A Weather Station

- Create an application that initially provides three display elements:
  - Current conditions
  - Weather statistics
  - Simple forecast

- Also provide an API so that other developers can write their own weather displays.
Overview

Weather Station

Humidity Sensor Device

Pressure Sensor Device

Temperature Sensor Device

Weather data object

Pulls data

Displays

Current conditions

Temp: 72°

Humidity: 60

Pressure: ↓

Weather-O-Rama Services

The Weather Object

public void measurementsChanged() {
    currentConditionsDisplay.update(….);
    weatherStatisticsDisplay.update(….);
    simpleForecastDisplay.update(….);
}

WeatherStation

WeatherData

getTemperature()

getHumidity()

getPressure()

measurementsChanged()

Interacts directly with the weather station. When data changes it invokes measurementsChanged()
Misguided Solution

```java
public class WeatherData {
    // instance variable declarations
    float temp = getTemperature();
    float humidity = getHumidity();
    float pressure = getPressure();

    currentConditionsDisplay.update(temp, humidity, pressure);
    statisticsDisplay.update(temp, humidity, pressure);
    forecastDisplay.update(temp, humidity, pressure);
}
// other WeatherData methods here
```

1. We are coding to concrete implementations, not interfaces. **True**
2. For every new display we need to alter code. **True**
3. We have no way to add (or remove) display elements at runtime. **True**
4. The display elements don’t implement a common interface. **Looks like they might.**
5. We haven’t encapsulated the part that changes. **True**
6. We are violating encapsulation of the weather class. **Not really**

A Better Solution
The Observer Pattern

Subject object manages data. Currently it holds a red triangle. When the data in the subject changes, the observers are notified. The observers have subscribed (registered to) the subject to receive updates when the subject's data changes.

Subject object

Dog object

Cat object

Mouse object

Observer Objects

The data can be sent directly to them, or they make a call back on the subject object to fetch the data.

Duck object

Duck object didn't register so it isn’t notified of changes.

Observer Objects

Observer

- Many-to-one dependency between objects
- Use when there are two or more views on the same “data”
- aka “Publish and subscribe” mechanism
- Choice of “push” or “pull” notification styles

Defines a one-to-many relationship between a set of objects.

When the state of one object changes, all of its dependents are notified.
Cat and Mouse

In this simple animation the mouse moves around the canvas – sometimes disappearing down an invisible mouse-hole and then reappearing elsewhere.

The cats observe the current position of the mouse and try to follow him around the canvas.

1 Traditional (from scratch) implementation.
2 Java inbuilt implementation

Disclaimer: The animation is only for display in class. The code I share with you is non-GUI based. Next lecture we will start working with GUIs.

Observer interface and concrete observer class implemented ‘from scratch’ in java.

1 Observer interface requires all concrete classes to implement a notify(Subject s) method.
2 The notify(Subject s) operation is called by Subject class when the subject changes.
3 The concrete observer (cat) makes a call-back onto the subject (mouse) object to get the data it wants, i.e. the mouse position.

```java
public interface Observer {
    public void notify(Subject s);
}

public class Cat implements Observer{
    Point mousePosition; // Current position of the mouse
    Point catPosition; // Current position of this cat
    Random rand = new Random();

    @Override
    public void notify(Subject s) {
        if(s instanceof Mouse) {
            mousePosition = ((Mouse)s).getPosition();
            moveCat();
        }
    }

    public Cat()
    { catPosition = new Point(rand.nextInt(800), rand.nextInt(800));

    }

    public void moveCat(){
        if(rand.nextInt(2)==1) { // Slow down the cat.
            if (catPosition.x - mousePosition.x < 0)
                catPosition.x++;
            else
                catPosition.x--;
            if (catPosition.y - mousePosition.y < 0)
                catPosition.y++;
            else
                catPosition.y--;
        }
        System.out.println("Cat: " + catPosition.x + " " + catPosition.y);
    }
}
```
Subject interface and concrete subject class implemented ‘from scratch’ in java.

Subject interface requires all concrete classes to implement three methods:

- Register observers
- Remove observers
- Notify observers

Here we see notification at work:

1. The mouseMove method is executed, the mouse moves, its state changes and it invokes notifyObservers()
2. All currently registered observers are notified of the change. “this” subject object is passed to them and used for call-backs. (See the cat example a few slides back!)
Java Inbuilt Observer

We can use the Java inbuilt Observer Pattern

<< Observer >>
https://docs.oracle.com/javase/8/docs/api/java/util/Observer.html

Observable
https://docs.oracle.com/javase/8/docs/api/java/util/Observable.html

Java: Observer Pattern

Java provides an inbuilt Observer Pattern in the form of:

• **An Observable superclass**
  Extend this class if you want something to be observed. i.e. this plays the role of the subject.

• **An Observer interface**
  Implement this interface if you want to register this object as an observer.
Before” (using the custom-built from-scratch Observer interface) ...and “after” (using Java’s inbuilt Observer interface).

What changed?

```
public class Cat implements Observer{
    Point mousePosition; // Current position of the mouse
    Point catPosition; // Current position of this cat
    Random rand = new Random();

    @Override
    public void notify(Subject s) {
        if(s instanceof Mouse)
            mousePosition = ((Mouse)s).getPosition();
        moveCat();
    }
}
```

```
public class Mouse extends Observable {
    Point mousePosition = new Point(0,0,0);
    Point targetPosition;
    Random rand;

    public Mouse(){
        System.out.println("Mouse: " + mousePosition + " + targetPosition:");
        notifyObservers();
    }
    public void moveMouse(){
        while (Math.abs(mousePosition.x-targetPosition.x) + 
        Math.abs(mousePosition.y-targetPosition.y) < 20){
            setMousePosition();
        }
        if (mousePosition.x - targetPosition.x < 0)
            mousePosition.x++;
        else
            mousePosition.x--;
        if (mousePosition.y - targetPosition.y < 0)
            mousePosition.y++;
        else
            mousePosition.y--;
        System.out.println("Mouse: " + mousePosition + " + targetPosition:");
        notifyObservers();
    }
}```
Casting

Upcasting:
- Java permits an object of a subclass type to be treated as an object of any superclass type.
- Upcasting is done automatically.
- Supports polymorphism
- Example:
  
  ```java
  List<Observer> observers = new LinkedList<Observer>();
  ```

Downcasting:
- Downcasting must be manually performed by the programmer
- Only sensible if the object being cast is an instance of that type
- We need to test the type before downcasting using:
  ```java
  if ([object] instanceof [class name]) do something;
  ```
- Example:

```java
public void notify(Subject s) {
  if(s instanceOf Mouse) {
      mousePosition = ((Mouse)s).getPosition();
      move(@); // do something
  }
}
```