CLOS: The basic parts & reading lisp

Michal Atlas

November 29, 2023

A Small tangent before we start, I swear it's relevant

What's in a symbol?

So, what exactly is CLOS?

Result of a long history of Object Systems

A part of the language?

A part of the ~~language?~~ standard

What even is part of the language

```
(print (macroexpand '(defun fun (x y) x)))
```

```
(PROGN
 (EVAL-WHEN (:COMPILE-TOPLEVEL) (SB-C:%COMPILER-DEFUN 'FUN T NIL NIL))
 (SB-IMPL::%DEFUN 'FUN
                  (SB-INT:NAMED-LAMBDA FUN
                      (X Y)
                    (BLOCK FUN X))))
```

What even is part of the language

```
(print (macroexpand '(defvar x 20)))
```

```
(PROGN
 (EVAL-WHEN (:COMPILE-TOPLEVEL) (SB-IMPL::%COMPILER-DEFVAR 'X))
 (SB-IMPL::%DEFVAR 'X (SB-C:SOURCE-LOCATION) '20))
```

Nothing much

```
;; In ECL
(print (macroexpand '(defvar x 20)))

(LOCALLY
  (DECLARE (SPECIAL X))
  (SI:*MAKE-SPECIAL 'X)
  (UNLESS (BOUNDP 'X) (SETQ X 20))
  (PROGN (EXT:OPTIONAL-ANNOTATION 'X 'EXT:LOCATION '(DEFVAR X) 'NIL) NIL NI
  (EVAL-WHEN (:COMPILE-TOPLEVEL) (SI::REGISTER-GLOBAL 'X))
  'X)
```

defclass

defclass

```
(defclass myclass () ())
```

defclass describe

```
(describe 'myclass)

COMMON-LISP-USER::MYCLASS
  [symbol]

MYCLASS names the standard-class #<STANDARD-CLASS COMMON-LISP-USER::MYCLASS
  Class precedence-list: MYCLASS, STANDARD-OBJECT, SB-PCL::SLOT-OBJECT,
                         T
  Direct superclasses: STANDARD-OBJECT
  No subclasses.
  No direct slots.
```

Classes are associated to but not resolved from symbols

```
(print (find-class 'myclass))

(print (boundp 'myclass))
```

```
#<STANDARD-CLASS COMMON-LISP-USER::MYCLASS>
NIL
```

slots

```
(defclass myclass ()
  (width height))
```

instantiation

```
(defparameter x (make-instance 'myclass))
(describe x)

#<MYCLASS {10038EA623}>
  [standard-object]

Slots with :INSTANCE allocation:
  WIDTH                          = #<unbound slot>
  HEIGHT                         = #<unbound slot>
```

slot assignment

```
; (slot-value x 'width)
(setf (slot-value x 'width) 20)


(print (slot-value x 'width))
```

```
20
```

A tangent about grammars

slots

```
(defclass myclass ()
  ((width :accessor width)
   height))


(setf (width x) 20)
(print (width x))
```

```
20
```

slots initarg

```
(defclass myclass ()
  ((width :accessor width :initarg :width)
   height))


(defparameter x (make-instance 'myclass :width 20))
(print (width x))
```

```
20
```

Did you catch that?

Redefining the class was defined in-place behaviour

other slot options

```
(defclass myclass ()
  ((slot-name
    :reader symbol1
    :reader symbol2
    :writer symbolw
    :accessor symbola
    :initarg :foo
    :initform '(some calculation)
    :allocation :instance ; or :class
    :type number
    :documentation "")))
```

Superclasses

Simple inheritance

```
(defclass patriot ()
  ((name :reader cry)
   (warcry :initform "For Glory" :reader cry)))

(defclass citizen (patriot) ())

(print (cry (make-instance 'citizen)))


"For Glory"
```

Conflicting inheritance

```
(defclass patriot ()
  ((warcry :initform "For Glory" :reader cry)
   (patriotism :initform 200)))

(defclass coward ()
  ((warcry :initform "For Money" :reader cry)
   (level :initform 'total)))

(defclass patriotic-coward (patriot coward) ())
(print (cry (make-instance 'patriotic-coward)))


(defclass cowardly-patriot (coward patriot) ())
(print (cry (make-instance 'cowardly-patriot)))
```

Changing classes

```
(setf jeff (make-instance 'cowardly-patriot))
(describe jeff)

#<COWARDLY-PATRIOT {1003F5B063}>
  [standard-object]

Slots with :INSTANCE allocation:
  WARCRY                         = "For Money"
  PATRIOTISM                     = 200
  LEVEL                          = TOTAL
```

Resolve

```
(change-class jeff 'patriot)
(describe jeff)

#<PATRIOT {1003F5B063}>
  [standard-object]

Slots with :INSTANCE allocation:
  WARCRY                         = "For Money"
  PATRIOTISM                     = 200
```

Add an instance allocated slot

```
(defclass patriot ()
  ((warcry :initform "For Glory" :reader cry)
   (group-motto :allocation :class :initform "God Save the Queen")))

(defclass coward ()
  ((warcry :initform "For Money" :reader cry)
   (group-motto :allocation :class :initform "God Save me")))
```

Back to jeff

```
(describe jeff)

#<PATRIOT {1003F5B063}>
  [standard-object]

Slots with :CLASS allocation:
  GROUP-MOTTO                    = "God Save the Queen"
Slots with :INSTANCE allocation:
  WARCRY                         = "For Money"
```

And changing him back

```
(change-class jeff 'coward)
(describe jeff)

#<COWARD {1003F5B063}>
  [standard-object]

Slots with :CLASS allocation:
  GROUP-MOTTO                    = "God Save me"
Slots with :INSTANCE allocation:
  WARCRY                         = "For Money"
```

Infecting the patriots

```
(setf (slot-value jeff 'group-motto) "Glory to cowards")

(describe (make-instance 'coward))

#<COWARD {10041586F3}>
  [standard-object]

Slots with :CLASS allocation:
  GROUP-MOTTO                    = "Glory to cowards"
Slots with :INSTANCE allocation:
  WARCRY                         = "For Money"
```

Class precedence-list: a.k.a. What about the diamond problem?

```
(require 'closer-mop)

(print (closer-mop:class-precedence-list (find-class 'cowardly-patriot)))
(print (closer-mop:class-precedence-list (find-class 'patriotic-coward)))
```

```
(#<STANDARD-CLASS COMMON-LISP-USER::COWARDLY-PATRIOT>
 #<STANDARD-CLASS COMMON-LISP-USER::COWARD>
 #<STANDARD-CLASS COMMON-LISP-USER::PATRIOT>
 #<STANDARD-CLASS COMMON-LISP:STANDARD-OBJECT>
 #<SB-PCL::SLOT-CLASS SB-PCL::SLOT-OBJECT> #<SB-PCL:SYSTEM-CLASS COMMON-LISP:T>)
(#<STANDARD-CLASS COMMON-LISP-USER::PATRIOTIC-COWARD>
 #<STANDARD-CLASS COMMON-LISP-USER::PATRIOT>
 #<STANDARD-CLASS COMMON-LISP-USER::COWARD>
 #<STANDARD-CLASS COMMON-LISP:STANDARD-OBJECT>
 #<SB-PCL::SLOT-CLASS SB-PCL::SLOT-OBJECT> #<SB-PCL:SYSTEM-CLASS COMMON-LISP:T>)
```

Multimethods

what's a congruent lambda-list???

defgeneric

```
(defgeneric name (lambda-list))
```

defmethod

```
(defmethod name (x)           (print "Jeff"))
(defmethod name ((x patriot)) (print "Harrison Ford"))
(defmethod name ((x coward))  (print "Chicken Little"))
```

defmethod description

```
(describe 'name)

COMMON-LISP-USER::NAME
  [symbol]

NAME names a generic function:
  Lambda-list: (LAMBDA-LIST)
  Derived type: (FUNCTION (T) *)
  Method-combination: STANDARD
  Methods:
    (NAME (COWARD))
    (NAME (PATRIOT))
    (NAME (T))
    (NAME :AROUND (COWARD))
```

defmethod usage

```
(name (make-instance 'patriot))
```

```
"Hello there"
"Harrison Ford"
```

:before method

```
(defmethod name :before ((x coward)) (print "Arghh, "))

(name (make-instance 'coward))


"START AROUND"
"Arghh, "
"Hello there"
"Chicken Little"
"General Kenobi"
"END AROUND"
```

:after method

```
(defmethod name :after ((x coward)) (print "General Kenobi"))

(name (make-instance 'coward))
```

```
"START AROUND"
"Arghh, "
"Hello there"
"Chicken Little"
"General Kenobi"
"END AROUND"
```

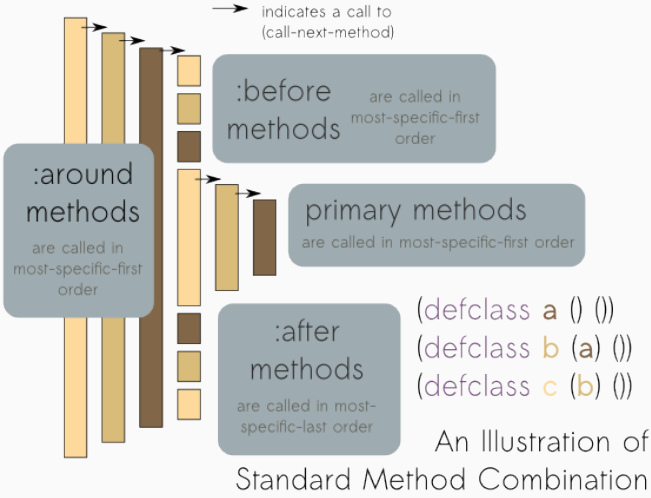
:before method

```
(defmethod name :before (x) (print "Hello there"))

(name (make-instance 'coward))
```

```
"START AROUND"
"Arghh, "
"Hello there"
"Chicken Little"
"General Kenobi"
"END AROUND"
```

the whole


indicates a call to
(call-next-method)
:before
methods
are called in
most-specific-first
order
:around
methods
are called in
most-specific-first
order
primary methods
are called in most-specific-first order
:after
methods
are called in most-
specific-last order
(defclass a () ())
(defclass b (a) ())
(defclass c (b) ())
An Illustration of
Standard Method Combination

:around methods

```
(defmethod name ((x coward))
  (print "START GENERAL PRIMARY")
  (call-next-method)
  (print "END GENERAL PRIMARY"))

(defmethod name :around ((x coward))
  (print "START AROUND")
  (call-next-method)
  (print "END AROUND"))

(name (make-instance 'coward))
```

Results

```
"START AROUND"
"Arghh, "
"Hello there"
"START GENERAL PRIMARY"
"Jeff"
"END GENERAL PRIMARY"
"General Kenobi"
"END AROUND"
```

trace

```
(trace name :methods t)

(name (make-instance 'patriot))

0: (NAME #<PATRIOT {1008DF6B33}>)
  1: ((SB-PCL::COMBINED-METHOD NAME) #<PATRIOT {1008DF6B33}>)
    2: ((METHOD NAME :BEFORE (T)) #<PATRIOT {1008DF6B33}>)
    2: (METHOD NAME :BEFORE (T)) returned "Hello there"
    2: ((METHOD NAME (PATRIOT)) #<PATRIOT {1008DF6B33}>)
    2: (METHOD NAME (PATRIOT)) returned "Harrison Ford"
  1: (SB-PCL::COMBINED-METHOD NAME) returned "Harrison Ford"
0: NAME returned "Harrison Ford"
```

bigtrace

```
0: (NAME #<COWARD {1008F18E73}>)
  1: ((METHOD NAME :AROUND (COWARD)) #<COWARD {1008F18E73}>)
    2: ((SB-PCL::COMBINED-METHOD NAME) #<COWARD {1008F18E73}>)
      3: ((METHOD NAME :BEFORE (COWARD)) #<COWARD {1008F18E73}>)
      3: (METHOD NAME :BEFORE (COWARD)) returned "Arghh, "
      3: ((METHOD NAME :BEFORE (T)) #<COWARD {1008F18E73}>)
      3: (METHOD NAME :BEFORE (T)) returned "Hello there"
      3: ((METHOD NAME (COWARD)) #<COWARD {1008F18E73}>)
        4: ((METHOD NAME (T)) #<COWARD {1008F18E73}>)
        4: (METHOD NAME (T)) returned "Jeff"
      3: (METHOD NAME (COWARD)) returned "END GENERAL PRIMARY"
      3: ((METHOD NAME :AFTER (COWARD)) #<COWARD {1008F18E73}>)
      3: (METHOD NAME :AFTER (COWARD)) returned "General Kenobi"
    2: (SB-PCL::COMBINED-METHOD NAME) returned "END GENERAL PRIMARY"
  1: (METHOD NAME :AROUND (COWARD)) returned "END AROUND"
0: NAME returned "END AROUND"
```

Accumulating a result from methods

```
(defmethod wage ((x bureaucrat)) (+ 10000 (call-next-method)))
(defmethod wage ((x soldier)) (+ 100 (call-next-method)))

(defclass steve (bureaucrat soldier) ())

(print (wage (make-instance 'steve)))
```

```
10100
```

Method combinations

```
(defgeneric wage-sum1 (x) (:method-combination +))
(defmethod wage-sum1 + ((x bureaucrat)) 10000)
(defmethod wage-sum1 + ((x soldier)) 100)

(defclass steve (bureaucrat soldier) ())

(print (wage-sum1 (make-instance 'steve)))
```

```
10100
```

Another syntax for methods

```
(defgeneric wage (x))
(defmethod wage ((x coward)) 20)

(defgeneric wage (x)
  (:method ((x coward)) 20))
```

Method combinations

```
(defgeneric wage-sum (x)
  (:method-combination +)
  (:method + ((x bureaucrat)) 10000)
  (:method + ((x soldier)) 100))

(defclass steve (bureaucrat soldier) ())

(print (wage-sum (make-instance 'steve)))
```

```
10100
```

Method combinations

```
(defgeneric wage-list (x)
  (:method-combination list)
  (:method list ((x bureaucrat)) 10000)
  (:method list ((x soldier)) 100))

(defclass steve (bureaucrat soldier) ())

(print (wage-list (make-instance 'steve)))
```

```
(10000 100)
```

Multimethods

update-instance-for-different-class

Thanks for listening