User View: OO System in Scheme

- **Class**: defined by a `make-<type>` procedure
  - 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: Using an Instance 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
OO System View in Scheme – with Inheritance

Abstract 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
```

User’s View of Class Definition

- A class is defined by a make-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

```scheme
(define (make-<type> self <arg1> <arg2> … <argn> )
  (let ((<super1>-part (make-<super1> self <args>)
         (<super2>-part (make-<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 …))))))
```

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

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

Object System II

- Create vs Make
- Environment Diagram
- Multiple Inheritance mechanics
- Asking a superclass
Abstract View – Class/Instance Diagrams

Class Diagram

Instance Diagram

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

User’s View: Why a “self” variable?

- Every class definition has access to a “self” variable
  - self is a pointer to the entire instance

- Why need this? How or when use self?
  - When implementing a method, sometimes you “ask” a part of yourself to do something
    - E.g. inside a BOOK method, we might...
      (ask named-object-part 'CHANGE-NAME 'mit-sicp)
  - However, sometimes we want to ask the whole instance to do something
    - E.g. inside a subclass, we might
      (ask self 'YEAR)
  - This mostly matters when we have subclass methods that shadow superclass methods, and we want to invoke one of those shadowing methods from inside the superclass

Create vs. Make

- Need notion of “self” – we use cons cell
- Don’t want multiple “self’s for each superclass – should only be one identity for each instance.
- Can’t create cons cell in “make-<type>”...
- Use “create-instance” HOP

Environment diagram for object system

- Three frames for each class-definition
  - application of make-<type>
    - self
    - args
    - let in make-<type>
      - binding of super-class handler
      - (other local state in let)
    - application of make-handler:
      - typename
      - method list
      - super-parts: list of super-class handlers
  - Handler env. is below these
  - Each class drops from GE
  - Super-part list links to handlers of super-classes
  - Self in each make-<type> frame links to single “instance”: tagged list with pointer to lowest sub-class handler.
Multiple Inheritance

- Handler list is searched in sequence
- Order of super-parts is important and determines which method is used when several super-classes specialize same method.
- In this Object system we only call one super-class method.

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A SNIFF</td>
<td>B SNIFF</td>
<td>C COUGH</td>
</tr>
</tbody>
</table>

Asking a superclass

- Generally always ask "self" to do things… we want more specialized method.
- But this can cause trouble—if we want to have change-name in BOOK also do the code for change-name in NAMED-OBJECT, we get an loop!
- Can also happen indirectly, when one method calls another which then calls the first.
- In this case appropriate to ask the super-class directly.
- ask works with handlers as well as objects (not documented, but in the code ☺)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
<td>TYPE</td>
<td>TYPE</td>
</tr>
<tr>
<td>self</td>
<td>book</td>
<td>copyright</td>
</tr>
<tr>
<td>is-a</td>
<td>is-a</td>
<td>CHANGE-NAME</td>
</tr>
<tr>
<td>NAMED-OBJECT</td>
<td>BOOK</td>
<td></td>
</tr>
<tr>
<td>name:</td>
<td>change-name:</td>
<td></td>
</tr>
<tr>
<td>TYPE</td>
<td>TYPE</td>
<td></td>
</tr>
<tr>
<td>PRIVATE</td>
<td>METHODS</td>
<td></td>
</tr>
<tr>
<td>BOOK</td>
<td>CHANGE-NAME</td>
<td></td>
</tr>
</tbody>
</table>

subclass 
is-a 
superclass 
private
variable
methods