

Objects, Modularity, and State

- Assignment and Local State
- The Environment Model of Evaluation
- Modeling with Mutable Data
- Object Oriented Programming
- Streams

Assignment and Local State

- One powerful design strategy (especially when modeling physical systems) is to *base the structure of the program on the structure of the thing being modeled.*
- For each object in the system, we define a corresponding computational object
- For each action in the system, we design a corresponding operation

Assignment and Local State

- What do we mean by “state?”
- What are some examples of objects with state (from real life, say)?

Assignment and Local State

- What do we mean by “state?”

An object is said to have state if its behavior is influenced by its history

We can characterize an object’s state by one or more *state variables*

Assignment and Local State

- Object-oriented view of a system decomposes system into computational objects
- Each computational object must have its own *local state variables* describing the object's actual state
- Since the states of the objects change, the variables must be able to change
- We must introduce *assignment*

Example: Bank Account

```
1 ]=> (withdraw 25)
```

```
;Value: 75
```

```
1 ]=> (withdraw 25)
```

```
;Value: 50
```

```
1 ]=> (withdraw 60)
```

```
;Value: Insufficient Funds
```

```
1 ]=> (withdraw 15)
```

```
;Value: 35
```

Example: Bank Account

- Something completely new has just happened
- We evaluated (`withdraw 25`) two different times
- And we got two different results!

Example: Bank Account

- To implement `withdraw` we need a state variable to keep track of the balance between calls to `withdraw`
- Each time `withdraw` is called, the value of the balance should be changed

Example: Bank Account

```
(define balance 100)
```

```
(define (withdraw amount)
```

```
  (if (>= balance amount)
```

```
      (sequence (set! balance
```

```
                  (- balance amount))
```

```
              balance)
```

```
      "Insufficient funds"))
```

The Special Form `set!` !

- Decrementing the balance is accomplished with the statement

```
(set! balance (- balance amount))
```

The Special Form `set!`!

- Do we really need `set!`?
- What's wrong with

```
(define balance (- balance amount))
```

The Special Form `set!` !

- The special form `set!` !

`(set! < name > < new – value >)`

- The value of the variable `<name>` is changed to be the value of `<new – value>`
- **Important:** The value of a `set!` expression is undefined.

Example: Bank Account

- Problems with `withdraw`?

Example: Bank Account

- Problems with `withdraw`?

`balance` is not a local state variable

Example: Bank Account

```
(define new-withdraw
  (let ((balance 100))
    (lambda (amount)
      (if (>= balance amount)
          (sequence
            (set! balance
                  (- balance amount))
            balance)
          "Insufficient funds"))))
```

Example: Bank Account

- Explain

Example: Bank Account

- Combining `set!` with local variables (e.g., created by `let`) is a general programming technique we will use for constructing computational objects with state
- We will have to introduce a new model for evaluation to handle this, however (the environment model)

Example: Bank Account

```
(define (make-withdraw balance)
  (lambda (amount)
    (if (>= balance amount)
        (sequence
         (set! balance (- balance amount))
         balance)
        "Insufficient funds")))
```

Example: Bank Account

- What will happen and why?

```
(define W1 (make-withdraw 100))
```

```
(define W2 (make-withdraw 100))
```

```
1 ]=> (W1 50)
```

```
;Value:
```

```
1 ]=> (W2 70)
```

```
;Value:
```

Example: Bank Account

```
(define W1 (make-withdraw 100))
```

```
(define W2 (make-withdraw 100))
```

```
1 ]=> (W1 50)
```

```
;Value: 50
```

```
1 ]=> (W2 70)
```

```
;Value: 30
```

Example: Bank Account

- A real bank account handles deposits and withdrawals
- How can we augment our objects to do both things?

Example: Bank Account

- A real bank account handles deposits and withdrawals
- How can we augment our objects to do both things?
 - Create an account object and pass messages to it!

Example: Bank Account

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (sequence (set! balance (- balance amount))
                  balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Unknown request -- MAKE-ACCOUNT"
                       m))))
  dispatch)
```

Example: Bank Account

- What happens and why?

```
(define acc (make-account 100))
```

```
1 ]=> ((acc 'withdraw) 50)
```

```
;Value:
```

```
1 ]=> ((acc 'withdraw) 60)
```

```
;Value:
```

```
1 ]=> ((acc 'deposit) 40)
```

```
;Value:
```

Example: Bank Account

- What happens and why?

```
(define acc (make-account 100))
```

```
1 ]=> ((acc 'withdraw) 50)
```

```
;Value: 50
```

```
1 ]=> ((acc 'withdraw) 60)
```

```
;Value: Insufficient Funds
```

```
1 ]=> ((acc 'deposit) 40)
```

```
;Value: 90
```

Example: Bank Account

- This is a little ugly

```
1 ]=> ((acc 'withdraw) 50)
```

- How could we fix it up with “syntactic sugar?”

Example: Bank Account

- This is a little ugly

```
1 ]=> ((acc 'withdraw) 50)
```

- How could we fix it up with “syntactic sugar?”

```
(define (withdraw account amount)  
  ((account 'withdraw) amount))
```

```
(define (deposit account amount)  
  ((account 'deposit) amount))
```

Exercise

- Define a procedure `flip` (with no parameters) that returns 1 the first time it is called, 0 the second time it is called, 1 the third time, etc.

```
1 ]=> (flip)
;Value: 1
```

```
1 ]=> (flip)
;Value: 0
```

```
1 ]=> (flip)
;Value: 1
```

Exercise

- The value of flip is the lambda expression *inside* the let expression

```
(define flip
  (let ((state 0))
    (lambda ()
      (set! state (if (= state 0) 1 0))
      state))))
```

Exercise

- Define a procedure `make-flip` that can be used to create the procedure `flip`

Exercise

```
(define (make-flip)
  (define flip
    (let ((state 0))
      (lambda ()
        (set! state (if (= state 0) 1 0))
        state)))
  flip)
```

Exercise

- Define a flip object that can flip itself and which can also be reset
- Hint: Use message passing
- Define a syntactically pleasing interface to the flip object

Exercise

- Example of behavior

```
(define a (make-flipper))
```

```
(flip a)  
;Value: 1
```

```
(flip a)  
;Value: 0
```

```
(reset a)  
;Value: 0
```

Exercise

```
(define (make-flipper)
  (let ((state 0))
    (define flip
      (lambda ()
        (set! state (if (= state 0) 1 0))
        state))
    (define reset
      (lambda ()
        (set! state 0)
        state))
    (define (dispatch m)
      (cond ((eq? m 'flip) flip)
            ((eq? m 'reset) reset)
            (else (error "Yow!" m))))
    dispatch))
```

Exercise

```
(define (flip a)
  ((a 'flip)))
```

```
(define (reset a)
  ((a 'reset)))
```

The Substitution Model Revisited

- Use the substitution model to evaluate

```
( (make-flip) )
```

The Substitution Model Revisited

- Use the substitution model to evaluate

```
((make-flip))
```

```
((lambda ()
```

```
  (set! state (if (= state 0) 1 0))
```

```
  state))
```

The Substitution Model Revisited

- Use the substitution model to evaluate

```
((make-flip))
```

```
((lambda ()
```

```
  (set! state (if (= state 0) 1 0))
```

```
  state))
```

```
(set! state (if (= state 0) 1 0)) state
```

The Substitution Model Revisited

- Use the substitution model to evaluate

```
((make-flip))
```

```
((lambda ()
```

```
  (set! state (if (= state 0) 1 0))
```

```
  state))
```

```
(set! state (if (= state 0) 1 0)) state
```

```
(set! state (if (= 0 0) 1 0)) 0
```

The Substitution Model Revisited

- Use the substitution model to evaluate

```
((make-flip))  
(lambda ()  
  (set! state (if (= state 0) 1 0))  
  state))  
(set! state (if (= state 0) 1 0)) state  
(set! state (if (= 0 0) 1 0)) 0  
(set! state 1) 0  
0
```

The Substitution Model Revisited

- What went wrong?

The Substitution Model Revisited

- What went wrong?
- The substitution model is insufficient to handle assignment