# Using Inheritance to Share Implementations

CS 5010 Program Design Paradigms

"Bootcamp"

Lesson 12.1

# Key Points for Lesson 12.2

- By the end of this lesson you should be able to:
  - Identify common parts of class implementations
  - Generalize these common parts into a superclass
  - Recover the original classes using inheritance.
  - Use the template-and-hook pattern

# 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: **squares.rkt**

# Video Demo: squares.rkt

- http://youtu.be/Yi0cWg_XOnM (7:36)

# 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 **`DraggableObject%`**.
- Then we'll have **`Square%`** and **`Ball%`** both inherit from **`DraggableObject%`**.
- Let's see what happens:

# Video Demo: unify-try1

- http://youtu.be/dDuta8azUY4 (5:11)

# Well, that didn't work

- Well, that didn’t work.
- Let’s go back and turn some of those functions into methods

# Video Demo: turn-differences-into-methods.rkt

- http://youtu.be/dzpRU5gF6yU (4:50)

# What we have accomplished so far

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

**`add-to-scene`**
**`place-at-left-edge`**
**`place-at-right-edge`**
**`would-hit-left-edge?`**
**`would-hit-right-edge?`**
**`inside-this?`**

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.

# 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% ()
 (field x y)
 (define radius ...)

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

 (define/public
  (on-mouse mx my mev)
  ...(inside-this? mx my)))

 (define/public (on-tick)
  ...(would-hit-left-edge?)...)

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

 (define (would-hit-left-edge?) ...)  )
```

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

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

 (define/public
  (on-mouse mx my mev)
  ...(inside-this? mx my)))

 (define/public (on-tick)
  ...(would-hit-left-edge?)...)

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

 (define (would-hit-left-edge?) ...)  )
```

# 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.

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

 (define/public (add-to-scene s) ...)
  ...)
 (define/public
  (on-mouse-inside-this? mx my mev)
  ...(inside-this? mx my)...)
 (define/public (inside-this? mx my)
  ...(inside-this? mx my)...)
 (define/public (on-tick)
  ...(would-hit-left-edge?)...)
 (define/public
  (would-hit-left-edge?) ...)  )
```

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

 (define/public (add-to-scene s) ...)
  ...)
 (define/public
  (on-mouse-inside-this? mx my mev)
  ...(inside-this? mx my)...)
 (define/public (inside-this? mx my)
  ...(inside-this? mx my)...)
 (define/public (on-tick)
  ...(would-hit-left-edge?)...)
 (define/public
  (would-hit-left-edge?) ...)  )
```

# Turning differing functions into methods

# 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: Specialize by Creating Subclasses

- 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 **squares.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.


# Move common methods into superclass

# What this accomplishes

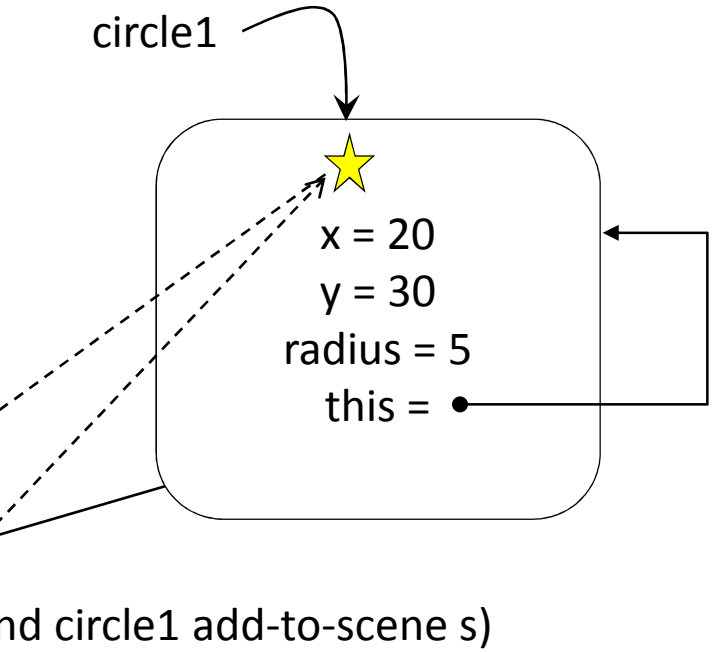
- We can think of the common method in the superclass as an abstraction or generalization of the methods in the classes.
- In the past, we generalized a set of functions by writing a single function with an extra argument. Depending on the value of the extra argument, we could get back one of our original 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. We call this *specialization by subclassing*.

# Subclassing in Action

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

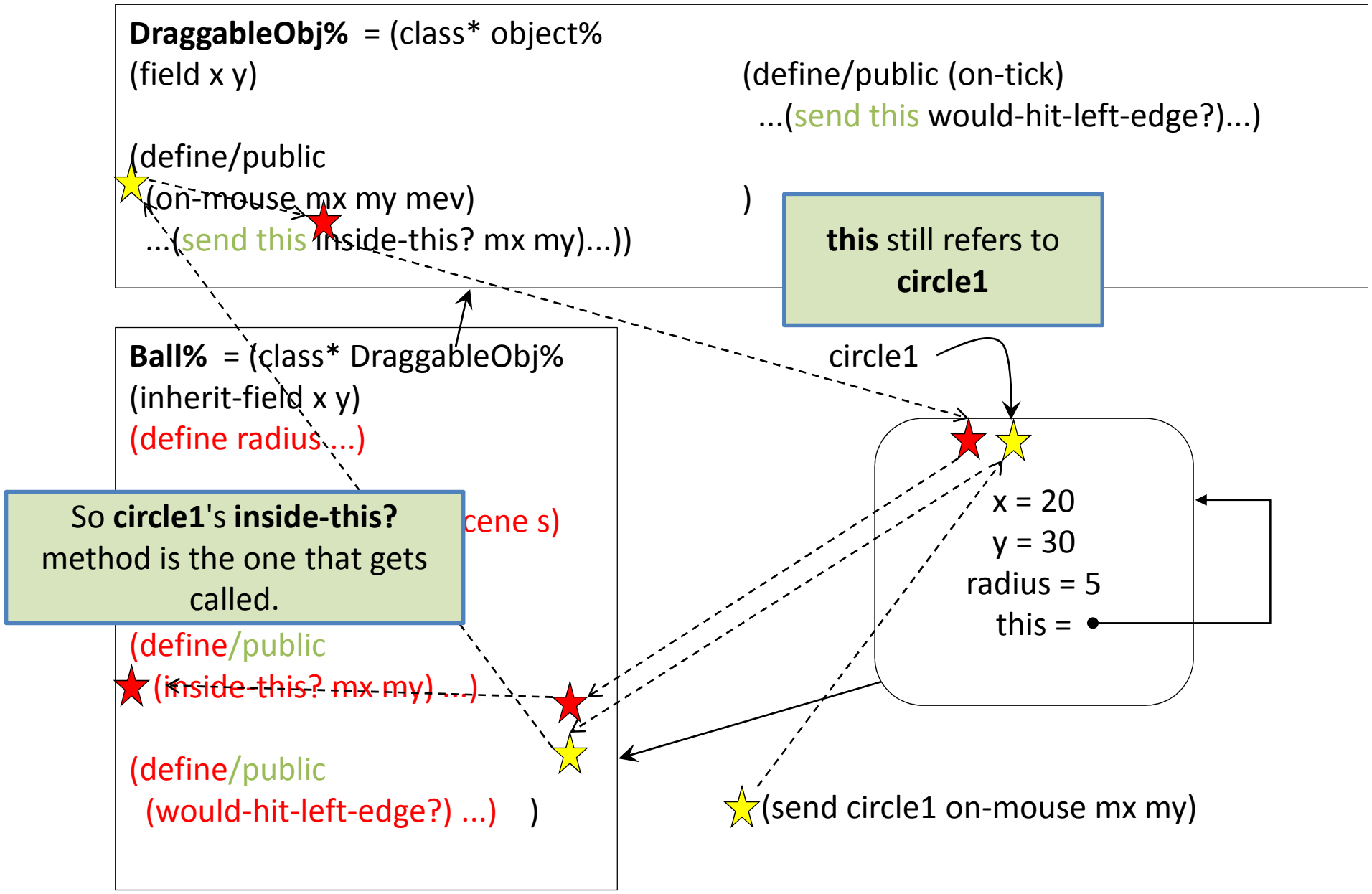
```
DraggableObj%  = (class* object%
(field x y)                                   (define/public (on-tick)
                                                ...(send this would-hit-left-edge?)...)
(define/public
  (on-mouse mx my mev)                        )
  ...(send this inside-this? mx my)...))
```

```
Ball%  = (class* DraggableObj%
(inherit-field x y)
(define radius ...)

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

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

(define/public
  (would-hit-left-edge?) ...)    )
```




Here's an example of method lookup where inheritance isn't involved

# Every object knows its own methods #1




# Every object knows its own methods #2

# 4. Use Templates and Hooks to Generalize Similar Methods

- 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 and a demo.

# 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)
```

**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**.

**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"))
```

# Video Demo: turn-differing fields into methods.rkt

- http://youtu.be/LjBSSlfsDNo (2:36)

# 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.

# Yet another way to share fields

- Create an init-field in the superclass
- Initialize it from the subclass, using super-new.
- This is useful for constants that are different in different subclasses.
- See 12-6-promote-similar-fields.rkt in the Examples folder (also pattern on next slide).

# Summary: 3 ways to share fields

**Initialize in superclass, inherit into subclass:**

```
(define super%
 (class …
   (init-field x)
   …x…))


(define sub1%
 (class super% …
   (super-new)
   (inherit-field x)
   …x…))
```

**Initialize in subclass, access via hook method:**

```
(define super%
 (class …
   (init-field x)
   (abstract get-x)
   …(send this get-x)…))


(define sub1%
 (class super% …
   (super-new)
   (init-field x)
   (define/override
     (get-x) x)
   …x…))
```

**Initialize in superclass:**

```
(define super%
 (class …
   (init-field x)
    …x…))


(define sub1%
 (class super% …
   (super-new
    [x “this is sub1”])
   …x…))
```

# Summary: Recipe for generalizing similar classes

1. Turn differing functions into methods
2. Move identical methods into a superclass
3. Specialize by subclassing
4. Create hooks to generalize similar methods