**Exercise 1: Implementing the Singleton Pattern**

**Scenario:** You need to ensure that a logging utility class in your application has only one instance throughout the application lifecycle to ensure consistent logging.

public class SingletonPatternExample {

    static class Logger {

        private static Logger instance;

        private Logger() {

            // Private constructor to prevent instantiation

        }

        public static synchronized Logger getInstance() {

            if (instance == null) {

                instance = new Logger();

            }

            return instance;

        }

        public void log(String message) {

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

        }

    }

    public static class SingletonTest {

        public static void main(String[] args) {

            Logger logger1 = Logger.getInstance();

            Logger logger2 = Logger.getInstance();

            if (logger1 == logger2) {

                System.out.println("Both logger instances are the same.");

            } else {

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

            }

            logger1.log("This is a test log message.");

            logger2.log("This message should come from the same logger instance.");

        }

    }

}

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

**Scenario:** You are developing a document management system that needs to create different types of documents (e.g., Word, PDF, Excel). Use the Factory Method Pattern to achieve this.

public class FactoryMethodPatternExample {

    // Step 2: Define Document Interface

    interface Document {

        void open();

    }

    // Step 3: Create Concrete Document Classes

    static class WordDocument implements Document {

        @Override

        public void open() {

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

        }

    }

    static class PdfDocument implements Document {

        @Override

        public void open() {

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

        }

    }

    static class ExcelDocument implements Document {

        @Override

        public void open() {

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

        }

    }

    // Step 4: Implement the Factory Method

    abstract static class DocumentFactory {

        // Factory method

        public abstract Document createDocument();

    }

    // Concrete factories

    static class WordDocumentFactory extends DocumentFactory {

        @Override

        public Document createDocument() {

            return new WordDocument();

        }

    }

    static class PdfDocumentFactory extends DocumentFactory {

        @Override

        public Document createDocument() {

            return new PdfDocument();

        }

    }

    static class ExcelDocumentFactory extends DocumentFactory {

        @Override

        public Document createDocument() {

            return new ExcelDocument();

        }

    }

    // Step 5: Test the Factory Method Implementation

    public static class FactoryTest {

        public static void main(String[] args) {

            // Create a Word document using the factory

            DocumentFactory wordFactory = new WordDocumentFactory();

            Document wordDocument = wordFactory.createDocument();

            wordDocument.open();

            // Create a PDF document using the factory

            DocumentFactory pdfFactory = new PdfDocumentFactory();

            Document pdfDocument = pdfFactory.createDocument();

            pdfDocument.open();

            // Create an Excel document using the factory

            DocumentFactory excelFactory = new ExcelDocumentFactory();

            Document excelDocument = excelFactory.createDocument();

            excelDocument.open();

        }

    }

}

**Exercise 3: Implementing the Builder Pattern**

**Scenario:** You are developing a system to create complex objects such as a Computer with multiple optional parts. Use the Builder Pattern to manage the construction process.

public class BuilderPatternExample {

    static class Computer {

        private final String CPU;

        private final String RAM;

        private final String storage;

        private final boolean hasGraphicsCard;

        private final boolean hasWiFi;

        private Computer(Builder builder) {

            this.CPU = builder.CPU;

            this.RAM = builder.RAM;

            this.storage = builder.storage;

            this.hasGraphicsCard = builder.hasGraphicsCard;

            this.hasWiFi = builder.hasWiFi;

        }

        public String getCPU() {

            return CPU;

        }

        public String getRAM() {

            return RAM;

        }

        public String getStorage() {

            return storage;

        }

        public boolean hasGraphicsCard() {

            return hasGraphicsCard;

        }

        public boolean hasWiFi() {

            return hasWiFi;

        }

        @Override

        public String toString() {

            return "Computer{" +

                    "CPU='" + CPU + '\'' +

                    ", RAM='" + RAM + '\'' +

                    ", Storage='" + storage + '\'' +

                    ", Has Graphics Card=" + hasGraphicsCard +

                    ", Has WiFi=" + hasWiFi +

                    '}';

        }

        public static class Builder {

            private String CPU;

            private String RAM;

            private String storage;

            private boolean hasGraphicsCard;

            private boolean hasWiFi;

            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(boolean hasGraphicsCard) {

                this.hasGraphicsCard = hasGraphicsCard;

                return this;

            }

            public Builder setWiFi(boolean hasWiFi) {

                this.hasWiFi = hasWiFi;

                return this;

            }

            public Computer build() {

                return new Computer(this);

            }

        }

    }

    public static class BuilderTest {

        public static void main(String[] args) {

            Computer gamingPC = new Computer.Builder()

                    .setCPU("Intel Core i9")

                    .setRAM("32GB")

                    .setStorage("1TB SSD")

                    .setGraphicsCard(true)

                    .setWiFi(true)

                    .build();

            Computer officePC = new Computer.Builder()

                    .setCPU("Intel Core i5")

                    .setRAM("16GB")

                    .setStorage("512GB SSD")

                    .setGraphicsCard(false)

                    .setWiFi(true)

                    .build();

            System.out.println("Gaming PC Configuration: " + gamingPC);

            System.out.println("Office PC Configuration: " + officePC);

        }

    }

}

**Exercise 4: Implementing the Adapter Pattern**

**Scenario:** You are developing a payment processing system that needs to integrate with multiple third-party payment gateways with different interfaces. Use the Adapter Pattern to achieve this.

// AdapterPatternExample.java

public class AdapterPatternExample {

    interface PaymentProcessor {

        void processPayment(double amount);

    }

    static class PayPal {

        void payWithPayPal(double amount) {

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

        }

    }

    static class Stripe {

        void charge(double amount) {

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

        }

    }

    static class PayPalAdapter implements PaymentProcessor {

        private final PayPal payPal;

        public PayPalAdapter(PayPal payPal) {

            this.payPal = payPal;

        }

        @Override

        public void processPayment(double amount) {

            payPal.payWithPayPal(amount);

        }

    }

    static class StripeAdapter implements PaymentProcessor {

        private final Stripe stripe;

        public StripeAdapter(Stripe stripe) {

            this.stripe = stripe;

        }

        @Override

        public void processPayment(double amount) {

            stripe.charge(amount);

        }

    }

    public static class AdapterTest {

        public static void main(String[] args) {

            PayPal payPal = new PayPal();

            Stripe stripe = new Stripe();

            PaymentProcessor payPalAdapter = new PayPalAdapter(payPal);

            PaymentProcessor stripeAdapter = new StripeAdapter(stripe);

            payPalAdapter.processPayment(100.0);

            stripeAdapter.processPayment(200.0);

        }

    }

}

**Exercise 5: Implementing the Decorator Pattern**

**Scenario:** You are developing a notification system where notifications can be sent via multiple channels (e.g., Email, SMS). Use the Decorator Pattern to add functionalities dynamically

// DecoratorPatternExample.java

public class DecoratorPatternExample {

    interface Notifier {

        void send(String message);

    }

    static class EmailNotifier implements Notifier {

        @Override

        public void send(String message) {

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

        }

    }

    abstract static class NotifierDecorator implements Notifier {

        protected final Notifier notifier;

        public NotifierDecorator(Notifier notifier) {

            this.notifier = notifier;

        }

        @Override

        public void send(String message) {

            notifier.send(message);

        }

    }

    static class SMSNotifierDecorator extends NotifierDecorator {

        public SMSNotifierDecorator(Notifier notifier) {

            super(notifier);

        }

        @Override

        public void send(String message) {

            super.send(message);

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

        }

    }

    static class SlackNotifierDecorator extends NotifierDecorator {

        public SlackNotifierDecorator(Notifier notifier) {

            super(notifier);

        }

        @Override

        public void send(String message) {

            super.send(message);

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

        }

    }

    public static class DecoratorTest {

        public static void main(String[] args) {

            Notifier emailNotifier = new EmailNotifier();

            Notifier smsEmailNotifier = new SMSNotifierDecorator(emailNotifier);

            Notifier slackSMSEmailNotifier = new SlackNotifierDecorator(smsEmailNotifier);

            slackSMSEmailNotifier.send("Hello, world!");

        }

    }

}

**Exercise 6: Implementing the Proxy Pattern**

**Scenario:** You are developing an image viewer application that loads images from a remote server. Use the Proxy Pattern to add lazy initialization and caching

// ProxyPatternExample.java

public class ProxyPatternExample {

    interface Image {

        void display();

    }

    static class RealImage implements Image {

        private final String filename;

        public RealImage(String filename) {

            this.filename = filename;

            loadFromServer();

        }

        private void loadFromServer() {

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

        }

        @Override

        public void display() {

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

        }

    }

    static class ProxyImage implements Image {

        private RealImage realImage;

        private final String filename;

        public ProxyImage(String filename) {

            this.filename = filename;

        }

        @Override

        public void display() {

            if (realImage == null) {

                realImage = new RealImage(filename);

            }

            realImage.display();

        }

    }

    public static class ProxyTest {

        public static void main(String[] args) {

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

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

            image1.display();

            image1.display();

            image2.display();

            image2.display();

        }

    }

}

**Exercise 7: Implementing the Observer Pattern**

**Scenario:** You are developing a stock market monitoring application where multiple clients need to be notified whenever stock prices change. Use the Observer Pattern to achieve this.

import java.util.ArrayList;

import java.util.List;

// ObserverPatternExample.java

import java.util.ArrayList;

import java.util.List;

interface Stock {

    void registerObserver(Observer observer);

    void deregisterObserver(Observer observer);

    void notifyObservers();

}

class StockMarket implements Stock {

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

    private double price;

    public void setPrice(double price) {

        this.price = price;

        notifyObservers();

    }

    public double getPrice() {

        return price;

    }

    @Override

    public void registerObserver(Observer observer) {

        observers.add(observer);

    }

    @Override

    public void deregisterObserver(Observer observer) {

        observers.remove(observer);

    }

    @Override

    public void notifyObservers() {

        for (Observer observer : observers) {

            observer.update();

        }

    }

}

interface Observer {

    void update();

}

class MobileApp implements Observer {

    private final StockMarket stockMarket;

    public MobileApp(StockMarket stockMarket) {

        this.stockMarket = stockMarket;

        this.stockMarket.registerObserver(this);

    }

    @Override

    public void update() {

        System.out.println("Mobile App Notification: Stock price updated to $" + stockMarket.getPrice());

    }

}

class WebApp implements Observer {

    private final StockMarket stockMarket;

    public WebApp(StockMarket stockMarket) {

        this.stockMarket = stockMarket;

        this.stockMarket.registerObserver(this);

    }

    @Override

    public void update() {

        System.out.println("Web App Notification: Stock price updated to $" + stockMarket.getPrice());

    }

}

public class ObserverPatternExample {

    public static void main(String[] args) {

        StockMarket stockMarket = new StockMarket();

        MobileApp mobileApp = new MobileApp(stockMarket);

        WebApp webApp = new WebApp(stockMarket);

        stockMarket.setPrice(100.0);

        stockMarket.setPrice(105.5);

        stockMarket.deregisterObserver(mobileApp);

        stockMarket.setPrice(110.0);

    }

}

**Exercise 8: Implementing the Strategy Pattern**

**Scenario:** You are developing a payment system where different payment methods (e.g., Credit Card, PayPal) can be selected at runtime. Use the Strategy Pattern to achieve this.

// StrategyPatternExample.java

public class StrategyPatternExample {

    interface PaymentStrategy {

        void pay(double amount);

    }

    static class CreditCardPayment implements PaymentStrategy {

        private final String cardNumber;

        private final String cardHolder;

        public CreditCardPayment(String cardNumber, String cardHolder) {

            this.cardNumber = cardNumber;

            this.cardHolder = cardHolder;

        }

        @Override

        public void pay(double amount) {

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

            System.out.println("Card Number: " + cardNumber);

            System.out.println("Card Holder: " + cardHolder);

        }

    }

    static class PayPalPayment implements PaymentStrategy {

        private final String email;

        public PayPalPayment(String email) {

            this.email = email;

        }

        @Override

        public void pay(double amount) {

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

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

        }

    }

    static class PaymentContext {

        private PaymentStrategy paymentStrategy;

        public void setPaymentStrategy(PaymentStrategy paymentStrategy) {

            this.paymentStrategy = paymentStrategy;

        }

        public void executePayment(double amount) {

            paymentStrategy.pay(amount);

        }

    }

    public static class StrategyTest {

        public static void main(String[] args) {

            PaymentContext paymentContext = new PaymentContext();

            PaymentStrategy creditCard = new CreditCardPayment("1234-5678-9876-5432", "John Doe");

            paymentContext.setPaymentStrategy(creditCard);

            paymentContext.executePayment(100.0);

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

            paymentContext.setPaymentStrategy(payPal);

            paymentContext.executePayment(200.0);

        }

    }

}

**Exercise 9: Implementing the Command Pattern**

**Scenario:** You are developing a home automation system where commands can be issued to turn devices on or off. Use the Command Pattern to achieve this.

// CommandPatternExample.java

public class CommandPatternExample {

    interface Command {

        void execute();

    }

    static class LightOnCommand implements Command {

        private final Light light;

        public LightOnCommand(Light light) {

            this.light = light;

        }

        @Override

        public void execute() {

            light.turnOn();

        }

    }

    static class LightOffCommand implements Command {

        private final Light light;

        public LightOffCommand(Light light) {

            this.light = light;

        }

        @Override

        public void execute() {

            light.turnOff();

        }

    }

    static class Light {

        public void turnOn() {

            System.out.println("Light is ON");

        }

        public void turnOff() {

            System.out.println("Light is OFF");

        }

    }

    static class RemoteControl {

        private Command command;

        public void setCommand(Command command) {

            this.command = command;

        }

        public void pressButton() {

            command.execute();

        }

    }

    public static class CommandTest {

        public static void main(String[] args) {

            Light livingRoomLight = new Light();

            Command lightOn = new LightOnCommand(livingRoomLight);

            Command lightOff = new LightOffCommand(livingRoomLight);

            RemoteControl remoteControl = new RemoteControl();

            remoteControl.setCommand(lightOn);

            remoteControl.pressButton();

            remoteControl.setCommand(lightOff);

            remoteControl.pressButton();

        }

    }

}

**Exercise 10: Implementing the MVC Pattern**

**Scenario:** You are developing a simple web application for managing student records using the MVC pattern.

// Model Class

public class MVCPatternExample{

    static 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;

        }

    }

    static class StudentView {

        public void displayStudentDetails(Student student) {

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

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

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

            System.out.println("Grade: " + student.getGrade());

        }

    }

    static class StudentController {

        private final Student model;

        private final 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(int id) {

            model.setId(id);

        }

        public int getStudentId() {

            return model.getId();

        }

        public void setStudentGrade(String grade) {

            model.setGrade(grade);

        }

        public String getStudentGrade() {

            return model.getGrade();

        }

        public void updateView() {

            view.displayStudentDetails(model);

        }

    }

    public static class MVCTest {

        public static void main(String[] args) {

            Student student = new Student("John Doe", 123, "A");

            StudentView view = new StudentView();

            StudentController controller = new StudentController(student, view);

            controller.updateView();

            controller.setStudentName("Jane Doe");

            controller.setStudentId(456);

            controller.setStudentGrade("B");

            controller.updateView();

        }

    }

}

**Exercise 11: Implementing Dependency Injection**

**Scenario:** You are developing a customer management application where the service class depends on a repository class. Use Dependency Injection to manage these dependencies.

public class DependencyInjectionExample {

    interface CustomerRepository {

        String findCustomerById(int id);

    }

    static class CustomerRepositoryImpl implements CustomerRepository {

        @Override

        public String findCustomerById(int id) {

            if (id == 1) {

                return "John Doe";

            } else if (id == 2) {

                return "Jane Smith";

            } else {

                return "Customer not found";

            }

        }

    }

    static class CustomerService {

        private final CustomerRepository customerRepository;

        public CustomerService(CustomerRepository customerRepository) {

            this.customerRepository = customerRepository;

        }

        public String getCustomerName(int id) {

            return customerRepository.findCustomerById(id);

        }

    }

    public static class DependencyInjectionTest {

        public static void main(String[] args) {

            CustomerRepository customerRepository = new CustomerRepositoryImpl();

            CustomerService customerService = new CustomerService(customerRepository);

            System.out.println("Customer with ID 1: " + customerService.getCustomerName(1));

            System.out.println("Customer with ID 2: " + customerService.getCustomerName(2));

            System.out.println("Customer with ID 3: " + customerService.getCustomerName(3));

        }

    }

}