Using Inheritance to Share Implementations

CS 5010 Program Design Paradigms

"Bootcamp"

Lesson 11.2

Key Points for Lesson 11.2

- By the end of this lesson you should be able to:
  - Identify common parts of class implementations
  - Generalize these common parts into a superclass
  - Identify the parts that differ, and turn these into abstract methods.
  - Recover the original classes using inheritance.

The Real Power of Inheritance

- The flashing-ball example was a good start, but it didn't illustrate the real power of inheritance.
- The real power of inheritance is that it enables you to abstract common parts of the *implementation* of similar classes.
- Let's try a somewhat more substantial example: 11-2-squares.rkt

Square%

```
(define Square%
  (class* object% (SBall<%>)

    (init-field w)  ;; the Wall that the square should
bounce off of

    ;; initial values of x, y (center of square)
    (init-field [x INIT-BALL-X])
    (init-field [y INIT-BALL-Y])
    (init-field [speed INIT-BALL-SPEED])

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

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

    (field [size 40])
    (field [half-size (/ size 2)])

    ;; register this square with the wall, and use the
    ;; result as the initial value of wall-pos
    (field [wall-pos (send w register this)])

    (super-new)
```

Square% is very similar to Ball%. I've marked the parts that are different in red. Everything else is the same.

```
;; Int -> Void
;; EFFECT: updates the square's idea of the wall's
;; position to the given integer.
(define/public (update-wall-pos n)
  (set! wall-pos n))

;; after-tick : -> Void
;; EFFECT: updates this square to the state it
;; should be in after a tick.
;; A selected square doesn't move.

(define/public (after-tick)
  (if selected?
    this
    (let ((x1     (next-x-pos))
          (speed1 (next-speed)))
      (begin
        (set! speed speed1)
        (set! x x1)))))

;; -> Integer
;; RETURNS: position of the square at the next tick
;; STRATEGY: use the square's cached copy of the
;;  wall position to set the upper limit of motion
(define (next-x-pos)
  (limit-value
    half-size
    (+ x speed)
    (-  wall-pos half-size)))
```

Square% (2)

```
;; Number^3 -> Number
;; WHERE: lo <= hi
;; RETURNS: val, but limited to the range [lo,hi]
(define (limit-value lo val hi)
  (max lo (min val hi)))

;; -> Integer
;; RETURNS: the velocity of the square at the next
;;  tick
;; STRATEGY: if the square will not be at its
;;  limit, return it unchanged. Otherwise, negate
;;  the velocity.
(define (next-speed)
  (if
    (< half-size
       (next-x-pos)
       (- wall-pos half-size))
    speed
    (- speed)))

(define/public (add-to-scene s)
  (place-image
    (square size
     (if selected? "solid" "outline")
     "green")
    x y s))
```

```
; after-button-down : Integer Integer -> Void
; GIVEN: the location of a button-down event
; STRATEGY: Cases on whether the event is in this
(define/public (after-button-down mx my)
  (if (in-this? mx my)
    (begin
      (set! selected? true)
      (set! saved-mx (- mx x))
      (set! saved-my (- my y)))
    this))

;; in-this? : Integer Integer -> Boolean
;; GIVEN: a location on the canvas
;; RETURNS: true iff the location is inside this.
(define (in-this? other-x other-y)
  (and
   (<= (- x half-size) other-x (+ x half-size))
   (<= (- y half-size) other-y (+ y half-size))))

; after-button-up : Integer Integer -> Void
; GIVEN: the location of a button-up event
; STRATEGY: Cases on whether the event is in this
; If this is selected, then unselect it.
(define/public (after-button-up mx my)
  (if (in-this? mx my)
    (set! selected? false)
    this))
```

Square% (3)

```
; after-drag : Integer Integer -> Void
; GIVEN: the location of a drag event
; STRATEGY: Cases on whether the square is
;  selected. If it is selected, move it so that the
;  vector from the center to the drag event is
;  equal to (mx, my)
(define/public (after-drag mx my)
  (if selected?
    (begin
      (set! x (- mx saved-mx))
      (set! y (- my saved-my)))
    this))

;; the square ignores key events
;; returns a nonsense value
(define/public (after-key-event kev) 23)

(define/public (for-test:x)          x)
(define/public (for-test:speed)      speed)
(define/public (for-test:wall-pos)   wall-pos)
(define/public (for-test:next-speed) (next-speed))
(define/public (for-test:next-x)     (next-x-pos))


))
```

Can we unify the common code?

- Looking at **`Square%`** and **`Ball%`**, we see that many of the method definitions have a lot in common.
- Let's try to move the common parts into a new class, which we'll call **`DraggableWidget%`**.
- Then we'll have **`Square%`** and **`Ball%`** both inherit from **`DraggableWidget%`**.
- Let's see what happens:

What about the methods that are different?

- We turn these into *abstract methods*.
- An abstract method is a method that is not defined in the superclass, but must be defined in any subclass that will have objects.
- For example, we write **(abstract add-to-scene)**.
- This declares add-to-scene to be an abstract method.
  - To *declare* a name means to introduce a name so that it can be referred to in the code, but to leave the definition until later
  - In this case, the definition will be supplied by the subclass.
- Let's look at the code:

We sometimes call abstract methods "hooks" because they act like hooks on which we can hang code in the subclass.

DraggableWidget% (1)

```
(define DraggableWidget%
  (class* object%

    (SBall<%>)

    ;; the Wall that the ball should bounce off of
    (init-field w)

    ;; initial values of x, y (center of ball)
    (init-field [x INIT-BALL-X])
    (init-field [y INIT-BALL-Y])
    (init-field [speed INIT-BALL-SPEED])

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

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

    ;; this is specific to Ball%
    ; (field [radius 20])
```

```
;; register this ball with the wall, and use the
;; result as the initial value of wall-pos
(field [wall-pos (send w register this)])

(super-new)

;; Int -> Void
;; EFFECT: updates the widget's idea of the wall's
;;   position by setting it to the given integer.
(define/public (update-wall-pos n)
  (set! wall-pos n))


;; after-tick : -> Void
;; EFFECT: updates this widget to the state it
;; should be in after a tick.
(define/public (after-tick)
  (if selected?
    this
    (let ((x1     (next-x-pos))
          (speed1 (next-speed)))
      (begin
        (set! speed speed1)
        (set! x x1)))))
```

Here we've defined the **DraggableWidget%** class by taking the **Ball%** class and commenting out all the code that is specific to **Ball%.**

DraggableWidget% (2)

```
    ;; -> Integer
    ;; position of the object at the next tick
    ;; (define (next-x-pos)
    ;;   (limit-value
    ;;     radius
    ;;     (+ x speed)
    ;;     (-  wall-pos radius)))
```

```
    ;; Number^3 -> Number
    ;; WHERE: lo <= hi
    ;; RETURNS: val, but limited to the range [lo,hi]
    (define (limit-value lo val hi)
      (max lo (min val hi)))
```

```
    ;; -> Integer
    ;; RETURNS: the velocity of the widget at the next
    ;;   tick
    ;; STRATEGY: if the ball will not be at its limit,
    ;;  return velocity unchanged. Otherwise, negate
    ;;  the velocity.
    ;; (define (next-speed)
    ;;   (if
    ;;     (< radius (next-x-pos) (- wall-pos radius))
    ;;     speed
    ;;     (- speed)))
```

```
    ;; also ball-specific
    ;; (define/public (add-to-scene s)
    ;;   (place-image
    ;;     (circle radius
    ;;       "outline"
    ;;       "red")
    ;;     x y s))
```

```
    ;; but we need to declare add-to-scene, so that
    ;; we'll satisfy the interface:
    (abstract add-to-scene)

    ; after-button-down : Integer Integer -> Void
    ; GIVEN: the location of a button-down event
    ; STRATEGY: Cases on whether the event is in this
    (define/public (after-button-down mx my)
      (if (in-this? mx my)
        (begin
          (set! selected? true)
          (set! saved-mx (- mx x))
          (set! saved-my (- my y)))
        this))
```

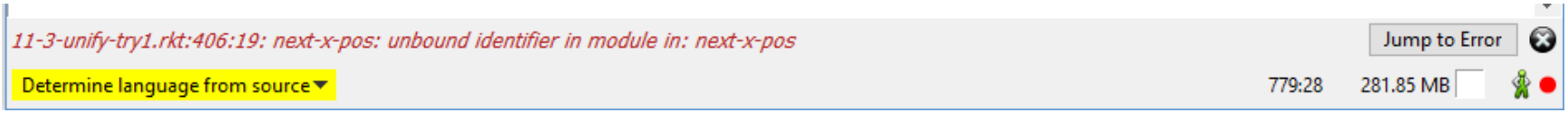
By writing **(abstract add-to-scene)**, we declare **add-to-scene** to be an abstract method. This is needed to make **DraggableWidget%** satisfy the **SBall<%>** interface. It also means that the system will complain if we try to create an object of class **DraggableWidget%** .

DraggableWidget% (3)

```
    ;; also circle-specific
    ;; in-this? : Integer Integer -> Boolean
    ;; GIVEN: a location on the canvas
    ;; RETURNS: true iff the location is inside this.
    ;; (define (in-this? other-x other-y)
    ;;   (<= (+ (sqr (- x other-x))
    ;;          (sqr (- y other-y)))
    ;;       (sqr radius)))

    ; after-button-up : Integer Integer -> Void
    ; GIVEN: the location of a button-up event
    ; STRATEGY: Cases on whether the event is in this
    ; If this is selected, then unselect it.
    (define/public (after-button-up mx my)
      (if (in-this? mx my)
        (set! selected? false)
        this))

    ; after-drag : Integer Integer -> Void
    ; GIVEN: the location of a drag event
    ; STRATEGY: Cases on whether the ball is selected.
    ; If it is selected, move it so that the vector
    ; from the center to the drag event is equal to
    ; (mx, my)
    (define/public (after-drag mx my)
      (if selected?
        (begin
          (set! x (- mx saved-mx))
          (set! y (- my saved-my)))
        this))
```

```
    ;; the widget ignores key events
    (define/public (after-key-event kev) 23)

    (define/public (for-test:x)          x)
    (define/public (for-test:speed)      speed)
    (define/public (for-test:wall-pos)   wall-pos)
    (define/public (for-test:next-speed) (next-speed))
    (define/public (for-test:next-x)     (next-x-pos))

    ))
```

But look what happens when we try to run this!



- What happened?
- **next-x-pos** is not defined in **DraggableWidget%** .
- How can we put it where it will be found?

Remember the story about inheritance and **this**

- If a method in the superclass refers to **this**, where do you look for the method?
- Answer: in the original object.
- Consider the following class hierarchy:

Searching for a method of **this**

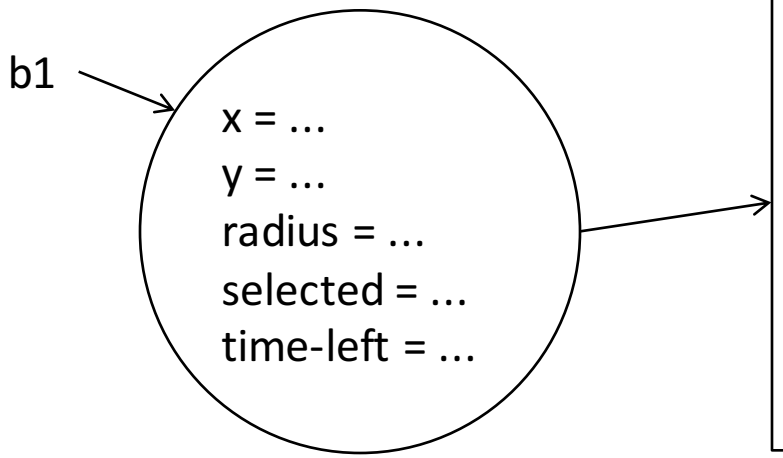
```
(define b1 (new FlashingBall% ...))
(send b1 m1 33)
```

When we send **b1** an **m1** message, what happens?

1) It searches its own methods for an **m1** method, and finds none.
2) It searches it superclass for an **m1** method. This time it finds one, which says to send **this** an **m2** message.
3) **this** still refers to **b1**. So **b1** starts searching for an **m2** method.
4) It finds the m2 method in its local table, and returns the string "right".

```
Ball% = (class* object% (SBall<%>)
 (field x y radius selected?)
 (define/public  (m1 x) (send this m2 x))
 (define/public  (m2 x) "wrong")

 )
```

```
FlashingBall% = (class* Ball% (SBall<%>)

(define/override  (m2 x) "right")
...)
```



Solution:

- Make **next-x-pos** a method of **Ball%** and **Square%**, and say

```
(let ((x1 (send this next-x-pos))
      ..etc..
```

The fix for next-x-pos

In DraggableWidget%:

```
(define/public (after-tick)
     (if selected?
       this
       (let ((x1     (send this next-x-pos) )
             (speed1 (send this next-speed)))
         (begin
           (set! speed speed1)
           (set! x x1)))))

   ;; to be supplied by the subclasses
   (abstract next-x-pos)
```

In Ball%:

```
    (define/override (next-x-pos)
      (limit-value
        radius
        (+ x speed)
        (-  wall-pos radius)))
```

In Square%:

```
    (define/override (next-x-pos)
      (limit-value
        half-size
        (+ x speed)
        (-  wall-pos half-size)))
```

In the superclass, we say **(send this next-x-pos)** and **(abstract next-x-pos)**, and in each subclass, we provide a definition of a **next-x-pos** method that works for that subclass.

We define the method in the subclass using **define/override**, since it is overriding a method declared in the superclass. This is a peculiarity of Racket; other object systems may have different syntax or different conventions for this situation.

And we do the same thing for each other function that is different in each subclass. In this example, these are **in-this?** and **next-speed**.

We document this by adding a new interface

```
;; Open hooks (abstract methods): these
;; methods must be supplied by each subclass.

(define DraggableWidgetHooks<%>
  (interface ()

    ;; Int Int -> Boolean
    ;; is the given location in this widget?
    in-this?

    ;; -> Int
    ;; RETURN: the next x position or speed
    ;;  of this widget
    next-x-pos
    next-speed

    ))
```

```
;; We require each subclass to implement the
;; Hooks interface of its superclass:

(define Ball%
  (class* DraggableWidget%

    (SBall<%> DraggableWidgetHooks<%>)

    ..etc..


    ))
```

As always, we need to document this design. We do this by adding a new interface that lists the methods that must be supplied by any subclass of DraggableWidget%.

We didn't include add-to-scene in this list because it's already in SBall<%>

What we have accomplished so far

So now the only differences between **`Ball%`** and **`Square%`** are in methods:

`add-to-scene`
`in-this?`
`next-x-pos`
`next-speed`

These are the methods that deal with the geometry of squares and balls so naturally they will be different. Everything else is taken care of in the superclass.

So Ball% and Square% consist only of these methods and the fields and functions they depend on.

New Ball% class

```
(define Ball%
  (class* DraggableWidget%

    ;; must implement SBall + the open hooks from the
superclass
    (SBall<%> DraggableWidgetHooks<%>)

    ;; inherit all these fields from the superclass:

    ;; initial values of x, y (center of ball) and
    ;; speed:
    (inherit-field x y speed)

    ;; position of the wall, updated by update-wall-pos
    (inherit-field wall-pos)

    ;; this field is local to Ball%
    (field [radius 20])

    (super-new)

    ;; -> Integer
    ;; position of the ball at the next tick
    (define/override (next-x-pos)
      (limit-value
        radius
        (+ x speed)
        (-  wall-pos radius)))
```

```
;; Number^3 -> Number
;; WHERE: lo <= hi
;; RETURNS: val, but limited to the range [lo,hi]
(define (limit-value lo val hi)
  (max lo (min val hi)))

;; -> Integer
;; RETURNS: the velocity of the ball at the next tick
(define/override (next-speed)
  (if
    (< radius (next-x-pos) (- wall-pos radius))
    speed
    (- speed)))

(define/override (add-to-scene s)
  (place-image
    (circle radius
      "outline"
      "red")
    x y s))

;; in-this? : Integer Integer -> Boolean
;; GIVEN: a location on the canvas
;; RETURNS: true iff the location is inside this.
(define/override (in-this? other-x other-y)
  (<= (+ (sqr (- x other-x)) (sqr (- y other-y)))
      (sqr radius)))

))
```

The new Ball% class consists only of things that are specific to Balls. All the things that are in common with Squares have been moved up to their generalization DraggableWidget%.

The Process in Pictures

- We start with the two classes **`Ball%`** and **`Square%`**. The black parts are the same and the red parts are different.

```
Ball% =
(class* object% (SBall<%>)
 (field x y)
 (define radius ...)

 (define/public (add-to-scene s)
  ...)

 (define/public
  (after-button-down mx my)
  ...(in-this? mx my)))

 (define/public (after-tick)
  ...(next-x-pos)...)

 (define (in-this? mx my) ...)

 (define (next-x-pos) ...)  )
```

```
Square% =
(class* object% (SBall<%>)
 (field x y)
 (define size ...)

 (define/public (add-to-scene s)
  ...)

 (define/public
  (after-button-down mx my)
  ...(in-this? mx my)))

 (define/public (on-tick)
  ...(next-x-pos)...)

 (define (in-this? mx my) ...)

 (define (next-x-pos) ...)  )
```

Starting Code

Step 1: Turn differing functions into methods

- The first thing we do is to turn the differing functions into methods. Each call **(f arg)** is replaced by **(send this f arg)** .
- This only comes up because Racket has both methods and functions.
- If we were in a language where everything was a method, this wouldn't be an issue.

Step 2: Move Common Methods into a Superclass

- We move the common methods into a superclass. We can think of the common method in the superclass as an abstraction or generalization of the methods in the classes.

Step 3: Make different methods into abstract methods

- In the past, we generalized a set of functions by writing a single function with an extra argument. Depending on the value of the extra argument, we could get back one of our original functions.
- Now instead of two functions, we have two methods, which differ only by being in two different classes.
- When we move the method into the superclass, the single method can behave like either of the original two methods.
- We don't give the generalized method an extra argument. Instead, depending on which class the method is called from, we get back the behavior of one of our original methods.
- We call this "*specialization by subclassing*."

Specialization in **11-4-turn-differences-into-methods.rkt**

- In this example, the on-mouse method in DraggableObj% will behave like the original on-mouse method of Ball% if it is called from Ball%. It will behave like the original on-mouse method of Square% if it is called from Square%.
- Let's see how this works.

```
DraggableWidget% = (class* object%
  (SBall<%>)
  (field x y)

  (define/public
   (after-button-down mx my mev)
   ...(send this in-this? mx my)...))

  (define/public (on-tick)
    ...(send this next-x-pos)...)

  )
```

```
Ball% = (class* DraggableWidget%
(SBall<%>
 DraggableWidgetHooks<%>)
(inherit-field x y)
(define radius ...)

(define/public (add-to-scene s)
  ...)

(define/public
  (in-this? mx my) ...)

(define/public
  (next-x-pos) ...)   )
```

```
Square% = (class* DraggableWidget%
(SBall<%>
 DraggableWidgetHooks<%>)
(inherit-field x y)
(define size ...)

(define/method (add-to-scene s)
  ...)

(define/public
  (in-this? mx my) ...)

(define/public
  (next-x-pos) ...)   )
```

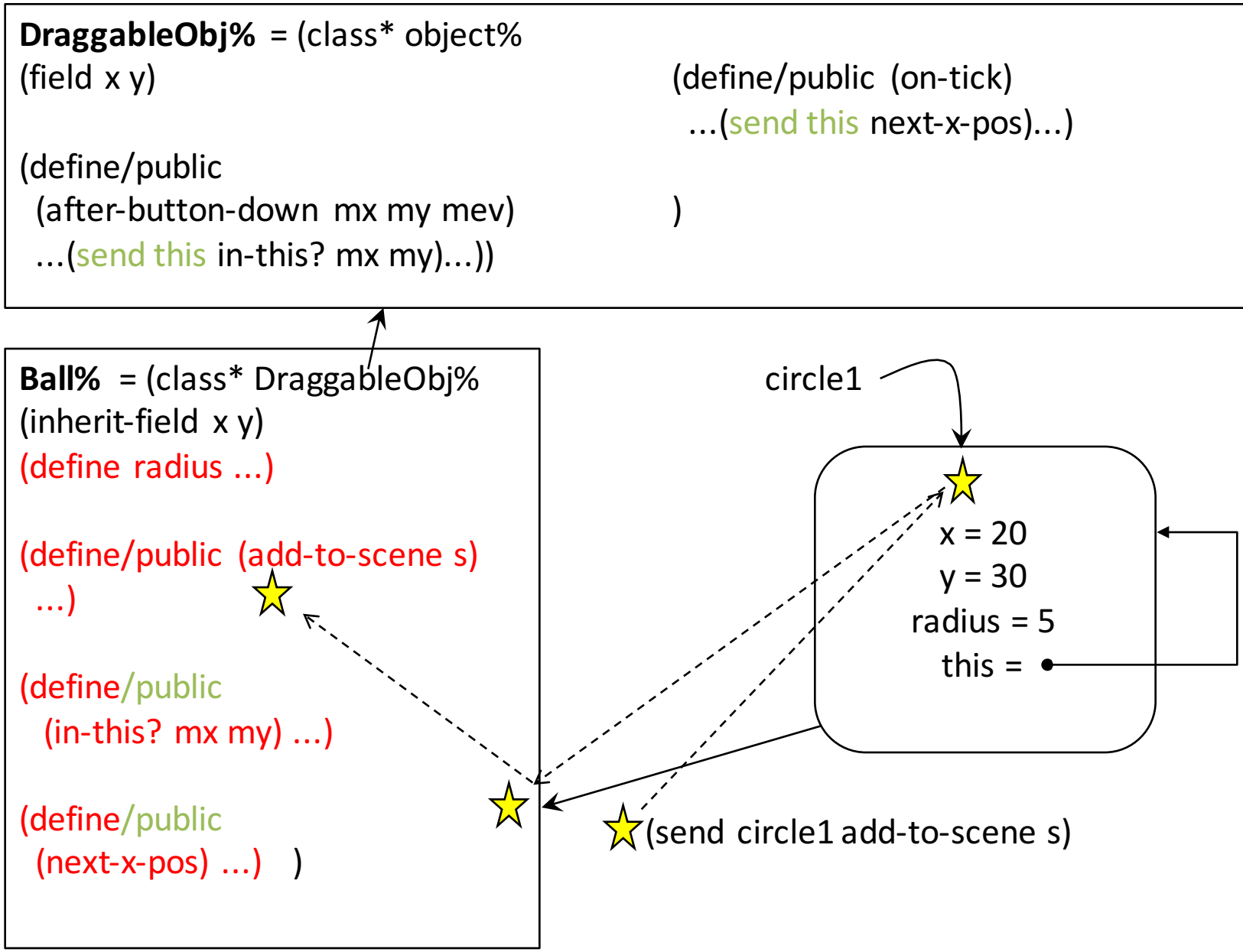
Move common methods into superclass

What this accomplishes

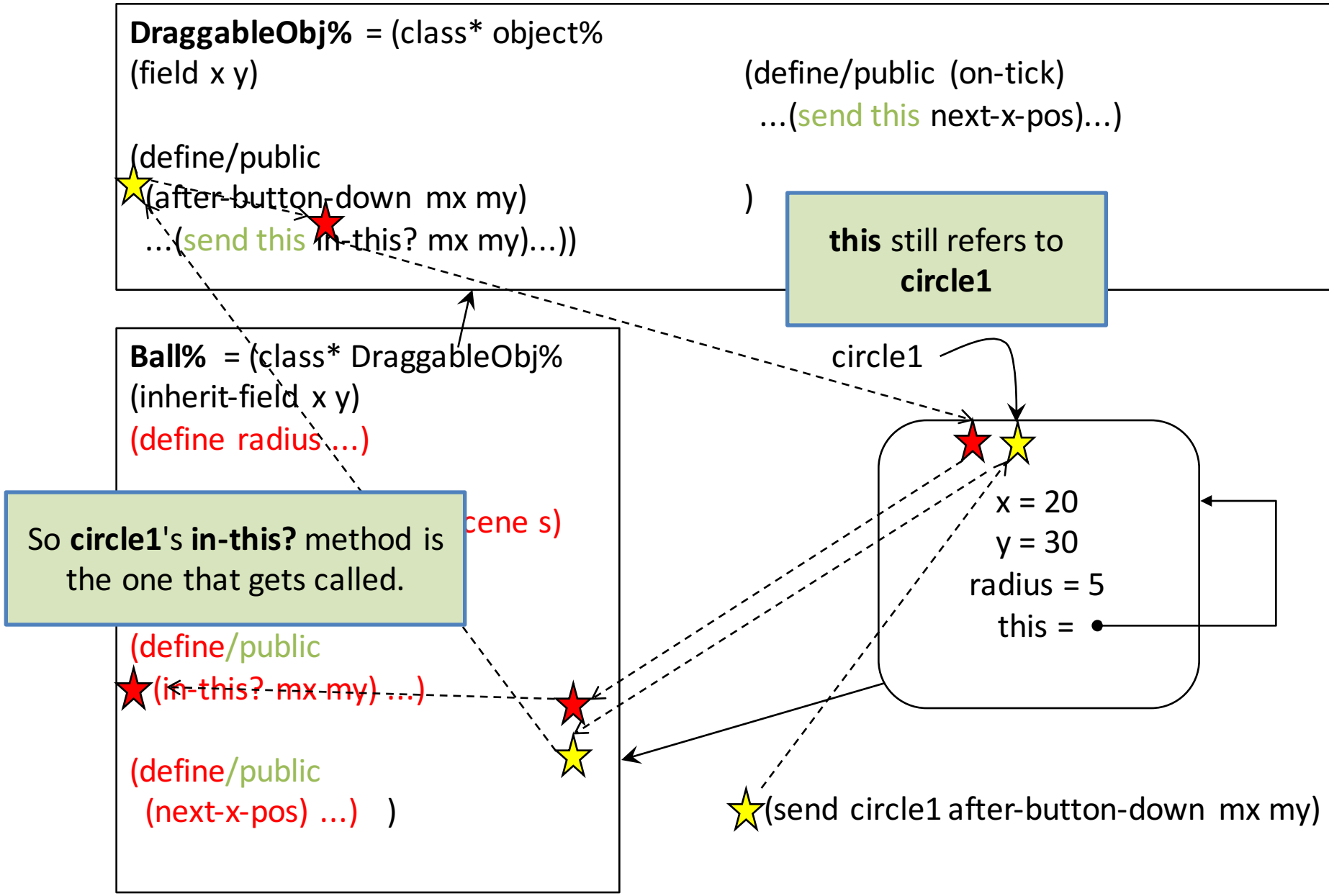
- We can think of the common method in the superclass as an abstraction or generalization of the methods in the classes.
- In the past, we generalized a set of functions by writing a single function with an extra argument. Depending on the value of the extra argument, we could get back one of our original functions.
- Now instead of two functions, we have two methods, which differ only by being in two different classes.
- When we move the method into the superclass, the single method can behave like either of the original two methods.

Subclassing in Action

- The animation on the next slide shows how sending a circle an **after-button-down** message winds up calling the circle's version of **in-this?**
- If we sent a square an **after-button-down** message, then we would wind up calling the square's version of **in-this?**, in exactly the same way.



Every object knows its own methods #1



Every object knows its own methods #2

4. Generalize Similar Methods by Adding Abstract Methods for the Differences

- We can do the same thing with methods that differ only in small ways.
- We move the common part of the method into the superclass, and have it refer to the differing parts by calling a method in the subclass.
- Here's an example.

Before:

In DraggableObject%:

```
(abstract add-to-scene)
```

In Ball%:

```
(define/override (add-to-scene s)
  (place-image
   (circle radius
           (if selected? "solid" "outline")
           "red")
    x y s))
```

In Square%:

```
(define/override (add-to-scene s)
  (place-image
   (square size
           (if selected? "solid" "outline")
           "green")
    x y s))
```

abstract creates an abstract method, so that **DraggbleObject%** will satisfy **StatefulWorldObj<%>** .
An **abstract** method must be defined by a **define/override** in every subclass.

After:

In DraggableObject%:

```
(define/public (add-to-scene s)
   (place-image
     (send this get-image)
     x y s))

 (abstract get-image)
```

In Ball%:

```
(define/override (get-image)
  (circle radius
          (if selected? "solid" "outline")
          "red"))
```

In Square%:

```
(define/override (get-image)
  (square size
         (if selected? "solid" "outline")
         "green"))
```

add-to-scene is now in the superclass. It uses an abstract method called **get-image** to retrieve the image.
Each subclass must provide a definition for **get-image**.

This is the *Template and Hook* pattern

- The superclass has incomplete behavior.
  - Superclasses leave *hooks* to be filled in by subclass.
- Parameterize a superclass by inheritance
- Subclasses supply methods for the hooks; these methods are called "at the right time"
- This is how "frameworks" work. A framework typically consists of a large set of general-purpose classes that you specialize by subclassing. Each subclass contains special purpose methods that describe the specialized behavior of objects of that subclass.

big-bang is sort of like this: you tell it what the hook functions are for each event and it calls each function when the event occurs.

Summary: Recipe for generalizing similar classes

1. Turn differing functions into methods
2. Move identical methods into a superclass
3. Make different methods into abstract methods
4. Generalize similar methods by adding abstract methods for the differences

Summary of the Files

- Study the relevant files in the examples folder:
  - 11-1-flashing-balls.rkt
  - 11-2-squares.rkt
  - 11-3-unify-try1.rkt
  - 11-4-turn-differences-into-methods.rkt
  - 11-5-generalize-methods-in-superclass.rkt
  - 11-6-after-review.rkt
    - Here I've cleaned up and produced the file as it might appear after Step 6: Program Review.
  - 11-7-separate-files/
    - Here I've separated the system into several files, with one file per class.

Next Steps

- Study the relevant files in the examples folder:
  - Do some diffs so you see exactly what changes between one version and the next.
- If you have questions about this lesson, ask them on the Discussion Board
- Go on to the next lesson