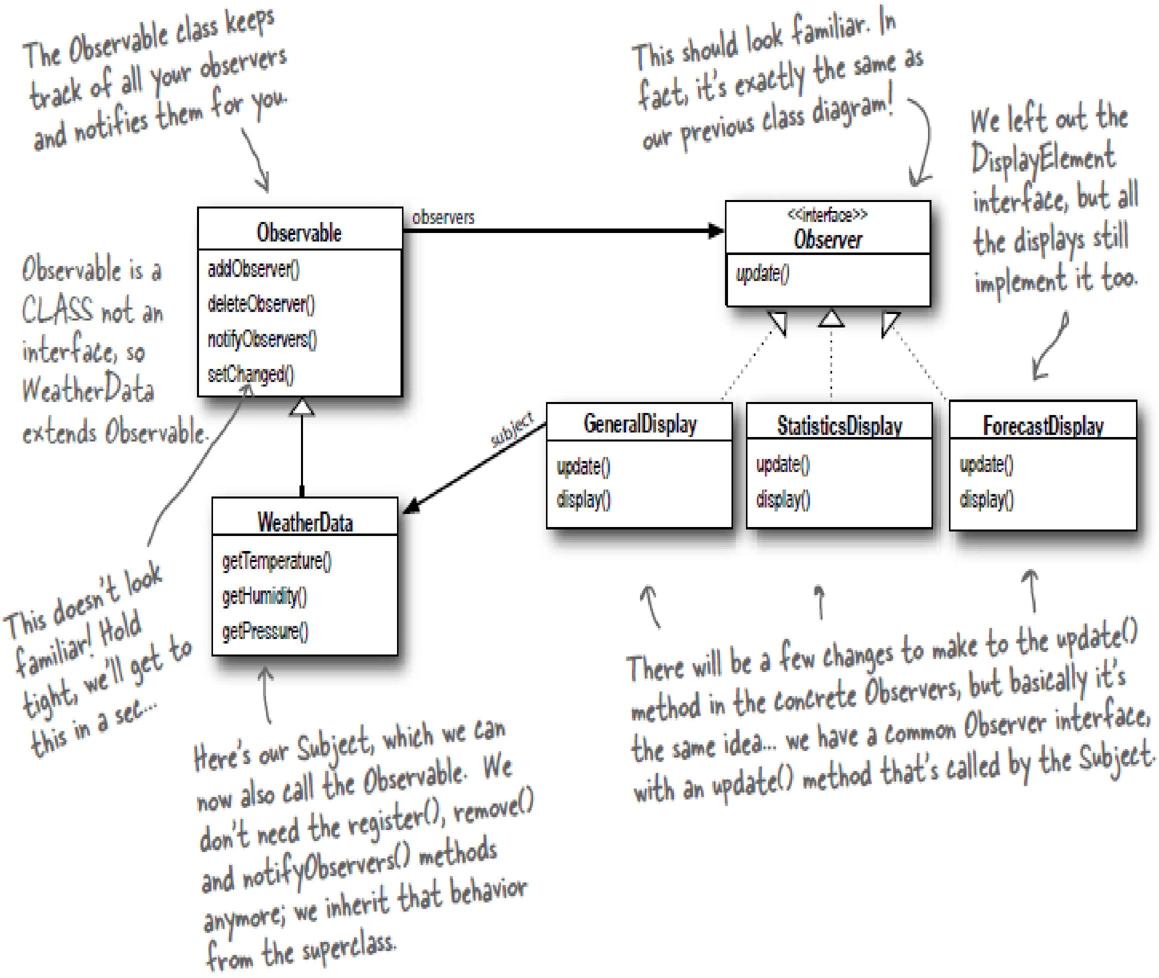
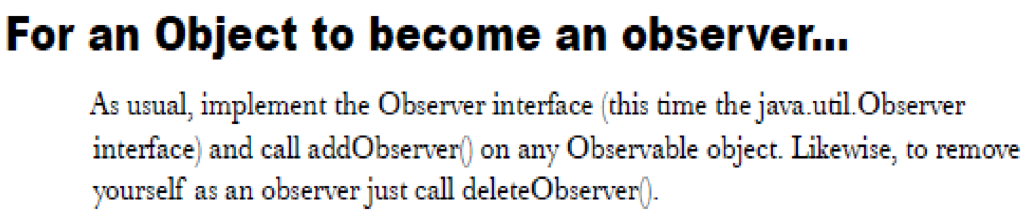
INSTRUCTION:

Follow the steps and rewrite the Weather Monitoring application in Lecture 3 using Java's built-in Observer Pattern

1. Observer interface and Observable class in the java.util package.


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
addObserver()
deleteObserver()
notifyObservers()
setChanged()
observers
<<interface>>
Observer
update()
Observable is a CLASS not an interface, so WeatherData extends Observable.
subject
GeneralDisplay
update()
display()
StatisticsDisplay
update()
display()
ForecastDisplay
update()
display()
WeatherData
getTemperature()
getHumidity()
getPressure()
This doesn't look familiar! Hold tight, we'll get to this in a sec...
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.
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.

2. How Java's built-in Observer Pattern works



For an Object to become an observer...

As usual, 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 of all 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:**

either `notifyObservers()` **or** `notifyObservers(Object arg)`

This version takes an arbitrary data object that gets passed to each Observer when it is notified.

data object

For an Observer to receive notifications...

It implements the update method, as before, but the signature of the method is a bit different:

```
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.

If you want to "push" data to the observers you can pass the data as a data object to the notifyObserver(arg) method. If not, then the Observer has to "pull" the data it wants from the Observable object passed to it. How? Let's rework the Weather Station and you'll see.

Wait, before we get to that, why do we need this setChanged() method? We didn't need that before.

The setChanged() method is used to signify that the state has changed and that notifyObservers() when it is called, should update its observers. If notifyObservers() is called without first calling setChanged(), the observers will NOT be notified. Let's take a look behind the scenes of Observable to see how this works:

Behind the Scenes

```
setChanged() {
    changed = true
}

notifyObservers(Object arg) {
    if (changed) {
        for every observer on the list {
            call update (this, arg)
        }
        changed = false
    }
}

notifyObservers() {
    notifyObservers(null)
}
```

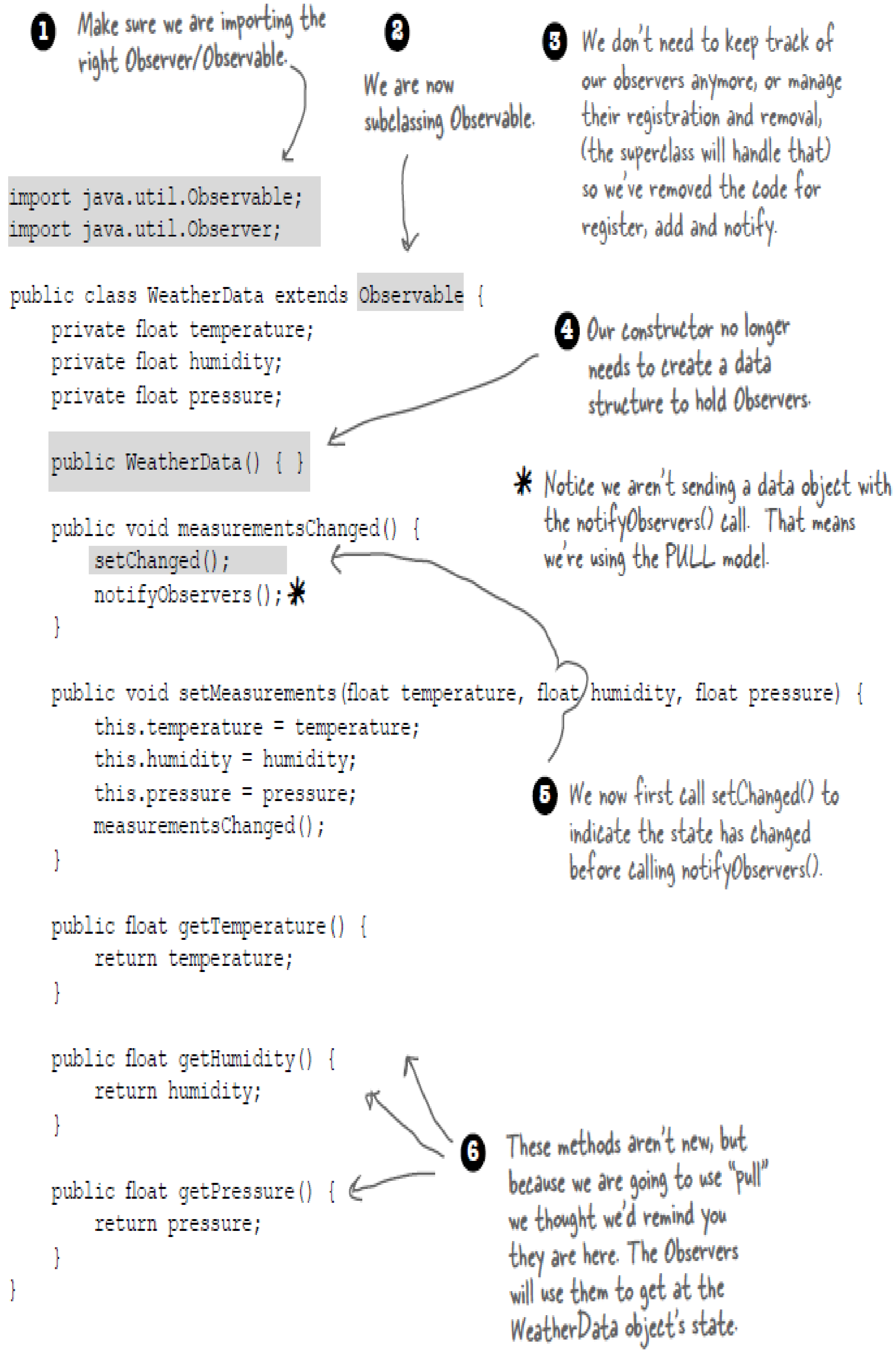
Pseudocode for the Observable Class.

The setChanged() method sets a changed flag to true.

notifyObservers() only notifies its observers if the changed flag is TRUE.

And after it notifies the observers, it sets the changed flag back to false.

3. Reworking the Weather Station with the built-in support



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

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;
    }
}
```

1 Make sure we are importing the right Observer/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 code for register, add and notify.

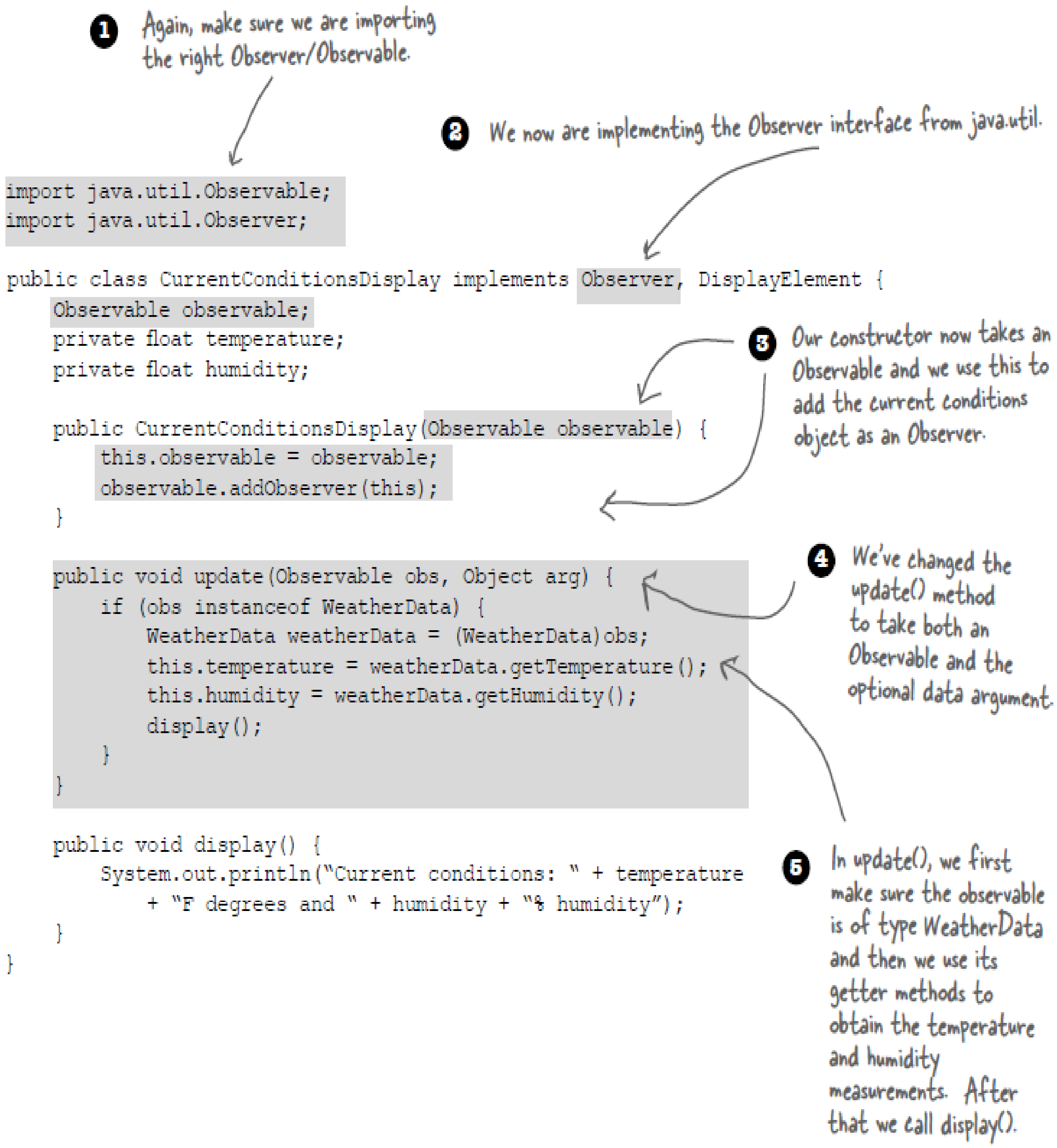
4 Our constructor no longer needs to create a data structure to hold Observers.

* 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.

4. Let’s rework the CurrentConditionsDisplay



```
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”);
    }
}
```

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

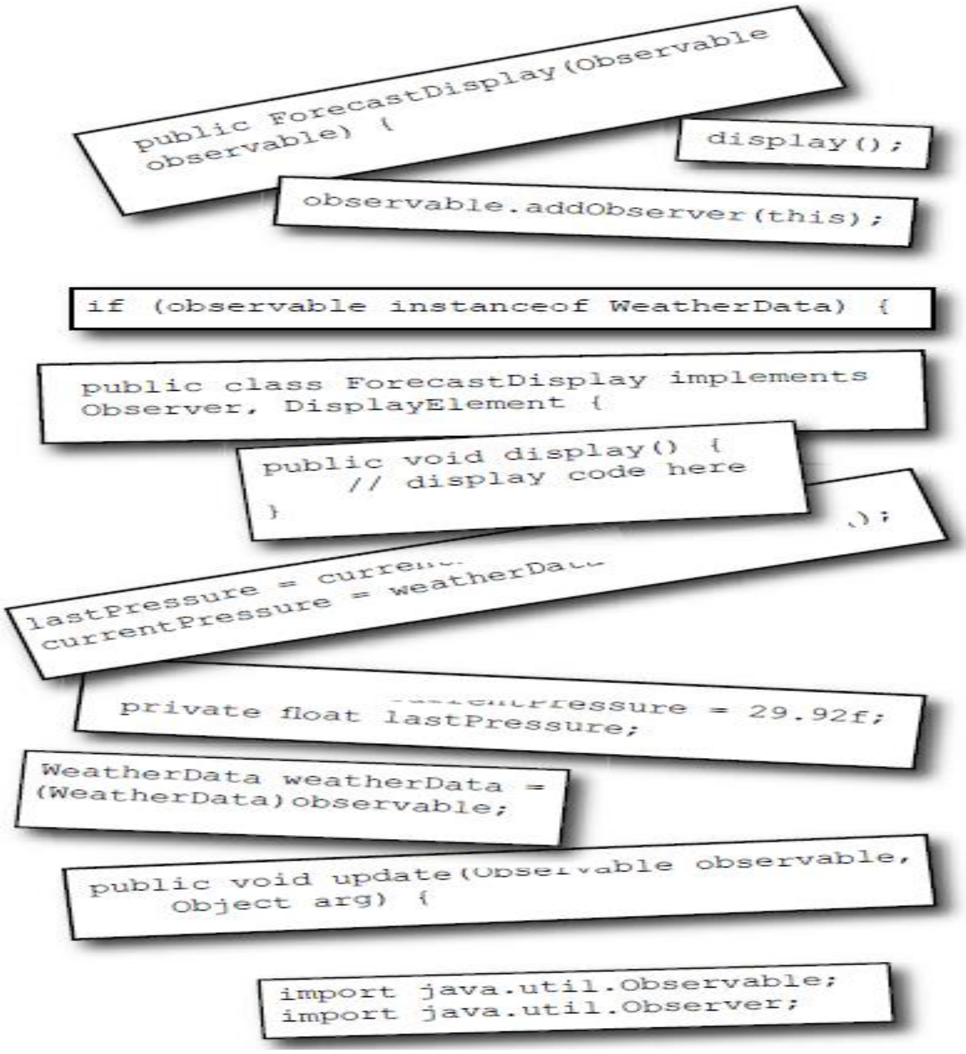
2 We now are implementing the Observer interface from java.util.

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().

5. The ForeCastDisplay class is all scrambled up on the fridge. Can you reconstruct the code snippets to make it work? Some of the curly braces feel on the floor and they were too small to pick up, so feel free to add as many of those as you need.



```
public ForecastDisplay(Observable
observable) {
```

```
display();
```

```
observable.addObserver(this);
```

```
if (observable instanceof WeatherData) {
```

```
public class ForecastDisplay implements
Observer, DisplayElement {
```

```
public void display() {
    // display code here
}
```

```
lastPressure = curren
currentPressure = weatherDat
                                        );
```

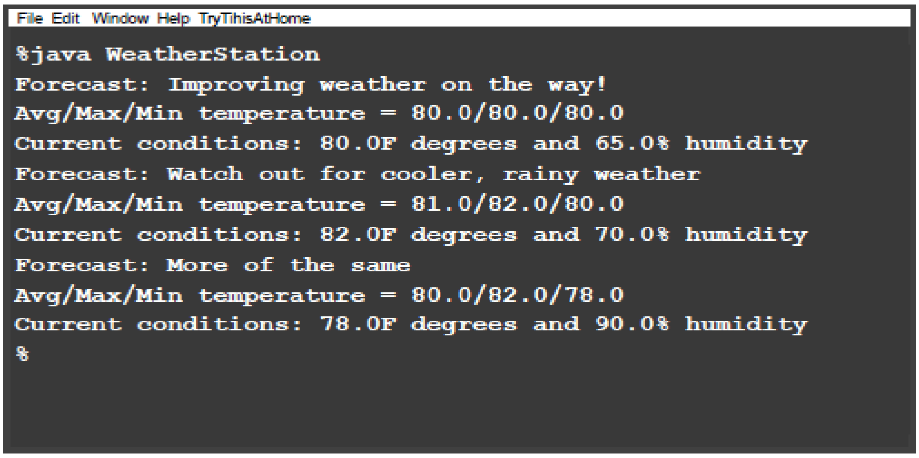
```
private float       entPressure = 29.92f;
                lastPressure;
```

```
WeatherData weatherData =
(WeatherData)observable;
```

```
public void update(Observable observable,
    Object arg) {
```

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

6. Run the new code. Do you notice anything different?

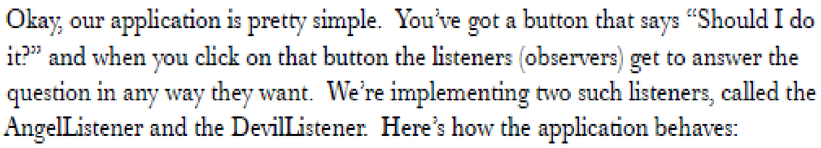


```
%java WeatherStation
Forecast: Improving weather on the way!
Avg/Max/Min temperature = 80.0/80.0/80.0
Current conditions: 80.0F degrees and 65.0% humidity
Forecast: Watch out for cooler, rainy weather
Avg/Max/Min temperature = 81.0/82.0/80.0
Current conditions: 82.0F degrees and 70.0% humidity
Forecast: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

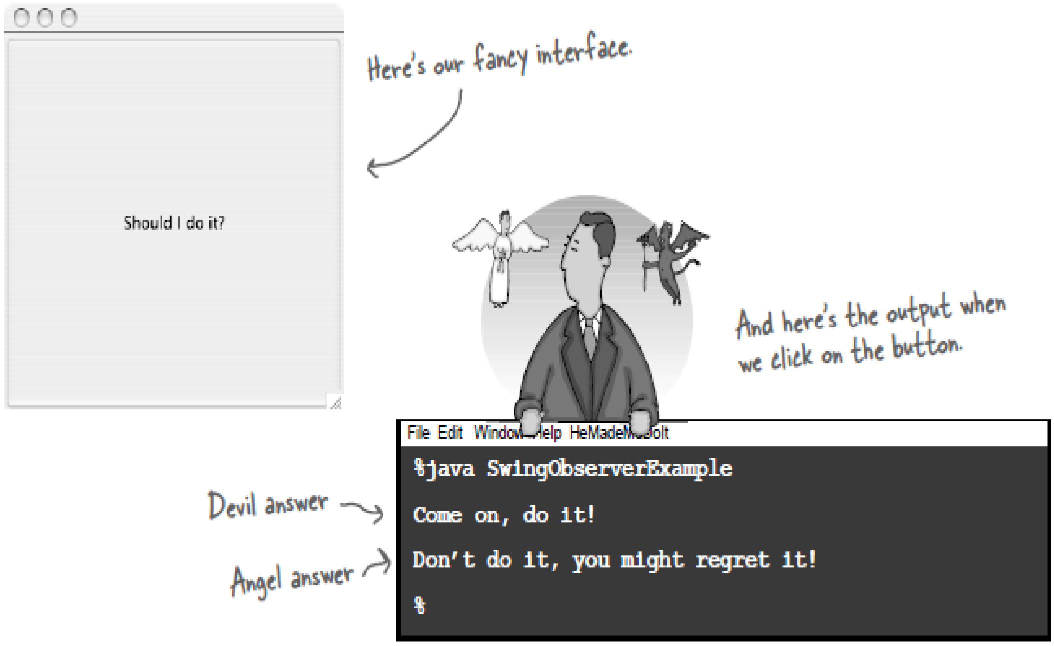
Never depend on the order of evaluation of the Observer notifications.

7. The dark side of java.util.Observable
   Observable is a class.

8. Swing also implements Observer Pattern. Observers are called listeners in Swing. Below is a little life-changing application. Type the code, run and compile it.



Okay, our application is pretty simple. 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:



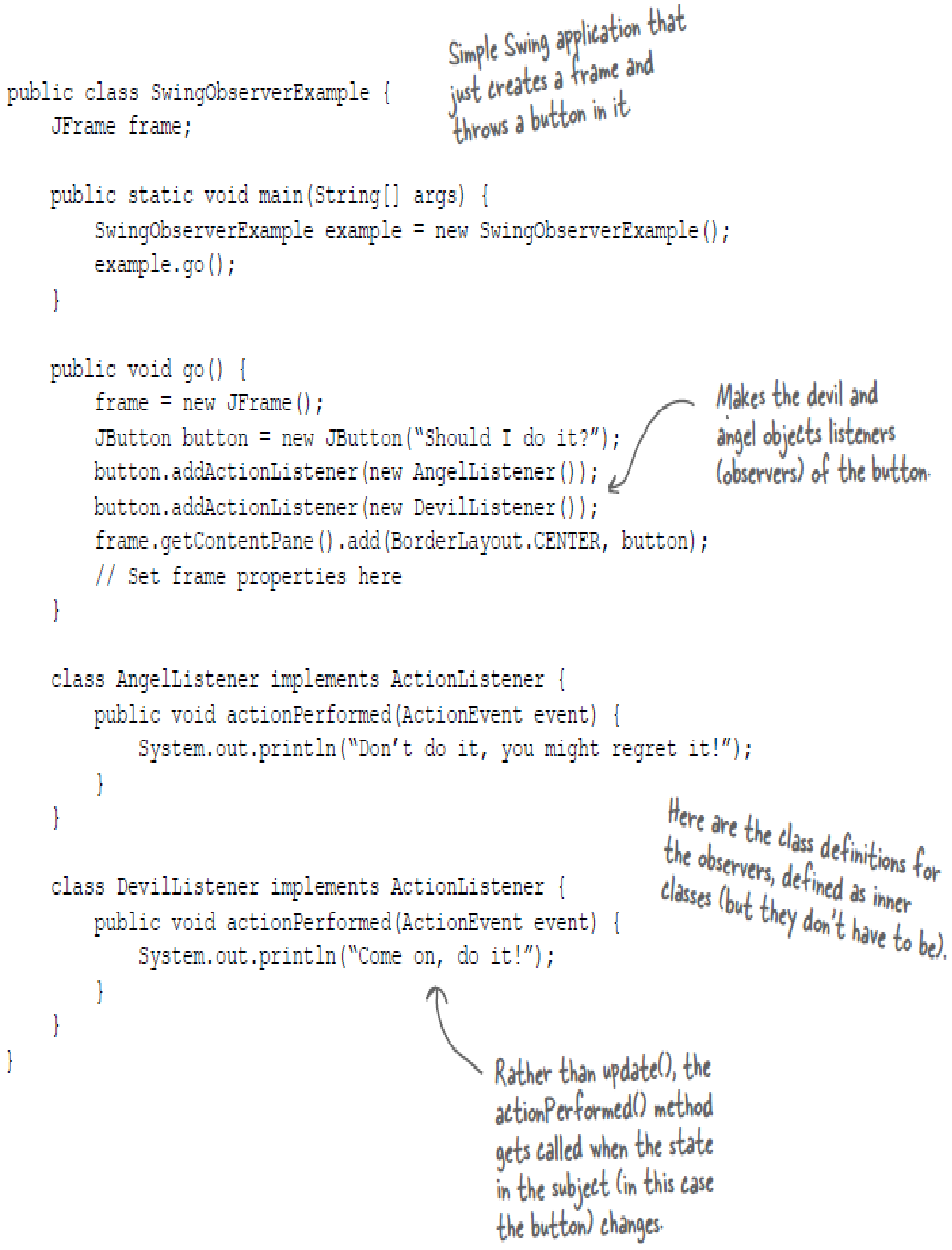
Here's our fancy interface.

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

Devil answer

Angel answer

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



```
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!”);
        }
    }
}
```

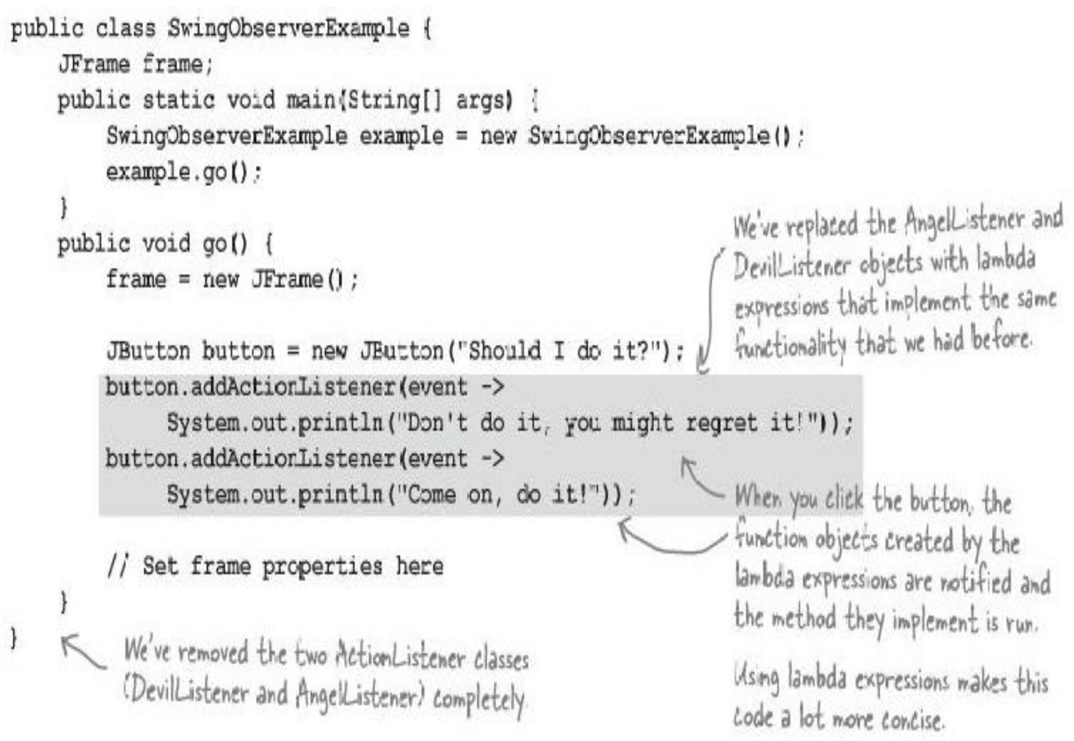
Simple Swing application that just creates a frame and throws a button in it

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.

9. Updated with lambda expression:



```
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(event ->
            System.out.println("Don't do it, you might regret it!"));
        button.addActionListener(event ->
            System.out.println("Come on, do it!"));

        // Set frame properties here
    }
}
```

We've replaced the AngelListener and DevilListener objects with lambda expressions that implement the same functionality that we had before.

When you click the button, the function objects created by the lambda expressions are notified and the method they implement is run.

Using lambda expressions makes this code a lot more concise.

We've removed the two ActionListener classes (DevilListener and AngelListener) completely.

```
package headfirst.designpatterns.observer.swing;

import java.awt.*;
import javax.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?");

                // Without lambdas
                //button.addActionListener(new AngelListener());
                //button.addActionListener(new DevilListener());

                // With lambdas
                button.addActionListener(event ->
                        System.out.println("Don't do it, you might regret it!")
                );
                button.addActionListener(event ->
                        System.out.println("Come on, do it!")
                );
                frame.getContentPane().add(BorderLayout.CENTER, button);

                // Set frame properties
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.getContentPane().add(BorderLayout.CENTER, button);
                frame.setSize(300,300);
                frame.setVisible(true);
        }

        /*
         * Remove these two inner classes to use lambda expressions instead.
         *
        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!");
                }
        }
        */

}
```