## Exercise 7: Implementing the Observer Pattern

### Step 1: Create a New Java Project

Create a new Java project named ObserverPatternExample.

### Step 2: Define Subject Interface

Create an interface Stock with methods to register, deregister, and notify observers.

**Code:**

// File: Stock.java

public interface Stock {

void registerObserver(Observer o);

void deregisterObserver(Observer o);

void notifyObservers();

}

### Step 3: Implement Concrete Subject

Create a class StockMarket that implements Stock and maintains a list of observers.

// File: StockMarket.java

import java.util.ArrayList;

import java.util.List;

public class StockMarket implements Stock {

private List<Observer> observers;

private double stockPrice;

public StockMarket() {

observers = new ArrayList<>();

}

public void setStockPrice(double stockPrice) {

this.stockPrice = stockPrice;

notifyObservers();

}

@Override

public void registerObserver(Observer o) {

observers.add(o);

}

@Override

public void deregisterObserver(Observer o) {

observers.remove(o);

}

@Override

public void notifyObservers() {

for (Observer o : observers) {

o.update(stockPrice);

}

}

}

### Step 4: Define Observer Interface

Create an interface Observer with a method update().

// File: Observer.java

public interface Observer {

void update(double stockPrice);

}

### Step 5: Implement Concrete Observers

Create classes MobileApp and WebApp that implement Observer.

// File: MobileApp.java

public class MobileApp implements Observer {

private String name;

public MobileApp(String name) {

this.name = name;

}

@Override

public void update(double stockPrice) {

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

}

}

// File: WebApp.java

public class WebApp implements Observer {

private String name;

public WebApp(String name) {

this.name = name;

}

@Override

public void update(double stockPrice) {

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

}

}

### Step 6: Test the Observer Implementation

Create a test class to demonstrate the registration and notification of observers.

// File: ObserverPatternTest.java

public class ObserverPatternTest {

public static void main(String[] args) {

StockMarket stockMarket = new StockMarket();

Observer mobileApp1 = new MobileApp("MobileApp1");

Observer mobileApp2 = new MobileApp("MobileApp2");

Observer webApp1 = new WebApp("WebApp1");

stockMarket.registerObserver(mobileApp1);

stockMarket.registerObserver(mobileApp2);

stockMarket.registerObserver(webApp1);

stockMarket.setStockPrice(100.00);

stockMarket.setStockPrice(105.50);

stockMarket.deregisterObserver(mobileApp1);

stockMarket.setStockPrice(110.00);

}

}

### Explanation:

1. **Stock Interface**: Defines methods for registering, deregistering, and notifying observers.
2. **StockMarket Class**: Implements Stock interface and maintains a list of observers. It notifies observers whenever the stock price changes.
3. **Observer Interface**: Defines the update method that will be called when the subject's state changes.
4. **MobileApp and WebApp Classes**: Implement the Observer interface and define the update method to handle stock price updates.
5. **ObserverPatternTest Class**: Demonstrates the registration and notification of observers by changing the stock price.

This implementation shows how the Observer Pattern can be used to notify multiple clients (observers) whenever the stock prices change.