**EXERCISES 1 TO 11(PATTERNS)**

**Exercise 1: Implementing the Singleton Pattern**

**CODE IMPLEMENTATION:**

 class Logger {

    private static Logger instance;

    private Logger() {}

    public static Logger getInstance() {

        if (instance == null) {

            instance = new Logger();

        }

        return instance;

    }

    public void log(String message) {

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

    }

}

public class SingletonPattern {

    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("Logger is a singleton, both references are the same instance.");

        } else {

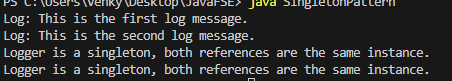
            System.out.println("Logger instances are different, Singleton pattern not implemented correctly.");

        }

    }

}

**OUTPUT:**

****

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

**CODE IMPLEMENTATION:**

interface Document {

    void open();

    void close();

    void save();

}

 class WordDocument implements Document {

    @Override

    public void open() {

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

    }

    @Override

    public void close() {

        System.out.println("Closing Word document.");

    }

    @Override

    public void save() {

        System.out.println("Saving Word document.");

    }

}

 class PdfDocument implements Document {

    @Override

    public void open() {

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

    }

    @Override

    public void close() {

        System.out.println("Closing PDF document.");

    }

    @Override

    public void save() {

        System.out.println("Saving PDF document.");

    }

}

 class ExcelDocument implements Document {

    @Override

    public void open() {

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

    }

    @Override

    public void close() {

        System.out.println("Closing Excel document.");

    }

    @Override

    public void save() {

        System.out.println("Saving Excel document.");

    }

}

 abstract class DocumentFactory {

    public abstract Document createDocument();

}

 class WordDocumentFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new WordDocument();

    }

}

 class PdfDocumentFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new PdfDocument();

    }

}

 class ExcelDocumentFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new ExcelDocument();

    }

}

public class FactoryMethodPattern {

    public static void main(String[] args) {

        DocumentFactory wordFactory = new WordDocumentFactory();

        Document wordDocument = wordFactory.createDocument();

        wordDocument.open();

        wordDocument.save();

        wordDocument.close();

        DocumentFactory pdfFactory = new PdfDocumentFactory();

        Document pdfDocument = pdfFactory.createDocument();

        pdfDocument.open();

        pdfDocument.save();

        pdfDocument.close();

        DocumentFactory excelFactory = new ExcelDocumentFactory();

        Document excelDocument = excelFactory.createDocument();

        excelDocument.open();

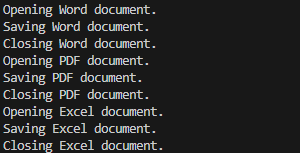
        excelDocument.save();

        excelDocument.close();

    }

}

**OUTPUT:**

****

**Exercise 3: Implementing the Builder Pattern**

**CODE IMPLEMENTATION :**

class Computer {

    private String CPU;

    private int RAM;

    private int storage;

    private String GPU;

    private String motherboard;

    private String powerSupply;

    private String coolingSystem;

    private Computer(Builder builder) {

        this.CPU = builder.CPU;

        this.RAM = builder.RAM;

        this.storage = builder.storage;

        this.GPU = builder.GPU;

        this.motherboard = builder.motherboard;

        this.powerSupply = builder.powerSupply;

        this.coolingSystem = builder.coolingSystem;

    }

    public String getCPU() {

        return CPU;

    }

    public int getRAM() {

        return RAM;

    }

    public int getStorage() {

        return storage;

    }

    public String getGPU() {

        return GPU;

    }

    public String getMotherboard() {

        return motherboard;

    }

    public String getPowerSupply() {

        return powerSupply;

    }

    public String getCoolingSystem() {

        return coolingSystem;

    }

    @Override

    public String toString() {

        return "Computer [CPU=" + CPU + ", RAM=" + RAM + "GB, storage=" + storage + "GB, GPU=" + GPU +

               ", motherboard=" + motherboard + ", powerSupply=" + powerSupply + ", coolingSystem=" + coolingSystem + "]";

    }

    public static class Builder {

        private String CPU;

        private int RAM;

        private int storage;

        private String GPU;

        private String motherboard;

        private String powerSupply;

        private String coolingSystem;

        public Builder setCPU(String CPU) {

            this.CPU = CPU;

            return this;

        }

        public Builder setRAM(int RAM) {

            this.RAM = RAM;

            return this;

        }

        public Builder setStorage(int storage) {

            this.storage = storage;

            return this;

        }

        public Builder setGPU(String GPU) {

            this.GPU = GPU;

            return this;

        }

        public Builder setMotherboard(String motherboard) {

            this.motherboard = motherboard;

            return this;

        }

        public Builder setPowerSupply(String powerSupply) {

            this.powerSupply = powerSupply;

            return this;

        }

        public Builder setCoolingSystem(String coolingSystem) {

            this.coolingSystem = coolingSystem;

            return this;

        }

        public Computer build() {

            return new Computer(this);

        }

    }

}

public class BuilderPattern {

    public static void main(String[] args) {

        Computer gamingComputer = new Computer.Builder()

                .setCPU("Intel i9")

                .setRAM(32)

                .setStorage(1000)

                .setGPU("NVIDIA RTX 3080")

                .setMotherboard("ASUS ROG STRIX")

                .setPowerSupply("750W")

                .setCoolingSystem("Liquid Cooling")

                .build();

        Computer officeComputer = new Computer.Builder()

                .setCPU("Intel i5")

                .setRAM(16)

                .setStorage(500)

                .setGPU("Integrated")

                .setMotherboard("ASUS Prime")

                .setPowerSupply("500W")

                .setCoolingSystem("Air Cooling")

                .build();

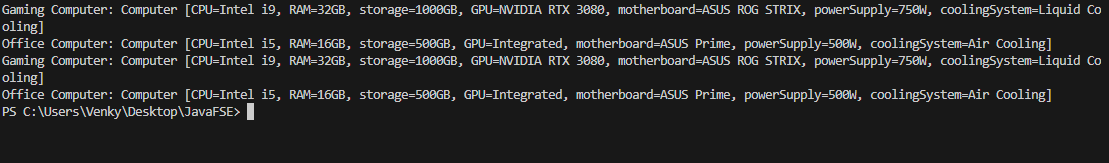
        System.out.println("Gaming Computer: " + gamingComputer);

        System.out.println("Office Computer: " + officeComputer);

    }

}

**OUTPUT:**

****

**Exercise 4: Implementing the Adapter Pattern**

**CODE IMPLEMENTATION:**

interface PaymentProcessor {

    void processPayment(double amount);

}

 class PayPalPayment {

    public void sendPayment(double amount) {

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

    }

}

 class StripePayment {

    public void makePayment(double amount) {

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

    }

}

 class SquarePayment {

    public void charge(double amount) {

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

    }

}

 class PayPalAdapter implements PaymentProcessor {

    private PayPalPayment payPalPayment;

    public PayPalAdapter(PayPalPayment payPalPayment) {

        this.payPalPayment = payPalPayment;

    }

    @Override

    public void processPayment(double amount) {

        payPalPayment.sendPayment(amount);

    }

}

 class StripeAdapter implements PaymentProcessor {

    private StripePayment stripePayment;

    public StripeAdapter(StripePayment stripePayment) {

        this.stripePayment = stripePayment;

    }

    @Override

    public void processPayment(double amount) {

        stripePayment.makePayment(amount);

    }

}

 class SquareAdapter implements PaymentProcessor {

    private SquarePayment squarePayment;

    public SquareAdapter(SquarePayment squarePayment) {

        this.squarePayment = squarePayment;

    }

    @Override

    public void processPayment(double amount) {

        squarePayment.charge(amount);

    }

}

public class AdapterPattern {

    public static void main(String[] args) {

        PaymentProcessor payPalProcessor = new PayPalAdapter(new PayPalPayment());

        payPalProcessor.processPayment(100.0);

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

        stripeProcessor.processPayment(200.0);

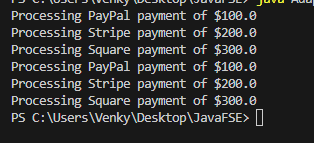
        PaymentProcessor squareProcessor = new SquareAdapter(new SquarePayment());

        squareProcessor.processPayment(300.0);

    }

}

**OUTPUT:**

****

**Exercise 5: Implementing the Decorator Pattern**

**CODE IMPLEMENTATION:**

interface Notifier {

    void send(String message);

}

 class EmailNotifier implements Notifier {

    @Override

    public void send(String message) {

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

    }

}

 abstract class NotifierDecorator implements Notifier {

    protected Notifier notifier;

    public NotifierDecorator(Notifier notifier) {

        this.notifier = notifier;

    }

    @Override

    public void send(String message) {

        notifier.send(message);

    }

}

 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 with message: " + message);

    }

}

 class SlackNotifierDecorator extends NotifierDecorator {

    public SlackNotifierDecorator(Notifier notifier) {

        super(notifier);

    }

    @Override

    public void send(String message) {

        super.send(message);

        sendSlackMessage(message);

    }

    private void sendSlackMessage(String message) {

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

    }

}

public class DecoratorPattern {

    public static void main(String[] args) {

        Notifier notifier = new EmailNotifier();

        notifier.send("Hello via Email!");

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

        notifier = new SMSNotifierDecorator(notifier);

        notifier.send("Hello via Email and SMS!");

        System.out.println("\nAdding Slack Notification:");

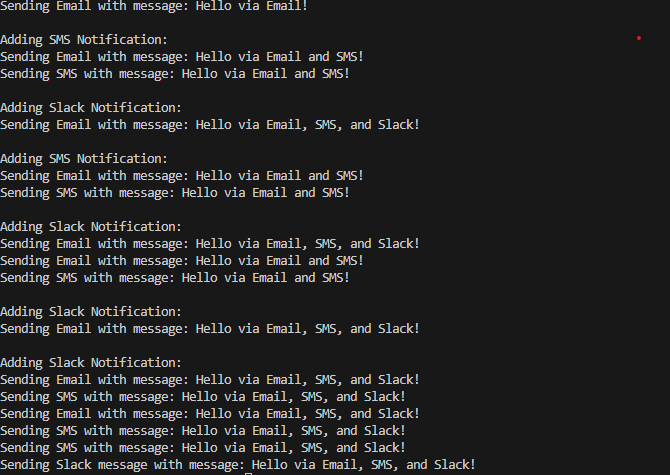
        notifier = new SlackNotifierDecorator(notifier);

        notifier.send("Hello via Email, SMS, and Slack!");

    }

}

**OUTPUT:**

****

**Exercise 6: Implementing the Proxy Pattern**

**CODE IMPLEMENTATION:**

interface Image {

    void display();

}

 class RealImage implements Image {

    private String filename;

    public RealImage(String filename) {

        this.filename = filename;

        loadImageFromDisk();

    }

    private void loadImageFromDisk() {

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

    }

    @Override

    public void display() {

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

    }

}

 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();

    }

}

public class ProxyPattern {

    public static void main(String[] args) {

        Image image1 = new ProxyImage("photo1.jpg");

        Image image2 = new ProxyImage("photo2.jpg");

        image1.display();

        System.out.println("");

        image1.display();

        System.out.println("");

        image2.display();

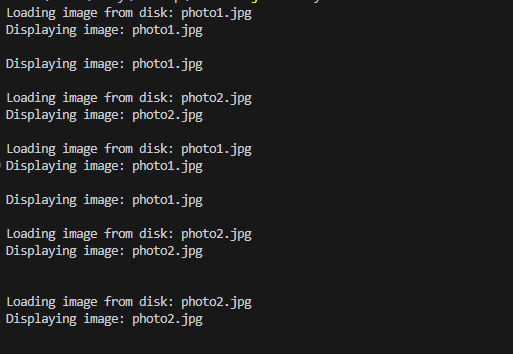
        System.out.println("");

        image2.display();

    }

}

**OUTPUT:**

****

**Exercise 7: Implementing the Observer Pattern**

**CODE IMPLEMENTATION:**

import java.util.ArrayList;

import java.util.List;

interface Stock {

    void registerObserver(Observer observer);

    void removeObserver(Observer observer);

    void notifyObservers();

}

 class StockMarket implements Stock {

    private List<Observer> observers;

    private double price;

    public StockMarket() {

        this.observers = new ArrayList<>();

    }

    @Override

    public void registerObserver(Observer observer) {

        observers.add(observer);

    }

    @Override

    public void removeObserver(Observer observer) {

        observers.remove(observer);

    }

    @Override

    public void notifyObservers() {

        for (Observer observer : observers) {

            observer.update(price);

        }

    }

    public void setPrice(double price) {

        this.price = price;

        notifyObservers();

    }

}

 interface Observer {

    void update(double price);

}

class MobileApp implements Observer {

    private String appName;

    public MobileApp(String appName) {

        this.appName = appName;

    }

    @Override

    public void update(double price) {

        System.out.println(appName + " received stock price update: " + price);

    }

}

 class WebApp implements Observer {

    private String appName;

    public WebApp(String appName) {

        this.appName = appName;

    }

    @Override

    public void update(double price) {

        System.out.println(appName + " received stock price update: " + price);

    }

}

 class ObserverPattern {

    public static void main(String[] args) {

        StockMarket stockMarket = new StockMarket();

        Observer mobileApp1 = new MobileApp("MobileApp1");

        Observer mobileApp2 = new MobileApp("MobileApp2");

        Observer webApp = new WebApp("WebApp");

        stockMarket.registerObserver(mobileApp1);

        stockMarket.registerObserver(mobileApp2);

        stockMarket.registerObserver(webApp);

        System.out.println("Setting stock price to 100.0");

        stockMarket.setPrice(100.0);

        System.out.println("\nSetting stock price to 200.0");

        stockMarket.setPrice(200.0);

        stockMarket.removeObserver(mobileApp2);

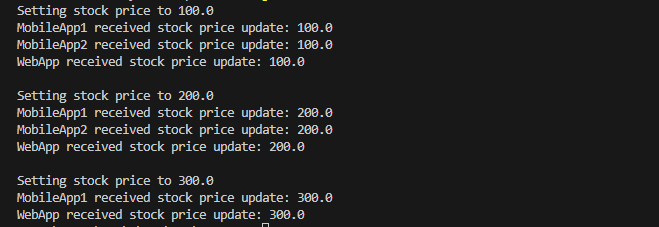
        System.out.println("\nSetting stock price to 300.0");

        stockMarket.setPrice(300.0);

    }

}

**OUTPUT:**

****

**Exercise 8: Implementing the Strategy Pattern**

**CODE IMPLEMENTATION:**

interface PaymentStrategy {

    void pay(double amount);

}

 class CreditCardPayment implements PaymentStrategy {

    private String cardNumber;

    private String cardHolderName;

    private String cvv;

    private String expirationDate;

    public CreditCardPayment(String cardNumber, String cardHolderName, String cvv, String expirationDate) {

        this.cardNumber = cardNumber;

        this.cardHolderName = cardHolderName;

        this.cvv = cvv;

        this.expirationDate = expirationDate;

    }

    @Override

    public void pay(double amount) {

        System.out.println(amount + " paid with credit card.");

    }

}

 class PayPalPayment implements PaymentStrategy {

    private String email;

    private String password;

    public PayPalPayment(String email, String password) {

        this.email = email;

        this.password = password;

    }

    @Override

    public void pay(double amount) {

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

    }

}

 class PaymentContext {

    private PaymentStrategy paymentStrategy;

    public void setPaymentStrategy(PaymentStrategy paymentStrategy) {

        this.paymentStrategy = paymentStrategy;

    }

    public void pay(double amount) {

        paymentStrategy.pay(amount);

    }

}

public class StrategyPattern {

    public static void main(String[] args) {

        PaymentContext context = new PaymentContext();

        PaymentStrategy creditCard = new CreditCardPayment("1234567890123456", "John Doe", "123", "12/23");

        context.setPaymentStrategy(creditCard);

        context.pay(100.00);

        PaymentStrategy payPal = new PayPalPayment("john.doe@example.com", "password");

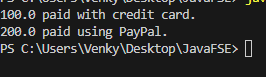
        context.setPaymentStrategy(payPal);

        context.pay(200.00);

    }

}

**OUTPUT:**

****

**Exercise 9: Implementing the Command Pattern**

**CODE IMPLEMENTATION:**

interface Command {

    void execute();

}

 class LightOnCommand implements Command {

    private Light light;

    public LightOnCommand(Light light) {

        this.light = light;

    }

    @Override

    public void execute() {

        light.turnOn();

    }

}

 class LightOffCommand implements Command {

    private Light light;

    public LightOffCommand(Light light) {

        this.light = light;

    }

    @Override

    public void execute() {

        light.turnOff();

    }

}

 class RemoteControl {

    private Command command;

    public void setCommand(Command command) {

        this.command = command;

    }

    public void pressButton() {

        command.execute();

    }

}

 class Light {

    private String location;

    public Light(String location) {

        this.location = location;

    }

    public void turnOn() {

        System.out.println(location + " light is ON");

    }

    public void turnOff() {

        System.out.println(location + " light is OFF");

    }

}

public class CommandPattern {

    public static void main(String[] args) {

        Light livingRoomLight = new Light("Living Room");

        Light kitchenLight = new Light("Kitchen");

        Command livingRoomLightOn = new LightOnCommand(livingRoomLight);

        Command livingRoomLightOff = new LightOffCommand(livingRoomLight);

        Command kitchenLightOn = new LightOnCommand(kitchenLight);

        Command kitchenLightOff = new LightOffCommand(kitchenLight);

        RemoteControl remote = new RemoteControl();

        remote.setCommand(livingRoomLightOn);

        remote.pressButton();

        remote.setCommand(livingRoomLightOff);

        remote.pressButton();

        remote.setCommand(kitchenLightOn);

        remote.pressButton();

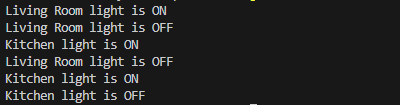
        remote.setCommand(kitchenLightOff);

        remote.pressButton();

    }

}

**OUTPUT:**

****

**Exercise 10: Implementing the MVC Pattern**

**CODE IMPLEMENTATION:**

class Student {

    private String id;

    private String name;

    private String grade;

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

        this.id = id;

        this.name = name;

        this.grade = 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;

    }

}

 class StudentView {

    public void displayStudentDetails(String studentName, String studentId, String studentGrade) {

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

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

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

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

    }

}

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 String getStudentName() {

        return model.getName();

    }

    public void setStudentId(String id) {

        model.setId(id);

    }

    public String getStudentId() {

        return model.getId();

    }

    public void setStudentGrade(String grade) {

        model.setGrade(grade);

    }

    public String getStudentGrade() {

        return model.getGrade();

    }

    public void updateView() {

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

    }

}

public class MVCPattern {

    public static void main(String[] args) {

        Student model = new Student("1", "John Doe", "A");

        StudentView view = new StudentView();

        StudentController controller = new StudentController(model, view);

        controller.updateView();

        controller.setStudentName("Jane Doe");

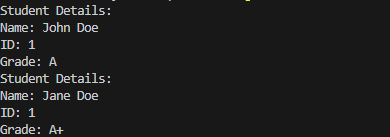
        controller.setStudentGrade("A+");

        controller.updateView();

    }

}

**OUTPUT:**

****

**Exercise 11: Implementing Dependency Injection**

**CODE IMPLEMENTATION:**

interface CustomerRepository {

    Customer findCustomerById(String customerId);

}

 class CustomerRepositoryImpl implements CustomerRepository {

    @Override

    public Customer findCustomerById(String customerId) {

        return new Customer(customerId, "John Doe");

    }

}

 class Customer {

    private String id;

    private String name;

    public Customer(String id, String name) {

        this.id = id;

        this.name = name;

    }

    public String getId() {

        return id;

    }

    public String getName() {

        return name;

    }

}

 class CustomerService {

    private final CustomerRepository customerRepository;

    public CustomerService(CustomerRepository customerRepository) {

        this.customerRepository = customerRepository;

    }

    public Customer getCustomerById(String customerId) {

        return customerRepository.findCustomerById(customerId);

    }

}

public class DependencyInjection {

    public static void main(String[] args) {

        CustomerRepository customerRepository = new CustomerRepositoryImpl();

        CustomerService customerService = new CustomerService(customerRepository);

        Customer customer = customerService.getCustomerById("1");

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

        System.out.println("Customer Name: " + customer.getName());

    }

}

**OUTPUT:**

****