Basics of Inheritance

CS 5010 Program Design Paradigms
"Bootcamp"
Lesson 11.1
Key Points for this Module

• Inheritance is a technique for generalizing over common parts of class implementations.
• When we create such a generalization, we specialize by subclassing.
• Languages with inheritance have many new design choices.
Module 11

Data Representations
- Basics
- Mixed Data
- Recursive Data
- Functional Data
- Objects & Classes
- Stateful Objects

Design Strategies
- Combine simpler functions
- Use a template
- Divide into Cases
- Call a more general function
- Recur on subproblem
- Communicate via State

Generalization
- Over Constants
- Over Expressions
- Over Contexts
- Over Data Representations
- Over Method Implementations
Key Points for Lesson 11.1

• By the end of this lesson you should be able to explain how objects find methods by searching up the inheritance chain.
• Use the overriding-defaults pattern to introduce small variations of a class.
Example: 11-1-flashing-balls

- Sometimes we want to define a new class that is just a small variation of an old class.
- For example, we might want to make a ball that flashes different colors.
- To do this, create a subclass that inherits from the old class (the "superclass").
- We call this the "overriding defaults" pattern.
- Let's look at some code.
FlashingBall%

;; FlashingBall% is like a Ball%, but it displays
;; differently: it changes color on every fourth tick

(define FlashingBall%
  (class* Ball% ; inherits from Ball%
    (SBall<%>) ; implements same interface
    (define FlashingBall%
      ; number of ticks between color changes
      (field [color-change-interval 4])

      ; time left til next color change
      (field [time-left color-change-interval])

      ; the list of possible colors, first elt is
      ; current color
      (field [colors (list "red" "green")])

      ; here are fields of the superclass that we need.
      (inherit-field radius x y selected?)

      ; the init-field w isn't declared here,
      ; so it is sent to the superclass.
      (super-new)
    )
)

define/override is used to define methods that override methods in the superclass

;; Scene -> Scene
;; RETURNS: a scene like the given one, but with the
;; flashing ball painted on it.
;; EFFECT: decrements time-left and changes colors if
;; necessary
(define/override (add-to-scene s)
  (begin
    ;; is it time to change colors?
    (if (zero? time-left)
        (change-colors)
        (set! time-left (- time-left 1)))
    ;; now paint this ball on the scene
    (place-image
      (circle radius
        (if selected? "solid" "outline")
        (first colors))
      x y s)))

;; -> Void
;; EFFECT: rotate the list of colors,
;; and reset time-left
(define (change-colors)
  (set! colors
    (append (rest colors) (list (first colors))))
  (set! time-left color-change-interval))

Inherit-fields is used to declare fields of the superclass that we want to make visible in the subclass
Features for Inheritance in Racket

• The Racket object system uses two features to implement inheritance: define/override and inherit-fields.
  – define/override is used to define methods that override methods in the superclass.
  – inherit-fields is used to declare fields of the superclass that we want to make visible in the subclass.
    • eg: x, y, selected?, radius in FlashingBall.
    • values are automatically supplied to the superclass on initialization.

Other languages do this differently, so watch out!
What fields are in the subclass?

- The init-fields of a subclass are the init-fields of the superclass plus any additional init-fields declared in the subclass.
- FlashingBall% doesn't declare any new init-fields, so its init-fields are the same as those of Ball%.
- init-fields of the subclass are automatically sent to the superclass, so when we create a FlashingBall%, we write

  (new FlashingBall% [x ...][y ...][speed ...])

- Those values become the values for the fields in Ball%, so they can be used by the methods in Ball%.
- x and y are also inherited fields, so they are visible to the methods in FlashingBall% as well.
The flashing ball was an example of the *overriding-defaults* pattern. In the overriding-defaults pattern:

- The superclass has a complete set of behaviors
- The subclass makes an incremental change in these behaviors by overriding some of them.
How does inheritance work?

• An object searches its inheritance chain for a suitable method.
• For FlashingBall% we have
  – FlashingBall% inherits from
  – Ball%, which inherits from
  – object%
• but the chain could be as long as you want.
• Here’s an example (be sure to watch the animation):
An object searches its inheritance chain for a suitable method

```
(define b1 (new FlashingBall% ...))

(send b1 add-to-scene s)

(send b1 on-tick)

(send b1 launch-missiles)

b1

x = ...
y = ...
radius = ...
selected? = ...
time-left = ...
```

---

**Ball%** = (class* object% (...) (field x y radius selected?))

```
(define/public (on-tick) ...)
(define/public (on-mouse ...) ...)
(define/public (add-to-scene s) ...) ...
```

**FlashingBall%** = (class* Ball% (...) (inherit-field x y radius selected?) (field time-left ...))

```
(define/public (on-tick) ...)  
(define/public (on-mouse ...) ...)  
(define/override (add-to-scene s)  
  (if (zero? time-left) ...)  
  (place-image ... x y s))  ...
```
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 \texttt{this}

\begin{itemize}
  \item \texttt{b1} is a \texttt{FlashingBall\%} object.
  \item When we send \texttt{b1} an \texttt{m1} message, what happens?
  \begin{enumerate}
    \item It searches its own methods for an \texttt{m1} method, and finds none.
    \item It searches it superclass for an \texttt{m1} method. This time it finds one, which says to send \texttt{this} an \texttt{m2} message.
    \item \texttt{this} still refers to \texttt{b1}. So \texttt{b1} starts searching for an \texttt{m2} method.
    \item It finds the \texttt{m2} method in its local table, and returns the string “right”.
  \end{enumerate}
\end{itemize}
• Sometimes the subclass doesn’t need to change the behavior of the superclass’s method; instead it just needs to add behavior to the existing method.

• `(super method args ...)` calls the method named method in the superclass of the class in which the method is defined.
Use case for super

```
(define the-superclass%
  (class* object% ()
    (define/public (m1 x)
      (... big-hairy function of x ...))))

(define the-subclass%
  (class* the-superclass% ()
    (define/public (m1 x)
      (... Same big hairy function, but now of x+1 ...))))
```

We don’t want to have to write out the big hairy function again. Can we avoid this repeated code?
Use case for super

(define the-superclass%
  (class* object% ()
     (define/public (m1 x)
       (... big-hairy function of x ...))))

(define the-subclass%
  (class* the-superclass% ()
     (define/public (m1 x)
       (super m1 (+ x 1))))))

This calls m1 in the superclass.
You can call any method in the super

(define the-superclass%
  (class* object% (...)
    (define/public (m1 x)
      (... big-hairy function of x ...))))

(define the-subclass%
  (class* the-superclass% (...)
    (define/public (m2 x)
      (super m1 (+ x 1))
      (define/public (m1 x) "this is noise")))

In Racket, you can't call (super m1 ...) unless m1 is already defined in the current class. This is a wart in the Racket object system. If we were in a different system, this would not be necessary. Sorry about that.
**this and super, summarized**

- The rules for this and super can be summarized as:
  - **this** is dynamic, **super** is static
- This simple rule can lead to interesting behavior
  - Do Guided Practices 11.1 and 11.2 to learn more about this.
- We will take great advantage of the dynamic nature of **this** in the next lesson.
Summary of Lesson 11.1

• We’ve seen how to define superclasses and subclasses in Racket, including \texttt{inherit-field} and \texttt{define/override}.

• We’ve seen the overriding-defaults pattern, in which a subclass overrides some methods of a complete superclass.

• We learned how \texttt{this} works with inheritance, and what \texttt{super} does.
Next Steps

• Study 11-1-flashing-balls.rkt in the Examples folder.
• If you have questions about this lesson, ask them on the Discussion Board.
• Do the Guided Practices 11.1 and 11.2
• Go on to the next lesson