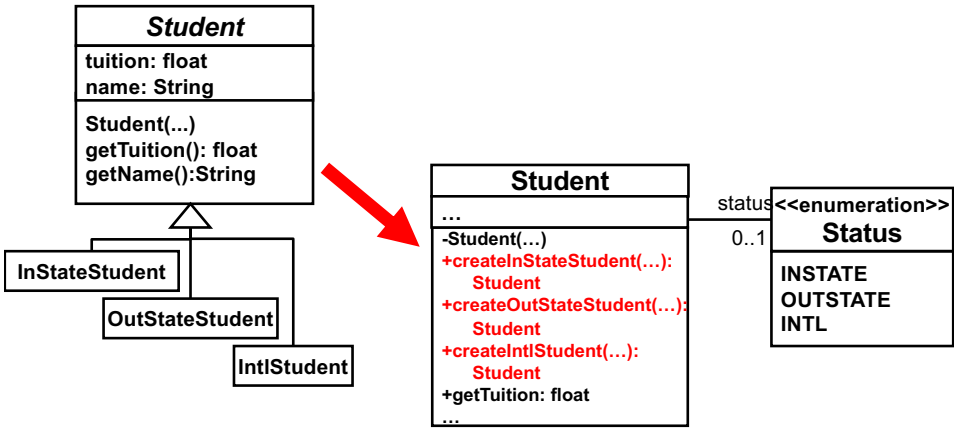
State Design Pattern

State Design Pattern

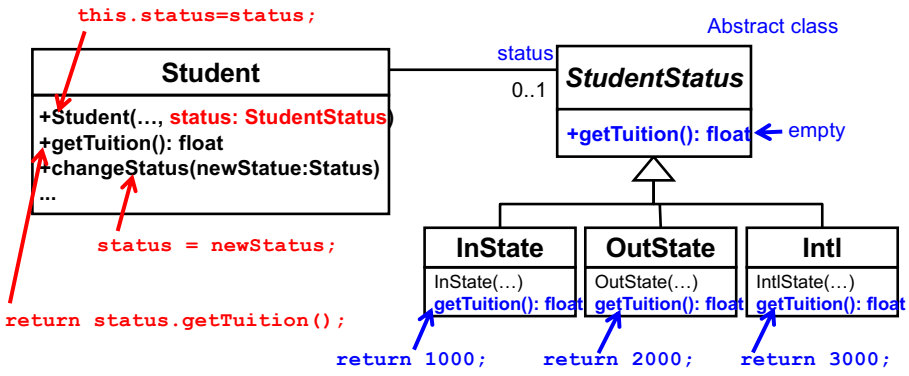
- Intent
  - Allows an object to change its behavior according to its state.
    - Allows an object to perform *state-dependent behaviors*.

An Example



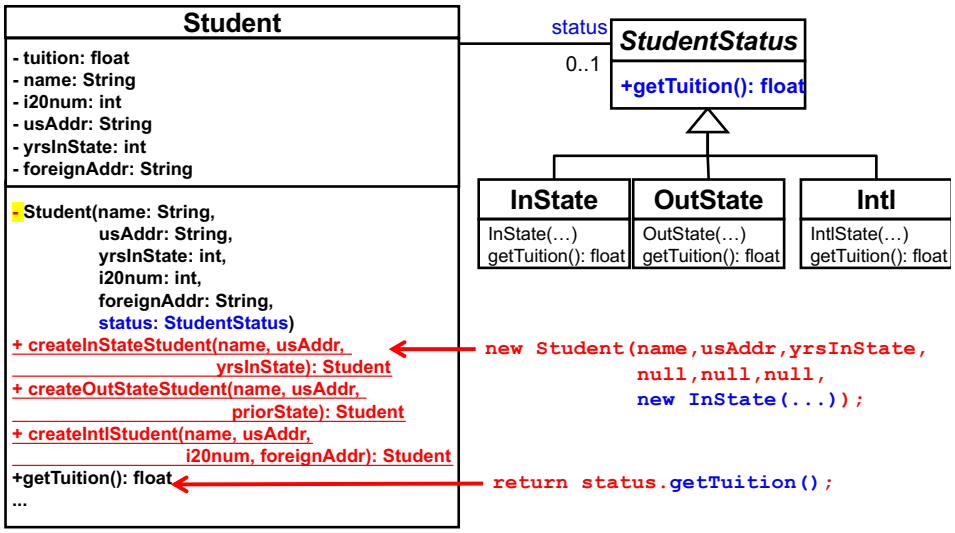
- c.f. previous lecture notes
- Allows each student to change his/her status dynamically
- Needs a conditional in getTuition()
  - Can eliminate the conditional with *State*.

Design Improvement with *State*



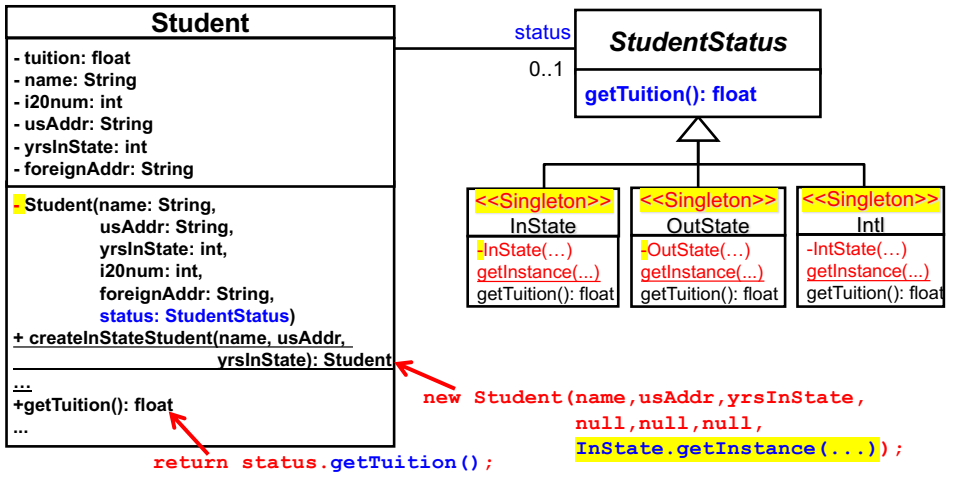

```
Student s1 = new Student( ..., new OutState(...) );
s1.getTuition();
s1.changeStatus(new InState);
s1.getTuition();
```

Adding *Static Factory Methods*




```
Student s1 = Student.createInStateStudent("John Smith",...);
s1.getTuition();
```


State Classes as *Singleton*




```
Student s1 = Student.createInStateStudent("John Smith",...);
s1.getTuition();
```

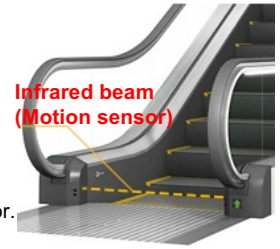

State-dependent Behaviors

- *State* design pattern
  - Allows each student (`Student` class instance) to change his/her behavior (i.e. returning different tuition $) according to his/her status.
    - State-dependent behavior: tuition calculation
- Benefits
  - Allows each student to change his/her status dynamically.
  - Can eliminate conditionals in `Student`'s methods.

Another Example: Firmware to Control Escalators

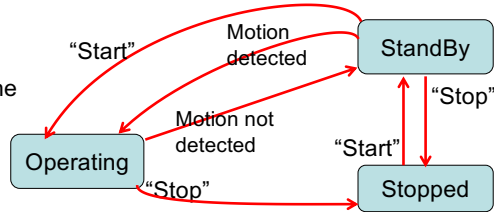
- An escalator performs different behaviors upon each event depending on its current state.
- Focus on an escalator's *behaviors* upon *events*.

- 4 Events
  - The "Start" button is pushed
  - The "Stop" button is pushed
  - Motion detected (with a motion sensor)
  - Motion not detected for a while (with a motion sensor)
- 3 states
  - Operating: Keeps moving escalator steps
  - Standby (idle)
    - Does not move steps because motion has not been detected for a while
    - Keeps running its motion sensor to possibly start moving steps
  - Stopped: Does not move steps. Does not run its motion sensor.

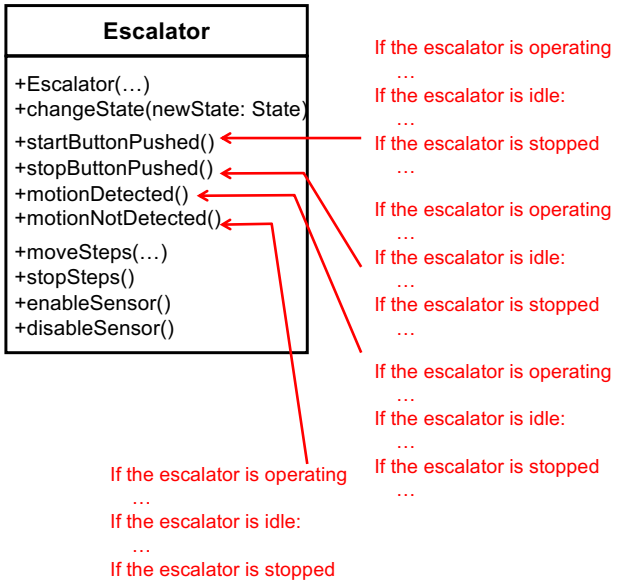


State-Dependent Behaviors

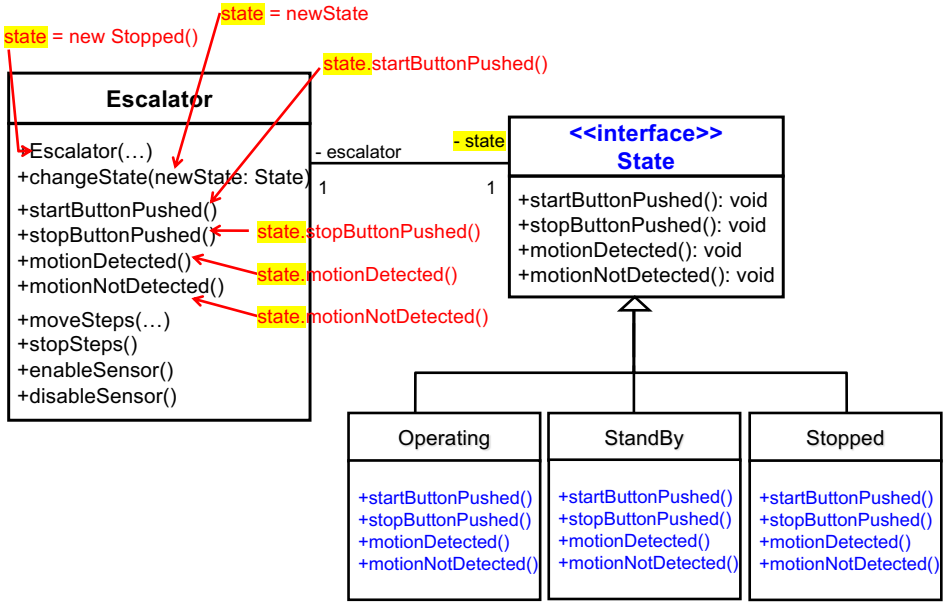
- When the "Start" button is pushed,
  - Does nothing (keeps moving steps)
    - If currently in "Operating"
  - Starts moving steps
    - If currently in "StandBy"
  - Enables the motion sensor and stands by
    - If currently in "Stopped"
- When the "Stop" button is pushed,
  - Does nothing (keeps the escalator stopped)
    - If currently in "Stopped"
  - Disables the motion sensor and stops the escalator
    - If currently in "StandBy"
  - Stops moving steps and disables the motion sensor
    - If currently in "Operating"
- When motion is detected,
  - Does nothing (keeps moving steps)
    - If currently in "Operating"
  - Starts moving steps
    - If currently in "StandBy"
- If motion is not detected,
  - Does nothing (keeps standing by)
    - If currently in "StandBy"
  - Stops moving steps and stands by
    - If currently in "Operating"



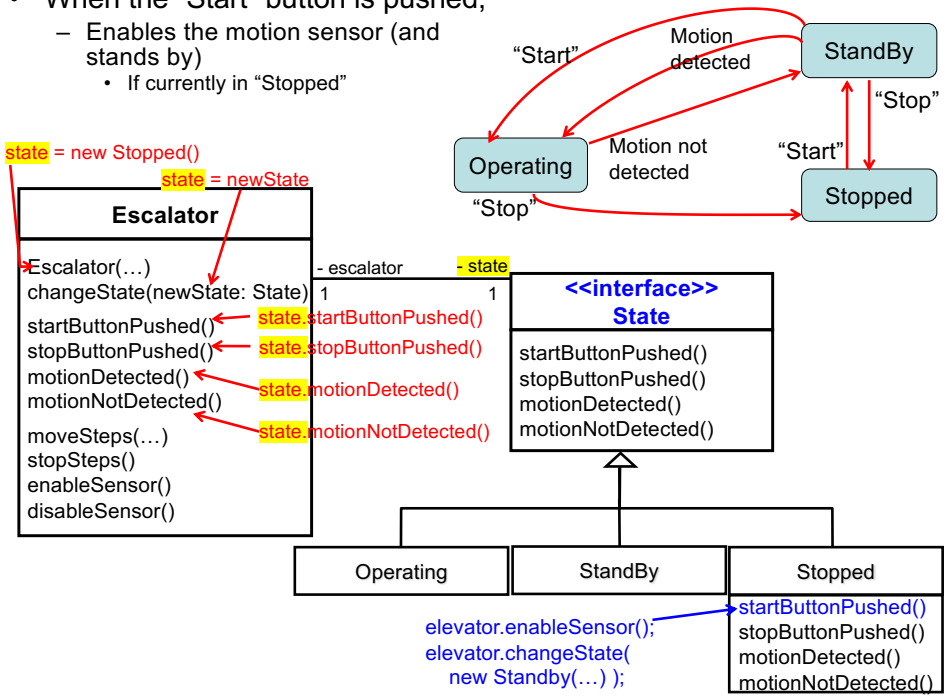
How to Implement State-dependent Behaviors



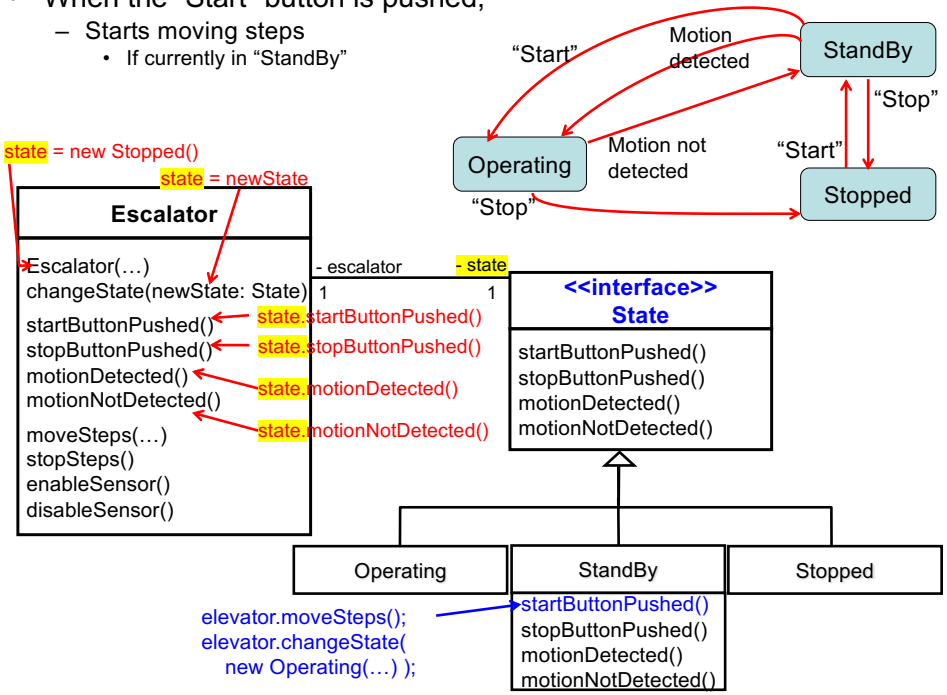
Using *State* Design Pattern



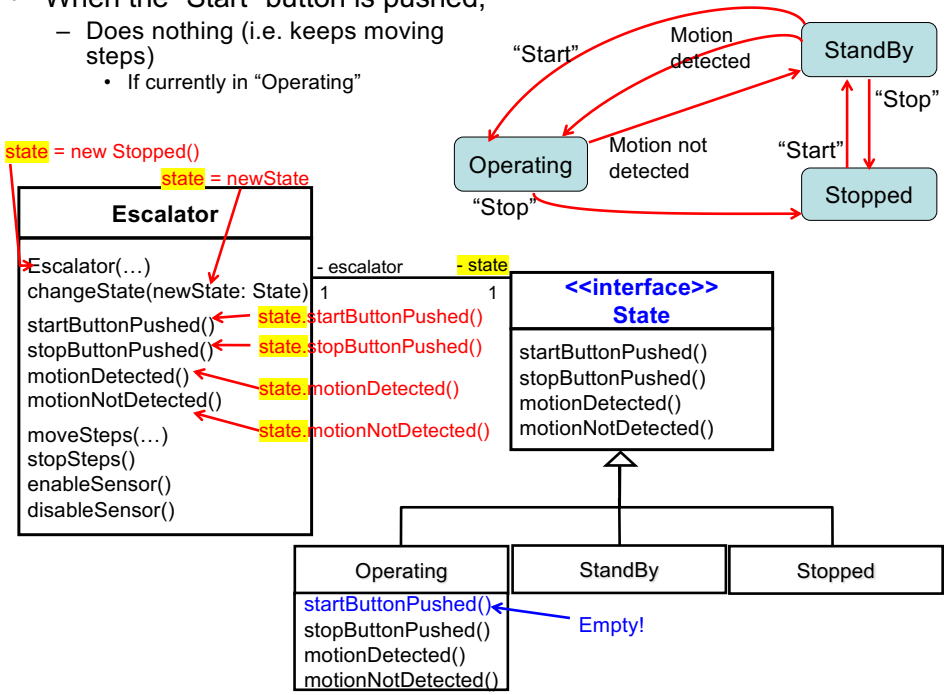
- When the "Start" button is pushed,
  - Enables the motion sensor (and stands by)
    - If currently in "Stopped"



- When the “Start” button is pushed,
  - Starts moving steps
    - If currently in “StandBy”



- When the “Start” button is pushed,
  - Does nothing (i.e. keeps moving steps)
    - If currently in “Operating”



Conditional-based or *State*-based Design

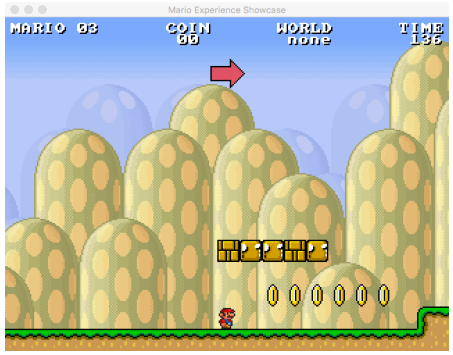
- Conditional-based
  - Maybe intuitive/straightforward to implement at first
  - Hard to maintain a long sequence of conditional branches
- *State*-based
  - May not be that intuitive/straightforward to implement at first
  - Easier (more principled/disciplined) to maintain
    - If a new event is added, just add an extra method to Escalator and each State subclass.
  - Initial cost may be higher, but maintenance cost (or total cost) should be lower over time
    - as changes are made in the future.

Note:

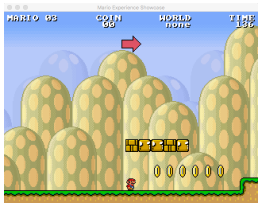
- Each **`State`** subclass and **`Escalator`** class can be *Singleton*.

One More Example: Game Characters

- Game characters often have state-dependent behaviors.
- Think of a simple 2D game like (classical) Super Mario



Handling User Inputs



Game loop

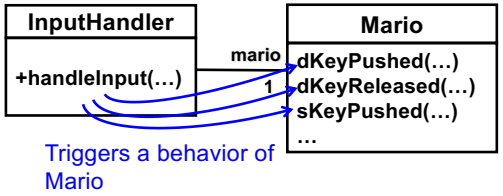
Detect an input from the user
→ Call InputHandler.handleInput()

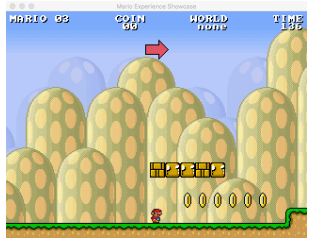
Update the scene (game world)

Render the game world

```
InputHandler ih = new InputHandler(...);
while(true){
  ih.handleInput(...);
  ...
}
```

- 5 types of inputs
  - The user can push the right arrow, left arrow, down arrow and "s" keys.
    - R arrow to move right
    - L arrow to move left
    - D arrow to duck
    - "s" to jump
  - The user releases the D arrow to stand up.
- **`InputHandler`**
  - **`handleInput()`**
    - identifies a keyboard input since the last game loop iteration (i.e. since the last frame).
    - 60 frames/s (FPS): One input per frame (i.e. during 1.6 msec)





For simplicity, let's focus on 3 inputs only here:
D arrow pushed, D arrow released, and "s" key pushed

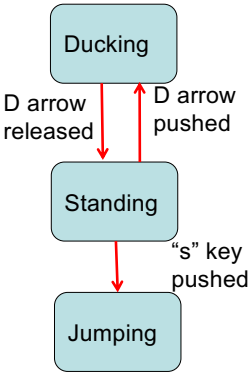
Game loop

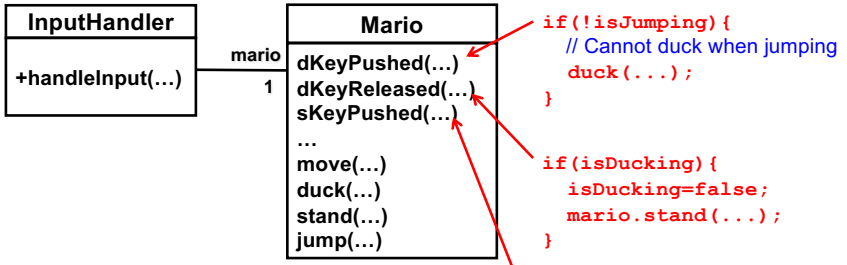
Detect an input from the user
→ Call InputHandler.handleInput()

Update the scene (game world)

Render the game world

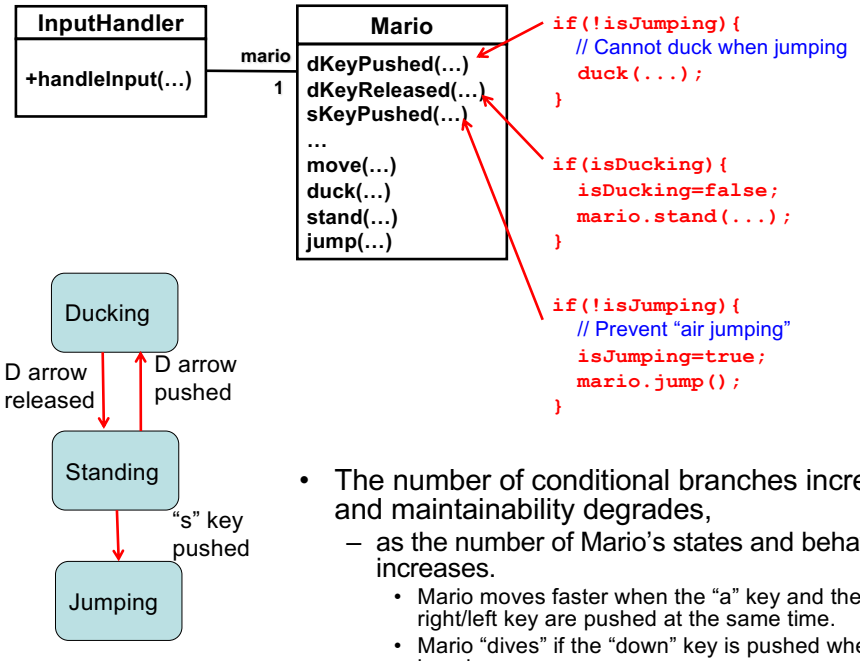
```
InputHandler ih = new InputHandler(...);
while(true){
  ih.handleInput(...);
    // If D arrow is pushed,
    //     call dKeyPushed() on Mario
    // If D arrow is released,
    //     ...
  ...
}
```

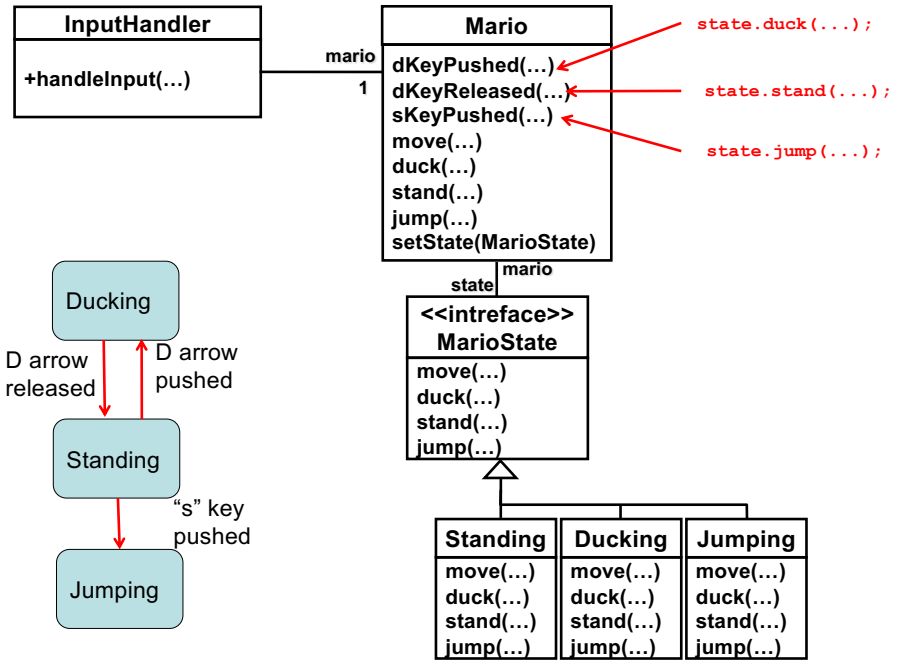


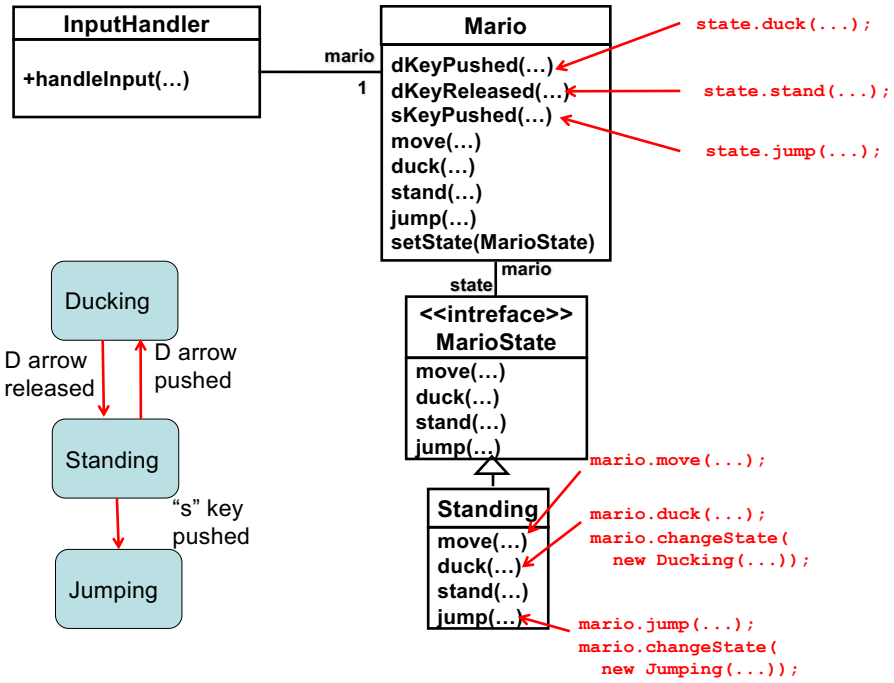
Mario differently behaves upon an event according to his current state.

Mario has and performs state-dependent behaviors



- The number of conditional branches increases and maintainability degrades,
  - as the number of Mario's states and behaviors increases.
    - Mario moves faster when the "a" key and the right/left key are pushed at the same time.
    - Mario "dives" if the "down" key is pushed when jumping.





HW 4

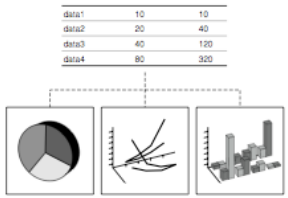
- Make YOUR OWN HW assignment and provide a solution to it.
- Come up with an example of the *State* design pattern in a particular application.
  - Do NOT use an example covered in this lecture note.
- Implement it yourself.
- Turn in:
  - Your implementation (code)
  - A short readme.txt file that explains what kind of app you consider and how your code implements the *State* design pattern.
  - Test code and an Ant script.

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)

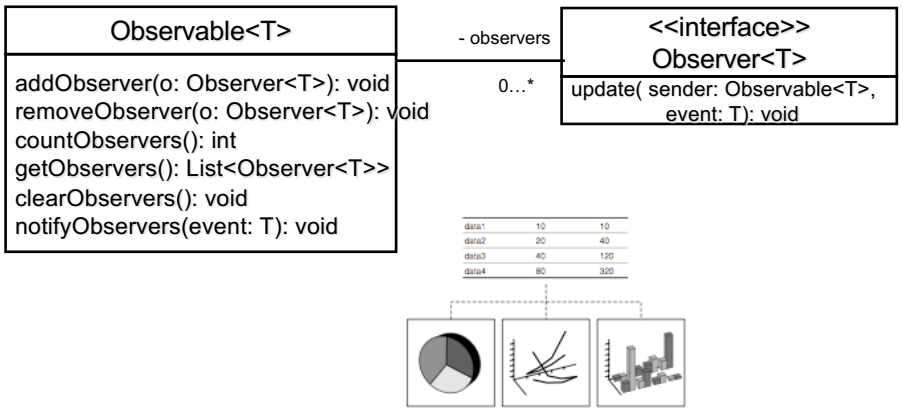
Event notification (publication)

Observers (subscribers, event listeners)

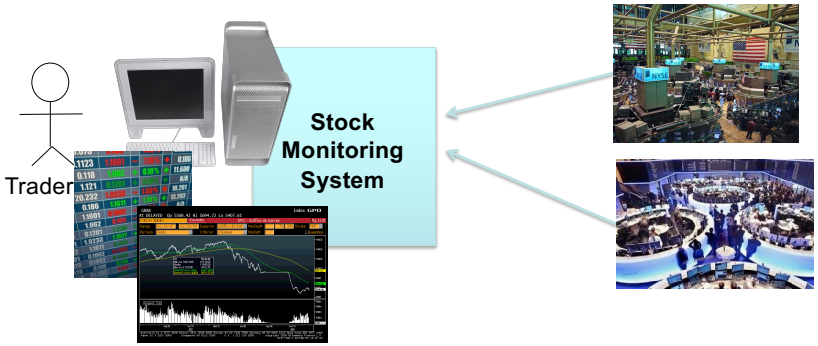
- Separate data processing from data management.
  - Data management: Observable
  - Data processing: Observers
    - e.g., Data analysis (e.g. feature extraction), graphical visualization, etc.

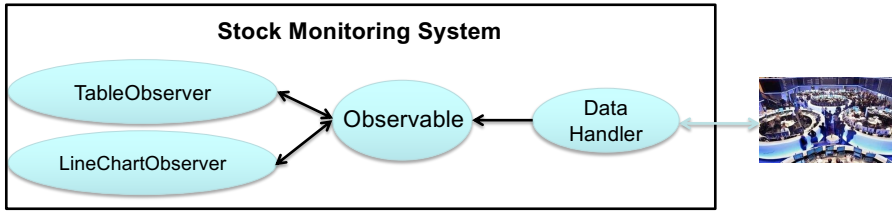
Simplified Version of MS Azure SDK

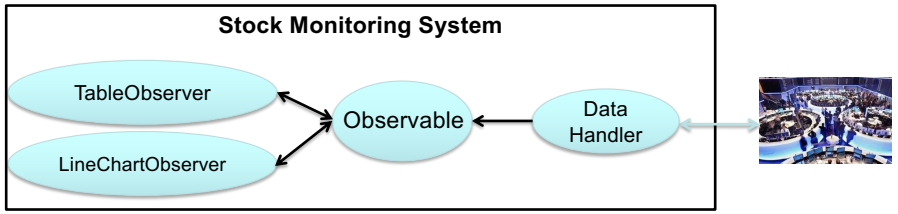
- `edu.umb.cs680.Observable`
- `edu.umb.cs680.Observer`



Example: Stock Price Monitoring

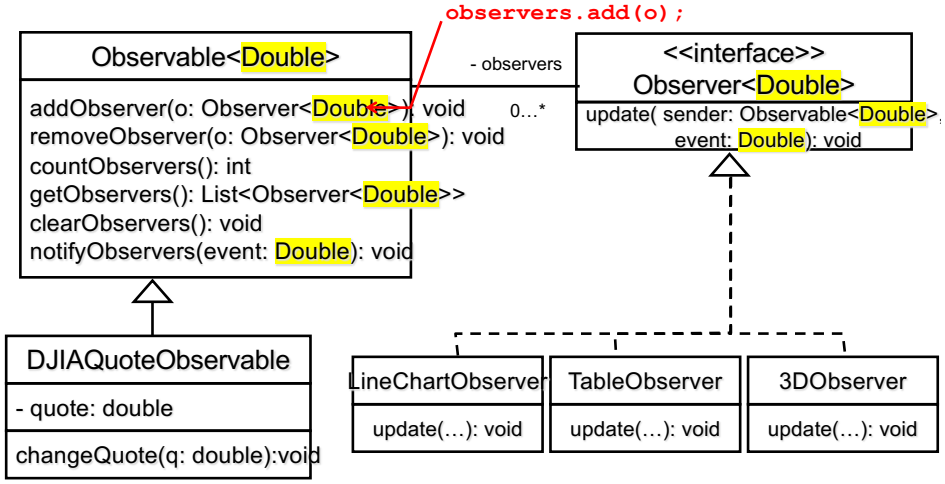






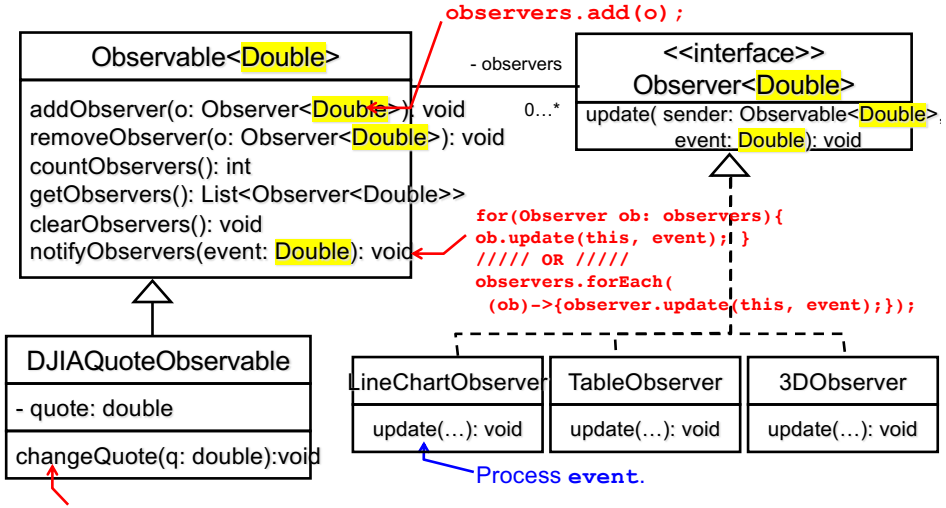
- Data handler: communication and data acquisition
  - Periodically fetches stock data from a stock exchange
  - Supplies the data to an Observable
- Observable: event notification
  - Notifies the data to Observers
- Observer: data visualization
  - Displays the data

Registration

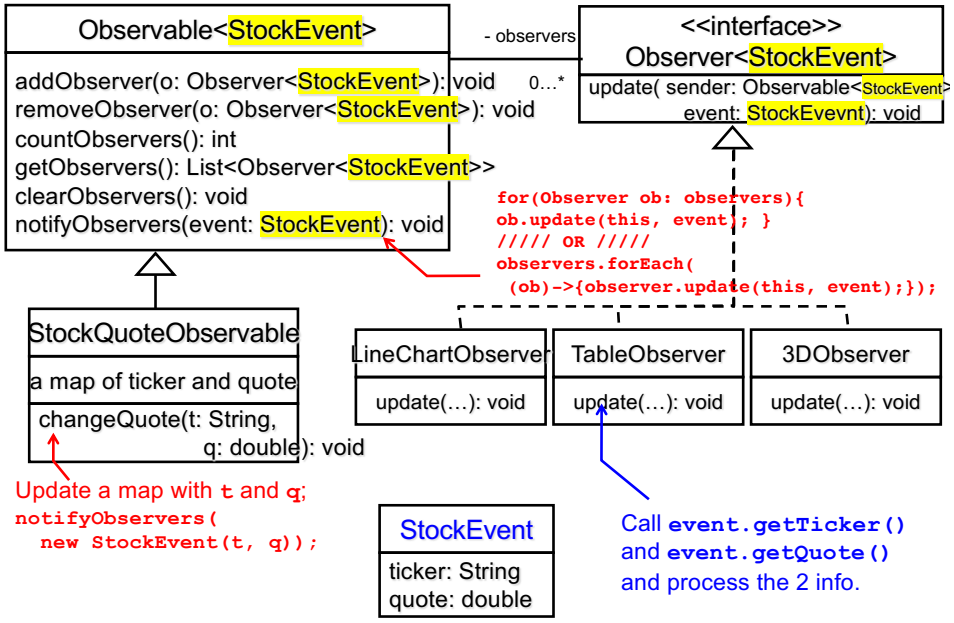


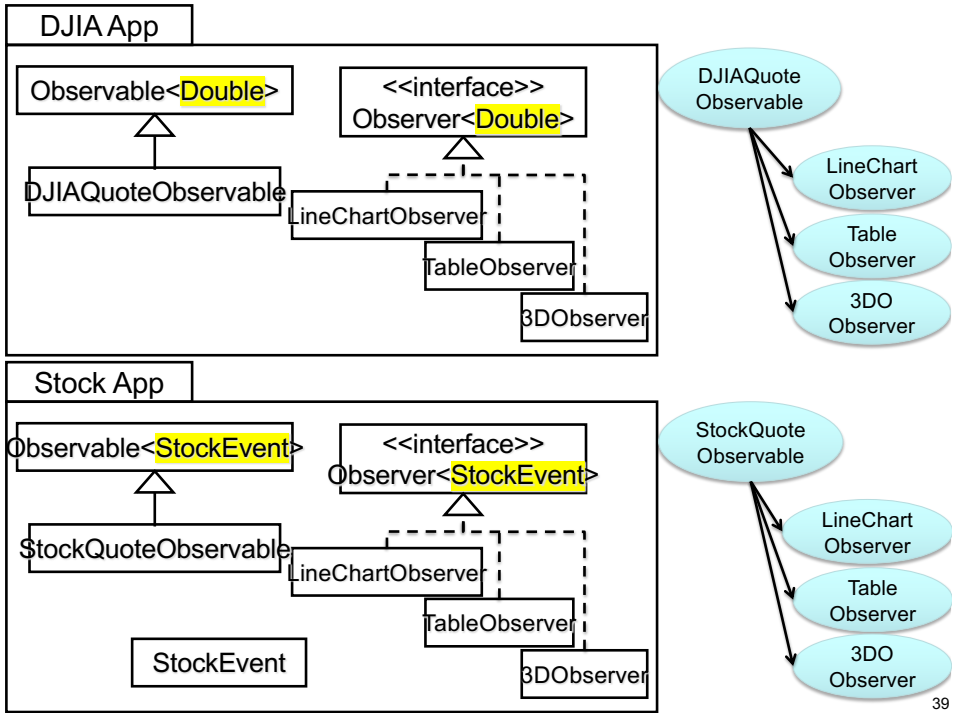
Each Observer calls addObserver() on DJIAQuoteObservable.

One-to-Many Event Notification



DJIAQuoteObservable calls notifyObservers().





39

HW 5

- Make YOUR OWN assignment and provide a solution to it.
- Come up with an example of the *Observer* design pattern in a particular application.
  - Do NOT use an example covered in this lecture note.
- 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.
  - They will be due on ~~Oct 31~~ Nov 7.