CLOS: the Common Lisp Object System

Christopher Mark Gore

http://www.cgore.com

cgore@cgore.com

@cgore

Thursday, February 6, AD 2014

Getting Started with Common Lisp

1. Install Linux.

   ```
   http://aptosid.com
   ```

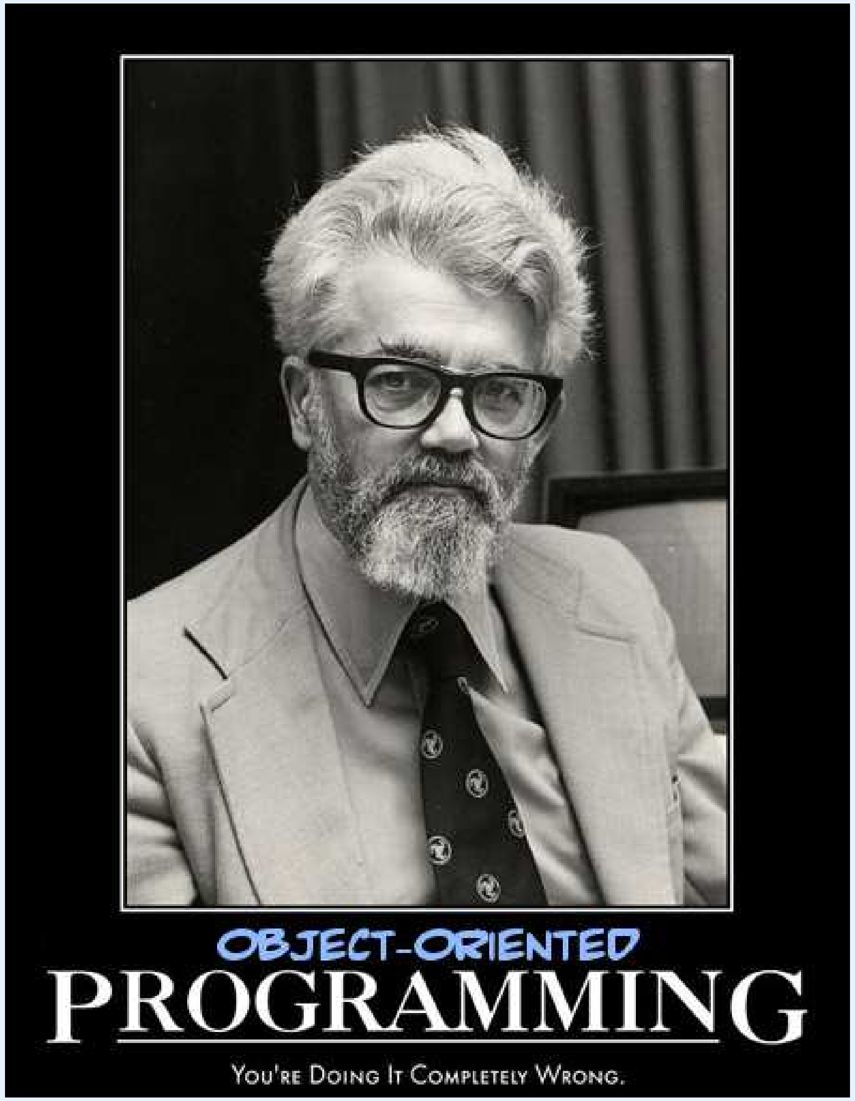
2. Install SBCL and some libraries.

   ```
   apt-get install sbcl{,-doc,-source} \
   cl-{asdf,cffi}
   ```

3. Install Emacs and SLIME *(Not strictly required.)*

   ```
   apt-get install emacs{,-goodies-el} cl-swank \
   cl-swank slime common-lisp-controller
   ```

$\textbf{Lisp} + \textbf{OOP} > \textbf{OOP} - \textbf{Lisp}$



Lisp $+$ OOP $>$ Lisp?

[Opinionated opinion:] Java or C++ style OOP doesn't help if you already have Lisp, and probably hurts, but the CLOS does help if you have the right sort of problem.

- Lisp is good at modeling computation.
- Functional programming is good at modeling verbs,
- Object-oriented programming is good at modeling nouns.
- CLOS allows FP for verbs and OOP for nouns to interact easily, with neither being the ***King Of All The Words***. (Cf. Steve Yegge's essay, *Execution in the Kingdom of Nouns*, `http://steve-yegge.blogspot.com/2006/03/execution-in-kingdom-of-nouns.html`).

OOP isn't *The* Answer

OOP is a useful tool, but it isn't the final solution to all things programming. It won't solve world hunger, but it does solve a restricted subset of the problem. Other things that are sometimes useful tools, but aren't ***the*** answer:

- FP,
- Lack of side effects,
- Unit tests/TDD/BDD/**EXTREME PROGRAMMING!**,
- Type systems,
- Monads,
- Your favorite thing in programming,
- ~~Lisp.~~ *[Yes it is.]*

Ancient History

Flavors (before 1982) worked on MIT Lisp Machines. It introduced the concepts of multiple inheritance and mixins. *(for lisp?)* It used a message-passing paradigm similar to Smalltalk (before 1972).

LOOPS worked on Xerox's Interlisp-D.

CommonLoops got LOOPS working in Common Lisp.

New Flavors (before 1985) introduced generic functions. *(for lisp?)*

Portable CommonLoops (PCL) eventually developed into the first implementation of CLOS.

Nouns and Verbs

Nouns are how our brain works with things.

*The **cat** was asleep in the **hallway**.*

```
cat[27].goToSleepInLocation(hallway[3]);
```

Verbs are how our brain works with actions.

*He **murdered** her in cold blood!*

```
(with-person (the-man) (murder the-woman))
```

Other Fun Parts of Speech . . .

Adjectives describe nouns.

*The **big**, **old**, **yellow** house burned to the ground.*

```
<house size="big" age="old" color="yellow"/>
```

Pronouns *(anaphors)* are shortcuts for nouns.

***We** walked down the street to meet **him**.*

```
(a?if him (person-to-meet?)  (go-to-meet him))
```

. . . Other Fun Parts of Speech

Adverbs change verbs.

He ***quickly*** *ran down the street.*

Prepositions links nouns and pronouns to other words.

The book is ***beneath*** *the table.*

Conjunctions link words, phrases, and clauses.

I ate the pizza ***and*** *the pasta.*

```
int i = 12; i++;
v = [1, 2, 14]
```

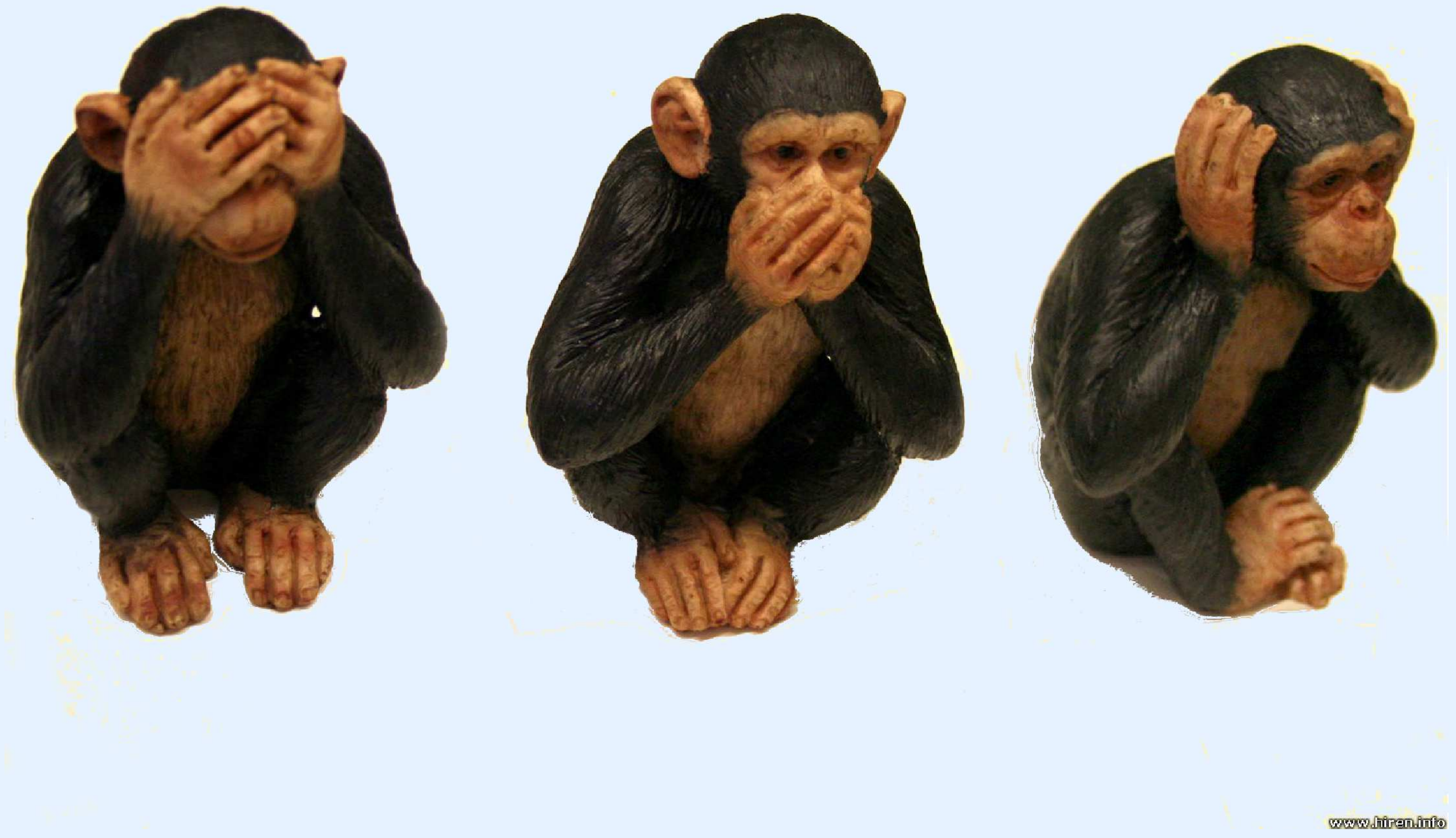
Interjections convey emotion.

Hey! *Put that down!*

```
i = 0x5f3759df - (i >> 1); // wtf?
```

OOP: How Classes See Their Methods

FP: How Functions See Their Data



The Basic Components in CLOS

Classes model nouns.

Instances are specific occurrences of nouns.

Generics model verbs.

Methods implement generics for specific classes.

The first two, classes, and instances, work as expected from any other normal OOP language. Generics and methods work quite differently though.

DEFCLASS

We define new classes with the `defclass` macro.

```
(defclass class-name (superclass-names) (slots))
```

Some examples:

```
(defclass point () (x y))
(defclass shape () ()) ; An abstract base class.
(defclass rectangle (shape) (p q))
(defclass circle (shape) (center radius))
```

We typically want to provide more for the slot definitions.

```
(defclass better-point ()
          ((x :accessor x :initarg :x :initform 0.0
              :type float)
           (y :accessor y :initarg :y :initform 0.0
              :type float)))
```

DEFCLASS Slot Definition Options

All of these are optional. This is ***not a hash map***, and some allow for multiple definitions.

:reader Defines a default getter for the slot.

:writer Defines a default setter for the slot.

:accessor Defines both a default getter and a default setter.

:allocation Either `:instance` for per-instance slots (this is the default) or `:class` for per-class slots.

:initarg This is a symbol to specify an initial value when you call `make-instance`.

:initform This is the default value at instantiation.

:type This can be any valid type specifier.

:documentation *But I thought code documents itself?*

MAKE-INSTANCE

We create new instances of a class with `make-instance`. *(The following examples assume more thorough slot definitions.)*

```
(let* ((o (make-instance 'point)) ; Defaults to (0,0)
       (p (make-instance 'point :x 1.0 :y 12.5))
       (q (make-instance 'point :x 5.0 :y 10.0))
       (r (make-instance 'rectangle :p p :q q))
       (c (make-instance 'circle
                         :center o :radius 12.5)))
  (x p) ; Returns 1.0
  ... Do more stuff ... )
```

DEFGENERIC

We define new generics with `defgeneric`.

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

Some examples:

```
(defgeneric min-x (thing))
(defgeneric max-x (thing))
(defgeneric min-y (thing))
(defgeneric max-y (thing))
(defgeneric height (thing))
(defgeneric width (thing))
(defgeneric area (thing))
```

These define the general layout of a set of methods all with the same name. *(SBCL will implicitly create them for you, with a warning.)*

DEFMETHOD

We define new methods with `defmethod`.

```
(defmethod min–x ((r rectange))
  (min (x (p r)) (x (q r))))
```

Implement `max-x`, `min-y`, and `max-y` in a similar manner.

```
(defmethod height ((r rectangle))
  (– (max–y r) (min–y r))
(defmethod width ((r rectangle))
  (– (max–x r) (min–x r)))
(defmethod area ((c circle))
  (* pi (expt (radius c) 2)))
(defmethod area ((r rectangle))
  (* (height r) (width r)))
```

Multiple Inheritance . . .

CLOS supports multiple inheritance, like C++ or Python. Java and Ruby *don't* have this, and I miss it when it isn't there.

```
(defclass animal () ())

(defclass swimming-animal (animal) ())
(defclass fish (swimming-animal) ())

(defclass winged-animal (animal) ())
(defclass bird (winged-animal) ())
(defclass penguin (bird swimming-animal) ())

(defclass mammal (animal) ())
(defclass whale (swimming-animal mammal) ())
(defclass bat (winged-animal mammal) ())
```

... Multiple Inheritance

```
(defclass device () ())

(defclass computer (device) ())
(defclass desktop (computer) ())
(defclass laptop (computer) ())

(defclass printer (device) ())
(defclass scanner (device) ())
(defclass fax (device) ())

(defclass all-in-one (printer scanner fax) ())
```

Multiple Dispatch: OOP Without the Self

"The self is a relation which relates itself to its own self,
or it is that in the relation
that the relation relates itself to its own self;
the self is not the relation
but that the relation relates itself to its own self."

— Søren Kierkegaard.

Single dispatch OOP assumes there is some special thing, the `self` or `this`. You can do this in CLOS too, by only specializing on the first argument to the methods, but then you are missing one of the best features of the CLOS.

```
(defmethod foo ((this square) x y) ...)

my_square.foo(x, y); // But what about x and y?
```

Let's Play a Game . . .

We can specialize on the classes of ***every argument***.

```
(defclass rock      () ())
(defclass paper     () ())
(defclass scissors  () ())
(defclass lizard    () ())
(defclass spock     () ())

(defgeneric play (a b))
(defmethod play (a b)
  (play b a)) ; Switch if no match.

(defmethod play ((a rock)     (b rock))     'tie)
(defmethod play ((a paper)    (b paper))    'tie)
(defmethod play ((a scissors) (b scissors)) 'tie)
(defmethod play ((a lizard)   (b lizard))   'tie)
(defmethod play ((a spock)    (b spock))    'tie)
```

. . . Let’s Play a Game

```
(defmethod play ((r  rock)      (sz scissors)) 'rock)
(defmethod play ((p  paper)     (sz scissors)) 'scissors)
(defmethod play ((r  rock)      (p  paper))    'paper)
(defmethod play ((r  rock)      (l  lizard))   'rock)
(defmethod play ((l  lizard)    (sp spock))    'spock)
(defmethod play ((sp spock)     (sz scissors)) 'spock)
(defmethod play ((sz scissors)  (l  lizard))   'scissors)
(defmethod play ((l  lizard)    (p  paper))    'lizard)
(defmethod play ((p  paper)     (sp spock))    'paper)
(defmethod play ((sp spock)     (r  rock))     'spock)
```

No Enforced Encapsulation

```
(defclass sphere ()
  ((x :initarg :x)
   (y :initarg :y)
   (z :initarg :z)
   (r :initarg :r)) ; No accessors defined!
(let ((s1 (make-instance 'sphere
               :x 12.66 :y 16.07 :z 22.01 :r 14.70)))
  (setf (slot-value s1 'radius) ; That was easy.
        (* 2.7 (slot-value s1 'x))))
```

Lambdas in Slots

You won't typically do this for code you write, but the code your code writes might like to do something similar.

```
(defclass j ()
  ((x :accessor x :initarg :x :initform 0.0 :type float)
   (y :accessor y :initarg :y :initform 0.0 :type float)
   (f :initform (lambda (this z)
                  (expt (+ (x this) (y this)) z)))))
(defgeneric f (this z))
(defmethod f ((this j) z)
  (funcall (slot-value this 'f) this z))
(let ((j1 (make-instance 'j :x 12 :y 33)))
  (f j1 7)) ; Returns 373,669,453,125.
```

MOP: The Meta Object Protocol

The CLOS is implemented in it.

It’s crazy!

Perhaps another day.

Questions?