Interfaces

CS 5010 Program Design Paradigms
"Bootcamp"
Lesson 10.2
Goals of this lesson

• See how an interfaces defines a common API for objects of possibly different classes.
• Learn how interfaces are expressed in the Racket object system
Our classes all responded to the same messages

- **World%**, **Bomb%**, and **Heli%** all respond to
  - after-tick
  - after-mouse-event
  - after-key-event
  - to-scene

- If we wanted to add another class of object to our system, we would probably want it to respond to these same messages.
Our classes all responded to the same messages (2)

• Furthermore, these methods all had similar contracts and similar purpose statements.

• If we wanted to add another class of object to our system, we would probably want its versions of these methods to have signatures and purpose statements that are similar to the existing ones.
Interfaces

• Remember: we are always looking for ways to document the design of our system

• We document this part of our design as an interface.

• An interface describes a group of classes with the same functions on them.

• World, Heli, Bombs, Bomb all implement this interface.
The **WorldObj<%>** interface

```racket
(define WorldObj<%> (interface ()
  ; -> WorldObj<%>
  ; GIVEN: no arguments
  ; RETURNS: the state of this object that should follow a
  ; tick.
  after-tick

  ; Number Number MouseEvent -> WorldObj<%>
  ; GIVEN: a location and a mouse event
  ; RETURNS: the state of this object that should follow the
  ; mouse event.
  after-mouse-event

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

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

In Racket, we write down an interface like this.

The convention is that names of interfaces end in `<%>`, which is silent, like `%`.

The interface lists the names of the methods that a class needs to have to satisfy this interface.

We list the signature and purpose statement for each method, in comments.
Function Signatures vs Method Signatures

In the functional version, we would write things like

```
;; bomb-after-key-event
;; : Bomb KeyEvent -> Bomb
```

In the OO version, our method signature is just

```
;; after-key-event
;; : KeyEvent -> WorldObj<%>
```

Instead of working on "the given bomb", the method works on "this object".
Connecting a class and an interface

(define World% (class* object% (WorldObj<%>))

; a Helicopter -- the helicopter in the game
(init-field heli)

; a ListOf<Bomb> -- the list of bombs that
; the UFO has dropped.
(init-field bombs)

...))

This specifies that the class World% implements the WorldObj<%> interface. (Remember, this used to be ()).

If we forget to include one of the required methods in WorldObj<%>, we'll get an error message at load time.

Interfaces make no difference at run time. They are part of the design. They are one of the very few design documents that Racket will enforce.
Interfaces Open Up New Possibilities

• In space-invaders, when you sent a message you always knew exactly what class the target was in.

• If you know what interface an object has, you can send a message to it, even if you don't know its class.

This is called "static dispatch".

This is called "dynamic dispatch".
Example using dynamic dispatch

- Let's imagine a system for manipulating shapes.
- We will do this first in the usual functional model, and then re-do it using an OO representation.
System Requirements

• Represent three kinds of shapes:
  – circle,
  – square
  – composite of two shapes
• Operations on shapes
  – weight : Shape -> Number
    • RETURNS: the weight of the given shape, assuming that each shape weighs 1 gram per pixel of area
  – add-to-scene : Shape Scene -> Scene
    • RETURNS: a scene like the given one, except that the given shape has been painted on it.
Video demos

- part 1 http://youtu.be/KRGiP0J62NU (4:51)
- part 2 http://youtu.be/ZUWRUScRpYs (6:13)

Again, these videos are a little old. They talk about “domain knowledge” and contain other things that don’t quite meet our current coding standards, sorry.
Self-Referential Data

(define composite%
  (class* object% (Shape<%>))
  (init-field front  ; Shape, the shape in front
   back)  ; Shape, the shape in back

  (super-new)

  (define/public (weight) (+ (send front weight)
                              (send back weight)))

  ;; all we know here is that front and back implement Shape<%>.  
  ;; we don't know if they are circles, squares, or other  
  ;; composites!

  (define/public (add-to-scene scene)
    (send front add-to-scene
     (send back add-to-scene scene)))
)

Observe how self-referential data appears in the OO version.

This is called the composite pattern

Recursion!

Doesn't care what kind of shape these are: it just works!
Interfaces and Signatures

• When a function takes an object as an argument, all it can do is to send messages to it.

• There might be many classes that accept the same set of messages.

• So interface method signatures should always be in terms of interfaces, not classes.

• We’ll see examples of this later.
Summary

• We’ve seen how an interface can be used to express an API that works for objects of several classes
• We’ve seen how to build recursive data using the composite pattern.