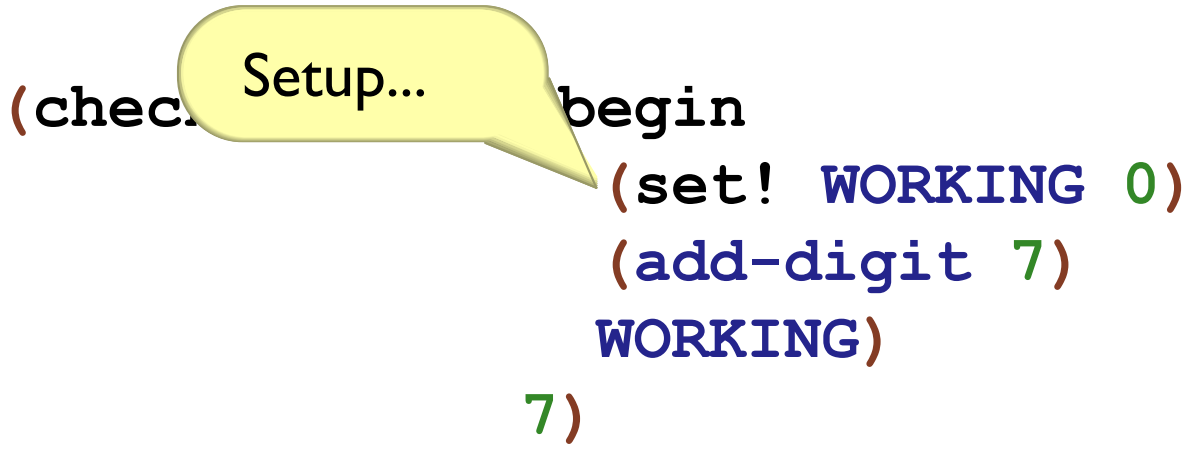
Testing Functions with State

```
(check-expect (begin
                (set! WORKING 0)
                (add-digit 7)
                WORKING)
              7)
```

Testing Functions with State



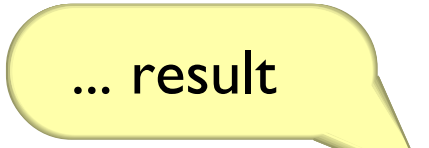
Testing Functions with State

```
(check-expect (begin
                (set! WORKING 0)
                (add-digit 7)
                WORKING)
              7)
```

... call ...

Testing Functions with State

```
(check-expect (begin
                (set! WORKING 0)
                (add-digit 7)
                WORKING)
              7)
```



Testing Functions with State

```
(check-expect (begin
                (set! WORKING 0)
                (add-digit 7)
                WORKING)
              7)
```

Problem: **WORKING** is left in a strange state

Testing Functions with State

```
(check-expect (begin
                (set! WORKING 0)
                (add-digit 7)
                (local [(define r WORKING)]
                  (begin
                    (set! WORKING 0)
                    r)))
              7)
```

Testing Functions with State

Setup ...

```
(check-equal? (begin
                (set! WORKING 0)
                (add-digit 7)
                (local [(define r WORKING)]
                  (begin
                    (set! WORKING 0)
                    r)))
              7)
```

Testing Functions with State

```
(check
                (begin
                  (set! WORKING 0)
                  (add-digit 7)
                  (local [(define r WORKING)]
                    (begin
                      (set! WORKING 0)
                      r)))
                7)
```

... call ...

Testing Functions with State

```
(check-expect (begin
                (define WORKING 0)
                (define (git 7)
                  (local [(define r WORKING)]
                    (begin
                      (set! WORKING 0)
                      r)))
                7)
```

... result ...

Testing Functions with State

```
(check-expect (begin
                (set! WORKING 0)
                (add-digit 7)
                (local [(define r WORKING)]
                  (begin
                    (set! WORKING 0)
                    r)))
              7)
```

... teardown

Testing Functions with State

```
(check-expect (begin
                (set! WORKING 53)
                (add-digit 1)
                (local [(define r WORKING)]
                  (begin
                    (set! WORKING 0)
                    r)))
              531)
```

Testing Functions with State

```
(check-expect (begin
                (set! TOTAL 3)
                (set! WORKING 5)
                (change-total * 5)
                (local [(define r (list TOTAL
                                        WORKING))]
                  (begin
                    (set! TOTAL 0)
                    (set! WORKING 0)
                    r)))
              (list 15 0))
```

Model–View–Controller

Suppose we want a GUI to manage a fish

Run

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

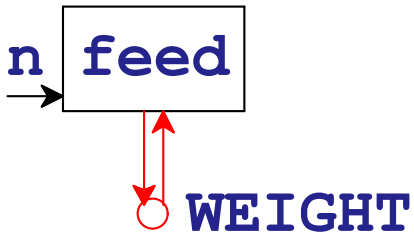
- The view and control are in the GUI
- The model is a fish with a weight

Design the model first

Fish Model

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

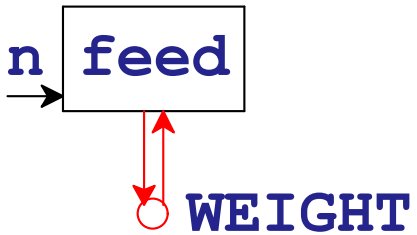
```
; feed : num -> num
;  Grows the fish by n, returns new size
;  Effect: adjusts the fish's weight
```



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
```




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

```
(check-expect (begin
                (set! WEIGHT 1)
                (local [(define r1 (feed 10))
                        (define r2 WEIGHT)]
                  (set! WEIGHT 0)
                  (list r1 r2)))
              (list 11 11))
```

Fish Model Implementation

```
(define WEIGHT 0)

; 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))

(check-expect (begin
                (set! WEIGHT 1)
                (local [(define r1 (feed 10))
                        (define r2 WEIGHT)]
                  (set! WEIGHT 0)
                  (list r1 r2)))
              (list 11 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

Copy

```
; 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`**

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

Evaluating create-fish

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

→

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

Evaluating create-fish

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

→

```
...
(define WEIGHT65 5)
(define (feed67 n)
  (begin
    (set! WEIGHT65 (+ WEIGHT65 n))
    WEIGHT65))
...
(create-window ...)
```

Multiple Fish

Every time we call **`create-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?

Problem: **`create-fish`** returns only a window

The renamed **`WEIGHT`** is completely hidden

Returning the Weight

Does this help?

```
; create-fish : num -> num
(define (create-fish init-weight)
  (local [(define WEIGHT init-weight)
          ...]
    (begin
      (create-window ...)
      WEIGHT)))
```

No:

```
(create-fish 5)
→ (local [(define WEIGHT 5) ...] ... WEIGHT)
→ (define WEIGHT73 5) ... WEIGHT73
→ (define WEIGHT73 5) ... 5
```

A variable is not a value

Variable Structs

A struct is a value:

```
(define-struct fish (weight))
(define sam (make-fish 3))
sam → (make-fish 3)
```

A struct is variable:

```
(fish-weight sam) → 3
(set-fish-weight! sam 4)
(fish-weight sam) → 4
```

Returning a Fish

```
(define-struct fish (weight))

; create-fish : num -> fish
(define (create-fish init-weight)
  (local [(define FISH (make-fish init-weight))
          ...]
    (begin
      (create-window ...)
      FISH)))
```

Variable Structs

Evaluating `make-fish` establishes a fish's identity:

```
(define samuel (make-fish 3))
(define sam samuel)

(fish-weight sam) → 3
(set-fish-weight! samuel 4)
(fish-weight sam) → 4
```

Evaluation with Variable Structs

```
(define samuel (make-fish 3))
(define sam samuel)
(fish-weight sam)
(set-fish-weight! samuel 4)
(fish-weight sam)
```

→

```
(define* FISH17 (make-fish 3))
(define samuel FISH17)
(define sam samuel)
(fish-weight sam)
(set-fish-weight! samuel 4)
(fish-weight sam)
```

define* binds an identifier as a value

Evaluation with Variable Structs

```
(define* FISH17 (make-fish 3))
(define samuel FISH17)
(define sam samuel)
(fish-weight sam)
(set-fish-weight! samuel 4)
(fish-weight sam)
```

→

```
(define* FISH17 (make-fish 3))
(define samuel FISH17)
(define sam FISH17)
(fish-weight sam)
(set-fish-weight! samuel 4)
(fish-weight sam)
```

Evaluation with Variable Structs

```
(define* FISH17 (make-fish 3))
(define samuel FISH17)
(define sam FISH17)
(fish-weight sam)
(set-fish-weight! samuel 4)
(fish-weight sam)
```

→

```
(define* FISH17 (make-fish 3))
(define samuel FISH17)
(define sam FISH17)
(fish-weight FISH17)
(set-fish-weight! samuel 4)
(fish-weight sam)
```

Evaluation with Variable Structs

```
(define* FISH17 (make-fish 3))
(define samuel FISH17)
(define sam FISH17)
(fish-weight FISH17)
(set-fish-weight! samuel 4)
(fish-weight sam)
```

→

```
(define* FISH17 (make-fish 3))
(define samuel FISH17)
(define sam FISH17)
3
(set-fish-weight! samuel 4)
(fish-weight sam)
```

Evaluation with Variable Structs

```
(define* FISH17 (make-fish 3))
(define samuel FISH17)
(define sam FISH17)
3
(set-fish-weight! samuel 4)
(fish-weight sam)
```

→

```
(define* FISH17 (make-fish 3))
(define samuel FISH17)
(define sam FISH17)
3
(set-fish-weight! FISH17 4)
(fish-weight sam)
```

Evaluation with Variable Structs

```
(define* FISH17 (make-fish 3))
(define samuel FISH17)
(define sam FISH17)
3
(set-fish-weight! FISH17 4)
(fish-weight sam)

→

(define* FISH17 (make-fish 4))
(define samuel FISH17)
(define sam FISH17)
3
(void)
(fish-weight sam)
```

Evaluation with Variable Structs

```
(define* FISH17 (make-fish 4))
(define samuel FISH17)
(define sam FISH17)
3
(void)
(fish-weight sam)
```

→

```
(define* FISH17 (make-fish 4))
(define samuel FISH17)
(define sam FISH17)
3
(void)
(fish-weight FISH17)
```

Evaluation with Variable Structs

```
(define* FISH17 (make-fish 4))
(define samuel FISH17)
(define sam FISH17)
3
(void)
(fish-weight FISH17)
```

→

```
(define* FISH17 (make-fish 4))
(define samuel FISH17)
(define sam FISH17)
3
(void)
4
```

Allocation

The step from

```
(make-fish 3)
```

to

```
(define* FISH89 (make-fish 3))
FISH89
```

is called ***allocation***

eq?

The `eq?` operator compares identity:

```
(define samuel (make-fish 3))
(define sam samuel)
(define gil (make-fish 3))

(equal? sam gil) → true
(eq? sam gil) → false
(eq? sam samuel) → true
```

Object Allocation

Java is the same:

- `new` allocates an object
- = changes a field's value
- == compares identity

Varying Fields

```
class Fish {
    int weight;
    Fish(int weight) { this.weight = weight; }
    void feed(int amt) {
        this.weight = this.weight + amt;
    }
    int getWeight() {
        return this.weight;
    }
}
```

Object Allocation and Identity

```
Fish samuel = new Fish(3);
Fish sam = samuel;
Fish gil = new Fish(3);

t.checkExpect(sam.getWeight(), 3);
sam.feed(1);
t.checkExpect(sam.getWeight(), 4);
t.checkExpect(gil.getWeight(), 3);
t.checkExpect(sam == samuel, true);
t.checkExpect(sam == gil, false);
```

Identities for non-Structs and non-Objects

Identity is sometimes underspecified:

- strings in Java
- numbers in Racket

Beware!