Last Time

The `add-digit` and `change-total` functions "remember" using `TOTAL`, `WORKING`, and `PREV-OP`
Designing Functions with State

- New design tool: organizational charts
- **Contract, Purpose, and Header** becomes **Contract, Purpose, Effect, and Header**
- Examples include starting state and effect
- Template includes potential assignments
; add-digit : num -> true
; Adds a digit to the number being entered
; Effect: extends number, updates GUI

(define (add-digit n)
  ... n
  ... WORKING ... (set! WORKING ... ...) ...)
; change-total : num (num num --> num) --> true
; Combines number and total
; Effect: sets total, resets number, sets op, updates GUI
(define (change-total n OP)
  ... n ... OP
  ... WORKING ... (set! WORKING ...)
  ... PREV-OP ... (set! PREV-OP ...)
  ... TOTAL ... (set! TOTAL ...) ...)

5
Examples

(begin
  (set! WORKING 0)
  (add-digit 5) "should be" true
  WORKING "should be" 5)

(begin
  (set! WORKING 10)
  (add-digit 5) "should be" true
  WORKING "should be" 105)
Examples

(begin
  (set! TOTAL 3)
  (set! WORKING 5)
  (set! PREV-OP *)
  (change-total 5 +) "should be" true
  TOTAL "should be" 15
  WORKING "should be" 0
  PREV-OP "should be" +)
Simpler Example

Suppose we want a GUI to manage a fish

New rule: keep the *view* and *control* separate from the *model*

- The view and control are in the GUI
- The model is a fish with a weight

Design the model first
Fish Model

• The only operation in the model is **feed**

; feed : num → num
; Grows the fish by n, returns new size
; Effect: adjusts the fish's weight

![Diagram showing the operation of the fish model with parameters and effect on weight.](image-url)
Fish Model

- The only operation in the model is `feed`

```scheme
(define (feed n)
  ... n ... WEIGHT
  ... (set! WEIGHT ...) ...)
```

```
(begin
  (set! WEIGHT 1)
  (feed 10) "should be" 11
  WEIGHT "should be" 11)
```
Fish Model Implementation

; feed : num -> num
; Grows the fish by n, returns new size
; Effect: adjusts the fish's weight
(define (feed n)
  (begin
    (set! WEIGHT (+ WEIGHT n))
    WEIGHT))

(begin
  (set! WEIGHT 1)
  (feed 10) "should be" 11
  WEIGHT "should be" 11)
Implementing the View and Controller

Use the GUI teachpack to construct view and control

- Message objects implement the view
- Button callbacks implement the control

Often, the model never calls the control
Complete Fish Program

; The model:
(define WEIGHT 3)
; feed : num -> num
; ...
(define (feed n)
  (begin
   (set! WEIGHT (+ n WEIGHT))
   WEIGHT))
... tests here ...

; The view:
(define msg (make-message (number->string WEIGHT)))
; The control:
(define (feed-button n)
  (make-button (string-append "Feed " (number->string n))
  (lambda (evt)
   (draw-message
    msg
    (number->string (feed n))))))
(create-window
  (list (list msg) (list (feed-button 1) (feed-button 3))))
Multiple Fish

As we saw last time, if we want multiple fish, we can use `local`

```scheme
(define (make-fish init-weight)
  (local [(define WEIGHT init-weight)
            (define (feed n)
              (begin
                (set! WEIGHT (+ WEIGHT n))
                WEIGHT))
          ...]
    (create-window ...)))
```
Evaluating make-fish

(define (make-fish init-weight)
  (local [(define WEIGHT init-weight)]
    (define (feed n)
      (begin
        (set! WEIGHT (+ WEIGHT n))
        WEIGHT))
    ...
    (create-window ...)))

(make-fish 5)

→

...
Evaluating make-fish

... 
(local [(define WEIGHT 5]
  (define (feed n)
    (begin
      (set! WEIGHT (+ WEIGHT n))
      WEIGHT))
...]
(create-window ...))

→

...
(define WEIGHT\textsubscript{65} 5)
(define (feed\textsubscript{67} n)
  (begin
    (set! WEIGHT\textsubscript{65} (+ WEIGHT\textsubscript{65} n))
    WEIGHT\textsubscript{65}))
...
(create-window ...)
Multiple Fish

Every time we call `make-fish` a new `WEIGHT` is created for the new fish.

We can make a whole aquarium....

- How can we get the current total weight of all fish?
- How can we auto-feed all fish?

Problem: `make-fish` returns only a window

The renamed `WEIGHT` is completely hidden.
Returning the Weight

Does this help?

; make-fish : num -> num
(define (make-fish init-weight)
    (local [(define WEIGHT init-weight)
            ...
            ]

    (begin
        (create-window ...)
        WEIGHT)))

No:

(make-fish 5)
→ (local [(define WEIGHT 5) ...] ... WEIGHT)
→ (define WEIGHT_{73} 5) ... WEIGHT_{73}
→ → (define WEIGHT_{73} 5) ... 5
Returning the Feeder

Only functions inside \texttt{make-fish} can see \texttt{WEIGHT}

So maybe \texttt{make-fish} should return a function:

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

(make-fish 5)
\end{verbatim}

\begin{verbatim}
→ (local [(define WEIGHT 5) (define (feed n) ... WEIGHT ...) ...]
    ... feed)
→ (define WEIGHT\_77 5) (define (feed\_81 n) ... WEIGHT\_77 ...) ... feed\_81
\end{verbatim}
Feeding an Aquarium

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

; make-fish : num -> live-fish
...
(define aquarium (list (make-fish 5)
                      (make-fish 3)
                      (make-fish 12)))

; aq-weight : list-of-live-fish -> num
(define (aq-weight l)
  (foldr (lambda (f r) (+ (f 0) r)) 0 l))

; feed-all : n list-of-live-fish -> ...
(define (feed-all n l)
  (map (lambda (f) (f n)) l))
The built-in function **for-each** is like **map**, but it returns **(void)**

; feed-all! : n list-of-live-fish -> (void)
; Feeds n to each live-fish in l
; Effect: each live-fish becomes heavier
(define (feed-all! n l)
  (for-each (lambda (f) (f n)) l))
The built-in function **for-each** is like **map**, but it returns (**void**) 

; feed-all! : n list-of-live-fish → (**void**) 
; Feeds n to each live-fish in l 
; Effect: each live-fish becomes heavier

(define (feed-all! n l)
  (for-each (lambda (f) (f n)) l))

(begin
  (define l (list (make-fish 1) (make-fish 2)))
  (feed-all! 3 l) "should be" (**void**) 
  l "should be" (list (make-fish 4) (make-fish 5)))

?)
The built-in function `for-each` is like `map`, but it returns `(void)`

```scheme
; feed-all! : n list-of-live-fish -> (void)
; Feeds n to each live-fish in l
; Effect: each live-fish becomes heavier
(define (feed-all! n l)
  (for-each (lambda (f) (f n)) l))

(begin
  (define l (list (make-fish 1) (make-fish 2)))
  (feed-all! 3 l) "should be" (void)
  l "should be" (list (make-fish 4) (make-fish 5)))
```

- This test doesn't completely capture the effect
for-each

The built-in function **for-each** is like **map**, but it returns *(void)*

; feed-all! : n list-of-live-fish -> (void)
; Feeds n to each live-fish in l
; Effect: each live-fish becomes heavier
(define (feed-all! n l)
  (for-each (lambda (f) (f n)) l))

(begin
  (define l (list (make-fish 1) (make-fish 2)))
  (feed-all! 3 l) "should be" (void)
  ((first l) 0) "should be" 4
  ((first (rest l)) 0) "should be" 5)
The built-in function `for-each` is like `map`, but it returns `(void)`:

```
; feed-all! : n list-of-live-fish -> (void)
; Feeds n to each live-fish in l
; Effect: each live-fish becomes heavier
(define (feed-all! n l)
  (for-each (lambda (f) (f n)) l))
```

```
(begin
  (define l (list (make-fish 1) (make-fish 2)))
  (feed-all! 3 l) "should be" (void)
  ((first l) 0) "should be" 4
  ((first (rest l)) 0) "should be" 5)
```

- Testing with state is often difficult
- Avoid this difficulty by avoiding state
A Tale of Two Fish Representations

; A fish is
;   num

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

• A **fish** represents a fish of a particular weight
  o Feed the fish ⇒ new value

• A **live-fish** represents a fish with a particular identity
  o Feed the fish ⇒ same value, new state
A Tale of Two Fish Representations

; A fish is
; num

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

live-fish is more closely reflects reality

• On the one hand, reflecting reality makes things more intuitive
• On the other hand, reality can be messy

Key question when designing a program: what to represent
Encapsulation

Packaging fish state with its operations is called *encapsulation*

More on encapsulation soon...
Design with State Summary

• Deciding to use state: often motivated by GUIs
  ○ Split into model and view/controller

• The design recipe for state
  ○ Charts (no handin artifact)
  ○ Effects (handin with purpose)
  ○ Template with assignments (handin optional)
  ○ Multi-step tests (handin as usual)

• Design for the single-instance case, then encapsulate if necessary