Encapsulation and Objects

Vectors and Identity
Encapsulation

Two lectures ago, we encapsulated a fish with a GUI:

; A live-fish is
; (num -> num)

; make-fish : num -> live-fish
(define (make-fish init-weight)
  (local [(define WEIGHT init-weight)
    (define (feed n) ...)
    ...
  ]
  (begin
    (create-window ...)
    feed)))

By returning feed, we enable programs that process groups of fish
Objects

Maybe we don't need the GUI, but we'd like to represent fish identities

```lisp
(define alice (new-fish 7))
(define bob (new-fish 6))
(define norman (new-fish 12))

(define my-favorite-fish (list alice norman))
(define all-my-fish (list alice bob norman))

(alice 4) "should be" 11

((first my-favorite-fish) 0) "should be" 11
((first all-my-fish) 0) "should be" 11
```
Objects

Maybe we don't need the GUI, but we'd like to represent fish identities

; A fish-object is
;   (num -> num)

; new-fish : num -> fish-object
(define (new-fish init-weight)
  (local [(define WEIGHT init-weight)
              (define (feed n)
                (begin
                  (set! WEIGHT (+ WEIGHT n))
                  WEIGHT))]
    feed))
How about armadillos?

; new-dillo : num bool -> (num -> num)
(define (new-dillo init-weight init-alive?)
  (local [(define WEIGHT init-weight)
           (define ALIVE? init-alive?)
           (define (feed n)
             (begin
               (set! WEIGHT (+ WEIGHT n))
               WEIGHT))])
  feed))

We can feed a dillo this way, but we can't check whether it's alive...
How can we return three functions?
Armadillo Objects

; A dillo-object is
;   (make-dillo (num -> num) (-> bool) (bool -> void))
(make-dillo (num -> num) (-> bool) (bool -> void))
(define-struct dillo (feed is-alive? set-alive))

; new-dillo : num bool -> dillo-object
(define (new-dillo init-weight init-alive?)
  (local [(define WEIGHT init-weight)
           (define ALIVE? init-alive?)
           (define (feed n)
             (begin
               (set! WEIGHT (+ WEIGHT n))
               WEIGHT))
           (define (is-alive?)
             ALIVE?)
           (define (set-alive a?)
             (set! ALIVE? a?)
             )]
  (make-dillo feed is-alive? set-alive)))
Armadillo Object Examples

(define cindy (new-dillo 5 true))
(define dan (new-dillo 8 true))

((dillo-feed cindy) 2) "should be" 7
((dillo-feed dan) 1) "should be" 9
((dillo-feed cindy) 0) "should be" 7

; run-over! : dillo -> void
(define (run-over! d)
  ((dillo-set-alive d) false))

((dillo-alive? dan)) "should be" true
(run-over! dan) "should be" (void)
((dillo-alive? dan)) "should be" false
((dillo-alive? cindy)) "should be" true
Disallowing Armadillo Resurrection

; A dillo-object is
;  (make-dillo (num -> num) (-> bool) (-> void))
(make-struct dillo (feed is-alive? run-over!))

; new-dillo : num bool -> dillo-object
(make-dillo (define (new-dillo init-weight init-alive?)
  (local [(define WEIGHT init-weight)
    (define ALIVE? init-alive?)
    (define (feed n)
      (begin
        (set! WEIGHT (+ WEIGHT n))
        WEIGHT))
    (define (is-alive?)
      ALIVE?)
    (define (run-over!)
      (set! ALIVE? false))]
  (make-dillo feed is-alive? run-over!)))
General Pattern for Encapsulating Objects

; A THING-object is
; (make-THING method-type ...)
(define-struct THING (METHOD ...))

; new-THING : init-type ... -> THING-object
(define (new-THING init-val ...)
  (local [(define STATE init-val)
    ...
    ; METHOD : method-type
    (define (METHOD arg ...)
      ...)]
    (make-THING METHOD ...)))

Note: implementation depends on the operations (a.k.a. methods) that you want
Encapsulation

- Groups related functions with data
- Controls access/modification of state

Encapsulation is a key idea behind languages like Java

Still, one other idea is more important:
- Data-driven design

One more idea is equally important:
- Extensible data definitions
Encapsulation and Objects

Vectors and Identity
A vector is an object with state

\[
\text{(define } v \text{ (vector 'a 'b 'c))}
\]

\[
(\text{vector-ref v 0) "should be" 'a}
\]

\[
(\text{vector-set! v 0 'd})
\]

\[
(\text{vector-ref v 0) "should be" 'd}
\]
Let's keep our armadillos in cages

- Each cage holds one armadillo
- If we have 5 cages, we can represent the set of cages with a vector of size 5

```scheme
(define cages
  (vector false false false false false false false false false false))
(define cindy (new-dillo 5 true))
(define dan (new-dillo 8 true))
(vector-set! cages 0 cindy)
(vector-set! cages 1 dan)
```
Moving Armadillos

• Implement `move-dillo` which takes a `dillo-object` and a cage number, and move the dillo to the cage number

```scheme
; move-dillo : dillo-object n -> void

; continuing from the previous example
(move-dillo cindy 3) "should be" (void)
(vector-ref cages 3) "should be" cindy
```
Moving Armadillos

First attempt:

```scheme
(define (move-dillo d n)
  (vector-set! cages n d))
```

Problem: the dillo is still in its old cage

```scheme
(move-dillo cindy 3) "should be" (void)
(vector-ref cages 3) "should be" cindy
(vector-ref cages 0) "should be" false
; but currently we get cindy
```
Finding and Moving Armadillos

(define (move-dillo d n)
  (begin
    (vector-set! cages
      ... false)
    (vector-set! cages n d)))
Finding and Moving Armadillos

(define (move-dillo d n)
  (begin
    (vector-set! cages
      (find-dillo d)
      false)
    (vector-set! cages n d)))

; find-dillo : dillo -> num
(define (find-dillo d)
  ...)
Finding and Moving Armadillos

\[
\text{(define (move-dillo d n)}
\begin{align*}
\text{(begin }& \text{(vector-set! cages } \\
& \text{(find-dillo d) } \\
& \text{false) } \\
& \text{(vector-set! cages n d))}) \end{align*}
\]

; find-dillo : dillo \rightarrow \text{num}
\text{(define (find-dillo d)}
\text{(find-dillo-at d 0))}

; find-dillo-at : dillo \text{num} \rightarrow \text{num}
\text{(define (find-dillo-at d n)}
\text{(cond } \\
[\text{(same? d (vector-ref cages n)) n}] \\
[\text{else (find-dillo-at d (add1 n))}])))
Comparing Armadillos

; same? : dillo-object dillo-object -> bool
(define (same? d1 d2)
  (and (= ((dillo-feed d1) 0)
          ((dillo-feed d2) 0))
       (boolean=? ((dillo-alive? d1))
                  ((dillo-alive? d2)))))

(define eddie (new-dillo 7 true))
(define fran (new-dillo 7 true))

(same? eddie fran) "should be" true

But that's not right – eddie and fran are different armadillos
Detecting an Armadillo

If \( d_1 \) and \( d_2 \) are the same, then feeding \( d_1 \) should grow \( d_2 \):

\[
\text{(define (same? } d_1 d_2) \\
\text{(local } [(\text{define orig-d2-size } ((\text{dillo-feed } d_2) 0))] \\
\text{(begin} \\
\text{((dillo-feed } d_1) 1) \\
\text{(local } [(\text{define later-d2-size } ((\text{dillo-feed } d_2) 0))] \\
\text{(begin} \\
\text{((dillo-feed } d_1) -1) \\
\text{(eq? later-d2-size orig-d2-size)))))])
\]

Granted, this is a harsh way to compare armadillos, so Advanced provides \textbf{eq}?

\[
\text{(define (same? } d_1 d_2) \\
\text{(eq? } d_1 d_2))
\]