The Observer Design Patterns

Learning outcomes

- Understand the publisher/subscriber mechanism in the implementation of Observer pattern.
- Identify the steps involved in implementing observer and observable interfaces
- Understand java built in functionality for the observer pattern
- Highlight how observer pattern can help in achieving loose coupling.

Observer Design: Customer-Store

Imagine that you have two types of objects: a **Customer** and a **Store**. The **customer** is interested in a particular brand of product (eg. a new model of the iPhone) which should become available in the store very soon.

- **Solution 1:** The customer could visit the store every day and check product availability.

Observer Design: Customer-Store

Imagine that you have two types of objects: a **Customer** and a **Store**. The **customer** is interested in a particular brand of product (eg. a new model of the iPhone) which should become available in the store very soon.

- **Solution 2:** Send emails to all customers each time a new product becomes available.

Observer Design: Customer-Store

Imagine that you have two types of objects: a **Customer** and a **Store**. The **customer** is interested in a particular brand of product (eg. a new model of the iPhone) which should become available in the store very soon.

- **Solution 3:** Observer design pattern.

Observer Design

- Used for building event-driven software:
  - User input devices such as a game controller
  - IoT sensor devices that collect and report information on an event basis.
  - Business rules. For example, if a bank customer transfers more than £10,000 between accounts a security service checks if that behavior is typical for the customer.
  - Service provisioning process for a cloud server. For example, milestones in a process can trigger events.
  - Workflow management systems
  - etc.

Example

- A publisher creates a new magazine and begins publishing issues
  - You subscribe and receive issues as long as you stay subscribed.
  - You can unsubscribe at any time
  - Others can also subscribe
  - If publisher ceases business, you stop receiving issues

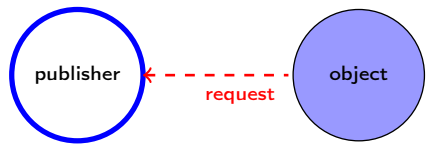
Example: Publishers and Subscribers

- Think about a publisher and the subscribers as sets of objects.

Example: Publishers and Subscribers

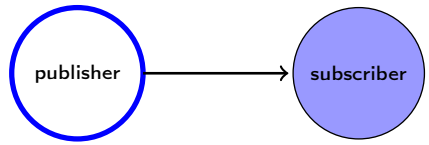
- Start with a publisher object to which any other object can send a request to subscribe.
- The publisher typically holds some type of data of interest (e.g stock quote, weather info., etc).
- When the data changes, the subscribers are notified.

Example: Publishers and Subscribers



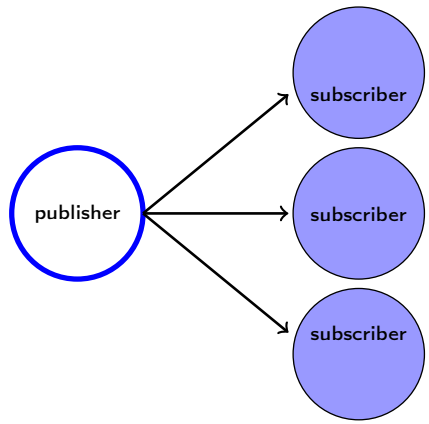
- A request is received by the publisher from a requesting object.

Example: Publishers and Subscribers



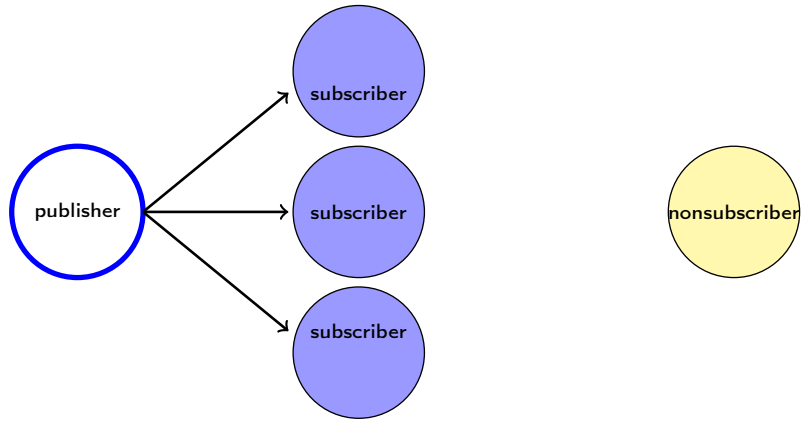
- When the request is received by the publisher, the requesting object is immediately becomes a subscriber.

Example: Publishers and Subscribers



- Any object can ask to be a subscriber.
- Also, you can have more than one subscriber.

Example: Publishers and Subscribers



- There will be objects that are not subscribers.

Example: Publishers and Subscribers

- How do we create a pattern around this idea?
- How can we implement it and explore its benefits?

The Observer Pattern

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

- **Also Known As** Dependents, Publish-Subscribe, Model-View pattern
- **Motivation:** The need to maintain consistency between related objects without making classes tightly coupled.

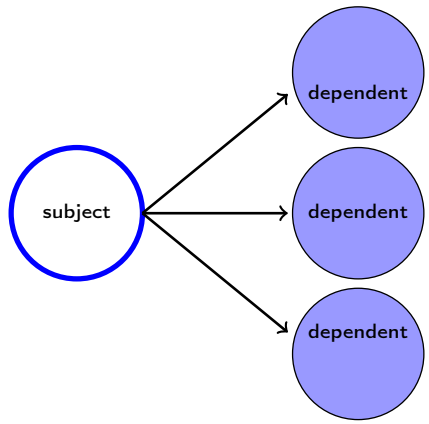
Observer Design Pattern

Design challenge:

- How can we achieve a one-to-many dependency between objects (i.e a publisher with many subscribers) without making the objects tightly coupled.
- How can we ensure that when one object changes state an open-ended number of dependent objects are updated automatically.

The Observer Pattern

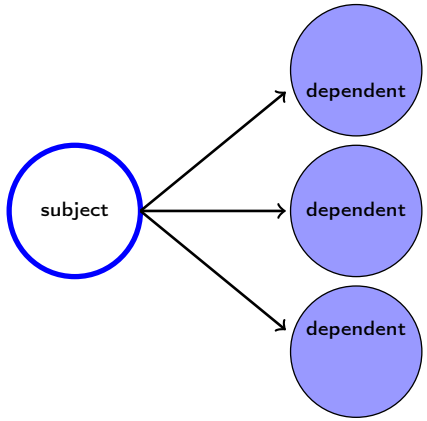
Defines a one-to-many relationship between a set of objects



- If the state changes in the subject, then the many dependents are notified of that state change.
- The dependent is the subscriber, or more commonly, the observer (the dependent relies on the subject for data).

The Observer Pattern

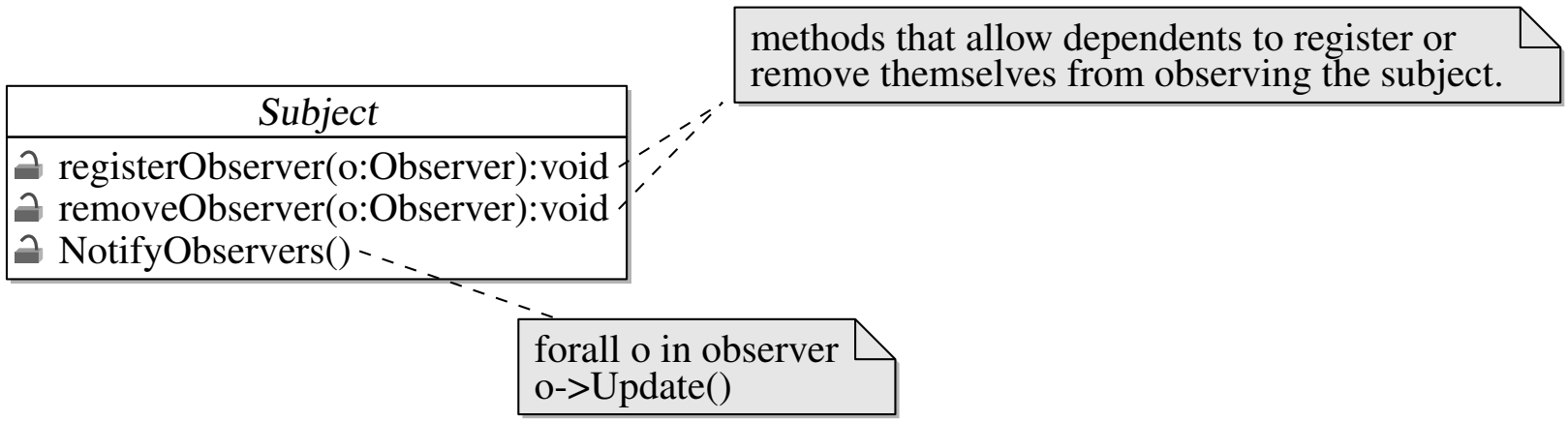
Defines a one-to-many relationship between a set of objects



- The subject owns the data:
  - That means there's only one copy of 'data', and the subject is the sole owner.
  - Thus, we end up with a design that's cleaner than many objects owning the same data.
- The dependents get notified when the subject 'data' changes.

Using Observer Pattern

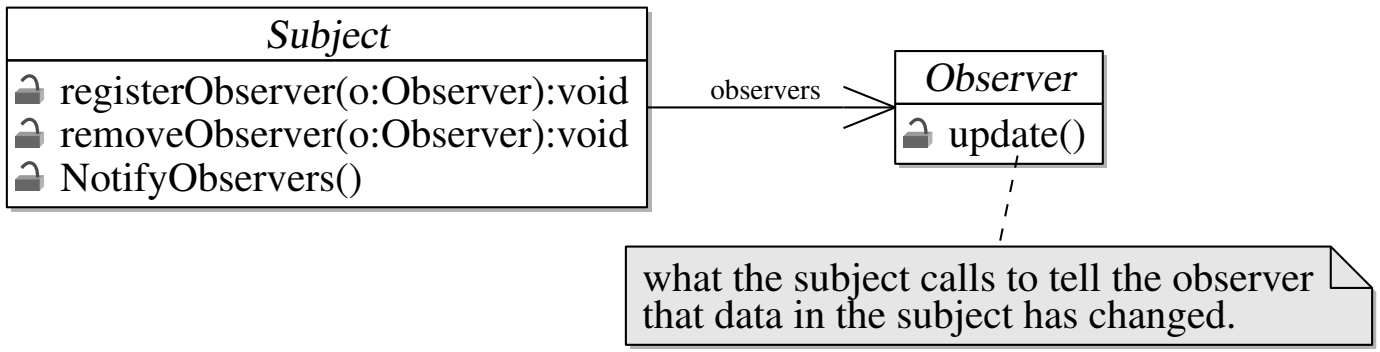
Step 1:



- Define the Subject interface

Using Observer Pattern

Step 2:

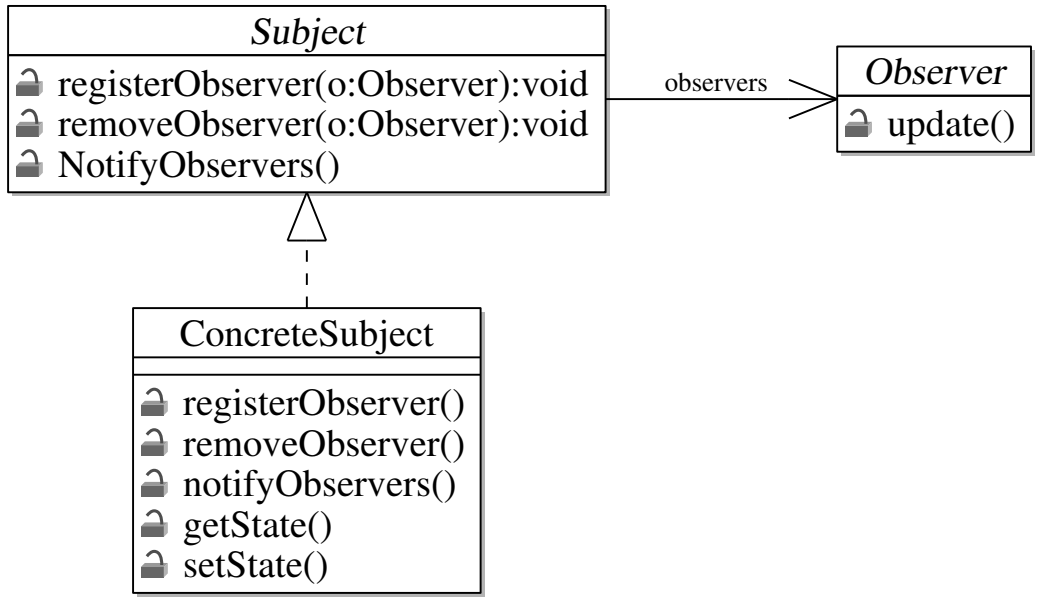


a. Define the Observer interface to communicate with the Subject

b. Define the **data communication protocol** .
This involves one of two design choices. Either:
   1. the observer is sent a new value as part of the update call; or
   2. the observer explicitly ask the subject for the new value.

Using Observer Pattern

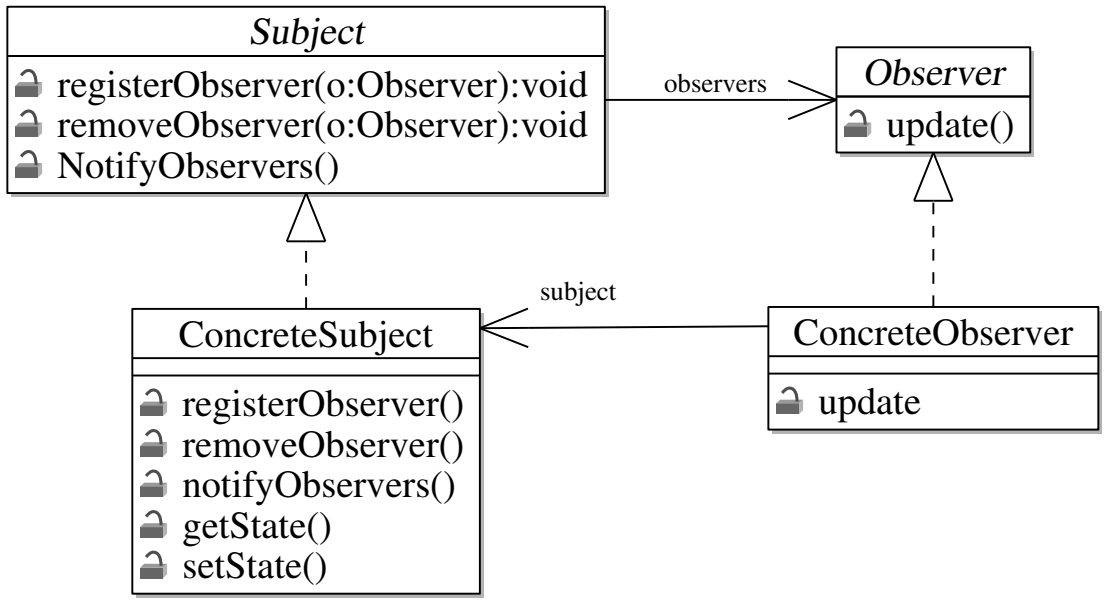
Step 3:



- The *ConcreteSubject* implements the *notifyObservers* method.
  - *notifyObservers* is called any time data changes
  - *notifyObservers* calls the *update* method on each observer
- It's common to implement getters and setters in the subject to get and set the state.

Using Observer Pattern

Step 4:



- The concrete observer is any class that implements the observable interface.
  - Implemented observable interface contains the update method and that is called by the subject as necessary.

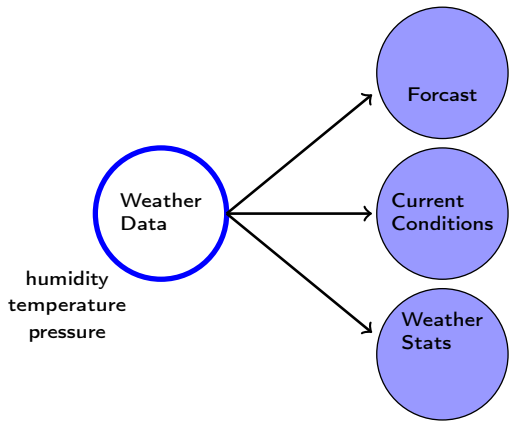
Observer Pattern - Example

The Weather Station

A weather station consists of a *weather data* class that can read weather information like the humidity, the temperature, and the pressure; and *display* classes that displays the weather data in various configurations, like the current conditions, the forecast, and statistics about the weather (e.g maximum and minimum temperature of the day). Each time the weather data object gets new weather data, it needs to update each of the displays.

Observer Pattern - Example

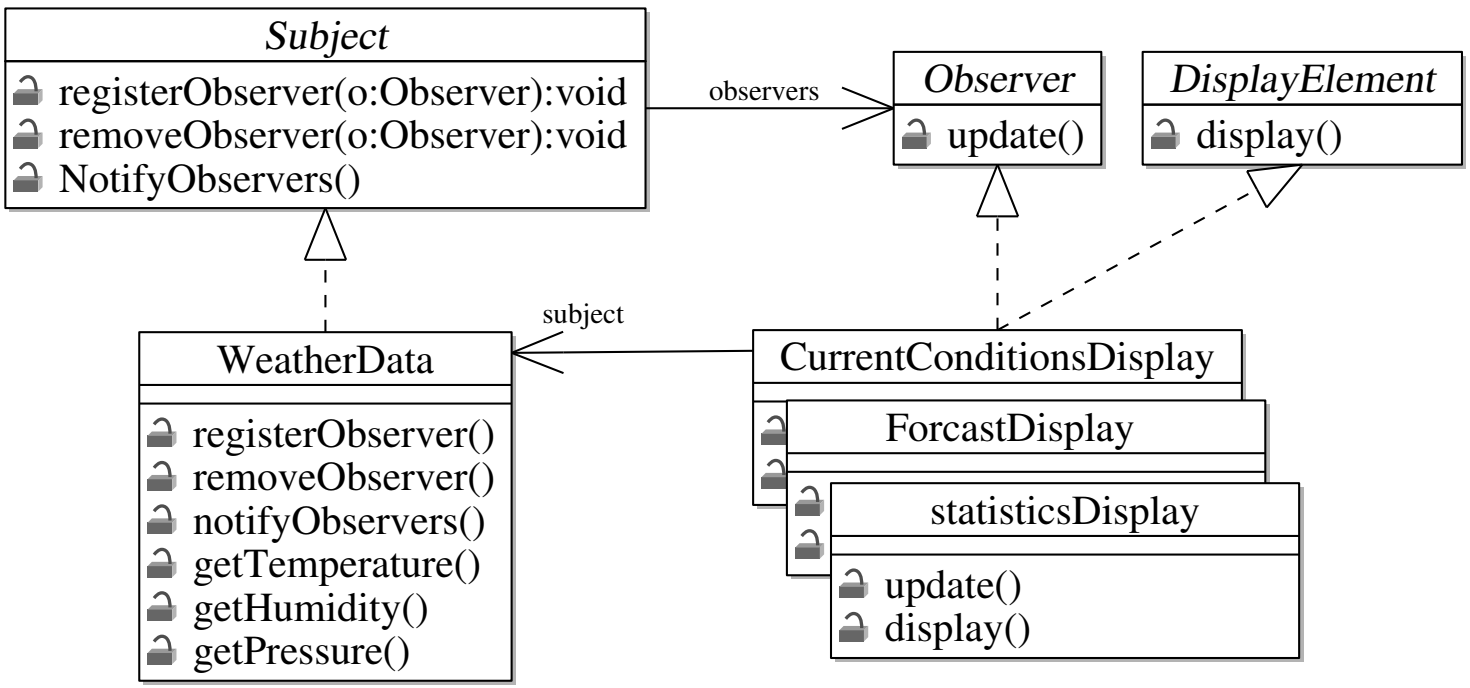
The Weather Station



The weather data class is the subject and the displays are the observers.

Observer Pattern - Example

The Weather Station



Example: The Weather Station

- Custom Observer/Observable

Java Observer and Observable Classes

- Java offers built in support for the observer pattern with the **Observable** class and the **Observer** interface, available in the **java.util** package.

Java Observer and Observable Classes

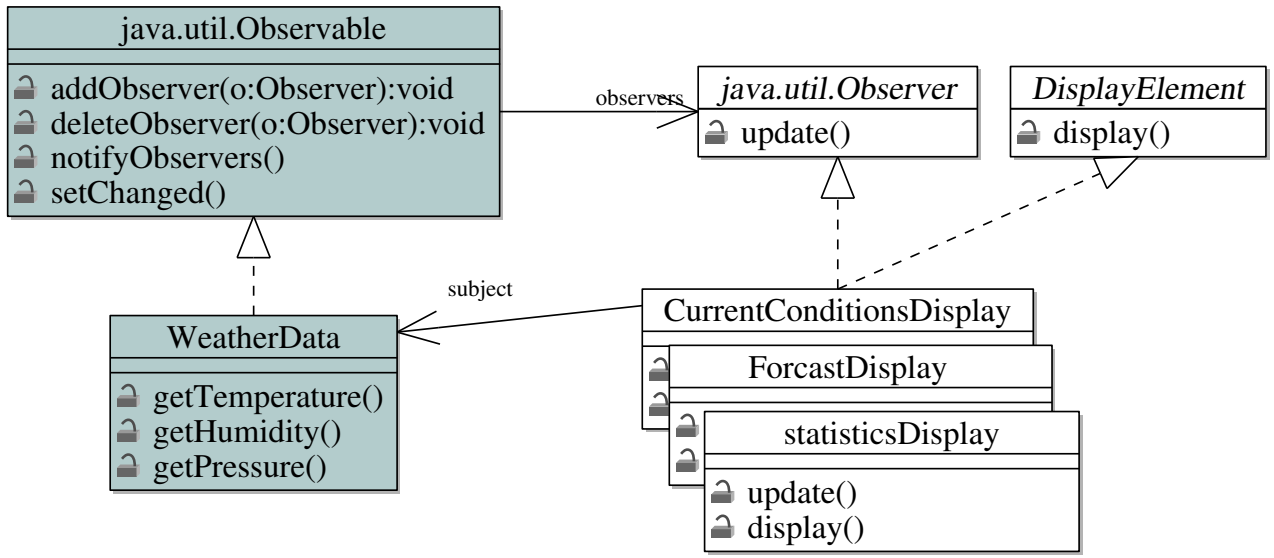
- **Observable class**
  - **java.util.Observable** class is analogous to the subject interface. To implement your own subject, you extend this class and inherit methods to manage the Observers.
  - Follows a two step process to notify observers

Java Observer and Observable Classes

- **Observer interface**
  - **java.util.Observer** interface is same as our previous Observer interface.
  - Observers can either **pull** data from the Subject or the Subject can **push** data to the Observers.

Weather Station: Using Java's built-in Observer Pattern

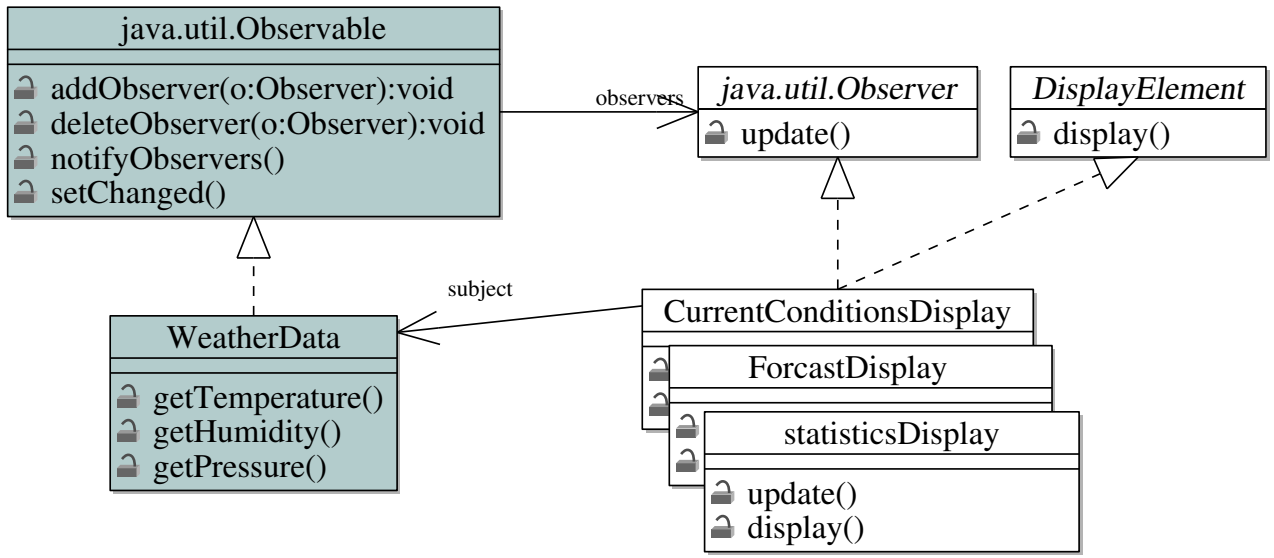
Revised class diagram



- The subject (**Observable**):
  - is now a class, rather than an interface, and implements the methods to add, delete, and notify observers.
  - Also implements a method `setChanged()` which is used in the notification process.

Weather Station: Using Java's built-in Observer Pattern

Revised class diagram



- The WeatherData class:
  - Now only implements the getters for the various weather data fields, as well as the set Measurements method

Weather Station: Using Java's built-in Observer Pattern

Two Steps to Notify Observers

1. Call setChanged() method
   - This indicates that the state in the subject has changed.

Weather Station: Using Java's built-in Observer Pattern

Two Steps to Notify Observers

2. Call notifyObservers()
   - for pull: **notifyObservers()**
     The observers to pull the updated data from the subject using the getter methods.
   - for push: **notifyObservers(Object arg)**
     The object **arg** contains the updated data

Weather Station: Using Java's built-in Observer Pattern

To Receive Notifications

- The update method in the observers is called

  update(Observable o, Object arg)

  - **Observable o**: is the subject, or observable, that is updating the observers.
  - **Object arg**: is the object that encapsulates the changed data (only useful when pushing data to observers)

Example: The Weather Station

- Built in Observer/Observable

Observer Pattern: Loose Coupling

- Subjects and observers are "loosely coupled"

  They interact, but have little knowledge of each other

  - **The subject only knows** that the observer implements a specific interface, and does not know (or need to know) concrete class or anything else

Observer Pattern: Loose Coupling

- Subjects and observers are "loosely coupled"

  They interact, but have little knowledge of each other

  - All the subject knows is that it has a list of objects that implement the observable interface.
  - The observers can **add**, **remove** or **replace** themselves from the list at anytime.

Observer Pattern: Loose Coupling

- Subjects and observers are "loosely coupled"

  They interact, but have little knowledge of each other

  - If you want to add new types of observers, modification of the subject is never needed.
  - Observers just need to implement the Observer interface
  - No changes to the subject or observers will affect the other

Object Oriented Design Principles

Design Principle:

Strive for loosely coupled designs between objects that interact.

- Object interdependencies are minimised
- Build more flexible OO architectures

Summary

Key:

- Loose coupling is an important OO design principle.
- loose coupling gives us the flexibility to vary our design without breaking the contract between interacting objects.

Disadvantage:

- The observer design pattern can cause memory leaks, known as the **lapsed listener problem**