Clase práctica Objetos II

1er cuatrimestre 2006

Estructuras de control en Smalltlak

En los lenguajes de programación imperativos existen, como parte del lenguaje, estructuras de control de flujo. Por ejemplo, el if:

```
If (condición) {
      sentencias
}
```

Al programar en Smalltalk, el equivalente a esa estructura tiene la forma:

```
condicion ifTrue: [sentencias].
```

La razón de esta forma particular del if proviene de una decisión de diseño del Smalltalk: todo es un objeto. Veamos en detalle las consecuencias de esta decisión.
Para comprender la construcción anterior, es necesario considerar, en primer lugar, que los Boolean también son objetos. En particular, la clase **Boolean** tiene dos subclases, **True** y **False**, cuyas únicas instancias son, respectivamente, las constantes **true** y **false**.
Cuando evaluamos una expresión booleana en Smalltalk, el resultado es alguna de esas constantes. Por ejemplo si evaluamos:

```
a := 3.
a < 5
```

obtenemos *true*. (Recordemos además que *a* es un objeto de clase *SmallInteger* y que < es un mensaje que *SmallInteger* sabe responder).
Podemos deducir que *ifTrue:* debe ser un mensaje keyword que la clase *True* sabe responder. El argumento del mensaje no puede ser otra cosa que un objeto. Asi que el argumento *[sentencias]* también es un objeto. Este tipo de objetos, llamados bloques o clausuras (son instancias de BlockClosure), son porciones de código suspendido, sin evaluar. Responden, en particular, al mensaje *value*, cuyo efecto es evaluar el código suspendido.
Antes de ver los bloques en mayor profundidad, analicemos un ejemplo de if. Consideremos que True define su método ifTrue: como la evaluación del argumento, mientras que False no hace nada.
Cuando evaluamos:

```
anObject < 5 ifTrue: [Transcript show: 'menor a cinco'].
```

lo que ocurrre es lo siguiente:

1. Se envia el mensaje < al objeto *anObject* con el argumento 5. El resultado (que dependerá de quién sea *anObject*) será la constante *true* o *false* (instancias de *True* o *False* respectivamente).
2. Al resultado (*true* o *false*) se le envia el mensaje *ifTrue:* con *[Transcript show: 'menor a cinco']* como argumento
3. Si el resultado de *anObject* < 5 fue true, el método *ifTrue:* definido en *True* indica que se evalúe el código entre corchetes.
4. Si el resultado fue *false*, el método *ifTrue:* definido en *False* no hace nada, el bloque no se evaluará y no habrá ningún efecto, tal como se espera.

Hasta aquí hemos visto lo que parece una solución ingeniosa para una estructura del tipo if utilizando solamente objetos y mensajes (veremos más adelante que en Smalltalk el flujo de ejecución se controla con objetos y mensajes. No hay construcciones sintácticas especiales para esto).
Sin embargo, la introducción de los bloques en Smalltalk permite resolver muchas otras situaciones de programación. Veamos primero algunos detalles de los bloques.

Late binding y polimorfismo

Notemos en el ejemplo anterior que ifTrue: tiene comportamientos distintos en True y en False. Esta cualidad de poder definir distintas respuestas a un mensaje de acuerdo al receptor se denomina **polimorfismo**.
Para que funcione el ejemplo anterior, es indispensable que la resolución del método que responde al mensaje se haga en tiempo de ejecución, porque el resultado de (anObject < 5) es desconocido en tiempo de compilación. Este mecanismo de resolución tardía del método que responde a un mensaje se llama **late-binding** y es lo que permite el polimorfismo en Smalltlak.

Bloques

- Los bloques son secuencias de instrucciones cuya ejecución es diferida, es decir, **el código que representan podría no ser ejecutado nunca**.
- Se notan escribiendo el código que los componen entre [].

"Todo es un objeto" (III)

- Los bloques son objetos. Por lo tanto pueden ser asignados a variables, pasados como parámetros o devueltos como resultado
- La evaluación de un bloque se realiza enviándole el mensaje **value**

Ejemplos:

Expresión	***Descripción***
[Ballena new deciAlgo]	Un bloque de código. El código dentro del bloque no ha sido evaluado todavía.
[Ballena new deciAlgo] value	Evalua el código dentro del bloque
miBloque := [Ballena new deciAlgo]. miBloque value.	Se puede almacenar en una variable la referencia a un bloque
b := [:a | a deciAlgo]. b value: Ballena new. b value: Perro new.	Los bloques pueden recibir parámetros. En ese caso se los evalua con los mensajes **value:**, **value:value:**, etc
b := [:a | |nombre| nombre := a nombre. Transcript show: nombre; cr.]. b value: (Humano new nombre: 'Juan Perez').	Los bloques pueden tener variables temporales que se declaran entre | |
p := Humano new nombre: 'Felipe'. [Transcript show: (p nombre);cr] value.	Dentro de un bloque, las variables en scope son las variables del scope en el que se definen los bloques. El binding de valores a variables se produce en tiempo de ejecución (el envío del mensaje value).
[Float pi * 4. 3 + 4] value	Los bloques devuelven el resultado de la evaluación de la última sentencia. En este caso, 7.

Control del flujo de ejecución

Evaluación condicional

- La clase Boolean y sus subclases definen los métodos:
    - **ifTrue:**
    - **ifTrue:ifFalse:**
    - **ifFalse:**

- **AND** y **OR** tambien son métodos de (las subclases de) Boolean:
    - **and: aBlock** recibe un bloque que evalua sólo si es necesario
    - **or: aBlock** recibe un bloque que evalua sólo si es necesario
    - **& aBoolean**
    - **| aBoolean**

  En todos los casos los argumentos de estos mensajes son bloques que se evaluarán dependiendo del receptor del mensaje (recordar que True y False son subclases de Boolean).

Ejemplos:

Expresión	***Descripción***
a := 5. a < 17 ifTrue: [Transcript show: 'menor'] ifFalse: [Transcript show: 'mayor']	La expresión "a < 17" devuelve un objeto Boolean. Si el objeto resultante es **true**, se ejecutará el primero bloque y si en cambio el objeto resultante es **false** se ejecutará el segundo bloque
a := 4. paridad := a odd ifTrue: [1] ifFalse: [0].	ifTrue:, ifFalse, etc. ***devuelven*** el resultado de evaluar el bloque. En Este caso *paridad* tendrá el valor 0.
a := 4. paridad := a odd ifTrue: [1].	Cuando no se devuelve ningún valor, se devuelve la constante especial ***nil*** (instancia de UndefinedObject). En este caso *paridad* tendrá el valor ***nil***. nil también es un objeto. Sabe responder, entre otros: isNil, que devuelve true (los demas objetos devuelven false)

Ciclos

- La clase Number define los métodos:
    - **to:do:** Que recibe otro número y un bloque con un 1 parámetro
    - **to:by:do:** Que recibe dos números (el segundo es el incremento) y un bloque con 1 parámetro
- La clase Integer define además el método:
    - **timesRepeat:** Que recibe un bloque sin parámetros

Ejemplos:

Expresión	***Descripción***
k := 0. 6 timesRepeat: [k := k + 1].	Suma 6 veces 1 la variable k
l := Loro new. 4 timesRepeat: [l deciAlgo].	Le pide 4 veces al loro l que diga algo
k := 1. 1 to: 6 do: [:i | k := k * i]. k.	Multiplica los números del 1 al 6, es decir, calcula 6!

k := 1. 1 to: 10 by: 2 do: [:i | k := k + i]. k	Suma todos los números impares entre 1 y 10

Ciclos tipo 'while'

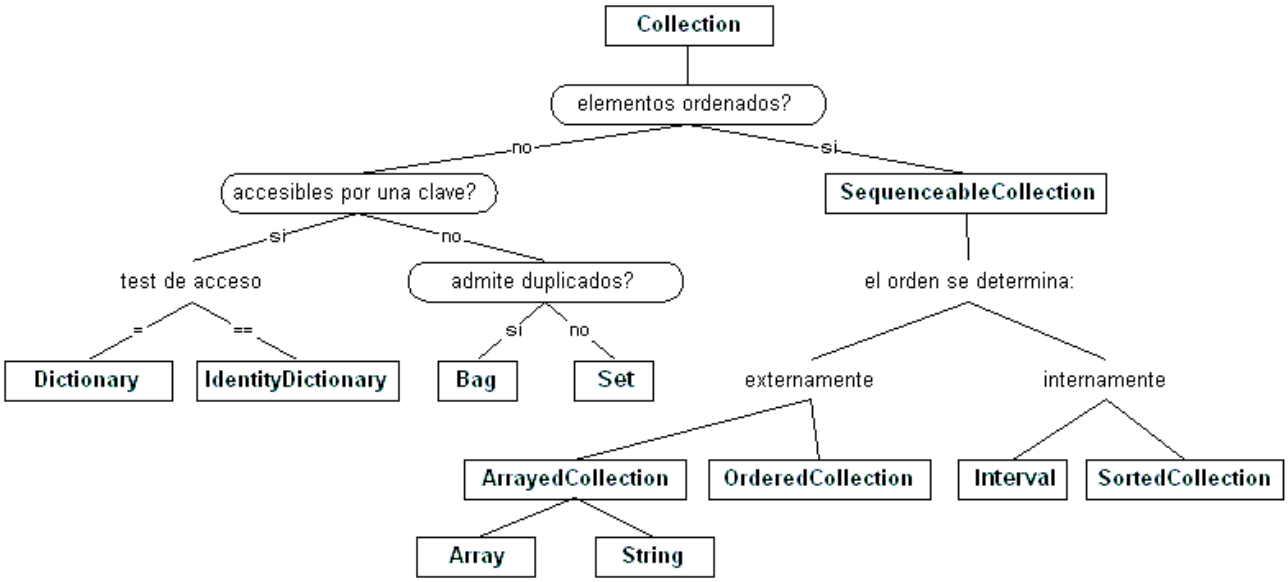
- BlockClosure define los métodos
    - **whileTrue:** Que recibe un bloque sin parámetros que será ejecutado mientras el bloque receptor devuelva true
    - **whileFalse:** Que recibe un bloque sin parámetros que será ejecutado mientras el bloque receptor devuelva false
    - **whileTrue** Equivale a whileTrue: []
    - **whileFalse** Equivale a whileFalse: []
    - **repeat** Que evalua el bloque receptor hasta que este interrumpa la evaluación por la fuerza

Ejemplos:

Expresión	***Descripción***
k := 0. [k < 6] whileTrue: [k := k + 1]. k.	Suma a k el número 1 hasta que k valga 6
k := 0. [k >= 6] whileFalse: [k := k + 1]. k.	Idem pero escrito de otra forma

Colecciones

Colecciones básicas



Nota: Las colecciones de Smalltalk son ***heterogéneas***. Es decir, no es necesario que los elementos sean todos de la misma clase.

Operaciones básicas:

Colección	***Operaciones***	***Ejemplos***
Array	at: at:put: size	a := #(1 2 3 4) copy. a at: 1. a at: 2 put: $c; yourself. a size
Set	add: size includes:	s := Set new. s add: 1; add: 1; add: 2; add: $a; add: $a; yourself. s size. s includes: 1. s includes: 4.
Bag (conjunto con repeticiones)	add: size includes: occurrencesOf:	b := Bag new. b add: 1; add: 1; add: 2; add: $a; add: $a; yourself. b size. b includes: 1. b includes: 4. b occurrencesOf: 1. b occurrencesOf: 2.
Dictionary	at:put: at: includes: (contiene valor) includesKey: (contiene clave) size keys	d := Dictionary new. d at: 'hola' put: 'hello'. d at: 'perro' put: 'dog'. d at: 'pitufo' put: 'smurff'. d at: 'pitufo'. d includes: 'perro'. “false” d includesKey: 'perro'. “true” d includesKey: 'loro'.
Interval (intervalo de números)	includes: do:	in := Interval from: 3 to: 30 by: 5. in includes: 4. in includes: 8. in do: [:i | Transcript show: i displayString; cr].
OrderedCollection	add: at: at:put: first last addFirst: addLast: removeFirst: removeLast: indexOf:	oC := OrderedCollection new. oC add: 3; add: 2; add: 5; yourself. oC add: 4 after: 2; yourself. "OrderedCollection sirve para implementar pilas y colas" oC addFirst: 50; yourself. oC addLast: 400; yourself. oC removeLast; yourself.
SortedCollection	add: todos los de OrderedCollection	x := SortedCollection new. x add: 3; add: 2; add: 5; yourself. "se puede cambiar el criterio de ordenamiento" x sortBlock: [:a :b | a >= b]; yourself.

Enumeracion de elementos

do: aBlock	bloque con un parámetro evalua el argumento aBlock para cada uno de los elementos del receptor
select: aBlock	bloque con un parámetro, evalua en booleano evalua el argumento aBlock para cada uno de los elementos del receptor genera una nueva coleccion del mismo tipo que el receptor con

	únicamente los elementos para los cuales aBlock evalua en **true**
reject: aBlock	bloque con un parámetro, evalua en booleano evalua el argumento aBlock para cada uno de los elementos del receptor genera una nueva coleccion del mismo tipo que el receptor con únicamente los elementos para los cuales aBlock evalua en **false**
collect: aBlock	bloque con un parámetro, evalua en cualquier cosa evalua el argumento aBlock para cada uno de los elementos del receptor genera una nueva coleccion del mismo tipo que el receptor, y para cada elemento del receptor agrega el resultado de aplicarle aBlock
detect: aBlock	bloque con un parámetro, evalua en booleano evalua el argumento aBlock para cada uno de los elementos del receptor y devuelve el primer elemento para el cual aBlock evaluó en true. Si ninguno da true, error
detect: aBlock ifNone: exceptionBlock	Idem anterior, pero devuelve el resultado de evaluar exceptionBlock si no encuentra el elemento
inject: thisValue into: binaryBlock	bloque con dos parámetros, evalua en lo mismo que el primer parámetro el primer parámetro del bloque es el resultado de la evaluacion anterior, empezando por **thisValue**. el segundo parámetro es cada uno de los elementos del receptor devuelve el resultado de la última evaluacion

Ejemplos:

Expresión	***Descripción***
n := 1 to: 10. x := 0. n do: [:each | x := x + each]. x.	Suma los enteros en el intervalo [1, 10] usando do:
n inject: 0 into: [:acum :each | acum + each]	Lo mismo, pero usando inject:into:
c := OrderedCollection new. c add: (Cuenta saldo: 100); add: (Cuenta saldo: 200); add: (Cuenta saldo: 300); yourself. c collect: [:each | each saldo].	Construye una OrderedCollection con los saldos de cada una de las cuentas
c inject: 0 into: [:acum :each | acum + each saldo]	Suma los saldos de todas las cuentas
c detect: [:each | each saldo > 150].	Busca la primera cuenta que tenga saldo mayor a 150

Identidad e igualdad

- **Equivalencia** (==): testea si dos objetos son el mismo objeto (es decir, la misma instancia)
- **Igualdad** (=): testea la igualdad semántica. Por ejemplo 'hola' = 'hola' es true, pero 'hola' == 'hola' es false.
- La igualdad debe ser reimplementada por cada clase de forma apropiada
- La implementación default de = es ==

==	equivalencia
=	igualdad
~=	no igualdad
~~	no equivalencia

Caso especial: **nil**

isNil	testea si el receptor es **nil**	== nil

notNil	testea si el receptor no es **nil**	~~ nil

Hash

- **hash** devuelve un entero. Dos objetos iguales deben devolver el mismo valor. Dos objetos distintos podrían devolver el mismo valor o no.
- Cada vez que se redefine =, hay que redefinir hash de forma acorde

Clases abstractas

- Una clase abstracta es aquella que nunca tendrá instancias
- Se usan para reunir comportamiento y *expectativas* comunes a todas las subclases
- Ejemplos: ***Magnitude*** y ***Animal***

- Métodos abstractos:
  - En Smalltalk: *subclassResponsibility* o *implementedBySubclass*
  - En C++: *pure virtual*
  - En Java: *abstract*
- Una clase que tiene métodos abstractos debería ser abstracta. Esto no es obligatorio en Smalltalk, pero es considerado buen diseño. Puede hacerse obligatorio redefiniendo el new de modo de emitir un mensaje de error.
- Muchas veces, las clases abstractas también tienen comportamiento concreto (ejemplo ***Magnitude>>max:***)
- Los métodos de una clase abstracta pueden ser de 3 tipos:
  - *abstractos*: no están implementados y deben ser implementados por las subclases
    Magnitude>> hash
  - *base*: están implementados y son autocontenidos. No generan ninguna obligación a las subclases
    Reloj >> deciLaHora
  - *template*: están implementados pero usan métodos abstractos, con lo cual el implementador de la subclase debe "completar" el comportamiento pero sin sobreescribir el método completo
    Magnitude>>max:

Convenciones y buenas prácticas

- Los nombres compuestos de varias palabras se escriben poniendo mayúsculas al inicio de cada palabra, por ejemplo: *unaVariable*, *UnaClase*
- Si existe una variable de instancia llamada *unaVariable*, los métodos para acceder y cambiar la variable se llaman respectivamente *unaVariable* y *unaVariable*:
- Es buena práctica acceder a las variables de instancia siempre a través de sus métodos de acceso, incluso desde otros métodos del mismo objeto. Esto permite cambiar la implementación. Si quiere indicarse que no debe cambiarse la variable desde afuera del objeto, se definen los métodos de acceso en la categoría de métodos 'private'
- Es convencional el uso de ciertas categorías de métodos. Por ejemplo, *accessing*, *private*, *copying*, *converting*, *printing*. Revisar las clases del sistema para otros casos.
- Es cómodo para el debugging que todos los objetos tengan una forma adecuada de imprimirse. El método *printString* devuelve un string de representación; es respondido por todos los objetos (está definido en *Object*) de manera estándar y generalmente inútil (por omisión, las instancias de *Reloj* devuelven 'a Reloj'). Para redefinirlo basta redefinir *printOn*: en la clase de interés (ver implementaciones para ver cómo se hace)
- Como las variables y los argumentos no están tipados, se suele "tipar el nombre" de las variables temporarias, y a veces de los argumentos. Por ejemplo *anObject* sería una variable que puede contener cualquier objeto. Otros ejemplos: *aNumber*, *anOrderedCollection*, *exceptionBlock*, *aReloj*.