# Converting from Immutable to Mutable Objects

CS 5010 Program Design Paradigms

"Bootcamp"

Lesson 10.4

# Key Points for Lesson 10.4

- We need to document our assumptions about statefulness in our interfaces.
- **`Void`** means that the function can return any value it wants, so the caller must ignore the returned value.
- A function that has a **Void** return contract must have an EFFECT, so we must document this as part of the purpose statement.
- We can transform a method definition that produces a new object into one that alters this object by doing a **set!** on the fields that should change.
- This is the only acceptable use of **set!** in this course.

# The first thing we do is introduce a new interface

```
;; Every stable (stateful) object that lives in the world must implement the
;; SWidget<%> interface.

(define SWidget<%>
  (interface ()

    ; -> Void
    ; GIVEN: no arguments
    ; EFFECT: updates this widget to the state it should have
    ; following a tick.
    after-tick

    ; Integer Integer -> Void
    ; GIVEN: a location
    ; EFFECT: updates this widget to the state it should have
    ; following the specified mouse event at the given location.
    after-button-down
    after-button-up
    after-drag

    ; KeyEvent -> Void
    ; GIVEN: a key event
    ; EFFECT: updates this widget to the state it should have
    ; following the given key event
    after-key-event

    ; Scene -> Scene
    ; GIVEN: a scene
    ; RETURNS: a scene like the given one, but with this object
    ; painted on it.
    add-to-scene
    ))
```

We adopt the convention that stateful things have names starting with "**S**". Thus **Swidget<%>** is the interface for stateful widgets.

**add-to-scene** still returns a scene

# New contracts

- Key contract (in **Swidget<%>** )

```
on-mouse :
    Integer Integer MouseEvent -> Void
```

- **Void** means that the function can return any value it wants.
- The caller of the function can’t rely on it returning any meaningful value
- So the caller must ignore the returned value

# If we don’t return a useful value, then what?

- A function that has a **Void** return contract must have an EFFECT.
- Must document this as part of the purpose statement:

# Example of an EFFECT in a purpose statement

```
; -> Void
; GIVEN: no arguments
; EFFECT: updates this widget to the
; state it should have following a tick.
after-tick
```

# Transforming the method definition

- We can change a function that produces a new object into one that alters this object by doing a set! on the fields that should change.
- Often this is only a small subset of the fields, so the new code is considerably shorter than the old one.
- When we do this, the new function no longer produces a meaningful value, so whoever calls it can no longer rely on its value. This is the meaning of the **Void** contract.
- In other languages, **Void** means that the method returns no value at all. In Racket, every function returns some value, so we use **Void** to mean a value that we don’t know and don’t care about.

We sometimes call this code “imperative”, because it deals in commands rather than values.

# The **Void** transformation: method definition

```
; after-button-down : Integer Integer -> Void
; GIVEN: the location of a button-down event
; STRATEGY: Cases on whether the event is near the wall
(define/public (after-button-down mx my)
  (if (near-wall? mx)
    ;; (new Wall%
    ;;   [pos pos]
    ;;   [selected? true]
    ;;   [saved-mx (- mx pos)])
    (begin
      (set! selected? true)
      (set! saved-mx (- mx pos))
      ~~this~~)
    42))
```

We change each method that produces a new wall into one that alters this wall by doing a **set!** on the fields that should change.

**begin** evaluates its subexpressions from left to right and returns the value of the last one.

We don't care what value is returned, so the first **this** can be omitted; the **begin** returns whatever it returns and we don't care.

However, an if still needs a value for the "else" case. The value is ignored, so we've put in a nonsense value, 42.

# Another example

```
; after-drag : Integer Integer -> Void
; GIVEN: the location of a drag event
; EFFECT: If the wall is selected, move it so that the
;  vector from its position to the drag event is equal to
;  saved-mx
; STRATEGY: Cases on whether the wall is selected.

(define/public (after-drag mx my)
  (if selected?
    ;; (new Wall%
    ;;   [pos (- mx saved-mx)]
    ;;   [selected? true]
    ;;   [saved-mx saved-mx])
    (set! pos (- mx saved-mx))
    ; this
    38))
```

Another nonsense value to be ignored

# We modify WorldState% to deal with both Widgets and SWidgets

```
(define (make-world-state objs sobjs)
  (new WorldState% [objs objs][sobjs sobjs]))

(define WorldState%
  (class* object% (WorldState<%>)

    (init-field objs)  ; ListOfWidget
    (init-field sobjs)  ; ListOfSWidget

    (super-new)

    ;; after-tick : -> WorldState<%>
    ;; STRATEGY: Use map on the Widgets in this World; use for-each on the
    ;; stateful widgets

    (define/public (after-tick)
      (new WorldState%
        [objs (lambda (obj) (send obj after-tick))]
        [sobjs (begin
                 (for-each
                   (lambda (obj) (send obj after-tick)))
                 sobjs)]))
```

**for-each** is like map, but it doesn't make a list from the results. Its contract is
**(X -> Void) ListOfX -> Void**
See the Racket documentation for more.

Other methods in WorldState% modified similarly(*)

(*) In the code, I actually used a HOF **process-widgets** to avoid having to write this out several times.

# And we have to initialize the world

```
;; initial-world : -> WorldState
;; RETURNS: a world with a stateful wall, and a ball that knows about
;; the wall.
(define (initial-world)
  (local
    ((define the-wall (new Wall%))
     (define the-ball (new Ball% [w the-wall])))
    (make-world-state
      (list the-ball)
      (list the-wall))))
```

# And now all's well with the world

- When the wall moves, it gets mutated with set!, but it retains its identity.
- The ball is still functional– at every tick you get a **new Ball%** , but only one wall ever gets created, and every incarnation of the ball sees it.
- Go run 10-2B-stateful-wall.rkt

# What do I write for the strategy?

- As in Week 09, a strategy should be a tweet-sized description of how your function or method works.
- Again as in Week 09, strategies are optional; write them if they are useful.
- Look at the examples in this lesson and in the example files.

# Review of Key Points for Lesson 10.4

- We need to document our assumptions about statefulness in our interfaces.
- `Void` means that the function can return any value it wants, so the caller must ignore the returned value.
- A function that has a **Void** return contract must have an EFFECT, so we must document this as part of the purpose statement.
- We can transform a method definition that produces a new object into one that alters this object by doing a **set!** on the fields that should change.
- This is the only acceptable use of **set!** in this course.

# Next Steps

- Study 10-2B-stateful-wall.rkt in the Examples folder.
- If you have questions about this lesson, ask them on the Discussion Board
- Do Guided Practice 10.1
  - Be sure to do this one– there is new material in there.
- Go on to the next lesson.