Converting from Immutable to Mutable Objects

CS 5010 Program Design Paradigms
"Bootcamp"
Lesson 11.4
Key Points for Lesson 11.4

- **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.
- 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.
Video Demonstration: stateful-box.rkt

- http://www.youtube.com/watch?v=pu_JQoy_yys (7:30)
The **Void** transformation: contracts

- Key contract (in Box%):

  ```plaintext
  on-mouse : 
    Integer 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

;; adjust-width : Number -> Void

;; **EFFECT:** adjusts the center and width of this box so that the width is new-width and the left edge is unchanged
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: function definition

```
(define (adjust-width new-width)
  ;; (new Box%)
  ;;      [x (+ (send this left-edge) (/ new-width 2))]
  ;;      [y y]
  ;;      [w new-width]
  ;;      [h h]
  ;;      [objects objects]
  ;;      [selected? selected?])
  (set! x (+ (send this left-edge) (/ new-width 2)))
  (set! w new-width))
```

We change a function that produces a new box into one that alters this object by doing a set! on the fields that should change.
World-after-tick in stateful-box.rkt

The world always has same box

IN Variant: the world’s ball always points to the world’s box
World after drag in stateful-box.rkt

```scheme
(new World%
[box (begin (send box after-mouse ...) box]
[ball (send ball after-mouse ...)])
```

The world always has same box

```
x = 220
box = 
```

WIN!

```
x = 252
w = 104
(right-edge)
=> 306
```

```
x = 220
box = .
```

ball after-mouse returns itself

INVARIANT: the world’s ball always points to the world’s box
Next step: a list of balls

Now let's have a list of balls instead, Keystroke "n" creates a new ball. Plan:

1. First, we'll write an add-object method in our world.
2. Then, we'll add a ball factory to the world. The ball factory will know about the world and the box, and on an "n" it will add a ball by calling the add-object method on the world.

But wait: the factory needs to know about the world, so it can send it add-object messages. So this means the world must have identity: it must be stateful.

So before we do anything else, let's just make the world stateful.
Video Demonstration: stateful-world.rkt

- [http://www.youtube.com/watch?v=RwLJtn5ceOo](http://www.youtube.com/watch?v=RwLJtn5ceOo) (7:52)
World-after-tick in stateful-world.rkt

\[(\text{send world after-tick}) = (\text{begin})\]
\[\quad (\text{send box after-tick})\]
\[\quad (\text{set! ball (send ball after-tick)})\]\n
\textbf{Invariant:} the world’s ball always points to the world’s box
World after drag in stateful-world.rkt

```
world

(send world on-mouse ...)
= (begin
   (send box on-mouse ...)
   (set! ball (send ball on-mouse ...)))
```

x = 252
w = 104
(right-edge) => 306

x = 220
box = 

ball on-mouse returns itself

INVARIANT: the world’s ball always points to the world’s box
Video Demonstration: ball-factory.rkt

• Next, we’ll add a ball factory to the world. This is a special purpose `StatefulWorldObject<%>` that processes keystrokes and adds objects to the world.

• [http://www.youtube.com/watch?v=cB3rnujW0qU](http://www.youtube.com/watch?v=cB3rnujW0qU)
Building the initial world in ball-factory.rkt

INVARIANT: the world’s ball always points to the world’s box

WIN!
Key Points for Lesson 11.4

- **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.
- 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.