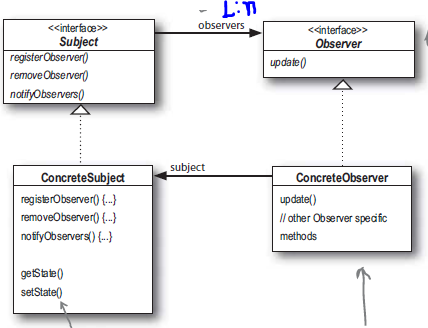
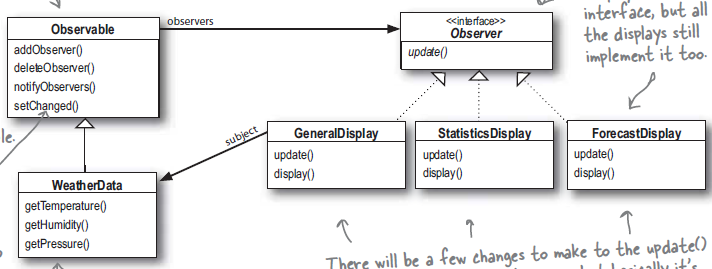
**OBSERVER PATTERN**

1. **Definition:** Observer pattern defines one-to-many dependency between objects such that when one object changes then its dependents are notified and updated automatically.



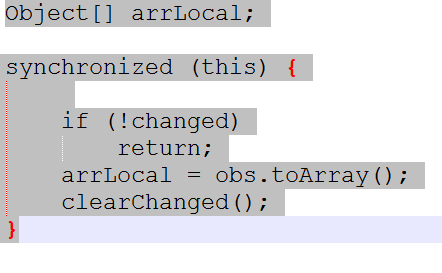
1. Since data state is maintained in subject and while updated calls observer this leads to cleaner OO design than letting many to control same data.
2. **Loose coupling**: Observer pattern provides design where subject and observers are loosely coupled i.e., the only thing Subject knows about observer that it implements Observer interface and the only thing Observer knows about Subject that it implements Subject interface.
3. **Java Applications**:



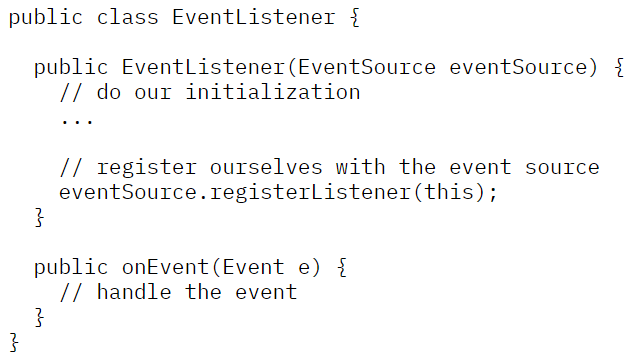
* 1. The setChanged() is there to signify that state has changed and notifyObservers() when it’s called. Its gives more control if state changes frequently but Subject don’t want to notify observers about every event.
  2. Don’t depend on the order of evaluation of observer notification because of the implementation of java.util.Observable.
  3. Observable is class and it doesn’t implement any interface.
     1. So we can’t extend it if our class want to be observable but already subclassed another class.
     2. Observable protect setChanged() method so we need to subclass it to use it. SO composition doesn’t work.
  4. Swing components that are the targets of user interaction fire events when the user interacts with them; data model classes fire events when the data has changed. The use of Observer in this way lets the controller be separated from the model, and the model be separated from the view, simplifying the development of GUI applications.
     1. AWT and Swing components, such as JButton or JTable, use the Observer pattern to decouple the generation of GUI events from their semantics within a given application. Similarly, the Swing model classes, such as TableModel and TreeModel, use Observer to decouple data model representation from view generation, which enables multiple, independent views of the same data.
  5. **Thread Safety**: Frequently, listeners are called in a different thread from which they were registered. To support registering listeners from different threads, whatever mechanism is used to store and manage the list of active listeners must be thread-safe.

Suppose you use a Vector<Listener> to store your list of listeners. While the Vector class is thread-safe, which means that its methods can be called without additional synchronization without risk of corrupting the Vector data structures, iterating a collection involves "check-then-act" sequences, which are at risk of failing if the collection is modified during iteration.

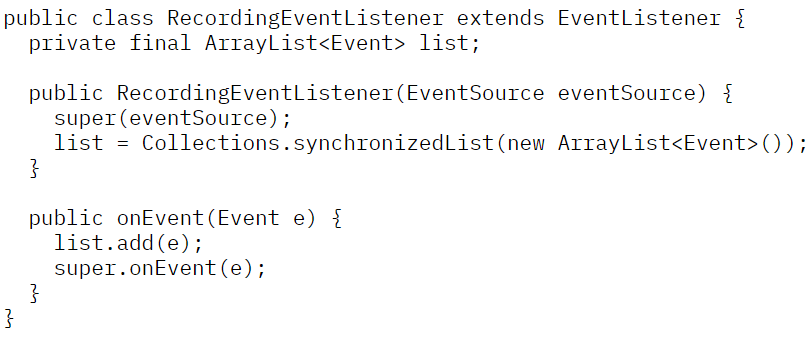
* + 1. One solution to this problem is to hold the lock on the Vector for the duration of iteration. These risks locking out other threads that might want to access the listener list for the duration of iteration.
    2. We can clone the Vector or call its toArray() method to retrieve its contents every time an event occurs. The worst result of any potential race-condition here is that:
       1. a newly-added Observer will miss a notification in progress.
       2. a recently unregistered Observer will be wrongly notified when it doesn't care.



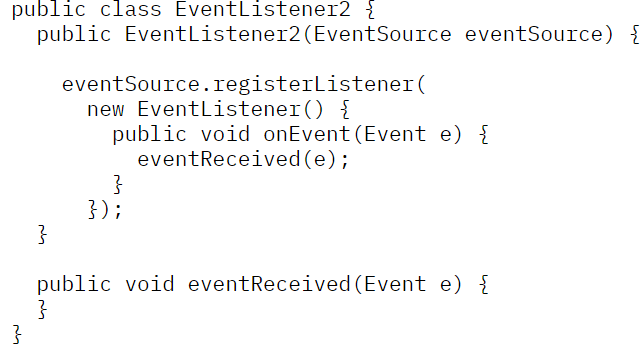
* + 1. If you use an Iterator to traverse the listener list, you'll have the same problem in a slightly different guise; rather than throwing NullPointerException, the iterator() implementation will throw ConcurrentModificationException if it detects that the collection has been modified since iteration began.
    2. The CopyOnWriteArrayList class, in java.util.concurrent, can help prevent this problem. It implements List and is thread-safe, but its iterators will not throw ConcurrentModificationException and do not require any additional locking during traversal.
    3. Initialization Safety: It is tempting to register a listener from its constructor, but it's a temptation you should avoid. Not only does it court the "lapsed listener" problem (which I discuss in a moment), but it creates several thread-safety problems.
       1. The problem is that it lets the "this" reference to the object escape before it is fully constructed.



* + - 1. risk of this approach emerges when the event listener is subclassed: Now, anything done by the subclass constructor happens after the EventListener constructor runs, and therefore after the EventListener has been published, creating race conditions.



* + - 1. Even if your listener class is final and therefore cannot be subclassed, you should still not allow the "this" reference to escape from the constructor -- doing so undermines some safety guarantees provided by the Java Memory Model. It is possible to let the "this" reference escape without the word "this" appearing in your program; publishing a nonstatic inner class instance has the same effect, because an inner class holds a reference to its enclosing object's "this" reference.



* + 1. In general, be aware of what thread event listeners will be called from, and ensure that any objects they touch are either thread-safe or protected with the appropriate synchronization (or thread-confinement, as with Swing model classes) everywhere they are accessed.
    2. Whenever you use the Observer pattern, you are coupling two separate components -- the observer and the observed, which generally have distinct lifecycles. One consequence of registering a listener is that it creates a strong reference from the observed object to the listener -- which prevents the listener (and any objects it references) from being garbage collected until the listener is unregistered.
       1. One approach that is sometimes suggested for dealing with lapsed listeners is to use weak references. While this approach is possible, it's fairly tricky to implement. For it to work, you need to find another object whose lifecycle is exactly the lifecycle of your listener and arrange for it to hold the strong reference to your listener, which is not always easy.
       2. To mitigate the effects of this without necessarily detecting the problem is to use a Set instead of a List for storing listeners; alternately, you could check the List for whether the listener is registered before registering it, and throw an exception (or log an error) if it is, so that evidence of the coding error can be gathered and acted on.
    3. Listener Transgression:
       1. One thing that a listener should *not* do is block for any perceptible amount of time; it has likely been called from an execution context that is expecting to get control back quickly.
       2. Listeners can make trouble for unwary event sources is throwing an unchecked exception. While most of the time, we never intend to throw unchecked exceptions, it does sometimes happen anyway.

When invoking unknown code (which listeners certainly qualify as), it is prudent to execute it in a try-catch block so that ill-behaved listeners do not do more damage than necessary.

* + 1. java.beans, java.util.concurrent , java.util.concurrent.Flow are alternatives of observer pattern.