**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.

**Code:**

// Singleton class for logging utility

class LogManager {

    // Single instance of LogManager

    private static LogManager uniqueInstance;

    // Private constructor to prevent instantiation from outside

    private LogManager() {

        // Initialization code for the logger

    }

    // Method to provide access to the single instance of LogManager

    public static LogManager getInstance() {

        if (uniqueInstance == null) {

            uniqueInstance = new LogManager();

        }

        return uniqueInstance;

    }

    // Method to log messages

    public void record(String message) {

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

    }

}

// Test class to verify Singleton pattern implementation

class SingletonDemo {

    public static void main(String[] args) {

        // Fetching the single instance of LogManager

        LogManager logger1 = LogManager.getInstance();

        LogManager logger2 = LogManager.getInstance();

        // Logging messages

        logger1.record("This is the first log entry.");

        logger2.record("This is the second log entry.");

        // Verifying that both logger1 and logger2 refer to the same instance

        if (logger1 == logger2) {

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

        } else {

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

        }

    }

}

**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.

// Interface representing a document with basic operations

interface FileDocument {

    void open();

    void close();

}

// Implementation for handling Word documents

class WordFile implements FileDocument {

    @Override

    public void open() {

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

    }

    @Override

    public void close() {

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

    }

}

// Implementation for handling PDF documents

class PdfFile implements FileDocument {

    @Override

    public void open() {

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

    }

    @Override

    public void close() {

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

    }

}

// Implementation for handling Excel documents

class ExcelFile implements FileDocument {

    @Override

    public void open() {

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

    }

    @Override

    public void close() {

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

    }

}

// Abstract factory class for creating document objects

abstract class DocumentCreator {

    public abstract FileDocument createFileDocument();

}

// Concrete factory for creating Word document instances

class WordFileFactory extends DocumentCreator {

    @Override

    public FileDocument createFileDocument() {

        return new WordFile();

    }

}

// Concrete factory for creating PDF document instances

class PdfFileFactory extends DocumentCreator {

    @Override

    public FileDocument createFileDocument() {

        return new PdfFile();

    }

}

// Concrete factory for creating Excel document instances

class ExcelFileFactory extends DocumentCreator {

    @Override

    public FileDocument createFileDocument() {

        return new ExcelFile();

    }

}

// Demonstration of the Factory Method Pattern

class FactoryMethodPatternDemo {

    public static void main(String[] args) {

        // Create and use a Word document

        DocumentCreator wordFactory = new WordFileFactory();

        FileDocument wordFile = wordFactory.createFileDocument();

        wordFile.open();

        wordFile.close();

        // Create and use a PDF document

        DocumentCreator pdfFactory = new PdfFileFactory();

        FileDocument pdfFile = pdfFactory.createFileDocument();

        pdfFile.open();

        pdfFile.close();

        // Create and use an Excel document

        DocumentCreator excelFactory = new ExcelFileFactory();

        FileDocument excelFile = excelFactory.createFileDocument();

        excelFile.open();

        excelFile.close();

    }

}

**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.

**Code:**

class BuilderPatternDemo {

    // Represents a Computer with various hardware components

    static class Computer {

        private String processor;

        private String memory;

        private String diskSpace;

        // Private constructor to ensure object creation is controlled via Builder

        private Computer(ComputerBuilder builder) {

            this.processor = builder.processor;

            this.memory = builder.memory;

            this.diskSpace = builder.diskSpace;

        }

        // Static nested Builder class for constructing Computer objects

        public static class ComputerBuilder {

            private String processor;

            private String memory;

            private String diskSpace;

            // Sets the processor type and returns the Builder for chaining

            public ComputerBuilder setProcessor(String processor) {

                this.processor = processor;

                return this;

            }

            // Sets the amount of memory and returns the Builder for chaining

            public ComputerBuilder setMemory(String memory) {

                this.memory = memory;

                return this;

            }

            // Sets the disk space and returns the Builder for chaining

            public ComputerBuilder setDiskSpace(String diskSpace) {

                this.diskSpace = diskSpace;

                return this;

            }

            // Builds and returns a Computer instance

            public Computer build() {

                return new Computer(this);

            }

        }

    }

    // Demonstrates the use of the Builder pattern to create a Computer object

static void main(String[] args) {

        // Construct a high-performance gaming PC using the Builder

        Computer gamingPC = new Computer.ComputerBuilder()

                .setProcessor("Intel Core i9")

                .setMemory("32GB")

                .setDiskSpace("1TB SSD")

                .build();

        // Output the specifications of the gaming PC

        System.out.println("Processor: " + gamingPC.processor);

        System.out.println("Memory: " + gamingPC.memory);

        System.out.println("Disk Space: " + gamingPC.diskSpace);

    }

}

**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.

interface PaymentGateway {

    void handlePayment(double amount);

}

class PayPalService {

    public void executePayment(double amount) {

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

    }

}

class StripeService {

    public void processCharge(double amount) {

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

    }

}

class AmazonPayService {

    public void performTransaction(double amount) {

        System.out.println("Processing payment of Rs." + amount + " through Amazon Pay.");

    }

}

class PayPalAdapter implements PaymentGateway {

    private PayPalService payPalService;

    public PayPalAdapter(PayPalService payPalService) {

        this.payPalService = payPalService;

    }

    @Override

    public void handlePayment(double amount) {

        payPalService.executePayment(amount);

    }

}

class StripeAdapter implements PaymentGateway {

    private StripeService stripeService;

    public StripeAdapter(StripeService stripeService) {

        this.stripeService = stripeService;

    }

    @Override

    public void handlePayment(double amount) {

        stripeService.processCharge(amount);

    }

}

class AmazonPayAdapter implements PaymentGateway {

    private AmazonPayService amazonPayService;

    public AmazonPayAdapter(AmazonPayService amazonPayService) {

        this.amazonPayService = amazonPayService;

    }

    @Override

    public void handlePayment(double amount) {

        amazonPayService.performTransaction(amount);

    }

}

class AdapterPatternDemo {

    public static void main(String[] args) {

        PayPalService payPalService = new PayPalService();

        StripeService stripeService = new StripeService();

        AmazonPayService amazonPayService = new AmazonPayService();

        PaymentGateway payPalAdapter = new PayPalAdapter(payPalService);

        PaymentGateway stripeAdapter = new StripeAdapter(stripeService);

        PaymentGateway amazonPayAdapter = new AmazonPayAdapter(amazonPayService);

        payPalAdapter.handlePayment(100.00);

        stripeAdapter.handlePayment(200.00);

        amazonPayAdapter.handlePayment(300.00);

    }

}

**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

interface NotificationService {

    void deliver(String message);

}

class EmailNotification implements NotificationService {

    @Override

    public void deliver(String message) {

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

    }

}

abstract class NotificationDecorator implements NotificationService {

    protected NotificationService wrappedNotifier;

    public NotificationDecorator(NotificationService notifier) {

        this.wrappedNotifier = notifier;

    }

    @Override

    public void deliver(String message) {

        wrappedNotifier.deliver(message);

    }

}

class SMSNotificationDecorator extends NotificationDecorator {

    public SMSNotificationDecorator(NotificationService notifier) {

        super(notifier);

    }

    @Override

    public void deliver(String message) {

        wrappedNotifier.deliver(message);

        sendSMS(message);

    }

    private void sendSMS(String message) {

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

    }

}

class SlackNotificationDecorator extends NotificationDecorator {

    public SlackNotificationDecorator(NotificationService notifier) {

        super(notifier);

    }

    @Override

    public void deliver(String message) {

        wrappedNotifier.deliver(message);

        sendSlack(message);

    }

    private void sendSlack(String message) {

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

    }

}

class DecoratorPatternDemo {

    public static void main(String[] args) {

        NotificationService emailNotifier = new EmailNotification();

        NotificationService smsEnhancedNotifier = new SMSNotificationDecorator(emailNotifier);

        NotificationService slackEnhancedNotifier = new SlackNotificationDecorator(smsEnhancedNotifier);

        slackEnhancedNotifier.deliver("Hello, this is a test notification!");

    }

}

**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

interface Picture {

    void show();

}

// Concrete class implementing the real image loading

class ActualImage implements Picture {

    private String filePath;

    public ActualImage(String filePath) {

        this.filePath = filePath;

        loadFromDisk();

    }

    private void loadFromDisk() {

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

    }

    @Override

    public void show() {

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

    }

}

// Proxy class that controls access to the actual image

class ImageProxy implements Picture {

    private String filePath;

    private ActualImage actualImage;

    public ImageProxy(String filePath) {

        this.filePath = filePath;

    }

    @Override

    public void show() {

        if (actualImage == null) {

            actualImage = new ActualImage(filePath);

        }

        actualImage.show();

    }

}

// Main class to test the proxy pattern

class ProxyPatternDemo {

    public static void main(String[] args) {

        Picture image1Proxy = new ImageProxy("image1.jpg");

        Picture image2Proxy = new ImageProxy("image2.jpg");

        // Image is loaded from disk for the first time

        image1Proxy.show();

        System.out.println("");

        // Image is not loaded from disk as it is already loaded

        image1Proxy.show();

        System.out.println("");

        // Image is loaded from disk for the first time

        image2Proxy.show();

        System.out.println("");

        // Image is not loaded from disk as it is already loaded

        image2Proxy.show();

    }

}

**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;

interface Market {

    void addObserver(Observer o);

    void removeObserver(Observer o);

    void notifyObservers();

}

class StockExchange implements Market {

    private List<Observer> observerList;

    private double currentStockPrice;

    public StockExchange() {

        this.observerList = new ArrayList<>();

    }

    @Override

    public void addObserver(Observer o) {

        observerList.add(o);

    }

    @Override

    public void removeObserver(Observer o) {

        observerList.remove(o);

    }

    @Override

    public void notifyObservers() {

        for (Observer o : observerList) {

            o.update(currentStockPrice);

        }

    }

    public void setStockPrice(double price) {

        this.currentStockPrice = price;

        notifyObservers();

    }

}

interface Observer {

    void update(double price);

}

class MobileNotification implements Observer {

    private String applicationName;

    public MobileNotification(String applicationName) {

        this.applicationName = applicationName;

    }

    @Override

    public void update(double price) {

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

    }

}

class WebNotification implements Observer {

    private String applicationName;

    public WebNotification(String applicationName) {

        this.applicationName = applicationName;

    }

    @Override

    public void update(double price) {

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

    }

}

class Implementing\_ObserverPatternDemo {

    public static void main(String[] args) {

        StockExchange stockExchange = new StockExchange();

        Observer mobileNotifier = new MobileNotification("MobileApp");

        Observer webNotifier = new WebNotification("WebApp");

        stockExchange.addObserver(mobileNotifier);

        stockExchange.addObserver(webNotifier);

        stockExchange.setStockPrice(100.00);

        stockExchange.setStockPrice(101.50);

        stockExchange.removeObserver(webNotifier);

        stockExchange.setStockPrice(102.75);

    }

}

**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.

// Define the PaymentMethod interface

interface PaymentMethod {

    void processPayment(double amount);

}

// Implement concrete payment methods

class Implementing\_the\_Strategy\_Pattern implements PaymentMethod {

    private String cardHolderName;

    private String cardNumber;

    private String securityCode;

    private String expirationDate;

    public Implementing\_the\_Strategy\_Pattern(String cardHolderName, String cardNumber, String securityCode, String expirationDate) {

        this.cardHolderName = cardHolderName;

        this.cardNumber = cardNumber;

        this.securityCode = securityCode;

        this.expirationDate = expirationDate;

    }

    @Override

    public void processPayment(double amount) {

        System.out.println("Processed payment of $" + amount + " using Credit Card.");

    }

}

class PayPalPayment implements PaymentMethod {

    private String accountEmail;

    private String accountPassword;

    public PayPalPayment(String accountEmail, String accountPassword) {

        this.accountEmail = accountEmail;

        this.accountPassword = accountPassword;

    }

    @Override

    public void processPayment(double amount) {

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

    }

}

// Implement the PaymentProcessor context

class PaymentProcessor {

    private PaymentMethod paymentMethod;

    public void setPaymentMethod(PaymentMethod paymentMethod) {

        this.paymentMethod = paymentMethod;

    }

    public void executePayment(double amount) {

        paymentMethod.processPayment(amount);

    }

}

// Test the Strategy Pattern

class StrategyPatternDemo {

    public static void main(String[] args) {

        PaymentProcessor processor = new PaymentProcessor();

        // Test payment using Credit Card

        processor.setPaymentMethod(new Implementing\_the\_Strategy\_Pattern("John Doe", "1234567890123456", "123", "12/23"));

        processor.executePayment(100.0);

        // Test payment using PayPal

        processor.setPaymentMethod(new PayPalPayment("john.doe@example.com", "password123"));

        processor.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.

// Define the Command interface

interface Action {

    void execute();

}

// Implement concrete command classes

class TurnOnLightCommand implements Action {

    private Light lightSource;

    public TurnOnLightCommand(Light lightSource) {

        this.lightSource = lightSource;

    }

    @Override

    public void execute() {

        lightSource.turnOn();

    }

}

class TurnOffLightCommand implements Action {

    private Light lightSource;

    public TurnOffLightCommand(Light lightSource) {

        this.lightSource = lightSource;

    }

    @Override

    public void execute() {

        lightSource.turnOff();

    }

}

// Implement the receiver class

class Light {

    public void turnOn() {

        System.out.println("The light is now on.");

    }

    public void turnOff() {

        System.out.println("The light is now off.");

    }

}

// Implement the invoker class

class Remote {

    private Action actionCommand;

    public void setAction(Action actionCommand) {

        this.actionCommand = actionCommand;

    }

    public void press() {

        actionCommand.execute();

    }

}

// Test the command pattern implementation

class CommandPatternDemo {

    public static void main(String[] args) {

        Light livingRoomLight = new Light();

        Action lightOnCommand = new TurnOnLightCommand(livingRoomLight);

        Action lightOffCommand = new TurnOffLightCommand(livingRoomLight);

        Remote remote = new Remote();

        // Turn the light on

        remote.setAction(lightOnCommand);

        remote.press();

        // Turn the light off

        remote.setAction(lightOffCommand);

        remote.press();

    }

}

**Exercise 10: Implementing the MVC Pattern**

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

// Define the Model class

class Student {

    private String studentId;

    private String studentName;

    private String studentGrade;

    public Student(String studentId, String studentName, String studentGrade) {

        this.studentId = studentId;

        this.studentName = studentName;

        this.studentGrade = studentGrade;

    }

    public String getStudentId() {

        return studentId;

    }

    public void setStudentId(String studentId) {

        this.studentId = studentId;

    }

    public String getStudentName() {

        return studentName;

    }

    public void setStudentName(String studentName) {

        this.studentName = studentName;

    }

    public String getStudentGrade() {

        return studentGrade;

    }

    public void setStudentGrade(String studentGrade) {

        this.studentGrade = studentGrade;

    }

}

// Define the View class

class StudentView {

    public void displayStudentInformation(String name, String id, String grade) {

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

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

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

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

    }

}

// Define the Controller class

class StudentController {

    private Student studentModel;

    private StudentView studentView;

    public StudentController(Student studentModel, StudentView studentView) {

        this.studentModel = studentModel;

        this.studentView = studentView;

    }

    public void setStudentName(String name) {

        studentModel.setStudentName(name);

    }

    public String getStudentName() {

        return studentModel.getStudentName();

    }

    public void setStudentId(String id) {

        studentModel.setStudentId(id);

    }

    public String getStudentId() {

        return studentModel.getStudentId();

    }

    public void setStudentGrade(String grade) {

        studentModel.setStudentGrade(grade);

    }

    public String getStudentGrade() {

        return studentModel.getStudentGrade();

    }

    public void updateStudentView() {

        studentView.displayStudentInformation(studentModel.getStudentName(), studentModel.getStudentId(), studentModel.getStudentGrade());

    }

}

// Test the MVC implementation

class MVCDemo {

    public static void main(String[] args) {

        // Create a Student model

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

        // Create a Student view

        StudentView view = new StudentView();

        // Create a Student controller

        StudentController controller = new StudentController(student, view);

        // Display initial student information

        controller.updateStudentView();

        // Update student information

        controller.setStudentName("Jane Smith");

        controller.setStudentGrade("B");

        // Display updated student information

        controller.updateStudentView();

    }

}

**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.

interface CustomerDataRepository {

    String retrieveCustomerById(String customerId);

}

// Implement Concrete Repository

class CustomerDataRepositoryImpl implements CustomerDataRepository {

    @Override

    public String retrieveCustomerById(String customerId) {

        // This is a mock implementation. In a real application, it would interact with a database.

        if (customerId.equals("1")) {

            return "John Doe";

        } else {

            return "Customer not found";

        }

    }

}

// Define Service Class

class CustomerManagementService {

    private CustomerDataRepository dataRepository;

    // Implement Dependency Injection

    public CustomerManagementService(CustomerDataRepository dataRepository) {

        this.dataRepository = dataRepository;

    }

    public String fetchCustomerInfo(String customerId) {

        return dataRepository.retrieveCustomerById(customerId);

    }

}

// Test the Dependency Injection Implementation

public class DependencyInjectionDemo {

    public static void main(String[] args) {

        // Create a CustomerDataRepository instance

        CustomerDataRepository dataRepository = new CustomerDataRepositoryImpl();

        // Inject the repository into the service

        CustomerManagementService service = new CustomerManagementService(dataRepository);

        // Use the service to get customer information

        String customerInfo = service.fetchCustomerInfo("1");

        System.out.println("Customer Info: " + customerInfo);

    }

}