ENLACE O BINDING

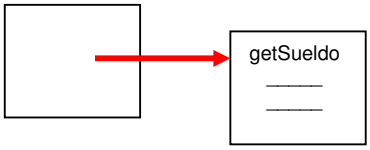
Vamos a estudiar un poquito cómo ocurre la magia del polimorfismo…

¿Qué sucede cuando envío un mensaje a un objeto? Se ejecuta el método, ok.
Ahora, si yo le pregunto el sueldo al empleado:
empleado sueldo
y tengo 3 tipos de empleado. ¿Qué código se va a ejecutar?
O mejor dicho: ¿puedo saber qué código se va a ejecutar de antemano?

No sabemos qué método se va a ejecutar hasta que el mensaje se envíe (en Runtime). En ese momento engancho el método (el código que se va a ejecutar) en base al objeto receptor de ese mensaje. Ese enganche entre operando y operador se conoce como binding, y el binding que usa Smalltalk es dinámico.

Binding estático: cuando compilo yo se exactamente qué código se va a ejecutar. En el programa generado tengo el puntero adonde comienza la función.

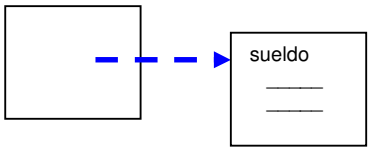
Empleado empleado;
empleado = …;
sueldo = getSueldo(empleado);



El enganche se hace en el momento de compilar (el ejecutable sabe exactamente a qué porción de código se debe llamar, Compile time).

Binding dinámico: cuando compilo dejo abierta la posibilidad de que el enlace con el código se asigne cuando ejecute esa instrucción. Pasado a objetos:

| empleado |
empleado ← …
empleado sueldo



El enganche se hace en el mismo momento que se ejecuta (Runtime).

¿Qué gano con tener binding dinámico?
Si yo no tuviera binding dinámico siempre sabría qué código ejecutaría, porque estaría obligado a hacer algo como

```
function sueldo ( e : Object ) {
   case e.type of
      jerarquico : sueldoJerarquico(e) |
      vip        : sueldoVIP(e) |
      comunacho  : sueldoComun(e)
   end;
}
```

y es justamente lo contrario a lo que juega el polimorfismo: tener varios objetos que entienden el mismo mensaje e intercambiarlos libremente sin que yo me entere.

El mejor ejemplo son los booleanos:

| a |

a ← … (algún valor entero)
a > 3 ifTrue: [… código …]

¿a > 3 es true o false? Depende de a. Hasta el momento en que evalúe no se si (a > 3) me va a devolver un objeto true o un objeto false, lo bueno es que el polimorfismo me permite trabajar con instancias True y False que entienden los mismos mensajes
ifTrue:
ifFalse:
ifTrue:ifFalse:
etc.

¿Cómo se implementan?

Clase True	*Clase False*

ifTrue: operand
  ^operand value
	ifTrue: operand
	  ^nil

ifFalse: operand
  ^nil
	ifFalse: operand
	  ^operand value

ifTrue: trueOperand ifFalse: falseOperand
  ^trueOperand value
	ifTrue: trueOperand ifFalse: falseOperand
	  ^falseOperand value

not
  ^false
	not
	  ^true

and: operand
  ^operand value
	and: operand
	  ^self

or: operand
  ^self
	or: operand
	  ^operand value

Los operand son objetos bloque, y como hemos visto en el módulo 5 les puedo enviar un mensaje value para ejecutarlos…

Por eso tener binding dinámico me permite tener polimorfismo: difiero hasta el último momento posible el enlace entre operador y operando.

Un lenguaje fuertemente tipado, ¿puede tener binding dinámico?
Sí, de hecho C++ y Java son fuertemente tipados y tienen binding dinámico:

Ejemplo en Java
```
Empleado empleado;
empleado = new EmpleadoJerarquico();   // es subclase de Empleado
…
empleado.getSueldo();
```

Lo único que me pide el compilador es que en la clase Empleado haya un método getSueldo(), pero la máquina virtual de Java va a ejecutar el código del objeto al que referencie la variable empleado (alguien puede cambiar en tiempo de ejecución la referencia a otro tipo de empleado).

Por eso para el compilador son dos cosas distintas:

- El chequeo de tipos (tipado fuerte/débil que exige que las variables estén asociadas a un tipo)
- El enlace entre operador y operando en tiempo de compilación (estático) / ejecución (dinámico)

En C++ por default el enlace es estático pero puedo definir funciones *virtuales* que habilitan el enlace dinámico.

SOBRECARGA NO IMPLICA BINDING DINÁMICO

El hecho de tener una función sobrecargada con parámetros de distinto tipo (o distinta cantidad de argumentos), ¿está hablando de un diferimiento entre operador y operando?
Si tenemos definidas dos funciones: `test(int)` y `test(char *)`, cuando invocamos desde el código cliente:

```
…
char c = test("prueba");
int v = 22;

test(v);
```

→ Se enlaza con la función test(char *)

→ Se enlaza con la función test(int)

En ambos casos sabemos en tiempo de compilación qué código va a ejecutar, en base a los tipos de los argumentos.

Por eso aquí no hay binding dinámico, y el concepto de sobrecarga no tiene ninguna asociación con el polimorfismo.

PARTE BONUS

ENVÍO MÚLTIPLE DE MENSAJES A UN MISMO OBJETO

Si necesitamos enviarle muchos mensajes a un objeto, puede resultarnos engorroso tener que decirle:
Flor, acordate de llevarme los cds.
Flor, ¿corregiste los parciales?
Flor, pedile a bedelía que nos cambien el aula.
¡¡¡¡Flor, Flor, Flor, Flor!!!!!

Entonces cuando tenemos algo como:
consultora := Consultora new.
consultora nombre: 'ENRON'.
consultora domicilio: 'Nazca 929'.
consultora addEmpleado: Empleado new nombre: 'Dodine'.
Etc. etc.

También podemos resolverlo de esta manera:

```
consultora := Consultora new
        nombre: 'ENRON';
        domicilio: 'Nazca 929';
        addEmpleado: Empleado new nombre: 'Dodine';
        etc,.
```

¿Cuál es el riesgo?
Que el último método no nos devuelva el objeto. Supongamos que el método addEmpleado estuviera codificado de la siguiente manera:

```
addEmpleado: unEmpleado
  self empleados add: unEmpleado.
  ^unEmpleado
```

Consultora estaría referenciando a un objeto de la clase Empleado, en lugar de un objeto de la clase Consultora. Entonces para asegurarse de que consultora va a apuntar al objeto al cual lo sentó en una mesa de café y le habló todo lo que le quiso hablar vamos a codificar un método que nos devuelva el mismo objeto receptor al cual le hablamos. Ese método se llama yourself:

```
yourself
   ^self
```

Y así hacemos:

```
consultora := Consultora new
        nombre: 'ENRON';
        domicilio: 'Nazca 929';
        addEmpleado: (Empleado new nombre: 'Dodine');
        …;
        …;
        yourself.
```

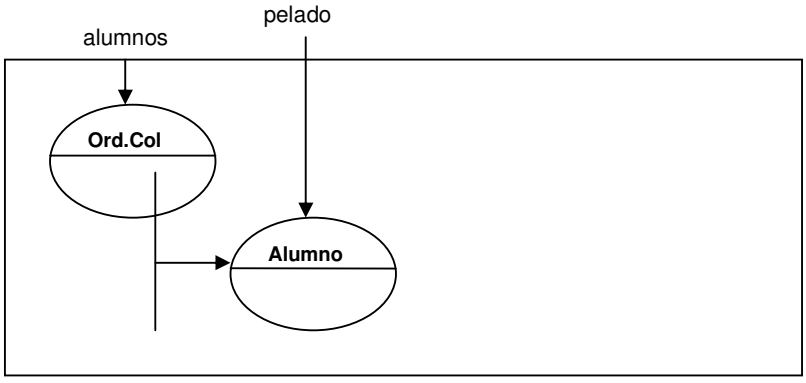
← nos devuelve el objeto que creamos originalmente (que fue el objeto receptor de todos los mensajes).

Esto es propio de Smalltalk, pero les va a permitir ser más claros cuando generen los scripts de prueba para el TP… ejem ejem, o sea…

IGUALDAD E IDENTIDAD

Cuando vimos los Sets, dijimos que era una colección que no tenía elementos duplicados. Ahora bien, ¿cómo sabemos que un elemento ya está en la colección?
Tenemos dos formas de preguntar: por identidad o por igualdad.

Identidad: decimos que dos objetos son idénticos si son el mismo objeto. Dentro del ambiente podemos tener dos referencias diferentes al mismo objeto, entonces si evalúo:
pelado == alumnos first
El resultado es true si:

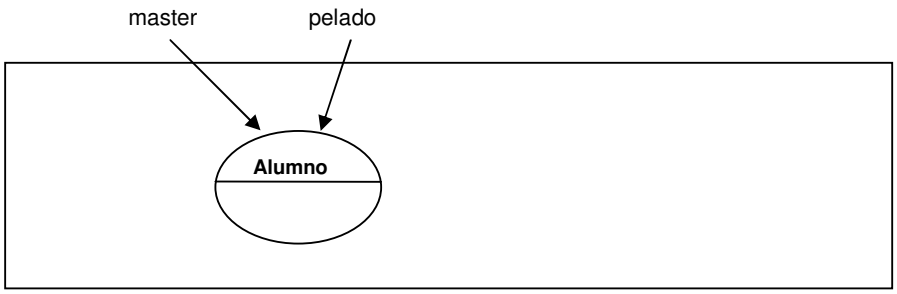


La identidad entre variables se da cuando referencian al mismo objeto. Un ejemplo más simple es:

```
pelado := Alumno new.
```

master := pelado.

Aquí está claro que pelado y master referencian al mismo objeto.



pelado == master

Igualdad: por defecto dos objetos son iguales si son el mismo objeto. Este "por defecto" nos lleva a mirar la definición del método = (igual) y el == (igual igual o identidad) en #Object:

== comparand

"Answer whether the <Object>, comparand, is the same, identical, object as the receiver.
The primitive should NOT fail.

N.B. This implementation cannot be overridden, and is never, in fact, received
(unless #perform'd), because #== is inlined by the Compiler."

<primitive: 110>
^self primitiveFailed

= comparand

"Answer whether the receiver and the <Object>, comparand, are considered equivalent (in whatever respect is appropriate for the class of the receiver).
By default two objects are equal if they are identical. This is the standard Smalltalk definition, though a better one is:

Two objects are equal if they are of the same species and
their contents are equal.

but this is a recursive definition, and is quite slow to implement in general."
<primitive: 110>
^self primitiveFailed

O sea, ambas implementaciones son iguales, pero resulta interesante lo que dice cada uno de los comentarios. En el contrato del método == de #Object, se lee clarito: "este método no se puede redefinir".

En general el igual nos alcanza como está definido, o sea

- si tengo dos alumnos con el mismo legajo…
- o tengo dos personas con el mismo DNI…
- o tengo dos materias que se llaman igual…

…hay algo que no está bien.

No obstante, sí puede pasar que una persona viva en un domicilio, abstracción que representamos con un objeto Domicilio que contiene un String llamado dirección. Tenemos a Chiara y Melina que viven en 'Juan Agustín García 2910'. Pero como los Strings los genera el Smalltalk, no puedo asegurar que se trate de

exactamente esa instancia (la misma para las dos). Entonces cuando pregunto si el domicilio de Chiara y Melina coincide, no está bueno confiar en el ==; más bien es preferible preguntar por =.

Necesita que el objeto domicilio sea el mismo, es poco confiable	*Aquí no dependemos de que sea exactamente el mismo objeto*
#Persona viveEnMismoDomicilioQue: otraPersona ^domicilio == otraPersona domicilio	**Persona** viveEnMismoDomicilioQue: otraPersona ^domicilio = otraPersona domicilio

Entonces en Domicilio redefinimos el = de esta manera:

```
(#Domicilio)
= anObject
   ^direccion = anObject direccion
```

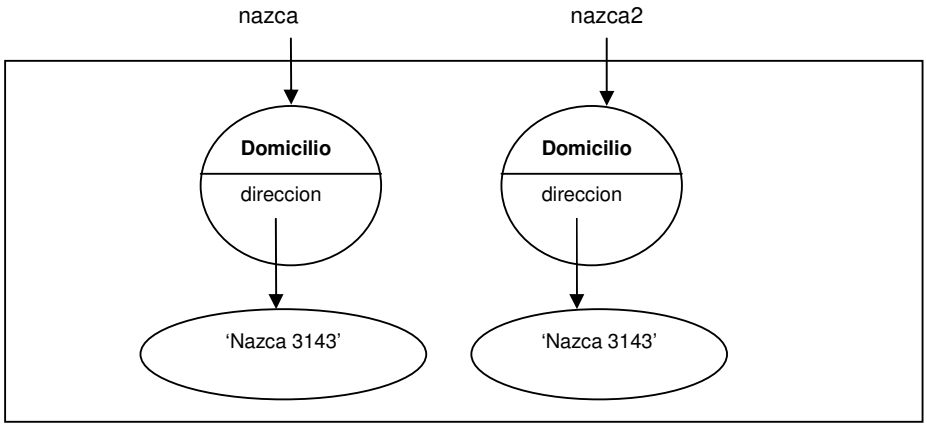
Ahora si yo quiero comparar un domicilio con otro objeto que no entienda dirección (pero que sea polimórfico con el domicilio en algún otro contexto), esto se rompe en tiempo de ejecución. ¿Qué tiene de malo? Quizás yo necesite considerar en una misma colección ambos objetos, entonces tengo que poder soportar que no se rompa. Para eso vamos a preguntar previamente si ambos objetos son de la misma "especie" (tipo):

```
(#Domicilio)
= anObject
   ^self species == anObject species and: [ direccion = anObject direccion ]
```

```
nazca := Domicilio new.
nazca direccion: 'Nazca 3143'.
nazca2 := Domicilio new.
nazca2 direccion: 'Nazca 3143'.
```

nazca == nazca2 → devuelve falso.
nazca = nazca2 → devuelve verdadero.

¿Y cómo están los objetos en memoria?
Si evalué línea por línea de a uno…



En general, cuando uno redefine el = también debe redefinir otro método que es el hash respetando el siguiente contrato: "si dos objetos son iguales, su número hash debe ser el mismo". Esto vale especialmente

para las colecciones Set, que usan tanto el = como el hash para almacenar y recuperar a los objetos que componen dicha colección.

Vemos la implementación de Object:

```
hash
    "Answer the <integer> hash value for the receiver. By default use the identity hash
    assigned at object creation time, which is temporally invariant.
    Equivalent objects (i.e. those that answer true for #=) MUST answer the same hash value, in order
    that they can be stored and retrieved successfully from hashed collections. Therefore, classes
    which reimplement either #= or #hash, will probably need to reimplement both."

    <primitive: 75>
    ^self primitiveFailed
```

Entonces, ¿cómo defino el hash para domicilio?
Si dos objetos son iguales, tienen que tener el mismo número de hash:

```
= anObject
  ^self species == anObject species and: [ direccion = anObject direccion ]
```

Si la igualdad depende de la dirección, entonces está bueno que el hash del domicilio también dependa del hash de su dirección:

```
hash
  ^direccion hash
```

Entonces si dos instancias de domicilio tienen la misma dirección:

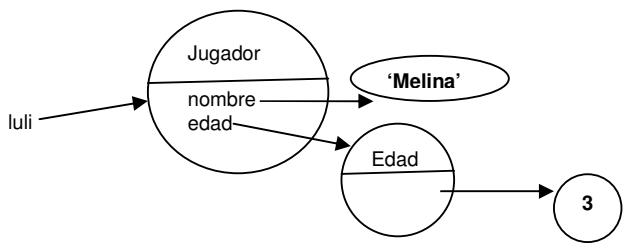
a) son iguales
b) comparten el mismo hash, que es justamente lo que me pide el contrato de Set.

Eso me permite hacer:
guiaDirecciones := Set new.
guiaDirecciones add: (Domicilio new direccion: 'Nazca 3143').
guiaDirecciones add: (Domicilio new direccion: 'Nazca 3143').
guiaDirecciones size → me devuelve 1, ya que el segundo alumno era "igual" al primero que agregué.

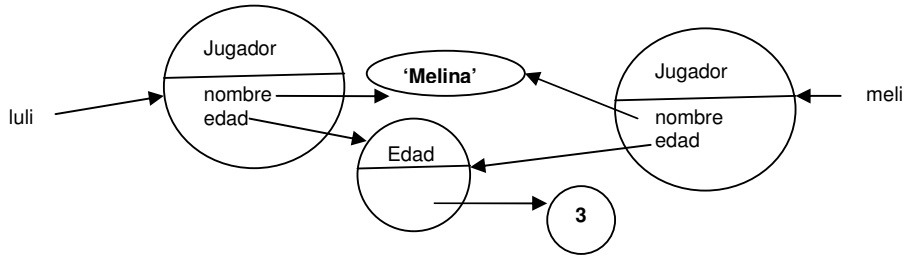
COPIA SUPERFICIAL Y COPIA PROFUNDA

En muchas ocasiones necesitaremos generar una copia de un objeto. Tenemos dos alternativas:

Shallow copy: copia superficial
Genero una nueva copia del objeto original y conservo las mismas referencias para sus atributos.
Si tenemos:



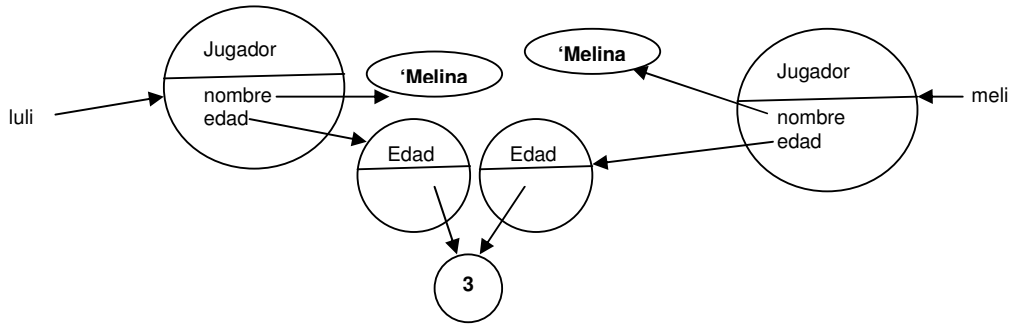
meli = luli shallowCopy
o
meli = luli copy (por defecto la copia es superficial)
Nos queda:



Deep copy

Se generan nuevas copias del objeto original y de sus atributos (en forma recursiva, anidando los n niveles de profundidad; el problema es identificar referencias circulares entre objetos para no ciclar indefinidamente).

meli = luli deepCopy



Nota: no siempre es posible obtener una nueva copia de todos los objetos (si el nombre fuera un Symbol, no se generaría una nueva copia).

¿Para qué necesitamos hacer copias de un objeto?

- Cuando no queremos que el objeto original tenga efecto colateral (porque queremos hacer una simulación que no afecte al objeto original, o porque tenemos que eliminar elementos de una colección mientras los vamos recorriendo, o porque queremos sumarizar totales sin afectar el objeto original)
- Cuando la creación del objeto es muy compleja o costosa, nos conviene copiar un objeto "prototipo" en lugar de generarlo desde cero. Ej: una imagen, un inventario de materiales, un juego de ajedrez, etc. (reemplazo toda la lógica que sigue a Ajedrez new por un juegoDeAjedrez copy).
- Cuando quiero transportar un objeto de un ambiente a otro (eso se llama "distribución")