

# FIB - Conceptes Avançats de Programació

## Introducció a l'assignatura

### Professors

|                 |  |
|-----------------|--|
| Title           | Jordi Delgado  |
| <u>Untitled</u> | <a href="mailto:jdelgado@cs.upc.edu">jdelgado@cs.upc.edu</a> |
| <u>Untitled</u> | <a href="#">Ω - S115 (-1)</a>                                |

## Llenguatges de programació

Orientació a Objectes (classes):

- **Estàtic:** Java (java.lang.reflect)
- **Dinàmic:** Smalltalk

Orientació a Prototipus:

- JavaScript (ignorant classes)

---

## Introducció a l'Smalltalk

### Què és l'Smalltalk?

- Llenguatge i Entorn Orientat a Objecte
  - Tot és un objecte: **Uniformitat**
- Origen de moltes innovacions
  - **Refactorització, IDEs, MVC, xUnit, ...**
- Dinàmic i complement interactius
  - Millora els seus successors (!)
- Llenguatge petit i uniforme
- Herència Simple

- Dinàmicament tipat
- Principi subjacent: TOT és un objecte
- Els objectes existeixen en una imatge persistent
- Té origen amb interfície gràfica

## Referències

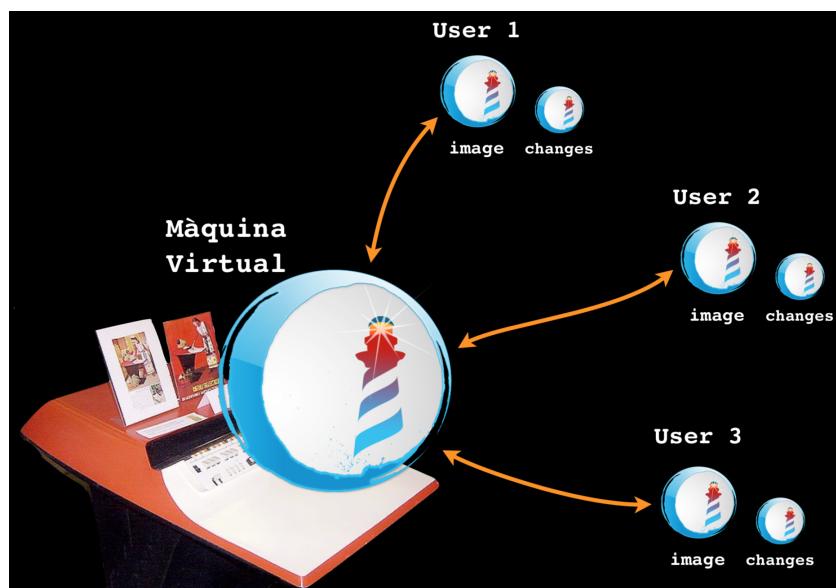
- Revista: byte - smalltalk
- [Pharo MOOC: Live Object Programming in Pharo](#)

## Objectes a Smalltalk

- Tot és un objecte. ABSOLUTAMENT TOT.
- Res passa si no és per **pas de missatges** (hi ha, però una excepció: l'assignació)
- L'estat dels objectes (atributs, camps, ...) és **privat**.
- Els mètodes són **públics** (els podem considerar privats per convenció).
- Les variables contenen **referències** (tipat dinàmic). (Els objectes no referenciats són eliminats *garbage collector*).
- Herència **simple**.

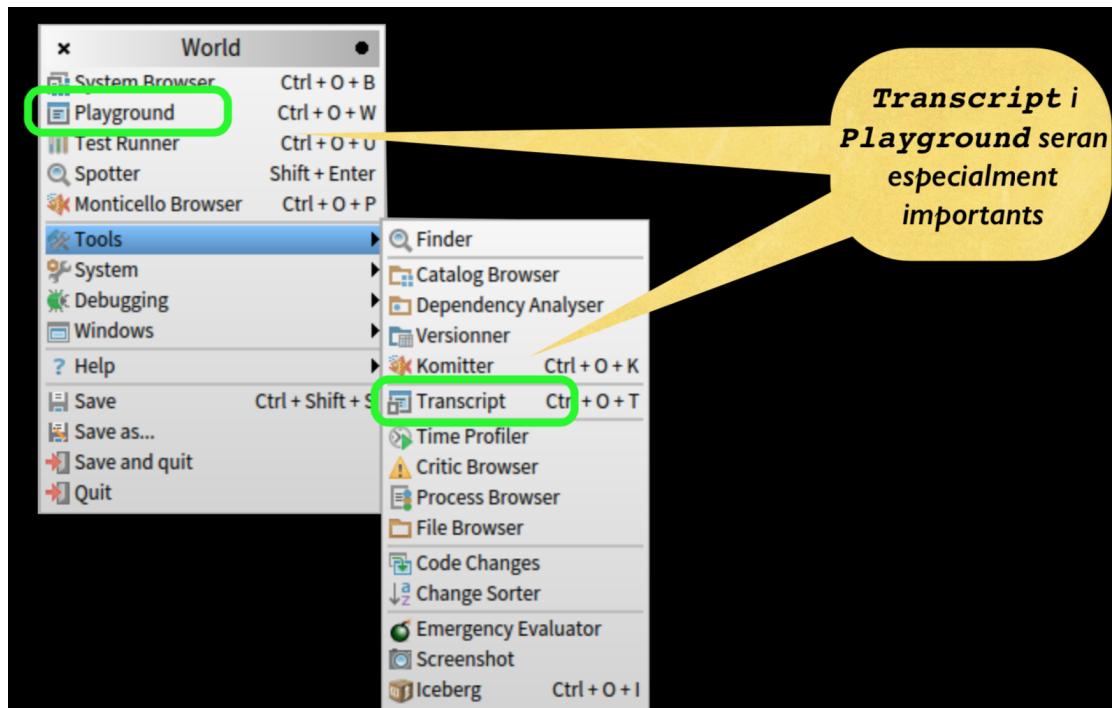
## Entorn

Un món on ho tenim tot a l'abast, fins i tot el codi font del sistema i de la màquina virtual (que, naturalment, està escrita en Smalltalk)



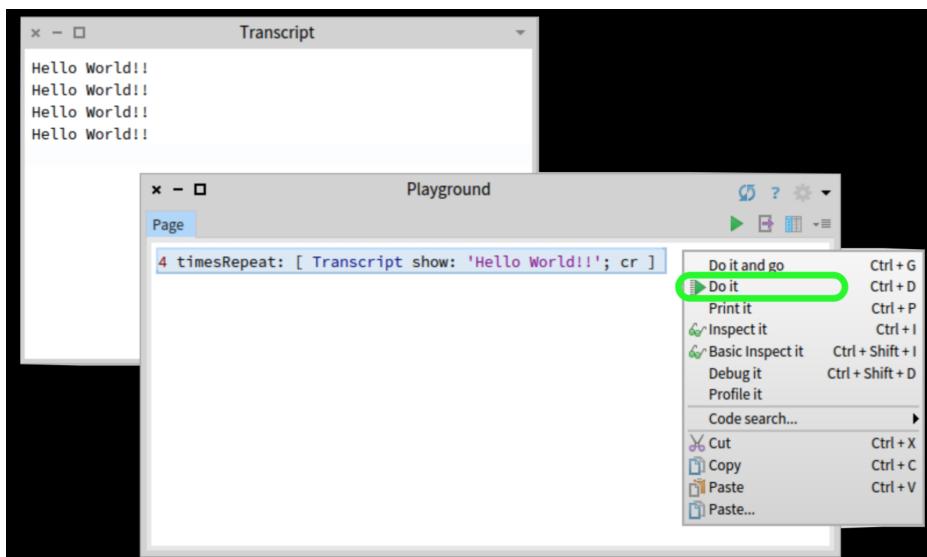
- Entorn que utilitzarem: **Pharo 6.1**

## Menú World per començar a interactuar amb el món

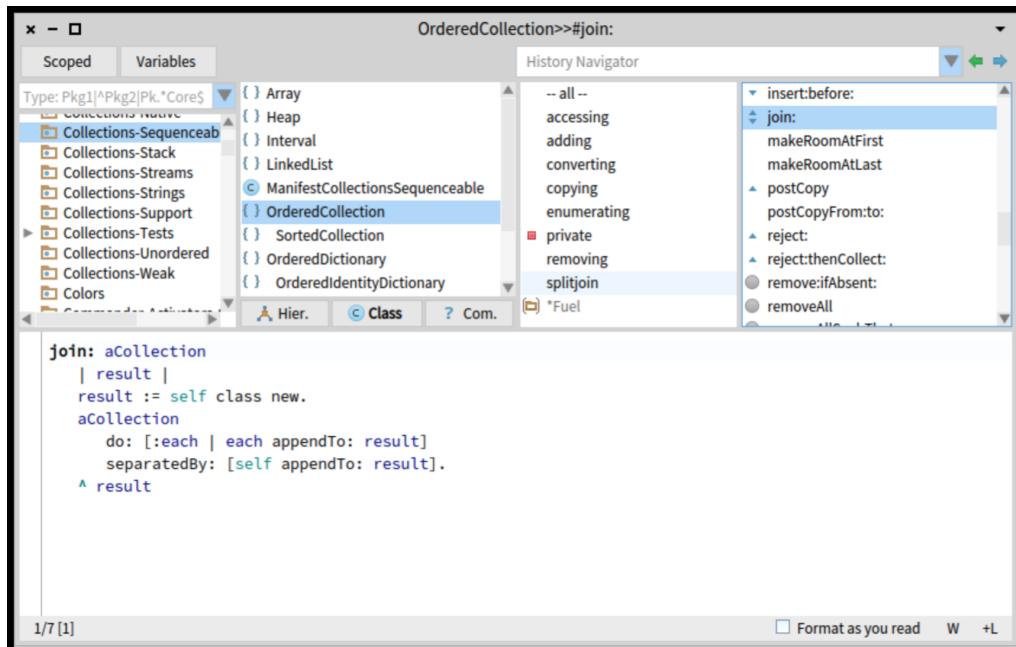


\$\$ Playground === Workspace \$\$

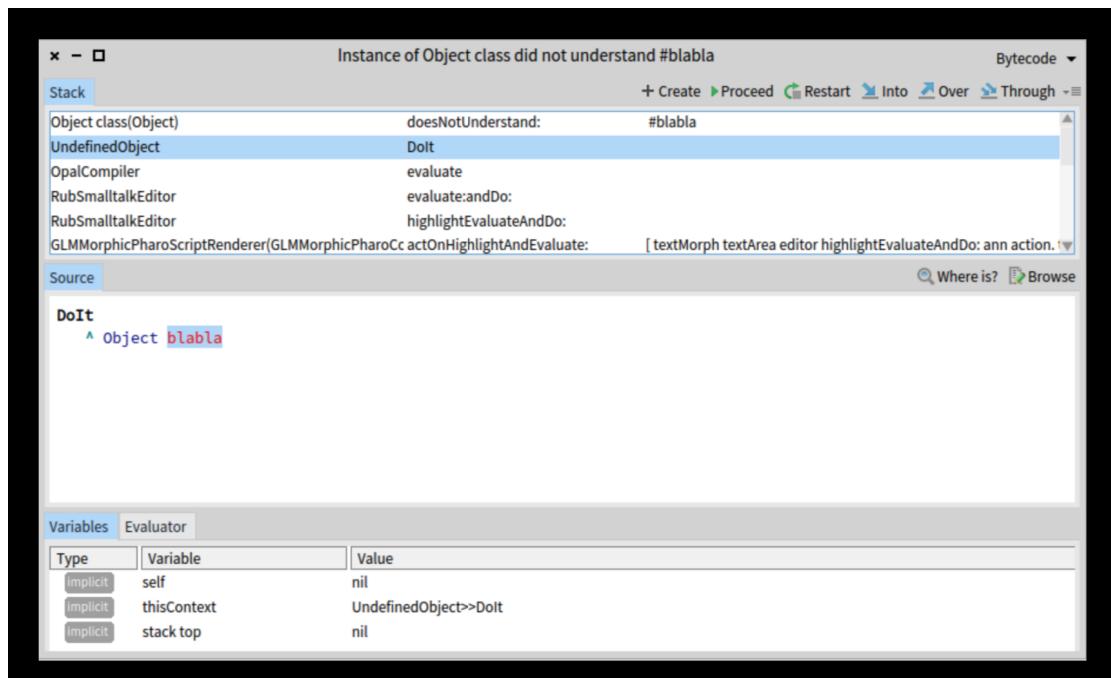
El Transcript és una mena d'estàndard output i el Playground és on provem el codi



## Usualment treballarem dins del Browser ...



## ... i del debugger



## Menus contextuais

- `accept` Compila un mètode o definició de classe
- `do it` Avalua una expressió
- `print it` Avalua una expressió i escriu el resultat

- `inspect it` Avalua una expressió i inspecciona el resultat

## Sintaxi

### Elements de la sintàxi

|                                  |                                     |
|----------------------------------|-------------------------------------|
| <code>^</code>                   | retorn                              |
| <code>" "</code>                 | comentari                           |
| <code>#</code>                   | símbol o array                      |
| <code>' '</code>                 | string                              |
| <code>[ ]</code>                 | bloc                                |
| <code>.</code>                   | separador d'instruccions            |
| <code>;</code>                   | cascada de missatges                |
| <code>   </code>                 | declaració de variables locals/bloc |
| <code>:=</code>                  | assignació                          |
| <code>\$ _</code>                | caràcter                            |
| <code>:</code>                   | final de selector de nom            |
| <code>&lt;primitive: &gt;</code> | crides a la MV                      |

## Exemples

|                                |   |
|--------------------------------|---|
| <code>comentari</code>         | <code>"això és un comentari"</code>       |
| <code>caràcter</code>          | <code>\$c \$g \$h \$# \$@</code>          |
| <code>string</code>            | <code>'això és una string'</code>         |
| <code>símbol</code>            | <code>#mac #linux #anonymous</code>       |
| <code>array</code>             | <code>#{(1 2 3 (1 3) \$a 'pozzi')}</code> |
| <code>array dinàmic</code>     | <code>{ 1+2 . 3/4}</code>                 |
| <code>(no Smalltalk-80)</code> |   |
| <code>enter</code>             | <code>2 2r1011</code>                     |
| <code>real</code>              | <code>1.4 6.03e-21 2.5e4</code>           |
| <code>booleà</code>            | <code>true false</code>                   |
| <code>pseudo variable</code>   | <code>self super</code>                   |

## Enviament de missatges

- Molts llenguatges defineixen els operadors bàsics i les estructures de control com a paraules clau del llenguatge.
- A Smalltalk només hi ha missatges
- Exemples:
  - `10 bitshift: 2` → Missatge `#bitshift:` enviat a l'objecte (número) 10, té com a paràmetre un altre número - el 2

- `(x > 1) ifTrue: [ Transcript show: 'més gran que 1'; cr]` ➔ Missatge `#ifTrue:` enviat a un booleà (resultat d'avaluar `(x>1)`) amb un bloc com a paràmetre.
- `#(a b c) do: [:each | each asString traceCr ]` ➔ Missatge `#do:` enviat a un array que escriu al Transcript els seus elements. Fixem-nos que `asString` és un missatge que s'envia a cada element per transformar-lo en una String
- `1 to: 10 do: [:each | each asString traceCr ]` ➔ Missatge `#to:do:` enviat a un objecte (número) 1 amb dos paràmetres, un enter (el 10) i un bloc. Aquí també fem servir `asString`

## Tres tipus de missatges

### Missatges Unaris

`Transcript cr , 5 factorial`

### Missatges Binaris

`3 + 4`

### Missatges de Paraula Clau

`Transcript show: 'Hola Món' , 5 raisedTo: 10 modulo: 5`

### Precedències

1. S'evalua d'esquerra a dreta
2. Missatges unaris tenen la precedència més alta
3. Després els binaris
4. Per últim, els clau
5. Sempre podem posar parèntesis per forçar l'ordre d'avaluació

Exemple:

|| \$2 raisedTo: 1 + 3 factorial\$\$  
 || equival a  
 || \$2 raisedTo: (1 + (3 factorial))\$\$

Exemple:

`$$1+2*3$$`

equival a

`$(1+2)*3$`

## Missatges binaris - Sitaxi

```
unReceptor unSelector unArgument
```

On `unSelector` pot ser un o dos caràcters dels següents:

`+ - / \ * ~ ` < > = @ % | & ! ? , $`

(el segon caràcter no pot ser \$)

Exemples:

- `2 + 3 - 5`
- `5 >= 7`
- `6=7`
- `'hola' , ' ', 'món' (3 @ 4) + (1 @ 2)`
- `2 << 5`

## Separadors de sentències

- Punts (.) - Separen sentències:
  - `Transcript show: 'Hola'.`
  - `Transcript show: ' '.`
  - `Transcript show: 'món' "aquí no cal"`
- Punt i coma (;) - Enviament d'una cascada de missatges al mateix objecte.
  - `Transcript show: 'Hola'; show: ' ';` show: 'món'.

## Més sintaxi

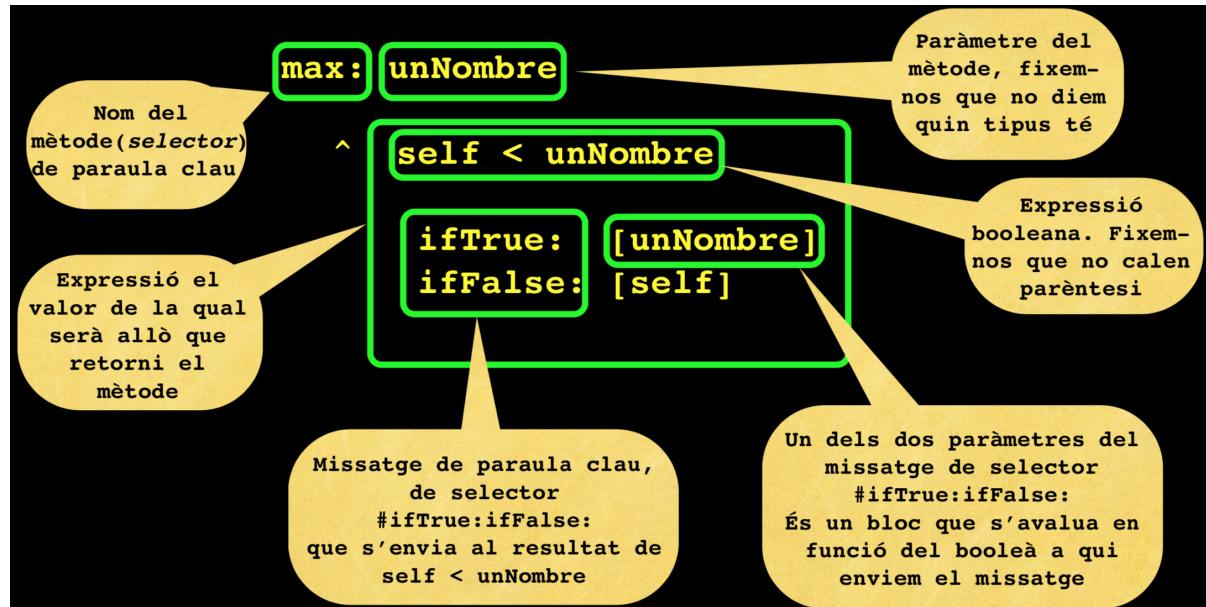
- Variables:
  - Declaració:
    - Es llisten entre barres verticals, `| x y z |`
  - Assignació:

- Té la forma `var := expressió`
- NO es fa per pas de missatges.
- Mètodes:
  - Retornen un valor **sempre** (`self` per defecte).
  - S'utilitza `^` per explicitar el valor de retorn
  - El selector d'un mètode és un símbol: `#collect:`
- Blocs (closures):
  - Es delimiten entre `( )`
  - Un bloc retorna el valor de la darrera expressió avaluada dins del bloc
  - Els paràmetres dels blocs es delimiten amb `:par |`

## Exemples

Definim un mètode per trobar el màxim de dos nombres:

```
max: unNombre
  ^ self < unNombre
    ifTrue: [unNombre]
    ifFalse: [self]
```



També podríem haver fet el retorn explícit, des dels blocs

```

max: unNombre
  self < unNombre
    ifTrue: [^unNombre]
    ifFalse: [^self]

```

Totes les estructures de control són envois de missatges

## Exemples d'iteracions; totes fan el mateix:

```

|n|      "Programant C en Smalltalk :)"
n := 1.
[ n <= 10 ] whileTrue:
  [ Transcript show: n asString; cr.
  n := n + 1 ]
1 to: 10 do: [ :n | Transcript show: n asString; cr]
(1 to: 10) do: [ :n | Transcript show: n asString; cr]

```

```
OrderedCollection >> collect:
```

### collect: aBlock

"Evaluate aBlock with each of my elements as the argument. Collect the resulting values into a collection that is like me. Answer the new collection. Override superclass in order to use addLast:, not at:put::."

```

| newCollection |
newCollection := self species new: self size.
firstIndex to: lastIndex do:
  [:index |
  newCollection addLast: (aBlock value: (array at: index))].
^ newCollection

```

## Crear objectes

Hi ha essencialment dues maneres de crear objectes:

- **Mètodes de classe:** Usualment, però no necessàriament, `new`  
Exemples:

```
OrderedCollection new
Array with: 1 with: 2
```

- **Mètodes constructors:** Missatges a objectes que crean instàncies d'altres classes.

Exemples:

```
1 @ 2 "un punt"  
1 / 2 "una fracció"
```

## Crear classes

Enviant missatges a altres classes!

```
Magnitude subclass: #Number  
instanceVariableNames: ''  
classVariableNames: ''  
package: 'Kernel-Numbers'
```

```
Object variableSubclass: #BlockClosure  
instanceVariableNames: 'outerContext startpc numArgs'  
classVariableNames: ''  
package: 'Kernel-Methods'
```

```
SequenceableCollection subclass: #OrderedCollection  
instanceVariableNames: 'array firstIndex lastIndex'  
classVariableNames: ''  
package: 'Collections-Sequenceable'
```

#subclass:instanceVariableNames:classVariableNames:package:

És un missatge amb 4 paràmetres!

## Exercici - Endevinar nombres

Fer un programa que generi a l'atzar un nombre entre 1 i 100 i demani a l'usuari que l'endevini. A cada resposta de l'usuari escriurem pistes com 'més gran' o 'més petit' per ajudar-lo. Al final, s'escriurà el nombre de vegades que l'usuari ho ha intentat.

### Ajuts

- Demanar inputs:
  - `UIManager >> #request:initialAnswer:title:` (atenció! retorna una `String`)
  - Enviarem aquest missatge a l'objecte `UIManager default`
- Nombres enters a l'atzar entre 1 i 100: `100 atRandom`
- Escriurem el programa en el `Playground` i els resultats en el `Transcript`

### Solució

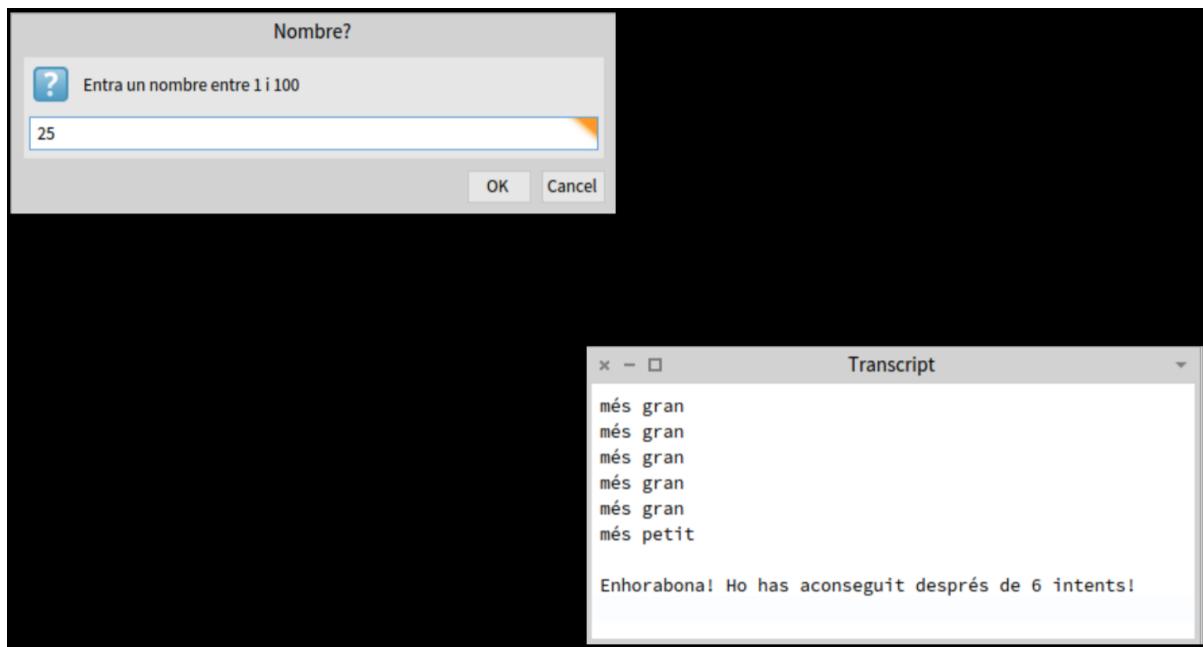
```

| nombreAEdevinar nombreProposat intents |

nombreAEdevinar := 100 atRandom.
intents := 0.
nombreProposat :=
(
    UIManager default request: 'Entra un nombre entre 1 i 100'
    initialAnswer: 0
    title: 'Nombre?'
) asInteger.

[nombreAEdevinar = nombreProposat]
whileFalse:
[
    Transcript show:
    (
        nombreProposat > nombreAEdevinar
        ifTrue: ['més petit']
        ifFalse: ['més gran']
    ); cr.
    nombreProposat :=
    (
        UIManager default
        request: 'Entra un nombre entre 1 i 100'
        initialAnswer: 0
        title: 'Nombre?'
    ) asInteger.
    intents := intents + 1
].
Transcript cr;
show: 'Enhorabona! Ho has aconseguit després de ', intents asString, ' intents!';
cr.

```



```

collect: aBlock
"Evaluate aBlock with each of my elements as the argument. Collect the
resulting values into a collection that is like me. Answer the new
collection. Override superclass in order to use addLast:, not at:put:."

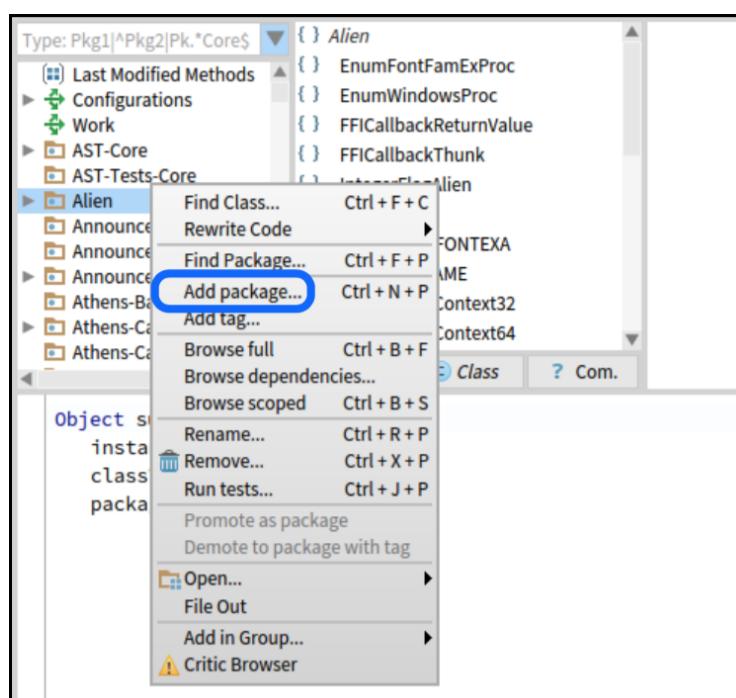
| newCollection |
newCollection := self species new: self size.
firstIndex to: lastIndex do:
[:index |
newCollection addLast: (aBlock value: (array at: index))].
^ newCollection

```

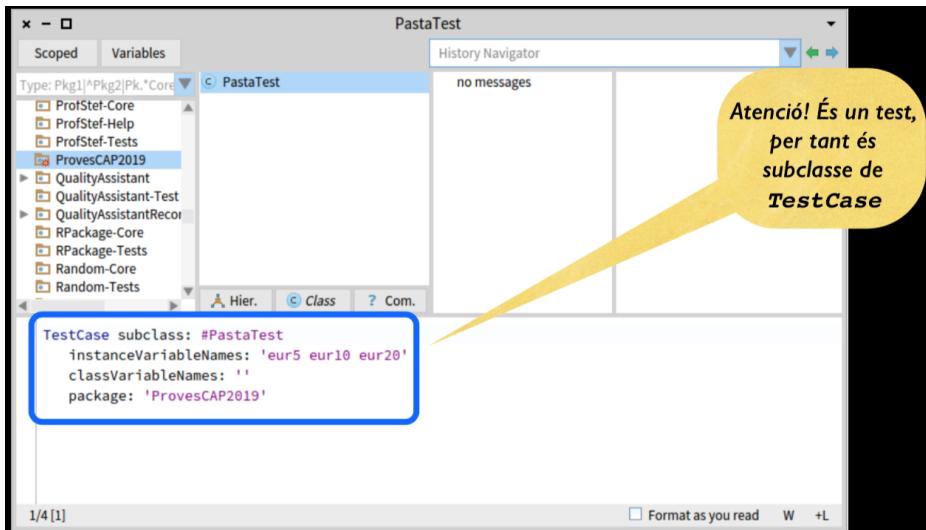
## Exercici: TDD en Smalltalk ➔ la classe Pasta

Volem implementar la classe **Pasta**, per poder manipular diferents quantitats en diferents monedes (euro, dolar, etc.) Programarem com és habitual, en el **System Browser**.

Afegirem un **Package** mitjançant el menu contextual de la finestra dels paquets.

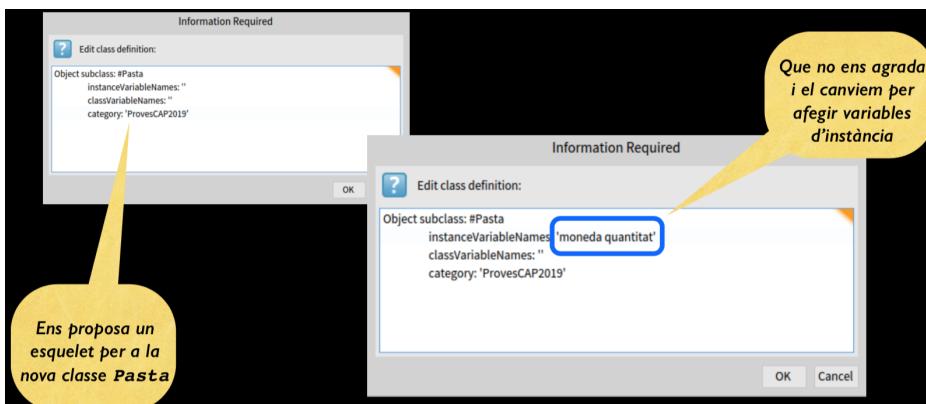
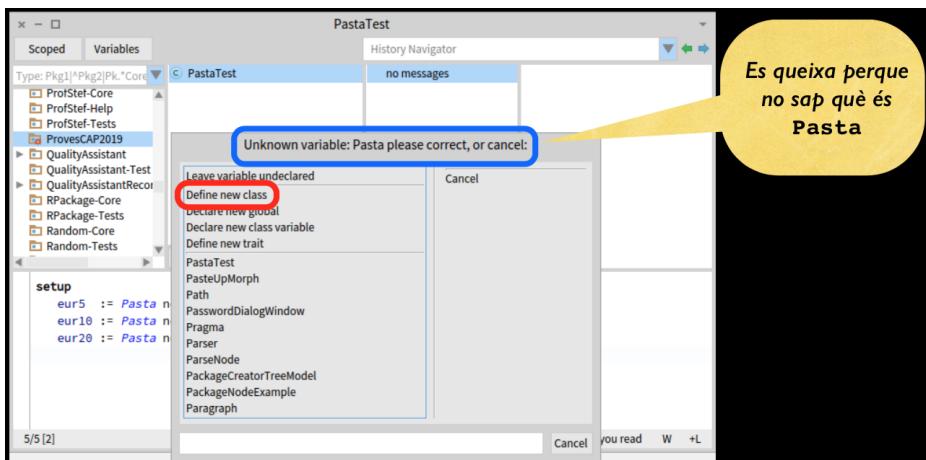


Anomenarem **Proves** (o com vulgueu) a la nova categoria i el primer que farem és definir la classe **PastaTest**.

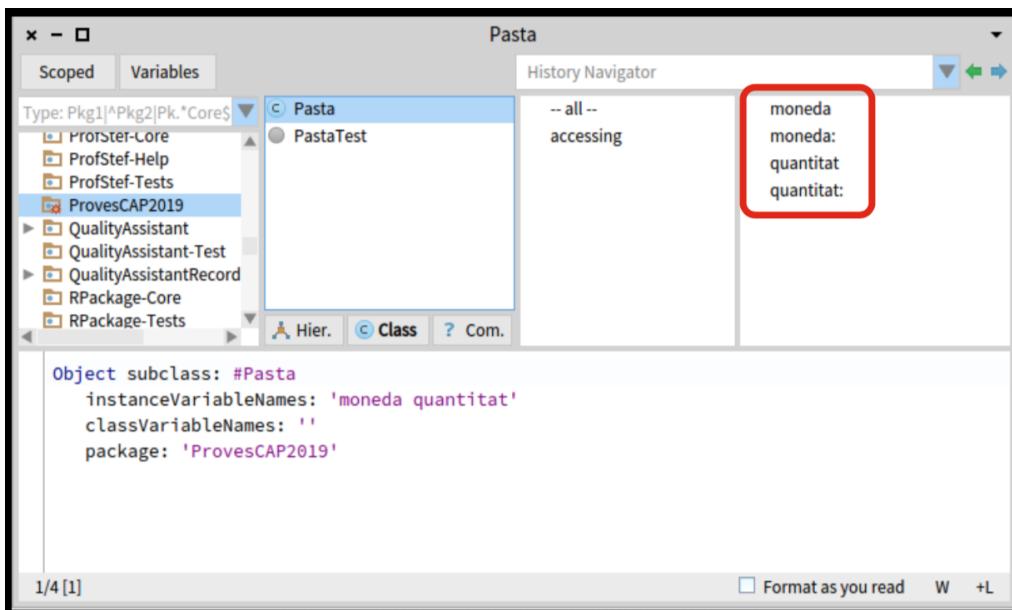
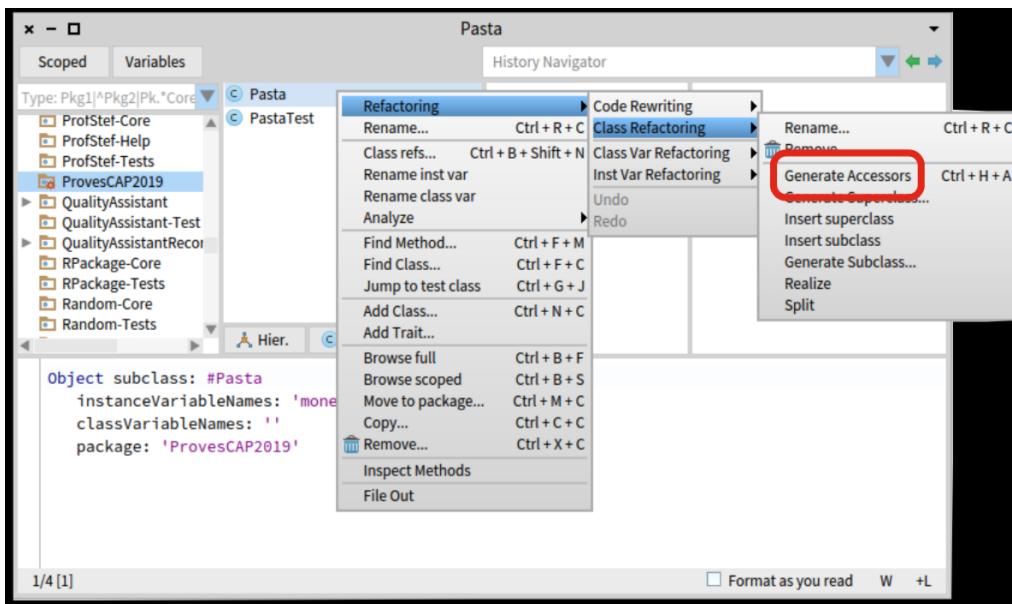


Crearem un mètode `setUp` per inicialitzar les variables d'instància de `PastaTest` és a dir, `eur5`, `eur10`, i `eur20`.

Atenció, el `setUp` s'executa abans de cada test!!



Crearem setters i getters per a totes les variables d'instància. Ho podem fer automaticament amb el menu contextual de les classes.



Amb aquesta infraestructura mínima, podem crear el nostre primer test, per mirar si les igualtats tenen algun sentit...

The screenshot shows the Eclipse IDE interface with the following details:

- Title Bar:** PastaTest>>#testIquals
- Left Side (Scope View):** Shows the package structure: Type: Pkg1|^Pkg2|Pk.\*Core\$ (expanded) and ProvesCAP2019 (selected).
- Middle Area:** Displays the code for the `testIquals` method:

```
testIquals
    self assert: eur5 = eur5.
    self assert: eur10 = (Pasta new moneda: 'EUR'; quantitat: 10).
    self assert: eur20 ~= eur10.
```
- Right Side (History Navigator):** Shows the execution history:
  - setUp
  - testIquals
- Bottom Status Bar:** Includes buttons for "Format as you read", "W", and "+L".

Podem mirar si el test és correcte...

The screenshot shows the Eclipse IDE interface after running the test, with the following details:

- Title Bar:** PastaTest>>#testIquals
- Middle Area:** Shows the test failure message: **TestFailure: Assertion failed**. Below it, the stack trace indicates the failure occurred in the `testIquals` method:

```
PastaTest(TestAssertion) assert:
PastaTest testIquals
PastaTest(TestCase) performTest
PastaTest(TestCase) runCase [ self setUp. self performTest ]
BlockClosure ensure:
PastaTest(TestCase) runCase
```
- Bottom Status Bar:** Includes buttons for "Proceed", "Abandon", "Debug", and "Report".
- Bottom Status Bar:** Includes buttons for "Format as you read", "W", and "+L".

NO HO ÉS!

```

PastaTest>>#testIguals
TestFailure: Assertion failed
Bytecode GT

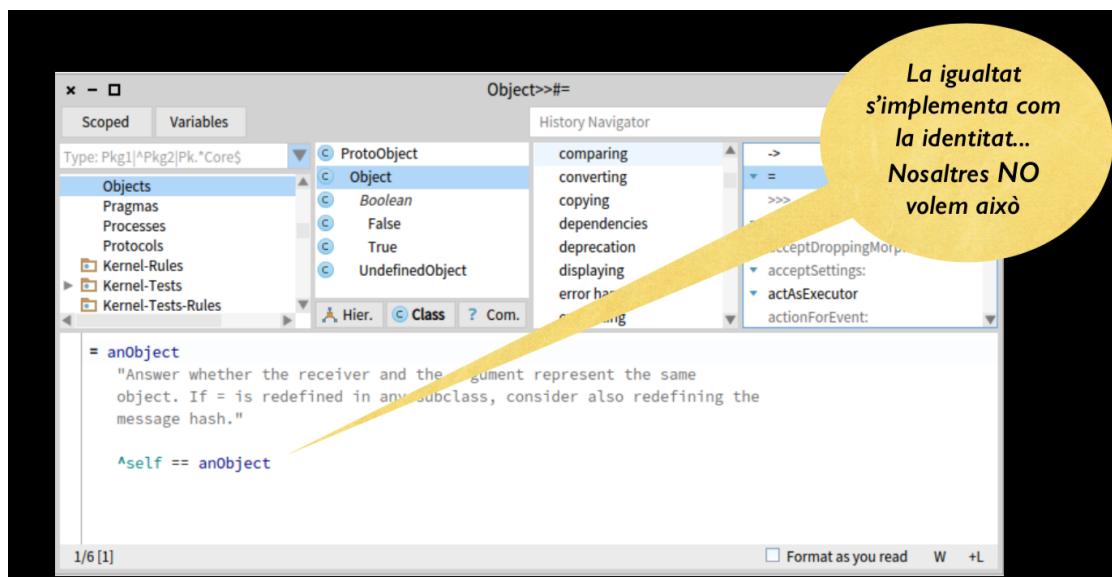
Stack
PastaTest(TestCase) assert:
PastaTest(TestCase) testIguals
PastaTest(TestCase) performTest
PastaTest(TestCase) runCase
[ self setUp, self performTest ]
Source SetUp
Where is? Browse

testIguals
    self assert: eur5 = eur5.
    self assert: eur10 = [Pasta new moneda: 'EUR'; quantitat: 10].
    self assert: eur20 ~~ eur10.

Variables Evaluator
Type Variable Value
implicit self PastaTest>>#testIguals
attribute eur10 a Pasta
attribute eur20 a Pasta

```

Hauríem de revisar què vol dir ser igual a... Com `Pasta` és subclasse d'`Object`, caldria revisar com aquest implementa el missatge `=` (ja que `Pasta` no l'implementa).



Per tant, caldrà que `Pasta` implementi el missatge `=`

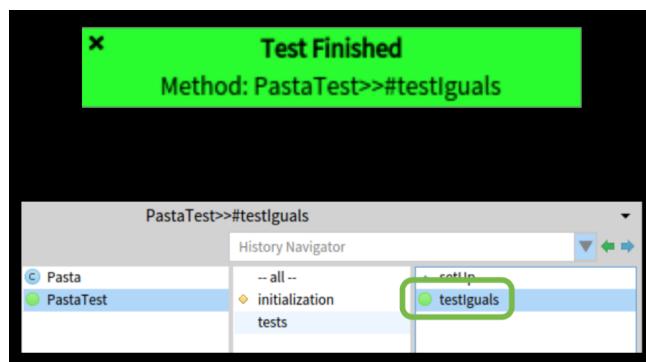
The screenshot shows the Rational Rose interface. On the left, the 'Variables' tab is selected in the 'History Navigator'. In the center, the code for the `Pasta` class is shown:

```
= unaPasta
  ^ self moneda = unaPasta moneda and: [ self quantitat = unaPasta quantitat ].
```

On the right, the 'History Navigator' pane shows the message `=` being implemented, with the message name `moneda` highlighted.

Per què no cal que `Pasta` implementi el missatge `~=` ???

Ara el test ja és correcte...



Podriem implementar la suma de monedes, i fer el test corresponent...

The screenshot shows the Rational Rose interface. On the left, the 'Variables' tab is selected in the 'History Navigator'. In the center, the code for the `Pasta` class is shown:

```
+ unaPasta
  ^ self moneda = unaPasta moneda
    ifTrue: [ Pasta new moneda: self moneda; quantitat: (self quantitat + unaPasta quantitat) ]
    ifFalse: [ self error: 'No es poden utilitzar monedes diferents' ]
```

On the right, the 'History Navigator' pane shows the message `+` being implemented, with the message name `moneda` highlighted.

```

Pasta>>#+
-- all --
accessing
arithmetic
+ =
moneda

PastaTest>>#testSumes
-- all --
initialization
tests
setUp
testquals
testSumes

testSumes
self assert: eur5 + eur5 = eur10.
self assert: eur5 + eur10 = (Pasta new moneda: 'EUR'; quantitat: 15).
self assert: (eur5 + eur5 + eur10) = eur20.

1/5 [1]

```

Voldriem tenir un constructor per a la classe `Pasta`, ja que la construcció `Pasta  
new moneda: x; quantitat: y` no ens agrada.

```

Pasta class>>-moneda:quantitat:
History Navigator
-- all --
instance creation
moneda:quantitat:

Atenció!!

```

```

Scoped Variables
Type: Pkg1|^Pkg2|Pk.*Core$ C Pasta
PragmaCollector
ProfStef-Core
ProfStef-Help
ProfStef-Tests
ProvesCAP2019
QualityAssistant
QualityAssistant-Test
QualityAssistantRecording
RPackage-Core

moneda: unaString quantitat: unNumero
^ self new moneda: unaString; quantitat: unNumero

2/2 [51]

```

Ara podem construir nous objectes de classe `Pasta` fent: `Pasta moneda: x quantitat: y`

Caldria canviar tot el codi on construim objectes de classe `Pasta` i utilitzar el nou constructor. Per exemple, la suma `+ ...`

```

Pasta>>#+
+ unaPasta
  ^ self moneda = unaPasta moneda
    ifTrue: [ Pasta moneda: self moneda quantitat: (self quantitat + unaPasta quantitat) ]
    ifFalse: [ self error: 'No es poden utilitzar monedes diferents' ]

```

3/5 [62] Format as you read W +L

Després caldrà tornar a passar els tests...

Però... no estem del tot satisfets. Estaria bé tenir un Constructor pels enters, un per a cada moneda:

`5 euro => Pasta moneda: 'EUR' quantitat: 5`

Primer es crea un nou protocol a la classe `Integer`, anomenat `*Proves`.  
*El nom del protocol és important*

```

Integer>>#euro
euro
  ^ Pasta moneda: 'EUR' quantitat: self

```

1/2 [1] Format as you read W +L

Podem redefinir el codi on construim euros i... passar altre cop els tests.

```

setUp
    eur5 := 5 euro.
    eur10 := 10 euro.
    eur20 := 20 euro.

```

I ara podríem continuar fent... però no ho farem. Guardeu la imatge si voleu conservar el que heu fet.

```

testSumes
    self assert: 5 euro + 5 euro = 10 euro.
    self assert: 5 euro + 10 euro = 15 euro.
    self assert: (5 euro + 5 euro + 10 euro) = 20 euro.

```

## Paquets (packages)

- Guardar la imatge és, però, molt poc pràctic. Per manipular codi amb certa agilitat a Pharo hi ha el concepte de **Package**, o Paquet.
- Il·lustrem-ho amb el nostre exemple. Hem creat un nou paquet anomenat **Proves** (o similar).
- Què hi ha dins el paquet?

- Totes les definicions de classes del paquet `Proves` o dels paquets que comencen amb `Proves-...`
- Tots els mètodes del protocol `*Proves` o `*Proves-...` allà on estiguin definits, no importa la classe (en el nostre cas, s'inclouria el mètode `euros` de la classe `Integer`).
- Tots els mètodes de les classes que pertanyen a les categories `Proves` i `Proves-...`, excepte aquells que pertanyen a protocols els noms dels quals comencin amb `*` (ja que pertanyen a altres paquets).
- Els paquets els gestionem mitjançant el `Monticello Browser`.
- Fixem-nos que ens marca el paquet `Proves` amb un asterisc (`*`), indicant-nos que cal actualitzar-lo (l'hem modificat i encara no l'hem guardat).
- Els paquets es guarden a **repositoris**, i SEMPRE n'hi ha un per defecte, al disc local, que dependrà de la nostra instal.lació de Pharo. Es diu `package-cache`.
- Ara bé, nosaltres preferirem guardar el codi on-line, à la GIT o Subversion. Per fer això caldrà fer servir `SmalltalkHub`.

Per poder crear repositoris a SmalltalkHub cal donar-se d'alta a  
<http://smalltalkhub.com>

Un cop donats d'alta com a usuaris, donem d'alta el projecte que associarem al paquet que volem gestionar des de `SmalltalkHub`

**Create a new project**

|                     |                          |  |
|---------------------|--------------------------|--|
| Project name        | ProvesCAP                | <small>NOTE</small> please be certain, it cannot be changed later. |
| Project website     | www.example.com          |  |
| Tags                | first tag, second tag    | <small>INFO</small> tags are separated by commas.                  |
| Project license     | MIT                      |  |
| Public write access | <input type="checkbox"/> |  |
| Description         |                          |  |

Ara ja puc fer servir el projecte SmalltalkHub que acabo de donar d'alta com a repositori pel paquet `Proves`, mitjançant `Monticello`.

**jdelgado / ProvesCAP** PUBLIC

Overview Source Commits Contributors Watchers Settings

**i Project infos** [Edit infos](#)

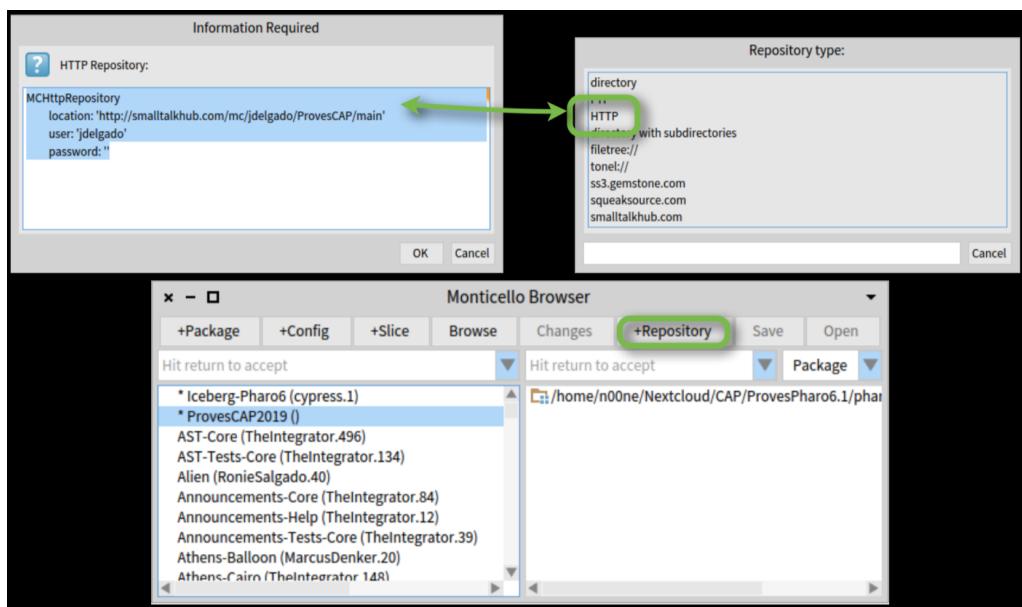
**Monticello registration**

|                  |  |
|------------------|--|
| MCHttpRepository | location: 'http://smalltalkhub.com/mc/jdelgado/ProvesCAP/main' |
|                  | user: ''   |
|                  | password: ''   |

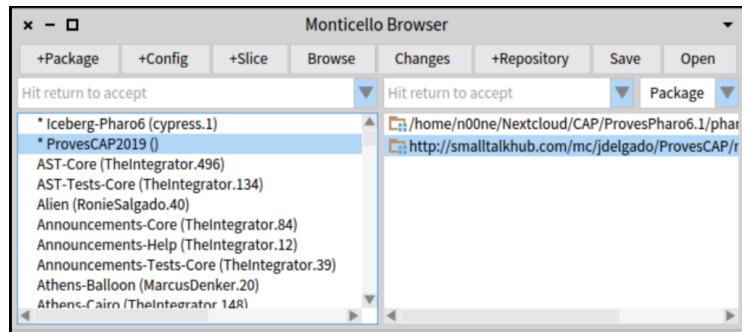
**About ProvesCAP**

Proves CAP

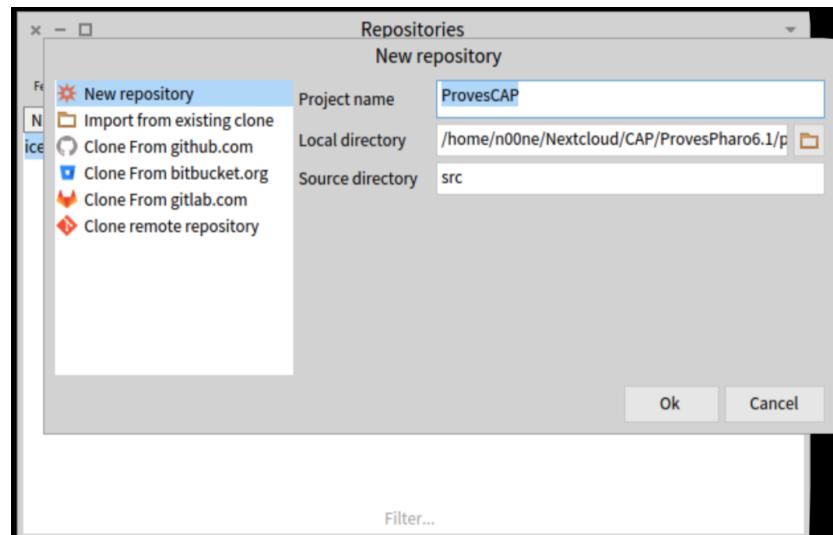
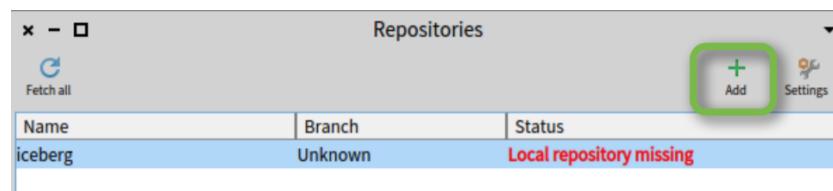
Tornem a Pharo i afegim al **Monticello Browser** el nou repository ...



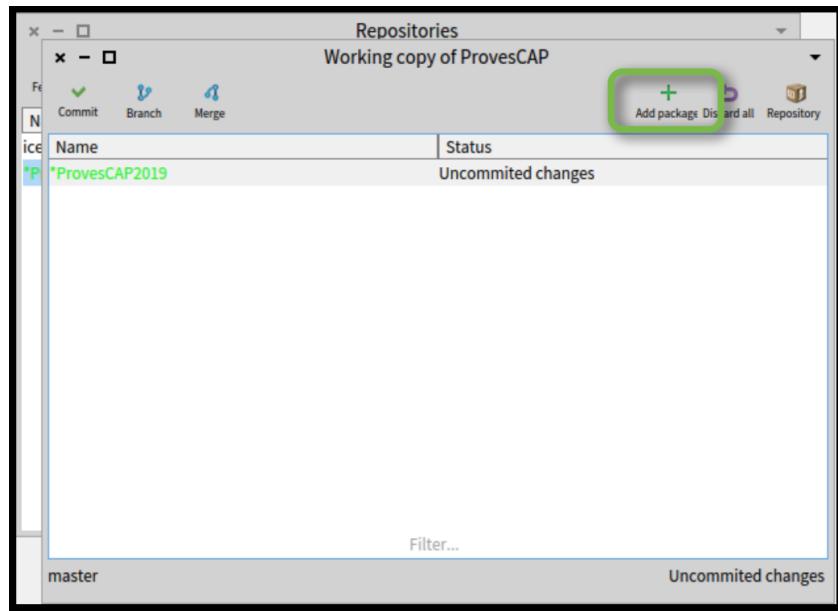
I ja podem vincular el paquet **Proves** al nou repositori SmalltalkHub dins el **Monticello Browser** (a més del repositori local). Com a mínim podem carregar paquets a la imatge (**Open→Load**) i guardar-los al repositori (Save), però podem fer més coses... Si voleu saber què més podem fer, llegiu el capítol 7 del llibre Deep into Pharo: <http://files.pharo.org/books-pdfs/deep-into-pharo/2013-DeepIntoPharo-EN.pdf>



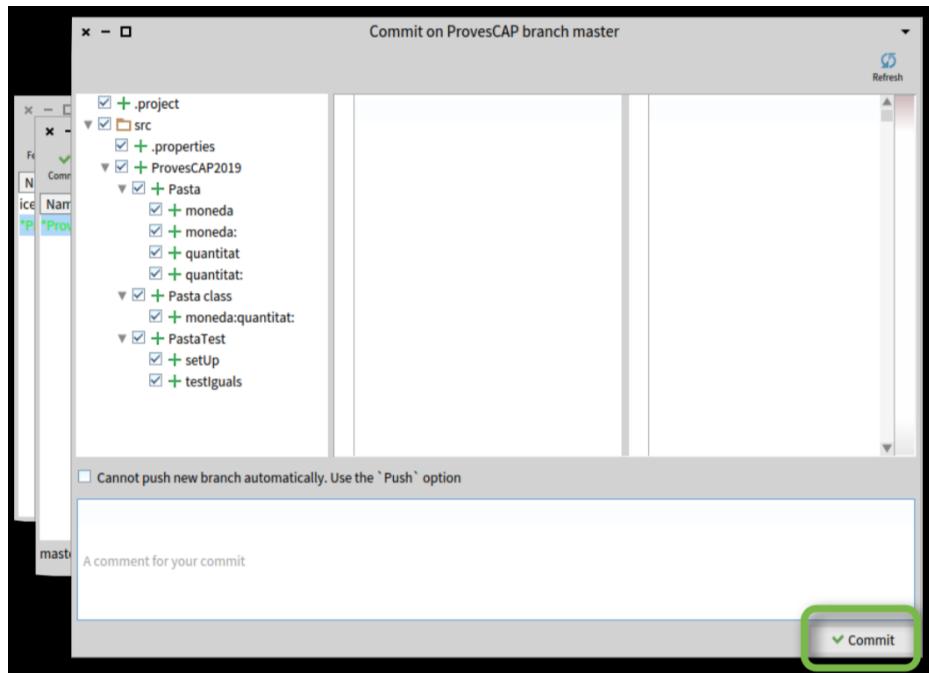
També podem gestionar el paquet `Proves` amb `Git` (a github, per exemple), utilitzant l'eina anomenada `Iceberg` (aquí suposarem que heu seguit les indicacions mencionades en la recepta d'instal·lació de Pharo que hem vist al principi i heu actualitzat Iceberg al vostre Pharo 6.1). Afegirem un repositori nou...

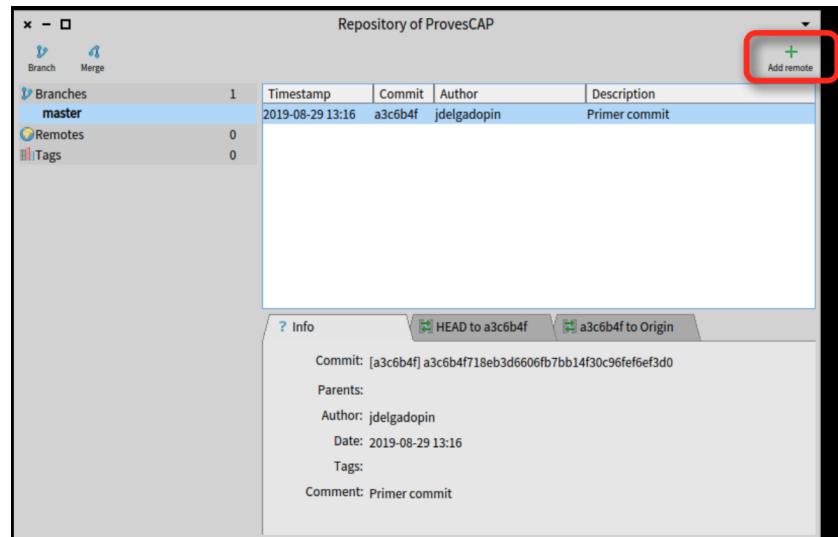


... i afegirem el paquet `Proves` a aquest repositori, amb `Add package`

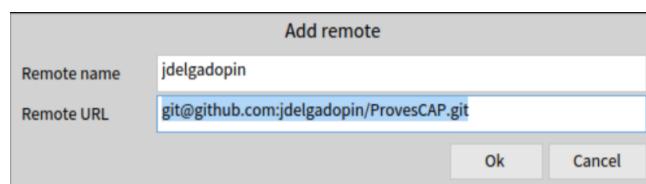


Ara, farem el primer (i únic en aquest exemple) **commit**, decidint què volem afegir i què no marcant-ho a la llista de fitxers. També hem de decidir quin repositori remot fem servir.

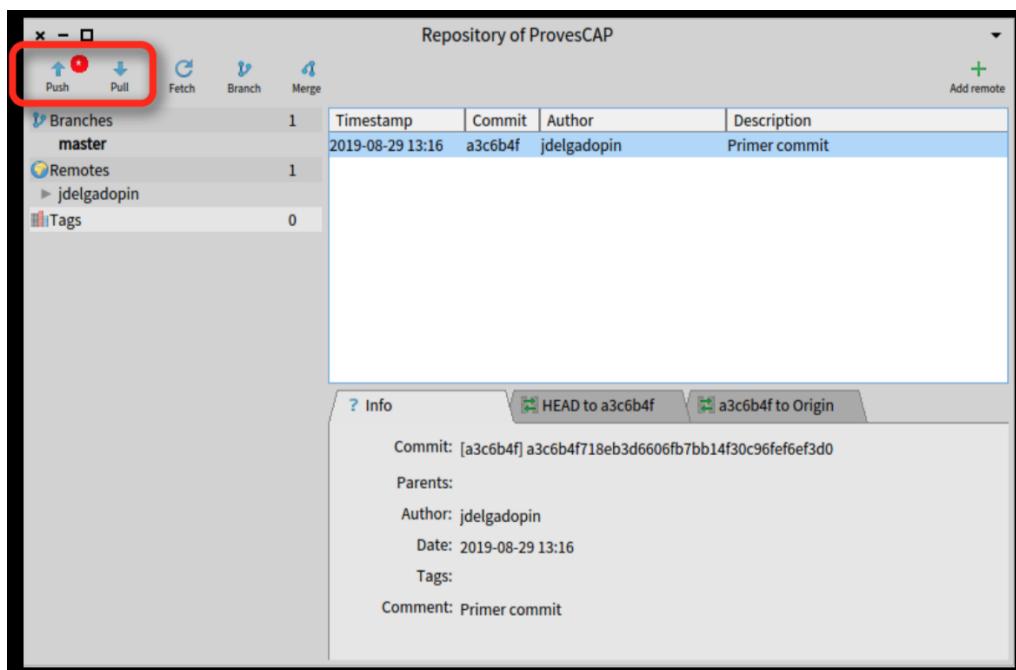


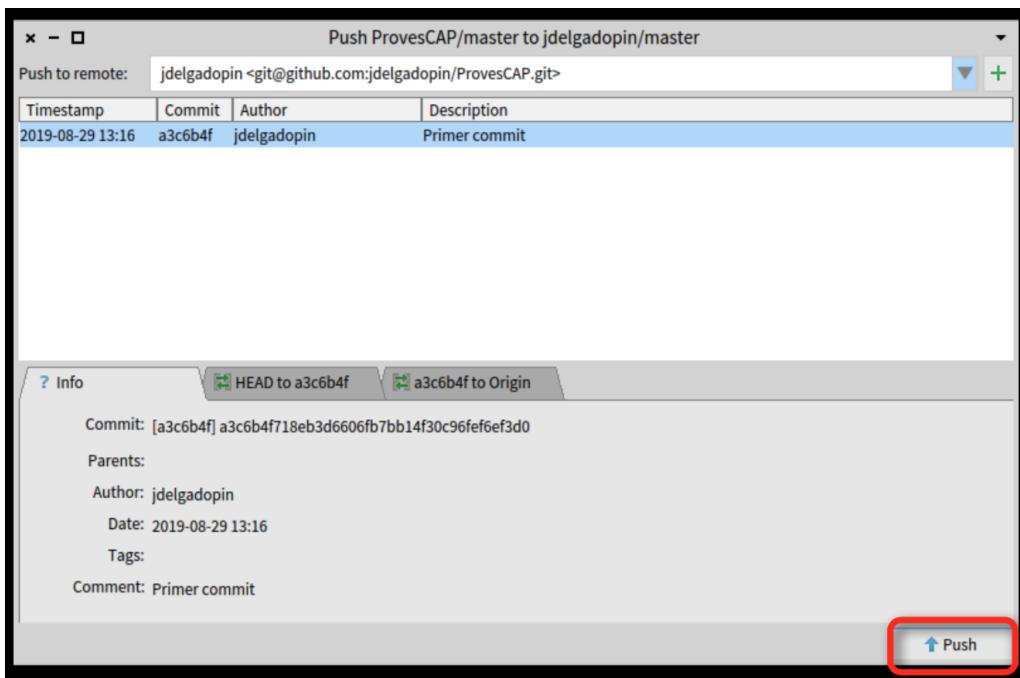


Vincularem el repositori remot (en aquest cas a github) al nostre repositori local (amb ssh segur que funciona):

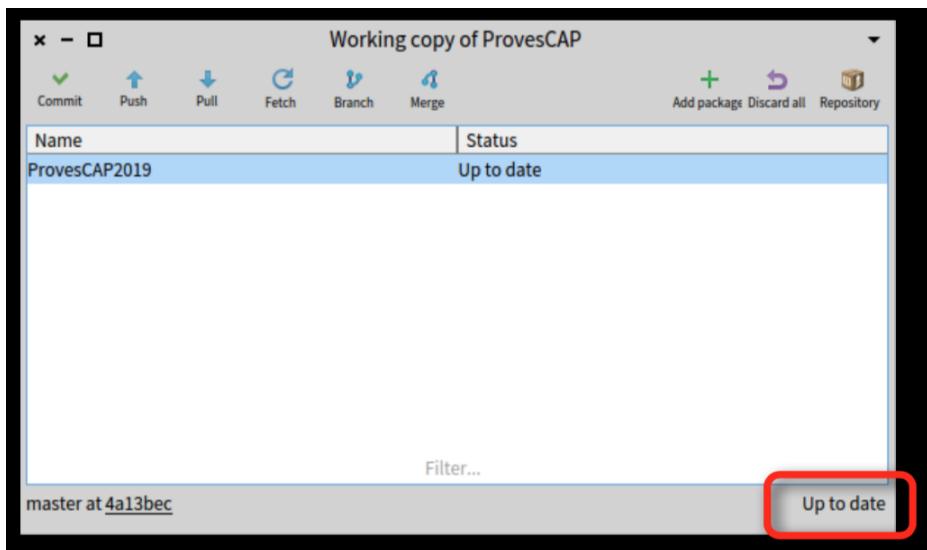


...i ja podem fer un **push**





Finalment, tot està en ordre...



Recordeu de consultar <https://github.com/pharo-vcs/iceberg/wiki/Tutorial>

## Recapitulem...

- `Tot` és un objecte. Absolutament tot. `Tots` els objectes són instància d'una classe. **TOTS!!** Les classes defineixen el comportament i l'estructura de les seves instàncies.
- Res passa si no és per `pas de missatges`
- L'estat dels objectes (atributs, camps,...) és `privat`

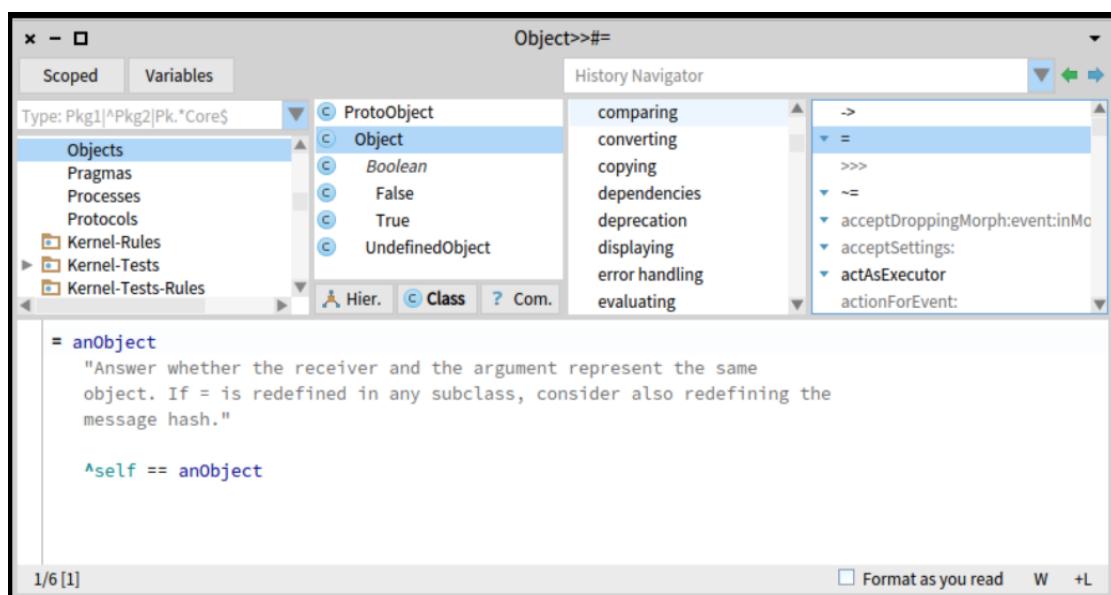
- Els mètodes són **públics** (els podem considerar privats per convenció)
  - Les variables contenen **referències** (tipat dinàmic). (els objectes no referenciats són eliminats, i.e. garbage collection)
  - Herència **simple**.

## La classe Object

- A l'estàndard Smalltalk-80, `Object` és l'arrel de la jerarquia de classes (a Pharo i Squeak l'arrel és `ProtoObject`, que només té a com a subclasse a `Object`)
  - Defineix el comportament comú i mínim per a tots els objectes del sistema: `Object numberOfMethods` → 429
  - Comparació d'objectes: `==`, `~~`, `isNil`, `=`, `~=`, `notNil`  
(a Pharo i Squeak `==`, `~~`, i `isNil` són a `ProtoObject`).
  - Representació textual d'objectes: `#printString`, `#printOn:`

## Identitat vs. Igualtat

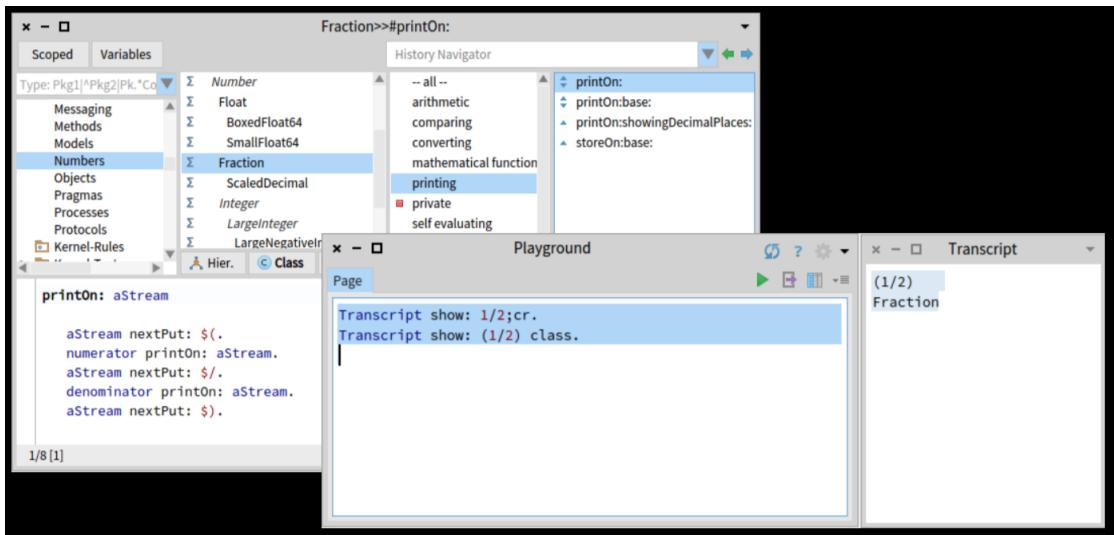
- `==` compara la identitat entre objectes. NO s'ha de sobreescriure mai!
  - `=` compara la igualtat entre objectes. Usualment es sobreescriu ja que la implementació per defecte és la identitat.
  - Però, si sobreescrivim `=` també cal sobreescriure `Object >> hash`



(no vam sobreescriure `Object >> hash` en l'exemple de la classe `Pasta` perque era un exemple de joquina)

## Representació Textual d'Objectes

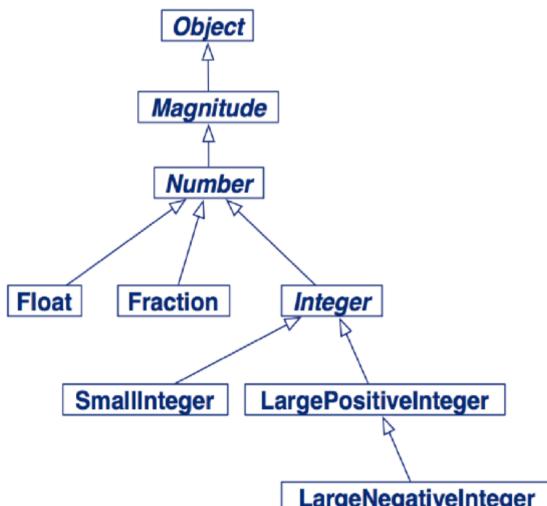
Normalment sobreescrivem `#printOn:` per donar representació textual als objectes. És el mètode per defecte utilitzat en diverses situacions, p.ex. pel `Transcript`



## Mètodes d'Object per donar suport al programador

- `#error:` genera un error
- `#doesNotUnderstand:` gestiona missatges no implementats
- `#halt`, `#halt:` invoca el debugger
- `#subClassResponsibility` el mètode que l'envia és abstracte
- `#shouldNotImplement` desactiva un missatge heretat

## Nombres



## Mètodes abstractes

**Number>>#+**

```
+ aNumber
    "Answer the sum of the receiver and aNumber."
    self subclassResponsibility
```

1/4 [1] Format as you read W +L

**Object>>#subclassResponsibility**

```
subclassResponsibility
    "This message sets up a framework for the behavior of the class' subclasses.
    Announce that the subclass should have implemented this message."
    SubclassResponsibility signalFor: thisContext sender selector
```

1/5 [1] Format as you read W +L

**SubclassResponsibility class**

```
SubclassResponsibility class
    instanceVariableNames: ''
```

1/2 [1] Format as you read W +L

The screenshot shows the Smalltalk Inspector interface with the title "SelectorException class>>#signalFor:". The left pane is a "History Navigator" showing a sequence of events:

- Type: Pkg1[^Pkg2]Pk.\*Core\$
- Chronology
- Classes
- Copying
- Exceptions
- Messaging
- Methods
- Models
- Numbers
- Objects
- Pragmas

The right pane shows the history of the message being sent:

- all -
- signaling

The message definition is shown in a red-bordered box:

```
signalFor: aSelector
    "Create and signal an exception for aSelector in the default receiver."
    ^ self new
        selector: aSelector;
        signal
```

The screenshot shows the Eclipse IDE interface with the Variables view open. The left pane displays a class hierarchy for `SelectorException`, with `SelectorException` selected. The right pane shows the History Navigator with a list of recent operations: `- all -`, `accessing`, `printing`, `messageText`, `selector`, `selector:`, and `standardMessageText`. Below the hierarchy, two lines of code are highlighted with a red box: `selector: aSelector` and `selector := aSelector`.

The screenshot shows the Smalltalk IDE interface with the following details:

- Scope:** Set to "Pkg1|^Pkg2|Pk.Core\$".
- Variables:** Tab selected.
- Exception > #signal:** The current class being viewed.
- History Navigator:** Shows a list of methods and objects related to the current context, with "signal" highlighted.
- Class Hierarchy:** A tree view showing the inheritance path from `ProtoObject` to `Exception`, which then branches into `Error`, `SelectorException`, and `SubclassResponsibility`.
- Code Editor:** Displays the following Smalltalk code:

```
signal
    "Ask ContextHandlers in the sender chain to handle this signal. The default is to
execute and return my defaultAction."
    signalContext := thisContext contextTag.
    signaler ifNil: [ signaler := self receiver ].
    ^ signalContext nextHandlerContext handleSignal: self
```
- Bottom Bar:** Includes buttons for "Format as you read" and "L" (likely a zoom or font size button).

## Coerció Automàtica

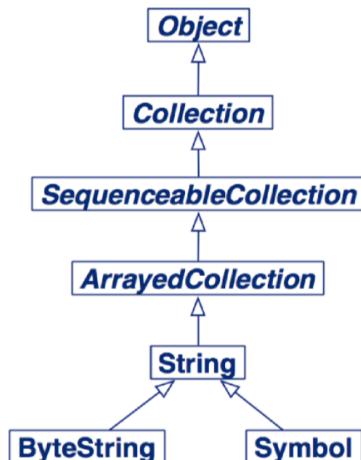
```

1+2.3 → 3.3 1 class → SmallInteger 1 class maxVal class → SmallInteger (1 class
maxVal + 1) class → LargePositiveInteger (1/3) + (2/3) → 1 1000 factorial / 999
factorial → 1000 2/3+1 → (5/3)

```

## Caràcters i Strings

- Caràcters: \$a \$F \$, \$( \$ñ \$4
- Caràcters no imprimibles: Character cr Character space Character tab
- Strings:
  - #mac asString → 'mac'
  - 12 printString → '12'
  - String with: \$A → 'A'
  - 'can''t' at: 4 → \$'
  - 'hello',' ','world' → 'hello world'



## Arrays Literals

- #('hola' #(1 2 3)) → #('hola' #(1 2 3))
- #(abc) → #(a#b#c)
- #(1+2) → #(1 #+ 2)
- #(Transcript show: 'hola') → #(Transcript #show: 'hola')

## Arrays i Arrays Literals

Els **Arrays** i els **Arrays literals** només es diferencien en el moment de ser creats: Els Arrays literals es coneixen en **temps de compilació** i en canvi els **Arrays** en temps d'execució.

Exemple:

El resultat d'avaluar

`#(Set new) ➔ #(#Set #new)`

és un Array amb dos símbols (NO és un Array amb una instància de Set)

El resultat d'avaluar

`Array with: (Set new) ➔ an Array(a Set())`

és un Array amb un element, una instància de Set

### Arrays: {} com a shortcut per a Array with:

(no és estàndard, però a Pharo i a Squeak està implementat)

Exemple:

`#(1+2.3) ➔ #(1#+2#.3){1+2.3} ➔ #(33) Arraywith: 1 + 2 with: 3  
➔ #(33)`

## Símbols i Strings

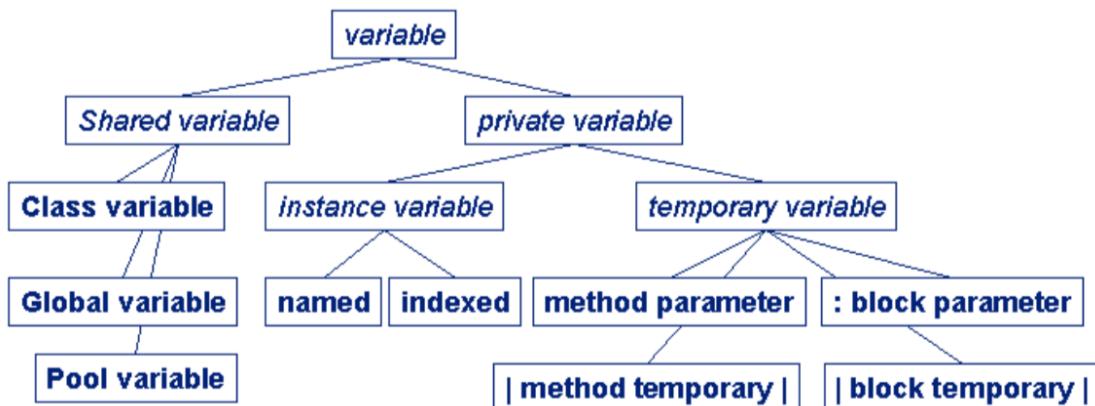
Els símbols són utilitzats com a claus úniques per a diccionaris i com a selectors de mètodes. Els símbols són `read-only`, immutables i únics, les `Strings` són mutables i no són úniques.

- `'calvin' = 'calvin'` ➔ `true`
- `'calvin' == 'calvin'` ➔ `true`
- `'cal','vin' = 'calvin'` ➔ `true`
- `'cal','vin' == 'calvin'` ➔ `false`
- `#calvin = #calvin` ➔ `true`
- `#calvin == #calvin` ➔ `true`
- `#cal,#vin = #calvin` ➔ `true`
- `#cal,#vin == #calvin` ➔ `false`
- `#cal,#vin ➔ 'calvin'`
- `(#cal,#vin) asSymbol == #calvin` ➔ `true`

## Variables

A *Smalltalk* les variables contenen referències a objectes, per això no cal dir quin tipus tenen (tipat dinàmic).

Hi ha una convenció: si el nom d'una variable comença per majúscula, aquesta és *shared*, si no, és local (*private*).



## Assignació

L'assignació lliga (*binds*) un nom a una referència a un objecte (això NO es fa amb pas de missatges)

- No podem assignar valors als paràmetres dels mètodes (no hi ha cap problema, utilitzar variables temporals)
- Noms (variables) diferents poden referenciar el mateix objecte (*aliasing*)
  - | p1 p2 |
  - p1 := 3@4. “un Punt”
  - p2 := p1.
  - p1 setX: 5 setY: 6.
  - p2 ➔ 5@6

## Variables d'Instància

Les variables d'instància definides a una determinada classe són privades de l'objecte instància de la classe, tot i que, lògicament, són visibles pels mètodes de la classe on estan definides i pels mètodes de les subclasses d'aquesta classe.

Tenen la mateixa vida que l'objecte instància de la classe on estan definides.

S'aconsella crear getters i setters per a cada variable d'instància i mai accedir-ne directament (per encapsular inicialitzacions i manipulacions). Poseu-los dins d'un

protocol anomenat private.

## Pseudo-variables

Definides al compilador, formen part de l'estàndard Smalltalk-80

- `nil` → referència a l' `UndefinedObject`
- `true` → instància única de la classe `True`
- `false` → instància única de la classe `False`
- `self` → referència a l'objecte al que pertany el mètode que s'està executant
- `super` → referència a l'objecte al que pertany el mètode que s'està executant, PERÒ la cerca del mètode comença a la superclasse de la classe on està definit el mètode que envia el missatge a `super`
- `thisContext` → reificació del context d'execució

## Blocs (closures)

- Un bloc (o closure) es pot entendre com un fragment de codi que capture el seu context lèxic en el moment de la creació.
- A Smalltalk els blocs són valors qualsevol, se'ls pot passar com paràmetre de missatges, guardar en variables, etc. - són valors de primera classe.
- S'utilitzen per ajornar l'avaluació de codi.
- Sintaxi:

```
[ :arg1 :arg2 ... |
| temp1 temp2 ...|
  expressio1.
  expressio2. ...
]
```

- Un bloc retorna el valor de la darrera expressió (avaluada) del bloc.
- Per avaluar un bloc cal enviar el missatge `#value`, `#value:`, `#value:value:`, etc. dependent dels paràmetres que tingui el bloc.

```
|sqr|
sqr := [ :n | n*n ].
```

```
sqr value: 5
```

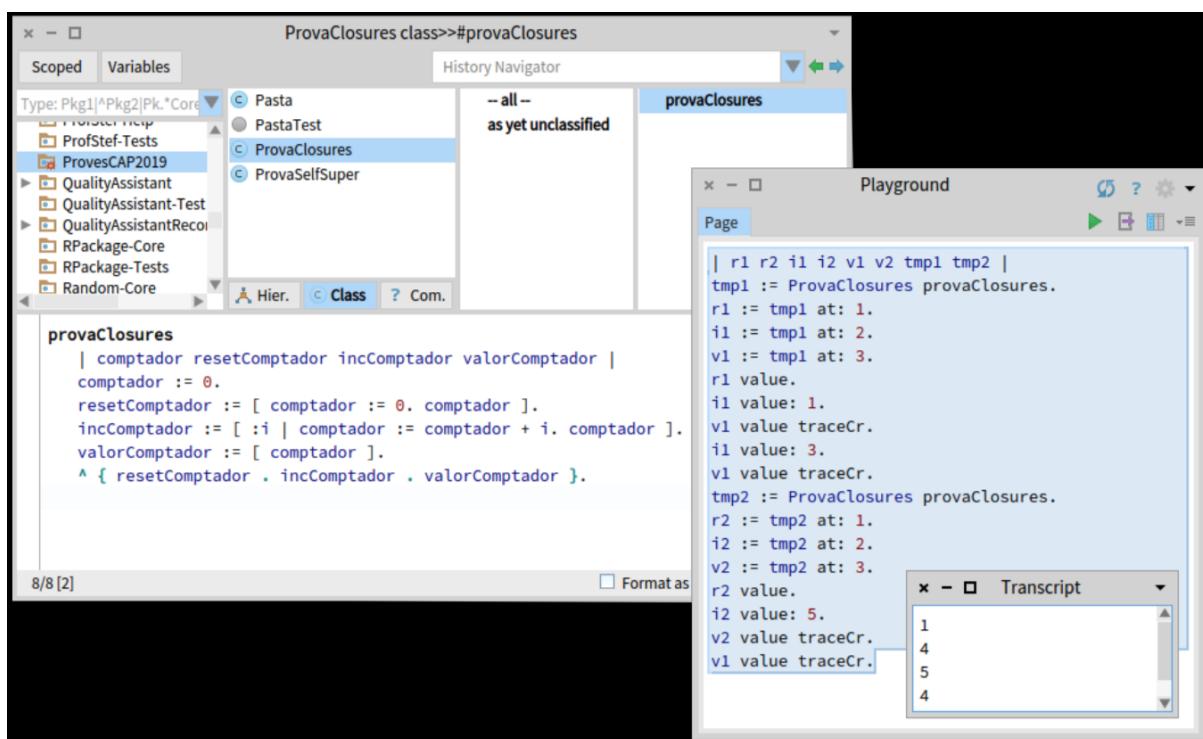
## ► 25

Recordem que un bloc retorna el valor de la darrera expressió (avaluada) del bloc.

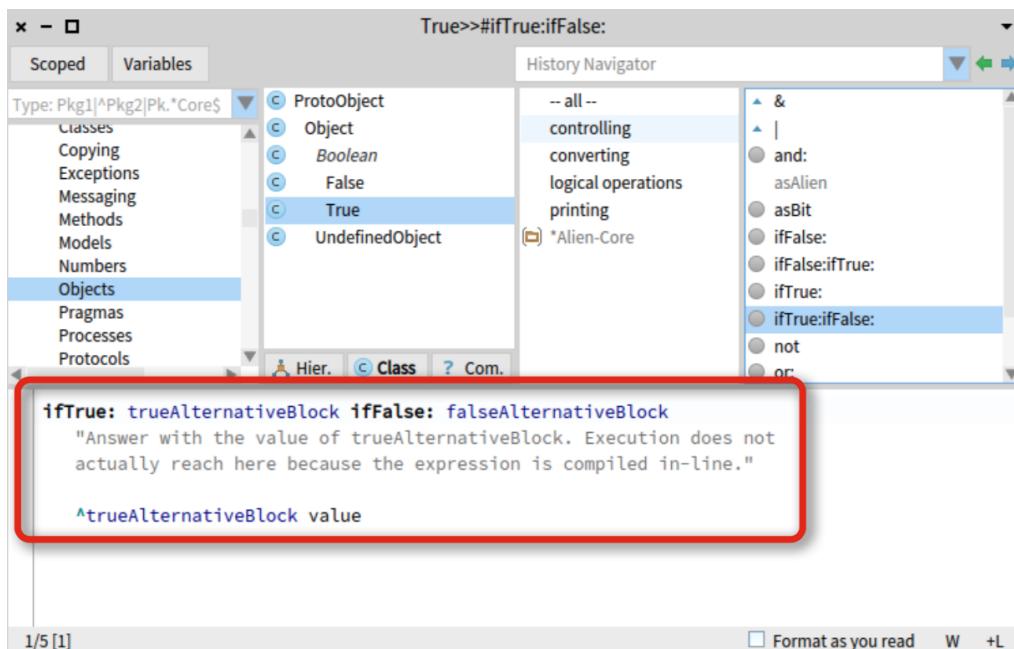
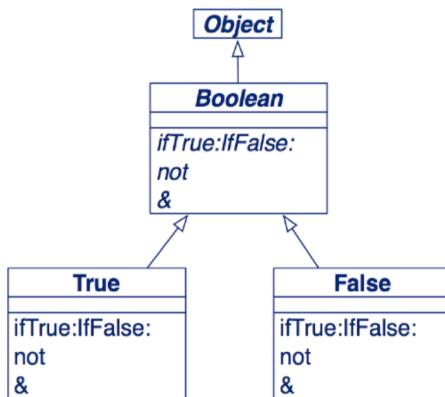
- Tenim fins a quatre `#value:value:value:value:`, per a més de quatre paràmetres cal utilitzar `#valueWithArguments:`, al que se li passa un `Array` amb els arguments.

Més exemples:

- `[2+3+4+5] value` ➔ 14
- `[:x | x+3+4+5] value:2` ➔ 14
- `[:x :y | x+y+4+5] value:2 value:3` ➔ 14
- `[:x :y :z | x+y+z+5] value:2 value:3 value:4` ➔ 14
- `[:x :y :z :w | x+y+z+w] value:2 value:3 value:4 value:5` ➔ 14



## Booleans

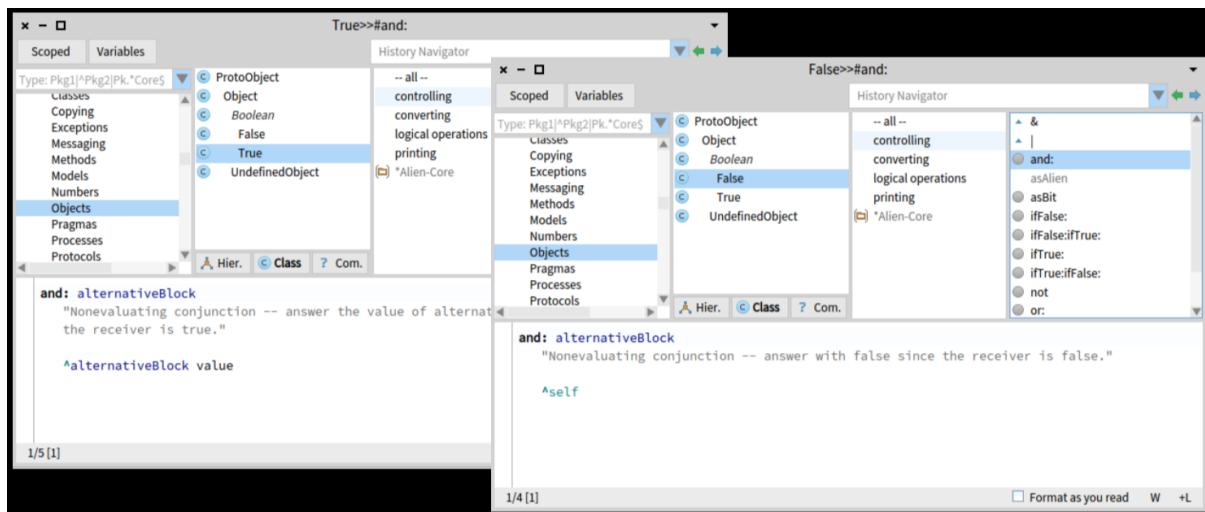


Com implementarieu `not`, `and:`, `or:`, `&`, etc... ?

`true` i `false`

`true` i `false` són les úniques instàncies (*singletons*) de les classes `True` i `False`, respectivament.

`and:` i `or:` no avaluen tots els seus arguments, depén del valor del booleà a qui enviem el missatge.



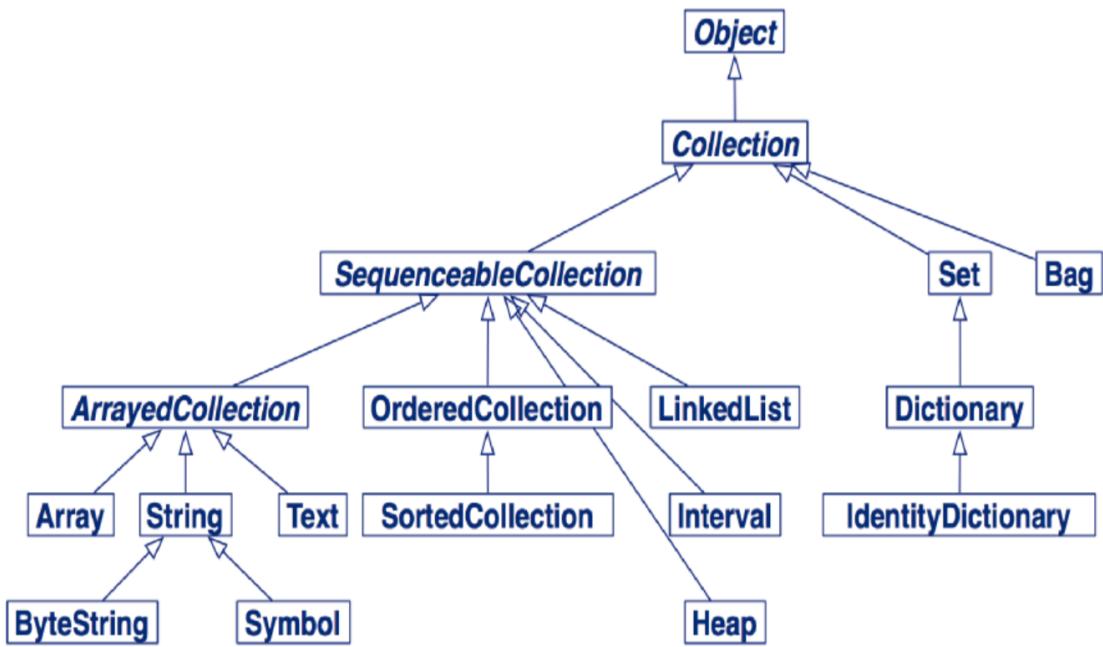
## Iteracions

Diversos tipus d'iteracions:

```
|n|
n := 1.
[ n <= 10 ] whileTrue:
    [ Transcript show: n asString; cr.
      n := n + 1 ]
1 to: 10 do: [ :n | Transcript show: n asString; cr]
(1 to: 10) do: [ :n | Transcript show: n asString; cr]
10 timesRepeat: [ Transcript show: '1'; cr]
```

Penseu, en cada cas, quin és l'objecte receptor?

## Collectors



- La jerarquia que penja de `Collection` ofereix moltes de les classes més útils d'un sistema Smalltalk. Un bon consell és, resistiu-vos de programar les vostres pròpies col.leccions! Segur que Smalltalk en té alguna que us farà el servei que us cal.
- Criteris de classificació:
  - **Accés:** indexat, seqüencial o basat en claus
  - **Mida:** fixada o dinàmica
  - **Tipus d'element:** fixat o arbitrari
  - **Ordre:** definible o cap
  - **Duplicats:** possibles o no
  - `Sequenceable` ordenada
  - `ArrayedCollection` mida fixada + índex enter
  - `Array` qualsevol mena d'element
  - `String` elements → caràcters
  - `IntegerArray` elements → enters
  - `Interval` progressió aritmètica
  - `LinkedList` encadenament dinàmic
  - `OrderedCollection` mida dinàmica + ordre d'arribada
  - `SortedCollection` ordre explícit

- `Bag` cap ordre + permet duplicitats
- `Set` cap ordre + no duplicitats
- `Dictionary` elements → associacions
- `IdentitySet` identificació basada en la identitat
- Accés → `#size` `#capacity` `#at:` `#at:put:`
- Test → `#isEmpty` `#includes:` `#contains:` `#occurrencesOf:`
- Afegir → `#add:` `#addAll:`
- Eliminar → `#remove:` `#remove:ifAbsent:` `#removeAll:`
- Enumerar → `#do:` `#collect:` `#select:` `#detect:` `#detect:ifNone:` `#reject:`  
`#inject:into:`
- Convertir → `#asBag` `#asSet` `#asOrderedCollection` `#asSortedCollection`  
`#assortedCollection:`
- Crear → `#with:` `#with:with:` `#with:with:with:` `#with:with:with:with:` `#withAll:`

Alguns exemples:

```

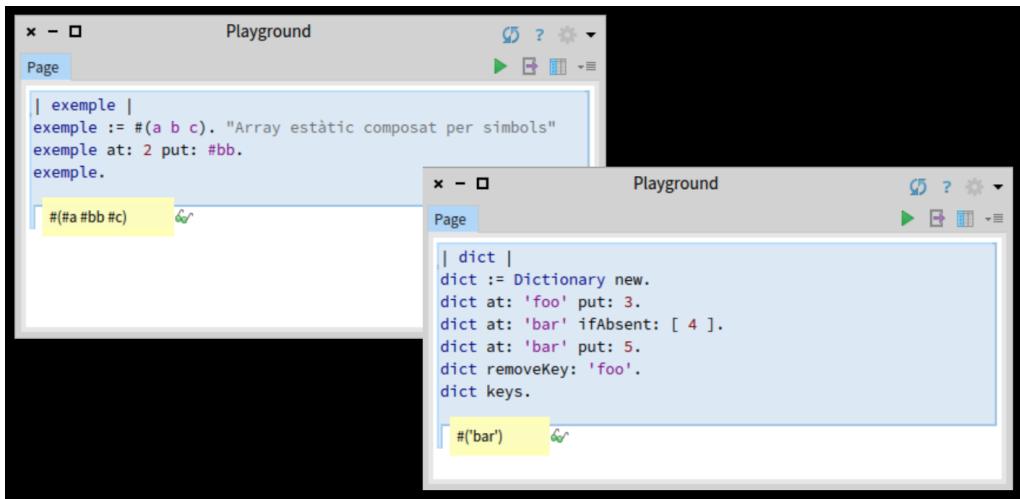
#(1 2 3 4) includes: 5 → false
#(1 2 3 4) size → 4
#(1 2 3 4) isEmpty → false
#(1 2 3 4) contains: [:some | some < 0] → false
#(1 2 3 4) do: [:each | Transcript show: each; cr] →
  Transcript
  1
  2
  3
  4

#(1 2 3 4) with: #(5 6 7 8)
  do: [:x :y | Transcript show: x+y; cr] →
  Transcript
  6
  8
  10
  12

#(1 2 3 4) select: [:each | each odd] → #(1 3)
#(1 2 3 4) reject: [:each | each odd] → #(2 4)
#(1 2 3 4) detect: [:each | each odd] → 1
#(1 2 3 4) collect: [:each | each even] → #(false true false true)
#(1 2 3 4) inject: 0
  into: [:sum :each | sum + each] → 10

```

Alguns missatges comuns:



## Snakes an Ladders

Per posar en context el que explicarem a partir d'ara prendrem com a exemple conductor el joc de [Snakes and Ladders](#)

The image shows a large block of Pharo code for the 'Snakes and Ladders' game. On the right side, there are two transcript windows. The top transcript shows the numbers 1, 2, 3, and 4. The bottom transcript shows the numbers 6, 8, 10, and 12, which are likely the results of various game calculations.

```

#(1 2 3 4) includes: 5 → false
#(1 2 3 4) size → 4
#(1 2 3 4) isEmpty → false
#(1 2 3 4) contains: [:some | some < 0] → false
#(1 2 3 4) do: [:each | Transcript show: each; cr] →
Transcript
1
2
3
4

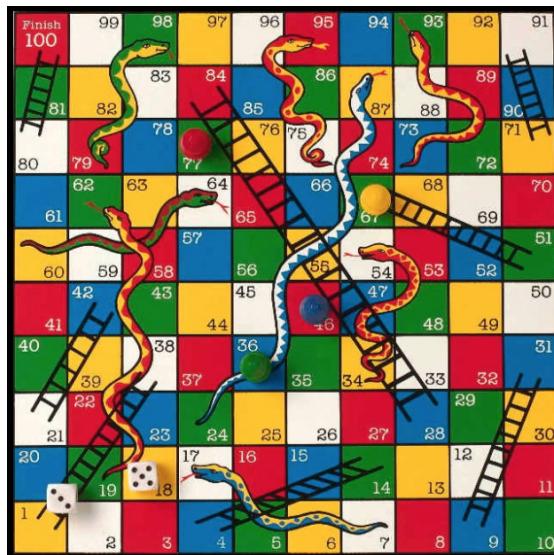
#(1 2 3 4) with: #(5 6 7 8)
    do: [:x :y | Transcript show: x+y; cr] →
Transcript
6
8
10
12

#(1 2 3 4) select: [:each | each odd] → #(1 3)
#(1 2 3 4) reject: [:each | each odd] → #(2 4)
#(1 2 3 4) detect: [:each | each odd] → 1
#(1 2 3 4) collect: [:each | each even] → #(false true false true)
#(1 2 3 4) inject: 0
    into: [:sum :each | sum + each] → 10

```

(Us passaré el fitxer `.st` pel Racó, només us caldrà fer `FileIn entire file`)

Aquest diagrama de classes mostra l'estructura del codi de Snakes and Ladders.



## Exemple: Cas d'ús

Screenshot of the Pharo Smalltalk IDE showing the code for the `example` method of the `SnakesAndLadders` class.

```

x - □
Scoped Variables
Type: Pkg1^Pkg2|Pk."Core$ History Navigator
Die
LoadedDie
GamePlayer
MetaclassHierarchyTest
SnakesAndLadders
SnakesAndLaddersTest
SnakesAndLaddersTest2
-- all --
documentation example
example2
example
"self example playToEnd"
^ (self new)
    add: FirstSquare new;
    add: (LadderSquare forward: 4);
    add: BoardSquare new;
    add: BoardSquare new;
    add: BoardSquare new;
    add: BoardSquare new;
    add: (LadderSquare forward: 2);
    add: BoardSquare new;
    add: BoardSquare new;
    add: BoardSquare new;
    add: (SnakeSquare back: 6);
    add: BoardSquare new;
    join: (GamePlayer named: 'Jack');
    join: (GamePlayer named: 'Jill');
yourself
1/19 [1]
Format as you read W +L

```

A yellow speech bubble points to the word `yourself` in the code, which is highlighted with a red rectangle.

*Veiem una gran cascada i yourself, que no sabem què és*

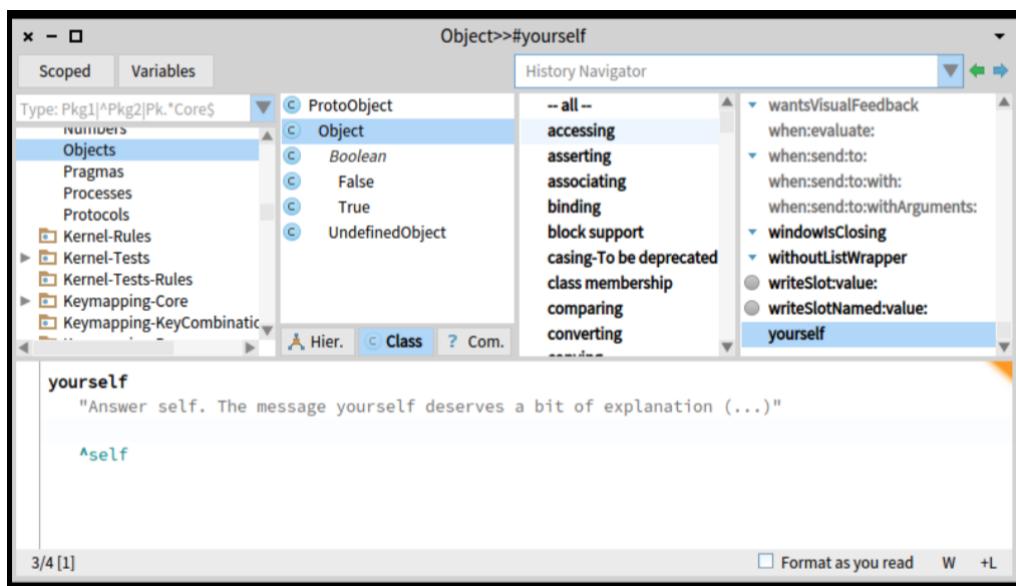
Quin format donem a múltiples missatges enviats al mateix receptor?

Utilitzeu una cascada. Separeu els missatges amb un `;`. Poseu cada missatge en una línia separada i sagneu un tabulador.

Utilitzeu cascades només per missatges amb un argument, com a molt.

Com podeu utilitzar el valor d'una cascada si el darrer missatge no retorna el receptor del missatge?

Acabeu la cascada amb el missatge `yourself`

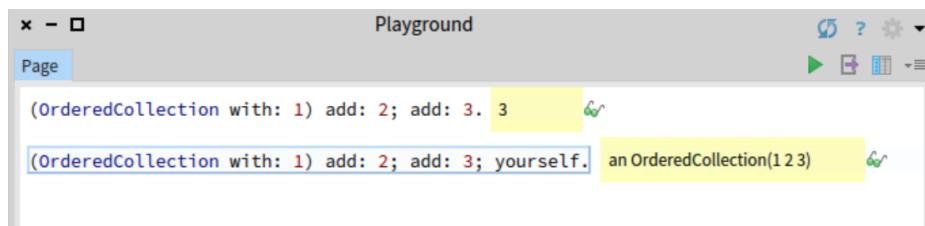


Sobre `yourself` ...

- L'efecte d'una cascada és enviar tots els missatges al receptor del primer missatge de la cascada:

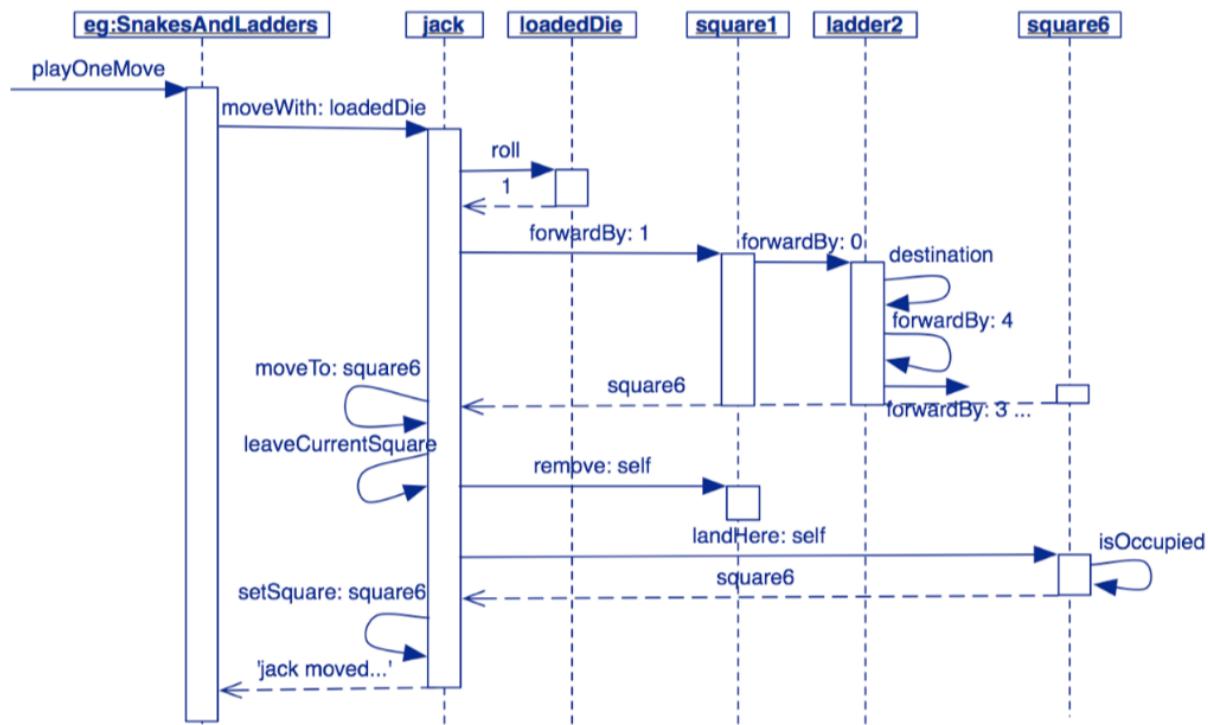
```
self new
add: FirstSquare new;
add: (LadderSquare forward: 4);
...
```

- El valor retornat per una cascada és el valor retornat pel darrer missatge enviat. Per aconseguir el receptor com a resultat de la cascada, hem d'enviar el missatge `yourself`.



## Distribuint responsabilitats...

En un sistema orientat a objectes ben dissenyat típicament trobarem molts mètodes petits, amb el nom triat amb cura. Això promou interfícies fluides, reutilitzables i amb facilitat de manteniment.



Un cop i només un

En un programa escrit en bon estil, tot el que cal dir es diu un sol cop.

Un munt de peces petites

El bon codi té invariablement mètodes petits i objectes petits. Només factoritzant el sistema en moltes peces petites d'estat i funció podem esperar satisfer la regla 'un cop i només un'

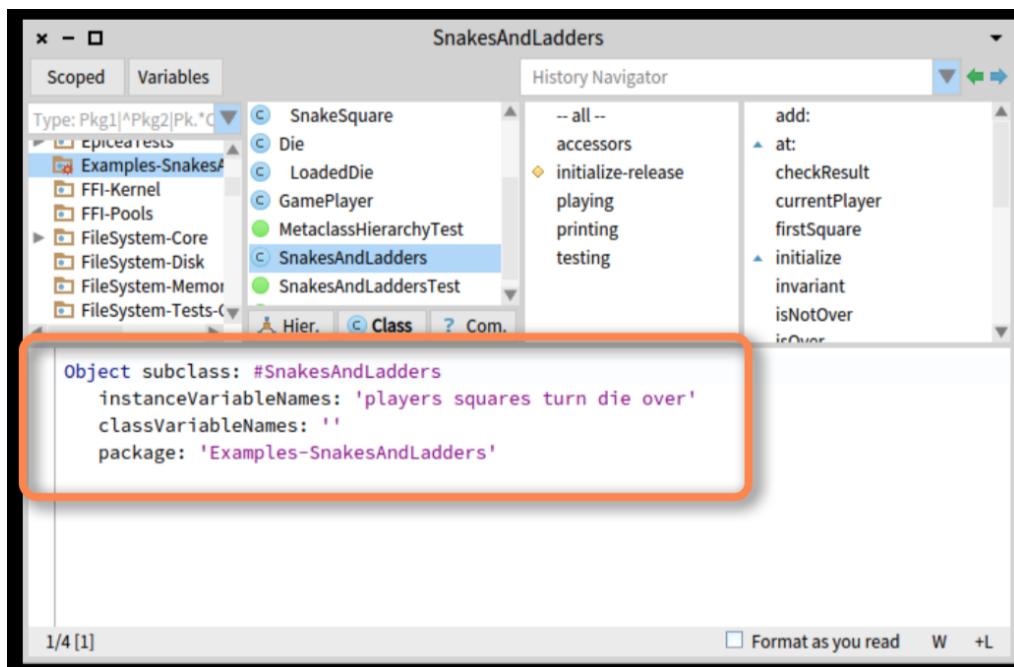
## Herència a Smalltalk

- Herència SIMPLE
- Estàtica per a les variables d'instància.
  - Les variables d'instància d'un objecte són aquelles variables definides en la classe de la que l'objecte és instància, a més de totes les variables

d'instància de les superclasses corresponents.

- Dinàmica pels mètodes.
  - Els mètodes corresponents als missatges enviats a un objecte (el receptor) es busquen en temps d'execució, en funció de la classe del receptor.

### Recordem que crear classes és només enviar missatges a altres classes



#subclass:instanceVariableNames:classVariableNames:package: és un missatge amb 4 paràmetres!

### Variables d'instància (amb nom)

- El seu nom comença amb una lletra minúscula.
- Cal declarar-les explícitament quan es crea la classe.
- El seu nom ha de ser únic dins la cadena d'erència
- El seu valor per defecte és nil
- Poden accedir-hi tots els mètodes de la classe i de les subclasses
- No hi poden accedir els mètodes de classe
- Els clients han d'utilitzar mètodes d'accés per consultar i/o modificar una variable d'instància.

**Consell de Disseny:** No accedir mai directament a les variables d'instància d'una superclasse des dels mètodes d'una subclasse. Així evitarem vincles forts entre classes.

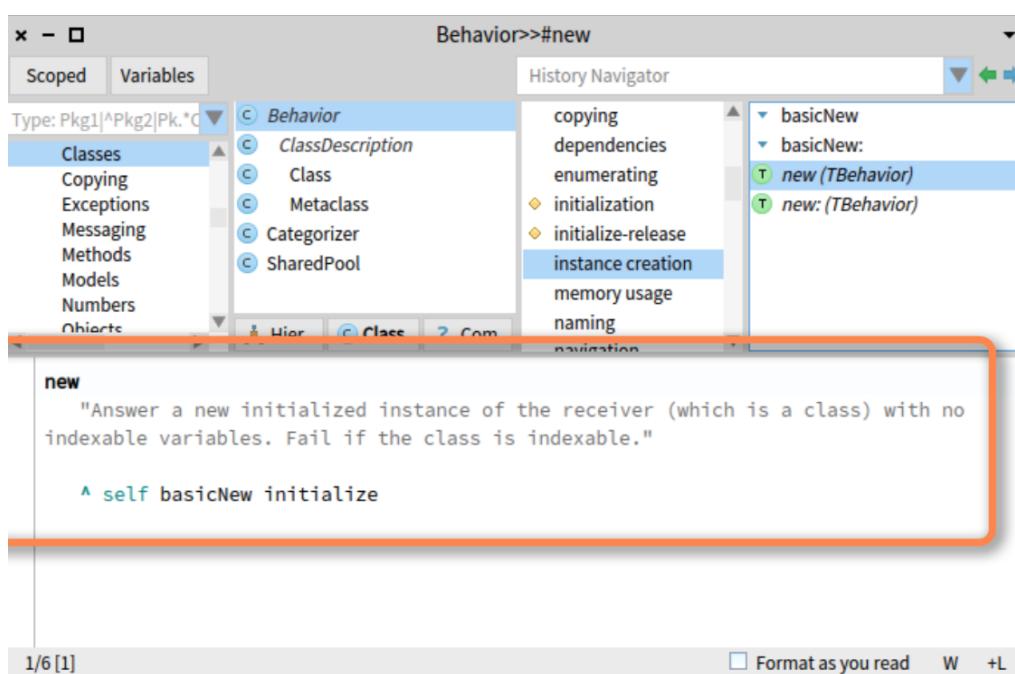
## Com inicialitzem objectes?

- **Problema:** Per crear una nova instància d'una classe hem d'enviar el missatge `new` a la classe. Els mètodes de classe, però, no poden accedir a les variables d'instància.

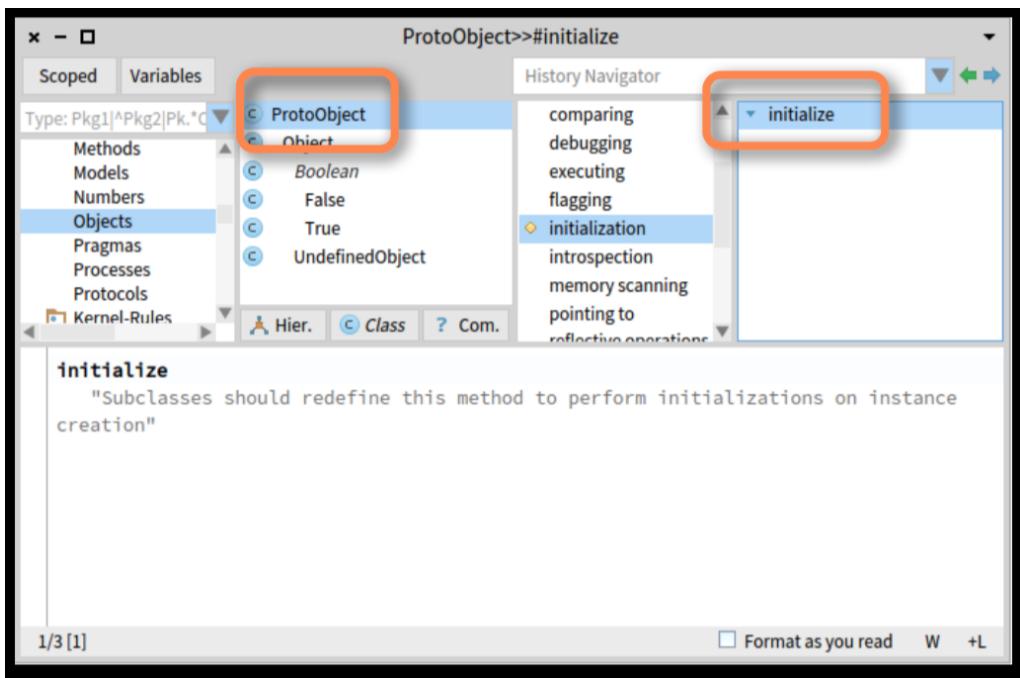
Com ho fem per inicialitzar els valors de les variables d'instància?

- **Solució:** Proporcionar un mètode (d'instància) anomenat `initialize` (s'aconsella utilitzar el protocol `initialize-release`) que inicialitzi les variables d'instància com calgui.

Aquest mètode sempre es crida en crear una classe amb `new`

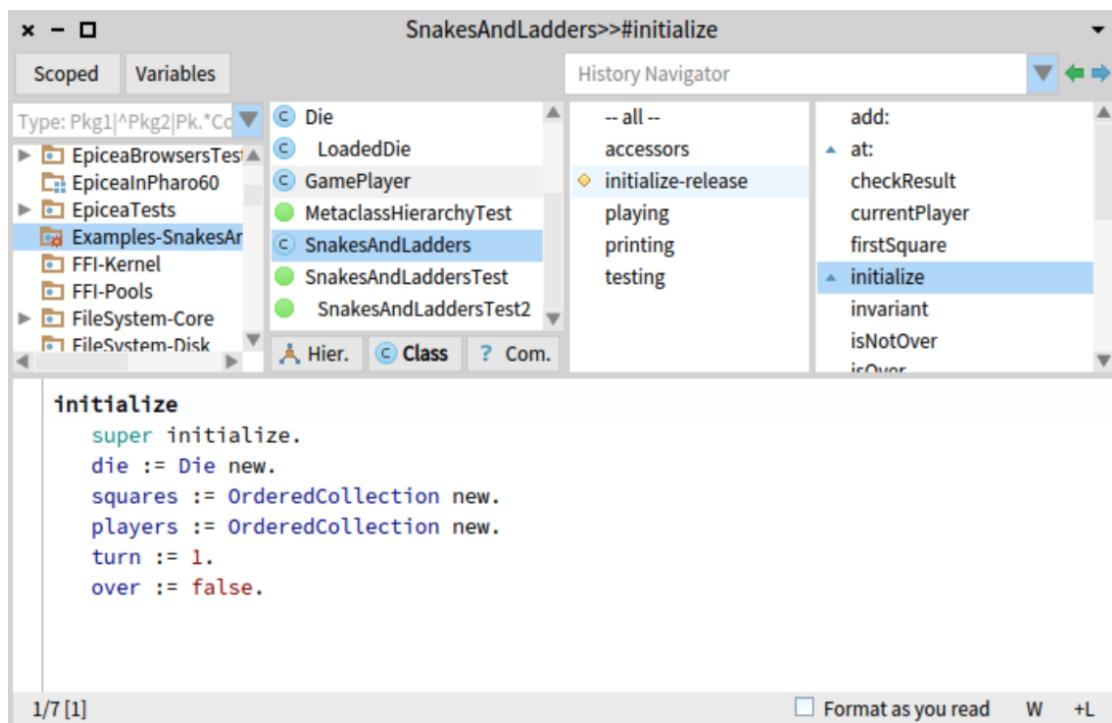


I si no en definim cap, de mètode `initialize`?



No passa res! Sempre en trobarà un a l'arrel de la jerarquia de classes.

### Exemple d' `initialize`



SnakesAndLadders>>#initialize

```

initialize
    super initialize.
    die := Die new.
    squares := OrderedCollection new.
    players := OrderedCollection new.
    turn := 1.
    over := false.

```

1/7 [1]

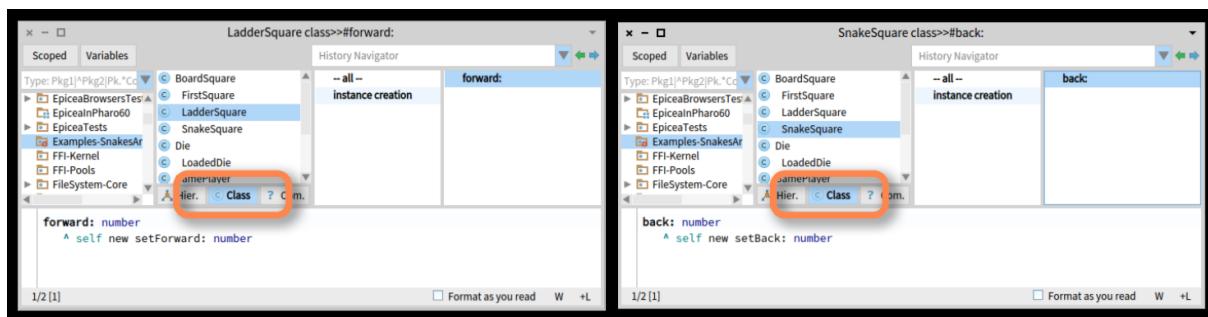
Format as you read W +L

Com podem utilitzar col·leccions la mida de les quals NO és coneguda en el moment de crear-les?

Utilitzeu `OrderedCollection` com la col·lecció de mida dinàmica per defecte.

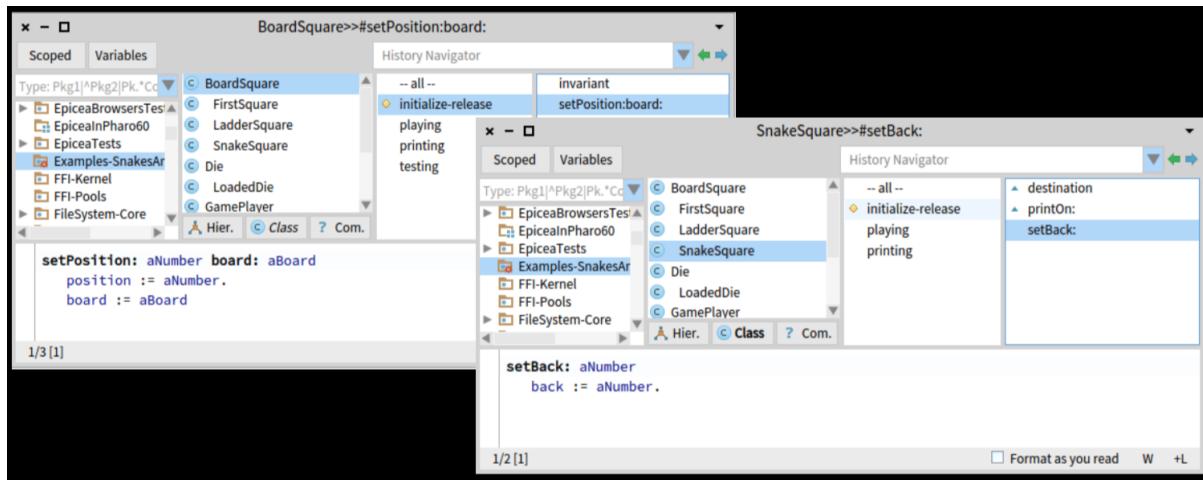
Com podem representar la creació d'instàncies?

Proporcioneu mètodes de classe, dins el protocol `instance creation`, que construeixin instàncies ben formades. Passeu tots els paràmetres que siguin necessaris.

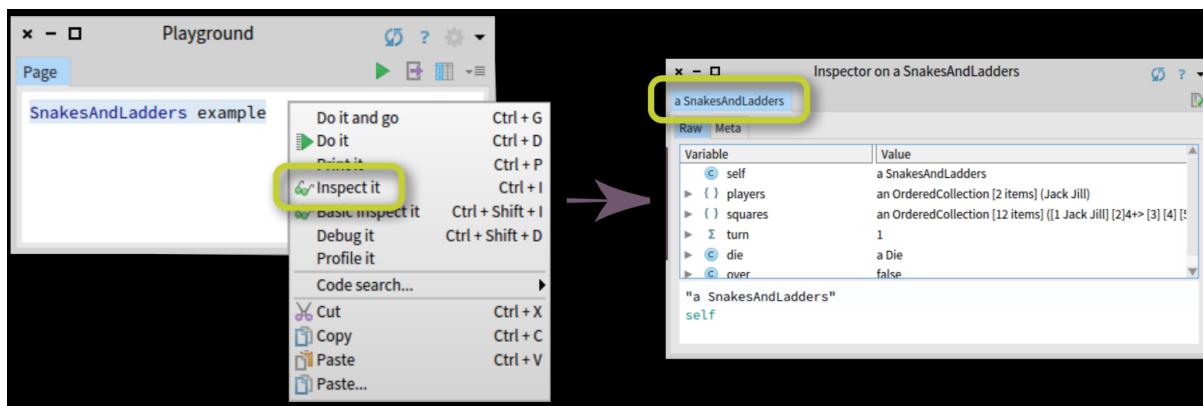


Com podeu donar valor a les variables d'instància a partir dels paràmetres del mètode constructor?

Proporcieu setters anomenats set i després el nom de la (les) variable(s)



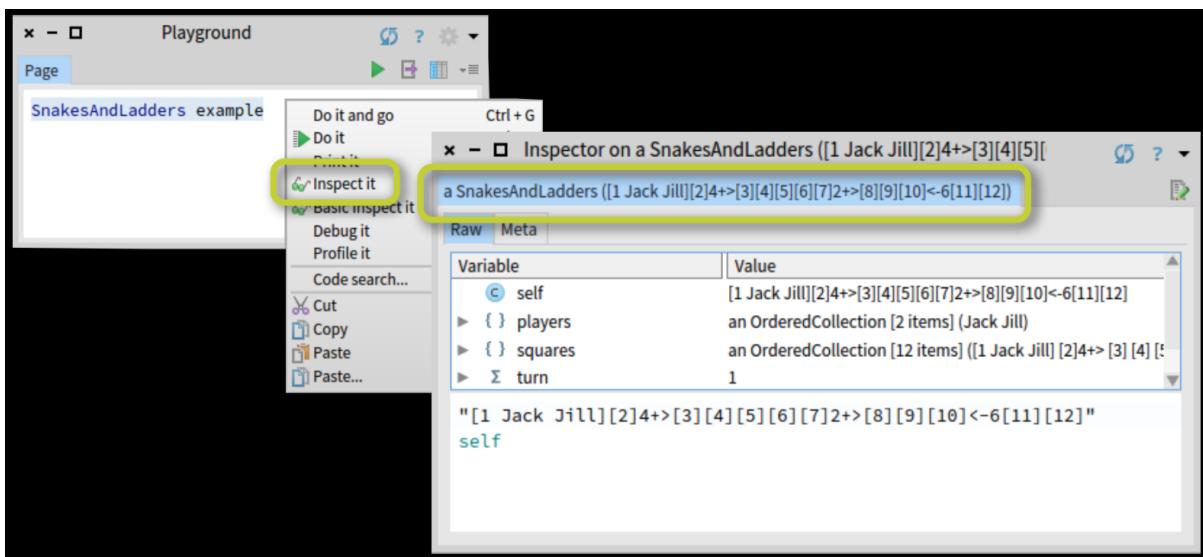
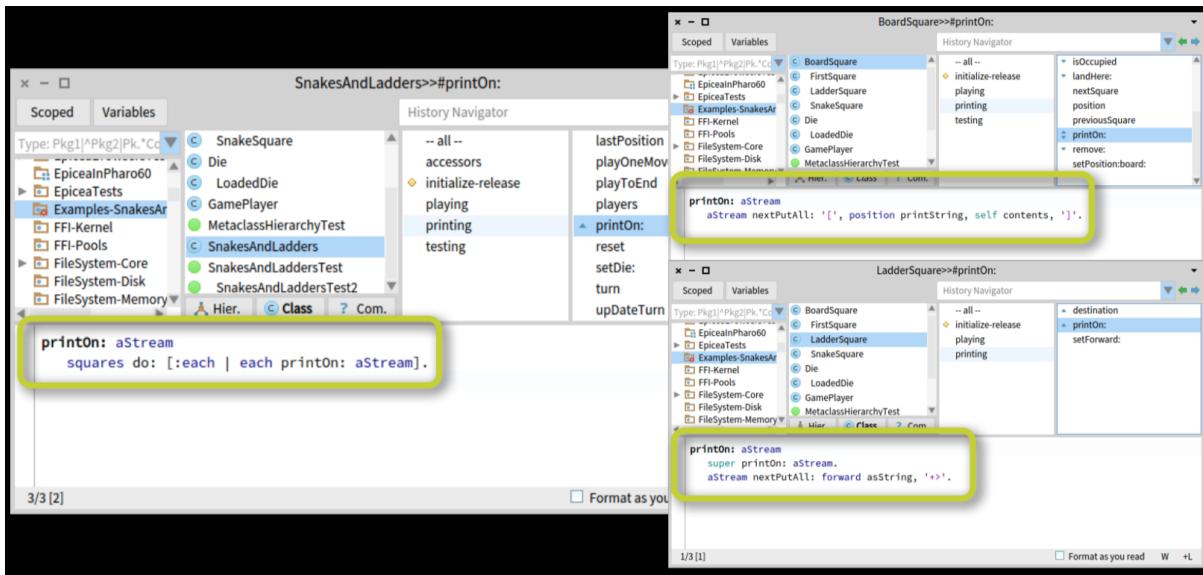
Com podem programar el mètode d'escriptura per defecte?



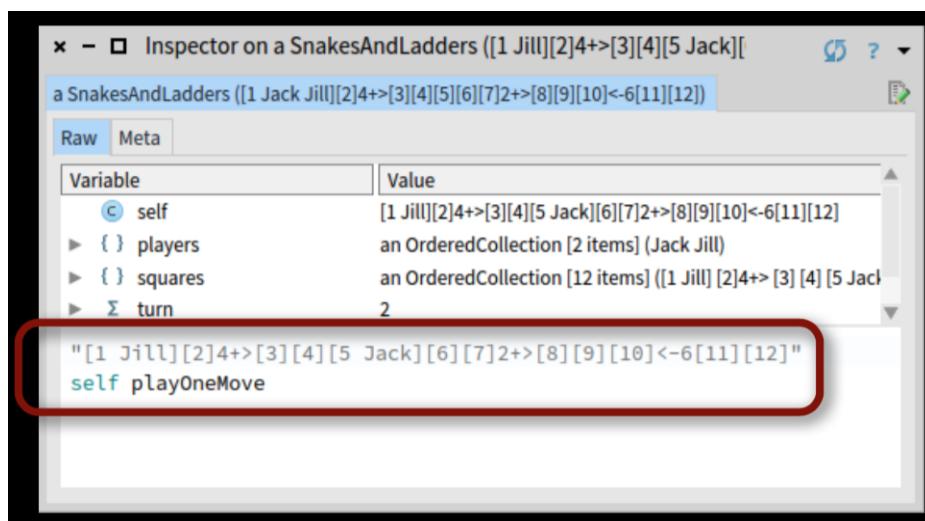
Vull coneixer l'estat del meu sistema, en aquest cas el joc que fem servir com a exemple. Puc inspeccionar-lo, però la representació textual per defecte de l'estat (a SnakesAndLadders) no és gaire informativa...

## Visualitzant l'estat

Cal sobreescriure el mètode `#printOn:`, que és el que fan servir totes les eines de l'entorn per escriure la representació textual.



Fins i tot podeu utilitzar l'inspector com a interficie pel joc...



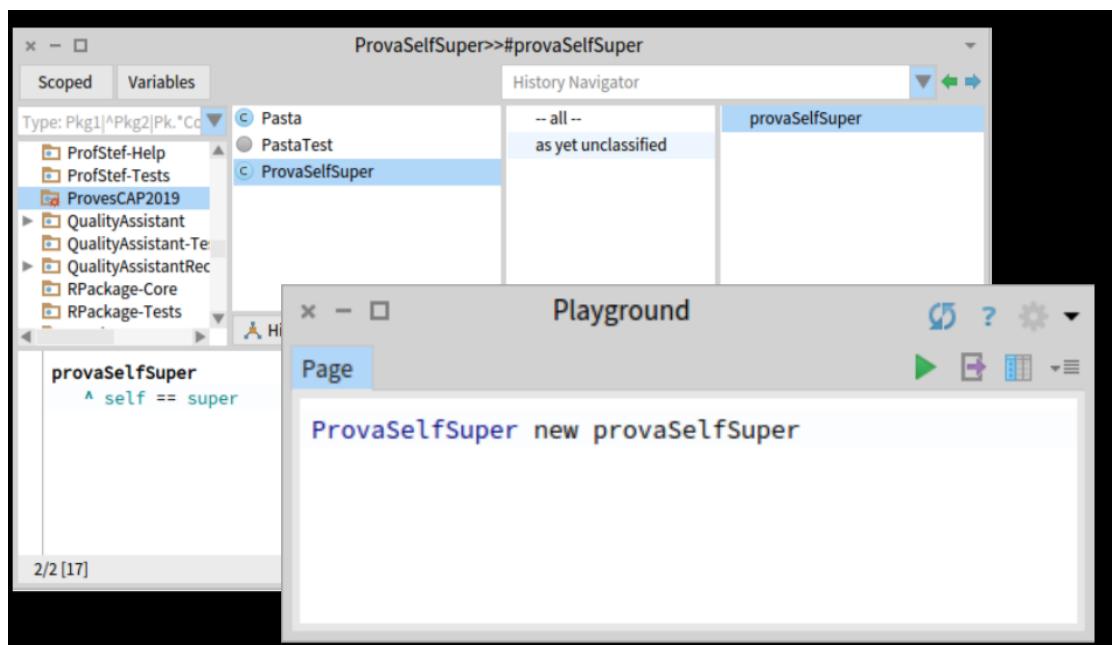
Com podem invocar comportament a les superclasses?

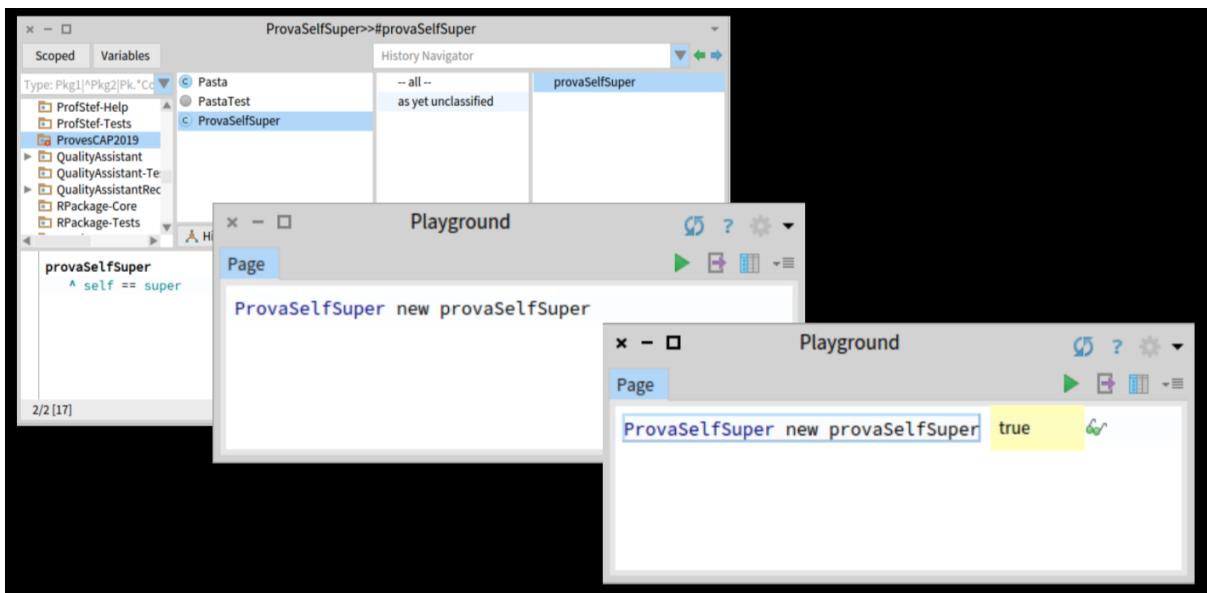
Invoqueu codi de la superclass explícitament enviant un missatge a super (i no a self)

Com funciona? `super` conté una referència a l'objecte receptor del missatge corresponent al mètode que s'està executant (igual que `self`!), però...

la cerca del mètode corresponent al missatge enviat a super canvia

Comença a buscar aquest mètode a la superclass de la classe que implementa el mètode que s'està executant.

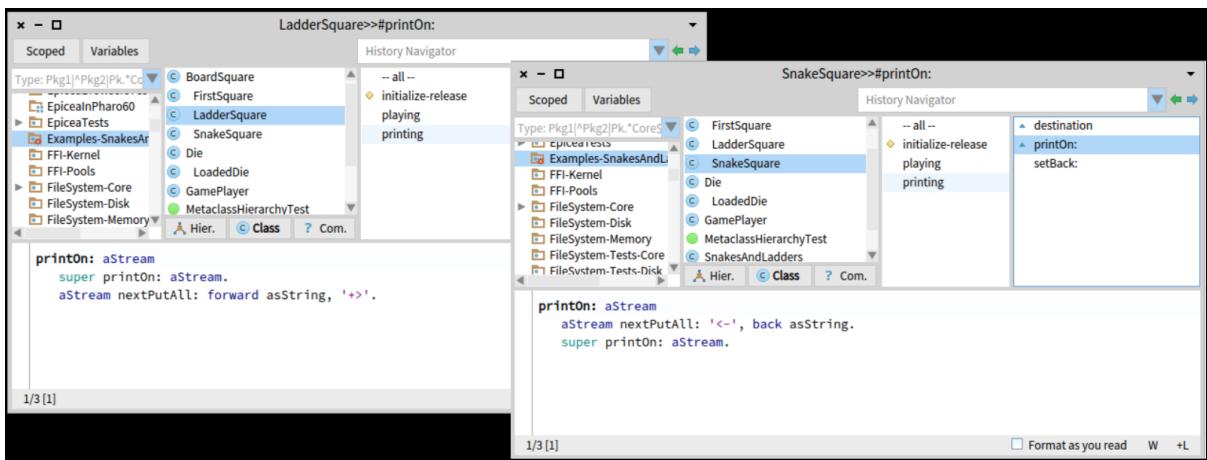




Cal utilitzar super amb cura. Canvieu super per self si això no modifica l'execució del codi...

Com podem afegir funcionalitat a la implementació d'un mètode heretat?

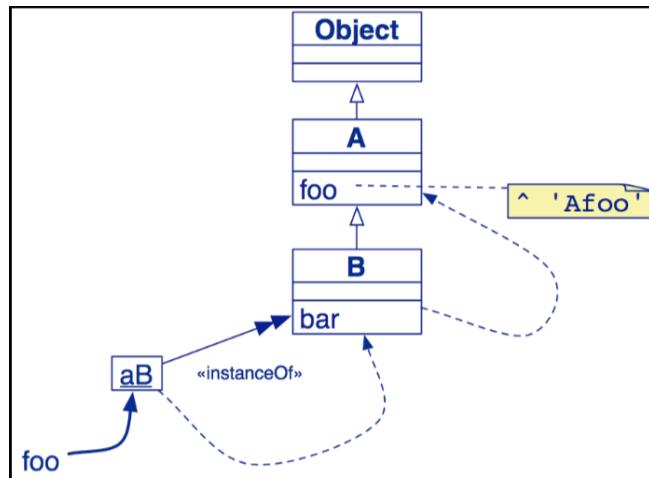
Sobreescriu el mètode i envieu un missatge a super en el mètode que sobreescriuvi



## Cerca normal de mètodes

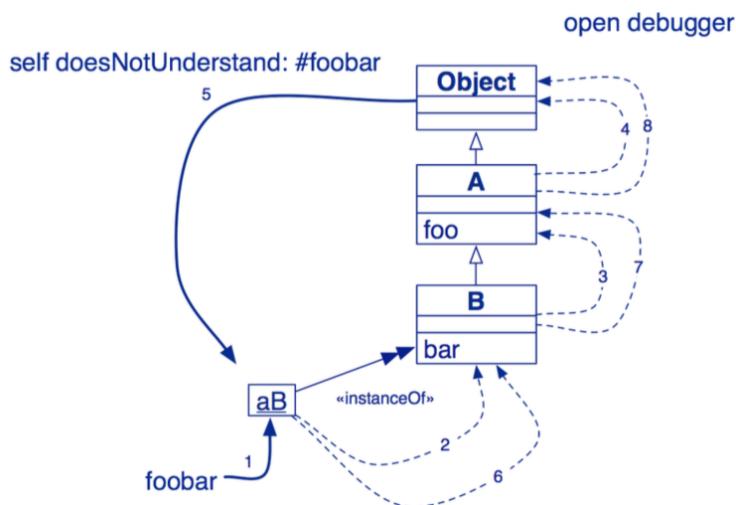
La cerca comença a la classe de l'objecte receptor del missatge:

1. Si el mètode és al diccionari de mètodes, s'utilitza.
2. Si no hi és, la cerca continua a la superclasse.



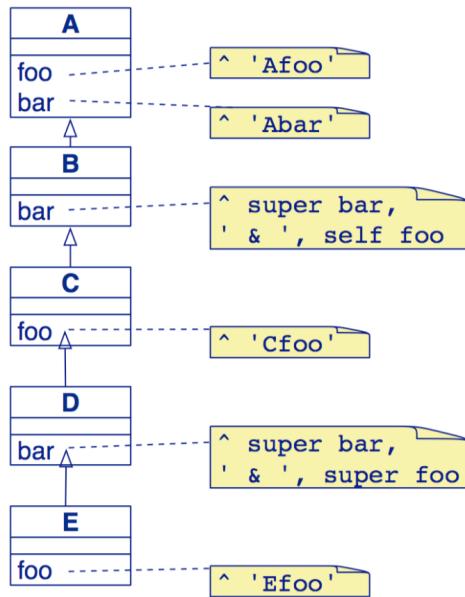
Si no es troba cap mètode això provoca un error.

Quan la cerca falla, s'envia un missatge d'error `#doesNotUnderstand:` a l'objecte i la cerca torna a començar amb aquest missatge...



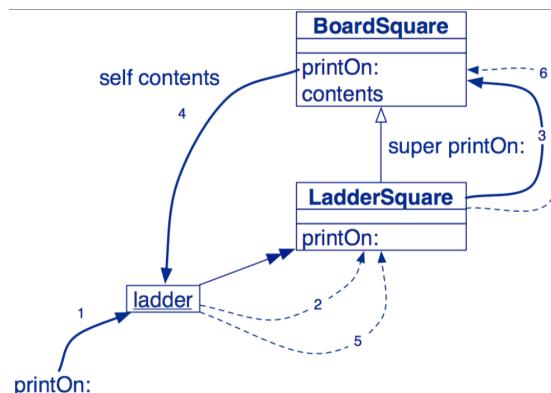
La implementació per defecte de `#doesNotUnderstand:` pot ser sobreescrita per qualsevol classe.

- `A new bar` → `'Abar'`
- `B new bar` → `'Abar & Afoo'`
- `C new bar` → `'Abar & Cfoo'`
- `D new bar` → `'Abar & Cfoo & Cfoo'` `E new bar` → `'Abar & Efoo & Cfoo'`



Els enviaments a `self` sempre es resolen dinàmicament.

Els enviaments a `super` es resolen estàticament.



Com podem dividir el nostre programa en mètodes?

Dividiu el vostre programa en mètodes que realitzen una tasca identifiable

Mantenui totes les operacions dins d'un mètode al mateix nivell d'`abstracció`.

Això farà que el vostre programa estigui compost de `molts mètodes petits, cada un d'ells de poques línies de codi`.

La majoria dels mètodes seran petits i `auto-documentats`

Algunes excepcions: algorismes complexos, scripts de configuració, tests, etc.

```

playOneMove
| result |
self assert: self invariant.
^ self isOver
ifTrue: ['The game is over!']
iffalse:
[result := (self currentPlayer moveWith: die), self checkResult.
self updateTurn.
result]

```

## Invariants

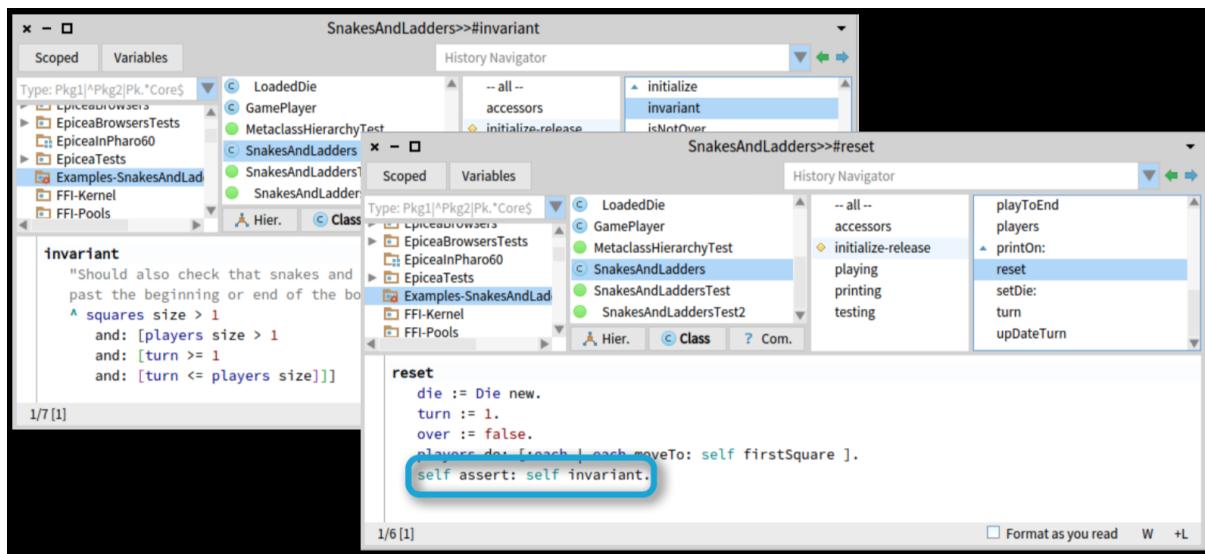
Podem fer servir `#assert:` per comprovar que l'estat dins d'un mètode, abans o després d'alguna acció, sigui el que s'espera.

```

roll: aNumber
self assert: ((1 to: 6) includes: aNumber).
roll := aNumber.

```

També podem establir un invariant de classe per comprovar-lo abans o després d'alterar l'estat de l'objecte.

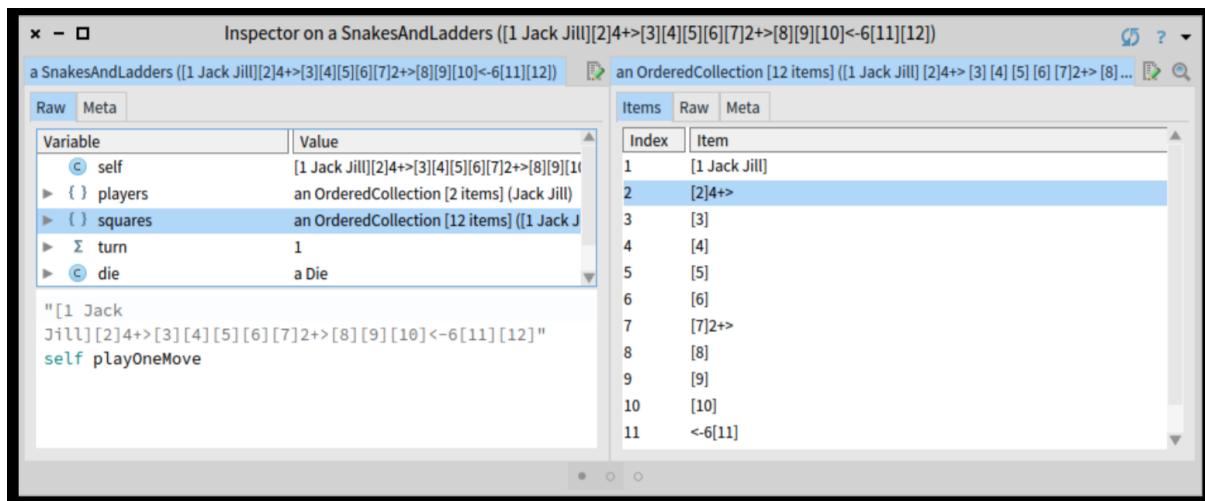


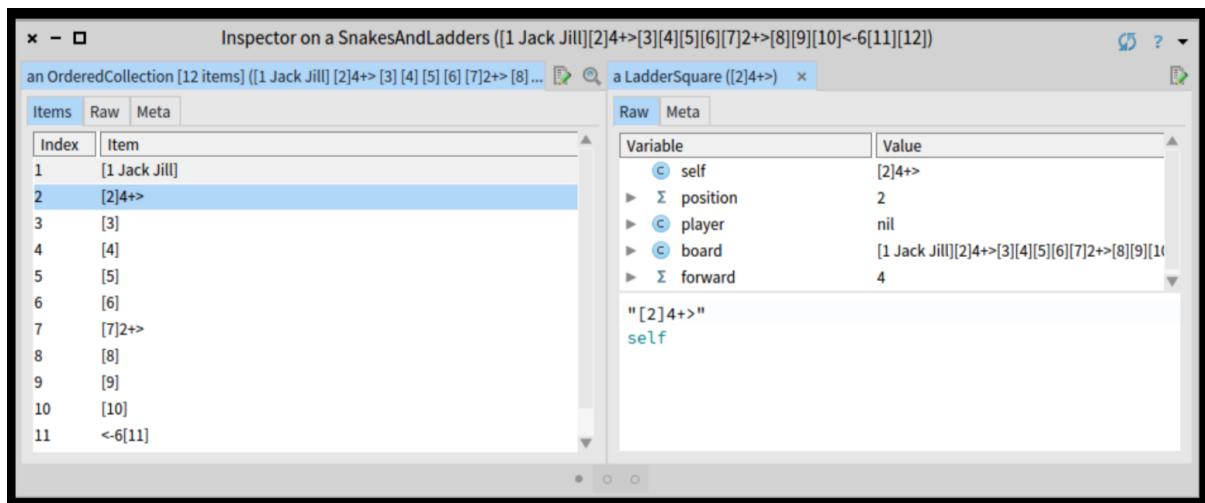
Com comentem els mètodes?

Comuniqueu informació important que no sigui òbvia a partir del codi. Poseu els comentaris al començament del mètode. **ConSELL:** Els comentaris poden fer pudor de codi disfressat! Proveu de refactoritzar el codi i canviar els noms dels mètodes per poder prescindir dels comentaris.

## Debugging: L'inspector

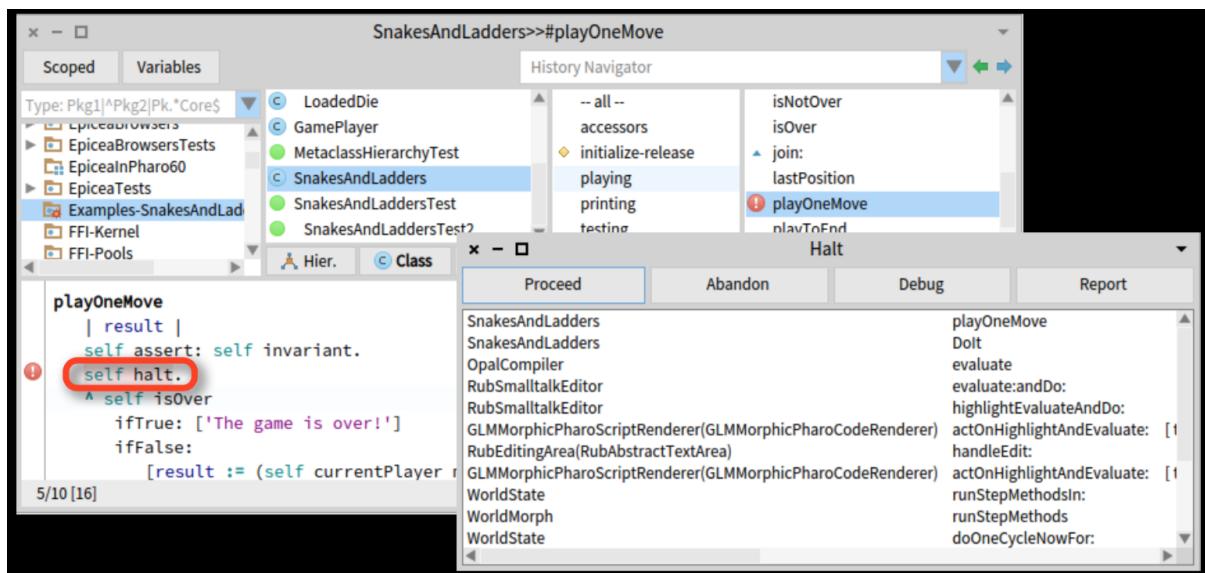
Podeu inspeccionar tot allò que vulgueu: Qualsevol expressió, la representació textual, interaccionar amb qualsevol objecte, inspeccionar variables d'instància, navegar pel sistema...





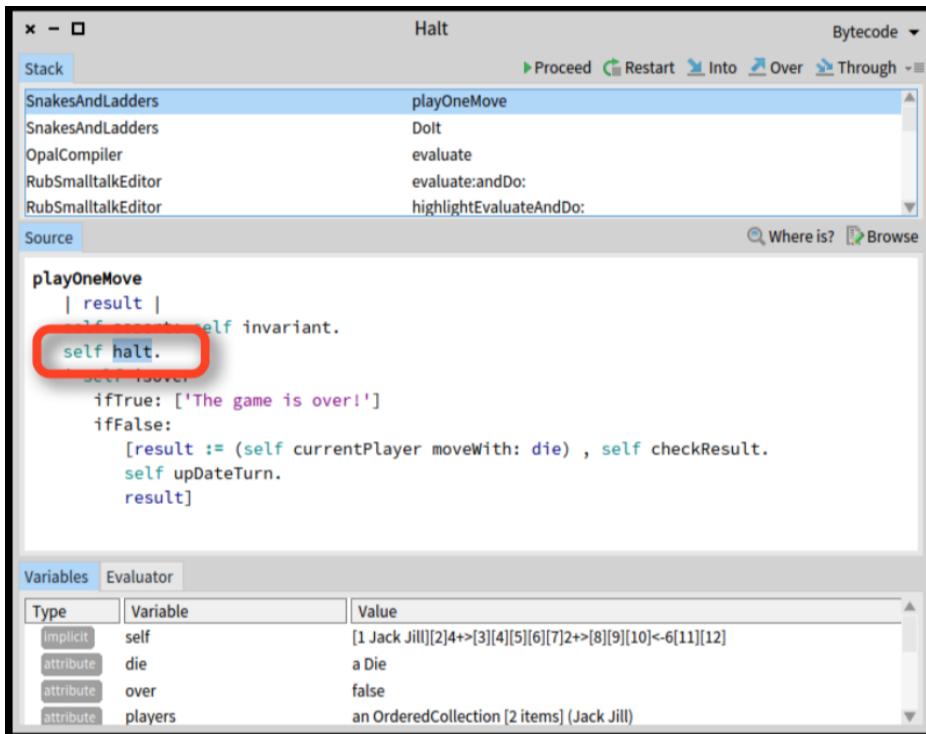
## Debugging: Breakpoints

Podeu enviar el missatge `self halt` per iniciar el debugger en qualsevol moment.



## Debugging

Amb el debugger puc executar els enviaments de missatges en un sol pas (`over`) o anar a veure els detalls de l'execució del mètode corresponent (`into`), i moltes altres coses també... EXPLOREU!



Apeneu a jugar amb el sistema. Exploreu els mètodes de la classe `SystemNavigation` i trobeu informació sobre el sistema enviant missatges a `SystemNavigation default`.

