

오피저버 패턴

이관우

kwlee@hansung.ac.kr

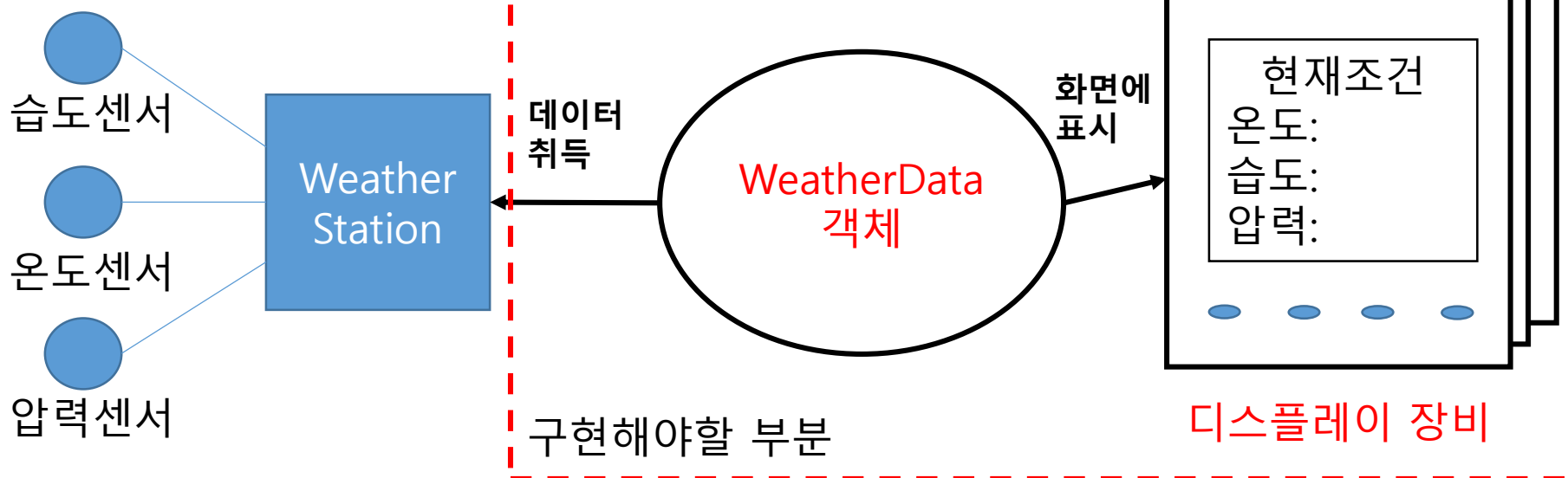
학습 목표

- 옵저버 패턴이 필요한 상황을 이해한다. (문제 상황)
- 옵저버 패턴의 작동 메커니즘을 이해한다. (해결 방안)
- 옵저버 패턴을 직접 구현해 본다. (구현)
- 자바 내장 옵저버 패턴을 사용해 보고, 이의 장단점을 분석해 본다. (대안구현 및 분석)
- 옵저버 패턴의 적용 예를 이해한다. (적용사례)

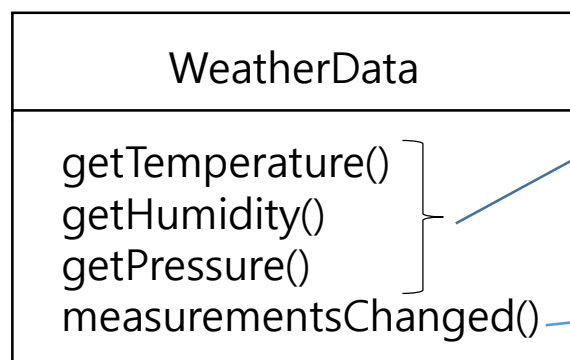
기상 모니터링 애플리케이션

- 구성 요소

- 기상 스테이션(Weather Station) : 실제 기상 정보를 수집하는 장비
- **WeatherData 객체** : 기상 스테이션으로부터 수집된 데이터
- **디스플레이 장비 종류**
 - 현재 조건 (온도, 습도, 압력)
 - 기상 통계
 - 기상 예보



기상 모니터링 애플리케이션 규격



측정값을 얻는 메소드

새로운 측정 데이터가 나올 때마다
호출되는 메소드
(어떻게 호출되는 지는 알필요 없음)

- 세 개의 디스플레이 항목(현재조건, 기상통계, 기상예보)을 구현
- 확장성
 - 향후에 새로운 디스플레이 항목을 추가 혹은 제거할 수 있어야 함

간단한 구현 방안

```
Public class WeatherData {
```

```
    // 인스턴스 변수 선언
```

```
    public void measurementsChanged() {
```

```
        float temp = getTemperature();  
        float humidity = getHumidity();  
        float pressure = getPressure();
```

측정값 얻어옴

```
        currentConditionDisplay.update(temp, humidity, pressure);  
        statisticsDisplay.update(temp, humidity, pressure);  
        forecastDisplay.update(temp, humidity, pressure);
```

디스플레이 갱신

```
    }
```

```
    // 기타 메소드
```

```
}
```

코드 분석

```
Public class WeatherData {
```

```
    // 인스턴스 변수 선언
```

```
    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);
```

```
    }
```

```
    // 기타 메소드
```

```
}
```

표준화된 데이터 전송
인터페이스



변화 가능한 부분
(WeatherData가 구체적인 디스플레이
개체들과 직접 상호작용함)

디자인 목표

```
Public class WeatherData {
```

```
// 인스턴스 변수 선언
```

```
public void measurement
```

```
float temp = getTemperature();
```

```
float humidity = getHumidity();
```

```
float pressure = getPressure();
```

```
currentConditionsDisplay.update(temp, humidity, pressure);  
statisticsDisplay.update(temp, humidity, pressure);  
forecastDisplay.update(temp, humidity, pressure);
```

```
}
```

```
// 기타 메소드
```

```
}
```

변화 가능한 부분을 어떻게
캡슐화하여 이의 변화가
WeatherData 클래스에 영향을
주지 않도록 설계할 것인가?

디자인 원칙

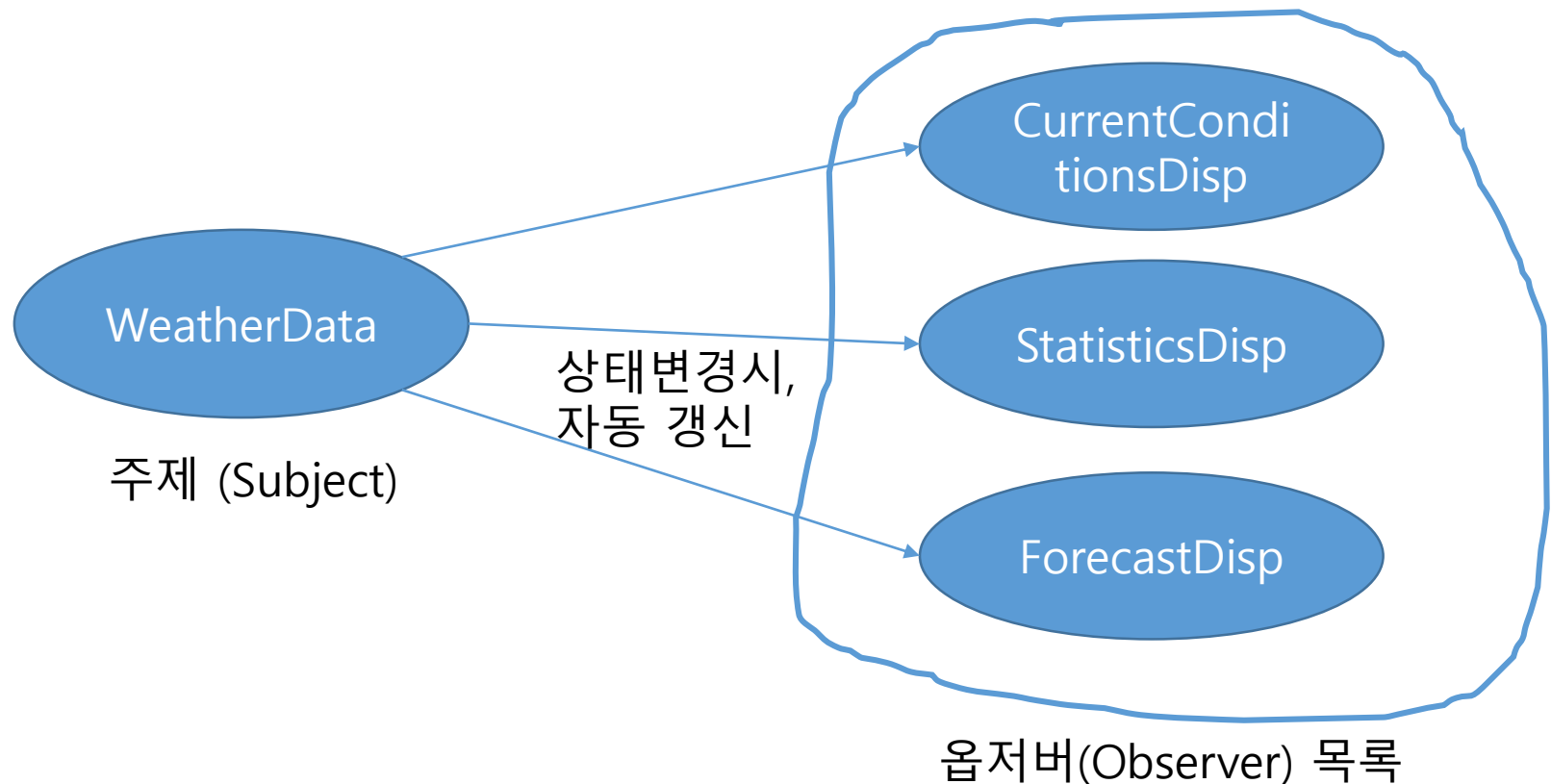
서로 상호작용하는 객체 사이에서는 가능하면
느슨하게 결합하는 디자인을 사용해야 한다



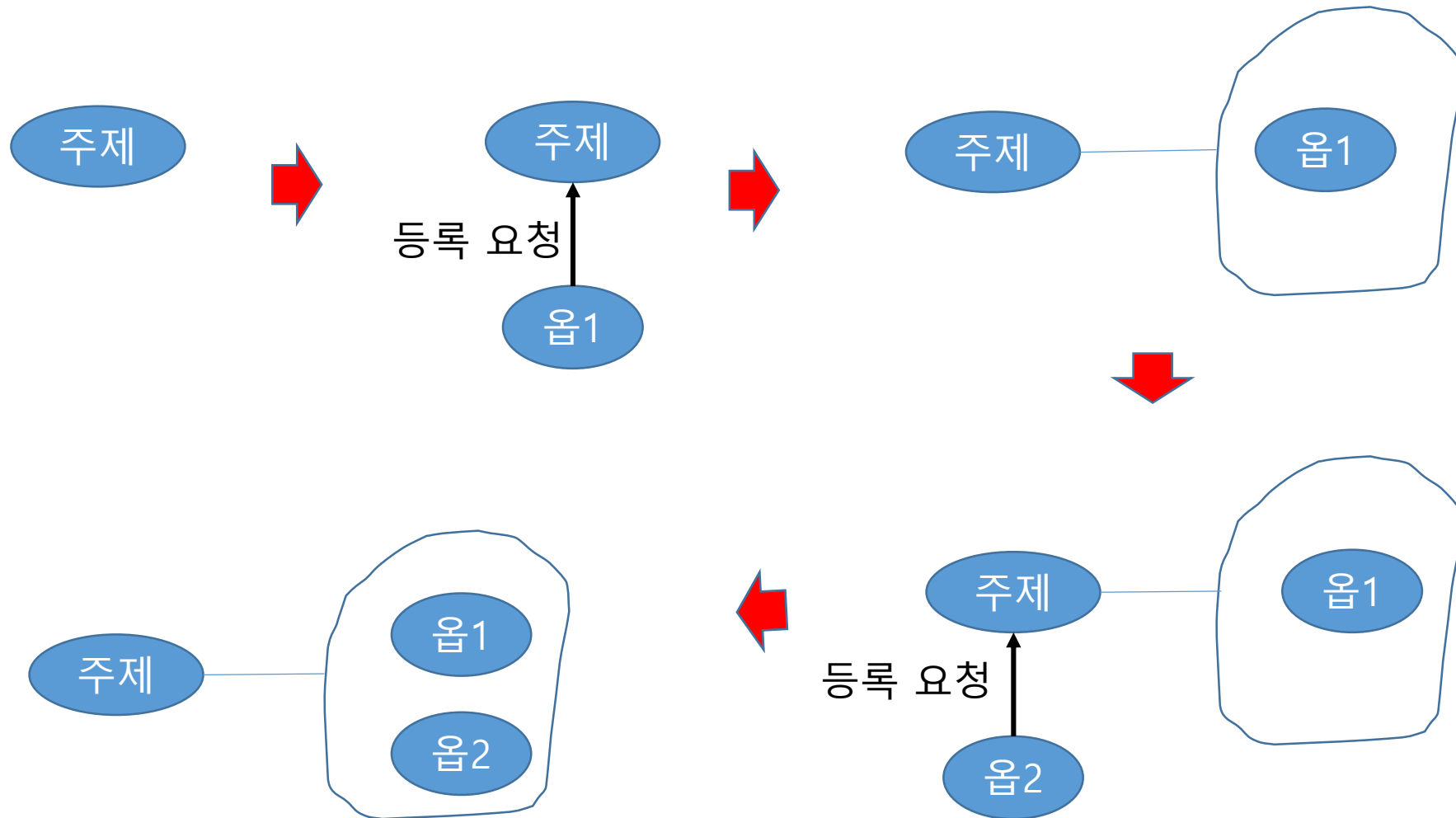
상호작용하는 두 객체가 서로에 대해서 구체적인 실체 및 구현에 대해서는 모르고,
단지 서로의 추상화된 인터페이스만을 알고 있음을 의미한다.

옵저버 패턴

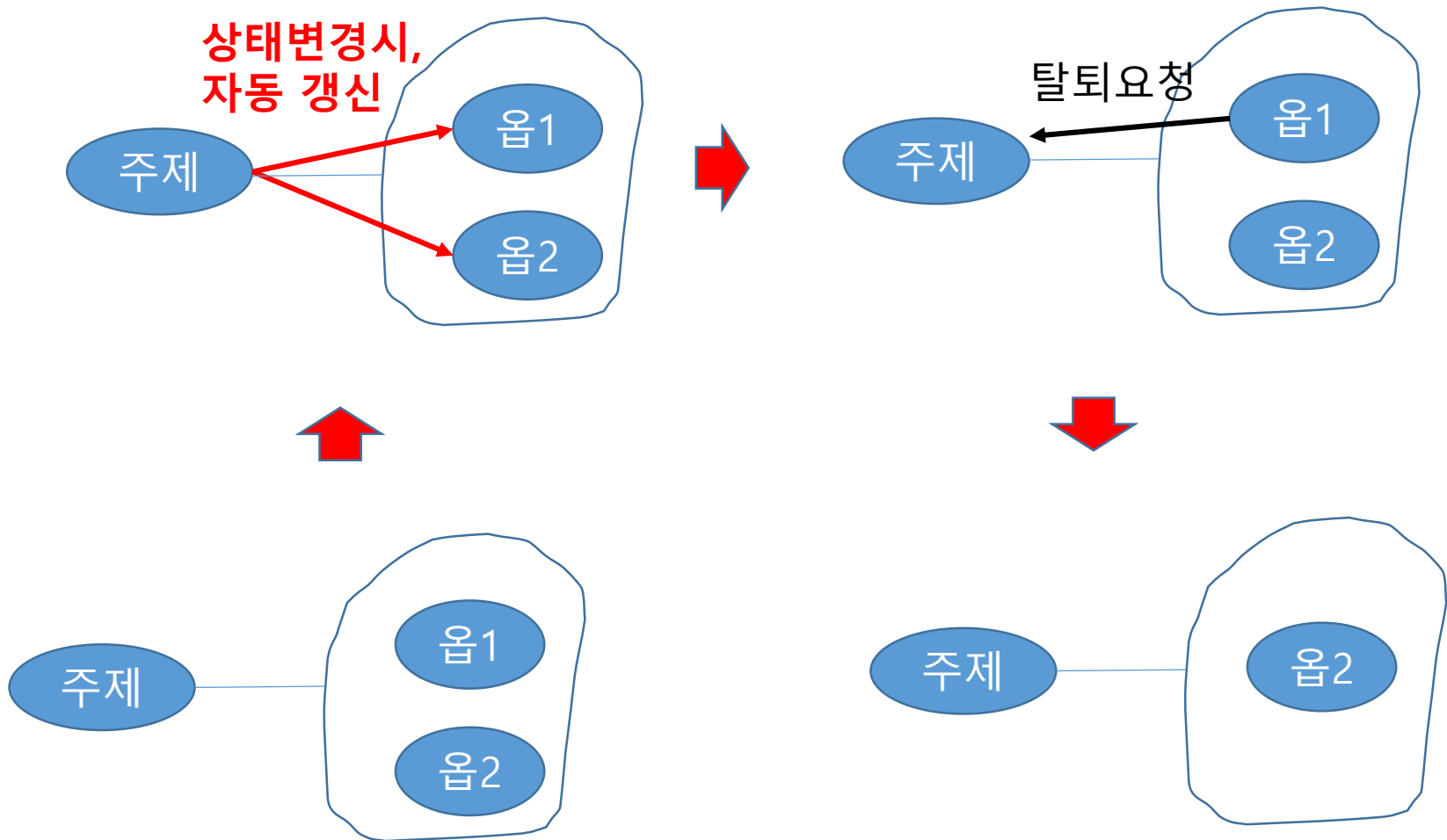
- 한 객체의 상태가 바뀌면 그 객체에 의존하는 다른 객체들한테 연락이 가고, 자동으로 내용이 갱신되는 방식으로 일대다 (one-to-many) 의존성을 정의



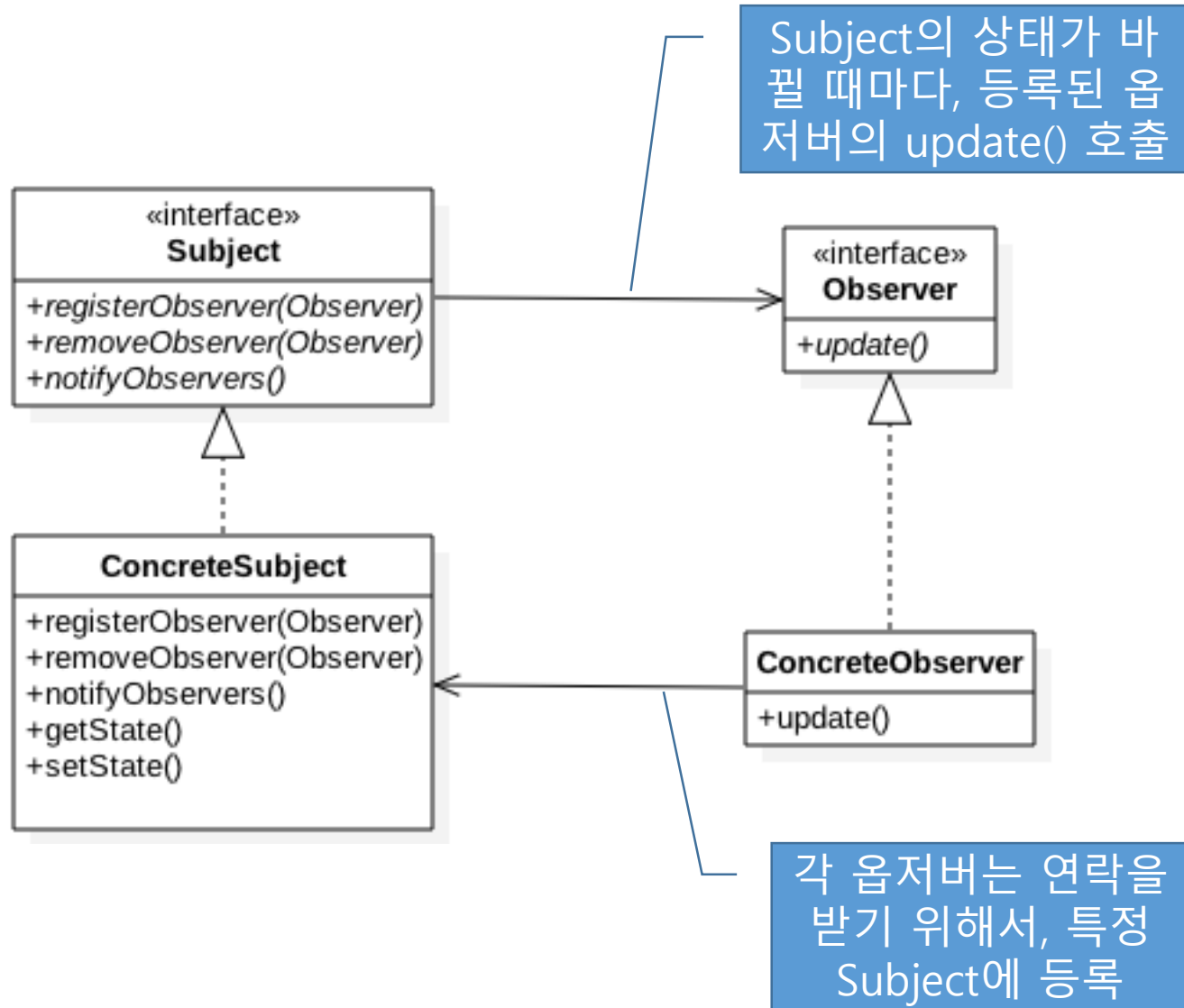
옵저버 패턴의 동작 원리



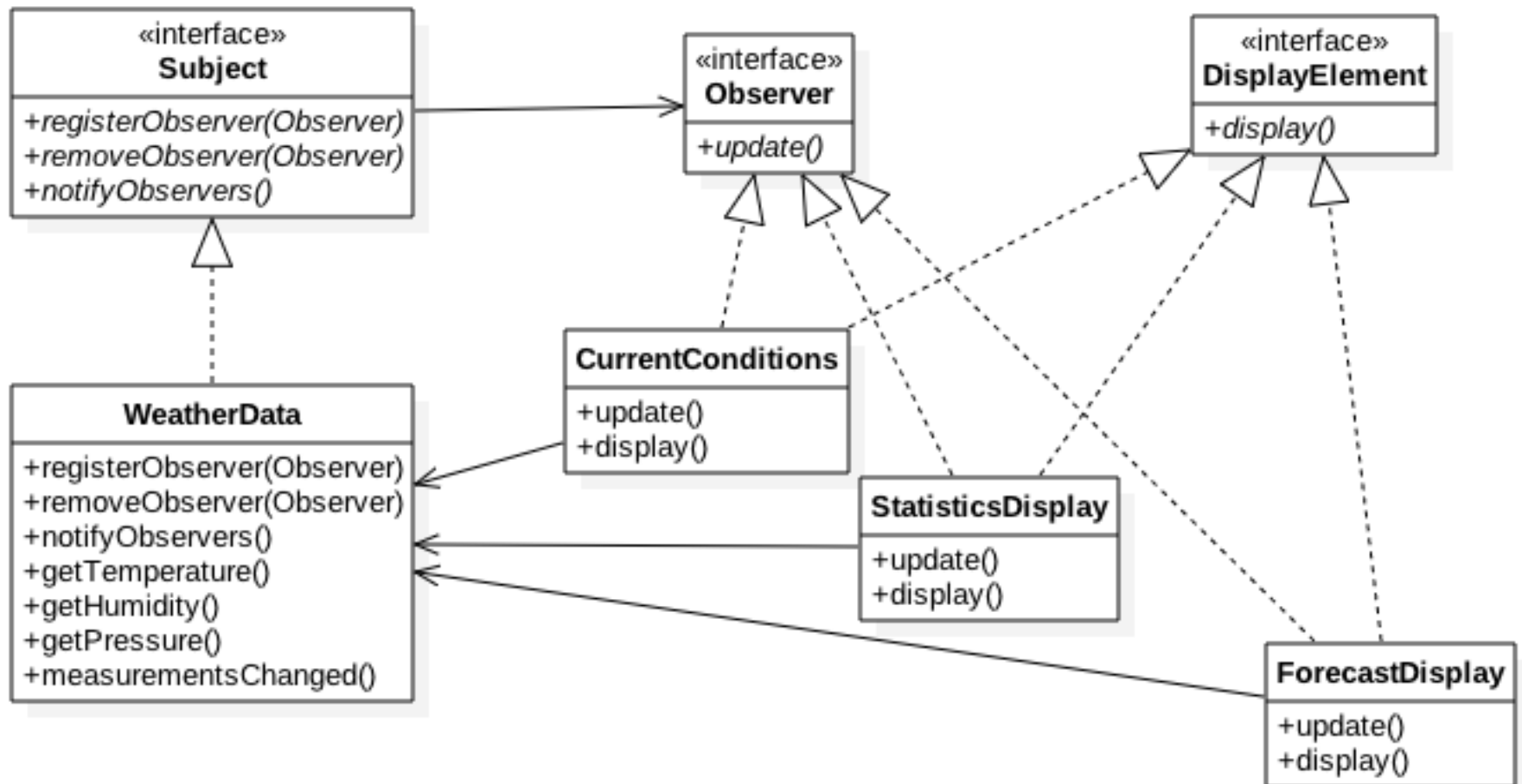
옵저버 패턴의 동작 원리



옵저버 패턴: 클래스 다이어그램



기상 스테이션 설계



기상 스테이션 구현

```
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();  
}
```

Subject 인터페이스 구현

```
public class WeatherData implements Subject {  
    private ArrayList<Observer> observers;  
    private float temperature;  
    private float humidity;  
    private float pressure;  
  
    public WeatherData() {  
        observers = new ArrayList<Observer>();  
    }  
  
    public void registerObserver(Observer o) {  
        observers.add(o);  
    }  
  
    public void removeObserver(Observer o) {  
        ...  
    }  
    ...  
}
```

옵저버를 관리하는 저장소

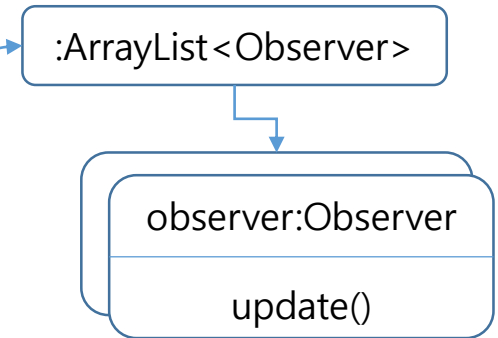
:ArrayList<Observer>

o:Observer

옵저버 객체 o를 리스트에 등록

Subject 인터페이스 구현

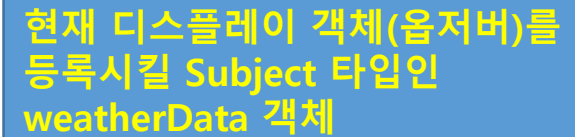
```
public class WeatherData implements Subject {  
    private ArrayList<Observer> observers;  
    ...  
    public void notifyObservers() {  
        for (Observer observer : observers) {  
            3 observer.update(temperature, humidity, pressure);  
        }  
    }  
    public void measurementsChanged() {  
        2 notifyObservers();  
    }  
    public void setMeasurements(float temperature, float humidity,  
                                float pressure) {  
        this.temperature = temperature;  
        this.humidity = humidity;  
        this.pressure = pressure;  
        1 measurementsChanged();  
    }  
}
```



Display 항목 구현

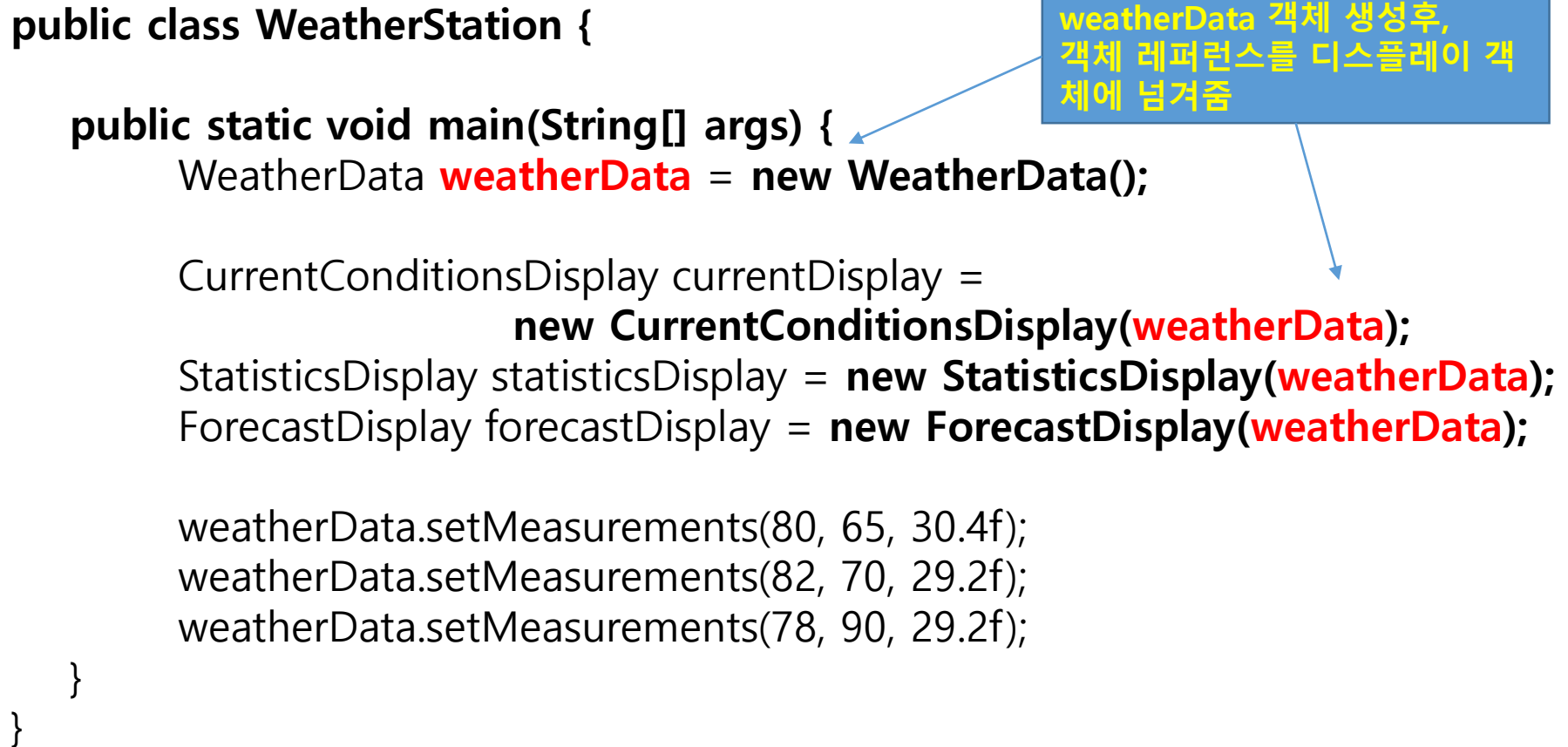
```
public class CurrentConditionsDisplay implements Observer, DisplayElement {  
    private float temperature;  
    private float humidity;  
    private Subject weatherData;  
  
    public CurrentConditionsDisplay(Subject weatherData) {  
        this.weatherData = weatherData;  
        weatherData.registerObserver(this);  
    }  
    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 ...);  
    }  
}
```

현재 디스플레이 객체(옵저버)를
등록시킬 Subject 타입인
weatherData 객체



기상 스테이션 테스트용 클래스

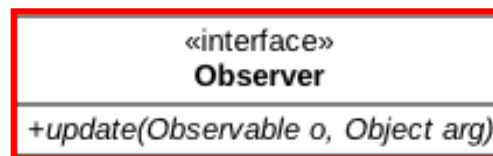
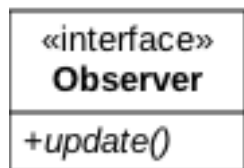
```
public class WeatherStation {  
  
    public static void main(String[] args) {  
        WeatherData weatherData = new WeatherData();  
  
        CurrentConditionsDisplay currentDisplay =  
            new CurrentConditionsDisplay(weatherData);  
        StatisticsDisplay statisticsDisplay = new 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);  
    }  
}
```



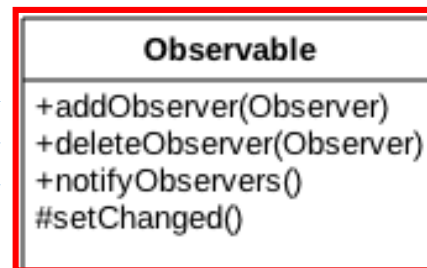
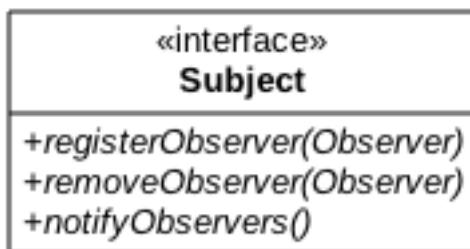
A blue box contains the text: "weatherData 객체 생성후, 객체 레퍼런스를 디스플레이 객체에 넘겨줌". Two arrows originate from this box. One arrow points to the **weatherData** variable in the line `WeatherData weatherData = new WeatherData();`. The other arrow points to the **weatherData** argument in the line `new CurrentConditionsDisplay(weatherData);`.

자바 내장 옵저버 패턴 사용하기

- **java.util.Observer** 인터페이스
 - 앞에서 설명한 옵저버 패턴의 **Observer** 인터페이스와 유사

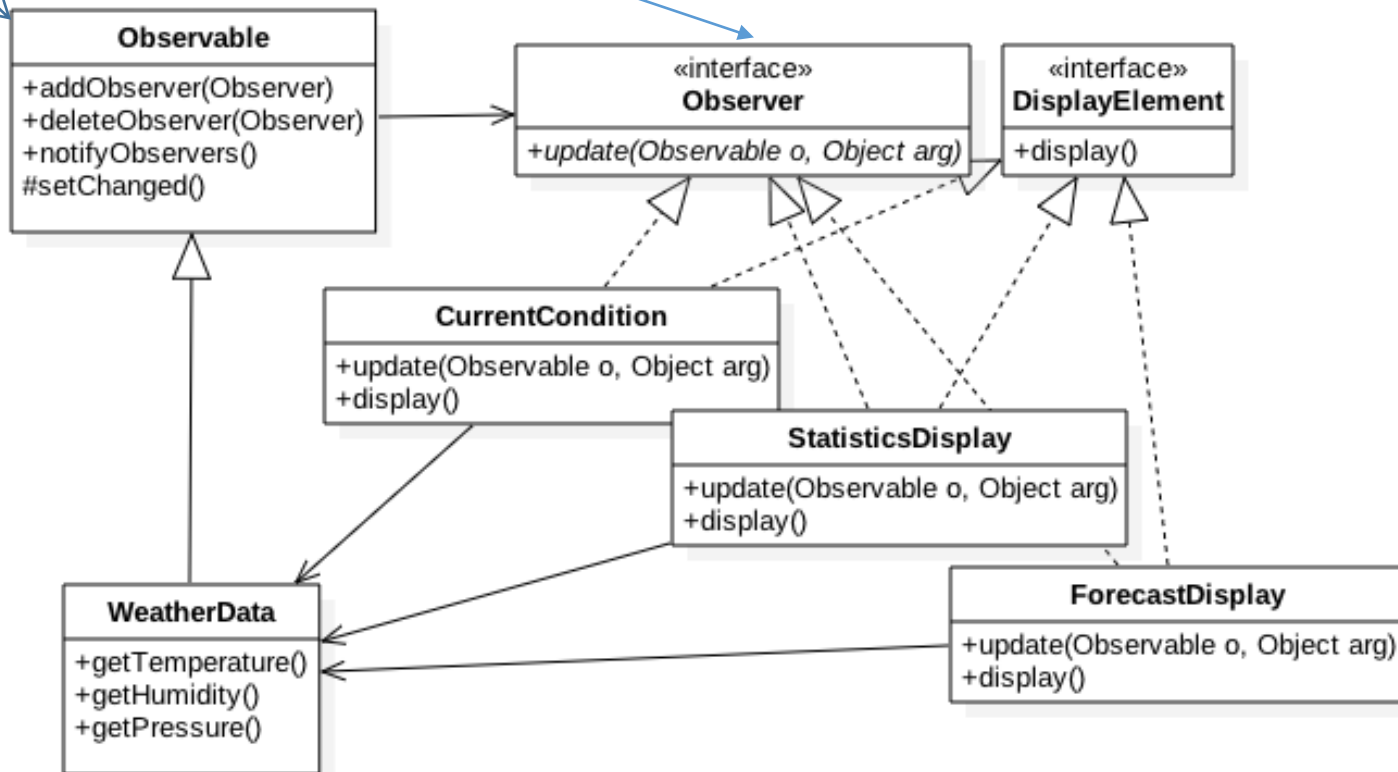


- **java.util.Observable** 클래스
 - 앞에서 설명한 옵저버 패턴의 **Subject** 인터페이스와 관련된 구현을 포함한 **클래스**



자바 내장 옵저버 패턴

- **java.util.Observable** (Subject 인터페이스를 구현한 클래스와 유사)
- **java.util.Observer** (인터페이스)



디스플레이 객체가 옵저버가 되는 방법

- **java.util.Observer** 인터페이스를 구현
- **java.util.Observable** 객체의 **addObserver()** 메소드를 호출

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

public class CurrentConditionsDisplay implements Observer, DisplayElement {
    Observable observable;
    private float temperature;
    private float humidity;

    public CurrentConditionsDisplay(Observable observable) {
        this.observable = observable;
        observable.addObserver(this);
    }
    ...
}
```

Observable 인스턴스인 WeatherData 객체에서 옵저버에게 연락을 돌리는 방법

java.util.Observable 클래스를 확장하여, 상태 정보를 가지는 WeatherData
서브클래스를 정의한 상태를 가정

```
import java.util.Observable;

public class WeatherData extends Observable {
    // 상태 정보
    private float temperature;
    private float humidity;
    private float pressure;

    ...

}
```

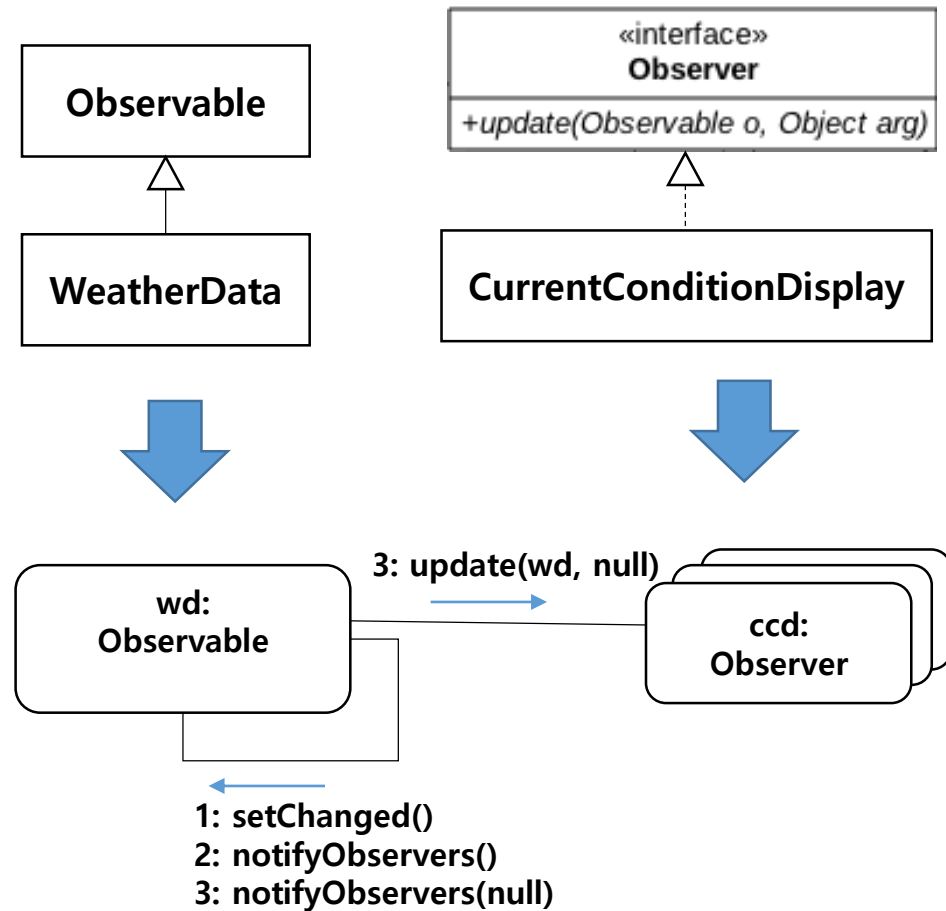
Observable 인스턴스인 WeatherData 객체에서 옵저버에게 연락을 돌리는 방법

1. **Observable** 객체의 상태가 변경되었다는 것을 알리기 위해 `setChanged()` (protected method) 호출
2. 그 다음으로, `notifyObservers()` 혹은 `notifyObservers(Object arg)` 호출

```
public class WeatherData extends Observable {  
    // 상태정보  
    ...  
    public void measurementsChanged() {  
        setChanged();  
        notifyObservers();  
    }  
  
    public void setMeasurements(float temperature, float humidity, float pressure) {  
        this.temperature = temperature;  
        this.humidity = humidity;  
        this.pressure = pressure;  
        measurementsChanged();  
    }  
}
```

무대 뒤에서

```
class Observable {  
    ...  
    setChanged() {  
        changed = true  
    }  
    notifyObservers(Object arg) {  
        if (changed) {  
            for all observers {  
                update(this, arg)  
            }  
            changed = false  
        }  
    }  
    notifyObservers() {  
        notifyObservers(null)  
    }  
    ...  
}
```



[중요] setChanged() 메소드를 호출해 주어야, notifyObservers() 가 호출되었을 때, 등록된 옵저버들에게 연락을 취하게 됨

setChanged() 메소드는 왜 필요할까요?

- Observable 객체의 상태 변화에 따라 옵저버들을 갱신하는 방법의 유연성을 위해서 제공된 것임

```
public void setMeasurements(float temperature, ...) {  
    this.temperature = temperature;  
    ...  
    setChanged();  
    notifyObservers();  
}
```

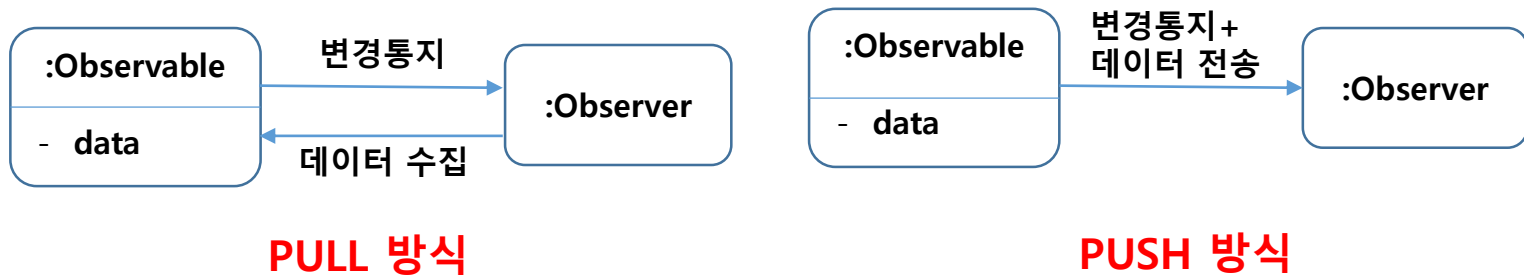
온도가 변할 때마다 옵저버들을 갱신



```
public void setMeasurements(float temperature, ...) {  
    if (Math.abs(temperature - this.temperature) > 0.5)  
        setChanged();  
    this.temperature = temperature;  
    ...  
    notifyObservers();  
}
```

온도차이가 0.5도 이상으로 변할 때마다 옵저버들을 갱신

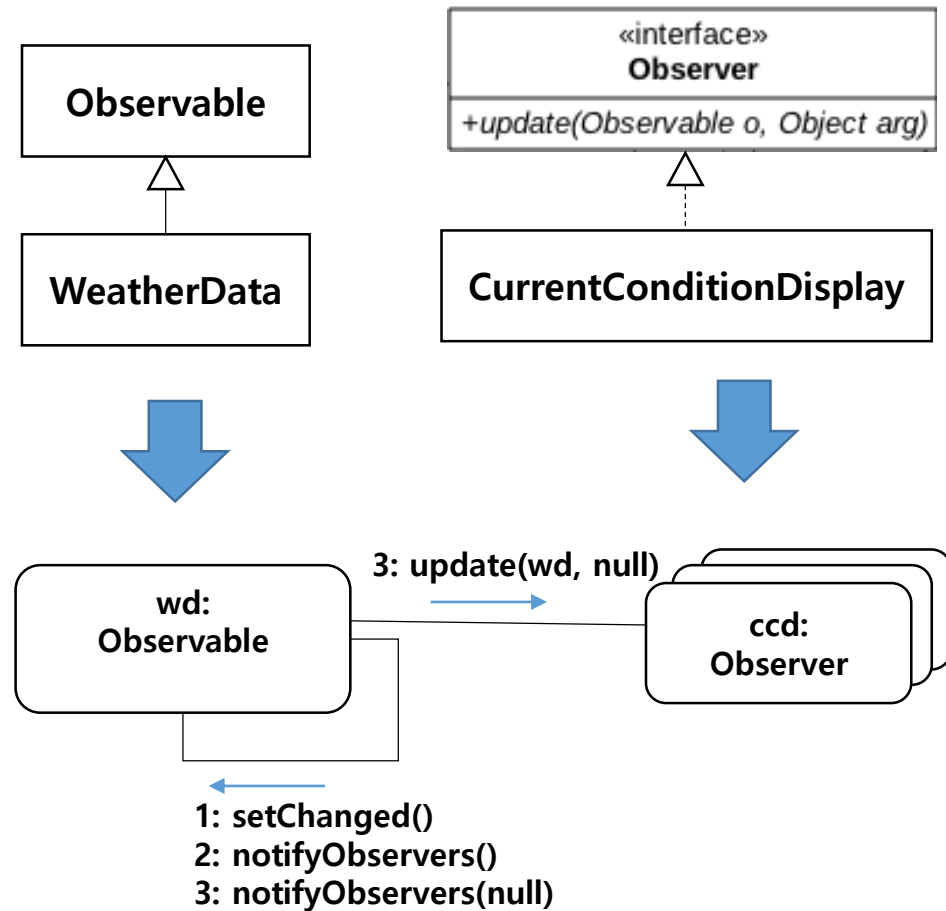
옵저버에게 연락을 돌리는 방법: PULL vs. PUSH



- **notifyObservers()** ← Pull 방식으로 옵저버들이 데이터를 가져감
- **notifyObservers(Object arg)** ← Push 방식으로 데이터를 옵저버들에게 전달

무대 뒤에서

```
class Observable {  
    ...  
    setChanged() {  
        changed = true  
    }  
    notifyObservers(Object arg) {  
        if (changed) {  
            for all observers {  
                update(this, arg)  
            }  
            changed = false  
        }  
    }  
    notifyObservers() {  
        notifyObservers(null)  
    }  
    ...  
}
```



[중요] `setChanged()` 메소드를 호출해 주어야, `notifyObservers()` 가 호출되었을 때, 등록된 옵저버들에게 연락을 취하게 됨

Display 구현 (Pull 방식)

```
public class CurrentConditionsDisplay implements Observer, DisplayElement {
    Observable observable;
    private float temperature;
    private float humidity;

    public CurrentConditionsDisplay(Observable observable) {
        this.observable = observable;
        observable.addObserver(this);
    }

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

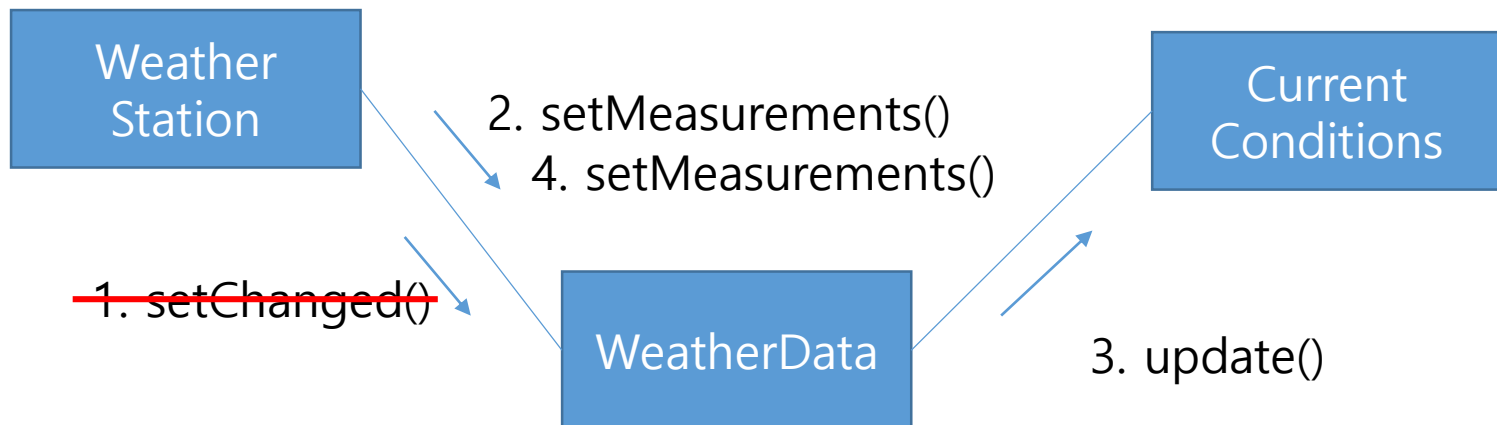
    public void display() {...}
}
```

토의

- Pull 방식과 Push 방식의 장단점은?
- 앞서 예시한 옵저버 패턴 구현 예제(기상 스테이션)에서 Subject가 Observer에게 연락을 돌리는 방식은 Push 혹은 Pull 중에 어떠한 방식인가?

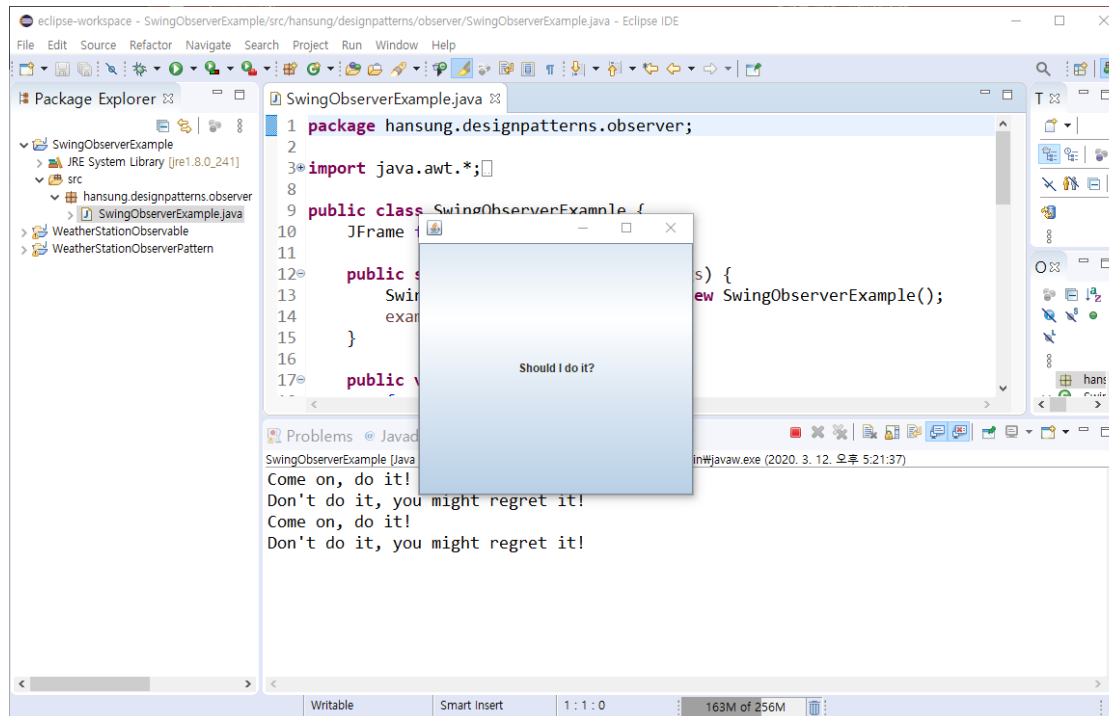
java.util.Observable의 단점

- Observable은 클래스이므로, 다른 슈퍼클래스를 확장하고 있는 클래스에 Observable 기능을 확장할 수 없음.
- Observable의 핵심 메소드 (setChanged()) 메소드는 protected 멤버임)를 외부에서 호출할 수 없음.



JDK에서 옵저버 패턴을 사용하는 부분

- 다음 간단한 Java Swing 애플리케이션을 실행해 봅시다.
 - <https://github.com/kwanulee/DesignPattern/tree/master/observer/SwingObserverExample>



다음 코드에서 옵저버 패턴을 쓰는 부분은?

```
public class SwingObserverExample {
    JFrame frame;
    public static void main(String[] args) {
        SwingObserverExample example = new SwingObserverExample();
        example.go();
    }

    public void go() {
        frame = new JFrame();
        JButton button = new JButton("Should I do it?");
        button.addActionListener(new AngelListener());
        button.addActionListener(new DevilListener());

        frame.getContentPane().add(BorderLayout.CENTER, button);
        // Set frame properties
        ....
    }
    class AngelListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            System.out.println("Don't do it, you might regret it!");
        }
    }

    class DevilListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            System.out.println("Come on, do it!");
        }
    }
}
```


다음 코드에서 옵저버 패턴을 쓰는 부분은?

```
public class SwingObserverExample {
    JFrame frame;
    public static void main(String[] args) {
        SwingObserverExample example = new SwingObserverExample();
        example.go();
    }

    public void go() {
        frame = new JFrame();
        JButton button = new JButton("Should I do it?");
        button.addActionListener(new AngelListener());
        button.addActionListener(new DevilListener());

        frame.getContentPane().add(BorderLayout.CENTER, button);
        // Set frame properties
        ....
    }

    class AngelListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            System.out.println("Don't do it, you might regret it!");
        }
    }

    class DevilListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            System.out.println("Come on, do it!");
        }
    }
}
```

핵심 정리

- 옵저버 패턴에서는 객체들 사이에 일대다 관계를 정의
- Subject 또는 Observable은 동일한 인터페이스를 써서 Observer에 연락을 취함
- Observable에서는 옵저버들이 Observer 인터페이스를 구현한다는 것을 제외하면 옵저버에 대해서 전혀 모르기 때문에, 이들 사이의 결합은 느슨한 결합이다.
- 옵저버 패턴을 이용하며 Subject 객체에서 데이터를 보내거나 (푸시방식) 가져오는(풀 방식)을 쓸수 있다.
- 스윙 및 여러 GUI 프레임워크에서 옵저버 패턴이 많이 쓰인다.