1

Common Lisp Object System (CLOS)

- Introduces functions in Lisp so that one can
  - define *classes* (`defclass`)
  - *instantiate* classes (`make-instance`)
  - assign/fetch values of *slots* (object variables)
  - define methods (`defmethod`)

- CLOS is an interesting object-oriented language because it supports multiple inheritance and multiple method attachment.

- Unlike OOT, and other imperative languages, were records and modules constitute the starting point for introducing objects in the language, in CLOS the starting point is atoms.

In many ways, Lisp is a much more appropriate foundation for defining object-oriented programming constructs than imperative languages.

@1993 John Mylopoulos

2

Slot Facets in CLOS

```
→ (defclass triangle (polygon)
            ((num-sides :initform 3)
             (base :accessor t-base :initarg :base)
             (altitude :accessor height :initarg :altitude)) )
nil
                ;; defines triangle as a subclass of polygon
                having 3 slots:
                    num-sides (initialized to 3)
                    base and altitude.
                ;; we can use the values of :accessor and
                :initarg facets:

→ (setf t1 (make-instance 'triangle :altitude 7))
#<triangle 115281776>
                ;; t1 is a triangle with its altitude slot set to 7.

→ (slot-value t1 'altitude)
7
                ;; we could also use "height" here.

→ (height t1)
7
                ;; since altitude's :accessor facet is "height"
```


@1993 John Mylopoulos

3

Slot Facets

Slot facets available in CLOS:

:initarg - allows the initialization of a slot during the creation of an instance

:reader - causes the creation of a generic function used to access the value of the slot

:accessor - causes the creation of a generic function used to fetch or update the value of the slot

:documentation - this is a class slot in the sense that it pertains to the class as an object in its own right

:initform - allows the assignment of a default value to a slot

@1993 John Mylopoulos

4

Redefining Classes in CLOS

Classes can be redefined by simply evaluating another defclass form. What happens to the slots and their values for instances of a redefined class? They may be preserved, re-initialized or deleted:

- when the same slot is specified in both definitions, the value of the slot is preserved.

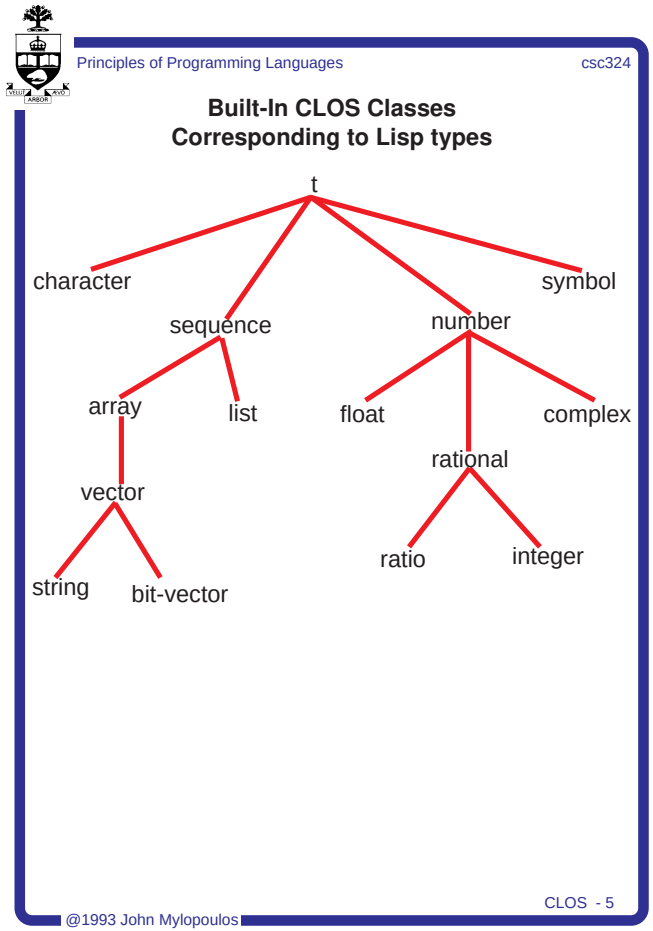
- when a slot is specified by the new definition but was not specified by the old definition, the slot is added to all existing instances and initialized according to :initform option.

- when a slot specified by the old definition is not specified by the new, the slot is deleted from the instances of the class and any value is discarded; however, discarding may be customized through the special generic function update-instance-of-redefined-class

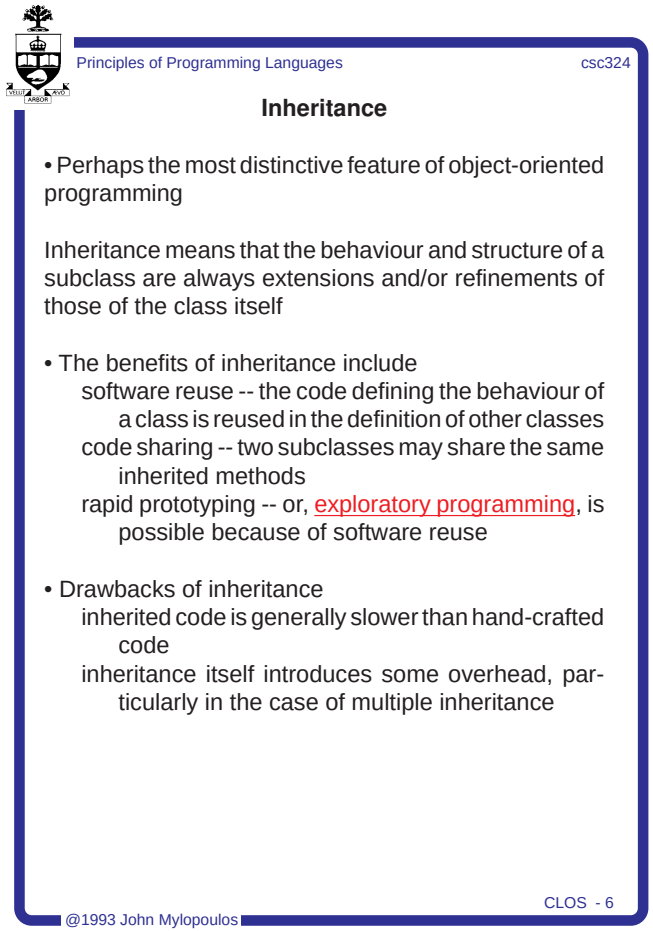
This feature is not available in compiler-based object-oriented languages such as OOT

@1993 John Mylopoulos

5


Principles of Programming Languages
csc324
Built-In CLOS Classes
Corresponding to Lisp types
t
character
sequence
number
symbol
array
list
float
complex
rational
vector
ratio
integer
string
bit-vector
CLOS - 5
@1993 John Mylopoulos

6


Principles of Programming Languages
csc324
Inheritance
• Perhaps the most distinctive feature of object-oriented programming
Inheritance means that the behaviour and structure of a subclass are always extensions and/or refinements of those of the class itself
• The benefits of inheritance include
software reuse -- the code defining the behaviour of a class is reused in the definition of other classes
code sharing -- two subclasses may share the same inherited methods
rapid prototyping -- or, exploratory programming, is possible because of software reuse
• Drawbacks of inheritance
inherited code is generally slower than hand-crafted code
inheritance itself introduces some overhead, particularly in the case of multiple inheritance
CLOS - 6
@1993 John Mylopoulos

7

What is the Value of a Slot?

Its most recent assignment

- For example, suppose

```
→ (defclass pentagon (polygon)
    ((num-sides :initform 5
                :initarg :num-sides)) )
```

- "First" value comes from defclass statement; after

```
→ (setf abc (make-instance
                        'pentagon))
```

the slot value of `num-sides` is 5

- "Second" value is from `make-instance` statement; after

```
→ (setf abc (make-instance
          'pentagon :num-sides 19))
```

the slot value of `num-sides` is 19

- Subsequent evaluations of forms

```
→ (setf (slot-value abc
                'num-sides) X  )
```

sets `num-sides` to *X*

@1993 John Mylopoulos

8

Inheritance: Going up the Generalization Hierarchy

```
→ (defclass polygon (figure)
        ((planar :initform 'yes)))
nil
            ;; Polygons are defined as planar figures

→ (defclass pentagon (polygon)
            ((num-sides :initform 5)))
nil
            ;; The definition of pentagons is silent on
planarity

→ (setf p1 (make-instance 'pentagon))
#<pentagon 1153378994>

→ (slot-value p1 'planar)
yes
          ;; Reasonable? Notice that evaluation of this form
          ;; requires going from p1 to its class, pentagon, to
          ;; its superclasses, polygon, figure,...
```


@1993 John Mylopoulos

9

Inheritance: The Specificity Rule

```
→ (defclass polygon (figure)
            ((num-sides :initform 8)) )
nil
                ;; Many polygons come with 8 sides...
→ (defclass pentagon (polygon)
          ((num-sides :initform 5)) )
nil
                ;;...but of course, pentagons have 5 sides
→ (setf p1 (make-instance 'polygon))
#<polygon 115331744>
→ (slot-value p1 'num-sides)
8
→ (setf p2 (make-instance 'pentagon))
#<pentagon 115347396>
→ (slot-value p2 'num-sides)
5
            ;; among p2's class and its superclasses, pentagon
            ;; is the most specific class that supplies a value
```

Specificity rule: When there are several classes from which an object inherits a value for a slot, pick the most specific class

@1993 John Mylopoulos

10

Inheritance and Overriding

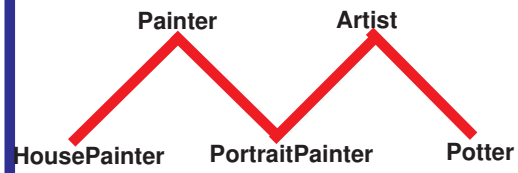
```
→ (defclass animal ()
          ((fly :initform 'no)) )
nil                ;; Most animals don't fly.
→ (setf phydeaux
             (make-instance 'animal)))
#<animal 122331744
→ (slot-value phydeaux 'fly)
no                 ;; All I know about phydeaux is that it's
                   an animal, so it doesn't fly.
→ (defclass bird (animal)
             ((fly :initform 'yes)) )
nil                ;; Most birds do fly.
→ (setf fred (make-instance 'bird))
#<bird 213313744>
→ (slot-value fred 'fly)
yes                ;; Specificity rule prevails
→ (defclass penguin (bird)
                ((fly :initform 'no)) )
nil                ;; Most penguins don't fly.
→ (setf tweety (make-instance 'penguin))
#<penguin 213378744>
→ (slot-value tweety 'fly)
no                ;; Specificity rule, again.
```


@1993 John Mylopoulos

11

Multiple Inheritance

There are many situations where a subclass needs to inherit from more than one other class



Most object-oriented languages do not support multiple inheritance because it is expensive to implement and because it can lead to semantic problems such as the following:

• Name ambiguity -- the same name may refer to different methods in the parent classes from which a class inherits.

• Inheriting from common ancestors -- the problem here is that a class A may have two parent classes which have a common ancestor; if that ancestor has an instance variable X, should A have one or two versions of that instance variable?

@1993 John Mylopoulos

12

Multiple Inheritance and Mixin Classes in CLOS

A class can be defined to be a subclass of several other classes, not just one.

Mixin classes are used in combination with other classes. They usually define general features of other classes. For example, suppose we want to have an author slot for (some) objects:

```
(defclass authored-object ()
    ( author :type person))
```

The :type facet indicates that the values of author must be instances of person. If we now want triangles to be authored objects, we define triangle as

```
(defclass triangle (shape authored-object)
    ( ...                                    ))
```


@1993 John Mylopoulos

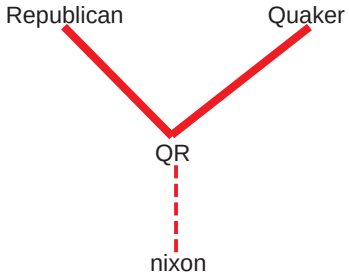
13

Multiple Inheritance

```
→ (defclass quaker ()
        ((pacifist :initform 'yes)) )
nil
→ (defclass republican ()
          ((pacifist :initform 'no)) )
nil
```

;; Quakers tend to be pacifists, republicans not.

```
→ (defclass QR (republican quaker) ())
nil
→ (setf nixon (make-instance 'QR) )
#<QR 114474355>
→ (slot-value nixon 'pacifist)
no
```



@1993 John Mylopoulos

14

Multiple Inheritance: Another Example

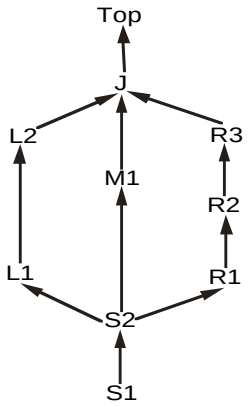
```
→ (defclass polygon (figure)
        ((planar :initform 'yes)) )
nil

→ (defclass pentagon (polygon)
          ((num-sides :initform 5)) )
nil

→ (defclass non-planar (figure)
            ((planar :initform 'no)) )
nil

→ (defclass funny-pents
          (non-planar pentagon) () )
nil

→ (setf fp1 (make-instance 'funny-pents))
#<funny-pents 115781784>

→ (slot-value fp1 'planar)
no
```

;; CLOS uses order of superclasses in the
;; declaration of funny-pents.

@1993 John Mylopoulos

15

Inheritance Ordering in CLOS



Ordering here: S1, S2, L1, L2, M1, R1, R2, R3, J, Top

```
(defclass J () (...))
(defclass L2 (J) (...))
(defclass L1 (L2) (...))
(defclass M1 (J) (...))
(defclass R3 (J) (...))
(defclass R2 (R3) (...))
(defclass R1 (R2) (...))
(defclass S2 (L1 M1 R1) (...))
(defclass S1 (S2) (...))
```


@1993 John Mylopoulos

16

Objects vs Pointers to Objects

OOT offers pointers to objects as values to be assigned to variables; CLOS offers objects as values

What's the difference?

• If a variable holds as value a pointer, the contents of the address pointed at might change unexpectedly, without anyone changing the variable

• I f a variable holds as value an object (or any other type of data structure, for that matter), then the variable can't change, except through an explicit assignment (but the internal state of the object that is the value of a variable can change)

Actually, Lisp is also guilty of some pointer-dependent features, e.g., `eq`

Pointers constitute an implementation-oriented language feature

@1993 John Mylopoulos

17

Methods and `Defmethod`

A *method* defines how does a generic function behave for a particular class of objects

- A method is like a function in that it takes arguments and returns a value
- ...but...a specific method only applies for objects that are instances of a particular class
- Many methods have the same name
- defmethod is like defun, but in addition to what arguments it takes and what it does, we must specify the class for which it is attached.

@1993 John Mylopoulos

18

Examples

```
→ (defmethod area ((f circle))
        (* pi (expt (radius f) 2)) )
nil

→ (defmethod area ((f triangle))
        (* (/ 1 2) (base f) (height f)) )
nil

→ (defmethod area ((f figure))
        '(I don't know -- but it's a figure))
nil
```

What is the value of (area FF) ?

```
;; If FF is a circle, use first definition.
;; If FF is a triangle, use second definition.
;; If FF is a figure
     but not a circle or triangle,
     use third definition
;; If FF is none of these, then ERROR.
```


@1993 John Mylopoulos

19

Details on `defmethod`

(`defmethod` <generic function name> <param class list> <form>)

defines a method where

<generic function name> is name of generic function

<form> is an arbitrary Lisp form

<param class list> ::= ({ (<param> <class>) })
is a list of parameter - class pairs

A given method only applies to specific class(es)

@1993 John Mylopoulos

20

What Method to Use?

```
→ (setf c1 (make-instance 'circle :radius 9))
#<circle 12376889>

→ (area c1)
                    ;; Notice that both first and third method
                    is applicable; first is more specific, so it
is
                    chosen.
254.469

→ (setf f1 (make-instance 'figure))
#<figure 199855325>

→ (area f1)
(I don't know, but it's a figure)

→ (area '(a b))
ERROR
                    ;; A more graceful solution:

→ (defmethod area ((f t)) '(Is not a figure at all!))
nil

→ (area '(a b))
(Is not a figure at all!)
                    ;; The class t covers everything.
```


@1993 John Mylopoulos

21

Selecting a Method

Shorthand: Replace (x t) by x in parameter lists
e.g., (defmethod area (f) '(Is not a figure at all!))

When single class argument, same ordering as for inheritance

When multiple class arguments...more complicated...:

→ (defmethod foo ((a figure) (b triangle)) '(foo 1))
nil

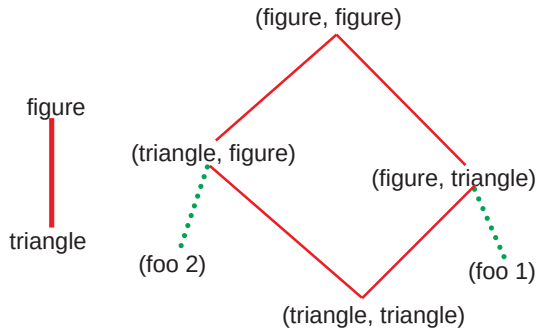
→ (defmethod foo ((a triangle) (b figure)) '(foo 2))
nil

→ (setf t1 (make-instance 'triangle))
#<triangle 112679904>

→ (foo t1 t1)
(foo 2)

@1993 John Mylopoulos

22



Look for most specific method with respect to first argument before looking at second argument

@1993 John Mylopoulos

23

Types of Methods

The methods we have seen so far are primary, intended to perform the bulk of the work for a generic function and a particular class. There are other types of methods as well:

• **before-methods** - methods called before the primary method, usually to perform set-up work

• **after-methods** - methods called after the primary method, usually to perform clean-up work

• **around-methods** - these are methods whose role is to surround all other methods; such methods usually perform some computation and use call-next-method to invoke the before methods, primary method and after methods; an around method can set up an environment to be in effect during the execution of other methods

The type of a method is indicated by a qualifier:

```
(defmethod blah-blah :around ...)
```


@1993 John Mylopoulos

24

Types of Methods

Consider, again the definition of the class triangle:

```
(defclass triangle (shape)
     ((side-a :accessor side-a :initarg :side-a)
      (side-b :accessor side-b :initarg :side-b)
      (side-c :accessor side-c :initarg :side-c)
      (number-of-sides :reader number-of-sides
          :initform 3 :allocation class)
      (area :reader area :initarg :area)))
```

To make sure the area slot has the right value initially, we define a constructor function make-triangle:

```
(defun make-triangle (side-a side-b side-c)
     (make-instance 'triangle
          :side-a side-a :side-b side-b :side-c side-c
          :area (area-of-triangle side-a side-b side-c)))
```

Now we define an after method which re-calculates the area of a triangle every time side-a is changed:

```
(defmethod (setf side-a) :after (new-length (tri triangle))
     (setf (area tri)
          (area-of-triangle
               new-length (side-b tri) (side-c tri))))
```


@1993 John Mylopoulos

25

Types of Methods (cont'd)

Say now we want to define an around method which prints the result of every area computation:

```
→ (defmethod area :around ((f t))
    (let ((result (call-next-method)))
    (format t "Area of ~A is ~A" f result)
    result))
nil
```


@1993 John Mylopoulos

26

Method Evaluation

START: Input is a list of applicable methods

AROUND STEP: Are there any uncalled around methods?
YES: Call the most specific; if body uses call-next-method, repeat this step, otherwise go to DONE

BEFORE STEP: Are there any uncalled before methods?
YES: Call the most specific; when body returns, repeat this step

PRIMARY STEP: Are there any uncalled primary methods?
YES: Call the most specific; if body uses call-next-method, repeat this step; return value to caller

AFTER STEP: Are there any uncalled after methods?
YES: Call the least specific; when body returns, repeat this step

DONE: Return control and values to caller

@1993 John Mylopoulos

27

Generic Functions and `defgeneric`

Every method defines automatically a generic function with the same name

Generic functions can also be defined explicitly through defgeneric

```
(defgeneric area (f figure)
     (:documentation "computes the area of a figure")
```

or

```
(defgeneric area (f figure &optional failure-mode))
     (:documentation "computes the area of a figure;
          if there is no method for computing the area
          of a figure and the failure-mode is  :no-error
          returns NIL else it returns a message stating
          the type of figure")
```


@1993 John Mylopoulos