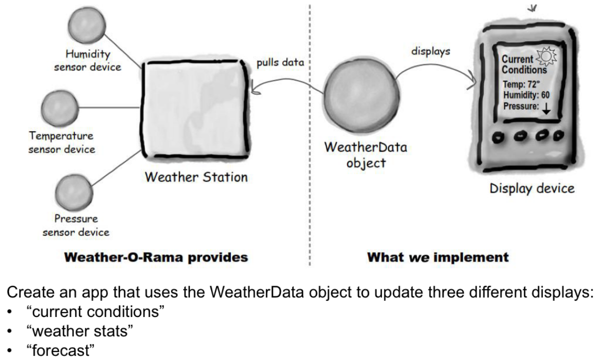
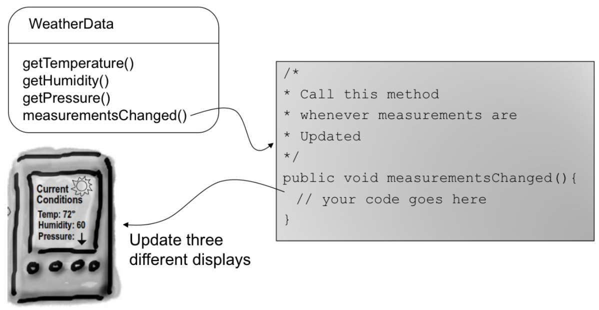
A Weather Monitoring Application



Create an app that uses the WeatherData object to update three different displays:

- "current conditions"
- "weather stats"
- "forecast"



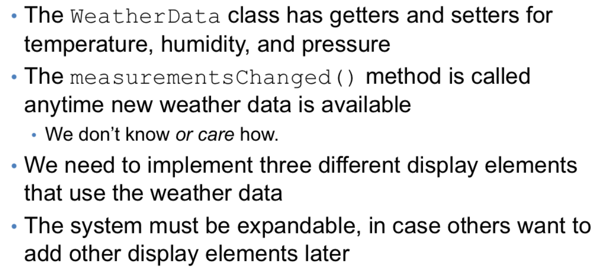
WeatherData

getTemperature()
getHumidity()
getPressure()
measurementsChanged()

```
/*
* Call this method
* whenever measurements are
* Updated
*/
public void measurementsChanged(){
  // your code goes here
}
```

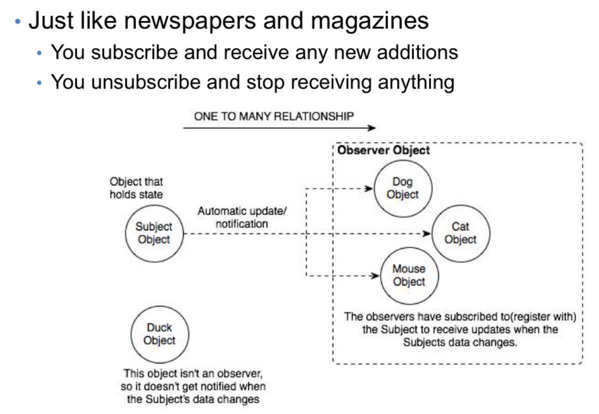
Update three different displays

Problem Specification



- The `WeatherData` class has getters and setters for temperature, humidity, and pressure
- The `measurementsChanged()` method is called anytime new weather data is available
  - We don't know *or care* how.
- We need to implement three different display elements that use the weather data
- The system must be expandable, in case others want to add other display elements later

Publish / Subscribe



- Just like newspapers and magazines
  - You subscribe and receive any new additions
  - You unsubscribe and stop receiving anything

ONE TO MANY RELATIONSHIP

Observer Object

Object that holds state

Subject Object

Automatic update/ notification

Dog Object

Cat Object

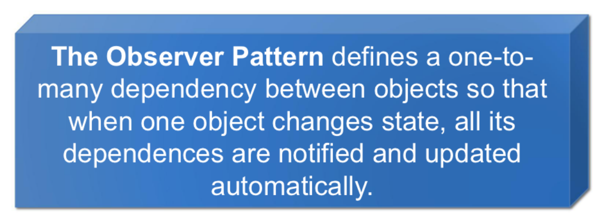
Mouse Object

The observers have subscribed to(register with) the Subject to receive updates when the Subjects data changes.

Duck Object

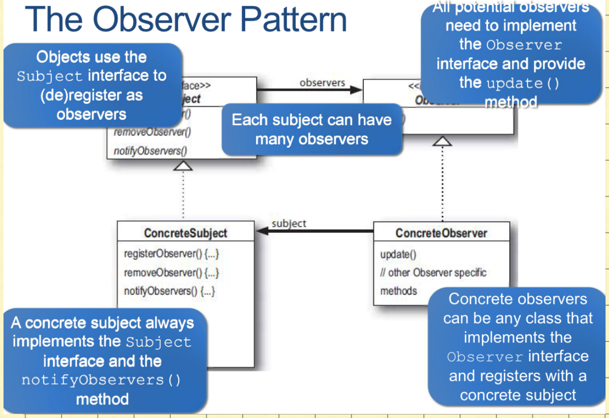
This object isn't an observer, so it doesn't get notified when the Subject's data changes

The Observer Pattern



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

观察者模式定义了一个一对多的依赖.



The Observer Pattern

Objects use the Subject interface to (de)register as observers

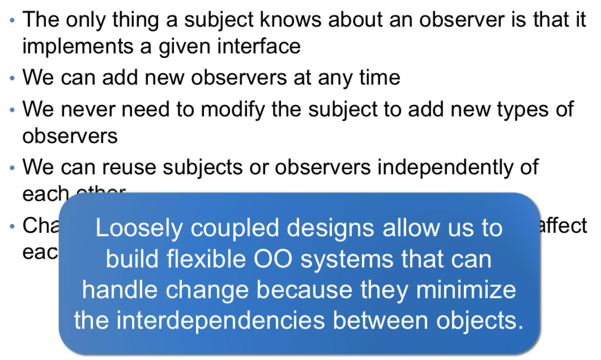
Each subject can have many observers

All potential observers need to implement the Observer interface and provide the update() method

A concrete subject always implements the Subject interface and the notifyObservers() method

Concrete observers can be any class that implements the Observer interface and registers with a concrete subject

The Power of Loose Coupling

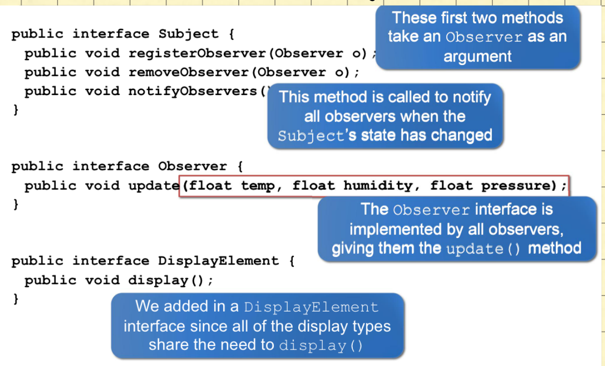


- The only thing a subject knows about an observer is that it implements a given interface
- We can add new observers at any time
- We never need to modify the subject to add new types of observers
- We can reuse subjects or observers independently of each other
- Cha affect eac

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

松耦合的设计能建立有弹性的OO系统，能应对变化，对象之间的依赖性降到了最低

Weather Data Interfaces



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

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

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

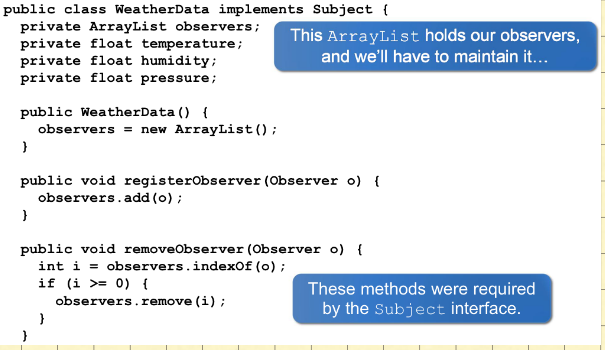
These first two methods take an Observer as an argument

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

The Observer interface is implemented by all observers, giving them the update() method

We added in a DisplayElement interface since all of the display types share the need to display()

Implementing the Subject Interface

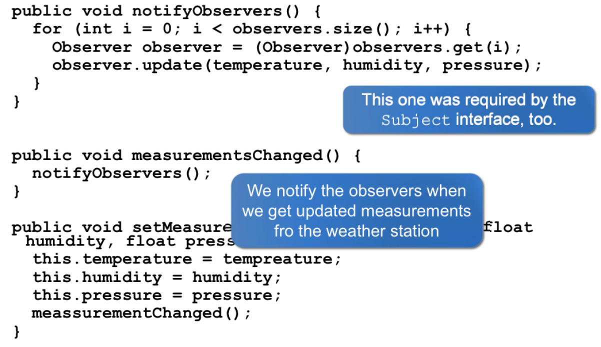


```
public class WeatherData implements Subject {
  private ArrayList observers;
  private float temperature;
  private float humidity;
  private float pressure;

  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);
    }
  }
```

This ArrayList holds our observers, and we'll have to maintain it…

These methods were required by the Subject interface.

Notify Methods

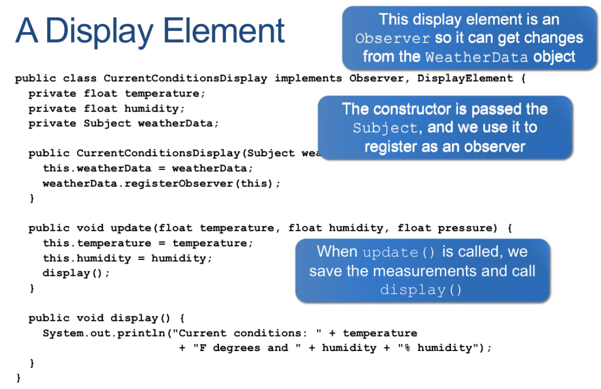


```
public void notifyObservers() {
  for (int i = 0; i < observers.size(); i++) {
    Observer observer = (Observer)observers.get(i);
    observer.update(temperature, humidity, pressure);
  }
}

public void measurementsChanged() {
  notifyObservers();
}

public void setMeasure                      float
  humidity, float press
  this.temperature = tempreature;
  this.humidity = humidity;
  this.pressure = pressure;
  meassurementChanged();
}
```

This one was required by the Subject interface, too.

We notify the observers when we get updated measurements fro the weather station

A Display Element



A Display Element

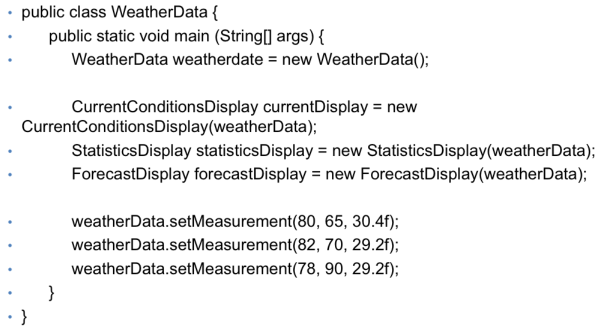
This display element is an Observer so it can get changes from the WeatherData object

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

  public CurrentConditionsDisplay(Subject wea
    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 Subject, and we use it to register as an observer

When update() is called, we save the measurements and call display()

Client Test



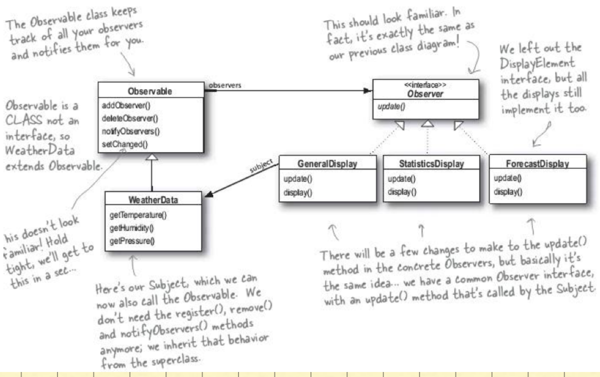
```
• public class WeatherData {
•       public static void main (String[] args) {
•            WeatherData weatherdate = new WeatherData();

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

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

The Observer Pattern in Java

- Java provides the `Observer` interface and the `Observable` class in the package `java.util`
  - Similar to `Subject` and `Observer`
- Enable both push and pull style interactions (as opposed to only push as before)

Java中有observer 接口和observable类
(Observer)　(Subject)
能支持拉/推两种方式.

Another Design...



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.

his 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.

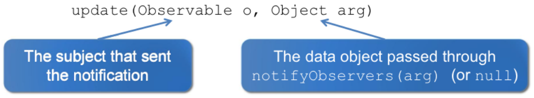
The Java Observer Pattern

- For an object to become an observer
  - Just implement the `Observer` interface (as before)
- For the observable to send notifications
  - Become observable by *extending* the `java.util.Observable` superclass
  - Call the `setChanged()` method to signify that the state of the object has changed
  - Call one of two notification methods:
    - `notifyObservers()`
    - `notifyObservers(Object arg)`

Notification Revisited

- For an observer to receive notifications
  - Provides a definition of the update method:

`update(Observable o, Object arg)`


The subject that sent the notification

The data object passed through `notifyObservers(arg)` (or `null`)

- To *push* data
  - Pass the data as a data object through the `notifyObservers(arg)` method
- To have the Observer *pull* data
  - The Observer must use the `Observable` object passed to it using the object's getters and setters

The Dark Side of Java Observables

- Observable is a class, not an interface
  - You have to *subclass* it, so you can't add the Observable behavior onto a class that already extends something else
    - Limits reuse potential
  - Because there's no Observable interface, you cannot create your own implementations of Observables
- Observable protects crucial methods
  - E.g., setChanged() can only be called by subclasses
    - Limits flexibilty; you cannot favor composition over inheritance