Paradigma
Orientado a Objetos

Módulo 02: Referencias. Estado. Compartir objetos. Identidad.

por Fernando Dodino

Versión 3.0
Septiembre 2024

Indice

1 Repaso

1.1 Mensaje y método

Mensaje y método no es sólo un juego de palabras. Me define de qué lado me pongo:

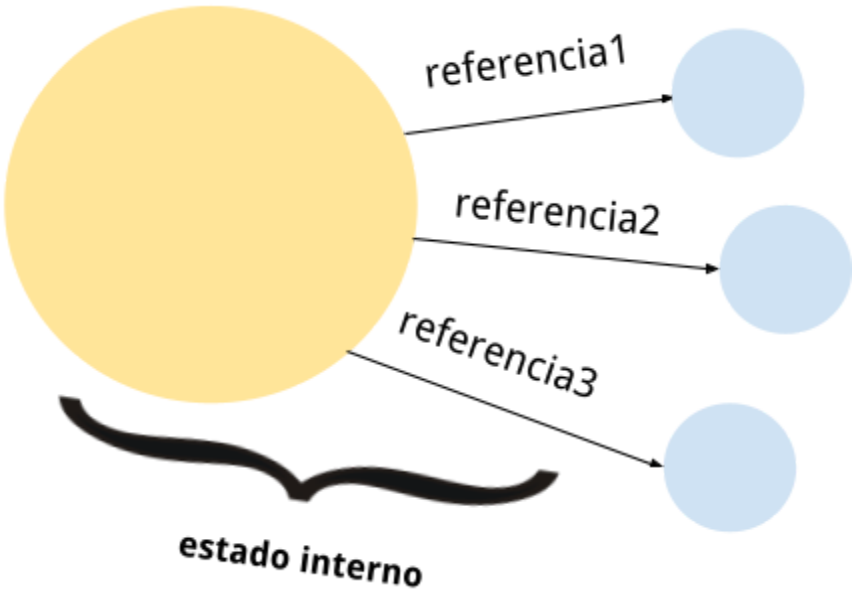
- **Mensaje**: soy usuario del objeto. Soy un observador que pido al objeto que haga algo (no quiero saber cómo lo resuelve, sólo quiero que lo haga).
- **Método**: yo soy un objeto, ejecuto código que está en mi definición, más allá de quién lo haya pedido.

1.2 Interfaz e implementación

- **Interfaz** es lo que un objeto publica para que otro objeto lo use (es el qué).
- **Implementación** es lo que un objeto encapsula para definir cómo se termina resolviendo un mensaje.

2 Estado de un objeto

Hemos visto que un objeto puede enviar mensajes a otros objetos, para lo cual los tiene que conocer. El conjunto de referencias que tiene un objeto representa su **estado**.



Al modelar una solución con objetos, buscaremos que el estado de un objeto sea manipulado únicamente por métodos definidos en el propio objeto. Cada referencia define un **atributo**, que tiene un nombre y un valor, el objeto apuntado.

3 Breve introducción a Wollok

Wollok es un lenguaje de programación **con fines didácticos**

- no obliga al desarrollador a definir tipos para los objetos
- es interpretado
- provee un entorno integrado de desarrollo o IDE donde trabajar

3.1 Primeros pasos en Wollok

Lo primero que hay que hacer es generar un nuevo proyecto Wollok siguiendo los pasos del sitio web.

```
wollok init --project aves
```

Eso crea por defecto un proyecto con un archivo *example.wlk* que podemos renombrar.

3.2 Definir un objeto en Wollok

Veremos un ejemplo que muestra la sintaxis básica: definimos a pepita, una golondrina que sabe

- volar: esto disminuye su energía en 10 unidades
- comer una cantidad (de alpiste en gramos): aumenta su energía en 2 unidades * la cantidad que coma

```
object pepita {
  var energia = 100

  method volar() {
    energia = energia - 10
  }

  method comer(cantidad) {
    energia = energia + 2 * cantidad
  }
```

```
}
```

Algunas observaciones:

- `object pepita`: define un nuevo objeto de nombre pepita, entre llaves deben definir atributos y comportamiento
- aquí vemos que no es obligatorio definir los tipos de los parámetros ni de los métodos que codificamos. ¿Qué pasa si un objeto no puede resolver un mensaje? El *method lookup* falla y aparecerá un mensaje de error.
- `var energia = 100`: define una referencia o atributo energia, con valor inicial 100 (un valor literal de *tipo* número).
- `method volar()`: define un método volar, sin parámetros. Entre llaves podemos ver el comportamiento.
- en el método comer vemos que recibimos un parámetro cantidad, que lo necesitamos para calcular la cantidad de energía que debemos aumentar.

4 Accessors

Como vimos en el párrafo Estado de un objeto, no es aconsejable que quien use a pepita manipule directamente sus variables, sino que debe enviarle mensajes que accedan o modifiquen las referencias a los objetos que conoce. Para ello existe cierto tipo de mensajes llamados **accessors**, cuyo fin es publicar la referencia (getter) o modificar dicha referencia (setter).

En el mismo ejemplo de pepita vamos a mostrar cómo definir un método getter y uno setter:

```
object pepita {
    ...

    // getter
    method energia() { return energia }

    // setter
    method energia(_energia) { energia = _energia }
```

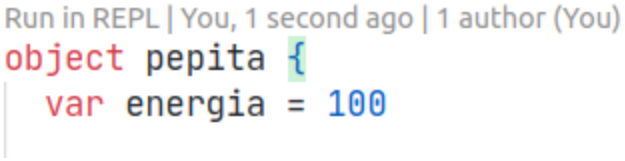
(ésta es la convención por defecto de Wollok)

El getter se define como un método de una sola línea, que devuelve algo.

El setter no, es un método que modifica el estado interno del objeto, no tiene necesidad de devolver nada.

5 Primeras pruebas en Wollok

Ahora que creamos un objeto pepita, podemos enviar mensajes en la consola interactiva, mediante el menú principal (F1 > Wollok: **Start a new REPL session**) o bien utilizando el menú contextual **Run in REPL** sobre pepita[1]:




```
> pepita.energia()
✓ 100
> pepita.volar()
✓
> pepita.energia()
✓ 90
> pepita.energia(170)
✓
> pepita.energia()
✓ 170
```

La sintaxis en general es

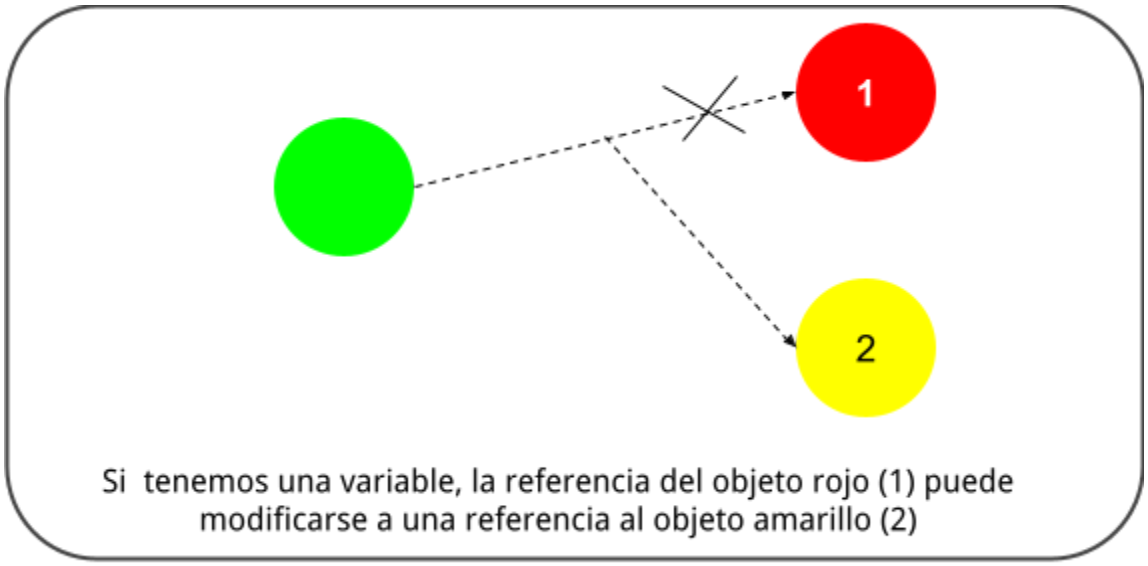
```
objeto.mensaje()
objeto.mensaje(parametro1)
objeto.mensaje(parametro1, parametro2)
```

6 Referencias en Wollok

Wollok tiene dos maneras de definir referencias

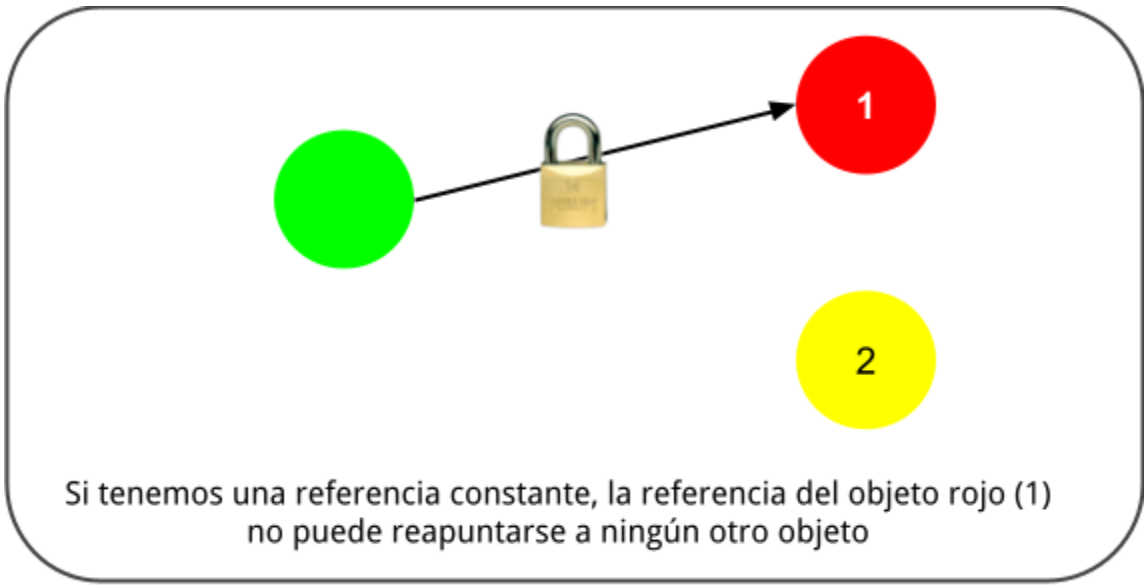
- **variables (var):** es una referencia que puede cambiar el objeto al que apunta
- **valores (const):** es una referencia fija a un objeto, que no puede cambiar (una vez inicializada no es válida una nueva operación de asignación)

[1] A futuro existirá un botón específico para ejecutar un archivo wlk



Si tenemos una variable, la referencia del objeto rojo (1) puede modificarse a una referencia al objeto amarillo (2)

```
var age = 10
age = 11
age = age + 1
```

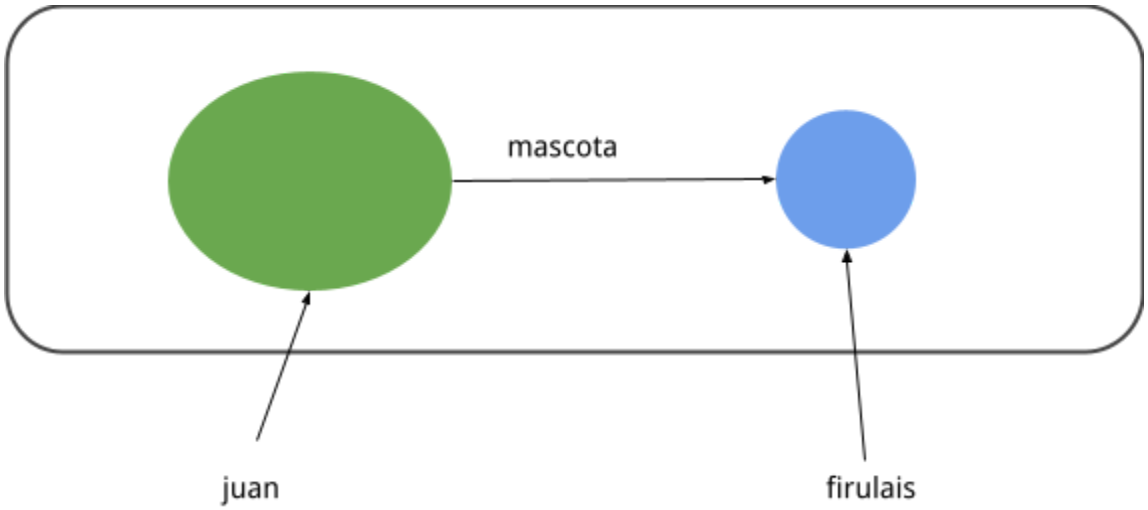


Si tenemos una referencia constante, la referencia del objeto rojo (1) no puede reapuntarse a ningún otro objeto

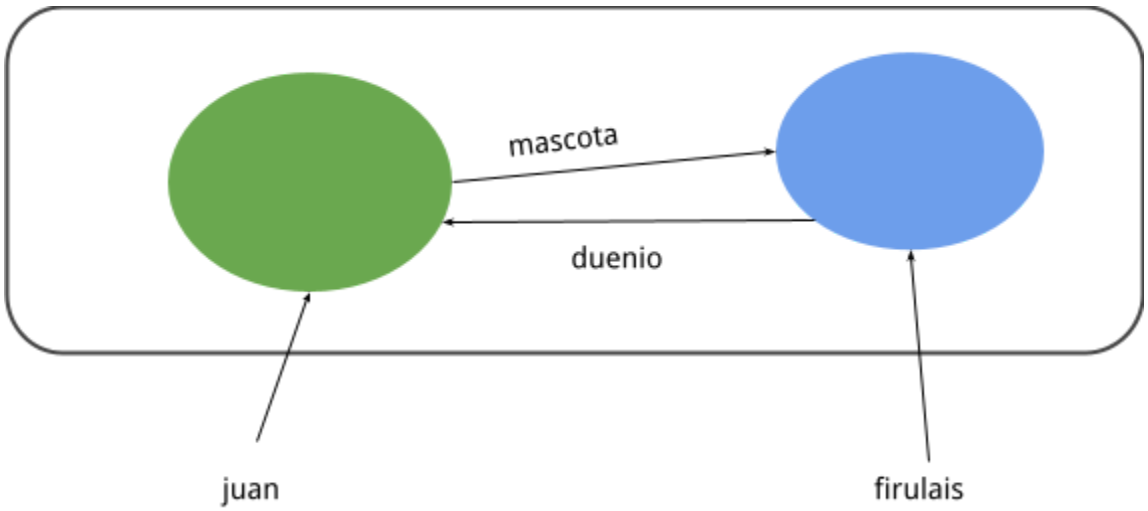
```
const adultAge = 21
adultAge = 18  // ESTO DA ERROR
```

7 Relaciones bidireccionales

Supongamos que tenemos el siguiente escenario: Juan tiene una mascota Firulais. Esto se grafica de la siguiente manera:



Esto implica que juan conoce a firulais, pero firulais no conoce a juan: la relación es unidireccional. Para establecer una relación bidireccional, necesitamos incorporar un atributo dueño en firulais:



Hacemos esto en Wollok:

```
object juan {
  var mascota

  method mascota(_mascota) { mascota = _mascota }
}
```

```
object firulais {
  var duenio

  method duenio(_duenio) { duenio = _duenio }
}
```

En la consola interactiva escribimos

```
> juan.mascota(firulais)
> firulais.duenio(juan)
```

Tener una relación bidireccional

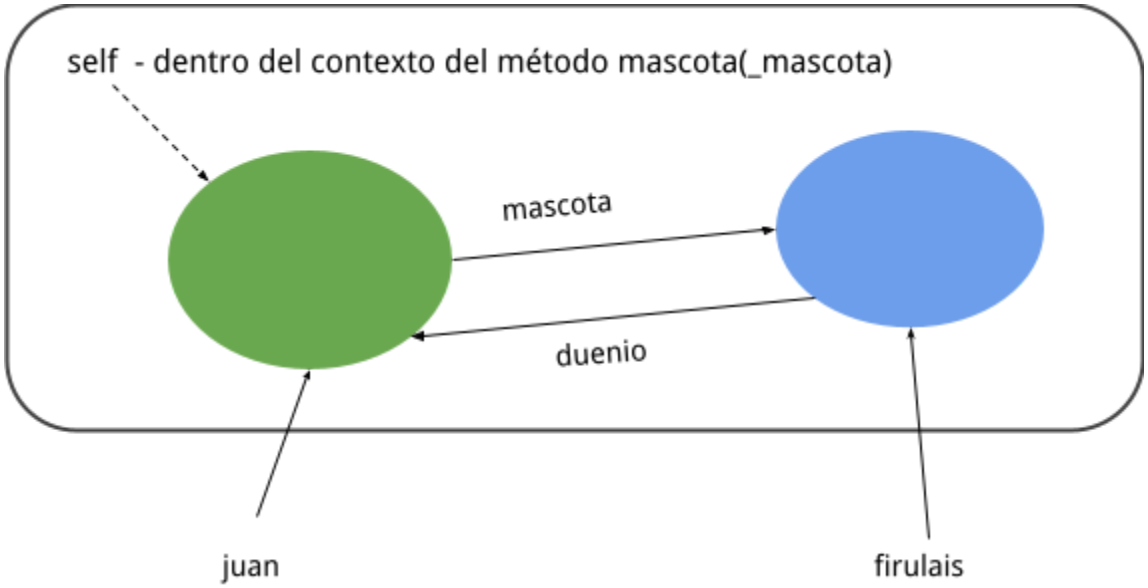
- es cómodo para **usarlo**, porque cualquiera de los dos objetos puede enviar un mensaje a otro
- pero requiere mantener ambas referencias sincronizadas, para evitar inconsistencias

Una alternativa puede ser que uno de los objetos sea el responsable de actualizar la otra referencia:

```
object juan {
   var mascota

   method mascota(_mascota) {
      mascota = _mascota
      mascota.duenio(self)
   }
}

object firulais {
   var duenio

   method duenio(_duenio) { duenio = _duenio }
}
```

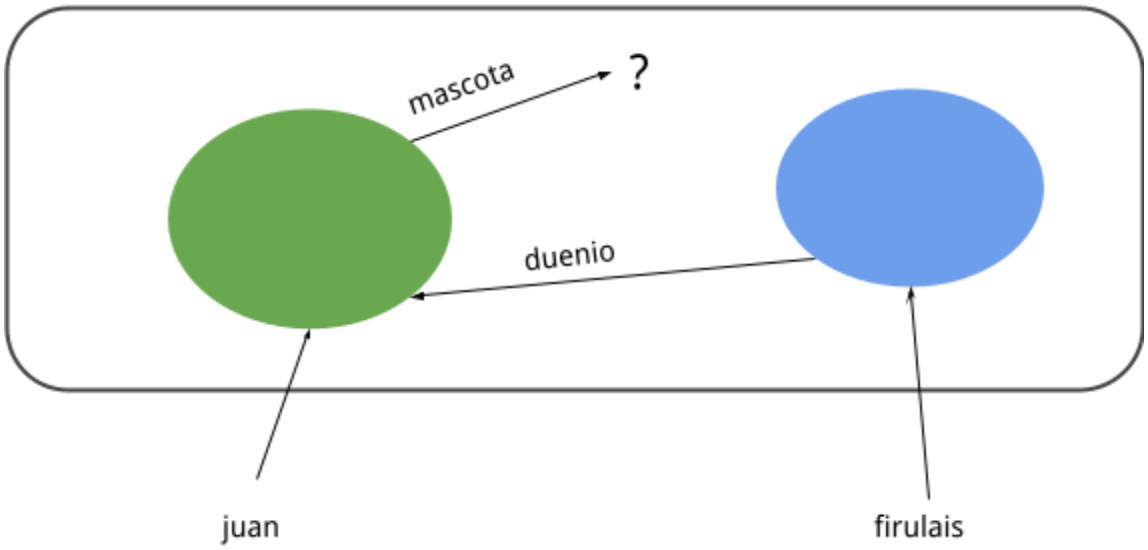
¿Qué es self? Dentro del contexto de un método es una referencia al propio objeto donde estamos ubicados.

```
> juan.mascota(firulais)
```



Esta solución tiene como desventaja que si enviamos el mensaje duenio a una mascota, podemos dejar una referencia sin definir en el dueño.

```
> firulais.duenio(juan)
```



8 Compartir valores

¿Qué pasa si juan y tefi quieren compartir actividades con firulais? Le pasamos la referencia al mismo objeto:

```
object tefi {
   ...
```

```
    method jugarCon(_mascota) { ... }
}

object juan {
    ...

    method alimentarA(_mascota) { ... }
}
```

Si tenemos los objetos tefi y juan...

```
> tefi.jugarCon(firulais)
> juan.alimentarA(firulais)
```

9 Referencias globales

Tanto juan, como tefi y firulais son **objetos autodefinidos**, o *well-known objects* (wko). Entonces cualquier objeto podría enviarle un mensaje sin necesidad de guardar una referencia propia:

```
object firulais {
   method jugar() {
      //
   }
}

object juan {
   method jugarConMascota(){
      firulais.jugar()
   }
}
```

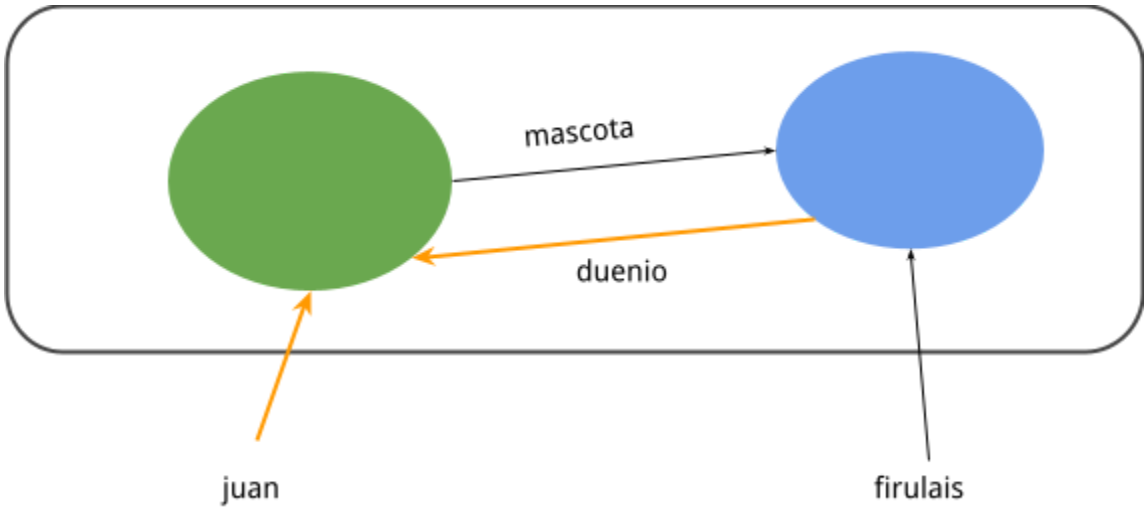
Ésta **no es una buena práctica**, porque ahora Juan ¡solo puede jugar con firulais! Acortar el camino utilizando los objetos autodefinidos como referencias es tentador, pero el costo que tiene cualquier acceso global es que nos lleva a soluciones rígidas, donde no es posible modificar el objeto al cual queremos enviarle el mensaje. Por eso, por más que nos parezca burocrático, es siempre

preferible que tengamos una referencia a los objetos autodefinidos: puede ser un parámetro, o bien atributos (definidos mediante *var* o *const*).

10 Identidad

En un sistema con objetos, cada objeto sabe que es él y ningún otro objeto más, por eso es el responsable de responder ante un mensaje. En el ambiente cada objeto tiene su propia **identidad**. En un sistema orientado a objetos es frecuente tener diferentes referencias y querer determinar si estamos hablando de un objeto u otro, entonces la definición de identidad nos dice que

Dos referencias son idénticas si apuntan al mismo objeto



En el ejemplo anterior, las referencias juan y duenio son idénticas, porque apuntan al mismo objeto

11 Resumen

Hemos visto que los objetos tienen estado interno, conformado por los atributos que son nombres de referencias hacia otros objetos. Las referencias pueden cambiar el objeto al que apuntan (en cuyo caso son referencias variables) o no (constantes). Cada referencia marca qué objeto conoce a otro, por eso para que la relación sea bidireccional necesitamos dos referencias entre los mismos objetos. Y además un objeto tiene identidad, que es lo que lo distingue de los demás objetos.