CPOA - Support TD 5

Version corrigée

Cette version comporte des indications pour les réponses aux exercices.

PreReq	1. I know how to code in Java.
	2. I know I need to think before I start coding.
	3. I know several design patterns already.
ObjTD	Learn the **Observer** pattern.
Duration	**1** TD and **2** TPs

1. Motivations

In this exercise, we will explore the need to maintain consistency between related objects without making classes tightly coupled.
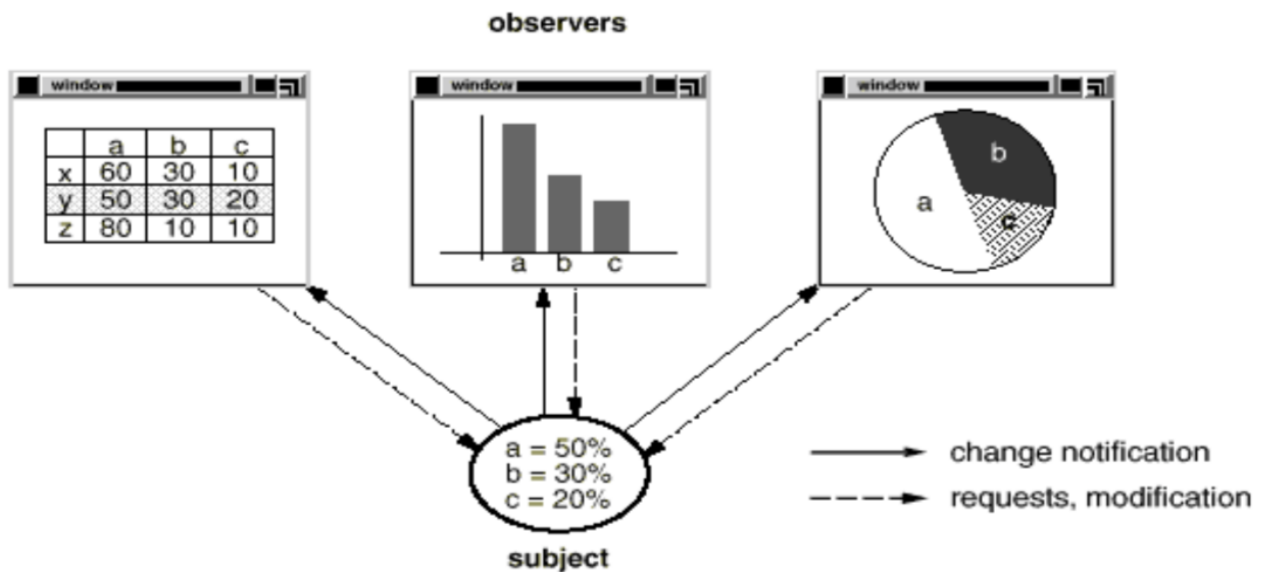

Figure 1. The classical Observer illustration

2. The Observer design pattern

2.1. Definition (sorry for the French)

Observateur définit une relation entre objets de type un-à-plusieurs, de façon que, lorsqu'un objet change d'état, tous ceux qui en dépendent en soient notifiés et soient mis à jour automatiquement.
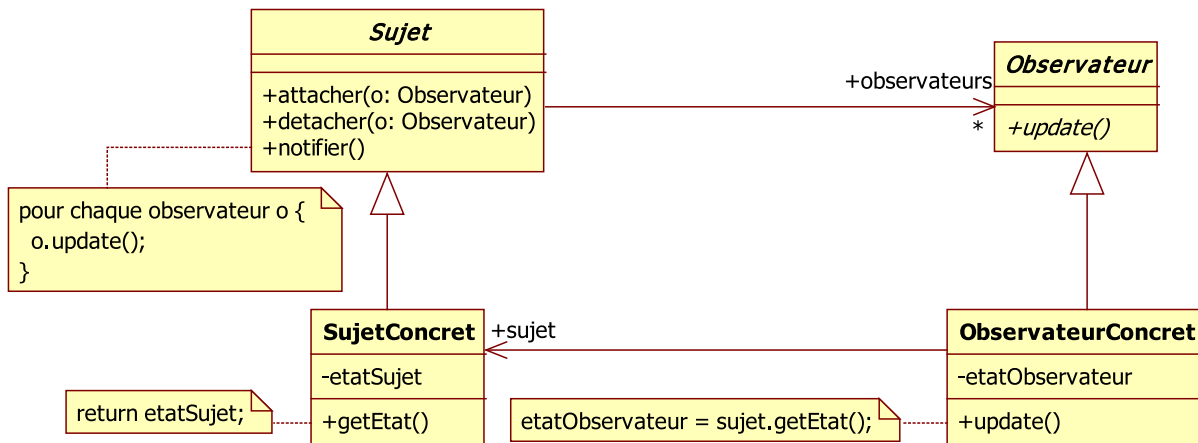
Figure 2. Modèle UML du patron Observer

Several precisions:

- the observers must be registered somehow
- the subject has to inform the observers (about changes).
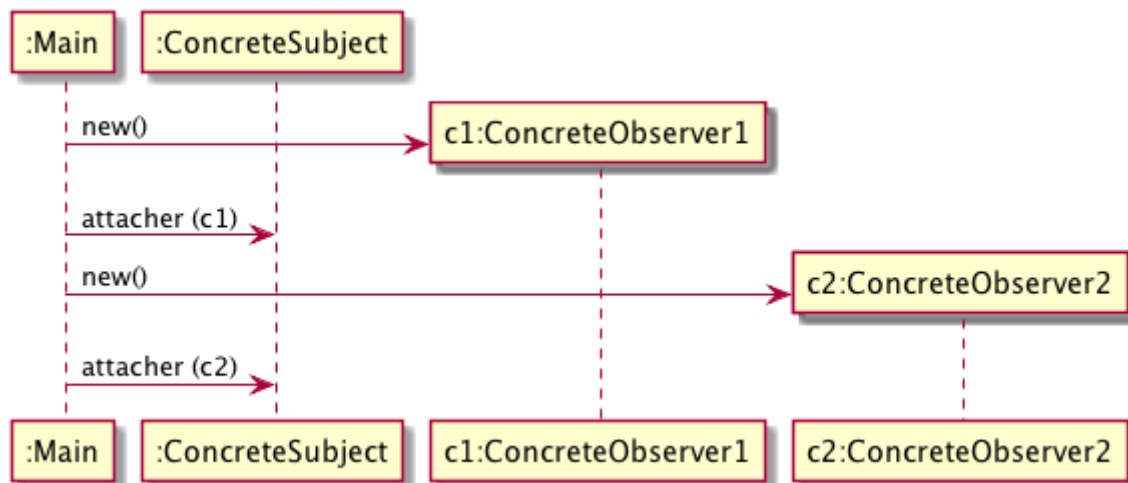

Figure 3. Example of use

There are several ways to implement the `attacher`, but in order to work properly it must be done:

- by an external object,
- by the `concreteObserver` itself.

2.2. Intent

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

2.3. Applicability

Use the Observer pattern in any of the following situations:

- When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects

in separate objects lets you vary and reuse them independently.

- When a change to one object requires changing others

- When an object should be able to notify other objects without making assumptions about those objects

2.4. Participants

- `Subject`
  - Keeps track of its `observers`
  - Provides an interface for attaching and detaching `Observer` objects

- `Observer`
  - Defines an interface for update notification

- `ConcreteSubject`
  - The object being observed
  - Stores state of interest to ConcreteObserver objects
  - Sends a notification to its observers when its state changes

- `ConcreteObserver`
  - The observing object
  - Stores state that should stay consistent with the subject's
  - Implements the Observer update interface to keep its state consistent with the subject's

2.5. Scenario

> **QUESTION**
>
> Let's see if you have undestood its utility. Make a sequence diagram that illustrate the use of the *Observer* pattern.
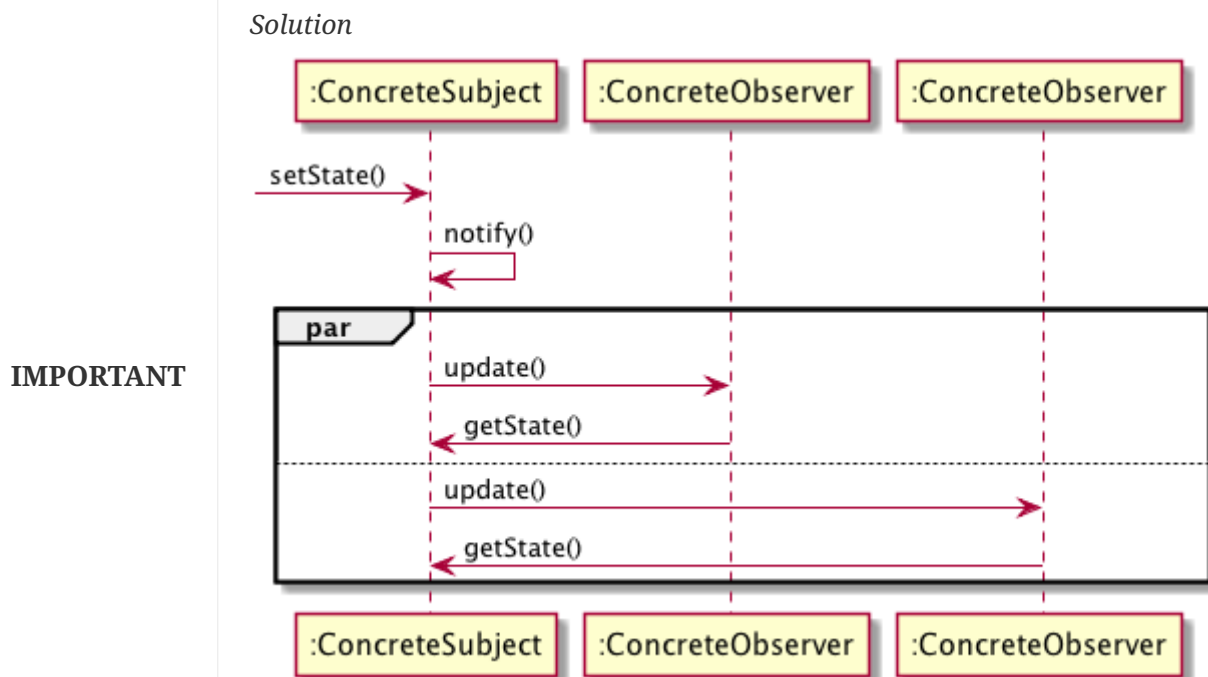
Figure 4. Example of use

2.6. Benefits

- Minimal coupling between the Subject and the Observer
  - Can reuse subjects without reusing their observers and vice versa
  - Observers can be added without modifying the subject
  - All subject knows is its list of observers
  - Subject does not need to know the concrete class of an observer, just that each observer implements the update interface
  - Subject and observer can belong to different abstraction layers
- Support for event broadcasting
  - Subject sends notification to all subscribed observers
  - Observers can be added/removed at any time

2.7. Liabilities

- Possible cascading of notifications
  - Observers are not necessarily aware of each other and must be careful about triggering updates
- Simple update interface requires observers to deduce changed item

3. Implementation Issues

3.1. Problems

Let's try to define an implementation.

> If you already know or have used the `Observer` Java implementation, try to forget it!

QUESTION

Here are a list of possible issues. For each of them try to provide some answers:

1. How does the subject keep track of its observers?
2. What if an observer wants to observe more than one subject?
3. Who triggers the update?
4. Make sure the subject updates its state before sending out notifications
5. How much info about the change should the subject send to the observers?
6. Can the observers subscribe to specific events of interest?
7. Can an observer also be a subject?
8. What if an observer wants to be notified only after several subjects have changed state?

IMPORTANT

Solution

1. Array, linked list
2. Have the subject tell the observer who it is via the update interface
3. For example:
   - The subject whenever its state changes
   - The observers after they cause one or more state changes
   - Some third party object(s)
4. Hard to enforce other than by systematic checking/testing
5. For example:
   - Push Model ⇒ Lots
   - Pull Model ⇒ Very Little
6. If so, it's publish-subscribe
7. Yes!
8. For example:
   - Use an intermediary object which acts as a mediator
   - Subjects send notifications to the mediator object which performs any necessary processing before notifying the observers

3.2. *Observer* **in Java**

Java provides the *Observable/Observer* classes as built-in support for the *Observer* pattern. The `java.util.Observable` class is the base `Subject` class. Any class that wants to be observed extends this class which:

- Provides methods to add/delete observers
- Provides methods to notify all observers
- A subclass only needs to ensure that its observers are notified in the appropriate mutators
- Uses a `Vector` for storing the observer references

The `java.util.Observer` interface is the `Observer` interface. It must be implemented by any observer class.

3.2.1. The java.util.Observable Class

QUESTION

Here is a list of methods of the `java.util.Observable` class:

```java
1 public Observable()
2 public synchronized void addObserver(Observer o)
3 protected synchronized void setChanged()
4 public synchronized void deleteObserver(Observer o)
5 protected synchronized void clearChanged()
6 public synchronized boolean hasChanged()
7 public void notifyObservers(Object arg)
8 public void notifyObservers()
```

Match them with their corresponding comments:

1. Adds an observer to the set of observers of this object

2. If this object has changed, as indicated by the hasChanged() method, then notify all of its observers and then call the clearChanged() method to indicate that this object has no longer changed. Each observer has its update() method called with two arguments: this observable object and the arg argument. The arg argument can be used to indicate which attribute of the observable object has changed.

3. Indicates that this object has changed

4. Indicates that this object has no longer changed, or that it has already notified all of its observers of its most recent change. This method is called automatically by `notifyObservers()`.

5. Same as above, but the arg argument is set to null. That is, the observer is given no indication what attribute of the observable object has changed.

6. Tests if this object has changed. Returns true if setChanged() has been called more recently than clearChanged() on this object; false otherwise.

7. Construct an `Observable` with zero observers

8. Deletes an observer from the set of observers of this object

Solution

```
 1 public Observable()
 2 // => Construct an `Observable` with zero Observers
 3 public synchronized void addObserver(Observer o)::
 4 // => Adds an observer to the set of observers of this object
 5 protected synchronized void setChanged()
 6 // => Indicates that this object has changed
 7 public synchronized void deleteObserver(Observer o)
 8 // => Deletes an observer from the set of observers of this
   object
 9 protected synchronized void clearChanged()
10 /* => Indicates that this object has no longer changed, or that
   it has already
11 notified all of its observers of its most recent change. This
   method is called
12 automatically by notifyObservers(). */
13 public synchronized boolean hasChanged()
14 /* => Tests if this object has changed. Returns true if
   setChanged() has been
15 called more recently than clearChanged() on this object; false
   otherwise. */
16 public void notifyObservers(Object arg)
17 /* => If this object has changed, as indicated by the
   hasChanged() method, then
18 notify all of its observers and then call the clearChanged()
   method to
19 indicate that this object has no longer changed. Each observer
   has its
20 update() method called with two arguments: this observable
   object and the
21 arg argument. The arg argument can be used to indicate which
   attribute of
22 the observable object has changed. */
23 public void notifyObservers()
24 /* => Same as above, but the arg argument is set to null. That
   is, the observer
25 is given no indication what attribute of the observable object
   has changed. */
```

3.2.2. The java.util.Observer Interface

```
 1 /**
 2 * This method is called whenever the observed object is changed. An
 3 * application calls an observable object's notifyObservers method to have all
 4 * the object's observers notified of the change.
 5 *
 6 * Parameters:
 7 * o - the observable object
 8 * arg - an argument passed to the notifyObservers method
 9 */
10 public abstract void update(Observable o, Object arg)
```

3.3. Observable/Observer Example

ConcreteSubject.java

```
 1 /**
 2 * A subject to observe!
 3 */
 4 public class ConcreteSubject extends Observable {
 5
 6   private String name;
 7   private float price;
 8   public ConcreteSubject(String name, float price) {
 9     this.name = name;
10     this.price = price;
11     System.out.println("ConcreteSubject created: " + name + " at " + price);
12   }
13   public String getName() {return name;}
14   public float getPrice() {return price;}
15   public void setName(String name) {
16     this.name = name;
17     setChanged();
18     notifyObservers(name);
19   }
20   public void setPrice(float price) {
21     this.price = price;
22     setChanged();
23     notifyObservers(new Float(price));
24   }
25 }
```

NameObserver.java

```java
// An observer of name changes.
public class NameObserver implements Observer {
  private String name;
  public NameObserver() {
    name = null;
    System.out.println("NameObserver created: Name is " + name);
  }
  public void update(Observable obj, Object arg) {
    if (arg instanceof String) {
      name = (String)arg;
      System.out.println("NameObserver: Name changed to " + name);
    } else {
      System.out.println("NameObserver: Some other change to
      subject!");
    }
  }
}
```

PriceObserver.java

```java
// An observer of price changes.
public class PriceObserver implements Observer {
  private float price;
  public PriceObserver() {
    price = 0;
    System.out.println("PriceObserver created: Price is " + price);
  }
  public void update(Observable obj, Object arg) {
    if (arg instanceof Float) {
      price = ((Float)arg).floatValue();
      System.out.println("PriceObserver: Price changed to " +
      price);
    } else {
      System.out.println("PriceObserver: Some other change to
      subject!");
    }
  }
}
```

TestObservers.java

```java
1 // Test program for ConcreteSubject, NameObserver and PriceObserver
2 public class TestObservers {
3   public static void main(String args[]) {
4     // Create the Subject and Observers.
5     ConcreteSubject s = new ConcreteSubject("Corn Pops", 1.29f);
6     NameObserver nameObs = new NameObserver();
7     PriceObserver priceObs = new PriceObserver();
8     // Add those Observers!
9     s.addObserver(nameObs);
10    s.addObserver(priceObs);
11    // Make changes to the Subject.
12    s.setName("Frosted Flakes");
13    s.setPrice(4.57f);
14    s.setPrice(9.22f);
15    s.setName("Sugar Crispies");
16  }
17 }
```

QUESTION

Donnez la trace d'exécution du code ci-dessus.

Solution

Test program output

IMPORTANT

```
1  ConcreteSubject created: Corn Pops at 1.29
2  NameObserver created: Name is null
3  PriceObserver created: Price is 0.0
4  PriceObserver: Some other change to subject!
5  NameObserver: Name changed to Frosted Flakes
6  PriceObserver: Price changed to 4.57
7  NameObserver: Some other change to subject!
8  PriceObserver: Price changed to 9.22
9  NameObserver: Some other change to subject!
10 PriceObserver: Some other change to subject!
11 NameObserver: Name changed to Sugar Crispies
```

3.4. Limitations of Observable/Observer in Java

Problem

Suppose the class which we want to be an observable is already part of an inheritance hierarchy, e.g.,:

```java
1 class SpecialSubject extends ParentClass
```

QUESTION

Since Java does not support multiple inheritance, how can we have ConcreteSubject extend both Observable and ParentClass ?

1. Propose a solution

2. Define its corresponding class diagram

3. Write the Java implementations (mainly SpecialSubject)

> 💡 Two methods of java.util.Observable are **protected** methods: setChanged() and clearChanged().

You can use the following test code:

```java
// Test program for SpecialSubject with a Delegated Observable.
public class TestSpecial {
  public static void main(String args[]) {
    // Create the Subject and Observers.
    SpecialSubject s = new SpecialSubject("Corn Pops", 1.29f);
    NameObserver nameObs = new NameObserver();
    PriceObserver priceObs = new PriceObserver();
    // Add those Observers!
    s.getObservable().addObserver(nameObs);
    s.getObservable().addObserver(priceObs);
    // Make changes to the Subject.
    s.setName("Frosted Flakes");
    s.setPrice(4.57f);
    s.setPrice(9.22f);
    s.setName("Sugar Crispies");
  }
}
```

That should produce the following results:

```
ConcreteSubject created: Corn Pops at 1.29
NameObserver created: Name is null
PriceObserver created: Price is 0.0
PriceObserver: Some other change to subject!
NameObserver: Name changed to Frosted Flakes
PriceObserver: Price changed to 4.57
PriceObserver: Price changed to 9.22
PriceObserver: Some other change to subject!
NameObserver: Name changed to Sugar Crispies
```
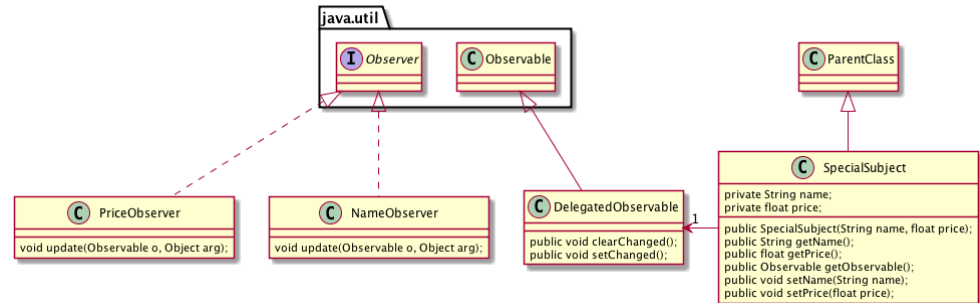
IMPORTANT

Solution

1. Here is one solution:

- Use Delegation

- We will have `SpecialSubject` contain an `Observable` object

- We will delegate the observable behavior that `SpecialSubject` needs to this contained `Observable` object

2. Corresponding class diagram

3. Corresponding Java code:

```
 1 /**
 2  * A subject to observe!
 3  * But this subject already extends another class.
 4  * So use a contained DelegatedObservable object.
 5  * Note that in this version of SpecialSubject we do
 6  * not duplicate any of the interface of Observable.
 7  * Clients must get a reference to our contained
 8  * Observable object using the getObservable() method.
 9  */
10 import java.util.Observable;
11
12 public class SpecialSubject extends ParentClass {
13     private String name;
14     private float price;
15     private DelegatedObservable obs;
16
17     public SpecialSubject(String name, float price) {
18       this.name = name;
19       this.price = price;
20       obs = new DelegatedObservable();
21       System.out.println("ConcreteSubject created: " + name + "
   at " + price);
22     }
23     public String getName() {return name;}
24     public float getPrice() {return price;}
25     public Observable getObservable() {return obs;}
26     public void setName(String string) {
27       name = string;
28       obs.setChanged();
29       obs.notifyObservers(name);
30     }
31     public void setPrice(float f) {
32       price = f;
33       obs.setChanged();
34       obs.notifyObservers(price);
35     }
36 }
```

Explanations:

What's this `DelegatedObservable` class? It implements the two methods of
`java.util.Observable` that are protected methods: `setChanged()` and
`clearChanged()`.

> **ℹ** Apparently, the designers of Observable felt that only
> subclasses of Observable (that is, "true" observable subjects)
> should be able to modify the state-changed flag.

If `SpecialSubject` contains an `Observable` object, it could not invoke the

`setChanged()` and `clearChanged()` methods on it. So we have `DelegatedObservable` extends `Observable` and override these two methods making them have public visibility.

> Java rule: A subclass can changed the visibility of an overridden method of its superclass, but only if it provides more access.

QUESTION

After studying the solution with your favorite teacher:

1. Do you see a problem to this implementation?

2. Can you provide a solution?

Solution

1. The problem: this version of `SpecialSubject` did not provide implementations of any of the methods of `Observable`. As a result, it had to allow its clients to get a reference to its contained `Observable` object using the `getObservable()` method. This may have dangerous consequences. A rogue client could, for example, call the `deleteObservers()` method on the `Observable` object, and delete all the observers! Let's have `SpecialSubject` not expose its contained `Observable` object, but instead provide "wrapper" implementations of the `addObserver()` and `deleteObserver()` methods which simply pass on the request to the contained `Observable` object.

2. The solution

```java
import java.util.Observer;

public class SpecialSubject2 extends ParentClass {
  private String name;
  private float price;
  private DelegatedObservable obs;
  public SpecialSubject2(String name, float price) {
    this.name = name;
    this.price = price;
    obs = new DelegatedObservable();
  }
  public String getName() {return name;}
  public float getPrice() {return price;}
  public void addObserver(Observer o) {
    obs.addObserver(o);
  }
  public void deleteObserver(Observer o) {
    obs.deleteObserver(o);
  }
  public void setName(String name) {
    this.name = name;
    obs.setChanged();
    obs.notifyObservers(name);
  }

  public void setPrice(float price) {
    this.price = price;
    obs.setChanged();
    obs.notifyObservers(new Float(price));
  }
}
```

IMPORTANT

QUESTION

Hence how to you modify the test?

Solution

```
//s.getObservable().addObserver(nameObs);
s.addObserver(nameObs);
//s.getObservable().addObserver(priceObs);
s.addObserver(priceObs);
```

To go further…

QUESTION

- For the `price` and `name` example, we could have decided that only the `PriceObserver` are notified of a price variation an that only the `NameObserver` are notified of a change of name. How would you do that?

- Make some connections with what you hav practiced today and the listeners used in Swing.

IMPORTANT

Solution

- Pas simple à faire car pas possible de "différencier" des types d'observateurs … Et le `notify` est déjà implémenté …

Mais faisable ensuite avec la solution finale et 2 types de add/remove observeurs : add/removeNameObservers et add/removePriceobservers et gérer 2 Observable internes.

- Voir le fichier Observer-suite.txt sur le github pour compléter.