DESIGN PATTERNS

Factory

- This is a creational pattern
- There is more to making objects than just calling new
- Often, we get into this situation when we have many related classes

```
Duck duck;
if (picnic)
  duck = new MallardDuck();
else if (hunting)
  duck = new DecoyDuck();
else if (bathtub)
  duck = new RubberDuck();
```

Factory

We might want to have some code for a pizza store

```
Pizza orderPizza()
 {
    Pizza pizza = new Pizza();
    pizza.prepare();
    pizza.bake()
  ….etc….
}
```

Factory

But if there could be many kinds of pizza, this is a problem

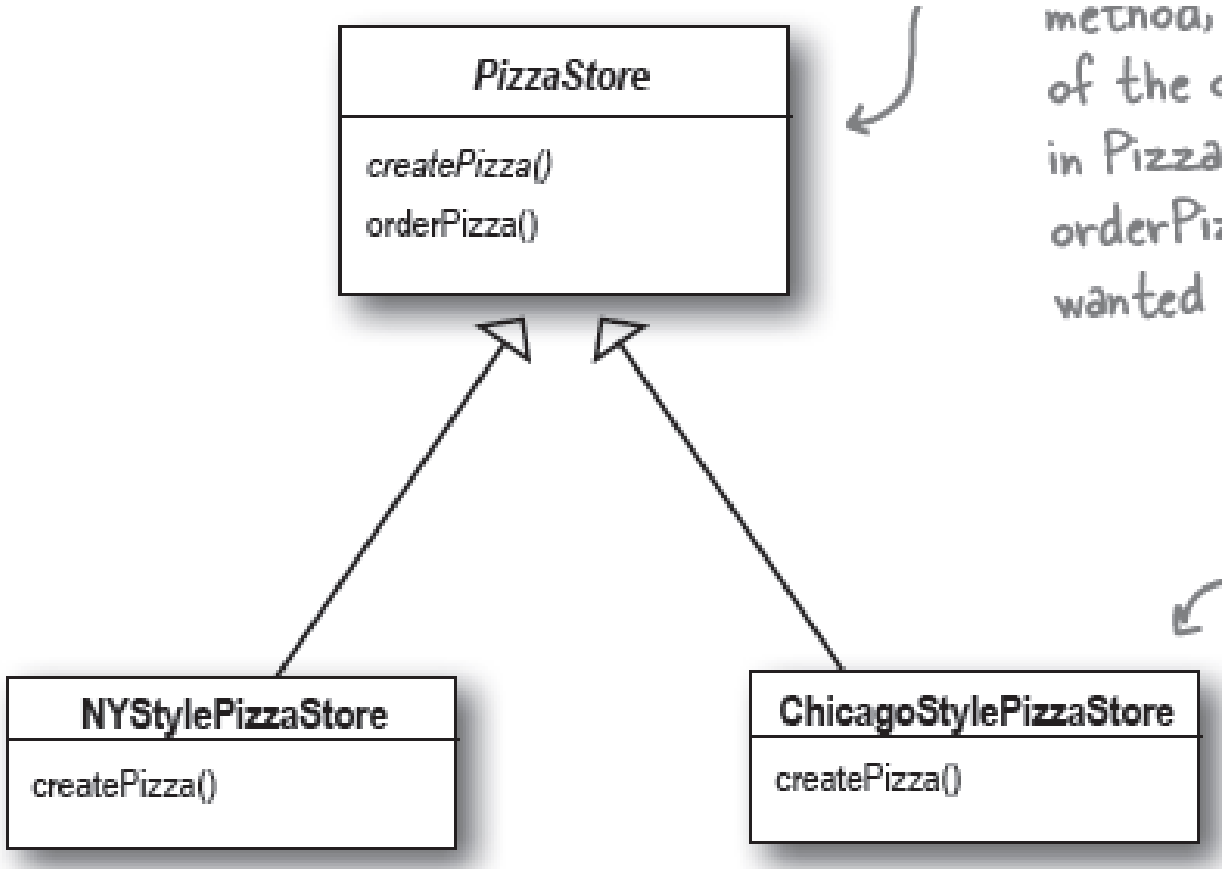
```
Pizza orderPizza(String type)
 {
    Pizza pizza;
    if (type.equals("cheese"))
       pizza = new CheesePizza();
    if (type.equals("greek"))
       pizza = new GreekPizza();
    if (type.equals("pepperoni");
       pizza = new PepperoniPizza();
    pizza.prepare();
    pizza.bake()
  ….etc….
}
```

Factory

- We need a way to separate this code from the rest of the code
- We could just have a method in the class that contains orderPizza(), but that would be a problem if we need to create pizzas in several classes
- So instead, make a class that is responsible for creating pizzas.
- This is called a **Factory** class
- In Week11 code, look at headfirst.factory.pizzas.SimplePizzaFactory.java

Pizza Framework

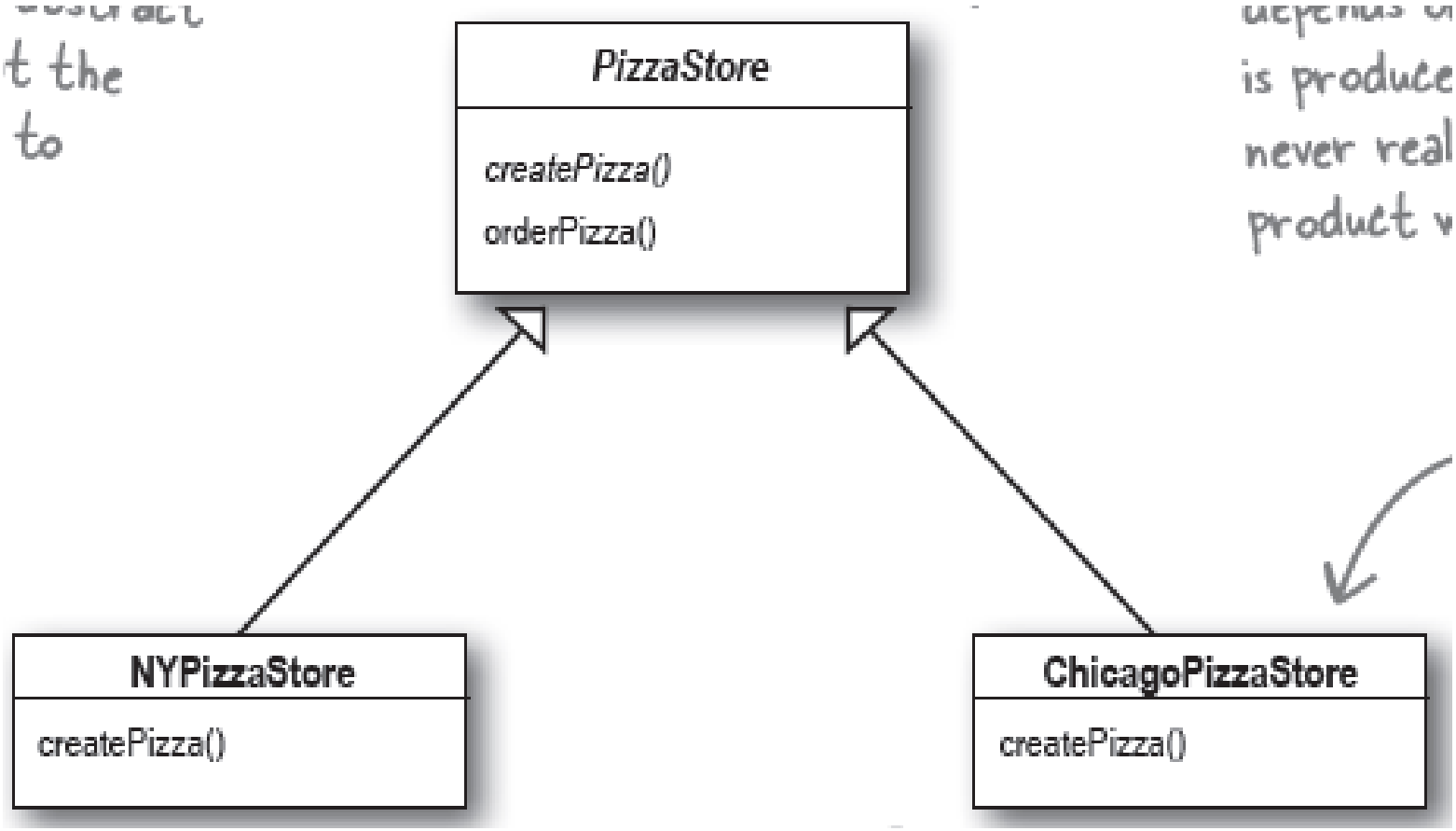
- We can extend this idea, so we can have different kinds of stores



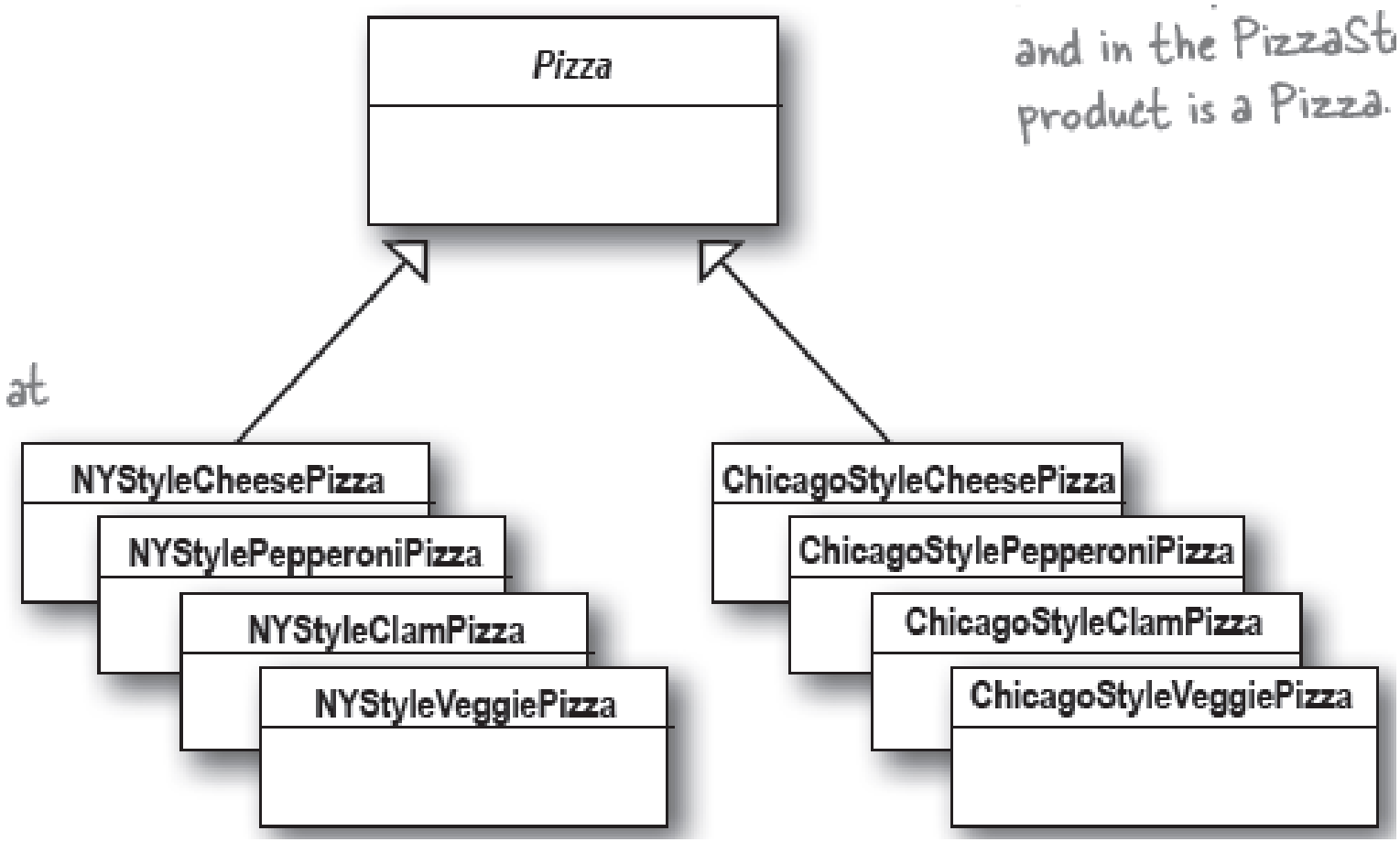
Pizza Framework

- Creating a pizza will no longer be in a separate factory
- Each subclass of PizzaStore will have its own version of createPizza()
- The superclass, PizzaStore, is abstract, and createPizza() is an abstract method
- Look at headfirst.factory.pizzafm
- The createPizza() method is called an abstract method

The Creator classes

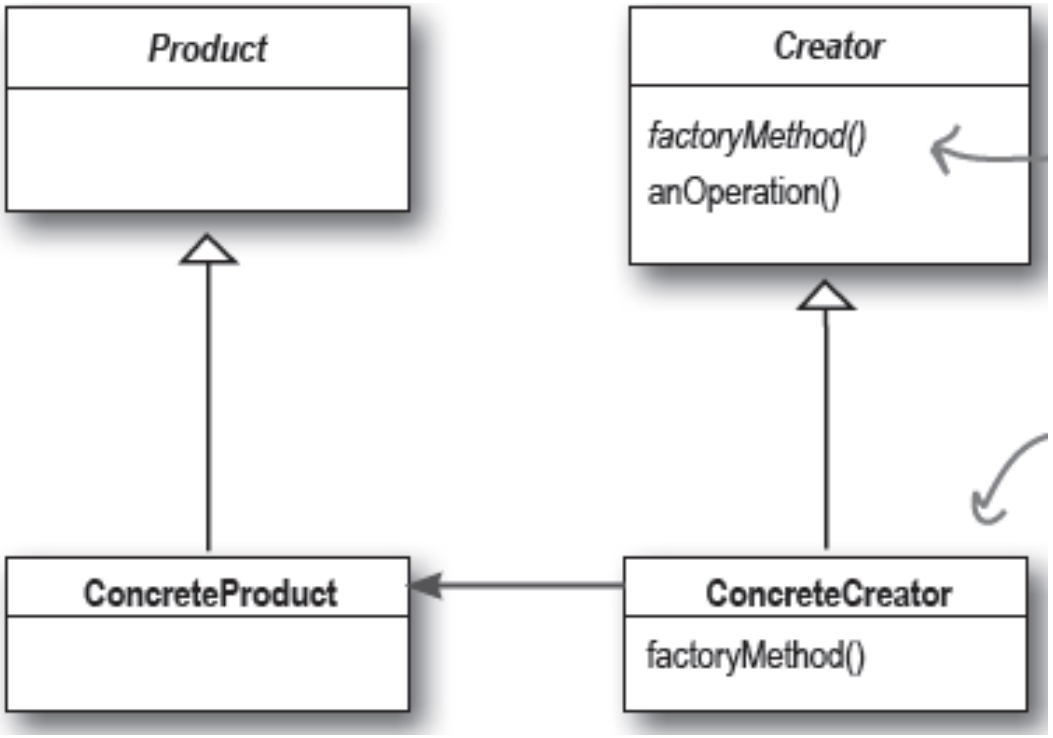


The Product classes

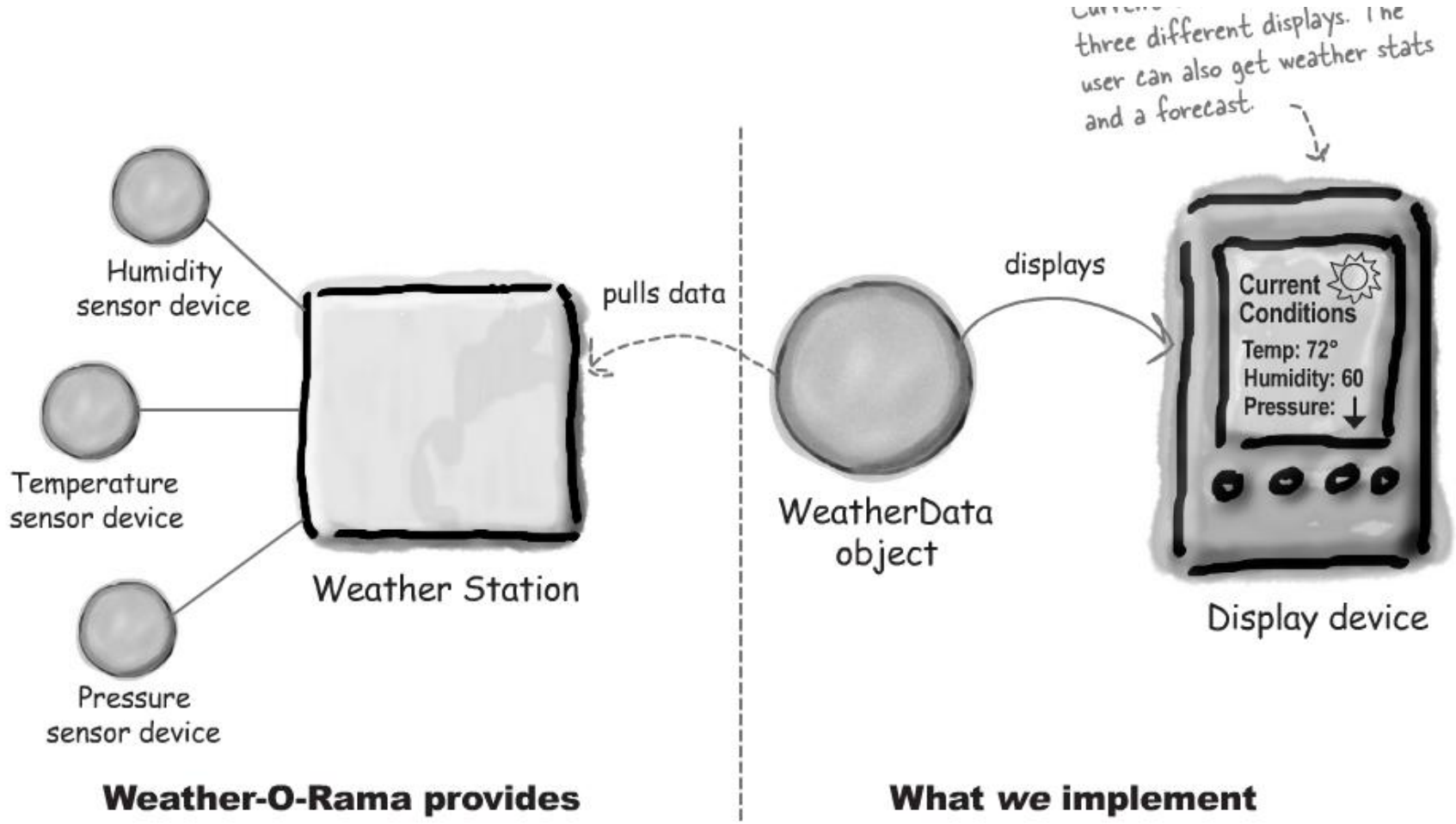


Factory method pattern

The Factory Method Pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.



Observer Pattern



Observer Pattern

The WeatherData class has a method called measurementsChanged() which is called whenever new weather data becomes available. How to implement it?

First try:

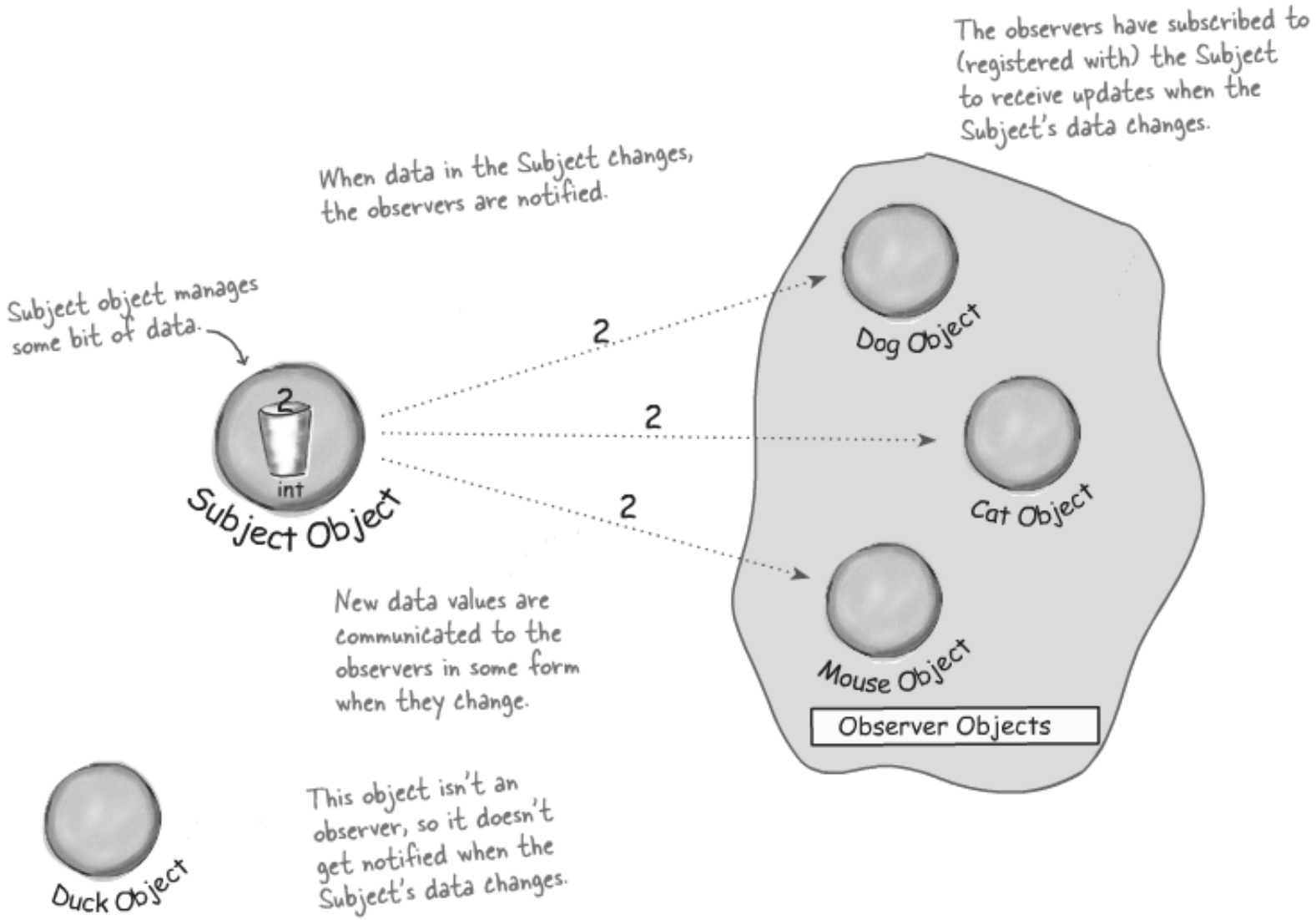
```
public class WeatherData {
    // instance variable declarations
    public void measurementsChanged() {
        float temp = getTemperature();
        float humidity = getHumidity();
        float pressure = getPressure();
        currentConditionsDisplay.update(temp,
 humidity, pressure);
        statisticsDisplay.update(temp, humidity,
 pressure);
        forecastDisplay.update(temp, humidity,
 pressure);
    }
```

Newspaper subscriptions

- A newspaper publisher goes into business and begins
- publishing newspapers.
- You subscribe to a particular publisher, and every time there is 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.

Observer pattern



Observer pattern

Context

- An object, called the subject, is source of events
- One or more observer objects want to be notified when such an event occurs.

Solution

- Define an observer interface type. All concrete observers implement it.
- The subject maintains a collection of observers.
- The subject supplies methods for attaching and detaching observers.
- Whenever an event occurs, the subject notifies all observers.

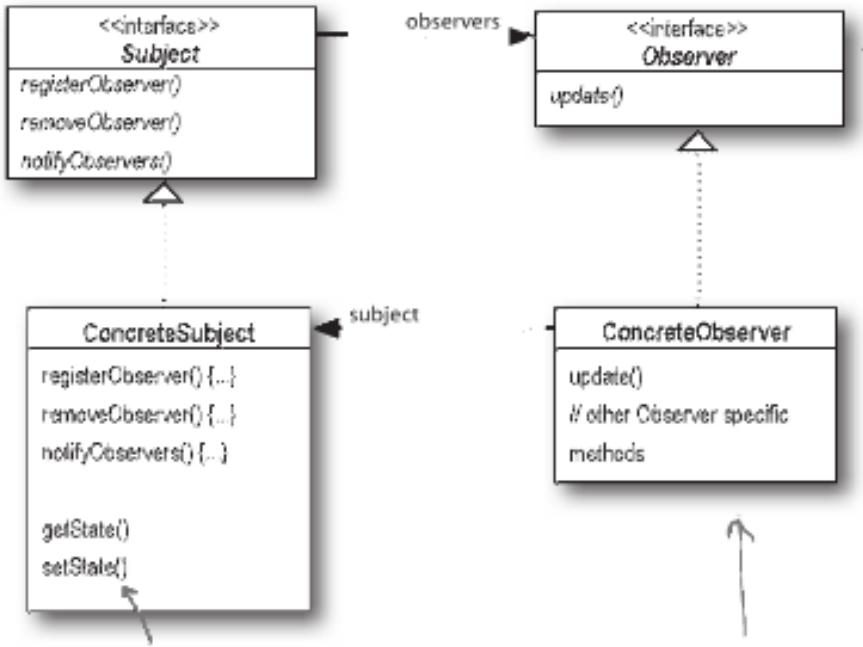
Observer Pattern

the class diagram

Here's the Subject interface. Objects use this interface to register as observers and also to remove themselves from 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. In addition to the register and remove methods, the concrete subject implements a notifyObservers() method that is used to update all the current observers whenever state changes.

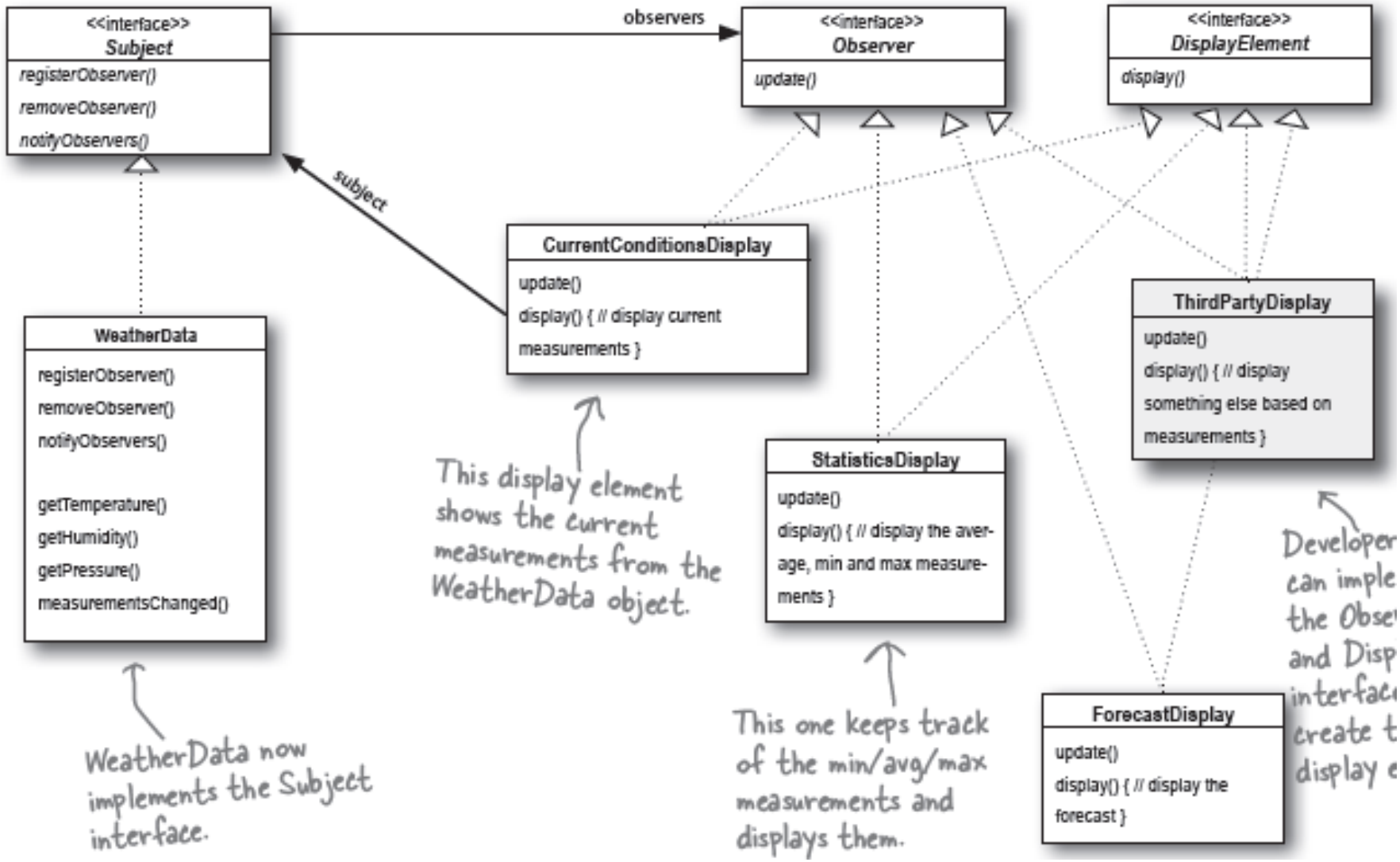
The concrete subject may also have methods for setting and getting its state (more about this later).

Concrete observers can be any class that implements the Observer interface. Each observer registers with a concrete subject to receive updates.

Observer pattern - buttons

Name in Design Pattern	Actual Name (Swing buttons)
`Subject`	`JButton`
`Observer`	`ActionListener`
`ConcreteObserver`	the class that implements the `ActionListener` interface type
`attach()`	`addActionListener()`
`notify()`	`actionPerformed()`

HeadFirst Implementation of Observer



HeadFirst Implementation of Observer

```
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.

Observer pattern - WeatherData

- Java has a built-in Observer interface, and an Observable class that works with it.
- A class that is a model must extend Observable.
- When the state of the model changes
  - Call stateChanged() to signify that the state has changed.
  - Call either notifyObservers() or notifyObservers(Object arg)

Observer interface

Public interface Observer

Method Summary

Methods

Modifier and Type	Method and Description
void	update(Observable o, Object arg) This method is called whenever the observed object is changed.

For an object to receive notifications, it must implement the Observer interface.

- This means it must implement the following method

```
update(Observable o, Object arg);
```

- O is the subject that sent the notification
- Arg is the data object that was passed through

Observer pattern - WeatherData

For the Observable to send notifications

1. It must extend Observable
2. It must call stateChanged() to signify that the state has changed
3. Then it must call either notifyObservers() or notifyObservers(arg) – arg is an arbitrary data object

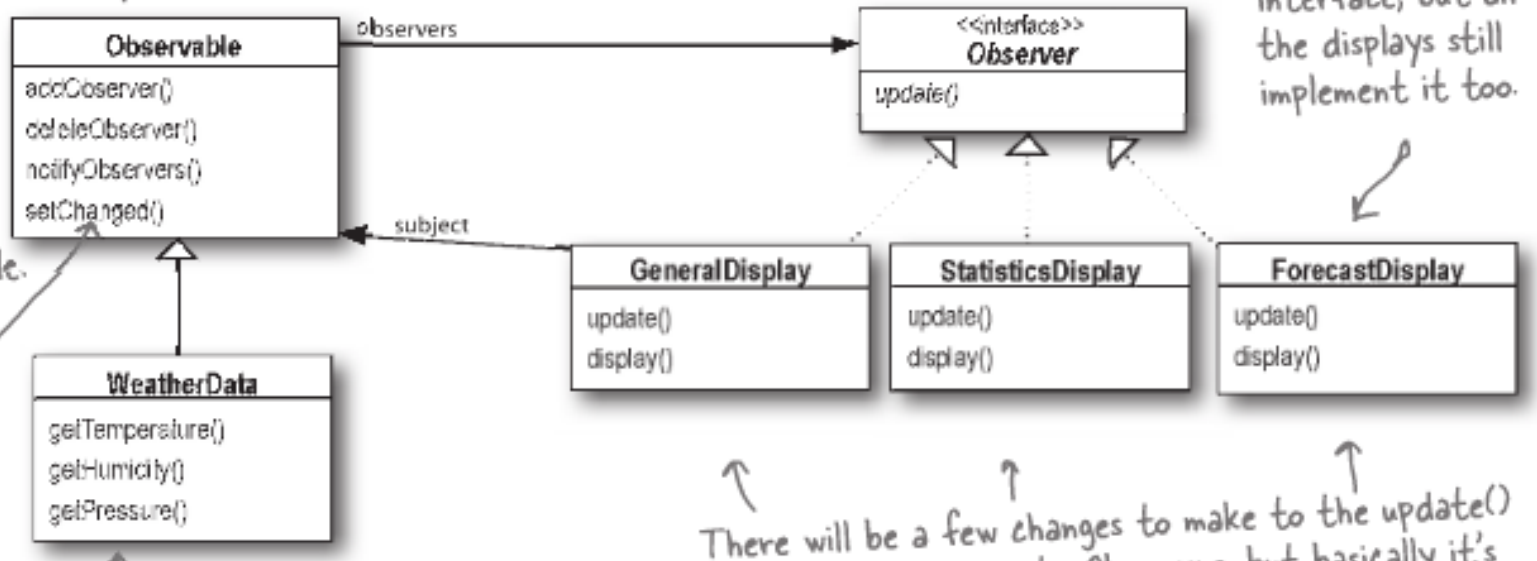
Observer pattern - WeatherData

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.



This 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.

Model View Controller

- This is a standard architecture used in GUI programming
- It makes use of the Observer pattern
- Some programs have multiple editable views
  - HTML Editor
  - WYSIWYG view
  - structure view
  - source view
- Editing one view updates the other
- Updates seem instantaneous

Model View Controller

- Model: data structure, no visual representation
- Views: visual representations
- Controllers: user interaction
- Controllers update model
- Model tells views that data has changed
- Views redraw themselves

Model View Controller

When a user types text into one of the windows

1. The controller tells the model to insert the text that the user typed
2. The model notifies all views of the change
3. All views repaint themselves, asking the model for the current text to do this.

Model View Controller

Minimal coupling

- Views do not know controllers, they only ask model for data
- Model does not know controllers, only knows which views to notify, but does not know how views actually work
- Controllers know how to update data of the model
- Adding more views, more controllers is simple

Having only one model avoids redundant storage of data and difficulties with updates and consistency.

Observer pattern - Temperature

- This example uses AWT classes to create a GUI
- There are several editable GUI interfaces
- The model is a class that represents one temperature
- It extends Observable

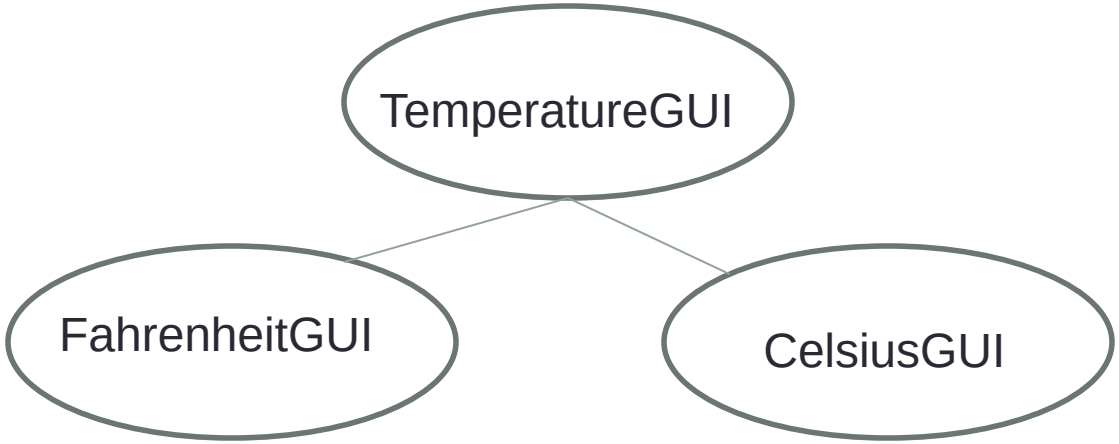
Observer pattern - Temperature

```
public class TemperatureModel extends java.util.Observable
{
 private double temperatureF = 32.0;
 public double getFahrenheit(){return temperatureF;}

 public double getCelsius(){return (temperatureF - 32.0) *
  5.0 / 9.0;}

 public void setFahrenheit(double tempF)
    {
     temperatureF = tempF;
     setChanged();
     notifyObservers();
   }
```

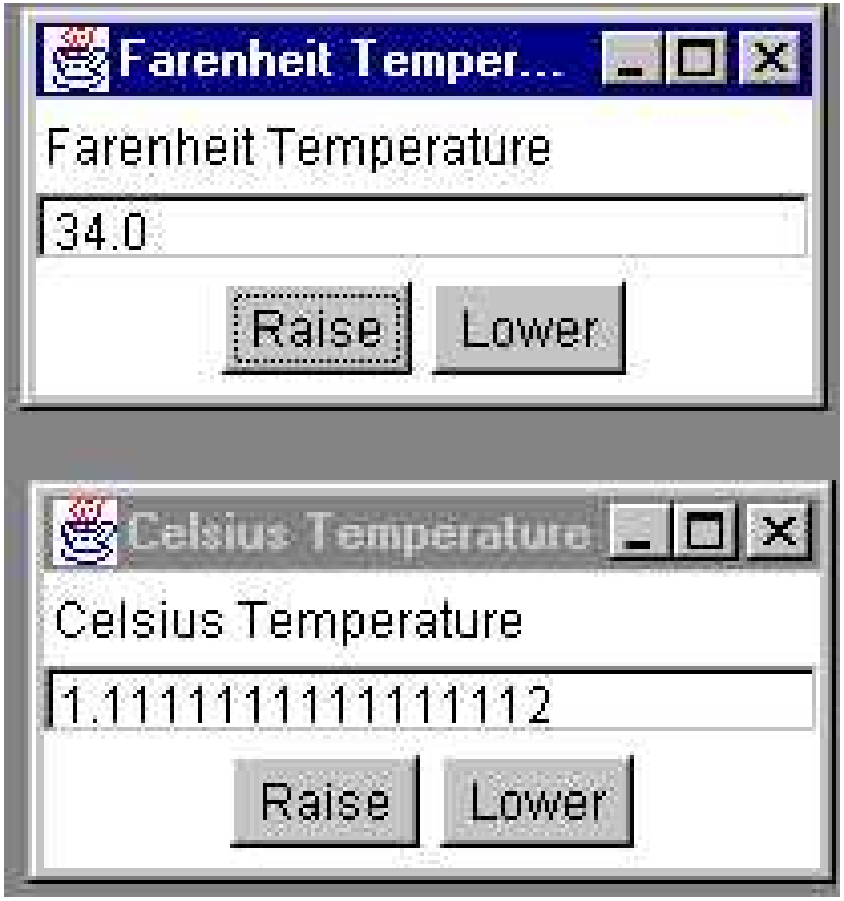
When state changes, call these two methods

Observer pattern - Temperature

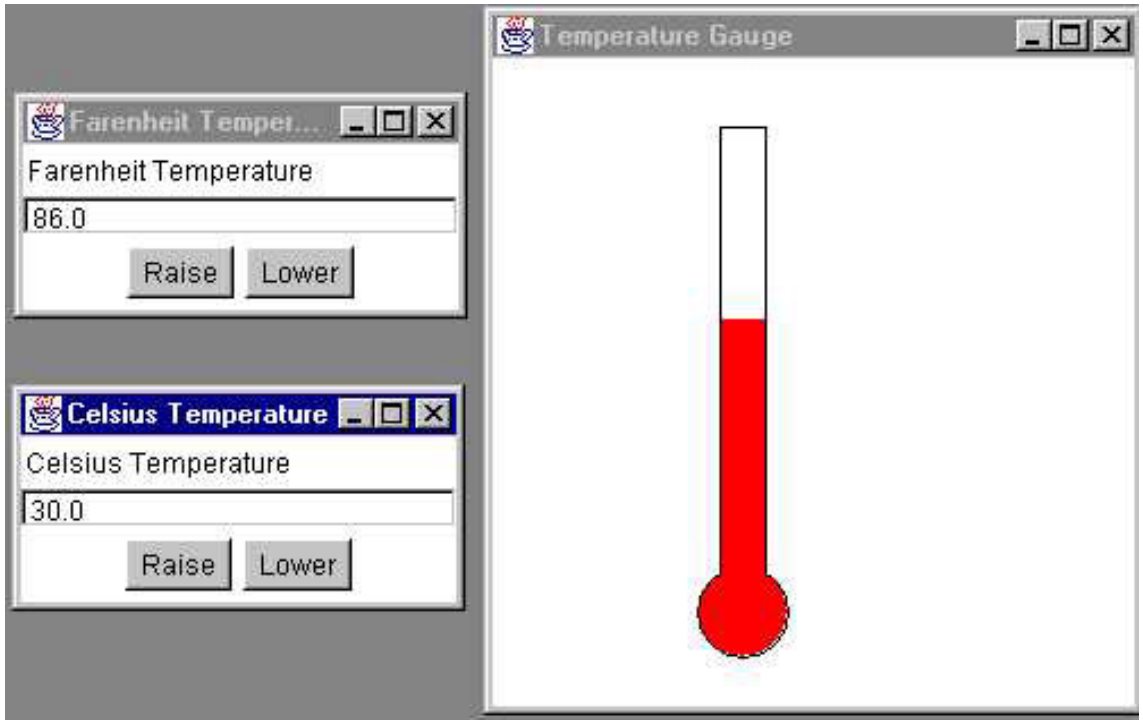
- TemperatureGUI is a superclass that contains code common to several GUI classes
- This design works because the FahrenheitGUI and CelsiusGUI look the same, so code can be reused



Observer pattern - Temperature



Observer pattern - Temperature



Disadvantages of Java Observer/Observable

- Observable is a class
- Crucial methods in Observable are protected rather than public
- You often end up creating your own Observer/Observable classes rather than using the builtin Java ones

Singleton

- This is useful when we want to ensure that there is exactly one object, and no more, of a given class in a system
- Why? Often this type of object represents a resource – a connection to the database, a socket that the system listens to, a logger object.
- example – a random number generator

Singleton

Context

- All clients need to access a single shared instance of a class.
- You want to ensure that no additional instances can be created accidentally.

Solution

- Define a class with a private constructor.
- The class constructs a single instance of itself.
- Supply a static method that returns a reference to the single instance.

Singleton

A Singleton candidate must satisfy three requirements:

- controls concurrent access to a shared resource.
- access to the resource will be requested from multiple, disparate parts of the system.
- there can be only one object.

Singleton

```
public class SingletonFrame extends JFrame {
    private static SingletonFrame myInstance;

    // the constructor
    private SingletonFrame() {
        this.setSize(400, 100);

        this.setTitle("Singleton Frame. Timestamp:" +
            System.currentTimeMillis());

        this.setDefaultCloseOperation(JFrame.HIDE_ON_CLOSE);
    }

    public static SingletonFrame getInstance() {
        if (myInstance == null)
            myInstance = new SingletonFrame();
        return myInstance;
    }
}
```

FileLogger

```
public class FileLogger extends SimpleLogger {
   private static FileLogger inst=null;
   private FileWriter outFile;

    private FileLogger()
    {
    System.err.println("in FileLogger constructor");
    }

public static FileLogger getInstance() {
if (inst == null)
inst = new FileLogger();

return inst;
   }
```

Usage

```
public class LoggerFactory {
   static public SimpleLogger getLogger(String
 type)
   {
   if (type.equals("File"))
   return FileLogger.getInstance();
   else
   return null;
   }
}
```