Brief Guide to CLOS

Written by Jeff Dalton, University of Edinburgh, <J.Dalton@ed.ac.uk>
Modified by Bruno Haible <haible@ma2s2.mathematik.uni-karlsruhe.de>

Contents:

[Think of upper case in things like DEFCLASS as literals -- I don't mean to imply that actual upper case would be used. "Thing*" means zero or more occurrences of thing; "thing+" means one or more. Curly brackets { and } are used for grouping, as in {a b}+, which means a b, a b a b, a b a b a b, etc.]

1. Defining classes.

You define a class with DEFCLASS:

```
(DEFCLASS class-name (superclass-name*)
  (slot-description*)
  class-option*)
```

For simple things, forget about class options.

A slot-description has the form (slot-name slot-option*), where each option is a keyword followed by a name, expression, or whatever. The most useful slot options are

```
:ACCESSOR function-name
:INITFORM expression
:INITARG symbol
```

(Initargs are usually keywords.)

DEFCLASS is similar to DEFSTRUCT. The syntax is a bit different, and you have more control over what things are called. For instance, consider the DEFSTRUCT:

```
(defstruct person
  (name 'bill)
  (age 10))
```

DEFSTRUCT would automatically define slots with expressions to compute default initial values, access-functions like PERSON-NAME to get and set slot values, and a MAKE-PERSON that took keyword initialization arguments (initargs) as in

```
(make-person :name 'george :age 12)
```

A DEFCLASS that provided similar access functions, etc, would be:

```
(defclass person ()
```

```
     ((name :accessor person-name
            :initform 'bill
            :initarg :name)
      (age :accessor person-age
           :initform 10
           :initarg :age)))
```

Note that DEFCLASS lets you control what things are called. For instance, you don't have to call the accessor PERSON-NAME. You could call it NAME.

In general, you should pick names that make sense for a group of related classes rather than rigidly following the DEFSTRUCT conventions.

You do not have to provide all options for every slot. Maybe you don't want it to be possible to initialize a slot when calling MAKE-INSTANCE (for which see below). In that case, don't provide an :INITARG. Or maybe there isn't a meaningful default value. (Perhaps the meaningful values will always be specified by a subclass.) In that case, no :INITFORM.

Note that classes are objects. To get the class object from its name, use (FIND-CLASS name). Ordinarily, you won't need to do this.

2. Instances

You can make an instance of a class with MAKE-INSTANCE. It's similar to the MAKE-x functions defined by DEFSTRUCT but lets you pass the class to instantiate as an argument:

```
   (MAKE-INSTANCE class {initarg value}*)
```

Instead of the class object itself, you can use its name. For example:

```
   (make-instance 'person :age 100)
```

This person object would have age 100 and name BILL, the default.

It's often a good idea to define your own constructor functions, rather than call MAKE-INSTANCE directly, because you can hide implementation details and don't have to use keyword parameters for everything. For instance, you might want to define

```
  (defun make-person (name age)
    (make-instance 'person :name name :age age))
```

if you wanted the name and age to be required, positional parameters, rather than keyword parameters.

The accessor functions can be used to get and set slot values:

```
   <cl> (setq p1 (make-instance 'person :name 'jill :age 100))
   #<person @ #x7bf826>

   <cl> (person-name p1)
   jill

   <cl> (person-age p1)
```

```
   100

   <cl> (setf (person-age p1) 101)
   101

   <cl> (person-age p1)
   101
```

Note that when you use DEFCLASS, the instances are printed using the #<...> notation, rather than as #s(person :name jill :age 100). But you can change the way instances are printed by defining methods on the generic function PRINT-OBJECT.

Slots can also be accessed by name using (SLOT-VALUE instance slot-name):

```
   <cl> (slot-value p1 'name)
   jill

   <cl> (setf (slot-value p1 'name) 'jillian)
   jillian

   <cl> (person-name p1)
   jillian
```

You can find out various things about an instance by calling DESCRIBE:

```
   <cl> (describe p1)
   #<person @ #x7bf826> is an instance of class
        #<clos:standard-class person @ #x7ad8ae>:
   The following slots have :INSTANCE allocation:
   age     101
   name    jillian
```

2. Inheritance of slot options

The class above had no superclass. That's why there was a "()" after "defclass person". Actually, this means it has one superclass: the class STANDARD-OBJECT.

When there are superclasses, a subclass can specify a slot that has already been specified for a superclass. When this happens, the information in slot options has to be combined. For the slot options listed above, either the option in the subclass overrides the one in the superclass or there is a union:

```
   :ACCESSOR  --  union
   :INITARG   --  union
   :INITFORM  --  overrides
```

This is what you should expect. The subclass can _change_ the default initial value by overriding the :INITFORM, and can add to the initargs and accessors.

However, the "union" for accessor is just a consequence of how generic functions work. If they can apply to instances of a class C, they can also apply to instances of subclasses of C.

(Accessor functions are generic. This may become clearer once generic functions are discussed, below.)

Here are some subclasses:

```
   <cl> (defclass teacher (person)
          ((subject :accessor teacher-subject
                    :initarg :subject)))
   #<clos:standard-class teacher @ #x7cf796>

   <cl> (defclass maths-teacher (teacher)
          ((subject :initform "Mathematics")))
   #<clos:standard-class maths-teacher @ #x7d94be>

   <cl> (setq p2 (make-instance 'maths-teacher
                    :name 'john
                    :age 34))
   #<maths-teacher @ #x7dcc66>

   <cl> (describe p2)
   #<maths-teacher @ #x7dcc66> is an instance of
        class #<clos:standard-class maths-teacher @ #x7d94be>:
    The following slots have :INSTANCE allocation:
    age        34
    name       john
    subject    "Mathematics"
```

Note that classes print like #<clos:standard-class maths-teacher @ #x7d94be>.
The #<...> notation usually has the form

```
   #<class-of-the-object ... more information ...>
```

So an instance of maths-teacher prints as #<MATHS-TEACHER ...>.
The notation for the classes above indicates that they are
instances of STANDARD-CLASS. DEFCLASS defines standard classes.
DEFSTRUCT defines structure classes.

4. Multiple inheritance

A class can have more than one superclass. With single inheritance
(one superclass), it's easy to order the superclasses from most to
least specific. This is the rule:

Rule 1: Each class is more specific than its superclasses.

In multiple inheritance this is harder. Suppose we have

```
  (defclass a (b c) ...)
```

Class A is more specific than B or C (for instances of A), but what if
something (an :INITFORM, or a method) is specified by B and C? Which
overrides the other? The rule in CLOS is that the superclasses listed
earlier are more specific than those listed later. So:

Rule 2: For a given class, superclasses listed earlier are more
        specific than those listed later.

These rules are used to compute a linear order for a class and all its
superclasses, from most specific to least specific. This order is the
"class precedence list" of the class.

The two rules are not always enough to determine a unique order, however, so CLOS has an algorithm for breaking ties. This ensures that all implementations always produce the same order, but it's usually considered a bad idea for programmers to rely on exactly what the order is. If the order for some superclasses is important, it can be expressed directly in the class definition.

5. Generic functions and methods

Generic function in CLOS are the closest thing to "messages". Instead of writing

```
(SEND instance operation-name arg*)
```

you write

```
(operation-name instance arg*)
```

The operations / messages are generic functions -- functions whose behavior can be defined for instances of particular classes by defining methods.

(DEFGENERIC function-name lambda-list) can be used to define a generic function. You don't have to call DEFGENERIC, however, because DEFMETHOD automatically defines the generic function if it has not been defined already. On the other hand, it's often a good idea to use DEFGENERIC as a declaration that an operation exists and has certain parameters.

Anyway, all of the interesting things happen in methods. A method is defined by:

```
(DEFMETHOD generic-function-name specialized-lambda-list
  form*)
```

This may look fairly cryptic, but compare it to DEFUN described in a similar way:

```
(DEFUN function-name lambda-list form*)
```

A "lambda list" is just a list of formal parameters, plus things like &OPTIONAL or &REST. It's because of such complications that we say "lambda-list" instead of "(parameter*)" when describing the syntax.

[I won't say anything about &OPTIONAL, &REST, or &KEY in methods. The rules are in CLtL, if you want to know them.]

So a normal function has a lambda list like (var1 var2 ...). A method has one in which each parameter can be "specialized" to a particular class. So it looks like:

```
((var1 class1) (var2 class2) ...)
```

The specializer is optional. Omitting it means that the method can apply to instances of any class, including classes that were not defined by DEFCLASS. For example:

```
(defmethod change-subject ((teach teacher) new-subject)
  (setf (teacher-subject teach) new-subject))
```

Here the new-subject could be any object. If you want to restrict

it, you might do something like:

```
(defmethod change-subject ((teach teacher) (new-subject string))
  (setf (teacher-subject teach) new-subject))
```

Or you could define classes of subjects.

Methods in "classical" object-oriented programming specialize only one parameter. In CLOS, you can specialize more than one. If you do, the method is sometimes called a multi-method.

A method defined for a class C overrides any method defined for a superclass of C. The method for C is "more specific" than the method for the superclass, because C is more specific that the classes it inherits from (eg, dog is more specific than animal).

For multi-methods, the determination of which method is more specific involves more than one parameter. The parameters are considered from left to right.

```
(defmethod test ((x number) (y number))
  '(num num))

(defmethod test ((i integer) (y number))
  '(int num))

(defmethod test ((x number) (j integer))
  '(num int))

(test 1 1)      =>  (int num), not (num int)
(test 1 1/2)    =>  (int num)
(test 1/2 1)    =>  (num int)
(test 1/2 1/2)  =>  (num num)
```

6. Method combination.

When more than one class defines a method for a generic function, and more than one method is applicable to a given set of arguments, the applicable methods are combined into a single "effective method". Each individual method definition is then only part of the definition of the effective method.

One kind of method combination is always supported by CLOS. It is called standard method combination. It is also possible to define new kinds of method combination. Standard method combination involves four kinds of methods:

* Primary methods form the main body of the effective method. Only the most specific primary method is called, but it can call the next most specific primary method by calling

  ```
  (call-next-method)
  ```

* :BEFORE methods are all called before the primary method, with the most specific :BEFORE method called first.

* :AFTER methods are all called after the primary method, with the most specific :AFTER method called last.

  * :AROUND methods run before the other methods. As with
    primary methods, only the most specific is called and the
    rest can be invoked by CALL-NEXT-METHOD. When the least
    specific :AROUND method calls CALL-NEXT-METHOD, what it
    calls is the combination of :BEFORE, :AFTER, and primary
    methods.

:BEFORE, :AFTER, and :AROUND methods are indicated by putting the
corresponding keyword as a qualifier in the method definition.
:BEFORE and :AFTER methods are the easiest to use, and a simple
example will show how they work:

```
(defclass food () ())

(defmethod cook :before ((f food))
  (print "A food is about to be cooked."))

(defmethod cook :after ((f food))
  (print "A food has been cooked."))

(defclass pie (food)
  ((filling :accessor pie-filling :initarg :filling :initform 'apple)))

(defmethod cook ((p pie))
  (print "Cooking a pie")
  (setf (pie-filling p) (list 'cooked (pie-filling p))))

(defmethod cook :before ((p pie))
  (print "A pie is about to be cooked."))

(defmethod cook :after ((p pie))
  (print "A pie has been cooked."))

(setq pie-1 (make-instance 'pie :filling 'apple))
```

And now:

```
<cl> (cook pie-1)
"A pie is about to be cooked."
"A food is about to be cooked."
"Cooking a pie"
"A food has been cooked."
"A pie has been cooked."
(cooked apple)
```

7. Quick reference

Square brackets ("[" and "]") indicate optional elements. A vertical
bar ("|") indicates an alternative. Thing* means zero or more
occurrence of thing, thing+ means one or more occurrence. "{" and "}"
are used for grouping.

Defining a class:

```
(DEFCLASS class-name (superclass-name*)
  (slot-description*))
```

Slot descriptions:

```
(slot-name slot-option*)
```

```
   :ACCESSOR function-name
   :INITFORM expression
   :INITARG keyword-symbol

Making instances:

   (MAKE-INSTANCE class {initarg value}*)

Method definitions:

   (DEFMETHOD generic-function-name [qualifier] specialized-lambda-list
     form*)

where

   generic-function-name is a symbol;

   qualifier is :BEFORE, :AFTER, :AROUND, or else omitted;

   specialized-lambda-list is
      ( {variable | (variable class-name)}* )

Some functions:

   (DESCRIBE instance)
   (DESCRIBE class)

   (FIND-CLASS class-name) -> class

   (CLASS-NAME class) -> symbol

   (CLASS-PRECEDENCE-LIST class) -> list of classes

   (CLASS-DIRECT-SUPERCLASSES class) -> list of classes

   (CLASS-DIRECT-SUBCLASSES class) -> list of classes
```