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

<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-translate</td>
<td>line-translate</td>
<td>2dshape-translate</td>
<td>3dshape-translate</td>
</tr>
</tbody>
</table>

Generic Operations

<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-translate</td>
<td>line-translate</td>
<td>2dshape-translate</td>
<td>3dshape-translate</td>
</tr>
<tr>
<td>color</td>
<td>point-color</td>
<td>line-color</td>
<td>2dshape-color</td>
<td>3dshape-color</td>
</tr>
</tbody>
</table>
### Generic Operations

- Adding new methods
  - Just create generic operations

<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>
</tbody>
</table>

### Generic Operations

- Adding new methods
  - Just create generic operations

<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

- 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>
<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

### 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
  - \( \text{do-something <data> <arg> ...} \)
  - \( \text{do-another-thing <data>} \)

**Object-based programming:**
- Organize system around objects that receive messages
  - \(<\text{object}> \ '\text{do-something <arg>} \)
  - \(<\text{object}> \ '\text{do-another-thing}\)
- An object encapsulates data and operations

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

A Space-Ship Object

```scheme
(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))))
)
```

Space-Ship Class

```
<table>
<thead>
<tr>
<th>class</th>
<th>private state</th>
<th>public messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHIP</td>
<td>position: velocity: num-torps:</td>
<td>POSITION VELOCITY MOVE ATTACK</td>
</tr>
</tbody>
</table>
```

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

Example – Environment Diagram

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

(enterprise 'MOVE) ==> DONE

(enterprise 'POSITION) ==> ?
```

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

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

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

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)))))
```

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**

**SHIP**
- position:
- velocity:
- num-torps:

**TORPEDO**
- position:
- velocity:

**PLANET**
- position:

**Procedure Diagram**

- **Coordinating with a clock**
  - **CLOCK**
    - The-time:
    - callbacks:
  - **CALLBACK**
    - Object:
    - message:
    - State info:
    - methods
    - Sends object message and data

- **The Universe and Time**
  ```Scheme```
  (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))
          else (error "No method" msg))))
  ```

- **Torpedo Implementation**
  ```Scheme```
  (define (make-torpedo position velocity)
    (define (explode torp)
      (display "torpedo goes off!"))
    (remove-from-universe torp))
  ```
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)`

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

Abstract View – Class/Instance Diagrams

Class Diagram                  Instance Diagram

Abstract View – with Inheritance

Class Diagram                  Instance Diagram

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

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

```scheme
(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
```

Abstract View

- self: TYPE
- `IS-A` NAMED-OBJECT
- name: sicp

OO System View in Scheme – with Inheritance

- **BOOK**
  - `copyright`: TYPE
  - `YEAR`: TYPE

User View

```scheme
(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

```scheme
(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 — we’ll see in the project that 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:

```scheme
(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))))
```

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>)
)
```

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)))))
```

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 (new-name) (set! name new-name)))
        (else (get-method message root-part))))
)
```

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 <inst> '<MESSAGE>))
(some-method <m-arg1> <m-arg2> ... <m-argm>)
```

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 (make-<type> self <arg1> <arg2> ... <argn>)
  (let ((<super1>-part (<super1> self <args>))
        (<super2>-part (<super2> self <args>))
        <other superclasses>)
    (lambda (message)
      (case message
        ((TYPE) (lambda () (type-extend '<type> <super1>-part
                                   <super2>-part ...)) )
        (else (get-method message <super1>-part
                           <super2>-part ...)))))
```

We will eventually replace this with some cleaner code
**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

```scheme
(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
```

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

**Different Views of Object-Oriented System**

- **An abstract view**
  - class and instance diagrams
  - terminology: messages, methods, inheritance, superclass, subclass, ...

- **Scheme OO system user view**
  - conventions on how to write Scheme code to:
    - define classes
    - inherit from other classes
    - create instances
    - use instances (invoke methods)

Next lecture:
- **Scheme OO system implementer view** (under the covers)
  - How implement instances, classes, inheritance, types