Object-Oriented Programming
MIT 6.001 Recitation

Word to the wise

Start project 4 early.

Case-sensitive

DrScheme is by default case-sensitive, MIT scheme is not. This can be a problem for the PSets. You can fix it by going to DrScheme's Choose language dialogue, click on show detail, and in Pretty Big, uncheck the case-sensitive checkbox.

Concepts

Class: the common structure of a type of object

Instance: a particular object of a given class

Inheritance: a subclass can extend the functionality of one or more superclasses

Always inherit from root object (often indirectly).

Shadowing of methods: lookup method following up a chain and finding first handler for that message

We go up the inheritance hierarchy to find a method not defined in the subclass.

The way to go back to the bottom is to ask self.

If there's a method that does what you want, use it! (not using it is an abstraction violation).

Almost always ask self, except when you extend (shadow) a method.

Create instance vs. make handler/container: why do some objects have a create-foo & a foo and other objects just have a foo?

Create-instance: makes a handler; creates a (tagged) instance that contains the handler; installs the object in the world if appropriate.

We cannot access the state of a superclass directly, we need to ask self.

The type method should always be overridden. The superclass must always be passed when calling make-handler.

Results from all messages are procedures. ask allows you to apply the procedure.

Template for class definitions:

```
(define (type self arg1 arg2 ... argn )
  (let (  (super1-part (super1 self args)
          ; other superclasses
          ; other local state )
    (make-handler
      type
      (make-methods
        message-name-1 method-1
        other messages and methods
      )
    super1-part othersuperparts ...)))
```

Note that arg1…argn are also stored as local state.

OOP review

* what's the difference between an instance and a class?

* what's the difference between a class inheritance hierarchy & environment diagram?

* what's the convention for the object procedures create-foo and foo? why do some objects only have a foo procedure?

* what 2 features should be present in the code of a class foo that inherits from another class bar?

* what's the difference between (ask self message) and (ask foo-part message)? in what cases do you have to use self and in what cases do you have to use foo-part?

OOP, inheritance and callbacks are ubiquitous for GUI implementation.

Consistency and proper assignment of responsibilities are crucial for a good design. Hence the importance of class and instance diagrams.

Diagrams

Look at the code below. Where do you find the list of states and method for a class such as named-object or container.

Draw a class diagram for a thing.

Draw a class diagram for a person

Draw the environment diagram for a thing instance (create-thing ‘stuff statacenter).

Use the classes

Create a “toy” mobile-thing instance.

Create a person bob.

Have bob take the thing. What happens exactly in terms of inheritance and message handling?

Draw an instance diagram.

How do you get the name of all the things carried by bob?

How do you get the name of their location?

Create a new method for the class person that takes all the things of another person.

Inheritance

We are going to derive new classes for the adventure game. As you go, ask yourself these questions:

1. What should class or classes should it inherit from?
2. What new fields does it need?
3. What new methods does it need?
4. What methods from its superclass(es) should it override? Should the behavior of its overrided methods completely replace what the supperclass does, or just augment it? How should superclass methods be called from an overriding method?

Ideas of classes

A **monk** refuses all possessions. (Hint: a person can be asked to TAKE something.)

A **bomb**, when triggered, destroys everything around it. (Hint: a thing has a LOCATION, which is a container with a list of THINGS.)

A **recorder** remembers everything it ever said and can replay it all on command. (Hint: a person can SAY something.)

Achile, never gets hurt.

A **person with a small pocket** can only carry 2 objects.

Derive a **weapons** system (include notions such as maximum of damage, ammunition).

Code

This code is often simplified for legibility, but it roughly follows that of project 4.

```
;;-------------------------------------------------------------
;; Instance

(define (make-instance)
  (list 'instance #f))

(define (create-instance maker . args)
  (let* ((instance (make-instance))
         (handler (apply maker instance args)))
    (set-instance-handler! instance handler)
    (if (method? (get-method 'INSTALL instance))
        (ask instance 'INSTALL))
    instance))


;;-------------------------------------------------------------
;; Handler

(define (make-handler typename methods . super-parts)
  […]    ;check for possible programmer errors (omitted for legibility
   (lambda (message)
     (case message
       ((TYPE) (lambda () (type-extend typename super-parts)))
       ((METHODS) (lambda ()
           (append (method-names methods)
             (append-map (lambda (x) (ask x 'METHODS)) super-parts))))
       (else
        (let ((entry (method-lookup message methods)))
           (if entry (cadr entry)
             (find-method-from-handler-list message super-parts)))))))


;;--------------------
;; ask

 (define (ask object message . args)
  (let ((method (get-method message object)))
    (cond ((method? method)
           (apply method args))
          (else
           (error "No method for" message 'in (safe-ask 'UNNAMED-OBJECT object 'NAME))))))
```

```
;;--------------------
;; named-object

(define (create-named-object name)   (create-instance named-object name))

(define (named-object self name)
  (let ((root-part (root-object self)))
    (make-handler
     'named-object
     (make-methods
      'NAME    (lambda () name)
      'INSTALL (lambda () 'installed)
      'DESTROY (lambda () 'destroyed))
     root-part)))


;;--------------------
;; container

(define (container self)
  (let ( (root-part (root-object self))
         (things '()))
    (make-handler
     'container
     (make-methods
      'THINGS      (lambda () things)
      'HAVE-THING? (lambda (thing) (not (null? (memq thing things))))
      'ADD-THING   (lambda (thing)
        (if (not (ask self 'HAVE-THING? thing))
            (set! things (cons thing things))) 'DONE)
      'DEL-THING   (lambda (thing)
         (set! things (delq thing things)) 'DONE))
     root-part)))


;;--------------------
;; thing

(define (create-thing name location)    ;symbol, location -> thing
  (create-instance thing name location))

(define (thing self name location)
  (let ((named-part (named-object self name)))
    (make-handler
     'thing
     (make-methods
      'INSTALL  (lambda ()
         (ask named-part 'INSTALL)
         (ask (ask self 'LOCATION) 'ADD-THING self))
      'LOCATION (lambda () location)
      'DESTROY  (lambda () (ask (ask self 'LOCATION) 'DEL-THING self))
      'EMIT     (lambda (text)
         (ask screen 'TELL-ROOM (ask self 'LOCATION)
          (append (list "At" (ask (ask self 'LOCATION) 'NAME)) text))))
     named-part)))
```

```
;;--------------------
;; mobile-thing

(define (create-mobile-thing name location)
  (create-instance mobile-thing name location))

(define (mobile-thing self name location)
  (let ((thing-part (thing self name location)))
    (make-handler
     'mobile-thing
     (make-methods
      'LOCATION  (lambda () location) ; This shadows message to thing-part!
      'CHANGE-LOCATION
        (lambda (new-location)     (ask location 'DEL-THING self)
                                   (ask new-location 'ADD-THING self)
                                   (set! location new-location))
      'ENTER-ROOM    (lambda () #t)
      'LEAVE-ROOM    (lambda () #t)
      'CREATION-SITE (lambda () (ask thing-part 'location)))
     thing-part)))

;;--------------------
;; place

(define (create-place name)     ; symbol -> place
  (create-instance place name))

(define (place self name)
  (let ((named-part (named-object self name))
        (container-part (container self))
        (exits '()))
    (make-handler
     'place
     (make-methods
      'EXITS (lambda () exits)
      'EXIT-TOWARDS (lambda (direction) (find-exit-in-direction exits direction))
      'ADD-EXIT (lambda (exit)
         (let ((direction (ask exit 'DIRECTION)))
         (if (ask self 'EXIT-TOWARDS direction) (error (list name "already has exit" direction))
               (set! exits (cons exit exits))) 'DONE)))
     container-part named-part)))

;;------------------------------------------------------------
;; exit

(define (create-exit from direction to)
  ; place, symbol, place -> exit
  (create-instance exit from direction to))

(define (exit self from direction to)
  (let ((named-object-part (named-object self direction)))
    (make-handler
     'exit
     (make-methods
      'INSTALL (lambda () (ask named-object-part 'INSTALL)
         (if (not (null? (ask self 'FROM))) (ask (ask self 'FROM) 'ADD-EXIT self)))
      'FROM         (lambda () from)
      'TO           (lambda () to)
      'DIRECTION    (lambda () direction)
      'USE (lambda (whom) (ask whom 'LEAVE-ROOM)
       (ask screen 'TELL-ROOM (ask whom 'location)(list (ask whom 'NAME) "moves from"
         (ask (ask whom 'LOCATION) 'NAME) "to" (ask to 'NAME)))
       (ask whom 'CHANGE-LOCATION to)
       (ask whom 'ENTER-ROOM)))
     named-object-part)))

(define (find-exit-in-direction exits dir)
  ; Given a list of exits, find one in the desired direction.
  (cond ((null? exits) #f)
        ((eq? dir (ask (car exits) 'DIRECTION))
         (car exits))
        (else (find-exit-in-direction (cdr exits) dir))))
(define (random-exit place)
  (pick-random (ask place 'EXITS)))
```

```
;;--------------------
;; person

(define (create-person name birthplace) ; symbol, place -> person
  (create-instance person name birthplace))

(define (person self name birthplace)
  (let ((mobile-thing-part (mobile-thing self name birthplace))
        (container-part    (container self))
        (health            3)
        (strength          1))
    (make-handler
     'person
     (make-methods
      'STRENGTH (lambda () strength)
      'HEALTH (lambda () health)
      'SAY (lambda (list-of-stuff) (ask screen 'TELL-ROOM (ask self 'location)
           (append (list "At" (ask (ask self 'LOCATION) 'NAME)
           (ask self 'NAME) "says --") list-of-stuff)) 'SAID-AND-HEARD)
      'HAVE-FIT (lambda () (ask self 'SAY '("Yaaaah! I am upset!"))
                  'I-feel-better-now)
      'PEOPLE-AROUND        ; other people in room...
         (lambda () (delq self (find-all (ask self 'LOCATION) 'PERSON)))
      'STUFF-AROUND         ; stuff (non people) in room...
         (lambda () (let* ((in-room (ask (ask self 'LOCATION) 'THINGS))
           (stuff (filter (lambda (x) (not (ask x 'IS-A 'PERSON))) in-room))) stuff))
      'PEEK-AROUND          ; other people's stuff...
         (lambda () (let ((people (ask self 'PEOPLE-AROUND)))
           (fold-right append '() (map (lambda (p) (ask p 'THINGS)) people)))
      'TAKE (lambda (thing)
         (cond ((ask self 'HAVE-THING? thing)  ; already have it
             (ask self 'SAY (list "I am already carrying" (ask thing 'NAME))) #f)
           ((or (ask thing 'IS-A 'PERSON) (not (ask thing 'IS-A 'MOBILE-THING)))
             (ask self 'SAY (list "I try but cannot take" (ask thing 'NAME)))  #F)
         (else (let ((owner (ask thing 'LOCATION)))
           (ask self 'SAY (list "I take" (ask thing 'NAME) "from" (ask owner 'NAME)))
           (if (ask owner 'IS-A 'PERSON) (ask owner 'LOSE thing self)
           (ask thing 'CHANGE-LOCATION self))
         thing))))
      'LOSE   (lambda (thing lose-to)
         (ask self 'SAY (list "I lose" (ask thing 'NAME)))
         (ask self 'HAVE-FIT)
         (ask thing 'CHANGE-LOCATION lose-to))
      'DROP (lambda (thing)
         (ask self 'SAY (list "I drop" (ask thing 'NAME) "at" (ask (ask self 'LOCATION) 'NAME)))
         (ask thing 'CHANGE-LOCATION (ask self 'LOCATION)))
      'GO-EXIT (lambda (exit) (ask exit 'USE self))
      'GO (lambda (direction) ; symbol -> boolean
         (let ((exit (ask (ask self 'LOCATION) 'EXIT-TOWARDS direction)))
         (if (and exit (ask exit 'IS-A 'EXIT)) (ask self 'GO-EXIT exit)
           (begin  (ask screen 'TELL-ROOM (ask self 'LOCATION)
                   (list "No exit in" direction "direction"))
           #F))))
      'SUFFER (lambda (hits perp)
         (ask self 'SAY (list "Ouch!" hits "hits is more than I want!"))
         (set! health (- health hits))
         (if (<= health 0) (ask self 'DIE perp))
         health)
      'DIE    (lambda (perp)
         (for-each (lambda (item) (ask self 'LOSE item (ask self 'LOCATION))) (ask self 'THINGS))
         (ask screen 'TELL-WORLD '("An earth-shattering, soul-piercing scream is heard..."))
         (ask self 'DESTROY))
      'ENTER-ROOM (lambda ()
         (let ((others (ask self 'PEOPLE-AROUND)))
         (if (not (null? others)) (ask self 'SAY (cons "Hi" (names-of others)))))
         #T))
       'HAS-A             ;your method here (exercise 2)
       'HAS-A-THING-NAMED ;your method here (exercise 2)
     mobile-thing-part container-part)))
```