Different Views of Object-Oriented System

- **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 instance 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

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
Different Views of Object-Oriented System

- **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>)`
    `=> #t`
    `(ask <instance> 'METHODS)`
    `=> (<METHOD1> ... <METHODn>)`

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
```

OO System View in Scheme – with Inheritance

```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) => #f
```

Abstract View

- **Instance**: created by a `create-type` procedure
- **Class**: defined by a `<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
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 (and in Project 4) 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!

Conventions on Handling Messages

• 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
• 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.

Compare old vs. new Torpedo

```
(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)
           (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 () ...) ...)
        moving-object-part)))
```

Alternative case syntax for message match:

• Yet another special form (syntactic sugar... yum!)
• `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>))
```

MAKE-HANDLER does a lot of work

```
(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)
                                 (append-map
                                  (lambda (x) (ask x 'METHODS)) super-parts))))
             (else (let ((entry (method-lookup message methods)))
                    (if entry
                        (cadr entry)
                        (find-method-from-handler-list message super-parts))))))))
```

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 case to inherit methods
```
(define (<type> self <arg1> <arg2> ... <argn>)
  (let ((super1-part (super1-part super2-part ... <arg>))
        (super2-part (super2-part other superclasses)
                    other local states))
    (make-handler 'type
      (make-methods 'METHODS1 (lambda () ...) ...
                  (other messages and methods) )
      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

\[(define (create-type <arg1> <arg2> … <argn>)
  (create-instance <type> <arg1> <arg2> … <argn>));\]
\[(define <instance> (create-type <arg1> <arg2> … <argn>));\]

User’s View Example: BOOK Class with Inheritance

- `create-book: symbol, number -> book`

\[(define (create-book name copyright)
  (create-instance book name copyright));\]

\[\textbf{instance creator for new class}\]
\[(define (book self name copyright)
  (let ((named-object-part (named-object self name)))
    (make-handler 'book
      (make-methods
        'YEAR (lambda () copyright))
    named-object)));\]

\[\textbf{superclass make superclass message handler}\]
\[\textbf{new method}\]

Another Example: NAMED-OBJECT Class

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

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

User’s View: Using an Instance

\[(define <inst> (create-type <arg1> <arg2> … <argn>));\]
\[(define some-method (get-method <instance> '<MESSAGE>));\]
\[(some-method <m-arg1> <m-arg2> … <m-argm>));\]

- 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

\[(define <inst> (create-type <arg1> <arg2> … <argn>));\]
\[(define some-method (get-method <instance> '<MESSAGE>));\]
\[(some-method <m-arg1> <m-arg2> … <m-argm>));\]
\[(ask <instance> '<MESSAGE> <m-arg1> <m-arg2> … <m-argm>));\]

Defining the root-object

\[(define (root-object self)
  (make-handler
    'root
    (make-methods
      'IS-A
        (lambda (type)
          (memq type (ask self 'TYPE))))));\]

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

User’s View: Type System

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

\[(define a-instance (create-A));\]
\[(define c-instance (create-C));\]
\[(ask a-instance 'TYPE) => (A root);\]
\[(ask c-instance 'TYPE) => (C A B root);\]
\[(ask c-instance 'IS-A 'C) => #t;\]
\[(ask c-instance 'IS-A 'A) => #f;\]
\[(ask c-instance 'IS-A 'B) => #f;\]
\[(ask a-instance 'IS-A 'C) => #f;\]
\[(ask a-instance 'IS-A 'A) => #t;\]
\[(ask a-instance 'IS-A 'B) => #f;\]
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**
  ```scheme
  (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)))
  ```

- **Implementer's View: Instances**
  ```scheme
  (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)))
  ```

- **Implementer’s View: method and ask**
  - method lookup:
    ```scheme
    (define (method? symbol)
      (case symbol
        (\noop (\no\)\)
        (\method (\method\)\)
        (true))
    )
    ```

- **Implementer's View of this in Environ. Model**
  ```scheme
  (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)))
  ```

- **Implementer's View of this in Environ. Model**
  ```scheme
  (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)))
  ```
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
    – 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
  (defclass book (named-object)
   (copyright)
   ...
  )

• 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.