Observer Pattern

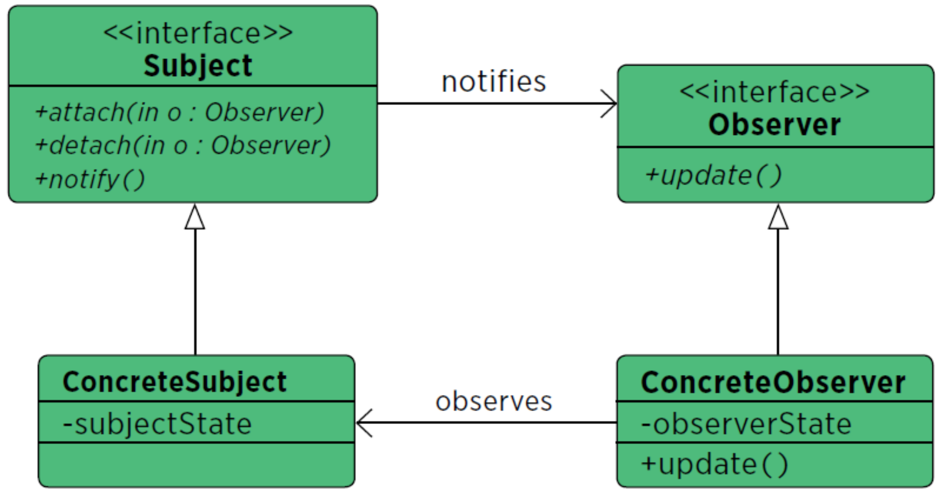
For the Complete Code, See the "Official" Head-First Design Patterns GitHub Repo:

https://github.com/bethrobson/Head-First-Design-Patterns/tree/master/src/headfirst/designpatterns/

And the course SVN repo:

svn://cosc436.net:65436/Examples/trunk

OBSERVER — Object Behavioral



Purpose	Lets one or more objects be notified of state changes in other objects within the system.
Use When	• State changes in one or more objects should trigger behavior in other objects • Broadcasting capabilities are required. • An understanding exists that objects will be blind to the expense of notification.
Example	This pattern can be found in almost every GUI environment. When buttons, text, and other fields are placed in applications the application typically registers as a listener for those controls. When a user triggers an event, such as clicking a button, the control iterates through its registered observers and sends a notification to each.

The Observer Pattern

Hey Jerry, I'm notifying everyone that the Patterns Group meeting moved to Saturday night. We're going to be talking about the Observer Pattern. That pattern is the best! It's the BEST, Jerry!

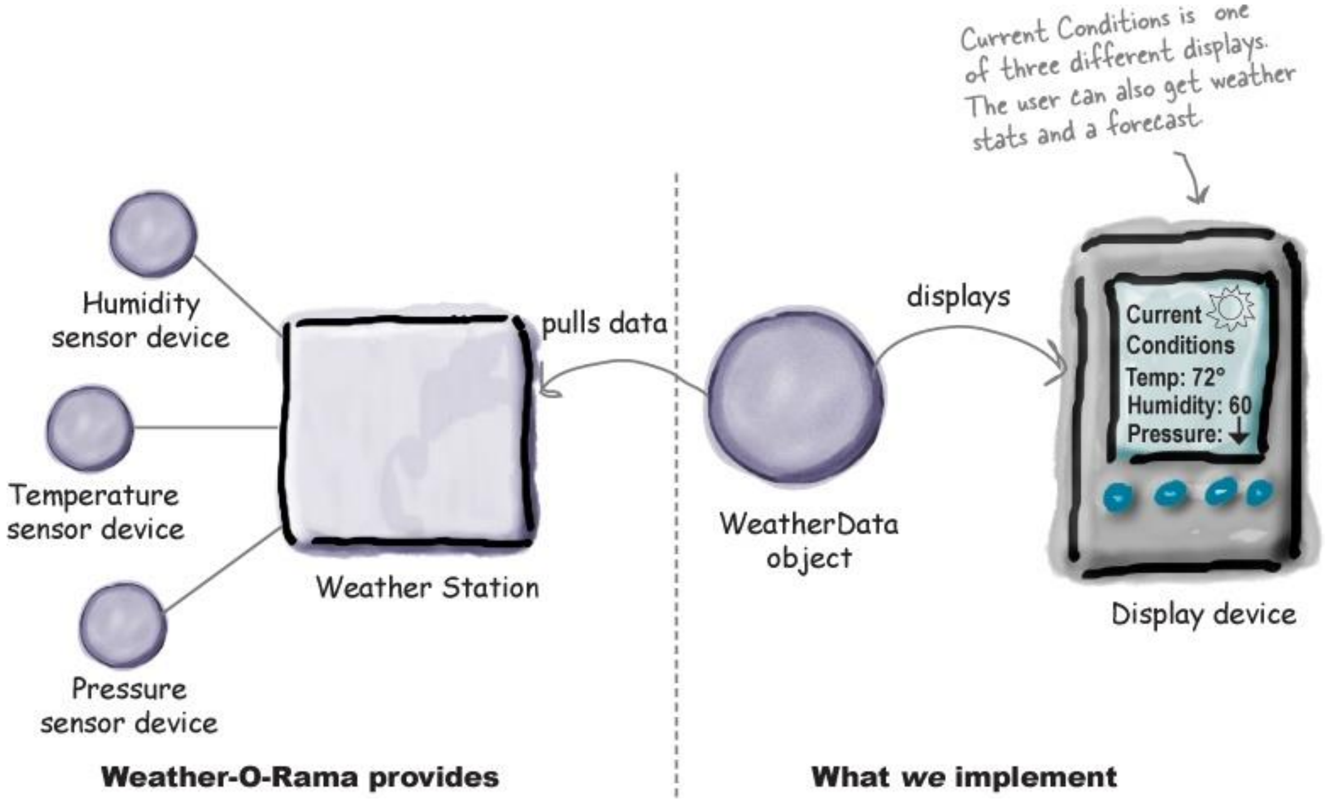
Don’t miss out when something interesting happens!

We’ve got a pattern that keeps your objects in the know when something they might care about happens.

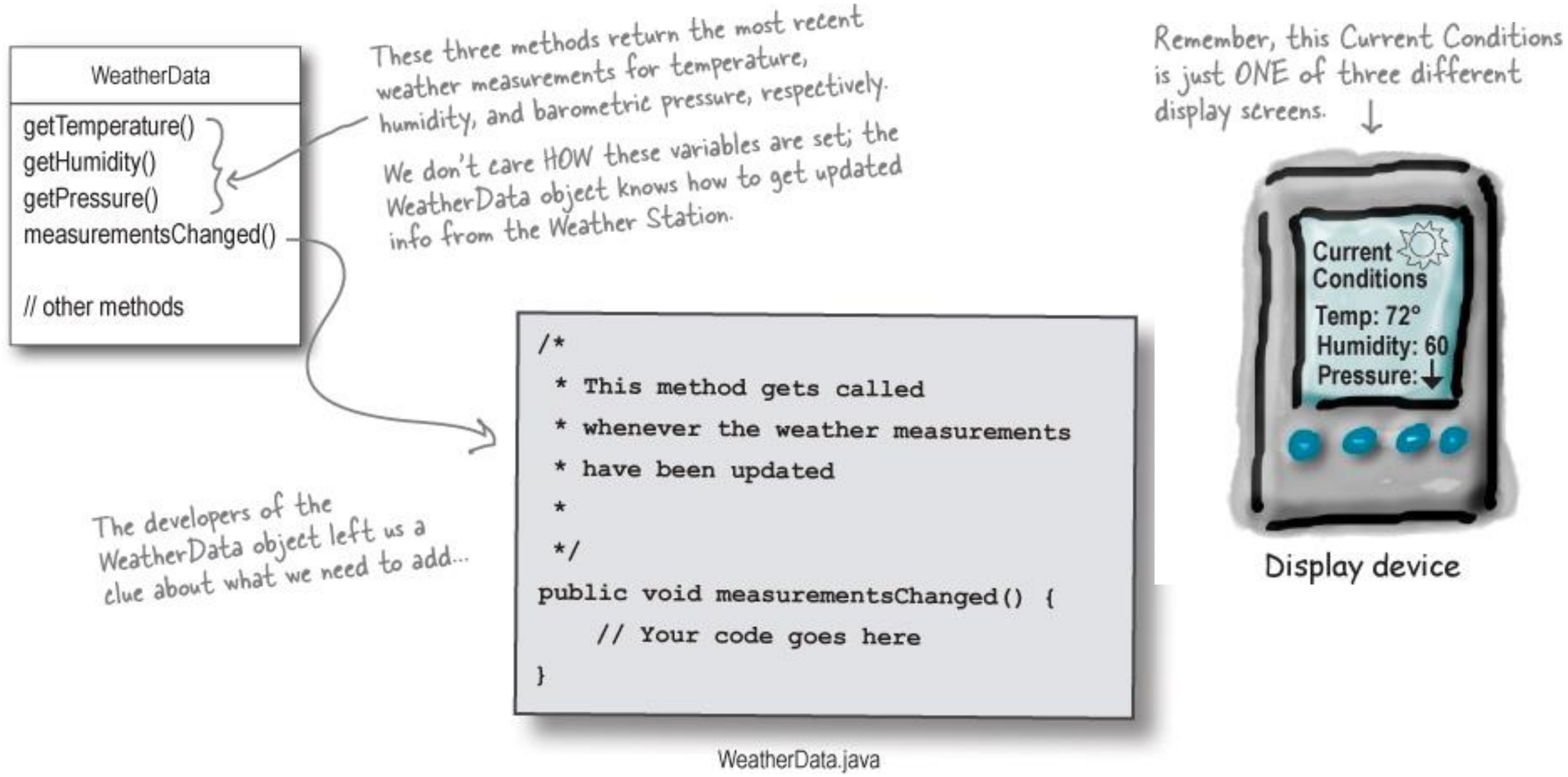
The Observer Pattern is one of the most heavily used patterns in the JDK, and it’s incredibly useful.

Build next-generation, Internet-based Weather Monitoring Station

The three parts in the system are the **weather station** (the physical device that acquires the actual weather data), the **WeatherData** object (that tracks the data coming from the Weather Station and updates the displays), and the **display** that shows users the current weather conditions.



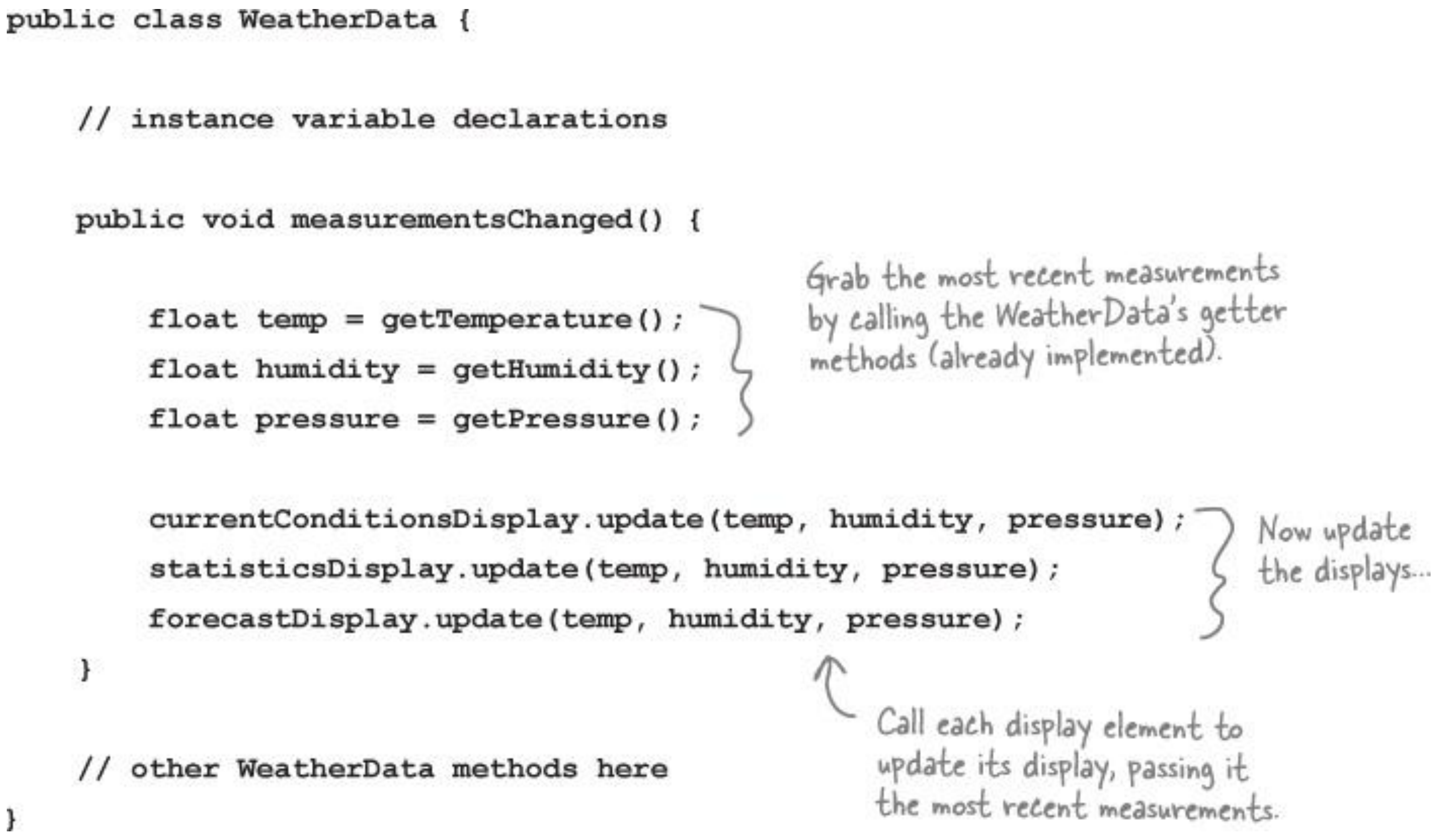
Unpacking the WeatherData class received from client


WeatherData
getTemperature()
getHumidity()
getPressure()
measurementsChanged()
// other methods
These three methods return the most recent weather measurements for temperature, humidity, and barometric pressure, respectively.
We don't care HOW these variables are set; the WeatherData object knows how to get updated info from the Weather Station.
The developers of the WeatherData object left us a clue about what we need to add...
Remember, this Current Conditions is just ONE of three different display screens.
Current Conditions
Temp: 72°
Humidity: 60
Pressure:
Display device
WeatherData.java

```
/*
 * This method gets called
 * whenever the weather measurements
 * have been updated
 *
 */
public void measurementsChanged() {
    // Your code goes here
}
```

Our job is to implement **measurementsChanged**() so that it updates the three displays for current conditions, weather stats, and forecast.

Possible implementation???



```
public class WeatherData {

    // instance variable declarations

    public void measurementsChanged() {

        float temp = getTemperature();
        float humidity = getHumidity();
        float pressure = getPressure();

        currentConditionsDisplay.update(temp, humidity, pressure);
        statisticsDisplay.update(temp, humidity, pressure);
        forecastDisplay.update(temp, humidity, pressure);
    }

    // other WeatherData methods here
}
```

Grab the most recent measurements by calling the WeatherData's getter methods (already implemented).

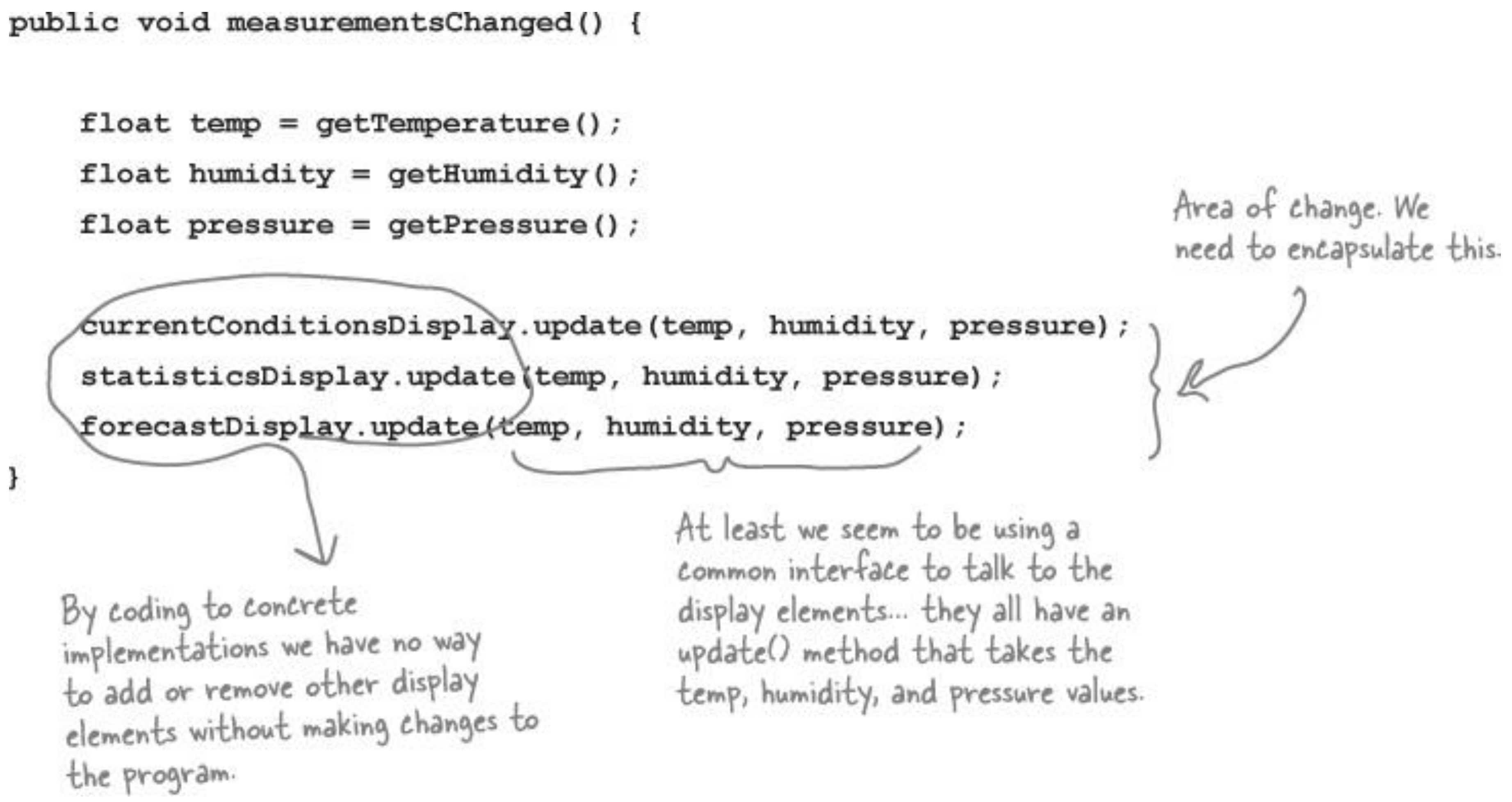
Now update the displays...

Call each display element to update its display, passing it the most recent measurements.

What is the problem?

- We are coding to concrete implementations, not interfaces.
- For every new display element we need to alter code.
- We have no way to add (or remove) display elements at run time.
- We haven’t encapsulated the part that changes.

What is the problem?



```
public void measurementsChanged() {

    float temp = getTemperature();
    float humidity = getHumidity();
    float pressure = getPressure();

    currentConditionsDisplay.update(temp, humidity, pressure);
    statisticsDisplay.update(temp, humidity, pressure);
    forecastDisplay.update(temp, humidity, pressure);
}
```

Area of change. We need to encapsulate this.

By coding to concrete implementations we have no way to add or remove other display elements without making changes to the program.

At least we seem to be using a common interface to talk to the display elements... they all have an update() method that takes the temp, humidity, and pressure values.

Like old-fashioned print magazine subscriptions.

- 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 [the Irony of technology making obsolete the metaphors used to describe the technology - ajc], other businesses constantly subscribe and unsubscribe to the newspaper.

Miss what’s going on in Objectville? No way, of course we subscribe!

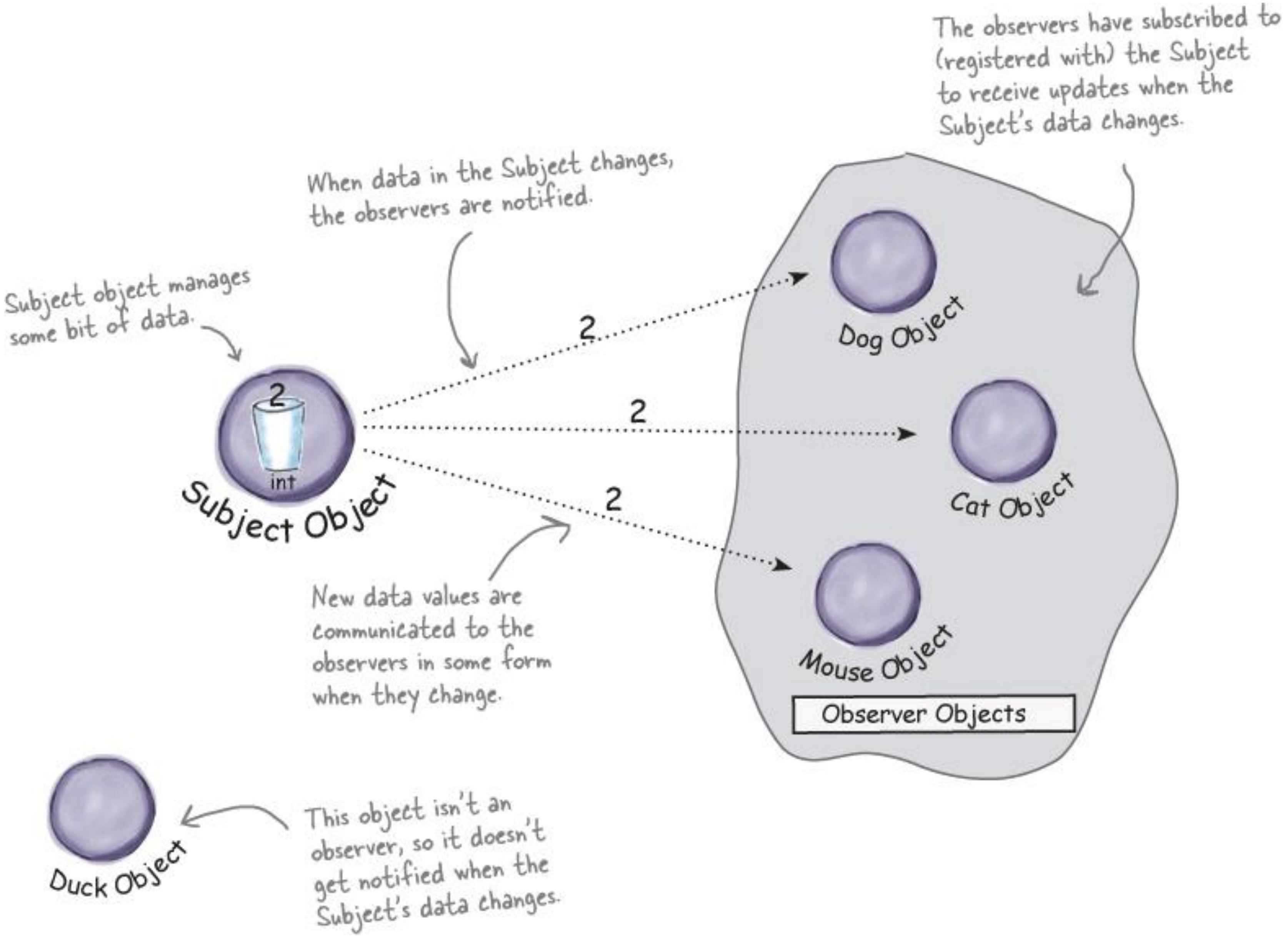
Observer Design Pattern

Problem
A large monolithic design does not scale well as new graphing or monitoring requirements are levied.

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

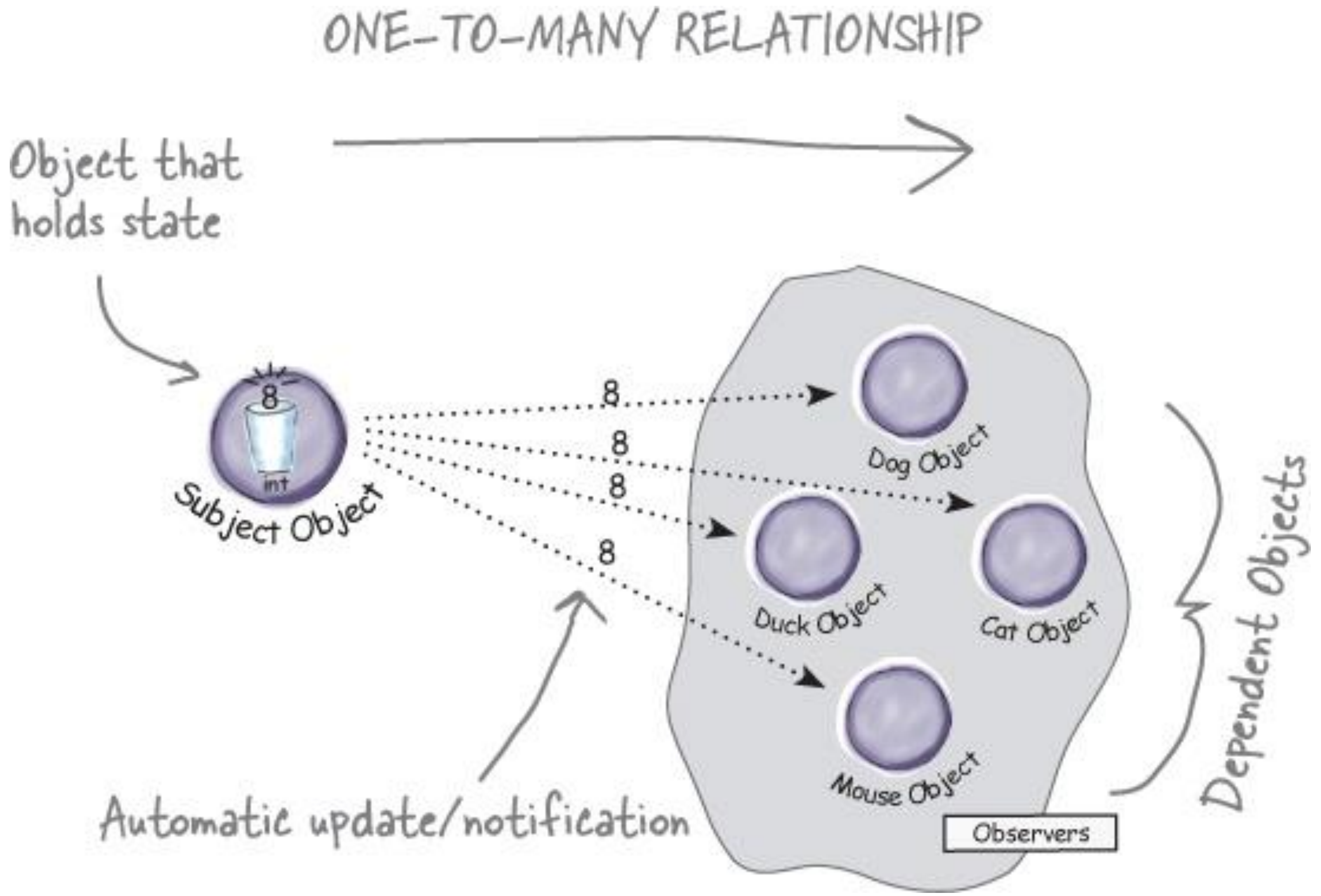
Encapsulate the core (or common or engine) components in a **Subject abstraction**, and the variable (or optional or user interface) components in an Observer hierarchy.

Publishers + Subscribers = Observer Pattern

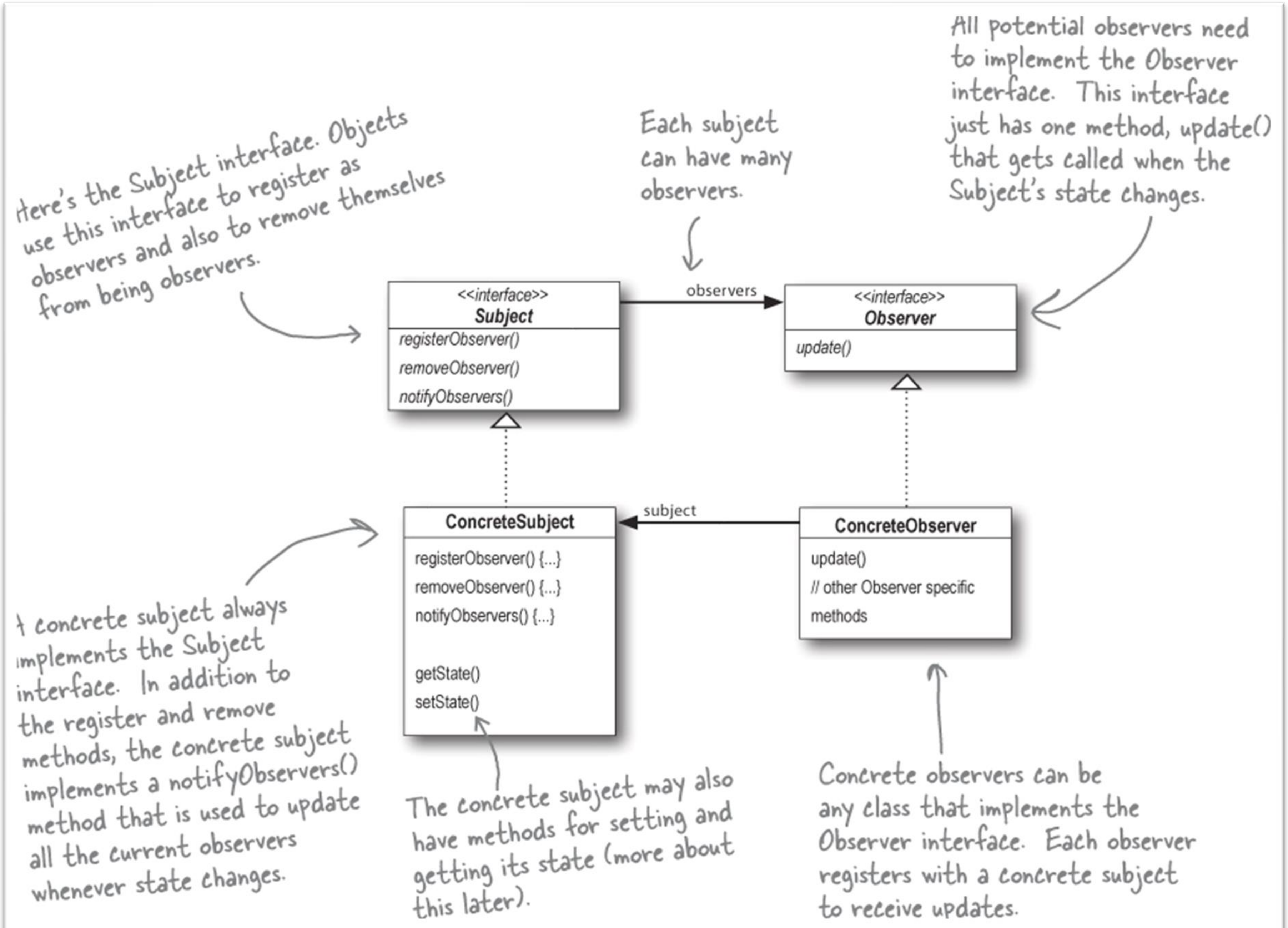

The observers have subscribed to (registered with) the Subject to receive updates when the Subject's data changes.
When data in the Subject changes, the observers are notified.
Subject object manages some bit of data.
2
int
Subject Object
2
2
2
Dog Object
Cat Object
Mouse Object
Observer Objects
New data values are communicated to the observers in some form when they change.
Duck Object
This object isn't an observer, so it doesn't get notified when the Subject's data changes.

Publishers + Subscribers = 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.



Observer Pattern


Here's the Subject interface. Objects use this interface to register as observers and also to remove themselves from being observers.
Each subject can have many observers.
All potential observers need to implement the Observer interface. This interface just has one method, update() that gets called when the Subject's state changes.
<<interface>>
Subject
registerObserver()
removeObserver()
notifyObservers()
observers
<<interface>>
Observer
update()
ConcreteSubject
registerObserver() {...}
removeObserver() {...}
notifyObservers() {...}
getState()
setState()
subject
ConcreteObserver
update()
// other Observer specific methods
A concrete subject always implements the Subject interface. In addition to the register and remove methods, the concrete subject implements a notifyObservers() method that is used to update all the current observers whenever state changes.
The concrete subject may also have methods for setting and getting its state (more about this later).
Concrete observers can be any class that implements the Observer interface. Each observer registers with a concrete subject to receive updates.

The power of Loose Coupling

When two objects are loosely coupled, they can interact, but have very little knowledge of each other.

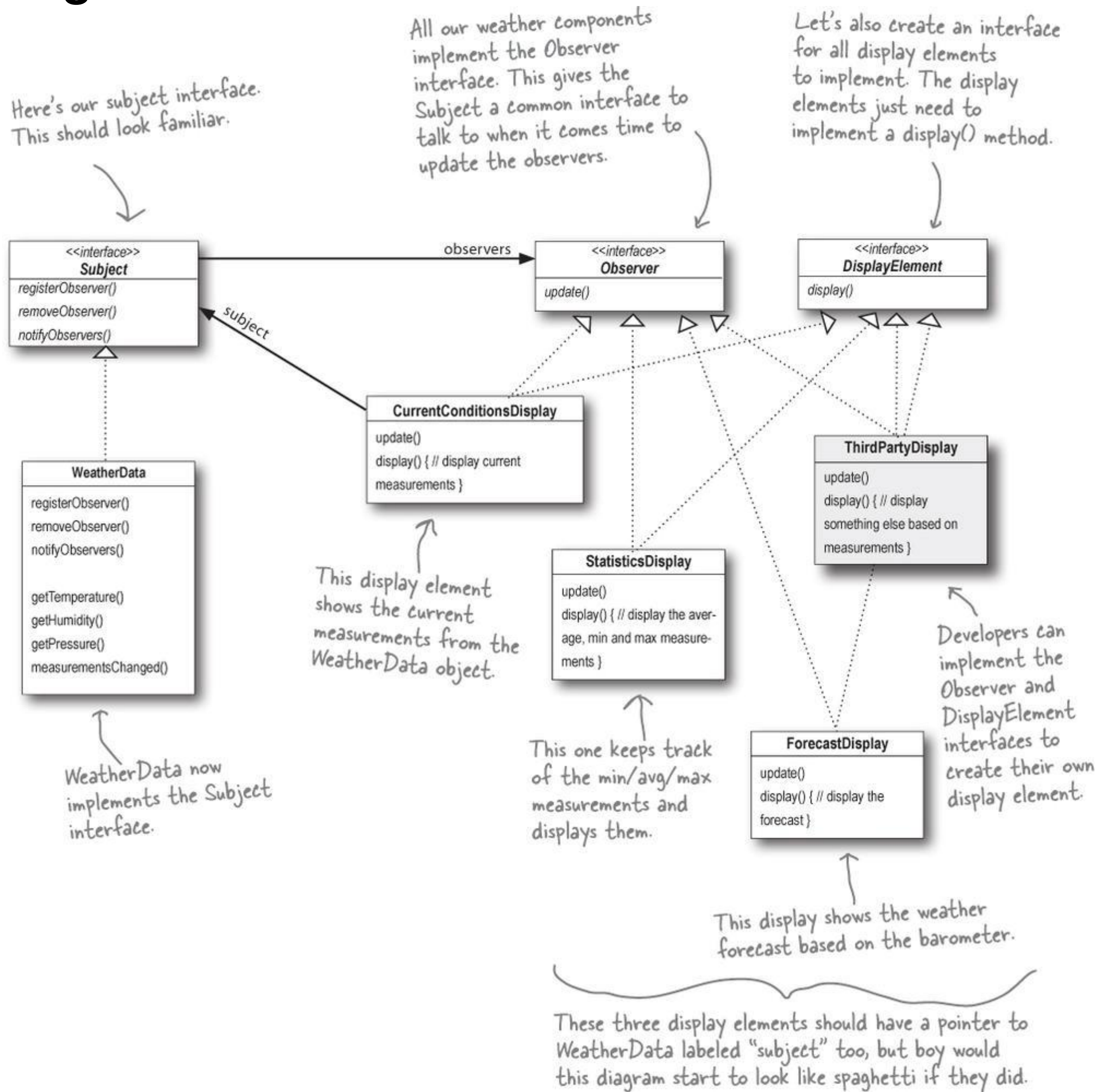
The Observer Pattern provides an object design where subjects and observers are loosely coupled.

Why?
The only thing the subject knows about an observer is that it implements a certain interface (the Observer interface).
We can add new observers at any time.
We never need to modify the subject to add new types of observers.
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.

DESIGN PRINCIPLE

Strive for loosely coupled designs between objects that interact.

Designing the Weather Station



Here's our subject interface. This should look familiar.
All our weather components implement the Observer interface. This gives the Subject a common interface to talk to when it comes time to update the observers.
Let's also create an interface for all display elements to implement. The display elements just need to implement a display() method.
<<interface>> Subject
registerObserver()
removeObserver()
notifyObservers()
observers
<<interface>> Observer
update()
<<interface>> DisplayElement
display()
subject
CurrentConditionsDisplay
update()
display() { // display current measurements }
WeatherData
registerObserver()
removeObserver()
notifyObservers()
getTemperature()
getHumidity()
getPressure()
measurementsChanged()
ThirdPartyDisplay
update()
display() { // display something else based on measurements }
StatisticsDisplay
update()
display() { // display the average, min and max measurements }
ForecastDisplay
update()
display() { // display the forecast }
This display element shows the current measurements from the WeatherData object.
WeatherData now implements the Subject interface.
This one keeps track of the min/avg/max measurements and displays them.
Developers can implement the Observer and DisplayElement interfaces to create their own display element.
This display shows the weather forecast based on the barometer.
These three display elements should have a pointer to WeatherData labeled "subject" too, but boy would this diagram start to look like spaghetti if they did.

Implementing the Weather Station

```
public interface Subject {
    public void registerObserver(Observer o);
    public void removeObserver(Observer o);
    public void notifyObservers();
}
```

Both of these methods take an Observer as an argument; that is, the Observer to be registered or removed.

This method is called to notify all observers when the Subject's state has changed.

```
public interface Observer {
    public void update(float temp, float humidity, float pressure);
}
```

These are the state values the Observers get from the Subject when a weather measurement changes

The Observer interface is implemented by all observers, so they all have to implement the update() method. Here we're following Mary and Sue's lead and passing the measurements to the observers.

```
public interface DisplayElement {
    public void display();
}
```

The DisplayElement interface just includes one method, display(), that we will call when the display element needs to be displayed.

```java
public class WeatherData implements Subject {
    private ArrayList<Observer> observers;
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() {
        observers = new ArrayList<Observer>();
    }

    public void registerObserver(Observer o) {
        observers.add(o);
    }

    public void removeObserver(Observer o) {
        int i = observers.indexOf(o);
        if (i >= 0) {
            observers.remove(i);
        }
    }

    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(temperature, humidity, pressure);
        }
    }

    public void measurementsChanged() {
        notifyObservers();
    }

    public void setMeasurements(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
    }

    // other WeatherData methods here
}
```

WeatherData now implements the Subject interface.

We've added an ArrayList to hold the Observers, and we create it in the constructor.

Here we implement the Subject interface.

When an observer registers, we just add it to the end of the list.

Likewise, when an observer wants to un-register, we just take it off the list.

Here's the fun part; this is where we tell all the observers about the state. Because they are all Observers, we know they all implement update(), so we know how to notify them.

We notify the Observers when we get updated measurements from the Weather Station.

Okay, while we wanted to ship a nice little weather station with each book, the publisher wouldn't go for it. So, rather than reading actual weather data off a device, we're going to use this method to test our display elements. Or, for fun, you could write code to grab measurements off the Web.

This display implements Observer so it can get changes from the WeatherData object.

It also implements DisplayElement, because our API is going to require all display elements to implement this interface.

```java
public class CurrentConditionsDisplay implements Observer, DisplayElement {
    private float temperature;
    private float humidity;
    private Subject weatherData;

    public CurrentConditionsDisplay(Subject weatherData) {
        this.weatherData = weatherData;
        weatherData.registerObserver(this);
    }

    public void update(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        display();
    }

    public void display() {
        System.out.println("Current conditions: " + temperature
            + "F degrees and " + humidity + "% humidity");
    }
}
```

The constructor is passed the weatherData object (the Subject) and we use it to register the display as an observer.

When update() is called, we save the temp and humidity and call display().

The display() method just prints out the most recent temp and humidity.

Get the Weather Station running

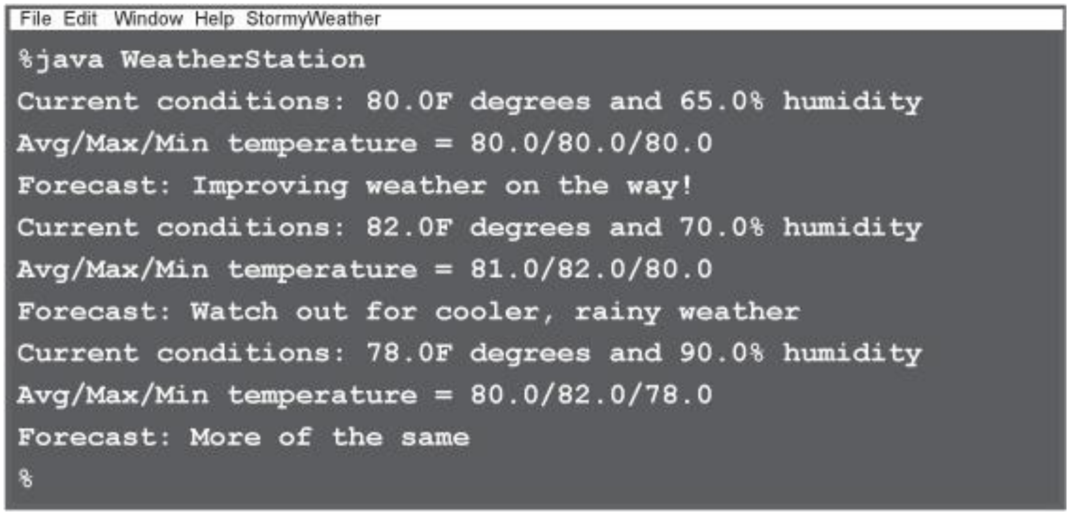
```
public class WeatherStation {

    public static void main(String[] args) {
        WeatherData weatherData = new WeatherData();

        CurrentConditionsDisplay currentDisplay =
            new CurrentConditionsDisplay(weatherData);
        StatisticsDisplay statisticsDisplay = new StatisticsDisplay(weatherData);
        ForecastDisplay forecastDisplay = new ForecastDisplay(weatherData);

        weatherData.setMeasurements(80, 65, 30.4f);
        weatherData.setMeasurements(82, 70, 29.2f);
        weatherData.setMeasurements(78, 90, 29.2f);
    }
}
```

First, create the WeatherData object.

If you don't want to download the code, you can comment out these two lines and run it.

Create the three displays and pass them the WeatherData object.

Simulate new weather measurements.



```
%java WeatherStation
Current conditions: 80.0F degrees and 65.0% humidity
Avg/Max/Min temperature = 80.0/80.0/80.0
Forecast: Improving weather on the way!
Current conditions: 82.0F degrees and 70.0% humidity
Avg/Max/Min temperature = 81.0/82.0/80.0
Forecast: Watch out for cooler, rainy weather
Current conditions: 78.0F degrees and 90.0% humidity
Avg/Max/Min temperature = 80.0/82.0/78.0
Forecast: More of the same
%
```

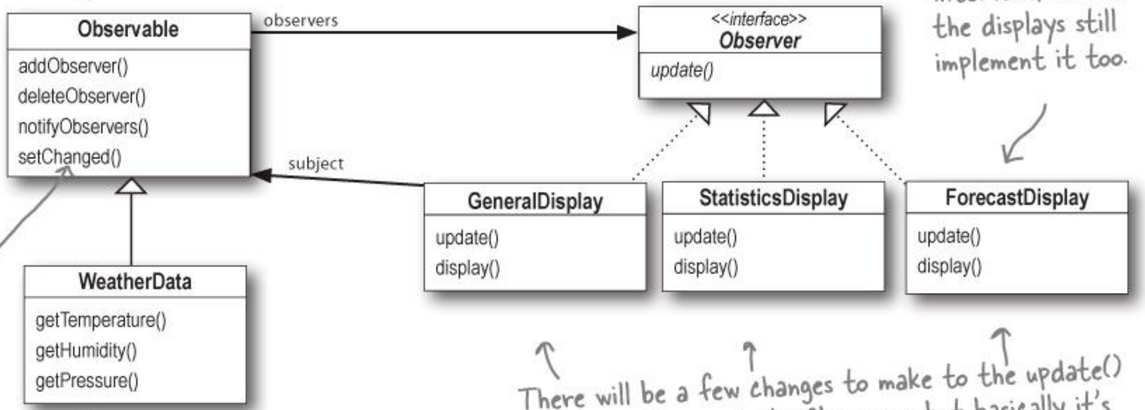
Java's built-in java.util.Observer and java.util.Observable

The Observable class keeps track of all your observers and notifies them for you.

This should look familiar. In fact, it's exactly the same as our previous class diagram!

We left out the DisplayElement interface, but all the displays still implement it too.

Observable is a CLASS, not an interface, so WeatherData extends Observable.



This doesn't look familiar! Hold tight, we'll get to this in a sec...

Here's our Subject, which we can now also call the Observable. We don't need the register(), remove(), and notifyObservers() methods anymore; we inherit that behavior from the superclass.

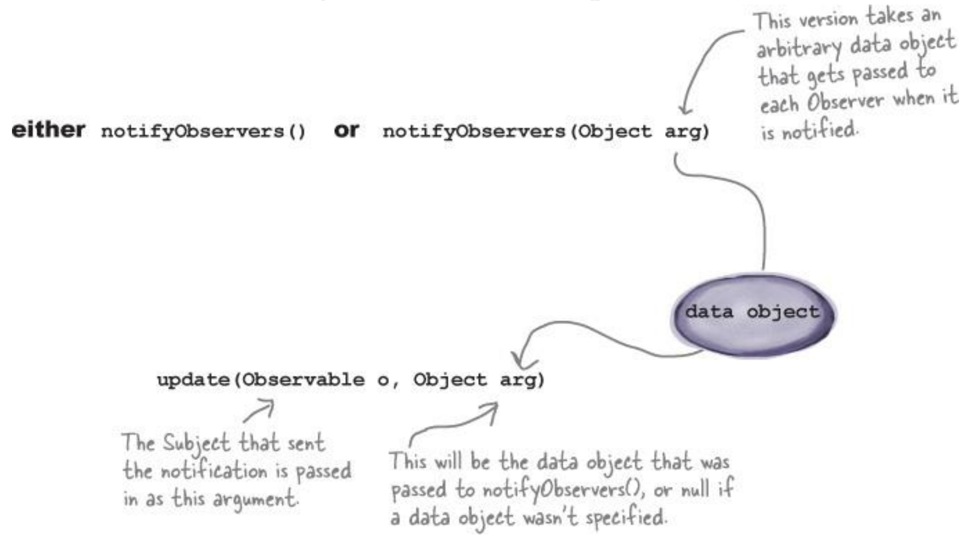
There will be a few changes to make to the update() method in the concrete Observers, but basically it's the same idea... we have a common Observer interface, with an update() method that's called by the Subject.

With Java's built-in support, all you have to do is extend Observable and tell it when to notify the Observers. The API does the rest for you.

How Java’s built-in Observer Pattern works

For an Object to become an observer...
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 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:


This version takes an arbitrary data object that gets passed to each Observer when it is notified.
either notifyObservers() or notifyObservers(Object arg)
data object
update(Observable o, Object arg)
The Subject that sent the notification is passed in as this argument.
This will be the data object that was passed to notifyObservers(), or null if a data object wasn't specified.

Reworking the Weather Station with the built-in support

1 Make sure we are importing the right Observable.

2 We are now subclassing Observable.

3 We don't need to keep track of our observers anymore, or manage their registration and removal (the superclass will handle that), so we've removed the registerObserver(), removeObserver() and notifyObservers() methods.

```
import java.util.Observable;

public class WeatherData extends Observable {
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() { }

    public void measurementsChanged() {
        setChanged();
        notifyObservers(); *
    }

    public void setMeasurements(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
    }

    public float getTemperature() {
        return temperature;
    }

    public float getHumidity() {
        return humidity;
    }

    public float getPressure() {
        return pressure;
    }
}
```

4 Our constructor no longer needs to create a data structure to hold Observers.

* Notice we aren't sending a data object with the notifyObservers() call. That means we're using the PULL model.

5 We now first call setChanged() to indicate the state has changed before calling notifyObservers().

6 These methods aren't new, but because we are going to use "pull" we thought we'd remind you they are here. The Observers will use them to get at the WeatherData object's state.

Reworking the Weather Station with the built-in support

1 Again, make sure we are importing the right Observer/Observable.

2 We now are implementing the Observer interface from java.util.

```
import java.util.Observable;
import java.util.Observer;

public class CurrentConditionsDisplay implements Observer, DisplayElement {
    Observable observable;
    private float temperature;
    private float humidity;

    public CurrentConditionsDisplay(Observable observable) {
        this.observable = observable;
        observable.addObserver(this);
    }

    public void update(Observable obs, Object arg) {
        if (obs instanceof WeatherData) {
            WeatherData weatherData = (WeatherData)obs;
            this.temperature = weatherData.getTemperature();
            this.humidity = weatherData.getHumidity();
            display();
        }
    }

    public void display() {
        System.out.println("Current conditions: " + temperature
            + "F degrees and " + humidity + "% humidity");
    }
}
```

3 Our constructor now takes an Observable and we use this to add the current conditions object as an Observer.

4 We've changed the update() method to take both an Observable and the optional data argument.

5 In update(), we first make sure the observable is of type WeatherData and then we use its getter methods to obtain the temperature and humidity measurements. After that we call display().

Other places you'll find the Observer Pattern in the JDK

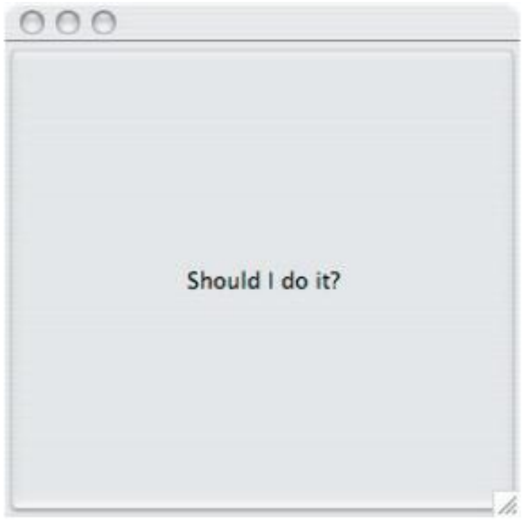
Both JavaBeans and Swing also provide their own implementations of the pattern

A simple example in the Swing API, the JButton.
It has a lot of add/remove listener methods. These methods allow you to add and remove **observers**, or, as they are called in Swing, **listeners**, to listen for various types of events that occur on the Swing component.

For instance, an **ActionListener** lets you "listen in" on any types of **actions** that might occur on a button, like a button press.

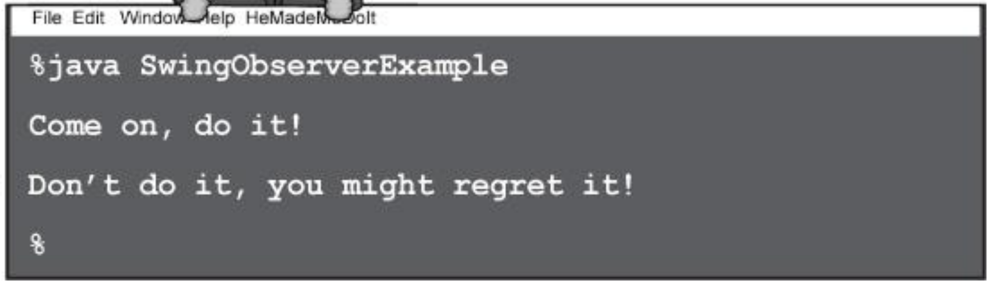
Other places you'll find the Observer Pattern in the JDK

You've got a button that says "Should I do it?" and when you click on that button the listeners (observers) get to answer the question in any way they want. We're implementing two such listeners, called the **AngelListener** and the **DevilListener**. Here's how the application behaves:



Here's our fancy interface.

And here's the output when we click on the button.



```
%java SwingObserverExample
Come on, do it!
Don't do it, you might regret it!
%
```

Devil answer

Angel answer

Other places you'll find the Observer Pattern in the JDK

```
public class SwingObserverExample {
    JFrame frame;
    public static void main(String[] args) {
        SwingObserverExample example = new SwingObserverExample();
        example.go();
    }
    public void go() {
        frame = new JFrame();

        JButton button = new JButton("Should I do it?");
        button.addActionListener(new AngelListener());
        button.addActionListener(new DevilListener());

        // Set frame properties here
    }

    class AngelListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            System.out.println("Don't do it, you might regret it!");
        }
    }

    class DevilListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            System.out.println("Come on, do it!");
        }
    }
}
```

Simple Swing application that just creates a frame and throws a button in it.

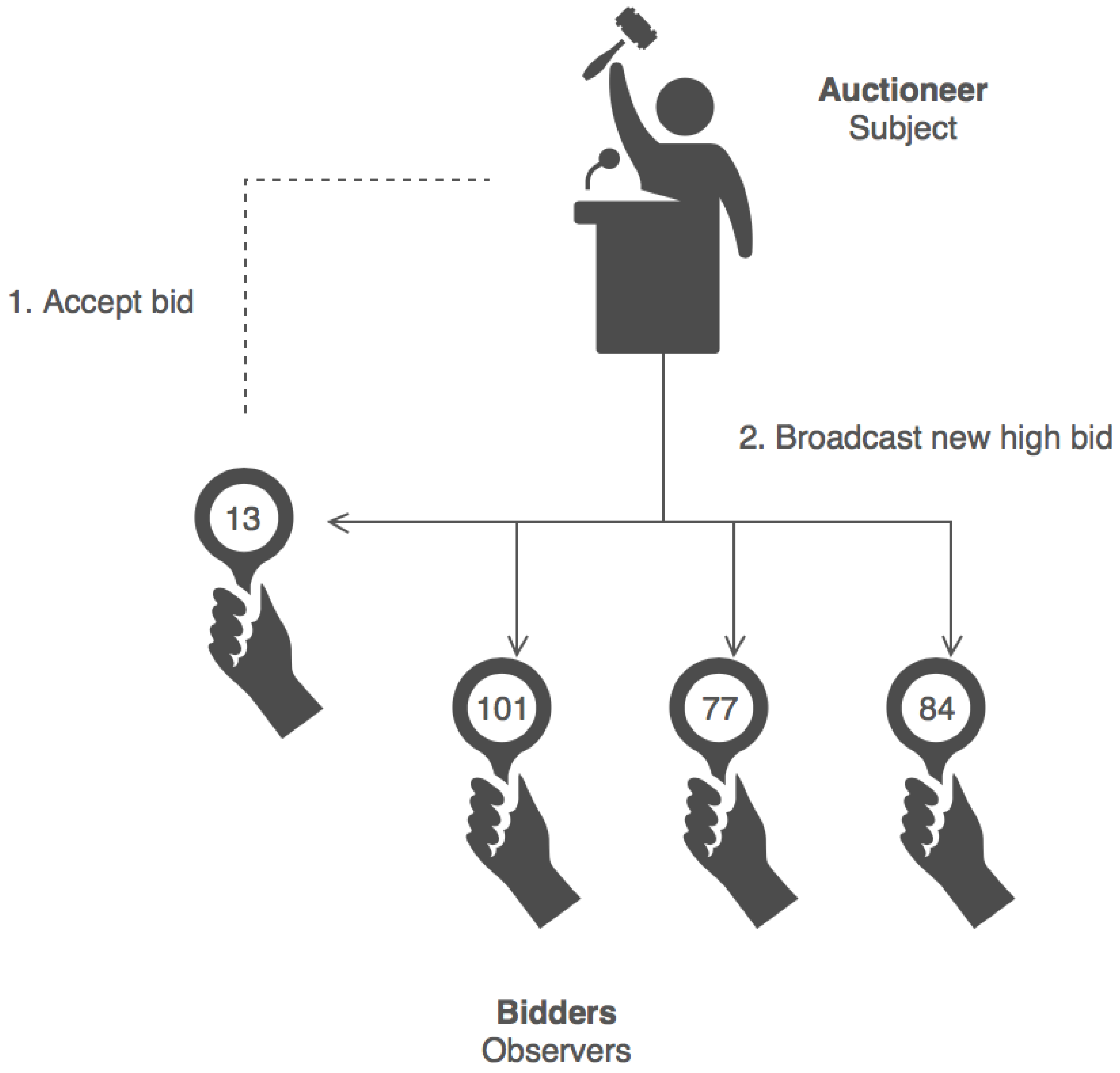
Makes the devil and angel objects listeners (observers) of the button.

Code to set up the frame goes here.

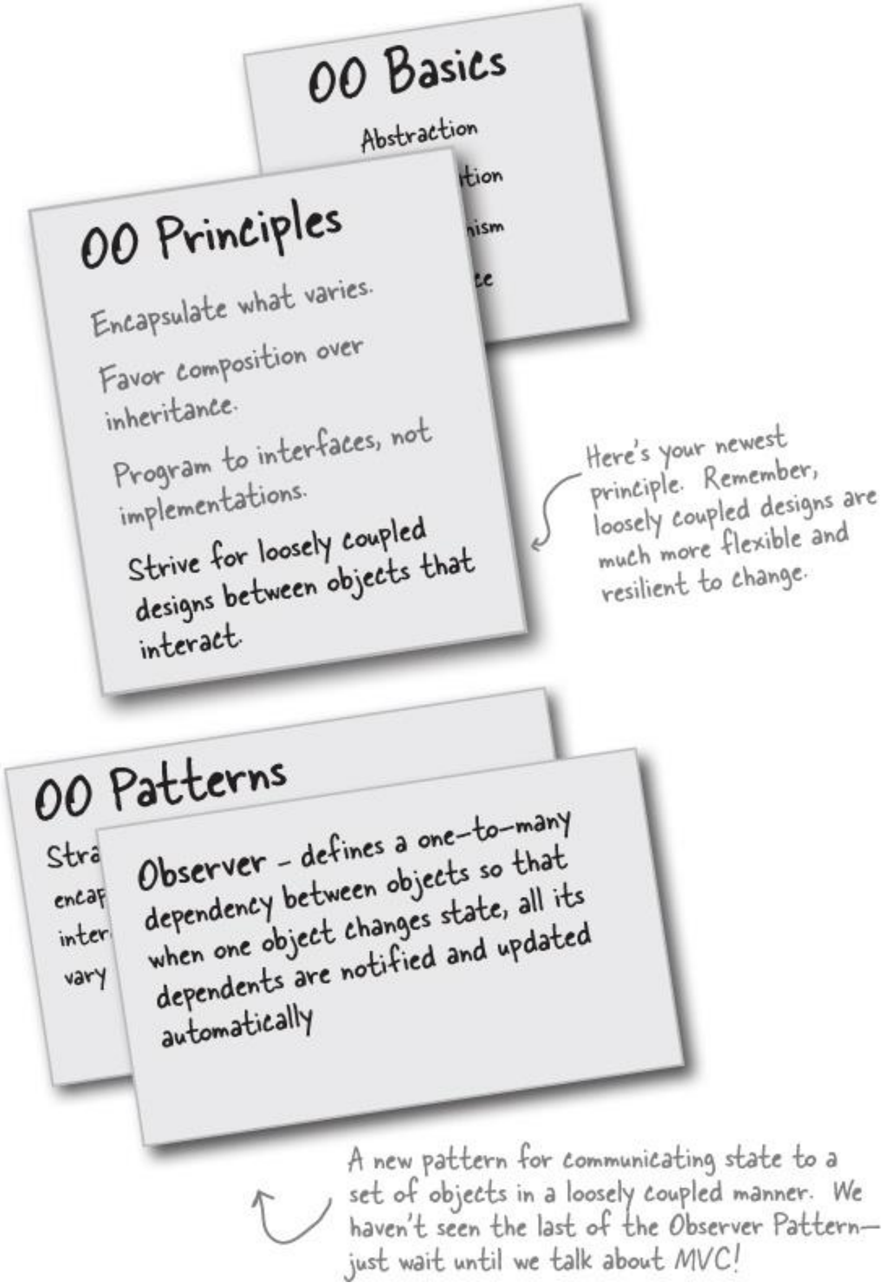
Here are the class definitions for the observers, defined as inner classes (but they don't have to be).

Rather than update(), the actionPerformed() method gets called when the state in the subject (in this case the button) changes.

Another example



Design Principles in the Observer Pattern



OO Basics

Abstraction

tion

nism

ce

OO Principles

Encapsulate what varies.

Favor composition over inheritance.

Program to interfaces, not implementations.

Strive for loosely coupled designs between objects that interact.

Here's your newest principle. Remember, loosely coupled designs are much more flexible and resilient to change.

OO Patterns

Stra

encap

inter

vary

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

A new pattern for communicating state to a set of objects in a loosely coupled manner. We haven't seen the last of the Observer Pattern—just wait until we talk about MVC!