

# Programming Languages (CS 550)

## Lecture 5 Summary Object Oriented Programming and Implementation\*

Jeremy R. Johnson

\*Lecture based on notes from SICP

# Theme

- ❖ Object oriented programming provides a convenient mechanism for modeling state. State is local to the object itself. Each object includes a set of functions (methods) through which its state can be accessed and changed.
- ❖ Inheritance is the major mechanism of object-oriented languages that allows sharing of data and operations among classes, as well as the ability to redefine these operations without modifying existing code (type polymorphism).
- ❖ Objects can be easily implemented in a language with first class functions (environments). Using an interpreter we can incorporate these features in a language.

# Outline

- ❖ Review concepts of OOP
- ❖ Implementing objects in scheme
  - Store local state in environment
  - Use message passing for method invocation
  - Inheritance stores internally superclass
- ❖ Creating an object oriented language
  - TOOL (Tiny Object Oriented Language)
  - Extend metacircular interpreter to support classes, class instantiation, generic functions, and methods

# Object Oriented Programming

- ❖ Classes
- ❖ Object instantiation (constructors)
- ❖ Instance variables
- ❖ Methods and method invocation
- ❖ Inheritance
  - Single and multiple inheritance
  - Abstract classes and dynamic binding
  - Subtype polymorphism

# Modeling State in Scheme

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Unknown request -- MAKE-ACCOUNT"
                       m))))
  dispatch)
```

```
(define acc (make-account 100))
((acc 'withdraw) 50)
50
((acc 'withdraw) 60)
"Insufficient funds"
((acc 'deposit) 40)
90
((acc 'withdraw) 60)
30
```

# Objects in Scheme

- ❖ An object is a procedure that, given a message as argument, returns another procedure called a method

```
(define (get-method object message)
  (object message))
```

```
(define (make-speaker)
  (define (self message)
    (cond ((eq? message 'say)
           (lambda (stuff) (display stuff)))
          (else (no-method "SPEAKER"))))
    self)
```

# Invoking a Method

- ❖ To ask an object to apply a method to some arguments we send the object a message asking for the appropriate method, and apply the method to the arguments□

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

# Example

(define george (make-speaker))

;Value: george

(ask george 'say '(the sky is blue))

(the sky is blue)

# Inheritance

❖ One thing we may want to do is define an object type to be a kind of some other object type

```
(define (make-lecturer)
  (let ((speaker (make-speaker)))
    (define (self message)
      (cond ((eq? message 'lecture)
             (lambda (stuff)
               (ask self 'say stuff)
               (ask self 'say '(you should be taking notes))))
            (else (get-method speaker message))))
      self))
```

# Example

```
(define gerry (make-lecturer))
```

```
;Value: gerry
```

```
(ask gerry 'say '(the sky is blue))
```

```
(the sky is blue)
```

```
;Unspecified return value
```

```
(ask gerry 'lecture '(the sky is blue))
```

```
(the sky is blue)(you should be taking notes)
```

```
;Unspecified return value
```

# More Inheritance

❖ The following example shows a bug in the implementation (self is lost)

```
(define (make-arrogant-lecturer)
  (let ((lecturer (make-lecturer)))
    (define (self message)
      (cond ((eq? message 'say)
             (lambda (stuff)
               (ask lecturer 'say (append '(it is obvious that) stuff))))
            (else (get-method lecturer message))))
      self))
```

# A Bug: Losing Self

❖ The problem is that when Albert calls internal lecturer, Albert's self is lost. Albert, his internal lecturer, and his internal lecturer's internal speaker, each have a self

(define albert (make-arrogant-lecturer))

(ask albert 'say '(the sky is blue))

(it is obvious that the sky is blue)

(ask albert 'lecture '(the sky is blue))

(the sky is blue)(you should be taking notes)

# Fixing the Bug

- ❖ We can fix this by changing the implementation so that all the methods keep track of self by taking self as an extra input

```
(define (make-speaker)
  (define (self message)
    (cond ((eq? message 'say)
           (lambda (self stuff) (display stuff)))
          (else (no-method message))))
  self)

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

# Multiple Inheritance

- ❖ We can have object types that inherit methods from more than one type

```
(define (make-singer)
  (lambda (message)
    (cond ((eq? message 'say)
           (lambda (self stuff)
             (display (append '(tra-la-la --) stuff))))
          ((eq? message 'sing)
           (lambda (self)
             (display '(tra-la-la))))
          (else (no-method "SINGER"))))))
```

# Multiple Inheritance (singer/lecturer)

```
(define ben
  (let ((singer (make-singer))
        (lecturer (make-lecturer)))
    (lambda (message)
      (let ((sing (get-method singer message))
            (lect (get-method lecturer message)))
        (if (method? sing)
            sing
            lect))))))
```

```
(ask ben 'sing)
(tra-la-la)
(ask ben 'say '(the sky is blue))
(tra-la-la -- the sky is blue)
(ask ben 'lecture '(the sky is blue))
(tra-la-la -- the sky is blue)
(tra-la-la -- you should be taking notes)
```

# Multiple Inheritance (lecturer/singer)

```
(define alyssa
  (let ((singer (make-singer))
        (lecturer (make-lecturer)))
    (lambda (message)
      (let ((sing (get-method singer message))
            (lect (get-method lecturer message)))
        (if (method? lect)
            lect
            sing))))))
```

```
(ask alyssa 'sing)
```

```
(tra-la-la)
```

```
(ask alyssa 'lecture '(the sky is blue))
```

```
(the sky is blue)(you should be taking notes)
```

# TOOL Overview

## ❖ Define classes

➤ (define-class *name superclass . slots*)

## ❖ Instantiate classes

➤ (make *class slot-names-and-values*)

## ❖ Generic functions (dispatch on signature)

➤ (define-generic-function *name*)

## ❖ Method definition

➤ (define-method *generic-function (params-and-classes . body)*)

# TOOL Examples

```
(load "teval")  
;Loading "teval.scm"... done  
;Value: set-binding-value!
```

```
(initialize-tool)
```

```
TOOL==> (define-class <cat> <object> size breed)  
(defined class: <cat>)
```

```
TOOL==> (define garfield (make <cat> (size 6) (breed 'wierd)))  
*undefined*
```

```
TOOL==> (get-slot garfield 'breed)  
wierd
```

```
TOOL==> (get-slot garfield 'size)
```

# TOOL Examples

```
TOOL==> (define-generic-function 4-legged?)  
(defined generic function: 4-legged?)
```

```
TOOL==> (define-method 4-legged? ((x <cat>)) true)  
(added method to generic function: 4-legged?)
```

```
TOOL==> (4-legged? garfield)  
#t
```

```
TOOL==> (define-method 4-legged? ((x <object>)) 'who-knows)  
(added method to generic function: 4-legged?)
```

```
TOOL==> (4-legged? 3)  
who-knows
```

# TOOL Examples

```
TOOL==> (define-generic-function say)
(defined generic function: say)
```

```
TOOL==> (define-method say ((x <cat>) (stuff <object>))
  (print 'meow:) (print stuff))
(added method to generic function: say)
```

```
TOOL==> (define-class <show-cat> <cat> awards)
(defined class: <show-cat>)
```

```
TOOL==> (define-method say ((cat <show-cat>) (stuff <object>))
  (print stuff)
  (print '(I am beautiful)))
(added method to generic function: say)
```

# TOOL Examples

```
TOOL==> (define Cornelius-Silverspoon-the-Third
  (make <show-cat>
    (size 'large)
    (breed '(Cornish Rex))
    (awards '((prettiest skin))))))
```

\*undefined\*

```
TOOL==> (say Cornelius-Silverspoon-the-Third '(feed me))
(feed me)
(i am beautiful)
#!unspecific
```

```
TOOL==> (define-method say ((cat <cat>) (stuff <number>))
  (print '(cats never discuss numbers)))
(added method to generic function: say)
```

```
TOOL==> (say fluffy 37)
(cats never discuss numbers)
#!unspecific
```

# TOOL Implementation

- ❖ Extend metacircular interpreter to handle class, generic function, and method definitions, and instance creations
- ❖ Provide code to apply a generic function

# Modified Scheme Interpreter eval

```
(define (tool-eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((quoted? exp) (text-of-quotation exp))
        ((variable? exp) (lookup-variable-value exp env))
        ((definition? exp) (eval-definition exp env))
        ((assignment? exp) (eval-assignment exp env))
        ;;((lambda? exp) (make-procedure exp env))
        ((conditional? exp) (eval-cond (clauses exp) env))
        ((generic-function-definition? exp)
         (eval-generic-function-definition exp env))
        ((method-definition? exp) (eval-define-method exp env))
        ((class-definition? exp) (eval-define-class exp env))
        ((instance-creation? exp) (eval-make exp env))
        ((application? exp)
         (tool-apply (tool-eval (operator exp) env)
                      (map (lambda (operand) (tool-eval operand env))
                           (operands exp))))
        (else (error "Unknown expression type -- EVAL >>" exp))))
```

# Modified Scheme Interpreter

## apply

```
(define (tool-apply procedure arguments)
  (cond ((primitive-procedure? procedure)
        (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment (parameters procedure)
                           arguments
                           (procedure-environment procedure))))
        ((generic-function? procedure)
         (apply-generic-function procedure arguments))
        (else (error "Unknown procedure type -- APPLY"))))
```

# TOOL Data Structures

## ❖ Class

- Class name, list of slots, list of ancestors

## ❖ Generic function

- Name, list of methods defined for function

## ❖ Method

- Specializers, procedure

## ❖ Instance

- Class, list of values for slots

# Defining Generic Functions

**(define-generic-function name)**

```
(define (eval-generic-function-definition exp env)
  (let ((name (generic-function-definition-name exp)))
    (let ((val (make-generic-function name)))
      (define-variable! name val env)
      (list 'defined 'generic 'function: name))))
```

# Defining Methods

```
(define-method generic-function (params-and-classes) . Body)
```

```
(define (eval-define-method exp env)
  (let ((gf (tool-eval (method-definition-generic-function exp) env)))
    (if (not (generic-function? gf))
        (error "Unrecognized generic function -- DEFINE-METHOD >> "
              (method-definition-generic-function exp))
        (let ((params (method-definition-parameters exp)))
          (install-method-in-generic-function
           gf
           (map (lambda (p) (paramlist-element-class p env))
                params)
           (make-procedure (make-lambda-expression
                           (map paramlist-element-name params)
                           (method-definition-body exp))
                          env))
          (list 'added 'method 'to 'generic 'function:
                (generic-function-name gf)))))))
```

# Defining and Instantiating Classes

```
(define-class name superclass . Slots) (make class slot-names-and-values)
```

```
(define (eval-define-class exp env)
  (let ((superclass (tool-eval (class-definition-superclass exp)
                               env)))
    (if (not (class? superclass))
        (error "Unrecognized superclass -- MAKE-CLASS >> "
              (class-definition-superclass exp))
        (let ((name (class-definition-name exp))
              (all-slots (collect-slots
                          (class-definition-slot-names exp)
                          superclass)))
          (let ((new-class
                 (make-class name superclass all-slots)))
            (define-variable! name new-class env)
            (list 'defined 'class: name))))))
```

# Applying Generic Functions

- ❖ Find all methods that are applicable
- ❖ Use first one (most specialized)
  - Method1 < Method2 (class method1 is a subclass of class method2)
- ❖ Extract procedure for that method and apply

```
(define (apply-generic-function generic-function arguments)
  (let ((methods (compute-applicable-methods-using-classes
                  generic-function
                  (map class-of arguments))))
    (if (null? methods)
        (error "No method found -- APPLY-GENERIC-FUNCTION")
        (tool-apply (method-procedure (car methods)) arguments))))
```

# Applying Generic Functions

```
(define (compute-applicable-methods-using-classes generic-function classes)
  (sort
   (filter
    (lambda (method)
      (method-applies-to-classes? method classes))
    (generic-function-methods generic-function))
   method-more-specific?))
```

```
(define (method-applies-to-classes? method classes)
  (define (check-classes supplied required)
    (cond ((and (null? supplied) (null? required)) true)
          ;;something left over, so number of arguments does not match
          ((or (null? supplied) (null? required)) false)
          ((subclass? (car supplied) (car required))      ;; class1 is subclass of class2 if
           (check-classes (cdr supplied) (cdr required)))  ;; class2 in ancestor list of class1
          (else false)
          ))
  (check-classes classes (method-specializers method)))
```

# Assignment 5

- ❖ Extend assignment 4 (mini language with first class functions) to support classes, objects, and inheritance
  - Class name ( $p_1, \dots, p_n$ )  
    <stmt-list>  
    End
  - The statement list defines the methods ( $method_1, \dots, method_n$ ) and attributes in the class
  - The class definition provides a constructor with parameters  $p_1, \dots, p_n$ , called by  $name(p_1, \dots, p_n)$ . The constructor returns an object which can access the methods and attributes of the class initialized by the constructor using the familiar dot notation.

# Assignment 5

## ❖ Example

```
Class list(init)
  L := init;
  Cons := proc(x) return cons(x,L) end;
  Car := proc() return car(L) end;
  Cdr := proc() return cdr(L) end;
end;
```

```
L := list([]);
L.Cons(3);
L.Cons(2);
L.Cons(1);
x := L.Car(); // = 1
M := L.Cdr(); // = [2,3]
```

# Assignment 5

## ❖ Inheritance

- Classes can be derived from other classes using the notation

Class name(p1,...,pn) : supername

<stmt-list>

End

- Objects in the derived class should be able to access methods and attributes from the derived class, however, they can be redefined in the subclass.