Tema 11: Programación orientada a objetos

Índice

1. La Programación Orientada a Objetos

En este tema vamos a introducir el paradigma de Programación Orientada a Objetos (POO), incidiendo en sus aspectos fundamentales. Vamos a utilizar de nuevo el lenguaje de programación Scheme (en concreto la librería `class.ss` de la implementación MzScheme) para resaltar los elementos esenciales de la POO que son comunes a los distintos lenguajes de programación orientados a objetos (Java, C#, C++, etc.).

1.1. Características principales

Una de las ideas fundamentales del paradigma de programación orientada a objetos es el concepto de *objeto* como una entidad que engloba datos y funciones.

En programación funcional el elemento principal que determina el funcionamiento de un programa son las funciones. Las funciones transforman datos, los cuales sólo tienen posibilidad de existir como valores que se pasan a una función o que son devueltos por ella. Pero ahí termina su existencia. No tienen vida fuera de las funciones.

En programación imperativa, por otra parte, los datos se pueden considerar los elementos fundamentales de un programa. Existen por si mismos y son usados y modificados por las funciones. La posibilidad de la programación imperativa de mantener y modificar un cierto estado (datos, valores de las variables) la hace muy potente para representar y modelar procesos del mundo real que serían complicados de expresar en forma de programación funcional. Las funciones son utilizadas para modificar el estado del programa.

Sin embargo, es posible un enfoque distinto que agrupe datos y funciones. Se trata del utilizado en el paradigma de Programación Orientada a Objetos (POO). En este enfoque, el concepto fundamental es el de *objeto*. Un objeto es una entidad con un estado (datos o *variables de instancia*) y unas funciones (*métodos*) que pueden acceder y modificar este estado. Para evaluar las funciones hay que *enviar un mensaje* al objeto solicitando que se ejecute alguno de sus métodos. Sólo es posible consultar el estado de un objeto mediante alguno de sus métodos. De esta forma, en POO se refuerza la filosofía de la *barrera de abstracción* y de la *ocultación de información*.

El enfoque de la POO permite modelar un dominio (problema a programar, un simulador de deportes, por ejemplo) de una forma muy cercana a la realidad. Los objetos del programa simulan los objetos (sustantivos) del dominio (por ejemplo, bicicleta, marchas, carretera, etc.). Y los métodos de los objetos permiten modelar perfectamente las acciones (verbos, por ejemplo cambiar de marcha, pedalear, etc.) que pueden realizar.

Entre las características de la POO podemos indicar los siguientes conceptos:

- Objetos (dinámicos/tiempo de ejecución) y clases (estáticas/tiempo de compilación).
- Los objetos agrupan datos (variables de instancia) y funciones (métodos), realizando de esa forma una ocultación de la información.
- Los métodos de los objetos se invocan mediante mensajes.
- Dispatch dinámico: cuando una operación es invocada sobre un objeto, el propio objeto determina qué código se ejecuta. Dos objetos con la misma interfaz pueden tener implementaciones distintas.
- Herencia: las clases se pueden definir utilizando otras clases como plantillas y modificando sus métodos y/o variables de instancia.

1.2. Historia de la Programación Orientada a Objetos

Podemos considerar como inicio del paradigma de Programación Orientada a Objetos (Object Oriented Programming) el lenguaje de programación Simula desarrollado por Kristen Nygaard y Ole-Johan Dahl en la mitad de los años 60 en el Centro de Computación Noruego (The Norwegian Computing Center). Simula se definió como un lenguaje de programación orientado a la simulación de procesos, con el que se podían definir distintos tipos de actividades. En este lenguaje aparecen por primera vez los conceptos de *clases* y *objetos*.

Después se desarrollaron los lenguajes Smalltalk, C++, Java y C#. (ver más detalles en http://www.rescomp.berkeley.edu/~hossman/cs263/paper.html).

1.3. Programación Orientada a Objetos en MzScheme

El lenguaje de programación MzScheme contiene una colección de librerías estándar, entre las que se encuentra la libería `class.ss` con la que se define una extensión orientada a objetos de Scheme.

Para usar la librería `class.ss` hay que importarla con:

```
(require (lib "class.ss"))
```

En los apartados siguiente veremos qué construcciones contiene esta librería.

2. Objetos y clases

Recordemos las estructuras en MzScheme:

```
(define-struct persona (nombre apellidos fecha-nacimiento nif))
```

Una estructura define un nuevo tipo agrupando un conjunto de campos. Se definen automáticamente un constructor, selectores y mutadores.

```
(define p1 (make-persona "Rafa" "Nadal" "1988-03-25" '21324299X))
```

```
(persona-nombre p1) -> "Rafa"
(persona-fecha-nacimiento p1) -> "1988-03-25"
(set-persona-fecha-nacimiento! p1 "1988-04-25")
```

Supongamos que queremos calcular la edad de una persona... la tenemos que definir como una función aparte.

```
(define (edad persona fecha-actual)
   (let ((año-nacimiento
            (string->number
               (substring (persona-fecha-nacimiento persona) 0 4)))
          (año-actual (string->number (substring fecha-actual 0 4))))
      (- año-actual año-nacimiento)))

(edad p1 "2006-05-23")
```

para obtener la fecha actual:

```
(require (lib "date.ss"))
(date-display-format 'iso-8601)
(define fecha-actual (date->string (seconds->date (current-seconds))))

(edad (persona-fecha-nacimiento p1) fecha-actual)
```

Vamos a hacer esto con Programación Orientada a Objetos

```
(require (lib "class.ss"))
```

Los primeros elementos que vamos a ver de POO:

- Cómo definir una clase: campos y métodos
- Cómo crear instancias de una clase
- Paso de mensajes: cómo ejecutar métodos de las instancias

Definimos una clase `persona%` que tiene como campos de inicialización el nombre y el dni y como resto de campos apellidos y fecha-nacimiento. Definimos los métodos `di-hola` y `edad` que podrán ser ejecutados por objetos ; de esta clase.

Una clase es (al igual que una estructura) una plantilla donde se define un conjunto de campos (también llamados en POO variables de instancia) y un conjunto de métodos (funciones) que podrán ser ejecutadas por los objetos de esa clase. Las estructuras sólo contienen datos, no métodos.

Otra diferencia importante es que las clases deben definir valores por defecto de los campos, y que es posible diferenciar entre campos de inicialización y el resto de campos.

Todas las clases son hijas de la clase `object%` (ya hablaremos más adelante de la herencia).

Versión inicial y simplificada de persona

```
(define persona%
   (class object%
      (init-field nombre nif)
      (field (apellidos null) (fecha-nacimiento null))

      (define/public (di-hola)
         (printf "Hola, soy ~a~%" nombre))

      (super-new)))
```

Creamos un objeto de tipo `persona%` (instanciamos un objeto) indicando los valores iniciales de los campos `nombre` y `nif`:

```
(define p1 (new persona% (nif '212121232) (nombre "Pepito")))
```

Pedimos que ejecute el método `di-hola`

```
(send p1 di-hola)
```

Ejemplo completo

```
(define persona%
   (class object%
      (init-field nombre nif)
      (field (apellidos null) (fecha-nacimiento null))

      (define/public (set-apellidos nuevos-apellidos)
         (set! apellidos nuevos-apellidos))

      (define/public (set-fecha-nacimento fecha)
         (set! fecha-nacimiento fecha))

      (define/public (get-nombre-completo)
         (if (not (null? apellidos))
            (string-append nombre " " apellidos)
            nombre))

      (define/public (di-hola)
         (define nombre-completo (send this get-nombre-completo))
         (printf "hola, soy ~a~%" nombre-completo))

      (define/public (get-edad fecha-actual)
         (let ((año-nacimiento
                  (string->number (substring fecha-nacimiento 0 4)))
               (año-actual (string->number (substring fecha-actual 0
4))))
         (- año-actual año-nacimiento)))

      (super-new)))


(define p1 (new persona% (nombre "Rafa") (nif '23434222N)))
```

Pedimos que ejecute el método `di-hola`

```
(send p1 di-hola)
```

Pedimos que se modifiquen sus apellidos y volvemos a hacer que diga hola:

```
(send p1 set-apellidos "Nadal")
(send p1 di-hola)
```

Modificamos su fecha de nacimiento y vemos su edad

```
(send p1 set-fecha-nacimento "1988-03-25")
(require (lib "date.ss"))
(date-display-format 'iso-8601)
(define fecha-actual (date->string (seconds->date (current-seconds))))
(send p1 get-edad fecha-actual)
```

Último ejemplo: haciendo amigos

```
(define persona%
   (class object%
      (init-field nombre)
      (field (amigos null))

      (define/public (di-hola)
         (printf "hola, me llamo ~a~%" nombre))

      (define/public (añade-amigo otro)
         (set! amigos (cons otro amigos)))

       (define/public (saludan-amigos)
          (for-each
              (lambda (f) (send f di-hola))
              amigos))

      (super-new)))

(define p1 (new persona% (nombre 'Frodo)))
(define p2 (new persona% (nombre 'Gandalf)))
(define p3 (new persona% (nombre 'Aragorn)))

(send p1 añade-amigo p2)
(send p1 añade-amigo p3)
(send p1 saludan-amigos)
```

3. Herencia

Utilizando la herencia es posible definir una nueva clase basándose en una clase ya existente. La nueva clase añade nuevos métodos y/o campos a los de la clase ya existente. Se dice que la nueva clase EXTIENDE la clase padre.

La herencia facilita la reutilización de código y la abstracción.

Recordamos la clase anterior, con unas pequeñas variaciones:

```
(define persona%
   (class object%
      (init-field nombre)
      (field (amigos '()))

      (define/public (di-hola)
         (printf "hola, me llamo ~a~%" nombre))

      (define/public (get-nombre)
         nombre)

      (define/public (es-amigo? otro)
         (if (memq otro amigos)  ;memq comprueba si otro está en
amigos usando la igualdad eq?
             #t
             #f))

      (define/public (añade-amigo otro)
         (if (not (es-amigo? otro)) ; tambien es posible llamar a los
métodos directamente
             (begin
                (set! amigos (cons otro amigos))
                (send otro añade-amigo this))))

       (define/public (saludan-amigos)
          (for-each
              (lambda (f) (send f di-hola))
              amigos))

      (super-new)))

(define frodo (new persona% (nombre "Frodo")))
(define gandalf (new persona% (nombre "Gandalf")))
(send frodo añade-amigo gandalf)
(send frodo saludan-amigos)
(send gandalf saludan-amigos)
```

Definimos la clase `mago%` que extiende la clase `persona%`

```
(define mago%
   (class persona%
      (init-field nombre-pila nivel-conjuro)
      (field (energia 100) (vida #t))
      (inherit-field nombre)

      (define/public (get-nivel-conjuro)
         nivel-conjuro)

      (define/public (get-energia)
```

```
          energia)

       (define/public (rayo)
          (set! energia (- energia 10))
          (if (< 0 energia)
             (set! vida #f)))

       (define/public (lanza-conjuro otro-mago)
          (define otro-nombre (send otro-mago get-nombre))
          (define nivel-otro (send otro-mago get-nivel-conjuro))
          (printf "Yo, ~a, lanzo un conjuro a ~a~%" nombre otro-nombre)
          (if (< nivel-otro nivel-conjuro)
             (begin
                (send otro-mago rayo)
                (printf "Mi conjuro te ha alcanzado, ~a~%"
otro-nombre))
             (printf "~a, admito que eres más poderoso que yo"
otro-nombre)))
```

Inicializo superclase pasando un nombre construido a partir del nombre de pila:

```
(super-new (nombre (string-append "Mago " nombre-pila)))))
(define gandalf (new mago% (nombre-pila "Gandalf") (nivel-conjuro
100)))
(define saruman (new mago% (nombre-pila "Saruman") (nivel-conjuro
90)))
(send gandalf lanza-conjuro saruman)
(send saruman lanza-conjuro gandalf)
```

Definimos la clase `enano%` que extiende `persona%` y redefine el saludo

```
(define enano%
   (class persona%
      (inherit-field nombre)

      (define/override (di-hola)
         (printf "Mmmm.. soy ~a y estoy hambriento!~%" nombre))

      (super-new)))

(define gimli (new enano% (nombre "Gimli")))
(send frodo añade-amigo gimli)
(send frodo saludan-amigos)
```

4. Interfaces

Las interfaces definen un conjunto de métodos sin especificar su implementación. Cuando una clase implementa una interfaz debe especificar todos los metodos contenidos en ella.

Para definir una clase que implementa una interfaz hay que usar la palabra clave `class*`

```
(define i-stack% (interface () push! pop! is-empty?))

(define stack%
  (class* object% (i-stack%)
      (init-field (name 'stack))
      (field (stack null))

      (define/public (push! v)
        (set! stack (cons v stack)))

      (define/public (pop!)
        (let ((v (car stack)))
          (set! stack (cdr stack))
          v))

      (define/public (is-empty?)
        (null? stack))

      (define/public (print-name)
        (display name)
        (newline))

      (super-new)))

(define fancy-stack%
  (class stack%
    (inherit-field name)

    (define/override (push! v)
      (super push! (cons 'fancy v)))

    (super-new)))

(define double-stack%
  (class stack%
    (inherit push!)

    (define/public (double-push! v)
       (push! v)
       (push! v))

    (super-new (name 'double-stack))))

(define safe-stack%
   (class stack%
      (inherit is-empty?)

      (define/override (pop!)
         (if (is-empty?)
            #f
            (super pop!)))

      (super-new)))
```

```
(define i-extended-stack% (interface (i-stack%) size))

(define extended-stack%
   (class* safe-stack% (i-extended-stack%)
      (inherit-field stack)

      (define/public (size)
         (length stack))

      (super-new)))
```

5. Funciones de bajo nivel de acceso a los campos

La función `(get-field field objeto)` devuelve el valor de un campo publico

```
(get-field name s1)
(get-field stack s1)
```

Se pueden obtener selectores y mutadores de los campos accediendo a la clase con las funciones

```
(class-field-accessor clase campo) -> devuelve un selector
(class-field-mutator clase campo) -> devuelve un mutador
```

Ejemplos:

```
(define name-mutator (class-field-mutator stack% name))
(define name-selector (class-field-accessor stack% name))
(name-mutator s1 'pepito)
(name-selector s1)
(get-field name s1)
(send s1 print-name)
```

Funciones auxiliares (reflexión):

```
(object? v)
(class? v)
(interface? v)
(class->interface class)
(object-interface object)
(is-a? v interface)
(is-a? v class)
(subclass? v class)
(implementation? v interface)
(interface-extension? v interface)
(interface->method-names interface)
(field-names object)
```

6. Definición de clases y modelo de entornos

La programación orientada a objetos en Scheme amplía el modelo de entornos, añadiendo nuevas funcionalidades que no se pueden explicar directamente con ese modelo.

Sin embargo, es posible utilizar las características del modelo de entornos en la POO.

Por ejemplo, la definición de clases funciona de forma similar a la definición de funciones, en el sentido de que todos los objetos creados una clase pueden acceder a las variables (datos y funciones) del entorno en el que se creo la clase. Esto hace posible la creación de VARIABLES DE CLASE, variables que sólo son accesibles desde los objetos de una clase, y que son compartidas por todos ellos.

Veamos un par de ejemplos

Definimos la variable pi y la función square en el entorno global:

```
(define pi 3.14159)
(define (square x) (* x x))
```

Definimos una clase que usa la variable y la función. Los objetos de esa clase pueden acceder a ambas.

```
(define circulo%
   (class object%
      (init-field (radio 0))
      (define/public (area)
         (* pi (square radio)))
      (super-new)))

(define circ1 (new circulo% (radio 10)))
(send circ1 area)
```

Ejemplo contador (mal construido, porque la variable total está en el entorno global)

```
(define total 0)
(define contador%
   (class object%
      (init-field (c 0))
      (define/public (inc)
         (set! c (+ 1 c))
         (set! total (+ 1 total))
         c)
      (define/public (get-total)
         total)
      (super-new)))

(define c1 (new contador%))
(define c2 (new contador%))
(define c3 (new contador%))
(send c1 inc)
(send c1 inc)
```

```
(send c1 inc)
(send c2 inc)
(send c2 inc)
(send c3 inc)
(send c1 get-total)
total
```

Ejemplo correcto final, usando un entorno local a la clase en el que se define la variable total

```
(define contador%
   (let ((total 0))
      (class object%
         (init-field (c 0))
         (define/public (inc)
            (set! c (+ 1 c))
            (set! total (+ 1 total))
            c)
         (define/public (get-total)
            total)
         (super-new))))

(define c1 (new contador%))
(define c2 (new contador%))
(define c3 (new contador%))
(send c1 inc)
(send c1 inc)
(send c1 inc)
(send c2 inc)
(send c2 inc)
(send c3 inc)
(send c1 get-total)
total
```

7. Referencias

Para saber más de los temas que hemos tratado en esta clase puedes consultar las siguientes referencias:

- Manual de la librería class.ss (http://download.plt-scheme.org/doc/mzlib/mzlib-Z-H-6.html#node_chap_6) de MzScheme
- Schematics:cookbook (lib "class.ss") Notes (http://schemecookbook.org/Cookbook/IntroductionToMzlibClasses) .
- Conceptos de Programación Orientada a Objetos (http://java.sun.com/docs/books/tutorial/java/concepts/index.html) del tutorial de Java de Sun.
- Scheme with Classes, Mixins, and Traits (http://people.cs.uchicago.edu/~robby/pubs/papers/aplas2006-fff.pdf) , artículo de Matthew Flatt1, Robert Bruce Findler2 y Matthias Felleisen de la Universidad de

Chicago.