Conceptes Avançats de Programació

Reflexió en Smalltalk

Basat en el curs

Dynamic Object Oriented Programming with Smalltalk

del Prof. Oscar Nierstrasz, Universitat de Berna

https://www.iam.unibe.ch/scg/svn_repos/Lectures/Smalltalk/

Jordi Delgado (jdelgado@cs.upc.edu)

CAP: Reflexió en Smalltalk

Abans de començar a tractar la reflexió en Smalltalk cal entendre com és que les classes també són objectes... i si ho són, de qui són instàncies?

Reifica el metamodel: *Un sistema reflexiu té un model del seu propi metamodel*

CAP: Reflexió en Smalltalk

Pròleg:

- **Les Metaclasses en 7 parts**

- **Classes indexades**

- **Variables de classe-instància**

- **Variables de classe**

Part del material és cortesia d'Stéphane Ducasse

CAP: Reflexió en Smalltalk

<u>Pròleg</u>:

- ***Les Metaclasses en 7 parts***

- Classes indexades

- Variables de classe-instància

- Variables de classe

CAP: Reflexió en Smalltalk

Les metaclasses en 7 parts:

1- Tot objecte és instància d'una classe

2- Tota classe hereta eventualment d'**`Object`**

3- Tota classe és instància d'una metaclasse

4- La jerarquia de metaclasses és equivalent a la jerarquia de classes

5- Tota metaclasse hereta de **`Class`** i **`Behavior`**

6- Tota metaclass és instància de **`Metaclass`**

7- La metaclasse de **`Metaclass`** és instància de **`Metaclass`**

Adaptat de Goldberg & Robson, *Smalltalk-80 — The Language*

CAP: Reflexió en Smalltalk

Les metaclasses en 7 parts:

1- ***Tot objecte és instància d'una classe***

2- Tota classe hereta eventualment d'**`Object`**

3- Tota classe és instància d'una metaclasse

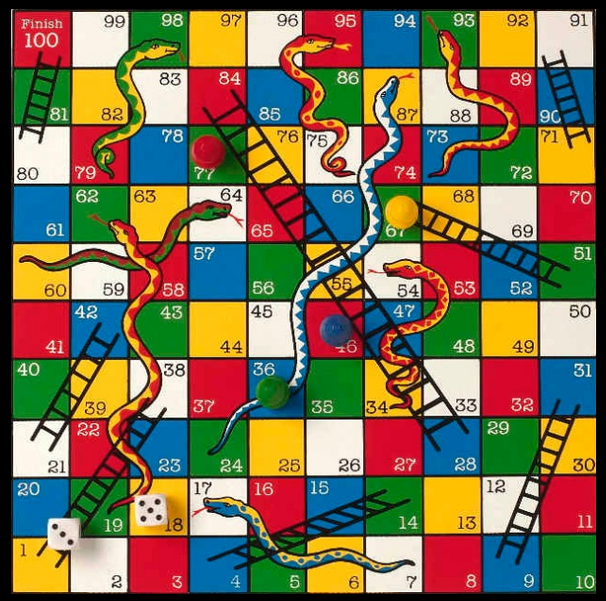
4- La jerarquia de metaclasses és equivalent a la jerarquia de classes

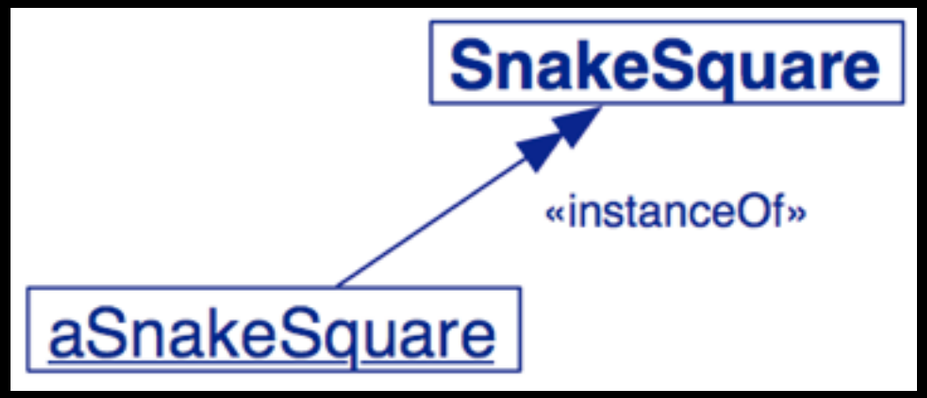
5- Tota metaclasse hereta de **`Class`** i **`Behavior`**

6- Tota metaclass és instància de **`Metaclass`**

7- La metaclasse de **`Metaclass`** és instància de **`Metaclass`**

CAP: Reflexió en Smalltalk





Recordeu el joc dels *Snakes and Ladders*...

CAP: Reflexió en Smalltalk

Les metaclasses en 7 parts:

1- Tot objecte és instància d'una classe

2- ***Tota classe hereta eventualment d'`Object`***

3- Tota classe és instància d'una metaclasse

4- La jerarquia de metaclasses és equivalent a la jerarquia de classes

5- Tota metaclasse hereta de **`Class`** i **`Behavior`**

6- Tota metaclass és instància de **`Metaclass`**

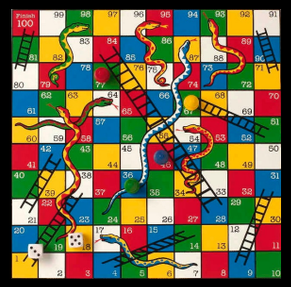
7- La metaclasse de **`Metaclass`** és instància de **`Metaclass`**

Adaptat de Goldberg & Robson, *Smalltalk-80 — The Language*

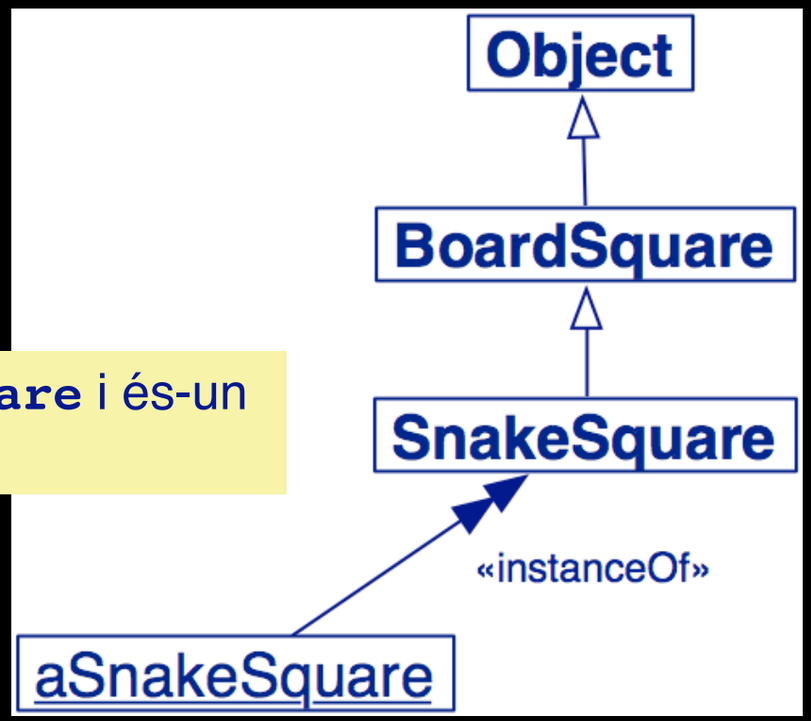
CAP: Reflexió en Smalltalk

Tot objecte **`és-un`** (*is-an*) **`Object`**

La classe de cada objecte hereta d'**`Object`**

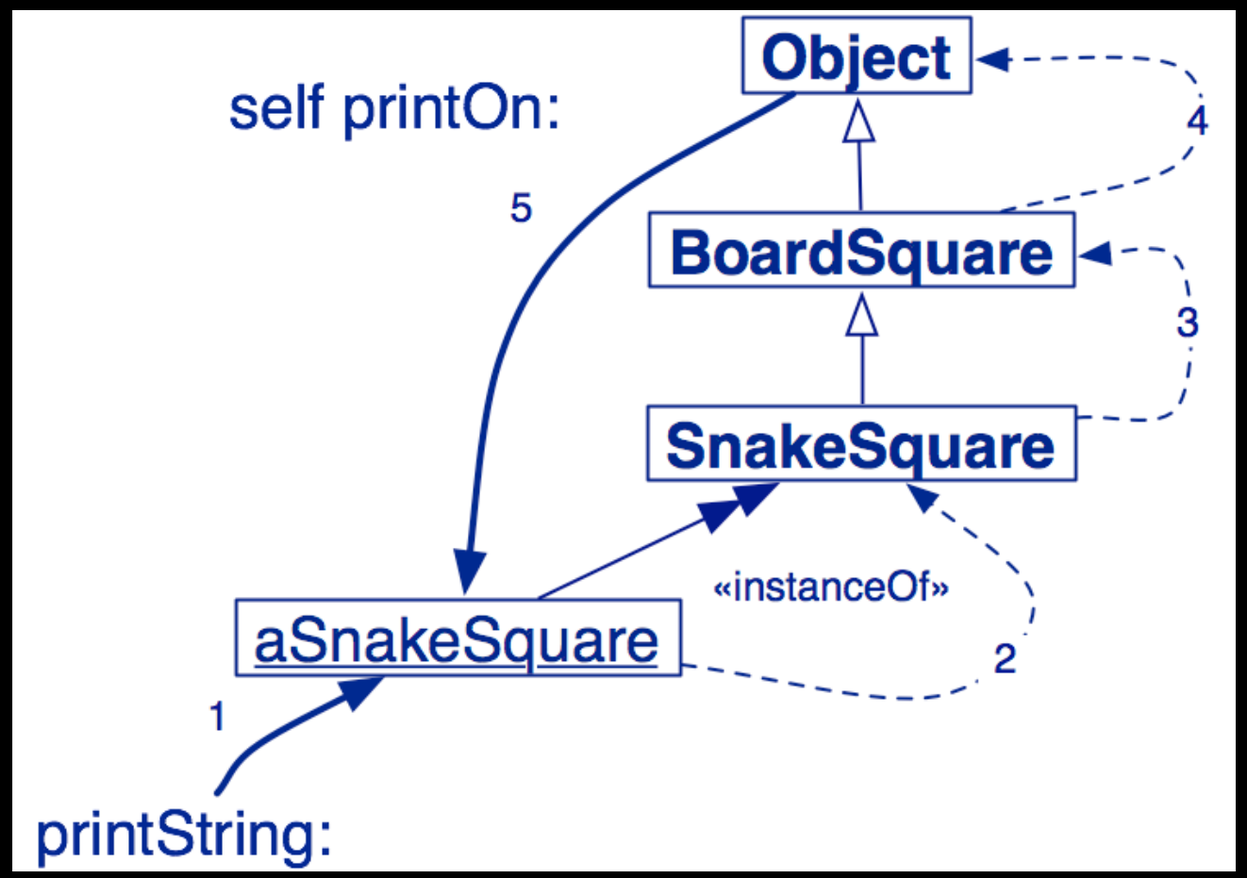


`aSnakeSquare` és-un **`SnakeSquare`** i és-un **`BoardSquare`** i és-un **`Object`**



CAP: Reflexió en Smalltalk

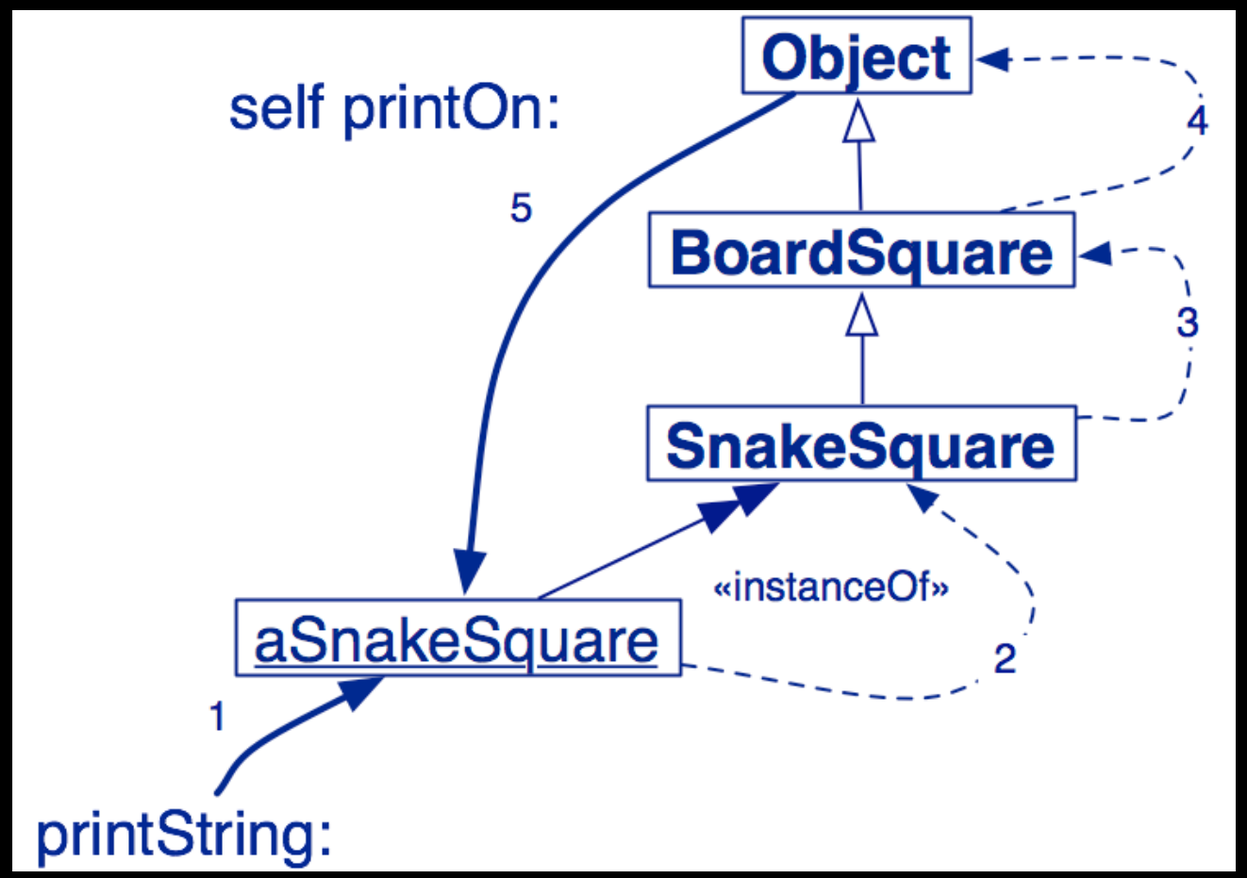
Quan un objecte rep un missatge, el mètode es busca al diccionari de mètodes de la seva classe, i, si cal, a les seves superclasses, fins arribar a **`Object`**.



CAP: Reflexió en Smalltalk

Responsabilitats d'**`Object`**:

- representa el comportament comú a tots els objectes (gestió d'errors, etc.).
- totes les classes haurien d'heretar d'**`Object`**



CAP: Reflexió en Smalltalk

Les metaclasses en 7 parts:

1- Tot objecte és instància d'una classe

2- Tota classe hereta eventualment d'**`Object`**

3- ***Tota classe és instància d'una metaclasse***

4- La jerarquia de metaclasses és equivalent a la jerarquia de classes

5- Tota metaclasse hereta de **`Class`** i **`Behavior`**

6- Tota metaclass és instància de **`Metaclass`**

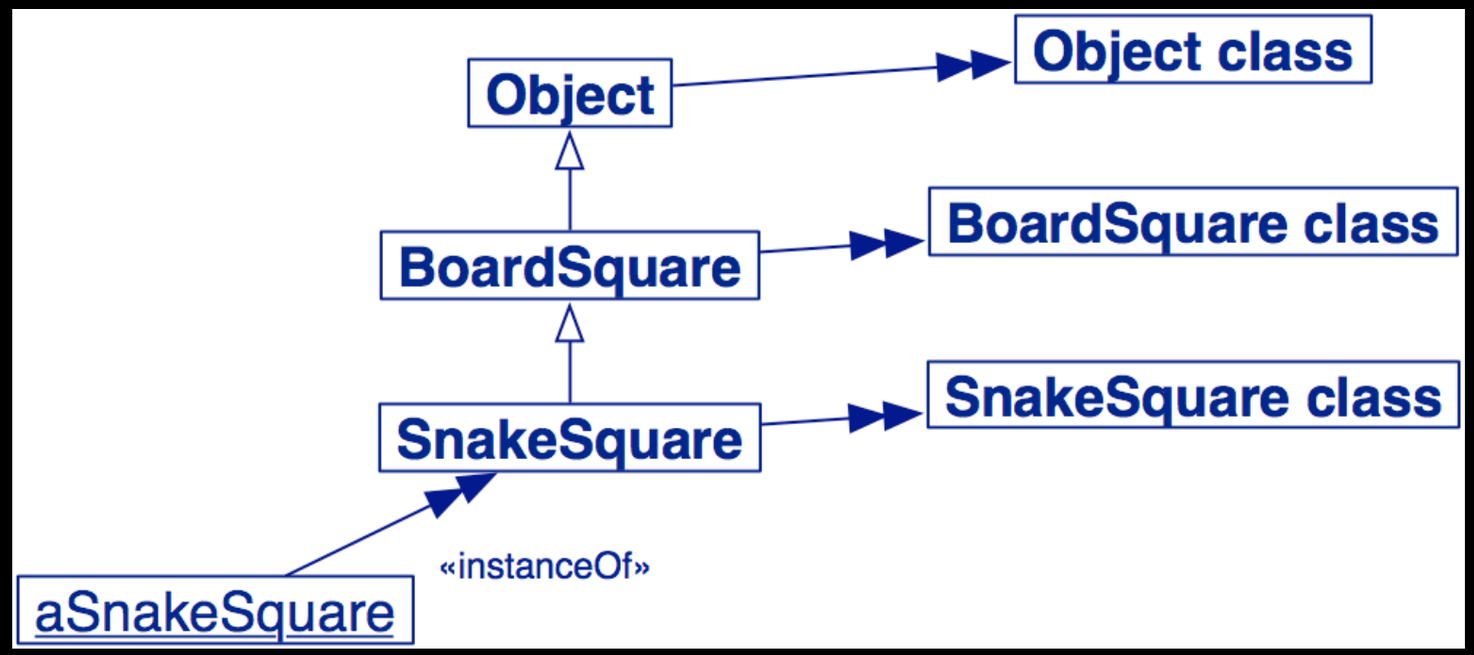
7- La metaclasse de **`Metaclass`** és instància de **`Metaclass`**

Adaptat de Goldberg & Robson, *Smalltalk-80 — The Language*

CAP: Reflexió en Smalltalk

Les classes també són objectes!!

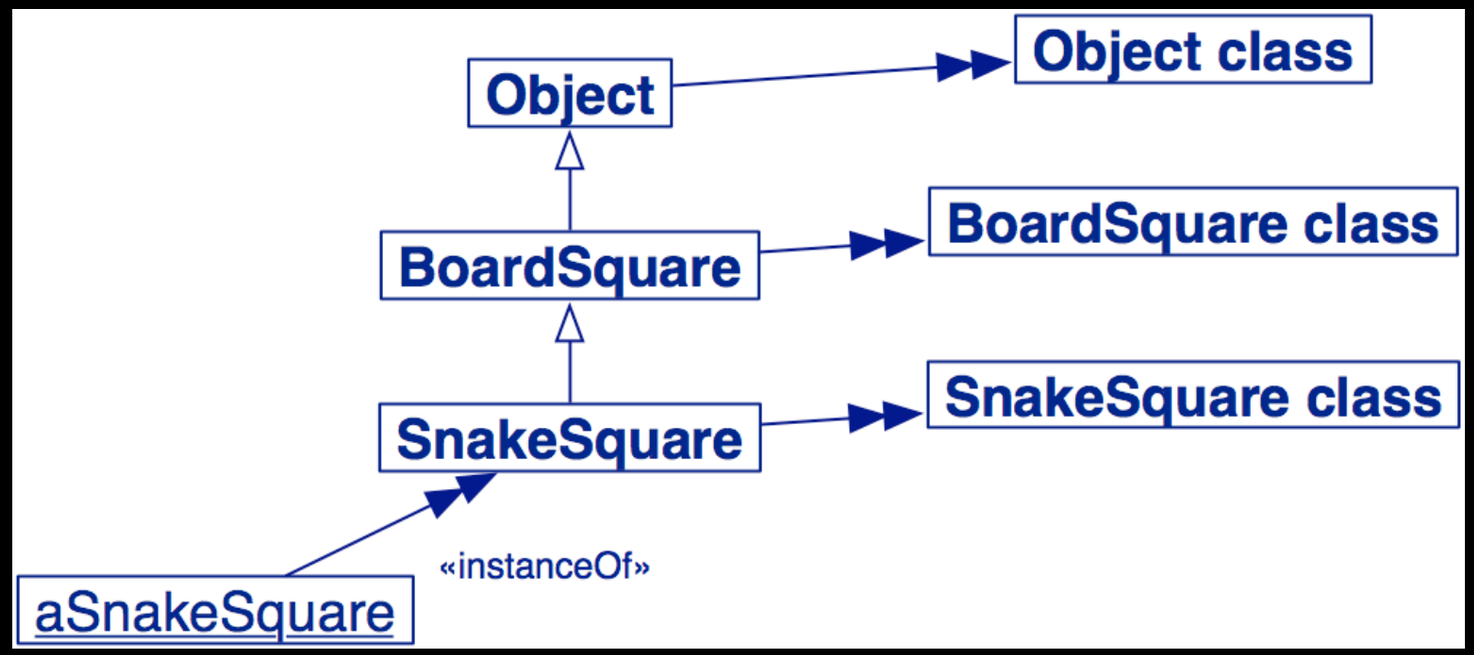
Cada classe, diguem-ne **`C`**, és l'única instància de la seva ***metaclasse***, anomenada ***`C class`***



CAP: Reflexió en Smalltalk

No hi ha metaclasses explícites

- Les metaclasses es construeixen *implícitament* quan es crean les classes
- No hi ha compartició de metaclasses, cada classe és *instància única* de la seva metaclasse



CAP: Reflexió en Smalltalk

Exemples:

```
BoardSquare allSubclasses →
      an OrderedCollection(LadderSquare FirstSquare SnakeSquare)

SnakeSquare allSubclasses → an OrderedCollection()

SnakeSquare allInstances  → an Array(<-6[11])

SnakeSquare instVarNames  → #('back')

      SnakeSquare back: 5 →  <-5[nil]

    SnakeSquare selectors → #(#setBack: #printOn: #destination)

     SnakeSquare canUnderstand: #new → false

SnakeSquare canUnderstand: #setBack: → true
```

CAP: Reflexió en Smalltalk

Les metaclasses en 7 parts:

1- Tot objecte és instància d'una classe

2- Tota classe hereta eventualment d'**`Object`**

3- Tota classe és instància d'una metaclasse

4- ***La jerarquia de metaclasses és equivalent a la jerarquia de classes***

5- Tota metaclasse hereta de **`Class`** i **`Behavior`**

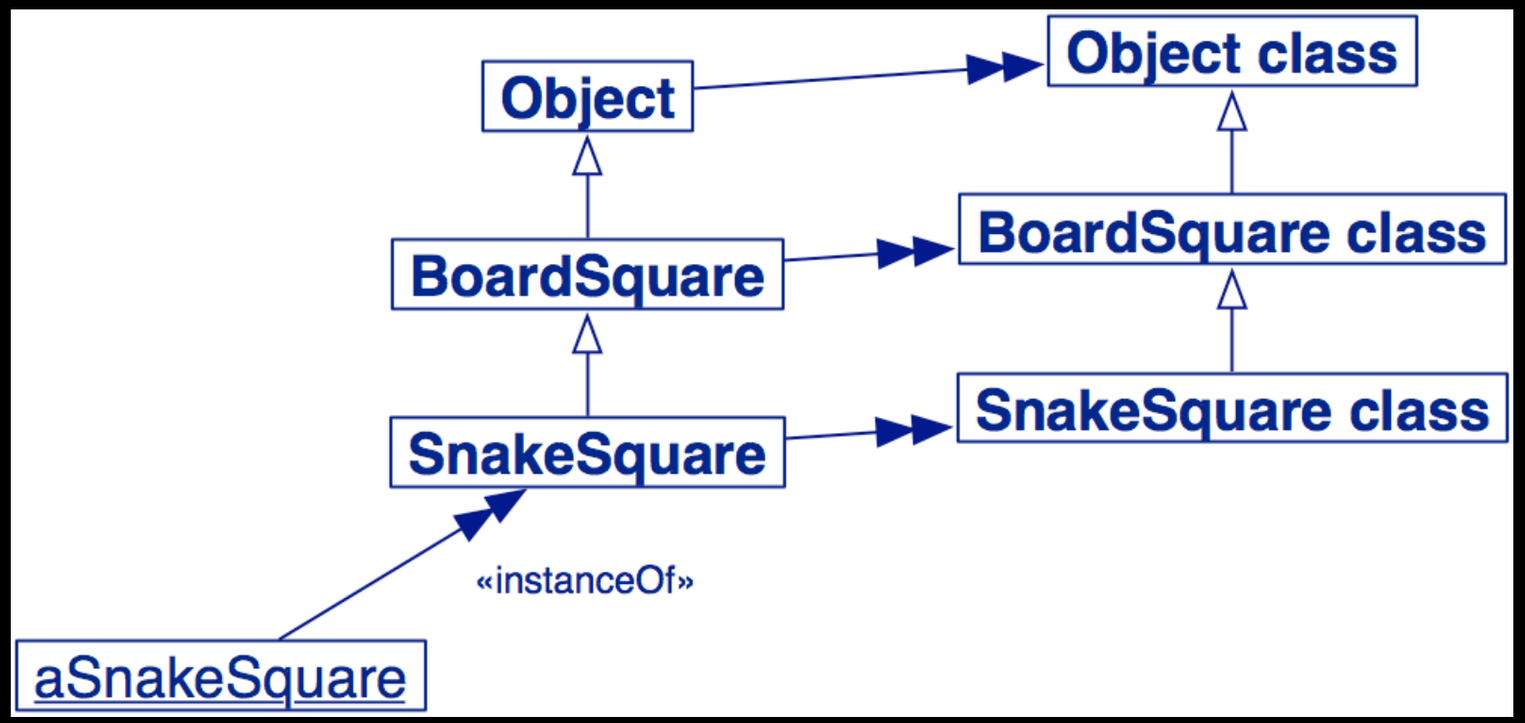
6- Tota metaclass és instància de **`Metaclass`**

7- La metaclasse de **`Metaclass`** és instància de **`Metaclass`**

Adaptat de Goldberg & Robson, *Smalltalk-80 — The Language*

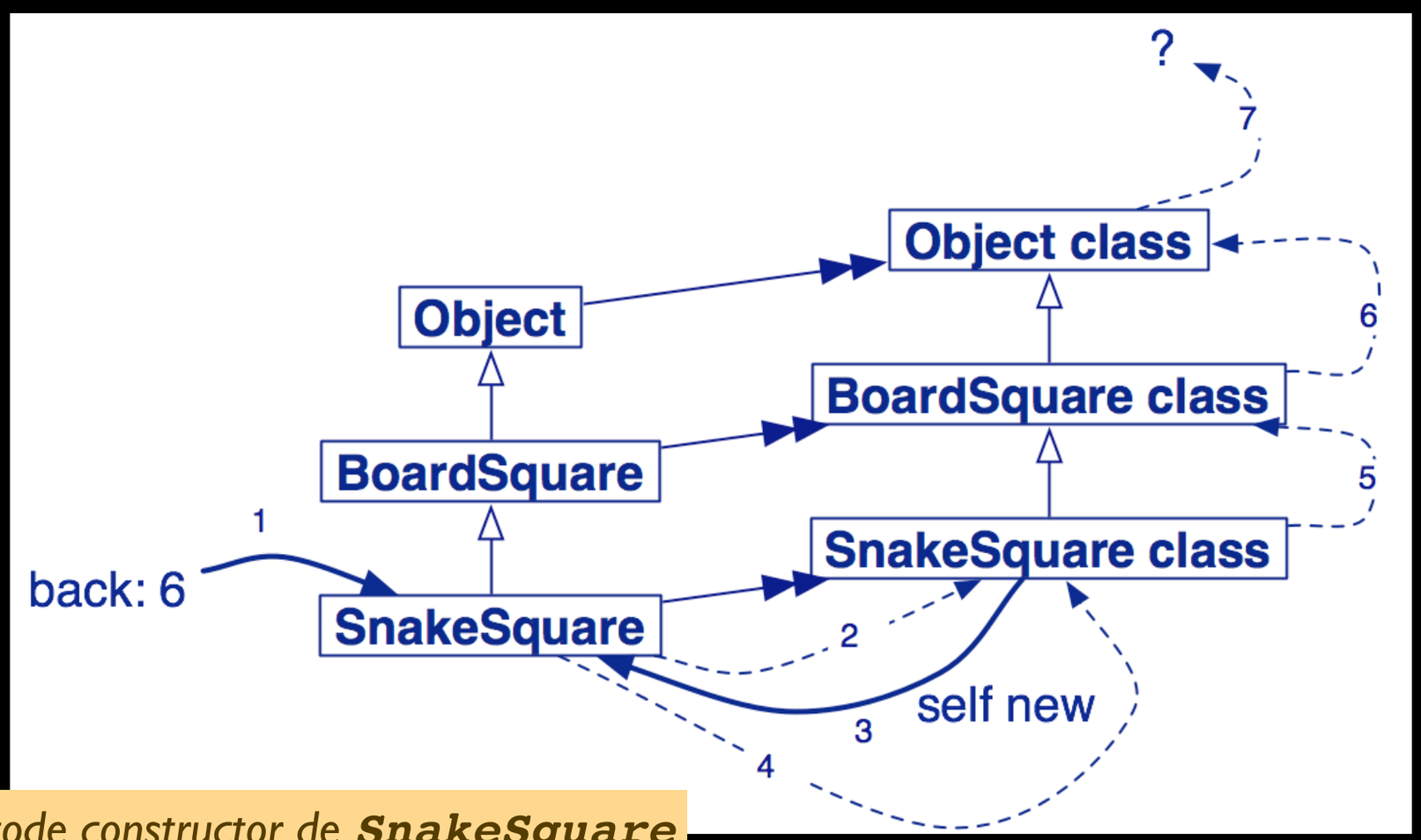
CAP: Reflexió en Smalltalk

La jerarquia de metaclasses és equivalent a la jerarquia de classes



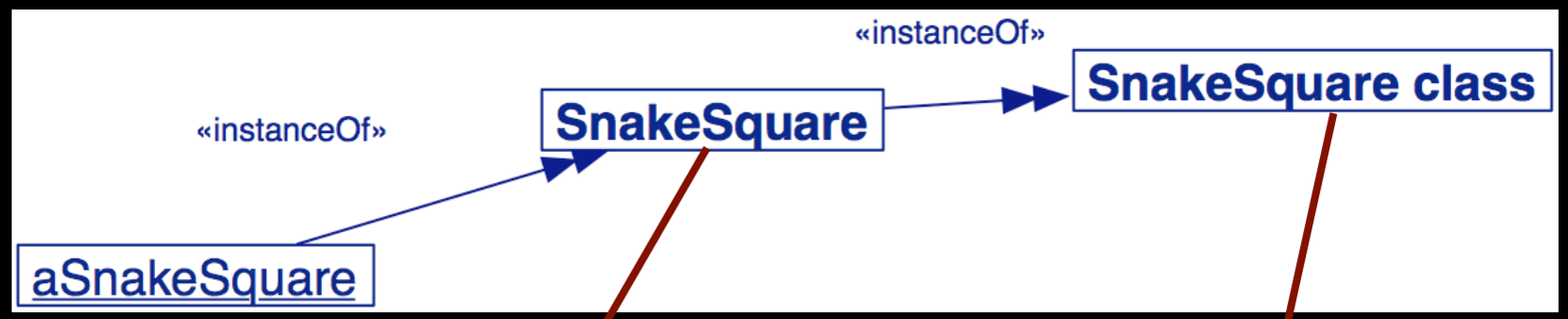
CAP: Reflexió en Smalltalk

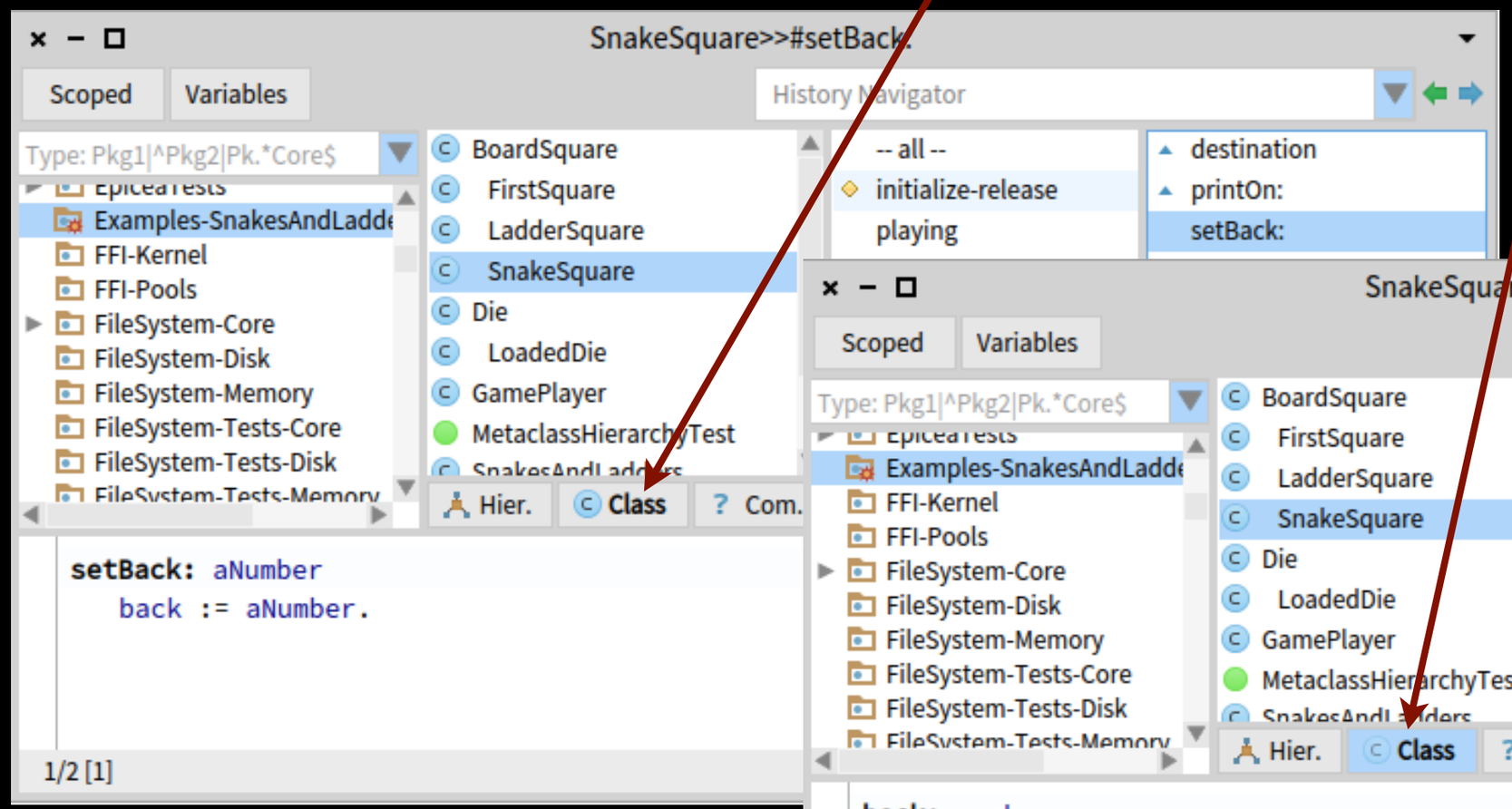
- **Les classes també són objectes, per tant tot el que hem après pels objectes és vàlid també per a les classes.**
- **Un enviament de missatge a una classe farà que es comenci a buscar el mètode corresponent al diccionari de mètodes de la metaclasse.**



#back: *és un mètode constructor de* ***SnakeSquare***

CAP: Reflexió en Smalltalk





CAP: Reflexió en Smalltalk

Les metaclasses en 7 parts:

1- Tot objecte és instància d'una classe

2- Tota classe hereta eventualment d'**`Object`**

3- Tota classe és instància d'una metaclasse

4- La jerarquia de metaclasses és equivalent a la jerarquia de classes

5- ***Tota metaclasse hereta de* `Class` *i* `Behavior`**

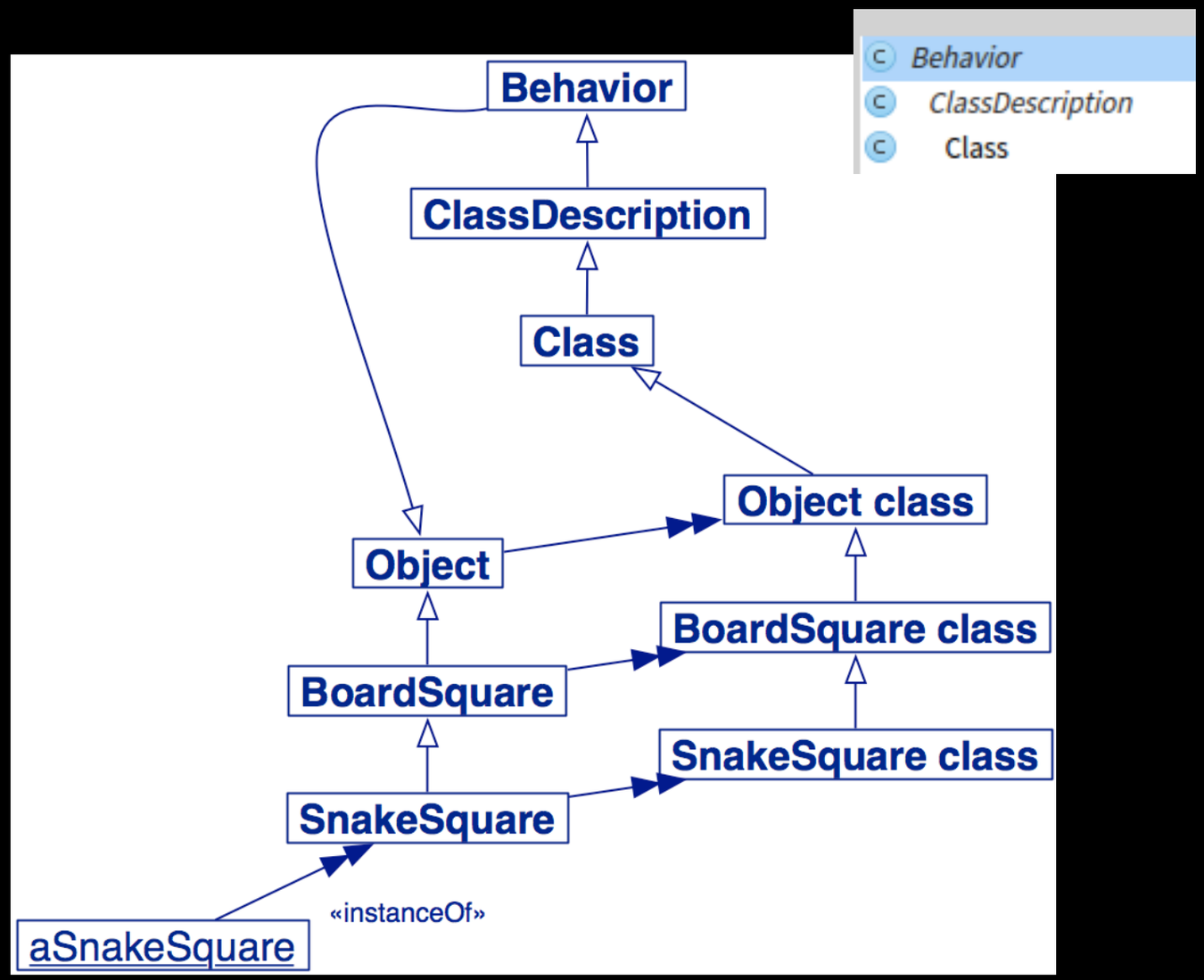
6- Tota metaclass és instància de **`Metaclass`**

7- La metaclasse de **`Metaclass`** és instància de **`Metaclass`**

Adaptat de Goldberg & Robson, *Smalltalk-80 — The Language*

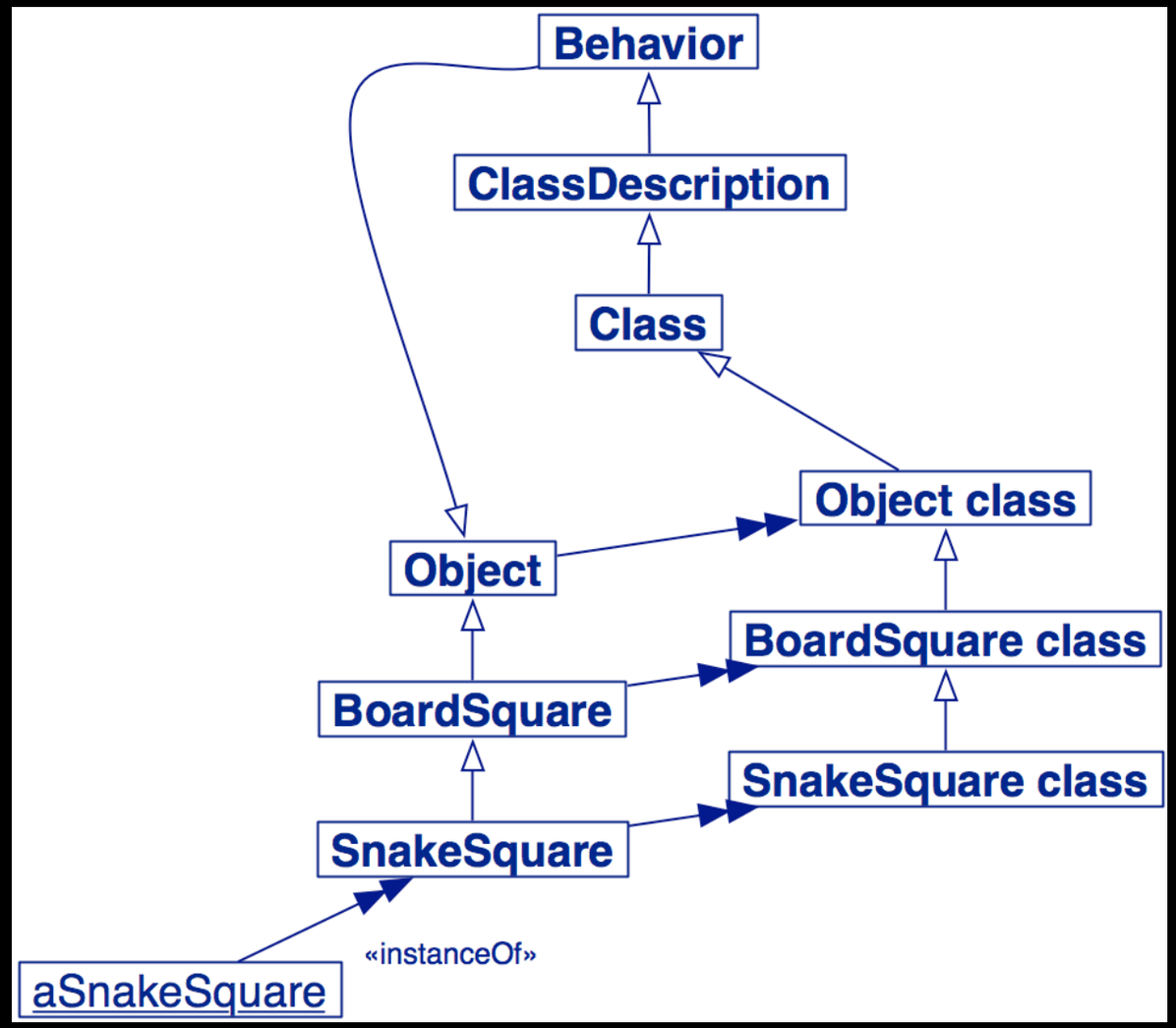
CAP: Reflexió en Smalltalk

***Cada classe* `és-una` `Class` ≡**
***La metaclasse de cada classe hereta de* `Class`**



CAP: Reflexió en Smalltalk

On està definit `new`?



CAP: Reflexió en Smalltalk

`Behavior` (responsabilitats)

- **Mínim estat necessari pels objectes que tenen instàncies**
- **Interfície bàsica pel compilador.**

Estat:

- enllaç amb la jerarquia de classes, diccionari de mètodes, descripció de les instàncies (representació i nombre)**.**

Mètodes:

- Crear un diccionar de mètodes, compilar mètodes.
- Crear instàncies (`#new`, `#basicNew`, `#new:`, `#basicNew:`)
- Manipular la jerarquia de classes (`#superclass:`)
- Accés (`#selectors`, `#allSelectors`, `#compiledMethodAt:`)
- Accés a instàncies i a variables (`#allInstances`, `#instVarNames`)
- Accés a la jerarquia de classes (`#superclass`, `#subclassesDo:`)
- *Testing* (`#hasMethods`, `#includesSelector:`, `#canUnderstand:`, `#inheritsFrom:`, `#isVariable`)

CAP: Reflexió en Smalltalk

`ClassDescription` (responsabilitats)

Afegeix un cert nombre d'utilitats a `Behavior`

- Variables d'instància amb nom (*named instance variables*).
- Organització dels mètodes en categories.
- La noció de nom (abstracte).
- Manteniment dels canvis en els *Change Sets* i el *Logging*.
- La majoria dels mecanismes necessaris per fer **`fileOut`**.

`ClassDescription` és una classe abstracte: Les utilitats que proporciona estan pensades per ser heretades per **`Class`** i **`Metaclass`**

CAP: Reflexió en Smalltalk

`Class` (responsabilitats)

Representa el comportament comú a totes les classes
- nom, compilació, emmagatzematge de mètodes, variables d'instància...

Representació pels noms de variables de classes.
- `#addClassVarName:`, `#classVarNamed:`.

`Class` hereta d'`Object` perque `Class` és un `Object`.

`Class` sap com crear instàncies, per tant totes les metaclasses han d'heretar de `Class`.

CAP: Reflexió en Smalltalk

Les metaclasses en 7 parts:

1- Tot objecte és instància d'una classe

2- Tota classe hereta eventualment d'**`Object`**

3- Tota classe és instància d'una metaclasse

4- La jerarquia de metaclasses és equivalent a la jerarquia de classes

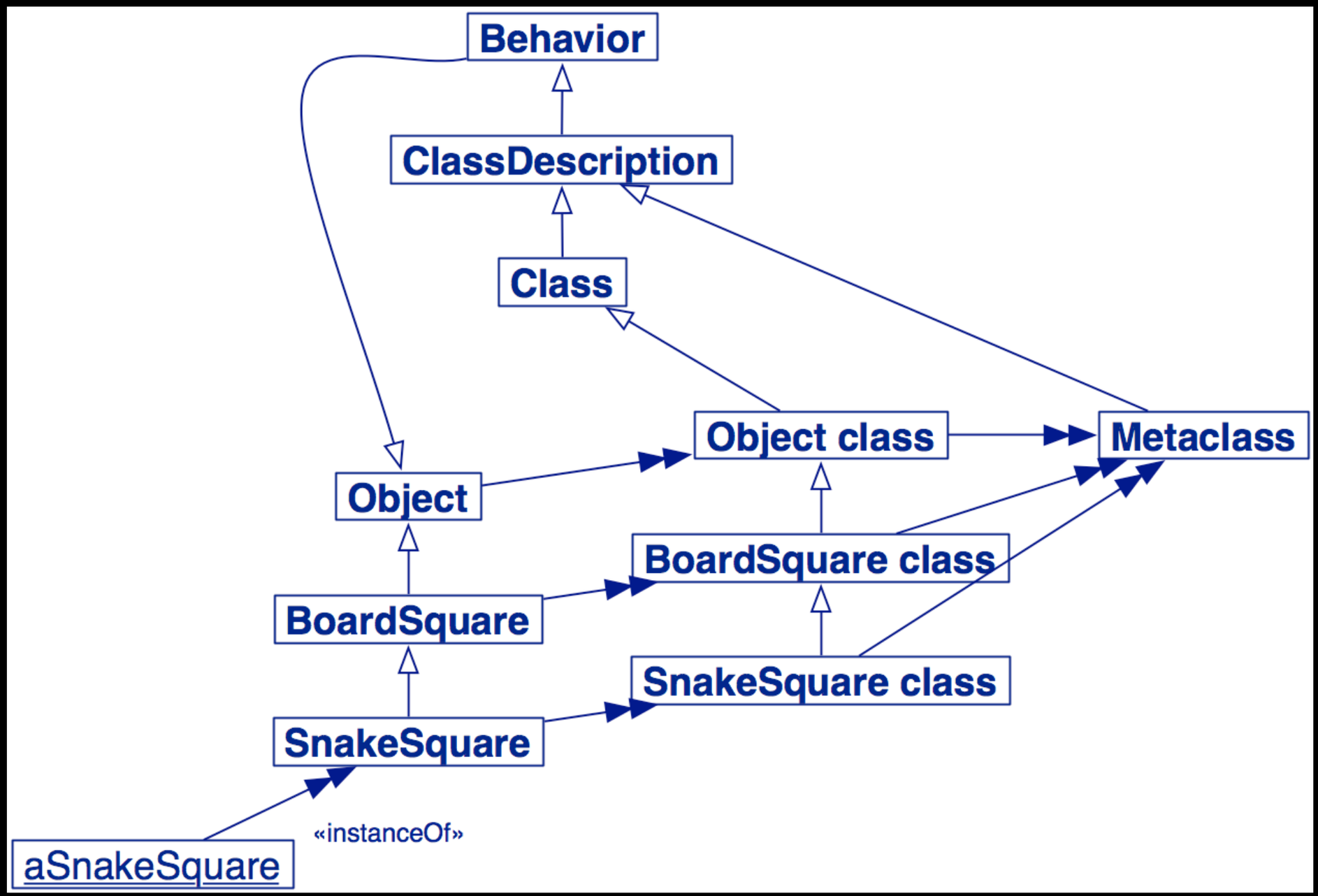
5- Tota metaclasse hereta de **`Class`** i **`Behavior`**

6- ***Tota metaclass és instància de*** **`Metaclass`**

7- La metaclasse de **`Metaclass`** és instància de **`Metaclass`**

Adaptat de Goldberg & Robson, *Smalltalk-80 — The Language*

CAP: Reflexió en Smalltalk



CAP: Reflexió en Smalltalk

`Metaclass` (responsabilitats)

Representa el comportament comú a les metaclasses.

- Crear instàncies (`#new`, `#instanceVariableNames:`)
- Crear instàncies inicialitzades de l'única instància de la metaclasse.
- Compilació de mètodes (`#subClasses`)
- Informació de la classe (*enllaços d'herència*, *variables d'instància*...)

CAP: Reflexió en Smalltalk

Les metaclasses en 7 parts:

1- Tot objecte és instància d'una classe

2- Tota classe hereta eventualment d'**`Object`**

3- Tota classe és instància d'una metaclasse

4- La jerarquia de metaclasses és equivalent a la jerarquia de classes

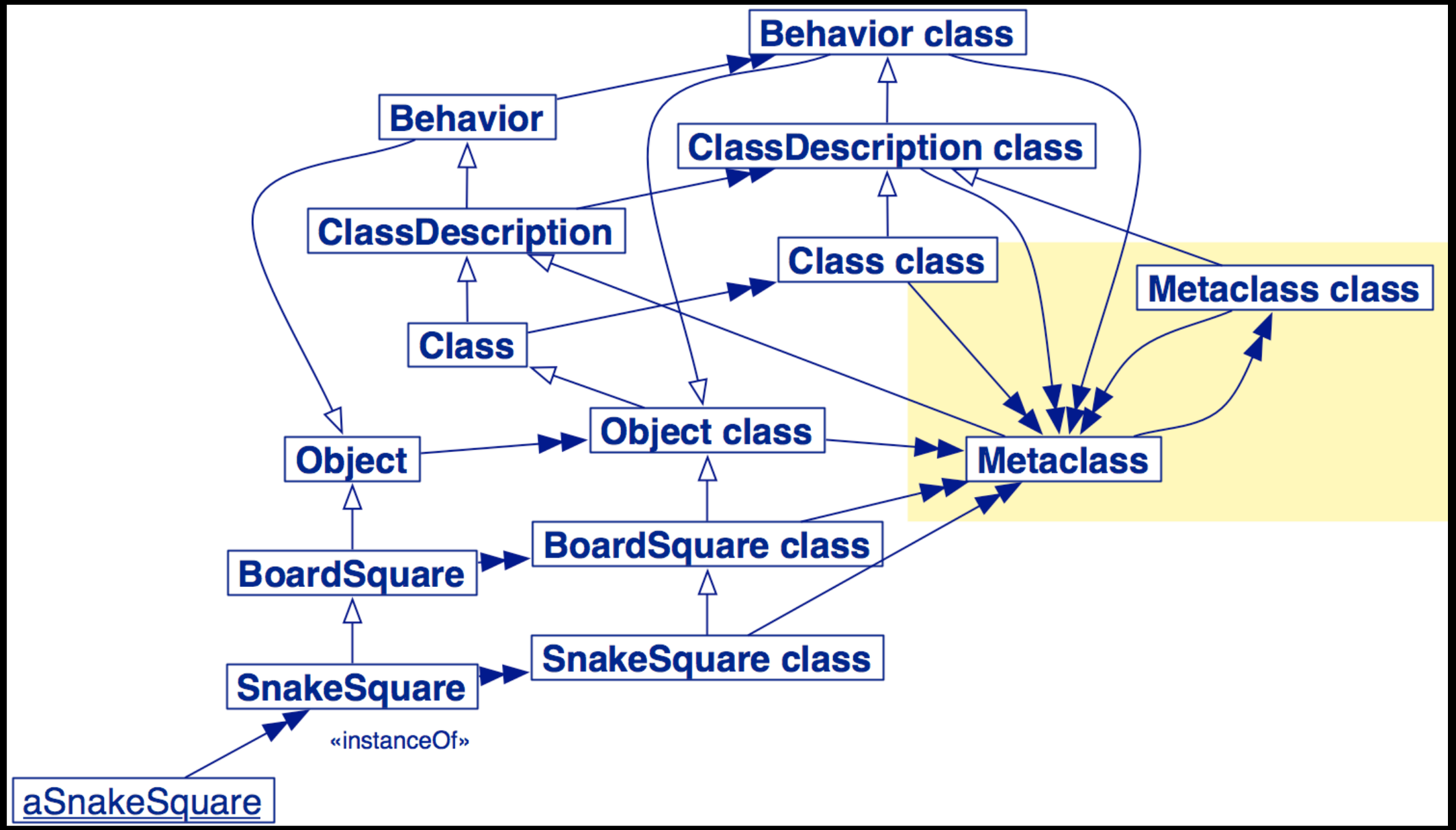
5- Tota metaclasse hereta de **`Class`** i **`Behavior`**

6- Tota metaclass és instància de **`Metaclass`**

7- ***La metaclasse de `Metaclass` és instància de `Metaclass`***

Adaptat de Goldberg & Robson, *Smalltalk-80 — The Language*

CAP: Reflexió en Smalltalk



CAP: Reflexió en Smalltalk

```
MetaclassHierarchyTest >> testHierarchy
   "The class hierarchy"
   self assert: SnakeSquare superclass = BoardSquare.
   self assert: BoardSquare superclass = Object.
   self assert: Object superclass superclass = nil.
   "The parallel metaclass hierarchy"
   self assert: SnakeSquare class name = 'SnakeSquare class'.
   self assert: SnakeSquare class superclass = BoardSquare class.
   self assert: BoardSquare class superclass = Object class.
   self assert: Object class superclass superclass = Class.
   self assert: Class superclass = ClassDescription.
   self assert: ClassDescription superclass = Behavior.
   self assert: Behavior superclass = Object.
   "The Metaclass hierarchy"
   self assert: SnakeSquare class class = Metaclass.
   self assert: BoardSquare class class = Metaclass.
   self assert: Object class class = Metaclass.
   self assert: Class class class = Metaclass.
   self assert: ClassDescription class class = Metaclass.
   self assert: Behavior class class = Metaclass.
   self assert: Metaclass superclass = ClassDescription.
   "The fixpoint"
   self assert: Metaclass class class = Metaclass
```

CAP: Reflexió en Smalltalk

<u>Pròleg</u>:

- Les Metaclasses en 7 parts

- ***Classes indexades***

- Variables de classe-instància

- Variables de classe

CAP: Reflexió en Smalltalk

Dues maneres de representar objectes

Variables d'Instància amb nom o *indexades*

- Amb nom `name` de `GamePlayer`

- Indexada: `#(Jack Jill) at: 1`

O, mirant-ho d'una altra manera

- Objectes amb referències a altres objectes

- Objectes amb *arrays* de *bytes* (*word*, *long*)

- Fem la diferència per raons d'eficiència: emmagatzemar *arrays* de *bytes* (com les *strings* de C) és més eficient que emmagatzemar un *array* de referències, cada una d'elles apuntant a un sol *byte.*

CAP: Reflexió en Smalltalk

Diferentes maneres de crear classes

Indexat	Amb nom	Mètode de definició
No	Yes	`#subclass: …`
Yes	Yes	`#variableSubclass: …`
Yes	No	`#variableByteSubclass: …`

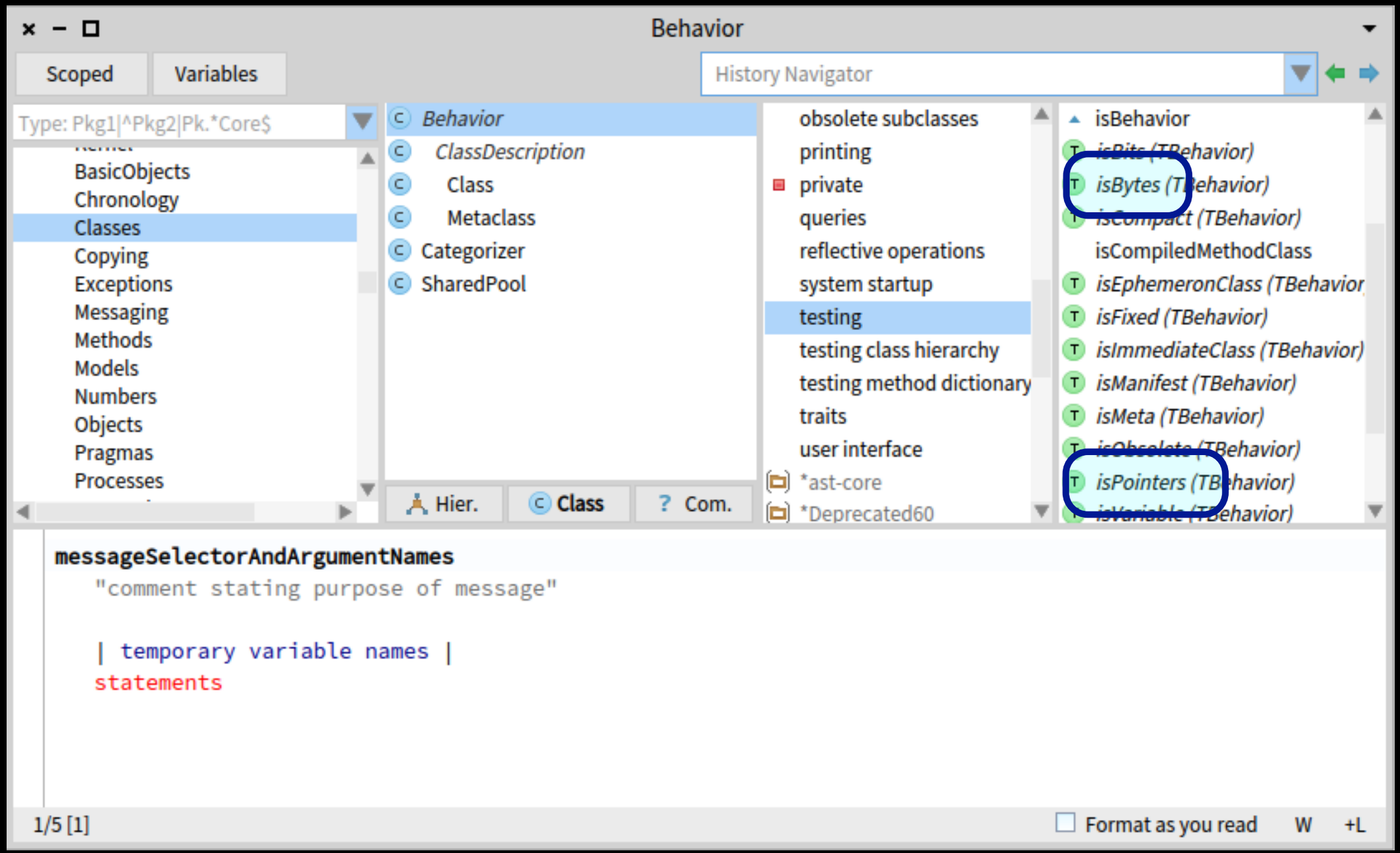
Veieu el protocol **`subclass creation`** de **`Class`**

Restriccions:

- **classes amb referències (*pointer classes*):** definides amb **`#subclass:`** poden tenir subclasses de qualsevol mena.
- **classes amb *bytes* (*byte classes*):** definides amb **`#variableSubclass:`** només poden tenir subclasses definides amb **`#variableSubclass:`** o amb **`#variableByteSubclass:`**

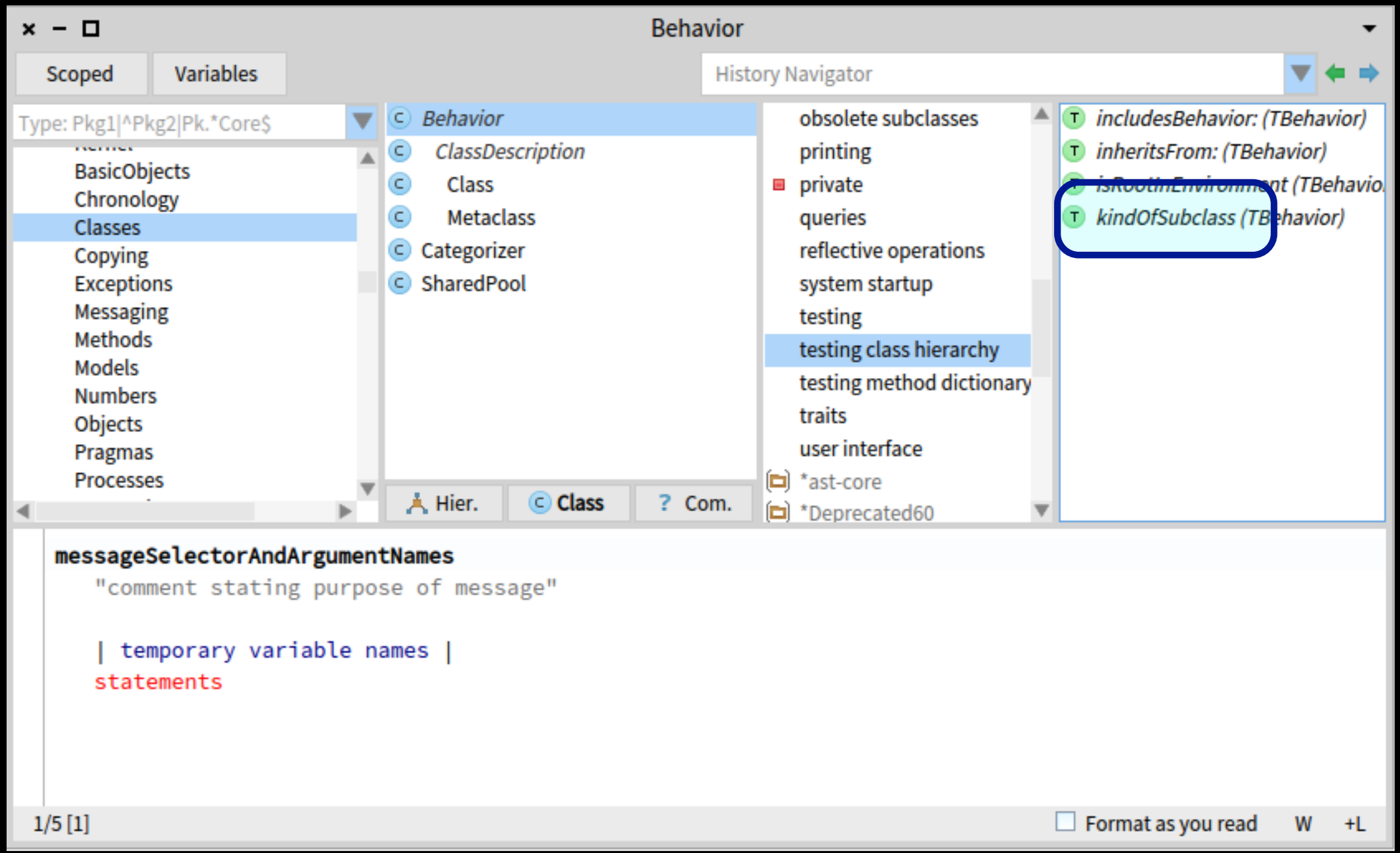
CAP: Reflexió en Smalltalk

Mètodes de *Testing*



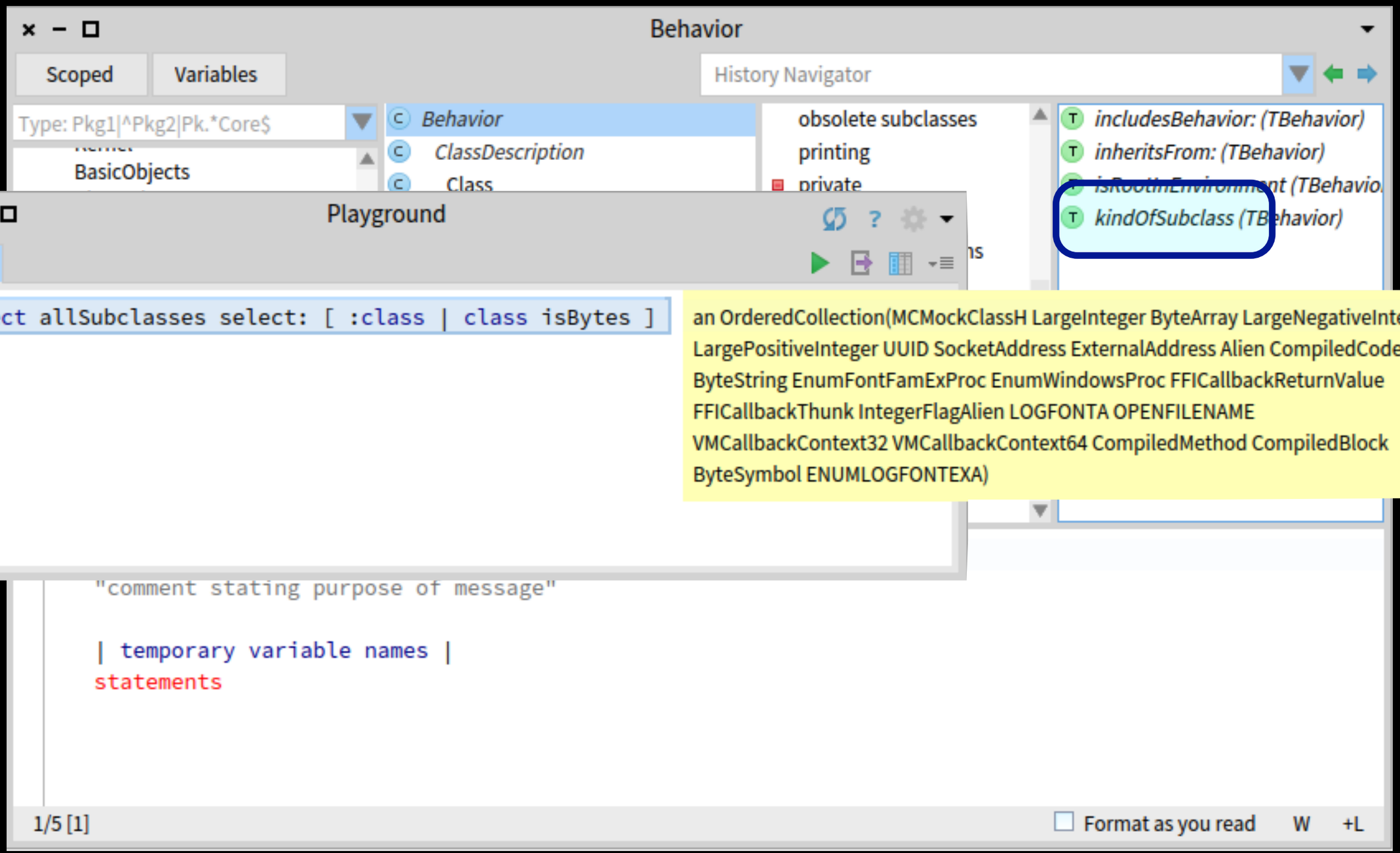
CAP: Reflexió en Smalltalk

Mètodes de *Testing*



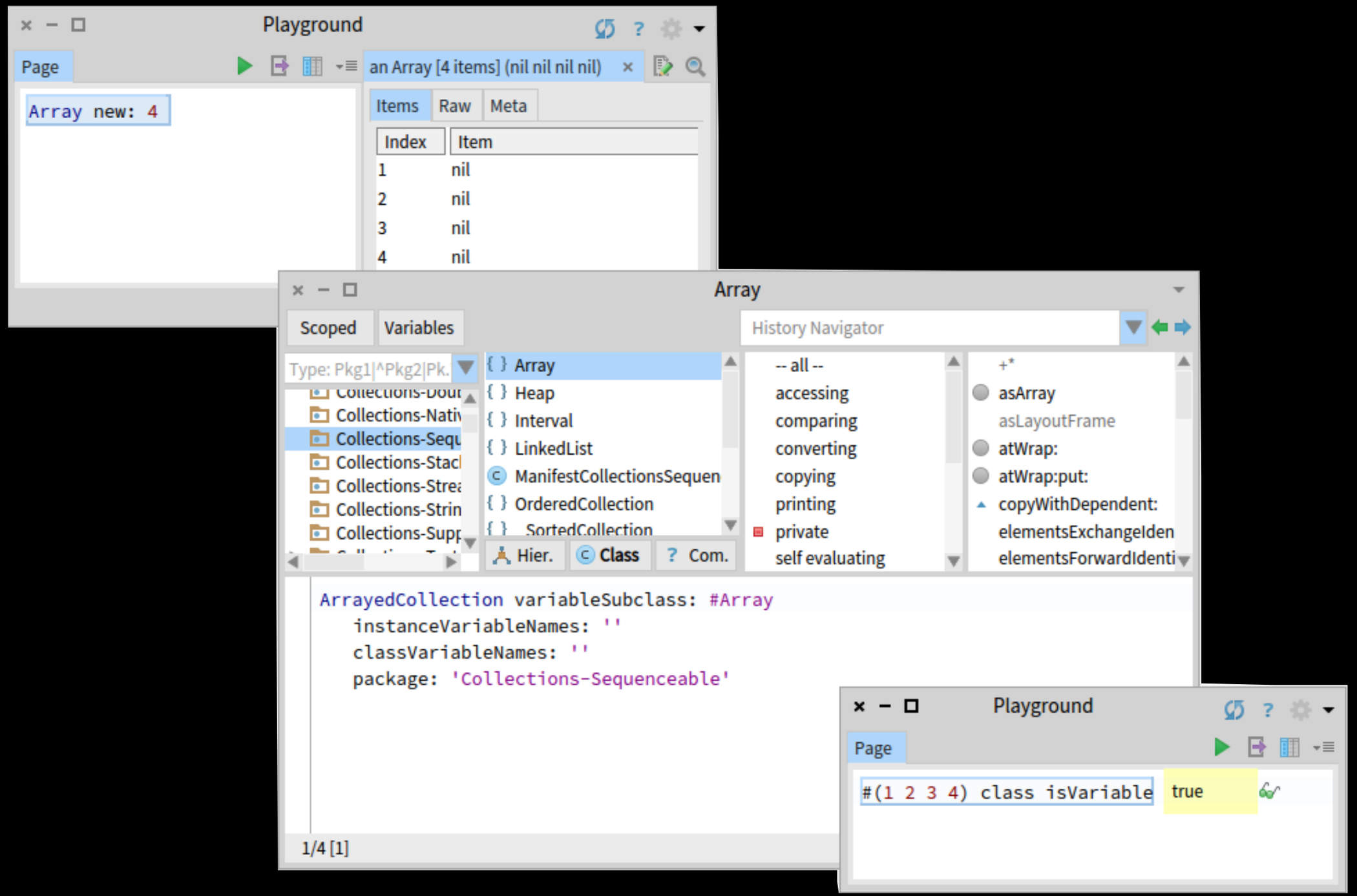
CAP: Reflexió en Smalltalk

Mètodes de *Testing*



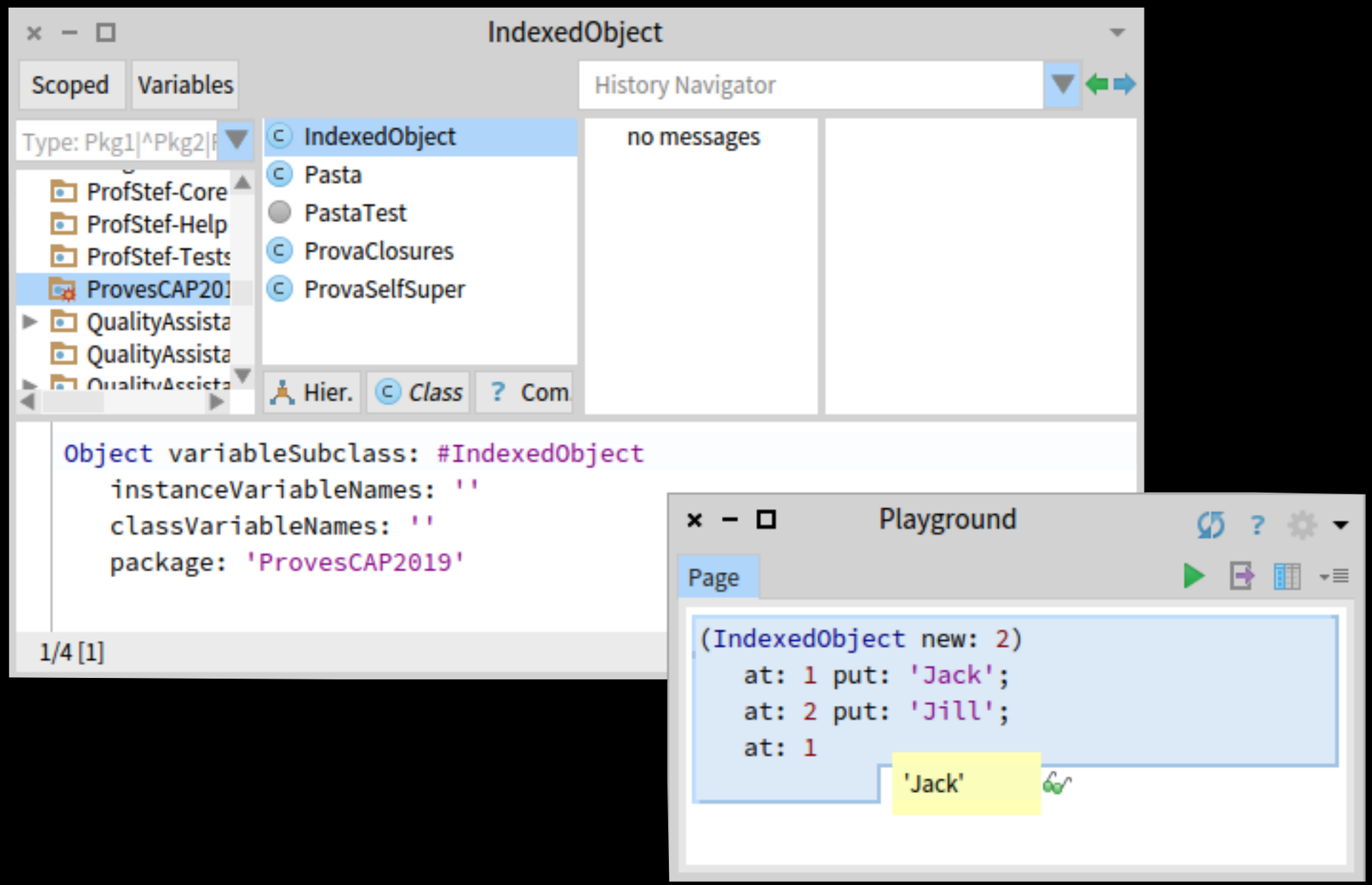
CAP: Reflexió en Smalltalk

Exemple



CAP: Reflexió en Smalltalk

Creació de classes indexades: Exemple



CAP: Reflexió en Smalltalk

Classes Indexades / Variables d'Instància

Una variable indexada s'afegeix implícitament a la llista de variables d'Instància

- Només ***UNA*** variable indexada (d'instància) per classe

- Accés amb `#at:` i amb `#at:put:`.
  (recordeu que aquests mètodes retornen el valor, no el receptor)

Les subclasses han de ser també indexades

CAP: Reflexió en Smalltalk

Pròleg:

- Les Metaclasses en 7 parts

- Classes indexades

- ***Variables de classe-instància***

- Variables de classe

CAP: Reflexió en Smalltalk

Variables de classe-instància

Les classes també són objectes

- Instàncies de la seva *metaclasse*
  (els mètodes es busquen en el diccionari de mètodes de la seva metaclasse)
- Així doncs, poden tenir variables d'instància

Quan la metaclasse defineix una variable d'instància nova, aleshores la seva instància (la classe) té una variable nova

- Que s'afegeix a `subclasses`, `superclass`, `methodDict`, etc.

Utilitzeu variables de classe-instància per representar l'estat privat d'una classe

- Per exemple, nombre d'instàncies, superclasse, etc.
- *No serveixen per representar informació que han de compartir les instàncies*

CAP: Reflexió en Smalltalk

Exemple: El patró `Singleton`

Una classe amb una sola instància:

La guardem en una variable de classe instància.

```
WebServer class
  instanceVariableNames: 'uniqueInstance'

WebServer class >> new
  self error: 'Use uniqueInstance to get the unique instance'

WebServer class >> uniqueInstance
   uniqueInstance isNil
    ifTrue: [uniqueInstance := self basicNew initialize].
   ^ uniqueInstance
```

CAP: Reflexió en Smalltalk

Pròleg:

- Les Metaclasses en 7 parts

- Classes indexades

- Variables de classe-instància

- ***Variables de classe***

CAP: Reflexió en Smalltalk

Variables de classe ≡ Variables compartides

Per compartir informació entre les instàncies d'una classe feu servir una *variable de classe*

- Compartida i directament accessible per totes les instàncies de la classe i subclasses

- Accessible tant a mètodes de classe com a mètodes d'instància.

- Comença amb una lletra majúscula.

CAP: Reflexió en Smalltalk

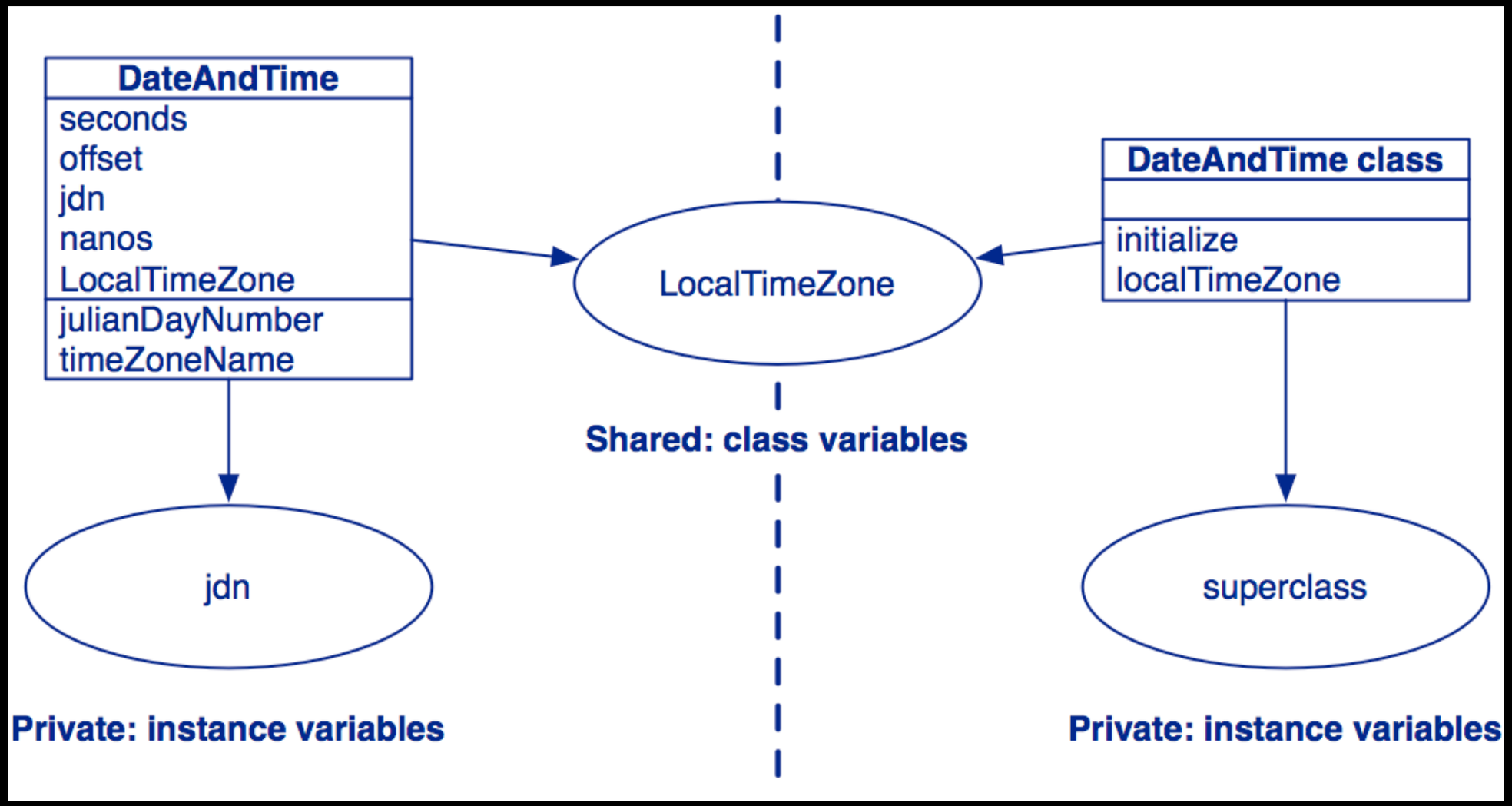
Inicialitzant Variables de Classe

Les variables de classe s'haurien d'inicialitzar en un mètode `initialize` *de classe*, o per inicialització *mandrosa* (*lazy*)

```
Magnitude subclass: #DateAndTime
  instanceVariableNames: 'seconds offset jdn nanos'
  classVariableNames: 'LocalTimeZone ...'
  package: 'Kernel-Chronology'

DateAndTime class >> localTimeZone
  "Answer the local time zone"

  ^ LocalTimeZone ifNil: [ LocalTimeZone := TimeZone default ]
```

CAP: Reflexió en Smalltalk

Variables de Classe vs. Variables d'Instància



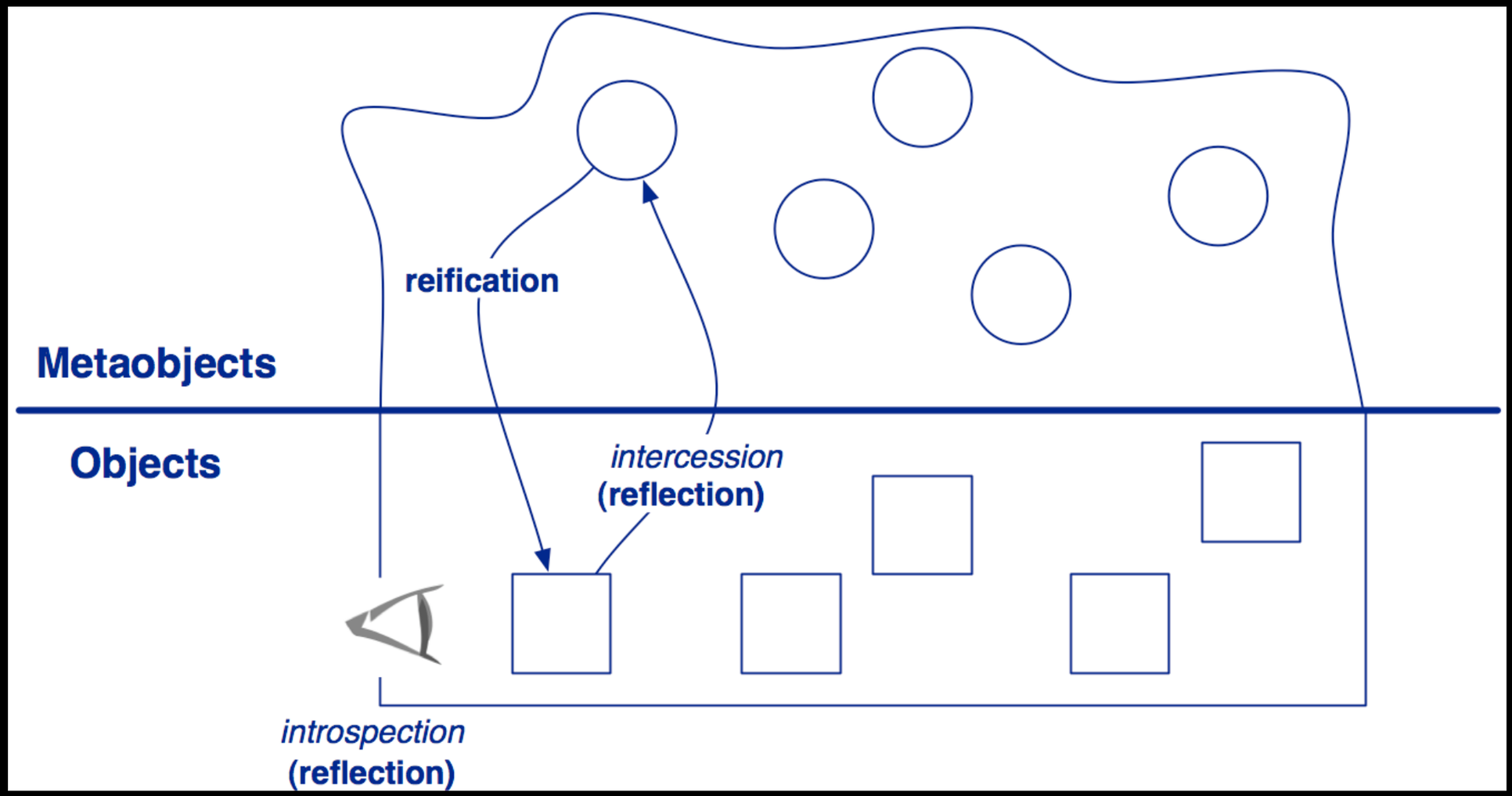
CAP: Reflexió en Smalltalk

And Now for Something Completely Different...

Part del material és cortesia d'Stéphane Ducasse i Marcus Denker

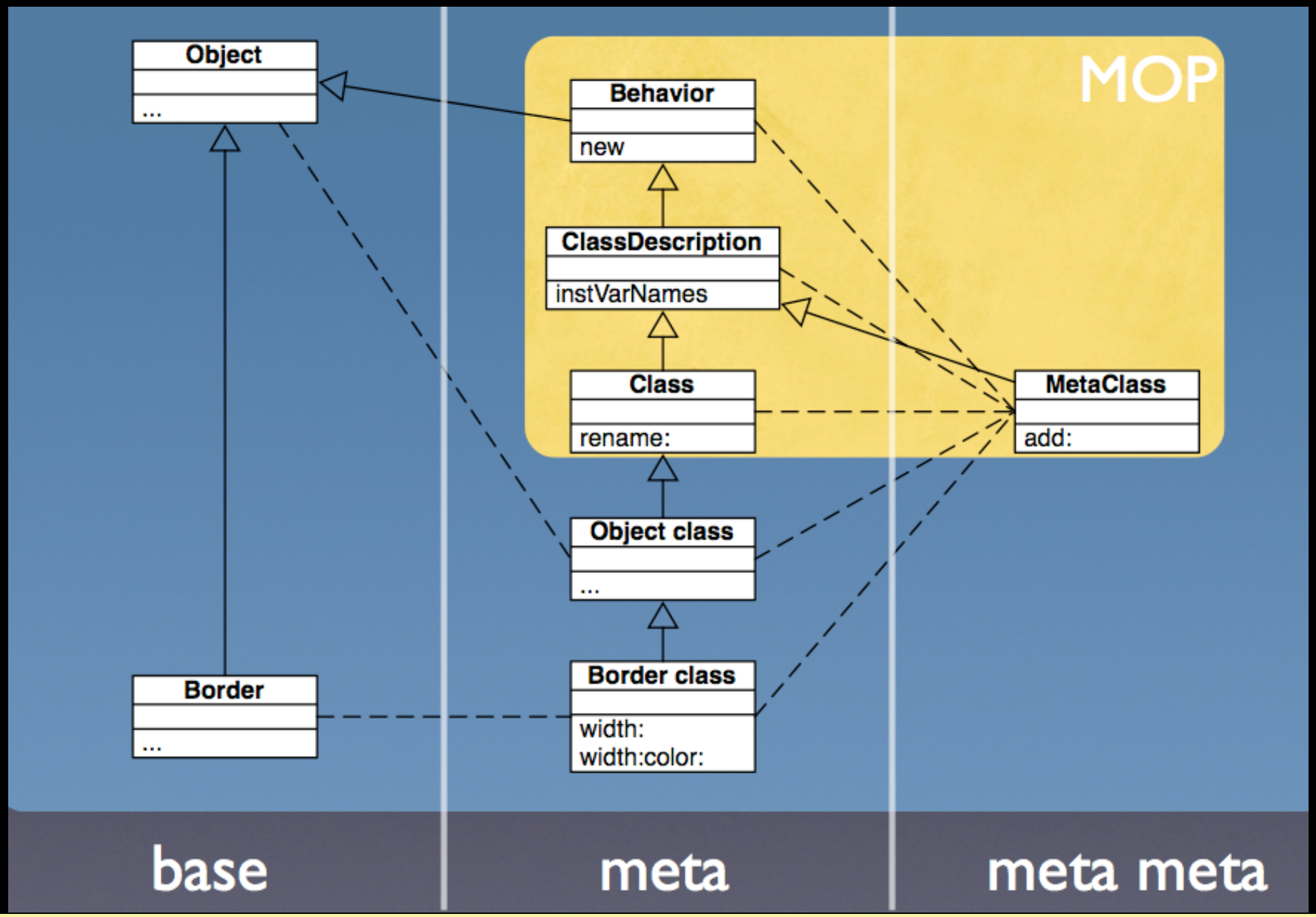
CAP: Reflexió en Smalltalk

Recordem: *Intercessió*, *Introspecció*, *Reificació*



CAP: Reflexió en Smalltalk

Recordem: *Intercessió*, *Introspecció*, *Reificació*



from ‘Smalltalk’, Johan Brichau (LIFL/PROG) & Roel Wuyts (ULB), Dynamic Languages Day, Feb. 2006

CAP: Reflexió en Smalltalk

La Reflexió ens permet *examinar* i a l'hora *alterar* els meta-objectes d'un sistema

Utilitzar la reflexió per modificar un sistema que s'està executant requereix tenir una mica de cura...

CAP: Reflexió en Smalltalk

Introspecció:

- Inspeccionar objectes
- Consultar el codi
- Accedir els contexts d'execució

Intercessió:

- Sobreescriure `#doesNotUnderstand:`
- Classes Anònimes
- Method Wrappers
- Continuacions

Part del material extret del treball de Marcus Denker i Stéphane Ducasse

CAP: Reflexió en Smalltalk

Introspecció:

- ***Inspeccionar objectes***
- Consultar el codi
- Accedir els contexts d'execució

Intercessió:

- Sobreescriure **`#doesNotUnderstand:`**
- Classes Anònimes
- Method Wrappers
- Continuacions

Part del material extret del treball de Marcus Denker i Stéphane Ducasse

CAP: Reflexió en Smalltalk

L’essencia d’una classe

- **Un format** (nombre de variables d’instància, etc.)

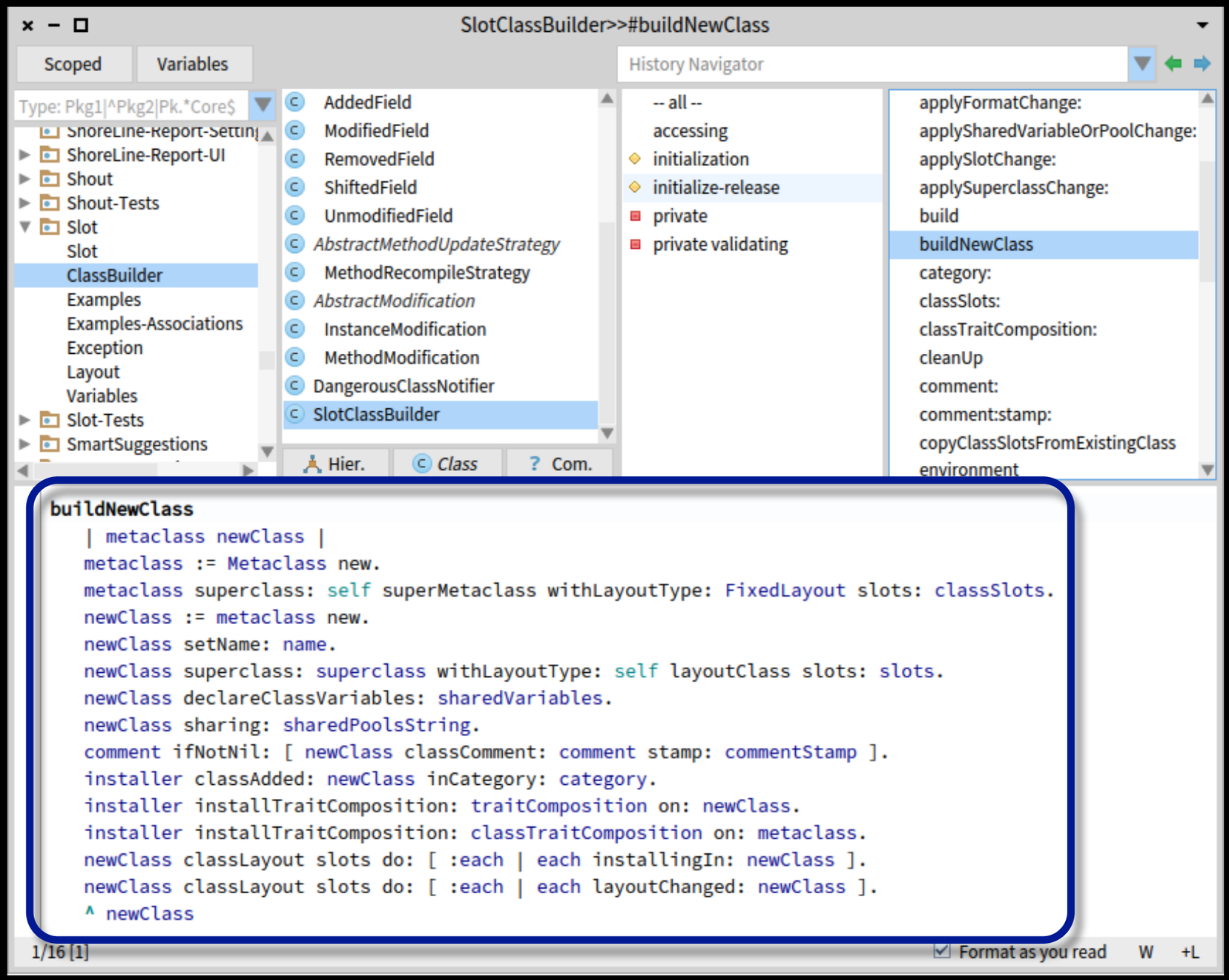
- **Una superclasse**

- **Un diccionari de mètodes**

Exercici: On es crea la metaclasse, i la classe com a instància?
Pista: començar per **`subclass:`**...

CAP: Reflexió en Smalltalk

L'essencia d'una classe



CAP: Reflexió en Smalltalk

L’essencia d’un objecte

- Una referència a una classe

- Valors

- Pot ser especial:
  - **`SmallInteger`**
  - Indexat
  - Classes compactes (**`CompiledMethod`**, **`Array`** ...)

CAP: Reflexió en Smalltalk

Metaobjectes vs. metaclasses

Cal distingir entre metaclasses i metaobjectes!

- Una ***metaclasse*** és una classe les instàncies de la qual són classes

- Un ***metaobjecte*** és un objecte que descriu o manipula altres objectes (*diferents metaobjectes poden controlar aspectes diferents dels objectes*)

CAP: Reflexió en Smalltalk

Alguns Metaobjectes

Estructura:

`Behavior`, `ClassDescription`, `Class`, `Metaclass`, `ClassBuilder`

Semántica:

`Compiler`, `Decompiler`, `IRBuilder`

Comportament:

`CompiledMethod`, `BlockContext`, `Message`, `Exception`

Control de l'estat:

`BlockContext`, `Process`, `ProcessorScheduler`

Recursos:

`WeakArray`

Noms:

`SystemDictionary`

Llibreries:

`MethodDictionary`, `ClassOrganizer`

CAP: Reflexió en Smalltalk

Metaoperacions

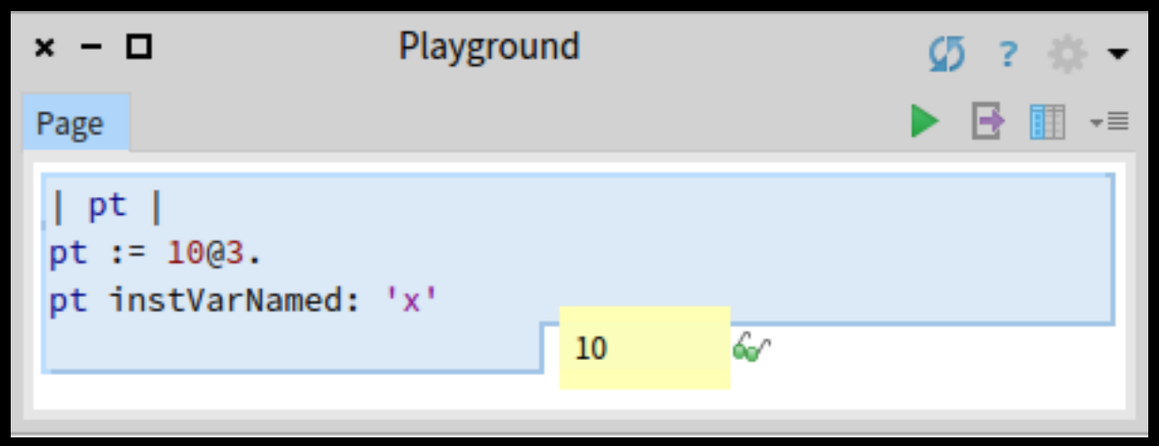
"*Meta-operations are operations that provide information about an object as opposed to information directly contained by the object ...They permit things to be done that are not normally possible*"

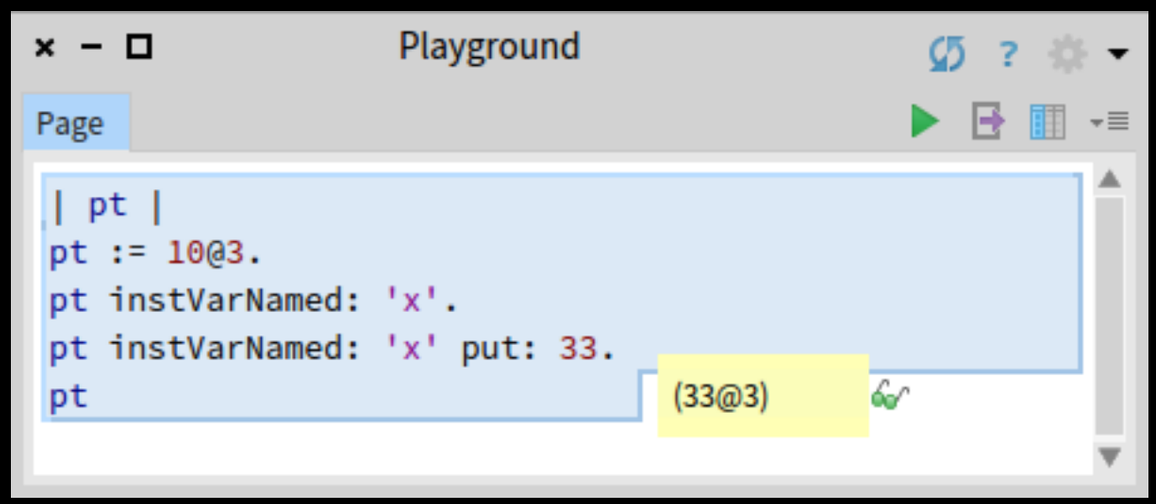
Inside Smalltalk

CAP: Reflexió en Smalltalk

Accés a l'estat

```
Object >> #instVarNamed:
Object >> #instVarNamed:put:
Object >> #instVarAt:
Object >> #instVarAt:put:
```





CAP: Reflexió en Smalltalk

Accés a la meta-informació

```
Object >> #class
Object >> #identityHash
```

```
'hello' class  → ByteString
(10@3) class  → Point
Smalltalk class  → SystemDictionary
Class class  → Class class
Class class class  → Metaclass
Class class class class  → Metaclass class

'hello' identityHash  → 843767808
Object identityHash  →  657408
5 identityHash  →  8322625
```

CAP: Reflexió en Smalltalk

Canvis

`Object >> #primitiveChangeClassTo:`

- canvia la classe de l'objecte receptor per la classe de l'objecte argument. Les dues classes haurien de tenir el mateix format, és a dir, la mateixa estructura a les seves instàncies.

`ProtoObject >> #become:`

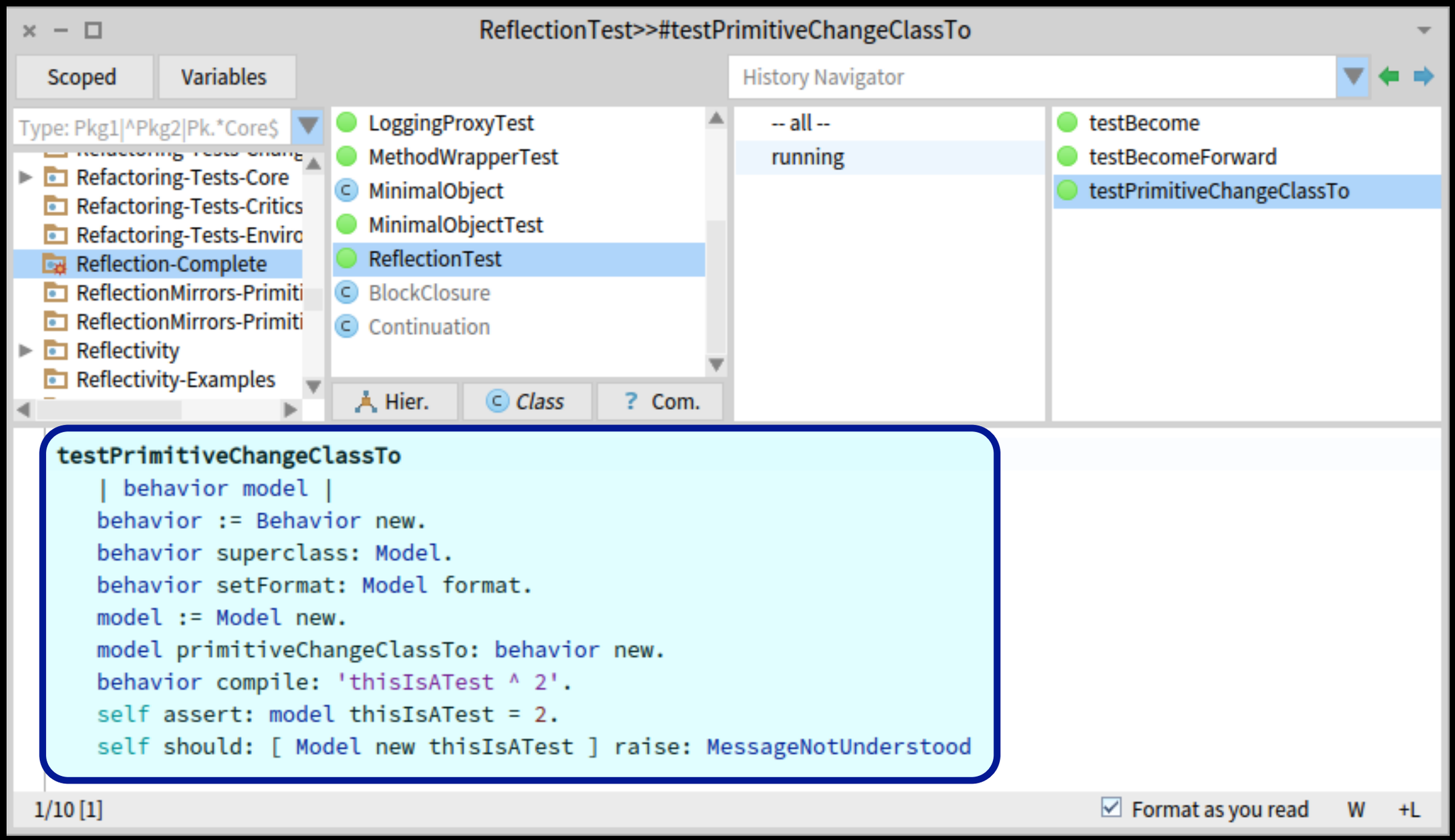
- Intercanvia *les referències* al receptor per les de l'argument, i a l'inrevés.
- Totes les variables del sistema (*tot ell*) que referenciaven el receptor ara apunten a l'argument i vice-versa.
- Falla si cap dels dos objectes és un **`SmallInteger`**

`Object >> #becomeForward:`

- Com **`become`** però només en una direcció

CAP: Reflexió en Smalltalk

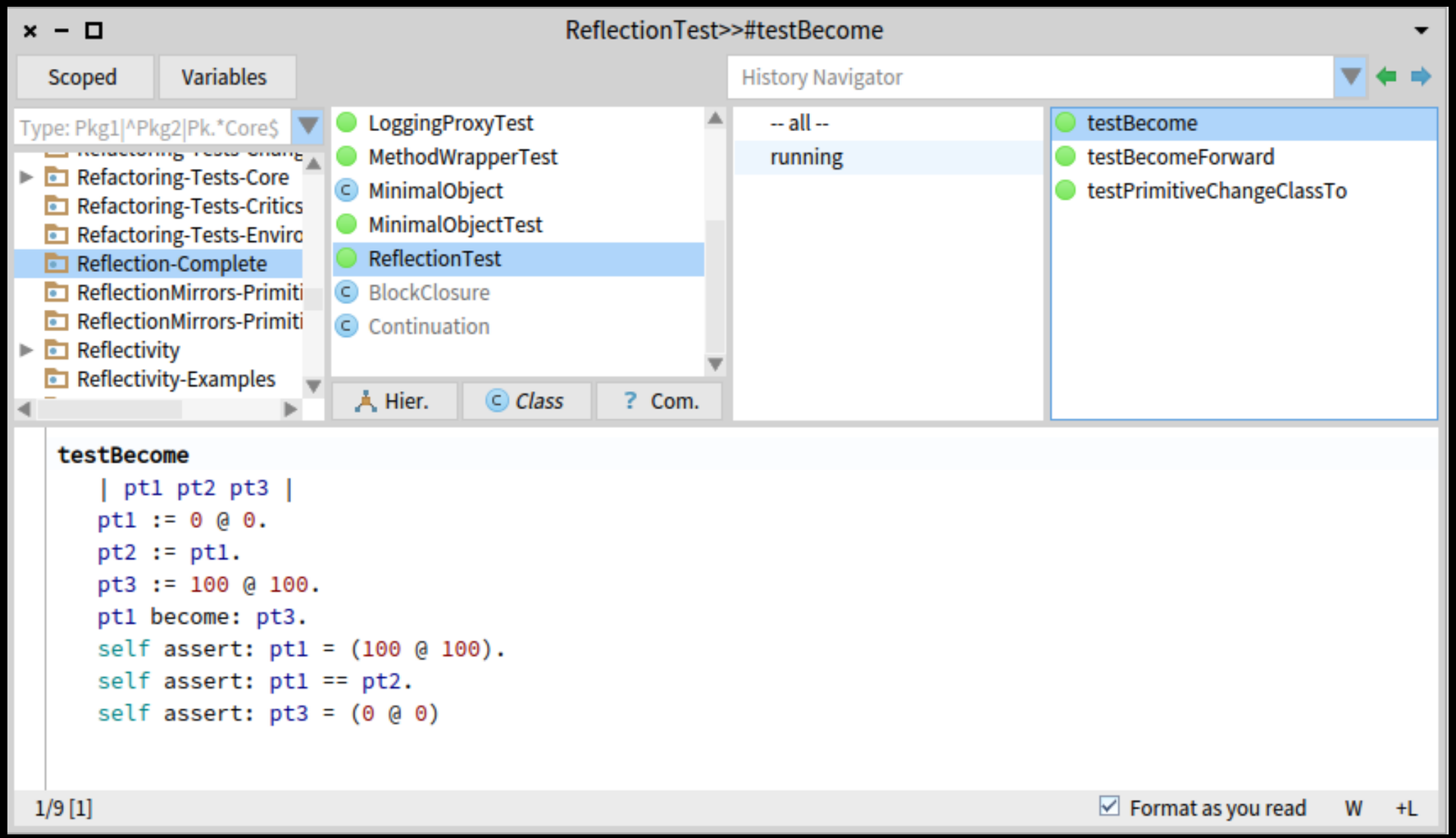
Implementar mètodes específics de la instància*



CAP: Reflexió en Smalltalk

`#become:`

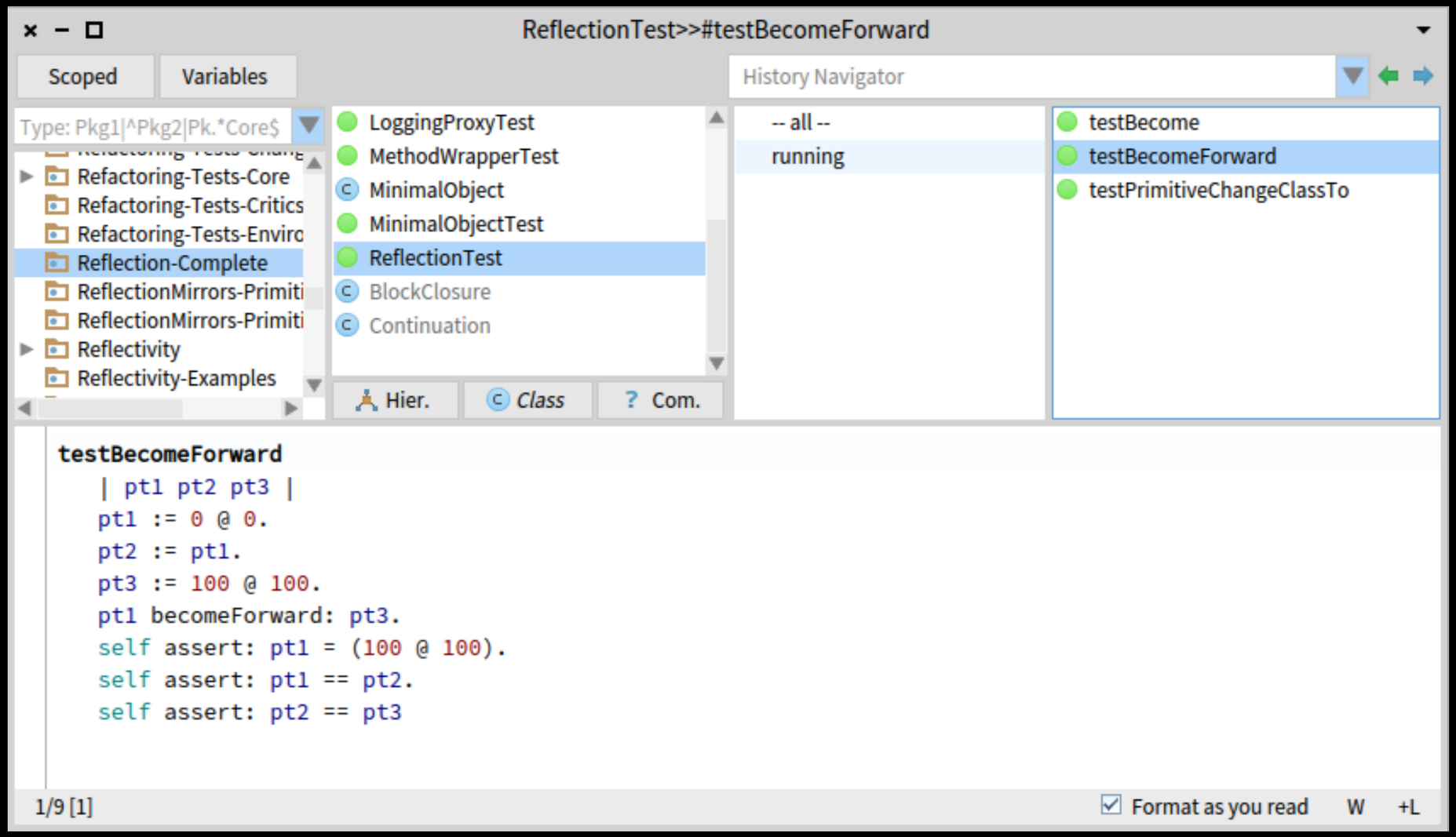
Intercanvia totes les referències d'un objecte a l'altre i vice-versa



CAP: Reflexió en Smalltalk

`#becomeForward:`

Intercanvia totes les referències d'un objecte a l'altre



CAP: Reflexió en Smalltalk

Introspecció:

- Inspeccionar objectes
- ***Consultar el codi***
- Accedir els contexts d'execució

Intercessió:

- Sobreescriure `#doesNotUnderstand:`
- Classes Anònimes
- Method Wrappers
- Continuacions

Part del material extret del treball de Marcus Denker i Stéphane Ducasse

CAP: Reflexió en Smalltalk

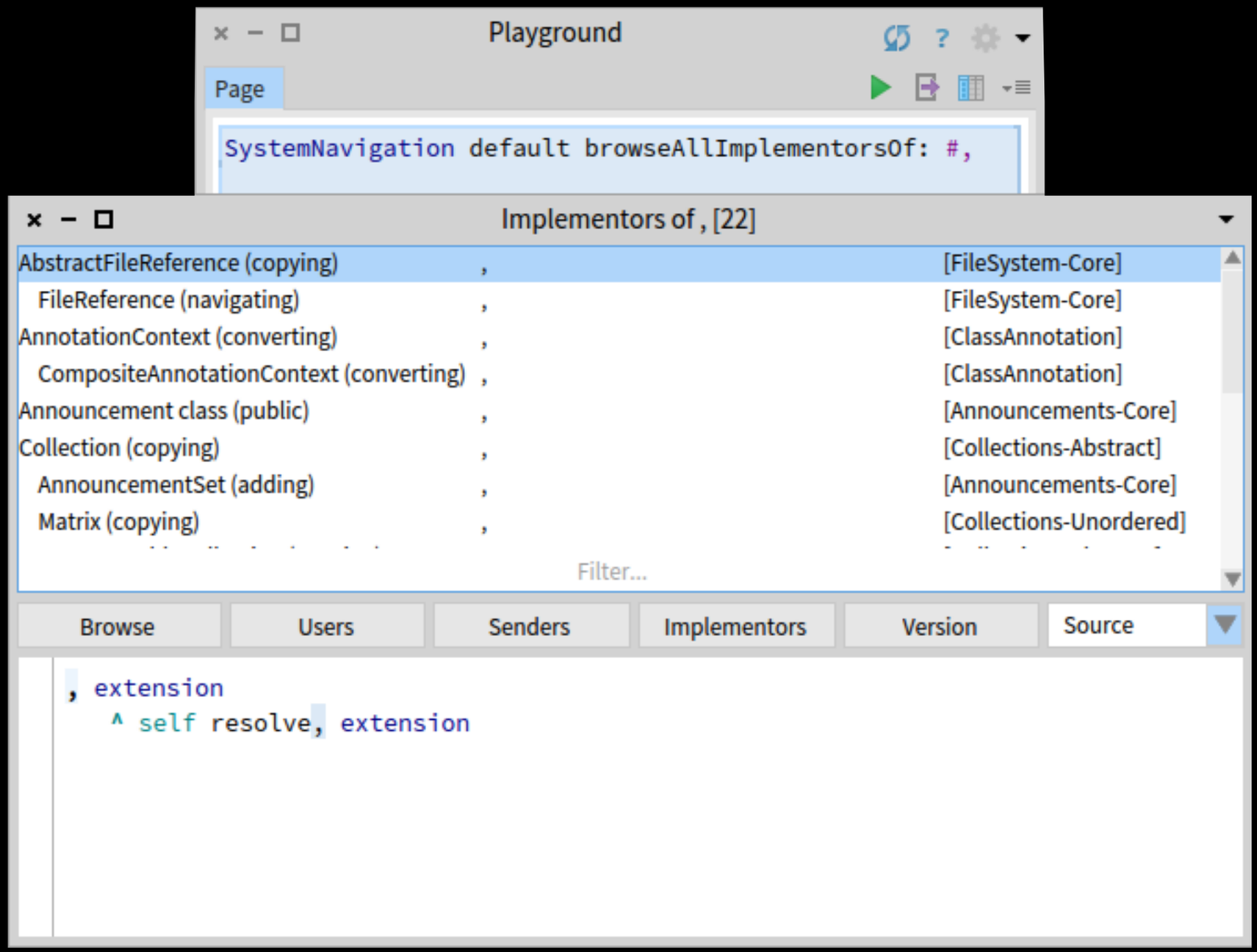
Code Metrics

```
Collection allSuperclasses size → 2
   Collection allSelectors size → 636
Collection allInstVarNames size → 0
     Collection selectors size → 200
  Collection instVarNames size → 0
    Collection subclasses size → 14
 Collection allSubclasses size → 101
       Collection linesOfCode → 1088
```

(en una imatge nova)

CAP: Reflexió en Smalltalk

System Navigation



CAP: Reflexió en Smalltalk

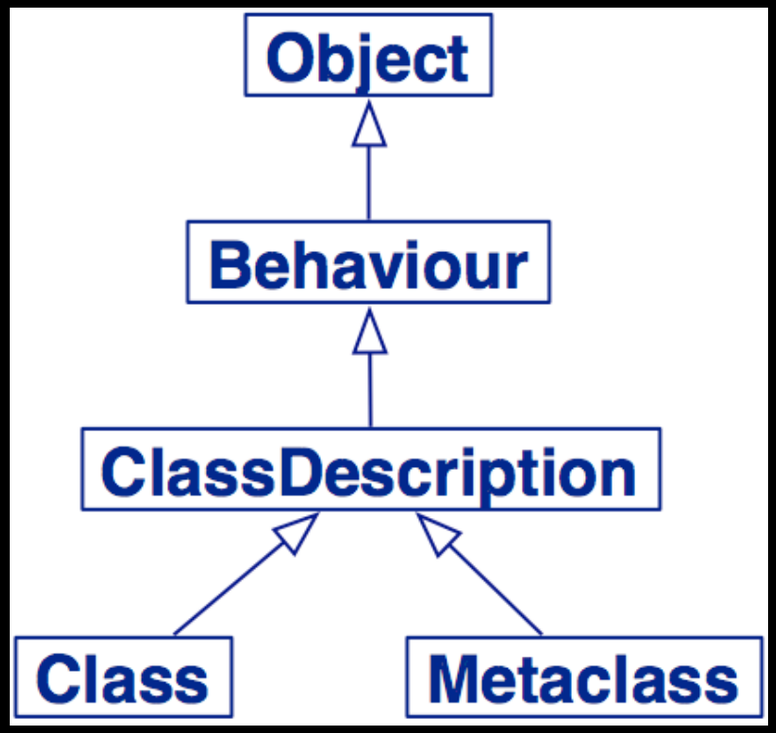
Recapitulem: Les classes també són objectes

`Object`*:*
- Arrel de l'herència
- Comportament per defecte.
- Comportament mínim

`Behavior`*:*
- Essència d'una classe
- Classe anònima.
- Format, diccionari de mètodes, superclasse

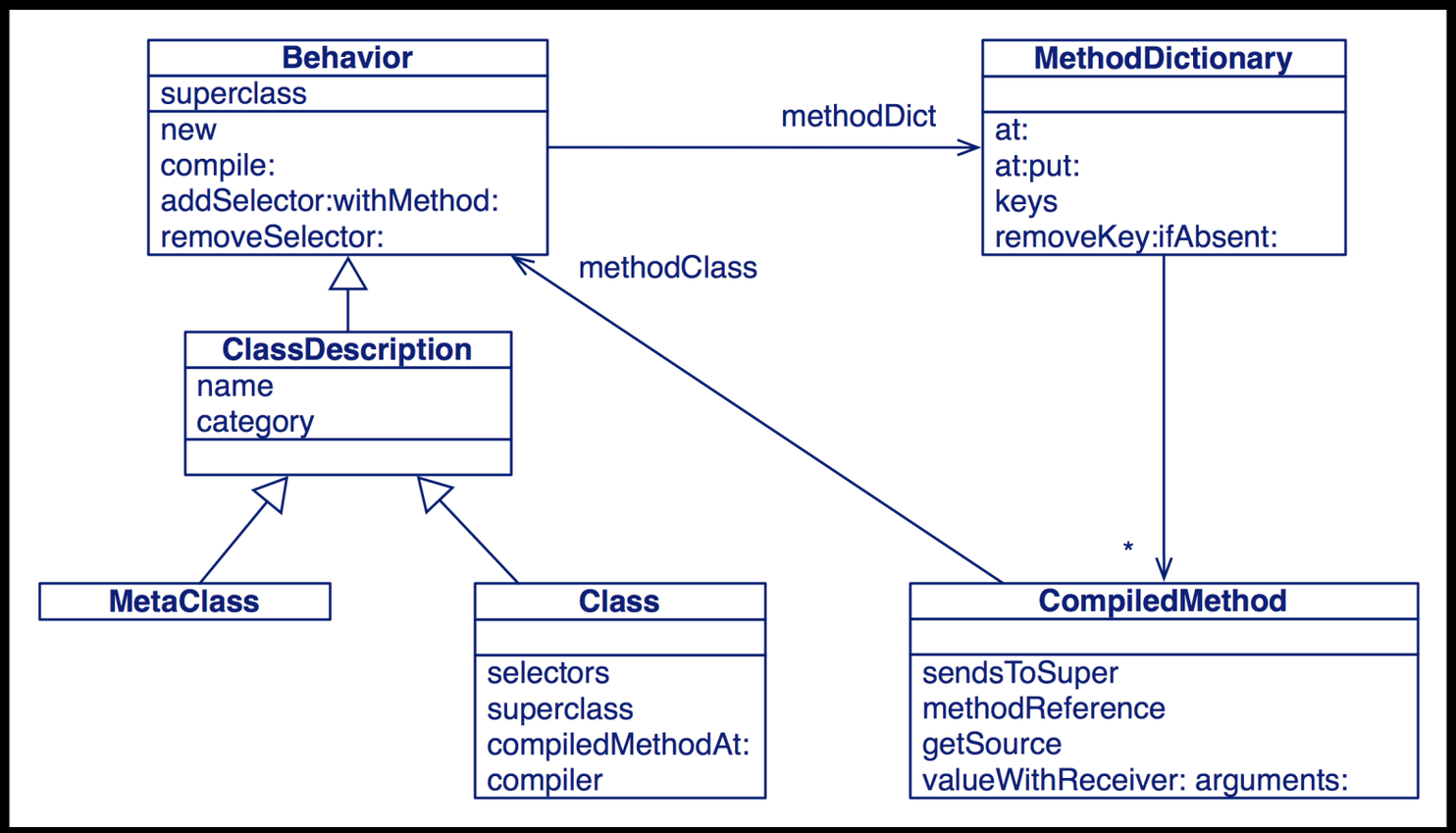
`ClassDescription`*:*
- Representació i organització

`Metaclass`*:*
- Única instància



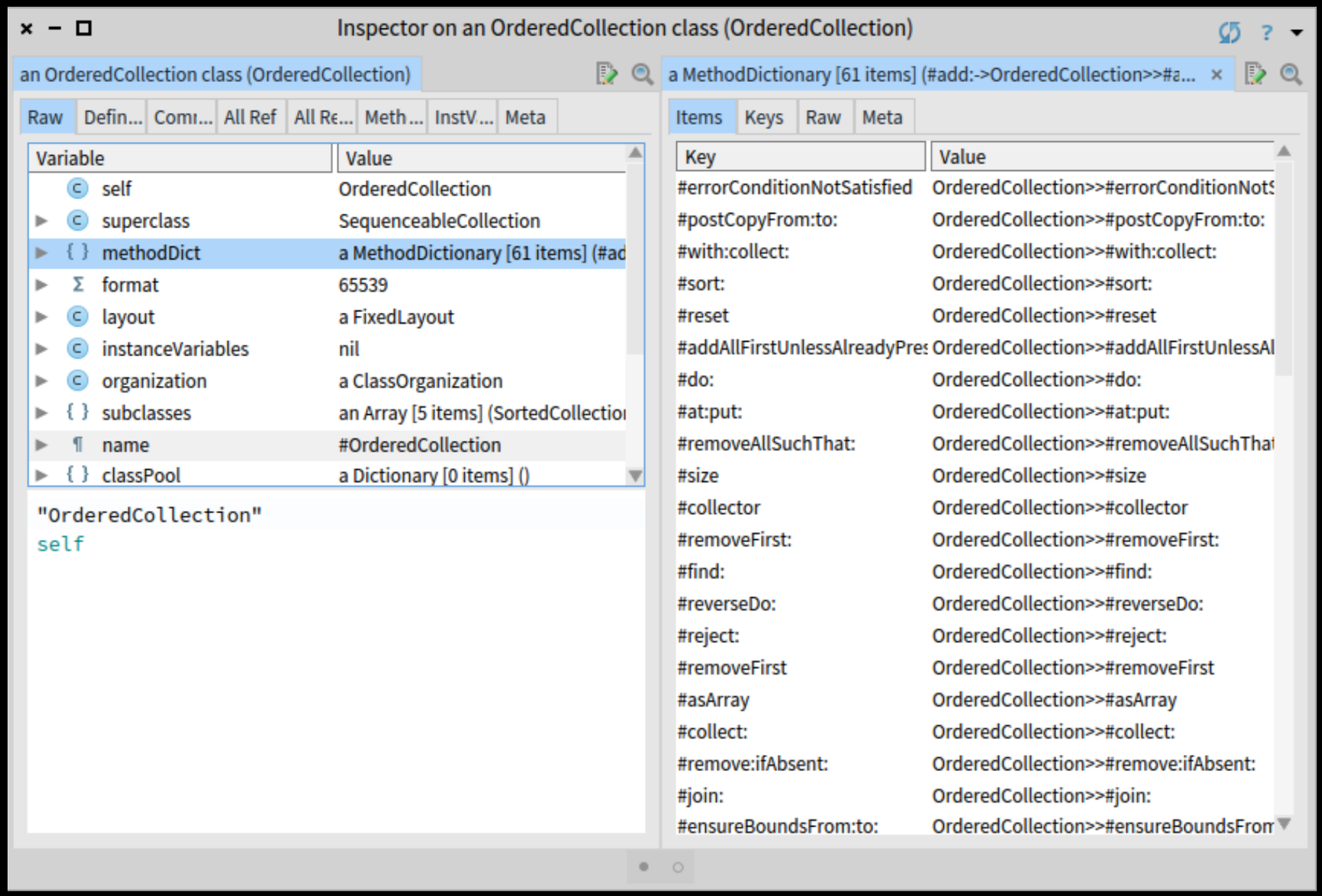
CAP: Reflexió en Smalltalk

Les classes són contenidors de `CompiledMethods`



CAP: Reflexió en Smalltalk

Les classes són contenidors de `CompiledMethods`



CAP: Reflexió en Smalltalk

Invocar un missatge pel nom

```
Object >> #perform:
Object >> #perform:withArguments:
```

Demanar un objecte que executi un missatge.
La cerca de mètodes és l'habitual

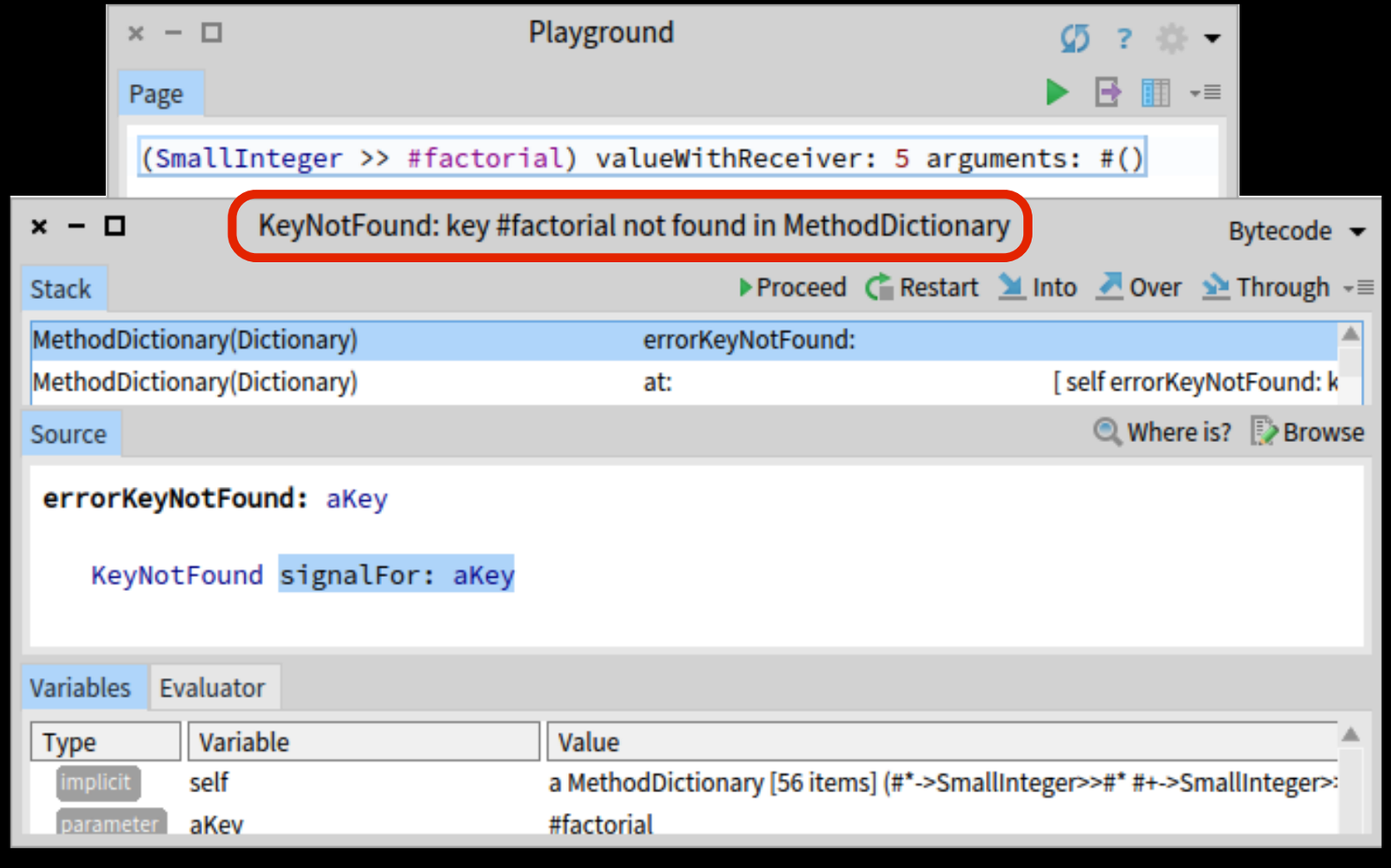
```
          5 factorial  → 120
5 perform: #factorial  → 120
```

CAP: Reflexió en Smalltalk

Executar un mètode compilat

`CompiledMethod >> #valueWithReceiver:arguments:`

*No es fa **cap** cerca de mètodes!!*

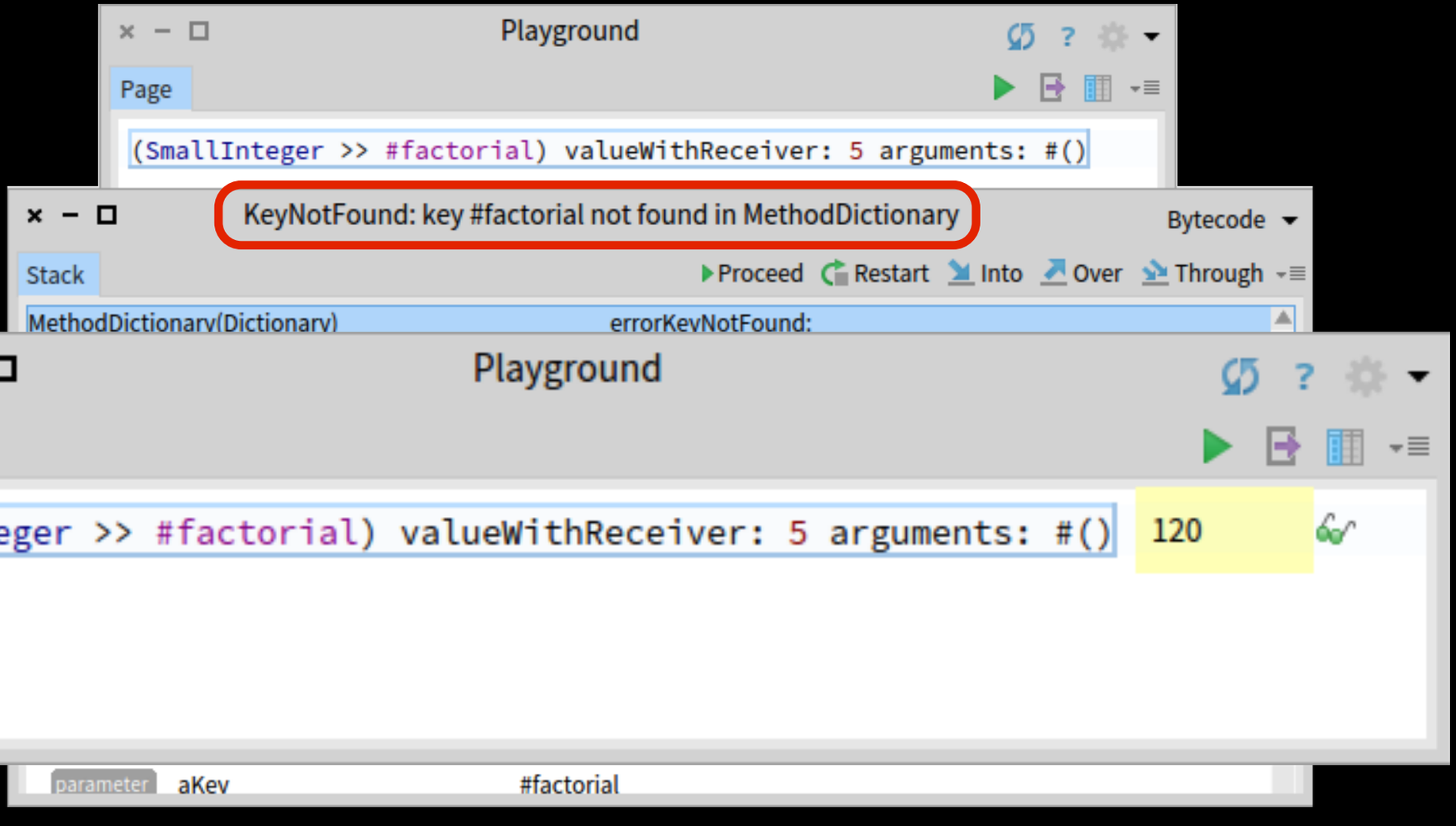


CAP: Reflexió en Smalltalk

Executar un mètode compilat

`CompiledMethod >> #valueWithReceiver:arguments:`

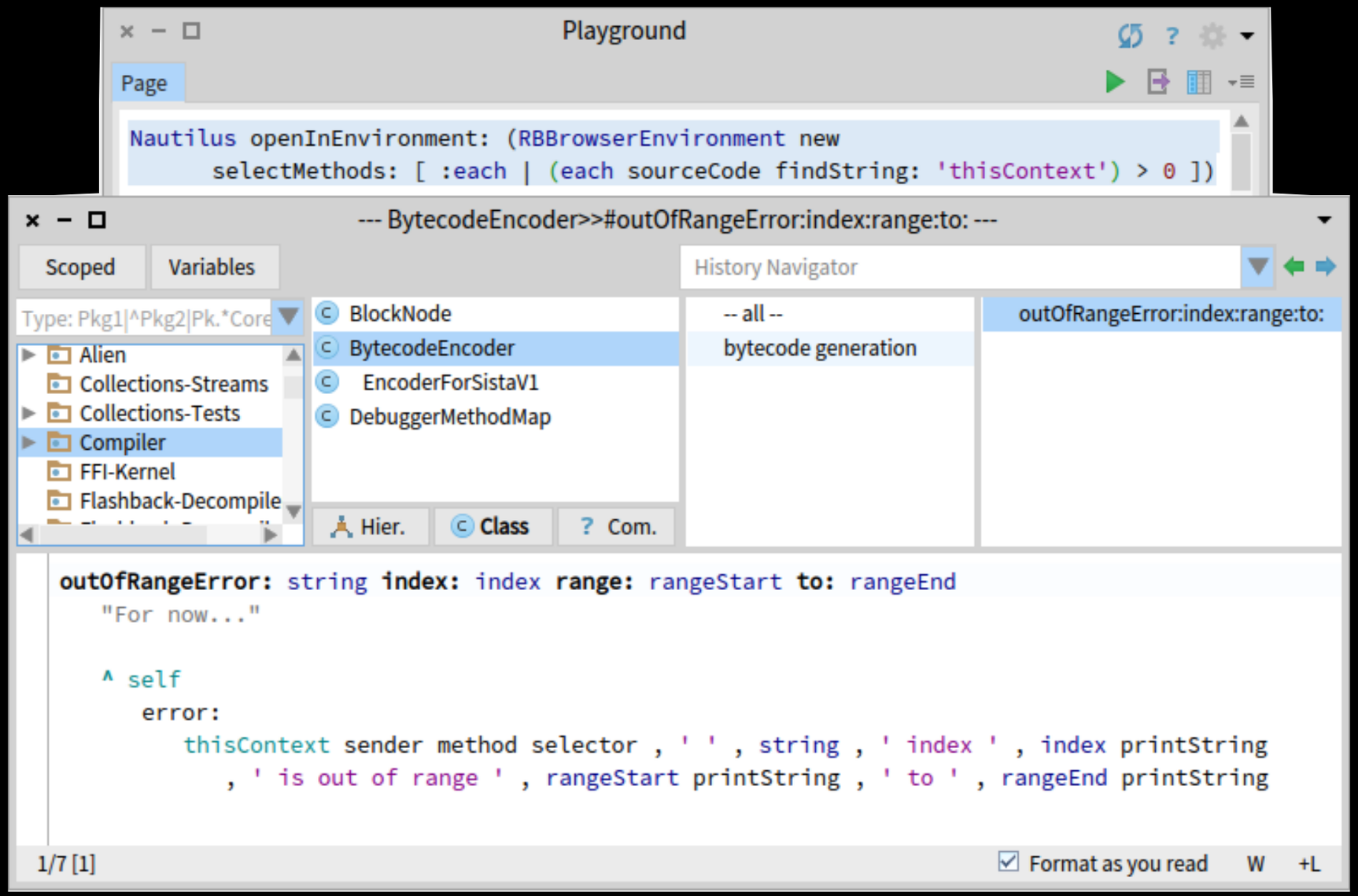
*No es fa **cap** cerca de mètodes!!*



CAP: Reflexió en Smalltalk

Exemple: Trobar els mètodes que fan referència a `thisContext` dins el codi font.

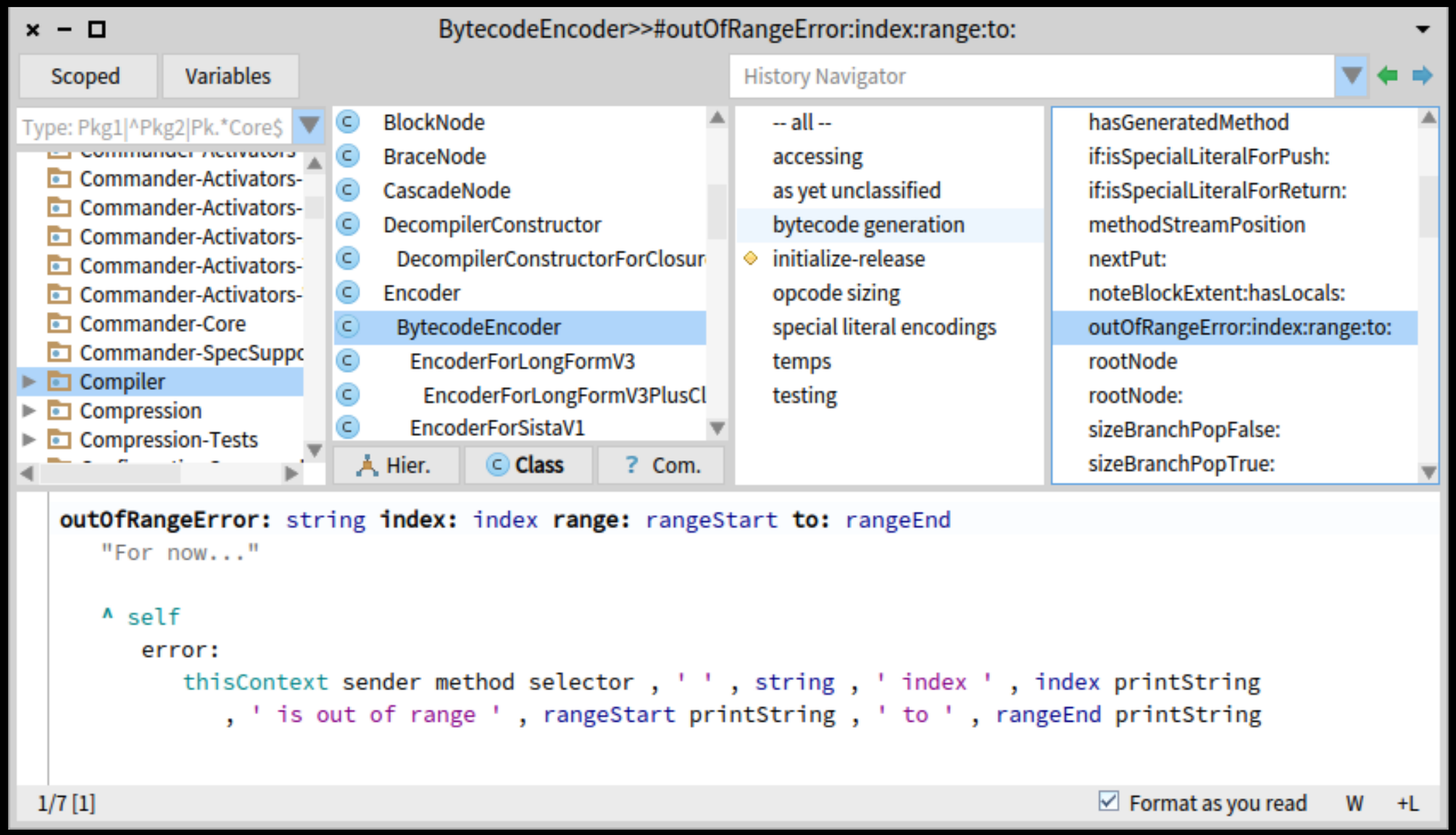
Utilitzant `RBBrowserEnvironment`



CAP: Reflexió en Smalltalk

Exemple: Trobar els mètodes que fan referència a `thisContext` dins el codi font.

Utilitzant `RBBrowserEnvironment`



CAP: Reflexió en Smalltalk

Introspecció:

- Inspeccionar objectes
- Consultar el codi
- ***Accedir els contexts d'execució***

Intercessió:

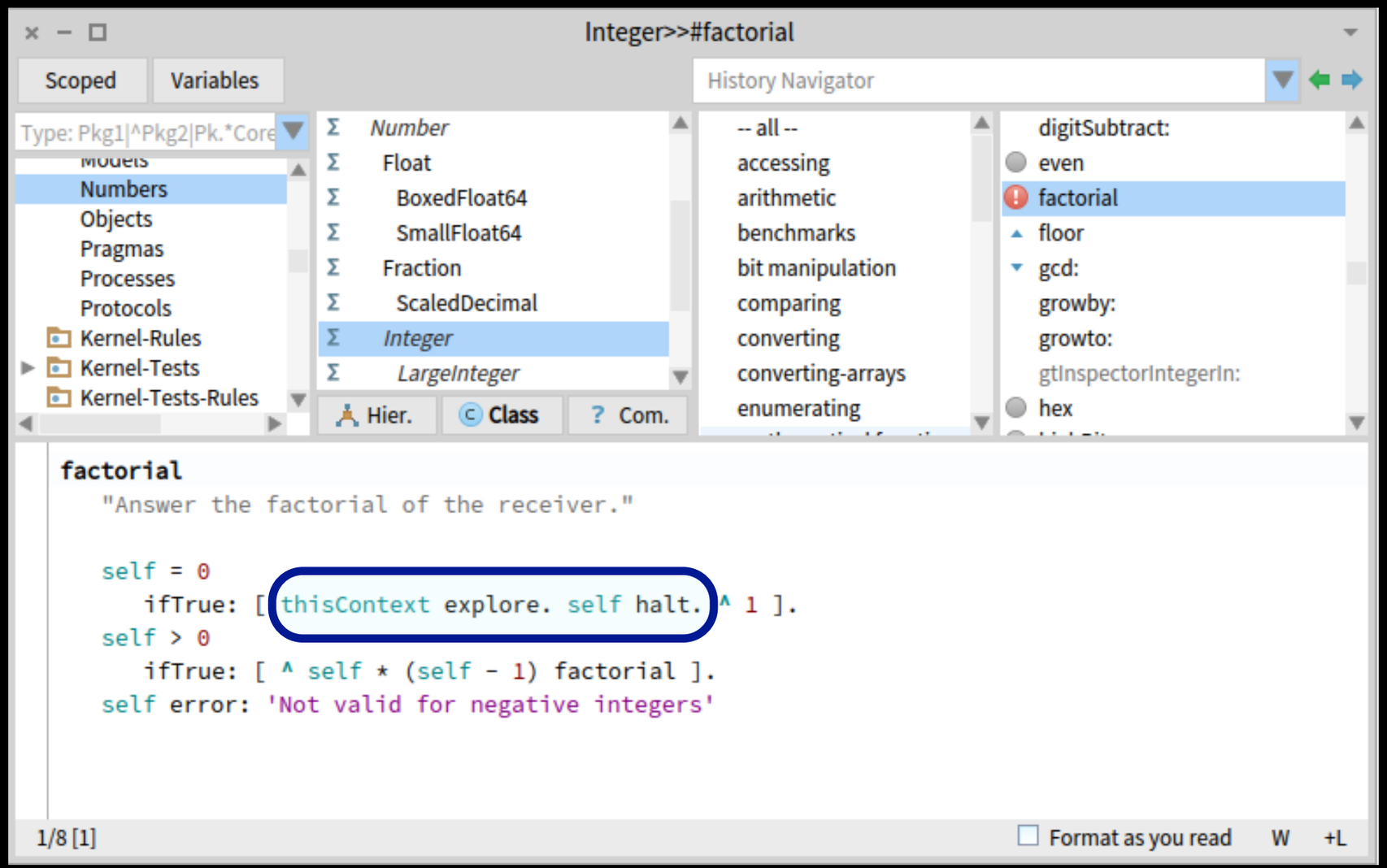
- Sobreescriure `#doesNotUnderstand:`
- Classes Anònimes
- Method Wrappers
- Continuacions

Part del material extret del treball de Marcus Denker i Stéphane Ducasse

CAP: Reflexió en Smalltalk

Accedir a la pila d'execució

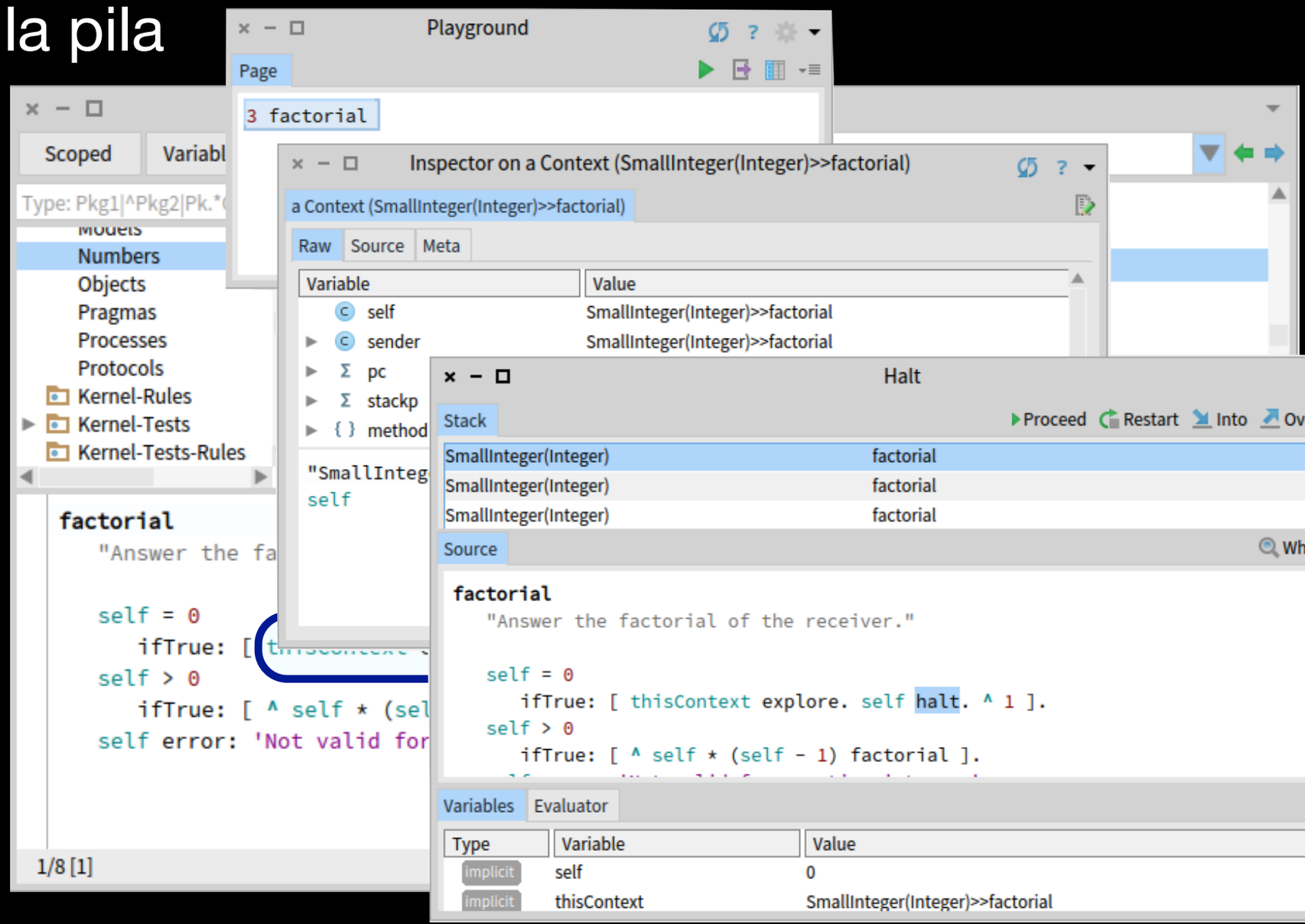
La pila d'execució pot ser ***reificada*** i ***manipulada*** quan desitjem. **`thisContext`** és una pseudo-variable que ens dóna accés a la pila



CAP: Reflexió en Smalltalk

Accedir a la pila d'execució

La pila d'execució pot ser ***reificada*** i ***manipulada*** quan desitjem. **`thisContext`** és una pseudo-variable que ens dóna accés a la pila



CAP: Reflexió en Smalltalk

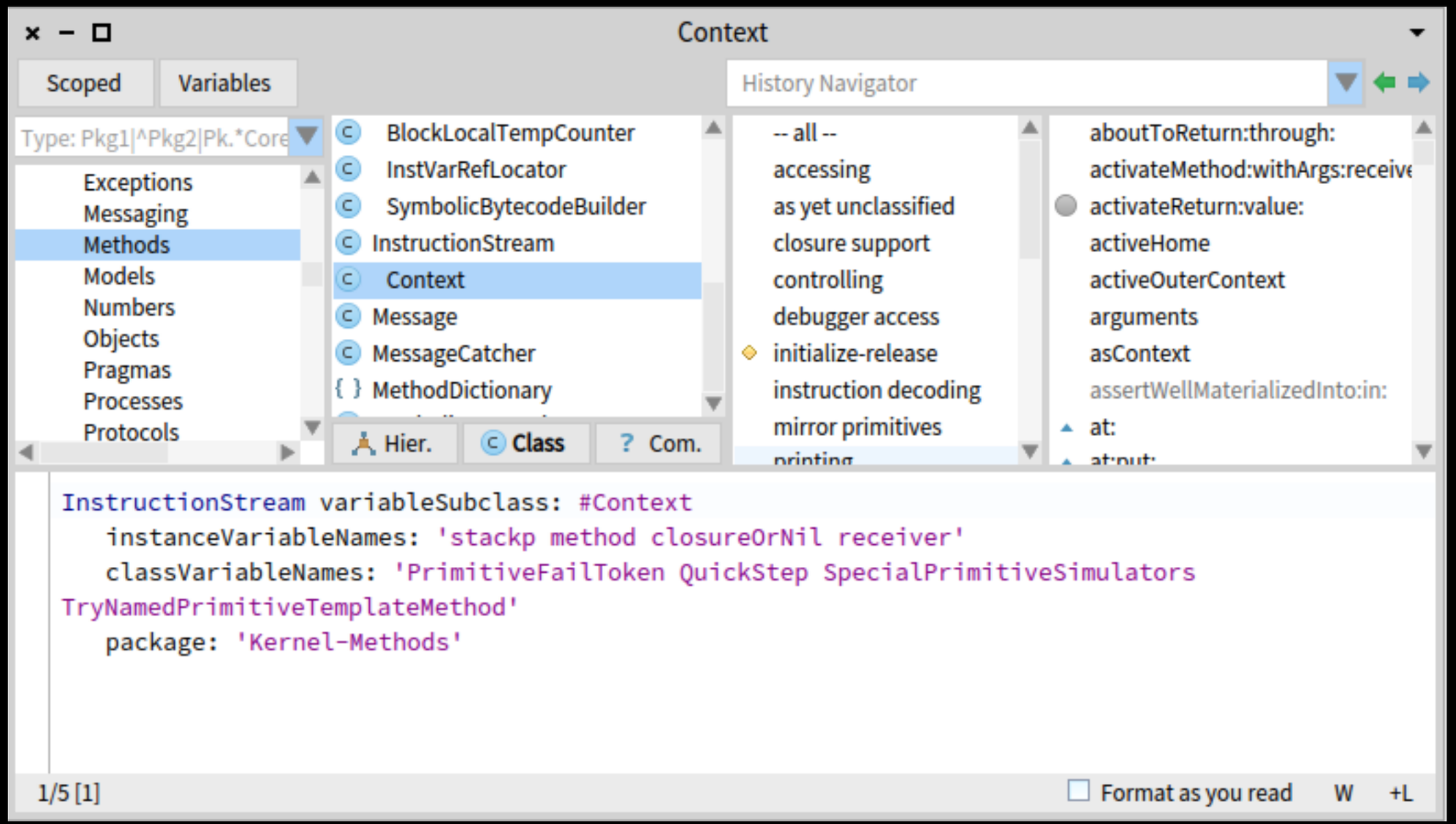
Què passa quan s'executa un mètode?

Cal espai per:
- variables temporals
- recordar on hem de retornar

TOT és un objecte!
- Modelitzem aquest espai amb objectes
- Classe **`Context`**

CAP: Reflexió en Smalltalk

Què passa quan s'executa un mètode?



CAP: Reflexió en Smalltalk

`Context`

`Context` gestiona l'espai associat a l'execució d'un **`CompiledMethod`**

- *Program counter* (pc)
- el mètode en sí
- *sender* i *receiver*

El *sender* és el previ **`Context`**

- La cadena de *senders* és una pila
- creix i decreix amb les invocacions i els retorns

CAP: Reflexió en Smalltalk

Exemple: Aturada contextual

No podem posar **`halt`** en mètodes que s'utilitzen sovint (p.ex. **`OrderedCollection>>#add:`**)

Idea: aturar-se només si el mètode ha estat invocat des d'algun altre mètode amb un *selector* determinat. Podriem implementar-lo així

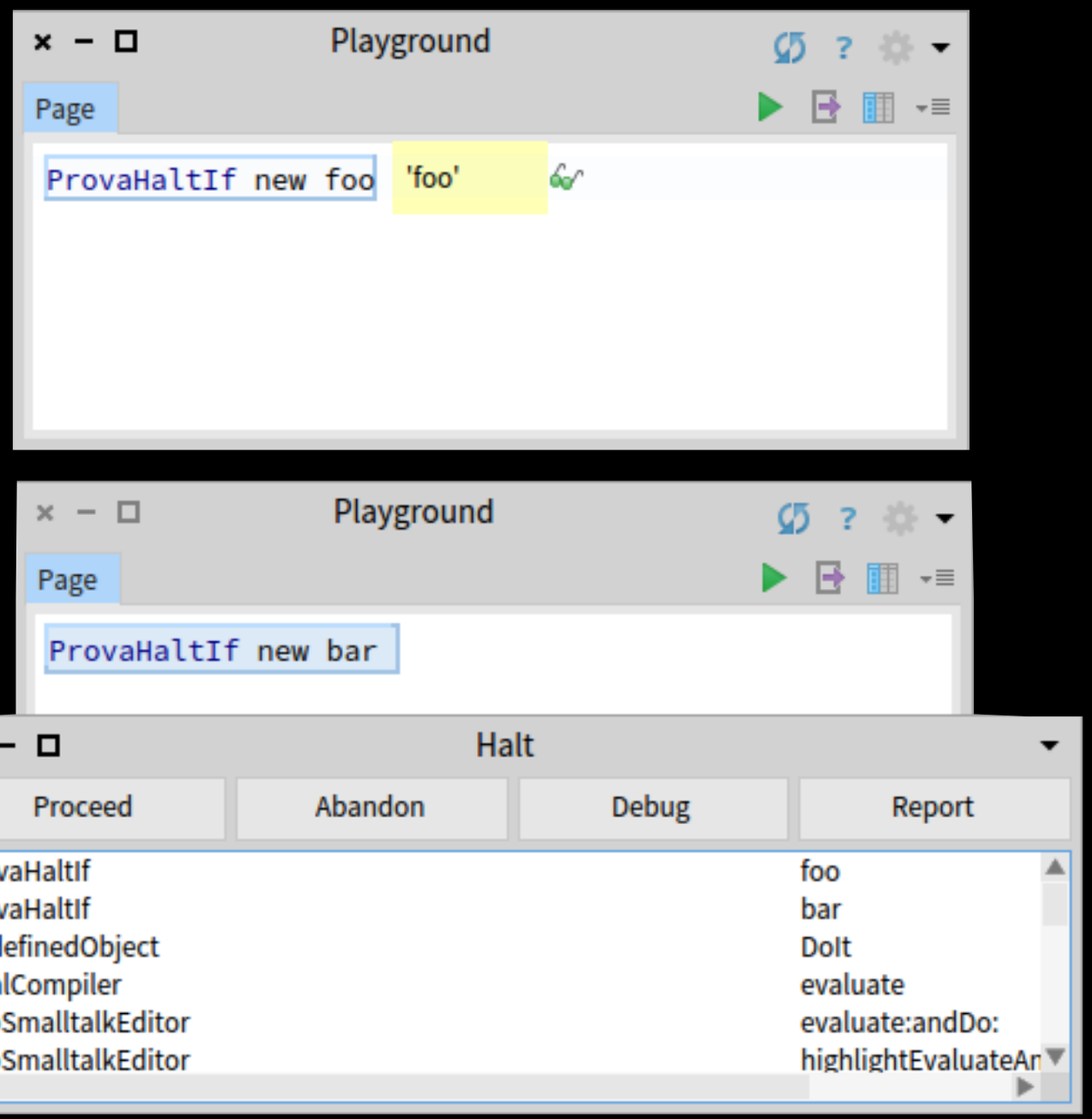
```
haltIf: aSelector
  | context |
  context := thisContext.
  [context sender isNil]
    whileFalse:
      [context := context sender.
      (context selector = aSelector)
          ifTrue: [ Halt signal ] ].
```

CAP: Reflexió en Smalltalk

Object >> haltIf:

```
ProvaHaltIf >> foo
  self haltIf: #bar.
  ^ 'foo'
```

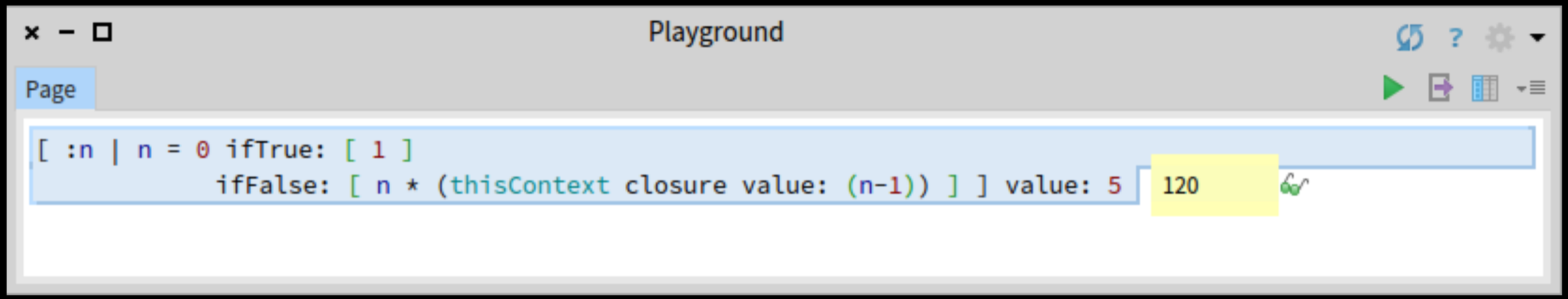
```
ProvaHaltIf >> bar
  ^ (self foo), 'bar'
```



CAP: Reflexió en Smalltalk

Exemple: Recursivitat en blocs anònims

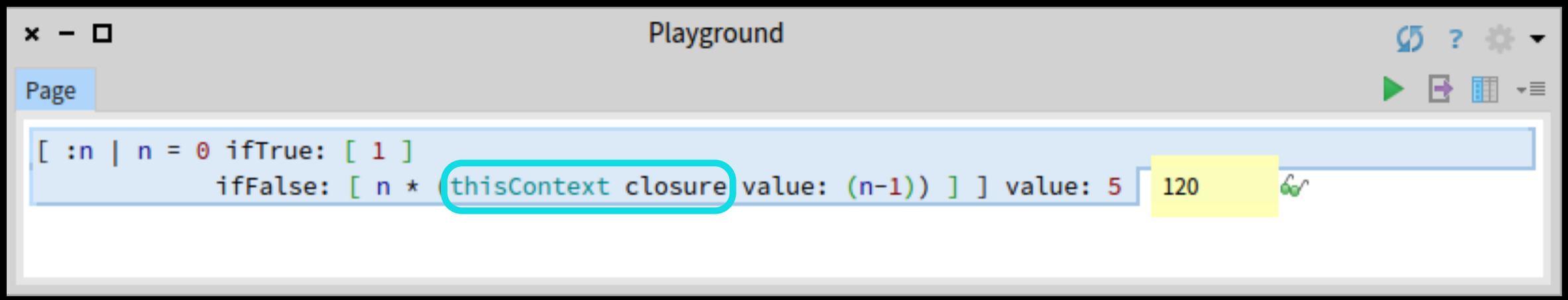
Com un bloc no referenciat per cap variable pot cridar-se ell mateix?



CAP: Reflexió en Smalltalk

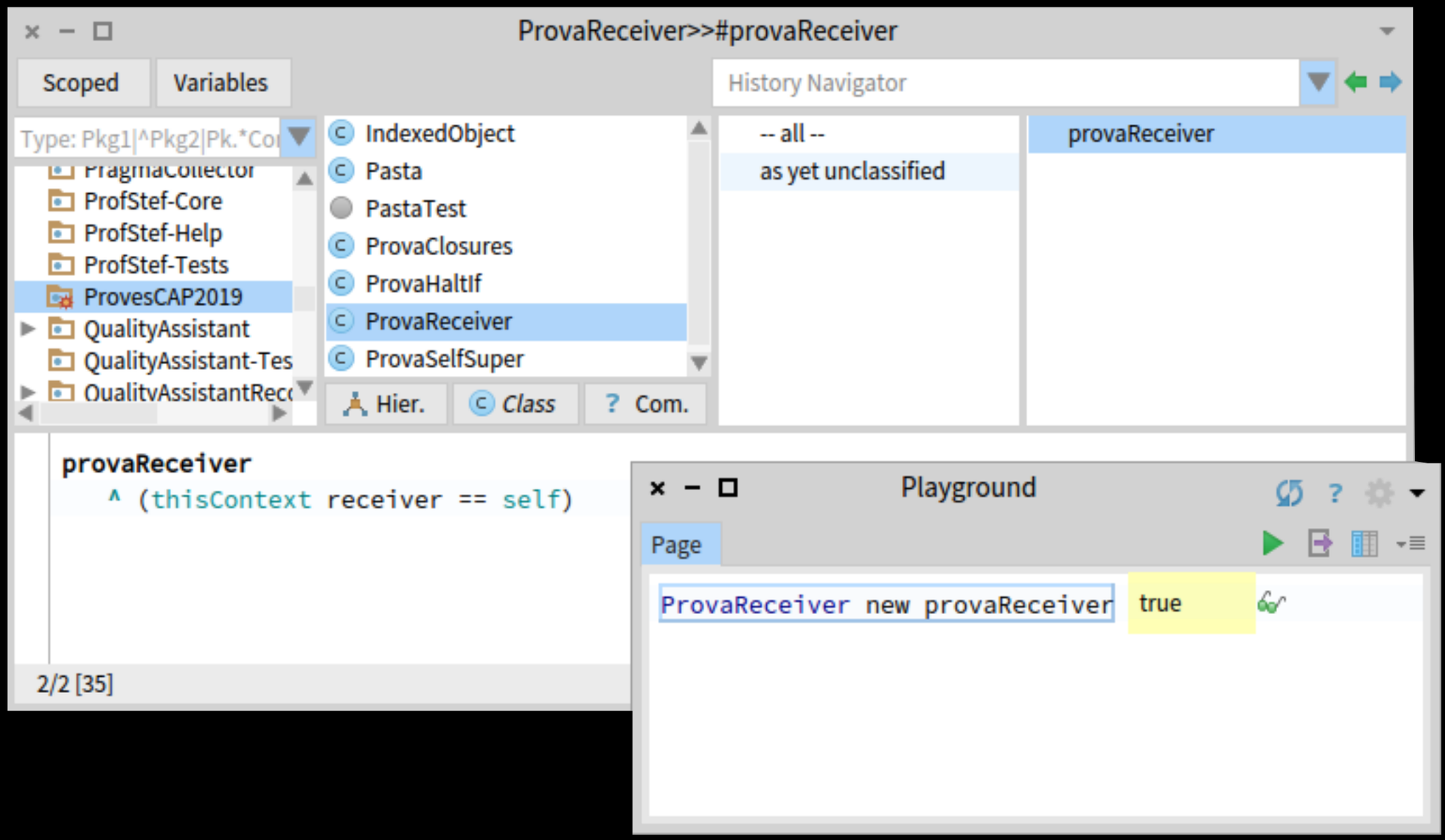
Exemple: Recursivitat en blocs anònims

Com un bloc no referenciat per cap variable pot cridar-se ell mateix?



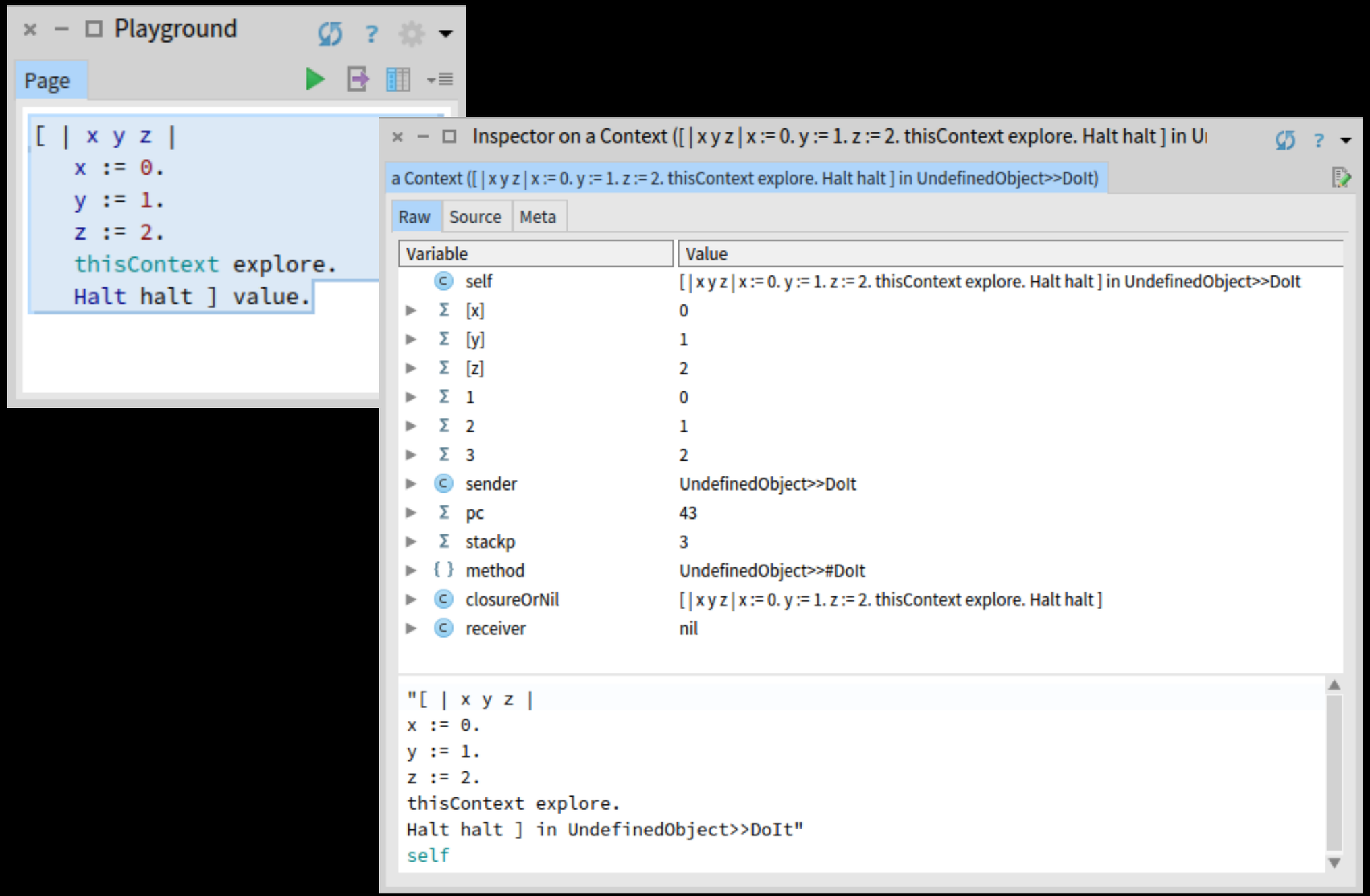
CAP: Reflexió en Smalltalk

Exemple: Cal `self` si tinc `thisContext`?



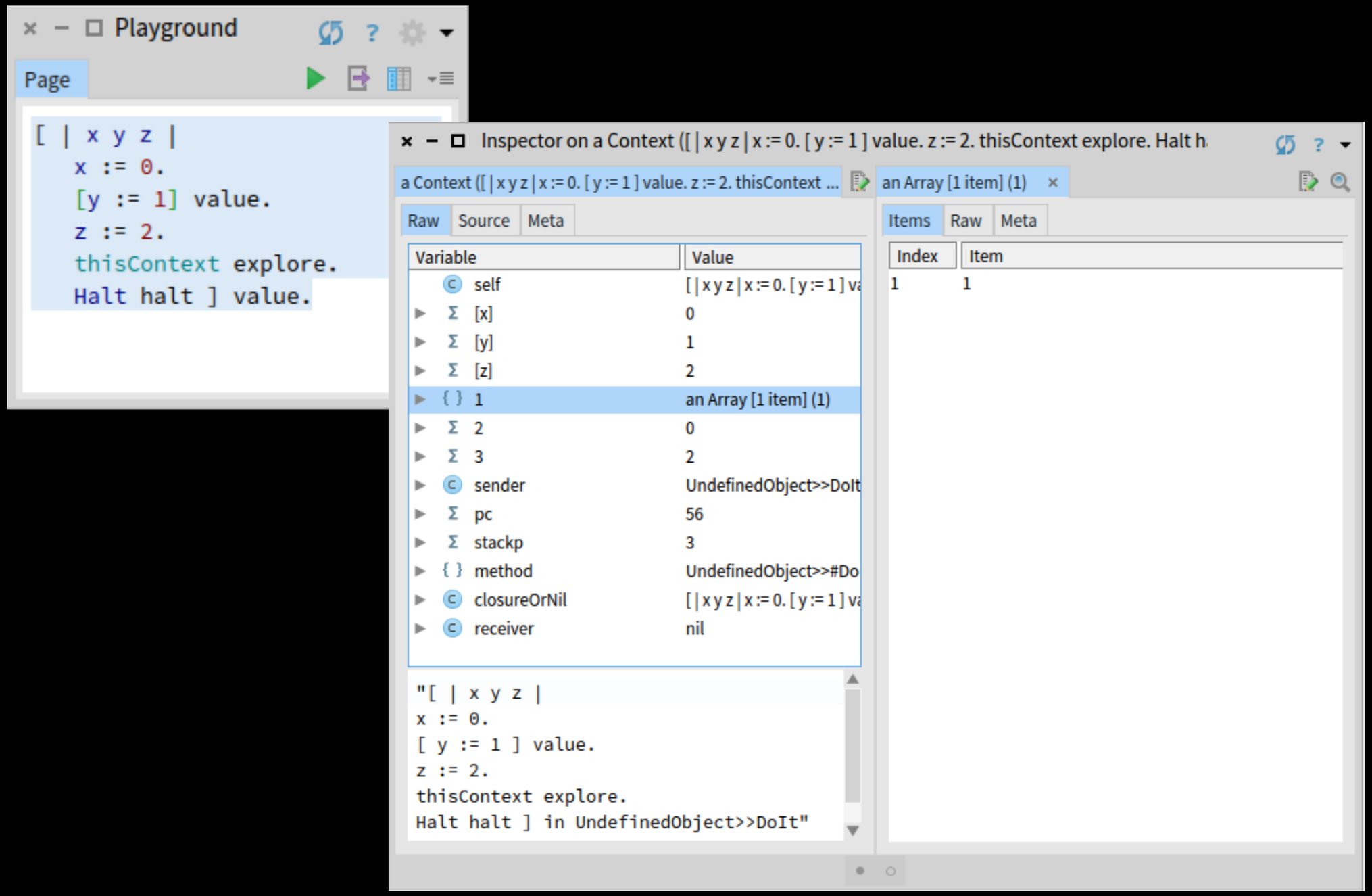
CAP: Reflexió en Smalltalk

Què es guarda en un `Context`?



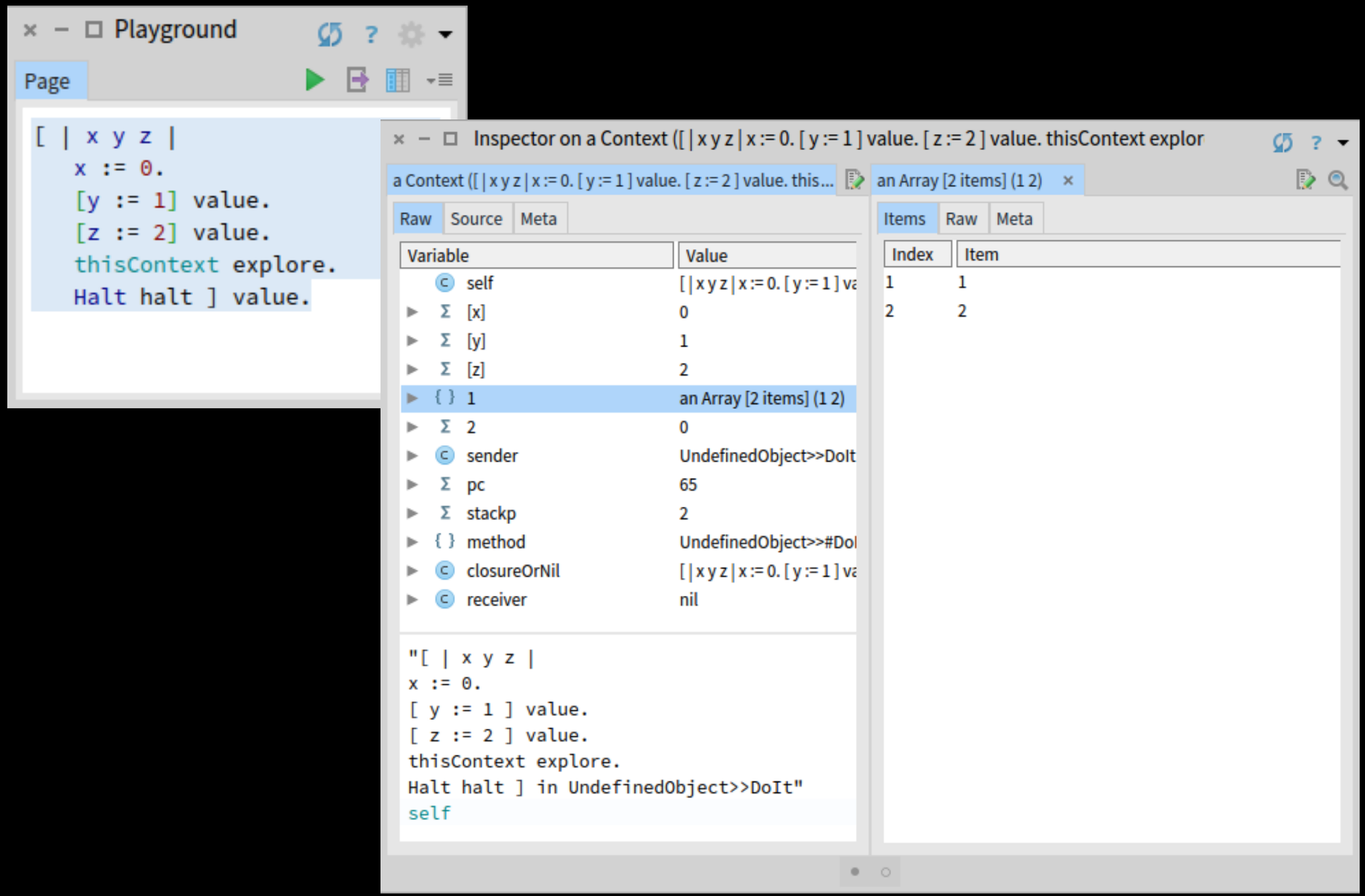
CAP: Reflexió en Smalltalk

Què es guarda en un `Context`?



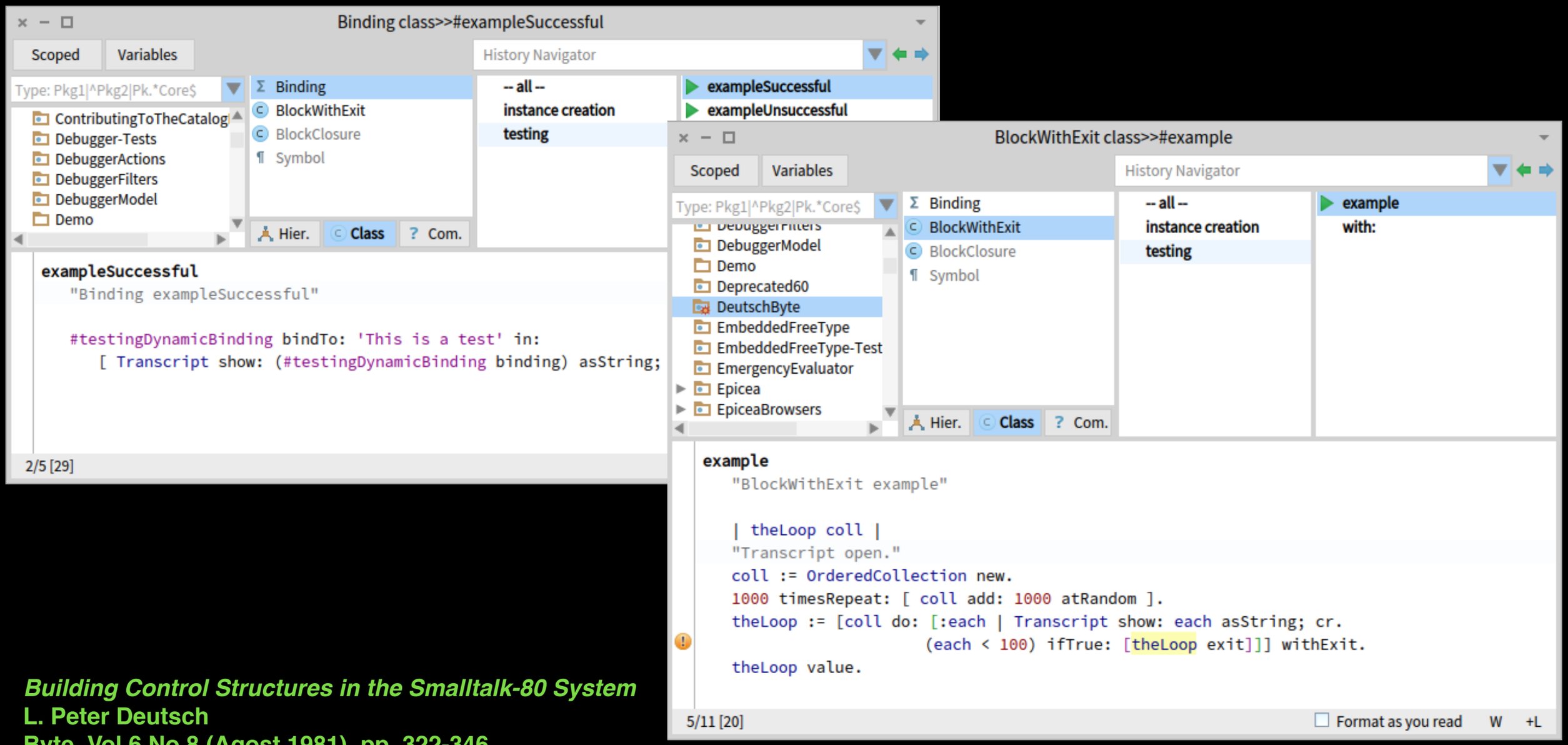
CAP: Reflexió en Smalltalk

Què es guarda en un `Context`?



CAP: Reflexió en Smalltalk

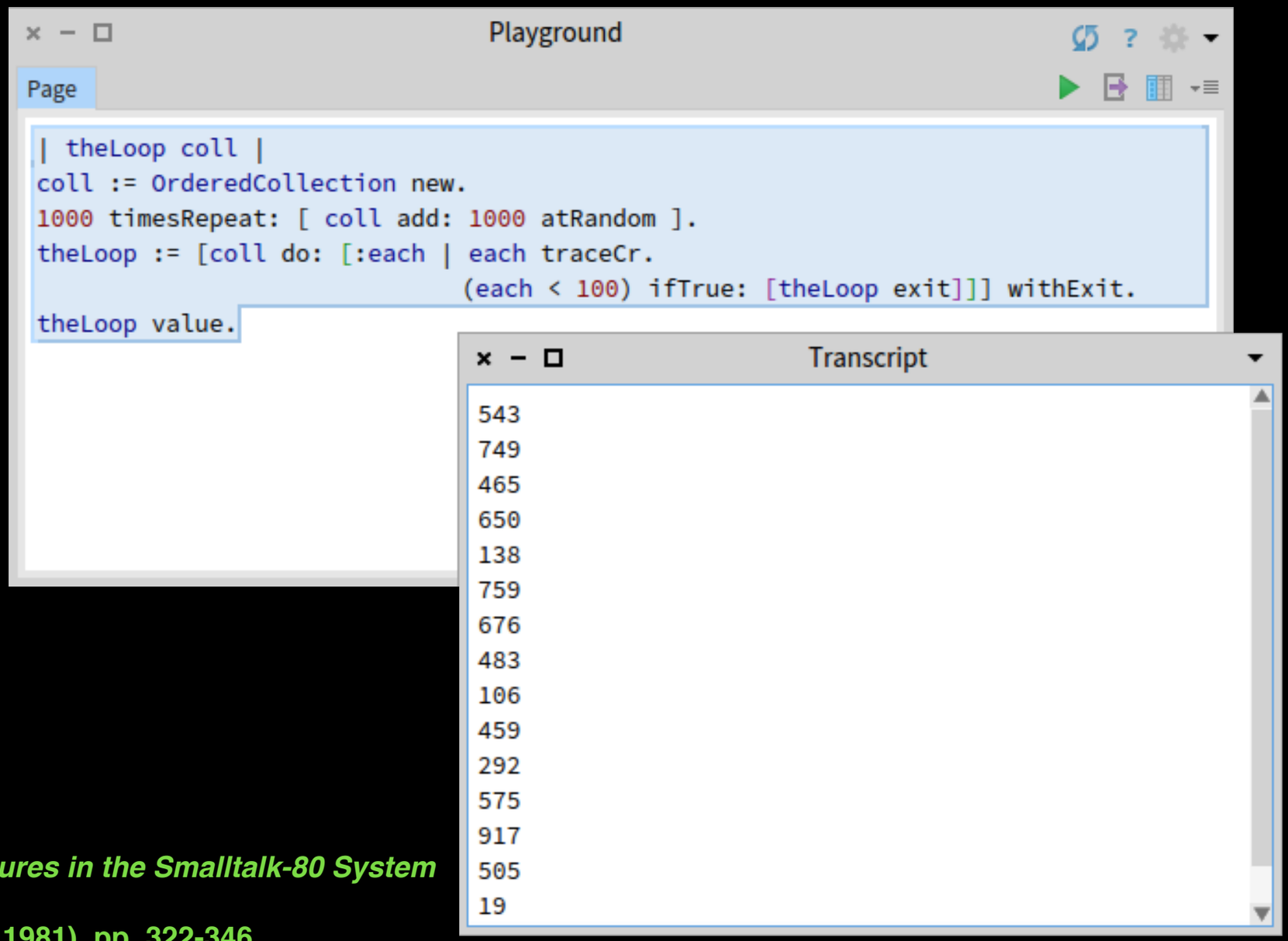
Exemples: `BlockWithExit` & `Binding`



Building Control Structures in the Smalltalk-80 System
L. Peter Deutsch
Byte, Vol.6 No.8 (Agost 1981), pp. 322-346
`https://archive.org/details/byte-magazine-1981-08/page/n335`

CAP: Reflexió en Smalltalk

Exemple: Puc tenir un `BlockWithExit` anònim?



Building Control Structures in the Smalltalk-80 System
L. Peter Deutsch
Byte, Vol.6 No.8 (Agost 1981), pp. 322-346
`https://archive.org/details/byte-magazine-1981-08/page/n335`

CAP: Reflexió en Smalltalk

Exemple: Puc tenir un `BlockWithExit` anònim?

Playground

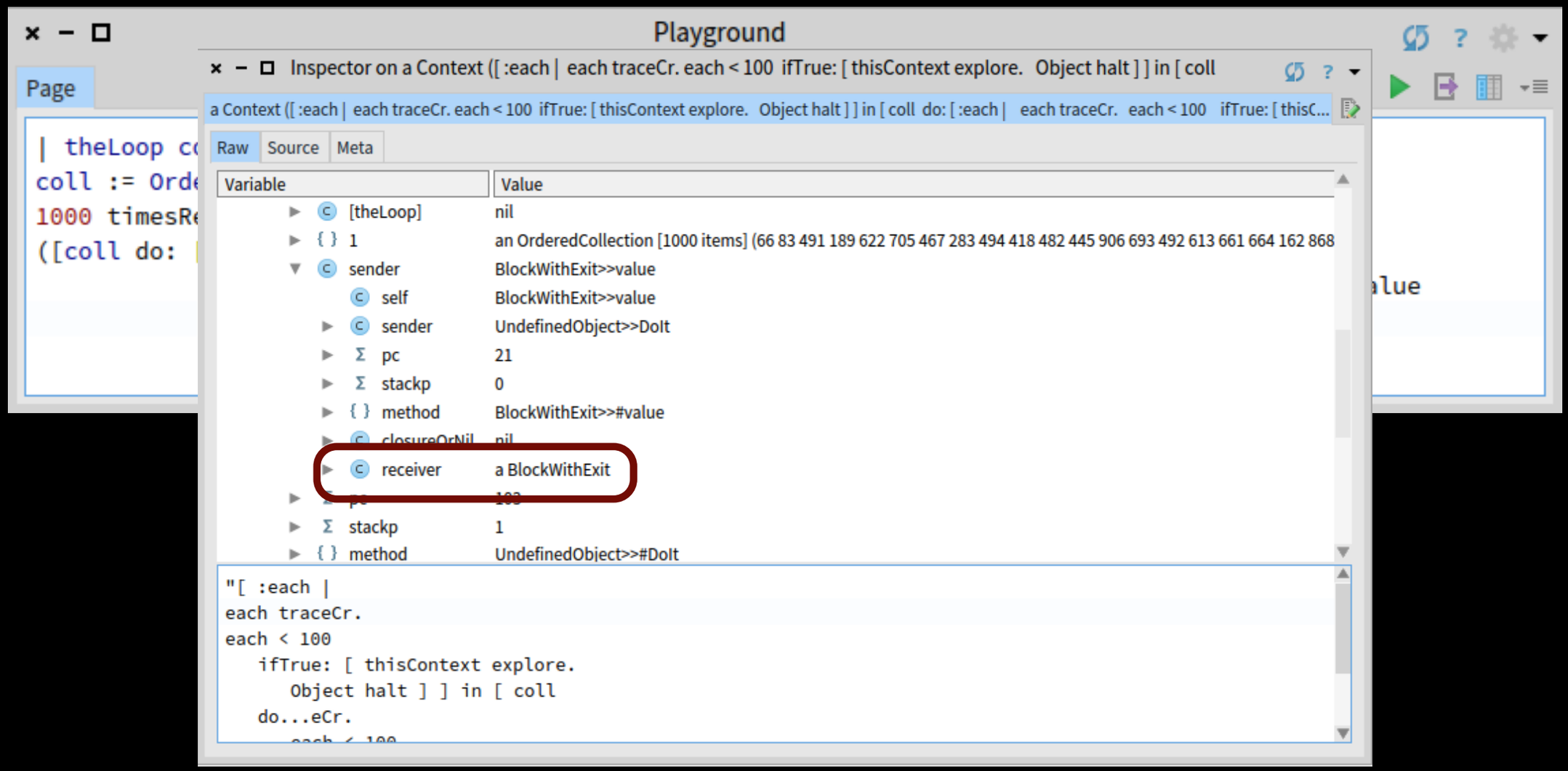
Page

```
| theLoop coll |
coll := OrderedCollection new.
1000 timesRepeat: [ coll add: 1000 atRandom ].
([coll do: [:each | each traceCr.
                (each < 100) ifTrue: [thisContext explore. Object halt "exit"]]] withExit) value
```

Building Control Structures in the Smalltalk-80 System
L. Peter Deutsch
Byte, Vol.6 No.8 (Agost 1981), pp. 322-346
https://archive.org/details/byte-magazine-1981-08/page/n335

CAP: Reflexió en Smalltalk

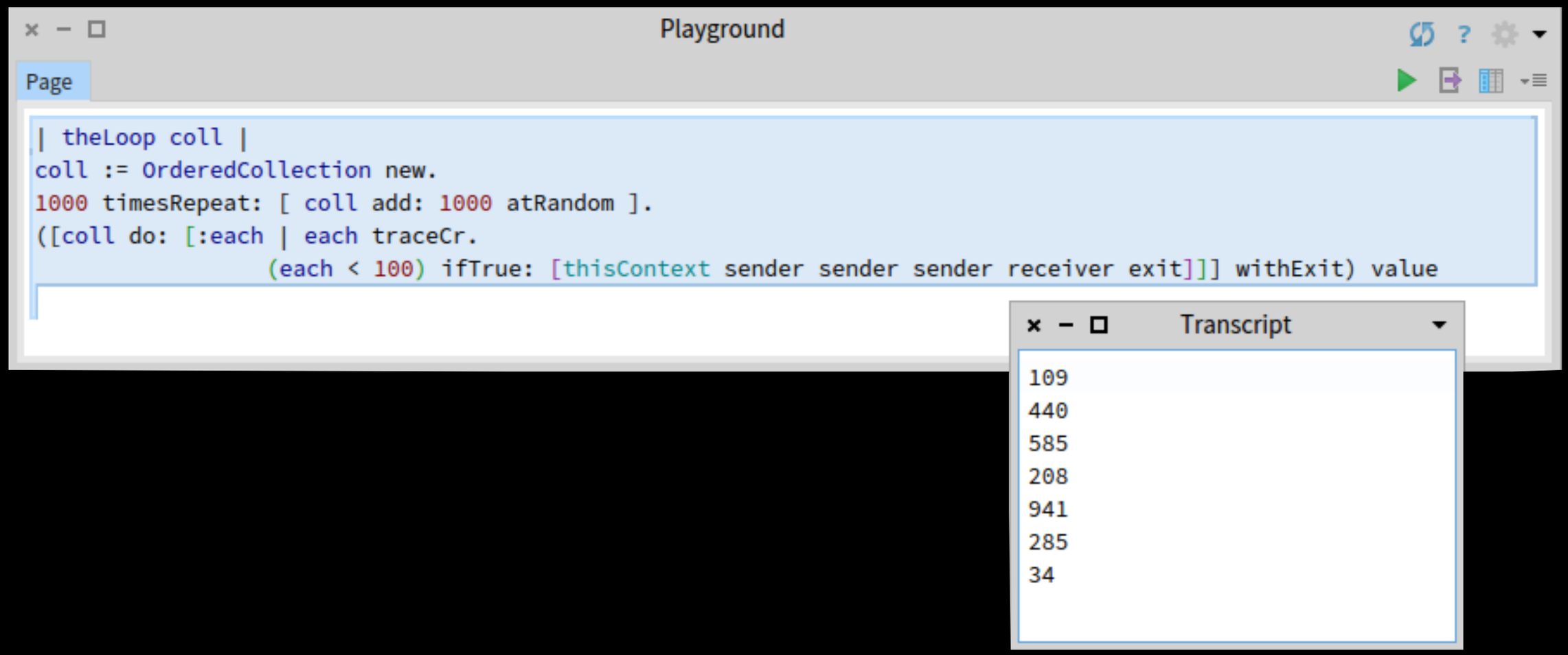
Exemple: Puc tenir un `BlockWithExit` anònim?



Building Control Structures in the Smalltalk-80 System
L. Peter Deutsch
Byte, Vol.6 No.8 (Agost 1981), pp. 322-346
`https://archive.org/details/byte-magazine-1981-08/page/n335`

CAP: Reflexió en Smalltalk

Exemple: Puc tenir un `BlockWithExit` anònim?



Building Control Structures in the Smalltalk-80 System
L. Peter Deutsch
Byte, Vol.6 No.8 (Agost 1981), pp. 322-346
`https://archive.org/details/byte-magazine-1981-08/page/n335`

CAP: Reflexió en Smalltalk

Introspecció:

- Inspeccionar objectes
- Consultar el codi
- Accedir els contexts d'execució

Intercessió:

- ***Sobreescriure*** **`#doesNotUnderstand:`**
- Classes Anònimes
- Method Wrappers
- Continuacions

Part del material extret del treball de Marcus Denker i Stéphane Ducasse

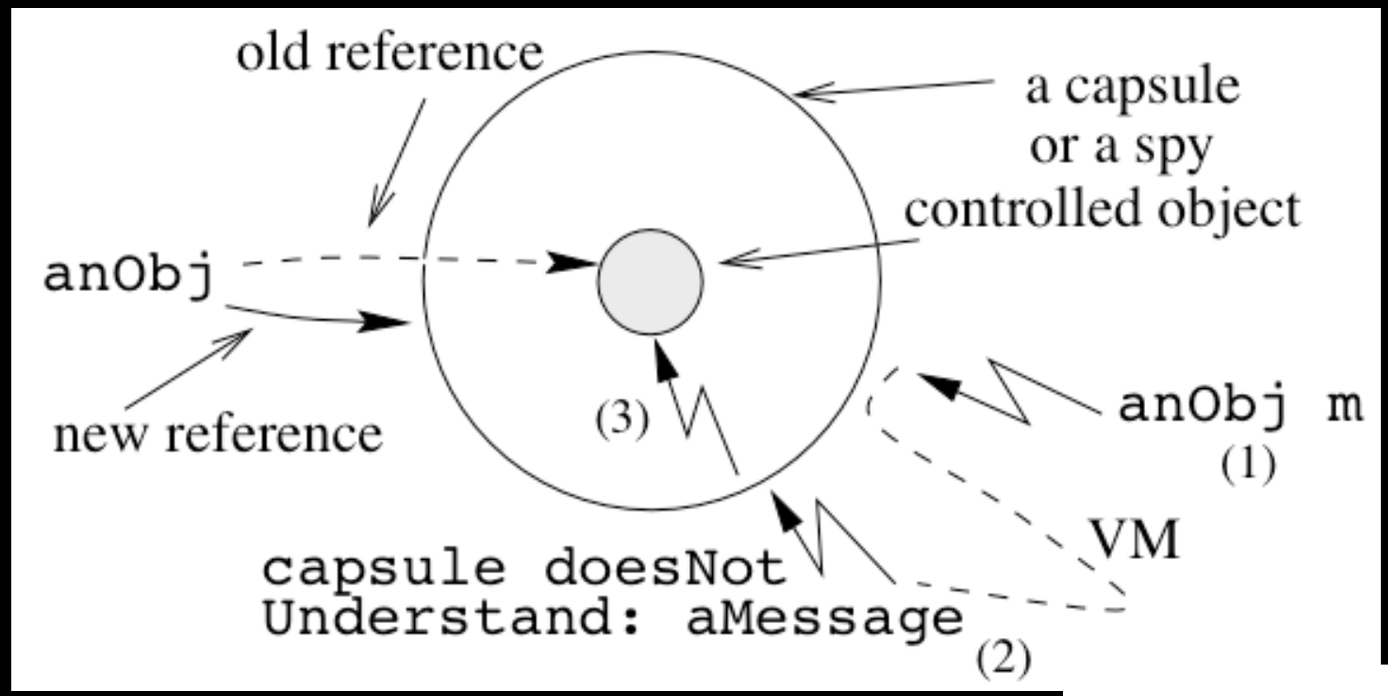
CAP: Reflexió en Smalltalk

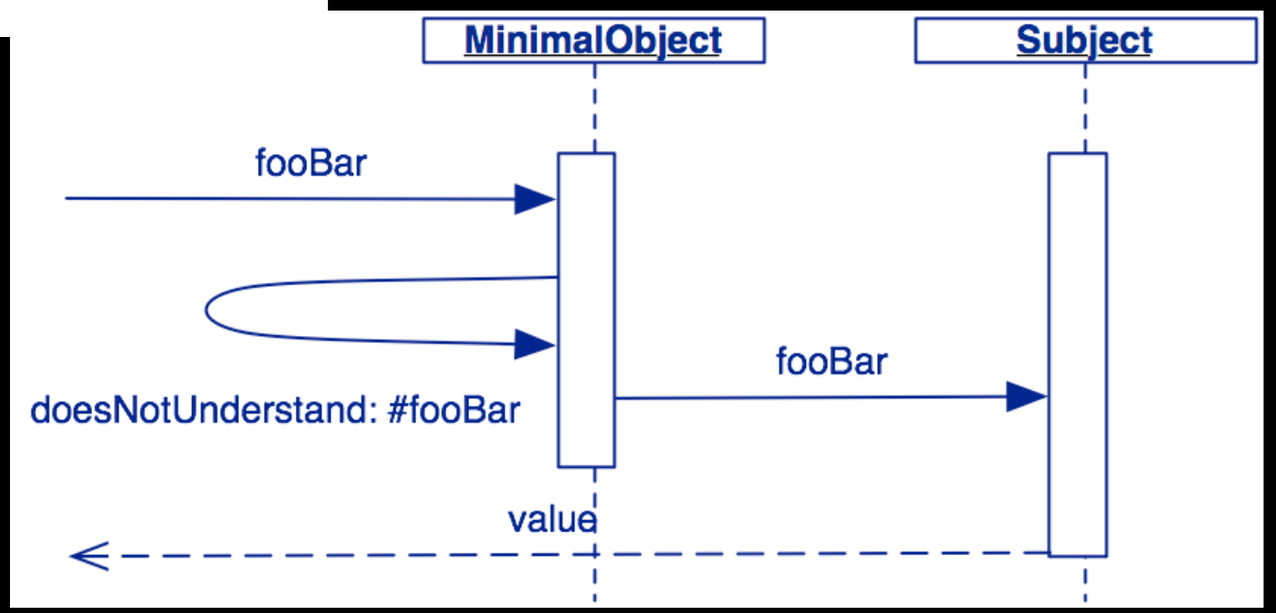
Sobreescriure `#doesNotUnderstand:`

Introduirem un objecte mínim
- embolica (*wraps*) un objecte normal
- no enten (quasi bé) res
- redefineix **`#doesNotUnderstand:`**
- superclase és **`nil`** o **`ProtoObject`**
- utilitza **`#become:`** per sustituir l'objecte a controlar

CAP: Reflexió en Smalltalk

L'objecte mínim en acció

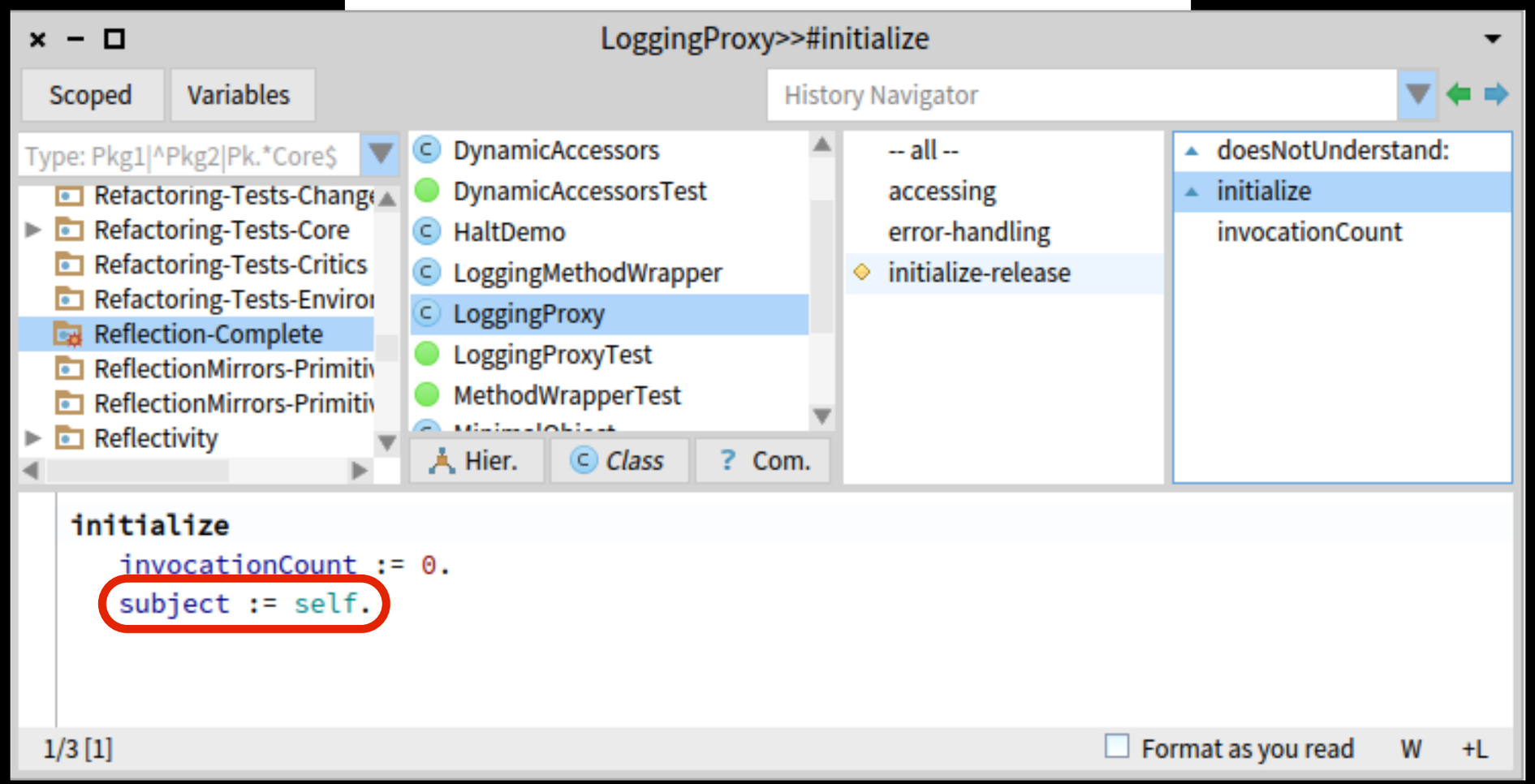




CAP: Reflexió en Smalltalk

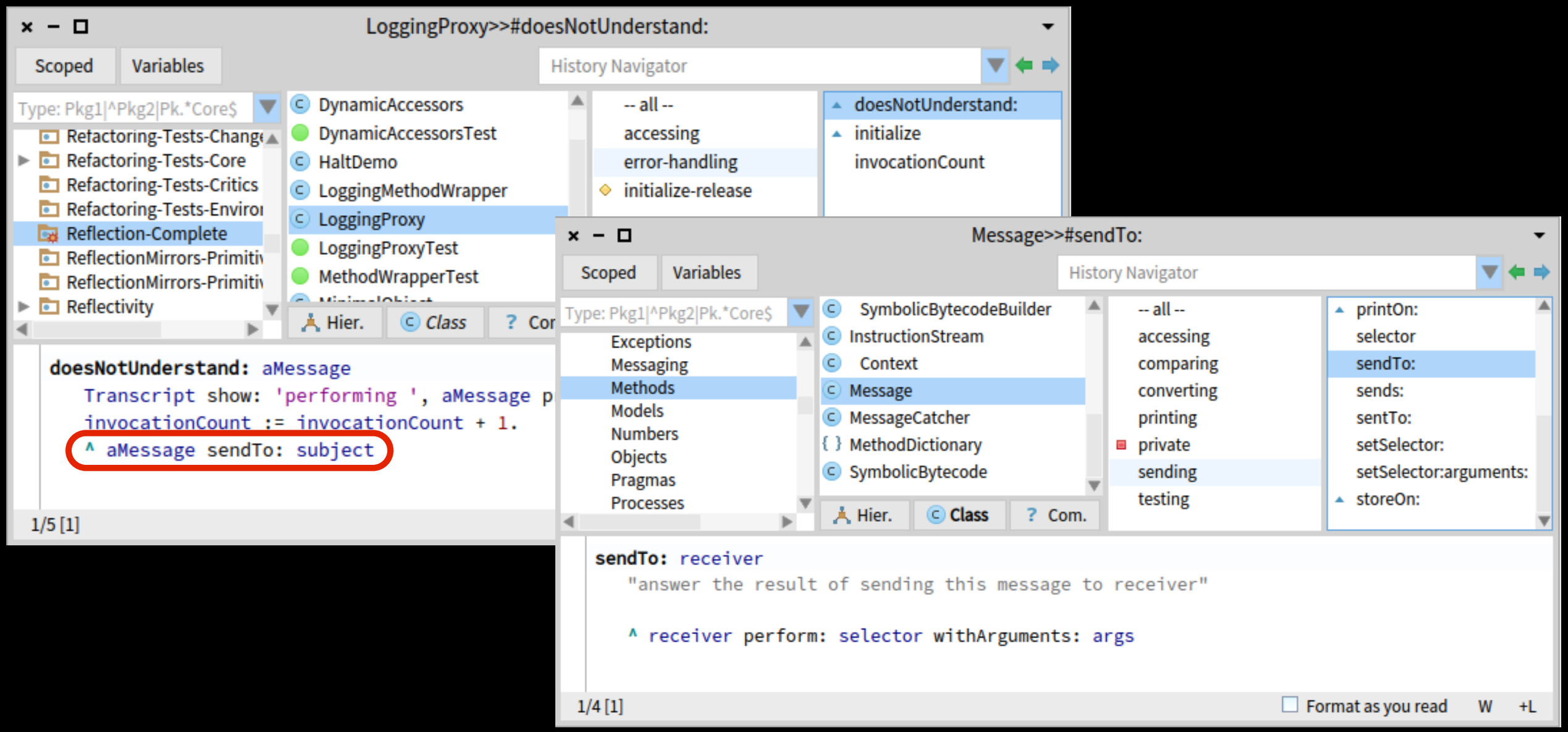
Logging d'enviaments de missatge utilitzant l'objecte mínim

```
ProtoObject subclass: #LoggingProxy
   instanceVariableNames: 'subject invocationCount'
   classVariableNames: ''
   package: 'Reflection-Complete'
```



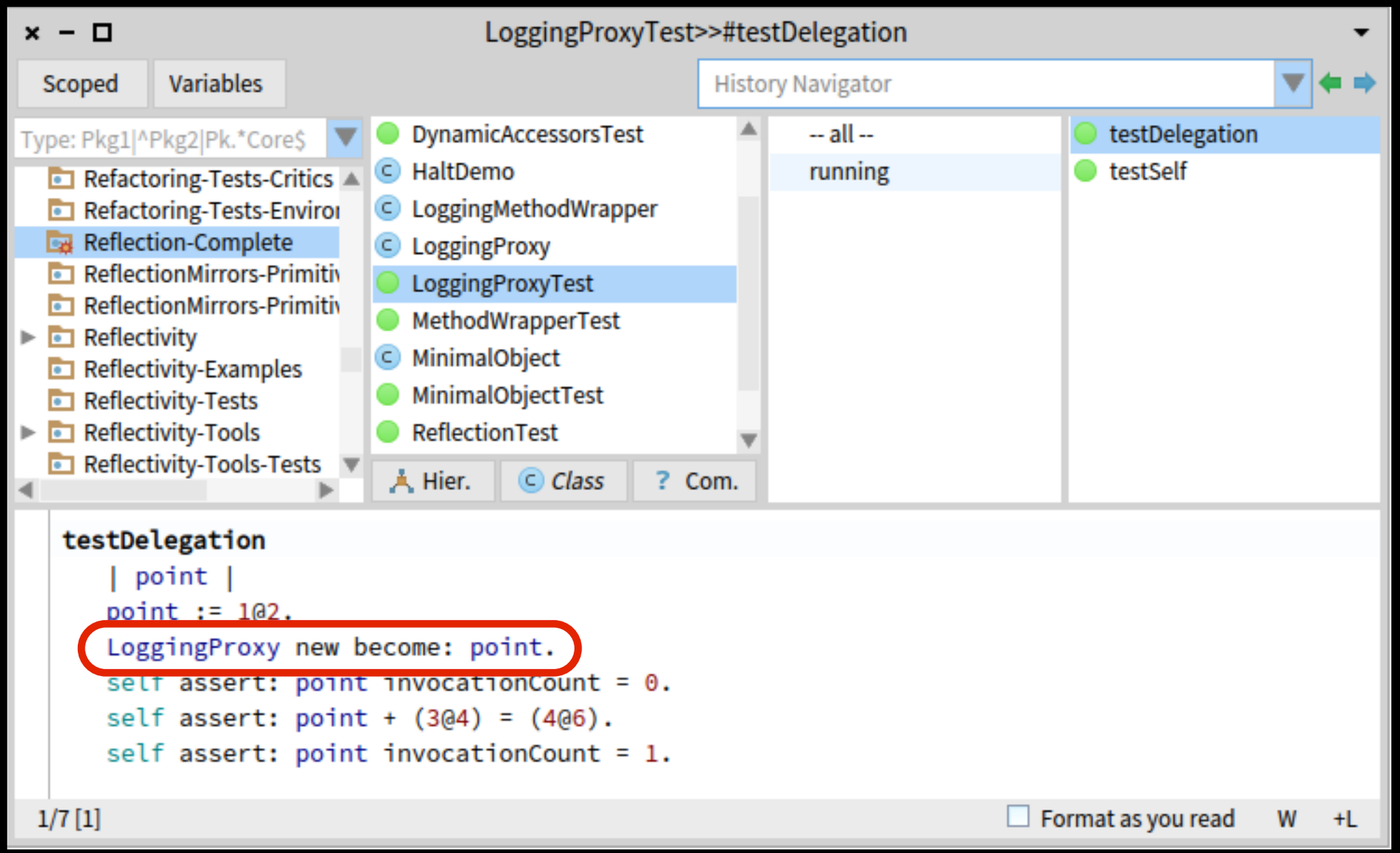
CAP: Reflexió en Smalltalk

Logging d'enviaments de missatge utilitzant l'objecte mínim



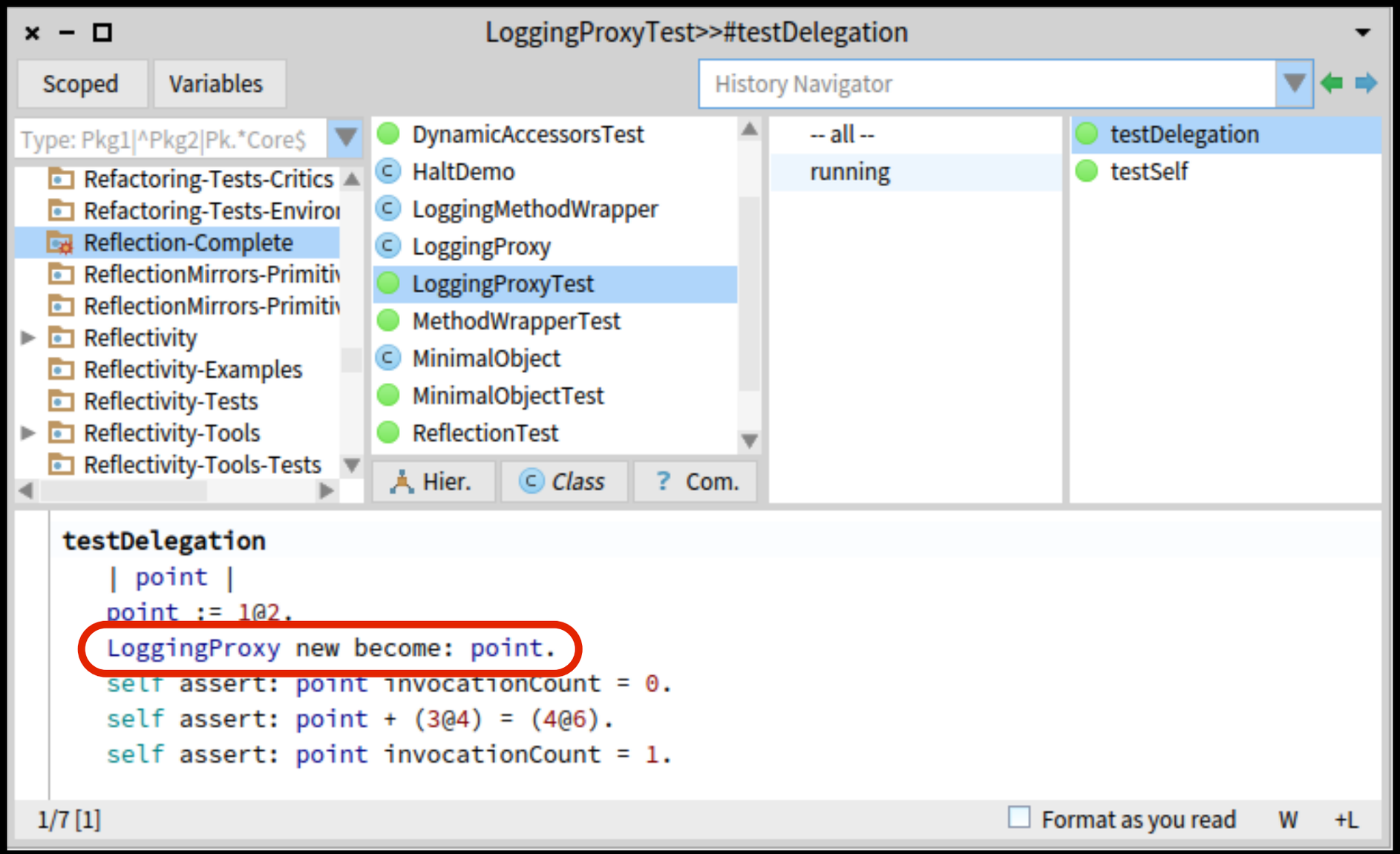
CAP: Reflexió en Smalltalk

Utilitzar `#become:` per instal.lar un *proxy*



CAP: Reflexió en Smalltalk

Com funciona això?



CAP: Reflexió en Smalltalk

Limitacions

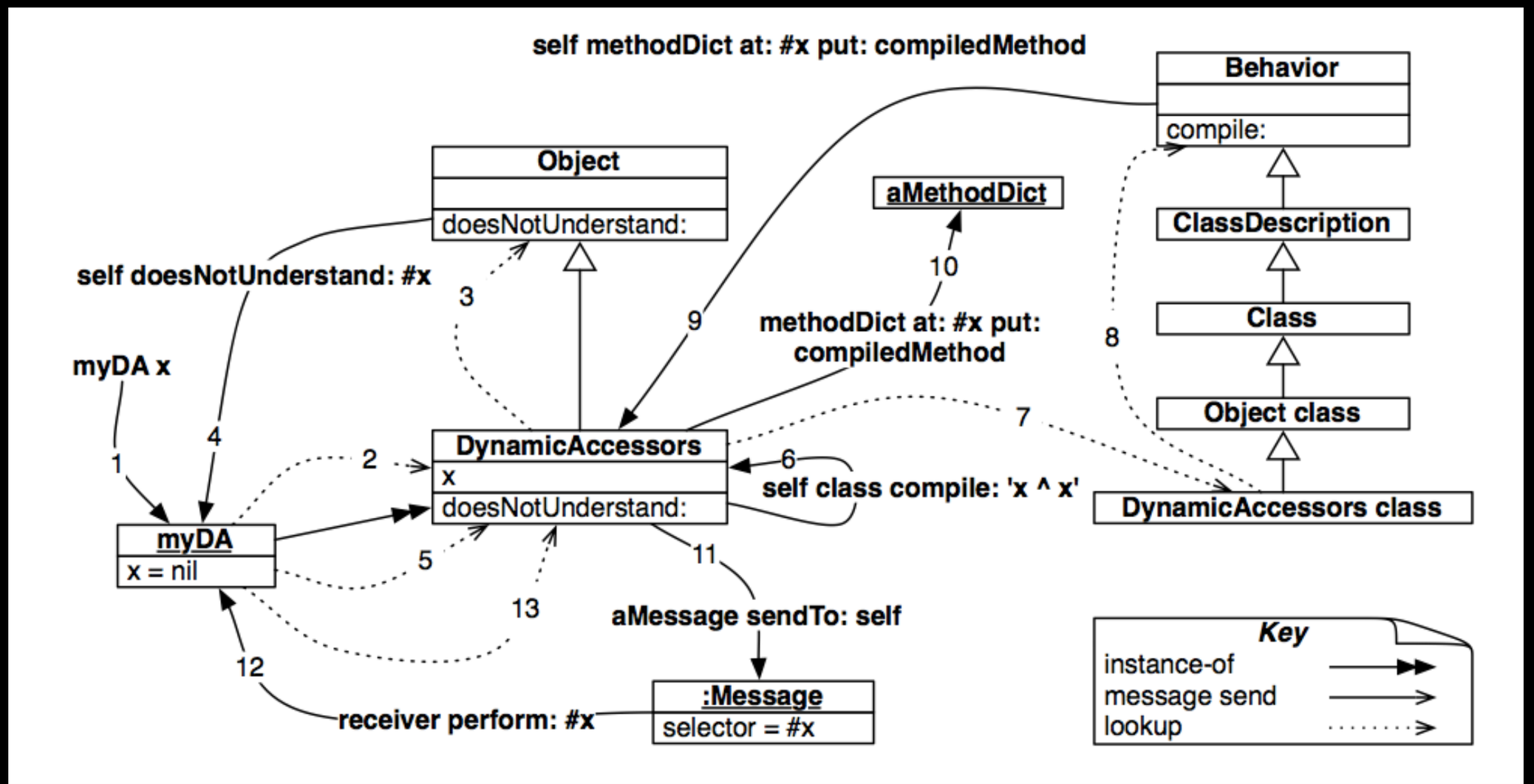
- El problema de **`self`**
  Missatges enviats per l'objecte a ell mateix no són interceptats

- El control de les classes és impossible
  No puc fer swap de classes

- Interpretació del protocol mínim
  Què fem amb els missatges que poden ser enviats tant a l'objecte embolicat com a l'objecte mínim?

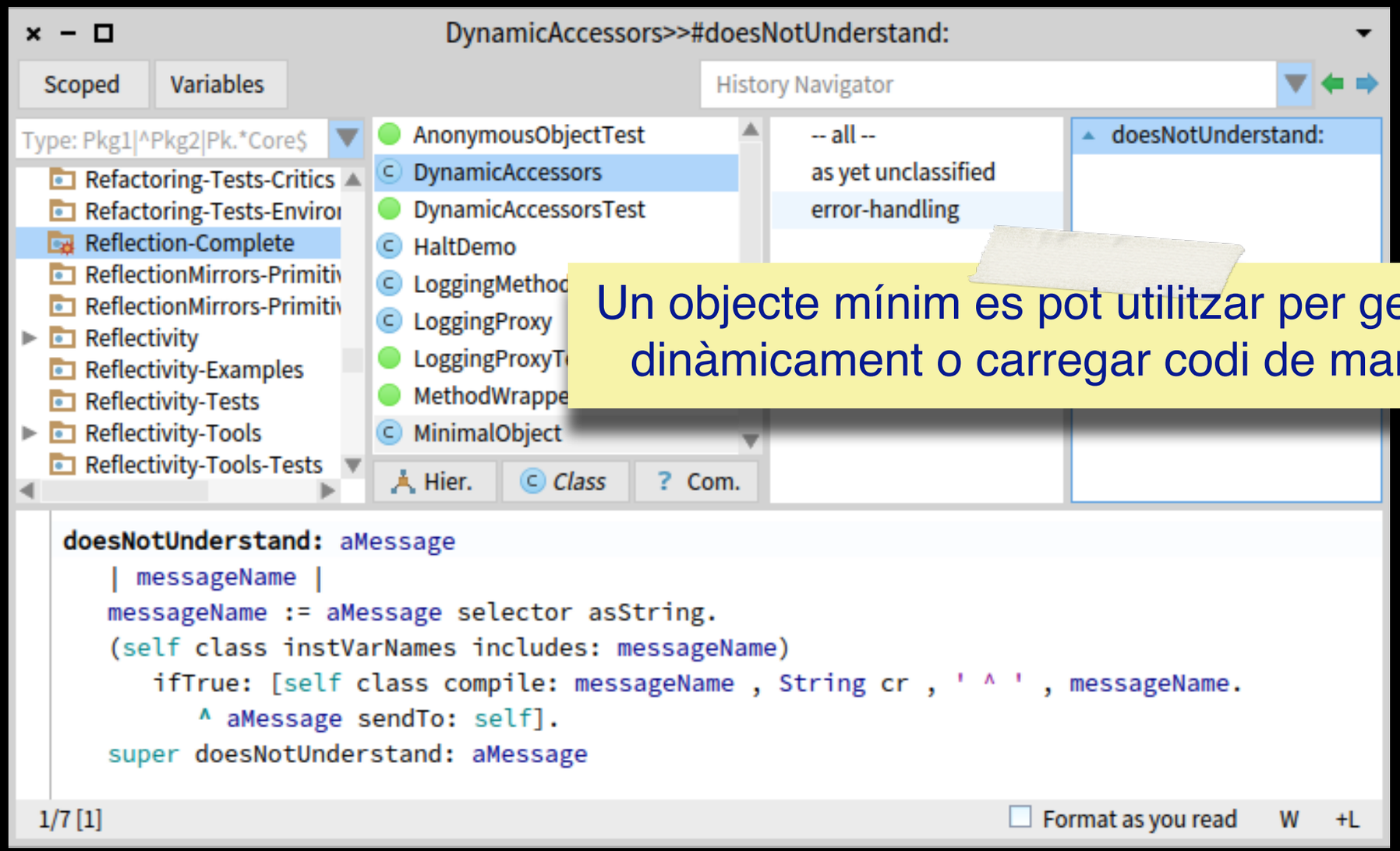
CAP: Reflexió en Smalltalk

Redefinir `#doesNotUnderstand:` per generar codi dinàmicament: Exemple: Els *getters* "*on demand*"



CAP: Reflexió en Smalltalk

Redefinir `#doesNotUnderstand:` per generar codi dinàmicament:
Exemple: Els *getters* "*on demand*"



CAP: Reflexió en Smalltalk

Introspecció:

- Inspeccionar objectes
- Consultar el codi
- Accedir els contexts d'execució

Intercessió:

- Sobreescriure `#doesNotUnderstand:`
- ***Classes Anònimes***
- Method Wrappers
- Continuacions

Part del material extret del treball de Marcus Denker i Stéphane Ducasse

CAP: Reflexió en Smalltalk

Control de missatges amb classes anònimes

Crear una classe *anònima*:

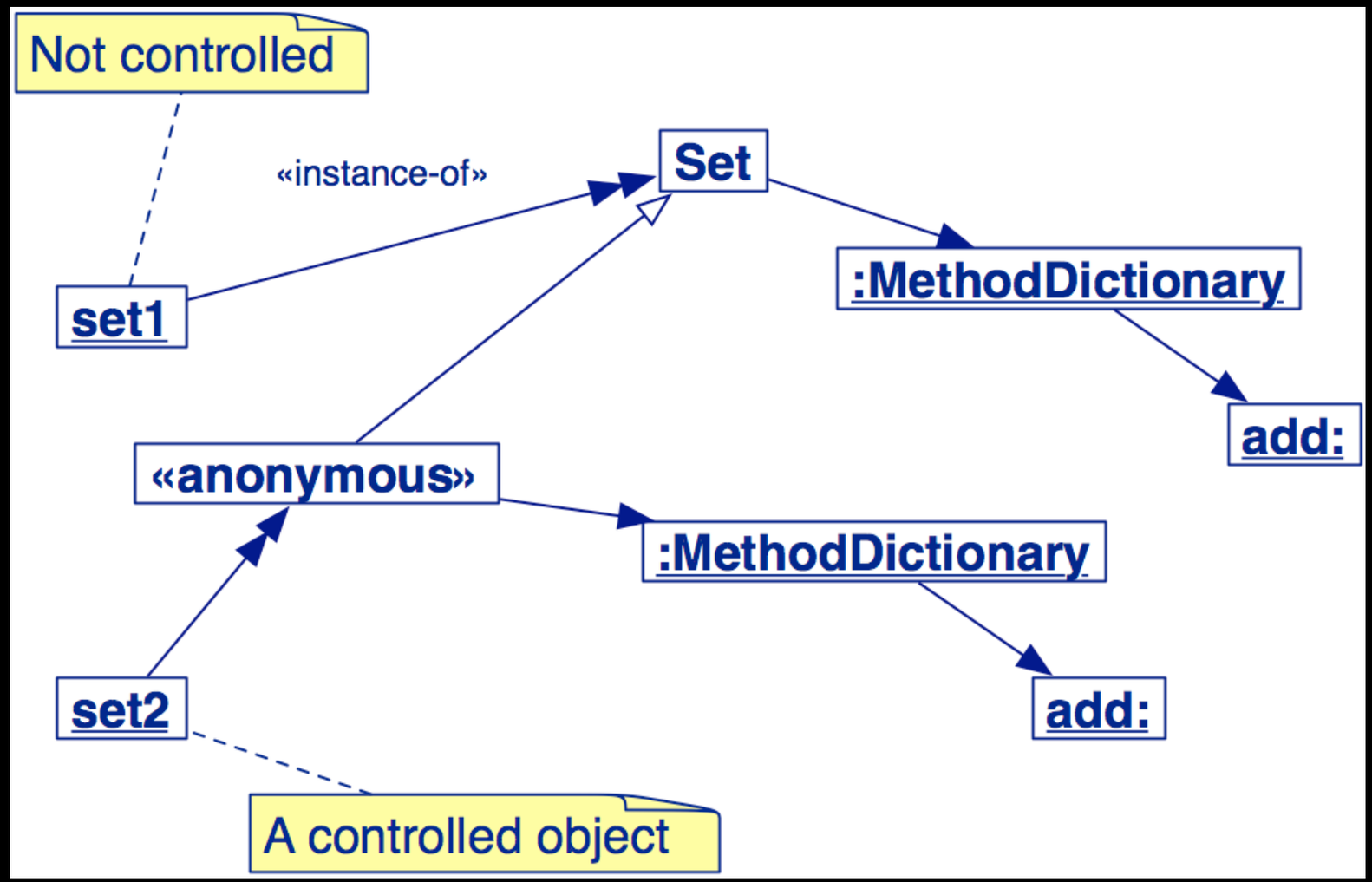
- Instància de **`Behavior`**

- Definir mètodes de control

- Interposar-la entre la instància i la classe

CAP: Reflexió en Smalltalk

Control sel.lectiu



CAP: Reflexió en Smalltalk

Classes anònimes a Pharo

Playground

Page

```
| set anonClass |
set := Set new.
anonClass := Behavior new.
anonClass superclass: Set;
   setFormat: Set format;
   methodDictionary: MethodDictionary new.

anonClass compile:
   'add: anObject
      Transcript show: ''adding '', anObject printString; cr.
      ^ super add: anObject'.

set add: 1.
set primitiveChangeClassTo: anonClass new.
set add: 2.
('set size = ' , (set size) asString) traceCr.
```

Transcript

```
adding 2
'set size = 2'
```

CAP: Reflexió en Smalltalk

Avaluació

- Control sel.lectiu

- El problema de **`self`** no hi és

- Eficient

- Transparent a l'usuari

CAP: Reflexió en Smalltalk

Introspecció:

- Inspeccionar objectes
- Consultar el codi
- Accedir els contexts d'execució

Intercessió:

- Sobreescriure `#doesNotUnderstand:`
- Classes Anònimes
- ***Method Wrappers***
- Continuacions

Part del material extret del treball de Marcus Denker i Stéphane Ducasse

CAP: Reflexió en Smalltalk

Substitució de mètodes

Primera aproximació:

- Afegir mètodes amb *mangled names*
  L'usuari, però, els pot veure.

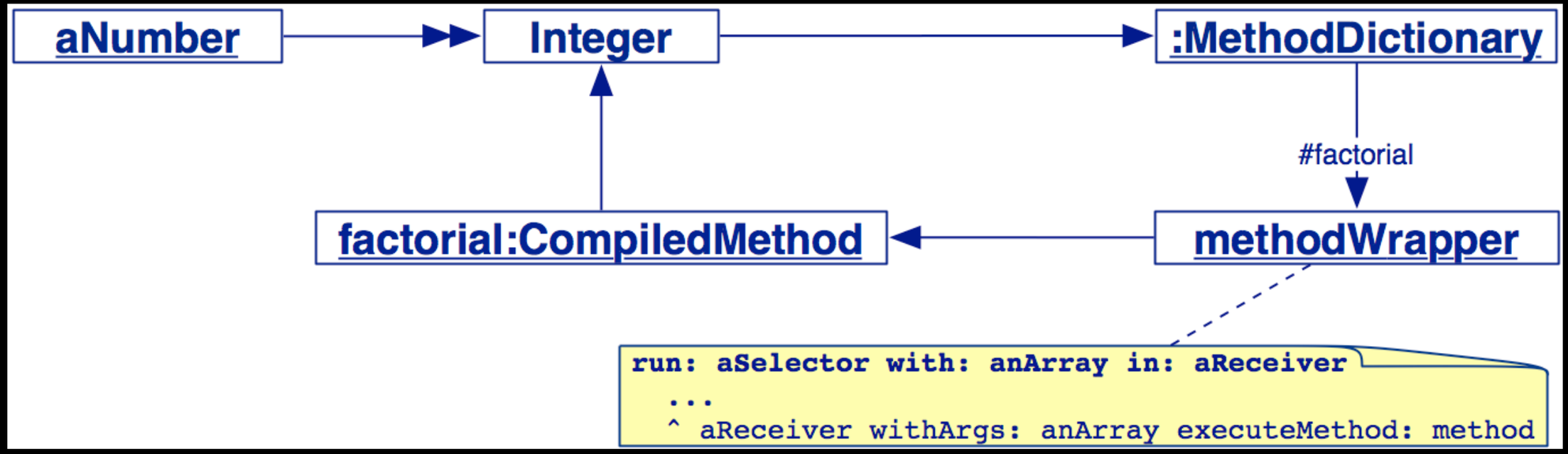
Segona aproximació:

- Embolicar els mètodes sense afegir res a l'interficie
  Substitueix el mètode per un objecte que implementi `#run:with:in:`

CAP: Reflexió en Smalltalk

Embolicar mètodes per fer accions abans i després

Un methodWrapper substitueix l'original **`CompiledMethod`** *en el diccionari de mètodes d'una classe i l'embolica realitzant algunes accions "before" i "after", per exemple*



CAP: Reflexió en Smalltalk

LoggingMethodWrapper

```
LoggingMethodWrapper>>initializeOn: aCompiledMethod
  method := aCompiledMethod.
  reference := aCompiledMethod methodReference.
  invocationCount := 0
```

```
LoggingMethodWrapper>>install
  reference actualClass methodDictionary
    at: reference selector
    put: self
```

uninstall és similar …

```
LoggingMethodWrapper>>run: aSelector with: anArray in: aReceiver
  invocationCount := invocationCount + 1.
  ^ aReceiver withArgs: anArray executeMethod: method
```

CAP: Reflexió en Smalltalk

Instal.lar un `LoggingMethodWrapper`

```
logger := LoggingMethodWrapper on: Integer>>#factorial.

logger invocationCount. 0
5 factorial.
logger invocationCount. 0

logger install.
[ 5 factorial ] ensure: [logger uninstall].
logger invocationCount. 6

10 factorial.
logger invocationCount. 6
```

CAP: Reflexió en Smalltalk

Avaluació

- Basat en classes
  totes les instàncies queden controlades
  - Només s'intercepten els missatges coneguts
  - Un sol mètode pot ser controlat
  - No cal compilar per instal.lar

CAP: Reflexió en Smalltalk

Introspecció:

- Inspeccionar objectes
- Consultar el codi
- Accedir els contexts d'execució

Intercessió:

- Sobreescriure **`#doesNotUnderstand:`**
- Classes Anònimes
- Method Wrappers
- ***Continuacions***

Part del material extret del treball de Marcus Denker i Stéphane Ducasse

CAP: Reflexió en Smalltalk

Continuation: *abstract representation of the control state of a computer program. A continuation reifies the program control state, i.e. the continuation is a data structure that represents the computational process at a given point in the process's execution; the created data structure can be accessed by the programming language, instead of being hidden in the runtime environment.*

http://en.wikipedia.org/wiki/Continuation

CAP: Reflexió en Smalltalk

Per a aquesta part del curs teniu un document al Racó: **`Continuacions.pdf`**

En aquest document trobareu explicacions molt més detallades del que segueix. A més, recomano ***molt*** que us llegiu el capítol 14, *Blocks: A Detailed Analysis*, del llibre ***Deep into Pharo***

(**`http://files.pharo.org/books-pdfs/deep-into-pharo/2013-DeepIntoPharo-EN.pdf`**)

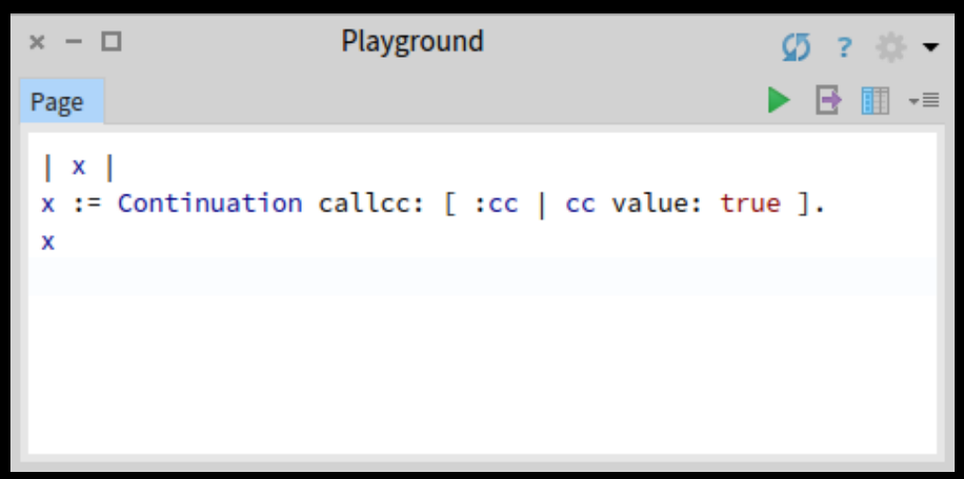
En el capítol esmentat es fan referències a les classes **`MethodContext`** i **`ContextPart`**. **`MethodContext`** era la classe de la que eren instàncies els *stack frames*. **`MethodContext`** era subclasse de **`ContextPart`**, i aquesta era subclasse de **`InstructionStream`**. En Pharo 6.1 la classe **`Context`** és una fusió de **`MethodContext`** i **`ContextPart`**, sustituint a aquestes dues classes. Aleshores, és **`Context`** la que és subclasse d'**`InstructionStream`**.

Tingueu-ho en compte en estudiar el capítol *Blocks: A Detailed Analysis*

CAP: Reflexió en Smalltalk

Amb aquesta infraestructura ja podem guardar i recuperar qualsevol moment de l'execució d'un programa (d'un *thread* en realitat, però no entrarem en aquests detalls).

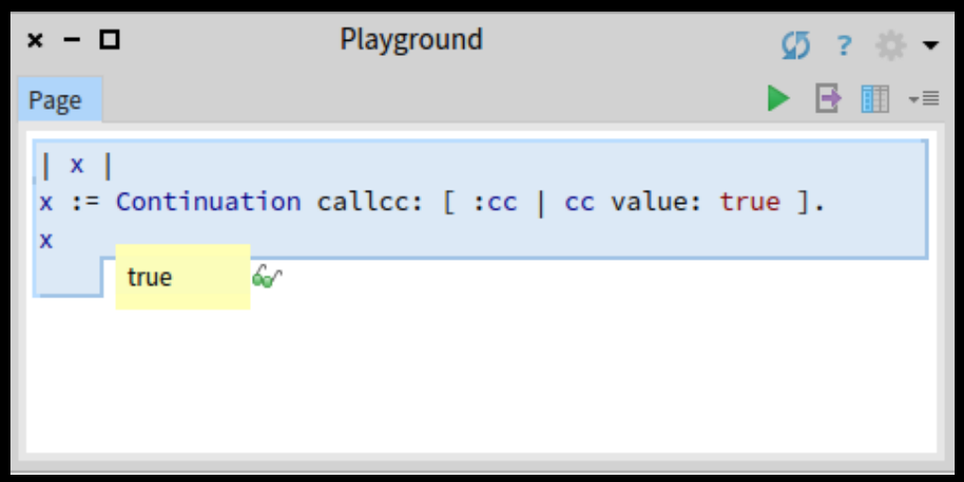
Exemples (*senzills*): Quin és el resultat d'avaluar aquests codis?



CAP: Reflexió en Smalltalk

Amb aquesta infraestructura ja podem guardar i recuperar qualsevol moment de l'execució d'un programa (d'un *thread* en realitat, però no entrarem en aquests detalls).

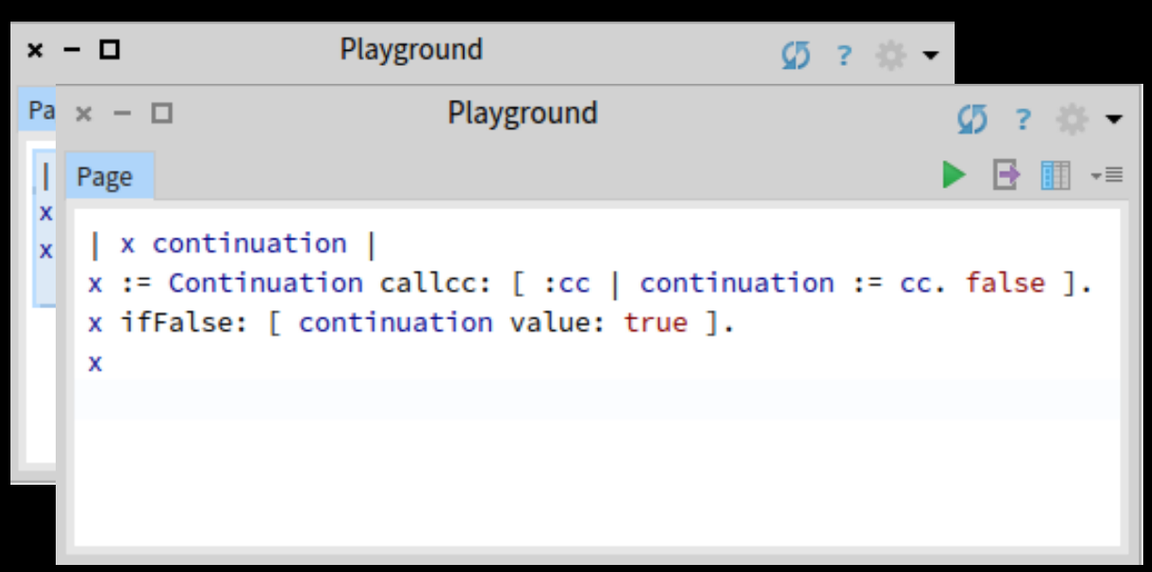
Exemples (*senzills*): Quin és el resultat d'avaluar aquests codis?



CAP: Reflexió en Smalltalk

Amb aquesta infraestructura ja podem guardar i recuperar qualsevol moment de l'execució d'un programa (d'un *thread* en realitat, però no entrarem en aquests detalls).

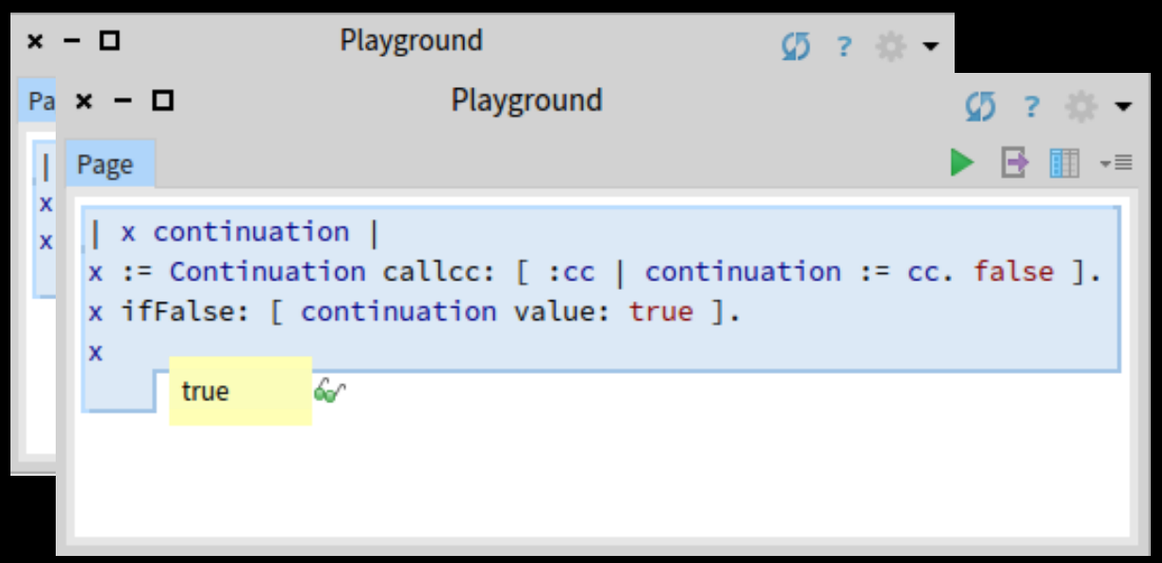
Exemples (*senzills*): Quin és el resultat d'avaluar aquests codis?



CAP: Reflexió en Smalltalk

Amb aquesta infraestructura ja podem guardar i recuperar qualsevol moment de l'execució d'un programa (d'un *thread* en realitat, però no entrarem en aquests detalls).

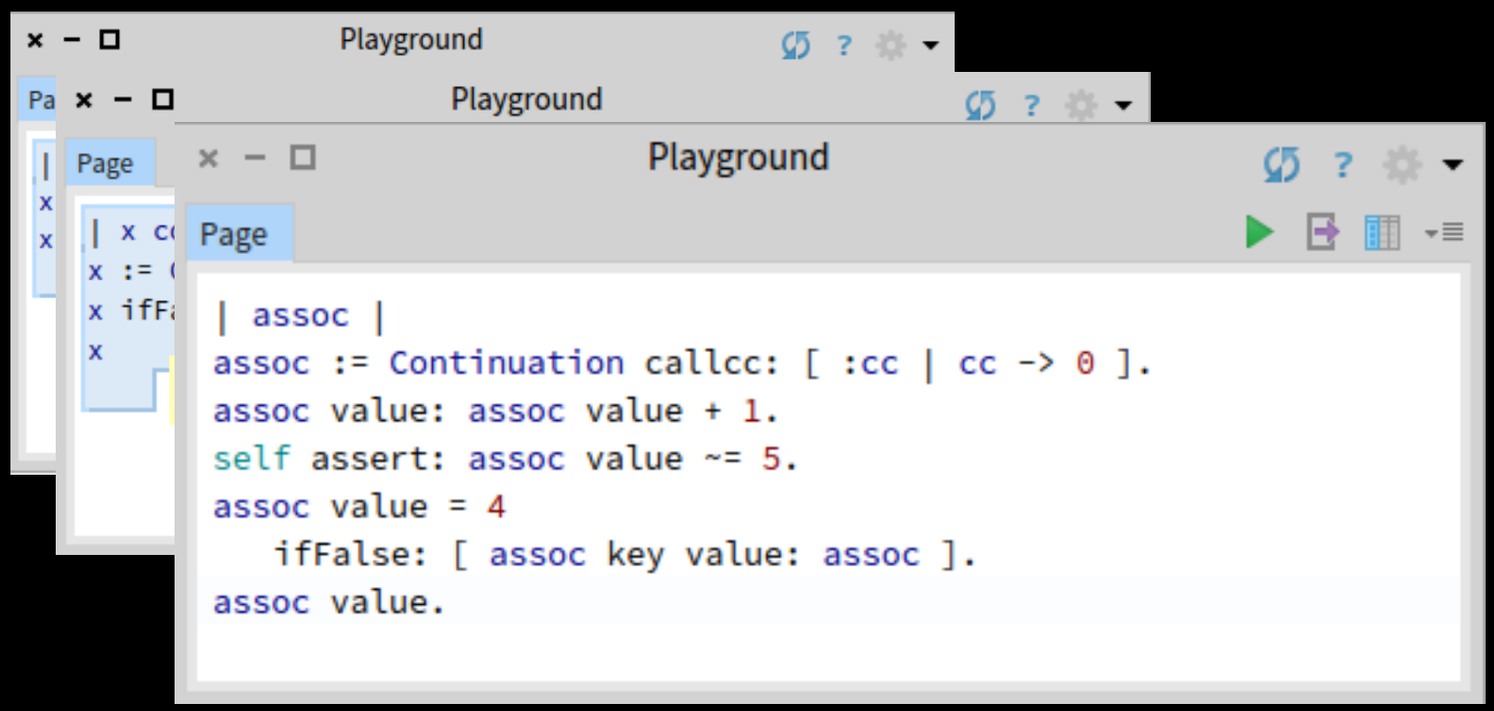
Exemples (*senzills*): Quin és el resultat d'avaluar aquests codis?



CAP: Reflexió en Smalltalk

Amb aquesta infraestructura ja podem guardar i recuperar qualsevol moment de l'execució d'un programa (d'un *thread* en realitat, però no entrarem en aquests detalls).

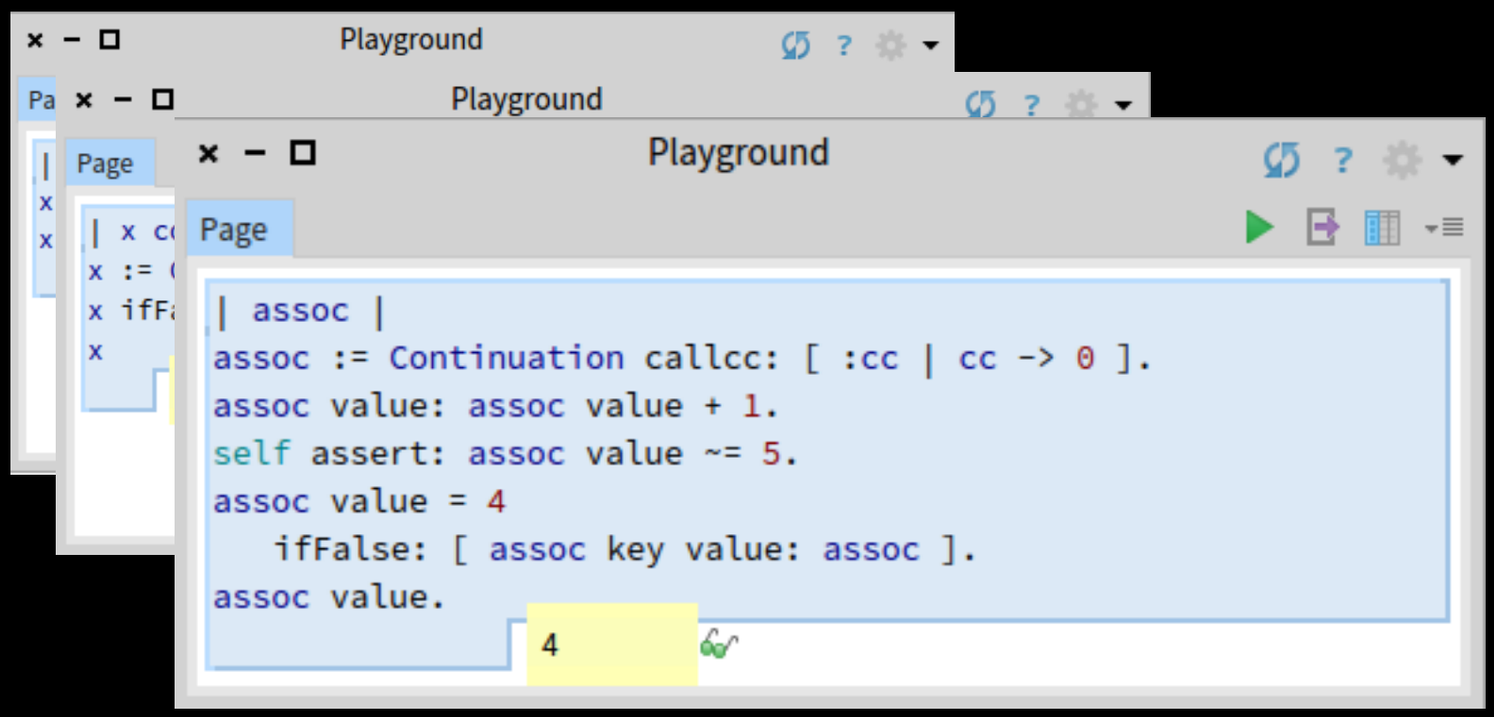
Exemples (*senzills*): Quin és el resultat d'avaluar aquests codis?



CAP: Reflexió en Smalltalk

Amb aquesta infraestructura ja podem guardar i recuperar qualsevol moment de l'execució d'un programa (d'un *thread* en realitat, però no entrarem en aquests detalls).

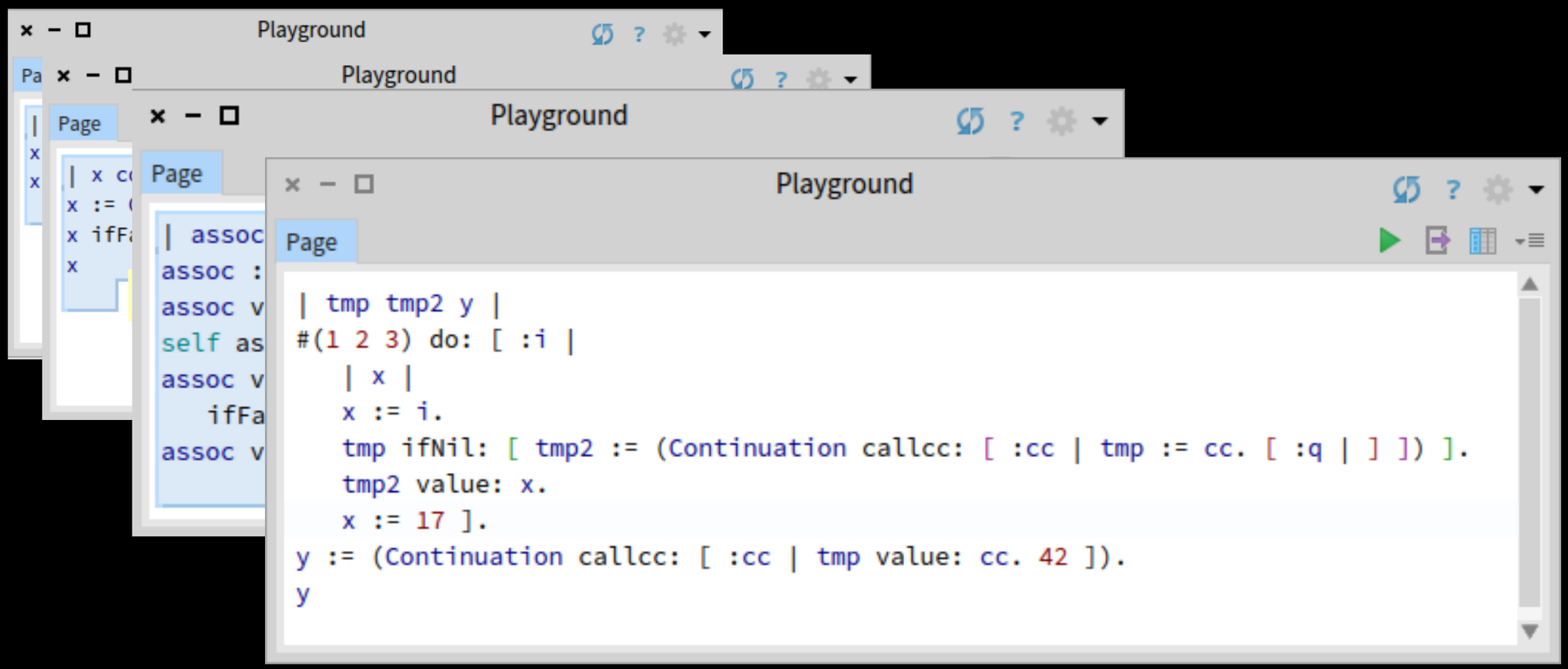
Exemples (*senzills*): Quin és el resultat d'avaluar aquests codis?



CAP: Reflexió en Smalltalk

Amb aquesta infraestructura ja podem guardar i recuperar qualsevol moment de l'execució d'un programa (d'un *thread* en realitat, però no entrarem en aquests detalls).

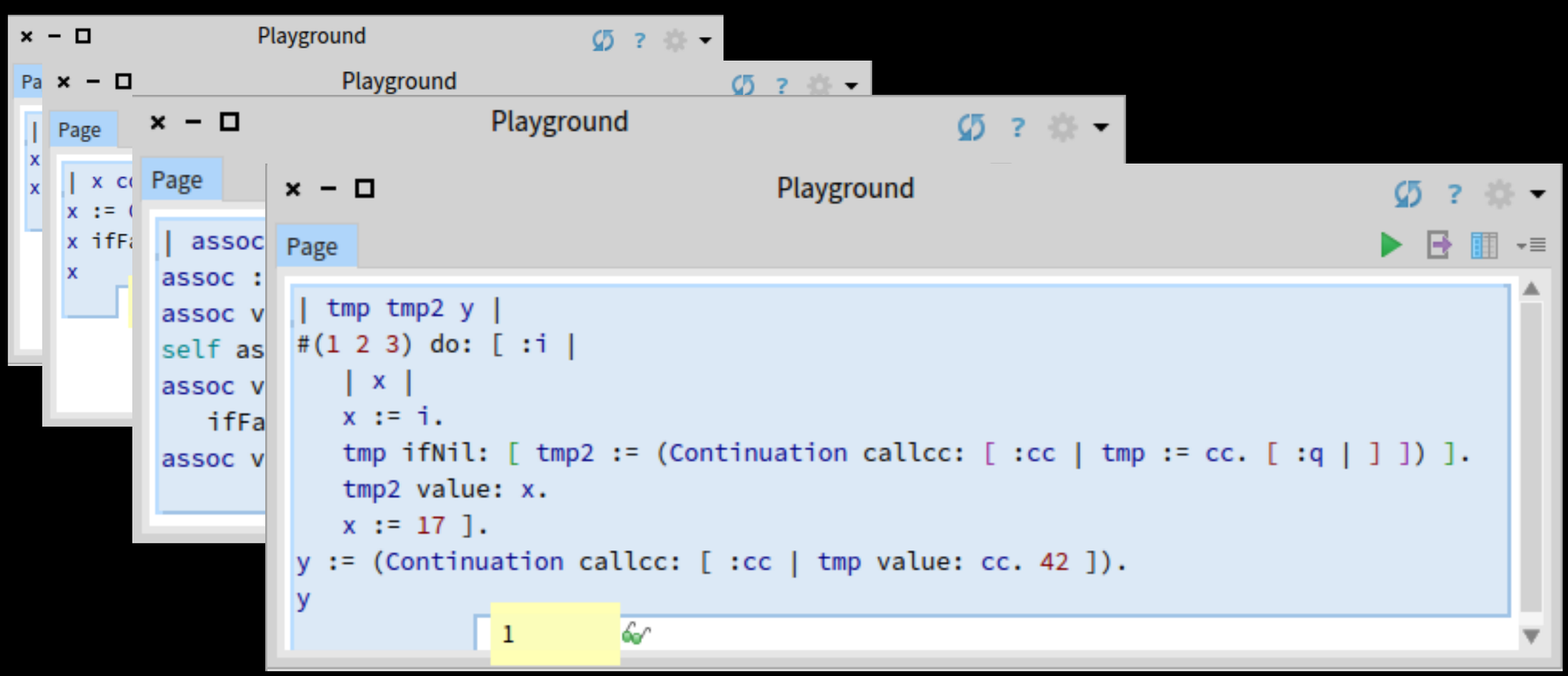
Exemples (*senzills*): Quin és el resultat d'avaluar aquests codis?



CAP: Reflexió en Smalltalk

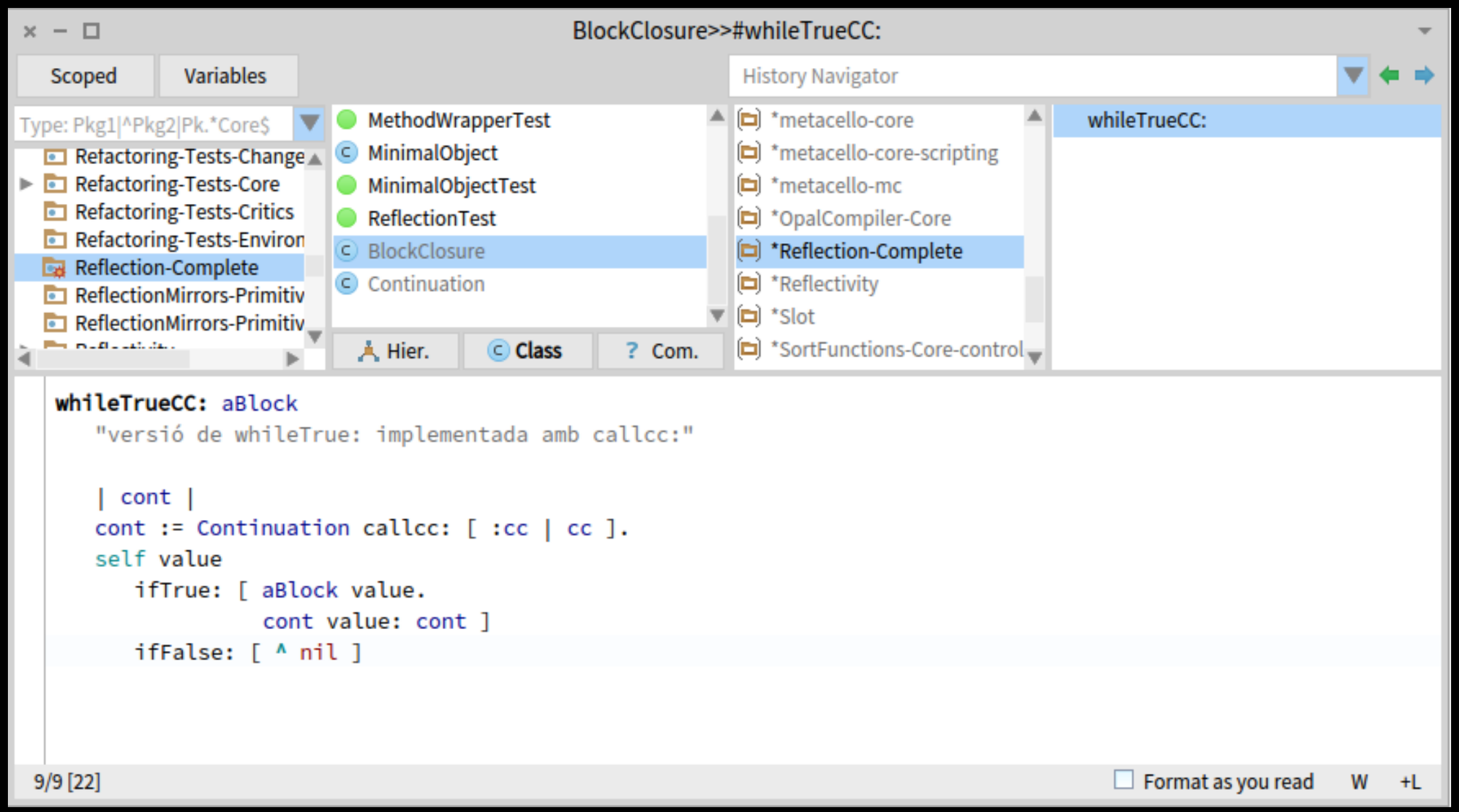
Amb aquesta infraestructura ja podem guardar i recuperar qualsevol moment de l'execució d'un programa (d'un *thread* en realitat, però no entrarem en aquests detalls).

Exemples (*senzills*): Quin és el resultat d'avaluar aquests codis?



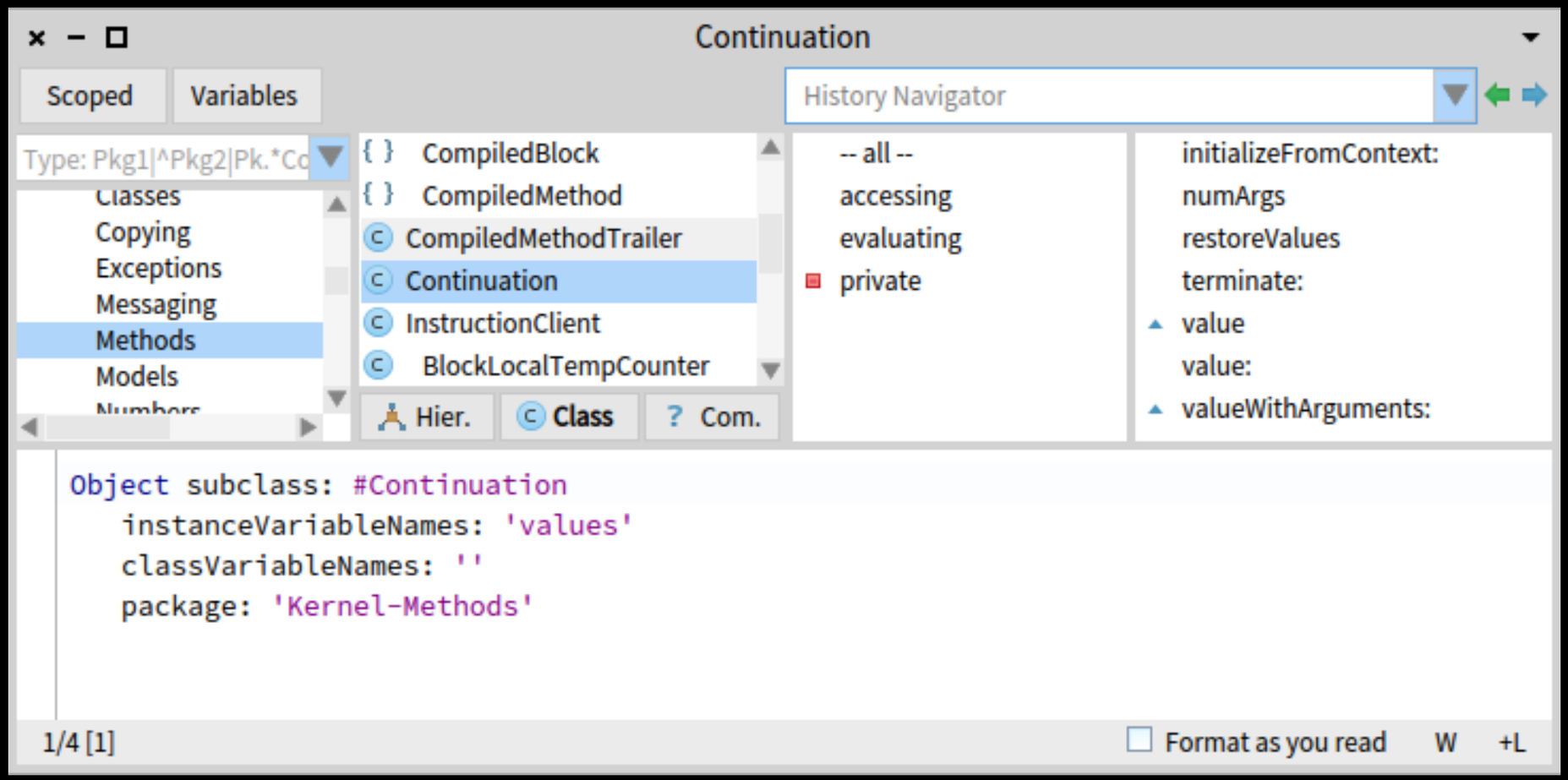
CAP: Reflexió en Smalltalk

Exemple: Implementar `BlockClosure>>#whileTrueCC:`, l'equivalent del `#whileTrue:`, sense cap construcció iterativa, només fent servir `#callcc:`



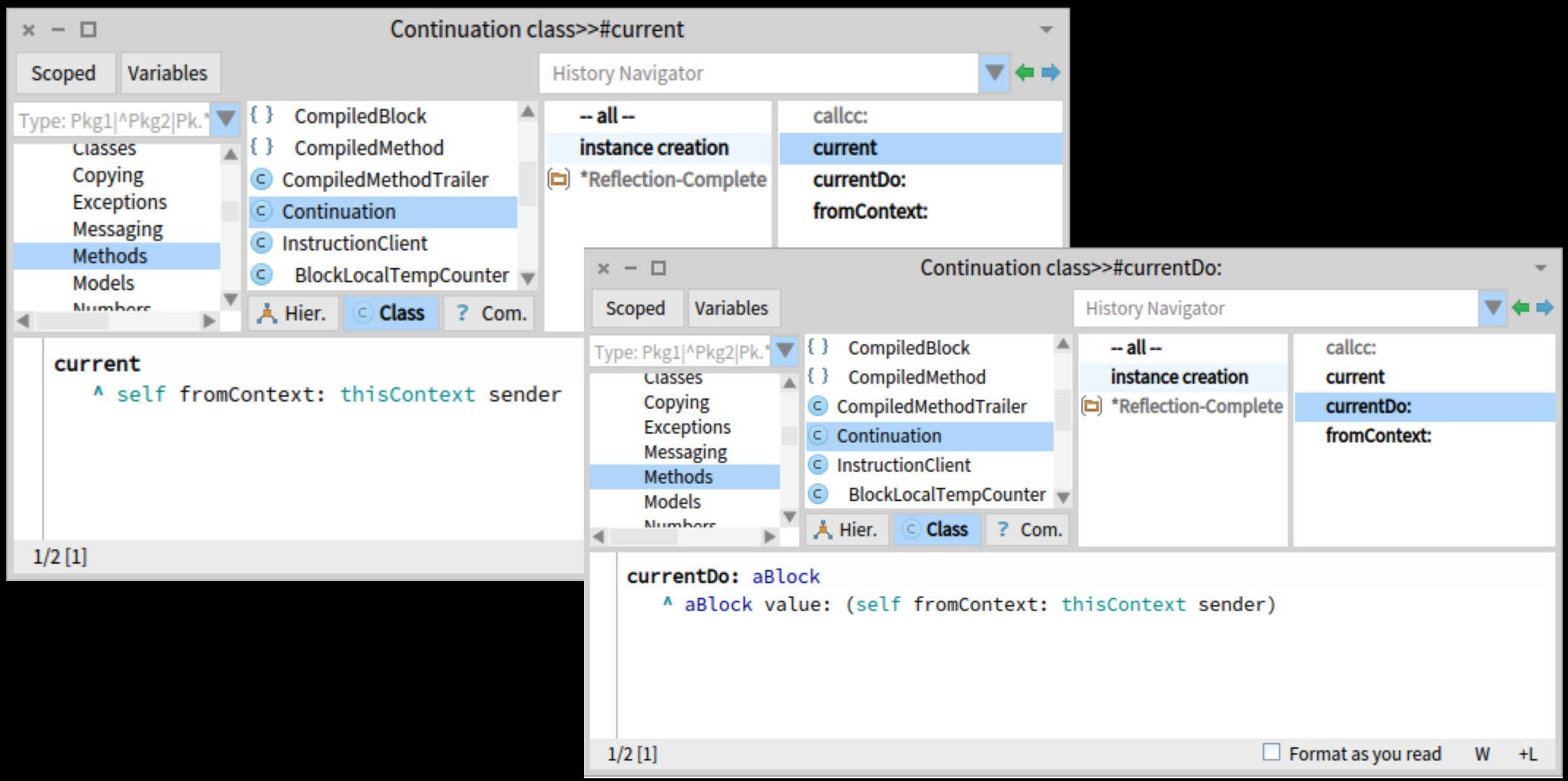
CAP: Reflexió en Smalltalk

A Pharo 6.1 existeix la classe **`Continuation`**. Serveix per guardar el context, és a dir, la pila d'execució



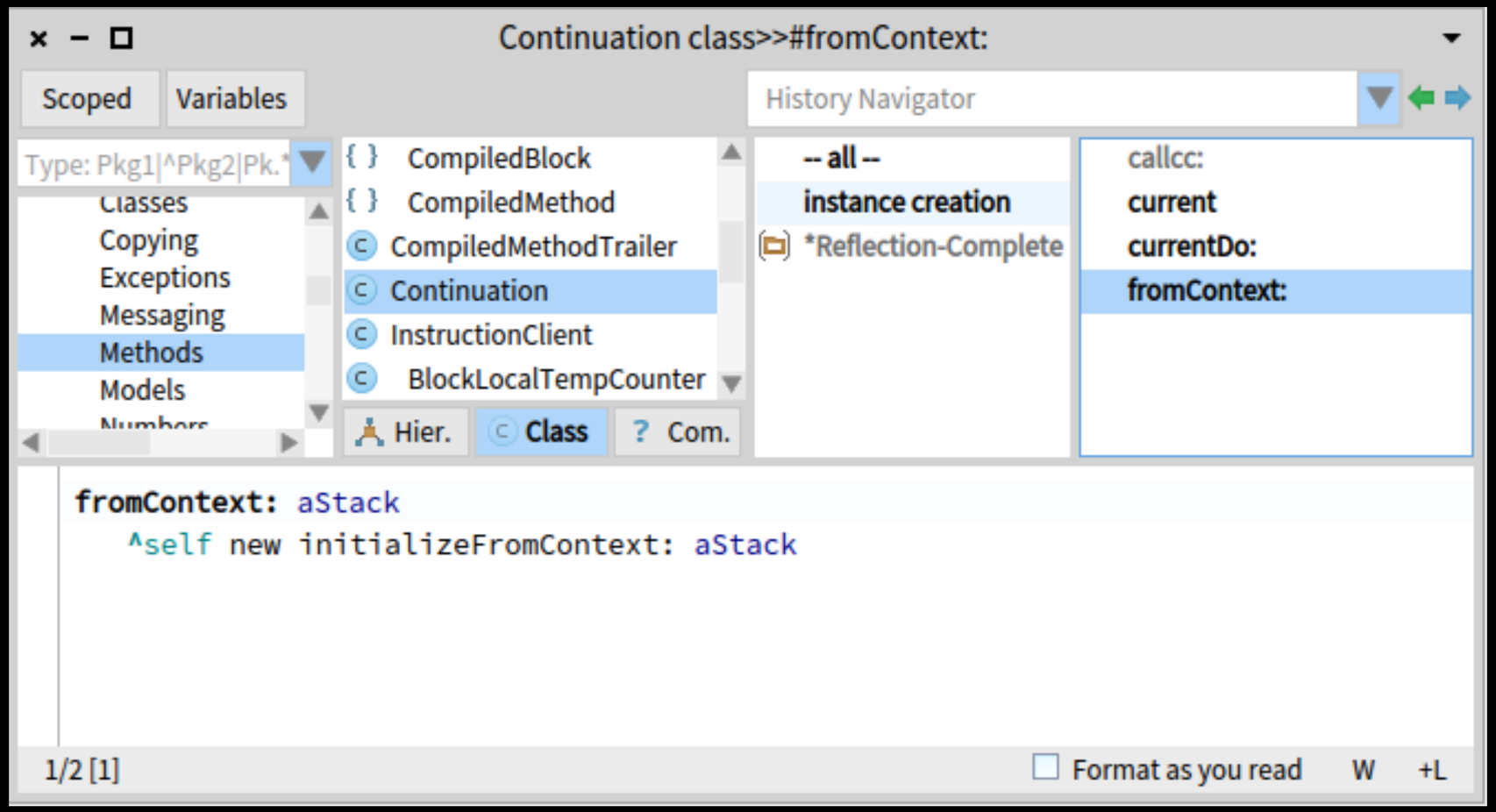
CAP: Reflexió en Smalltalk

S'instancia amb **`Continuation class>>#current`**
o amb **`Continuation class>>#currentDo:`**



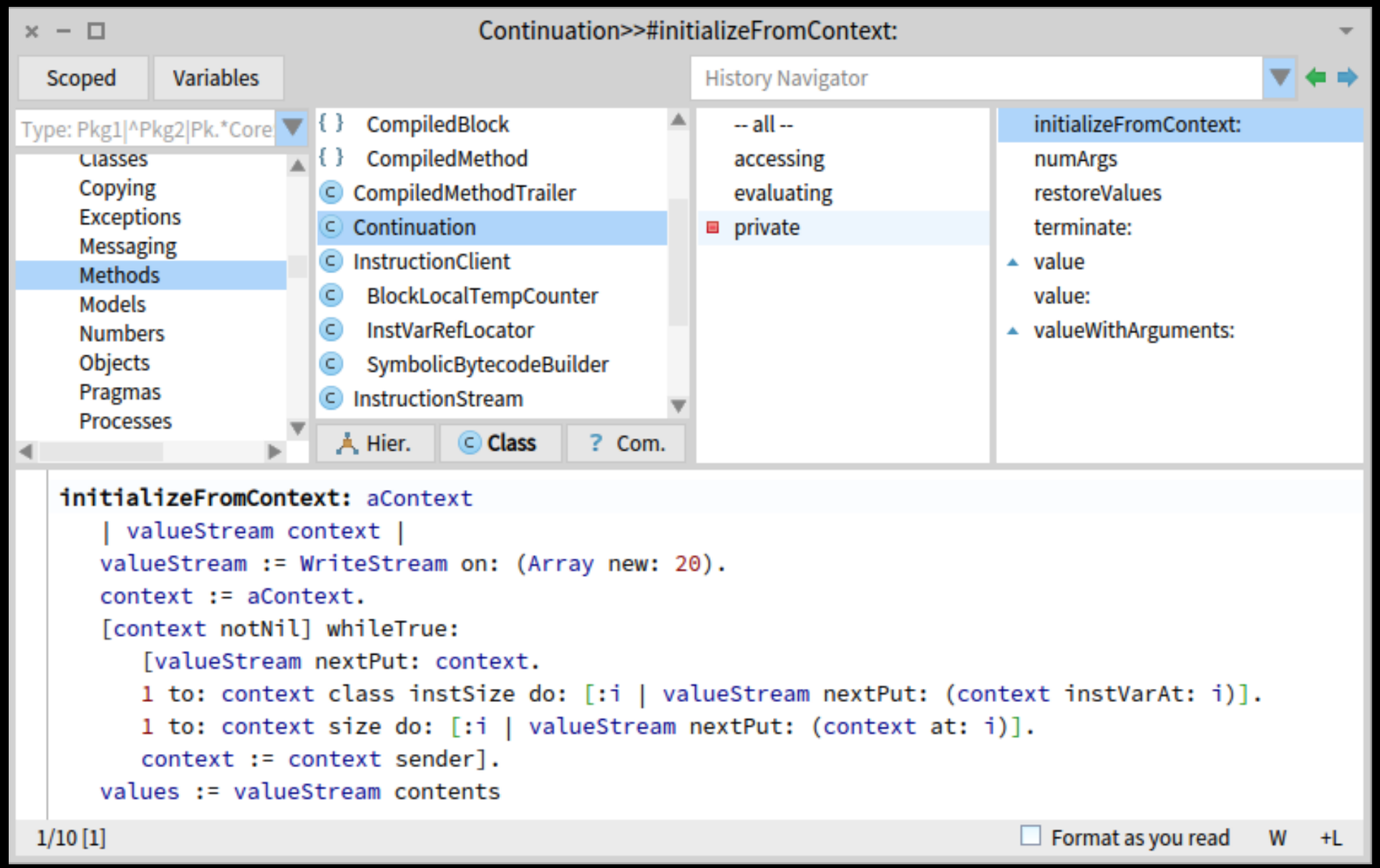
CAP: Reflexió en Smalltalk

Així doncs, és clar que el mètode que crea la instancia és **`Continuation class>>#fromContext:`**



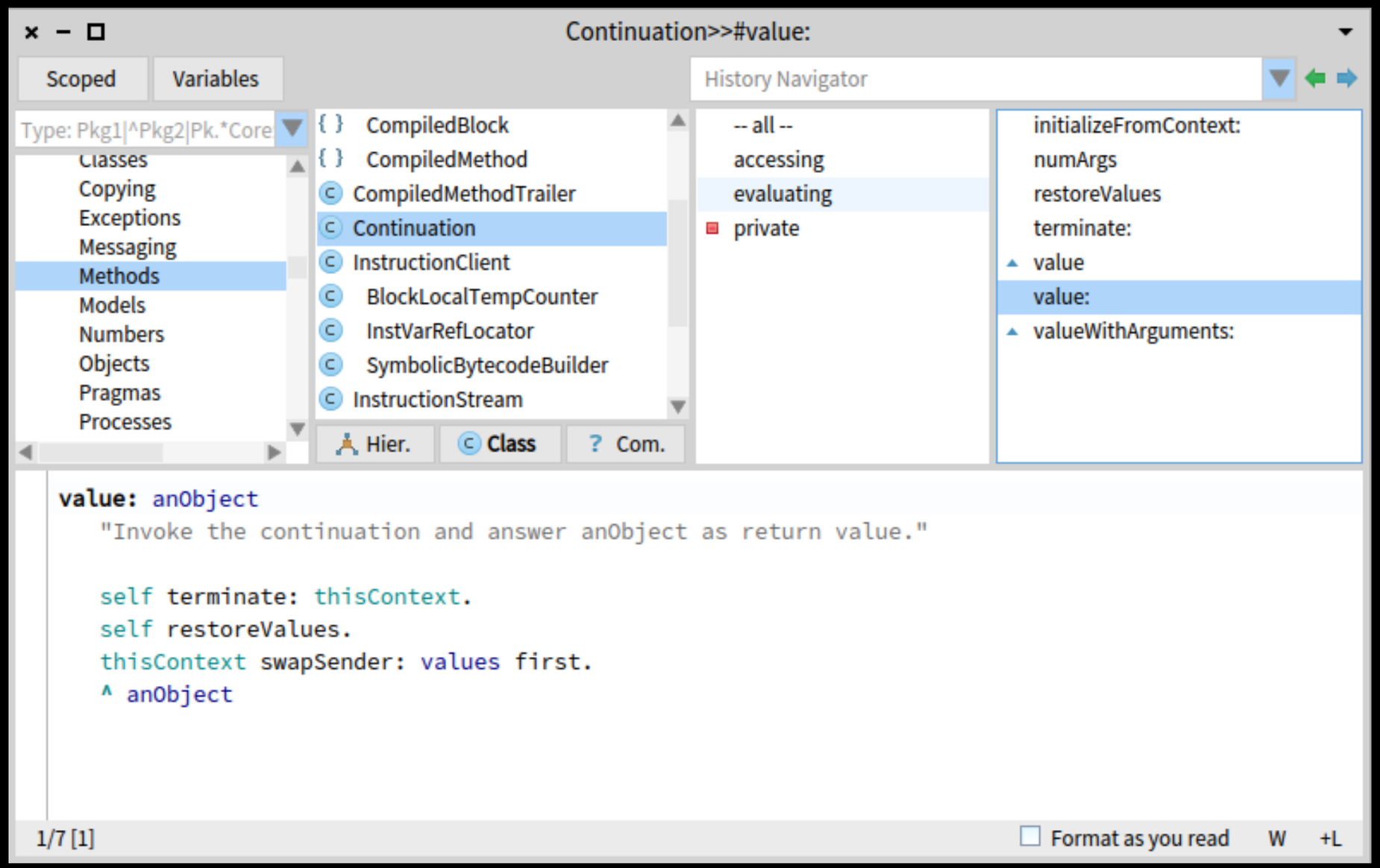
CAP: Reflexió en Smalltalk

Cal guardar tota la informació que tenim a la pila d'execució, però considerant que Pharo 6.1 internament recicla les instàncies de **`Context`**, el que cal guardar va més enllà dels elements enllaçats des del context argument **`aContext`**



CAP: Reflexió en Smalltalk

Un cop la pila d'execucio està guardada a la variable d'instància **`values`** de la instància de **`Continuation`**, cal poder reinstaurar-la quan desitjem fer-ho. Per a això tenim **`Continuation>>#value:`**



CAP: Reflexió en Smalltalk

Continuation>>#terminate:

Scoped | Variables

History Navigator

Type: Pkg1|^Pkg2|Pk.*Core

Classes
Copying
Exceptions
Messaging
Methods
Models
Numbers
Objects
Pragmas

CompiledBlock
CompiledMethod
CompiledMethodTrailer
Continuation
InstructionClient
BlockLocalTempCounter
InstVarRefLocator
SymbolicBytecodeBuilder

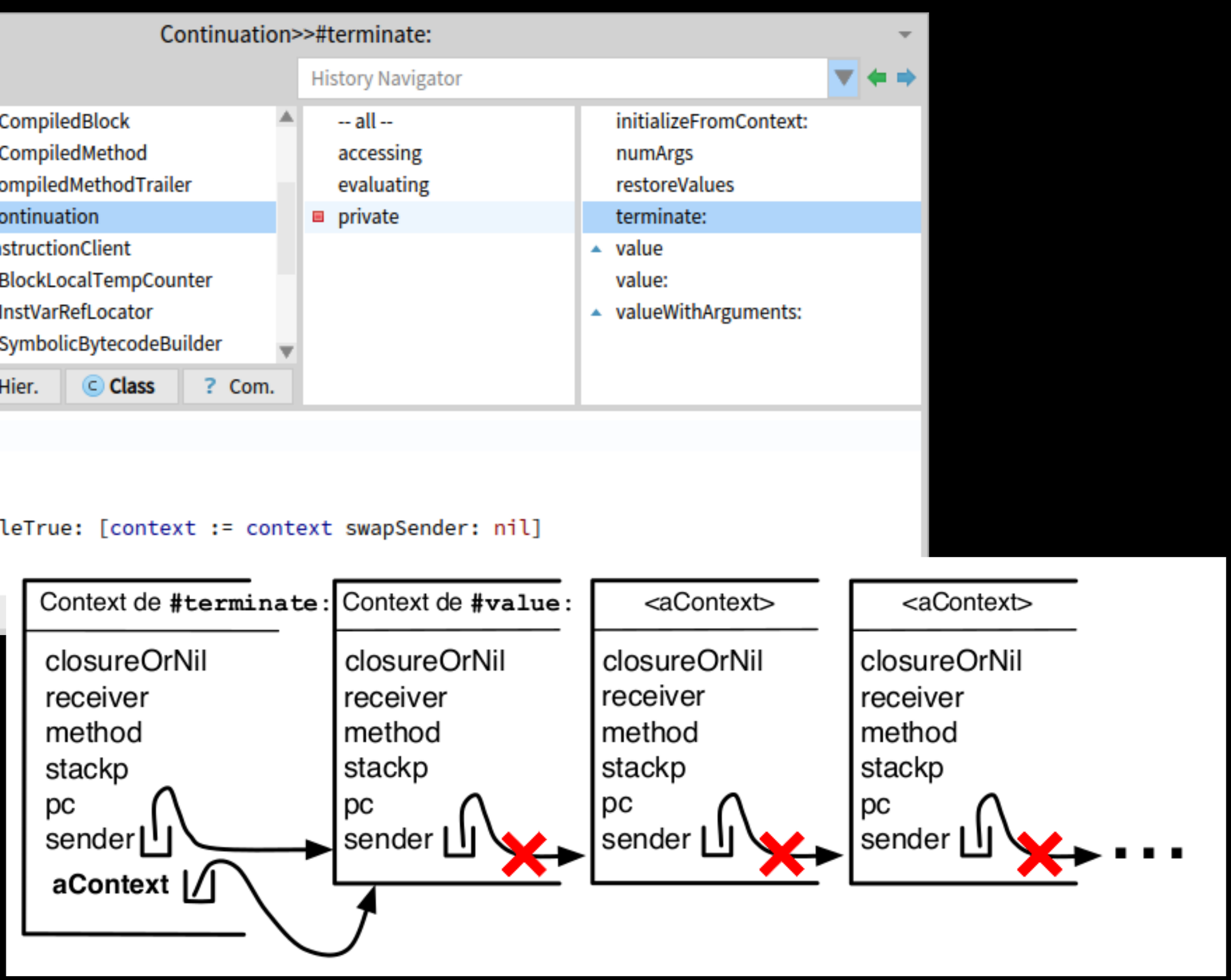
Hier. | Class | Com.

-- all --
accessing
evaluating
private

initializeFromContext:
numArgs
restoreValues
terminate:
value
value:
valueWithArguments:

```
terminate: aContext
    | context |
    context := aContext.
    [context notNil] whileTrue: [context := context swapSender: nil]
```

1/4 [1]



CAP: Reflexió en Smalltalk

Continuation>>#restoreValues

Scoped | Variables | History Navigator

Type: Pkg1|^Pkg2|Pk.*Core

Classes
Copying
Exceptions
Messaging
Methods
Models
Numbers
Objects
Pragmas

CompiledBlock
CompiledMethod
CompiledMethodTrailer
Continuation
InstructionClient
BlockLocalTempCounter
InstVarRefLocator
SymbolicBytecodeBuilder

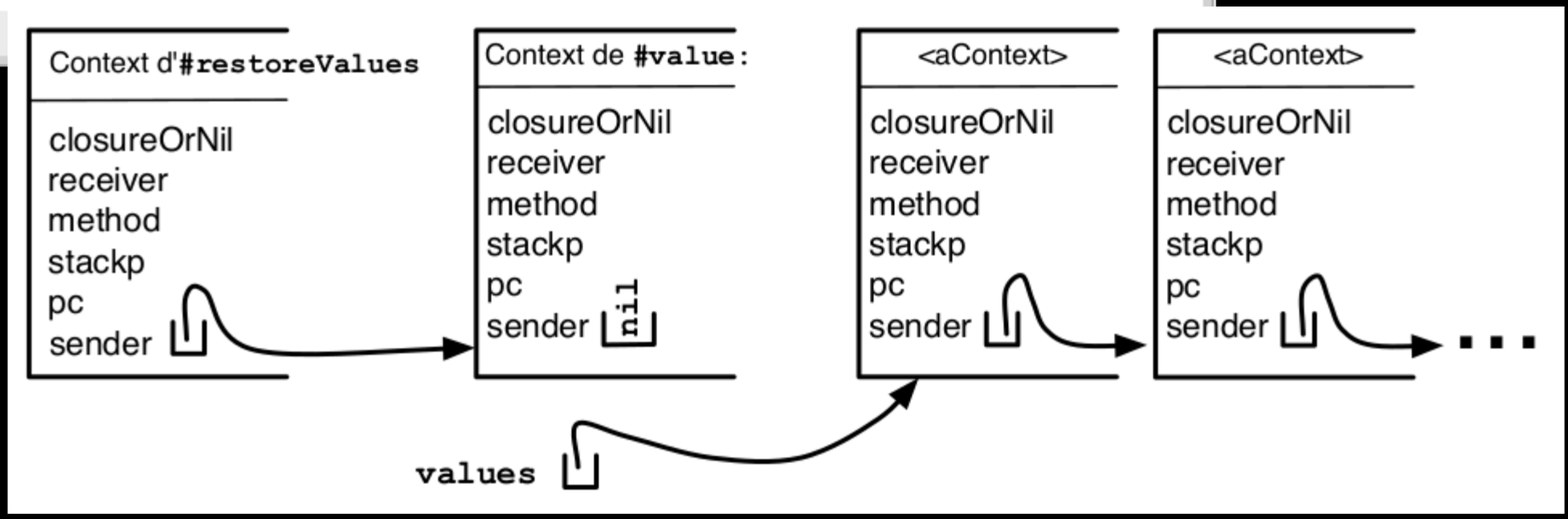
Hier. | Class | Com.

-- all --
accessing
evaluating
private

initializeFromContext:
numArgs
restoreValues
terminate:
value
value:
valueWithArguments:

```
restoreValues
    | valueStream context |
    valueStream := values readStream.
    [valueStream atEnd] whileFalse:
        [context := valueStream next.
        1 to: context class instSize do: [:i | context instVarAt: i put: valueStream next].
        1 to: context size do: [:i | context at: i put: valueStream next]]
```

1/7 [1]



CAP: Reflexió en Smalltalk

Continuation>>#value:

Scoped | Variables | History Navigator

Type: Pkg1|^Pkg2|Pk.*Core

Classes
Copying
Exceptions
Messaging
Methods
Models
Numbers
Objects
Pragmas

CompiledBlock
CompiledMethod
CompiledMethodTrailer
Continuation
InstructionClient
BlockLocalTempCounter
InstVarRefLocator
SymbolicBytecodeBuilder

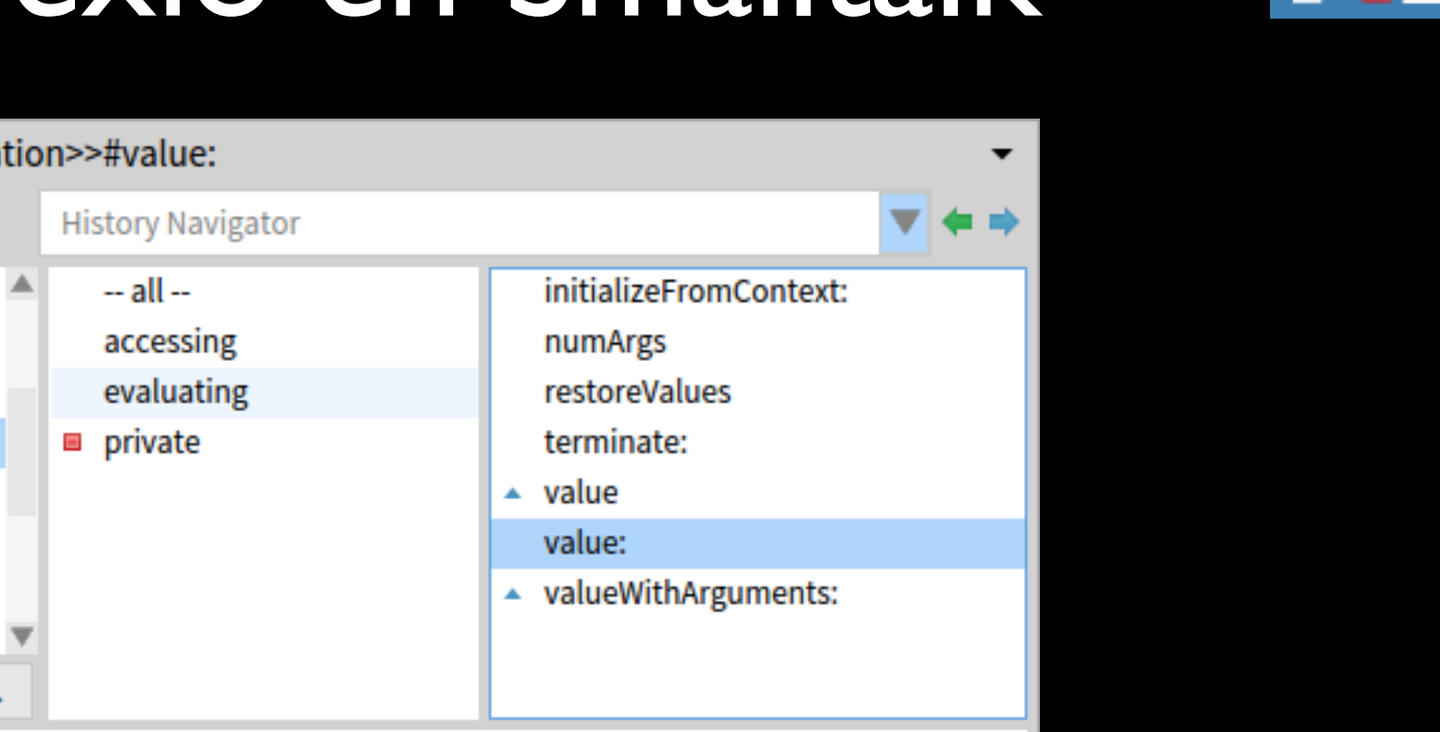
Hier. | Class | Com.

-- all --
accessing
evaluating
private

initializeFromContext:
numArgs
restoreValues
terminate:
value
value:
valueWithArguments:

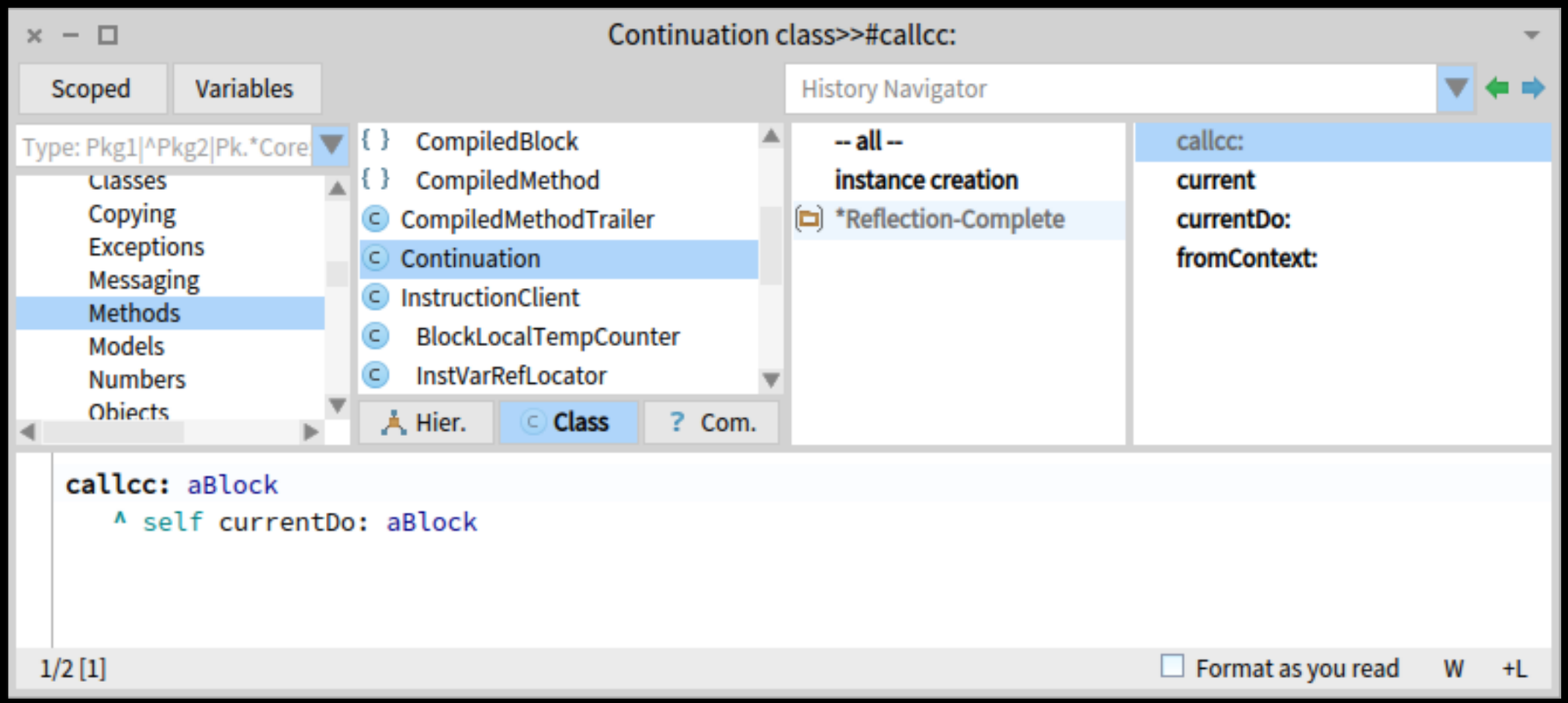
```
value: anObject
    "Invoke the continuation and answer anObject as return value."

    self terminate: thisContext.
    self restoreValues.
    thisContext swapSender: values first.
    ^ anObject
```

1/7 [1]



CAP: Reflexió en Smalltalk

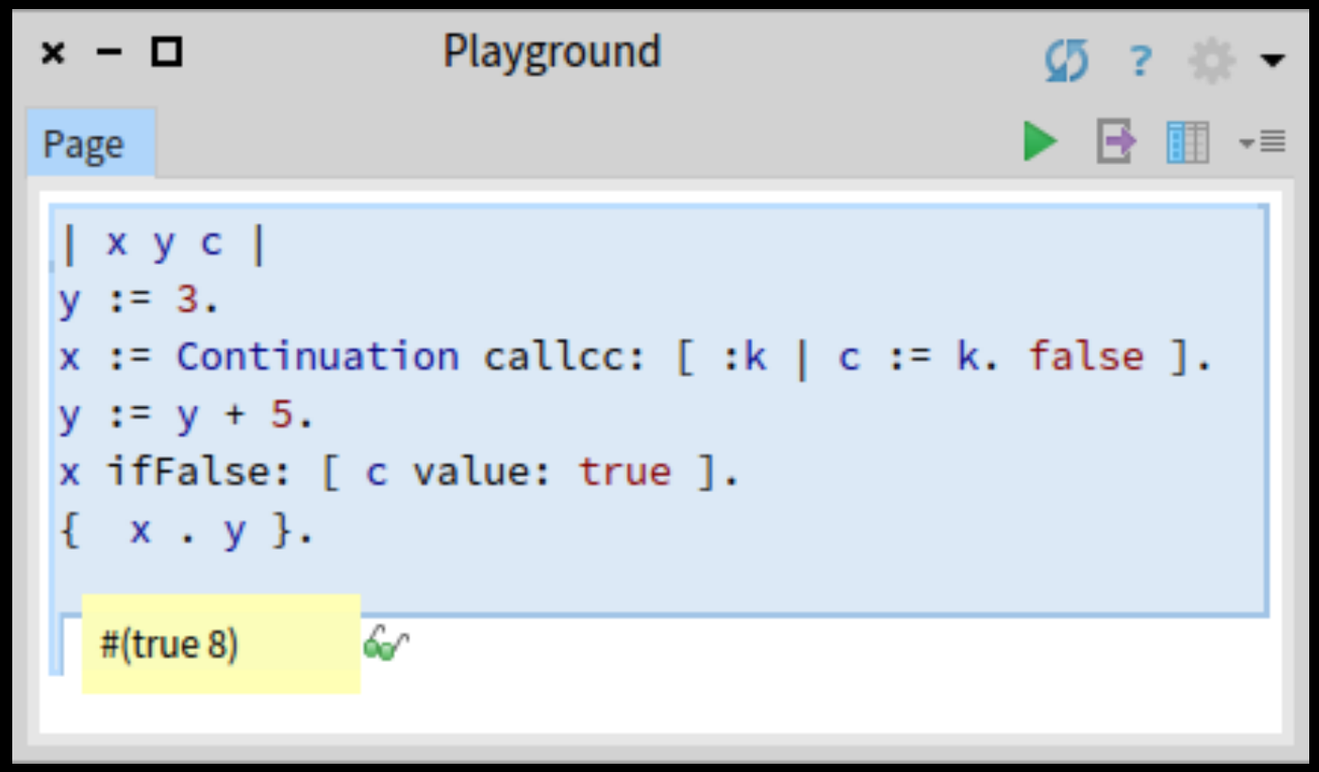
Finalment, volem crear instàncies de **`Continuation`** de manera controlada. Afegirem un mètode a **`Continuation class`** anomenat **`#callcc:`** que espera un bloc d'un paràmetre **`[ :k | . . . ]`** com a argument



CAP: Reflexió en Smalltalk

Exemple: Interacció entre Closures i Continuacions

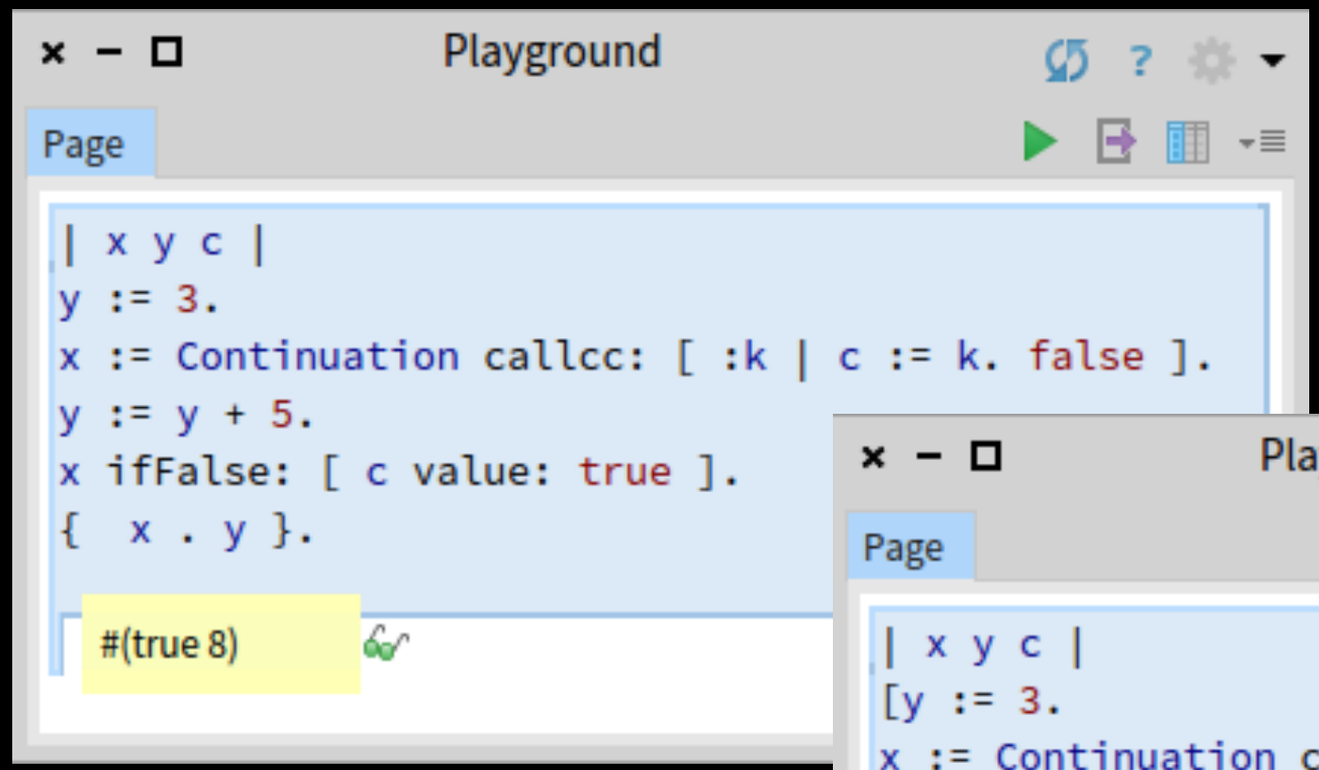
(recordeu allò que explicàvem sobre què es guarda en un `Context`?)

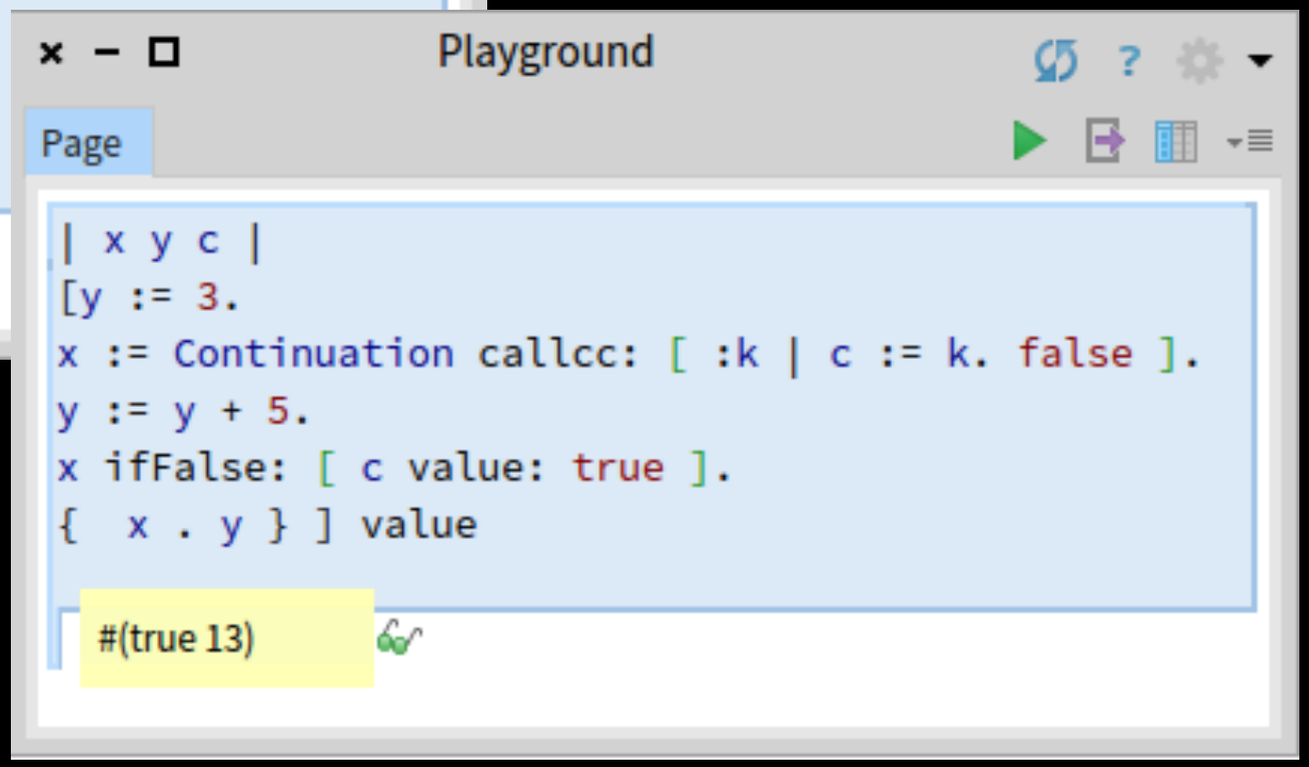


CAP: Reflexió en Smalltalk

Exemple: Interacció entre Closures i Continuacions

(recordeu allò que explicàvem sobre què es guarda en un `Context`?)





CAP: Reflexió en Smalltalk

Finalment, les conseqüències de tenir continuacions són força subtils...

Playground

Page

```
" Es cert que:  false = ([ :x | false ] value: <qualsevol cosa>) ?!? "
" Es a dir, per a tot programa P amb un 'forat' P = P[[forat]] on es pugui"
" posar un boolea, no importa si posem 'false' o [ :x | false ] value: <qualsevol cosa>"
" P[[ false ]] = P[[ ([ :x | false ] value: <qualsevol cosa>) ]]"
"                                                              "
" Si, si no tenim continuacions!"
"                                                              "
" En altre cas... "
" Si P[[ ]] es Continuation callcc: [ :k | [[ ]] ] "
" i <qualsevol cosa> = (k value: true) "

(Continuation callcc: [ :k | false ]) == (Continuation callcc: [ :k | ([ :x | false] value: (k value: true)) ])   false
```

CAP: Reflexió en Smalltalk