**WEEK 1 Hands On**

**Design Patterns and Principles**

**Exercise 1: Implementing the Singleton Pattern**

**CODE**

**Logger.java**

public class Logger {

    // Private static variable

    private static Logger instance = new Logger();

    // Private constructor

    private Logger() {

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

    }

    // Public static method

    public static Logger getInstance() {

        return instance;

    }

    // Sample method to simulate logging

    public void log(String message) {

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

    }

}

**TestSingleton.java**

public class TestSingleton {

    public static void main(String[] args) {

        // Two instances of Logger

        Logger logger1 = Logger.getInstance();

        Logger logger2 = Logger.getInstance();

        // Using the logger to print messages

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

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

        // Checking both references

        if (logger1 == logger2) {

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

        } else {

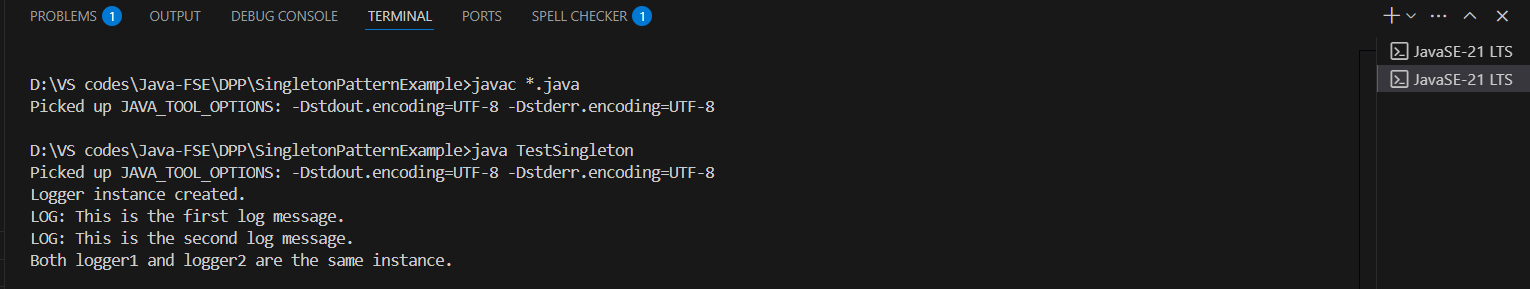
            System.out.println("logger1 and logger2 are different instances.");

        }

    }

}

**OUTPUT**

****

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

**CODE**

**Document.java**

package DPP.FactoryMethodPatternExample;

public interface Document {

    void open();

}

**DocumentFactory.java**

package DPP.FactoryMethodPatternExample;

public abstract class DocumentFactory {

    public abstract Document createDocument();

}

**ExcelDocument.java**

package DPP.FactoryMethodPatternExample;

public class ExcelDocument implements Document {

    @Override

    public void open() {

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

    }

}

**ExcelDocumentFactory.java**

package DPP.FactoryMethodPatternExample;

public class ExcelDocumentFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new ExcelDocument();

    }

}

**PdfDocument.java**

package DPP.FactoryMethodPatternExample;

public class PdfDocument implements Document {

    @Override

    public void open() {

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

    }

}

**PdfDocumentFactory.java**

package DPP.FactoryMethodPatternExample;

public class PdfDocumentFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new PdfDocument();

    }

}

**WordDocument.java**

package DPP.FactoryMethodPatternExample;

public class WordDocument implements Document {

    @Override

    public void open() {

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

    }

}

**WordDocumentFactory.java**

package DPP.FactoryMethodPatternExample;

public class WordDocumentFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new WordDocument();

    }

}

**TestFactoryMethod.java**

package DPP.FactoryMethodPatternExample;

public class TestFactoryMethod {

    public static void main(String[] args) {

        // Create Word document

        DocumentFactory wordFactory = new WordDocumentFactory();

        Document wordDoc = wordFactory.createDocument();

        wordDoc.open();

        // Create PDF document

        DocumentFactory pdfFactory = new PdfDocumentFactory();

        Document pdfDoc = pdfFactory.createDocument();

        pdfDoc.open();

        // Create Excel document

        DocumentFactory excelFactory = new ExcelDocumentFactory();

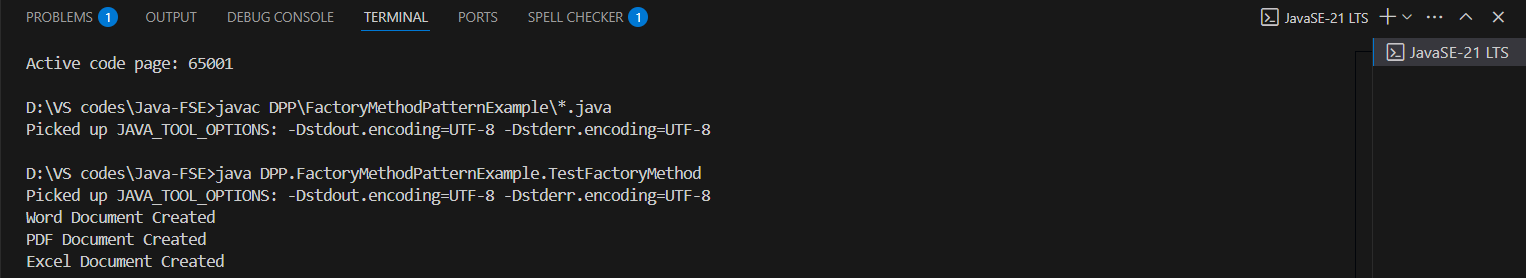
        Document excelDoc = excelFactory.createDocument();

        excelDoc.open();

    }

}

**OUTPUT**



**Exercise 3: Implementing the Builder Pattern**

**CODE**

**Computer.java**

package DPP.BuilderPatternExample;

public class Computer {

    private String cpu;

    private String ram;

    private String storage;

    // Private constructor

    private Computer(Builder builder) {

        this.cpu = builder.cpu;

        this.ram = builder.ram;

        this.storage = builder.storage;

    }

    // Getters for testing or displaying

    public String getCpu()

    {

        return cpu;

    }

    public String getRam()

    {

        return ram;

    }

    public String getStorage()

    {

        return storage;

    }

    // To display configuration

    public void showSpecs() {

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

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

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

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

    }

    // Static nested Builder class

    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;

        }

        // build()

        public Computer build() {

            return new Computer(this);

        }

    }

}

**TestBuilderPattern.java**

package DPP.BuilderPatternExample;

public class TestBuilderPattern {

    public static void main(String[] args) {

        Computer basicComputer = new Computer.Builder()

            .setCpu("Intel i3")

            .setRam("8GB")

            .setStorage("256GB SSD")

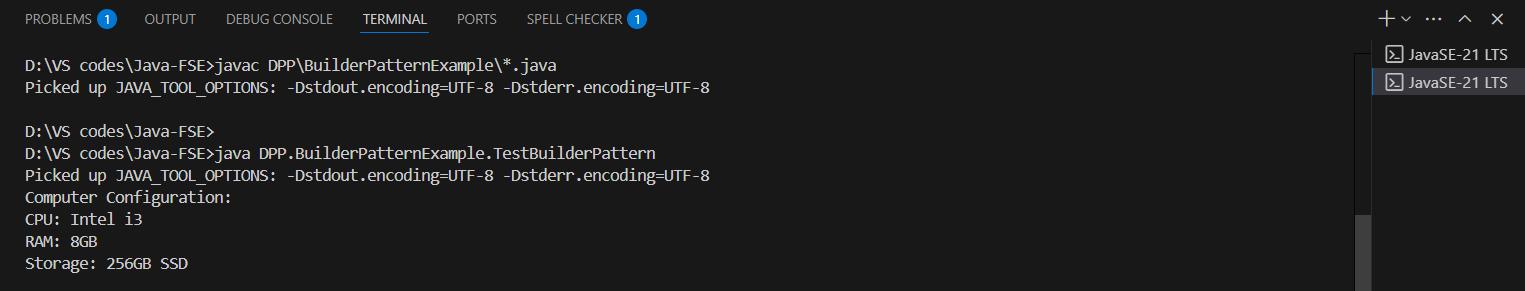
            .build();

        basicComputer.showSpecs();

    }

}

**OUTPUT**



**Exercise 4: Implementing the Adapter Pattern**

**CODE**

**PaymentProcessor.java**

package DPP.AdapterPatternExample;

public interface PaymentProcessor {

    void processPayment(double amount);

}

**PayPalAdapter.java**

package DPP.AdapterPatternExample;

public class PayPalAdapter implements PaymentProcessor {

    private PayPalGateway paypal;

    public PayPalAdapter(PayPalGateway paypal) {

        this.paypal = paypal;

    }

    @Override

    public void processPayment(double amount) {

        paypal.makePayment(amount);

    }

}

**PayPalGateway.java**

package DPP.AdapterPatternExample;

public class PayPalGateway {

    public void makePayment(double amount) {

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

    }

}

**StripeAdapter.java**

package DPP.AdapterPatternExample;

public class StripeAdapter implements PaymentProcessor {

    private StripeGateway stripe;

    public StripeAdapter(StripeGateway stripe) {

        this.stripe = stripe;

    }

    @Override

    public void processPayment(double amount) {

        stripe.payAmount(amount);

    }

}

**StripeGateway.java**

package DPP.AdapterPatternExample;

public class StripeGateway {

    public void payAmount(double amount) {

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

    }

}

**TestAdapterPattern.java**

package DPP.AdapterPatternExample;

public class TestAdapterPattern {

    public static void main(String[] args) {

        // Using PayPal

        PayPalGateway paypal = new PayPalGateway();

        PaymentProcessor paypalProcessor = new PayPalAdapter(paypal);

        paypalProcessor.processPayment(250.00);

        // Using Stripe

        StripeGateway stripe = new StripeGateway();

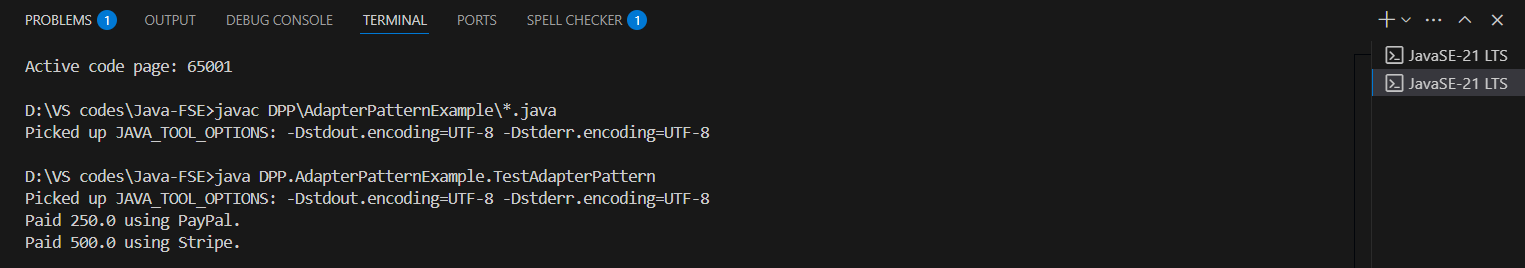
        PaymentProcessor stripeProcessor = new StripeAdapter(stripe);

        stripeProcessor.processPayment(500.00);

    }

}

**OUTPUT**

****

**Exercise 5: Implementing the Decorator Pattern**

**CODE**

**Notifier.java**

package DPP.DecoratorPatternExample;

public interface Notifier {

    void send(String message);

}

**EmailNotifier.java**

package DPP.DecoratorPatternExample;

public class EmailNotifier implements Notifier {

    @Override

    public void send(String message) {

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

    }

}

**NotifierDecorator.java**

package DPP.DecoratorPatternExample;

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);

    }

}

**SlackNotifierDecorator.java**

package DPP.DecoratorPatternExample;

public class SlackNotifierDecorator extends NotifierDecorator {

    public SlackNotifierDecorator(Notifier notifier) {

        super(notifier);

    }

    @Override

    public void send(String message) {

        super.send(message);

        sendSlack(message);

    }

    private void sendSlack(String message) {

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

    }

}

**SMSNotifierDecorator.java**

package DPP.DecoratorPatternExample;

public class SMSNotifierDecorator extends NotifierDecorator {

    public SMSNotifierDecorator(Notifier notifier) {

        super(notifier);

    }

    @Override

    public void send(String message) {

        super.send(message);

        sendSMS(message);

    }

    private void sendSMS(String message) {

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

    }

}

**TestDecoratorPattern.java**

package DPP.DecoratorPatternExample;

public class TestDecoratorPattern {

    public static void main(String[] args) {

        // Base notifier (only Email)

        Notifier emailOnly = new EmailNotifier();

        emailOnly.send("System Maintenance at 9 PM");

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

        // Email + SMS

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

        emailAndSMS.send("Your OTP is 7784");

        System.out.println("\n Email + SMS + Slack ");

        // Email + SMS + Slack

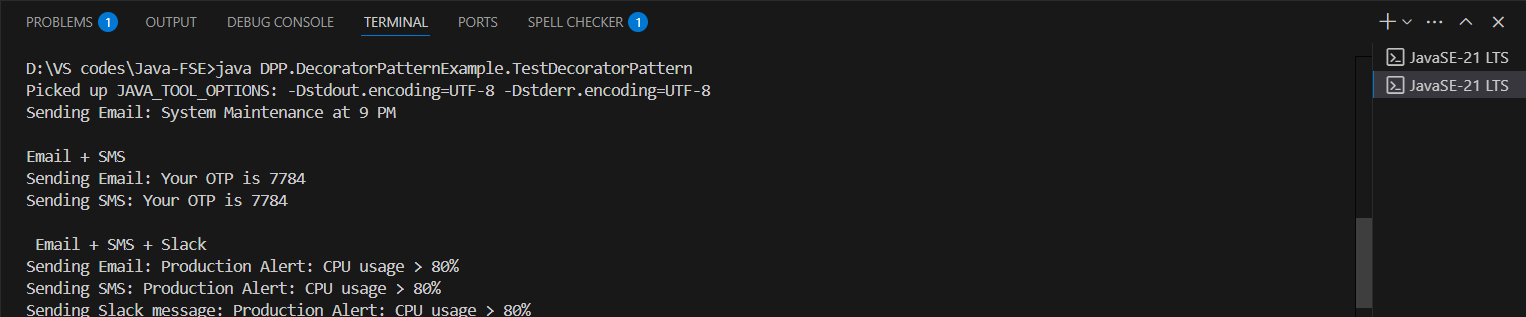
        Notifier allChannels = new SlackNotifierDecorator(new SMSNotifierDecorator(new EmailNotifier()));

        allChannels.send("Production Alert: CPU usage > 80%");

    }

}

**OUTPUT**



**Exercise 6: Implementing the Proxy Pattern**

**CODE**

**Image.java**

package DPP.ProxyPatternExample;

public interface Image {

    void display();

}

**ProxyImage.java**

package DPP.ProxyPatternExample;

public class ProxyImage implements Image {

    private String fileName;

    private RealImage realImage;

    public ProxyImage(String fileName) {

        this.fileName = fileName;

    }

    @Override

    public void display() {

        if (realImage == null) {

            realImage = new RealImage(fileName);

        } else {

            System.out.println("Image already loaded. Using cached image.");

        }

        realImage.display();

    }

}

**RealImage.java**

package DPP.ProxyPatternExample;

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);

    }

    @Override

    public void display() {

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

    }

}

**TestProxyPattern.java**

package DPP.ProxyPatternExample;

public class TestProxyPattern {

    public static void main(String[] args) {

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

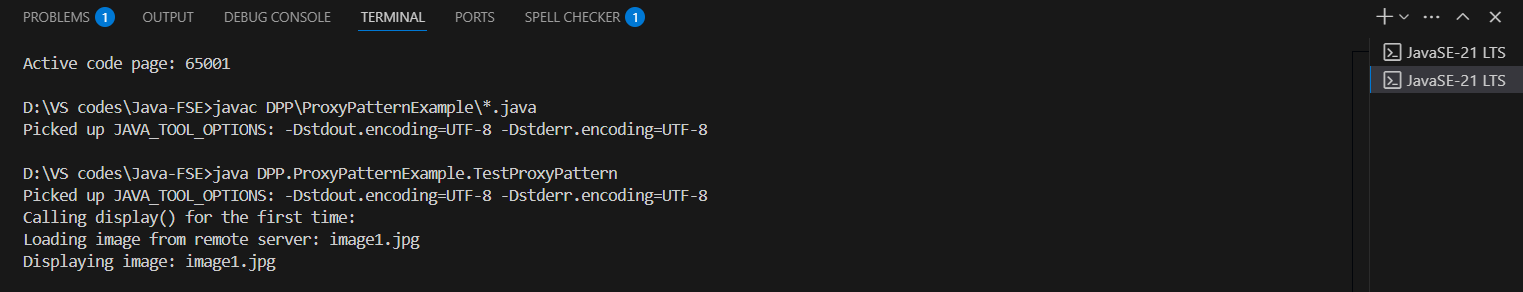
        System.out.println("Calling display() for the first time:");

        image1.display();  // Loads and displays

    }

}

**OUTPUT**



**Exercise 7: Implementing the Observer Pattern**

**CODE**

**Observer.java**

package DPP.ObserverPatternExample;

public interface Observer {

    void update(String stockName, double price);

}

**MobileApp.java**

package DPP.ObserverPatternExample;

public class MobileApp implements Observer {

    private String appName;

    public MobileApp(String appName) {

        this.appName = appName;

    }

    @Override

    public void update(String stockName, double price) {

        System.out.println(appName + " - Mobile App Notification: " + stockName + " is now Rs." + price);

    }

}

**WebApp.java**

package DPP.ObserverPatternExample;

public class WebApp implements Observer {

    private String siteName;

    public WebApp(String siteName) {

        this.siteName = siteName;

    }

    @Override

    public void update(String stockName, double price) {

        System.out.println(siteName + " - Web App Notification: " + stockName + " is now Rs." + price);

    }

}

**Stock.java**

package DPP.ObserverPatternExample;

public interface Stock {

    void registerObserver(Observer o);

    void removeObserver(Observer o);

    void notifyObservers();

}

**StockMarket.java**

package DPP.ObserverPatternExample;

import java.util.ArrayList;

import java.util.List;

public class StockMarket implements Stock {

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

    private String stockName;

    private double price;

    public void setStockPrice(String stockName, double price) {

        this.stockName = stockName;

        this.price = price;

        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, price);

        }

    }

}

**TestObserverPattern.java**

package DPP.ObserverPatternExample;

public class TestObserverPattern {

    public static void main(String[] args) {

        // Create StockMarket

        StockMarket market = new StockMarket();

        // Create observers

        Observer mobileApp1 = new MobileApp("InvestorPro");

        Observer webApp1 = new WebApp("StockWatch");

        // Register observers

        market.registerObserver(mobileApp1);

        market.registerObserver(webApp1);

        // Simulate stock price changes

        market.setStockPrice("Apple", 150.75);

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

        market.setStockPrice("GOOGLE", 2801.50);

        // Deregister one observer

        market.removeObserver(mobileApp1);

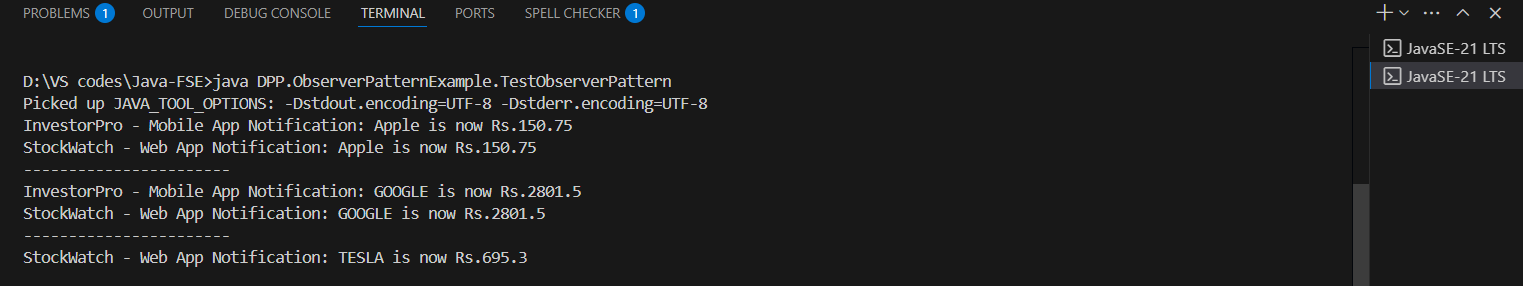
        System.out.println("-----------------------");

        market.setStockPrice("TESLA", 695.30);

    }

}

**OUTPUT**



**Exercise 8: Implementing the Strategy Pattern**

**CODE**

**PaymentStrategy.java**

package DPP.StrategyPatternExample;

public interface PaymentStrategy {

    void pay(double amount);

}

**PaymentContext.java**

package DPP.StrategyPatternExample;

public class PaymentContext {

    private PaymentStrategy strategy;

    public void setPaymentStrategy(PaymentStrategy strategy) {

        this.strategy = strategy;

    }

    public void executePayment(double amount) {

        if (strategy == null) {

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

        } else {

            strategy.pay(amount);

        }

    }

}

**CreditCardPayment.java**

package DPP.StrategyPatternExample;

public class CreditCardPayment implements PaymentStrategy {

    private String cardNumber;

    private String cardHolder;

    public CreditCardPayment(String cardNumber, String cardHolder) {

        this.cardNumber = cardNumber;

        this.cardHolder = cardHolder;

    }

    @Override

    public void pay(double amount) {

        System.out.println("Paid Rs." + amount + " using Credit Card (Holder: " + cardHolder + ")");

    }

}

**PayPalPayment.java**

package DPP.StrategyPatternExample;

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 Rs." + amount + " using PayPal (Account: " + email + ")");

    }

}

**TestStrategyPattern.java**

package DPP.StrategyPatternExample;

public class TestStrategyPattern {

    public static void main(String[] args) {

        PaymentContext context = new PaymentContext();

        //Credit Card Payment

        context.setPaymentStrategy(new CreditCardPayment("1234-5678-9012-3456", "John Doe"));

        context.executePayment(250.00);

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

        //PayPal Payment

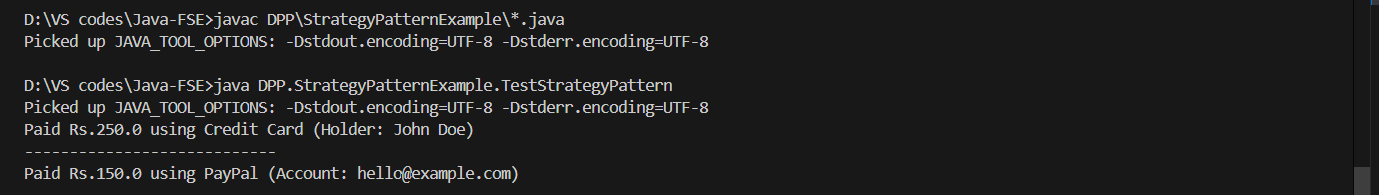
        context.setPaymentStrategy(new PayPalPayment("hello@example.com"));

        context.executePayment(150.00);

    }

}

**OUTPUT**



**Exercise 9: Implementing the Command Pattern**

**CODE**

**Command.java**

package DPP.CommandPatternExample;

public interface Command {

    void execute();

}

**Light.java**

package DPP.CommandPatternExample;

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**

package DPP.CommandPatternExample;

public class LightOffCommand implements Command {

    private Light light;

    public LightOffCommand(Light light) {

        this.light = light;

    }

    @Override

    public void execute() {

        light.turnOff();

    }

}

**LightOnCommand.java**

package DPP.CommandPatternExample;

public class LightOnCommand implements Command {

    private Light light;

    public LightOnCommand(Light light) {

        this.light = light;

    }

    @Override

    public void execute() {

        light.turnOn();

    }

}

**RemoteControl.java**

package DPP.CommandPatternExample;

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 assigned.");

        }

    }

}

**TestCommandPattern.java**

package DPP.CommandPatternExample;

public class TestCommandPattern {

    public static void main(String[] args) {

        // Receiver

        Light livingRoomLight = new Light();

        // Concrete Commands

        Command lightOn = new LightOnCommand(livingRoomLight);

        Command lightOff = new LightOffCommand(livingRoomLight);

        // Invoker

        RemoteControl remote = new RemoteControl();

        // Turn ON the light

        remote.setCommand(lightOn);

        remote.pressButton();

        // Turn OFF the light

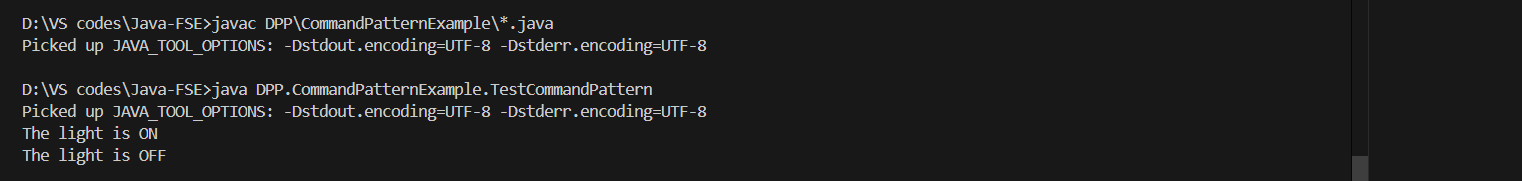
        remote.setCommand(lightOff);

        remote.pressButton();

    }

}

**OUTPUT**



**Exercise 10: Implementing the MVC Pattern**

**CODE**

**Student.java**

package DPP.MVCPatternExample;

public class Student {

    private String name;

    private String id;

    private String grade;

    // Constructor

    public Student(String name, String 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 String getId() { return id; }

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

    public String getGrade() { return grade; }

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

}

**StudentController.java**

package DPP.MVCPatternExample;

public class StudentController {

    private Student student;

    private StudentView view;

    public StudentController(Student student, StudentView view) {

        this.student = student;

        this.view = view;

    }

    // Setters

    public void setStudentName(String name) {

        student.setName(name);

    }

    public void setStudentId(String id) {

        student.setId(id);

    }

    public void setStudentGrade(String grade) {

        student.setGrade(grade);

    }

    // Getters

    public String getStudentName() {

        return student.getName();

    }

    public String getStudentId() {

        return student.getId();

    }

    public String getStudentGrade() {

        return student.getGrade();

    }

    // Update the view

    public void updateView() {

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

    }

}

**StudentView.java**

package DPP.MVCPatternExample;

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);

    }

}

**MVCTest.java**

package DPP.MVCPatternExample;

public class MVCTest {

    public static void main(String[] args) {

        // model object

        Student student = new Student("Lingoth", "S001", "A");

        // view object

        StudentView view = new StudentView();

        // controller object

        StudentController controller = new StudentController(student, view);

        // Initial display

        controller.updateView();

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

        // Updating model via controller

        controller.setStudentName("Lokkesh");

        controller.setStudentGrade("A+");

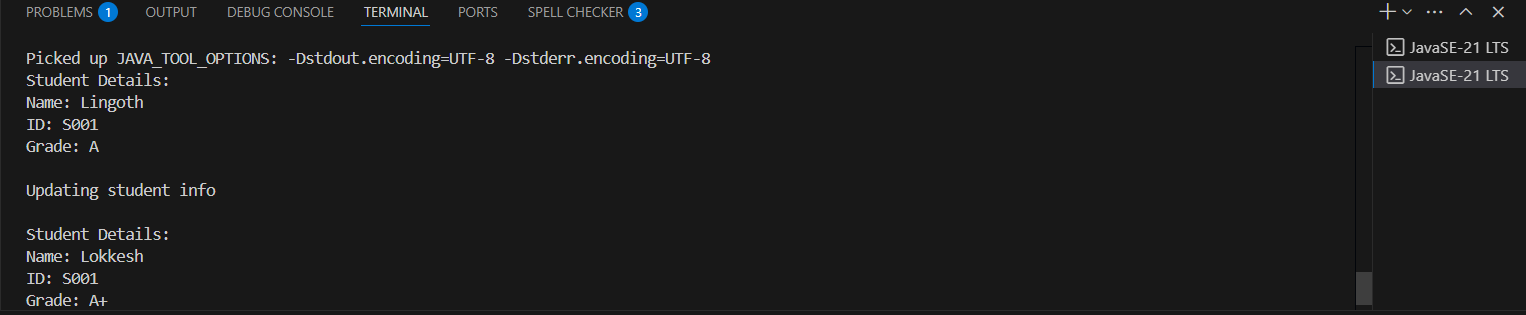
        // Updated display

        controller.updateView();

    }

}

**OUTPUT**



**Exercise 11: Implementing Dependency Injection**

**CODE**

**Customer.java**

package DPP.DependencyInjectionExample;

public class Customer {

    private String id;

    private String name;

    private String email;

    public Customer(String id, String name, String email) {

        this.id = id;

        this.name = name;

        this.email = email;

    }

    public void display() {

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

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

        System.out.println("Email     : " + email);

    }

}

**CustomerRepository.java**

package DPP.DependencyInjectionExample;

public interface CustomerRepository {

    Customer findCustomerById(String id);

}

**CustomerRepositoryImpl.java**

package DPP.DependencyInjectionExample;

import java.util.HashMap;

import java.util.Map;

public class CustomerRepositoryImpl implements CustomerRepository {

    private Map<String, Customer> customers = new HashMap<>();

    public CustomerRepositoryImpl() {

        // Sample customers

        customers.put("C001", new Customer("C001", "Lingoth", "lingoth@example.com"));

        customers.put("C002", new Customer("C002", "Lokkesh", "lokk@example.com"));

    }

    @Override

    public Customer findCustomerById(String id) {

        return customers.get(id);

    }

}

**CustomerService.java**

package DPP.DependencyInjectionExample;

public class CustomerService {

    private CustomerRepository customerRepository;

    // Constructor based Dependency Injection

    public CustomerService(CustomerRepository customerRepository) {

        this.customerRepository = customerRepository;

    }

    public void getCustomerDetails(String id) {

        Customer customer = customerRepository.findCustomerById(id);

        if (customer != null) {

            System.out.println("Customer found:");

            customer.display();

        } else {

            System.out.println("Customer not found for ID: " + id);

        }

    }

}

**TestDependencyInjection.java**

package DPP.DependencyInjectionExample;

public class TestDependencyInjection {

    public static void main(String[] args) {

        // Creating repository

        CustomerRepository repository = new CustomerRepositoryImpl();

        // Injecting repository into service

        CustomerService service = new CustomerService(repository);

        // Using the service

        service.getCustomerDetails("C001");

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

        service.getCustomerDetails("C999");

    }

}

**OUTPUT**

