

Clase práctica: Smalltalk

(Parte 2: los objetos contraatacan)

Paradigmas de Lenguajes de Programación

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

7 de Junio de 2018

Temas del día

- ➊ Profundizando conceptos
- ➋ Igualdad y hash
- ➌ Metaprogramación

Revisitando el mensaje collect:

¿Qué devuelven las siguientes colaboraciones?

- `#hola collect: [:x | Unicode toUppercase: x]`.
- `(Interval from: 1 to: 5) collect: [:x | x*2]`.

Pista: los símbolos e intervalos son inmutables.

Revisitando el mensaje collect:

¿Qué devuelven las siguientes colaboraciones?

- `#hola collect: [:x | Unicode toUppercase: x]`.
- `(Interval from: 1 to: 5) collect: [:x | x*2]`.

Pista: los símbolos e intervalos son inmutables.

Veámoslo en el entorno.

Revisitando el mensaje `collect`:

¿Qué devuelven las siguientes colaboraciones?

- `#hola collect: [:x | Unicode toUppercase: x]`.
- `(Interval from: 1 to: 5) collect: [:x | x*2]`.

Pista: los símbolos e intervalos son inmutables.

El mensaje `species`

Las clases *Interval* y *ByteSymbol* redefinen el método `species` para poder responder a `collect`:

```
Interval >> species
```

```
^Array.
```

```
ByteSymbol >> species
```

```
^ByteString.
```

Implementemos detectMin:

Volviendo al ejercicio de la clase pasada...

Implementemos detectMin:

Volviendo al ejercicio de la clase pasada...

#detectMin:

Agregar a la clase Collection un método con la siguiente interfaz:

`detectMin: aBlock`

- `aBlock` es un bloque con un parámetro de entrada cuya evaluación devuelve un número.
- El método debe evaluar el bloque en todos los elementos de la colección receptora, y devolver el elemento que minimiza el valor obtenido.
- Se asume que la colección receptora no está vacía.

Implementemos detectMin:

Volviendo al ejercicio de la clase pasada...

#detectMin:

Agregar a la clase Collection un método con la siguiente interfaz:

`detectMin: aBlock`

- aBlock es un bloque con un parámetro de entrada cuya evaluación devuelve un número.
 - El método debe evaluar el bloque en todos los elementos de la colección receptora, y devolver el elemento que minimiza el valor obtenido.
 - Se asume que la colección receptora no está vacía.
-
- ¿Cómo inicializar un primer valor?
 - ¿Funciona para Set?

Implementemos detectMin:

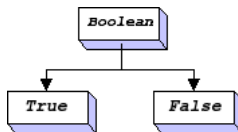
Solución según Pharo:

```
detectMin: aBlock
  | minElement minValue |
  self do: [:each | | val |
    minValue ifNotNil: [
      (val := aBlock value: each) < minValue ifTrue: [
        minElement := each.
        minValue := val]]
    ifNil: ["first element"
      minElement := each.
      minValue := aBlock value: each]].
  ^minElement
```

¿Pero... cómo se implementa el ifTrue:?

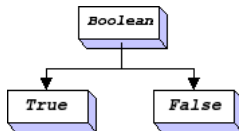
¿Pero... cómo se implementa el ifTrue?:?

Recordar: *Boolean* tiene dos subclases.



¿Pero... cómo se implementa el ifTrue:?

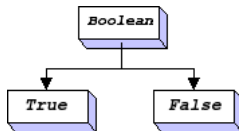
Recordar: *Boolean* tiene dos subclases.



```
True >> ifTrue: unBloque  
^unBloque value.
```

¿Pero... cómo se implementa el ifTrue:?

Recordar: *Boolean* tiene dos subclases.



```
True >> ifTrue: unBloque
```

```
^unBloque value.
```

```
False >> ifTrue: unBloque
```

```
^nil.
```

Otros métodos de *Boolean*

- `ifFalse:`
- `ifTrue:ifFalse:`
- `&`
- `|`
- `and:`
- `or:`
- `not`

Otros métodos de *Boolean*

- `ifFalse:`
- `ifTrue:ifFalse:`
- `&`
- `|`
- `and:`
- `or:`
- `not`

¿Por qué los booleanos no entienden el mensaje `whileTrue:?`

Otros métodos de *Boolean*

- `ifFalse:`
- `ifTrue:ifFalse:`
- `&`
- `|`
- `and:`
- `or:`
- `not`

¿Por qué los booleanos no entienden el mensaje `whileTrue:?`
¿Qué objetos lo entienden?

Jugando con clausuras

Agregar a la clase BlockClosure el método de clase
generarBloqueInfinito que devuelve un bloque b1 tal que:

b1 value devuelve un arreglo de 2 elementos #(1 b2)

b2 value devuelve un arreglo de 2 elementos #(2 b3)

⋮

b_i value devuelve un arreglo de 2 elementos #(i b_{i+1})

Igualdad

¿Valen las siguientes comparaciones?

- `1 == 1`

Igualdad

¿Valen las siguientes comparaciones?

- `1 == 1`
- `1 = 1`

Igualdad

¿Valen las siguientes comparaciones?

- `1 == 1`
- `1 = 1`
- `c1 := Bag with: 1.`
`c2 := Bag with: 1.`
`c1 == c2`

Igualdad

¿Valen las siguientes comparaciones?

- `1 == 1`
- `1 = 1`
- `c1 := Bag with: 1.`
`c2 := Bag with: 1.`
`c1 == c2`
- `c1 = c2`

Igualdad

¿Valen las siguientes comparaciones?

- `1 == 1`
- `1 = 1`
- `c1 := Bag with: 1.`
`c2 := Bag with: 1.`
`c1 == c2`
- `c1 = c2`

Si se redefine `#=`, entonces se debe redefinir `#hash`. Sino Set y Dictionary no van a operar correctamente.

$$a = b \Rightarrow \text{hash}(a) = \text{hash}(b)$$

Metaprogramación

Todo es un objeto

- Las clases, mensajes y métodos no escapan.
- Podemos manipularlos dentro de nuestros programas.
- ¿Para qué puede servir esto?

Metaprogramación

Todo es un objeto

- Las clases, mensajes y métodos no escapan.
- Podemos manipularlos dentro de nuestros programas.
- ¿Para qué puede servir esto?

Ya vimos un ejemplo: ¿qué ocurriría cuando le mandaba un mensaje al objeto equivocado?

Ejercicio

Definir los objetos necesarios para que esto funcione:

```
|content settings|  
content  := Dictionary newFromPairs: #(foo bar).  
settings := Settings withContent: content.
```

```
settings foo          (retorna #bar)  
settings foo: #baz  
settings foo          (retorna #baz)
```

Pista: ¿qué devuelve (Message selector: #foo) arguments?

Ejercicio

Definir una clase cuyas instancias hagan lo siguiente al recibir un mensaje de la forma `obj foo: bar`:

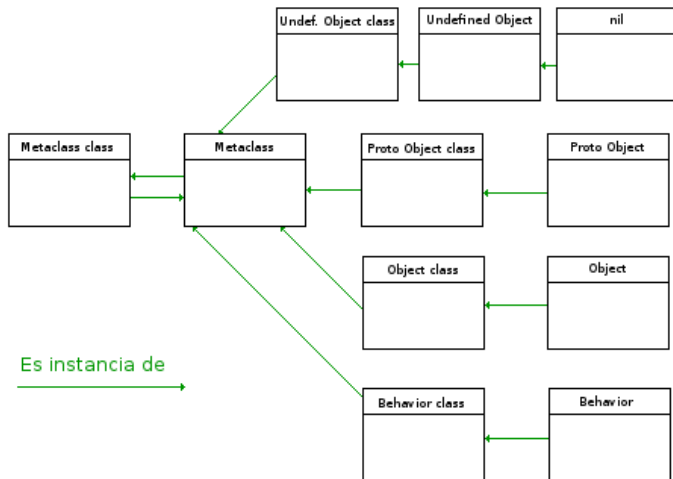
- agregar la variable de instancia `foo` con valor inicial `bar` (asumir que solo se observa mediante el `getter`),
- agregar métodos `foo` y `foo:` para obtener y redefinir el valor de la variable.

Ejemplo:

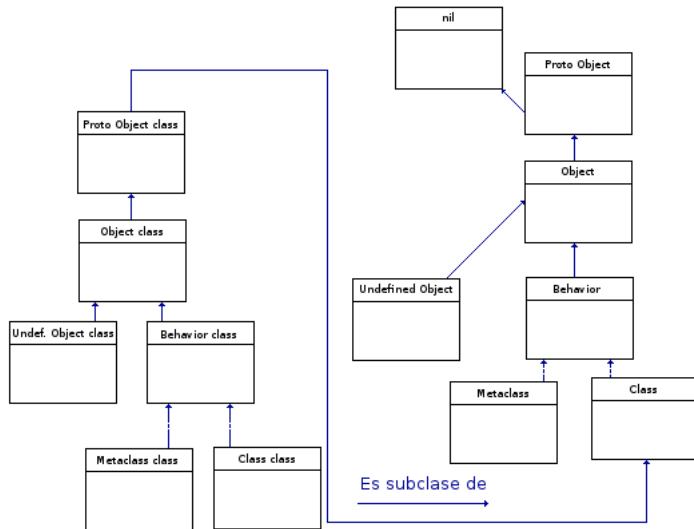
```
aGenerator := CodeGenerator new.  
aGenerator intValue: 10; intValue.
```

devuelve 10.

El metamodelo (instancias y clases)



El metamodelo (subclases y superclases)



2do recuperatorio, 1er cuatrimestre de 2015

Se extendió la clase `OrderedCollection` con una subclase `AutoMapList`, la cual, al recibir un mensaje, primero intenta responderlo como lo haría una `OrderedCollection`, y si no lo entiende, lo delega a sus elementos, siempre y cuando todos puedan responderlo. De otra forma, falla como fallaría habitualmente. En otras palabras, es una lista que puede 'mapear' mensajes a sus elementos de manera implícita.

- Definir el método `AutoMapList >> respondsTo: aSelector` que devuelva verdadero si la lista o todos sus elementos responden al mensaje con selector `aSelector`.
- Definir lo necesario para que los mensajes se redirijan cuando corresponda. Por ejemplo:
 - ▶ `(AutoMapList with: 1 with: 4) size` devuelve 2.
 - ▶ `(AutoMapList with: 1 with: 4) + 10` devuelve una `AutoMapList` que contiene al 11 y al 14.
 - ▶ `(AutoMapList with: 1 with: 4) lala` produce una excepción.

Solución

```
AutoMapList>>respondsTo: aSelector  
  ^super respondsTo: aSelector or: [self allRespondTo: aSelector]
```

```
AutoMapList>>doesNotUnderstand: aMessage  
  ^(self allRespondTo: aMessage selector)  
    ifTrue: [ self collect: [ :each | aMessage sendTo: each ] ]  
    ifFalse: [ super doesNotUnderstand: aMessage ]
```

```
AutoMapList>>allRespondTo: aSelector  
  ^self allSatisfy: [ :each | each respondsTo: aSelector ]
```

Último ejercicio: clases unitarias

En este ejercicio incorporaremos al lenguaje Smalltalk la capacidad de crear clases unitarias (en inglés *singleton*). Así como cada clase puede crear subclases de sí misma al recibir el mensaje `subclass:instancaVariableNames:classVariableNames:package:`, ahora también podrá crear clases con una única instancia.

Implementar el método de instancia `singletonSubclass: nombreSubclase` para la clase `Class`.

El comportamiento esperado es el siguiente: cuando una clase reciba el mensaje `singletonSubclass:` con un símbolo, debe crear una subclase de sí misma cuyo nombre sea el símbolo pasado como parámetro. Además, la nueva clase deberá redefinir el método `new`, de manera que si ya existe una instancia no cree una nueva, y en cambio devuelva la instancia ya existente.

Solución

```
Class>>singletonSubclass: aClassName
| subclass |
subclass := self subclass: aClassName
            instanceVariableNames: ''
            classVariableNames: 'instance'
            package: self package name.
subclass class compile: 'new
    instance ifNil:[instance := super new].
    ^instance'.
^subclass
```


FIN

```
self tengoDudas whileTrue: [self consultar].  
^clase fin.
```