Lecture 02: The Observer Pattern

SE313, Software Design and Architecture

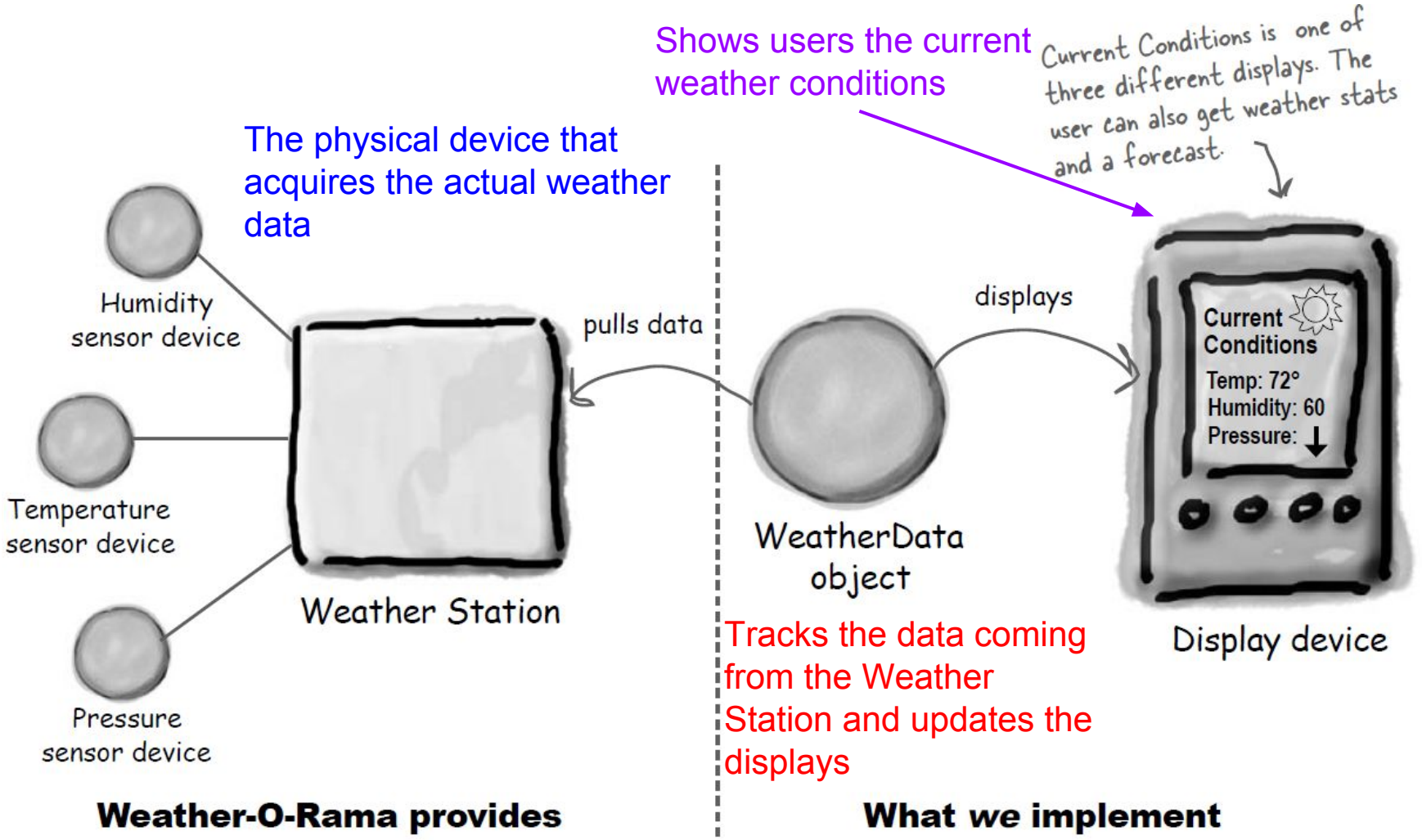
Damla Oguz

Chapter 2: The Observer Pattern

- **Don’t miss out when something interesting happens!**

Statement of Work

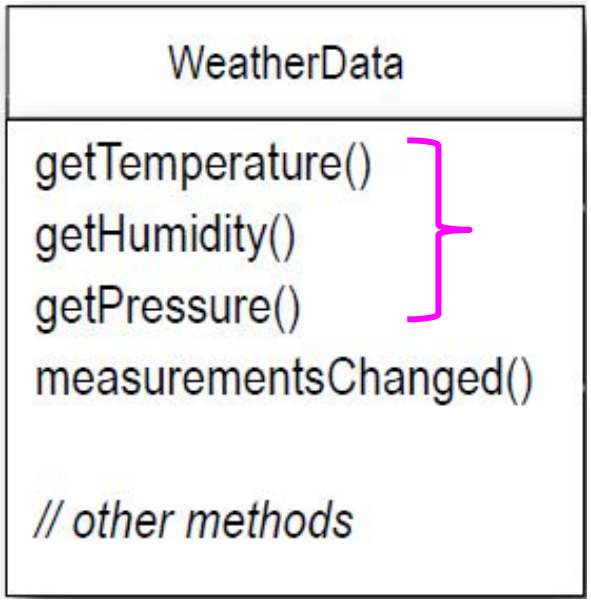
- We would like to build an Internet-based Weather Monitoring Station, Weather-O-Rama.
- We have WeatherData object which tracks current weather conditions
  - temperature
  - humidity
  - barometric pressure
- The application to be created must provide three display elements:
  - current conditions
  - weather statistics
  - a simple forecast
- Further, this is an expandable weather station.


Shows users the current weather conditions
Current Conditions is one of three different displays. The user can also get weather stats and a forecast.
The physical device that acquires the actual weather data
Humidity sensor device
Temperature sensor device
Pressure sensor device
Weather Station
pulls data
WeatherData object
displays
Current Conditions
Temp: 72°
Humidity: 60
Pressure:
Display device
Tracks the data coming from the Weather Station and updates the displays
Weather-O-Rama provides
What we implement

The Weather Monitoring application overview

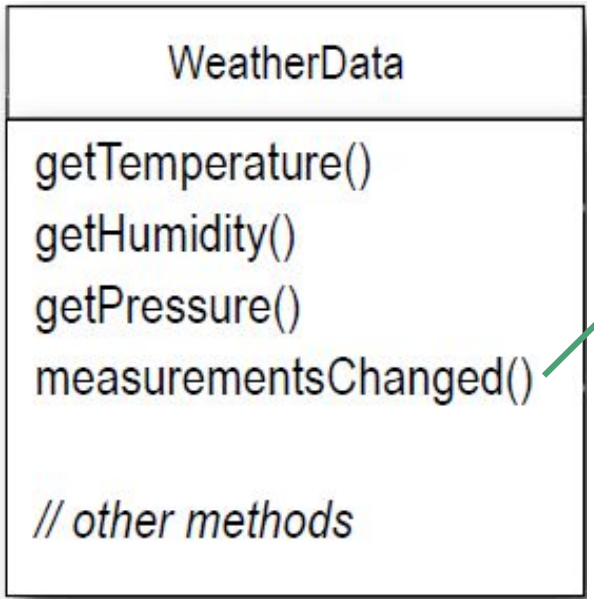
- Our job is to create an application that uses the WeatherData object to update three displays for
  - current conditions,
  - weather statistics, and
  - a forecast.

Unpacking the WeatherData class



- 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.

Unpacking the WeatherData class (cont.)



WeatherData.java

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

- The developers of WeatherData object left us a clue about what we need to add...

What needs to be done?

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

Display device

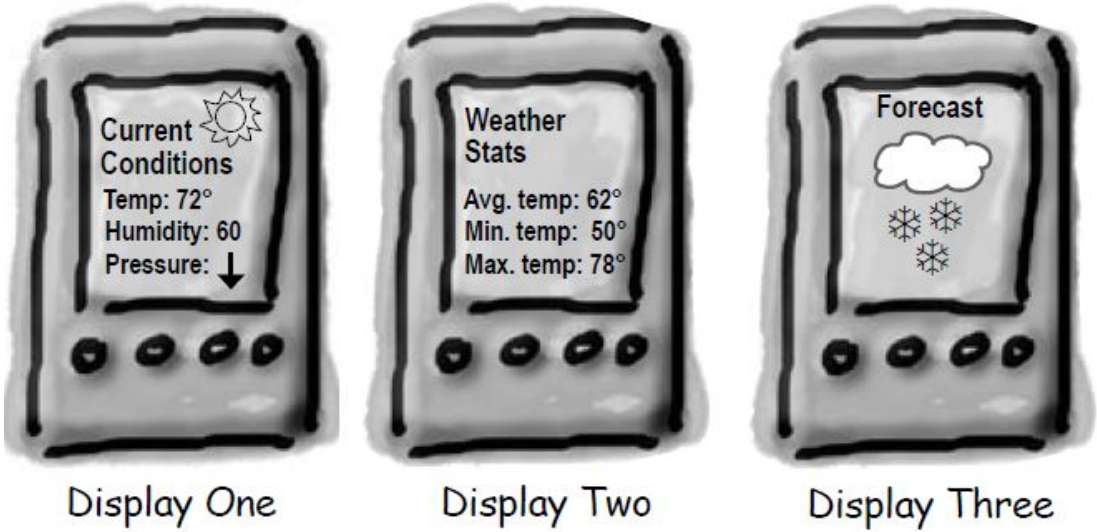
Remember, this Current Conditions is just ONE of three different display screens.

What do we know so far?

- The WeatherData class has getter methods for three measurement values: temperature, humidity and barometric pressure.
  - **getTemperature()**
  - **getHumidity()**
  - **getPressure()**
- The **measurementsChanged()** method is called any time new weather measurement data is available.

What do we know so far? (cont.)

- We need to implement three display elements that use the weather data:
  - a current conditions display,
  - a statistics display
  - a forecast display
- These displays must be updated each time WeatherData has new measurements.



What do we know so far? (cont.)

- The system must be expandable.
  - Other developers can create new custom display elements.
  - Users can add or remove as many display elements as they want to the application.

Future displays

First implementation possibility (misguided)

```
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
}
```

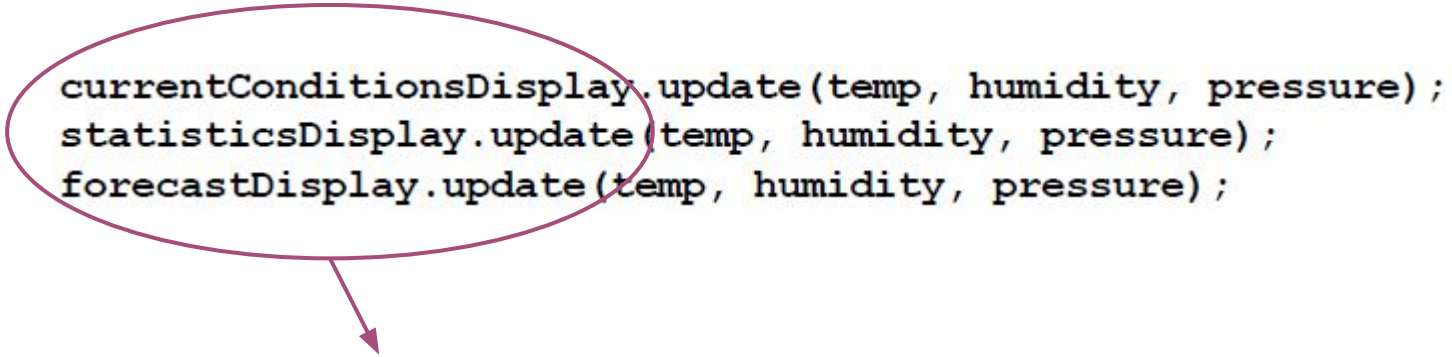
Grab the most recent measuremets by calling the WeatherData's getter methods (already implemented).

Now update the displays...

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

What’s wrong with our implementation?

- We are coding to concrete implementations, not interfaces.



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

What’s wrong with our implementation?

- We are coding to concrete implementations, not interfaces.

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

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

What’s wrong with our implementation?

- We haven’t encapsulated the part that changes.

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

Area of change, we need to encapsulate this.

Based on our first implementation, the followings apply

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

We’ll take a look at Observer, then come back and figure out how to apply it to the weather monitoring application.

Meet the Observer Pattern

Newspaper or magazine subscription works:

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

Publishers + Subscribers = Observer Pattern

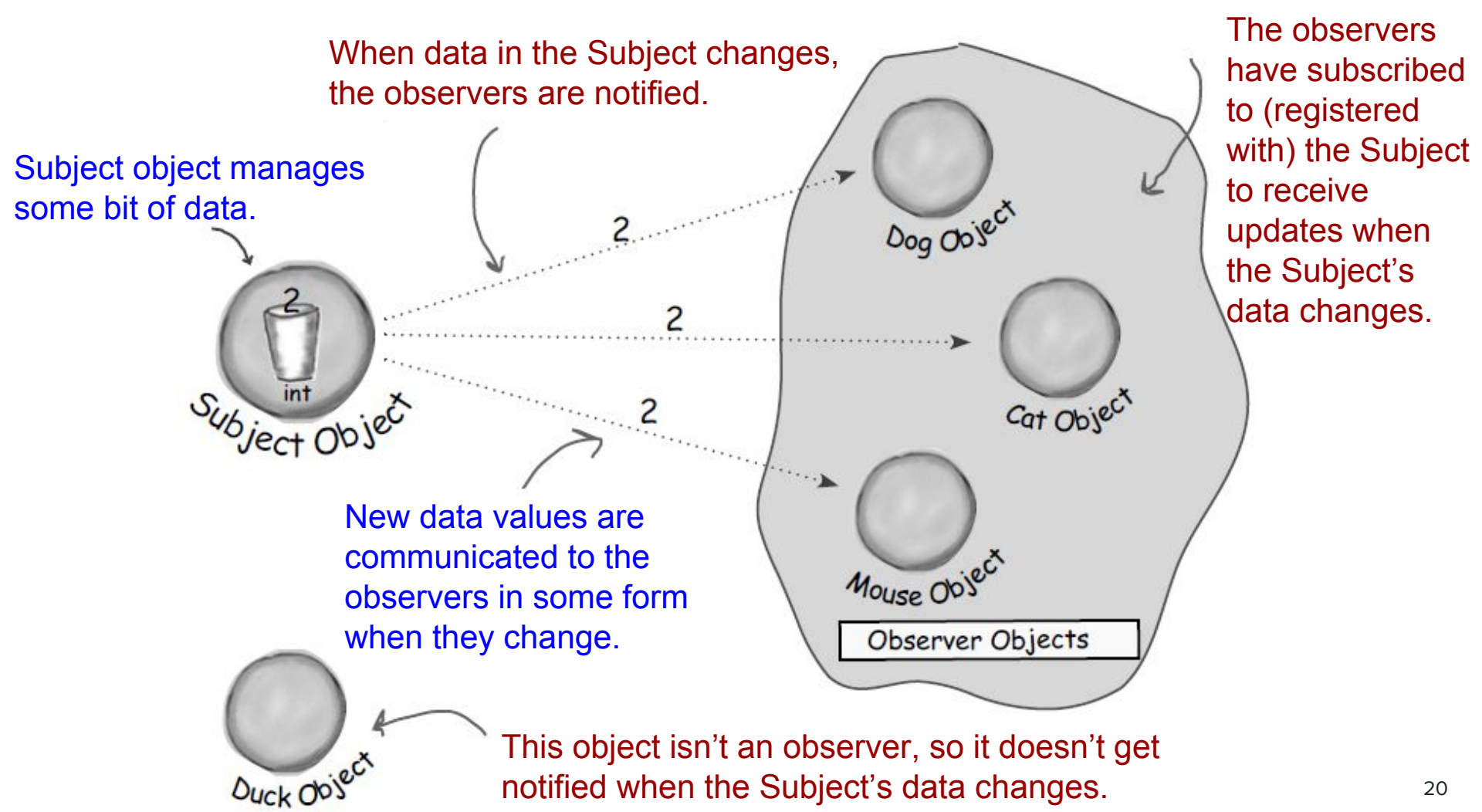
The publisher  the **SUBJECT**

The subscribers  the **OBSERVERS**

Publishers + Subscribers = Observer Pattern

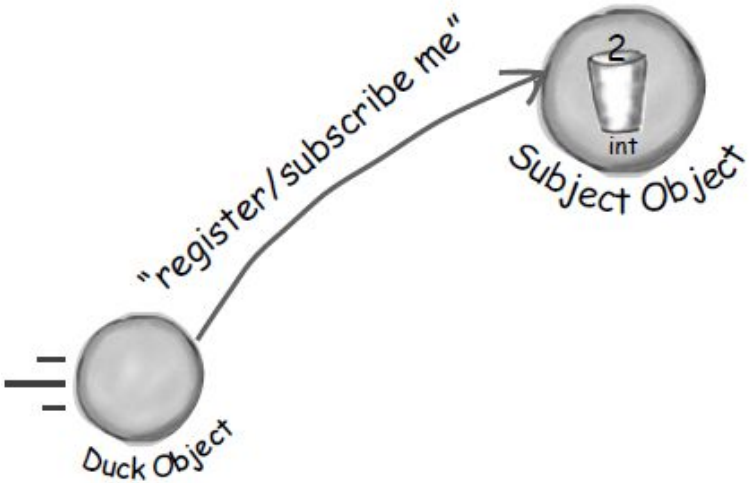
The **SUBJECT** is the object that contains the state and controls it.

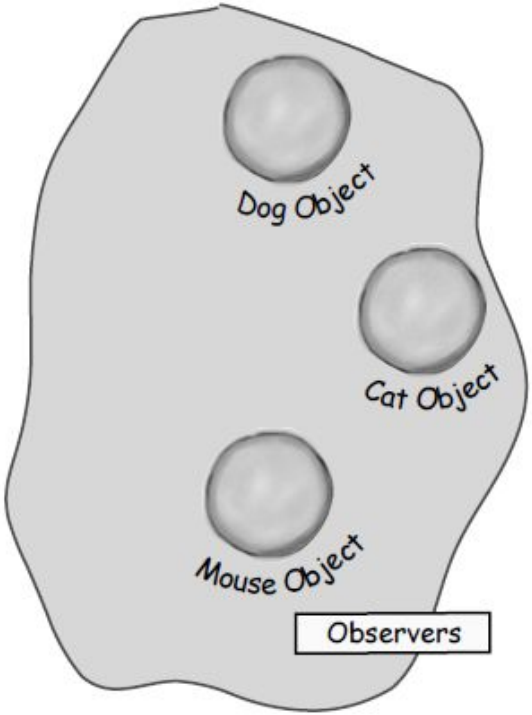
The **OBSERVERS** use the state, even if they do not own it.


When data in the Subject changes, the observers are notified.
The observers have subscribed to (registered with) the Subject to receive updates when the Subject's data changes.
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.

A day in the life of the Observer Pattern

A Duck object tells the Subject that it wants to become an observer.

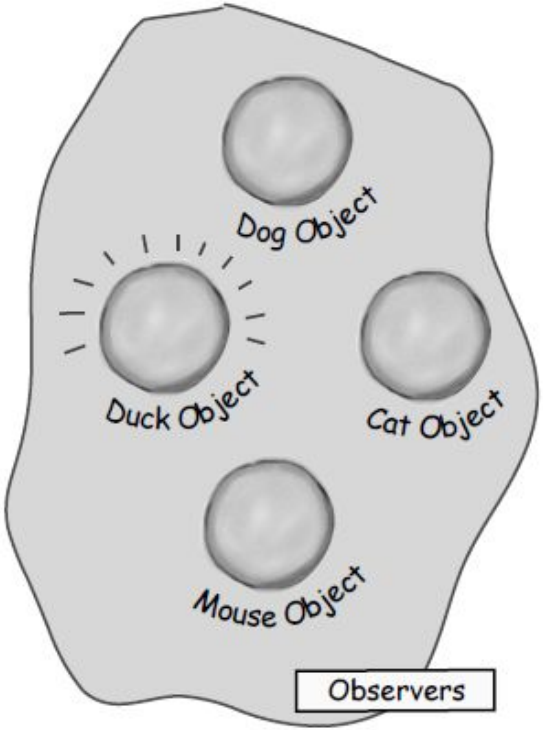




A day in the life of the Observer Pattern (cont.)

The Duck object is now an official observer.

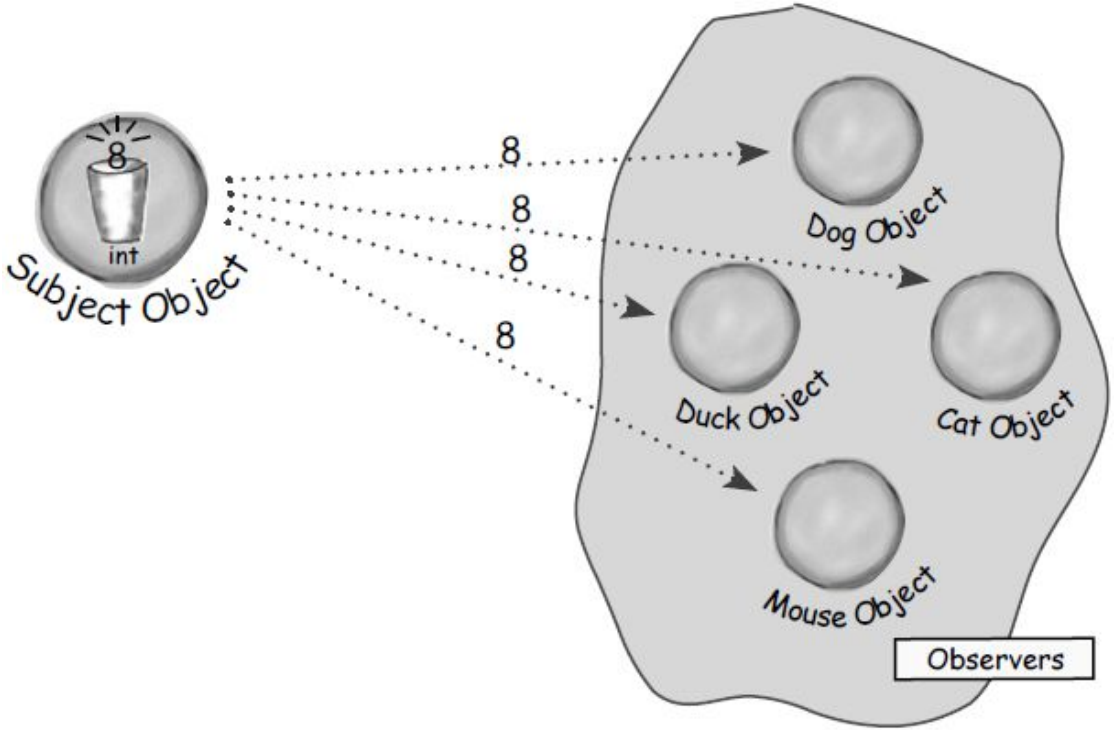
Duck is waiting for the next notification so he can get an int.



A day in the life of the Observer Pattern (cont.)

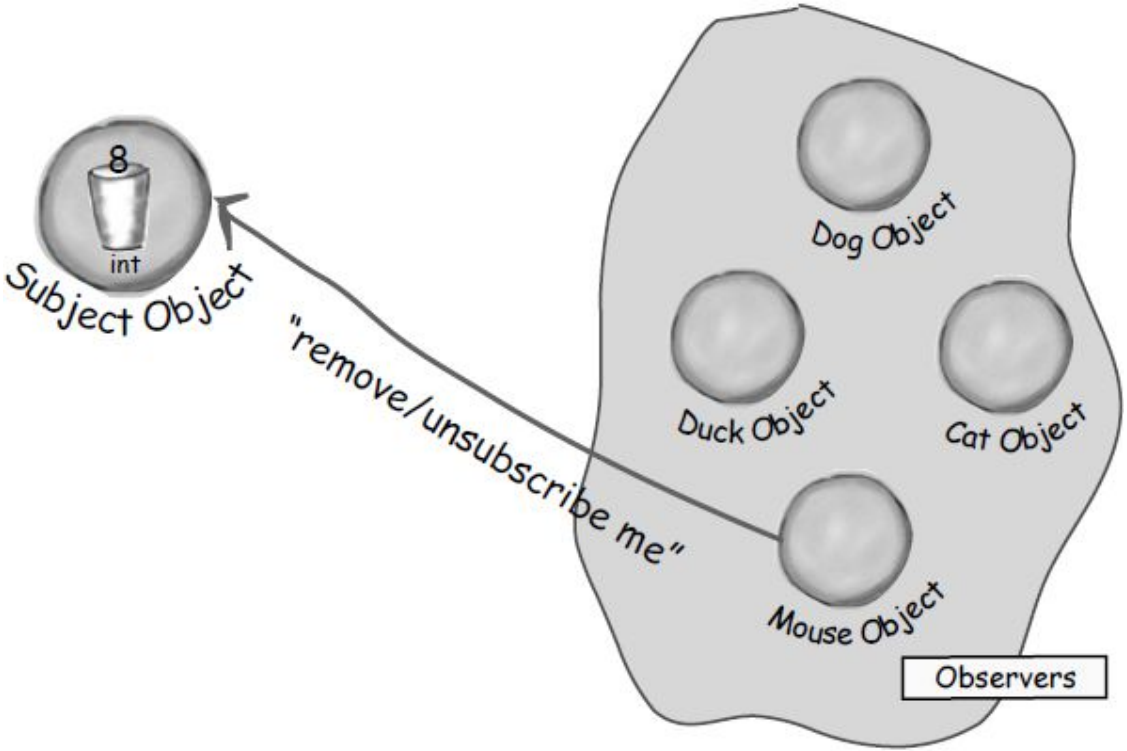
The Subject gets a new data value!

Now Duck and all the rest of the observers get a notification that the Subject has changed.



A day in the life of the Observer Pattern (cont.)

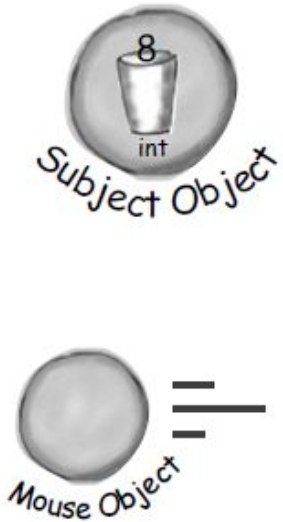
The Mouse object asks to be removed as an observer.

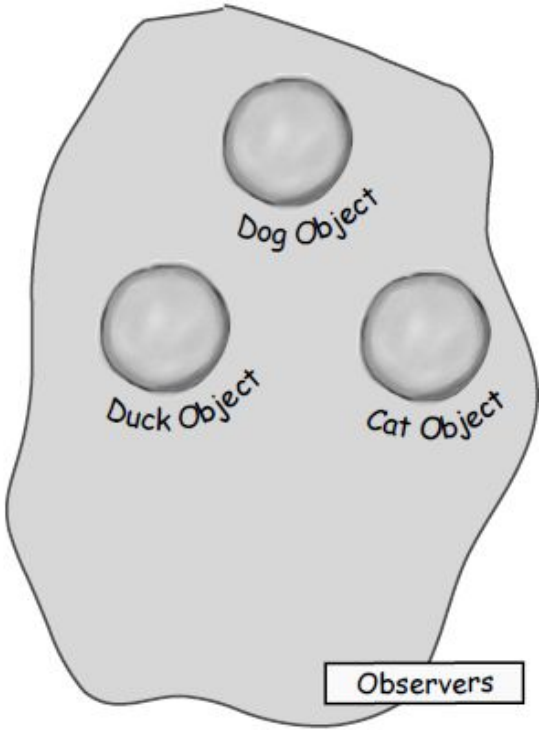


A day in the life of the Observer Pattern (cont.)

Mouse is out of here!

The Subject acknowledges the Mouse's request and removes it from the set of observers.

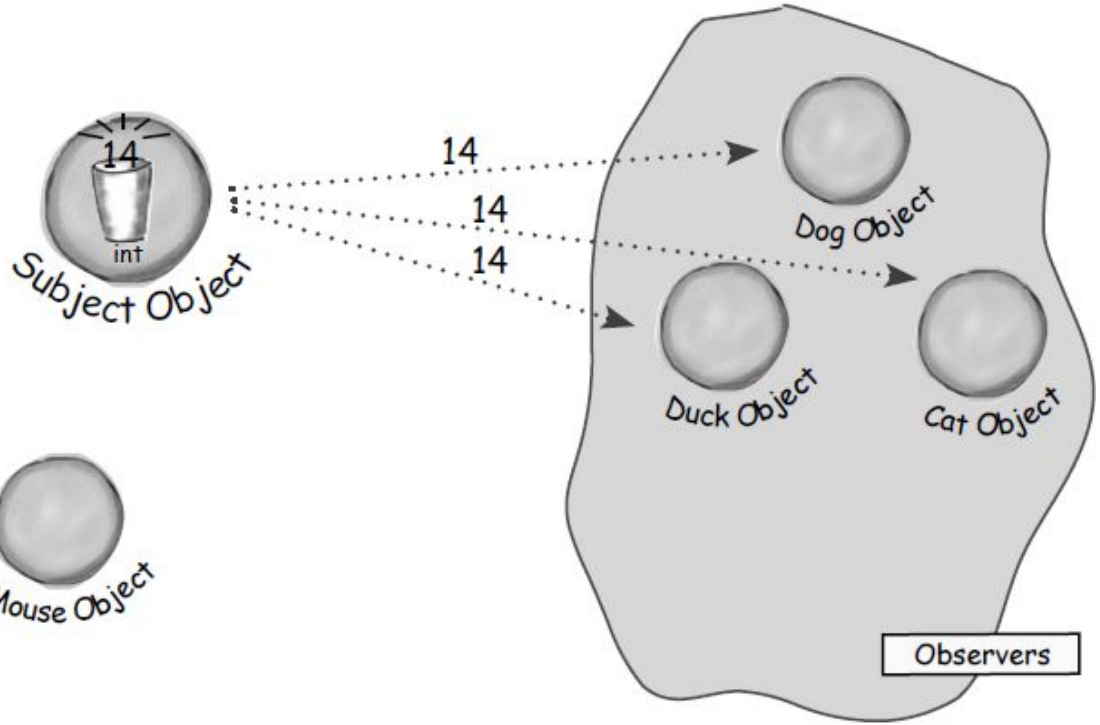




A day in the life of the Observer Pattern (cont.)

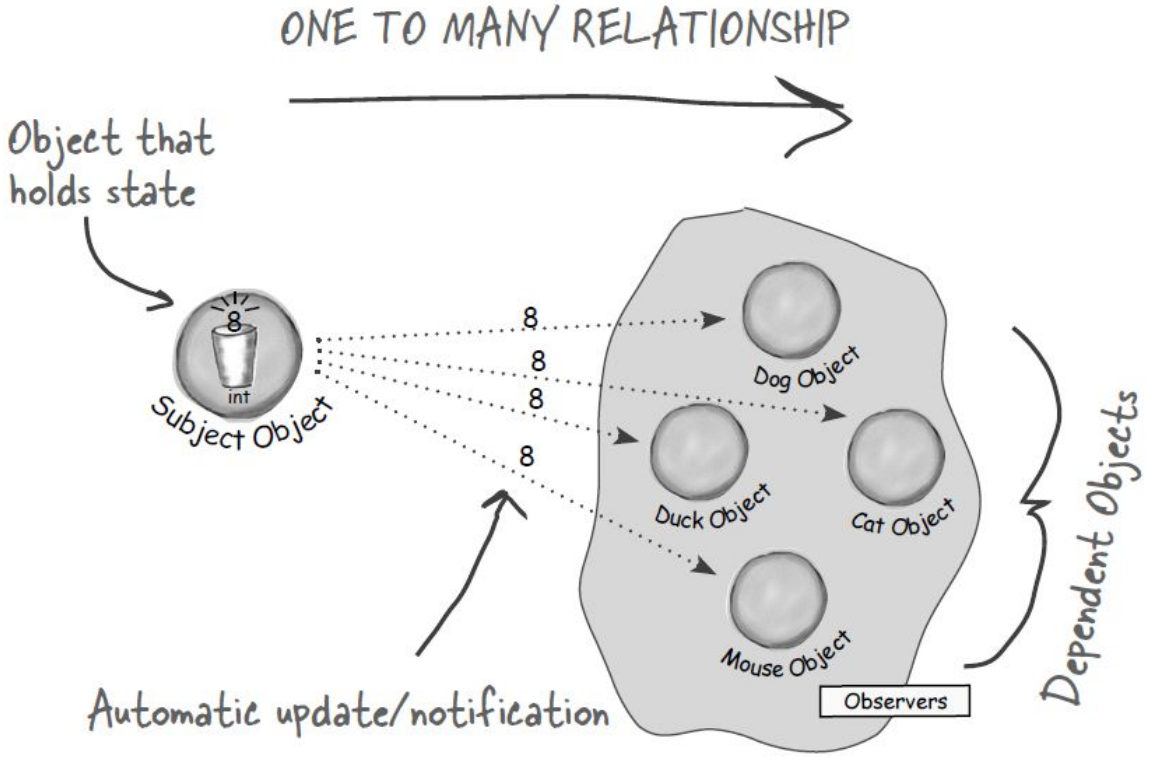
The Subject has another new int.

All the observers get another notification, except for the Mouse who is no longer included.

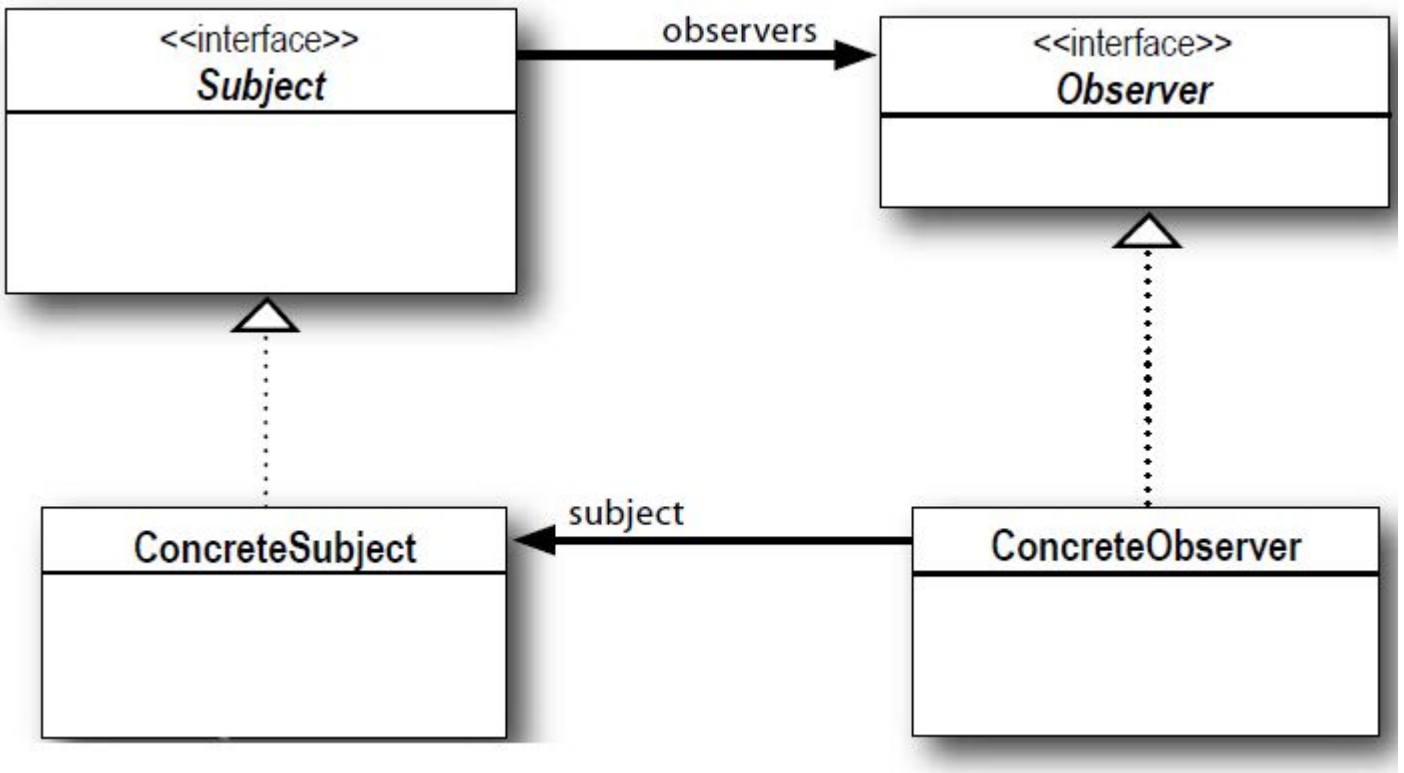


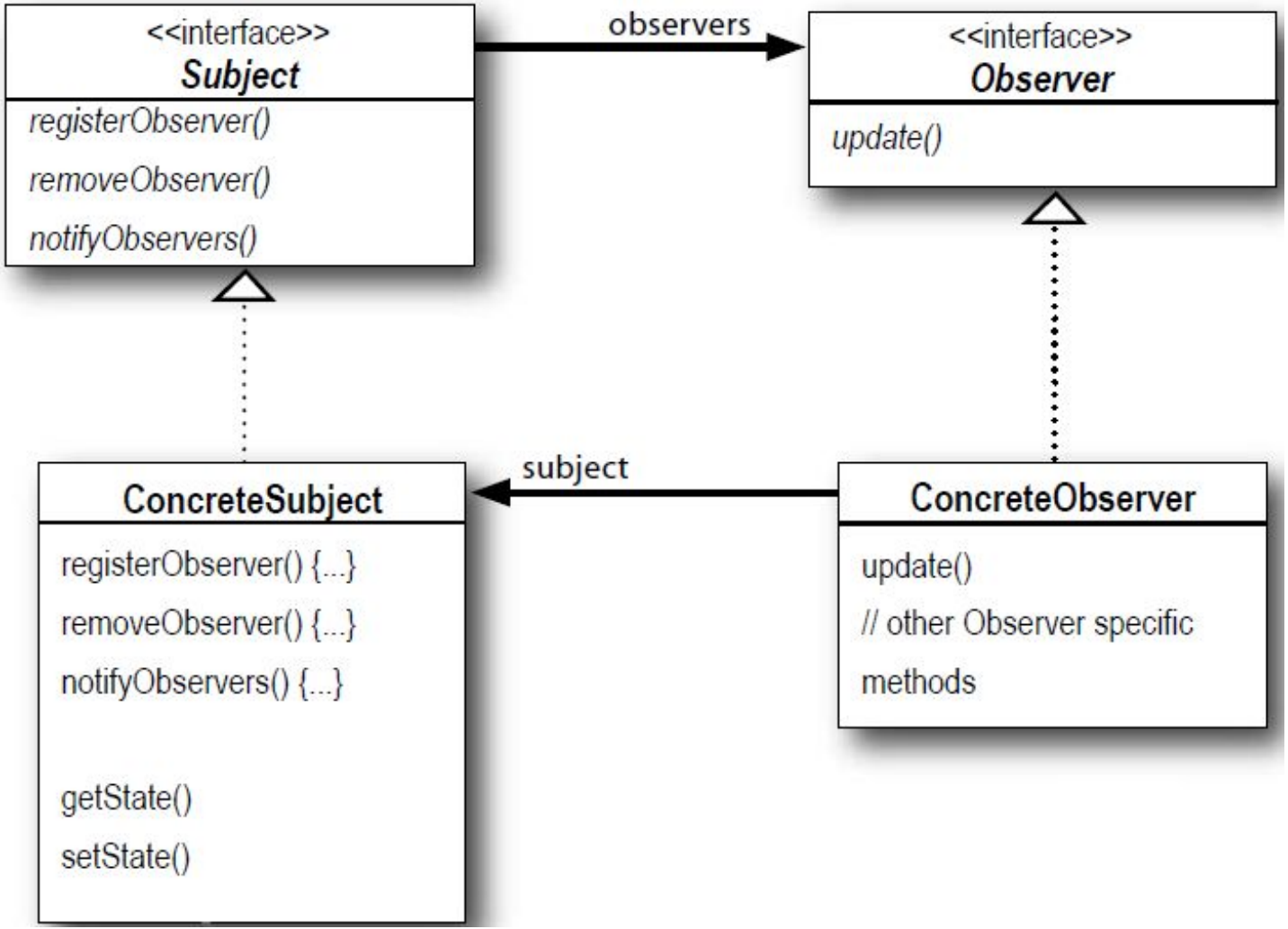
The Observer Pattern defined

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



The Observer Pattern defined: the class diagram

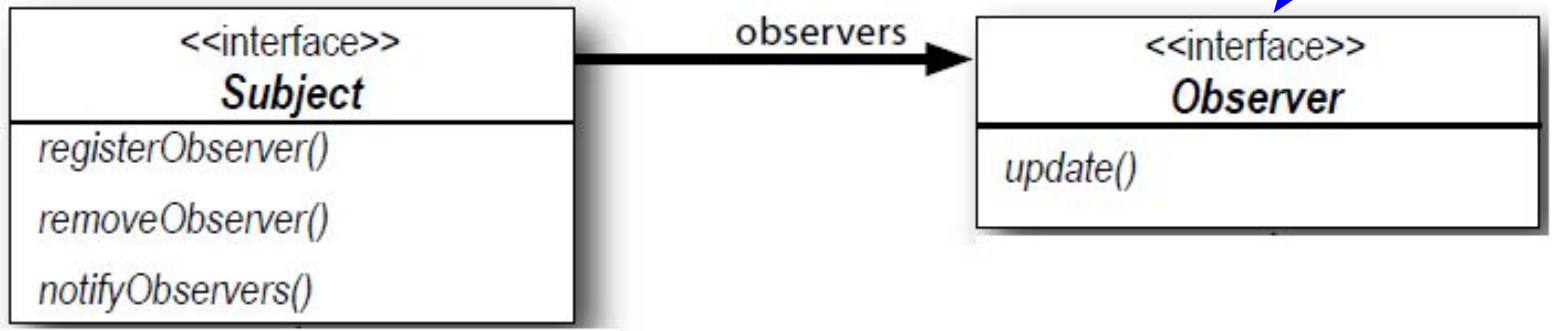



<<interface>>
Subject
registerObserver()
removeObserver()
notifyObservers()
observers
<<interface>>
Observer
update()
ConcreteSubject
registerObserver() {...}
removeObserver() {...}
notifyObservers() {...}
getState()
setState()
subject
ConcreteObserver
update()
// other Observer specific
methods

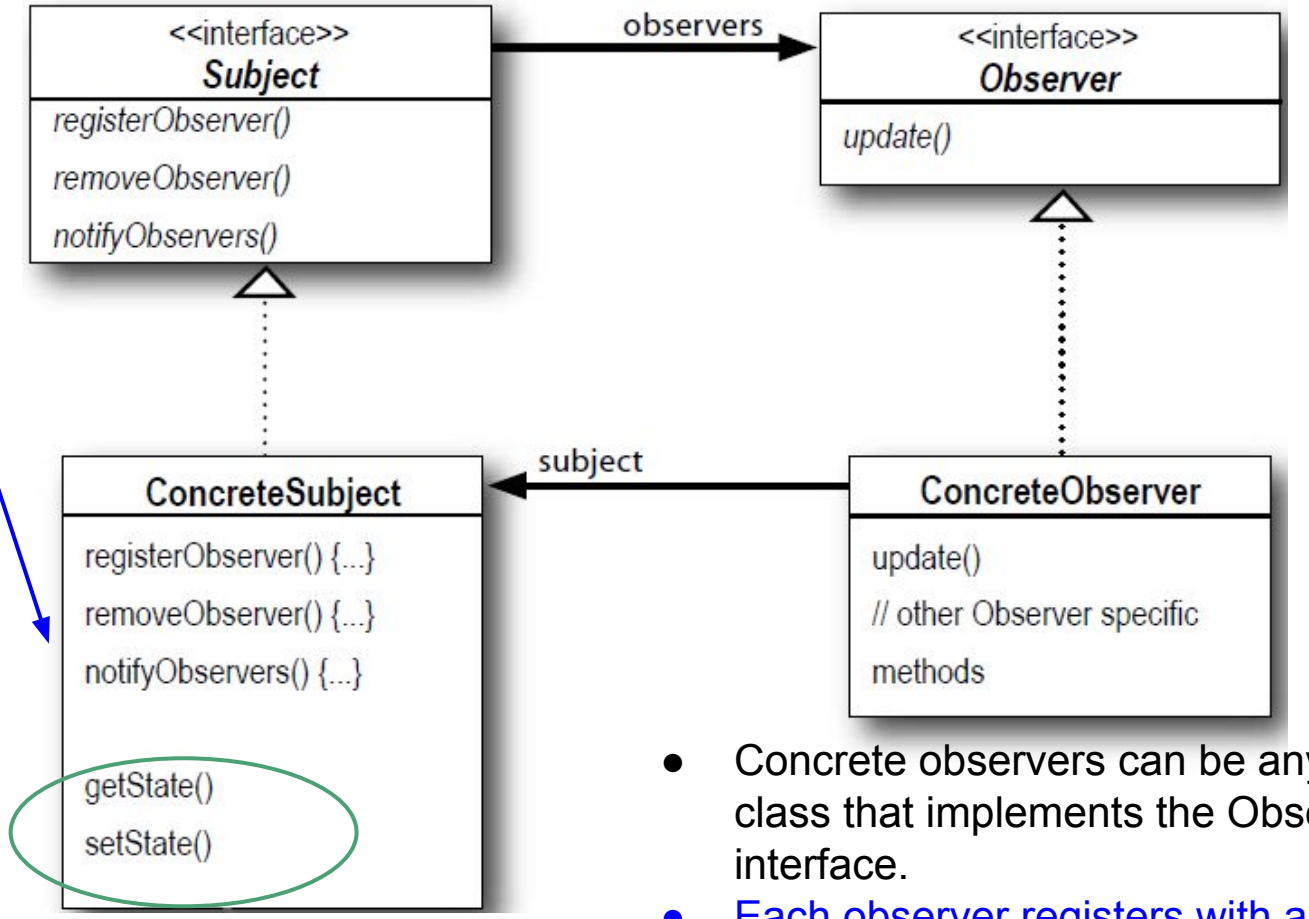
- Here is the Subject interface.
- Objects use this interface to register as observers and also to remove themselves being observers.

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.



- A concrete subject always implements the Subject interface.
- notifyObservers() method is used to update all the current observers whenever state changes.



The concrete subject may also these methods.

- 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, they have very **little knowledge** of each other
- Loosely coupled designs allow us to build flexible OO systems that can handle change
  - because they minimize the interdependency between objects

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

The power of Loose Coupling (cont.)

- The Observer Pattern provides an object design where subjects and observers are loosely coupled.
  - **The only thing the subject knows about an observer is that it implements a certain interface** (the Observer interface).
    - It does not 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 observers that implement Observer interface.

The power of Loose Coupling (cont.)

- **We never need to modify the subject to add new types of observers.**
  - All we have to do is implement the Observer interface in the new class and register as an observer.
- **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 each other.**
  - We are free to make changes to either, as long as the objects still meet their obligations to implement the subject or observer interfaces.

Back to the Weather Station project

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

- One: The WeatherData class
- Many: The various display elements that use the weather measurements

Back to the Weather Station project (cont.)

*The **SUBJECT** is the object that contains the state and controls it.*

- **WeatherData** class has state...
  - temperature
  - humidity
  - barometric pressure

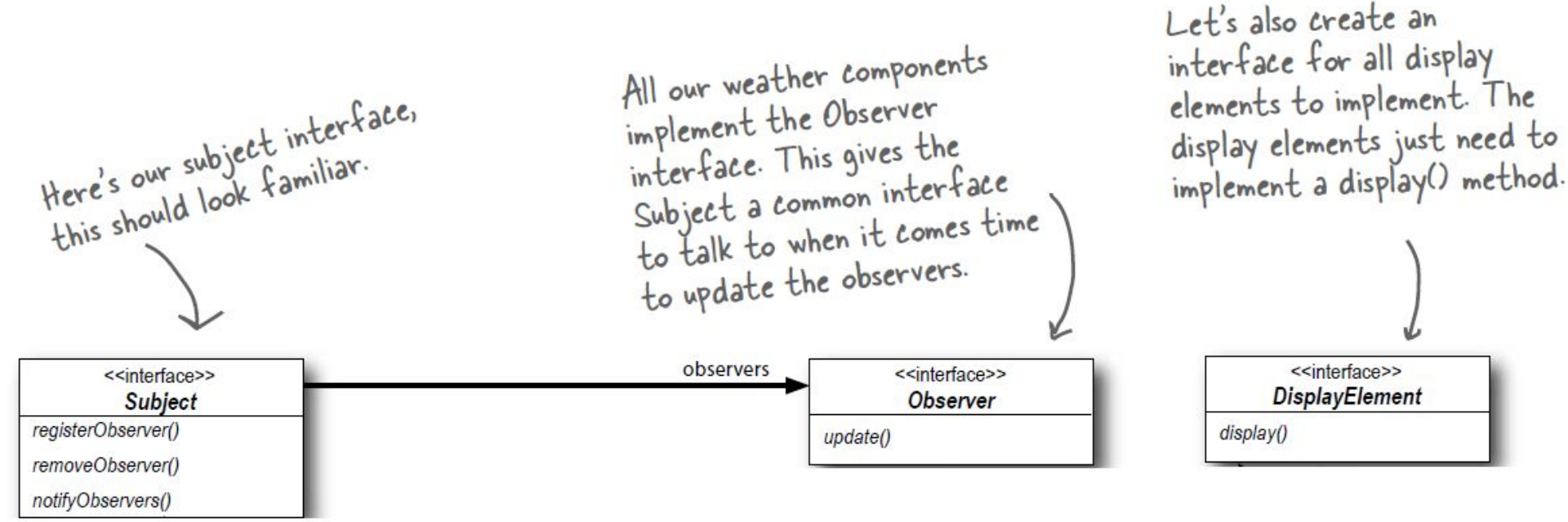
and those definitely change

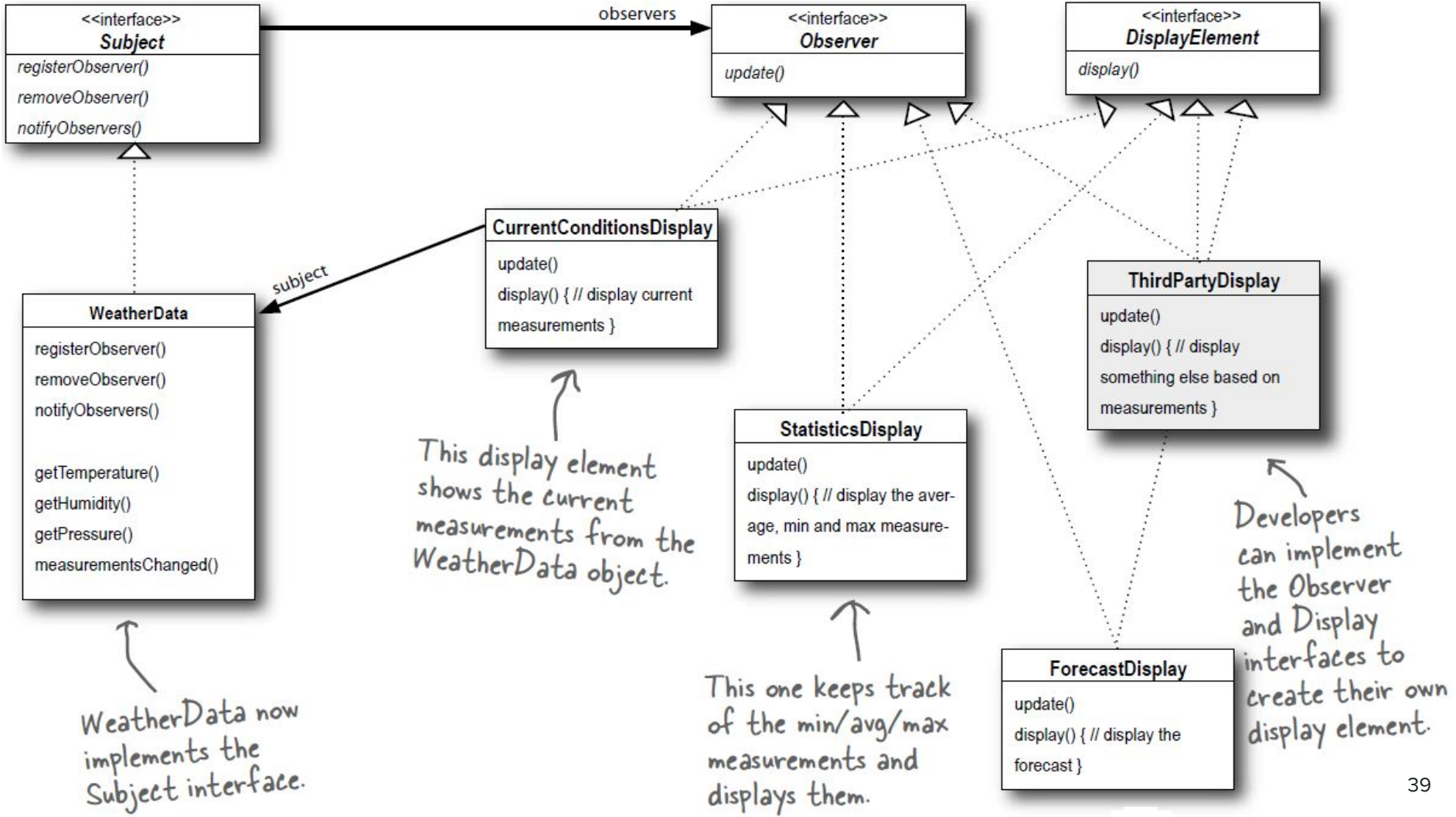
Back to the Weather Station project (cont.)

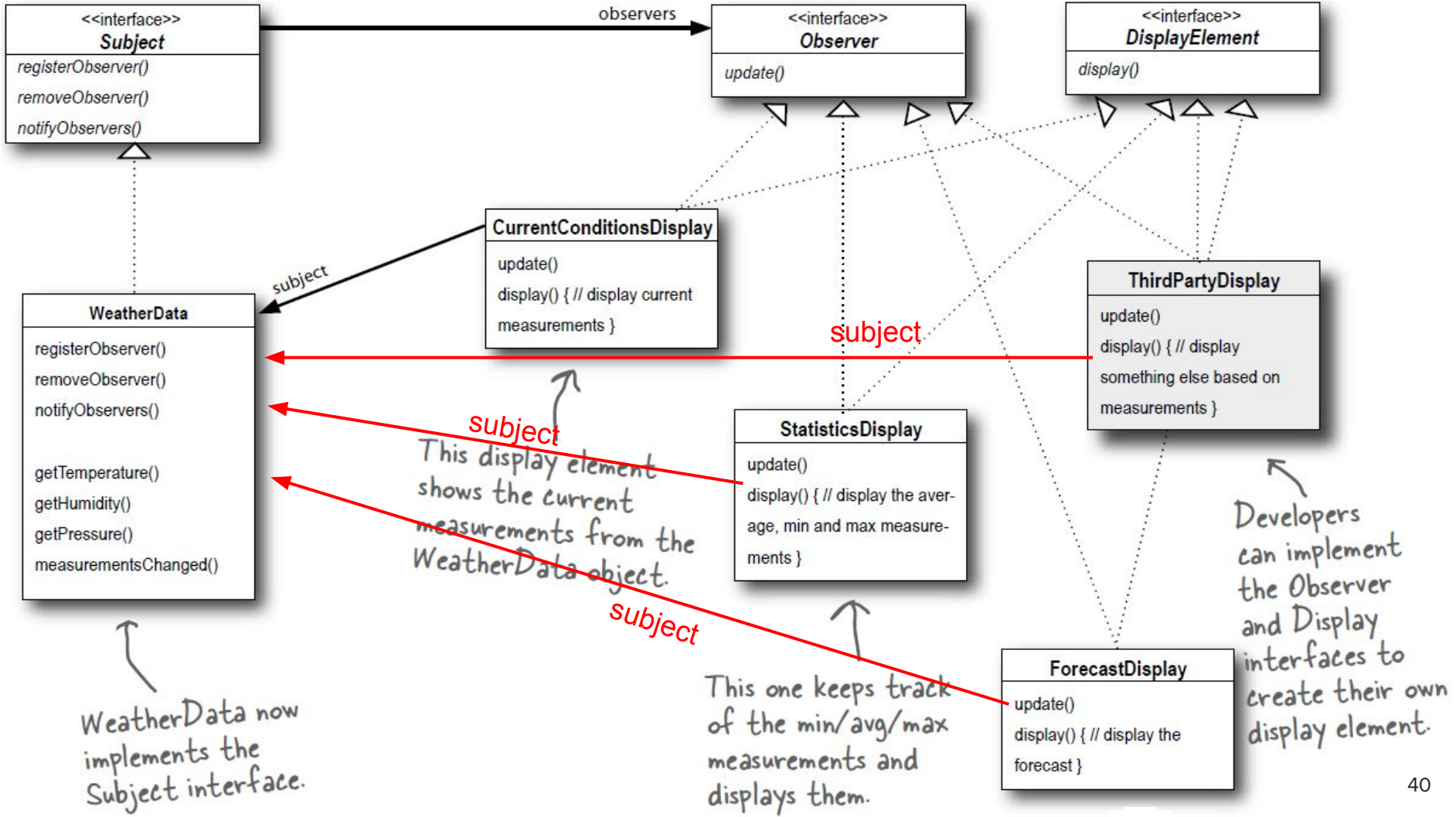
*The **OBSERVERS** use the state, even if they do not own it.*

- **Display elements** must be notified when the measurements change so that they can use them.

Designing the Weather Station




<<interface>>
Subject
registerObserver()
removeObserver()
notifyObservers()
observers
<<interface>>
Observer
update()
<<interface>>
DisplayElement
display()
CurrentConditionsDisplay
update()
display() { // display current measurements }
subject
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 Display interfaces to create their own display element.


<<interface>>
Subject
registerObserver()
removeObserver()
notifyObservers()
observers
<<interface>>
Observer
update()
<<interface>>
DisplayElement
display()
CurrentConditionsDisplay
update()
display() { // display current measurements }
subject
WeatherData
registerObserver()
removeObserver()
notifyObservers()
getTemperature()
getHumidity()
getPressure()
measurementsChanged()
subject
ThirdPartyDisplay
update()
display() { // display something else based on measurements }
subject
StatisticsDisplay
update()
display() { // display the average, min and max measurements }
subject
This display element shows the current measurements from the WeatherData object.
ForecastDisplay
update()
display() { // display the forecast }
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 Display interfaces to create their own display element.

Implementing the Weather Station: Interfaces

```
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.

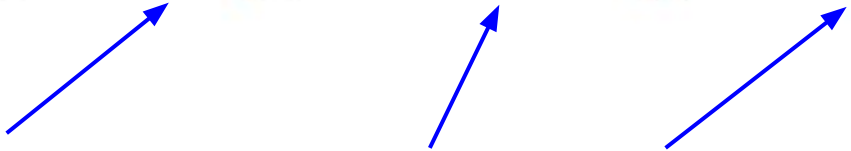
- The Observer to be registered or removed.

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

Implementing the Weather Station: Interfaces (cont.)

- The Observer interface is implemented by all observers, so they all have to implement the update() method.

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



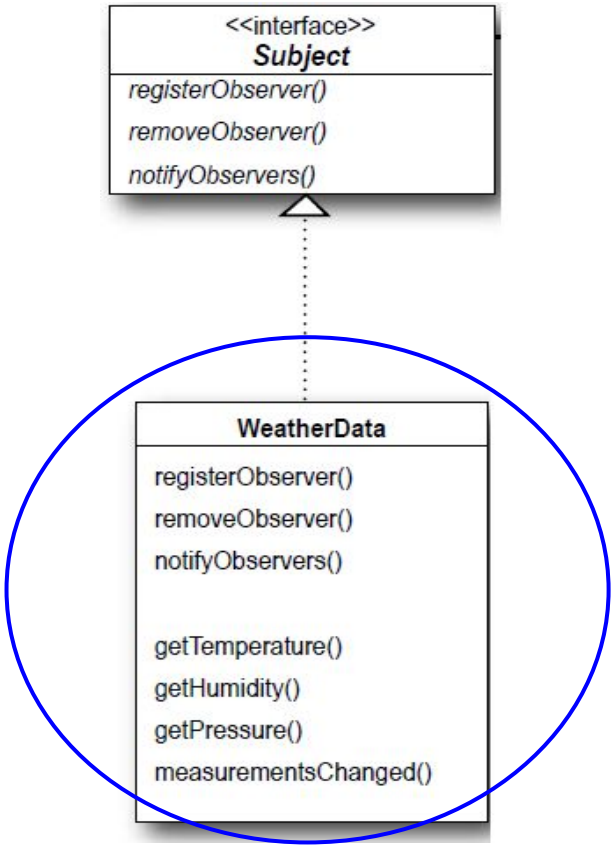
These are the state values the Observers get from the Subject when a weather measurement changes.

Implementing the Weather Station: Interfaces (cont.)

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

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

Implementing the Subject interface in WeatherData



Implementing the Subject interface in WeatherData

```
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);
        }
```

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

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

Implementing the Subject interface in WeatherData

```
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);
        }
}
```

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

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.

Implementing the Subject interface in WeatherData

```
public void measurementsChanged() {
        notifyObservers();
}
```

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

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

Rather than reading actual weather data off a device, we are going to use this method to test our display elements.

Implementing the Subject interface in WeatherData

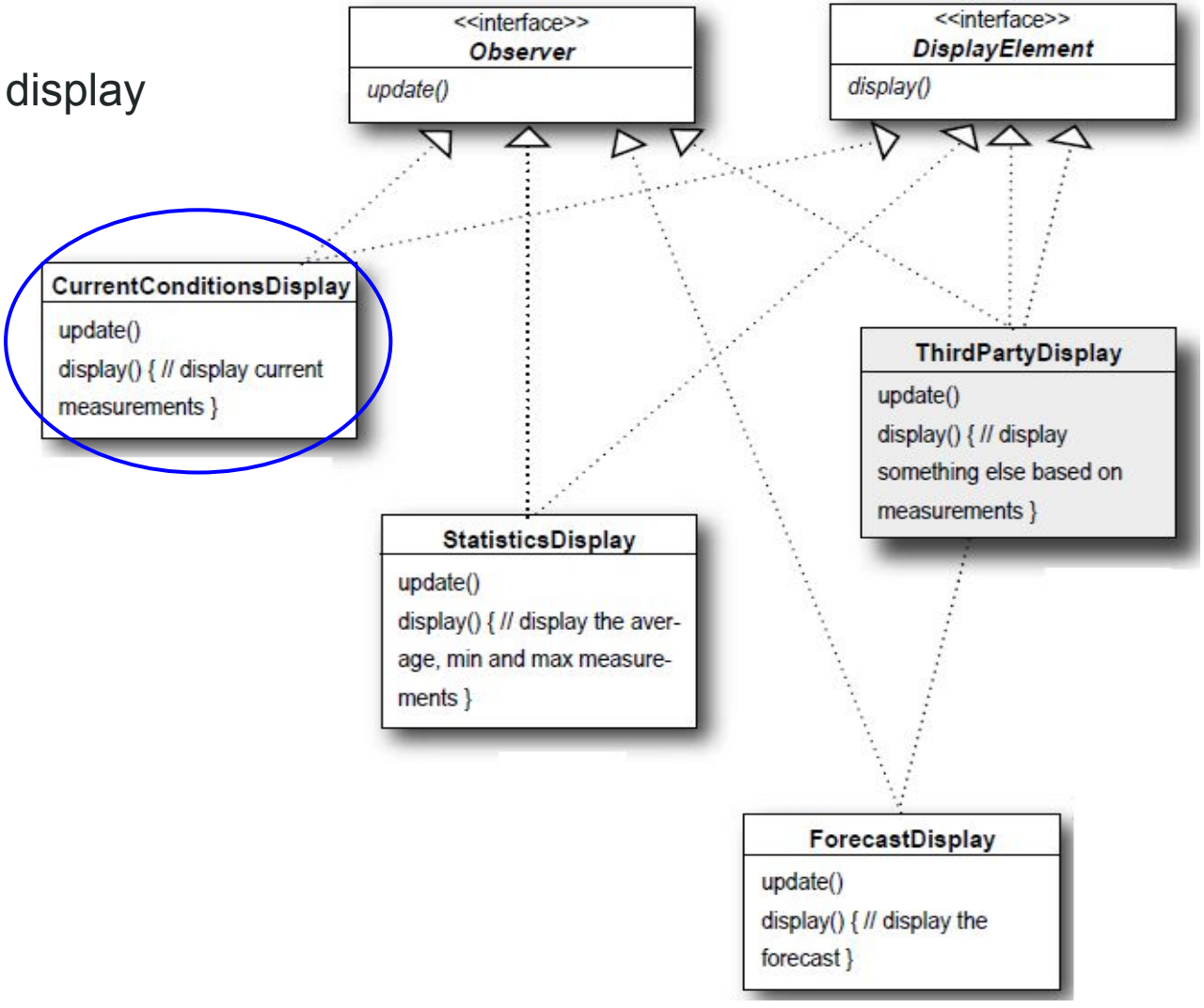
```
public float getTemperature() {
        return temperature;
}

public float getHumidity() {
        return humidity;
}

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

Other WeatherData methods...

Now, let's build those display elements



```
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");
        }
}
```

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.

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

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

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

Power up the Weather Station

The WeatherStation is ready to go, all we need is some code to glue everything together.

```
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);
        }
}
```

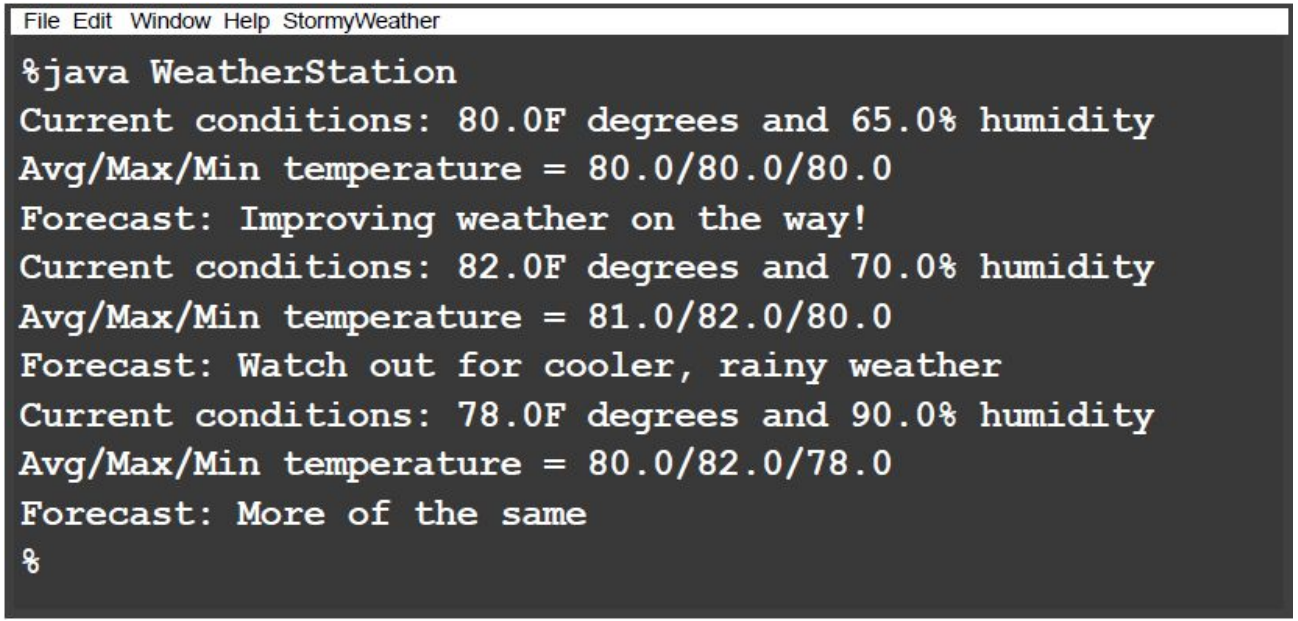
First, create the WeatherData object.

Create the three displays and pass them the WeatheraData object.

Simulate new weather measurements.

Power up the Weather Station

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



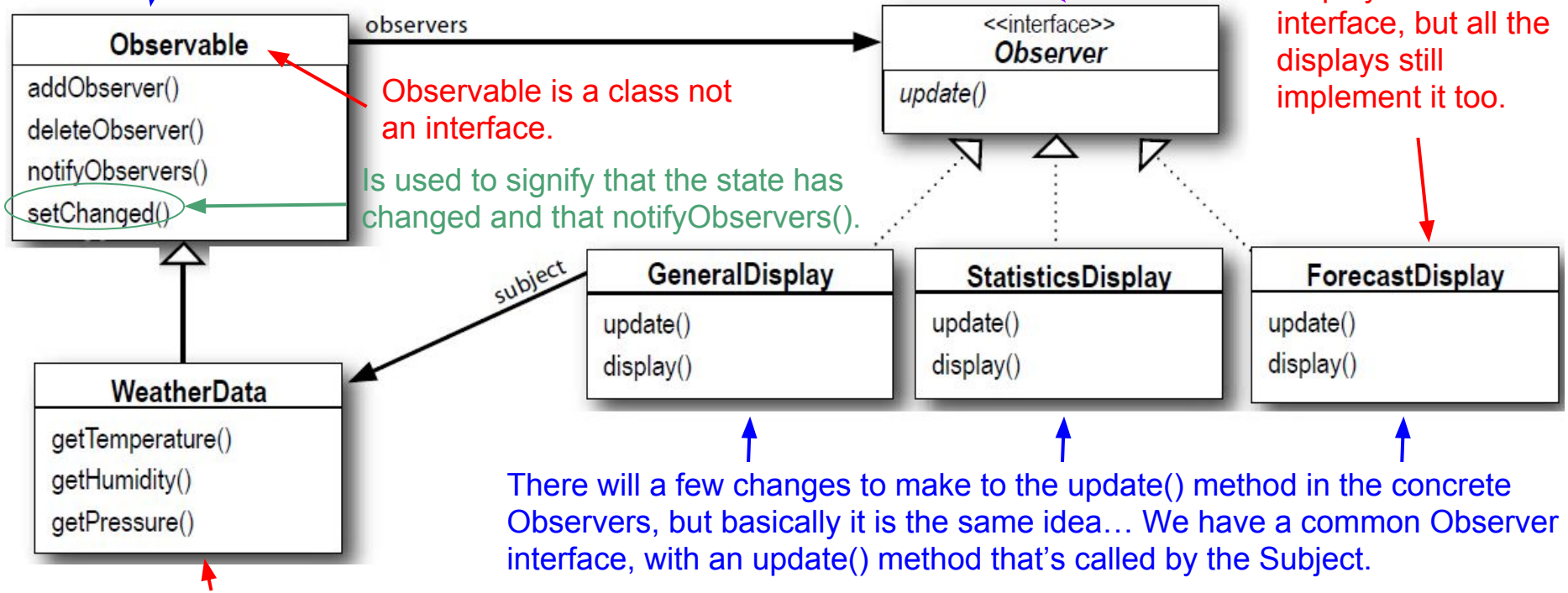
Using Java’s built-in Observer Pattern

- Java provides the **Observer interface** and the **Observable class** in the java.util package.
- With Java’s built-in support, all you have to do is **extend Observable** and **tell it when to notify the Observers**.

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.



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

Here is our Subject, which we can now also call the Observable. We do not need the register(), remove() and notifyObserver() methods anymore; we inherit that behavior from the superclass.

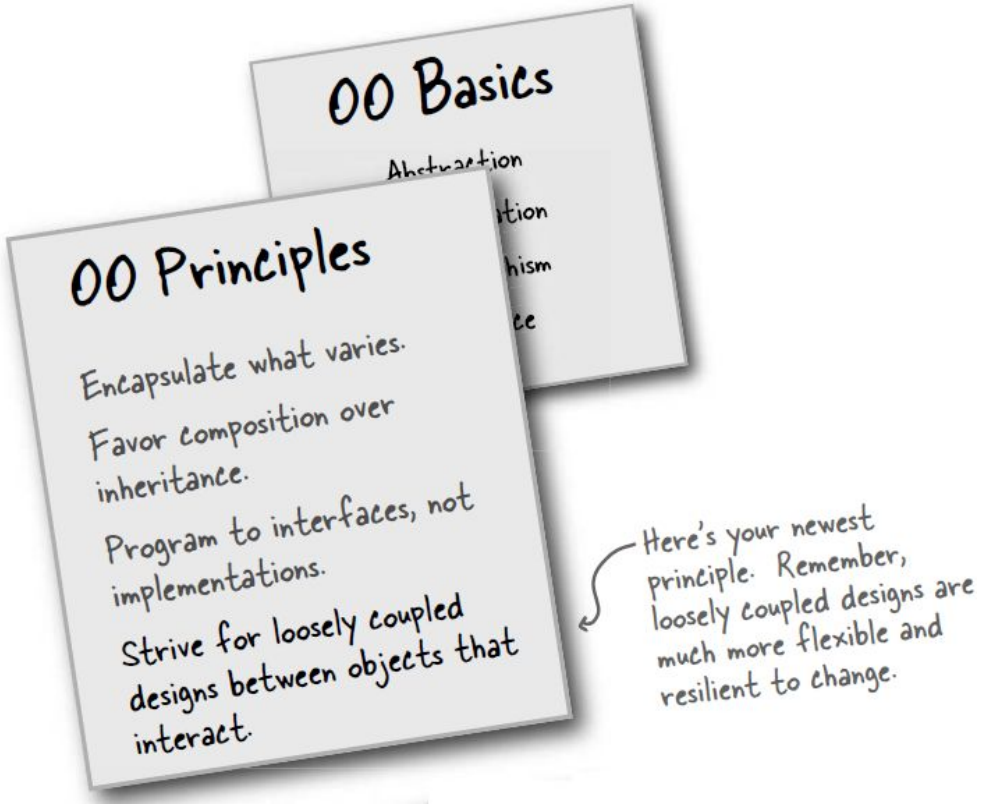
The dark side of java.util.Observable

- Observable is a ***class***, not an ***interface***, and worse, it doesn't even ***implement*** an interface.
  - You have to ***subclass*** it.
  - You cannot add on the Observable behavior to an existing class that already extends another superclass.
    - This limits its reuse potential.

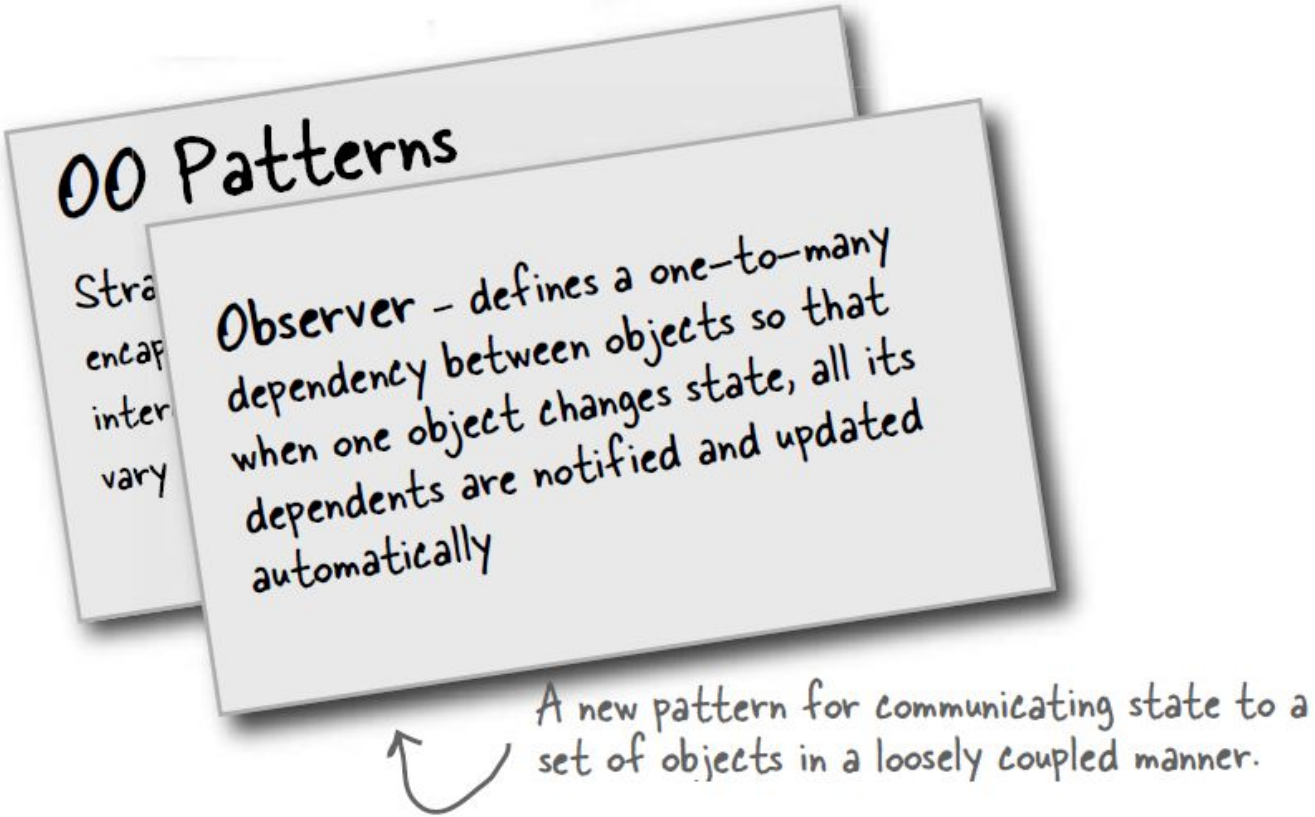
Recommendations

- Write the weather station application by using the Observer Pattern.
- Try to rework the application by using Java’s built-in Observer Pattern.
- Read Chapter 2 from the book.
- https://github.com/bethrobson/Head-First-Design-Patterns/tree/master/src/headfirst/designpatterns

Tools for your Design Toolbox



Tools for your Design Toolbox (cont.)



References

Material in this lecture is taken from Freeman, E., Robson, E., Bates, B., & Sierra, K., *Head First Design Patterns: A Brain-Friendly Guide*, O'Reilly Media, Inc., 2004.