Callbacks and Interacting Objects

CS 5010 Program Design Paradigms

"Bootcamp"

Lesson 10.8

The agreement between publisher and subscriber

- The publisher and subscriber must agree on a *protocol* for exchanging messages.
- The protocol consists of:
  - A publisher-side method for an object to subscribe to the messages
  - A subscriber-side method that the publisher can call to deliver the messages
  - An agreement on what messages mean and how they are represented.

Information and its Representation as Data (again!!)

Doing pub-sub without relying on a common method name

- You might have several different classes of subscribers, who want to do different things with a published message.
- Maybe you don't know the name of the subscriber's receiver method
- Solution: instead of registering an object, register a *function* to be called.
  - `f : X -> Void` where X is the kind of value being published
- To publish a value, call each of the registered functions
  - It's a callback!
- These functions are called *delegates* or *closures*.

No more update-wall-pos method

```
(define SBall<%>
  (interface (SWidget<%>)

    ;; ; Int -> Void
    ;; ; EFFECT: updates the ball's cached value of the
wall's position
    ;; update-wall-pos

    ))
```

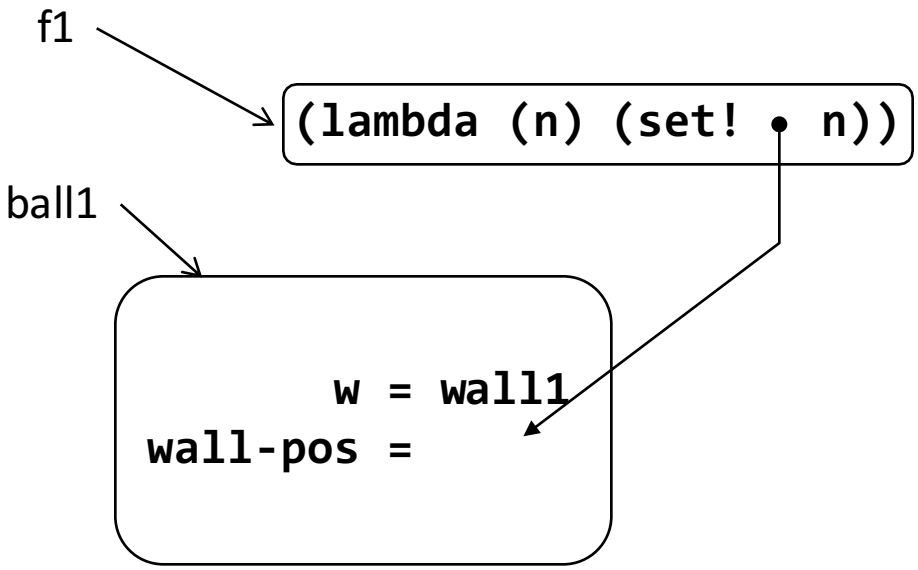
The Wall keeps a list of callback functions

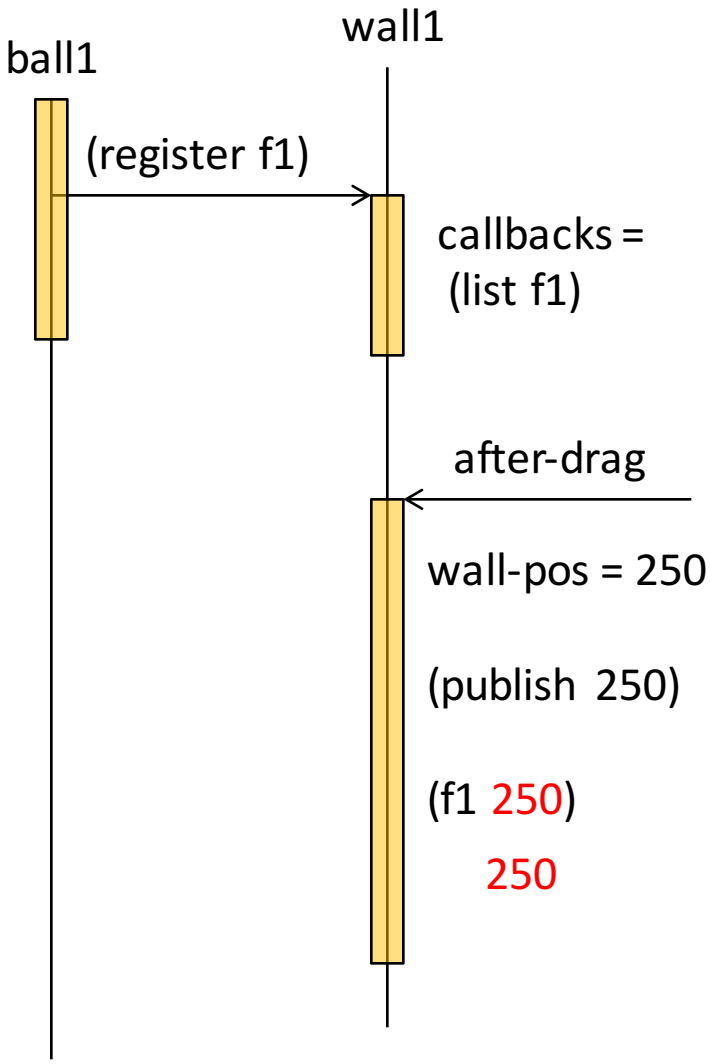
```
(define Wall%
  (class* object% (SWall<%>)
    ....

     ;; the list of registered balls
     ;; ListOf(Ball<%>)
     (field [balls empty])

    ;; the list of registered
    ;; callbacks
    ;; ListOf(Int -> Void)
    (field [callbacks empty])

    ;; (Int -> X) -> Int
    ;; EFFECT: registers the given
    ;;   callback
    ;; RETURNS: the current position
    ;;   of the wall
    (define/public (register c)
      (begin
        (set! callbacks
          (cons c callbacks))
        pos))
```

```
(define/public (after-drag mx my)
     (if selected?
       (begin
         (set! pos (- mx saved-mx))
         (for-each
           (lambda (callback)
             (send b update-wall-pos pos)
             (callback pos))
           callbacks))
       this))
```

The wall keeps a list of callback functions instead of a list of Balls. When the wall moves, it calls each registered function instead of sending a message to each registered ball.

Publishing through a delegate



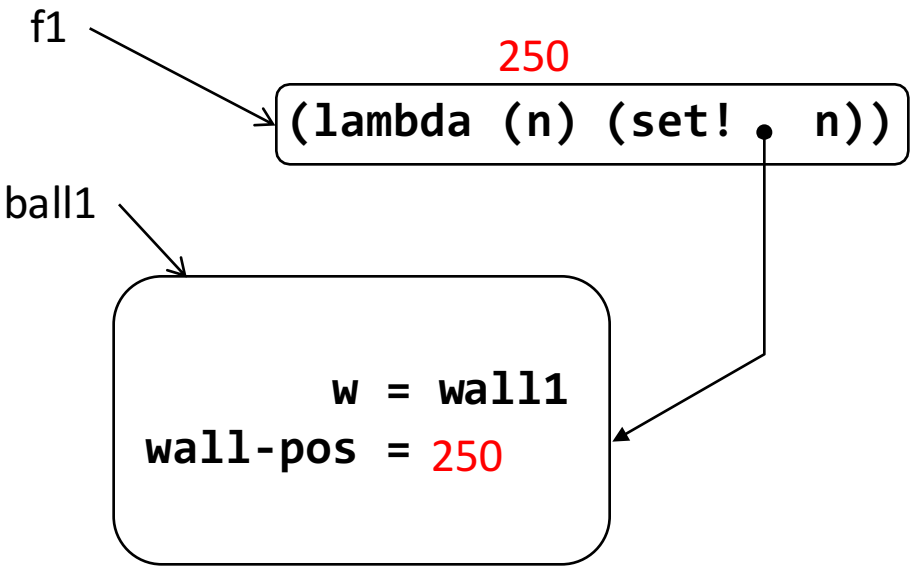


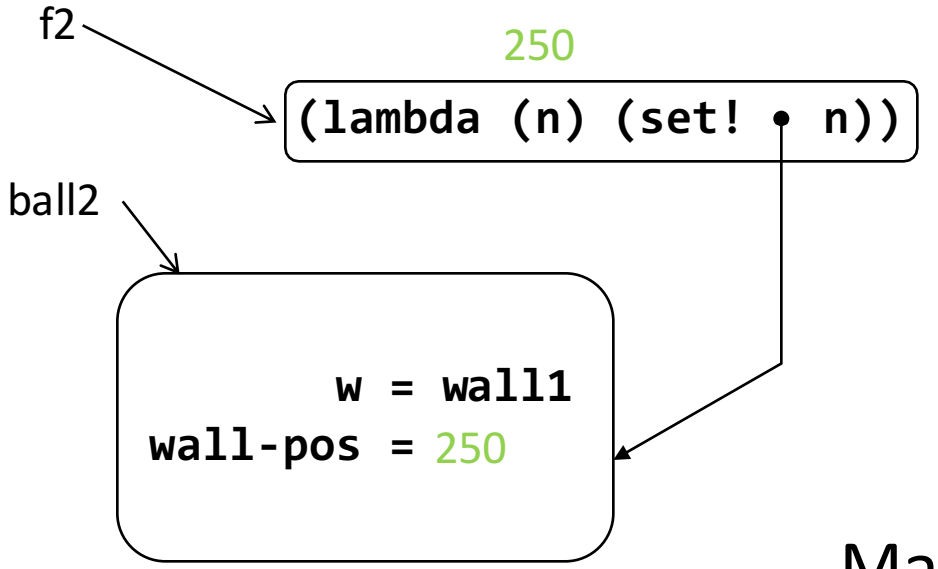
Whose **wall-pos**?

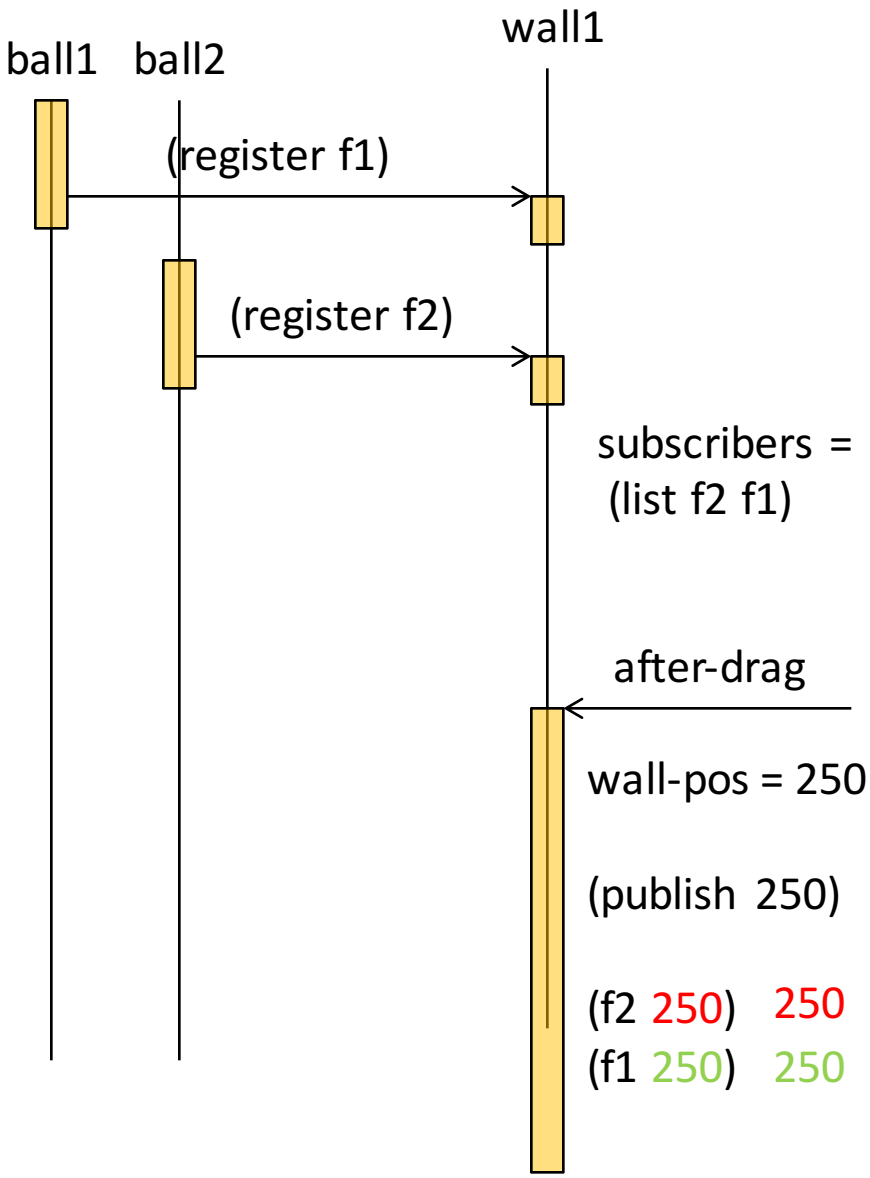
- When we write

  (lambda (n) (set! wall-pos n))

  we are referring to the **wall-pos** field in this object.
- The next slide shows a similar diagram illustrating what happens when there are two balls in the world.
- Each ball has its own delegate, which refers to its own **wall-pos** field.







Many balls, many delegates

Extending pub-sub

- Now that each ball knows about the wall, the ball could send the wall other kinds of messages.

Example: 10-8-communicating objects

- In this version, we'll allow the balls to interact with the wall directly.
- When a ball is selected, the key event "a" attracts the wall. It makes the wall move 50% closer to the ball.
- Similarly "r" repels the wall and moves the wall 50% farther away.
- Note this relies on the ball handling the keystrokes.

Protocol for this communication

- The ball will have an **update-wall-pos** method (as in 10-6-push-model-fixed).
- The wall will have a **move-to** method.
- The ball will call the move-to method with the x-position the wall should move to.
- The ball will use its cached version of wall-pos to calculate the desired new position for the wall.

move-to

```
(define SWall<%>
  (interface (SWidget<%>)

    ; SBall<%> -> Int
    ; GIVEN: An SBall<%>
    ; EFFECT: registers the ball
    ;  to receive position updates
    ;  from this wall.
    ; RETURNS: the x-position of the
    ;  wall
    register

    ; Int -> Void
    ; EFFECT: moves the wall to the given
    ; position.  Notifies all the
    ; registered balls about the change.
    move-to

    ))
```

In the interface

```
(define Wall%
  (class* object% (SWall<%>)

    ; the x position of the wall
    (init-field [pos INITIAL-WALL-POSITION])

     ...

    ; move-to : Integer -> Void
    ; EFFECT: moves the wall to the specified
    ; position, and reports the new position
    ; to all registered balls
    (define/public (move-to n)
      (set! pos n)
      (for-each
        (lambda (b)
         (send b update-wall-pos pos))
        balls))
```

In the class definition.

The for-each is repeated code, and should probably be moved to a help function

… and in **Ball%**

```
;; KeyEvent -> Void
(define/public (after-key-event kev)
  (if selected?
    (cond
      [(key=? kev "a") (attract-wall)]
      [(key=? kev "r") (repel-wall)])
    this))

(define (attract-wall)
  (send w move-to (- wall-pos (/ (- wall-pos x) 2))))

(define (repel-wall)
  (send w move-to (+ wall-pos (/ (- wall-pos x) 2))))
```

Many other protocols could accomplish the same thing

- Ball could send the wall the distance to move (either positive or negative), and the wall could move that distance.
- Or the wall could have two methods, **attract** and **repel**, and the ball could send **(/ (- wall-pos x) 2)** to one or the other of the methods.

Yet another protocol (part 1)

Introduce a data type of messages, say something like:

```
A MoveMessage is one of
-- (make-move-left NonNegInt)
-- (make-move-right NonNegInt)
Interp: the NonNegInt is the distance to
move
```

Yet another protocol (part 2)

- Then the receiver method in the wall will decode the message and move to the right position.
- This protocol generalizes: you could send the wall messages in an arbitrary complicated way.
- For example:

Wall choreography

```
;; A WallDance is a ListOfMoveMessage

;; WallDance -> Void
(define/public (interpret-dance msg)
  (cond
    [(empty? msg) this]
    [else (begin (interpret-move (first msg))
                 (interpret-dance (rest msg)))]))
```

Now the ball can give the wall a whole sequence of instructions in a single message.

WallDance is a programming language!

Extending pub-sub

- What if we wanted to deal with multiple messages?

Design #1: Separate subscription lists

- Each kind of message would have its own subscription list and its own method name
- Good choice if different groups of methods want to see different sets of messages.

Design #2: Single subscription list

- Better if most classes want to see most of the same messages.
- All subscribers now see all the messages
- The object can simply ignore the messages it's not interested in.

Variations on Design #2

- Could have different receiver methods for different messages:
  - This is what we did in **Widget<%>**, with **after-tick**, **after-key-event**, etc.
  - **add-stateful-object** was the equivalent of **register**.
- Or could have a single receiver method, but complex messages
  - sometimes called a "message bus"
  - this is how IP works: each device on the bus just listens for the messages directed to it.
  - this generalizes to large message sets

Summary: Reasons to use publish-subscribe

- Metaphor:
  - "you" are an information-supplier
  - You have many people that depend on your information
- Your information changes rarely, so most of your dependents' questions are redundant
- You don't know who needs your information

Module Summary

- Objects may need to know each other's identity:
  - either to *pull* information from that object
  - or to *push* information to that object
- Publish-subscribe enables you to send information to objects you don't know about
  - objects register with you ("subscribe")
  - you send them messages ("publish") when your information changes
  - must agree on protocol for transmission
    - eg: **(*method-name <data>*)**
    - eg: call a registered closure with some data
  - it's up to receiver to decide what to do with the data.

Next Steps

- Study the relevant files in the Examples folder:
  - 10-6-push-model-fixed.rkt
  - 10-7-callbacks.rkt
  - 10-8-interacting-objects.rkt
- If you have questions about this lesson, ask them on the Discussion Board
- Do Problem Set #10.