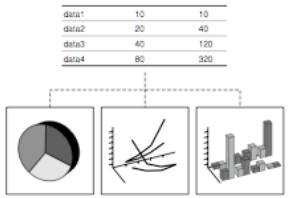
# *Observer* Design Pattern

# *Observer* Design Pattern

- Intent
  - Event notification
    - Define a one-to-many dependency between objects so that, when one object changes its state, all its dependents are notified automatically
- a.k.a
  - Publish-Subscribe (pub/sub)
  - Event source - event listener
- Two key participants (classes/interfaces)
  - Observable (model, publisher or subject)
    - Propagates an event to its dependents (observers) when its state changes.
  - Observer (view and subscriber)
    - Receives events from an observable object.



# Example



**Observable (publisher, event source)**

Registration (subscription) for an event (state change)

Event notification (publication)

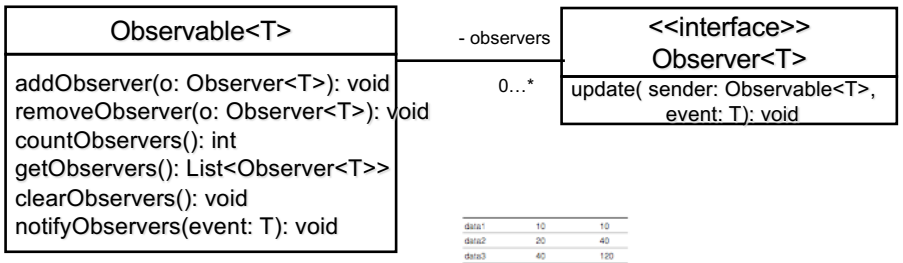
**Observers (subscribers, event listeners)**

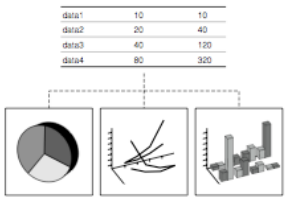
- Separate data processing from data management.
  - Data management: Observable
  - Data processing: Observers
    - e.g., Data analysis (e.g. feature extraction), graphical visualization, etc.



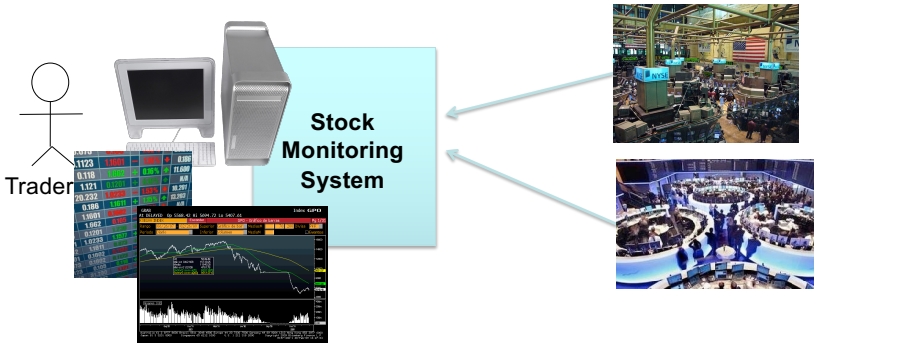
# Simplified Version of MS Azure SDK

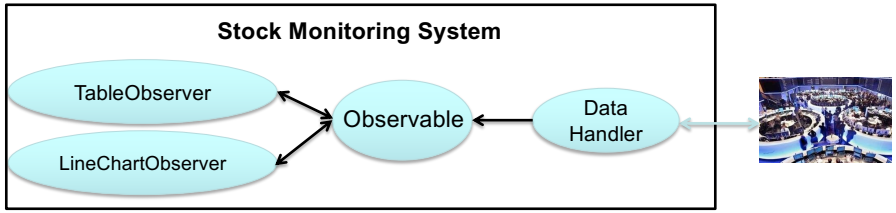
- `umbcs680.observer.Observable`
- `umbcs680.observer.Observer`



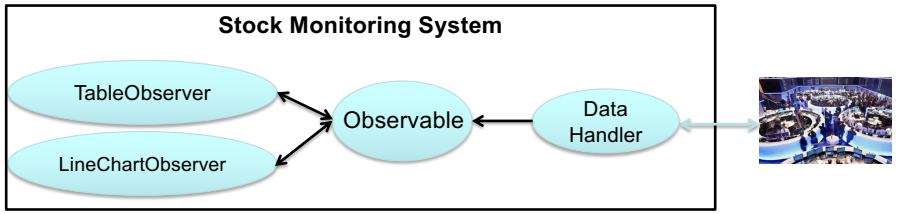



# Example: Stock Price Monitoring



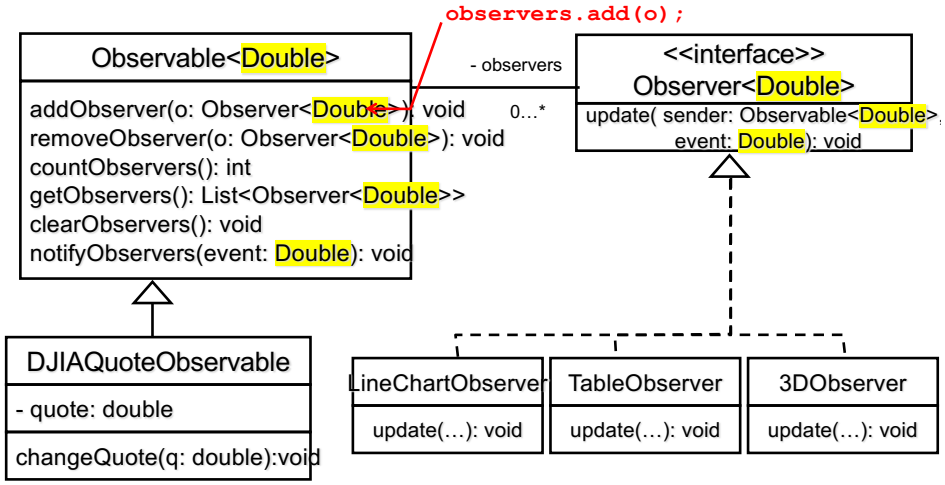



- Data handler: communication and data acquisition
  - Periodically fetches stock data from a stock exchange
  - Supplies the data to an Observable
- Observable: event notification
  - Notifies the data to Observers
- Observer: data visualization
  - Displays the data



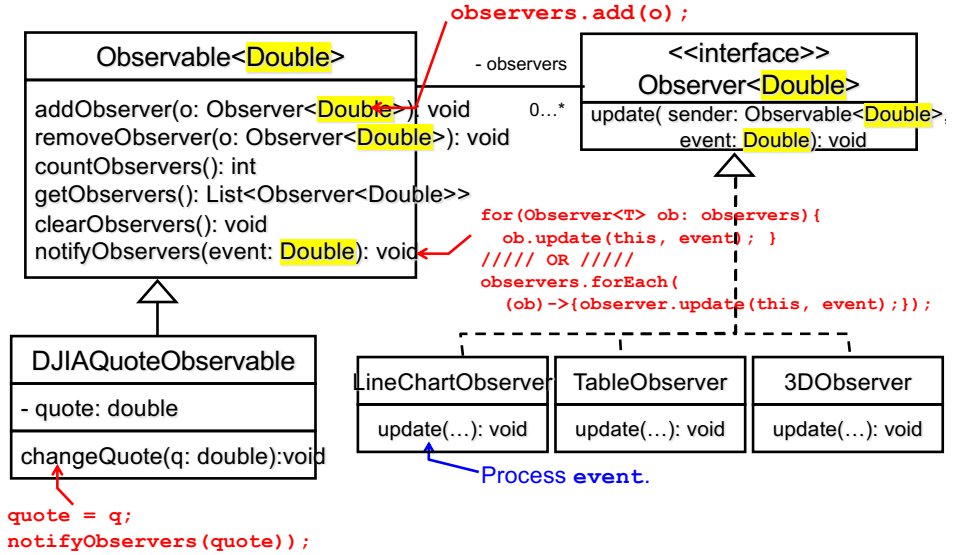
# Registration



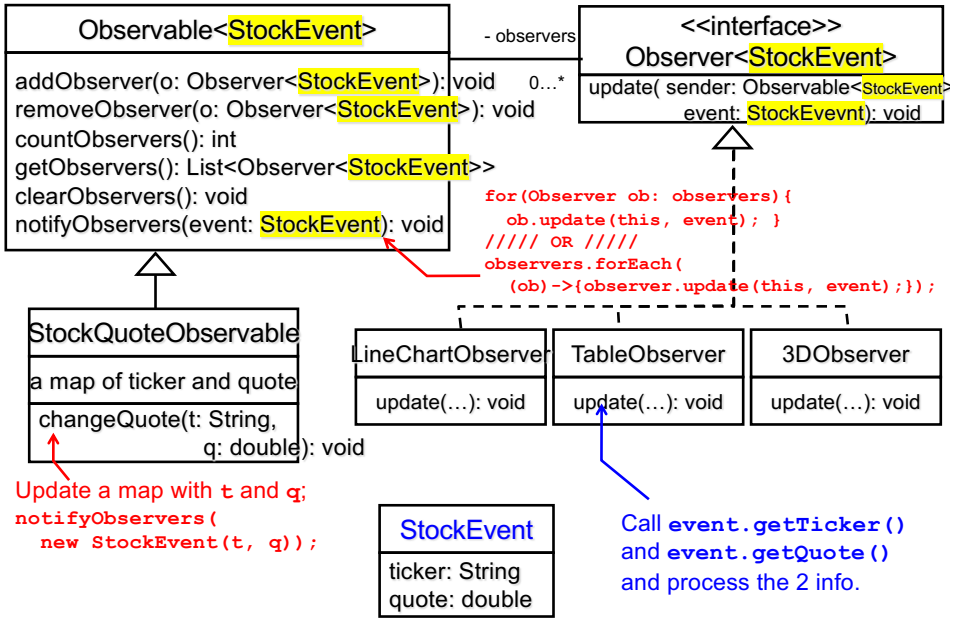

Each `observer` calls `addObserver()` on a `DJIAQuoteObservable`.

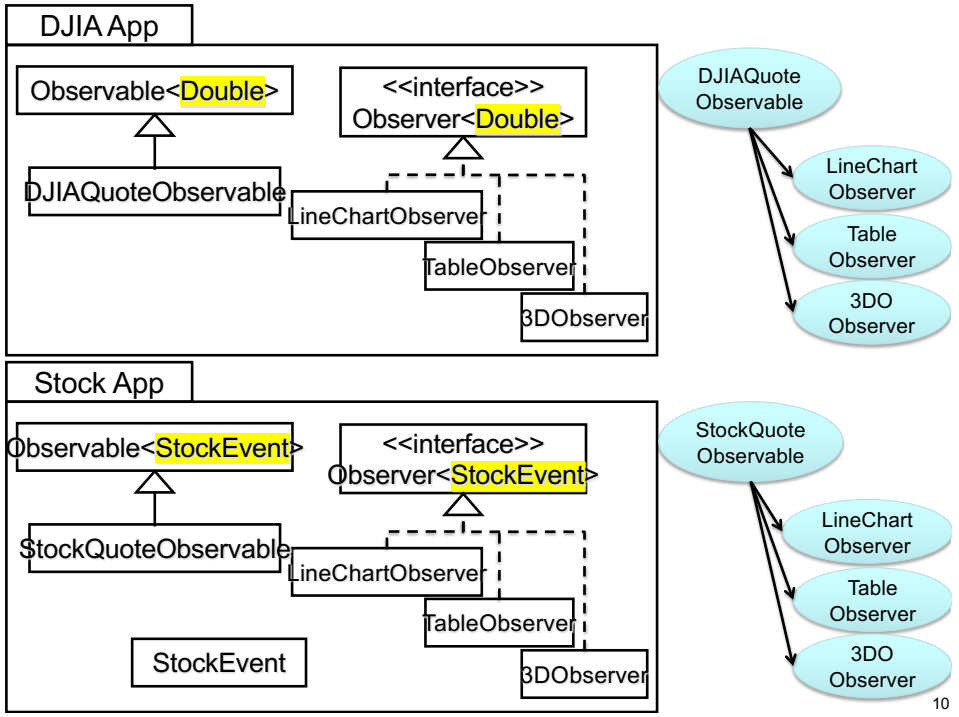


# One-to-Many Event Notification




A `DJIAQuoteObservable` calls `notifyObservers()`.

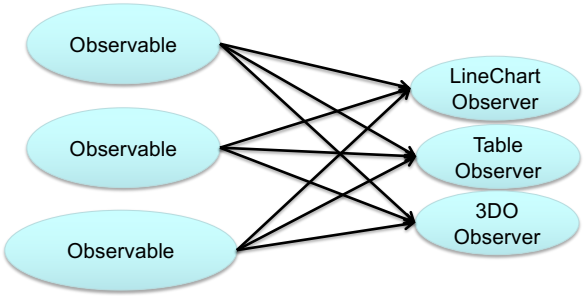




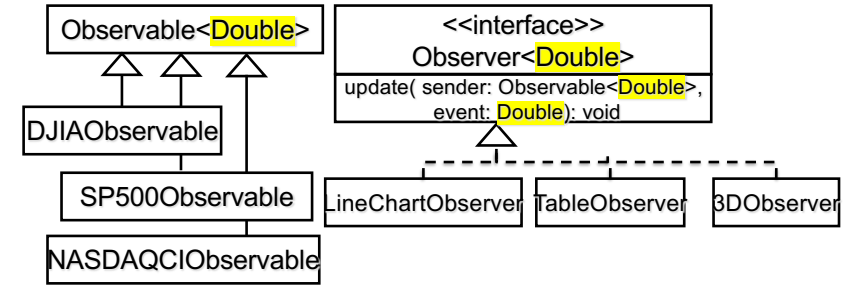

# What about Many-to-many Event Notification?

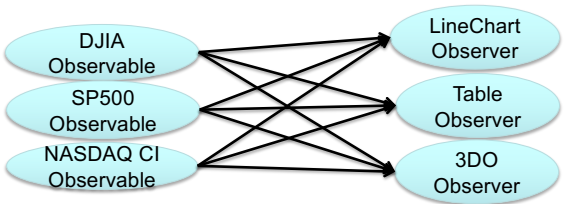
- *Observer* works well to design one-to-many event notification.



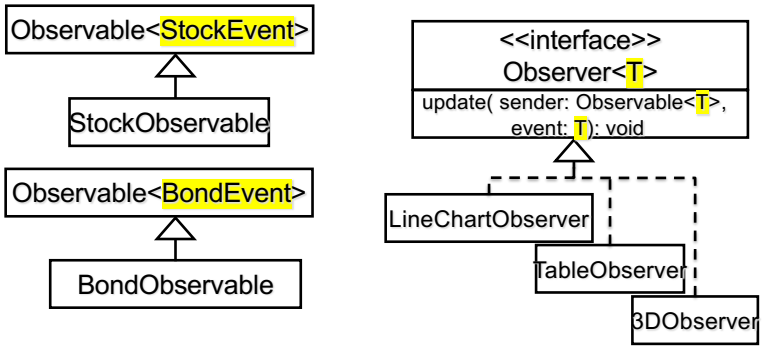



- Many-to-many event notification is possible as far as observables send out the same type of events.
  - Need to use conditionals in update() though.



- 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.





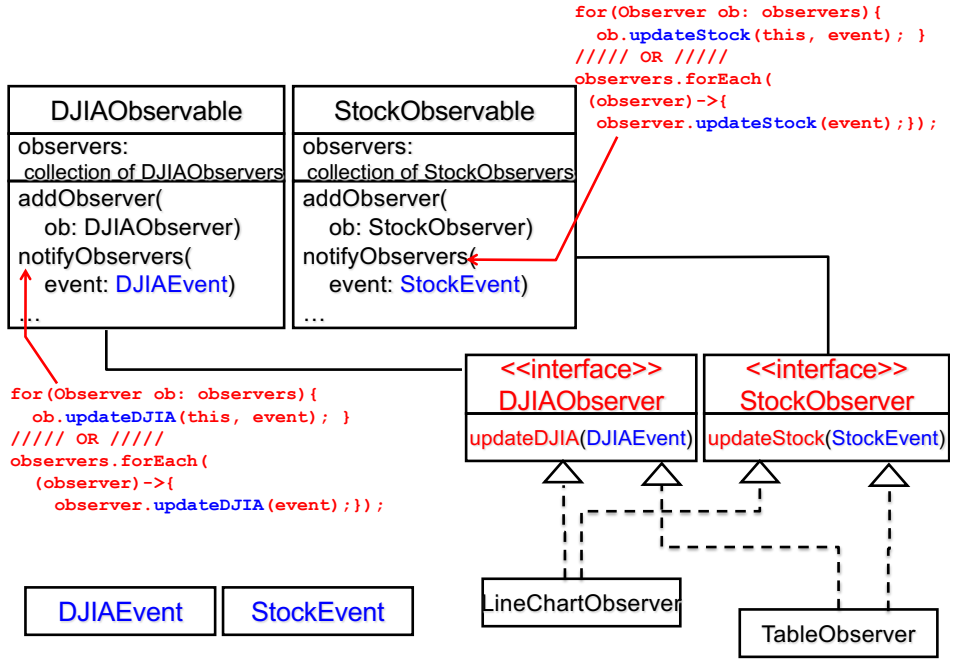

# Multicast Design Pattern



# Multicast Design Pattern

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

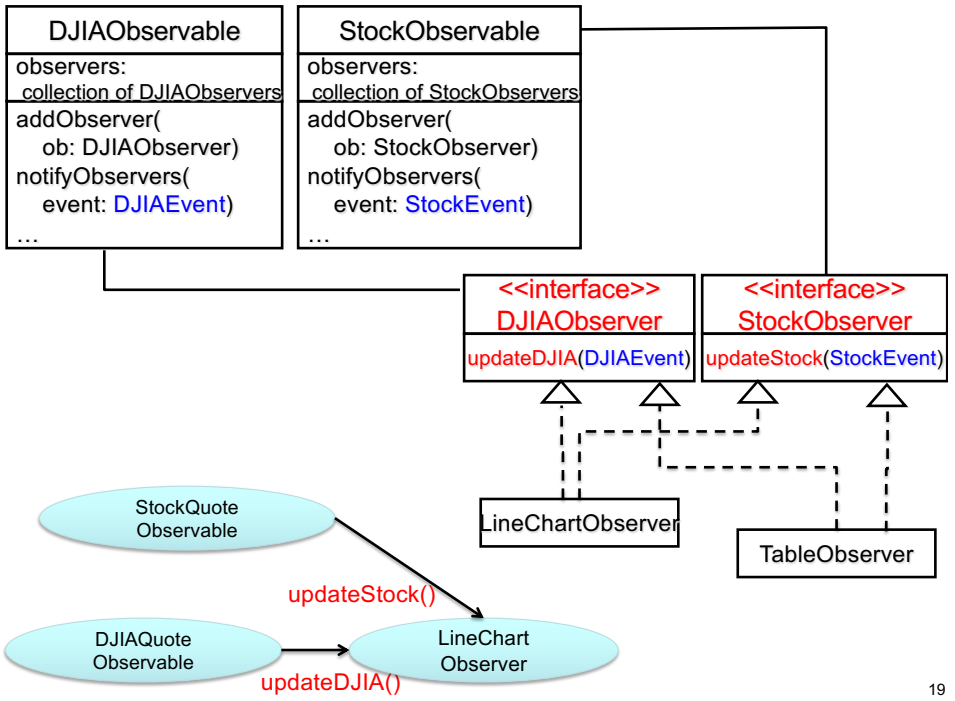




# Points

- 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

- Make YOUR OWN assignment and provide a solution to it.
- Come up with an example of the *Multicast* design pattern in a particular application.
  - Do NOT use an example covered in my lecture notes.
- Implement it yourself with the supplied `Observable` and `Observer`.
  - DO NOT modify them. Keep them as they are, and just "use" them.
- Turn in:
  - Your implementation (code)
  - A short readme.txt file that explains what kind of app you consider and how your code implements the *Observer* design pattern.
  - Test code and an Ant script.
- This is the last one in the first half of HWs.
  - Your HW 1 to 5 solutions will be due at March 17.