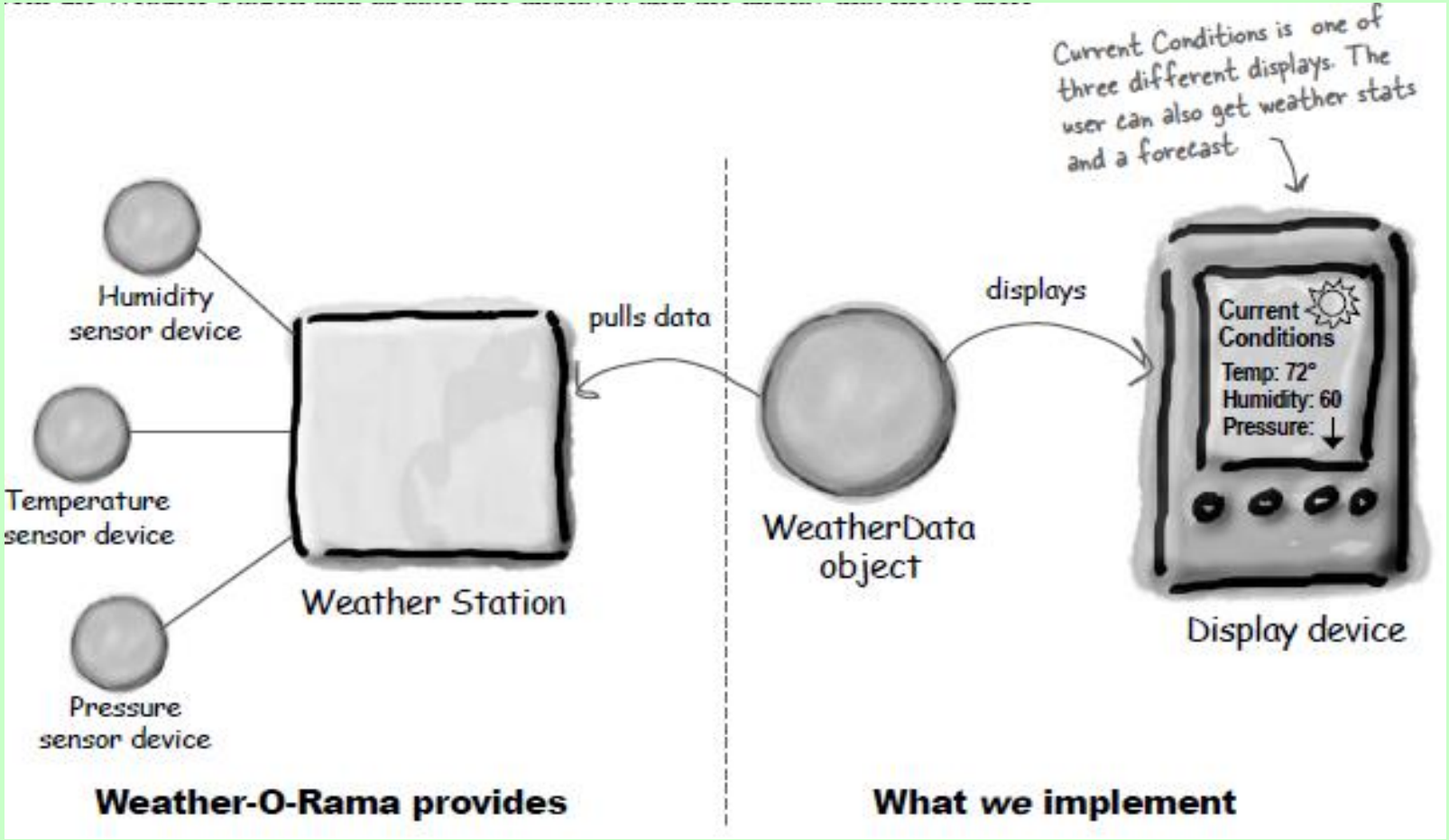
The Observer Pattern

1. Keeping the Objects in the know

Internet-based Weather Monitoring Station



Internet-based Weather Monitoring Station

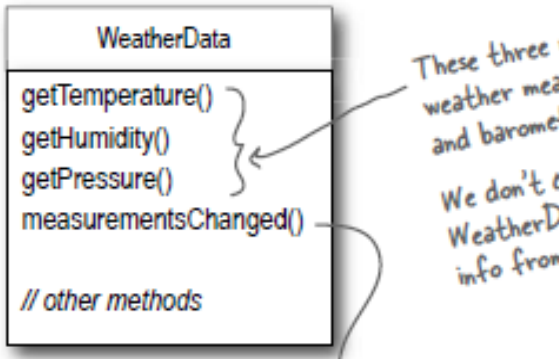
- An application that initially provides three display elements:
  - current conditions,
  - weather statistics
  - simple forecast,

- All updated in real time as the WeatherData object acquires the most recent measurements.

- WeatherData object tracks current weather conditions (temperature, humidity, and barometric pressure).

Unpacking the WeatherData class

As promised, the next morning the WeatherData source files arrive. Peeking inside the code, things look pretty straightforward:



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...

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

WeatherData.java

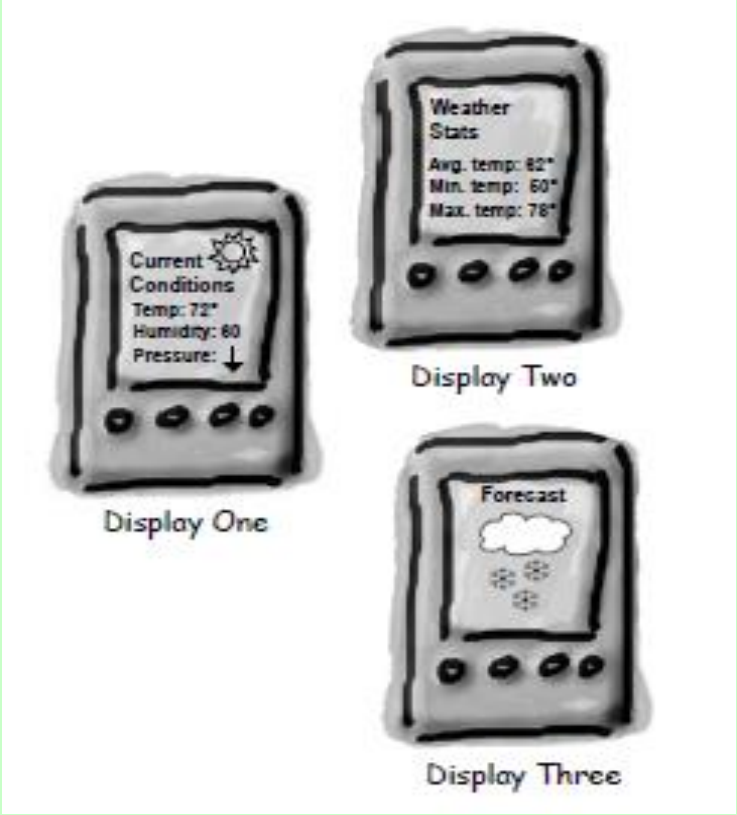
Internet-based Weather Monitoring Station

- The WeatherData class has getter methods for three measurement values:
  - temperature,
  - humidity and
  - barometric pressure.

- The measurementsChanged() method is called any time new weather measurement data is available.
  - We don't know or care how this method is called; we just know that it is.

- The system must be expandable

Internet-based Weather Monitoring Station



First Solution

```
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
}
```

What’s wrong with our implementation?

- Think back to all those Last week concepts and principles...

What's wrong with our implementation?

```
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 takes the temp, humidity, and pressure values.

Newspaper Subscription

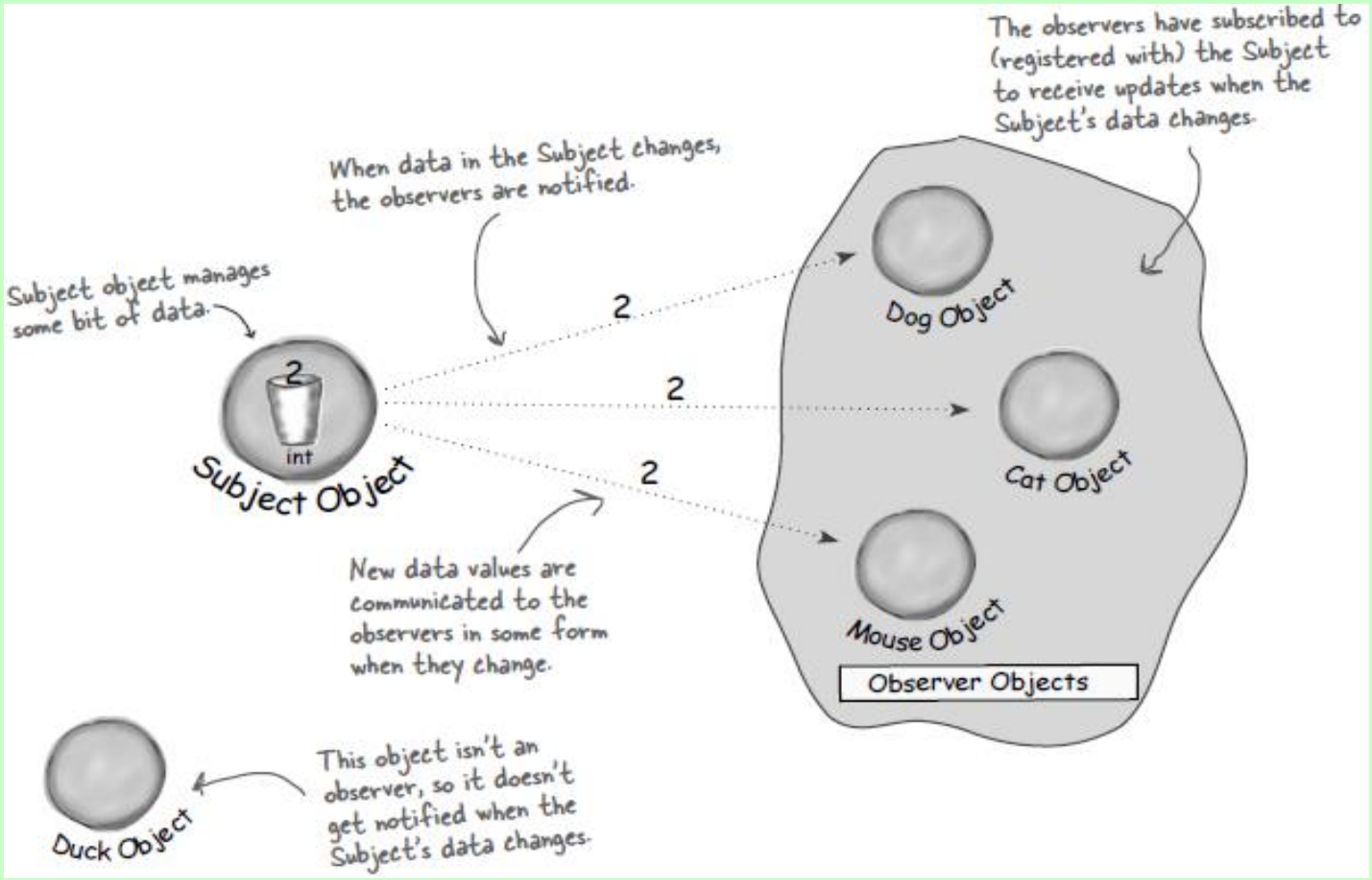
- A newspaper publisher goes into business and begins publishing newspapers.
- You subscribe to a particular publisher, and every time there's a new edition it gets delivered to you. As long as you remain a subscriber, you get new newspapers.
- You unsubscribe when you don't want papers anymore, and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

Other Examples for Subscription

- ?

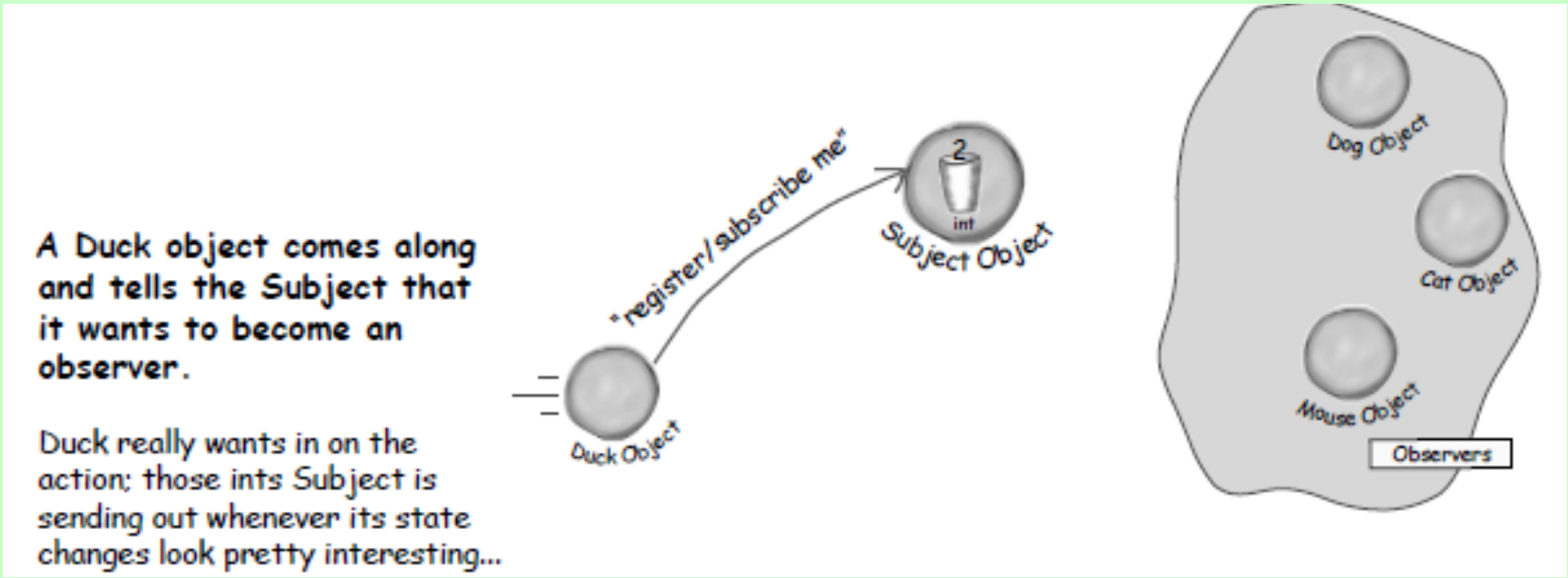
Observer Pattern

- Publishers + Subscribers = Observer Pattern



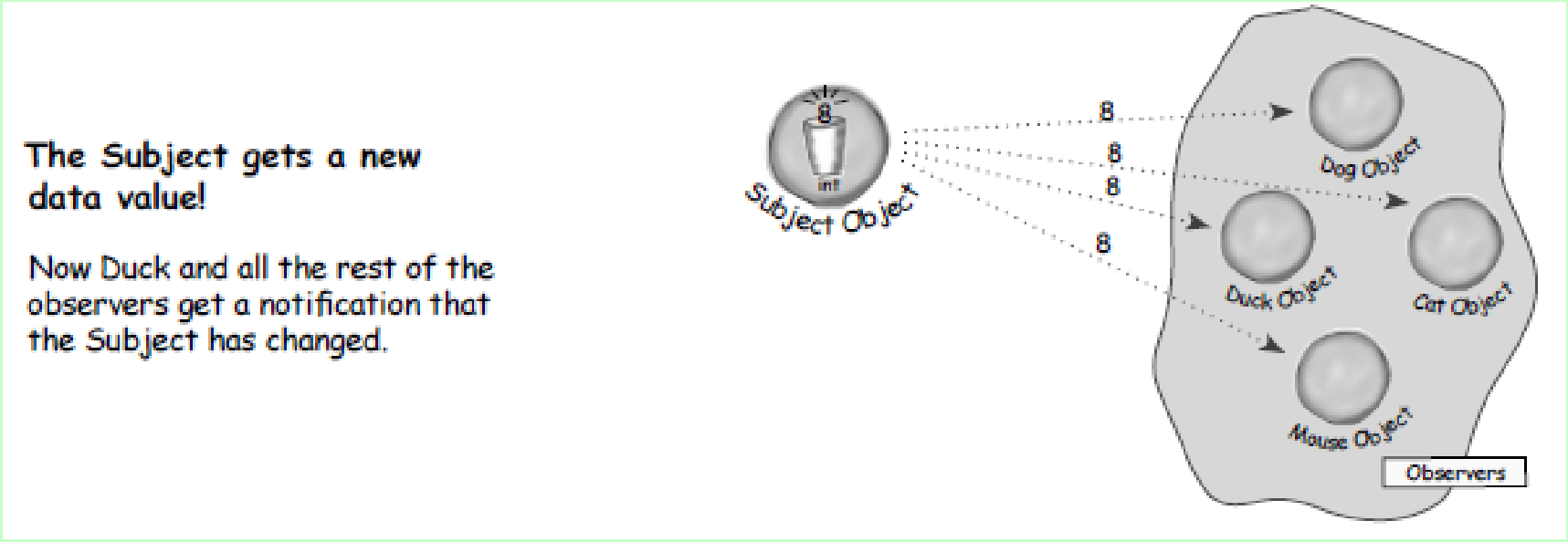
Observer Pattern

- Subscribtion



Observer Pattern

- Subscribtion

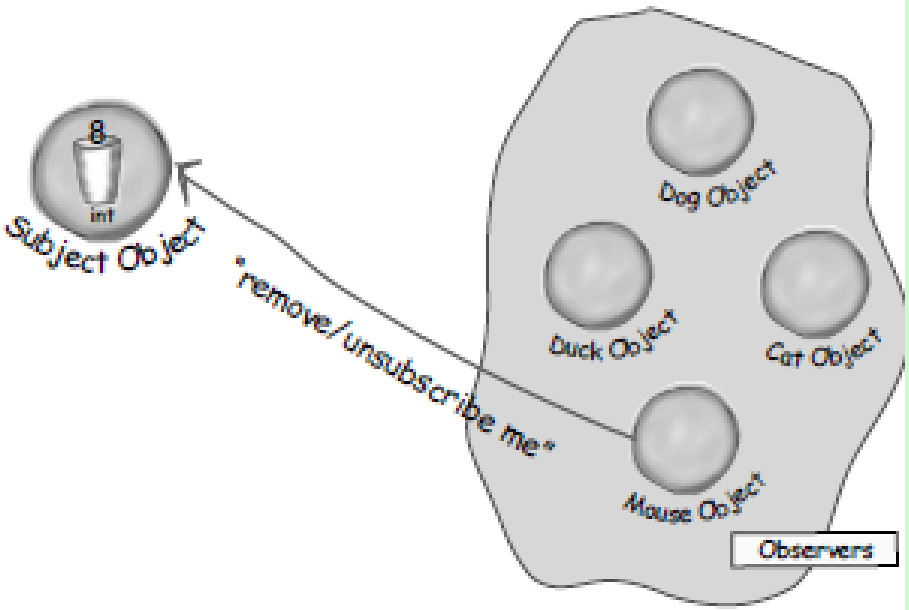


Observer Pattern

- Removal

The Mouse object asks to be removed as an observer.

The Mouse object has been getting ints for ages and is tired of it, so it decides it's time to stop being an observer.

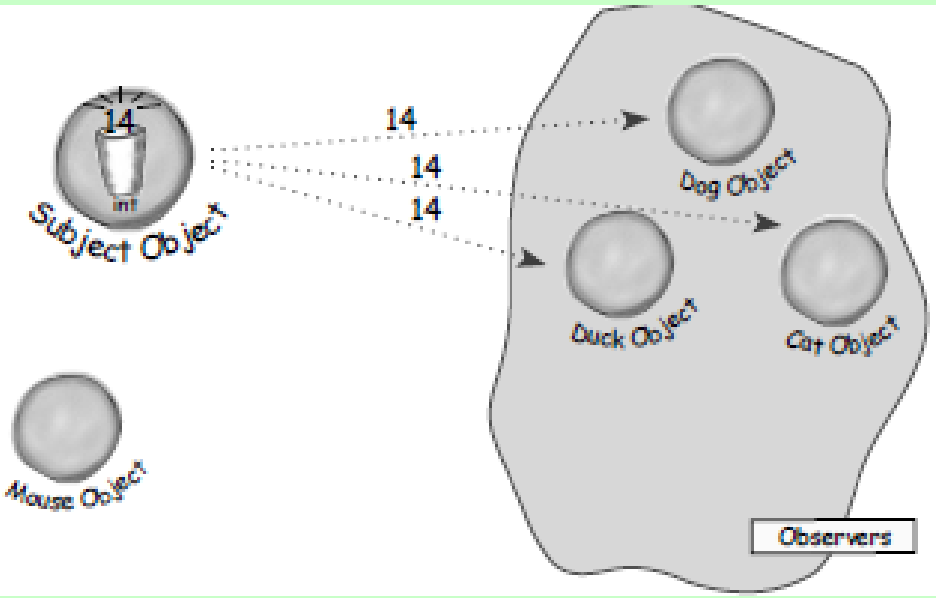


Observer Pattern

- Removal

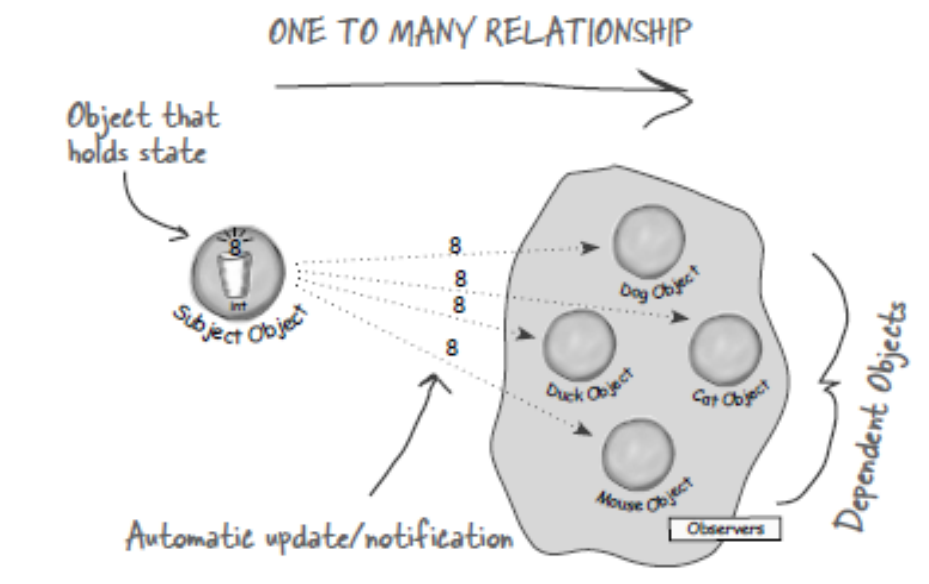
The Subject has another new int.

All the observers get another notification, except for the Mouse who is no longer included. Don't tell anyone, but the Mouse secretly misses those ints... maybe it'll ask to be an observer again some day.

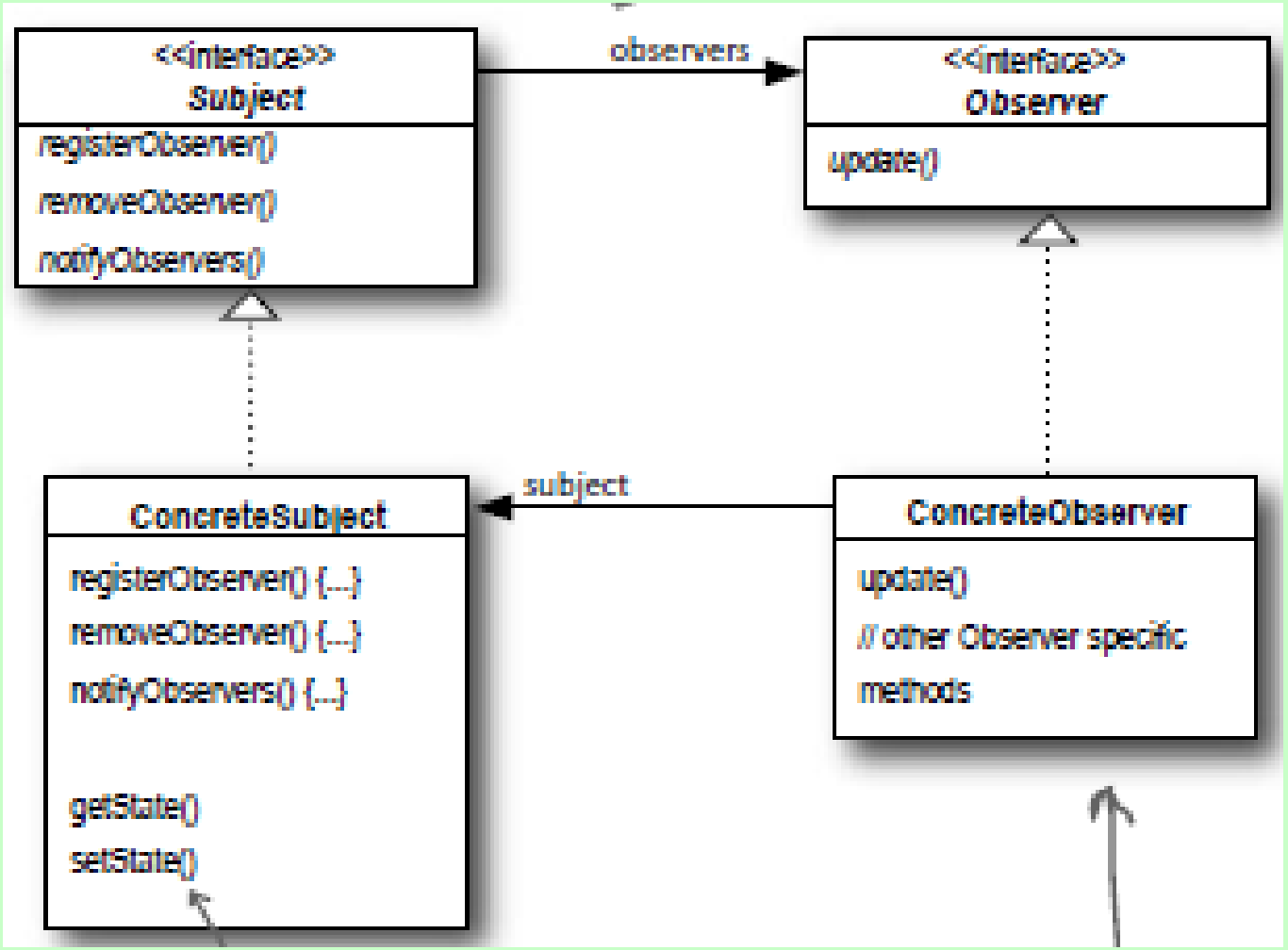


Observer Pattern

- The Observer Pattern defines a one-to-many relationship between a set of objects.
- When the state of one object changes, all of its dependents are notified.



Observer Pattern



Design Principle

- Strive for loosely coupled designs between objects that interact.

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

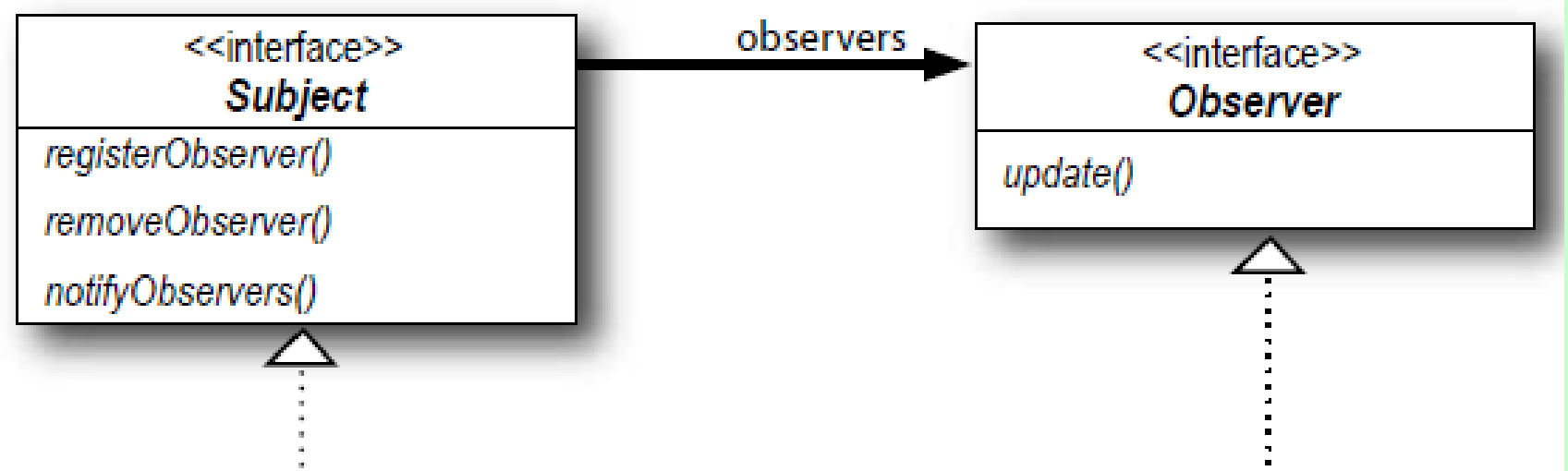
Loose Coupling

- 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.

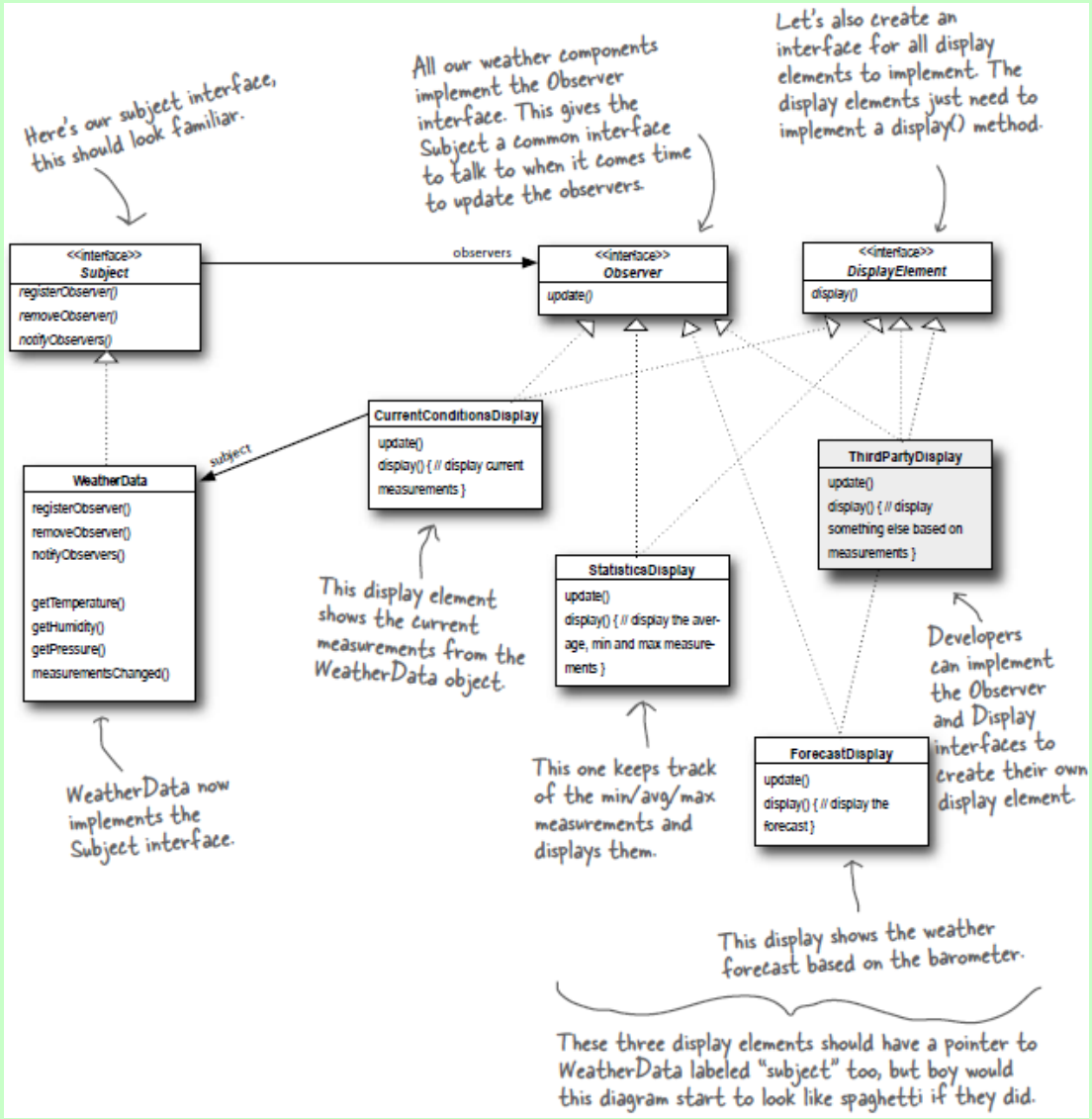
Observer Pattern

- The only thing the subject knows about an observer is that it implements a certain 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.
- Changes to either the subject or an observer will not affect the other.

Redesign Weather Station



Redesign Weather Station



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.

Implementing the WeatherData

```
public class WeatherData implements Subject {
    private ArrayList observers;
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() {
        observers = new ArrayList();
    }

    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 (int i = 0; i < observers.size(); i++) {
            Observer observer = (Observer)observers.get(i);
            observer.update(temperature, humidity, pressure);
        }
    }

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

WeatherData now implements the Subject interface.

We've 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.

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.

Here we implement the Subject interface.

Implementing the WeatherData

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

    // other WeatherData methods here
}
```

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.

Implementing 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");
    }
}
```

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.

Implementing Weather Station

```
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);
    }
}
```

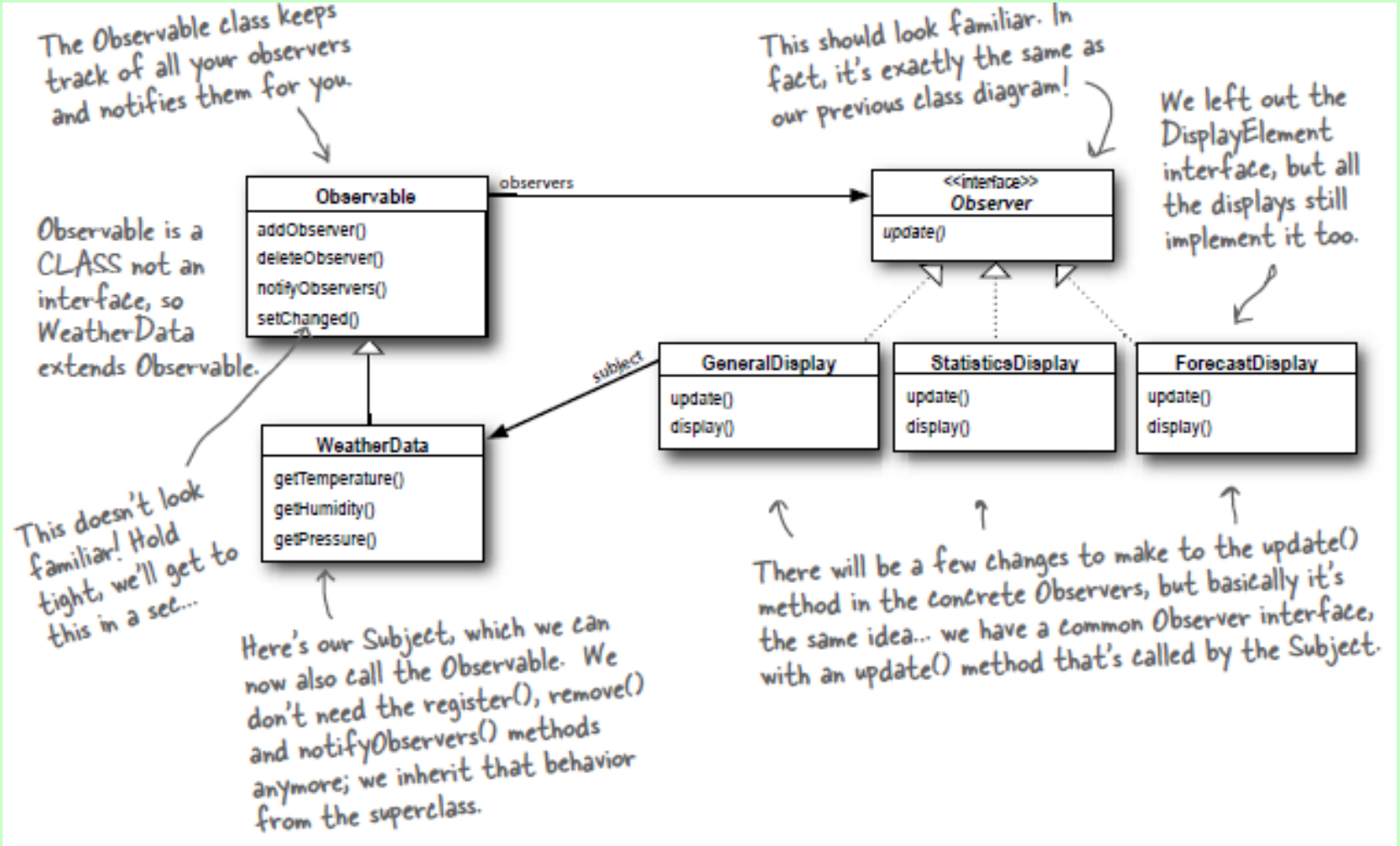
First, create the WeatherData object.

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's built-in Observer Pattern



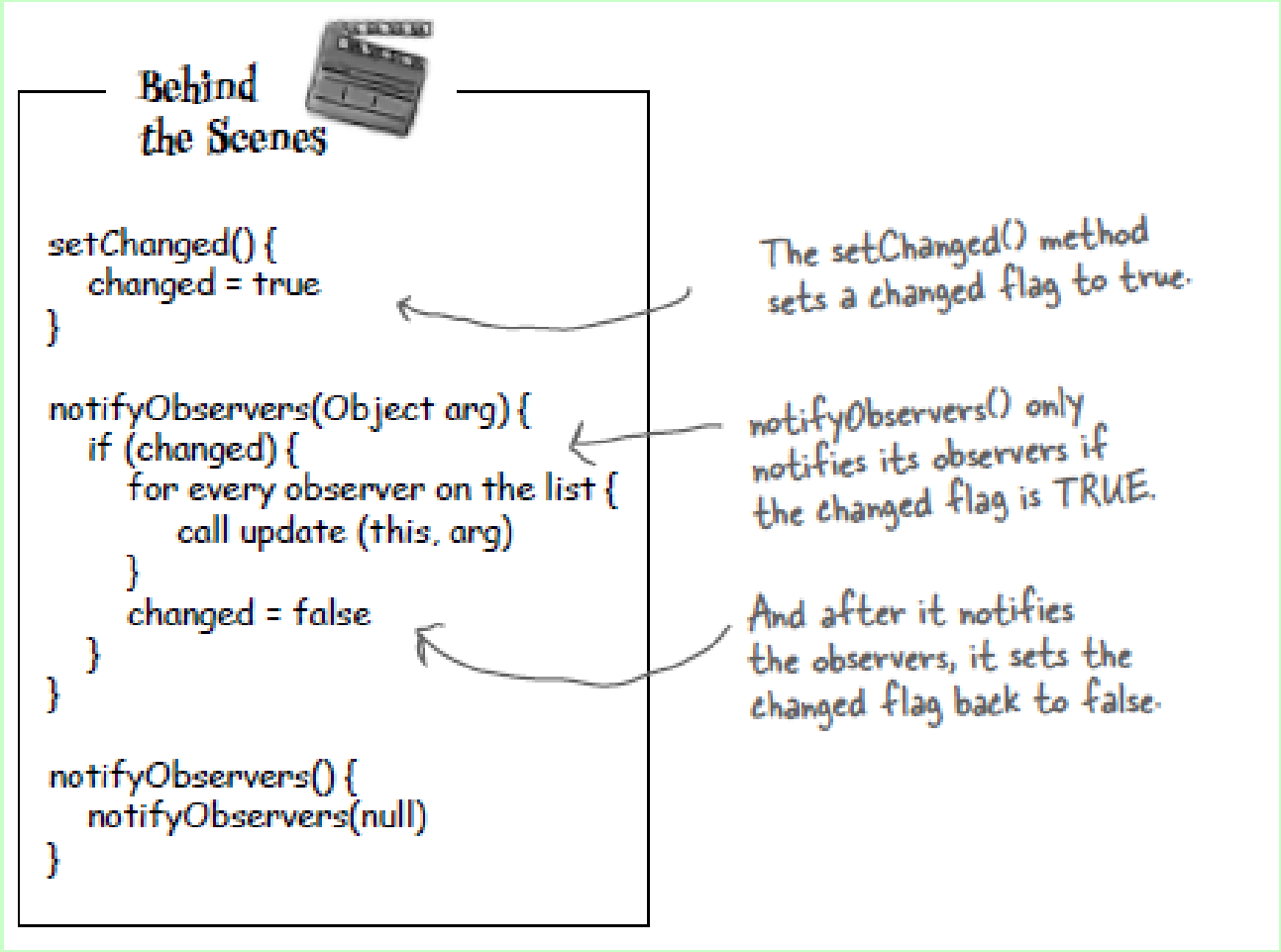
To Become an Observable

- Extend the java.util.Observable superclass
- To send notifications
  - First call the setChanged() method to signify that the state has changed in your object
  - Then, call one of two notifyObservers() methods:
    - notifyObservers()
    - notifyObservers(Object arg)

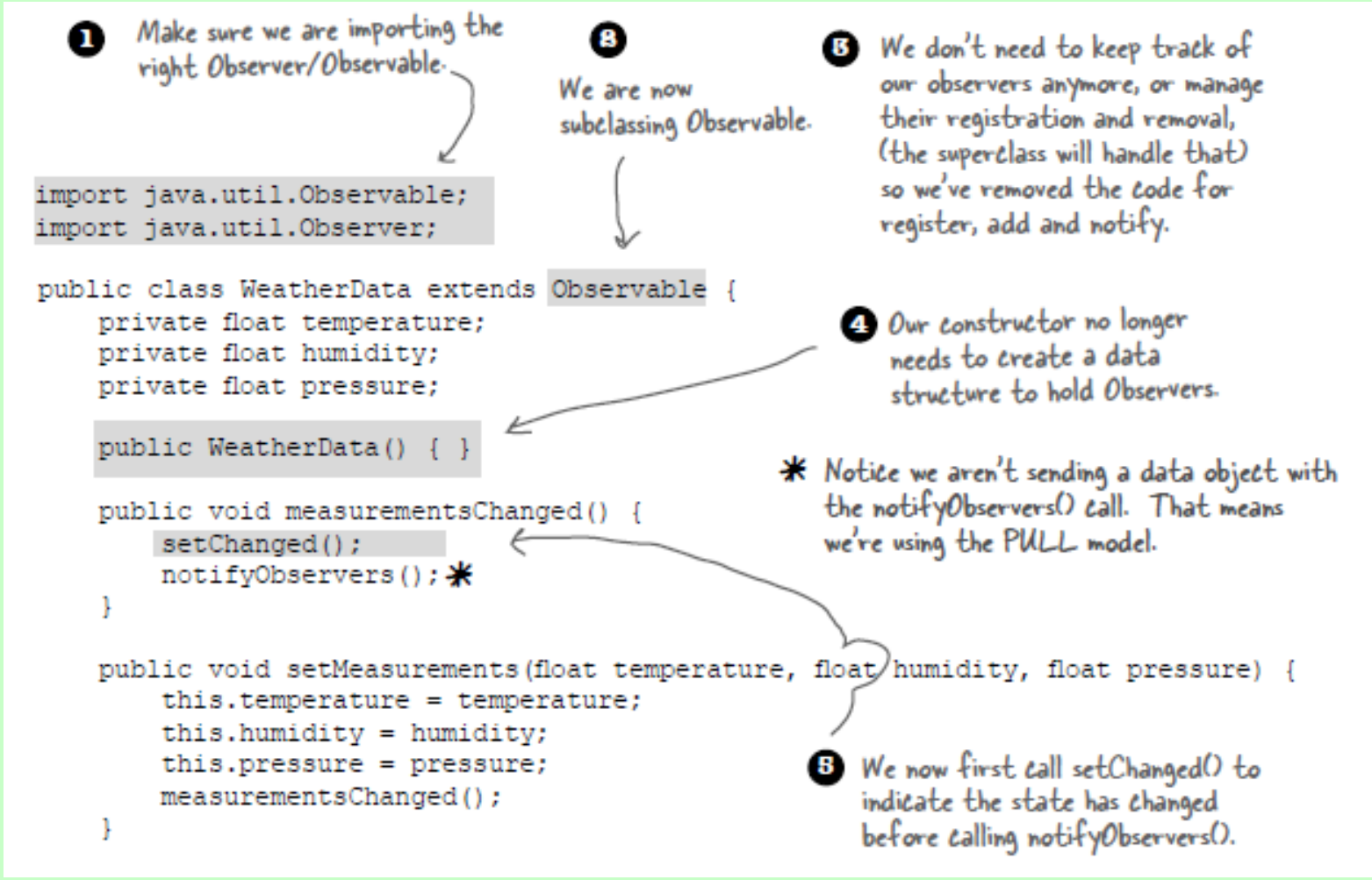
To Become an Observer

- Implement the java.util.Observer interface
- To receive notifications
  - Implement the update method
    - update(Observable o, Object arg)

Observable Class



WeatherData extending Observable



WeatherData extending Observable

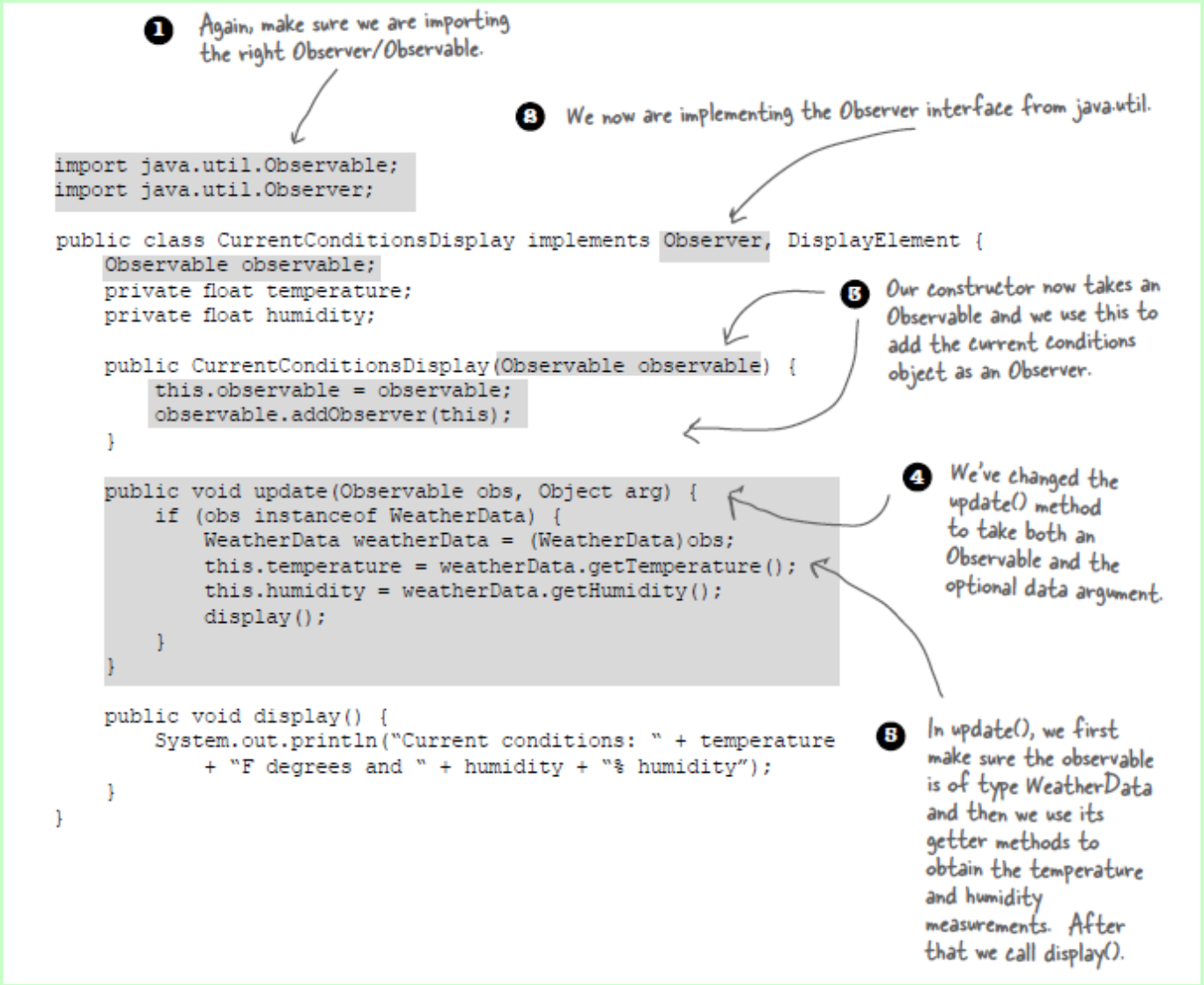
```
    public float getTemperature() {
        return temperature;
    }

    public float getHumidity() {
        return humidity;
    }

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

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.

Display implementing Observer




```
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");
    }
}
```

PropertyChangeSupport

Here is an example of `PropertyChangeSupport` usage that follows the rules and recommendations laid out in the JavaBeans™ specification:

```
public class MyBean {
    private final PropertyChangeSupport pcs = new PropertyChangeSupport(this);

    public void addPropertyChangeListener(PropertyChangeListener listener) {
        this.pcs.addPropertyChangeListener(listener);
    }

    public void removePropertyChangeListener(PropertyChangeListener listener) {
        this.pcs.removePropertyChangeListener(listener);
    }

    private String value;

    public String getValue() {
        return this.value;
    }

    public void setValue(String newValue) {
        String oldValue = this.value;
        this.value = newValue;
        this.pcs.firePropertyChange("value", oldValue, newValue);
    }

    [...]
}
```

Observer Pattern in Swing

- JButton's superclass, AbstractButton
  - has a lot of add/remove listener methods
  - these methods allow you to add and remove observers
  - to listen for various types of events that occur on the Swing component.
- an ActionListener lets you "listen in" on any types of actions that might occur on a button, like a button press

Observer Pattern in Swing

```
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());
        frame.getContentPane().add(BorderLayout.CENTER, button);
        // 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.

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.

Observer Pattern in Android

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Button btnArray = findViewById(R.id.btnArrayAdapter);
        btnArray.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                Log.d(MainActivity.class.getName(), msg: "array button clicked");

                Intent intent = new Intent( packageContext: MainActivity.this,
                        ArrayAdapterActivity.class);
                startActivity(intent);
                //ArrayAdapterActivity activity = new ArrayAdapterActivity()ş
            }
        });
```

Observer Pattern in Android

```
public class ArrayAdapterActivity extends ListActivity {


    static final  String[] ANIMALS = new String[]
            {"Cat", "Dog", "Bee","Bird"};
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        //setContentView(R.layout.activity_array_adapter);

        setListAdapter(new ArrayAdapter<String>( context: this,
                R.layout.activity_array_adapter, ANIMALS));

        ListView l = getListView();
        getListView().setTextFilterEnabled(true);

        l.setOnItemClickListener(new AdapterView.OnItemClickListener() {
            @Override
            public void onItemClick(AdapterView<?> adapterView, View view, int i, long l) {
                Toast.makeText( context: ArrayAdapterActivity.this,
                        ((TextView)view).getText(),Toast.LENGTH_SHORT).show();
            }
        });
```

Summary

- The Observer Pattern defines a one-to-many relationship between objects
- Observables (Subjects), update Observers using a common interface.
- Observers are loosely coupled in that the Observable knows nothing about them,
- You can push or pull data from the Observable when using the pattern

Summary

- Don’t depend on a specific order of notification for your Observers.
- Java has several implementations of the Observer Pattern, including the general purpose java.util.Observable.
- Create your own Observable implementation if needed.
- Swing makes heavy use of the Observer Pattern, as do many GUI frameworks.

References

- Design Patterns: Elements of Reusable Object-Oriented Software By Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides (1995). Addison-Wesley. ISBN 0-201-63361-2.
- **Head First Design Patterns** By Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates
  First Edition October 2004
  ISBN 10: 0-596-00712-4
- http://www.uwosh.edu/faculty_staff/huen/262/f09/slides/10_Strategy_Pattern.ppt