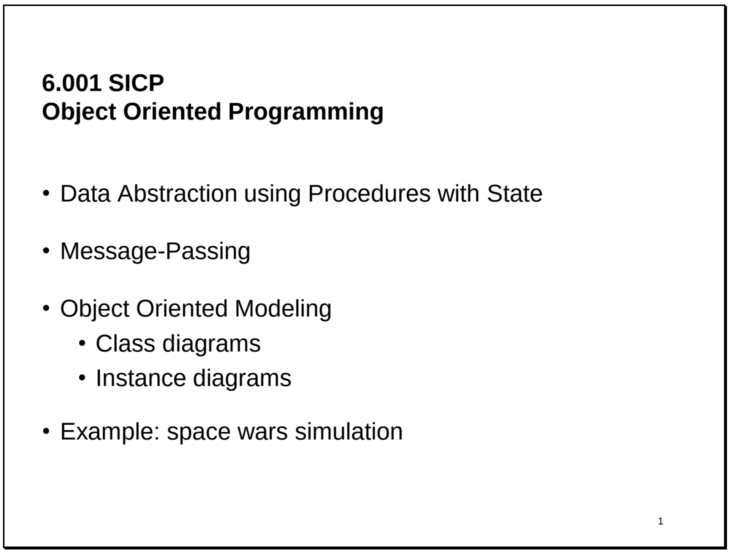
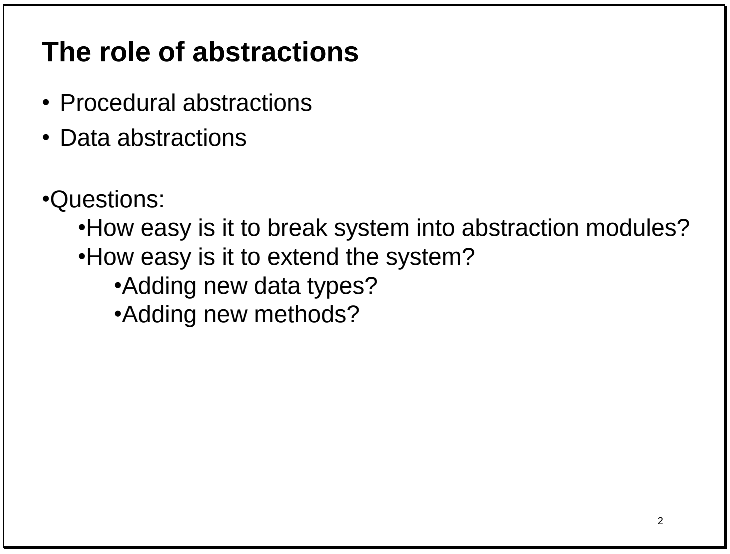
Slide 1



6.001 SICP
Object Oriented Programming

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

Slide 2

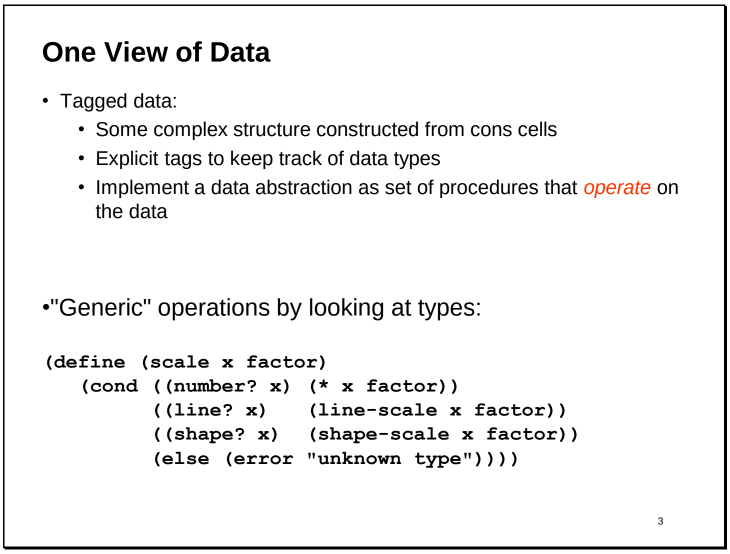


The role of abstractions

- Procedural abstractions
- Data abstractions

•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?

Slide 3



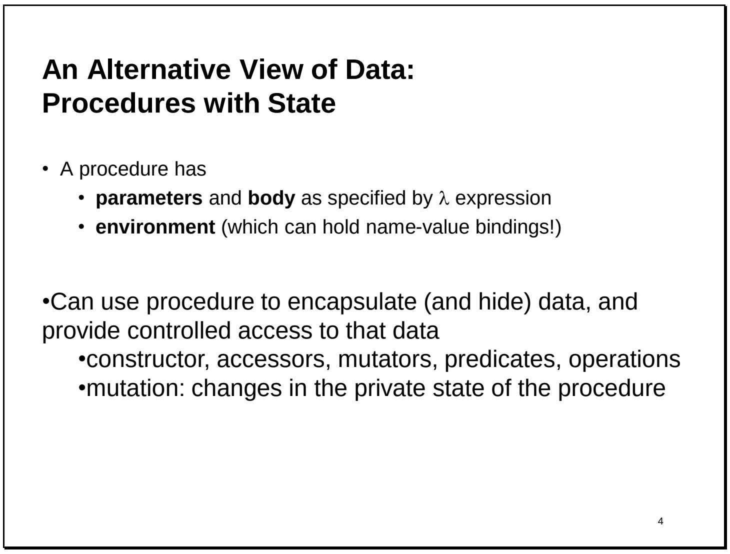
One View of Data

- Tagged data:
  - Some complex structure constructed from cons cells
  - Explicit tags to keep track of data types
  - Implement a data abstraction as set of procedures that *operate* on the data

•"Generic" operations by looking at types:

```
(define (scale x factor)
   (cond ((number? x)  (* x factor))
         ((line? x)    (line-scale x factor))
         ((shape? x)   (shape-scale x factor))
         (else (error "unknown type"))))
```

Slide 4



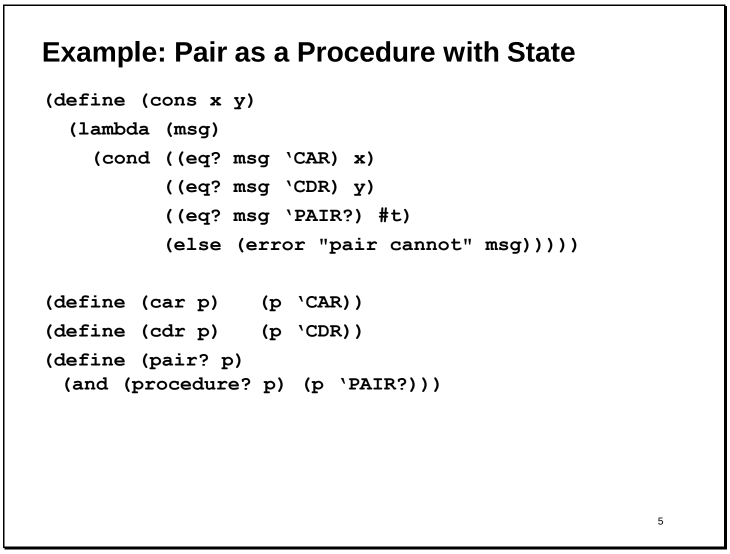
An Alternative View of Data: Procedures with State

- 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

- •constructor, accessors, mutators, predicates, operations
- •mutation: changes in the private state of the procedure

Slide 5

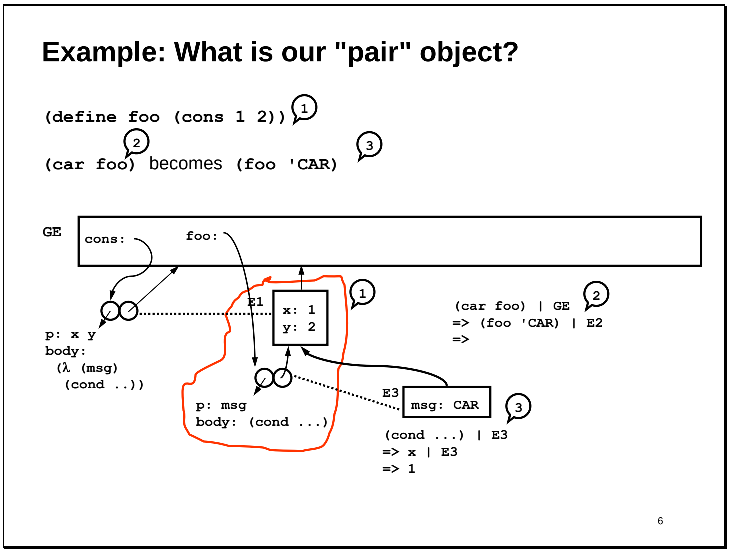


Example: Pair as a Procedure with State

```
(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?)))
```

Slide 6



Example: What is our "pair" object?

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

`(car foo)` becomes `(foo 'CAR)`

GE

cons:

foo:

E1

x: 1
y: 2

p: x y
body:
  (λ (msg)
   (cond ..))

p: msg
body: (cond ...)

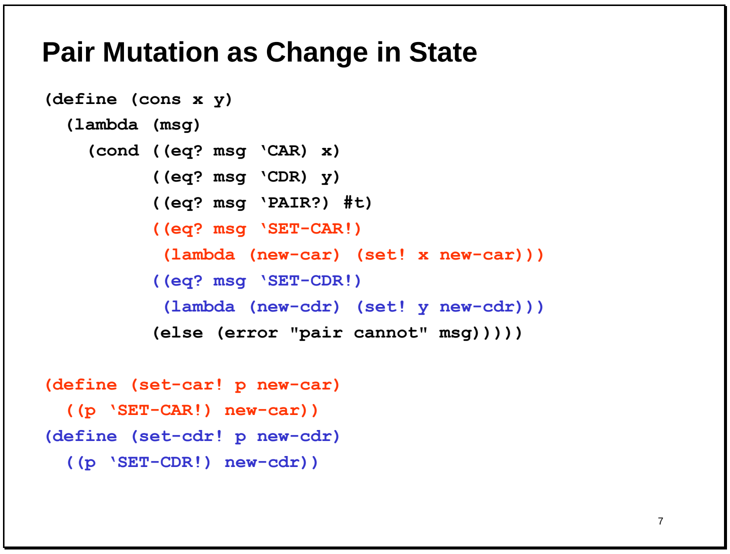
```
(car foo) | GE
=> (foo 'CAR) | E2
=>
```

E3

msg: CAR

```
(cond ...) | E3
=> x | E3
=> 1
```

Slide 7



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))
```

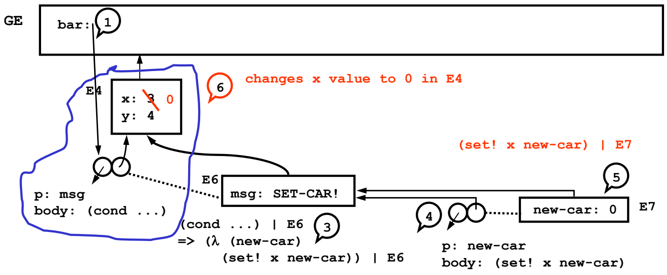
Slide 8

Example: Mutating a pair object

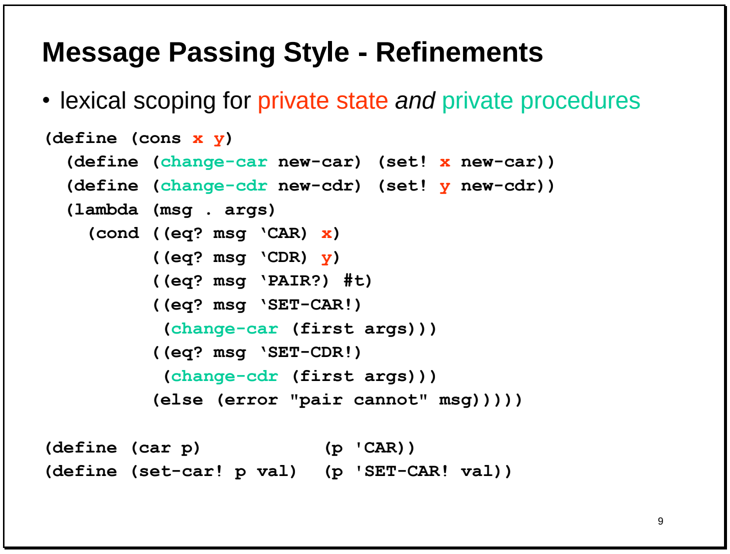
```
(define bar (cons 3 4))
```

```
(set-car! bar 0)
```

```
(set-car! bar 0) | GE
=> ((bar 'SET-CAR!) 0) | E5
```



Slide 9



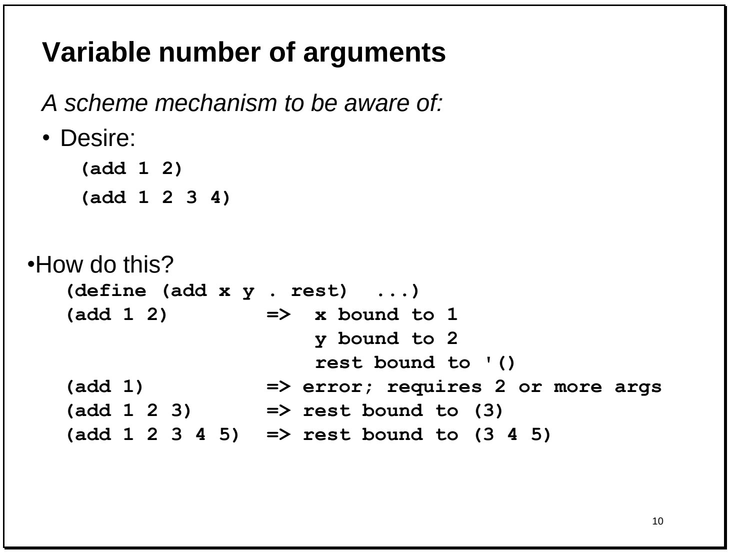
Message Passing Style - Refinements

- 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))
```

Slide 10



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)
```

____

____

____

____

____

____

____

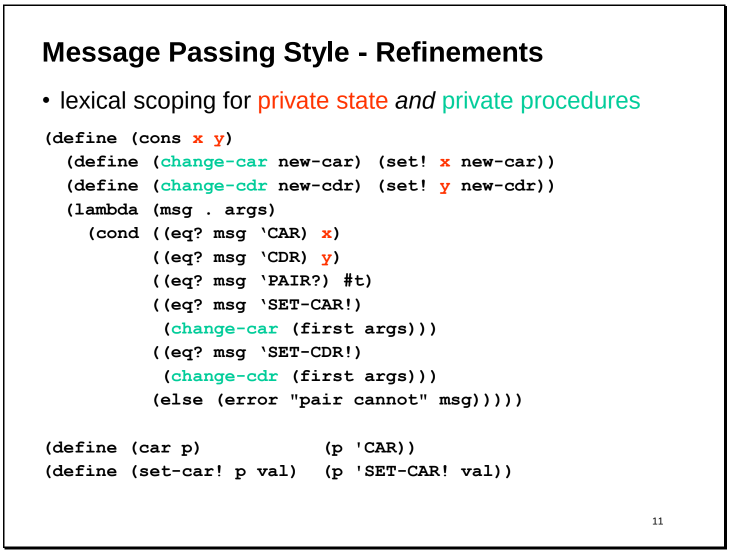
____

____

____

____

Slide 11



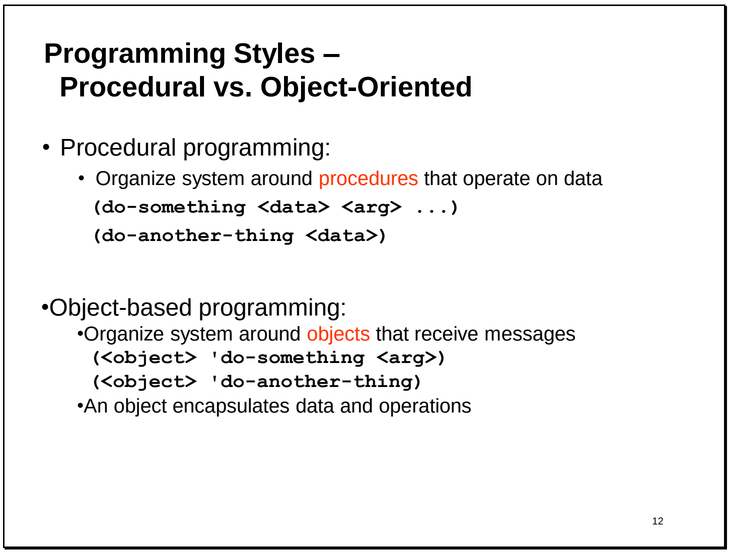
Message Passing Style - Refinements

- 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))
```

Slide 12



Programming Styles –
Procedural vs. Object-Oriented

- Procedural programming:
  - Organize system around procedures that operate on data

```
(do-something <data> <arg> ...)
(do-another-thing <data>)
```

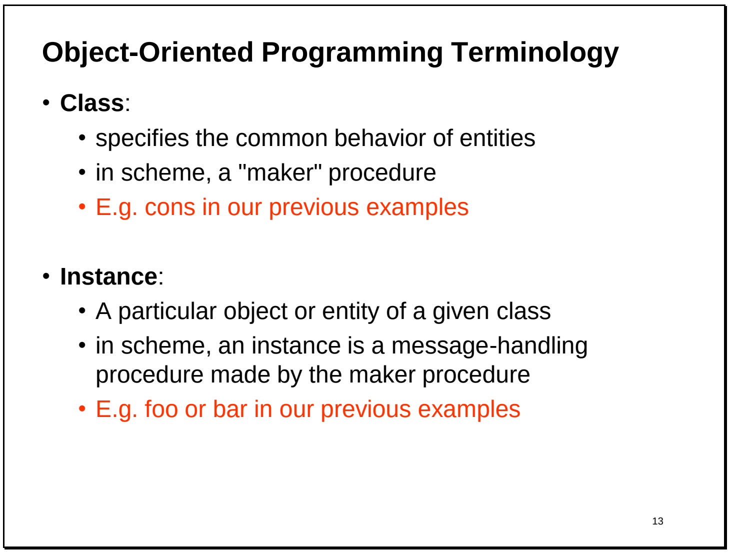
•Object-based programming:

•Organize system around objects that receive messages

```
(<object> 'do-something <arg>)
(<object> 'do-another-thing)
```

•An object encapsulates data and operations

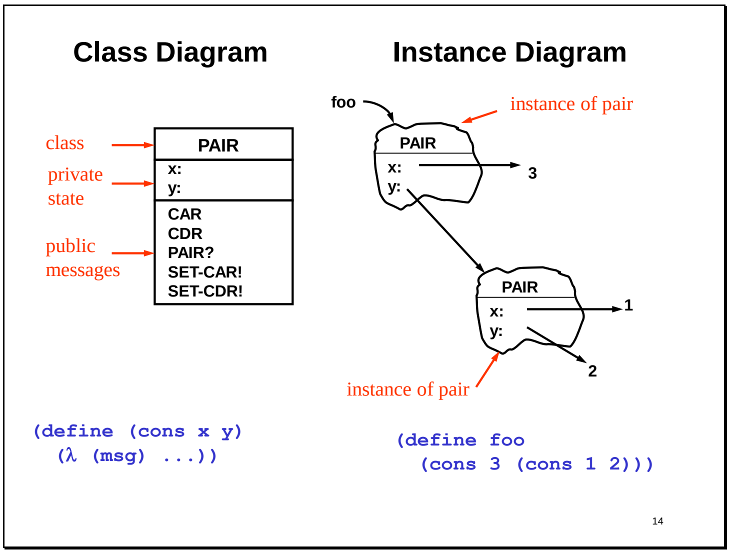
Slide 13



Object-Oriented Programming Terminology

- **Class**:
  - specifies the common behavior of entities
  - in scheme, a "maker" procedure
  - E.g. cons in our previous examples
- **Instance**:
  - A particular object or entity of a given class
  - in scheme, an instance is a message-handling procedure made by the maker procedure
  - E.g. foo or bar in our previous examples

Slide 14



Class Diagram

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

Instance Diagram

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

____

____

____

____

____

____

____

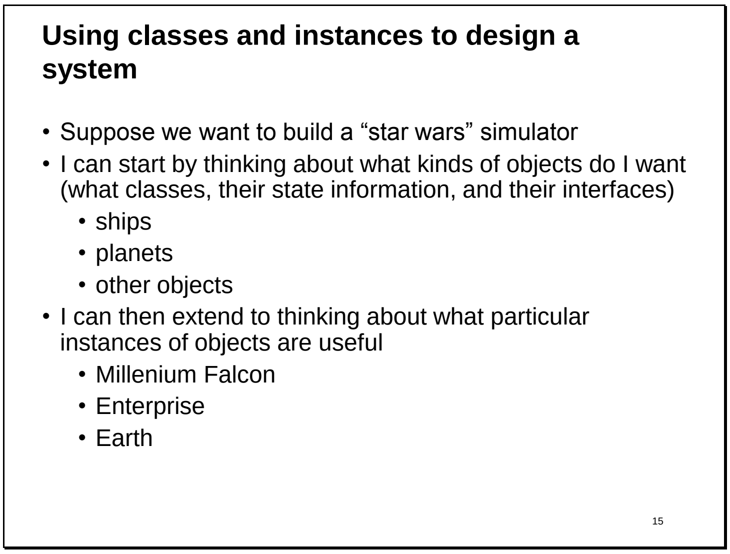
____

____

____

____

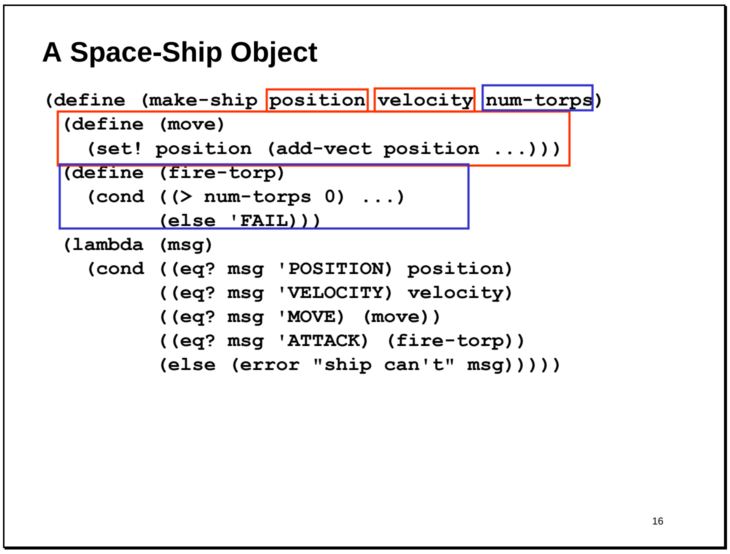
Slide 15



Using classes and instances to design a system

- Suppose we want to build a “star wars” simulator
- 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

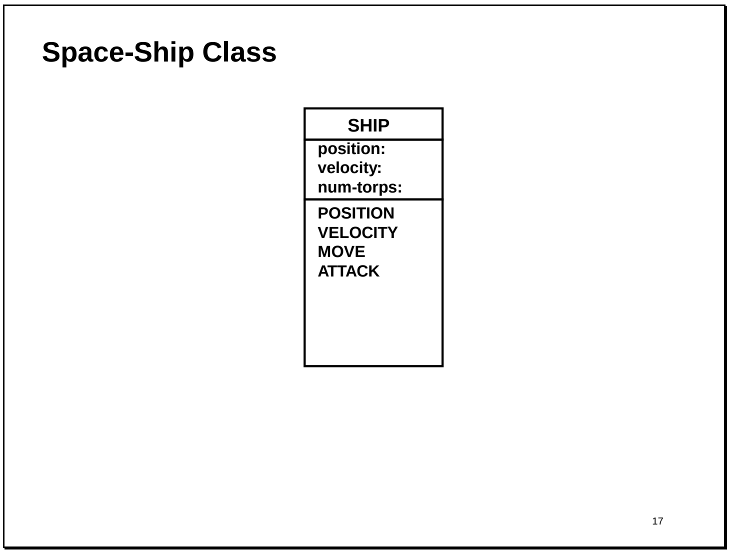
Slide 16



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)))))
```

Slide 17



Space-Ship Class

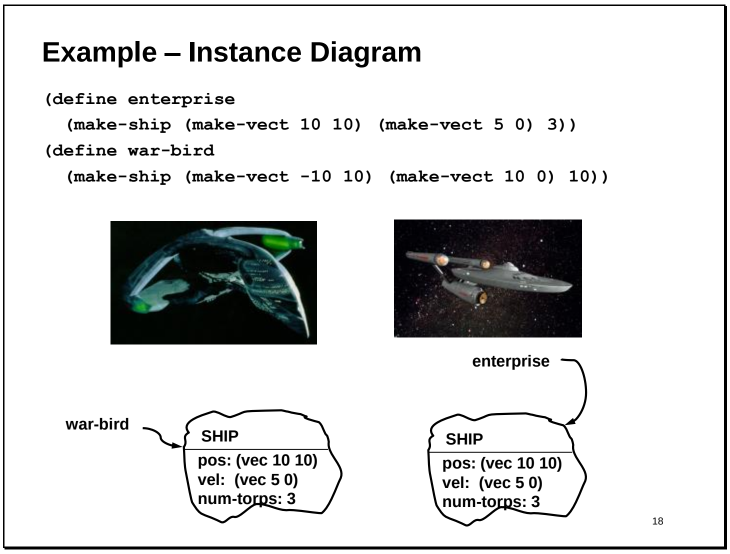
SHIP

position:
velocity:
num-torps:

POSITION
VELOCITY
MOVE
ATTACK

17

Slide 18



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))
```

enterprise

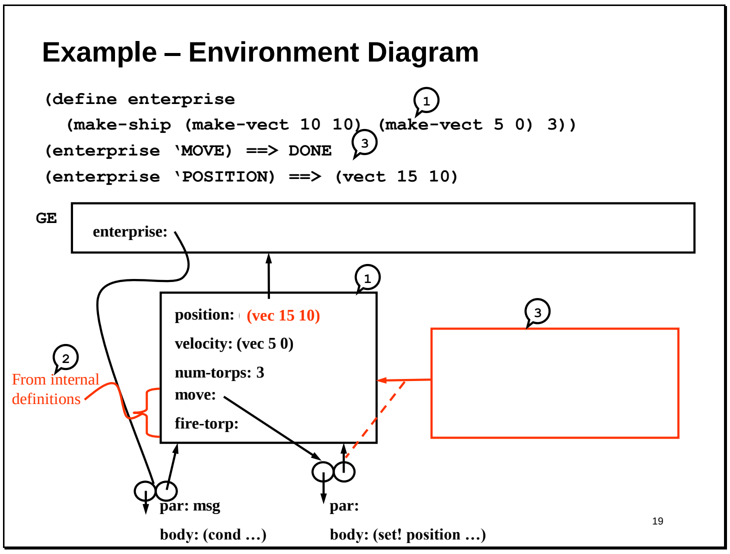
war-bird

SHIP
pos: (vec 10 10)
vel: (vec 5 0)
num-torps: 3

SHIP
pos: (vec 10 10)
vel: (vec 5 0)
num-torps: 3

18

Slide 19



Example – Environment Diagram

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

GE

enterprise:

position: (vec 15 10)

velocity: (vec 5 0)

num-torps: 3

move:

fire-torp:

From internal definitions

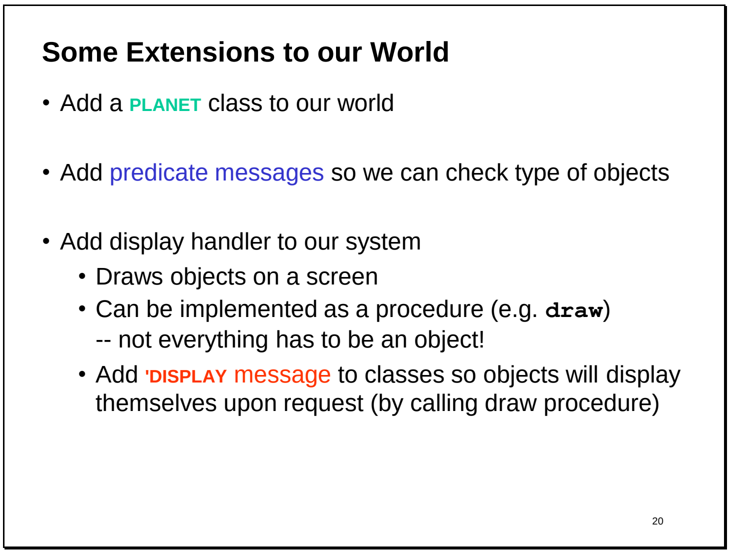
par: msg

body: (cond …)

par:

body: (set! position …)

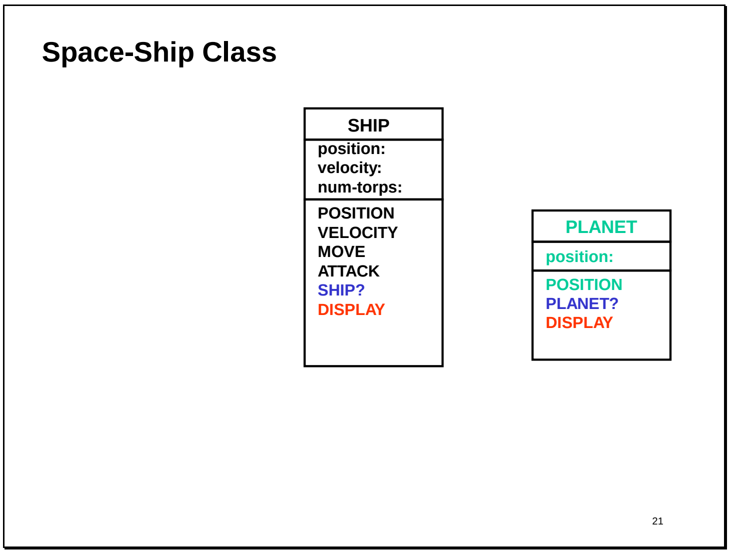
Slide 20



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)

Slide 21



Space-Ship Class

SHIP

position:
velocity:
num-torps:

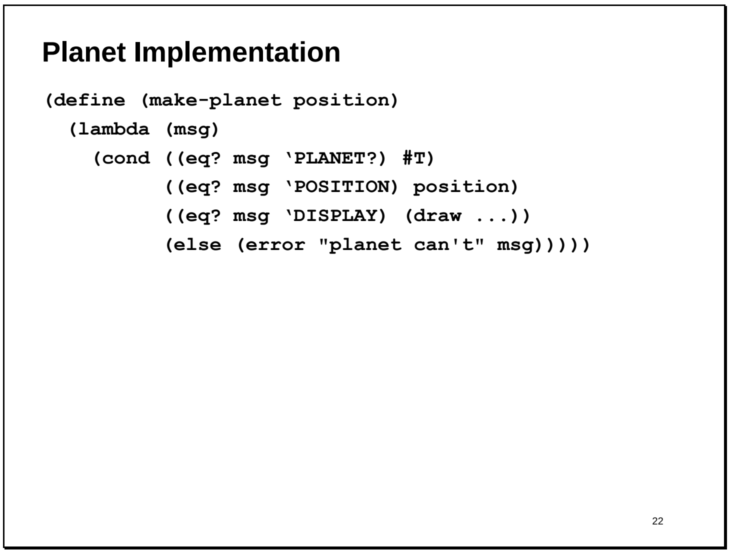
POSITION
VELOCITY
MOVE
ATTACK
SHIP?
DISPLAY

PLANET

position:

POSITION
PLANET?
DISPLAY

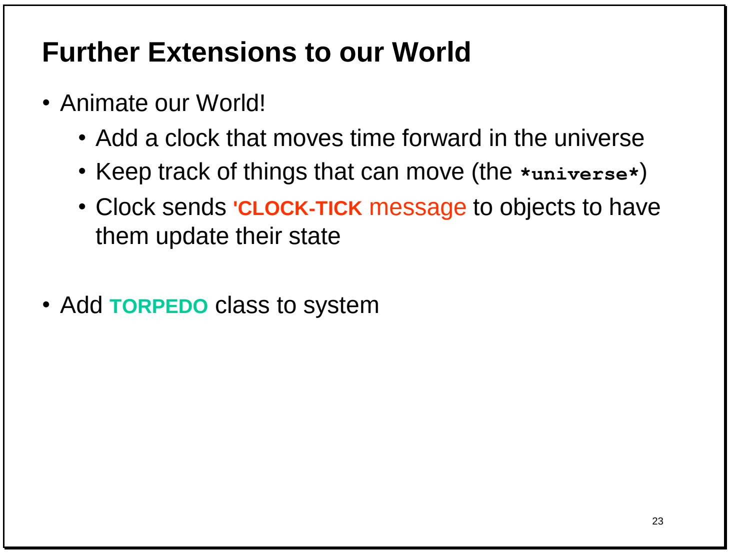
Slide 22



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)))))
```

Slide 23

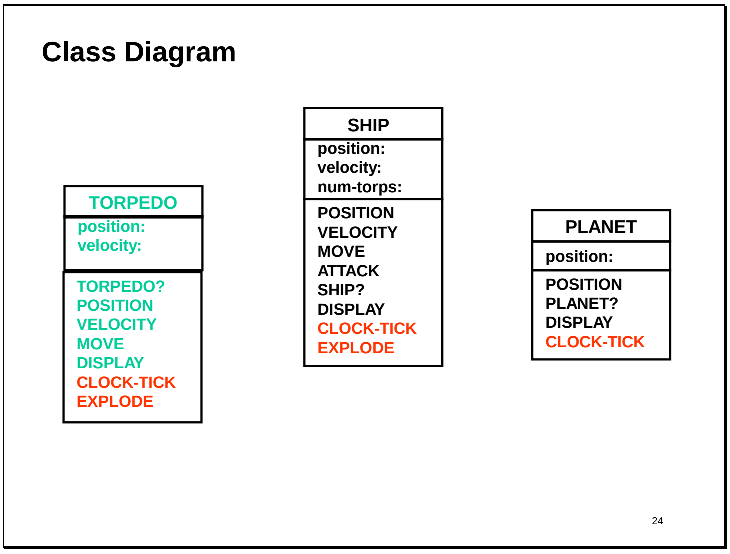


Further Extensions to our World

- Animate our World!
  - Add a clock that moves time forward in the universe
  - Keep track of things that can move (the `*universe*`)
  - Clock sends **'CLOCK-TICK** message to objects to have them update their state

- Add **TORPEDO** class to system

Slide 24



Class Diagram

SHIP

position:
velocity:
num-torps:

POSITION
VELOCITY
MOVE
ATTACK
SHIP?
DISPLAY
CLOCK-TICK
EXPLODE

TORPEDO

position:
velocity:

TORPEDO?
POSITION
VELOCITY
MOVE
DISPLAY
CLOCK-TICK
EXPLODE

PLANET

position:

POSITION
PLANET?
DISPLAY
CLOCK-TICK

____

____

____

____

____

____

____

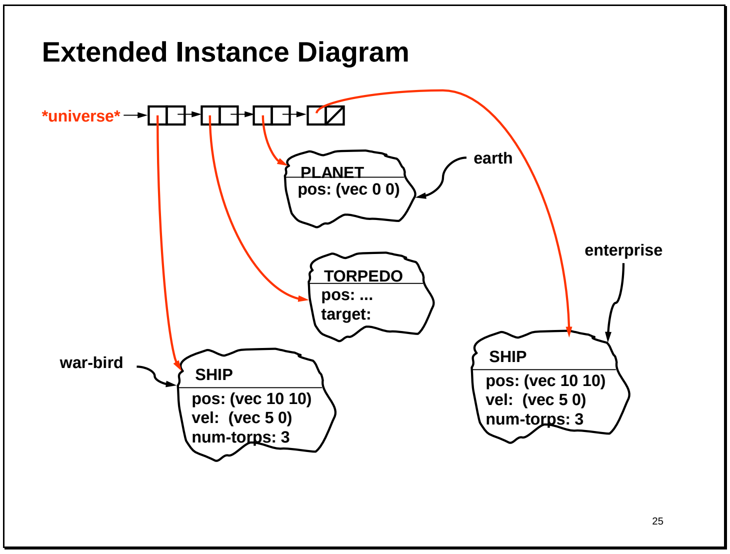
____

____

____

____

Slide 25



Extended Instance Diagram

universe

PLANET
pos: (vec 0 0)

earth

TORPEDO
pos: ...
target:

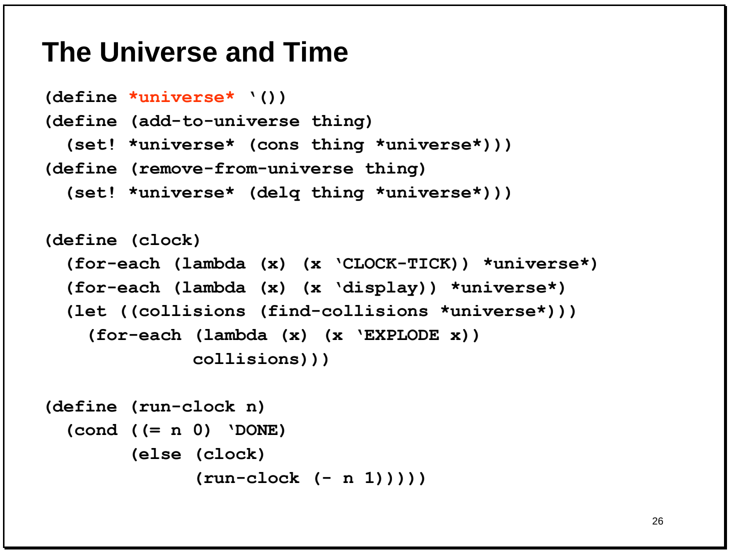
enterprise

war-bird

SHIP
pos: (vec 10 10)
vel: (vec 5 0)
num-torps: 3

SHIP
pos: (vec 10 10)
vel: (vec 5 0)
num-torps: 3

Slide 26



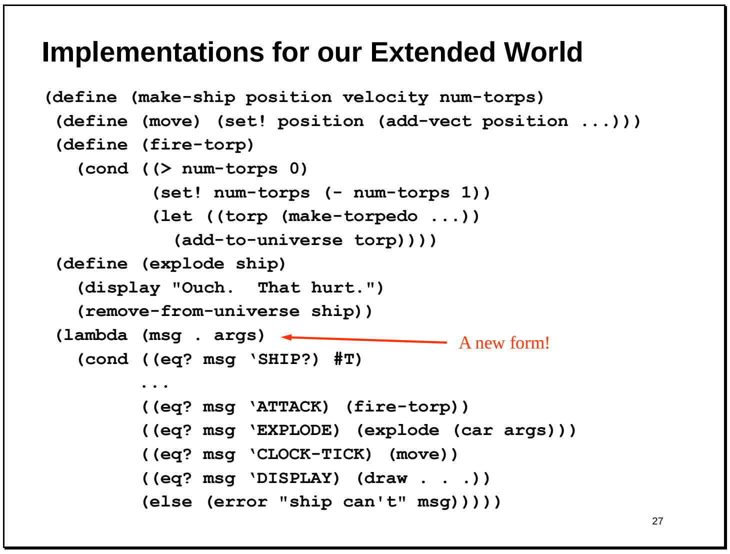
The Universe and Time

```
(define *universe* `())
(define (add-to-universe thing)
  (set! *universe* (cons thing *universe*)))
(define (remove-from-universe thing)
  (set! *universe* (delq thing *universe*)))

(define (clock)
  (for-each (lambda (x) (x `CLOCK-TICK)) *universe*)
  (for-each (lambda (x) (x `display)) *universe*)
  (let ((collisions (find-collisions *universe*)))
    (for-each (lambda (x) (x `EXPLODE x))
              collisions)))

(define (run-clock n)
  (cond ((= n 0) `DONE)
        (else (clock)
              (run-clock (- n 1)))))
```

Slide 27

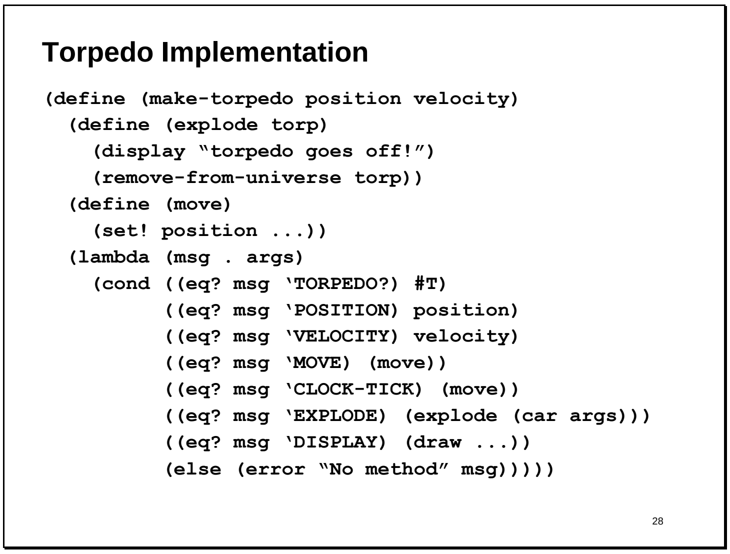


Implementations for our Extended World

```
(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))
 (lambda (msg . args)
   (cond ((eq? msg ‘SHIP?) #T)
         ...
         ((eq? msg ‘ATTACK) (fire-torp))
         ((eq? msg ‘EXPLODE) (explode (car args)))
         ((eq? msg ‘CLOCK-TICK) (move))
         ((eq? msg ‘DISPLAY) (draw . . .))
         (else (error "ship can't" msg)))))
```

A new form!

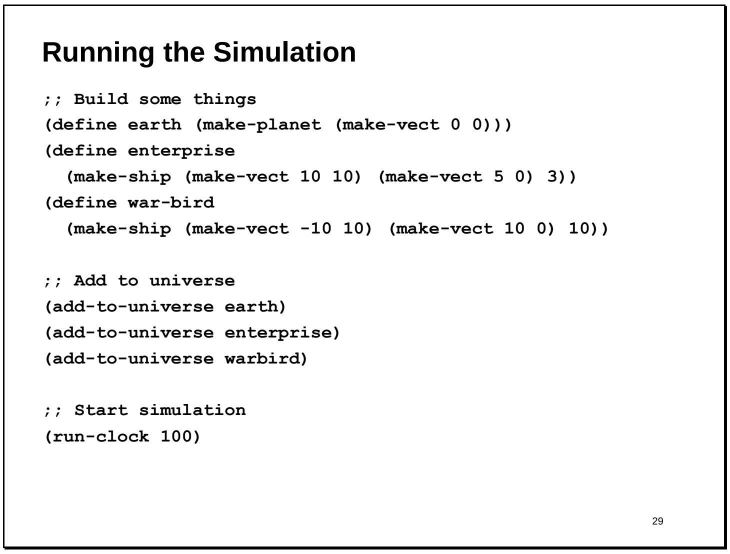
Slide 28



Torpedo Implementation

```
(define (make-torpedo position velocity)
  (define (explode torp)
    (display “torpedo goes off!”)
    (remove-from-universe torp))
  (define (move)
    (set! position ...))
  (lambda (msg . args)
    (cond ((eq? msg ‘TORPEDO?) #T)
          ((eq? msg ‘POSITION) position)
          ((eq? msg ‘VELOCITY) velocity)
          ((eq? msg ‘MOVE) (move))
          ((eq? msg ‘CLOCK-TICK) (move))
          ((eq? msg ‘EXPLODE) (explode (car args)))
          ((eq? msg ‘DISPLAY) (draw ...))
          (else (error “No method” msg)))))
```

Slide 29



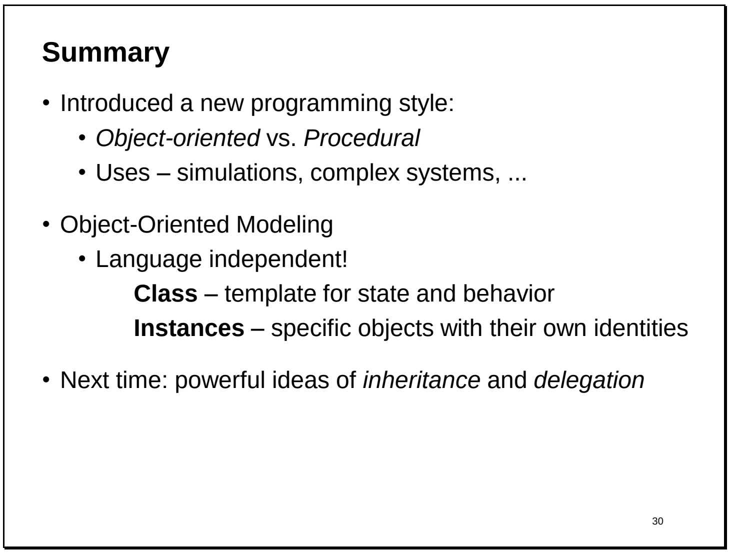
Running the Simulation

```
;; 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))

;; Add to universe
(add-to-universe earth)
(add-to-universe enterprise)
(add-to-universe warbird)

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

Slide 30



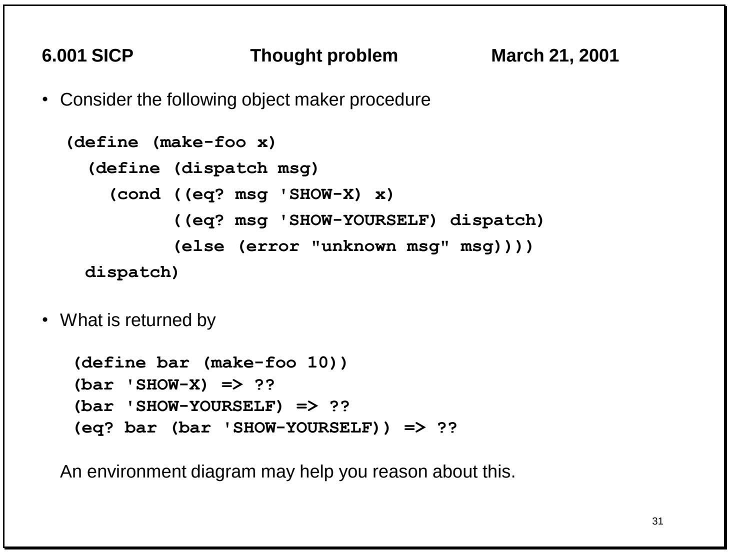
Summary

- 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*

Slide 31



6.001 SICP **Thought problem** **March 21, 2001**

- Consider the following object maker procedure

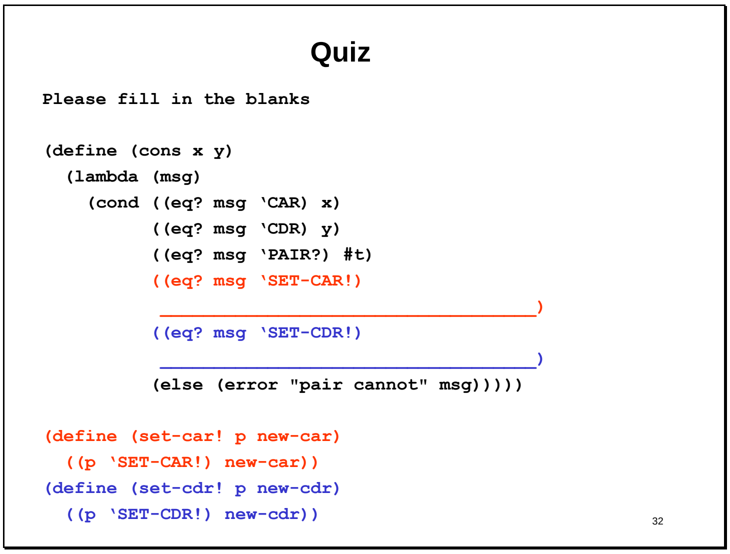
```
(define (make-foo x)
  (define (dispatch msg)
    (cond ((eq? msg 'SHOW-X) x)
          ((eq? msg 'SHOW-YOURSELF) dispatch)
          (else (error "unknown msg" msg))))
 dispatch)
```

- What is returned by

```
(define bar (make-foo 10))
(bar 'SHOW-X) => ??
(bar 'SHOW-YOURSELF) => ??
(eq? bar (bar 'SHOW-YOURSELF)) => ??
```

An environment diagram may help you reason about this.

Slide 32



Quiz

Please fill in the blanks

```
(define (cons x y)
  (lambda (msg)
    (cond ((eq? msg 'CAR) x)
          ((eq? msg 'CDR) y)
          ((eq? msg 'PAIR?) #t)
          ((eq? msg 'SET-CAR!)
           ___________________________________)
          ((eq? msg 'SET-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))
```