

The state of an object

- Objects have properties that change over time, e.g., snowballs, cars, people
- The **state** of an object is the collection of the values of all of its properties at a given moment in time

Representing objects

- To represent an object, we must represent its state
- The measurable properties of an object can be represented by **local state variables**
- To make the values of local state variables change over time, an **assignment operator** is needed, but this will make program behavior harder to predict

Bank account example

```
(bank-statement) --> 1000
(withdraw 50) --> 950
(withdraw 50) --> 900
(withdraw 990) -->
"insufficient funds"
(withdraw 50) --> 850
withdraw is not a mathematical function!
```

An implementation

```
(define balance 1000)
(define (withdraw amount)
  (if (>= balance amount)
      (begin
        (set! balance
              (- balance amount))
        balance)
      "insufficient funds"))
```

Withdrawing from more than one fund

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

A fund is represented by a (withdraw) procedure

A history of withdrawals

```
(define w1 (make-withdraw 1000))
(define w2 (make-withdraw 500))
(w1 90) --> 910
(w2 80) --> 420
(w1 430) --> 480
(w2 430) --> "insufficient funds"
(w1 430) --> 50
```

More realistic bank accounts

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance
                  (- balance amount))
              balance)
        "insufficient funds")))
```

(the new feature)

```
(define (deposit amount)
  (set! balance (+ balance amount))
  balance)
```

(the bank teller)

```
(define (dispatch msg)
  (cond ((eq? msg 'withdraw) withdraw)
        ((eq? msg 'deposit) deposit)
        (else
         (error
          "You can't do that here:"
          msg))))
dispatch)
```

Tale of two bank accounts

```
(define acc1 (make-account 1000))
(define acc2 (make-account 1200))
((acc1 'withdraw) 400) --> 600
((acc2 'deposit) 300) --> 1500
((acc1 'deposit) 200) --> 800
((acc2 'close) -->
```

You can't do that here: close

A benefit of modeling objects

- Often allows more modular program design
- Example: a random number generator

A random number generator (after Park and Miller)

```
(define random-init 123) ; must not be 0
(define (rand-update x)
  (remainder (* 16807 x) 2147483647))
(define rand
  (let ((x random-init))
    (lambda ()
      (set! x (rand-update x))
      (/ x 2147483647.0))))
```


Estimating π without rand

```
(define (estimate-pi trials)
  (* 4.0 (random-circle-test trials
        random-init)))

(define (random-circle-test trials init-ri)
  (define (iter trials-left trials-passed ri0)
    (let ((ri1 (random-update ri0)))
      (let ((ri2 (random-update ri1)))
        (let ((x (- (/ ri1 2147483647.0) 0.5))
              (y (- (/ ri2 2147483647.0) 0.5)))
```

continued

```
(cond ((= trials-left 0)
      (/ trials-passed trials))
      ((<= (+ (square x) (square y))
            0.25)
      (iter (- trials-left 1)
            (+ trials-passed 1)
            ri2))
      (else (iter (- trials-left 1)
                  trials-passed
                  ri2))))))

(iter trials 0 init-ri))
```

Comparison

- First version factors out random number generation as a separable process
- First version allows extraction of Monte Carlo simulation as a general method
- The second version intertwines the two processes
- The second version requires more variables passed in procedure calls

The dark side of assignment

- As we saw, procedures that use assignment may not return the same output for the same input, so they don't compute mathematical functions
- Generally speaking, procedures written without (re)assignment do compute mathematical functions, so this style is called **functional programming**

Behavior comparison

```
(define (make-simplified-withdraw balance)
  (lambda (amount)
    (set! balance (- balance amount))
    balance))
(define W (make-simplified-withdraw 30))
(W 20) --> 10
(W 20) --> -10
```

continued (without set!)

```
(define (make-decrementer balance)
  (lambda (amount) (- balance amount)))
(define D (make-decrementer 30))
(D 20) --> 10
(D 20) --> 10
```

The substitution model succeeds

```
((make-decrementer 30) 20)
((lambda (amount) (- 30 amount))
 20)
(- 30 20)
10
```

The substitution model fails

```
((make-simplified-withdraw 30) 20)
((lambda (amount)
  (set! balance (- 30 amount))
  30)
 20) [balance in set! expr. not
evaluated, so not replaced]
(set! balance (- 30 20))
30
[does't return the new balance]
```

Why substitution fails

- Substitution model treats variables simply as names for values
- Assignment treats variables as places where values can be stored
- Assignment can change the value that is stored at a given place

When are two things the same?

- Philosophically speaking, two things are the same if there are no properties that can distinguish them
- Notion of sameness might depend on the properties that are available for making distinctions

Substituting equals for equals

```
(define D1 (make-decrementer 30))  
(define D2 (make-decrementer 30))  
(D1 20) --> 10  
at this point, (D1 20) --> 10 and  
(D2 20) -->10
```

D1 and D2 compute the same mathematical function, so they could be considered to be the same

In (most) any computation, D1 and D2 can be substituted for each other without affecting the result

(They differ only in location in memory)

Objects with different states are not the same

```
(define W1 (make-simplified-withdraw 30))  
(define W2 (make-simplified-withdraw 30))  
(W1 20) --> 10
```

At this point, (W1 20) --> -10, but

```
(W2 20) --> 10
```

W1 and W2 can't be the same, and substituting one for the other in an expression will change the value of that expression

Referential transparency

- If, for any expression in a language, the value of the expression does not change when substituting different ways of referring to the same thing for each other in the expression, the language is said to be **referentially transparent**.
- Assignment can change what a variable is referring to, so a language with assignment is not referentially transparent.

A philosophical problem

If objects are allowed to change their properties and still be the same object, what changes are allowed without changing an object's identity?

Imperative programming

- Programming that makes extensive use of the assignment operator is called **imperative programming**
- Imperative programming is more sensitive to the order in which expressions are evaluated than is functional programming

Factorial (functional programming style)

```
(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product)
              (+ counter 1))))
  (iter 1 1))
```

(Arguments of calls to iter can be evaluated in either order)

Factorial (imperative prog. style)

```
(define (factorial n)
  (let ((product 1) ;Order of set!
        (counter 1) ;statements is
                    ;important, one more
                    ;thing for a
                    ;programmer to
                    ;product
                    ;begin (set! product ;worry about
                              (* counter product))
                              (set! counter (+ counter 1)))
    (iter)))
(iter)))
```