A Case Study: Space Invaders

CS 5010 Program Design Paradigms "Bootcamp"

Lesson 9.2

© Mitchell Wand, 2012-2015
This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

Goals of this lesson

- Use the idea of an interface to write a small interactive system

Let's see a demo!

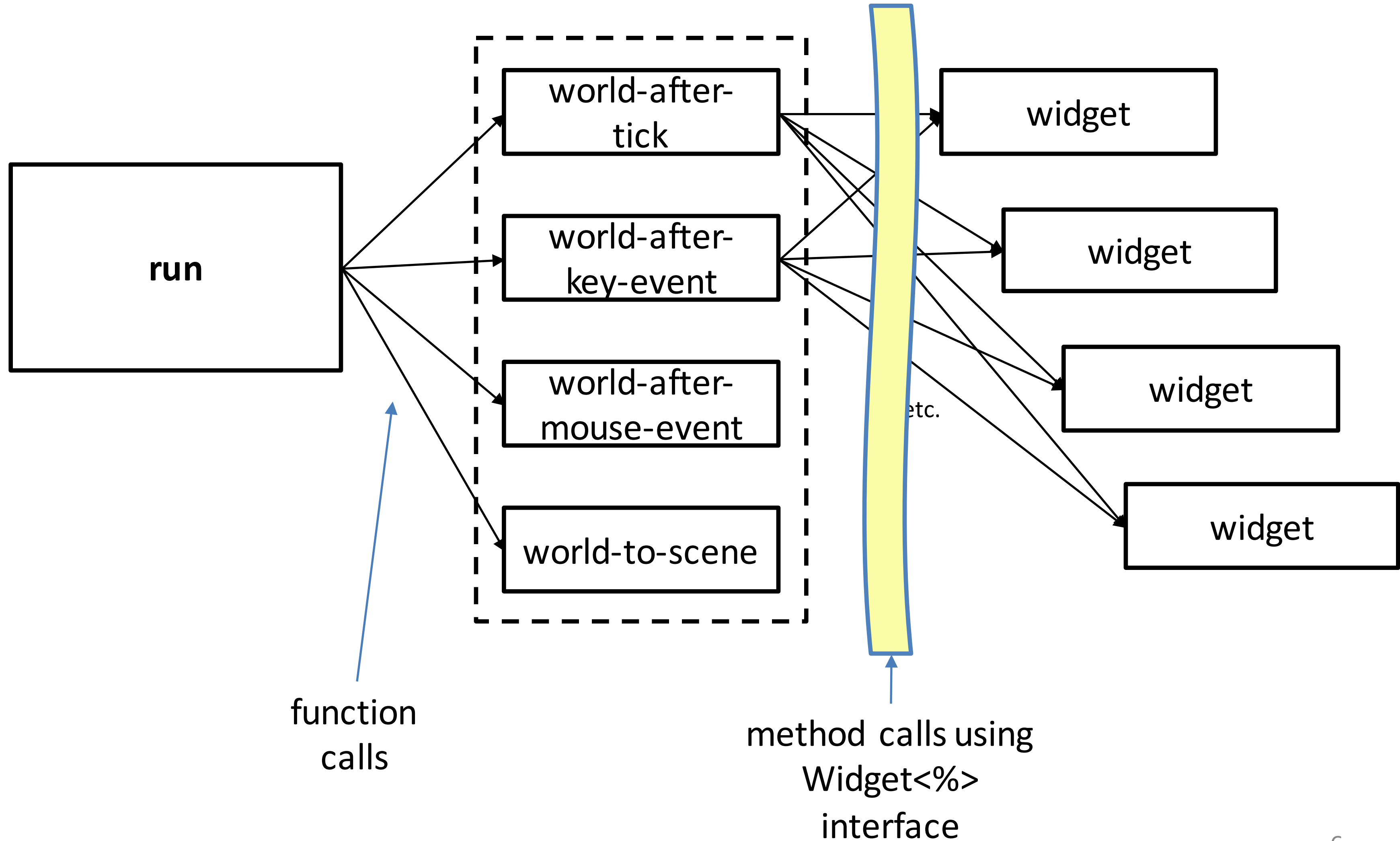
Let’s design a system!

- We will have some things living on a canvas.
- We’ll call these things *widgets*. We’ll represent widgets as objects.
- First step is to figure out what messages these objects should respond to.

System Design (2)

- big-bang will call our world-after-XX functions
- Each world-after-XX function will send an appropriate message to each widget.
- We can use our previous experience with big-bang to guide us.

System Architecture



What messages should a widget respond to?

- Our big-bang functions will send each widget the appropriate message at each event.
- Two easy ones:
  - **(send widget1 after-tick)** should return the state of **widget1** after a tick
  - **(send widget1 after-key-event kev)** should return the state of **widget1** after the given key event

What about display?

- We discovered that the right way to write display code was to write **add-to-scene** .
- So we'll say:
  - **(send widget1 add-scene s)** returns a Scene like **s**, but with **widget1** painted on it.

What about mouse events?

- We wrote a bunch of things like

```
(define (rect-after-mouse-event r mx my mev)
  (cond
    [(mouse=? mev "button-down") (rect-after-button-down r mx my)]
    [(mouse=? mev "drag") (rect-after-drag r mx my)]
    [(mouse=? mev "button-up") (rect-after-button-up r)]
    [else r]))
```

Let’s only do this once…

- We’ll put three methods in our interface:
  - **after-button-down**
  - **after-button-up**
  - **after-drag**
- It will be the responsibility of **world-after-mouse-event** to do cases on the mouse event and send the appropriate message to each widget.
  - this is sometimes called “demultiplexing”.

Our **Widget<%>** interface

```
;; Every object that lives in the world
;; must implement the Widget<%> interface.

(define Widget<%>
  (interface ()

    ; -> Widget<%>
    ; GIVEN: no arguments
    ; RETURNS: the state of this object
    ; that should follow after a tick
    after-tick

    ; Integer Integer -> Widget<%>
    ; GIVEN: x and y coordinates for a location
    ; RETURNS: the state of this object
    ; that should follow the specified
    ; mouse event at the given location.
    after-button-down
    after-button-up
    after-drag

    ; KeyEvent -> Widget<%>
    ; GIVEN: a key event
    ; RETURNS: the state of this object
    ; that should follow after 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
    ))
```

Some vocabulary

- We wrote:

```
; Integer Integer -> Widget<%>
; GIVEN: x and y coordinates for a location
; RETURNS: the state of this object
; that should follow the specified
; mouse event at the given location.
after-button-down
after-button-up
after-drag
```

- this object : **this** always refers to the object that receives the message
- the specified mouse event: "specified" refers to which of the three functions in this group we are talking about (e.g., after-button-down talks about what should follow a button-down event)
- the given location: "given" always refers to the arguments of the method call, e.g. (send obj after-button-down 10 20) refers to a button-down event at (10, 20)
- We will use this terminology consistently in our purpose statements when referring to different quantities.

Let’s look at the code for the world

```
;; Data Definitions

;; A Time is a NonNegative Integer

;; A Widget is an object whose class
;; implements Widget<%>

(define-struct world-state
   (objects time))

;; A WorldState is a
;; (make-world-state
;;    ListOfWidget Time)

;; INTERP: (make-world-state lst t)
;;  represents a world containing
;;  the widgets in lst at time t
;;  (in ticks).
```

```
; run : PosReal -> WorldState
; GIVEN: a frame rate, in secs/tick
; EFFECT: runs an initial world at
; the given frame rate
; RETURNS: the final state of the
; world
; STRATEGY: deliver events to the
;  event handler functions
(define (run rate)
  (big-bang (initial-world)
    (on-tick world-after-tick rate)
    (on-draw world-to-scene)
    (on-key  world-after-key-event)
    (on-mouse
      world-after-mouse-event)))
```

Nothing exciting here. We put a time component in the worldstate to illustrate that the worldstate might have more things in it than just the list of widgets.

world-after-tick

```
;; world-after-tick : WorldState -> WorldState
;; Use HOF map on the Widgets in w
(define (world-after-tick w)
  (let ((objs (world-state-objects w))
        (t (world-state-time w)))
    (make-world-state
      (map
        (lambda (obj) (send obj after-tick))
        objs)
      (+ 1 t))))
```

I used **let** instead of **local**. You can use either one.

On a tick, the world sends an **after-tick** message to each of the widgets, and assembles the results with **map** to get the new list of widgets. It also increments its timer.

world-to-scene

```
;; world-to-scene : WorldState -> Scene
;; Use HOF foldr on the Widgets in w
(define (world-to-scene w)
  (foldr
    ;; Widget Scene -> Scene
    (lambda (obj scene)
      (send obj add-to-scene scene))
    EMPTY-CANVAS
    (world-state-widgets w)))
```

world-to-scene is similar, except it does a **foldr** to assemble the scene.

world-to-mouse-event

```
;; world-after-mouse-event
;;  : WorldState Nat Nat MouseEvent -> WorldState
;; STRATEGY: Cases on mev
(define (world-after-mouse-event w mx my mev)
  (cond
    [(mouse=? mev "button-down")
     (world-after-button-down w mx my)]
    [(mouse=? mev "drag")
     (world-after-drag w mx my)]
    [(mouse=? mev "button-up")
     (world-after-button-up w mx my)]
    [else w]))
```

world-after-mouse-event decides which mouse event it is looking at, and calls the appropriate specialized function. See how we follow the data definitions!

world-after-button-down

```
; WorldState Nat Nat -> WorldState
; STRATEGY: Use HOF map on the widgets in w
(define (world-after-button-down w mx my)
  (let ((objs (world-state-widgets w))
        (t (world-state-time w)))
    (make-world-state
      (map
        (lambda (obj)
         (send obj after-button-down mx my))
        objs)
      t)))
```

world-after-button-down follows the pattern of **world-after-tick**. **world-after-button-up** and **world-after-drag** are similar.

world-after-key-event

```
;; world-after-key-event : WorldState KeyEvent -> WorldState
;; STRATEGY: Cases on kev
;; "b" and "h" create new bomb and new helicopter;
;; other keystrokes are passed on to the widgets in the world.

(define (world-after-key-event w kev)
  (let ((objs (world-state-widgets w))
        (t (world-state-time w)))
    (cond
      [(key=? kev NEW-BOMB-EVENT)
       (make-world-state
        (cons (new-bomb t) objs)
        t)]
      [(key=? kev NEW-HELI-EVENT)
       (make-world-state
        (cons (new-heli) objs)
        t)]
      [else
       (make-world-state
        (map
         (lambda (obj) (send obj after-key-event kev))
         (world-state-widgets w))
        t)])))
```

world-after-key-event responds to “b” and “h” itself to add new widgets to the world. Other key events are passed to the objects.

Next we'll build some widgets

- We have two classes of widgets: Bombs and Helicopters
- Bombs drop from the top of the screen. They are not draggable.
- Helicopters rise from the bottom of the screen. They are selectable and draggable, like the rectangles in our screensavers.

We’ll start with Bomb%

```
;; A Bomb is a (new Bomb% [x Integer][y Integer])
;; A Bomb represents a bomb.
;; in this version, the bomb just falls.

;; Handy to have a functional interface.
;; We don't use t, now but we might do so later.
(define (new-bomb t)
  (new Bomb% [x BOMB-INITIAL-X][y BOMB-INITIAL-Y]))

(define Bomb%
  (class* object% (Widget<%>)
    (init-field x y)  ; the bomb's x and y position

    ;; private data for objects of this class.
    ;; these can depend on the init-fields.

    ;; image for displaying the bomb
    (field [BOMB-IMG (circle 10 "solid" "red")])
    ;; the bomb's speed, in pixels/tick
    (field [BOMB-SPEED 8])

    (super-new)

    ;; after-tick : -> Bomb
    ;; RETURNS: A bomb like this one, but as it should
    ;; be after a tick
    ;; DETAILS: the bomb moves vertically by BOMB-SPEED
    (define/public (after-tick)
      (new Bomb% [x x][y (+ y BOMB-SPEED)]))
```

```
;; to-scene : Scene -> Scene
;; RETURNS: a scene like the given one, but with
;; this bomb painted on it.
(define/public (add-to-scene scene)
  (place-image BOMB-IMG x y scene))

;; the bomb doesn't have any other behaviors, so it
;; responds to each of these messages by returning
;; itself, unchanged.
(define/public (after-button-down mx my) this)
(define/public (after-drag mx my) this)
(define/public (after-button-up mx my) this)
(define/public (after-key-event kev) this)

;; test methods, to test; the bomb state.
(define/public (for-test:x) x)
(define/public (for-test:y) y)

))
```

after-tick returns a new bomb in the right location. Since **Bomb%** implements **Widget<%>**, the value returned is a Widget, so this method satisfies the contract given for it in the **Widget<%>** interface.

for-test:x and **for-test:y** are NOT in the **Widget<%>** interface. They are added here for testing purposes ONLY.

…and on to Heli%

```
;; A Heli is a (new Heli% [x Integer][y Integer]
;;                        [selected? Boolean]
;;                        [saved-mx Integer]
;;                        [saved-my Integer])
;; A Heli represents a heli.
(define Heli%
  (class* object% (Widget<%>)

    ;; the init-fields are the values that may vary
    ;; from one heli to the next.

    ; the x and y position of the center of the heli
    (init-field x y)

    ; is the heli selected? Default is false.
    (init-field [selected? false])

    ;; if the heli is selected, the position of
    ;; the last button-down event inside the heli,
    ;; relative to the heli's center.  Else any value.
    (init-field [saved-mx 0] [saved-my 0])
```

Note how we've put an interpretation on each of the fields of the object. This is just like what we did when we put an interpretation on each of the fields of a **struct**.

```
;; private data for objects of this class.
;; these can depend on the init-fields.

; the heli's radius
(field [r 15])
; image for displaying the heli
(field [HELI-IMG (circle r "outline" "blue")])
; the heli's speed, in pixels/tick.
; negative means that it moves upwards.
(field [HELISPEED -4])

(super-new)

;; after-tick : -> Heli
;; RETURNS: A heli like this one, but as it should
;; be after a tick.
;; DETAILS: a selected heli doesn't move. An
;; unselected heli moves vertically by HELISPEED.
;; STRATEGY: Cases on selected?
(define/public (after-tick)
  (if selected?
    this
    (new Heli%
      [x x]
      [y (+ y HELISPEED)]
      [selected? selected?]
      [saved-mx saved-mx]
      [saved-my saved-my])))
```

Heli% (2)

```
;; after-key-event : KeyEvent -> Heli
;; RETURNS: A heli like this one, but as it should
;; be after the given key event.
;; DETAILS: a heli ignores key events
(define/public (after-key-event kev)
  this)

;; after-button-down : Integer Integer -> Heli
;; GIVEN: the location of a button-down event
;; RETURNS: A heli like this one, but as it should
;; be after a button-down event at the given
;; location.
;; STRATEGY: Cases on whether the event is in the
;; helicopter
(define/public (after-button-down mx my)
  (if (in-heli? mx my)
    (new Heli%
      [x x][y y]
      [selected? true]
      [saved-mx (- mx x)]
      [saved-my (- my y)])
    this))
```

```
;; after-button-up : Integer Integer -> Heli
;; GIVEN: the location of a button-up event
;; RETURNS: A heli like this one, but as it should
;; be after a button-up event at the given
;; location.
;; DETAILS: If the heli is selected, then unselect
;; it, otherwise ignore.
;; STRATEGY: Cases on whether the event is in the
;; helicopter.
(define/public (after-button-up mx my)
  ...etc...)


;; after-drag : Integer Integer -> Heli
;; GIVEN: the location of a drag event
;; ...etc...
(define/public (after-drag mx my) ...etc...)


;; to-scene : Scene -> Scene
;; RETURNS: a scene like the given one, but with
;; this heli painted on it.
(define/public (add-to-scene scene)
  (place-image HELI-IMG x y scene))
```

Heli% (3)

```
;; in-heli? : Integer Integer -> Boolean
    ;; GIVEN: a location on the canvas
    ;; RETURNS: true iff the location is inside this
    ;; heli.

    (define (in-heli? other-x other-y)
      (<= (+ (sqr (- x other-x)) (sqr (- y other-y)))
          (sqr r)))

    ;; test methods, to probe the heli state.
    ;; Note that we don't have a probe for radius.

    ;; -> Int
    (define/public (for-test:x) x)
    ;; -> Int
    (define/public (for-test:y) y)
    ;; -> Boolean
    (define/public (for-test:selected?) selected?)
    ;; -> (list Int Int Boolean)
    (define/public (for-test:heli-state)
      (list x y selected?))

    ))
```

Let’s do it again

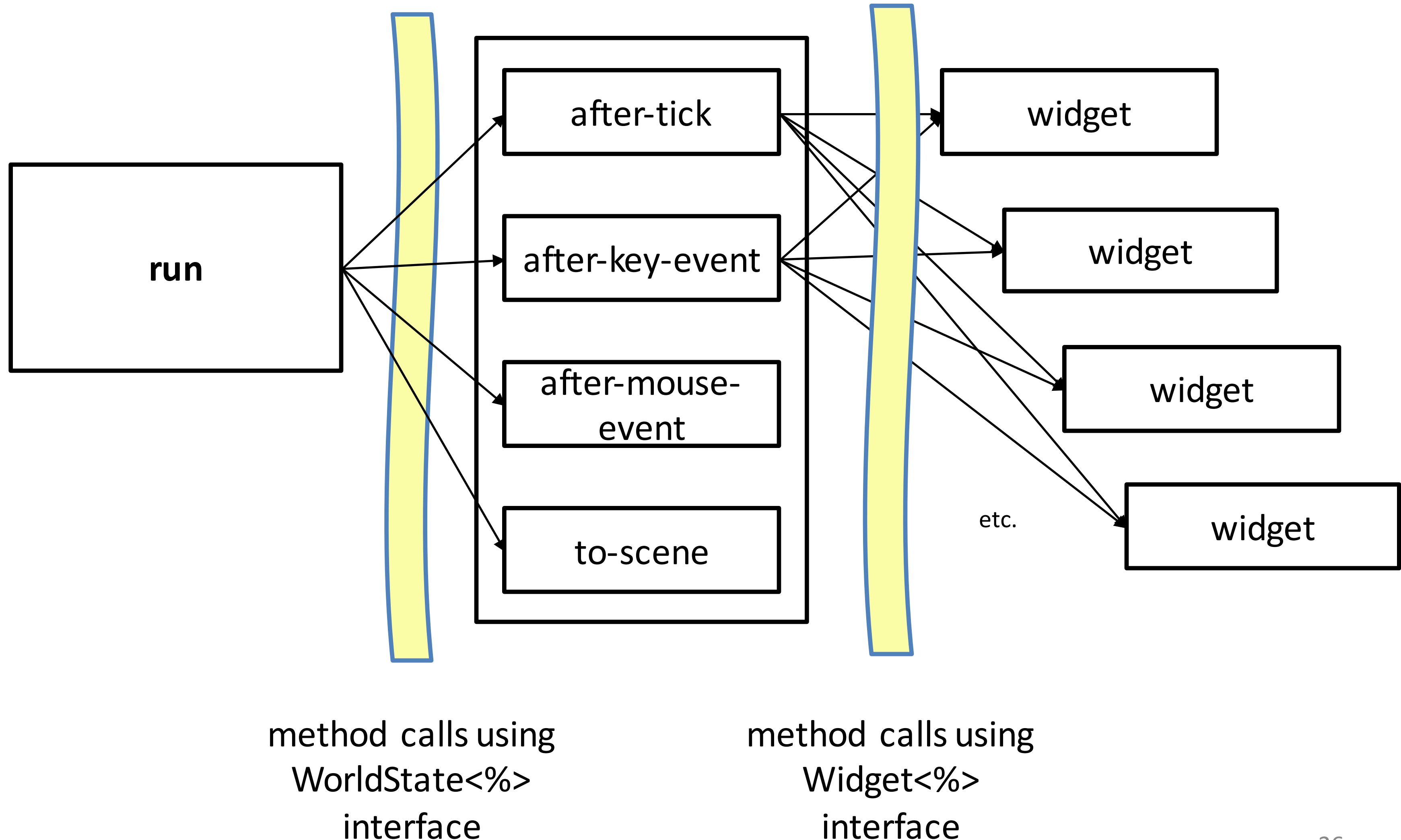
- Let’s make the World into an object.
- We’ll write a WorldState<%> interface and a class WorldState% that implements it.
- We’ll create an initial world, which is an object of class WorldState%.
- Our big-bang function will send messages to the world.

Don’t get agitated about World vs WorldState. I’ve not been entirely consistent about this. ☹

The WorldState<%> interface

```
(define WorldState<%>
  (interface ()

    ; -> WorldState<%>
    ; GIVEN: no arguments
    ; RETURNS: the state of the world at the next tick
    after-tick

    ; Integer Integer MouseEvent-> WorldState<%>
    ; GIVEN: a location
    ; RETURNS: the state of the world that should follow the
    ; given mouse event at the given location.
    after-mouse-event


    ; KeyEvent -> WorldState<%>
    ; GIVEN: a key event
    ; RETURNS: the state of the world that should follow the
    ; given key event
    after-key-event

    ; -> Scene
    ; GIVEN: a scene
    ; RETURNS: a scene that depicts the world
    to-scene
    ))
```

System Architecture



The new **run** function

```
; run : PosReal -> World
; GIVEN: a frame rate, in secs/tick
; EFFECT: runs an initial world at the given frame rate
; RETURNS: the final state of the world
(define (run rate)
  (big-bang (initial-world)
    (on-tick
      (lambda (w) (send w after-tick))
      rate)
    (on-draw
      (lambda (w) (send w to-scene)))
    (on-key
      (lambda (w kev)
        (send w after-key-event kev)))
    (on-mouse
      (lambda (w mx my mev)
        (send w after-mouse-event mx my mev)))))
```

Compare this to the **run** function in slide 13.

The class WorldState%

```
;; A WorldState is a
;;  (make-world-state ListOfWidget Time)

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

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

    (init-field objs) ;  ListOfWidget
    (init-field t)    ;  Time

    (super-new)

    ;; after-tick : -> WorldState<%>
    ;; Use HOF map on the Widgets in this World

    (define/public (after-tick)
      (make-world-state
        (map
          (lambda (obj) (send obj after-tick))
          objs)
        (+ 1 t)))
```

```
;; to-scene : -> Scene
;; Use HOF foldr on the Widgets in this World

(define/public (to-scene)
  (foldr
    (lambda (obj scene)
      (send obj add-to-scene scene))
    EMPTY-CANVAS
    objs))


;; after-key-event : KeyEvent -> WorldState<%>
;; STRATEGY: Cases on kev

(define/public (after-key-event kev)
  ...)
```

We define a function make-world-state so we can reuse the code from our previous version.

The class WorldState% (2)

```
;; world-after-mouse-event
;; : Nat Nat MouseEvent -> WorldState<%>
;; STRATGY: Cases on mev
(define/public (after-mouse-event mx my mev)
  (cond
    [(mouse=? mev "button-down")
     (world-after-button-down mx my)]
    [(mouse=? mev "drag")
     (world-after-drag mx my)]
    [(mouse=? mev "button-up")
     (world-after-button-up mx my)]
    [else this]))

;; the next few functions are local functions,
;; not in the interface.

(define (world-after-button-down mx my)
  (make-world-state
    (map
      (lambda (obj)
       (send obj after-button-down mx my))
      objs)
    t))

(define (world-after-button-up mx my)
     (make-world-state
       (map
         (lambda (obj)
          (send obj after-button-up mx my))
         objs)
       t))

   (define (world-after-drag mx my)
     (make-world-state
       (map
         (lambda (obj)
          (send obj after-drag mx my))
         objs)
       t))

   ))
```

Why do it this way?

- The Widget<%> interface didn't change, so we didn't need any other changes in the code.
- Not much difference in this example, but making the World into an object will become important next week.

Summary

- We’ve seen how an interface can be used to express an API that works for objects of several classes
- We’ve seen two designs of a small example that illustrate the use of interfaces.

Next Steps

- Study the files in the Examples folder:
  - 09-2-space-invaders-1.rkt
  - 09-2A-space-invaders-2.rkt
- If you have questions about this lesson, ask them on the Discussion Board
- Go on to the next lesson