



Dr. Salvador Godoy Calderón



## Mapa...

*Fortran (John Backus, 1957)*

*LISP (McCarthy, 1958)*

*Algol 60 (1958 IAL, 1960)*

*Pascal (Niklaus Wirth, 1968)*

*C (Dennis Ritchie, 1969)*

*Modula (Niklaus Wirth, 1970)*

*Alan Kay, 1978, concepto de OO*

*Smalltalk (Alan Kay, 1980)*

*C++ (Bjarne Stroustrup, 1979)*



*Alan Kay:*

*“Yo inventé el término **orientado a objetos** y puedo decirles que C++ no era lo que yo tenía en mente ...”*

*Dr. Salvador Godoy Calderón*

## Sorpasivamente...

Aunque *LISP* es anterior al desarrollo de la orientación a objetos, el estándar *Common Lisp* fue creado resumiendo diversos modelos de OO, particularmente el modelo de *Smalltalk*...

El modelo completo de objetos se integró en la extensión llamada *CLOS* que es parte del estándar de *Common Lisp* desde el año 1994...

El modelo OO de *CLOS* es único pero no es bien aceptado por los partidarios de otros modelos como C++ o Java...

*Dr. Salvador Godoy Calderón*

### ***Los argumentos...***

Diferentes lenguajes establecen lo que consideran fundamental en un modelo OO:

- ◆ Encapsulamiento “estricto”... (*C++, Eiffel, Java*)
- ◆ Paso de mensajes... (*Actor, C#*)
- ◆ Prototipos... (*JavaScript*)

*CLOS (Common Lisp Object System)* es una extensión para orientación a objetos basada en las siguientes características:

- ◆ Funciones genéricas
- ◆ Herencia múltiple

*Dr. Salvador Godoy Calderón*

### ***Lo importante de entender...***

En *CLOS*:

- ◆ Se definen clases e instancias de cada clase (obetos).
- ◆ Se pueden definir jerarquías de clases y las instancias correspondientes heredan elementos definidos en clases ancestro...
- ◆ Existen variables compartidas por todos los elementos de una clase, así como variables específicas a alguna instancia...
- ◆ Los métodos NO PERTENECEN a las clases...

*Dr. Salvador Godoy Calderón*



## Funciones genéricas y métodos...

### Funciones genéricas...

Una función genérica es la descripción abstracta de una operación, especificando su nombre y lista de argumentos, pero no su implementación...

```
>> (defgeneric visualiza (dato) )  
#<Standard-Generic-Function VISUALIZA>
```

Estas funciones pueden aceptar argumentos de cualquier tipo y debe existir una definición de método para cada tipo posible de argumento...

## Métodos...

Los métodos especifican el comportamiento de una función genérica según el tipo de datos de sus argumentos...

```
>> (defgeneric visualiza (dato))
#<Standard-Generic-Function VISUALIZA>
```

```
(defmethod visualiza ((dato number))
  ... )

(defmethod visualiza ((dato string))
  ... )

(defmethod visualiza ((dato list))
  ... )
```

Dr. Salvador Godoy Calderón

## Diferencia...

La principal diferencia de un modelo OO **basado en funciones genéricas** y uno con paso de mensajes, es el hecho de que los métodos no requieren ser parte de alguna clase...

A partir de la *especialización* del argumento, en los métodos, la función genérica decide cuál método invocar en cada caso (*despachar la función*)...

Existen diversas formas de especializar un argumento y, por supuesto, diversos parámetros a especializar...

Dr. Salvador Godoy Calderón

*Ejemplo...*

```
>> (defgeneric dato (d))
#<Standard-Generic-Function DATO>

>> (defmethod dato ((d number))
      (format t "Recibí un número = ~a~% " d) )
#<Standard-Method DATO (NUMBER) {1003F6BB83}>

>> (defmethod dato ((d string))
      (format t "Recibí una cadena = ~a~% " d) )
#<Standard-Method DATO (STRING) {100401C2A3}>
```

Dr. Salvador Godoy Calderón

*Ejemplo...*

```
>> (dato 45.2)
Recibí un número = 45.2
NIL

>> (dato "hola" )
Recibí una cadena = hola
NIL

>> (dato '(a b c) )
debugger invoked on a SIMPLE-ERROR in thread
#<THREAD "main thread" RUNNING {1002A8AF63}>:
  There is no applicable method for the generic function
    #<STANDARD-GENERIC-FUNCTION DATO (2)>
when called with arguments ((A B C)).
```

Dr. Salvador Godoy Calderón

*Ejemplo...*

```
>> (defmethod dato ((d list))
      (format t "Recibí una lista = ~a~%" d) )
#<STANDARD-METHOD DATO (LIST) {10044FD303}>
```

```
>> (dato '(a b c d e) )
Recibí una lista = (A B C D E)
NIL
```

```
>> (dato nil )
Recibí una lista = NIL
NIL
```

Dr. Salvador Godoy Calderón

*Recomendación...*

Aunque una función genérica no tiene cuerpo, es recomendable colocar, por lo menos, la opción **:documentation** para describir el propósito de la función:

```
>> (defgeneric visualiza (dato)
      (:documentation
       "Permite vizualizar un dato de diferentes tipos"))
#<standard-generic-function visualiza>
```

Diferente formato que en las funciones normales:

```
>> (defun suma (dato)
      "Suma 1 al argumento..."
      (+ dato 1))
SUMA
```

Dr. Salvador Godoy Calderón

## Multidespacho...

Posiblemente, la característica más peculiar (y poderosa) del modelo CLOS es que soporta Multidespacho de métodos...

Ello significa que los métodos pueden especializarse para varios de sus argumentos o inclusive para una combinación de ellos...

La mayoría de otros modelos OO sopertan sólo Despacho simple (sobre su primer argumento)...

Dr. Salvador Godoy Calderón

## Ejemplo...

Función genérica inicial

`(defgeneric f (x y) )`

`(defmethod f ((x integer) y) 1)`

`(f 1 "hola") => 1`

← Despacho simple

`(defmethod f ((x integer) (y real)) (+ x y))`

`(f 1 2.0) => 3.0`

← Despacho Múltiple

Dr. Salvador Godoy Calderón



## Definición...

La definición de una clase es muy simple:

```
>> (defclass <nombre clase> (<nombre superclases>)
           <lista de campos slots>)
```

Se puede usar cualquier símbolo para nombrar una clase y *Common Lisp* mantiene un espacio de nombres diferente para clases, variables y funciones...

```
(defclass cuentaBancaria() ...)
(defclass cuentaCheques (cuentaBancaria) ...)
(defclass cuentaAhorros (cuentaBancaria) ...)
```

*Creando objetos...*

Para crear una instancia de una clase usamos la función **make-instance** con argumento igual al nombre de la clase que se desea instanciar:

```
>> (defclass cuentaBancaria()
      (dueño
       saldo))
#<STANDARD-CLASS CUENTABANCARIA>

>> (setq MiCuenta (make-instance 'cuentaBancaria))
#<CUENTABANCARIA {1004C993D3}>
```

Dr. Salvador Godoy Calderón

*Objetos y sus ranuras...*

La función **slot-value** permite leer y escribir el valor de un campo en algún objeto particular:

```
>> (setq miCuenta (make-instance 'cuentaBancaria))
#<cuentaBancaria {1004C993D3}>

>> (setf (slot-value miCuenta 'dueño) "Salvador Godoy")
#<cuentaBancaria {100027830f2}>

>> (setf (slot-value miCuenta 'saldo) 45000.50)
#<cuentaBancaria {10003991ff6}>
```

Dr. Salvador Godoy Calderón

### *Inicialización de ranuras...*

Para no tener que inicializar campos por separado se pueden usar las opciones **:initarg** e **:initform** en la definición de una clase...

La opción **:initform** proporciona una expresión que será evaluada para dar valor inicial por omisión a un campo...

La opción **:initarg** permite indicar la llave con la que se dará valor inicial a un campo al ejecutar **make-instance**...

Dr. Salvador Godoy Calderón

### *Inicialización de ranuras...*

```
>> (defclass cuentaBancaria()
      ((dueño :initarg :cliente
              :initform " ")
       (saldo :initarg :capital
              :initform 0.0)))
#<standard-class cuentaBancaria>
```

```
>> (setq miCuenta (make-instance 'cuentaBancaria
                                    :cliente "Salvador Godoy"
                                    :capital 45000.50))
#<cuentaBancaria {10046D8893}>
```

Dr. Salvador Godoy Calderón

### *Funciones de acceso a ranuras...*

Es tradicional, en un modelo OO, limitar el acceso al contenido de una clase, especificando una interfaz con la cual se puede lograr dicho acceso...

Para cada campo en una clase, se puede definir una función de lectura y una función de escritura...

También se puede definir una única función de acceso a un campo y que sirve, tanto para lectura, como para escritura...

Dr. Salvador Godoy Calderón

### *Funciones de lectura y escritura...*

Resulta trivial definir una función de acceso por lectura a un campo de alguna clase...

```
(defun saldo? (cuenta)
  (slot-value cuenta 'saldo))
```

Pero, si se van a definir sub-clases de **cuentaBancaria**, conviene definir las funciones de acceso como funciones genéricas...

```
(defgeneric saldo? (cuenta))
(defmethod saldo? ((cuenta cuentaBancaria))
  (slot-value cuenta 'saldo))
```

Dr. Salvador Godoy Calderón

*Funciones de acceso genéricas...*

```
(defclass cuentaBancaria ()
  ((dueño :initarg :cliente
    :initform " "
    :reader dueño?)
   (saldo :initarg :capital
    :initform 0.0
    :reader saldo?))

(defgeneric saldo? (cuenta))
(defmethod saldo? ((cuenta cuentaBancaria))
  (slot-value cuenta 'saldo))

(defgeneric dueño? (cuenta))
(defmethod dueño? ((cuenta cuentaBancaria))
  (slot-value cuenta 'dueño))
```

Dr. Salvador Godoy Calderón

*Ejemplo de uso...*

```
>> (setq miCuenta (make-instance 'cuentaBancaria
                                     :cliente "Salvador Godoy"
                                     :capital 45000.50))
#<CUENTABANCARIA {10046D8893}>

>> (saldo? miCuenta)
45000.50

>> (dueño? miCuenta)
SALVADOR GODOY
```

Dr. Salvador Godoy Calderón

*El caso de las funciones de escritura...*

Para campos en los cuales se desea que el usuario de la clase pueda modificar su valor, se requiere una función de escritura de tipo **SETF...**

Estas funciones se comportan de la misma forma que **setf**, **aref** y **gethash...**

```
(defun (setf dueño) (nombre cuenta)
      (setf (slot-value cuenta 'dueño) nombre))
```

Dr. Salvador Godoy Calderón

*Sin embargo...*

Al igual que una función de acceso por lectura, seguramente queremos una función genérica...

```
(defgeneric (setf dueño) (cuenta nombre))

(defmethod (setf dueño) ((cuenta cuentaBancaria) nombre)
  (setf (slot-value cuenta 'dueño) nombre))
```

```
(defgeneric dueño? (cuenta))

(defmethod dueño? ((cuenta cuentaBancaria))
  (slot-value cuenta 'dueño))
```

Dr. Salvador Godoy Calderón

*La opción :accessor...*

Como es muy común requerir, tanto función de lectura, como de escritura para una ranura, también existe la opción **:accessor** que crea ambas funciones de forma automática...

```
(defclass cuentaBancaria()
  ((dueño :initarg :cliente
    :initform (error "Debe tener nombre")
    :accessor nombre-dueño)
   (saldo :initarg :capital
    :initform 0
    :accessor saldo-de-cuenta)) )
```

Dr. Salvador Godoy Calderón

*Ejemplo de uso...*

```
>> (setq miCuenta (make-instance 'cuentaBancaria
                                     :cliente "Salvador Godoy"
                                     :capital 45000.50))
#<CUENTABANCARIA {10046D8893}>

>> (nombre-dueño miCuenta)
SALVADOR GODOY
>> (setf (nombre-dueño miCuenta) "Juan Pérez")
JUAN PÉREZ

>> (saldo-de-cuenta miCuenta)
45000.50
>> (setf (saldo-de-cuenta miCuenta) 45.25)
45.25
```

Si Juan se va a quedar con mi cuenta, no se quedará con mi dinero !!!

Dr. Salvador Godoy Calderón

*Observaciones...*

Todo objeto puede ser descrito por el método **describe**, que indica el nombre y valor en cada uno de sus campos:

```
>> (defclass Class1 ()
      ((a :initform 1) (b :initform 2) (c :initform 3)))
#<STANDARD-CLASS CLASS1>

>> (setq c1 (make-instance 'Class1))
C1

>> (describe c1)
<CLASS1 {1003D52CE3}>
[standard-object]

Slots with :INSTANCE allocation:
A = 1
B = 2
C = 3
```

Dr. Salvador Godoy Calderón

*Para obsesivos...*

Una opción de “ocultar” el nombre de los campos es sobreescribir el método **describe** ...

```
>> (defmethod describe ((c cuentaBancaria))
      (format t "Cuenta bancaria con ~a pesos ~%"
              (slot-value c 'saldo)))
```

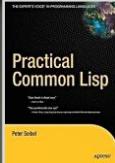
```
>> (describe miCuenta)
Cuenta bancaria con 45000.50 pesos
```

Dr. Salvador Godoy Calderón

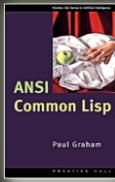


## Recursos adicionales...

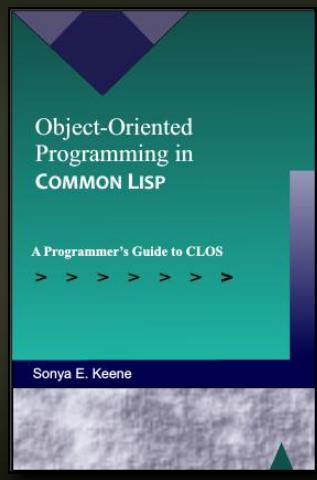
*Libros...*



Capítulos 16 y 17 en  
*Practical Common Lisp*  
de Peter Siebel



Capítulo 11 en  
*ANSI Common Lisp*  
de Paul Graham



*Object Oriented Programming in COMMON LISP*  
A Programmer's Guide to CLOS  
> > > > > >

*Object Oriented Programming in Common Lisp*  
*A programmers guide to CLOS*  
Sonya E. Keene (1989)

Dr. Salvador Godoy Calderón

*en la Web...*

*Fundamentals of CLOS*  
<http://cl-cookbook.sourceforge.net/clos-tutorial/>

*A Brief Guide to CLOS*  
<http://www.aiai.ed.ac.uk/~jeff/clos-guide.html>

*The Common Lisp Object System*  
<http://www.cs.northwestern.edu/academics/courses/325/readings/clos.php>

*Common Lisp Object System*  
<http://www.dreamsongs.com/CLOS.html>

Dr. Salvador Godoy Calderón

