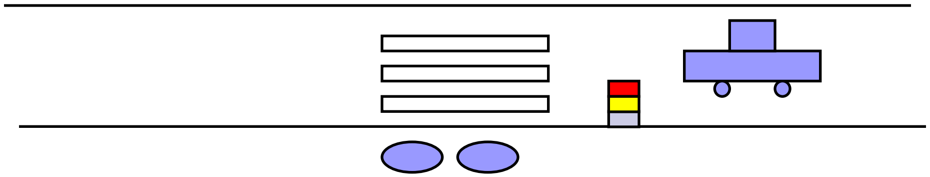
Behavioral Patterns

Observer

Back to Traffic Flow

- Consider the Traffic flow simulation



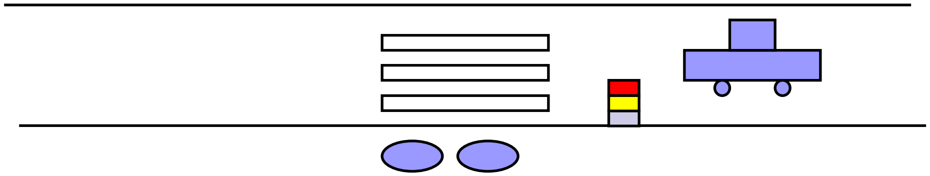
- When a traffic light on a crosswalk changes to red
  - □ Motorized vehicles stop
  - □ Pedestrians cross the road
- Polling is not always a good idea
  - □ Like “are we there yet?” every other second

Common problem

- What if a group of objects needs to update themselves when some object changes state?
  - Common problem!
    - Data changes, update the view (MVC)
    - Traffic Light changes, vehicles and peds need to update themselves (take action)
    - Following in social media
- Solution: subscribe and get notifications
- i.e. The **Observer** pattern

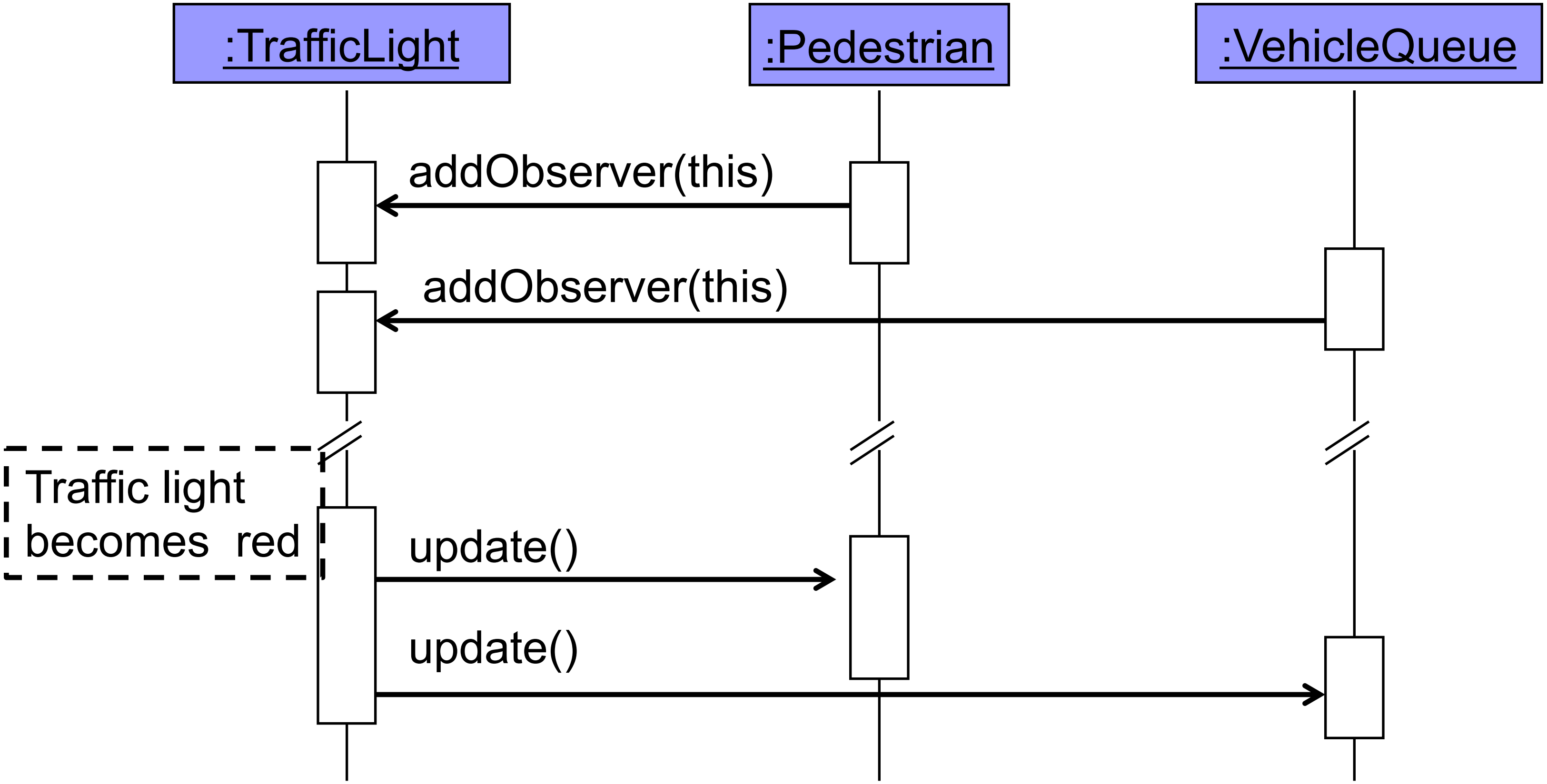
Back to Traffic Flow

- Consider the Traffic flow simulation



- When a traffic light on a crosswalk changes to red
  - □ Motorized vehicles stop
  - □ Pedestrians cross the road
- Polling is not always a good idea
- The waiting queues and people on each side should be notified when light changes
  - □ **Observer pattern**

Observer Pattern

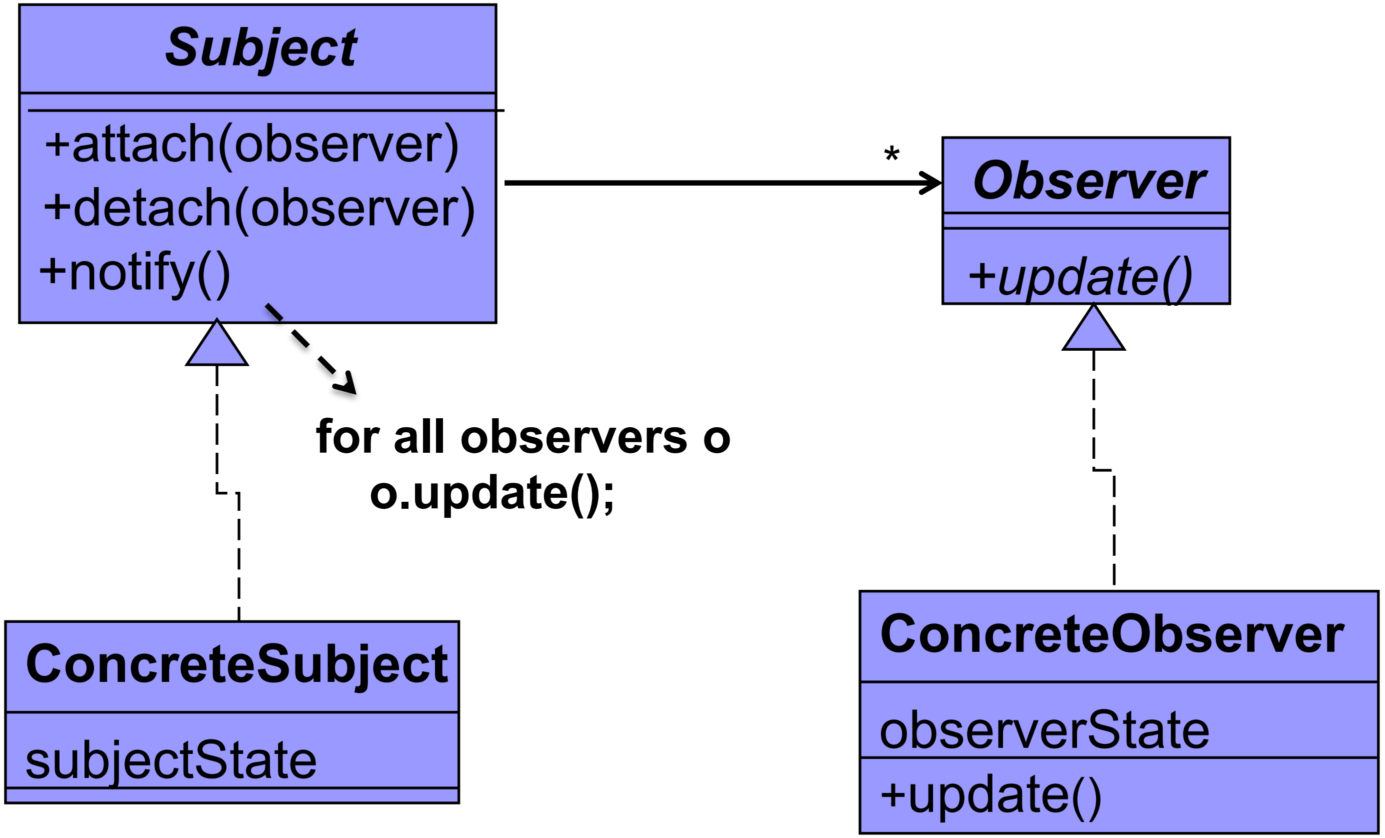


Traffic light notifies all of its observers at once
TrafficLight::notify(){ for each observer o o.update();}

Observer Pattern

- There is a one-to-many dependence
  - Many objects are interested in the Traffic Light
- Observers (Waiting queues, pedestrians) register to the Subject (Traffic Light)
  - Observers can change at runtime
- Change of state (light change) of the Subject notifies the observers
  - Observers get notified about any events that happen in the Subject
  - Change of state, events ….
- Observers take action upon notification
  - Update themselves

Structure: Observer Pattern



Subject

+attach(observer)
+detach(observer)
+notify()

for all observers o
o.update();

*

Observer

+update()

ConcreteSubject

subjectState

ConcreteObserver

observerState

+update()

Participants?

Observer

- **Intent**: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- A.k.a. Publish/Subscribe
- Motivation:
  - Need to maintain consistency between related objects without tight coupling

Exercise: Observing Traffic Light

- Make the Traffic Light observable

```
public interface Subject{
   public void attach(Observer o);
   public void remove(Observer o);
   public void notify();
}
public class TrafficLight implements Subject{
  private List<Observer> observers;
  private RGY light;  //state variable
  …/ /implement instance methods
 //implement the Subject interface
```

```
public class TrafficLight implements Subject{
  private List<Observer> observers;
  private RGY light;  //state variable
  public void attach(Observer o)
  {observers.add(o);}
  public void remove(Observer o)
  {observers.remove(o);}
  public void notify(){ //broadcasting the event
     for(var ob:observers)
       ob.update();
  }
  public void changeLight(){
      //change to Red, Yellow, Green
      notify();    //issue-2: who and when should notify
  }
}
```

Exercise: Observing Traffic Light

```
public interface Observer{
   public void update(); //issue-1
}
public class Pedestrian implements Observer{
   public void update(){ //issue-1
       //cross the road if light is green
       //stop observing the light issue-3
   }
    //other instance methods
}
public class MVehicleQueue implements Observer{
   public void update(){…}
}
```

Exercise: Observing Traffic Light

```
public interface Observer{
   public void update(); //issue-1
}
public class Pedestrian implements Observer{..}
public class VehicleQueue implements Observer{..}
```

- See the loose coupling
- The only thing common between these two classes is the Observer interface.
- Subject is unaware of their primary job
- OCP: new observer types do not affect the subject

Observer -- Loose coupling

- Subject only know interface.
  - Don’t know or care what any concrete observer does
- Add new observers at anytime
  - Runtime registration and remove
- Do not modify subject when new kinds of observers come to play
- Reuse subject and observer independently of each other
- Changes in subject or an observer will not affect each other –as long as they implement the interfaces
- Dependency Inversion

When to use Observer: Applicability

- A change to one object requires changing other objects and the actual set of objects is *unknown beforehand* or changes dynamically
  - notify other objects without making assumptions about who they are
- Need to maintain consistency between related objects without tight coupling
- An abstraction has two aspects, one dependent on the other. Encapsulate these aspects in separate objects lets you vary and reuse them independently

Implementation Issues / Choices

Issue1: arguments of `update()`

- □ Pedestrian will cross the road if the `light` of `TrafficLight` object is `Green`

- ■ Choice 1: Subject sends the data

```
public void update (RGY light)
```

- □ Push model: push the data observers need
- □ Subject sends detailed information to observers about its state
- □ observers less reusable, fixed update method signature

Exercise: Observing Traffic Light

```
public class Pedestrian implements Observer{
    public void update(RGY light){ //issue-1
        if(light==RGY.Green)
            //cross the road
    }//other methods…
}
```

- Pro:Pedestrian is unaware it is observing a TrafficLight object.
- Cons: When subject’s code change, maybe it pushes more
- Cons: Not all observers may need all the data subject pushes

Implementation Issues / Choices

Issue1: arguments of `update()`

- □ Pedestrian will cross the road if the `light` of `TrafficLight` object is `Green`

- ■ Choice 1: Subject sends the data

```
public void update (RGY light);
```

- □ **Push** model: push the data observers need

- ■ Choice 2: Observers query the subject

```
public void update (Subject s);
```

- □ **Pull** model: Send the Subject reference, observers asks for details explicitly afterwards
- □ Subject is ignorant of observers
- □ Observers call Subject back to get the state

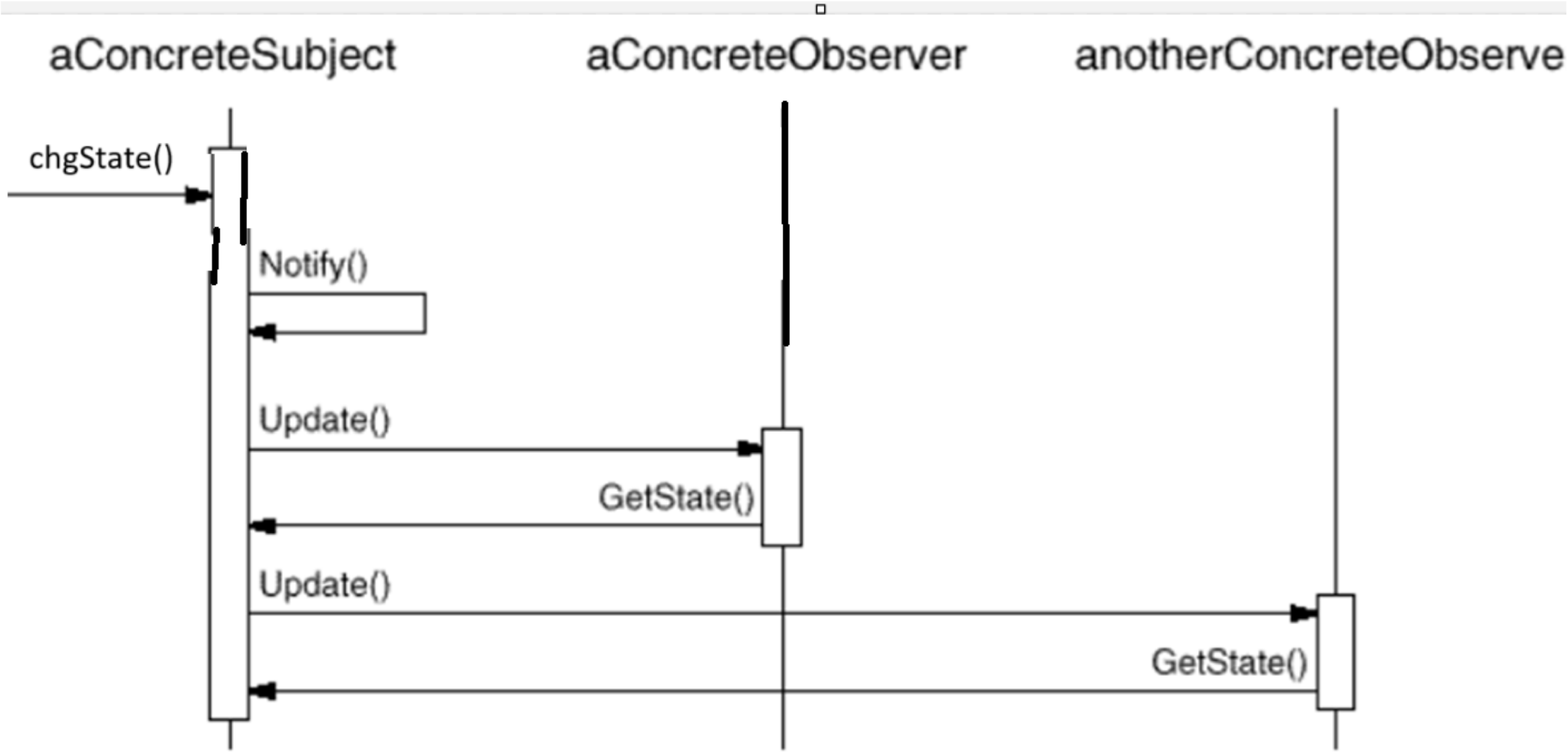
Exercise: Observing Traffic Light

```
public class Pedestrian implements Observer{
   public void update(Subject s){ //issue-1
      if(s.equals(mylight)){
         if(s.getLight()==RGY.Green)
             //cross the road
   }  }//other methods...
   private Subject mylight;
}
```

- the Subject interface needs **getter** methods

```
public interface Subject{
   public RGY getLight();
   public void attach(Observer o);
   public void remove(Observer o);
   public void notify();}
```

Observer -Collaborations



- Pull model is mostly preferred

Push vs Pull Models

- Pull Model is more decoupled
  - Pro: observers are reusable
  - Pro: subject does not assume what observers need
  - Cons: observers have to call subject
    - 1K subscribers/observers call your method to check what happened
  - Cons: subject has to provide getter methods
- Push model is more efficient
  - Pro: Subject controls what info to reveal
  - Cons: restricted observers

Implementation issues /Choices

Issue2: Who triggers the *notify* and when?

- Subject calls *notify* after each state changing operation
  - Pro: Clients do not have to remember to call notify
  - Cons: when series of operation, consecutive updates results in inefficiency
- Clients (users of subject) call *notify* at right time
  - Pro: update after a series of operation
  - Cons: client may forget it

Implementation issues /Choices

Issue2: Who triggers the *notify* and when?

- Subject calls *notify* after each state changing operation
  - Pro: Clients do not have to remember to call notify
  - Cons: when series of operation, consecutive updates results in inefficiency
  - `setChanged()` after series of operations,
    - Notify ignores when `isChanged` is not set
- Clients (users of subject) call *notify* at right time

Implementing issues -3

- Observing more than one subject
  - Which subject is notifying?
  - Subject passes itself via update method as a parameter

- How does an Observer remove itself?
  1. Store Subject references
  2. Sometimes observers remove themselves as a reaction to some event.
  - Subject passes itself via update method as a parameter, remove yourself using the reference

Exercise: Observing Traffic Light

```
public class Pedestrian implements Observer{
    public void update(RGY light){ //push
        if(light==RGY.Green){
            //cross the road
        //how to stop observing the light?
            mysubject.remove(this);
        }
    }
    public void atIntersection(Subject s){
        mySubject=s;
    }
    //other methods…
}
```

Exercise: Observing Traffic Light

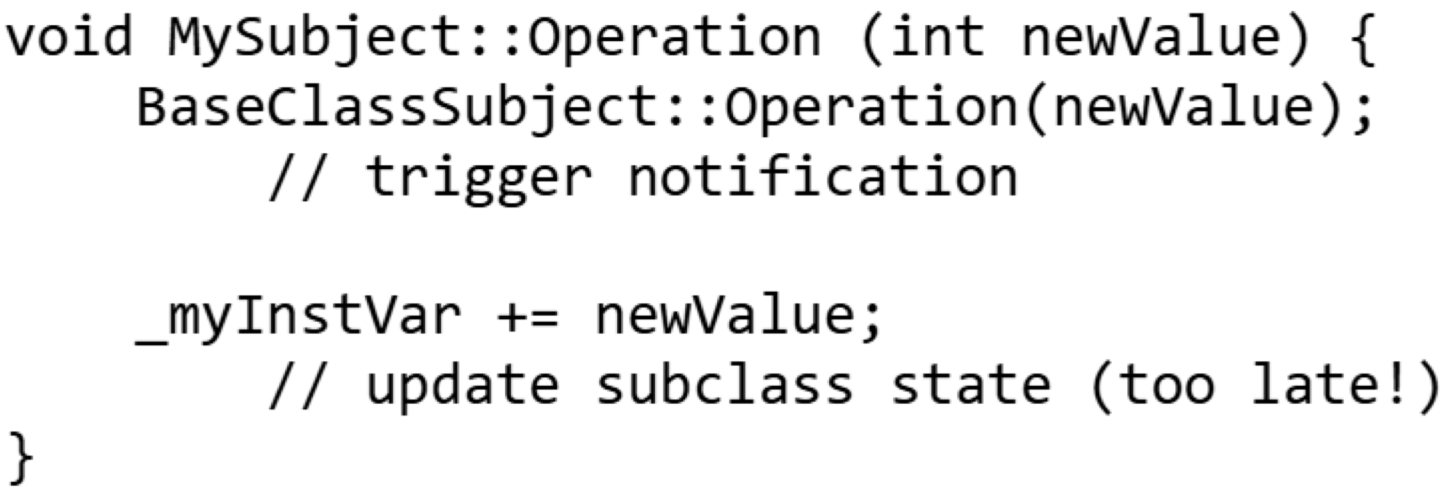
```
public class Pedestrian implements Observer{
    public void update(Subject s){ //pull
        if(s.equals(mylight)){
            if(s.getLight()==RGY.Green)
                //cross the road
         s.remove(this);//stop observing the light
    }
    public void atIntersection(Subject s){
         mylight=s;
    }//other methods...
    private Subject mylight;
}
```

Implementation issues -4,5,6

- Can an observer be also a subject?
- Do not assume an order of updates in Subject::notify
- Destructor method of Subject (C++)
  - Deleting a subject should not produce dangling references in its observers.
  - Make the subject notify its observers as it is deleted so that they can reset their reference to it.
  - Do **not** delete observers.
    - other objects may reference them
    - they may be observing other subjects as well.

Implementation issues -7

- Make sure the subject updates its state before sending out notifications
  - Do not notify in the middle of change
  - A problem caused by inheritance usually



```
void MySubject::Operation (int newValue) {
    BaseClassSubject::Operation(newValue);
        // trigger notification

    _myInstVar += newValue;
        // update subclass state (too late!)
}
```

BaseClassSubject calls notify itself inside the operation (see issue-2)

```
public class CSubject{
int state = 0;
public change(int increment){
   state = state + increment;
   notify();
}
//…
}
```

make the order of operation Constant

1. Make the change
2. Notify

```
public class CSubSubject extends CSubject{
 int additionalState = 0;
 public change(int increment){
   super.change(increment); //notified

   additionalState = additionalState + increment;
  // the state is changed after the notifiers are updated
 }
}
```

Template Method to rescue

```
public class CSubject{
  int state = 0;
  public void final change(int increment) {
     doUpdateState(increment);
     notifyObservers();
  }
  public void doUpdateState(int increment){
      state = state + increment; }
}
class CSubSubject extends CSubject{
   public void doUpdateState(int increment){
       super.doUpdateState(increment); // the observers are not notified
       additionalState = additionalState + increment;
}
```

Implementation issue (optional)

- Lambda and function references (advanced)
  - □ If you understood the mechanics of Observer, you may replace the observer interface with function references

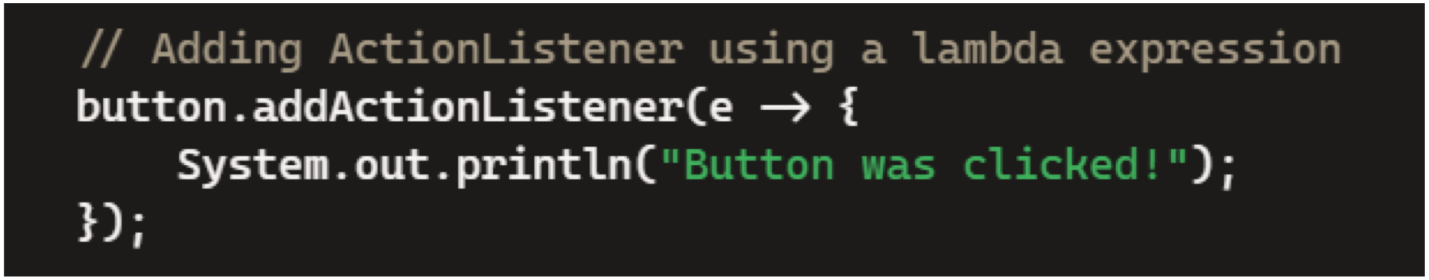
```
class Subject{…
  typedef std::function<void(Subject*)> ObserverFunction;
  void attach(ObserverFunction observer) {
       observers.push_back(observer); }
   void notify() { for (auto& observer : observers) {
observer(this); }
};
```

Implementation issue (optional)

- Lambda and function references (advanced)
- Observer attaches itself with a lambda

subject.attach(this { this->update(s); }); //inside an observer

- Another Example:



```
// Adding ActionListener using a lambda expression
button.addActionListener(e -> {
    System.out.println("Button was clicked!");
});
```

Exercise 2

- Weather station keep tracks of the recent temperature and pressure
  - setMeasurement(float temperature, float pressure)
- There are several weather data display
  - CurrentConditionDisplay
    - Prints current conditions
  - ForecastDisplay
    - Calculate prediction using the current data

Subject

Which update model does this code implement? Push or Pull

```
public class WeatherStation implements Subject {
    private List<Observer> observers=new ArrayList<>();
    private float temperature;
    private float pressure;
    public void registerObserver(Observer o) { observers.add(o);}
    public void removeObserver(Observer o) { observers.remove(o);    }
    public void notifyObservers() {
        for (Observer observer : observers)
            observer.update(temperature,  pressure);
    }
    public void setMeasurements(float temperature,  float pressure) {
        this.temperature = temperature;
        this.pressure = pressure;
        notifyObservers();
    }
```

Weather data Observer

- How to implement a ForecastDisplay
  - Gets current data from 2 stations
  - Makes prediction and display
  - pull model or push model?

Alternative: Pull model

```
public class ForecastDisplay implements Observer {
    private WeatherStation myStation;
    public ForecastDisplay(WeatherStation station) {
        this.myStation = station;
        station.registerObserver(this); // Register itself
    }
public void update(Subject s) {
     if (s == myStation) {
            float temp = myStation.getTemperature(); //pull
            float pressure = myStation.getPressure(); //pull
           //do the work
            System.out.println("Forecasting with temp: " + temp);
        }
    }
}
```

Alternative: Pull model

```
public class ForecastDisplay implements Observer {
   private WeatherStation mySta
   public ForecastDisplay(Weath
      this.myStation = station;
      station.registerObserver(this
   }
public void update(Subject s) {
    if (s == myStation) {
          float temp = myStation.getTemperature(); //pull
          float pressure = myStation.getPressure(); //pull
         //do the work
          System.out.println("Forecasting with temp: " + temp);
       }
   }
}
```

```
// Inside WeatherStation class
public void notifyObservers() {
   for (Observer observer : observers) {
      observer.update(this); // "this" is the Subjec
   }
}
```

Observer – Consequences-1

- Subject and observers are loosely coupled
  - Subject need not know concrete observers just knows each observer implements the update()
  - Can reuse subjects without reusing their observers and vice versa
  - Observers can be added without modifying the subject
  - Subject and observer can belong to different abstraction layers

Observer – Consequences-2

- Support for broadcasting
  - All observers are notified
  - The subject does not care how many interested objects exist
  - Freedom to add/remove Observers any time
  - It's up to the observer to handle or ignore a notification
- Unexpected updates
  - An operation on the subject may cause a cascade of updates to observers and their dependent objects

Event based systems

- Publish/subscribe is observer
- Event based architectures
- Put queues/topics or messaging services in between
  - Publisher(Subject) send events/notifications/messages to the topic/queue
  - Subscribers get notified and consume events/messages from the topic/queue
  - Security, availability, recoverability, persistency,….

This code is not thread safe (Java)

```
public class Subject{     //lazy init and not thread safe
   private List<Observer> observers=null;
  private void createObservers(){
           observers=new ArrayList<Observer>();}
  public void attach (Observer o)  {
          if(observers==null) createObservers();
          observers.add(o);}
  public void detach (Observer o) {
       if (observers!=null) observers.remove(o); }
  public void notify () {
          if (observers==null) return;
          for (Observer o: observers)   o.update(this);}
}
```

Not thread safe

```
public void detach (Observer o) {  observers.remove(o); }
public void notify () {
        if (observers==null) return;
        for (Observer o: observers)   o.update(this);
}
```

- What happens one thread is executing detach(o) while another thread is in notify()?
- ConcurrentModificationException

Not thread safe

```
public void attach (Observer o)  {
        if(observers==null) createObservers();
        observers.add(o);
}
```

- What happens when thread 1 see null observers and just before calling createObservers, thread 2 started and completed attach(o)?

Not thread-safe   (C++)

- No lazy initialization but still not thread safe

```
void Subject::attach (Observer* o)  { observers-
>push_back(o);}
 void Subject::detach (Observer* o) {  observers-
>remove(o); }
 void Subject::notify () {
    for (auto it = _observers.begin(); it != observers.end();
++it)
           it->update(this);
 }
```

- Race condition: What happens one thread is executing detach(o) while another thread is in notify()?

Thread safe Subject

- Use a thread safe Collection

```
public class Subject{
   private final List<Observer> observers= new
             CopyOnWriteArrayList<Observer>(); //in concurrency
package
  public void attach (Observer o)  {
          observers.add(o);}
  public void detach (Observer o) {  observers.remove(o); }
  public void notify () {
          for (Observer o: observers)   o.update(this);}
}  //final makes it initiated once. –no lazy init
   //CopyOnWriteArrayList is thread safe data structure
```

There are libraries for C++ that provides thread safe data structures (e.g. Boost)

Thread safe Subject -synchronized

```
public class Subject{
   private List<Observer> observers=null;
   private void createObservers(){
           observers=new ArrayList<Observer>();}
  public void synchronized attach (Observer o)  {
          if(observers==null) createObservers();
          observers.add(o);}
  public void synchronized detach (Observer o) {
            if(observers!=null) observers.remove(o); }
  //see next slide
```

This is Java synchronized.

There are C++ libraries with locking mechanism

Thread safe Subject -synchronized

```
  public void synchronized detach (Observer o) {
          observers.remove(o); }
  public void notify () {
      List<Observer> copy;
      synchronized(this){
          if (observers==null) return;
          copy=new ArrayList<>(observers); //shallow copy
      }
      for (Observer o: copy)   o.update(this);}
} //lazy init and pull method
```

- If notify is synchronized, deadlock!
  - Use reentrant locks or concurrent.CopyOnWriteArrayList

Observer –Known uses

- Event based systems
- Listeners in Java
  - implementations of `java.util.EventListener`
  - All Listeners in javax.swing
  - javax.servlet.http.HttpSessionBindingListener
- Any publish/subscriber is an observer
- Submit a task to cloud, do something else while waiting for task ended notification
- READ https://gameprogrammingpatterns.com/observer.html

Layered Systems and Observers

- In layered systems, objects in different layers should be loosely coupled
- communication between layers
  - Top layer calls bottom layer to perform task
    - FACADE
  - If bottom object calls upper one directly, it violates layering
    - put them in the same layer?
    - classes end up in inappropriate layers
  - observer
    - upper layer observes the lower layer subject
    - coupling only at interface level

Related patterns

- Mediator (next)
  - when we have cases of complex cases of many subjects an many observers
- Chain of responsibility (next week)
  - Observer notifies all registered handlers at once.
  - CoR sequentially looks for handlers
    - The linked observers in game patterns reading is CoR
- Command (next week)