Observer:

Definition: A Behavioral pattern that defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically

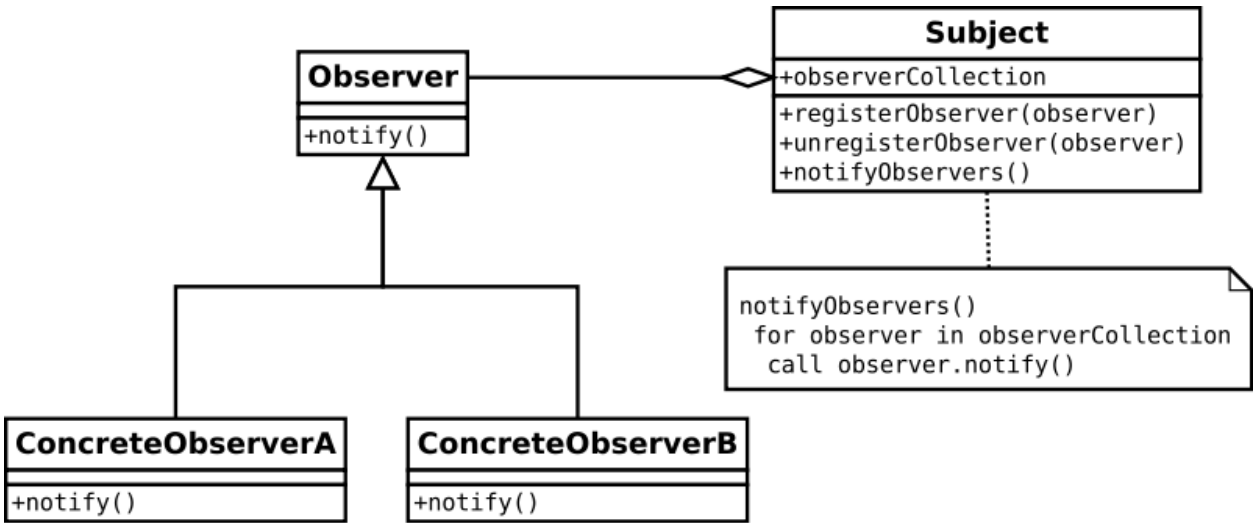
Quick analogy: Newspaper (Spokesman Review) and subscribers
→someone subscribes to paper, they automatically get it when an update (new paper) occurs
→update is handled by the newspaper company (Subject): it distributes the update to its list of subscribers
→a subscriber can unsubscribe – they are removed from subject's list of subscribers

From HeadFirst, we have a WeatherData example where that data needs to be displayed in different ways (CurrentConditions, Statistics, Forecast) whenever the data changes. Note that other displays could be added later and that current displays could be removed later. To allow for easy communication and registering an observer with a subject, create an interface for Subject and interface for Observer. This promotes loose coupling – classes don't have to know about one another other than how to pass messages via interfaces.

Subject: register, remove, notify – concrete class will contain a list of observers
Observer: notify/update – all observers implement this – each concrete observer will contain a reference the subject (part of observer construction includes specifying the subject that will be observed).

In this example, update method passes info as params to observers. This is known as push paradigm. Can also have pull, which requires subject to provide interface for observers to grab the data they want (subject needs to have get methods for the data that has been updated).

UML for pattern:



Note that Java has Observer interface and Observable class built in (java.util). Subject will extend Observable (this places limitations on subject!) – Observers will implement Observer. See Java API for Observer and Observable.

Java implementation of pattern (from Wikipedia).

```
/* File Name : EventSource.java */
//Alex Pantaleev
package obs;

import java.util.Observable;          //Observable is here
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class EventSource extends Observable implements Runnable {
    public void run() {
        try {
            final InputStreamReader isr = new InputStreamReader( System.in );
            final BufferedReader br = new BufferedReader( isr );
            while( true ) {
                String response = br.readLine();
                setChanged();
                notifyObservers( response );
            }
        }
        catch (IOException e) {
            e.printStackTrace();
        }
    }
}
/* File Name: ResponseHandler.java */

package obs;

import java.util.Observable;
import java.util.Observer;  /* this is Event Handler */

public class ResponseHandler implements Observer {
    private String resp;
    public void update (Observable obj, Object arg) {
        if (arg instanceof String) {
```

```
            resp = (String) arg;
            System.out.println("\nReceived Response: "+ resp );
        }
    }
}
/* Filename : MyApp.java */
/* This is the main program */

package obs;

public class MyApp {
    public static void main(String args[]) {
        System.out.println("Enter Text >");

        // create an event source - reads from stdin
        final EventSource evSrc = new EventSource();

        // create an observer
        final ResponseHandler respHandler = new ResponseHandler();

        // subscribe the observer to the event source
        evSrc.addObserver( respHandler );

        // starts the event thread
        Thread thread = new Thread(evSrc);
        thread.start();
    }
}
```

Java uses Observer for much of its GUI (Swing). Listeners are observers. See HeadFirst example of this (SwingObserverExample).