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

Abstract View: OO Terminology

- **Class**
  - Defines what is common to all instances of that class
    - Provides local state variables
    - Provides methods which implement desired behaviors
  - Inheritance enables inclusion of other class variables & methods
    - Subclass vs. superclass
      - The subclass specializes the superclass by extending the state/behavior of the superclass
    - Classes have “is-a” relationships with other classes
      - Establishes a type hierarchy

Abstract View: OO Terminology

- **Instance**
  - An object created to the “plan” given by a class definition
  - Each instances has its own identity
    - Local state: the instance can perform based on its own state
    - An instance has a type corresponding to the class(es)

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

User View: OO System in Scheme

- Instance: created by a create-<type> procedure
  - 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>
      (ask <instance> 'METHODS) ⇒ (<METH1> ... <METHn>)

User View: Using an Instance in Scheme

(define x (create-named-object 'sicp))
(ask x 'NAME) => sicp
(ask x 'CHANGED-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 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

(define x (create-named-object 'sicp))
(ask x 'NAME) => sicp
(ask x 'CHANGED-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 x (create-named-object 'sicp))
(ask x 'NAME) => sicp
(ask x 'CHANGED-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

User View: Using an Instance in Scheme

(define x (create-named-object 'sicp))
(ask x 'NAME) => sicp
(ask x 'CHANGED-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 x (create-named-object 'sicp))
(ask x 'NAME) => sicp
(ask x 'CHANGED-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

User View: Using an Instance in Scheme

(define x (create-named-object 'sicp))
(ask x 'NAME) => sicp
(ask x 'CHANGED-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 x (create-named-object 'sicp))
(ask x 'NAME) => sicp
(ask x 'CHANGED-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
User's View of Class Definition

• Apology: Object oriented programming is implemented in Scheme, not part of Scheme
  – Therefore, difficult to separate (cleanly) the use of OOP from the implementation of OOP
  – E.g., last time, you saw the "guts" of a simple OOP:
    • objects were implemented as procedures
    • state of objects was Scheme variables (in environments)
• Today we use a more sophisticated implementation, but still show some of the "guts"
  – Simplified by conventional patterns
• We could hide much of this with new special forms, but don't!
• Exercise for the listener: Why not?

Conventions on Handling Messages

• Response to every message is a method
• A method is a procedure that can be applied to actually do the work

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

(define (torpedo self position velocity target)
  (let ((moving-object-part
        (moving-object
          self
          position
          velocity)))
    (make-handler 'torpedo
      (make-methods
        'EXPLODE
        (lambda () ...)
        ...
        (else
          (error "No method" msg))))))
```

Conventions on Handling Messages

• Object behaviors are specified using message-handlers
  • Instead of simply returning (lambda (msg ...), we call make-handler to do it for us. It also does the following:
    – Checks for errors
    – Automatically defines methods for TYPE and METHODS messages
    – Implements inheritance of methods from superclasses.

```
(define (make-handler typename methods . super-parts)
  (cond ;check for possible programmer errors
    ((not (symbol? typename))
      (error "bad typename" typename))
    ...
    (else
      (named-lambda (handler message)
        (case message
          ((TYPE)
            (lambda () (type-extend typename super-parts)))
          ((METHODS)
            (lambda ()
              (append method-names methods)))
          (else
            (find-method-from-handler-list message super-parts)))))
```

Compare old vs. new Torpedo
Big Step: User’s View of Class Definition

- A class is defined by a `define <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 be able to inherit methods

```lisp
(define (<type> self <arg1> <arg2> … <argn> )
  (let ((<super1>-part (<super1> self <args>)
                 (<super2>-part (<super2> self <args>)
                 <other superclasses>
                 <other local state>)
      (make-handler '<type>
        (make-methods
         'METHOD1 (lambda () ...)
         <other messages and methods>)
      <super1>-part
      <super2>-part ...)))
```

- A class is defined by a `define` 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 be able to inherit methods

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

```lisp
(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
- `book`: self, name, copyright
- `create-<type>` is instance creator for new class
- `create-instance book name copyright`

```lisp
(define (create-book name copyright)
  (create-instance book name copyright))
(define (book self name copyright)
  (let ((named-object-part (named-object self name)))
    (make-handler 'book
      (make-methods
       'YEAR (lambda () copyright))
    named-object)))
```

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

Another Example: NAMED-OBJECT Class

- `create-named-object`: symbol -> named-object
- `named-object`: self, name
- `create-instance named-object`

```lisp
(define (create-named-object name)   ; symbol -> named-object
  (create-instance named-object name))
(define (named-object self name)
  (let ((root-part (root-object self)))
    (make-handler
      'named-object
      (make-methods
       'NAME    (lambda () name)
       'CHANGE-NAME (lambda (new-name)
                     (set! name new-name))
       'INSTALL (lambda () 'installed)
       'DESTROY (lambda () 'destroyed)
       'root-part)))
```

Defining the root-object

- `root-object`: self
- `ask`: type
- `IS-A`: lambda
- `MEMQ`: type (ask self 'TYPE))

```lisp
(define (root-object self)
  (make-handler
    'root
    (make-methods
     'IS-A
     (lambda (type) (memq type (ask self 'TYPE))))))
```

- We can begin to see the use of the self variable
- But more later!

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

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

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

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

• 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)
  ➔ Scheme OO system implementer view (under the covers)
    - How we implement instances, classes, inheritance, types

Reminder: Example Class/Instance Diagram

Implementer’s View of this in Environ. Model

Implementer’s View: get-method and ask
- method lookup:
  
```
(define (get-method message object)
  (object message))
```
- "ask" an object to do something - combined method retrieval and application to args.

```
(define (ask object message . args)
  (let ((method (get-method message object)))
    (if (method? method)
        (apply method args)
        (error "No method for message" message))))
```

```
(apply op args) → (op arg1 arg2 ... argn)
```

Implementer’s View: Instances

```
(define (make-instance)
  (list 'instance #f))

(define (instance? x)
  (and (pair? x) (eq? (car x) 'instance)))

(define (instance-handler instance) (cadr instance))

(define (set-instance-handler! instance handler)
  (set-car! (cdr instance) handler))

(define (create-instance maker . args)
  (let* ((instance (make-instance))
         (handler (apply maker instance args)))
    (set-instance-handler! instance handler)
    (if (method? (get-method 'INSTALL instance))
        (ask instance 'INSTALL)
        instance))
```

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed euismod, lorem vel egestas hendrerit, odio massa congue est, at dapibus magna nibh feugiat nisi.
Implementer's View of this in Environ. Model

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

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...
      ```lisp
      (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
        ```lisp
        (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
        - Remember IS-A in root-object!
- Next time: An example OO design to illustrate our OO system

OOP Languages hide more details

- Common Lisp Object System
  ```lisp
  (defclass book (named-object)
    (copyright)
    ..
  )
  ```
- Java
  ```java
  public class book extends namedObject {
    Date copyright;
    ..
  }
  ```
- …but in all of these, there are tell-tale aspects of the implementation that peek through, e.g., when a method for a subclass also needs to call the same method on a superclass.