**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.

**Steps:**

1. **Create a New Java Project:**
   1. Create a new Java project named **BuilderPatternExample**.
2. **Define a Product Class:**
   1. Create a class **Computer** with attributes like **CPU**, **RAM**, **Storage**, etc.
3. **Implement the Builder Class:**
   1. Create a static nested Builder class inside Computer with methods to set each attribute.
   2. Provide a **build()** method in the Builder class that returns an instance of Computer.
4. **Implement the Builder Pattern:**
   1. Ensure that the **Computer** class has a private constructor that takes the **Builder** as a parameter.
5. **Test the Builder Implementation:**
   1. Create a test class to demonstrate the creation of different configurations of Computer using the Builder pattern.

**Code:**

**Computer.java**

package BuiderPatternPackage;

public class Computer {

private String CPU;

private String RAM;

private String storage;

private Computer(Builder builder) {

this.CPU = builder.CPU;

this.RAM = builder.RAM;

this.storage = builder.storage;

}

public static class Builder {

private String CPU;

private String RAM;

private String storage;

public Builder setCPU(String CPU) {

this.CPU = CPU;

return this;

}

public Builder setRAM(String RAM) {

this.RAM = RAM;

return this;

}

public Builder setStorage(String storage) {

this.storage = storage;

return this;

}

public Computer build() {

return new Computer(this);

}

}

public void showSpecs() {

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

}

}

**Test.java**

package BuiderPatternPackage;

public class Test {

public static void main(String[] args) {

// TODO Auto-generated method stub

Computer myPC = new Computer.Builder()

.setCPU("Intel i7")

.setRAM("16GB")

.setStorage("1TB SSD")

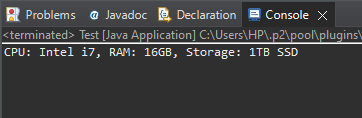
.build();

myPC.showSpecs();

}

}

**Output:**



**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.

**Steps:**

1. **Create a New Java Project:**
   1. Create a new Java project named **AdapterPatternExample**.
2. **Define Target Interface:**
   1. Create an interface **PaymentProcessor** with methods like **processPayment()**.
3. **Implement Adaptee Classes:**
   1. Create classes for different payment gateways with their own methods.
4. **Implement the Adapter Class:**
   1. Create an adapter class for each payment gateway that implements PaymentProcessor and translates the calls to the gateway-specific methods.
5. **Test the Adapter Implementation:**
   1. Create a test class to demonstrate the use of different payment gateways through the adapter.

**Code:**

**PaymentProcessor.java**

public interface PaymentProcessor {

void processPayment();

}

**PayPal.java**

class PayPal {

void payNow() {

System.out.println("Payment done via PayPal!!");

}

}

**PayPalAdapter.java**

class PayPalAdapter implements PaymentProcessor {

private PayPal Paypal;

PayPalAdapter(PayPal Paypal) {

this.Paypal = Paypal;

}

public void processPayment() {

Paypal.payNow();

}

}

**Test.java**

public class Test {

public static void main(String[] args) {

// TODO Auto-generated method stub

PayPal pay = new PayPal();

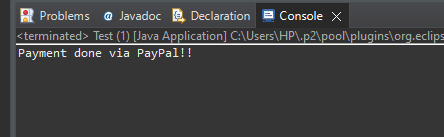
PaymentProcessor processor = new PayPalAdapter(pay);

processor.processPayment();

}

}

**Output:**



**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.

**Steps:**

1. **Create a New Java Project:**
   1. Create a new Java project named **DecoratorPatternExample**.
2. **Define Component Interface:**
   1. Create an interface **Notifier** with a method **send()**.
3. **Implement Concrete Component:**
   1. Create a class **EmailNotifier** that implements Notifier.
4. **Implement Decorator Classes:**
   1. Create abstract decorator class **NotifierDecorator** that implements **Notifier** and holds a reference to a **Notifier** object.
   2. Create concrete decorator classes like **SMSNotifierDecorator**, **SlackNotifierDecorator** that extend **NotifierDecorator**.
5. **Test the Decorator Implementation:**
   1. Create a test class to demonstrate sending notifications via multiple channels using decorators.

**Code:**

**Notifier.java**

interface Notifier {

void send(String message);

}

**EmailNotifier.java**

class EmailNotifier implements Notifier {

public void send(String message) {

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

}

}

**NotifierDecorator.java**

abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

}

**SMSNotifierDecorator.java**

class SMSNotifierDecorator extends NotifierDecorator {

SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

notifier.send(message);

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

}

}

**Test.java**

public class Test{

public static void main(String[] args) {

Notifier email = new EmailNotifier();

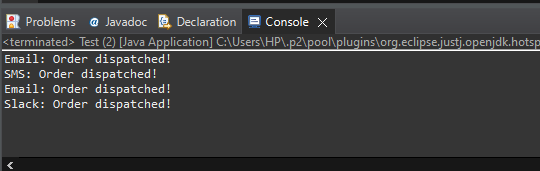
Notifier emailWithSMS = new SMSNotifierDecorator(email);

emailWithSMS.send("Order dispatched!");

}

}

**Output:**



**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.

**Steps:**

1. **Create a New Java Project:**
   1. Create a new Java project named **ProxyPatternExample**.
2. **Define Subject Interface:**
   1. Create an interface Image with a method **display()**.
3. **Implement Real Subject Class:**
   1. Create a class **RealImage** that implements Image and loads an image from a remote server.
4. **Implement Proxy Class:**
   1. Create a class **ProxyImage** that implements Image and holds a reference to RealImage.
   2. Implement lazy initialization and caching in **ProxyImage**.
5. **Test the Proxy Implementation:**
   1. Create a test class to demonstrate the use of **ProxyImage** to load and display images.

**Code:**

**Image.java**

interface Image {

void display();

}

**RealImage.java**

class RealImage implements Image {

private String fileName;

public RealImage(String fileName) {

this.fileName = fileName;

loadFromDisk();

}

private void loadFromDisk() {

System.out.println("Loading " + fileName);

}

public void display() {

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

}

}

**ProxyImage.java**

class ProxyImage implements Image {

private RealImage realImage;

private String fileName;

public ProxyImage(String fileName) {

this.fileName = fileName;

}

public void display() {

if (realImage == null) {

realImage = new RealImage(fileName);

}

realImage.display();

}

}

**Test.java**

public class Test {

public static void main(String[] args) {

// TODO Auto-generated method stub

Image image = new ProxyImage("cat.jpg");

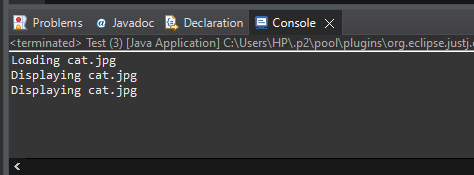
image.display();

image.display();

}

}

**Output:**



**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.

**Steps:**

1. **Create a New Java Project:**
   1. Create a new Java project named **ObserverPatternExample**.
2. **Define Subject Interface:**
   1. Create an interface **Stock** with methods to **register**, **deregister**, and **notify** observers.
3. **Implement Concrete Subject:**
   1. Create a class **StockMarket** that implements **Stock** and maintains a list of observers.
4. **Define Observer Interface:**
   1. Create an interface Observer with a method **update().**
5. **Implement Concrete Observers:**
   1. Create classes **MobileApp**, **WebApp** that implement Observer.
6. **Test the Observer Implementation:**
   1. Create a test class to demonstrate the registration and notification of observers.

**Code:**

**Stock.java**

public interface Stock {

void register(Observer o);

void deregister(Observer o);

void notifyObservers();

}

**StockMarket.java**

import java.util.\*;

class StockMarket implements Stock {

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

private float price;

public void register(Observer o) {

observers.add(o);

}

public void deregister(Observer o) {

observers.remove(o);

}

public void setPrice(float price) {

this.price = price;

notifyObservers();

}

public void notifyObservers() {

for (Observer o : observers) {

o.update(price);

}

}

}

**Observer.java**

interface Observer {

void update(float price);

}

**MobileApp.java:**

class MobileApp implements Observer {

public void update(float price) {

System.out.println("Mobile App: Stock price updated to ₹" + price);

}

}

**WebApp.java:**

class WebApp implements Observer {

public void update(float price) {

System.out.println("Web App: Stock price updated to ₹" + price);

}

}

**Test.java:**

public class Test {

public static void main(String[] args) {

// TODO Auto-generated method stub

StockMarket stockMarket = new StockMarket();

Observer mobileApp = new MobileApp();

Observer webApp = new WebApp();

stockMarket.register(mobileApp);

stockMarket.register(webApp);

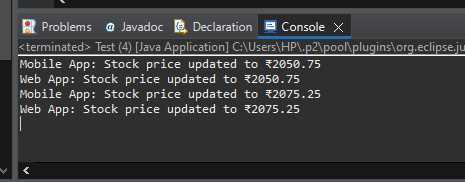
stockMarket.setPrice(2050.75f);

stockMarket.setPrice(2075.25f);

}

}

**Output:**



**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.

**Steps:**

1. **Create a New Java Project:**
   1. Create a new Java project named **StrategyPatternExample**.
2. **Define Strategy Interface:**
   1. Create an interface PaymentStrategy with a method **pay()**.
3. **Implement Concrete Strategies:**
   1. Create classes **CreditCardPayment**, **PayPalPayment** that implement **PaymentStrategy**.
4. **Implement Context Class:**
   1. Create a class **PaymentContext** that holds a reference to **PaymentStrategy** and a method to execute the strategy.
5. **Test the Strategy Implementation:**
   1. Create a test class to demonstrate selecting and using different payment strategies.

**Code:**

**PaymentStrategy.java**

interface PaymentStrategy {

void pay(int amount);

}

**CreditCardPayment.java**

class CreditCardPayment implements PaymentStrategy {

public void pay(int amount) {

System.out.println("Paid ₹" + amount + " via Credit Card");

}

}

**PayPalPayment.java**

class PayPalPayment implements PaymentStrategy {

public void pay(int amount) {

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

}

}

**PaymentContext.java**

class PaymentContext {

private PaymentStrategy strategy;

public void setPaymentStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void pay(int amount) {

strategy.pay(amount);

}

}

**Test.java**

public class Test {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

context.setPaymentStrategy(new CreditCardPayment());

context.pay(1000);

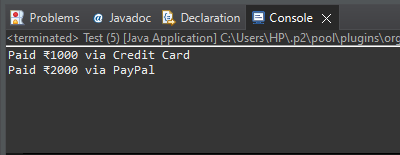
context.setPaymentStrategy(new PayPalPayment());

context.pay(2000);

}

}

**Output:**



**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.

**Steps:**

1. **Create a New Java Project:**
   1. Create a new Java project named **CommandPatternExample**.
2. **Define Command Interface:**
   1. Create an interface Command with a method **execute()**.
3. **Implement Concrete Commands:**
   1. Create classes **LightOnCommand**, **LightOffCommand** that implement Command.
4. **Implement Invoker Class:**
   1. Create a class **RemoteControl** that holds a reference to a Command and a method to execute the command.
5. **Implement Receiver Class:**
   1. Create a class **Light** with methods to turn on and off.
6. **Test the Command Implementation:**
   1. Create a test class to demonstrate issuing commands using the **RemoteControl**.

**Code:**

**Command.java**

interface Command {

void execute();

}

**LightOnCommand.java**

class LightOnCommand implements Command {

private Light light;

LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.on();

}

}

**LightOffCommand.java**

class LightOffCommand implements Command {

private Light light;

LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.off();

}

}

**RemoteControl.java**

class RemoteControl {

private Command command;

void setCommand(Command command) {

this.command = command;

}

void pressButton() {

command.execute();

}

}

**Light.java**

class Light {

void on() {

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

}

void off() {

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

}

}

**Test.java**

public class Test {

public static void main(String[] args) {

Light light = new Light();

RemoteControl remote = new RemoteControl();

remote.setCommand(new LightOnCommand(light));

remote.pressButton();

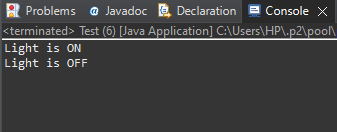
remote.setCommand(new LightOffCommand(light));

remote.pressButton();

}

}

**Output:**



**Exercise 10: Implementing the MVC Pattern**

**Scenario:**

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

**Steps:**

1. **Create a New Java Project:**
   1. Create a new Java project named **MVCPatternExample**.
2. **Define Model Class:**
   1. Create a class **Student** with attributes like **name, id, and grade**.
3. **Define View Class:**
   1. Create a class **StudentView** with a method **displayStudentDetails()**.
4. **Define Controller Class:**
   1. Create a class **StudentController** that handles the communication between the model and the view.
5. **Test the MVC Implementation:**
   1. Create a main class to demonstrate creating a **Student**, updating its details using **StudentController**, and displaying them using **StudentView**.

**Code:**

**Student.java**

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 int getId() {

return id;

}

public String getGrade() {

return grade;

}

public void setGrade(String grade) {

this.grade = grade;

}

}

**StudentView.java**

public class StudentView {

public void displayStudentDetails(Student s) {

System.out.println("Student: " + s.getName() + ", ID: " + s.getId() + ", Grade: " + s.getGrade());

}

}

**StudentController.java**

public class StudentController {

private Student model;

private StudentView view;

public StudentController(Student model, StudentView view) {

this.model = model;

this.view = view;

}

public void updateGrade(String grade) {

model.setGrade(grade);

}

public void show() {

view.displayStudentDetails(model);

}

}

**Test.java**

public class Test {

public static void main(String[] args) {

// TODO Auto-generated method stub

Student student = new Student("Satyam", 101, "B");

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

controller.show();

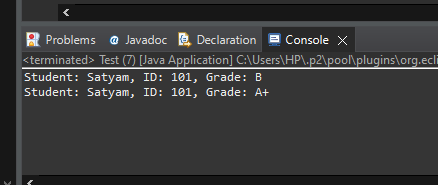
controller.updateGrade("A+");

controller.show();

}

}

**Output:**



**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.

**Steps:**

1. **Create a New Java Project:**
   1. Create a new Java project named **DependencyInjectionExample**.
2. **Define Repository Interface:**
   1. Create an interface **CustomerRepository** with methods like **findCustomerById()**.
3. **Implement Concrete Repository:**
   1. Create a class **CustomerRepositoryImpl** that implements **CustomerRepository**.
4. **Define Service Class:**
   1. Create a class **CustomerService** that depends on **CustomerRepository**.
5. **Implement Dependency Injection:**
   1. Use constructor injection to inject **CustomerRepository** into **CustomerService**.
6. **Test the Dependency Injection Implementation:**
   1. Create a main class to demonstrate creating a **CustomerService** with **CustomerRepositoryImpl** and using it to find a customer.

**Code:**

**CustomerRepository.java**

interface CustomerRepository {

String findCustomerById(int id);

}

**CustomerRepositoryImpl.java**

class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(int id) {

return "Customer ID: " + id + " (John Doe)";

}

}

**CustomerService.java**

class CustomerService {

private CustomerRepository repository;

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void showCustomer(int id) {

String customer = repository.findCustomerById(id);

System.out.println(customer);

}

}

**Main.java**

public class Main {

public static void main(String[] args) {

CustomerRepository repo = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repo);

service.showCustomer(1001);

}

}

**Output:**

