

## 6.001 SICP Object Oriented Programming

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

1

## 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?

3

## 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 set of procedures that operate on the data

• "Generic" operations by looking at types:

```
(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"))))
```

4

## Generic Operations

	Point	Line	2-dShape	3-dShape
<b>scale</b>	point-scale	line-scale	2dshape-scale	3dshape-scale
<b>translate</b>	point-trans	line-trans	2dshape-trans	3dshape-trans
<b>color</b>	point-color	line-color	2dshape-color	3dshape-color

7

## Generic Operations

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

	Point	Line	2-dShape	3-dShape	curve
<b>scale</b>	point-scale	line-scale	2dshape-scale	3dshape-scale	scale-c
<b>translate</b>	point-trans	line-trans	2dshape-trans	3dshape-trans	trans-c
<b>color</b>	point-color	line-color	2dshape-color	3dshape-color	color-c
<b>new-op</b>	...	...	...	...	...

10

## Views of The World

	Point	Line	2-dShape	3-dShape	curve
<b>scale</b>	point-scale	line-scale	2dshape-scale	3dshape-scale	scale-c
<b>color</b>	point-color	line-color	2dshape-color	3dshape-color	color-c
<b>new-op</b>	...	...	...	...	...

Diagram: A blue oval labeled "Data object" encircles the "Line" column. A red oval labeled "Generic operation" encircles the "color" row. A red line connects the "color" row to the "color-c" cell.


11

## 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

12

## Data Objects: Simple (?) Example

- What's a cons cell (pair)?
  - 
- What about a pair as a data object?

	<b>PAIR</b>
data	car-contents cdr-contents
operations	CAR CDR PAIR? SET-CAR! SET-CDR!

13

## 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

15

## Scheme OOP: Procedures with State

- A procedure has
  - **parameters** and **body** as specified by  $\lambda$  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

16

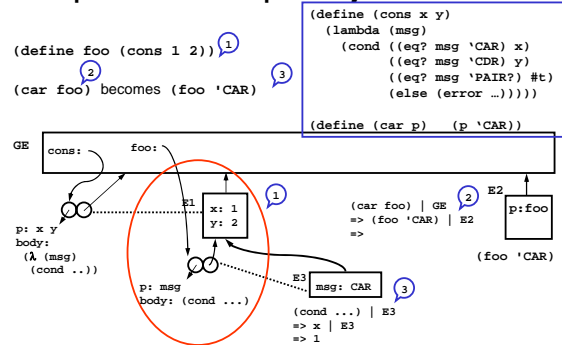
## Example: Pair as a Procedure with State

```
(define (cons x y)
  (lambda (msg)
    (cond ((eq? msg 'CAR) x)
          ((eq? msg 'CDR) y)
          ((eq? msg 'PAIR?) #t)
          (else (error "pair cannot" msg)))))

(define (car p) (p 'CAR))
(define (cdr p) (p 'CDR))
(define (pair? p)
  (and (procedure? p) (p 'PAIR?)))
```

17

## Example: What is our "pair" object?



18

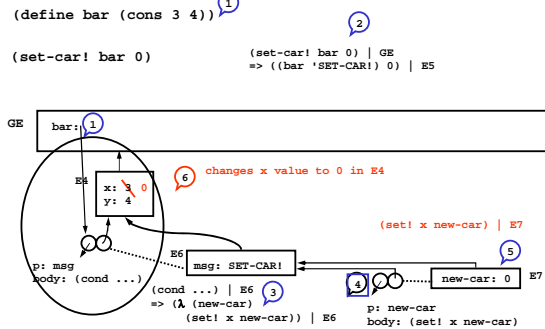
## Pair Mutation as Change in State

```
(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))
```

19

## Example: Mutating a pair object



20

## Message Passing Style - Refinements

- lexical scoping for **private state** and **private procedures**

```
(define (cons x y)
  (define (change-car new-car) (set! x new-car))
  (define (change-cdr new-cdr) (set! y new-cdr))
  (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))
```

21

## 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)
```

22

## Message Passing Style - Refinements

- lexical scoping for **private state** and **private procedures**

```
(define (cons x y)
  (define (change-car new-car) (set! x new-car))
  (define (change-cdr new-cdr) (set! y new-cdr))
  (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))
```

23

## 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

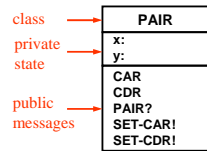
24

## 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

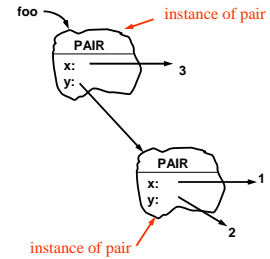
25

## Class Diagram



```
(define (cons x y)
  (lambda (msg) ...))
```

## Instance Diagram



```
(define foo
  (cons 3 (cons 1 2)))
```

26

## 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

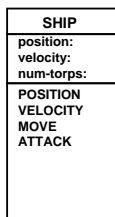
27

## A Space-Ship Object

```
(define (make-ship position velocity num-torps)
  (define (move)
    (set! position (add-vect position ...)))
  (define (fire-torp)
    (cond ((> num-torps 0) ...)
          (else 'FAIL)))
  (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)))))
```

28

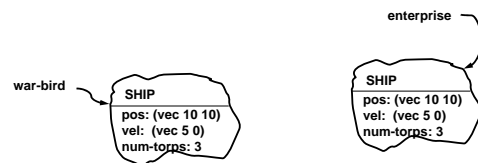
## Space-Ship Class



29

## Example – Instance Diagram

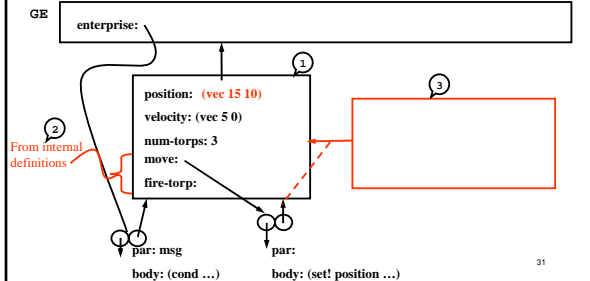
```
(define enterprise
  (make-ship (make-vect 10 10) (make-vect 5 0) 3))
(define war-bird
  (make-ship (make-vect -10 10) (make-vect 10 0) 10))
```



30

### Example – Environment Diagram

```
(define enterprise
  (make-ship (make-vec 10 10) (make-vec 5 0) 3))
(enterprise 'MOVE) ==> DONE
(enterprise 'POSITION) ==> (vec 15 10)
```



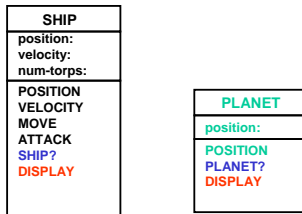
31

### 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)

32

### Space-Ship Class



33

### Planet Implementation

```
(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)))))
```

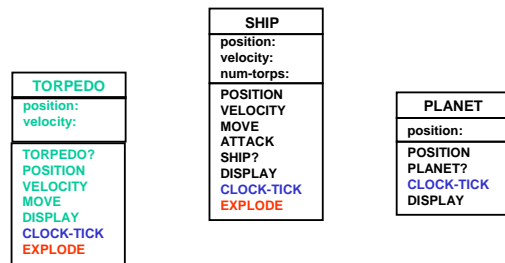
34

### 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

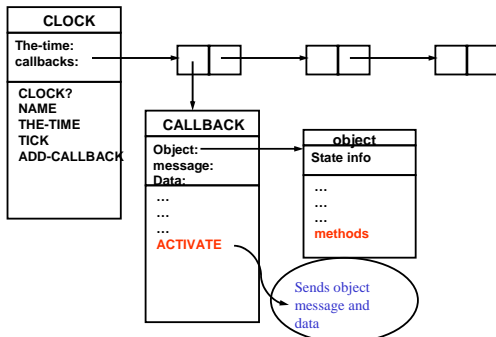
35

### Class Diagram



36

## Coordinating with a clock



37

## 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)))
    )))
  
```

40

## Controlling the clock

```

;; Clock callbacks
;;
;; A callback is an object that stores a target object,
;; message, and arguments. When activated, it sends the target
;; object the message. It can be thought of as a button that
;; executes an action at every tick of 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))))
    )))
  
```

41

## 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 (explode ship)
    (display "Ouch. That hurt.") (remove-from-universe ship))
  (ask clock 'ADD-CALLBACK
        (make-clock-callback 'moveit me 'MOVE))
  (define (me msg . args)
    (cond ((eq? msg 'SHIP?) #T)
          ...
          ((eq? msg 'ATTACK) (fire-torp))
          ((eq? msg 'EXPLODE) (explode (car args)))
          (else (error "ship can't" msg))))
  ME)
  
```

42

## 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))))
  ME)
  
```

43

## Running the Simulation

```

;; Build some things
(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))

;; Start simulation
(run-clock 100)
  
```

44

## 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*

45