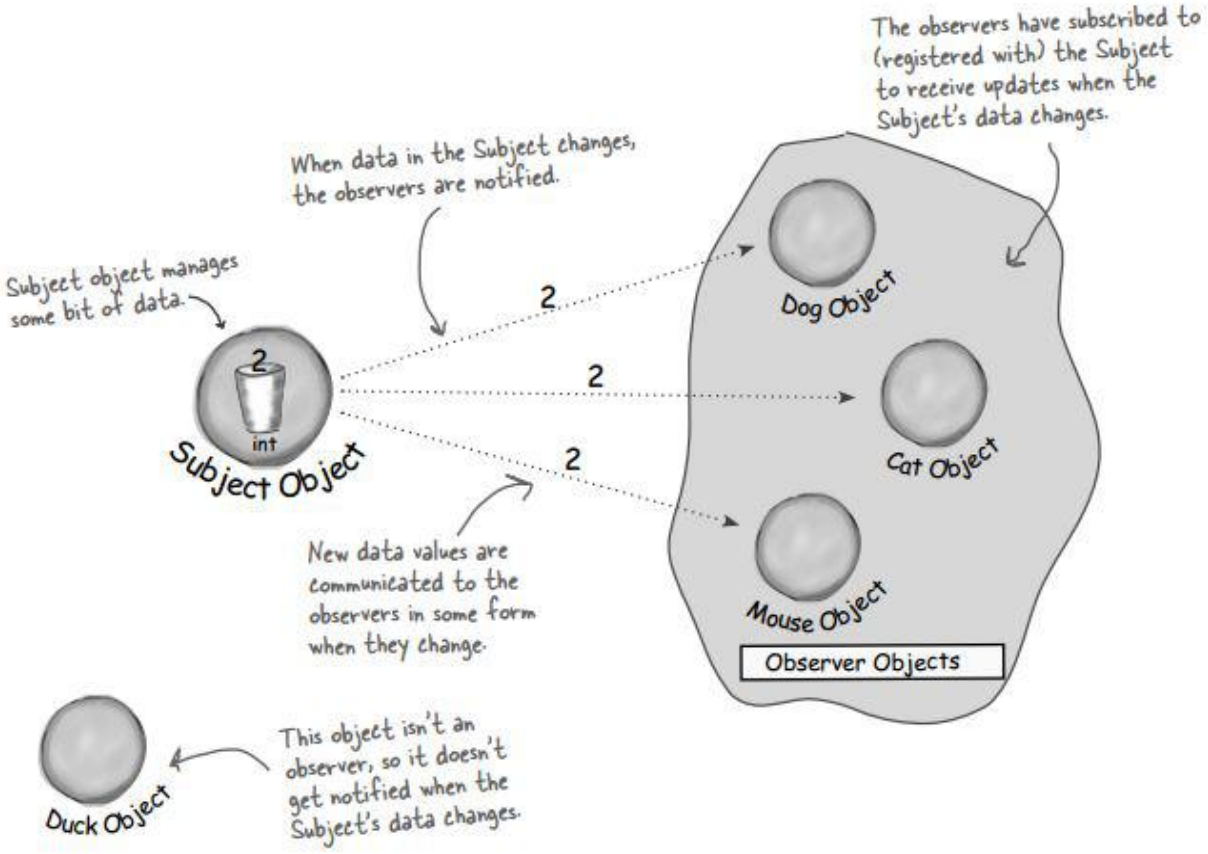
Observer Pattern

Observer Pattern:

You know how newspaper or magazine subscriptions work?

- A newspaper publisher goes into business and begins publishing newspapers.
- You subscribe to a particular publisher, and every time there's a new edition it gets delivered to you. As long as you remain a subscriber, you get new newspapers.
- You unsubscribe when you don't want papers anymore, and they stop being delivered.
- While the publisher remains in business, people, hotels, airlines and other businesses constantly subscribe and unsubscribe to the newspaper.

Publishers + Subscribers = Observer Pattern:
If you understand newspaper subscriptions, you pretty much understand the Observer Pattern, only we call the publisher the SUBJECT and the subscribers the

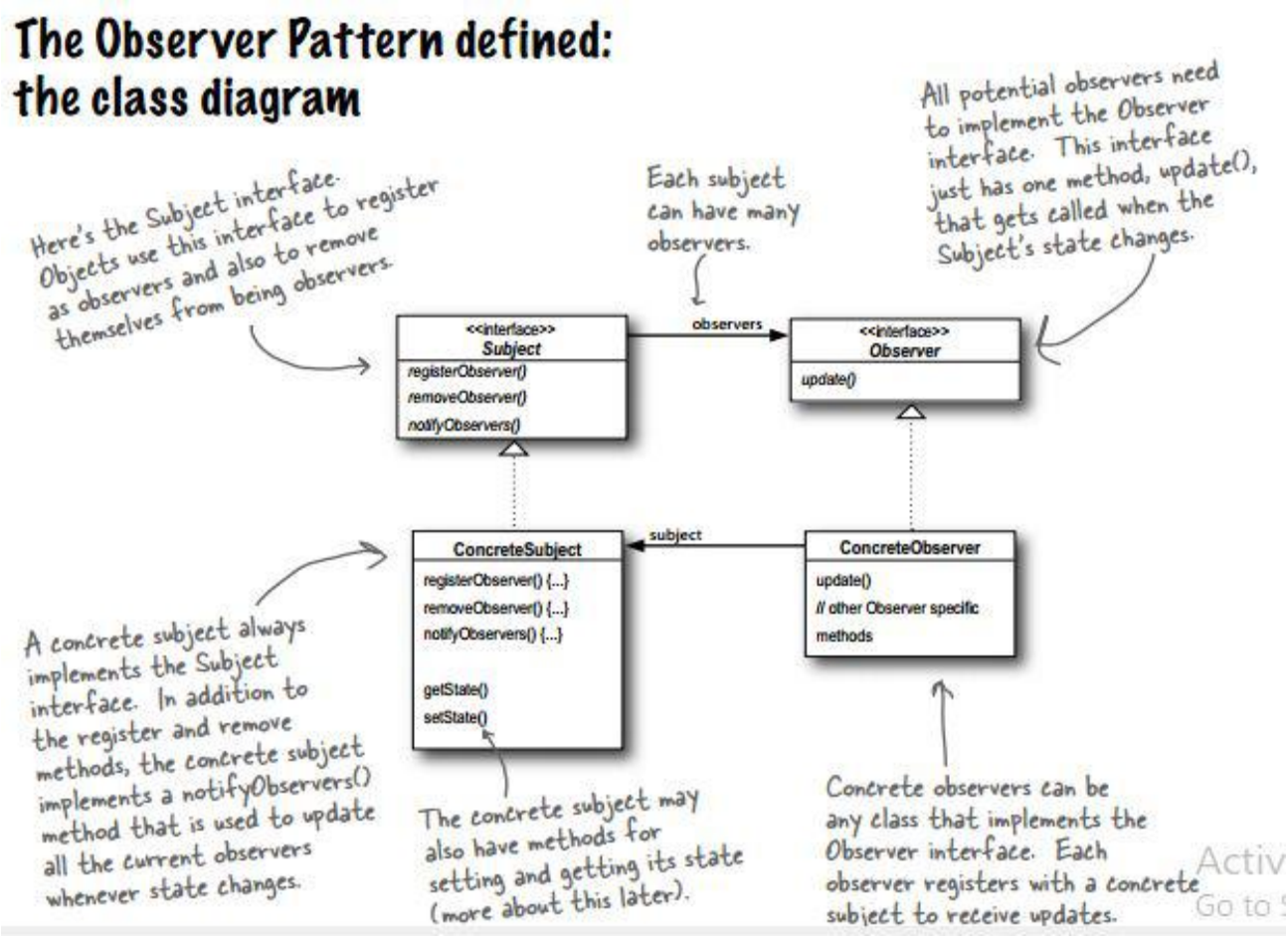


OBSERVERS.

Observer Pattern

Observer Pattern:
The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

The subject and observers define the one-to-many relationship. The observers are dependent on the subject such that when the subject's state changes, the observers get notified. Depending on the style of notification, the observer may also be updated with new values.



Because the subject is the sole owner of that data, the observers are dependent on the subject to update them when the data changes. This leads to a cleaner OO design than allowing many objects to control the same data.

When two objects are loosely coupled, they can interact, but have very little knowledge of each other. The Observer Pattern provides an object design where subjects and observers are loosely coupled.

Observer Pattern

Design Principle :
Strive for loosely coupled designs between objects that interact.

Why?

The only thing the subject knows about an observer is that it implements a certain interface(the Observer interface). It doesn't need to know the concrete class of the observer, what it does, or anything else about it.

We can add new observers at any time. Because the only thing the subject depends on is a list of objects that implement the Observer interface, we can add new observers whenever we want. In fact, we can replace any observer at runtime with another observer and the subject will keep purring along. Likewise, we can remove observers at any time.

We never need to modify the subject to add new types of observers. Let's say we have a new concrete class come along that needs to be an observer. We don't need to make any changes to the subject to accommodate the new class type, all we have to do is implement the Observer interface in the new class and register as an observer. The subject doesn't care; it will deliver notifications to any object that implements the Observer interface.

We can reuse subjects or observers independently of each other. If we have another use for a subject or an observer, we can easily reuse them because the two aren't tightly coupled.

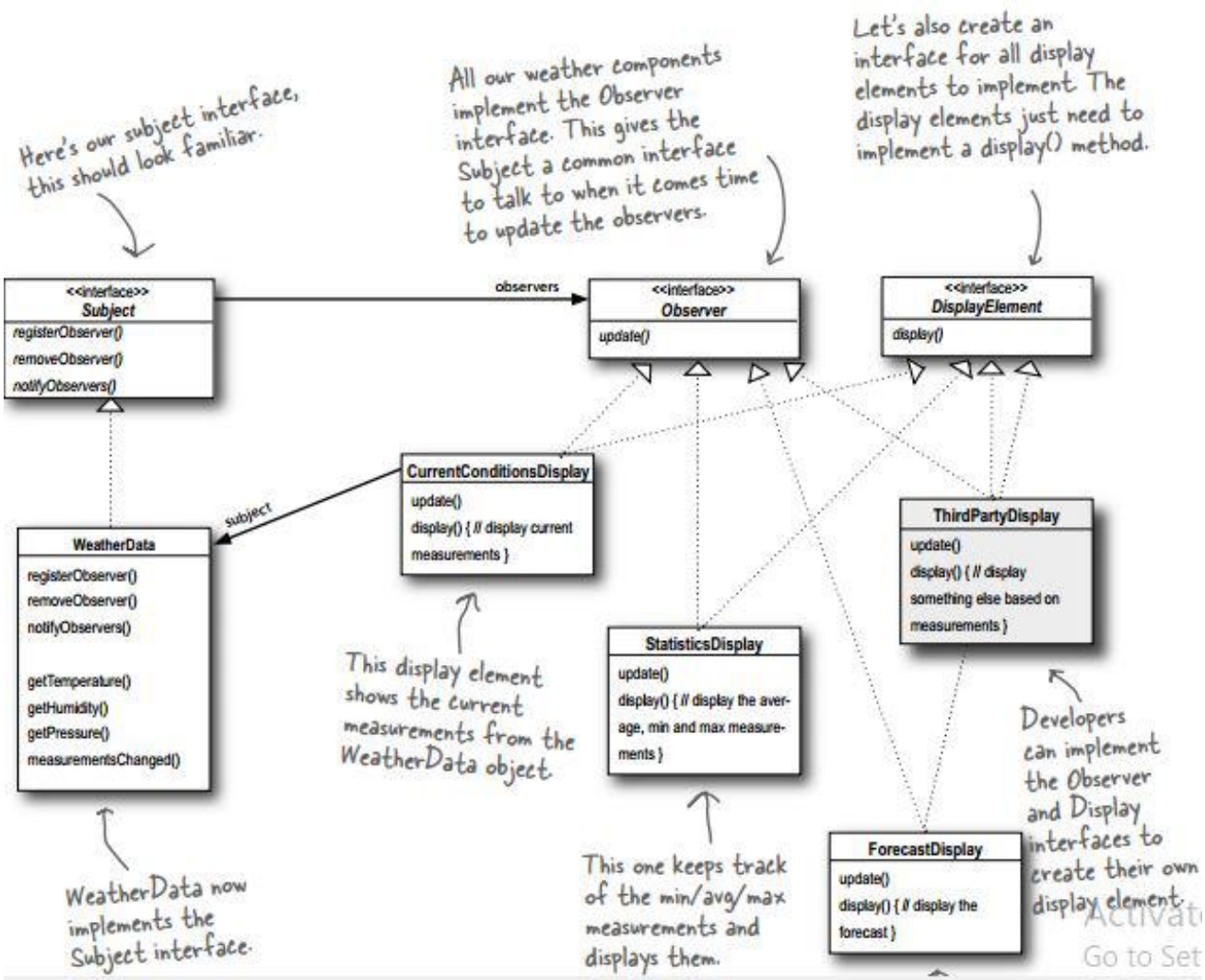
Changes to either the subject or an observer will not affect the other. Because the two are loosely coupled, we are free to make changes to either, as long as the objects still meet their obligations to implement the subject or observer interfaces.

Loosely coupled designs allow us to build flexible OO systems that can handle change because they minimize the inter-dependency between objects.

Designing the Weather Station:
The WeatherData object knows how to talk to the physical Weather Station, to get updated data. The WeatherData object then updates its displays for the three different display elements: Current Conditions (shows temperature, humidity, and pressure), Weather Statistics, and a simple forecast. Current Conditions is one of three different displays. The user can also get weather stats and a forecast.

Our job, if we choose to accept it, is to create an app that uses the WeatherData object to update three displays for current conditions, weather stats, and a forecast.

Observer Pattern



Note:we can use arraylist to store observer objects in subject.

How Java's built-in Observer Pattern works

The built in Observer Pattern works a bit differently than the implementation that we used on the Weather Station. The most obvious difference is that WeatherData (our subject) now extends the Observable class and inherits the add, delete and notify Observer methods (among a few others). Here's how we use Java's version:

For an Object to become an observer...

As usual, implement the Observer interface (this time the java.util.Observer interface) and call addObserver() on any Observable object. Likewise, to remove yourself as an observer just call deleteObserver().

For the Observable to send notifications...

First of all you need to be Observable by extending the java.util.Observable superclass. From there it is a two step process:

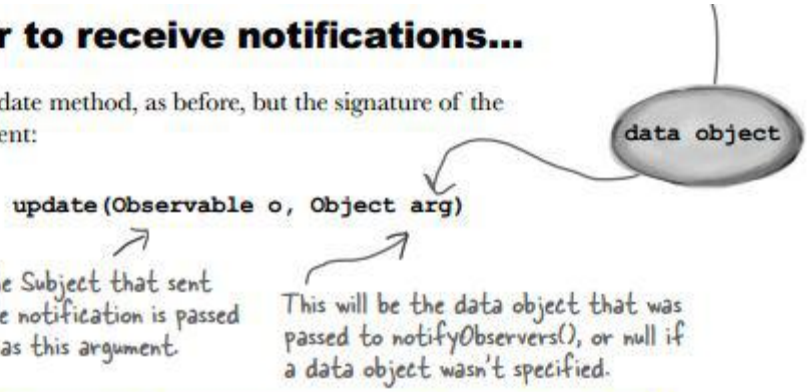
1. **You first must call the setChanged() method to signify that the state has changed in your object**

2. **Then, call one of two notifyObservers() methods:**

either `notifyObservers()` **or** `notifyObservers(Object arg)`

For an Observer to receive notifications...

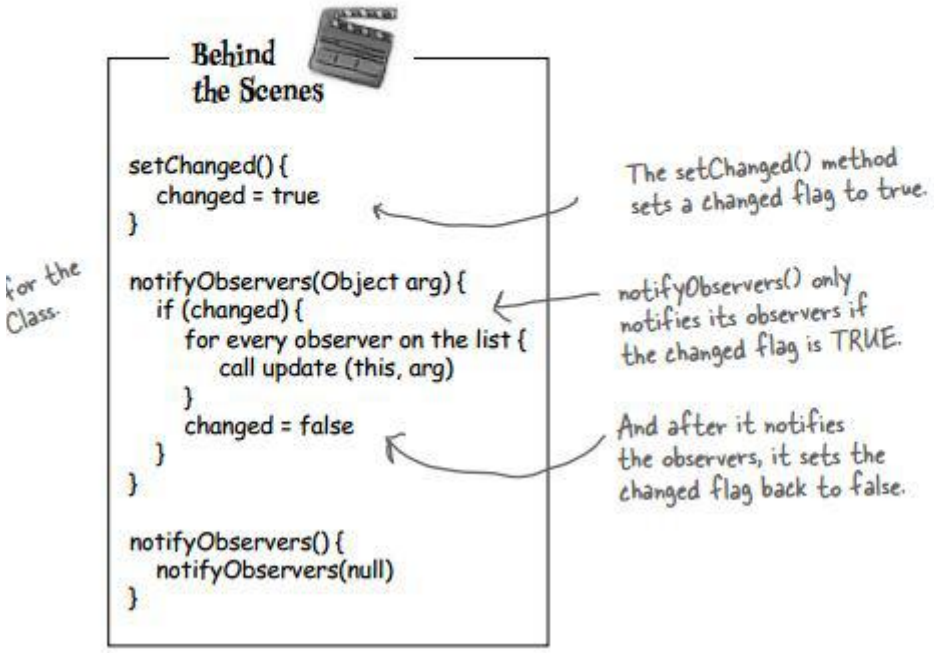
It implements the update method, as before, but the signature of the method is a bit different:



If you want to "push" data to the observers you can pass the data as a data object to the notifyObserver(arg) method. If not, then the Observer has to "pull" the data it wants from the Observable object passed to it. How? Let's rework the Weather Station and you'll see.

Observer Pattern

The setChanged() method is used to signify that the state has changed and that notifyObservers(), when it is called, should update its observers. If notifyObservers() is called without first calling setChanged(), the observers will NOT be notified. Let's take a look behind the scenes of Observable to see how this works:



The dark side of java.util.Observable

Yes, good catch. As you've noticed, Observable is a class, not an interface, and worse, it doesn't even implementan interface. Unfortunately, the java.util.Observable implementation has a number of problems that limit its usefulness and reuse. That's not to say it doesn't provide some utility, but there are some large potholes to watch out for.

Observable protects crucial methods:

If you look at the Observable API, the setChanged() method is protected. So what? Well, this means you can't call setChanged() unless you've subclassed Observable. This means you can't even create an instance of the Observable class and compose it with your own objects, you have to subclass. The design violates a second design principle here…favor composition over inheritance.