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

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?

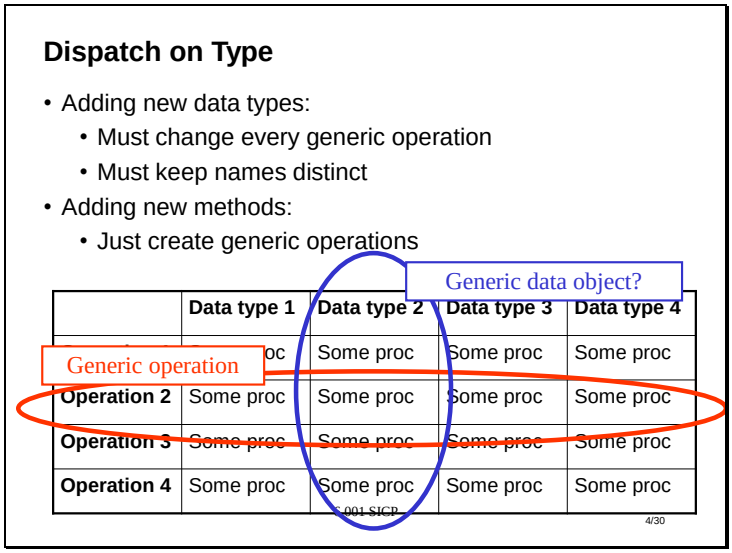
One View of Data

- 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


Dispatch on Type
• Adding new data types:
• Must change every generic operation
• Must keep names distinct
• Adding new methods:
• Just create generic operations
Generic data object?
Generic operation
Data type 1
Data type 2
Data type 3
Data type 4
Some proc
Operation 2
Operation 3
Operation 4
4/30

Slide 5

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
  - •Procedure application creates private environment
  - •Need access to that environment
    - •constructor, accessors, mutators, predicates, operations
    - •mutation: changes in the private state of the procedure

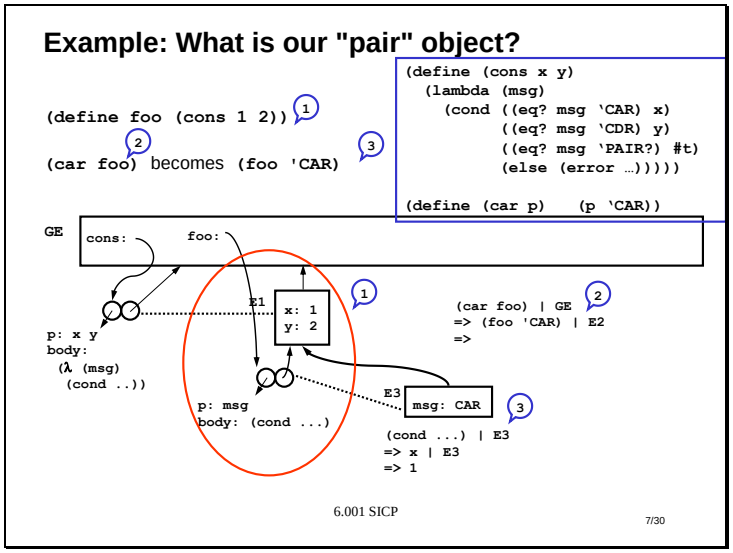
Slide 6

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 7


Example: What is our "pair" object?
(define foo (cons 1 2))
(car foo) becomes (foo 'CAR)
(define (cons x y)
(lambda (msg)
(cond ((eq? msg 'CAR) x)
((eq? msg 'CDR) y)
((eq? msg 'PAIR?) #t)
(else (error …)))))
(define (car p) (p 'CAR))
GE
cons:
foo:
E1
x: 1
y: 2
p: x y
body:
(λ (msg)
(cond ..))
p: msg
body: (cond ...)
E3
msg: CAR
(car foo) | GE
=> (foo 'CAR) | E2
=>
(cond ...) | E3
=> x | E3
=> 1
6.001 SICP
7/30

Slide 8

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))
```

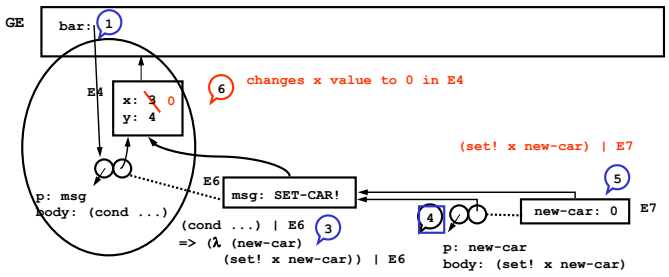
Slide 9

Example: Mutating a pair object

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

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

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



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))
```

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)
  (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))
```

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