CSE 1729:Principles of Programming

Lecture 20 : Object Oriented Design in Scheme

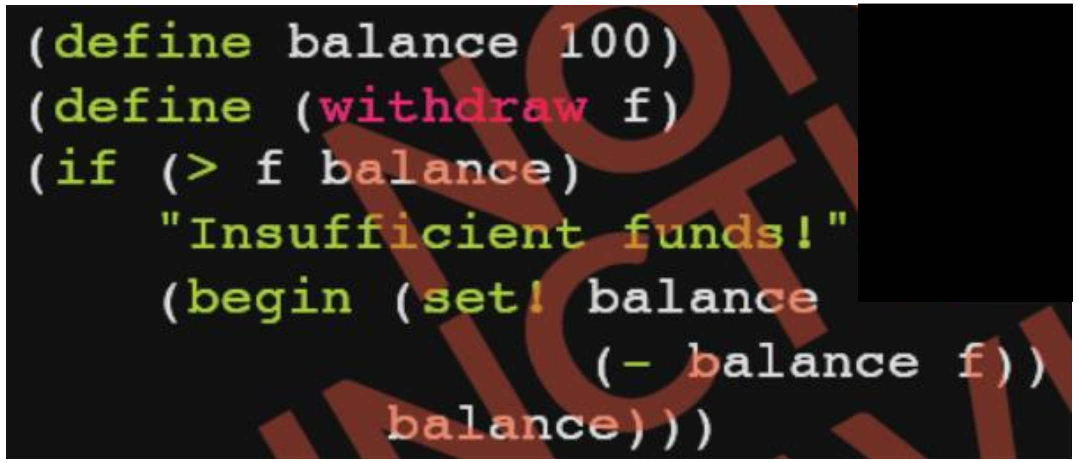
Kaleel Mahmood

Department of Computer Science and Engineering

University of Connecticut

Recall The Bank Account Balance Example

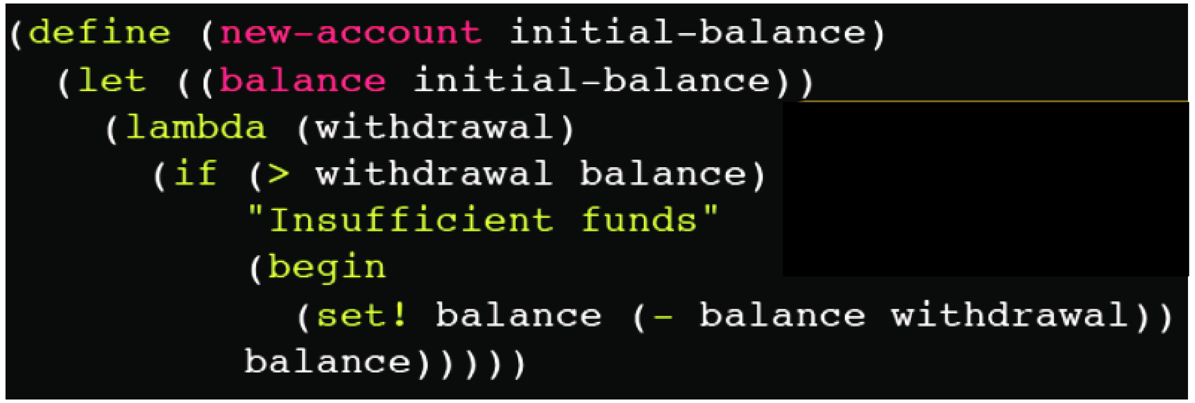
Bad Code: Anyone can change the balance, no private variables.



```
(define balance 100)
(define (withdraw f)
(if (> f balance)
    "Insufficient funds!"
    (begin (set! balance
                 (- balance f))
          balance)))
```

[redacted]

Better Code: balance can only be changed using the withdraw function



```
(define (new-account initial-balance)
  (let ((balance initial-balance))
    (lambda (withdrawal)
      (if (> withdrawal balance)
          "Insufficient funds"
          (begin
            (set! balance (- balance withdrawal))
          balance)))))
```

[redacted]

OBJECTS

- In our bank account example, the only way to change the balance is through the withdrawal function. Thus, while there is destructive assignment, it is carried out behind an abstraction barrier.
- This fits into a paradigm called *object-oriented* programming:
  - Data objects are permitted private "state" information that can change over time, but
  - All interaction with these objects are carried out by specifically-crafted functions; no other functions have access to the state.

A SHORT ASIDE: TOKENS IN SCHEME

- In Scheme, you can work with “token” values

```
> 'green
green
> (eq? 'green 'Bach)
#f
> (eq? 'green 'green)
#t
```

- No operations are defined on tokens, except for equality.

A MORE SOPHISTICATED BANK ACCOUNT EXAMPLE

The idea:

- As before, we will establish an environment in which balance lives.
- Functions will be defined in this environment with access to the balance, and passed out of the environment.

- Now in this code we have a single method with different functionality based on the token call.

```
(define (new-account initial-balance)
  (let ((balance initial-balance))
    (lambda (method x)
      (cond ((eq? method 'deposit)
              (begin
                (set! balance
                      (+ balance x))
               balance))
            ((eq? method 'withdrawal)
             (if (> x balance)
                 "Insufficient funds"
                 (begin
                   (set! balance
                         (- balance x))
                   balance)))))))
```

TWO EXAMPLES

- Putting this to work

```
> (define my-acct (new-account 200))
> (my-acct 'deposit 250)
450
> (my-acct 'withdrawal 300)
150
```

What happens if we try to access the variable “balance” directly in the global environment?

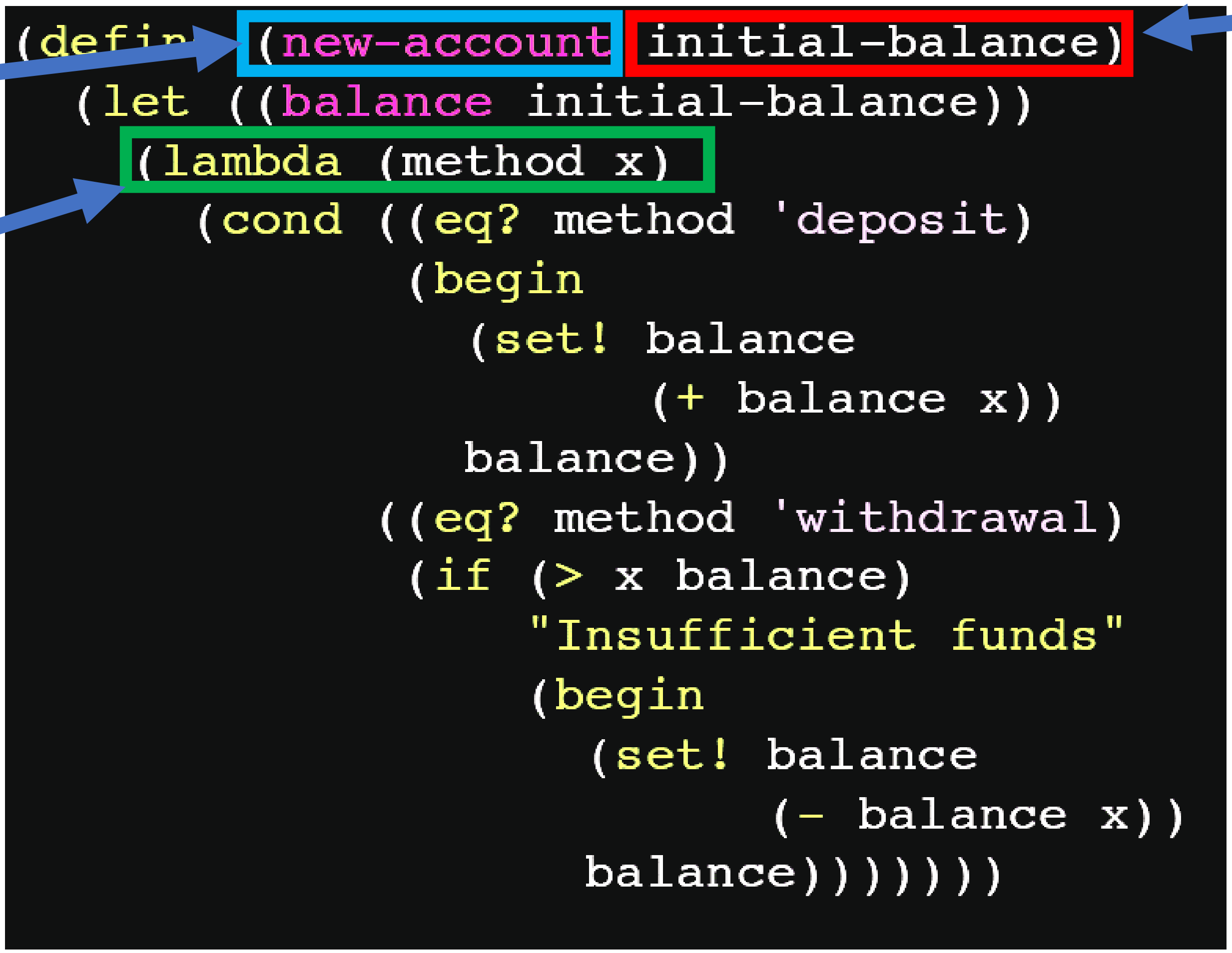
```
> balance
. . reference to undefined identifier: balance
>
```

- The function my-acct is a *dispatcher*;
  - it can provide different functionality depending on the first parameter.
- *Note that balance is undefined in this environment.*

What is the big design picture in Scheme?

Name of the type of object

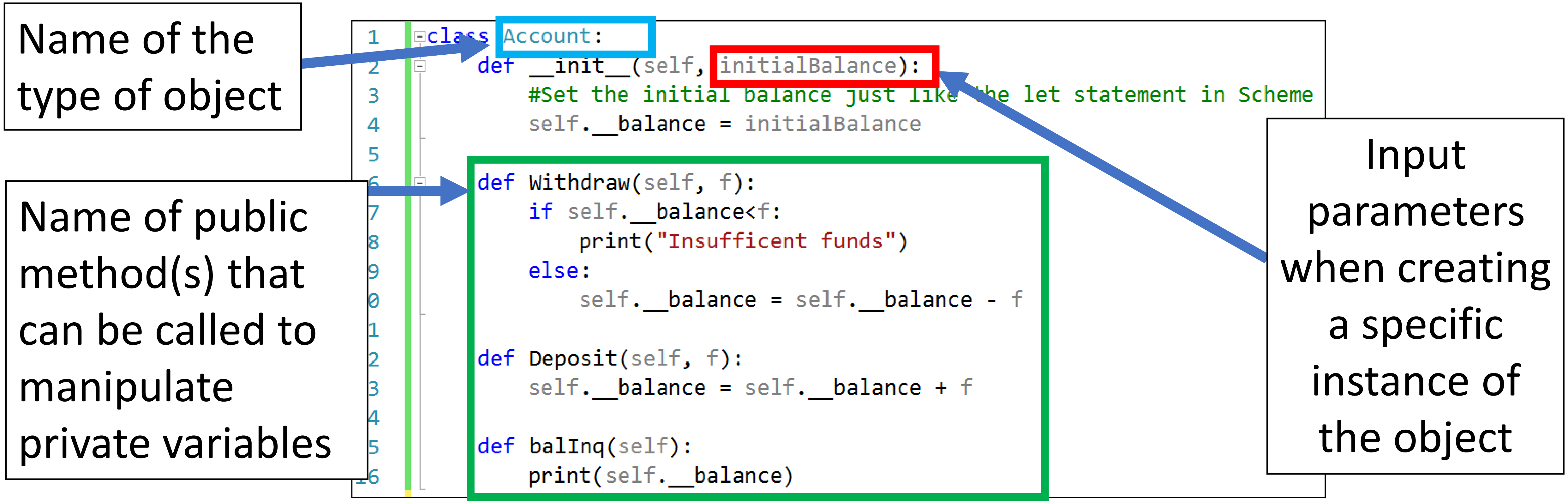
Name of public method that can be called to manipulate private variables



```
(define (new-account initial-balance)
  (let ((balance initial-balance))
    (lambda (method x)
      (cond ((eq? method 'deposit)
                (begin
                  (set! balance
                        (+ balance x))
                  balance))
            ((eq? method 'withdrawal)
             (if (> x balance)
                 "Insufficient funds"
                 (begin
                   (set! balance
                         (- balance x))
                   balance)))))))
```

Input parameters when creating a specific instance of the object

Confused? Let's see the same thing in Python


Name of the type of object
Name of public method(s) that can be called to manipulate private variables
Input parameters when creating a specific instance of the object

```
class Account:
    def __init__(self, initialBalance):
        #Set the initial balance just like the let statement in Scheme
        self.__balance = initialBalance

    def Withdraw(self, f):
        if self.__balance<f:
            print("Insufficent funds")
        else:
            self.__balance = self.__balance - f

    def Deposit(self, f):
        self.__balance = self.__balance + f

    def balInq(self):
        print(self.__balance)
```

REFINING THE SOLUTION: A FIRST ORDER DISPATCHER

- We can slightly improve the solution by allowing the dispatcher to return the appropriate function (rather than applying it itself).
- Methods can now have varying number of arguments.

```scheme
(define (new-account initial-balance)
  (let ((balance initial-balance))
    (define (deposit f)
      (begin
        (set! balance
              (+ balance f))
        balance))
    (define (withdraw f)
      (if (> f balance)
          "Insufficient funds"
          (begin
            (set! balance
                  (- balance f))
            balance)))
    (define (bal-inq) balance)
    (lambda (method)
      (cond ((eq? method 'deposit) deposit)
            ((eq? method 'withdraw) withdraw)
            ((eq? method 'balance-inquire) bal-inq)))))
```

The methods

The dispatcher

USING THE FIRST ORDER DISPATCHER

- Note that the dispatcher now returns the requested *function*.

```
> (define my-account (new-account 500))
> (my-account 'deposit)
#<procedure:deposit>
> ((my-account 'deposit) 200)
700
> ((my-account 'withdraw) 250)
450
> ((my-account 'balance-inquire))
450
```

OBJECTS, IN GENERAL

- Implemented in Scheme by defining an object creator.
  - The creator builds an environment containing the private state variables of the object.
  - It returns a function (which references these local variables in its body). The function can be a dispatcher, as in the last example.
- In general, these functions with access to the private state variables are called *methods* for this object.

ANOTHER EXAMPLE: THE SET ADT IMPLEMENTED AS AN OBJECT

```
(define (set-object)
  (let ((S-list '()))
    (define (is-member? x)
      (define (member-iter remnant)
        (cond ((null? remnant) #f)
              ((eq? x (car remnant)) #t)
              (else (member-iter (cdr remnant)))))
      (member-iter S-list))
    (define (insert x)
      (set! S-list (cons x S-list)))
    (define (empty) (eq? S-list '()))
   (lambda (method)
     (cond ((eq? method 'empty) empty)
           ((eq? method 'member) is-member?)
           ((eq? method 'insert) insert)))))
```

A private variable

The methods;
as defined they have access to S-list

The dispatcher;
returned to caller

THIS SET OBJECT, IN ACTION

```
> (define S (set-object))
> (S 'empty)
#<procedure:empty>
> ((S 'empty))
```

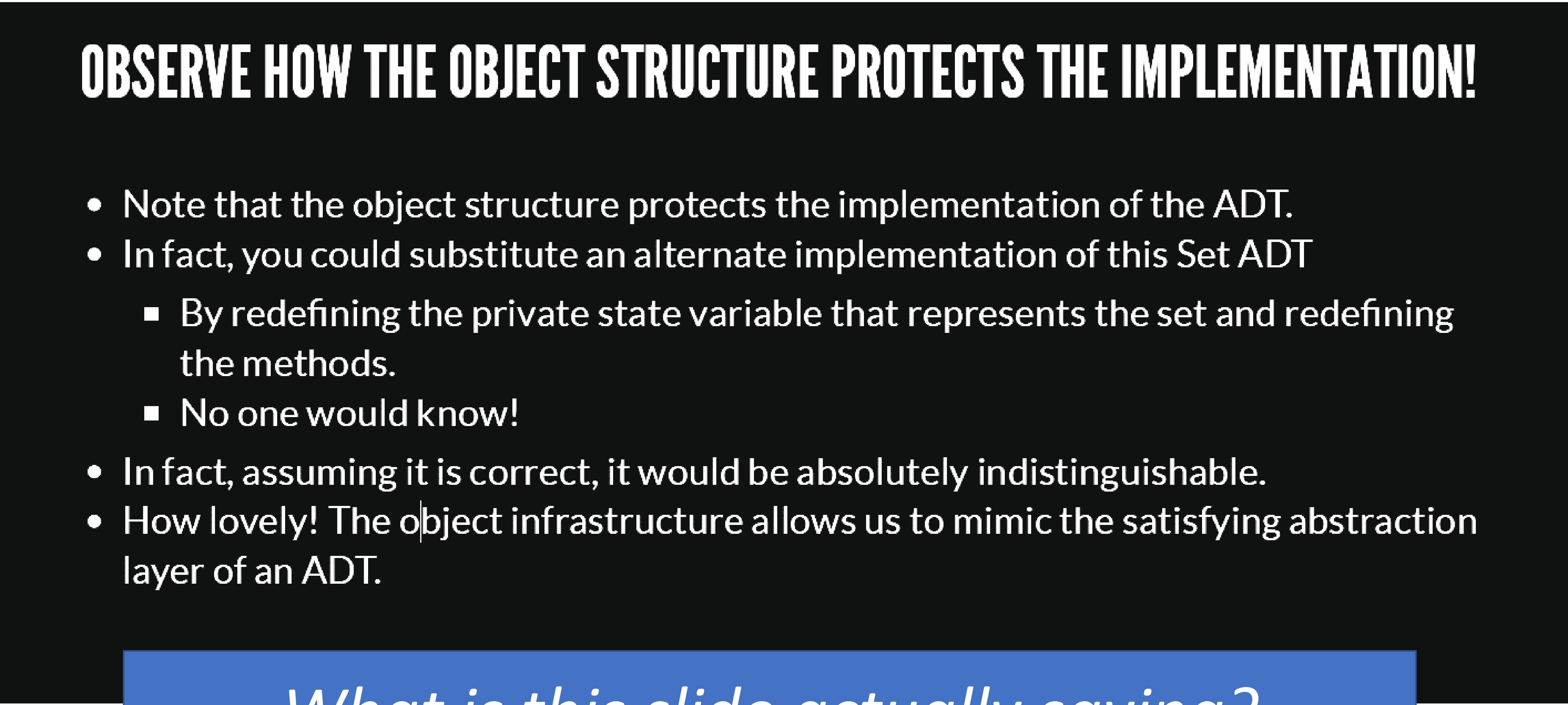
This call creates an environment in which S-value lives

```
#t
> ((S 'insert) 0)
> ((S 'member) 0)
#t
```

S-value does not exist in *this* environment

```
> ((S 'member) 1)
#f
> ((S 'insert) 1)
> ((S 'member) 1)
#t
```

However, these methods can change it!

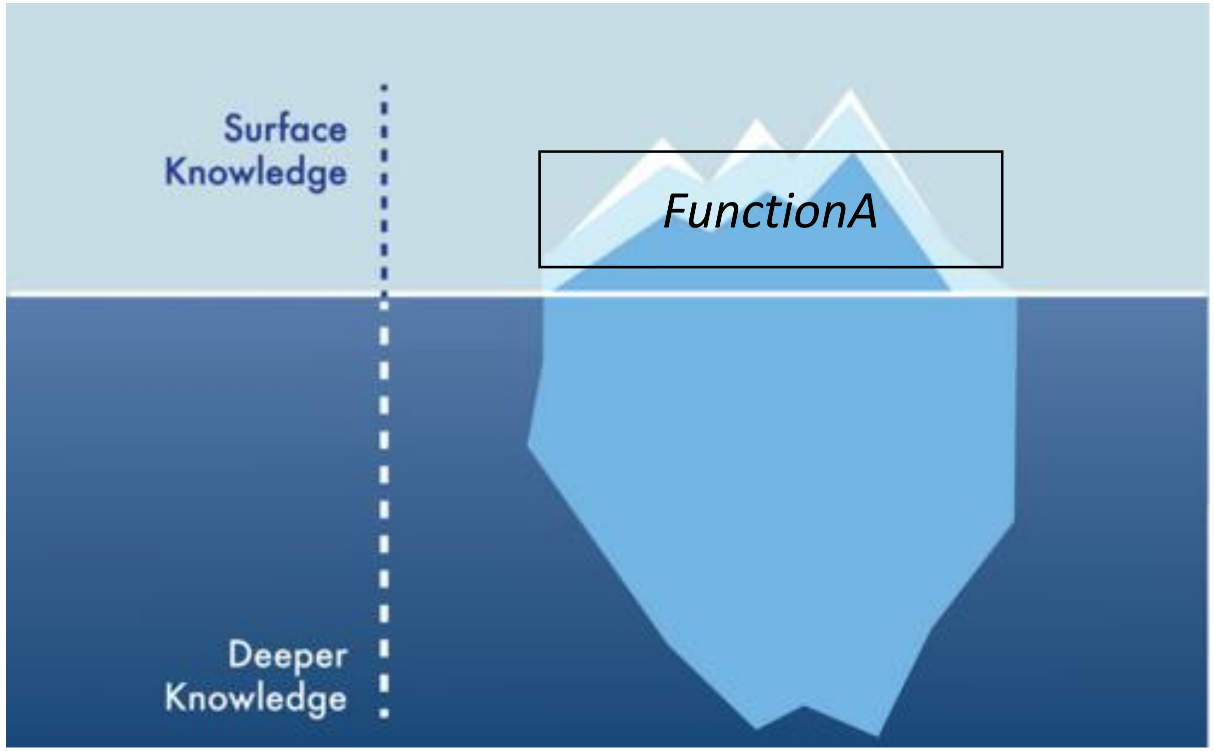


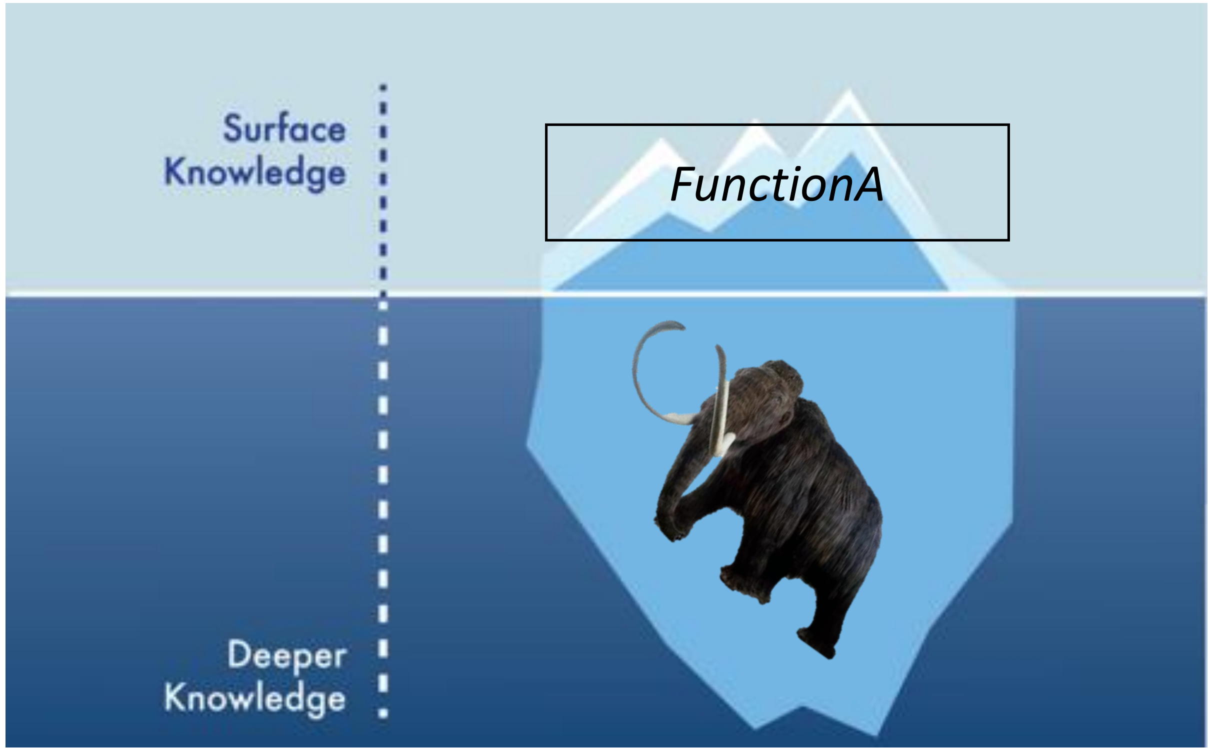
OBSERVE HOW THE OBJECT STRUCTURE PROTECTS THE IMPLEMENTATION!

- Note that the object structure protects the implementation of the ADT.
- In fact, you could substitute an alternate implementation of this Set ADT
  - By redefining the private state variable that represents the set and redefining the methods.
  - No one would know!
- In fact, assuming it is correct, it would be absolutely indistinguishable.
- How lovely! The object infrastructure allows us to mimic the satisfying abstraction layer of an ADT.

What is this slide actually saying?

As long we are given the same method name, we don’t need to look deeper into the code.





In the above pictures both functions have the same interface but what is going on behind the scenes can be very different.

A COMPLEX NUMBER OBJECT

```scheme
(define (make-complex x y)
  (define (square z) (* z z))
  (define (length)
    (sqrt (+ (square x)
             (square y))))
  (define (real-part) x)
  (define (imag-part) y)
  (define (conjugate) (make-complex x (- y)))
  (define (add c) (make-complex (+ x ((c 'real-part)))
                                (+ y ((c 'imag-part)))))
 (lambda (method)
   (cond ((eq? method 'real-part) real-part)
         ((eq? method 'imag-part) imag-part)
         ((eq? method 'length) length)
         ((eq? method 'conjugate) conjugate)
         ((eq? method 'add) add))))
```

Methods with access to x and y

Methods passed back by a dispatcher

EXAMPLES

```
> (define a (make-complex 4 5))
> ((a 'imag-part))
5
```

[redacted]

```
> (define b ((a 'conjugate))
```

Conjugate returns a new object

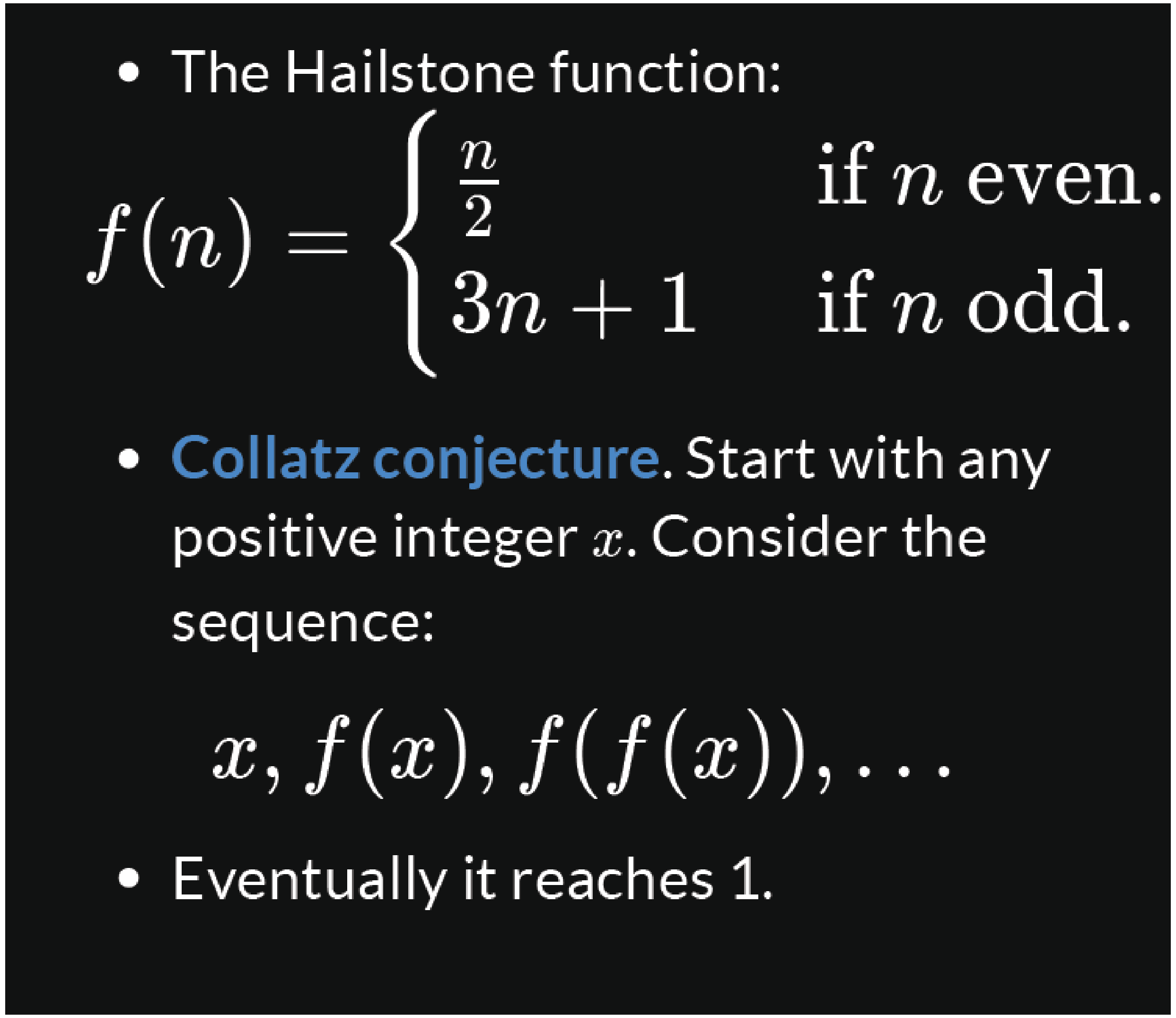
```
> ((b 'imag-part))
-5
> ((((b 'add) (make-complex 5 6)) 'length))
9.055385138137417
>
```

add takes an object as a argument

INTERESTING PROBLEM: MODELING SUM BY OBJECTS

- Note that some methods return objects!
- How do you model addition of complex numbers?
  - In the previous example, we sent one complex to the method of another, and used accessor methods.
  - Alternatively, you could write a function that takes two complexes (as objects) and adds them. Note that this typically requires only “read access” to the internal variables.
- In this case, the object structure does not protect destructive assignment, but it does hide implementation.

One More Example of Modeling Mutable Variables



- The Hailstone function:

$$f(n) = \begin{cases} \frac{n}{2} & \text{if } n \text{ even.} \\ 3n+1 & \text{if } n \text{ odd.} \end{cases}$$

- Collatz conjecture. Start with any positive integer x. Consider the sequence:

$$x, f(x), f(f(x)), \ldots$$

- Eventually it reaches 1.

MODELING SEQUENCES VIA MUTABLE DATA

```
(define (hailstone x)
  (define (next)
    (if (eq? (modulo x 2) 0)
        (begin (set! x (/ x 2)) x)
        (begin (set! x (+ (* 3 x) 1)) x)))
  next)

(define hseq (hailstone 100))
```

environment contains x

Passes `next` back

- Models the hailstone sequence we discussed before. The environment produced by hailstone remembers the current x value.
- The next function updates x and returns the new value.

THE HAILSTONE SEQUENCE IN HIDDEN MUTABLE ACTION

```
> (define hseq (hailstone 25))
> (hseq)
76
> (hseq)
38
> (hseq)
19
> (hseq)
58
> (hseq)
29
> (hseq)
88
> (hseq)
44
> (hseq)
22
> (hseq)
11
> (hseq)
34
> (hseq)
17
> (hseq)
52
> (hseq)
26
> (hseq)
13
> (hseq)
40
> (hseq)
20
> (hseq)
10
> (hseq)
5
> (hseq)
16
> (hseq)
8
> (hseq)
4
> (hseq)
2
> (hseq)
1
>
```

Figure Sources

- https://i0.wp.com/SailEMagazine.com/wp-content/uploads/2014/08/MaryamZainal_TheIcebergTheory_Aug2014_.png?resize=500%2C330
- https://i.guim.co.uk/img/media/584a34b3b86e9aa90eeec81f4d507aaf77c6779e/0_160_4728_2836/master/4728.jpg?width=465&quality=45&auto=format&fit=max&dpr=2&s=321997dee4f9a5adef63ae9184bf4756