Functions vs. Classes

CS 5010 Program Design Paradigms

"Bootcamp"

Lesson 9.3

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

Goals for this Lesson

- In this lesson, we'll illustrate the relationship between the functional version of the shapes and the object-oriented version.

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.

Code outline (functional version)

```
(define-struct my-circle (x y r color))
(define-struct my-square (x y l color))
(define-struct my-composite (front back))

;; A Shape is one of
;; -- (make-my-circle Number Number Number ColorString)
;; -- (make-my-square Number Number Number ColorString)
;; -- (make-my-composite Shape Shape)
```

Code outline (2)

```
;; weight : Shape -> Number
;; GIVEN: a shape
;; RETURNS: the weight of the shape, assuming that each
;; shape weighs 1 gram per pixel of area.
;; STRATEGY: Use template for Shape on s

(define (weight s)
  (cond
    [(my-circle? s)    (my-circle-weight s)]
    [(my-square? s)    (my-square-weight s)]
    [(my-composite? s) (my-composite-weight s)]))

;; add-to-scene : Shape Scene -> Scene
;; RETURNS: a scene like the given one, but with the
;; given shape painted on it.
;; STRATEGY: Use template for Shape on s

(define (add-to-scene s scene)
  (cond
    [(my-circle? s) (my-circle-add-to-scene s scene)]
    [(my-square? s) (my-square-add-to-scene s scene)]
    [(my-composite? s) (my-composite-add-to-scene s scene)]))
```

In real code, I probably wouldn't break these out into help functions, but I've done it here to help make my point.

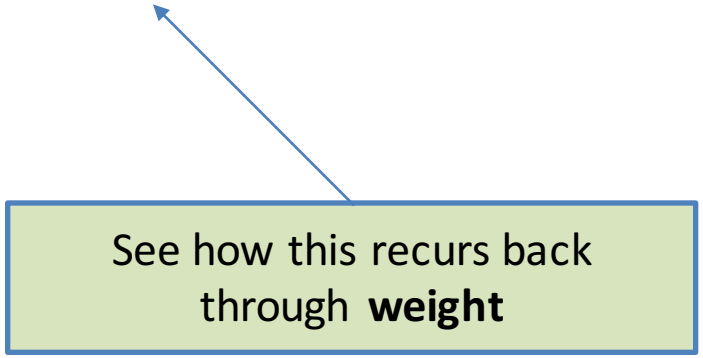
6 small functions left to write:

- my-circle-add-to-scene
- my-square-add-to-scene
- my-composite-add-to-scene
- my-circle-weight
- my-square-weight
- my-composite-weight

A few of the help functions

```
(define (my-circle-weight s) (* pi (my-circle-r s) (my-circle-r s)))
(define (my-square-weight s) (* (my-square-l s) (my-square-l s)))
(define (my-composite-weight s) (+ (weight (my-composite-front s))
                                   (weight (my-composite-back s))))

(define (my-composite-add-to-scene s scene)
  ;; paint the back image first,
  ;; then the front image
  (add-to-scene (my-composite-front s)
    (add-to-scene (my-composite-back s)
      scene)))
```



Code Outline (OO version)

```
;;; INTERFACE:

;; all geometric shapes support these methods in all contexts
;; a Shape is an object of a class that implements Shape<%>.

(define Shape<%>
  (interface ()

    ;; weight : -> Number
    ;; RETURNS: the weight of this shape
    weight

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

    ))
```

Code Outline (OO:2)

```
;; A Circle is a
;; (new Circle% [x Integer][y Integer]
;;              [r Integer][c ColorString])
;; REPRESENTS: a circle on the canvas
(define Circle%
  (class* object% (Shape<%>)
    (init-field
      x  ; Integer, x-position of center
      y  ; Integer, y-position of center
      r  ; Integer, radius
      c)  ; ColorString, color of circle

    (field [IMG (circle r "solid" c)])

    (super-new)
```

```
;; weight : -> Integer
;; RETURNS: the weight of this shape
;; DETAILS: this shape is a circle
;; STRATEGY: combine simpler functions
(define/public (weight) (* pi r r))

;; add-to-scene : Scene -> Scene
;; RETURNS: a scene like the given one,
;; but with this shape painted on it.
;; DETAILS: this shape is a circle
;; STRATEGY: call a more general function
(define/public (add-to-scene s)
   (place-image IMG x y s))

))
```

For each method, we copy down the contract and purpose statement from the interface, with perhaps additional details relating to this class.

Code Outline (OO:3)

```
;; A Square is a (new Square% [x Integer][y Integer][l Integer][c ColorString])

;; REPRESENTS: a square parallel to sides of canvas
(define Square%
  (class* object% (Shape<%>)
    (init-field x  ; Integer, x pixels of center from left
                y  ; Integer, y pixels of center from top
                l  ; Integer, length of one side
                c) ; ColorString

    (field [IMG (rectangle l l "solid" c)])

    (super-new)

    ;; weight : -> Real
    ;; RETURNS: the weight of this shape
    ;; DETAILS: this shape is a square
    ;; STRATEGY: combine simpler functions

    (define/public (weight) (* l l))

    ;; add-to-scene : Scene -> Scene
    ;; RETURNS: a scene like the given one, but with this shape
    ;; painted on it.
    ;; DETAILS: this shape is a square
    ;; STRATEGY: call a more general function

    (define/public (add-to-scene s) (place-image IMG x y s))

    ))
```

Code Outline (OO:4)

```
;; A Composite is a (new Composite% [front Shape][back Shape])
;; a composite of front and back
(define Composite%
  (class* object% (Shape<%>)
    (init-field
      front   ; Shape, the shape in front
      back    ; Shape, the shape in back
      )

    (super-new)

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

    ;; weight : -> Number
    ;; RETURNS: the weight of this shape
    ;; DETAILS: this shape is a composite
    ;; STRATEGY: recur on the components
    (define/public (weight) (+ (send front weight)
                               (send back weight)))

    ;; add-to-scene : Scene -> Scene
    ;; RETURNS: a scene like the given one, but with this shape
    ;; painted on it.
    ;; DETAILS: this shape is a composite
    ;; strategy: recur on the components
    (define/public (add-to-scene scene)
      (send front add-to-scene
        (send back add-to-scene scene)))

    ))
```

The Big Picture

- The functional version and the OO version are really the same. They just have the pieces grouped differently.
- Here are a couple of slides that illustrate what happened.
- We had 6 little functions to write. Let's see where they wound up in the functional version, and then in the OO version.

The Big Picture: Functional

my-circle-weight

my-square-weight

my-composite-weight

my-circle-add-to-scene

my-square-add-to-scene

my-composite-add-to-scene

define weight:

- my-circle-weight
- my-square-weight
- my-composite-weight

define add-to-scene:

- my-circle-add-to-scene
- my-square-add-to-scene
- my-composite-add-to-scene

When we call **weight** or **add-to-scene**, we use a **cond** expression to determine what kind of shape we were dealing with, so the appropriate code is evaluated.

The Big Picture: Classes

my-circle-weight

my-square-weight

my-composite-weight

my-circle-add-to-scene

my-square-add-to-scene

my-composite-add-to-scene

When we invoke a method on an object, the object already knows what class it belongs to, so the correct piece of code is evaluated directly. We no longer need to write a **cond**.

class circle:

- my-circle-weight
- my-circle-add-to-scene

class square:

- my-square-weight
- my-square-add-to-scene

class composite:

- my-composite-weight
- my-composite-add-to-scene

Functional vs. OO organization

Here's another way of visualizing the same thing. Here we have six small rectangles corresponding to our six pieces of functionality.

Functional:	Square	Circle	Composite
weight			
add-to-scene			

OO:	Square	Circle	Composite
weight			
add-to-scene			

In the functional organization, all the pieces corresponding to **weight** are written together (symbolized here by outlining them in red), and all the pieces corresponding to **add-to-scene** are written together (outlined in green).

In the object-oriented organization, all the pieces for **square** are written together (the red outline in the lower table), all the pieces for **circle** are written together (the orange outline), and all the pieces for composite are written together (the purple outline).

Adding a New Data Variant

If we add a new kind of data, such as a triangle, what will we need to change?

We will need 2 pieces of code: to compute the weight of a triangle and to display it.

Functional:	Square	Circle	Composite
weight			
add-to-scene			

OO:	Square	Circle	Composite
weight			
add-to-scene			

In the functional organization, the two cells correspond to different portions of our file, so we will need to edit two pieces of our file: the **weight** function and the **add-to-scene** function.

In the object-oriented organization, we will add the two pieces in a single place in our file: the new **triangle** class.

Adding a New Operation

Functional:	Square	Circle	Composite
weight			
add-to-scene			
move	new code 1	new code 2	new code 3

OO:	Square	Circle	Composite
weight			
add-to-scene			
move	new code 1	new code 2	new code 3

If we add a new operation such as **move**, what needs to change?

In the functional organization, we add the new code in a single function definition, the function **move**, symbolized by the blue outline above.

In the object-oriented organization, we must add a **move** method in each of our classes.

Extensibility

	Functional Org.	O-O Org.
New Data Variant	requires editing in many places	all edits in one place
New Operation	all edits in one place	requires editing in many places

What's the tradeoff?

- Object-oriented organization is better when new data variants are more likely than new operations.
- Functional organization is better when new operations are more likely than new data variants.
- In the real world, you may not have a choice:
  - this decision is up to the system architects
  - or may need compatibility with an existing system
- There are ways to get the best of both worlds
  - but these are beyond the scope of this course
  - this is called "the expression problem"

Summary

- You should now be able to draw diagrams that explain the organization of O-O programs vs. functional programs.

Next Steps

- Review examples 09-3 through 09-5 in the examples folder.
- If you have questions about this lesson, ask them on the Discussion Board
- Go on to the next lesson