

Fundamentos de lenguajes de programación

Robinson Duque, M.Eng, Ph.D

Universidad del Valle

robinson.duque@correounivalle.edu.co

Programa de Ingeniería de Sistemas
Escuela de Ingeniería de Sistemas y Computación



Este documento es una adaptación del material original de los profesores
Carlos Andres Delgado y Carlos Alberto Ramírez

- 1 Conceptos Fundamentales de la Programación Orientada a Objetos.
- 2 Sintaxis de la programación orientada a objetos
- 3 Ejemplos y Semántica
- 4 Implementación Conceptos Programación Orientada a objetos
 - Aspectos Generales
 - Una implementación simple
 - Objetos Planos

Objetos y Clases

- El mundo real puede ser visto como un conjunto de entidades que poseen un *estado* y un *comportamiento* que controla o es controlado por un estado.
- Por ejemplo, los gatos pueden comer, hacer ruido, saltar, y dormirse, y dichas actividades son controladas por el estado actual en el que se encuentren, incluyendo qué tan hambrientos o cansados estén.
- La programación orientada a objetos es una tecnología útil para resolver el problema de mantener la consistencia de los estados y asegurar que las variables que constituyen dichos estados sean actualizadas de una manera coordinada, a través de una interfaz.

Objetos y Clases

- En la programación orientada a objetos, un *objeto* consiste de *campos* (para almacenar estados) y *métodos* (para representar comportamientos) asociados que tienen acceso a los campos.
- La operación de llamado a un método es vista como el envío del nombre del método y sus argumentos al objeto a manera de *mensaje*.

Objetos y Clases

- Muchas veces es necesario trabajar con objetos con comportamientos similares.
- Para facilitar el *compartimiento* (del inglés *sharing*) de métodos, la programación orientada a objetos provee *clases*, las cuales son estructuras que especifican los campos y métodos de cada objeto.
- Los objetos son *instancias* de alguna clase.

Herencia

- A veces, se necesita definir una nueva clase, que tiene sólo unas pequeñas modificaciones con respecto a una clase existente (la adición o cambio de un método ó la adición o cambio de un campo).
- En este caso, se dice que la nueva clase *hereda de* o *extiende* la clase existente.

Polimorfismo

- Otra característica importante de los lenguajes orientados a objetos es el *polimorfismo*. Este concepto tiene varias acepciones.
- En este contexto (herencia o subclases) *polimorfismo* significa que una instancia de una subclase pueda hacer el rol de un objeto de su *superclase* (clase de la cual hereda) de modo que pueda ser usada en los mismos lugares que una instancia de su superclase.
- *Polimorfismo* también puede significar que un valor puede tener más de un tipo.

Programación Orientada a Objetos

En general, las características de un lenguaje orientado a objeto son:

- Los *objetos* encapsulan el comportamiento (métodos) y el estado (almacenado en los campos).
- Las *clases* agrupan los objetos que difieren sólo en su estado.
- La *herencia* permite a nuevas clases ser derivadas de algunas clases existente.
- El *polimorfismo* permite que un mismo tipo de mensaje pueda ser enviado a objetos de diferentes clases.

Sintaxis

- El lenguaje del curso debe ser extendido mediante reglas de producción que permitan la programación orientada o objetos.
- Un programa será ahora, una secuencia de declaraciones de clases seguida de una expresión:

$\langle \text{programa} \rangle \quad ::= \quad \{ \langle \text{class-decl} \rangle \}^* \langle \text{expresión} \rangle$

a-program (class-decls body)

$\langle \text{class-decl} \rangle \quad ::= \quad \text{class } \langle \text{identificador} \rangle \text{ extends } \langle \text{identificador} \rangle$
 $\{ \text{field}(\langle \text{identificador} \rangle) \}^* \{ \langle \text{method-decl} \rangle \}^*$

a-class-decl(class-name super-name
fields-ids method-decls)

$\langle \text{method-decl} \rangle \quad ::= \quad \text{method } \langle \text{identificador} \rangle \{ \{ \langle \text{identificador} \rangle \}^* (,) \}$
 $\langle \text{expresión} \rangle$

a-method-decl (method-name ids body)

Sintaxis

Se deben agregar reglas de producción para las expresiones, de modo que se puedan crear objetos, invocar métodos y hacer super llamados:

$\langle \text{expresión} \rangle ::= \text{new } \langle \text{identificador} \rangle (\{ \langle \text{expresión} \rangle \}^*(,))$

`new-object-exp (class-name rand)`

$\langle \text{expresión} \rangle ::= \text{send } \langle \text{expresión} \rangle \langle \text{identificador} \rangle (\{ \langle \text{expresión} \rangle \}^*(,))$

`method-app-exp (obj-exp method-name rand)`

$\langle \text{expresión} \rangle ::= \text{super } \langle \text{identificador} \rangle (\{ \langle \text{expresión} \rangle \}^*(,))$

`super-call-exp (method-name rand)`

Sintaxis

Se extiende el conjunto de valores expresados y denotados del lenguaje de la siguiente manera:

$$\begin{aligned}\text{Valor Denotado} &= \text{Ref(Valor Expresado)} \\ \text{Valor Expresado} &= \text{Número} + \text{ProcVal} + \text{Obj} + \\ &\quad \text{Lista(Valores Expresados)}\end{aligned}$$

Ejemplos y Semántica

Ejemplos:

```
class c1 extends object
  field i
  field j
  method initialize (x)
    begin
      set i = x;
      set j = -(0,x)
    end
  method countup (d)
    begin
      set i = +(i,d);
      set j = -(j,d)
    end
  method getstate () list(i,j)
```

Ejemplos y Semántica

Ejemplos

- El código anterior declara una clase `c1` que hereda de `object`.
- Cada objeto de la clase `c1` contendrá dos campos llamados `i` y `j`, y tres métodos llamados `initialize`, `countup` y `getstate`.
- Los nombres de los métodos corresponden a los tipos de mensajes a los que las instancias de `c1` pueden responder.

Ejemplos y Semántica

Ejemplos

Las clases definidas en el código anterior se pueden usar de la siguiente manera:

```
let t1 = 0
    t2 = 0
    o1 = new c1(3)
in begin
    set t1 = send o1 getstate();
    send o1 countup(2);
    set t2 = send o1 getstate();
    list(t1,t2)
end
```

Ejemplos y Semántica

Ejemplos

- En la expresión anterior se crean dos variables `t1` y `t2` y un objeto de la clase ya definida, `o1`.
- Cuando un objeto es creado (por medio de un `new`), su método `initialize` es invocado.
- En este caso `initialize` iguala `i` a 3 y `j` a -3.
- Luego, en el cuerpo del `let`, el método `getstate` de `o1` es invocado, retornando la lista `(3,-3)`.

Ejemplos y Semántica

Ejemplos

- Después el método `countup` de `o1` es invocado, cambiando el valor de los dos campos a 5 y -5.
- En la siguiente línea el método `getstate` es invocado, retornando la lista `(5,-5)`.
- Por último, el valor de `list(t1,t2)` (que es `((3 -3) (5 -5))`) es retornado como el valor de todo el programa.

Ejemplos y Semántica

Ejemplos:

```
class interior_node extends object
  field left
  field right
  method initialize (l, r)
    begin
      set left = l;
      set right = r
    end
  method sum () +(send left sum(),send right sum())

class leaf_node extends object
  field value
  method initialize (v) set value v
  method sum () value
```

Ejemplos y Semántica

Ejemplos

```
let
  o1 = new interior_node(
    new interior_node(new leaf_node(3),
                      new leaf_node(4)),
    new leaf_node(5))
in
  send o1 sum()
```

- En el código anterior, se tienen dos clases de nodos de un árbol: `interior_node` y `leaf_node`.
- Para encontrar la suma de las hojas de un nodo, se invoca el método `sum`.

Ejemplos y Semántica

Ejemplos

- Sin embargo, no se sabe a cuál clase de nodo se envía dicho mensaje; simplemente cada nodo acepta el mensaje `sum` y usa su método `sum` propio.
- Lo anterior es llamado *despacho dinámico* y es usado para implementar el polimorfismo de subclases.
- La expresión `1et` construye un árbol con dos nodos interiores y tres hojas.
- Se envía un mensaje `sum` al nodo `o1`; `o1` envía mensajes `sum` a sus subárboles, y así sucesivamente, retornando `12`.

Ejemplos y Semántica

Ejemplos:

Un método puede invocar otros métodos del mismo objeto usando el identificador `self`, el cual está ligado al objeto en el cual el método es invocado.

```
class oddeven extends object
  method initialize () 1
  method even (n)
    if zero?(n) then 1 else send self odd(sub1(n))
  method odd (n)
    if zero?(n) then 0 else send self even(sub1(n))

let o1 = new oddeven()
in send o1 odd(13)
```

Ejemplos y Semántica

Ejemplos herencia

- La herencia permite al programador definir nuevas clases con la modificación incremental de las clases existentes.
- Si una clase *c2* extiende una clase *c1*, se dice que *c1* es el *padre* de *c2*, o que *c2* es el *hijo* de *c1*.
- Como la herencia define a *c2* como una extensión de *c1*, ésta última debe estar definida antes que *c2*.
- Es por esto que se introduce una clase *object* sin métodos o campos, de la cual heredan todas las clases que se vayan a crear.
- Además, como *object* no tiene método *initialize*, es imposible crear un objeto de dicha clase.

Ejemplos y Semántica

Ejemplos herencia

Se tiene el siguiente código:

```
class c1 extends object
  field x
  field y
  method initiliaze () 1
  method setx1 (v) set x = v
  method sety1 (v) set y = v
  method getx1 () x
  method gety1 () y
```

```
class c2 extends c1
  field y
  method sety2 (v) set y = v
  method getx2 () x
  method gety2 () y
```

Ejemplos y Semántica

Ejemplos herencia

```
let
  o2 = new c2()
in
  begin
    send o2 setx1(101);
    send o2 sety1(102);
    send o2 sety2(999);
    list(send o2 getx1(), % retorna 101
          send o2 gety1(), % retorna 102
          send o2 getx2(), % retorna 101
          send o2 gety2()) % retorna 999
  end
```

Ejemplos y Semántica

Ejemplos herencia

- En el código anterior, un objeto de clase c2 tiene dos campos llamados y: uno declarado en c1 y el otro declarado en c2.
- Los métodos declarados en c1 ven los campos x y y de c1.
- En c2, la referencia a x en el método getx2 se refiere al campo x de c1, mientras que la referencia a y en el método gety2 se refiere al campo y de c2.

Ejemplos y Semántica

Ejemplos herencia

- Si un método m de una clase $c1$ es redeclarado en una de sus subclases $c2$, se dice que el nuevo método *anula* el método viejo.
- Si un mensaje m es enviado a un objeto de clase $c2$, el nuevo método es usado.

Ejemplos y Semántica

Ejemplos herencia

Ejemplo:

```
class c1 extends object
  method initialize () 1
  method m1 () 1
  method m2 () 100
  method m3 () send self m2()
```

```
class c2 extends c1
  method initialize () 1
  method m2 () 2
```

Ejemplos y Semántica

Ejemplos herencia

```
let
    o1 = new c1()
    o2 = new c2()
in
    list(send o1 m1(), % retorna 1
         send o1 m2(), % retorna 100
         send o1 m3(), % retorna 100
         send o2 m1(), % retorna 1 (de c1)
         send o2 m2(), % retorna 2 (de c2)
         send o2 m3()) % retorna 2 (m3 de c1 llama a m2 de c2)
```

Ejemplos y Semántica

Ejemplos herencia

- Cuando o2 envía el mensaje m3, el cuerpo del método en c1 es evaluado, con `self` ligado a o2.
- Sin embargo, el método para el mensaje m2, es m2 de c2.
- Cuando el método a ser ejecutado, no se determina sin conocer la clase actual del objeto en el cual el método es invocado (solo se conocerá en tiempo de ejecución), se dice que hay *despacho dinámico de métodos*.
- Si el método a ser ejecutado se determina desde la declaración de la superclase, se dice que hay *despacho estático de métodos*.
- En nuestro lenguaje usaremos despacho dinámico de métodos.

Ejemplos y Semántica

Ejemplos:

Un ejemplo de super llamados:

```
class point extends object
  field x
  field y
  method initialize (initx, inity)
    begin
      set x = initx;
      set y = inity
    end
```

```
class colorpoint extends point
  field color
  method initialize (initx, inity, initcolor)
    begin
      super initialize (initx, inity);
      set color = initcolor
    end
```

Aspectos Generales

- Cuando se incorporan los conceptos de programación orientada a objetos, un programa consta de un conjunto de declaraciones de clase y de una expresión.
- Cuando un programa es evaluado, las declaraciones de clases son procesadas por `elaborate-class-decls!`, y luego la expresión es evaluada.
- El trabajo de `elaborate-class-decls!` es almacenar las declaraciones de clases de forma que ellas se hagan accesibles para cuando se necesiten.

Aspectos Generales

El procedimiento `eval-program` estará definido de la siguiente manera:

```
(define eval-program
  (lambda (pgm)
    (cases program pgm
      (a-program (c-decls exp)
        (elaborate-class-decls! c-decls)
        (eval-expression exp (init-env))))))
```

Aspectos Generales

- Cuando se incorporan objetos al lenguaje se definen tres nuevos tipos de expresiones (creación, invocación de métodos y super llamados).
- Se deben agregar tres nuevos casos a `eval-expression` para estas tres nuevas reglas de producción.
- Cuando una expresión `send` es evaluada, los operandos y la expresión de objeto son evaluados.
- Luego, `find-method-and-apply` encuentra el método asociado con el nombre del método en la declaración del objeto, y dicho método es aplicado a sus argumentos.

Aspectos Generales

Se debe incorporar el siguiente caso al procedimiento `eval-expression` para las expresiones de invocación de métodos (`send`):

```
(method-app-exp (obj-exp method-name rands)
  (let ((args (eval-rands rands env))
        (obj (eval-expression obj-exp env)))
    (find-method-and-apply
      method-name (object->class-name obj) obj args)))
```

Aspectos Generales

`find-method-and-apply` toma cuatro argumentos:

- 1 un nombre de método,
- 2 el nombre de la clase en la cual se comenzará a buscar el método,
- 3 el valor de `self` y
- 4 la lista de argumentos.

Aspectos Generales

Un super llamado es similar a la invocación de métodos normal, excepto que el método es buscado en la superclase de la clase huésped de la expresión.

```
(super-call-exp (method-name rands)
  (let ((args (eval-rands rands env))
        (obj (apply-env env 'self))))
  (find-method-and-apply
    method-name (apply-env env '%super) obj args)))
```

El nombre de la superclase será ligada a una variable especial llamada %super.

Aspectos Generales

- El último caso que debe considerarse en el procedimiento `eval-expression` es el de creación de objetos.
- Cuando una expresión `new` es evaluada, los operandos son evaluados y un nuevo objeto del nombre de la clase es creado.
- Luego, su método `initialize` es llamado, pero su valor es ignorado.
- Por último, el objeto es retornado.

Aspectos Generales

Se debe añadir la siguiente clausula al procedimiento `eval-expression`:

```
(new-object-exp (class-name rands)
  (let ((args (eval-rands rands env))
        (obj (new-object class-name)))
    (find-method-and-apply
      'initialize class-name obj args)
    obj))
```

Aspectos Generales

- Cualquier implementación debe proveer los procedimientos `elaborate-class-decls!`, `find-method-and-apply`, `object->class-name`, y `new-object`.
- Además, cada implementación de proveer las estructuras de datos y otros procedimientos que se requieran.

Implementación simple

- Una declaración de clase contiene toda la información que se necesita, incluyendo el nombre de la clase, el de superclase, sus campos y sus declaraciones de métodos.
- Por lo tanto, se representarán las clases y métodos por sus declaraciones.

Implementación simple

Una declaración de clase es un tipo de dato definido de la siguiente manera:

```
(define-datatype class-decl class-decl?  
  (a-class-decl  
    (class-name symbol?)  
    (super-class-name symbol?)  
    (field-ids (list-of symbol?))  
    (methods-decls (list-of method-decl?))))
```


Implementación simple

Una declaración de método está definida de la siguiente manera:

```
(define-datatype method-decl method-decl?  
  (a-method-decl  
    (method-name symbol?)  
    (methods-args (list-of symbol?))  
    (body expression?)))
```

Implementación simple

- El repositorio de declaraciones de clases se construirá usando la variable global `the-class-env`.
- El procedimiento `elaborate-class-decls!` modificará el contenido de la variable `the-class-env` con las declaraciones de clase que se están definiendo.

```
(define the-class-env '())
```

```
(define elaborate-class-decls!  
  (lambda (c-decls)  
    (set! the-class-env c-decls)))
```

Implementación simple

El procedimiento `lookup-class` busca un nombre de clase en `the-class-env` y retorna su correspondiente declaración.

```
(define lookup-class
  (lambda (name)
    (let loop ((env the-class-env))
      (cond
        ((null? env)
         (eopl:error 'lookup-class
          "Unknown class ~s" name))
        ((eqv? (class-decl->class-name (car env)) name)
         (car env))
        (else (loop (cdr env)))))))
```

Implementación simple

- Un objeto es representado como una lista de *partes*, donde cada parte corresponde a cada clase en la cadena de herencia.
- Cada parte consiste del nombre de la clase y un vector que mantiene el estado de la clase.

Implementación simple

Ejemplo

Ejemplo:

```
class c1 extends object
  field x
  field y
  method initialize ()
    begin
      set x = 11;
      set y = 12
    end
  method m1 () ... x ... y ...
  method m2 () ... send self m3() ...
```

Implementación simple

Ejemplo

```
class c2 extends c1
  field y
  method initialize ()
    begin
      super initialize();
      set y = 22
    end
  method m1 (u, v) ... x ... y ...
  method m3 () ...
```

Implementación simple

Ejemplo

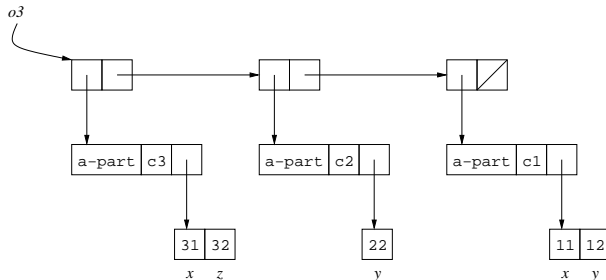
```
class c3 extends c2
  field x
  field z
  method initialize ()
    begin
      super initialize();
      set x = 31;
      set z = 32
    end
  method m3 () ... x ... y ... z ...

let o3 = new c3()
in send o3 m1(7,8)
```

Implementación simple

Ejemplo

- La representación de o3 se muestra en la siguiente figura:



- La declaración de la primera parte de la lista representa el punto más bajo de la cadena de clases.
- Mientras más se avance en la lista, más cerca se estará del tope de la jerarquía.

Implementación simple

Cada una de las *partes* mostradas anteriormente, se definen con el tipo de dato `part`:

```
(define-datatype part part?  
  (a-part  
    (class-name symbol?)  
    (fields vector?)))
```

Implementación simple

- Para construir un objeto, se construye una lista de partes, dado un nombre de clase.
- Si el nombre es `object`, entonces se sabe que se ha llegado al tope de la cadena de herencia y no hay más partes a construir.
- De lo contrario, se encuentra la declaración de clase correspondiente al nombre de clase dado y se retorna una lista cuyo primer elemento es la primera parte y cuya cola es obtenida haciendo un llamado recursivo con su superclase.

Implementación simple

El procedimiento `new-object` estará definido de la siguiente manera:

```
(define new-object
  (lambda (class-name)
    (if (eqv? class-name 'object)
        '()
        (let ((c-decl (lookup-class class-name)))
          (cons
            (make-first-part c-decl)
            (new-object (class-decl->super-name c-decl))))))))
```

Implementación simple

```
(define make-first-part
  (lambda (c-decl)
    (a-part
     (class-decl->class-name c-decl)
     (make-vector (length (class-decl->field-ids c-decl)))))))
```

Implementación simple

- En el procedimiento `find-method-and-apply`, se buscan las clases a través de la cadena de herencia hasta que se encuentre una clase que declare un método que sea igual al nombre de método dado.
- Cuando esto sucede, se hace un llamado a `apply-method` con la declaración de método encontrada, el nombre de la clase huésped, `self` y los argumentos.

Implementación simple

El procedimiento `find-method-and-apply` estará definido de la siguiente manera:

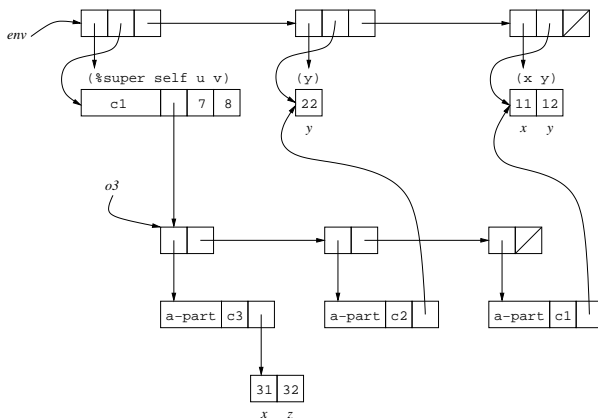
```
(define find-method-and-apply
  (lambda (m-name host-name self args)
    (if (eqv? host-name 'object)
        (eopl:error 'find-method-and-apply
                     "No method for name ~s" m-name)
        (let ((m-decl (lookup-method-decl m-name
                                           (class-name->method-decls host-name))))
          (if (method-decl? m-decl)
              (apply-method m-decl host-name self args)
              (find-method-and-apply m-name
                                     (class-name->super-name host-name)
                                     self args))))))
```

Implementación simple

- Para aplicar un método, se debe ejecutar el cuerpo del método en un ambiente en el cual cada variable esté ligada al valor apropiado, es decir, un ambiente en el cual el primer elemento contenga la ligadura para `%super`, para `self` y los parámetros formales del método.
- El resto del ambiente provee una ligadura para cada campo que es visible desde el método (aquellos campos que empiezan con la clase huésped).

Implementación simple

La siguiente figura muestra el ambiente para la aplicación del ejemplo mostrado anteriormente:



Implementación simple

- Si se ejecuta `send o3 m1(7,8)`, entonces los campos visibles desde el método `m1` son aquellos que comienzan en la parte de `o3`, que corresponden al método `m1` de la clase `c2`.
- De allí que los nombres de clase dan una *vista* del objeto; se puede encontrar la vista con el procedimiento:

```
(define view-object-as  
  (lambda (parts class-name)  
    (if (eqv? (part->class-name (car parts)) class-name)  
        parts  
        (view-object-as (cdr parts) class-name))))
```

Implementación simple

- De la vista del objeto, se puede generar un ambiente que consiste de un elemento para cada parte.
- Cada elemento liga los campos de una parte usando el vector ya construido.
- Para esto se utiliza el procedimiento `build-field-env`.

Implementación simple

El procedimiento `build-field-env` estará definido de la siguiente manera:

```
(define build-field-env
  (lambda (parts)
    (if (null? parts)
        (empty-env)
        (extend-env-refs
         (part->field-ids (car parts))
         %      (part->fields      (car parts))
         (build-field-env (cdr parts))))))

(define extend-env-refs
  (lambda (syms vec env)
    (extended-env-record syms vec env)))
```

Implementación simple

Finalmente, el procedimiento `apply-method` estará definido de la siguiente manera:

```
(define apply-method
  (lambda (m-decl host-name self args)
    (let ((ids (method-decl->ids m-decl))
          (body (method-decl->body m-decl))
          (super-name (class-name->super-name host-name)))
      (eval-expression body
        (extend-env
          (cons '%super (cons 'self ids))
          (cons super-name (cons self args))
          (build-field-env
            (view-object-as self host-name)))))))
```

Objetos Planos

- Es deseable que no se construyan todos esos elementos cada vez que se hace un llamado a un método.
- Sería mejor representar todo el almacenamiento de un objeto como un vector, en vez de una lista de partes. Es decir,

```
(define-datatype object object?  
  (an-object  
    (class-name symbol?)  
    (fields vector?)))
```

- Los campos se almacenarán tomando los de la clase mas “vieja” primero.

Objetos Planos

Retomando el ejemplo mostrado anteriormente:

```
class c1 extends object
  field x
  field y
  method initialize ()
    begin
      set x = 11;
      set y = 12
    end
  method m1 () ... x ... y ...
  method m2 () ... send self m3() ...
```

Objetos Planos

```
class c2 extends c1
  field y
  method initialize ()
    begin
      super initialize();
      set y = 22
    end
  method m1 (u, v) ... x ... y ...
  method m3 () ...
```

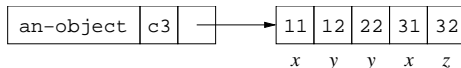
Objetos Planos

```
class c3 extends c2
  field x
  field z
  method initialize ()
    begin
      super initialize();
      set x = 31;
      set z = 32
    end
  method m3 () ... x ... y ... z ...

let o3 = new c3()
in send o3 m1(7,8)
```


Objetos Planos

- En este ejemplo, un objeto de clase c1 tendrá sus campos organizados como (x y); un objeto de clase c2 tendrá sus campos como (x y y), siendo el segundo y el que pertenece a c2; y un objeto de clase c3 será organizado como (x y y x z).
- La siguiente figura muestra la representación del objeto o3:



Objetos Planos

- Como se espera que los métodos de la clase `c3` se refieran al campo `x` declarado en `c3`, y no al declarado en `c1`, se debe cambiar la implementación de los ambientes, para que se use la ocurrencia de la variable que está más a la derecha del vector.

Objetos Planos

Para soportar esto, se redefine `rib-find-position`:

```
(define rib-find-position
  (lambda (name symbols)
    (list-find-last-position name symbols)))

(define list-find-last-position
  (lambda (sym los)
    (let loop
      ((los los) (curpos 0) (lastpos #f))
      (cond
        ((null? los) lastpos)
        ((eqv? sym (car los))
         (loop (cdr los) (+ curpos 1) curpos))
        (else (loop (cdr los) (+ curpos 1) lastpos))))))
```

Objetos Planos

- Como no se ha cambiado la representación de las clases ni de los métodos, sólo se deben considerar dos procedimientos: `new-object` y `find-method-and-apply`.
- El procedimiento `new-object` crea el vector haciendo un llamado a `rool-up-field-length`.

Objetos Planos

- `rool-up-field-length` es un procedimiento recursivo que empieza con un nombre de clase y encuentra el número total de campos que deben ser almacenados para un objeto de dicha clase.
- Si la clase es `object`, no hay campos; de lo contrario, el número de campos es la suma del número de campos necesarios para la clase padre y el número de campos de la clase actual.

Objetos Planos

El procedimiento `new-object` estará definido de la siguiente manera:

```
(define new-object
  (lambda (class-name)
    (an-object
     class-name
     (make-vector (roll-up-field-length class-name))))))
```

```
(define roll-up-field-length
  (lambda (class-name)
    (if (eqv? class-name 'object)
        0
        (+ (roll-up-field-length
             (class-name->super-name class-name))
           (length (class-name->field-ids class-name))))))
```

Objetos Planos

- El procedimiento `find-method-and-apply` no cambia, ya que no se trabaja con la representación del objeto.
- Sin embargo, `apply-method` si se redefine: como hay sólo un vector de campos, se construye un solo elemento para los campos.

Objetos Planos

El procedimiento `apply-method` estará definido de la siguiente manera:

```
(define apply-method
  (lambda (m-decl host-name self args)
    (let ((ids (method-decl->ids m-decl))
          (body (method-decl->body m-decl))
          (super-name (class-name->super-name host-name))
          (field-ids (roll-up-field-ids host-name))
          (fields (object->fields self)))
      (eval-expression body
        (extend-env
          (cons '%super (cons 'self ids))
          (cons super-name (cons self args))
          (extend-env-refs field-ids fields (empty-env)))))))
```


Objetos Planos

- El procedimiento `apply-method` llama a `roll-up-fields-ids` para construir una lista igual de identificadores.
- La lista de identificadores generalmente es del mismo tamaño que el vector de campos cuando la clase huésped y la clase de `self` son la misma.
- Si la clase huésped está más arriba, en la cadena de clases, entonces habrá más elementos en el vector que en la lista.

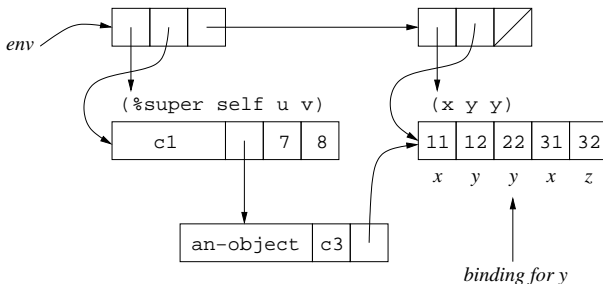
Objetos Planos

El procedimiento `roll-up-fields-ids` estará definido de la siguiente manera:

```
(define roll-up-field-ids
  (lambda (class-name)
    (if (eqv? class-name 'object)
        '()
        (append
         (roll-up-field-ids
          (class-name->super-name class-name))
         (class-name->field-ids class-name)))))
```

Objetos Planos

La siguiente figura muestra el ambiente para la aplicación en `send o3 m1(7,8)`, en la representación de objetos planos:



Objetos Planos

- Aquí se ve que el vector puede ser más largo que la lista de identificadores.
- La lista de identificadores es sólo (x y y) ya es que sólo dichas variables son visibles desde m1 de c2.

Objetos Planos

- Para evitar el llamado a `roll-up-field-ids` en cada llamado a un método, se necesita calcular esta información y almacenarla con el método.
- Además, se almacenará el nombre del método de la superclase, para usarlo en super llamados.

Objetos Planos

De esta manera, se crea un nuevo tipo de dato para mantener toda esta información:

```
(define-datatype method method?  
  (a-method  
    (method-decl method-decl?)  
    (super-name symbol?)  
    (field-ids (list-of symbol?))))
```

Objetos Planos

- La información almacenada es *estática*, es decir, no depende de valores denotados o expresados que puedan aparecer cuando el programa es ejecutado.
- De esta manera, es mucho mejor calcularla exáctamente una vez por clase. Para hacerlo, se necesita un tipo de dato para las clases.

Objetos Planos

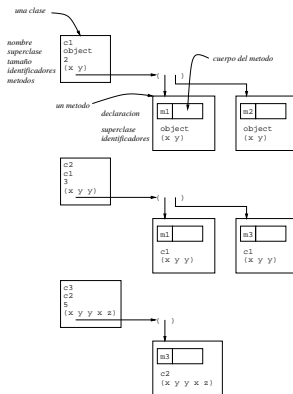
El tipo de dato class estará definido así:

```
(define-datatype class class?
  (a-class
    (class-name symbol?)
    (super-name symbol?)
    (field-length integer?)
    (field-ids (list-of symbol?))
    (methods method-environment?)))

(define method-environment? (list-of method?))
```


Objetos Planos

La siguiente figura muestra las estructuras de clases y métodos para el ejemplo mostrado anteriormente:



Objetos Planos

Las clases se contruyen redefiniendo el procedimiento
`elaborate-class-decls!`:

```
(define elaborate-class-decls!  
  (lambda (c-decls)  
    (initialize-class-env!)  
    (for-each elaborate-class-decl! c-decls)))
```

Objetos Planos

```
(define elaborate-class-decl!  
  (lambda (c-decl)  
    (let ((super-name (class-decl->super-name c-decl)))  
      (let ((field-ids (append  
                        (class-name->field-ids super-name)  
                        (class-decl->field-ids c-decl))))  
        (add-to-class-env!  
          (a-class  
            (class-decl->class-name c-decl)  
            super-name  
            (length field-ids)  
            field-ids  
            (roll-up-method-decls  
              c-decl super-name field-ids))))))))
```

Objetos Planos

El procedimiento `roll-up-method-decls` cambia cada declaración de método a un método, y retorna la lista de métodos:

```
(define roll-up-method-decls
  (lambda (c-decl super-name field-ids)
    (map
      (lambda (m-decl)
        (a-method m-decl super-name field-ids))
      (class-decl->method-decls c-decl))))
```

Objetos Planos

- Se deben ajustar los procedimientos `find-method-and-apply` y `apply-method` para esta representación.
- `find-method-and-apply` no cambia excepto que cada referencia a una declaración de método es ahora una referencia a un método.
- El procedimiento `apply-method` ahora toma un método (en vez de una declaración de método) como primer argumento, y obtiene la lista de identificadores del método (en vez de llamar a `rool-up-fields-ids`).

Objetos Planos

El procedimiento `apply-method` se redefine de la siguiente manera:

```
(define apply-method
  (lambda (method host-name self args)
    (let ((ids (method->ids method))
          (body (method->body method))
          (super-name (method->super-name method))
          (field-ids (method->field-ids method))
          (fields (object->fields self)))
      (eval-expression body
        (extend-env
          (cons '%super (cons 'self ids))
          (cons super-name (cons self args))
          (extend-env-refs field-ids fields (empty-env)))))))
```

Objetos Planos

Por último, se cambia `new-object` para obtener la información requerida de la clase:

```
(define new-object
  (lambda (class-name)
    (an-object
     class-name
     (make-vector (class-name->field-length class-name))))))
```

Preguntas

?

Próxima sesión

- Objetos y tipos.