Object-Oriented Programming

Lecturer:
Professor Albert R. Meyer

October 30, 1997

Bank Account Object

```
(define account1
  (let ((balance 100))      ;AN INSTANCE VARIABLE

        ;THE METHODS:
    (define (deposit amount)
      (if (> amount 0)
          (begin
            (set! balance (+ balance amount))
            "deposit is complete")
          "only positive deposits are allowed"))
    (define (withdraw amount)
      (let ((new-balance (- balance amount)))
        (if (< new-balance 0)
            "insufficient funds"
            (begin
              (set! balance new-balance)
              "withdrawal is complete"))))
    (define (show)
      balance)

            ;THE MESSAGE-PASSING OBJECT:
    (lambda (m)
                    ;MESSAGE        ;METHOD
      (cond ((eq? m 'withdraw)      withdraw)
            ((eq? m 'deposit)       deposit)
            ((eq? m 'show-balance) show)
            (else (fail))))))
```

Using Accounts

```
(define (ask-account account message . args)
  (cond
   ((eq? message 'withdraw)
    ((account 'withdraw) (car args)))
   ((eq? message 'deposit)
    ((account 'deposit) (car args)))
   ((eq? message 'show-balance)
    ((account 'withdraw)))
   (else
    (error
     "ASK-ACCOUNT unknown message: " message))))

(ask-account account1 'show-balance)
;Value: 100

(ask-account account1 'deposit 5)
;Value: "deposit is complete"

(ask-account account1 'show-balance)
;Value: 105
```

Cons-Cell Object

Simulate mutable lists with a message-passing object:

```
(define (make-cell a d)
            ;INSTANCE VARIABLES:
  (let ((the-car a)
        (the-cdr d))

            ;METHODS
    (define (set-the-car! v) (set! the-car v))
    (define (set-the-cdr! v) (set! the-cdr v))
    (define (car-thunk) the-car)
    (define (cdr-thunk) the-cdr)

            ;MESSAGE-PASSING OBJECT
    (lambda (m)
      (cond ((eq? m 'car) car-thunk)
            ((eq? m 'cdr) cdr-thunk)
            ((eq? m 'car!) set-the-car!)
            ((eq? m 'cdr!) set-the-cdr!)
            (else (fail))))))
```

Using Cells

```
(define (ask-cell cell message . args)
  (cond
   ((eq? message 'car)
    ((cell 'car)) )
   ((eq? message 'car!)
    ((cell 'car!) (car args)) )
   ((eq? message 'cdr) ...)
   ((eq? message 'cdr!) ...)
   (else
    (error
     "ASK-CELL unknown message: " message))))

(define (car cell)
 (ask-cell cell 'car))

(define (set-car! cell val)
  (ask-cell cell 'car! val))
```

The ASK Abstraction

```
(define (ask object message . args)
  (let ((method (get-method message object)))
    (if (fail? method)
        (error object "has no method: " message)
        (apply method args))))

  ;look through the objects for the named method:
(define (get-method message . objects)
  (if (null? objects)
      (fail)
      (let ((method ((car objects) message)))
        (if (fail? method)
            (apply get-method
                   (cons message (cdr objects)))
            method))))
```

ASK Account and Cell

```
(ask-account account1 'show-balance)
                ;same effect as:
(ask account1 'show-balance)


(define new-cell (make-cell 3 4))

(ask-cell new-cell 'car)
                ;same effect as:
(ask new-cell 'car)
```

The Object Interface

Henceforth, ”OBJECT” will be a new kind of FIRST-CLASS SCHEME VALUE.

The only primitives which come with this new kind of value are:

```
(ASK object message arg1 arg2 ...)
```

```
(RESPONDS-TO? object message)
```

For example, with the implementation above, we could

```
(define (responds-to? object message)
  (not (fail? (object message))))
```

NOTE: **RESPONDS-TO?** is not needed in strongly typed languages (like Java).

Classes of Objects

How to construct these new OBJECT values?

CLASSES specify object CONTRUCTORS.

```
(define-class
  (CLASS-NAME                ;KEYWORD
    bank-account)

  (SLOTS                     ;KEYWORD
   (balance 'uninitialized))

  (CONSTRUCTOR               ;KEYWORD
   (lambda (initial-balance)
     (set! balance initial-balance)))

  (METHODS                   ;KEYWORD
    (DEPOSIT                    ;MESSAGE
     (lambda (amount)           ;METHOD
       (if (> amount 0) ...)))
    (WITHDRAW                   ;message
     (lambda (amount)           ;method
       (let ((new-balance (- balance amount)))
         (if (< new-balance 0) ...))))
    (SHOW-BALANCE               ;message
     (lambda () balance))))     ;method

(define account1 (NEW bank-account 100))
```

account1 is an **INSTANCE** of the class **bank-account**.

The Cons-Cell Class

```
(define-class
  (CLASS-NAME cons-cell)

  (SLOTS (the-car 'uninitialized)
         (the-cdr 'uninitialized))

  (CONSTRUCTOR
   (lambda (a d)
     (begin
       (set! the-car a)
       (set! the-cdr d))

  (METHODS
    (CAR
     (lambda () the-car))
    (CAR!
     (lambda (v) (set! the-car v)))
    (CDR ...)
    (CDR! ..)))

(define new-cell (NEW cons-cell 3 4))

(ask new-cell 'car)
;Value: 3

(ask new-cell 'car! 7)
(ask new-cell 'car)
;Value: 7
```

Self

Add an **UPDATE** method for **bank-account**s which **ASK**s the account *itself* to deposit or withdraw:

```
(define-class
  (CLASS-NAME bank-account)

  (SLOTS (balance 'uninitialized))

  (CONSTRUCTOR (lambda (initial-balance) ...))

  (METHODS
   (DEPOSIT
    (lambda (amount)
      (if (> amount 0) ...)))
   (WITHDRAW
    (lambda (amount) ...))
   (SHOW-BALANCE (lambda () balance))
   (UPDATE                ;;ADDED MESSAGE
    (lambda (amount)
      (if
       (> amount 0)
       (ask SELF 'deposit amount)            ;****
       (ask SELF 'withdraw (- amount))))))) ;****
```

In JAVA and C++, **self** is called **this**.

The Named-Class

From Problem Set 7:

```
(define-class
  (CLASS-NAME named-class)

  (SLOTS (name 'not-initialized-yet))

  (CONSTRUCTOR
   (lambda (initial-name)
     (set! name initial-name))

  (METHODS
    (INSTANCE-OF?
     (lambda (class-name)
       (eq? class-name 'NAMED-CLASS)))
    (NAME
     (lambda () name))
    (SAY
     (lambda (words)
       (for-each display words)
       'NUF-SAID))))
```

INHERITANCE

A objects which are **mobile** have **names**, but **also** have a **location** and methods for moving around.

The **`mobile-class`** **INHERITS** from the **`named-class`**.

This means that every mobile object has a **named part**.

```
(define-class
  (CLASS-NAME mobile-class)
  (PARTS (named-part))            ;***
  (SLOTS (location 'not-initialized-yet))
  (CONSTRUCTOR
   (lambda (initial-name initial-location)
     (set! named-part
           (NEW-PART              ;***
            named-class self initial-name))
     (set! location initial-location)
     (ask location 'ADD-THING self))
  (METHODS
    (INSTANCE-OF?
     (lambda (class-name)
       (or (eq? class-name 'MOBILE-CLASS)
           (ask named-part
                'INSTANCE-OF? class-name))))
    (PLACE (lambda () location))
    (GO-TO
       (lambda (new-place)
         (begin
          (ask location 'DEL-THING self)
          (set! location new-place)
          (ask location 'ADD-THING self))))))
```

Inherited Methods

Mobile objects do not have an explicit **NAME** message. They *inherit* the NAME method of their named part:

```
(define mobile-obj
   (new mobile-class ...))

(ask mobile-obj 'name)
                ;winds up as
(ask <named-part-of mobile-obj> 'name)
```

Persons inherit mobility

person objects are **mobile**, but **also** have **possessions and restlessness** and methods for them.

```
(define-class
  (CLASS-NAME person-class)
  (PARTS (mobile-part))
  (SLOTS (possessions '())
         (restlessness 'uninitialized))
  (CONSTRUCTOR
   (lambda (initial-name
            initial-location
            initial-restlessness)
     (set! mobile-part
           (NEW-PART
            mobile-class self
            initial-name initial-location))
     (set! restlessness initial-restlessness)))
  (METHODS
     (INSTANCE-OF?
      (lambda (class-name)
        (if (eq? class-name 'PERSON-CLASS)
            true
            (ask mobile-part
                 'INSTANCE-OF? class-name))))
     (POSSESSIONS
      (lambda () possessions))
     (DESCRIBE-POSSESSIONS ...)
     (SAY ...)))
```

```
(DESCRIBE-POSSESSIONS
 (lambda ()
                          ;*** ASK SELF 'SAY:
   (ask self 'SAY
        (if (null? possessions)
            '("nothing.")
            (map (lambda (p) (ask p 'NAME))
                 possessions)))))
(SAY
 (lambda (words)
   (begin
     (map display
          (list
           "At " (ask (ask self 'PLACE) 'NAME)
           ": "  (ask self 'NAME) " says -- "))

                          ;*** ASK NAMED-PART 'SAY:
     (ask named-part 'SAY words))))
```

New Part

```
(define named-obj
  (NEW named-class 'anna))
```

causes the **self** of **named-obj** to be a

named-class MESSAGE-PASSING OBJECT.

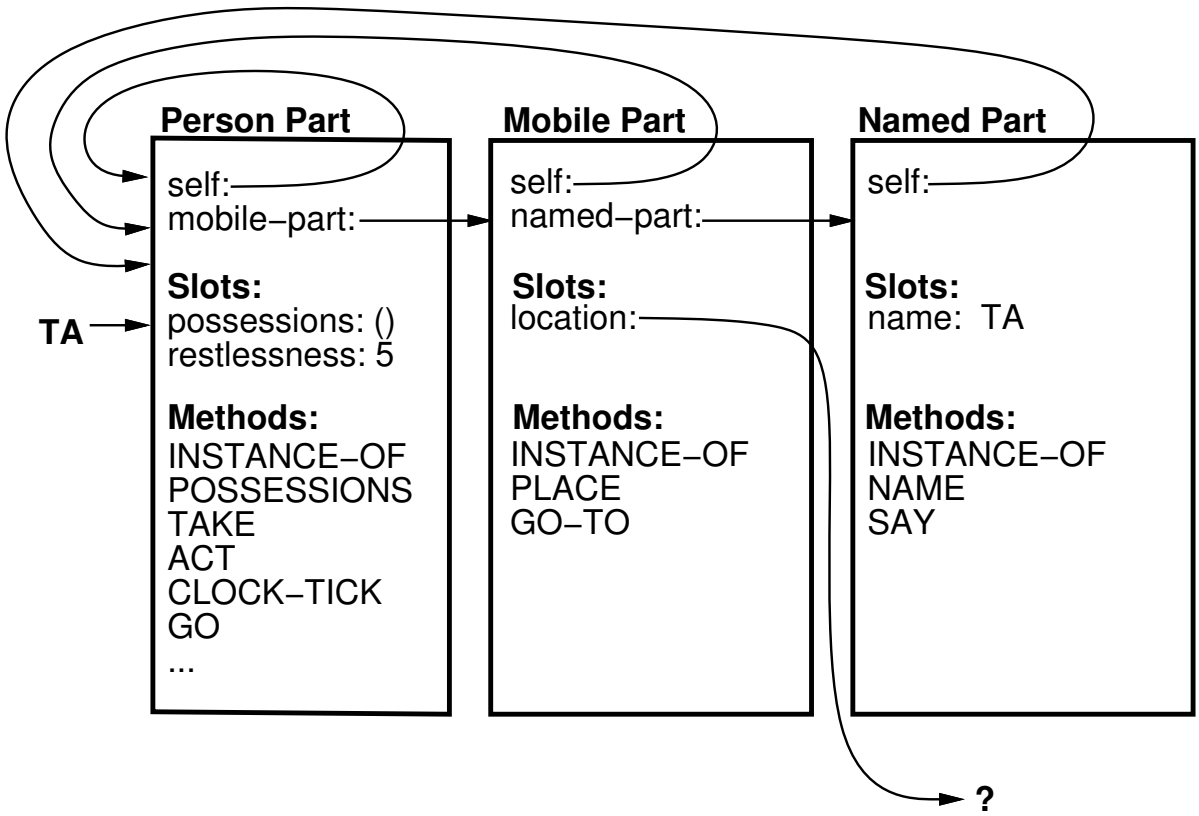
```
(define named-obj
  (NEW-PART named-class <OTHER-OBJECT> 'anna))
```

causes the **self** of **named-obj** to be

<OTHER-OBJECT>.

The TA object

```
(define TA-office (new place-class 'TA-office))
(define TA (new person-class 'TA TA-office 5))
```



Summary

- An *Object* has local variables called *instance variables* and has procedures called *methods* which can effect them.
- An object is a *black box*: one can only **`ASK`** it to do things or see what messages it **`RESPONDS-TO?`**.
- *Classes* specify objects.
- An Object is an *instance* of the Class which specifies it.
- The **`NEW`** procedure constructs instances of a class.
- A class C may *inherit* from other classes D, E, ... (multiple inheritance is possible). This means that every C-object will have a D-object and an E-object and ... as *parts*.
- An object has a *SELF* which is the ”bottom-most” subobject that the object is part of.
- Problem Set 7 has 14 pages of class definitions.
- NEXT TIME: IMPLEMENTING CLASSES AND OBJECTS.