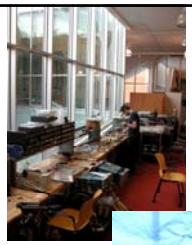
EECS is everywhere!

- EECS Freshman Open House
  - If you are thinking about majoring in Course 6 (or even curious), please come to the EECS open house, Friday, 3:30 – 5:00 in 34-401
  - Talk with faculty and students about department, about degree programs, about career opportunities
  - Hear about the new curriculum
  - Get free "swag"



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 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"))))
```


Generic Operations

- Adding new methods
  - Just create generic operations

	Point	Line	2-dShape	3-dShape
scale	point-scale	line-scale	2dshape-scale	3dshape-scale
translate	point-trans	line-trans	2dshape-trans	3dshape-trans
color	point-color	line-color	2dshape-color	3dshape-color

Generic Operations

- Adding new methods
  - Just create generic operations

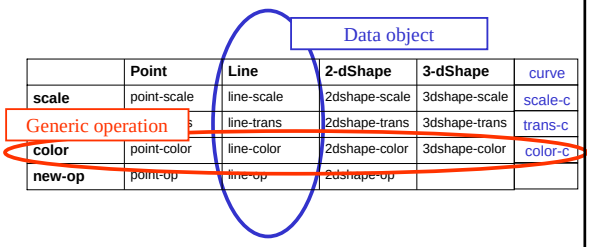
	Point	Line	2-dShape	3-dShape
scale	point-scale	line-scale	2dshape-scale	3dshape-scale
translate	point-trans	line-trans	2dshape-trans	3dshape-trans
color	point-color	line-color	2dshape-color	3dshape-color
new-op	point-op	line-op	2dshape-op	

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
scale	point-scale	line-scale	2dshape-scale	3dshape-scale	scale-c
translate	point-trans	line-trans	2dshape-trans	3dshape-trans	trans-c
color	point-color	line-color	2dshape-color	3dshape-color	color-c
new-op	point-op	line-op	2dshape-op	…	

Views of The World



	Point	Line	2-dShape	3-dShape	curve
scale	point-scale	line-scale	2dshape-scale	3dshape-scale	scale-c
		line-trans	2dshape-trans	3dshape-trans	trans-c
color	point-color	line-color	2dshape-color	3dshape-color	color-c
new-op	point-op	line-op	2dshape-op		

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

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

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 (i.e. specific procedures that apply to that object, and handle local state associated with that object)

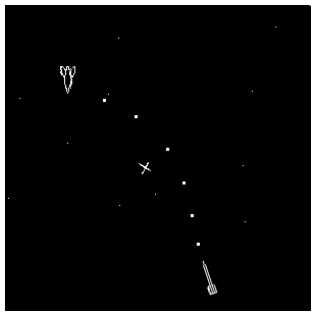
Object-Oriented Programming Terminology

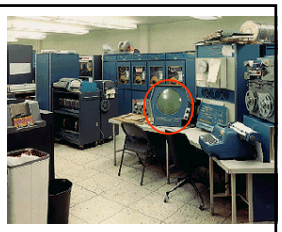
- **Class**:
  - specifies the common behavior of entities
  - in scheme, a <type> procedure
- **Instance**:
  - a particular object or entity of a given class
  - in scheme, an instance is a message-handling procedure made by a create-<type> procedure

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

16





SPACEWAR: the *original* video game
first realized on the MIT PDP-1 in 1962
PDP-1 – 100KHz, 4K RAM, $100,000

17

A Space-Ship Object

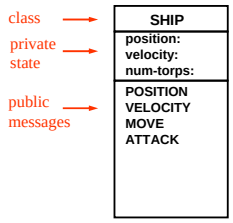
```
(define (ship position velocity num-torps)
  (define (move)
    (set! position (add-vect position ...)))
  (define (fire-torp)
    (cond ((> num-torps 0) ...)
          (else 'FAIL)))
  (lambda (msg)
    (cond ((eq? msg 'POSITION) position)
          ((eq? msg 'VELOCITY) velocity)
          ((eq? msg 'MOVE) (move))
          ((eq? msg 'ATTACK) (fire-torp))
          (else (error "ship can't" msg)))))
```

Note the internal state (passed in as parameters in this case), and the object-specific procedures.

Note value returned is procedure with access to internal state

18

Space-Ship Class



19

Creating instances of a class

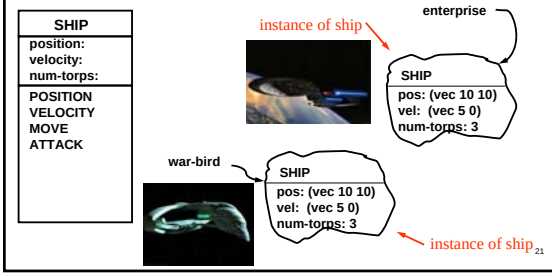
- The definition of ship specifies the properties of a class
  - Every instance of a ship will have its own version of position, velocity, etc.; and will have its own procedures for accessing that state
- Need a mechanism for creating specific instances of this class
- For now, we will use a simple instantiation – this will get extended in the next lecture

```
(define (create-ship pos vel torp)
  (create-instance ship pos vel torp))

(define (create-instance type . args)
  (apply type args)) ;; for ship, think of as evaluating
                     ;; (ship pos vel trop)
```

20

Example – Instance Diagram

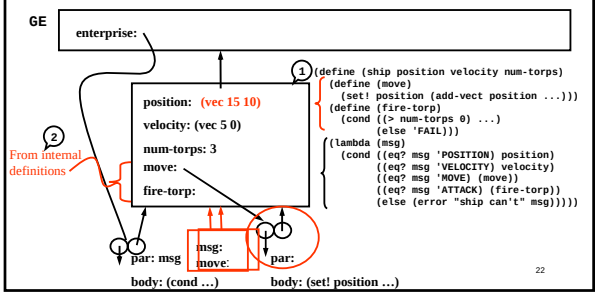
```
(define enterprise
  (create-ship (make-vector 10 10) (make-vector 5 0) 3))
(define war-bird
  (create-ship (make-vector -10 10) (make-vector 10 0) 10))
```



21

Example – Environment Diagram

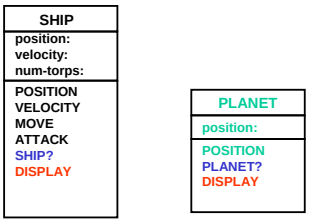
```
(define enterprise
  (ship (make-vector 10 10) (make-vector 5 0) 3)) ; skipping step
(enterprise 'MOVE) ==> DONE
(enterprise 'POSITION) ==>  ?      (vec 15 10)
```



Filling out our World
-- how do we think about programming in this space"?

- 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



Planet Implementation

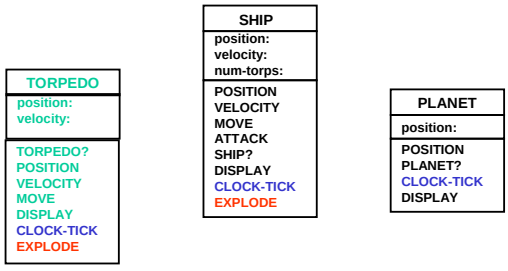
```
(define (planet position)
  (lambda (msg)
    (cond ((eq? msg 'PLANET?) #T)
          ((eq? msg 'POSITION) position)
          ((eq? msg 'DISPLAY) (draw ...))
          (else (error "planet can't" msg)))))
```

This is like our tags in data structures

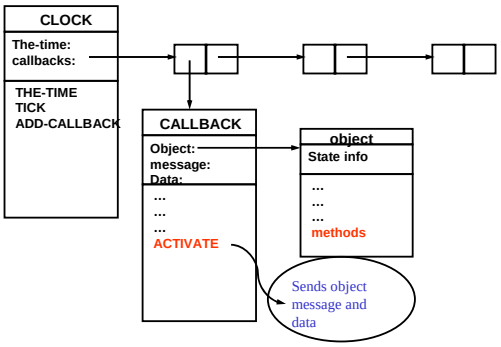
Keeping time…

- 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

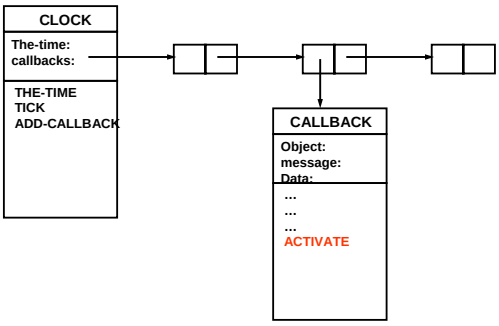


Coordinating with a clock



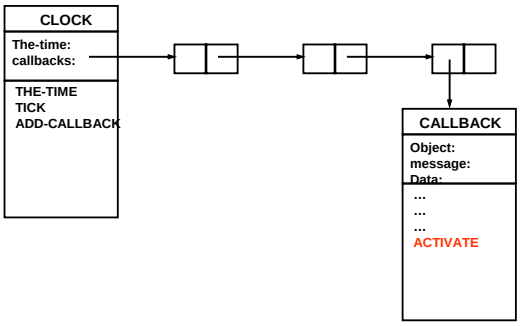
28

Coordinating with a clock



29

Coordinating with a clock



30

Torpedo Implementation

```
(define (torpedo position velocity)
  (define (explode torp)
    (display "torpedo goes off!")
    (remove-from-universe torp))
  (define (move)
    (set! position ...))
  (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))))
```

name who to call message to send

```
((clock 'ADD-CALLBACK)
   (clock-callback 'moveit me 'MOVE))
  ME)
```

32

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)
```

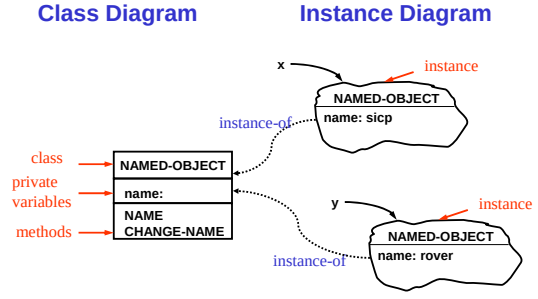
33

Summary, so far…

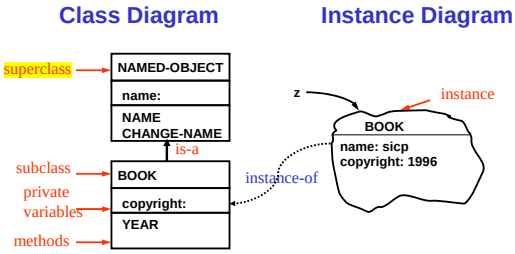
- Introduced a new programming style:
  - *Object-oriented* vs. *Procedural*
  - Uses – simulations, complex systems, ...
- Object-Oriented Modeling
  - Language independent!
    - **Class** – template for state and behavior
    - **Instances** – specific objects with their own identities
- Next: *inheritance* and *delegation*

35

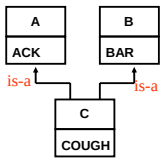
Abstract View – Class/Instance Diagrams



Abstract View – with Inheritance



Abstract View: Multiple Inheritance



- Superclass & Subclass
  - A is a **superclass** of C
  - C is a **subclass** of both A & B
    - C "is-a" B
    - C "is-a" A
- A subclass **inherits** the state variables and methods of its superclasses
  - Class C has methods **ACK**, **BAR**, and **COUGH**

User View: OO System in Scheme

- **Class**: defined by a **<type>** procedure (e.g. **named-object**)
  - Defines what is common to all instances of that class
    - Provides local state variables
    - Provides a message handler to implement methods
    - Specifies what superclasses and methods are inherited
  - Root class: **root-object**
    - All user defined classes should inherit from either **root-object** class or from some other superclass
  - Types:
    - Each class should specialize the TYPE method

User View: OO System in Scheme

- **Instance**: created by a **create-<type>** procedure (e.g. **create-named-object**)
  - Each instance has its own identity in sense of **eq?**
  - One can invoke methods on the instance:
    `(ask <instance> '<message> <arg1> … argn>)`
  - Default methods for all instances:
    `(ask <instance> 'TYPE)`
    ⇒ **`(<type> <supertype> …)`**

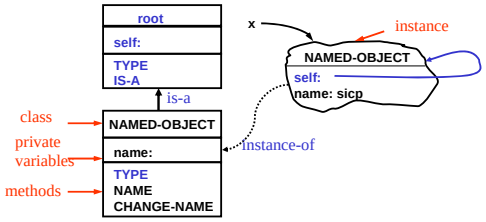
    `(ask <instance> 'IS-A <some-type>)`
    ⇒ **`<boolean>`**

A sidebar on interacting with objects

This is just sending a message to an object together with some arguments

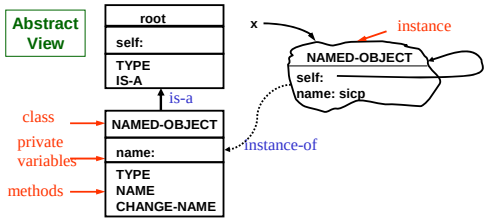
```
(define (ask object message . args)
  (let ((method (get-method message object)))
    (cond ((method? method)
           (apply method args))
          (else
           (error "No method for" message 'in
                  (safe-ask 'UNNAMED-OBJECT object 'NAME))))))

(define (get-method message . objects)
  (find-method-from-list message objects))

(define (find-method-from-list message objects)
  (if (null? objects)
      (no-method)   ;; we are suppressing a few details here
      (let ((method ((car objects) message)))
        (if (not (eq? method (no-method)))
            method
            (find-method-from-list message (cdr objects))))))
```

OO System in Scheme



- **Named-object** inherits from our **root** class
  - Gains a "self" variable: each instance can refer to itself
  - Gains an IS-A method
  - Specializes a TYPE method

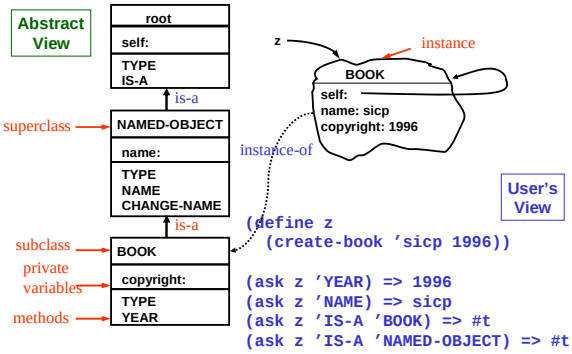
User View: Using an Instance in Scheme



User View

```
(define x (create-named-object 'sicp))
(ask x 'NAME) => sicp
(ask x 'CHANGE-NAME 'sicp-2nd-ed)
(ask x 'NAME) => sicp-2nd-ed)

(ask x 'TYPE) => (named-object root)
(ask x 'IS-A 'NAMED-OBJECT) => #t
(ask x 'IS-A 'CLOCK) => #f
```


OO System View in Scheme – with Inheritance




```
(define z
  (create-book 'sicp 1996))

(ask z 'YEAR) => 1996
(ask z 'NAME) => sicp
(ask z 'IS-A 'BOOK) => #t
(ask z 'IS-A 'NAMED-OBJECT) => #t
```


An Intermediate Step: Message Handlers

- Object behaviors are specified using **message-handlers**
- Response to every **message** is a **method**
- A **method** is a procedure that can be applied to actually do the work

```
(define (make-named-object-handler name)
  (lambda (message)
    (cond ((eq? message 'NAME)
           (lambda () name))
          ((eq? message 'CHANGE-NAME)
           (lambda (new-name) (set! name new-name)))
          (else (no-method)))))
```

This is an illustrative example –in the project we will clean this up to insert handlers inside each class

Alternative *case* syntax for message match:

- `case` is more general than this (see Scheme manual), but our convention for message matching will be:

```
(case message
 ((<msg-1>) <method-1>)
 ((<msg-2>) <method-2>)
 ...
 ((<msg-n>) <method-n>)
 (else <expr>))))
```


An Intermediate Step: Handler with case syntax

- Object behaviors are specified using **message-handlers**
- Response to every **message** is a **method**
- A **method** is a procedure that can be applied to actually do the work

```
(define (make-named-object-handler name)
  (lambda (message)
    (case message
      ((NAME)
       (lambda () name))
      ((CHANGE-NAME)
       (lambda (new-name) (set! name new-name)))
      (else (no-method))) ))
```

Big Step: User's View of Class Definition

- A class is defined by a **<type>** procedure
  - **inherited classes**
  - **local state (must have "self" as first argument)**
  - **message handler** with **messages** and **methods** for the class
    - must have a **TYPE** method as shown
    - must have **(else (get-method …))** case to inherit methods

```
(define (<type> self <arg1> <arg2> … <argn> )
  (let ((<super1>-part (<super1> self <args>)
        (<super2>-part (<super2> self <args>)
        <other superclasses>
        <other local state> )
    (lambda (message)
      (case message
        ((TYPE) (lambda ()
                  (type-extend '<type> <super1>-part
                                       <super2>-part …)) )
        <other messages and methods>
        (else (get-method message <super1>-part
                                  <super2>-part …))))))
```

We will eventually replace this with some cleaner code

48

User's View: Instance Creation

- User should provide a **create-<type>** procedure for each class
  - Uses the **create-instance** higher order procedure to
    - Generate an instance object
    - Make and add the **message handler** for the object
    - Return the instance object
- An instance is created by applying the **create-<type>** procedure

```
(define (create-<type> <arg1> <arg2> … <argn>)
  (create-instance <type> <arg1> <arg2> … <argn>) )

(define <instance> (create-<type> <arg1> <arg2> … <argn>))
```

49

User's View Example: BOOK Class with Inheritance

```
; create-book: symbol, number -> book
(define (create-book name copyright)
  (create-instance book name copyright))


(define (book self name copyright)
  (let ((named-object-part (named-object self name)))
    (lambda (message)
      (case message
        ((TYPE) (lambda () (type-extend 'book named-object)))
        ((YEAR) (lambda () copyright))
        (else (get-method message named-object-part))))))
```

instance creater for new class

message handler for new class

local state for class

superclass

make superclass message handler

new message

new method

use inherited methods

50

Another Example: NAMED-OBJECT Class

```
(define (create-named-object name) ; symbol -> named-object
  (create-instance named-object name))

(define (named-object self name)
  (let ((root-part (root-object self)))
    (lambda (message)
      (case message
        ((TYPE)
         (lambda () (type-extend 'named-object root-part)))
        ((NAME)
         (lambda () name))
        ((CHANGE-NAME)
         (lambda (newname) (set! name newname)))
        (else (get-method message root-part))))))
```

- In this example, **named-object** only inherits from **root-object**

51

User's View: Using an Instance

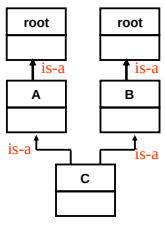
- Method lookup: get-method for <MESSAGE> from instance
- Method application: apply that method to method arguments
- Can do both steps at once:
  - **ask** an instance to do something

```
(define <inst> (create-<type> <arg1> <arg2> … <argn>))

(define some-method (get-method <instance> '<MESSAGE>))
(some-method <m-arg1> <m-arg2> … <m-argm>)

(ask <instance> '<MESSAGE> <m-arg1> <m-arg2> … <m-argm>)
```

52

User's View: Type System

- With inheritance, an instance can have multiple types
  - all objects respond to TYPE message
  - all objects respond to IS-A message




```
(define a-instance (create-A))
(define c-instance (create-C))

(ask a-instance 'TYPE) => (A root)
(ask c-instance 'TYPE) => (C A B root)

(ask c-instance 'IS-A 'C)=> #t
(ask c-instance 'IS-A 'B)=> #t
(ask c-instance 'IS-A 'A)=> #t
(ask c-instance 'IS-A 'root)=> #t

(ask a-instance 'IS-A 'C)=> #f
(ask a-instance 'IS-A 'B)=> #f
(ask a-instance 'IS-A 'A)=> #t
```

53