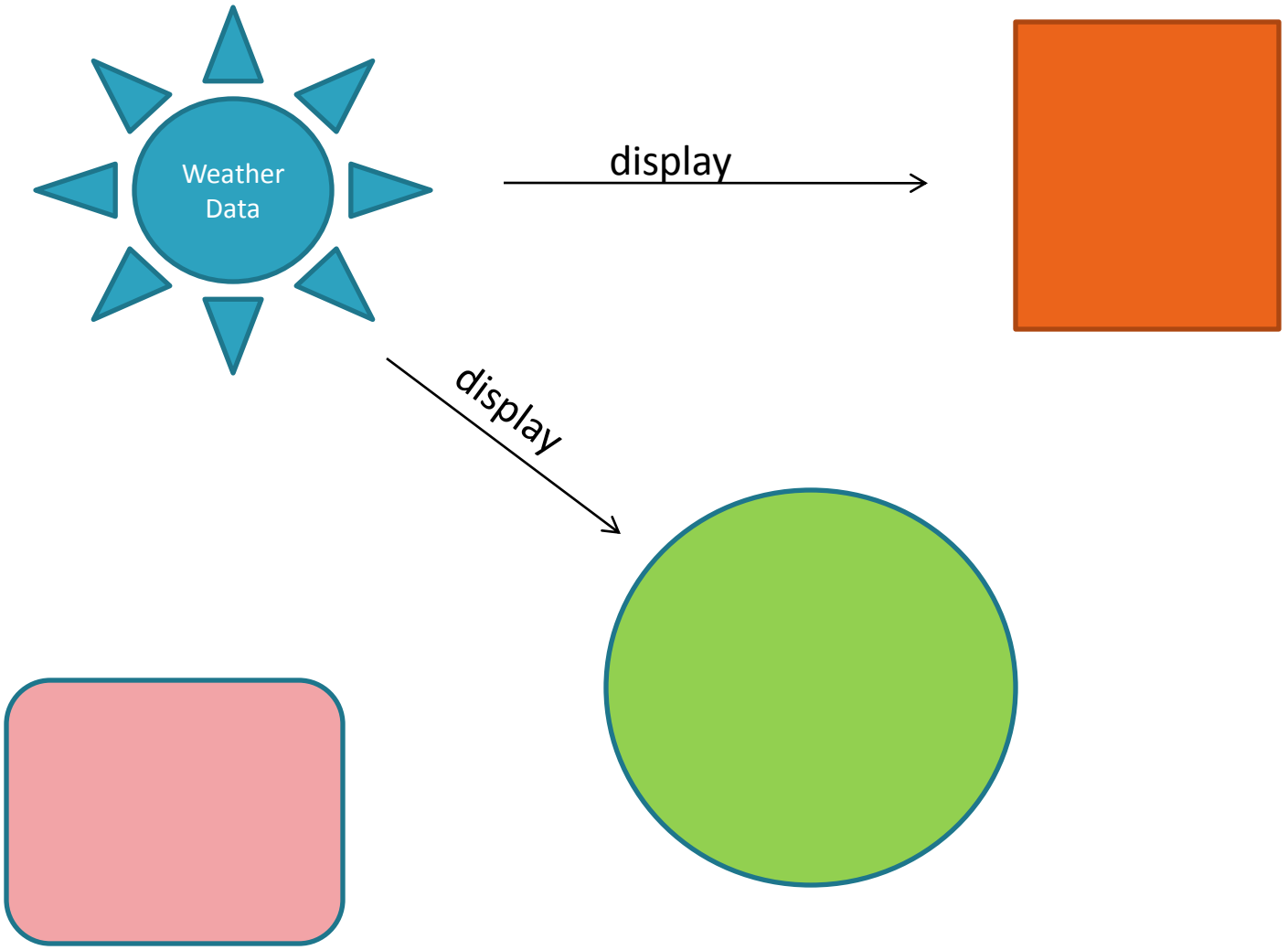
Design Pattern

Observer Pattern

Weather Data



TestDisplay

```
package nonpattern;
import javax.swing.*;

public class TextDisplay extends JPanel {

    JLabel label;

    public TextDisplay() {
        label = new JLabel("  ");
        add(label);
    }

    public void update(int temp) {
        label.setText(""+temp);
        validate();
    }
}
```

SliderDisplay

```
package nonpattern;
import javax.swing.*;

public class SliderDisplay extends JPanel {

    JSlider slider;

    public SliderDisplay() {
        slider = new JSlider();
        add(slider);
    }

    public void update(int value) {
        slider.setValue(value);
        validate();
    }

}
```

WeatherData

```
package nonpattern;

public class WeatherData {
    int temperature;
    TextDisplay text;
    SliderDisplay slider;

    public void addDisplay(TextDisplay display) {
        this.text = display;
    }

    public void addDisplay(SliderDisplay display) {
        this.slider = display;
    }
```

WeatherData

```
    public void measurementChanged() {
        temperature = getTemperature();

        text.update(temperature);
        slider.update(temperature);
    }

    public int getTemperature() {
        return (int)(Math.random()*100);
    }

}
```

Main Application

```
public class MainApp extends JFrame {

    TextDisplay text;
    // add new display here
    SliderDisplay slider;

    public static void main(String[] args) {
        final WeatherData weather = new WeatherData();
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new MainApp(weather).initGUI();
            }
        });
```

Main Application

```
        while(true) {
            try {
                Thread.sleep(1000);
                weather.measurementChanged();
            } catch (InterruptedException ex) {
                Logger.getLogger(
                        MainApp.class.getName()).log(
                        Level.SEVERE, null, ex);
            }
        }
    }
```

Main Application

```
public MainApp(WeatherData w) {
    super("Non Pattern Application");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // add display here
    text = new TextDisplay();
    w.addDisplay(text);

    // init new display
    slider = new SliderDisplay();
    w.addDisplay(slider);
}
```

Main Application

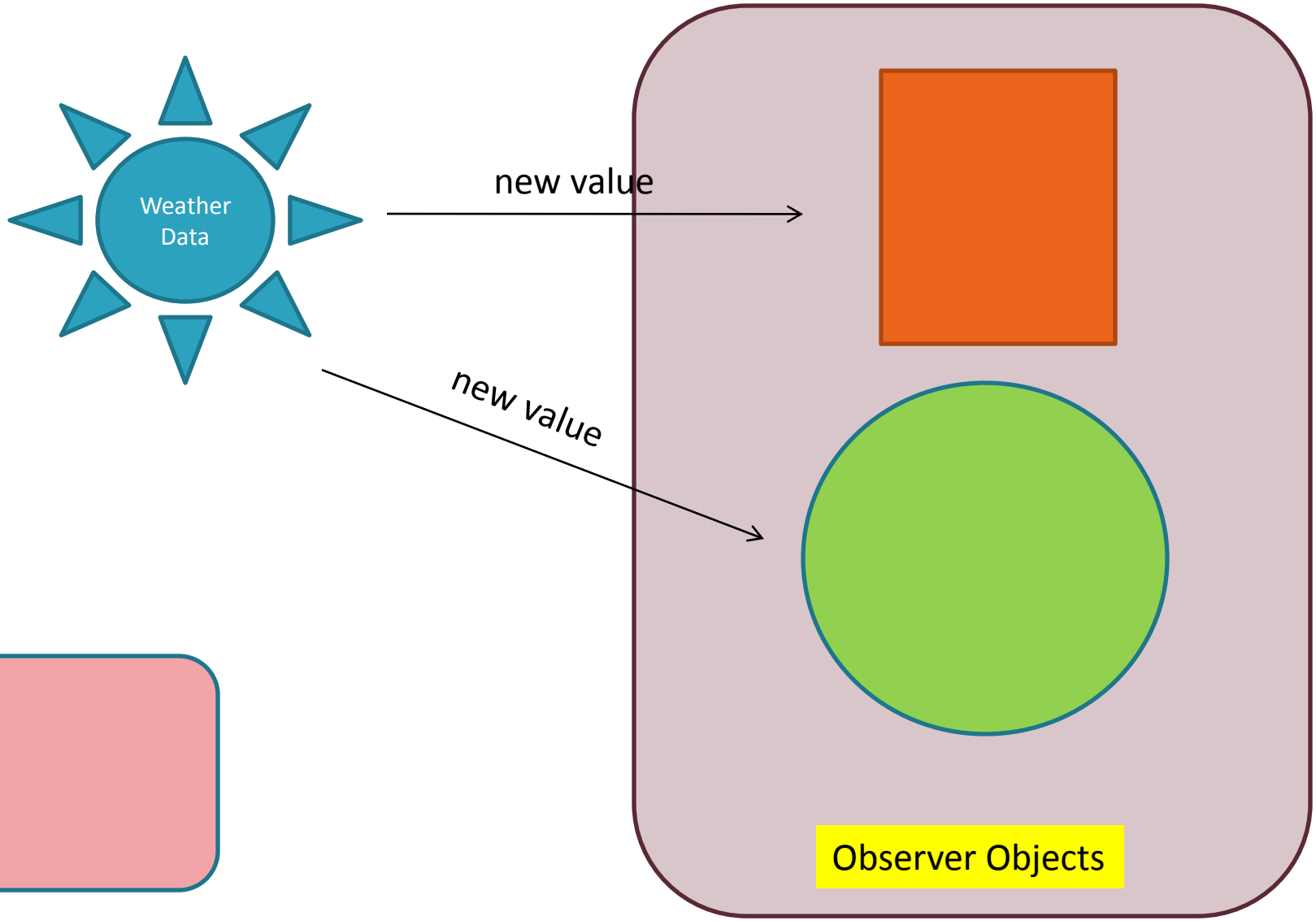
```
    public void initGUI() {
        Container cp = getContentPane();
        cp.setLayout(new BorderLayout());

        cp.add(text, BorderLayout.LINE_END);

        // add new display to frame
        cp.add(slider,BorderLayout.PAGE_END);

        pack();
        setVisible(true);
    }
}
```

New Display to Be Added

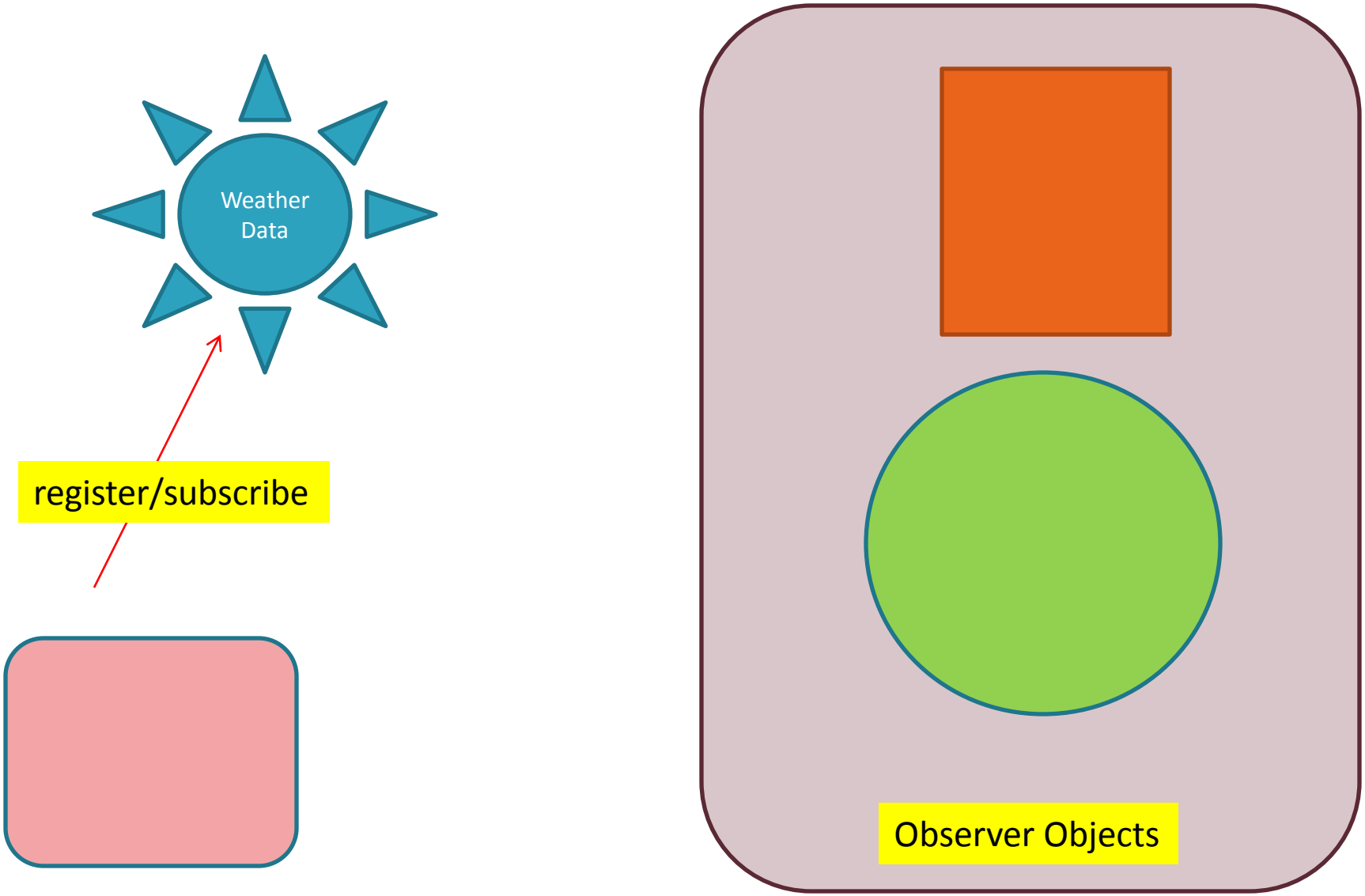
- WeatherData
  - add new display field
  - add new methods to set the display field
  - add new display update statement

- Need to change the program!

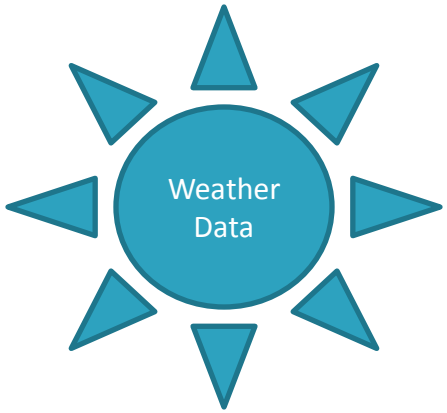
Observer Pattern

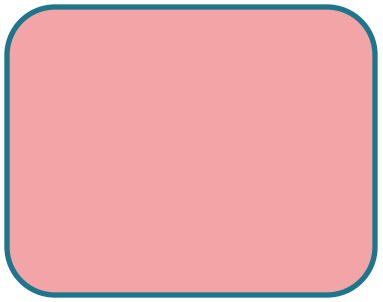


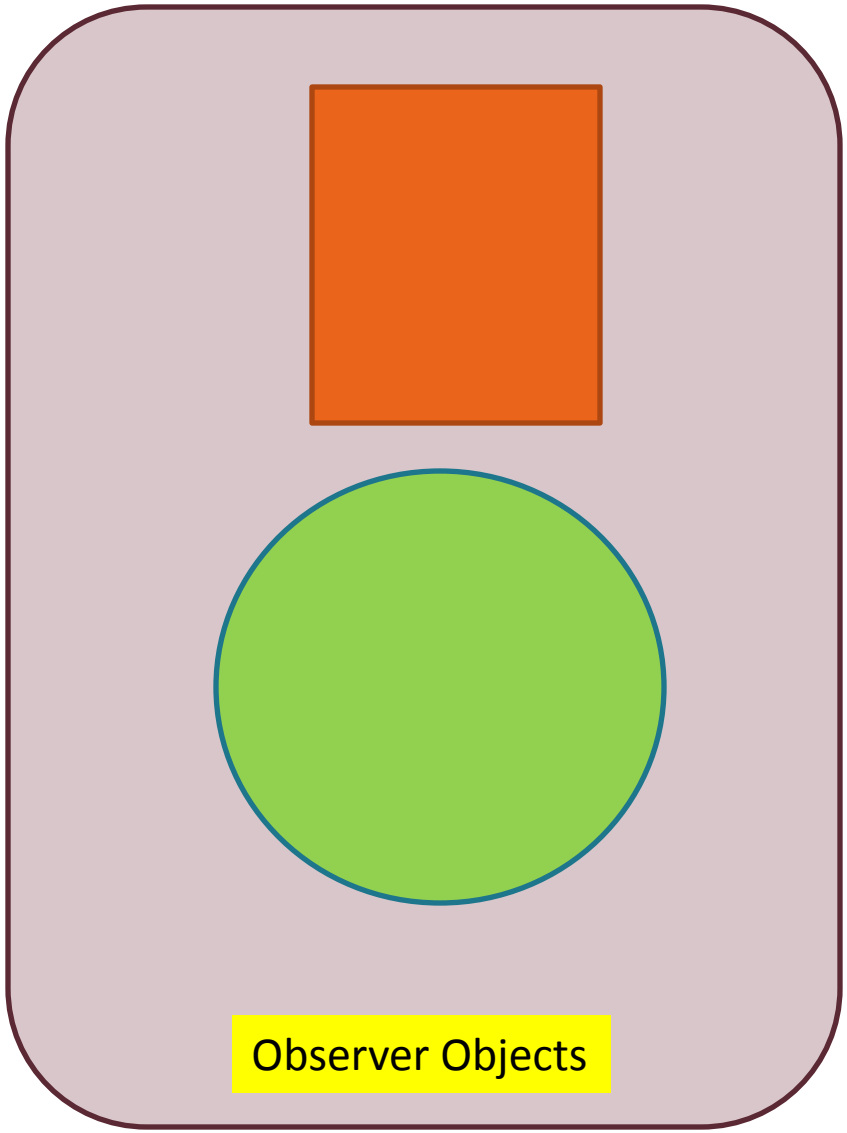
Observer Pattern



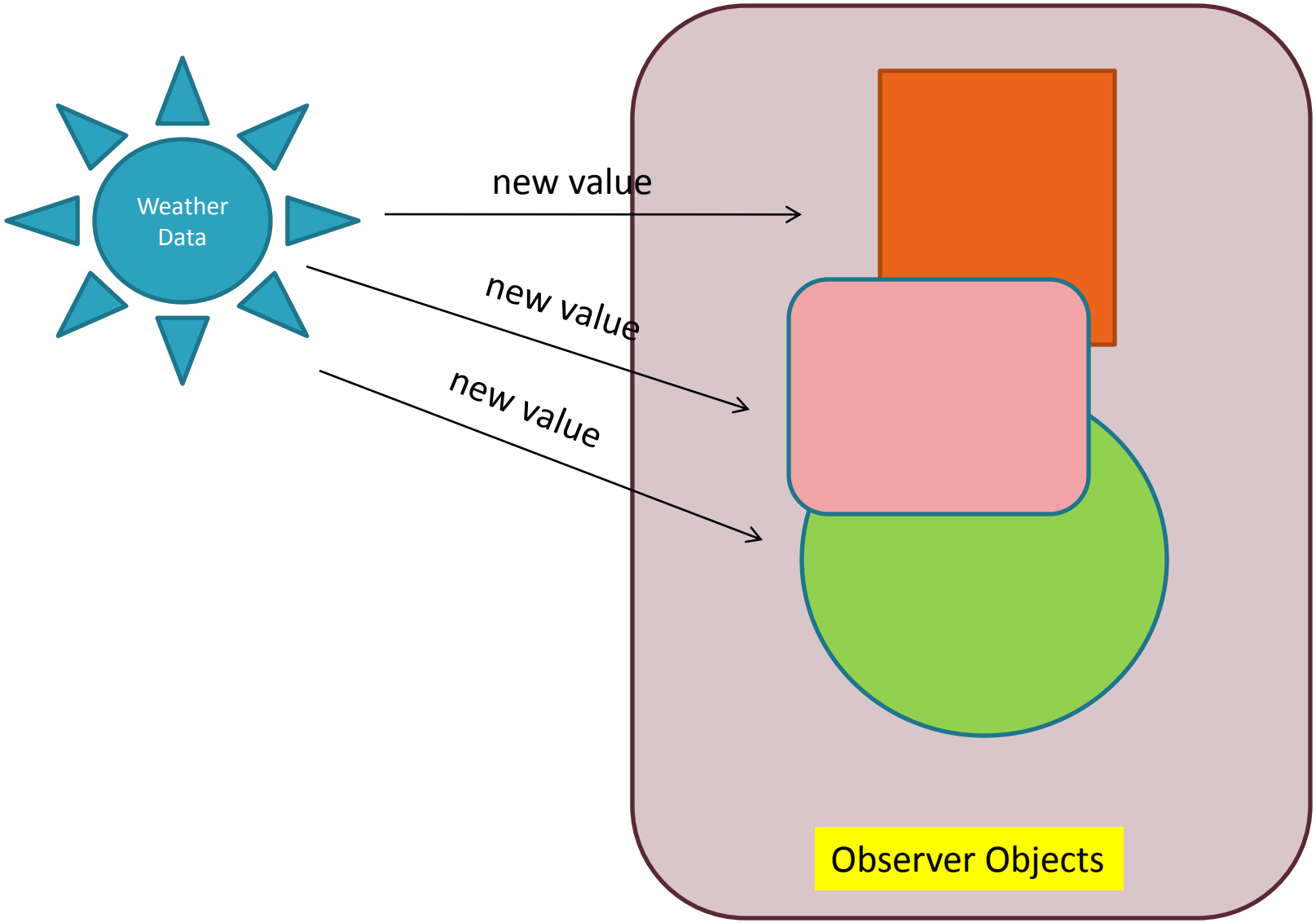
Observer Pattern



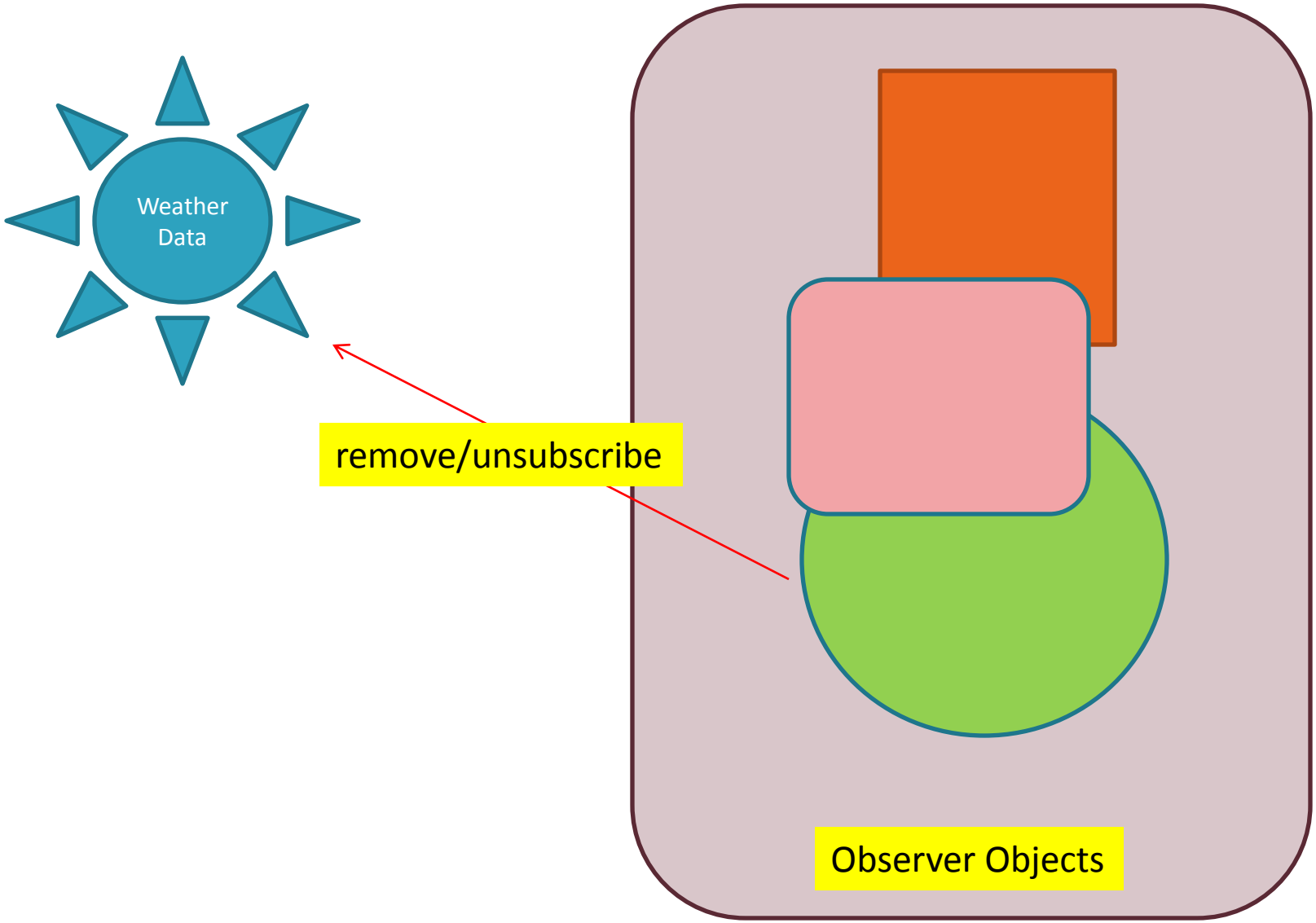




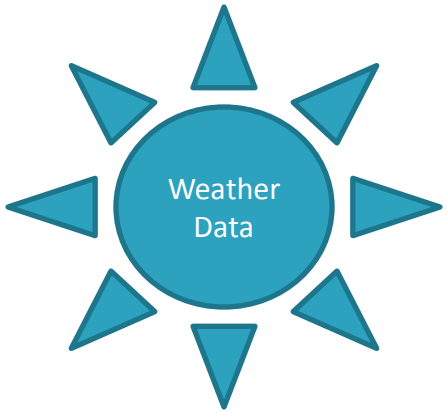
Observer Pattern

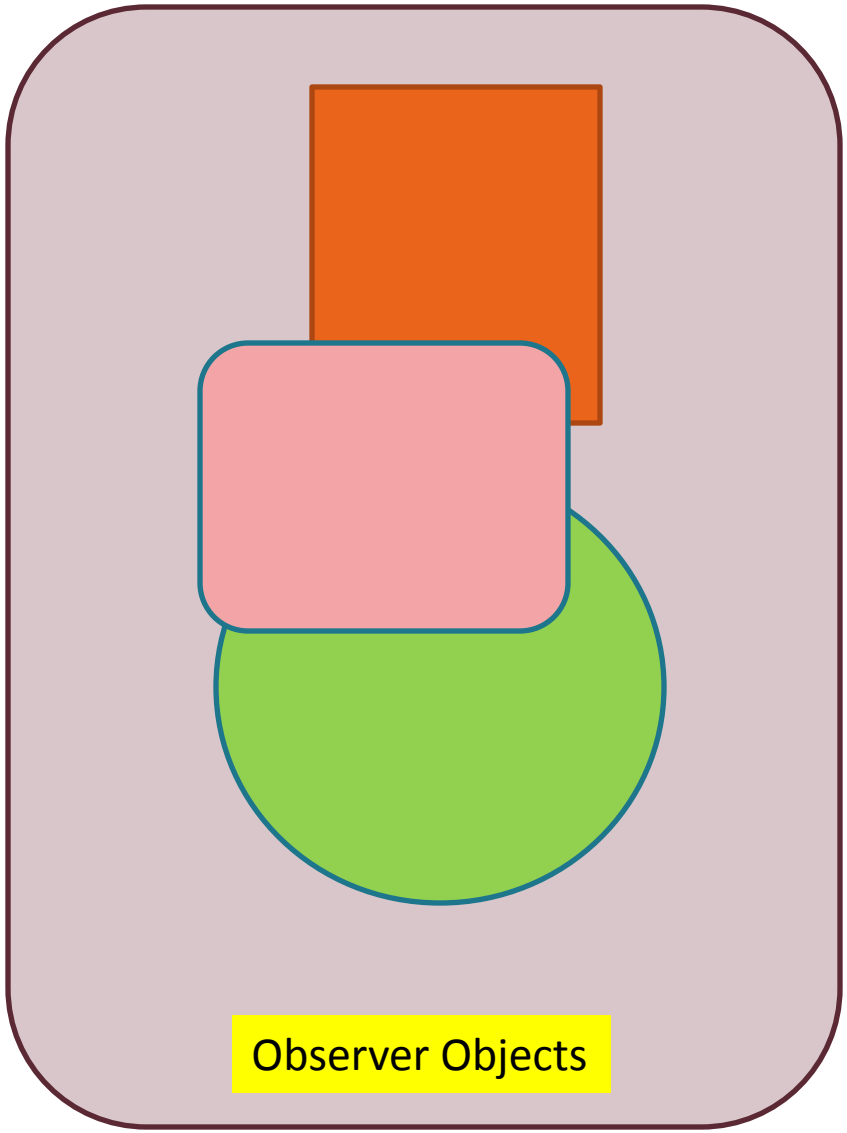


Observer Pattern

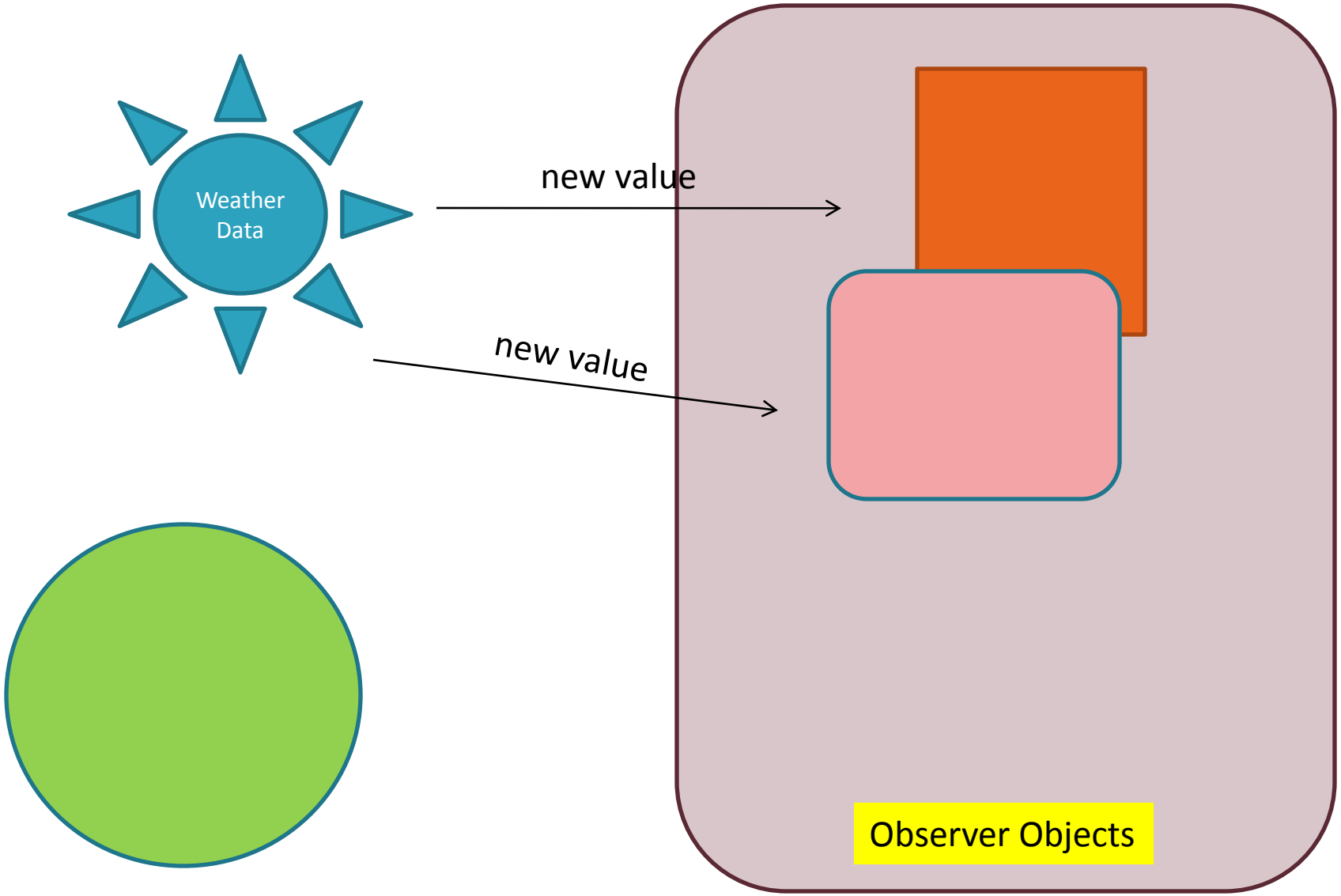


Observer Pattern





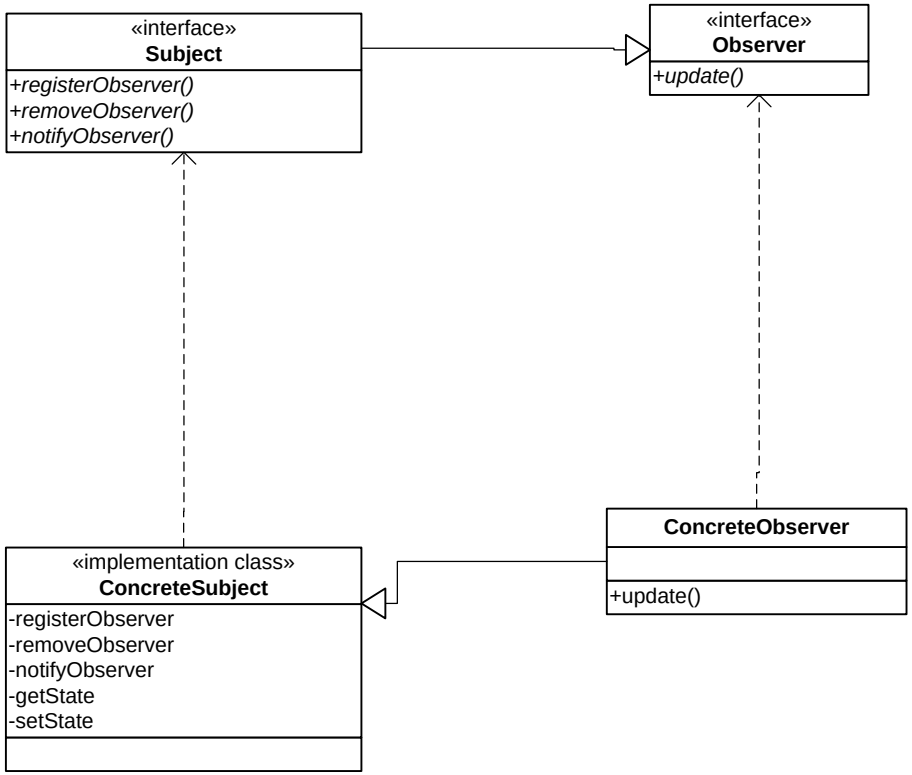
Observer Pattern



Observer Pattern

- one-to-many dependency between objects
- when object changes state →
- all dependencies are notified and update automatically.

Class Diagram



Advantages

- Subjects (data) only implements a certain interface (Observer interface)
- new observers can be add at any time
- never have to modify subject (data) when new observer is added
- reuse subject(data) and observer
- change to subject will not affect on observer and vice versa

Interfaces

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

public interface Observer {
    public void update(int data);
}

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

New WeatherData

```
public class WeatherData implements Subject {
    private ArrayList observers;
    private int temperature;

    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);
        }
    }
```

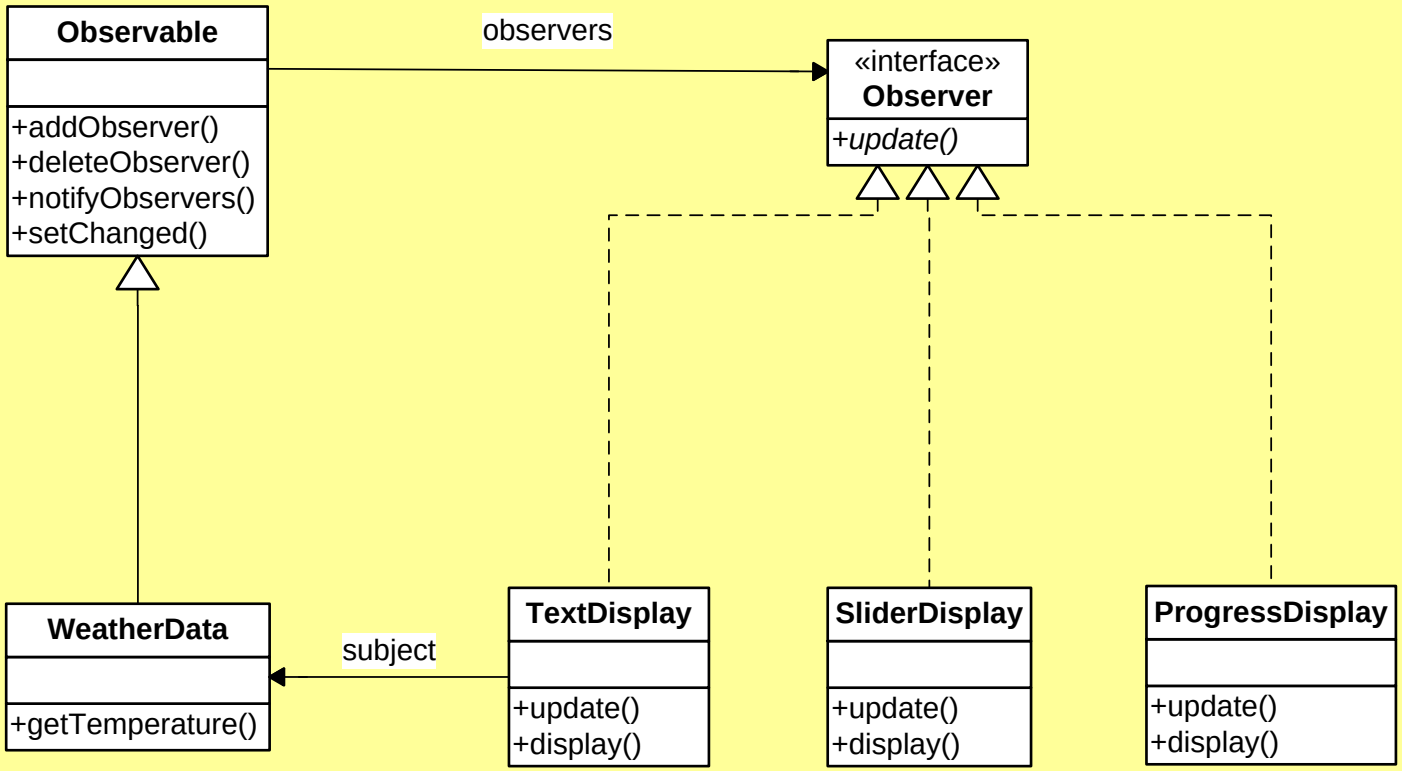
new WeatherData

```
    public void notifyObservers() {
        for (int i = 0; i < observers.size(); i++) {
            Observer observer = (Observer)observers.get(i);
            observer.update(temperature);
        }
    }

    public void measurementChanged() {
        temperature = getTemperature();
        notifyObservers();
    }

    public void setMeasurements(int temperature) {
        this.temperature = temperature;
        measurementChanged();
    }

    public int getTemperature() {
        return (int)(Math.random()*100);
    }
}
```

New TextDisplay

```
public class TextDisplay extends JPanel
        implements Observer, DisplayElement {

    int temperature;
    WeatherData weather;
    JLabel label;

    public TextDisplay(WeatherData weather) {
        this.weather = weather;
        weather.registerObserver(this);
        label = new JLabel(" ");
        add(label);
    }

    public void update(int data) {
        this.temperature = data;
        display();
    }

    public void display() {
        label.setText(""+temperature);
        validate();
    }
}
```

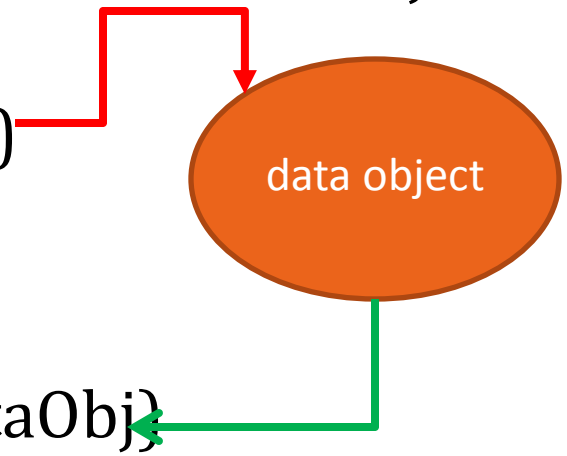
Java Build-in Observer Pattern



How it work? (1)

- Implement the Observer interface (java.util.Observer)
  - addObserver()
  - deleteObserver()

- To send notification
  - call setChanged() to change the state of the object
  - call notifyObservers() or notifyObservers(Object dataObj)



- Observers receive notifications
  - update(Observable o, Object dataObj)

How it work? (2)

```
setChanged() {
    changed = true;
}

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

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

WeatherData

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

public class WeatherData extends Observable {
    private int temperature;

    public void measurementChanged() {
        setChanged();
        notifyObservers();
    }

    public void setMeasurements(int temperature) {
        this.temperature = temperature;
        measurementChanged();
    }

    public int getTemperature() {
        return (int)(Math.random()*100);
    }
}
```

TextDisplay (1)

```
public class TextDisplay extends JPanel
        implements Observer, DisplayElement {

    Observable observable;
    private int temperature;
    JLabel label;

    public TextDisplay(Observable ob) {
        label = new JLabel("  ");
        add(label);
        this.observable = ob;
        observable.addObserver(this);
    }
```

TextDisplay (2)

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

  public void display() {
    label.setText(""+temperature);
    validate();
  }

}
```