Mobile Computing
Professor Pushpendra Singh
Indraprastha Institute of Information Technology Delhi
Lecture 22
Observer Pattern and MVC

Hello and welcome to the class, let us first revise what we have learnt so far. We have learned about basic android programming, we learned about fundamentals of android applications, how to use android studio to crate android applications and then we learned one of the basic component of android applications that is Activities. We learned about Activity life cycle and we learned about that what happens when Activity life cycle when we interact with the user device such as by pressing back button or by pressing home button. Then recently we also learned how to create more than one Activity in a single android application and then how to use intents to communicate between different Activities.

Today we are going to learn something different but very useful, it i called Observer Pattern. You may have heard about design patterns earlier. Design patterns are recommended practice of programming which has evolved after lot many years, discussions, arguments between the developer communities. The patterns have been proposed as the necessary as well as recommended practice of program. So, today we are going to learn Observer Pattern which is very closely related to every see that we use in our android applications. So, let us start with what is Observer Pattern and how does it relate with MVC or even with the public subscriber.

(Refer Slide Time: 1:51)

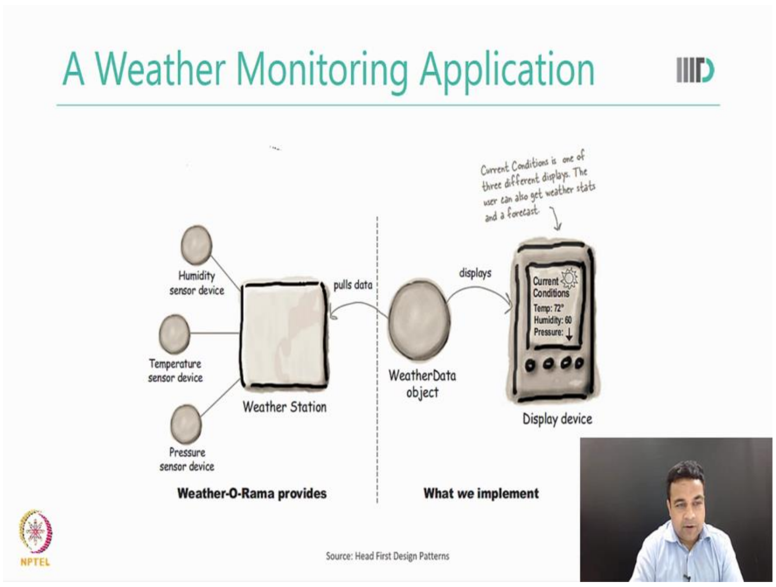
Observer Pattern

- Do not miss out when something interesting happens

Source: Head First Design Patterns

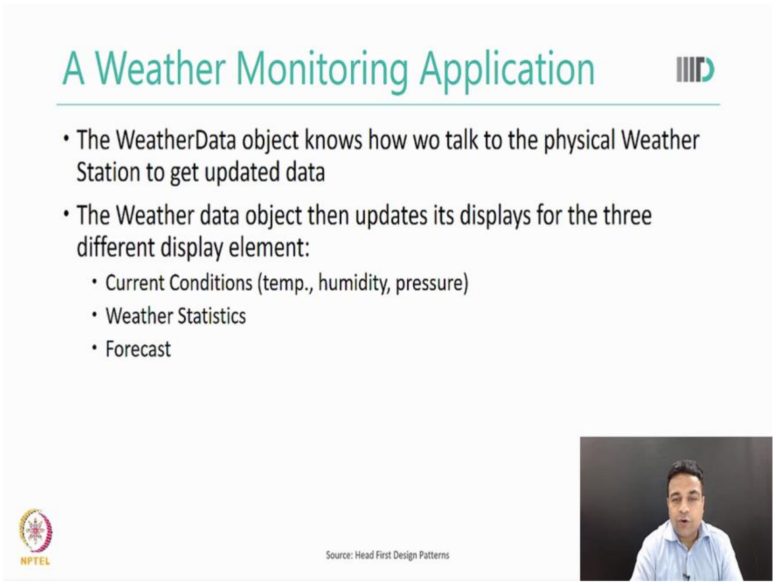
So the Observer Pattern is based on a simple premise that one should not miss out when something interesting happens. Now what that interesting thing could be? For example, if you are subscribing to let us say a news item or a newspaper, you would not like to miss out when new news is published. Or if you are subscribing to a weather station you will not like to miss out if a weather update has been done, the Observer Pattern in Java enables that. Let us understand with a simple example.

(Refer Slide Time: 2:31)



Let us say that we have a weather monitoring application. This application has 3 components; the 1 component is the actual weather station where the different values related to weather are updated. For example, there may be a humidity sensor device which updates the humidity value. There may be temperature sensor device which updates the temperature value and there may be a pressure sensor device which updates the pressure. Now, all these weather station data is pulled by a weather data object. And then, this data object tries to display this data on a number of devices. Some of these devices could be mobile phones and some of these devices could be web browsers, some of these devices could even be smart watches.

(Refer Slide Time: 3:51)



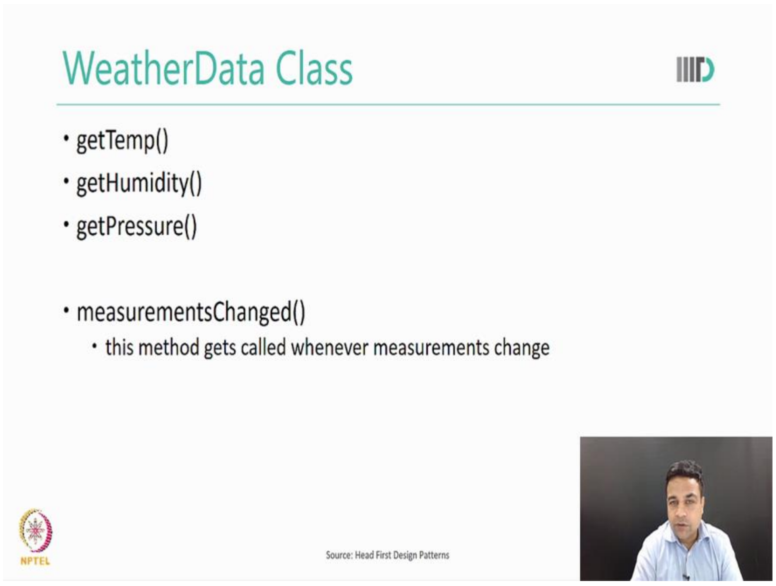
A Weather Monitoring Application

- The WeatherData object knows how wo talk to the physical Weather Station to get updated data
- The Weather data object then updates its displays for the three different display element:
  - Current Conditions (temp., humidity, pressure)
  - Weather Statistics
  - Forecast

Source: Head First Design Patterns

Now you can see that this is very similar to what we see in the MVC. Let us go further in our example, suppose you want to implement it, so how do we implement such that whenever the weather station updates any data, the view is updated on all the display devices that is our task. So, our weather data object knows how to talk to the physical weather station to get updated data. And our weather data object then updates its displays for the 3 different display elements that we are currently concentrating. So, in this example we will be considering only 3 display elements. One is a one is a simple display element that presents current conditions that is current values, another provides statistics by giving average, min and max of the value and the a one provides the weather forecast.

(Refer Slide Time: 4:44)



WeatherData Class

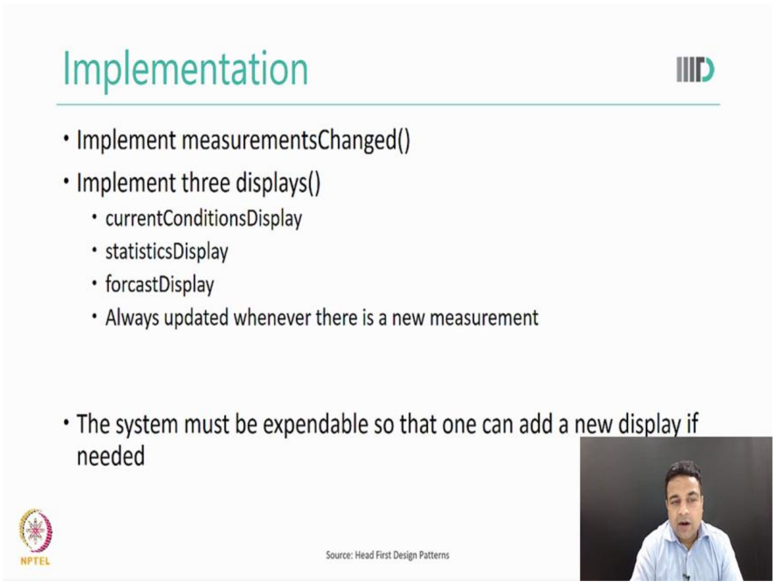
- getTemp()
- getHumidity()
- getPressure()

- measurementsChanged()
  - this method gets called whenever measurements change

Source: Head First Design Patterns

These displays could be running on three different types of devices or a single device or multiple types of devices having different combinations. Now, let us see that what we will have in our weather data class. So, our weather data class will be very simple it will try to get the actual sensor values and then it will have a method which will be called when any of these measurements is changed.

(Refer Slide Time: 4:59)



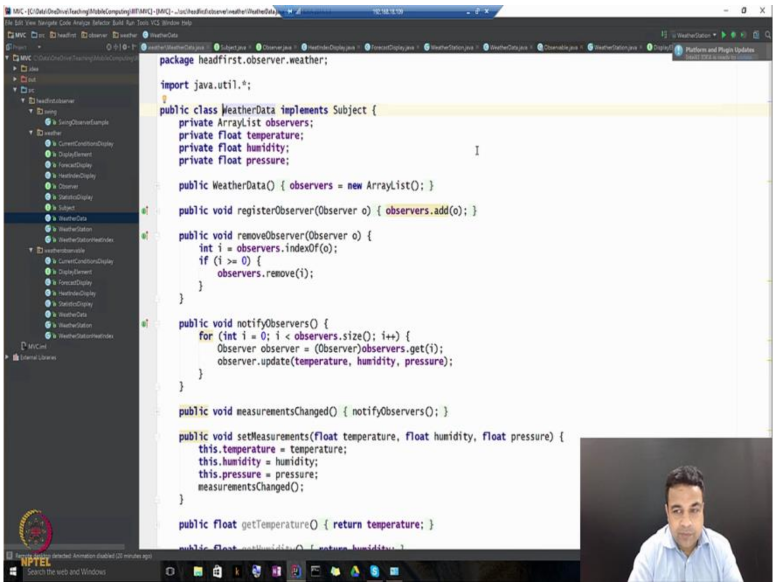
Implementation

- Implement measurementsChanged()
- Implement three displays()
  - currentConditionsDisplay
  - statisticsDisplay
  - forcastDisplay
  - Always updated whenever there is a new measurement
- The system must be expendable so that one can add a new display if needed

Source: Head First Design Patterns

Now, we will have to see that how do we implement this measurements changed and how do we implement it for the 3 displays which will be displaying the new measurements. Number 1 thing that we have to keep in mind is that the displays are always updated whenever there is a new measurement, and the system must be expendable so that one can add a new display if it is needed.

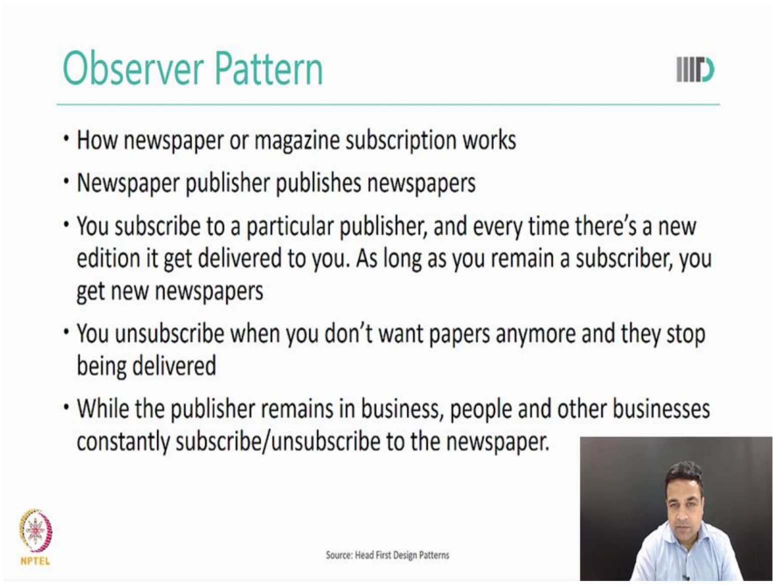
(Refer Slide Time: 5:28)



```java
package headfirst.observer.weather;

import java.util.*;

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(); }

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

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

So, here is a possible implementation of measurement changes, it first tries to get the temperature, humidity, the pressure readings and then it updates each of the display one by one. Now, let us see how the Observer Pattern fits here does or how the Observer Patterns does fits in real life. Let us take a very simple example that you encounter in your daily life.

(Refer Slide Time: 5:54)



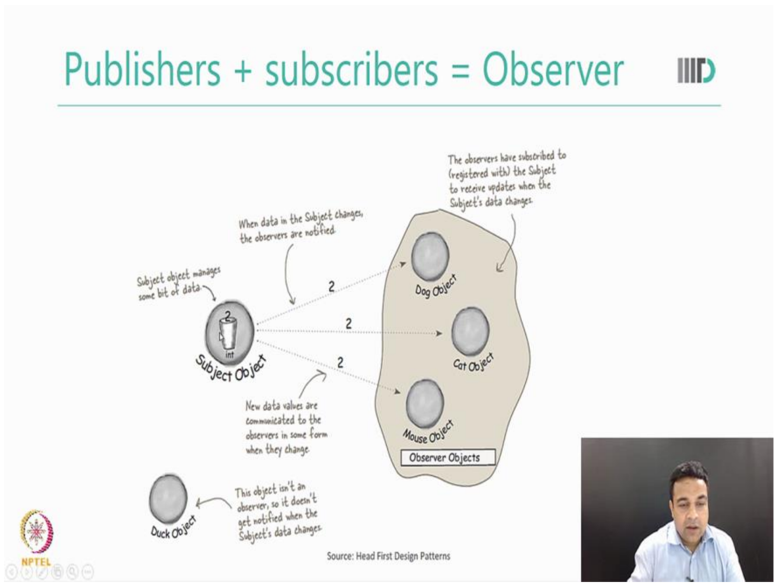
Observer Pattern

- How newspaper or magazine subscription works
- Newspaper publisher publishes newspapers
- You subscribe to a particular publisher, and every time there's a new edition it get 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 and other businesses constantly subscribe/unsubscribe to the newspaper.

Source: Head First Design Patterns

Many of you may be reading newspapers. Some of you may be reading one newspaper; some of you may be reading more than one newspaper. Now what do you do when you want to receive a newspaper? You simply call a newspaper person and you ask that whether that person should deliver a Times of India or a Hindustan Times or Amarjala newspaper to you. Your neighbor may be asking for a different newspaper and your another neighbor may be asking for more than one newspaper. So, which means that you may subscribe to any number of the newspaper and as long as you remain a subscriber you will get these newspapers. And one day you may decide that you may do not want to receive any newspaper or maybe you do not want to receive the Times of India. That day you unsubscribe from either one or all of the newspaper and then subsequently you stop receiving the newspaper updates.

(Refer Slide Time: 7:18)

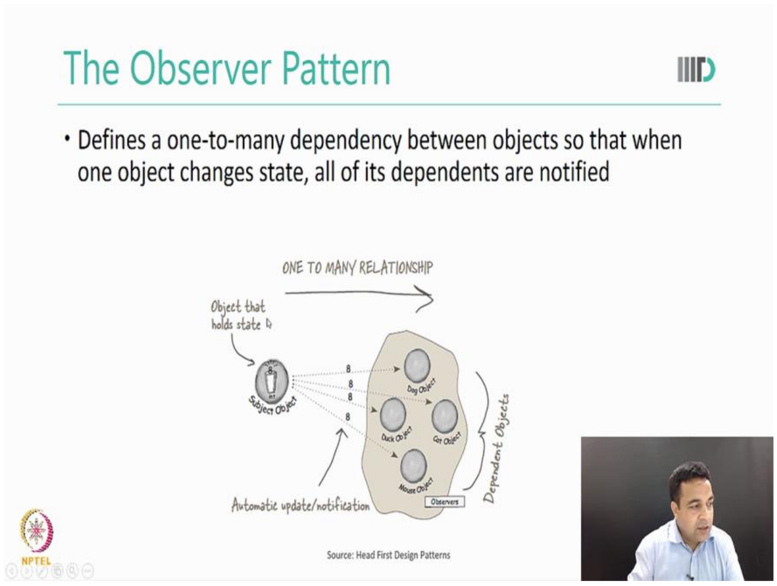


Publishers + subscribers = Observer

Subject object manages some bit of data.
2
int
Subject Object
When data in the Subject changes, the observers are notified.
2
2
2
New data values are communicated to the observers in some form when they change.
The observers have subscribed to (registered with) the Subject to receive updates when the Subject's data changes.
Dog Object
Cat Object
Mouse Object
Observer Objects
Duck Object
This object isn't an observer, so it doesn't get notified when the Subject's data changes.
Source: Head First Design Patterns
NPTEL

Now, just because you subscribe or unsubscribe from newspaper should not affect your neighbor’s subscription or subscription and similarly your neighbor’s subscription, etc should not affect you. Let us try to understand it from the simple way. Here you can see that this is your subject, which is of interest and these are different Observer Objects. These Observer Objects have subscribed themselves with the subject to receive updates when the subject data changes. So, whenever there is an interesting change in the data of the subject these values are notified to the Observer Objects. And if there is an object which is not an Observer, that object does not get the notification.

(Refer Slide Time: 7:54)

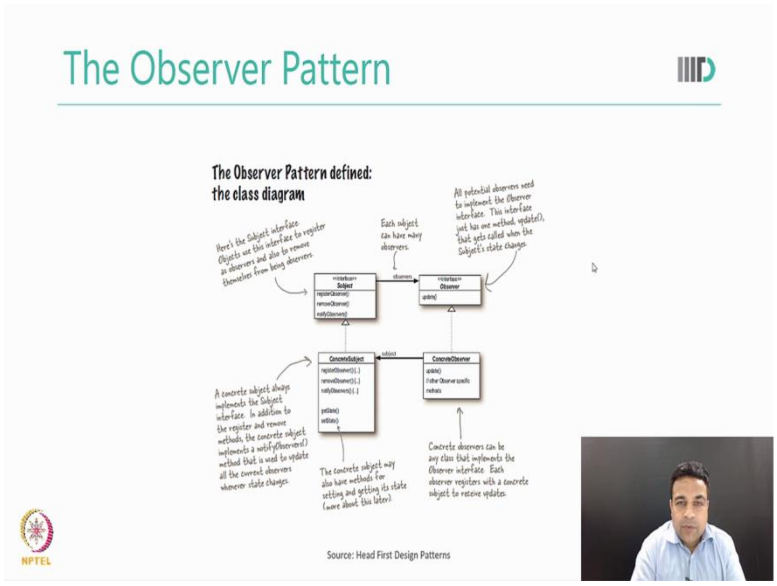


The Observer Pattern

- Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified

Now, here with one subject it could be multiple Observers. Similarly a single Observer may subscribe to different subjects.

(Refer Slide Time: 8:08)



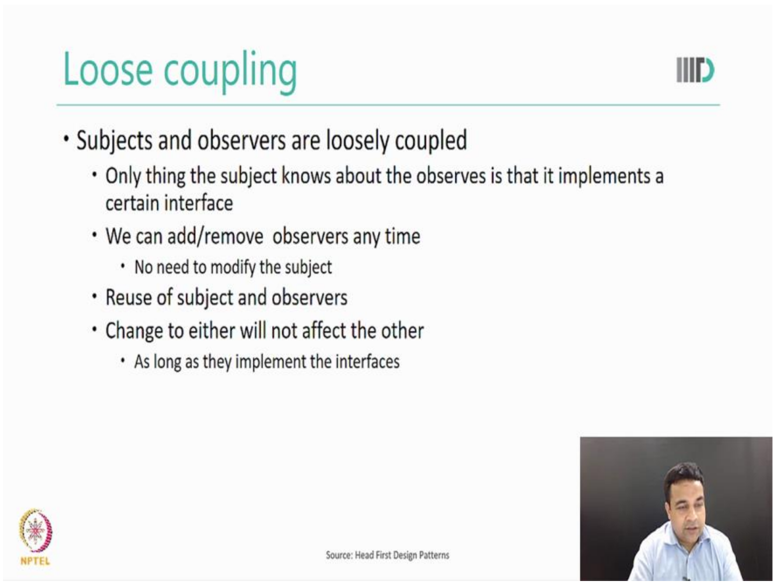
The Observer Pattern

The Observer Pattern defined: 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.
<<interface>> Subject
registerObserver()
removeObserver()
notifyObservers()
observers
<<interface>> Observer
update()
ConcreteSubject
registerObserver() {...}
removeObserver() {...}
notifyObservers() {...}
getState()
setState()
subject
ConcreteObserver
update()
// other Observer specific methods
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.
Source: Head First Design Patterns
NPTEL

Now, let us see how it all gets implemented in Java. We are currently assuming that Java is not providing us any facility to implement Observer Pattern and we have to write everything from the scratch. Later on we will also see examples of Java classes which implement Observer Patterns. So, what we need to do is we need to declare subject interface and the object that use this interface to register as Observers and also to remove themselves from being Observers. So, various subject interfaces, the Observers who want to observe this subject will have to register as Observers or remove themselves as Observers. For the subject, it must have something like notify observers so that it can notify all the Observers.

Now all the potential Observers need to implement the Observer interface. The Observer interface has only one method let us say update which gets called whenever the subject's state gets changed. For a concrete implementation the concrete subject will implement the subject interface and that is it will implement the Register Observer, Remover Observer and Notify Observer methods and it will also have something like the de-active state, static state and when there is a change in the state it needs to call the notify methods. The Concrete Observer needs to implement the Observer interface and it needs to only implement one method that is the update which is run when the state of the subject is changed.

(Refer Slide Time: 10:07)



Loose coupling

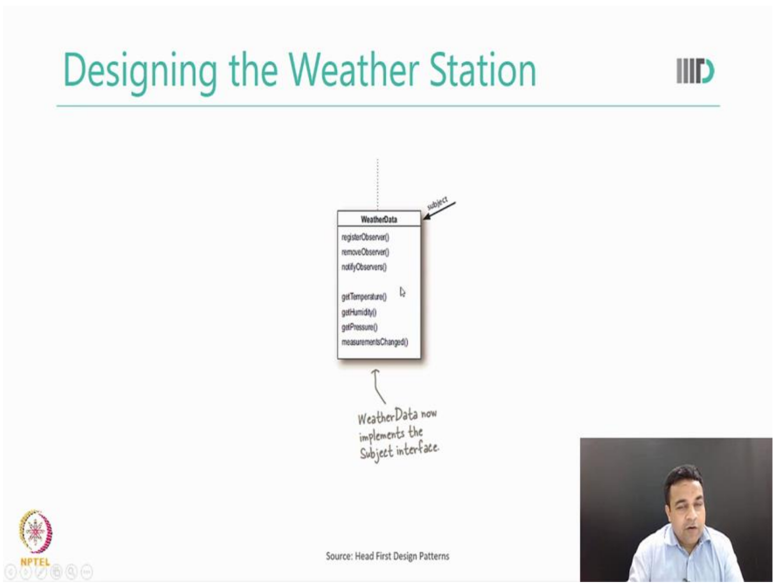
- Subjects and observers are loosely coupled
  - Only thing the subject knows about the observes is that it implements a certain interface
  - We can add/remove observers any time
    - No need to modify the subject
  - Reuse of subject and observers
  - Change to either will not affect the other
    - As long as they implement the interfaces

Source: Head First Design Patterns

There are few important things, the subjects and the Observers are loosely coupled, which means that they only know the subject knows only thing about Observers that it implements a certain interface and it need not to know anything more about the Observer. Also we can remove and add Observers at any time, without changing anything in the subject. So, this allows us the reuse of subject and Observers and the change to either of them will not affect or should not affect the other as long as they implement the interfaces that you have.

Now let us go back to our weather application. So, our weather data was one object that had many states temperature, humidity, pressure that change. And this was the state in which our Observers were interested. So our displays for the observers which were many and of different types, so only thing our Observer need to implement was the update. So, that is how we can possibly design our weather station. So, our interface for the subject should have methods such as Register Observer, Remover Observer and Notify Observer, while our interface for Observer should have only method called update. So, all our weather components implement the Observer interface this gives the subject the common interface to talk to and when it comes time to update the Observers. And then we can have simple interface to make it more modular program called display element which has the display interface. And because each of the Observer will be doing a different type of display they will implement this method as per there.

(Refer Slide Time: 12:01)

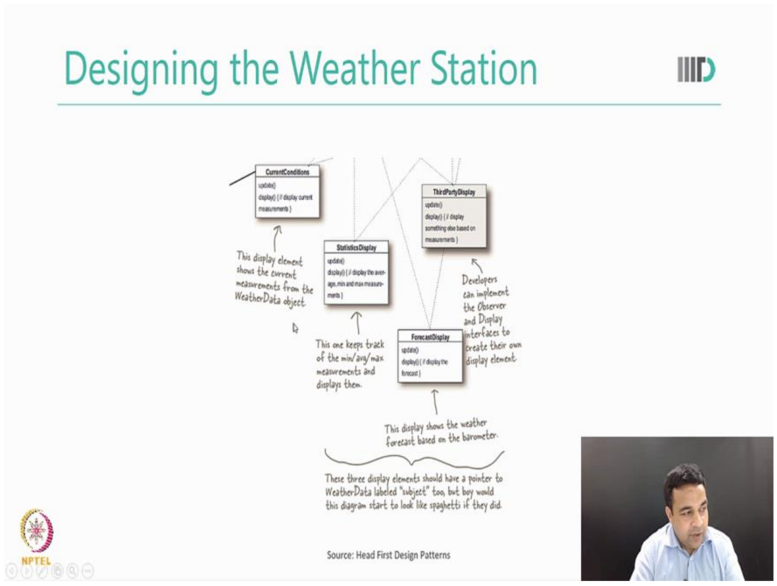


Designing the Weather Station

Source: Head First Design Patterns

Now, forever weather data we only need the 3 methods that we defined earlier and then some methods which takes account of the state of the weather something like get temperature, get humidity and get sensors and then the measurements changed is called whenever there is a change in any of these variables.

(Refer Slide Time: 12:28)



Designing the Weather Station

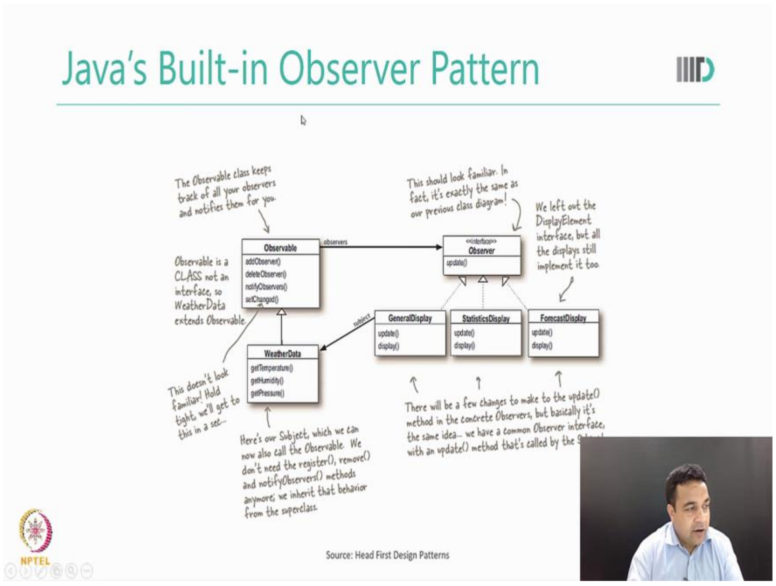
CurrentConditions
update()
display() { // display current measurements }
ThirdPartyDisplay
update()
display() { // display something else based on measurements }
StatisticsDisplay
update()
display() { // display the average, min and max measurements }
ForecastDisplay
update()
display() { // display the forecast }
This display element shows the current measurements from the WeatherData object
This one keeps track of the min/avg/max measurements and displays them.
Developers can implement the Observer and Display interfaces to create their own display element
This display shows the weather forecast based on the barometer.
These three display elements should have a pointer to WeatherData labeled "subject" too, but boy would this diagram start to look like spaghetti if they did.

Source: Head First Design Patterns

Fine now the display element shows the current measurements from the weather data objects. The display elements we have defined 3 types; statistics, current conditions and the forecast display. All the display elements display a different type of it and there could be another third party display which implement the same method and may display a different value depending on the weather data.

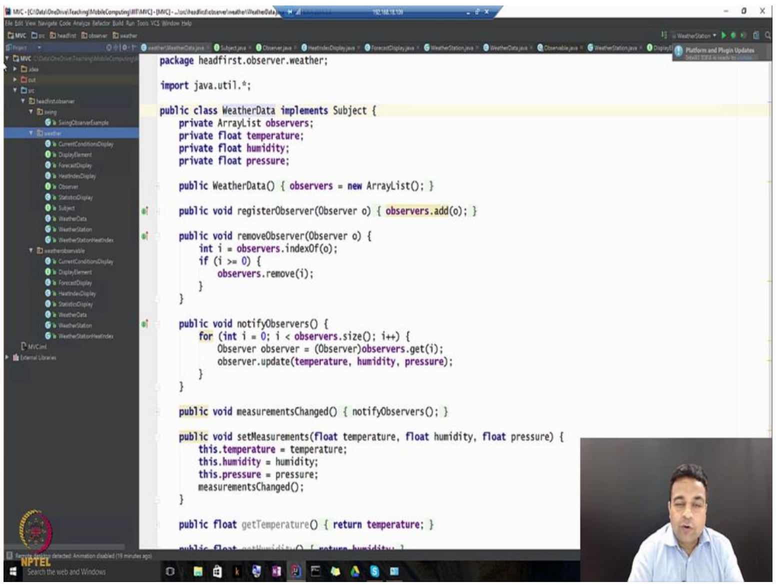
If we look overall then we have few interfaces mainly 3 interfaces. Out of them 2 interfaces are subject and observer which we described earlier. And one is an optional interface called display element because it is given just a way to display the name. All our displays that is, all our Observers need to implement the interface observer as well as the interface display element. While our subject weather data only implement our interface subject.

(Refer Slide Time: 13:43)


Java's Built-in Observer Pattern
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...
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
Source: Head First Design Patterns
NPTEL

Now, Java does provide an Observer Pattern by default, so in Java whatever we were saying for the subject is called Observable and whatever you were calling as Observer is called Observer point. Observable class takes care of all the methods that we need that is add observer, delete observer, notify observer and set change. While the Observer interface has the same update method that we had earlier. Now, let us look into a program and try to understand it.

(Refer Slide Time: 14:34)



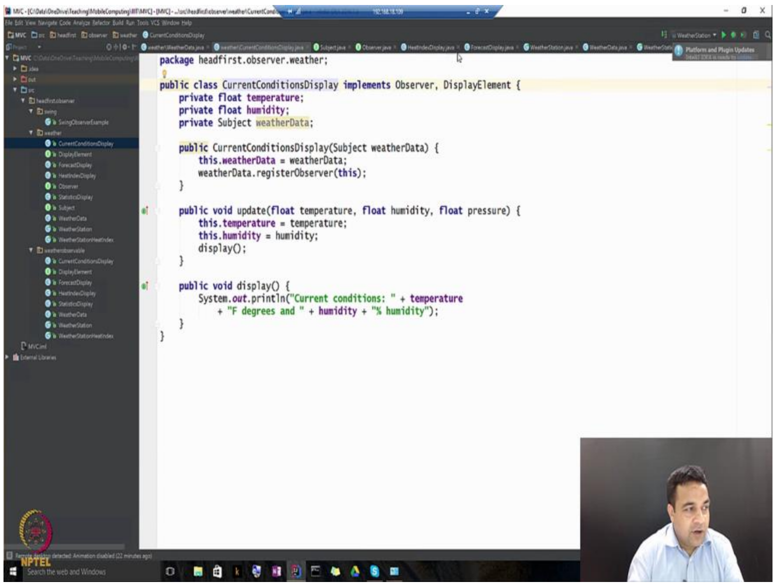
```java
package headfirst.observer.weather;

import java.util.*;

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(); }

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

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

I have created a simple program based on our example that we were discussing. So, in this program, first part of the program is made without using the Java classes for Observer Pattern and then the program is remade by using the Java classes for Observer Pattern. So, first let us see the program where we do not use any Java specific class for Observer Pattern. So, this is our first example, let us first look into our interfaces. So our subject interface has 3 methods – Register Observer, Remove Observer and Notify Observer. Our Observer interface has only one method which is update, takes the value of temperature, humidity and pressure. And then our display element interface has only one method called display.

Now, let us look at into the concrete implementation of our subject that is provided in the weather details. So, let us see how we are implementing for example, Register Observer. If we want to have functionality in which we can add or remove Observer depending on how they come and go, we need to choose a data structure which permits. One such data structure is Array, Array list can be extended. Unlike Array which cannot be extended that is why we are using Arrays. So, we use an Array list and whenever a new Observers wants to view the state of our subject or it gets interested in the state of our subject we add that as an Observer. So Observers are added using the register Observer method it means that they are added into the Array list.

Similarly when an Observer leaves, Remove Observer method is called with the index and that is remove from the Observers, erased. Now, the Notify Observers does nothing more except going through the Observers list and then calling the update method of each of the Observer. And whenever measurements changed that is whenever there is a change in a temperature or humidity or pressure, Notify Observer method is called which in turn calls the update method of all the Observers. Then in order to take in to account the state of the object, we have the method called set measurements here the new value of temperature, humidity and pressure are all set.

(Refer Slide Time: 17:19)

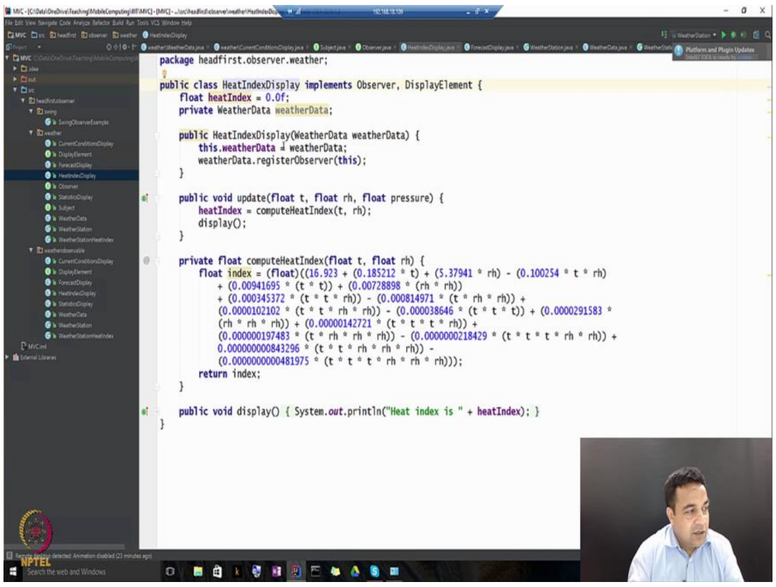


```java
package headfirst.observer.weather;

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");
    }
}
```

Now, let us look into the implementation of an Observer, a concrete implementation. For this I will choose, but I will say current conditions display. Current conditions display is an Observer so it implements the Observers interface and because it needs to display its value it also interface the display element. So, there are the values; temperature, humidity and may be pressure as well. So, Current condition display does not do anything, it gets initialized, it registers itself with the weather data. So, if you remember this was our subject and weather data was our concrete implementation of the subject. So, what Current conditions display is doing that it is registering itself to the weather data using the Register Observer method of the weather data.

So, here the Current condition display is called with the weather data in the constructor and then it sets the weather data to itself and then register to its own weather data subject variable and then it register itself as one of the Observer of the weather data. This act will add this display into the Array list of the weather data. Now, what happens whenever the update method of this display is called? Whenever the update is called it changes the value of its temperature and humidity and it displays them. The display is coming from the display element interface and does nothing but only does is system broad, upbraid print along align in our case. Similarly other Observers have been implemented.

(Refer Slide Time: 19:16)

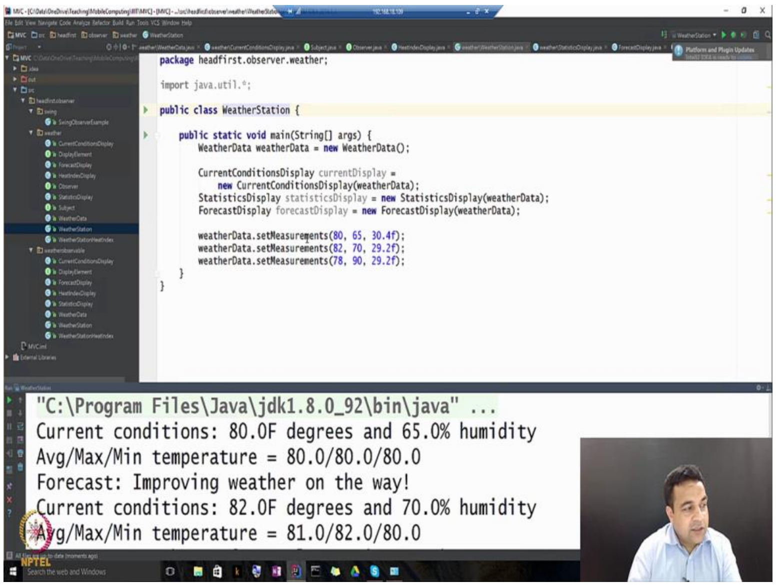


```java
package headfirst.observer.weather;

public class HeatIndexDisplay implements Observer, DisplayElement {
    float heatIndex = 0.0f;
    private WeatherData weatherData;

    public HeatIndexDisplay(WeatherData weatherData) {
        this.weatherData = weatherData;
        weatherData.registerObserver(this);
    }

    public void update(float t, float rh, float pressure) {
        heatIndex = computeHeatIndex(t, rh);
        display();
    }

    private float computeHeatIndex(float t, float rh) {
        float index = (float)((16.923 + (0.185212 * t) + (5.37941 * rh) - (0.100254 * t * rh)
            + (0.00941695 * (t * t)) + (0.00728898 * (rh * rh))
            + (0.000345372 * (t * t * rh)) - (0.000814971 * (t * rh * rh)) +
            (0.0000102102 * (t * t * rh * rh)) - (0.000038646 * (t * t * t)) + (0.0000291583 *
            (rh * rh * rh)) + (0.00000142721 * (t * t * t * rh)) +
            (0.000000197483 * (t * rh * rh * rh)) - (0.0000000218429 * (t * t * t * rh * rh)) +
            0.000000000843296 * (t * t * rh * rh * rh)) -
            (0.0000000000481975 * (t * t * t * rh * rh * rh)));
        return index;
    }

    public void display() { System.out.println("Heat index is " + heatIndex); }
}
```

For example, Forecast display again take the weather data object, registers itself and updates, though its display is different than the display of Current conditions. And same thing is for the heat index which registers itself similar way, updates itself in a almost similar way but its display is very much different than the other two displays because it tries to complete a heat index using a mathematical formula. And then we have another display, another observer which is statistics display which does nothing accepts adding itself and then it tries to calculate min, max average and displays loose values. So, this is the concrete implementation of our subject and Observers, I will just quickly repeat.

The subject, need to have a data structure in which it can store the Observers. The Observers take the subject object as part of their constructor and rest of themselves as Observers. Implement the update method and hopefully they are registered as the Observers. Now let us check this program, so, here is a class that checks it. It creates a weather data that is a subject and it creates 3 objects, 3 Observers of that subject. So, current conditions display, statistics display, forecast displays are 3 Observers of the same subject. And then one by one I am changing the measurements. Let us run this program and check the output for it. So, we will go to weather station you right click and we will say run weather station, let us come down.

(Refer Slide Time: 20:59)

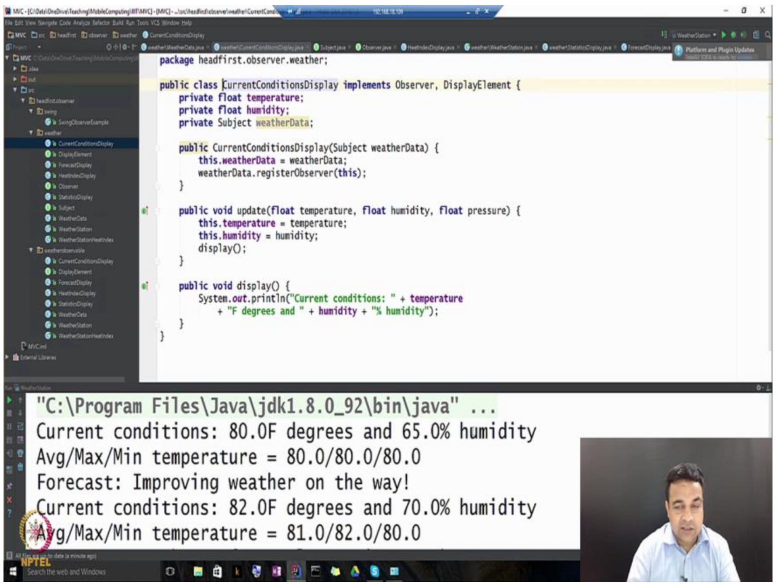


```java
package headfirst.observer.weather;

import java.util.*;

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);
    }
}
```

```
"C:\Program Files\Java\jdk1.8.0_92\bin\java" ...
Current conditions: 80.0F degrees and 65.0% humidity
Avg/Max/Min temperature = 80.0/80.0/80.0
Forecast: Improving weather on the way!
Current conditions: 82.0F degrees and 70.0% humidity
Avg/Max/Min temperature = 81.0/82.0/80.0
```

So, when the first set measurement is called, let us see what the work flow is. The set measurements will come here and will in turn call measurements changed. The measurement changed will then in turn call Notify Observers. The Notify Observers will then call the update method of each of the Observer that is where the update method is defined. Now, let us go back and see the output. So, for the Current conditions for example, the update method calls the display and display only prints out its statements and we can see this statement here and all.

(Refer Slide Time: 21:45)

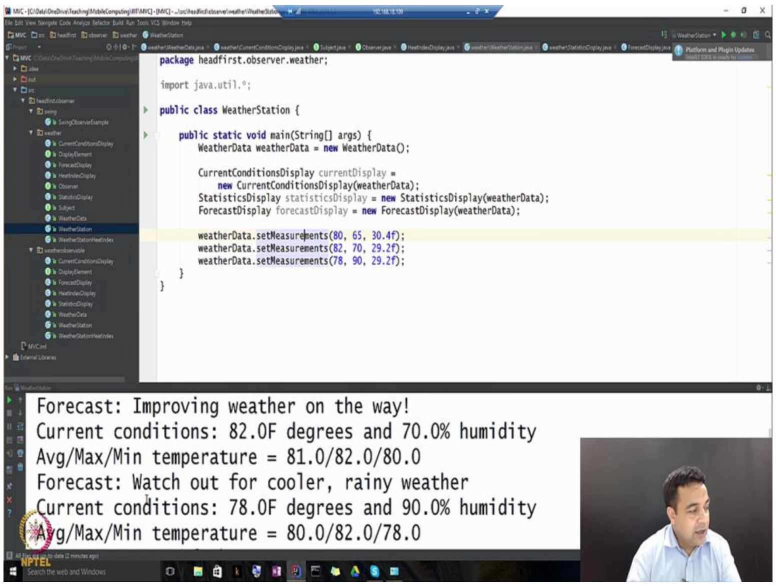


```java
package headfirst.observer.weather;

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");
    }
}
```

```
"C:\Program Files\Java\jdk1.8.0_92\bin\java" ...
Current conditions: 80.0F degrees and 65.0% humidity
Avg/Max/Min temperature = 80.0/80.0/80.0
Forecast: Improving weather on the way!
Current conditions: 82.0F degrees and 70.0% humidity
Avg/Max/Min temperature = 81.0/82.0/80.0
```

Current condition: 80.0F degrees and 65.0% humidity. Similar things will happen with respect to the display of statistics and forecast. So, if you check the statistics display its nothing but prints average max, min as you can see the average max, min and then the forecast display also does nothing else accept display the forecast and a good tag lines for example, improving weather on the way that is what is published.

(Refer Slide Time: 22:34)

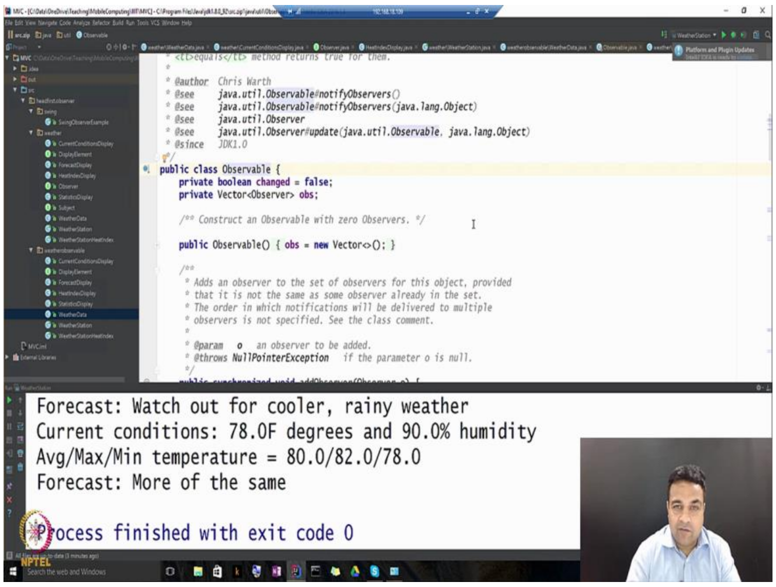


```java
package headfirst.observer.weather;

import java.util.*;

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);
    }
}
```

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

Now, when the measurements got changed, in real life these measurements will be changed because the sensor that was sensing the values, registered a new value. So, whenever the measurements got changed all the displays will be updated and as you can see the new new 3 lines shows the new updates, you see now the forecast has changed. And then when again the measurements change and again the displays are updated. So, you saw that how easy it was to implement the Observer Pattern with nothing more than the standards of our classes. Now, let us look again into what Java provides to us and how we can use the observable and the observer of the Java and implement the observer pattern. So, we will be using the same example, so let me go and show you.

(Refer Slide Time: 23:49)

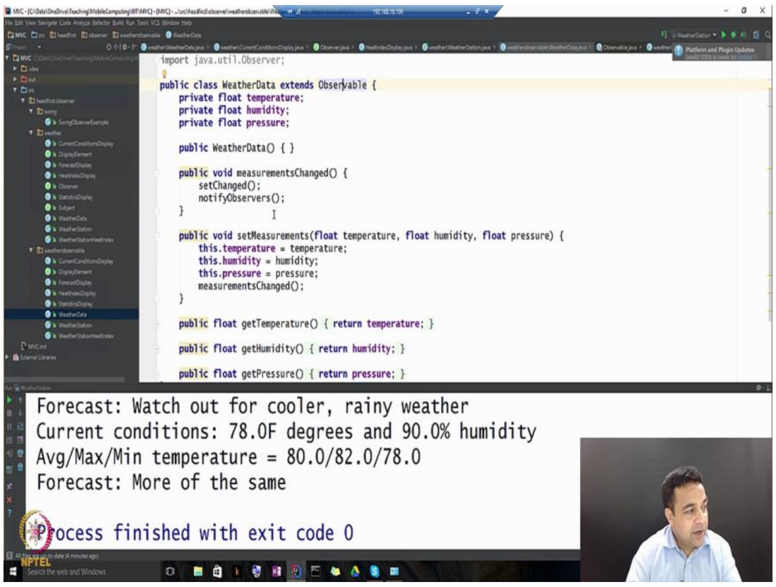


```
 * <tt>equals</tt> method returns true for them.
 *
 * @author  Chris Warth
 * @see     java.util.Observable#notifyObservers()
 * @see     java.util.Observable#notifyObservers(java.lang.Object)
 * @see     java.util.Observer
 * @see     java.util.Observer#update(java.util.Observable, java.lang.Object)
 * @since   JDK1.0
 */
public class Observable {
    private boolean changed = false;
    private Vector<Observer> obs;

    /** Construct an Observable with zero Observers. */

    public Observable() { obs = new Vector<>(); }

    /**
     * Adds an observer to the set of observers for this object, provided
     * that it is not the same as some observer already in the set.
     * The order in which notifications will be delivered to multiple
     * observers is not specified. See the class comment.
     *
     * @param   o   an observer to be added.
     * @throws NullPointerException   if the parameter o is null.
     */
```

```
Forecast: Watch out for cooler, rainy weather
Current conditions: 78.0F degrees and 90.0% humidity
Avg/Max/Min temperature = 80.0/82.0/78.0
Forecast: More of the same

Process finished with exit code 0
```

So, the second implementation uses the Java classes, so here our weather data is same but it now extends Observable and the Observable is one of the Java classes which I have imported. If I go into the code of Observable by pressing control on my key board, and then pressing click, I can see how Observable is implemented in Java. That is another beauty of Java programming language that all the code is available for you to see. And you see that Observable is implemented pretty much the same way that we implemented our own subject. There are some add Observer methods there would be a Delete Observer method and there is a Notify Observer. We were using Array list they are using number. Let us come back so now you need not to implement the Array list of managing Java is do not need for you. All you need to do is to have the measurement changed and the set measurement method.

(Refer Slide Time: 24:38)

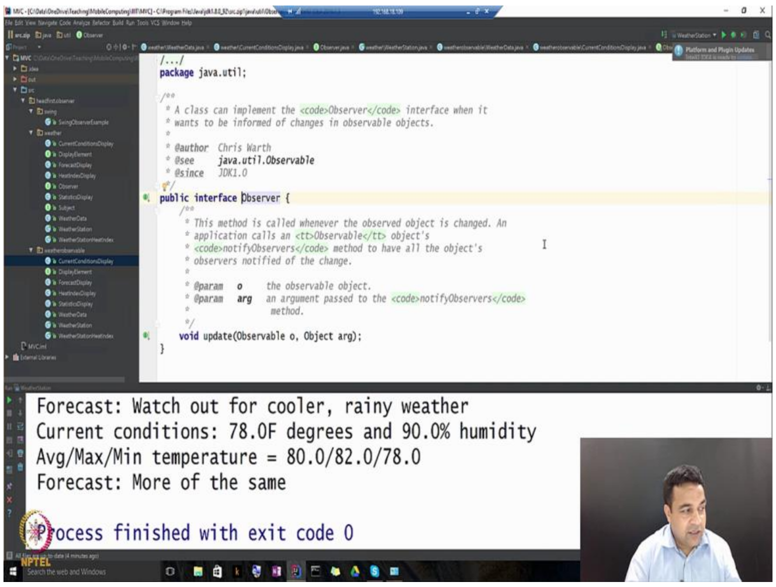


```
import java.util.Observer;

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; }
```

```
Forecast: Watch out for cooler, rainy weather
Current conditions: 78.0F degrees and 90.0% humidity
Avg/Max/Min temperature = 80.0/82.0/78.0
Forecast: More of the same

Process finished with exit code 0
```

In our earlier weather data, we were also implementing register and remove now we need not to do that. Only thing we need not we need to do is measurement changed. Now, let us see what is the difference in the Observer? So, let us pick the Current conditions display Observer which is same as what we picked earlier. Now this is implementing an Observer and here is the Observer that it is implementing.

(Refer Slide Time: 25:10)



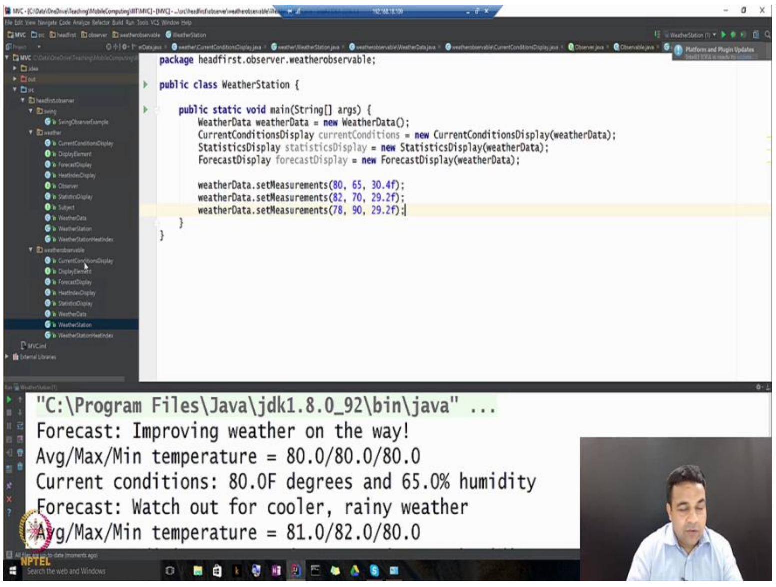
```
/.../
package java.util;

/**
 * A class can implement the <code>Observer</code> interface when it
 * wants to be informed of changes in observable objects.
 *
 * @author  Chris Warth
 * @see     java.util.Observable
 * @since   JDK1.0
 */
public interface Observer {
    /**
     * This method is called whenever the observed object is changed. An
     * application calls an <tt>Observable</tt> object's
     * <code>notifyObservers</code> method to have all the object's
     * observers notified of the change.
     *
     * @param   o     the observable object.
     * @param   arg   an argument passed to the <code>notifyObservers</code>
     *                 method.
     */
    void update(Observable o, Object arg);
}
```

```
Forecast: Watch out for cooler, rainy weather
Current conditions: 78.0F degrees and 90.0% humidity
Avg/Max/Min temperature = 80.0/82.0/78.0
Forecast: More of the same

Process finished with exit code 0
```

Let me click it, so this Observer is also an interface find by Java it also has only one method called update, which takes the Observable as one of the parameter and object as another of it. So, let us go back and see just like the last program, this one also takes our subject as a part of the constructor, adds itself to that subject and then calls the update. So, whatever we were doing earlier ourselves Java is doing it now for us using the Observer and Observable.

Let me also run this program and hopefully I will get a similar output. So, this is exactly same. Let me done it by right click then I do this and as you see that every time just like the last program, I am again getting all the 3 outputs being displayed.

(Refer Slide Time: 26:18)

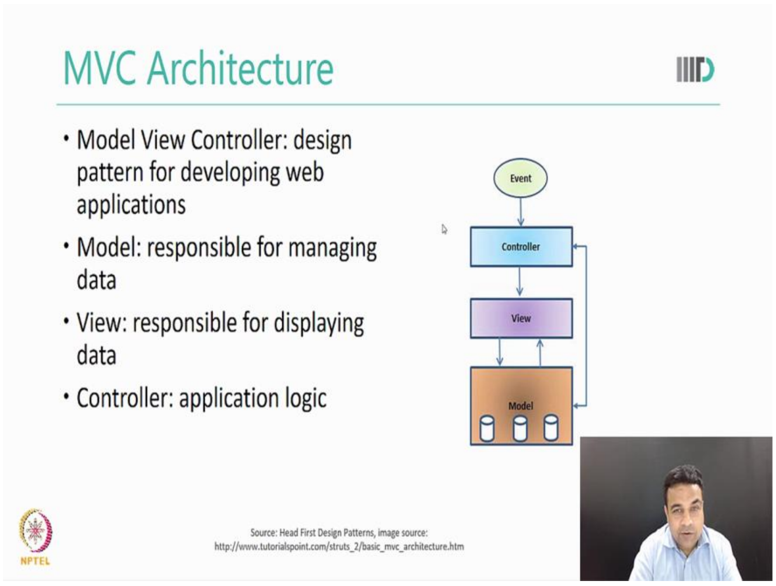


```java
package headfirst.observer.weatherobservable;

public class WeatherStation {

    public static void main(String[] args) {
        WeatherData weatherData = new WeatherData();
        CurrentConditionsDisplay currentConditions = 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);
    }
}
```

```
"C:\Program Files\Java\jdk1.8.0_92\bin\java" ...
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
```

So, this these are 2 very simple programs showing you how you can use Observer Pattern in a Java program when you need to define multiple Observer for a single subject which has some interesting states. Why I discussed Observer Pattern is based on that we are using and developing android application which use this MVC architecture patterns and MVC is very related to Observer Pattern as you can now guess.

(Refer Slide Time: 26:53)



MVC Architecture

- Model View Controller: design pattern for developing web applications
- Model: responsible for managing data
- View: responsible for displaying data
- Controller: application logic

Event
Controller
View
Model

Source: Head First Design Patterns, image source: http://www.tutorialspoint.com/struts_2/basic_mvc_architecture.htm

We have discussed MVC earlier before we started improving our math quiz applications, but here is a quick repetition. In MVC also, there is a Model, a View, a Controller, while the View is responsible for displaying data, Model is responsible for managing data and Controller is for the application logic. You can see the relation between the subject and observers. If something changes in the model, the view is updated. This is all for today’s class, it was a little bit different and away from the android programming, but understanding some of these patterns is very important to understand and to learn how to develop good android applications.

Thank you.