6.001 SICP
Object Oriented Programming

- Data Abstraction using Procedures with State
- Message-Passing
- Object Oriented Modeling
  - Class diagrams
  - Instance diagrams
- Example: spacewar simulation

The role of abstractions

- Procedural abstractions
- Data abstractions

Goal: treat complex things as primitives, and hide details

Questions:
- How easy is it to break system into abstraction modules?
- How easy is it to extend the system?
  - Adding new data types?
  - Adding new methods?

One View of Data

- Data structures
  - Some complex structure constructed from cons cells
    - point, line, 2dshape, 3dshape
  - Explicit tags to keep track of data types
    - (define (make-point x y) (list 'point x y))
  - Implement a data abstraction as a set of procedures that operate on the data

*“Generic”* operations by looking at types:

```scheme
(define (scale x factor)
  (cond ((point? x) (point-scale x factor))
        ((line? x) (line-scale x factor))
        ((2dshape? x) (2dshape-scale x factor))
        ((3dshape? x) (3dshape-scale x factor))
        (else (error "unknown type"))))
```

Generic Operations

- Adding new methods
  - Just create generic operations
- Adding new data types
  - Must change every generic operation
  - Must keep names distinct

<table>
<thead>
<tr>
<th></th>
<th>Point</th>
<th>Line</th>
<th>2-dShape</th>
<th>3-dShape</th>
</tr>
</thead>
<tbody>
<tr>
<td>scale</td>
<td>point-scale</td>
<td>line-scale</td>
<td>2dshape-scale</td>
<td>3dshape-scale</td>
</tr>
<tr>
<td>translate</td>
<td>point-trans</td>
<td>line-trans</td>
<td>2dshape-trans</td>
<td>3dshape-trans</td>
</tr>
<tr>
<td>color</td>
<td>point-color</td>
<td>line-color</td>
<td>2dshape-color</td>
<td>3dshape-color</td>
</tr>
<tr>
<td>new-op</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Views of The World

<table>
<thead>
<tr>
<th></th>
<th>Point</th>
<th>Line</th>
<th>2-dShape</th>
<th>3-dShape</th>
<th>curve</th>
</tr>
</thead>
<tbody>
<tr>
<td>scale</td>
<td>point-scale</td>
<td>line-scale</td>
<td>2dshape-scale</td>
<td>3dshape-scale</td>
<td>scale-c</td>
</tr>
<tr>
<td>translate</td>
<td>point-trans</td>
<td>line-trans</td>
<td>2dshape-trans</td>
<td>3dshape-trans</td>
<td>trans-c</td>
</tr>
<tr>
<td>color</td>
<td>point-color</td>
<td>line-color</td>
<td>2dshape-color</td>
<td>3dshape-color</td>
<td>color-c</td>
</tr>
<tr>
<td>new-op</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Thinking About Data Objects

• A data type, but….
  • it has operations associated with it
  • we want both the generic concept (a line), and a specific instance (line17)
  • the specific instance can have private data associated with it (e.g., its endpoints)
• AKA: object oriented programming

Data Objects: Simple (?) Example

• What’s a cons cell (pair)?
  •
• What about a pair as a data object?

Scheme OOP: Procedures with State As Objects

• Procedure’s state supplies the mechanism for building an object
  • holds the private data for that object
  • allows us to associate operations with the object

Scheme OOP: Procedures with State

• A procedure has
  • parameters and body as specified by λ expression
  • environment (which can hold name-value bindings!)
• Can use procedure to encapsulate (and hide) data, and provide controlled access to that data
  • Procedure application creates private environment
  • Need access to that environment
    • constructor, accessors, mutators, predicates, operations
    • mutation: changes in the private state of the procedure

Example: Pair as a Procedure with State

\[
\text{Example: What is our "pair" object?}
\]

\[
\text{Example: What is our "pair" object?}
\]

\[
\text{Example: What is our "pair" object?}
\]
Pair Mutation as Change in State

```
(define (cons x y)
  (lambda (msg)
    (cond ((eq? msg 'CAR) x)
          ((eq? msg 'CDR) y)
          ((eq? msg 'PAIR?) #t)
          ((eq? msg 'SET-CAR!) (lambda (new-car) (set! x new-car)))
          ((eq? msg 'SET-CDR!) (lambda (new-cdr) (set! y new-cdr)))
          (else (error "pair cannot" msg))))

(define (set-car! p new-car)
  ((p 'SET-CAR!) new-car))

(define (set-cdr! p new-cdr)
  ((p 'SET-CDR!) new-cdr))
```

Example: Mutating a pair object

```
(define bar (cons 3 4))
(set-car! bar 0)
(set-cdr! bar 0)
```

Message Passing Style - Refinements

- lexical scoping for private state and private procedures

```
(define (cons x y)
  (lambda (msg . args)
    (cond ((eq? msg 'CAR) x)
          ((eq? msg 'CDR) y)
          ((eq? msg 'PAIR?) #t)
          ((eq? msg 'SET-CAR!) (change-car (first args)))
          ((eq? msg 'SET-CDR!) (change-cdr (first args)))
          (else (error "pair cannot" msg))))

(define (car p)           (p 'CAR))
(define (set-car! p val)  (p 'SET-CAR! val))
```

Variable number of arguments

A scheme mechanism to be aware of:
- Desire:
  - (add 1 2)
  - (add 1 2 3 4)

- How do this?
  - (define (add x y . rest)  ...)
  - (add 1 2)        =>  x bound to 1
    y bound to 2
    rest bound to '()
  - (add 1)          => error; requires 2 or more args
  - (add 1 2 3)      => rest bound to (3)
  - (add 1 2 3 4 5)  => rest bound to (3 4 5)

Message Passing Style - Refinements

- lexical scoping for private state and private procedures

```
(define (cons x y)
  (lambda (msg . args)
    (cond ((eq? msg 'CAR) x)
          ((eq? msg 'CDR) y)
          ((eq? msg 'PAIR?) #t)
          ((eq? msg 'SET-CAR!) (change-car (first args)))
          ((eq? msg 'SET-CDR!) (change-cdr (first args)))
          (else (error "pair cannot" msg))))

(define (car p)           (p 'CAR))
(define (set-car! p val)  (p 'SET-CAR! val))
```

Programming Styles – Procedural vs. Object-Oriented

- Procedural programming:
  - Organize system around procedures that operate on data
    - (do-something <data> <arg> ...)
    - (do-another-thing <data>)

- Object-based programming:
  - Organize system around objects that receive messages
    - (object) 'do-something <arg>
    - (object) 'do-another-thing
  - An object encapsulates data and operations
Object-Oriented Programming Terminology

- **Class**:  
  - specifies the common behavior of entities  
  - in scheme, a "maker" procedure  

- **Instance**:  
  - A particular object or entity of a given class  
  - in scheme, an instance is a message-handling procedure made by the maker procedure

Using classes and instances to design a system

- Suppose we want to build a spacewar game  
- I can start by thinking about what kinds of objects do I want (what classes, their state information, and their interfaces)  
  - ships  
  - planets  
  - other objects  
- I can then extend to thinking about what particular instances of objects are useful  
  - Millenium Falcon  
  - Enterprise  
  - Earth

A Space-Ship Object

```scheme
(define make-ship (lambda (position velocity num-torps)  
    (lambda (msg)  
      (cond ((eq? msg 'POSITION) position)  
            ((eq? msg 'VELOCITY) velocity)  
            ((eq? msg 'MOVE) (move))  
            ((eq? msg 'ATTACK) (fire-torp))  
            (else (error "ship can't" msg))))))
```

Example – Instance Diagram

```scheme
(define enterprise (make-ship (make-vect 10 10) (make-vect 5 0) 3))  
(define war-bird (make-ship (make-vect -10 10) (make-vect 5 0) 0))
```
Example – Environment Diagram

(define enterprise
  (make-ship (make-vect 10 10) (make-vect 5 0) 3))
(enterprise 'POSITION) => (vect 15 10)

Some Extensions to our World

- Add a PLANET class to our world
- Add predicate messages so we can check type of objects
- Add display handler to our system
  - Draws objects on a screen
  - Can be implemented as a procedure (e.g. draw) -- not everything has to be an object!
  - Add DISPLAY message to classes so objects will display themselves upon request (by calling draw procedure)

Space-Ship Class

```lisp
(ship
  (position: (vec 15 10)
    (velocity: (vec 5 0)
      num-torps: 3
      fire-torp:))
  par: msg
  body: (cond ...) body: (set! position ...))
```

Planet Implementation

```lisp
(define (make-planet position)
  (lambda (msg)
    (cond ((eq? msg 'PLANET?) #t)
        ((eq? msg 'POSITION) position)
        ((eq? msg 'DISPLAY) (draw ...))
        (else (error "planet can't" msg))))
```

Further Extensions to our World

- Animate our World!
  - Add a clock that moves time forward in the universe
  - Keep track of things that can move (the universe)
  - Clock sends ACTIVATE message to objects to have them update their state
- Add TORPEDO class to system

Class Diagram
Coordinating with a clock

The Universe and Time

(define (make-clock . args)
  (let ((the-time 0)
        (callbacks '()))
    (lambda (message)
      (case message
        ((CLOCK?) (lambda (self) #t))
        ((NAME) (lambda (self) name))
        ((THE-TIME) (lambda (self) the-time))
        ((TICK) (lambda (self)
                   (map (lambda (x) (ask x 'activate)) callbacks)
                   (set! the-time (+ the-time 1))))
        ((ADD-CALLBACK) (lambda (self cb)
                         (set! callbacks (cons cb callbacks)) 'added))))

Controlling the clock

(define (make-clock-callback name object msg . data)
  (lambda (message)
    (case message
      ((CLOCK-CALLBACK?) (lambda (self) #t))
      ((NAME) (lambda (self) name))
      ((OBJECT) (lambda (self) object))
      ((MESSAGE) (lambda (self) msg))
      ((ACTIVATE) (lambda (self)
                   (apply-method object object msg data))))

Implementations for our Extended World

(define (make-ship position velocity num-torps)
  (define (move) (set! position (add-vect position ...)))
  (define (fire-torp) (cond ((> num-torps 0) (set! num-torps (- num-torps 1))
                               (let ((torp (make-torpedo ...))
                                     (add-to-universe torp)))
                               (define (me msg . args)
                                 (cond ((eq? msg 'SHIP?) #T)
                                       ((eq? msg 'ATTACK) (fire-torp))
                                       ((eq? msg 'EXPLODE) (explode (car args)))
                                       ((eq? msg 'POSITION) position)
                                       ((eq? msg 'VELOCITY) velocity)
                                       ((eq? msg 'MOVE) (move))
                                       ((eq? msg 'EXPLODE) (explode (car args)))
                                       ((eq? msg 'DISPLAY) (draw ...))
                                       (else (error "No method" msg)))))

Torpedo Implementation

(define (make-torpedo position velocity)
  (define (explode torp)
    (display "torpedo goes off!"
    (remove-from-universe torp))
  (define (move) (set! position ...))
  (ask clock 'ADD-CALLBACK
    (make-clock-callback 'moveit me 'MOVE))
  (define (me msg . args)
    (cond ((eq? msg 'TORPEDO?) #T)
          ((eq? msg 'POSITION) position)
          ((eq? msg 'VELOCITY) velocity)
          ((eq? msg 'MOVE) (move))
          ((eq? msg 'EXPLODE) (explode (car args)))
          ((eq? msg 'DISPLAY) (draw ...))
          (else (error "No method" msg))))

Running the Simulation

(define earth (make-planet (make-vect 0 0)))
(define enterprise (make-ship (make-vect 10 10) (make-vect 5 0) 3))
(define war-bird (make-ship (make-vect -10 10) (make-vect 10 0) 10))
(run-clock 100)
Summary

• Introduced a new programming style:
  • Object-oriented vs. Procedural
  • Uses – simulations, complex systems, ...

• Object-Oriented Modeling
  • Language independent!
    Class – template for state and behavior
    Instances – specific objects with their own identities

• Next time: powerful ideas of inheritance and delegation