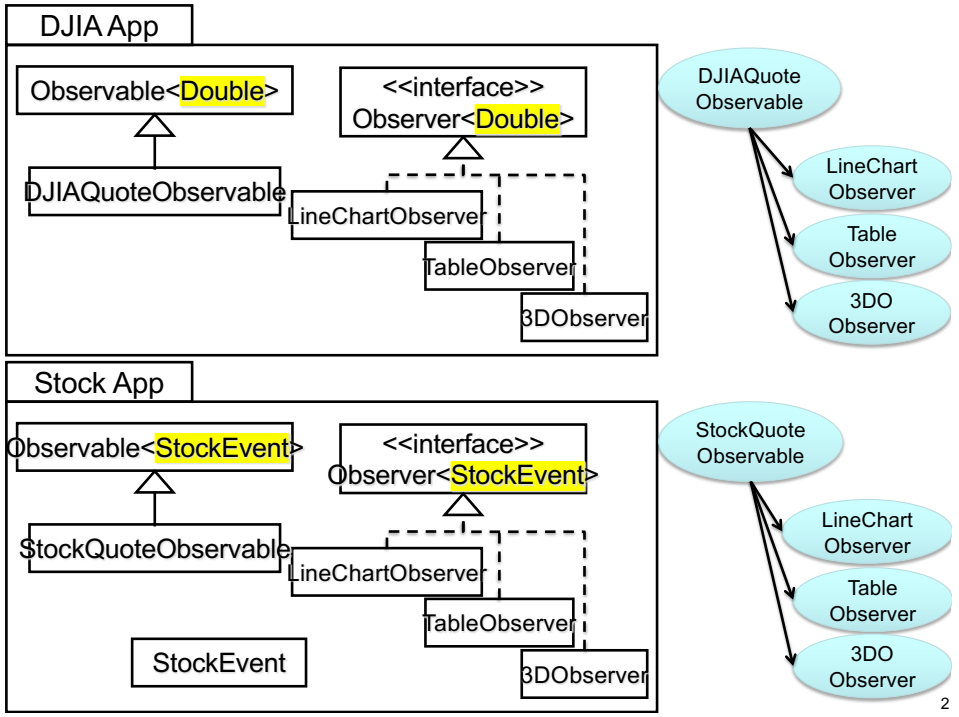
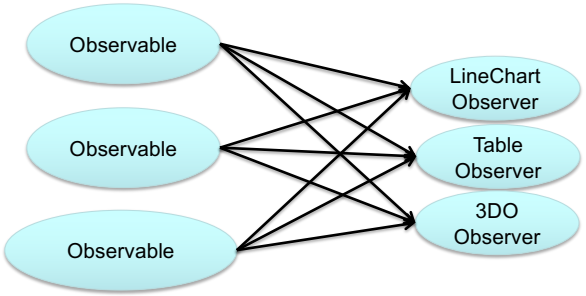
One-to-many Event Notification

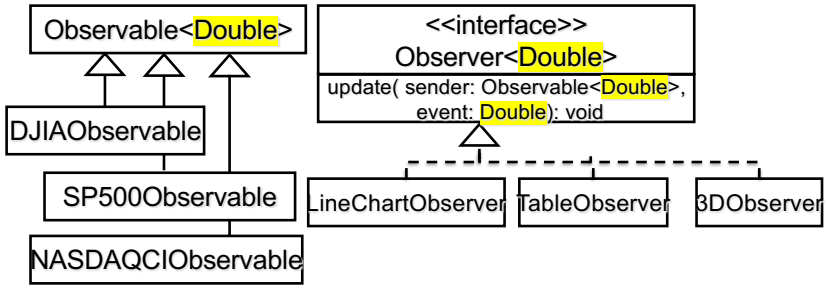
- *Observer* works well to design one-to-many event notification.

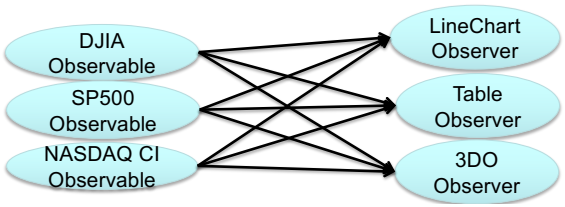


What about Many-to-many Event Notification?

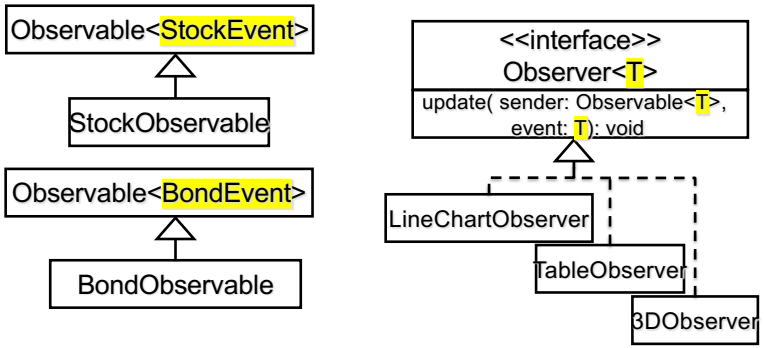


- Many-to-many event notification is possible as far as observables send out the same type of events.
  - Need to use conditionals in update() though.





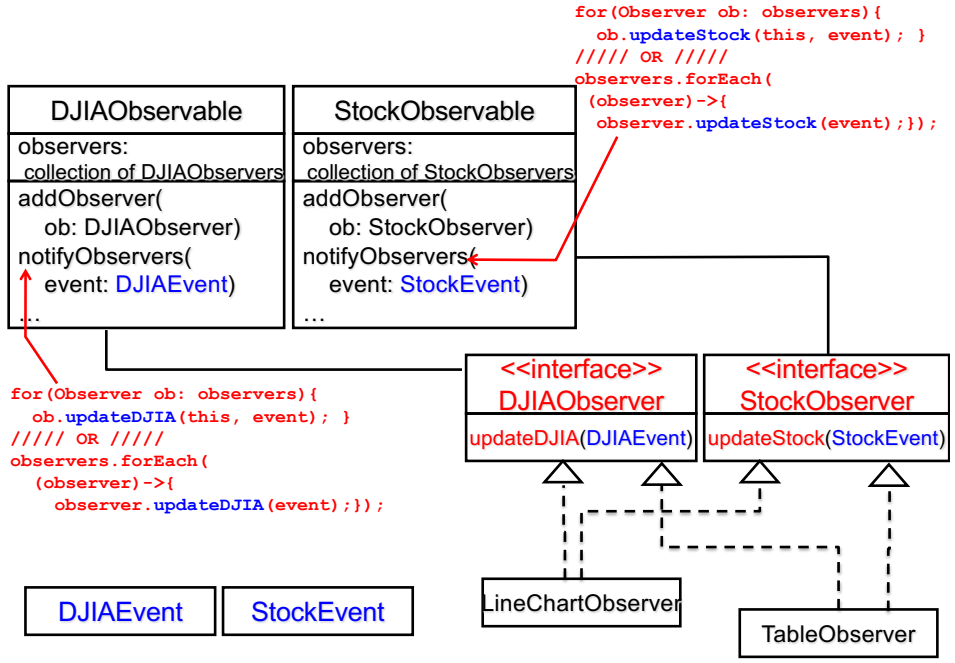
- Many-to-many event notification is NOT possible if observables send out different types of events.
  - If an observer is designed to
    - receive **`StockEvent`**s, it cannot receive **`BondEvent`**s.
    - receive **`BondEvent`**s, it cannot receive **`StockEvent`**s.



Multicast Design Pattern

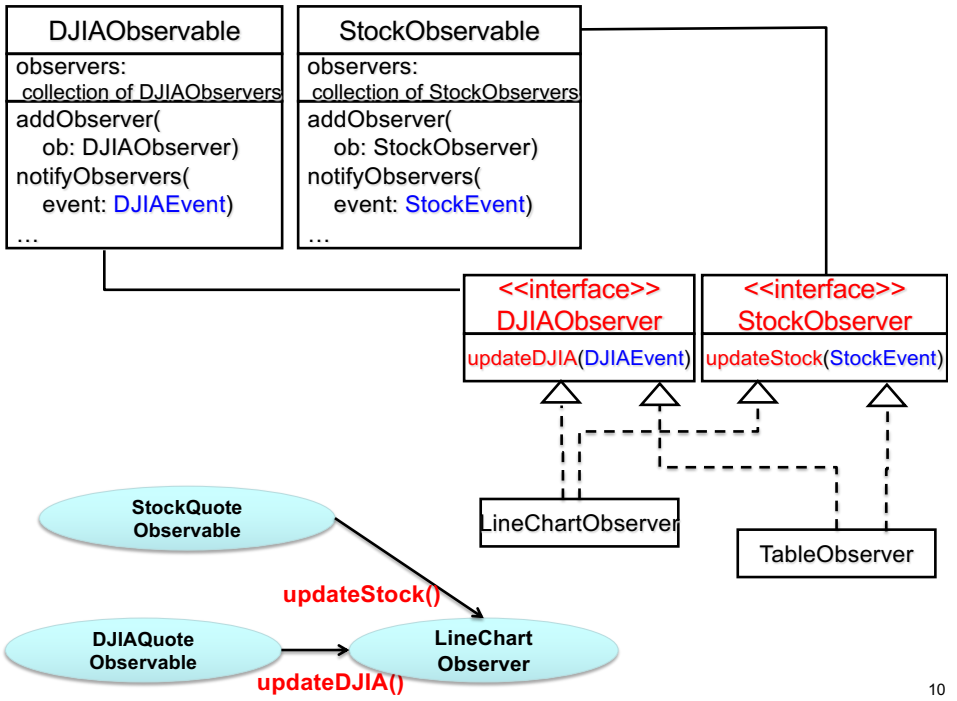
Multicast Design Pattern

- Intent
  - Same as the intent of *Observer*
  - Allows for "many-to-many" event notification



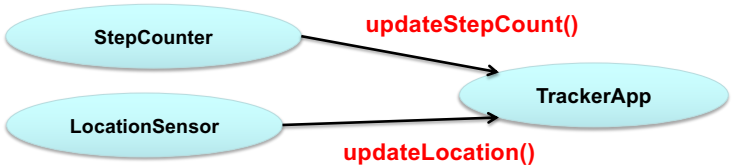
Points

- The `Observer` interface no longer exists.
- A pair of an observable class and an observer interface is defined for each event.
  - e.g., `DJIAObservable` and `DJIAObserver` for `DJIAEvent`
- Each observer interface has a method specific to an event type.
  - e.g., `DJIAObserver` has `updateDJIA(DJIAEvent ev)`
- Each observer class can implement more than one observer interface,
  - so it can receive multiple types of events.
    - e.g., A `LineChartObserver` can receive `DJIAEvent`s and `StockEvent`s with `updateDJIA()` and `updateStockEvent()`.



HW 5

- Implement the following event notification with *Multicast*.
  - Use 2 observable classes: `StepCounter` and `LocationSensor`.
  - Use 2 observer interfaces: `StepCountObserver` and `LocationObserver`.
  - Use 2 different event classes: `StepCount` and `Location`.
    - You can design these event classes as you like.



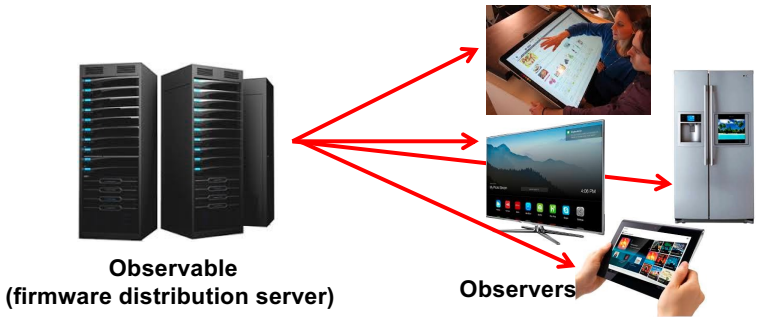
Half-Push/Half-Pull Design Pattern

"Push" and "Pull" in Event Notification

- Push
  - Observer and Multicast
    - Publish-subscribe (pub/sub)
  - Pros:
    - Low workload/traffic from observers to an observable
  - Cons:
    - An observable needs to perform error handling for unavailable (e.g., sleeping, turned-off or dead) observers.
      - An observable assumes that all observers are always available by default.
      - Need to keep track which observers have received events.
- Pull
  - a.k.a. Polling
  - Observers (periodically) contact an observable to collect data/events.
  - Pros:
    - No error handling in an observable regarding unavailable observers.
  - Cons:
    - Potentially huge incoming workload/traffic on an observable.

Half-Push/Half-Pull

- Hybrid of push and pull schemes
  - http://www.hillside.net/plop/2009/papers/Security/Half-push-Half-polling.pdf
- Example scenario
  - Firmware update on home appliances and consumer electronics.



- Pull
  - Each observer contacts an observable when it boots up.
  - If an event (i.e., a firmware update) is available, the observer downloads it from the observable or consults with the user.
  - Pros: No error handling necessary in the observable.
  - Cons: Huge incoming traffic on the observable.
- Push
  - Each observer registers itself to the observable.
  - Whenever an event is available, the observable pushes it to registered observers.
  - Pros: Limited incoming traffic on the observable
  - Cons: Need error handling in the observable. Need to keep track which observers have not responded and which observers have installed which versions of updates.

- Half-push/half-pull
  - Each observer registers itself to an observable.
  - Whenever an event (i.e. firmware update) is available, the observable schedules when it sends out the event to which observers. The observable then *pushes* an "update schedule" to each observer.
    - The observable ignores unavailable observers. No error handling is performed.
  - According to a given schedule, each observer *pulls* an event (i.e., downloads a firmware update) from the observable.
    - Update schedules need to be prepared carefully so that not too many observers contact the observable at the same time.
  - When an observer boots up, it requests an update schedule to the observable.
  - Pros: Modest incoming traffic on the observable. No error handling is necessary on the observable.