        payPalGateway.payWithPayPal(amount);

    }

}

**StripeAdapter.java**

public class StripeAdapter implements PaymentProcessor {

    private StripeGateway stripeGateway;

    public StripeAdapter() {

        this.stripeGateway = new StripeGateway();

    }

    public StripeAdapter(StripeGateway stripeGateway) {

        this.stripeGateway = stripeGateway;

    }

    @Override

    public void processPayment(double amount) {

        stripeGateway.payWithStripe(amount);

    }

}

**Test.java**

public class AdapterPatternTest {

    public static void main(String[] args) {

        PayPalGateway pg = new PayPalGateway();

        StripeGateway sg = new StripeGateway();

        PaymentProcessor pa = new PayPalAdapter(pg);

        PaymentProcessor sa = new StripeAdapter(sg);

        System.out.println("Processing payment with PayPal:");

        pa.processPayment(100.0);

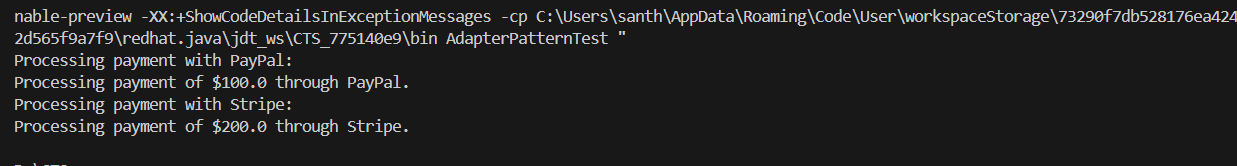
        System.out.println("Processing payment with Stripe:");

        sa.processPayment(200.0);

    }

}

**Output:**



**Exercise 5: Decorator Pattern**

**Notifier.java**

public interface Notifier {

    void send(String message);

}

**EmailNotificationDecorator.java**

public class EmailNotificationDecorator implements Notifier {

    protected Notifier notifier;

    public EmailNotificationDecorator(Notifier notifier) {

        this.notifier = notifier;

    }

    @Override

    public void send(String message) {

        notifier.send(message);

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

    }

}

**NotificationDecorator.java**

public abstract class NotificationDecorator implements Notifier {

    protected Notifier decoratedNotifier;

    public NotificationDecorator(Notifier decoratedNotifier) {

        this.decoratedNotifier = decoratedNotifier;

    }

    @Override

    public void send(String msg) {

        decoratedNotifier.send(msg);

    }

}

**SMSNotificationDecorator.java**

public class SMSNotificationDecorator extends NotificationDecorator {

    public SMSNotificationDecorator(Notifier notifier) {

        super(notifier);

    }

   private void sendSMS(String message) {

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

    }

}

**BasicNotification.java**

public class BasicNotification implements Notifier {

    @Override

    public void send(String message) {

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

    }

}

**Test.java**

public class Test {

    public static void main(String[] args) {

        Notifier basicNotification = new BasicNotification();

        Notifier emailNotification = new EmailNotificationDecorator(basicNotification);

        Notifier smsNotification = new SMSNotificationDecorator(basicNotification);

        System.out.println("Sending Basic Notification:");

        basicNotification.send("hello");

        System.out.println("\n Sending Email Notification:");

        emailNotification.send("hello1");

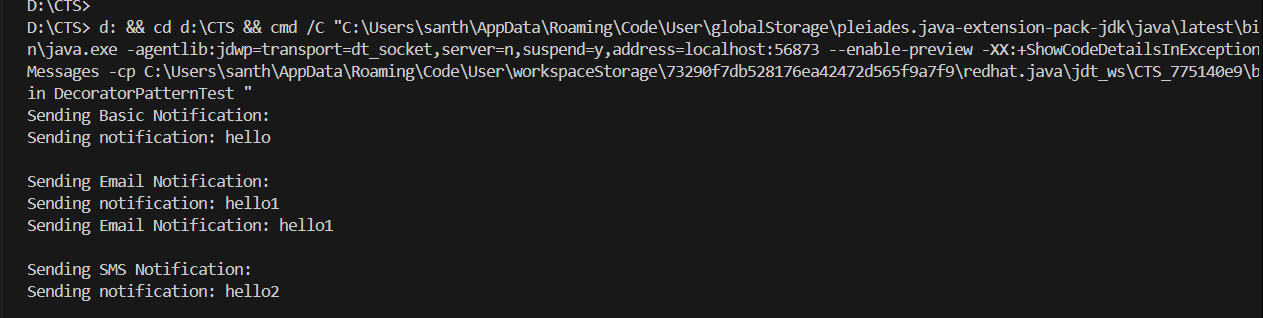
        System.out.println("\nSending SMS Notification:");

        smsNotification.send("hello2");

    }

}

**Output:**



**Exercise 6: Proxy Pattern**

**Image.java**

public interface Image {

    void display();

}

**RealImage.java**

public class RealImage implements Image {

    private String filename;

    public RealImage(String filename) {

        this.filename = filename;

        loadImageFromDisk();

    }

    private void loadImageFromDisk() {

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

    }

    @Override

    public void display() {

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

    }

}

**ProxyImage.java**

public class ProxyImage implements Image {

    private RealImage realImage;

    private String imageName;

    public ProxyImage(String imageName) {

        this.imageName = imageName;

    }

    @Override

    public void display() {

        if (realImage == null) {

            realImage = new RealImage(imageName);

        }

        realImage.display();

    }

}

**ProxyPatternTest.java**

public class ProxyPatternTest {

    public static void main(String[] args) {

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

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

        image1.display();

        System.out.println();

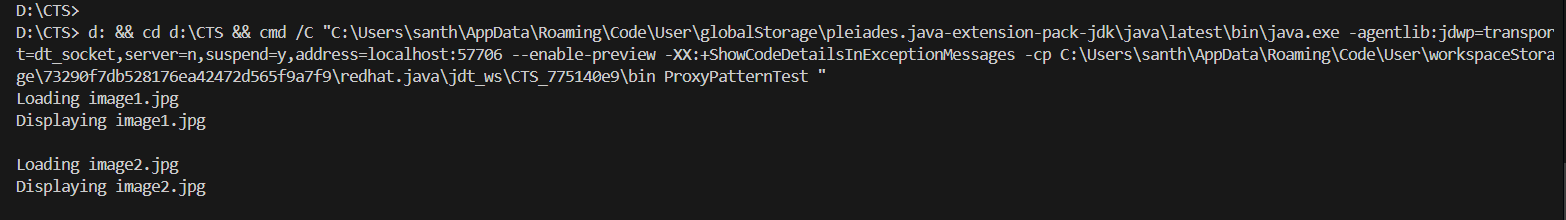
        image2.display();

        System.out.println();

    }

}

**Output:**



**Exercise 7: Observer Pattern**

**Observer.java**

package ObserverPatternExample;

public interface Observer {

    void update(int price);

}

**Stock.java**

package ObserverPatternExample;

public interface Stock {

    void register(Observer o);

    void deregister(Observer o);

    void notifyObservers();

}

**StockMarket.java**

package ObserverPatternExample;

import java.util.ArrayList;

import java.util.List;

public class StockMarket implements Stock {

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

    private int stockPrice;

    @Override

    public void register(Observer o) {

        observers.add(o);

    }

    @Override

    public void deregister(Observer o) {

        observers.remove(o);

    }

    @Override

    public void notifyObservers() {

        for (Observer o : observers) {

            o.update(stockPrice);

        }

    }

    public void setPrice(int price) {

        this.stockPrice = price;

        notifyObservers();

    }

}

**MobileApp.java**

package ObserverPatternExample;

public class MobileApp implements Observer {

    @Override

    public void update(int price) {

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

    }

}

**Test.java**

package ObserverPatternExample;

public class Main {

    public static void main(String[] args) {

        StockMarket stockMarket = new StockMarket();

        Observer mobile = new MobileApp();

        Observer web = new WebApp();

        stockMarket.register(mobile);

        stockMarket.register(web);

        stockMarket.setPrice(100);

        stockMarket.setPrice(150);

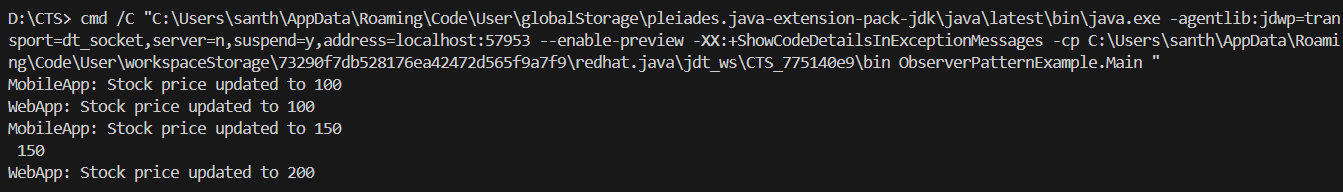
        stockMarket.deregister(mobile);

        stockMarket.setPrice(200);

    }

}

**Output:**



**Exercise 8: Strategy Pattern**

**PaymentStrategy.java**

package StrategyPattern;

public interface PaymentStrategy {

    void pay(double amount);

}

**CreditCardPayment.java**

package StrategyPattern;

public class CreditCardPayment implements PaymentStrategy {

    public void pay(double amount) {

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

    }

}

**PayPalPayment.java**

package StrategyPattern;

public class PayPalPayment implements PaymentStrategy {

    public void pay(double amount) {

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

    }

}

**PaymentContext.java**

package StrategyPattern;

public class PaymentContext {

    private PaymentStrategy strategy;

    public void setStrategy(PaymentStrategy strategy) {

        this.strategy = strategy;

    }

    public void executePayment(double amount) {

        strategy.pay(amount);

    }

}

**Test.java**

package StrategyPattern;

public class Test {

    public static void main(String[] args) {

        PaymentContext context = new PaymentContext();

        context.setStrategy(new CreditCardPayment());

        context.executePayment(100.0);

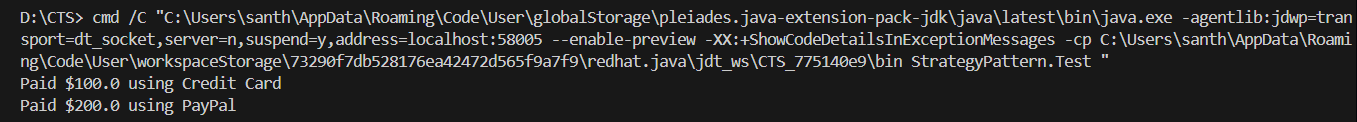
        context.setStrategy(new PayPalPayment());

        context.executePayment(200.0);

    }

}

**Output:**



**Exercise 9: Command Pattern**

**Command.java**

public interface Command {

    void execute();

}

**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");

    }

}

**LightOnCommand.java**

public class LightOnCommand implements Command {

    private Light light;

    public LightOnCommand(Light light) {

        this.light = light;

    }

    @Override

    public void execute() {

        light.turnOn();

    }

}

**LightOffCommand.java**

public class LightOffCommand implements Command {

    private Light light;

    public LightOffCommand(Light light) {

        this.light = light;

    }

    @Override

    public void execute() {

        light.turnOff();

    }

}

**RemoteControl.java**

public class RemoteControl {

    private Command command;

    public void setCommand(Command command) {

        this.command = command;

    }

    public void pressButton() {

        if (command != null) {

            command.execute();

        } else {

            System.out.println("No command set.");

        }

    }

}

**Test.java**

public class Test {

    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();

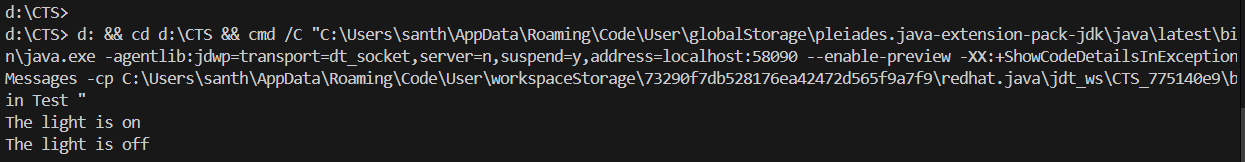
        remote.setCommand(lightOff);

        remote.pressButton();

    }

}

**Output:**



**Exercise 10: MVC Pattern**

**Student.java**

package controller;

public class Student {

    private String name;

    private int id;

    private String grade;

    public Student(String name, int id, String grade) {

        this.name = name;

        this.id = id;

        this.grade = grade;

    }

    public String getName() {

        return name;

    }

    public void setName(String name) {

        this.name = name;

    }

    public int getId() {

        return id;

    }

    public void setId(int id) {

        this.id = id;

    }

    public String getGrade() {

        return grade;

    }

    public void setGrade(String grade) {

        this.grade = grade;

    }

}

**StudentView.java**

package controller;

public class StudentView {

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

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

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

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

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

    }

}

**StudentController.java**

package controller;

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());

    }

}

**Test.java**

package controller;

public class Test {

    public static void main(String[] args) {

        Student student = new Student("John", 1, "A");

        StudentView view = new StudentView();

        StudentController controller = new StudentController(student, view);

        controller.updateView();

        controller.setStudentName("Jane");

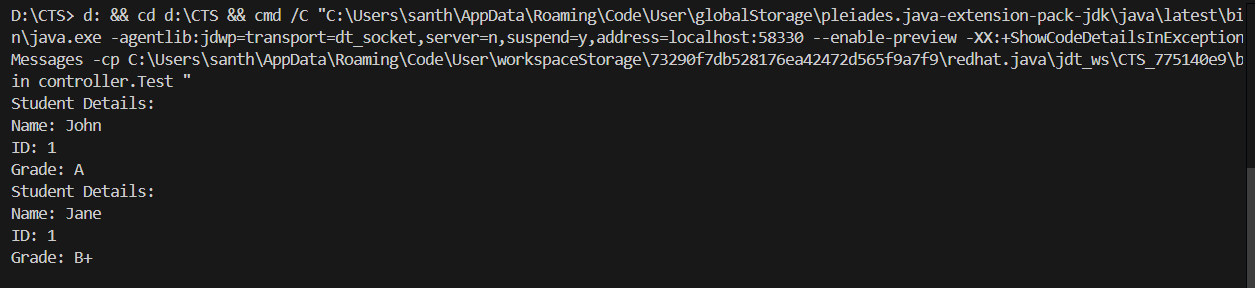
        controller.setStudentGrade("B+");

        controller.updateView();

    }

}

**Output:**



**Exercise 11: Dependency Injection**

**CustomerRepository.java**

public interface CustomerRepository {

    String findCustomerById(int id);

}

**CustomerRepositoryImpl.java**

public class CustomerRepositoryImpl implements CustomerRepository {

    @Override

    public String findCustomerById(int id) {

        return "Customer with ID " + id + " found.";

    }

}

**CustomerService.java**

public class CustomerService {

    private CustomerRepository repo;

    public CustomerService(CustomerRepository repo) {

        this.repo = repo;

    }

    public String getCustomer(int id) {

        return repo.findCustomerById(id);

    }

}

**Test.java**

public class Test {

    public static void main(String[] args) {

        CustomerRepository repo = new CustomerRepositoryImpl();

        CustomerService service = new CustomerService(repo);

        String customerInfo = service.getCustomer(123);

        System.out.println(customerInfo);

    }

}

**Output:**

