**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:***

public class SingletonPatternExample {

static class Logger {

private static Logger instance;

private Logger() {

System.out.println("Logger initialized.");

}

public static Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

public void log(String message) {

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

}

}

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

Logger logger2 = Logger.getInstance();

logger1.log("First message");

logger2.log("Second message");

if (logger1 == logger2) {

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

} else {

System.out.println("Different instances exist. Singleton failed.");

}

}

}

***OUTPUT:***

Logger initialized.

Log: First message

Log: Second message

Both logger instances are the same. Singleton works!

**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.**

***CODE:***

public class FactoryMethodPatternExample {

interface Document {

void open();

}

static class WordDocument implements Document {

public void open() {

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

}

}

static class PdfDocument implements Document {

public void open() {

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

}

}

static class ExcelDocument implements Document {

public void open() {

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

}

}

abstract static class DocumentFactory {

public abstract Document createDocument();

}

static class WordDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

static class PdfDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

static class ExcelDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

public static void main(String[] args) {

DocumentFactory wordFactory = new WordDocumentFactory();

Document word = wordFactory.createDocument();

word.open();

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdf = pdfFactory.createDocument();

pdf.open();

DocumentFactory excelFactory = new ExcelDocumentFactory();

Document excel = excelFactory.createDocument();

excel.open();

}

}

***OUTPUT:***

Opening Word document.

Opening PDF document.

Opening Excel document.

**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:***

public class BuilderPatternExample {

static class Computer {

private String cpu;

private String ram;

private String storage;

private String gpu;

private Computer(Builder builder) {

this.cpu = builder.cpu;

this.ram = builder.ram;

this.storage = builder.storage;

this.gpu = builder.gpu;

}

public void showSpecs() {

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

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

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

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

System.out.println("GPU: " + gpu);

System.out.println();

}

static class Builder {

private String cpu;

private String ram;

private String storage;

private String gpu;

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 setGPU(String gpu) {

this.gpu = gpu;

return this;

}

public Computer build() {

return new Computer(this);

}

}

}

public static void main(String[] args) {

Computer gamingPC = new Computer.Builder()

.setCPU("Intel Core i9")

.setRAM("32GB")

.setStorage("1TB SSD")

.setGPU("NVIDIA RTX 4090")

.build();

Computer officePC = new Computer.Builder()

.setCPU("Intel Core i5")

.setRAM("16GB")

.setStorage("512GB SSD")

.build();

gamingPC.showSpecs();

officePC.showSpecs();

}

}

***OUTPUT:***

Computer Configuration:

CPU: Intel Core i9

RAM: 32GB

Storage: 1TB SSD

GPU: NVIDIA RTX 4090

Computer Configuration:

CPU: Intel Core i5

RAM: 16GB

Storage: 512GB SSD

GPU: null

**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.**

***CODE:***

public class AdapterPatternExample {

interface PaymentProcessor {

void processPayment(double amount);

}

static class PayPalGateway {

public void sendPayment(double amount) {

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

}

}

static class StripeGateway {

public void makeStripePayment(double value) {

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

}

}

static class PayPalAdapter implements PaymentProcessor {

private PayPalGateway payPalGateway;

public PayPalAdapter() {

this.payPalGateway = new PayPalGateway();

}

public void processPayment(double amount) {

payPalGateway.sendPayment(amount);

}

}

static class StripeAdapter implements PaymentProcessor {

private StripeGateway stripeGateway;

public StripeAdapter() {

this.stripeGateway = new StripeGateway();

}

public void processPayment(double amount) {

stripeGateway.makeStripePayment(amount);

}

}

public static void main(String[] args) {

PaymentProcessor paypal = new PayPalAdapter();

PaymentProcessor stripe = new StripeAdapter();

paypal.processPayment(250.0);

stripe.processPayment(450.0);

}

}

***OUTPUT:***

Processing PayPal payment of $250.0

Processing Stripe payment of $450.0

**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.**

***CODE:***

public class DecoratorPatternExample {

interface Notifier {

void send(String message);

}

static class EmailNotifier implements Notifier {

public void send(String message) {

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

}

}

static abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

public void send(String message) {

notifier.send(message);

}

}

static class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

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

}

}

static class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

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

}

}

public static void main(String[] args) {

Notifier basicNotifier = new EmailNotifier();

Notifier smsNotifier = new SMSNotifierDecorator(basicNotifier);

Notifier slackAndSmsNotifier = new SlackNotifierDecorator(smsNotifier);

System.out.println("Sending single-channel notification:");

basicNotifier.send("System maintenance at 10 PM");

System.out.println("\nSending dual-channel notification (Email + SMS):");

smsNotifier.send("Your password was changed");

System.out.println("\nSending multi-channel notification (Email + SMS + Slack):");

slackAndSmsNotifier.send("New login from unrecognized device");

}

}

***OUTPUT:***Sending single-channel notification:

Email: System maintenance at 10 PM

Sending dual-channel notification (Email + SMS):

Email: Your password was changed

SMS: Your password was changed

Sending multi-channel notification (Email + SMS + Slack):

Email: New login from unrecognized device

SMS: New login from unrecognized device

Slack: New login from unrecognized device

**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.**

***CODE:***

public class ProxyPatternExample {

interface Image {

void display();

}

static class RealImage implements Image {

private String filename;

public RealImage(String filename) {

this.filename = filename;

loadFromServer();

}

private void loadFromServer() {

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

}

public void display() {

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

}

}

static class ProxyImage implements Image {

private String filename;

private RealImage realImage;

public ProxyImage(String filename) {

this.filename = filename;

}

public void display() {

if (realImage == null) {

realImage = new RealImage(filename);

} else {

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

}

realImage.display();

}

}

public static void main(String[] args) {

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

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

System.out.println("First display (nature.jpg):");

image1.display();

System.out.println("\nSecond display (nature.jpg):");

image1.display();

System.out.println("\nFirst display (space.jpg):");

image2.display();

}

}

***OUTPUT:***

First display (nature.jpg):

Loading image from server: nature.jpg

Displaying image: nature.jpg

Second display (nature.jpg):

Using cached image: nature.jpg

Displaying image: nature.jpg

First display (space.jpg):

Loading image from server: space.jpg

Displaying image: space.jpg

**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.**

***CODE:***

import java.util.\*;

public class ObserverPatternExample {

interface Observer {

void update(String stockName, double price);

}

interface Stock {

void register(Observer o);

void deregister(Observer o);

void notifyObservers();

void setPrice(double price);

}

static class StockMarket implements Stock {

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

private String stockName;

private double stockPrice;

public StockMarket(String stockName, double initialPrice) {

this.stockName = stockName;

this.stockPrice = initialPrice;

}

public void register(Observer o) {

observers.add(o);

}

public void deregister(Observer o) {

observers.remove(o);

}

public void notifyObservers() {

for (Observer o : observers) {

o.update(stockName, stockPrice);

}

}

public void setPrice(double price) {

this.stockPrice = price;

notifyObservers();

}

}

static class MobileApp implements Observer {

private String name;

public MobileApp(String name) {

this.name = name;

}

public void update(String stockName, double price) {

System.out.println("MobileApp [" + name + "]: " + stockName + " price updated to $" + price);

}

}

static class WebApp implements Observer {

private String name;

public WebApp(String name) {

this.name = name;

}

public void update(String stockName, double price) {

System.out.println("WebApp [" + name + "]: " + stockName + " price updated to $" + price);

}

}

public static void main(String[] args) {

StockMarket teslaStock = new StockMarket("TSLA", 700.00);

Observer mobileClient = new MobileApp("Alice");

Observer webClient = new WebApp("Dashboard");

teslaStock.register(mobileClient);

teslaStock.register(webClient);

System.out.println("First price update:");

teslaStock.setPrice(720.00);

System.out.println("\nSecond price update after deregistering WebApp:");

teslaStock.deregister(webClient);

teslaStock.setPrice(735.50);

}

}

***OUTPUT:***

First price update:

MobileApp [Alice]: TSLA price updated to $720.0

WebApp [Dashboard]: TSLA price updated to $720.0

Second price update after deregistering WebApp:

MobileApp [Alice]: TSLA price updated to $735.5

**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.**

***CODE:***

public class StrategyPatternExample {

interface PaymentStrategy {

void pay(double amount);

}

static class CreditCardPayment implements PaymentStrategy {

public void pay(double amount) {

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

}

}

static class PayPalPayment implements PaymentStrategy {

public void pay(double amount) {

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

}

}

static class PaymentContext {

private PaymentStrategy strategy;

public void setPaymentStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void executePayment(double amount) {

if (strategy != null) {

strategy.pay(amount);

} else {

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

}

}

}

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

context.setPaymentStrategy(new CreditCardPayment());

context.executePayment(1000.00);

context.setPaymentStrategy(new PayPalPayment());

context.executePayment(500.50);

}

}

***OUTPUT:***

Paid $1000.0 using Credit Card.

Paid $500.5 using PayPal.

**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.**

***CODE:***

public class CommandPatternExample {

interface Command {

void execute();

}

static class Light {

public void turnOn() {

System.out.println("The light is ON");

}

public void turnOff() {

System.out.println("The light is OFF");

}

}

static class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOn();

}

}

static class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOff();

}

}

static class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

if (command != null) {

command.execute();

} else {

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

}

}

}

public static void main(String[] args) {

Light livingRoomLight = new Light();

Command lightOn = new LightOnCommand(livingRoomLight);

Command lightOff = new LightOffCommand(livingRoomLight);

RemoteControl remote = new RemoteControl();

System.out.println("Pressing ON button:");

remote.setCommand(lightOn);

remote.pressButton();

System.out.println("\nPressing OFF button:");

remote.setCommand(lightOff);

remote.pressButton();

}

}

***OUTPUT:***

Pressing ON button:

The light is ON

Pressing OFF button:

The light is OFF

**Exercise 10: Implementing the MVC Pattern**

**Scenario:**

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

***CODE:***

public class MVCPatternExample {

// Model

static class Student {

private String name;

private String id;

private String grade;

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;

}

}

// View

static 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);

}

}

// Controller

static class StudentController {

private Student model;

private StudentView view;

public StudentController(Student model, StudentView view) {

this.model = model;

this.view = view;

}

public void setStudentName(String name) {

model.setName(name);

}

public void setStudentId(String id) {

model.setId(id);

}

public void setStudentGrade(String grade) {

model.setGrade(grade);

}

public void updateView() {

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

}

}

public static void main(String[] args) {

Student student = new Student("John Doe", "ST101", "A");

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

System.out.println("Initial Student Info:");

controller.updateView();

System.out.println("\nUpdated Student Info:");

controller.setStudentName("Jane Smith");

controller.setStudentId("ST202");

controller.setStudentGrade("B+");

controller.updateView();

}

}

***OUTPUT:***

Initial Student Info:

Student Details:

Name: John Doe

ID: ST101

Grade: A

Updated Student Info:

Student Details:

Name: Jane Smith

ID: ST202

Grade: B+

**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.**

***CODE:***

public class DependencyInjectionExample {

// Model class

static 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;

}

}

// Step 2: Repository Interface

interface CustomerRepository {

Customer findCustomerById(String id);

}

// Step 3: Concrete Repository Implementation

static class CustomerRepositoryImpl implements CustomerRepository {

public Customer findCustomerById(String id) {

// Dummy data

if (id.equals("C101")) {

return new Customer(id, "Alice Johnson");

} else {

return new Customer(id, "Unknown Customer");

}

}

}

// Step 4: Service Class that depends on Repository

static class CustomerService {

private CustomerRepository repository;

// Step 5: Constructor Injection

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void printCustomerDetails(String id) {

Customer customer = repository.findCustomerById(id);

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

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

}

}

// Step 6: Main class to test DI

public static void main(String[] args) {

CustomerRepository repository = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repository);

System.out.println("Fetching Customer Details:");

service.printCustomerDetails("C101");

System.out.println("\nFetching Unknown Customer:");

service.printCustomerDetails("C999");

}

}

***OUTPUT:***

Fetching Customer Details:

Customer ID: C101

Customer Name: Alice Johnson

Fetching Unknown Customer:

Customer ID: C999

Customer Name: Unknown Customer