

## 2. Smalltalk — a reflective language

Oscar Nierstrasz



# Birds-eye view



**Less is More** — simple syntax and semantics uniformly applied can lead to an expressive and flexible system, not an impoverished one.



# Roadmap



- > Smalltalk Basics
- > Demo: modeling Call Graphs

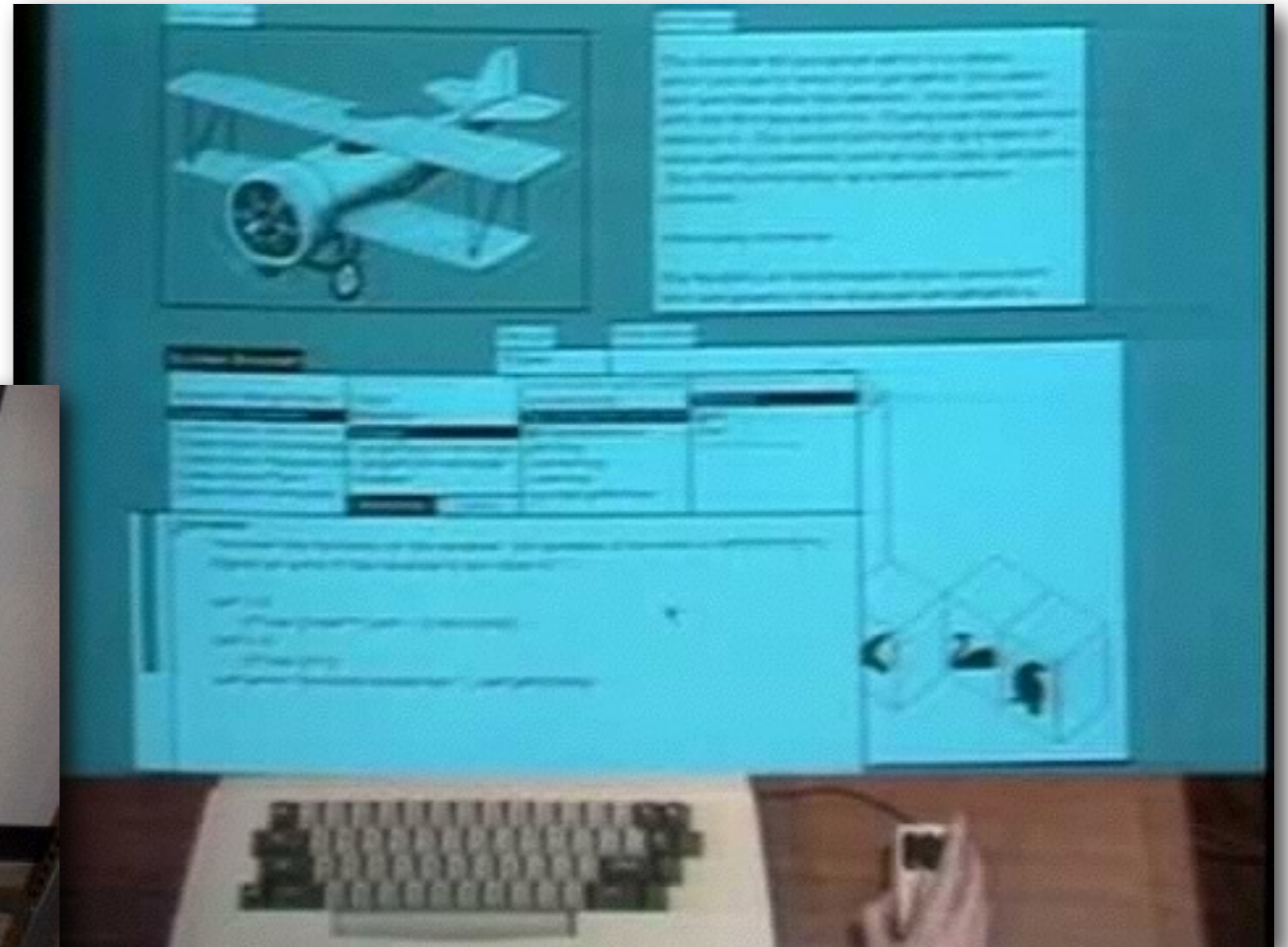
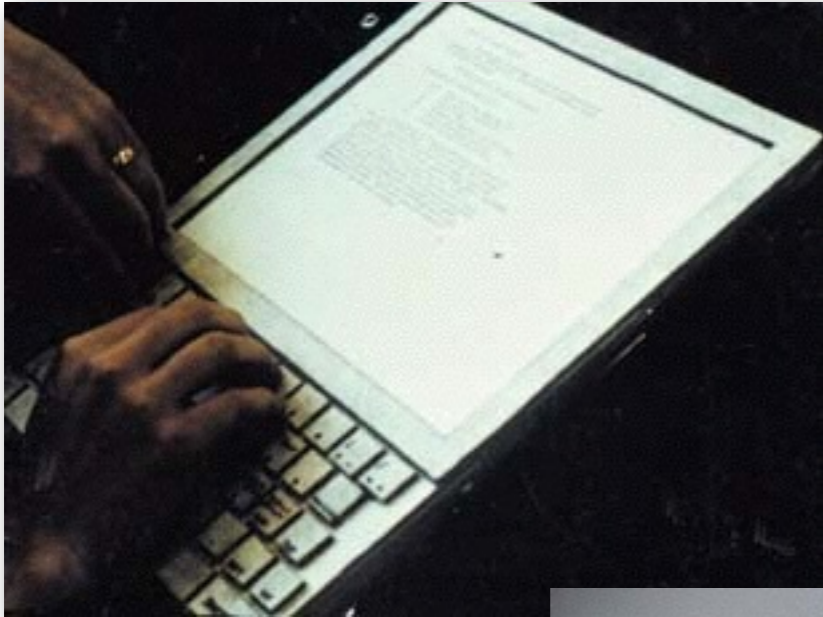
# Roadmap



- > **Smalltalk Basics**
- > Demo: modeling Call Graphs

# The origins of Smalltalk

Dynabook project (1968)



Alto — Xerox PARC (1973)

# Don't panic!

New Smalltalkers often think they need to understand all the details of a thing before they can use it.

Try to answer the question

***“How does this work?”***

with

***“I don't care”.***

— Alan Knight. Smalltalk Guru

# Two things to remember ...

**Everything is an object**



**Everything happens by  
sending messages**

# The Smalltalk object model

- > **Every object is an instance of one class**
  - ... which is also an object
  - Single inheritance
- > **Dynamic binding**
  - All variables are dynamically typed and bound
- > **State is private to objects**
  - “Protected” for subclasses
  - Encapsulation boundary is the object, not the class!
- > **Methods are public**
  - “private” methods by convention only

# Smalltalk Syntax

*Every expression is a message send*

> Unary messages

```
5 factorial  
Transcript cr
```

> Binary messages

```
3 + 4  
'hi', ' there'
```

> Keyword messages

```
Transcript show: 'hello world'  
2 raisedTo: 32  
'hello' at: 1 put: $y
```

# Precedence

*First unary, then binary, then keyword:*

2 raisedTo: 1 + 3 factorial

128

*Same as:*

2 raisedTo: (1 + (3 factorial))

*Use parentheses to force order:*

1 + 2 \* 3

1 + (2 \* 3)

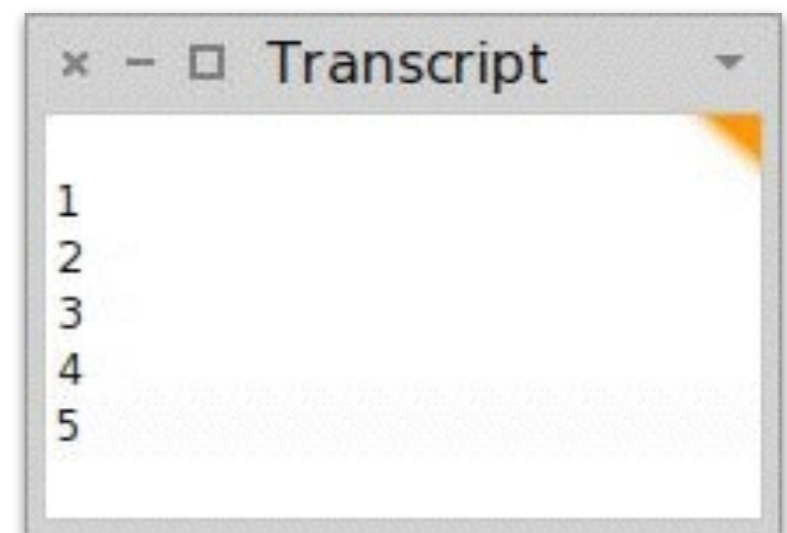
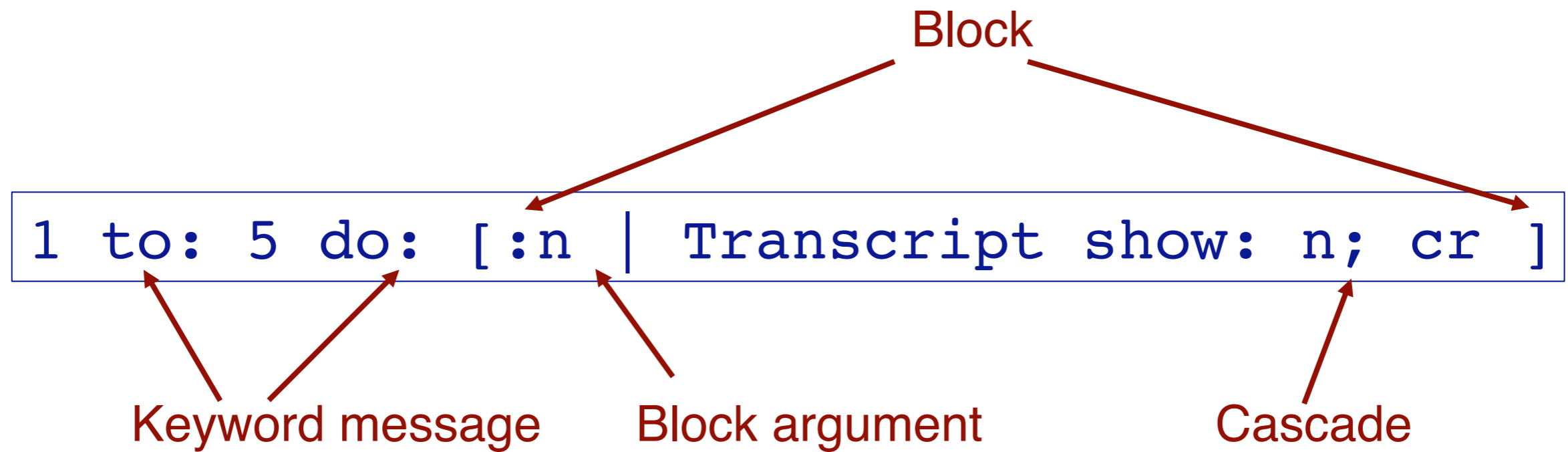
9 (!)

7

# Literals and constants

|                                 |             |
|---------------------------------|-------------|
| <i>Strings &amp; Characters</i> | 'hello' \$a |
| <i>Numbers</i>                  | 1 3.14159   |
| <i>Symbols</i>                  | #yadayada   |
| <i>Arrays</i>                   | #( 1 2 3 )  |
| <i>Pseudo-variables</i>         | self super  |
| <i>Constants</i>                | true false  |

# Blocks

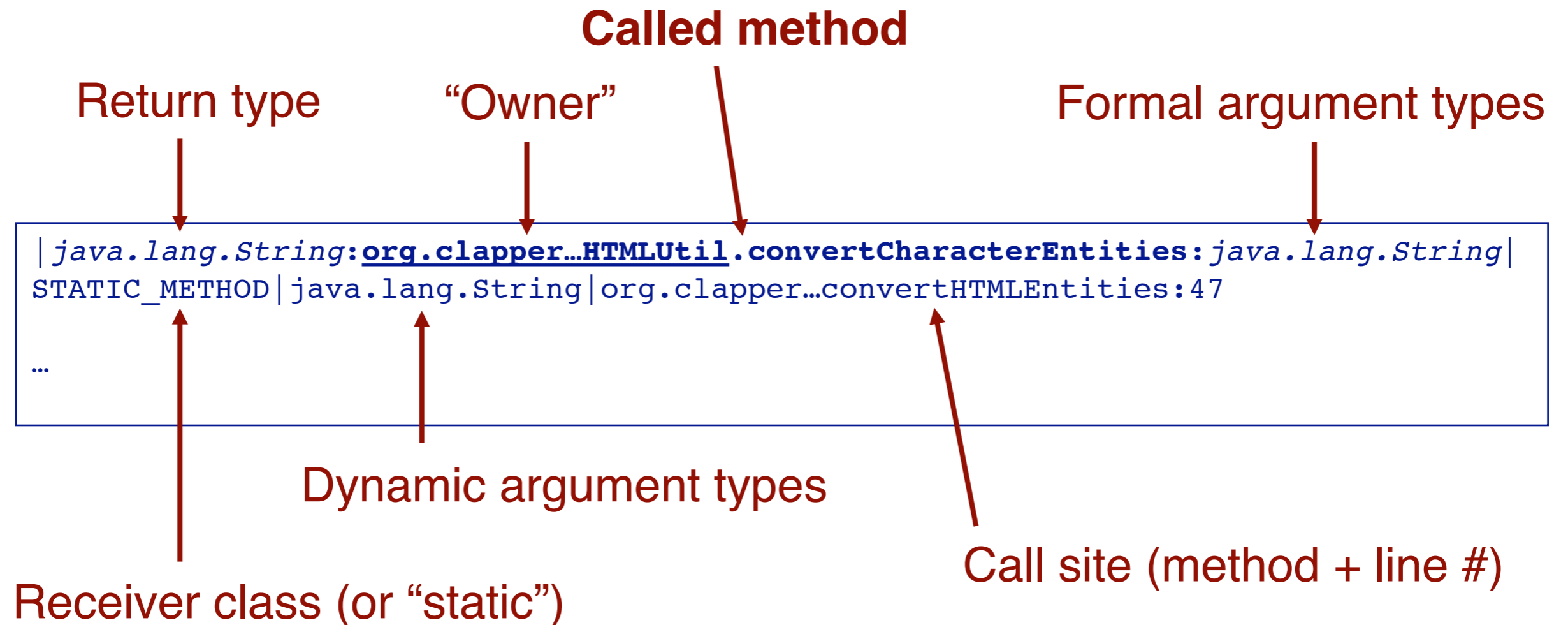


# Roadmap



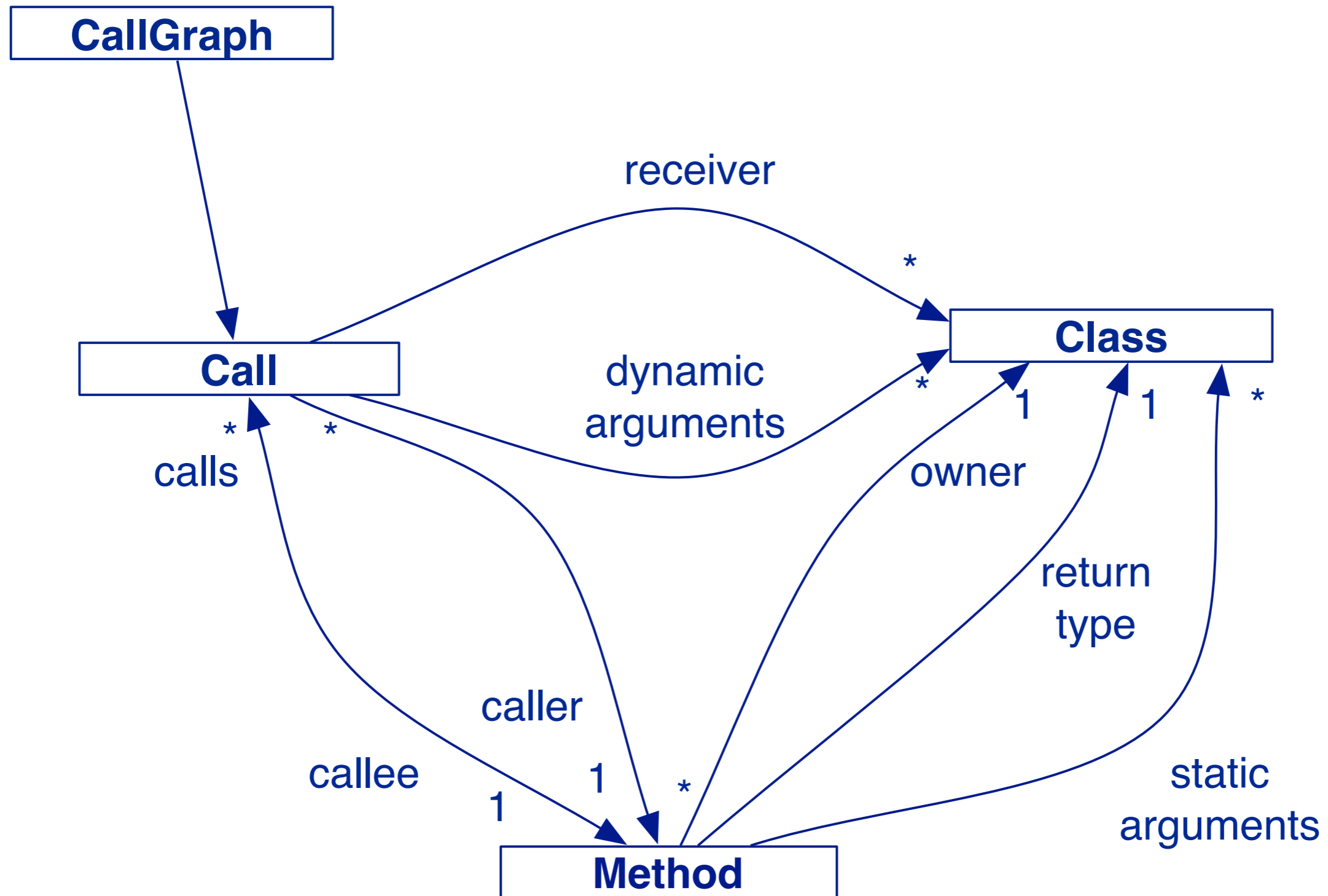
- > Smalltalk Basics
- > **Demo: modeling Call Graphs**

# Task: analyze call graph logs





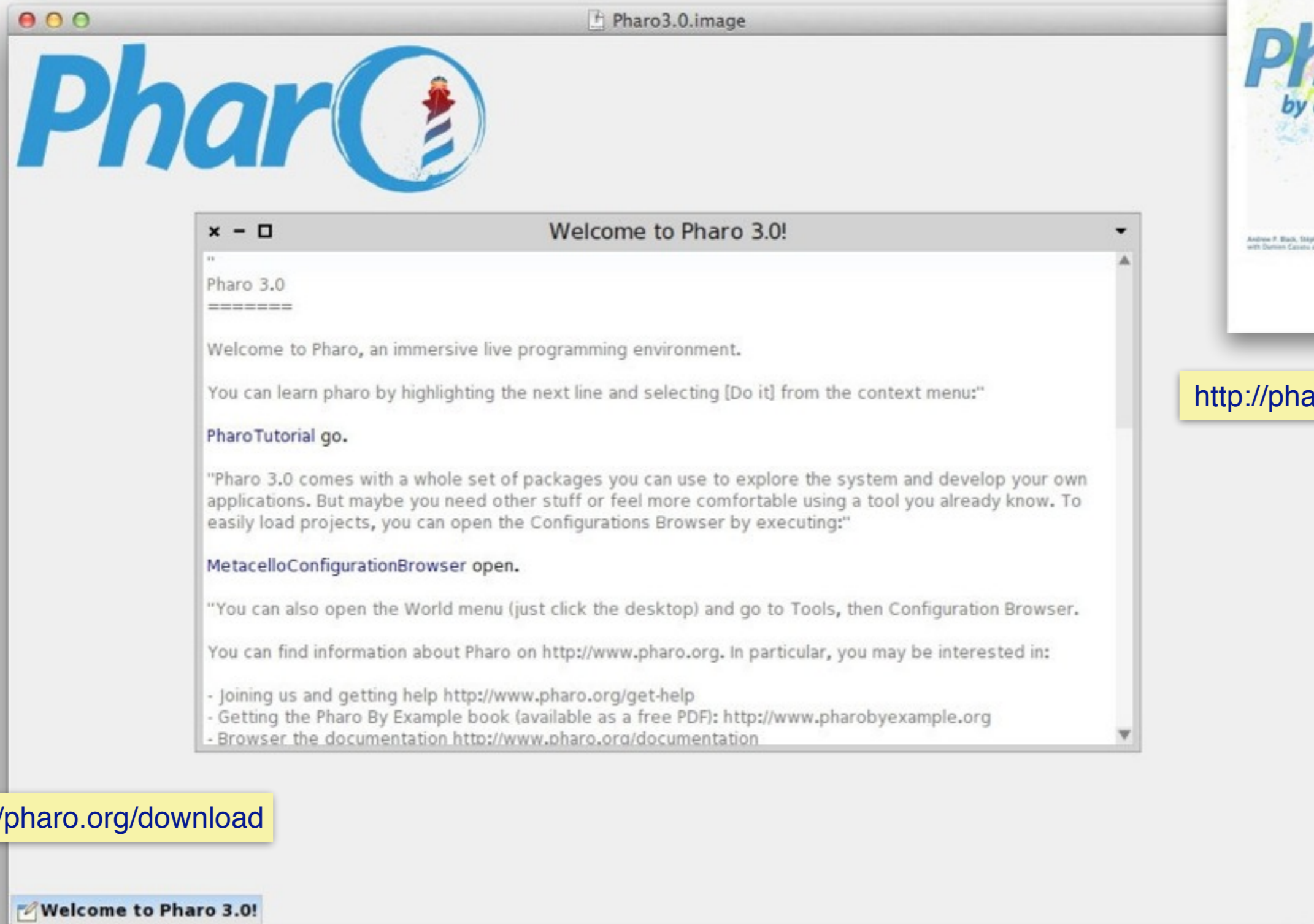
# How to reconstruct the model from the log?



# Questions of interest

- > How many calls are there?
- > How many methods are called?
- > How many classes are accessed?
- > Which methods are static?
- > Which methods are called most frequently?
- > What is the depth of the call graph?
- > Which methods are called by more than one caller?
- > Which methods are potentially polymorphic? (multiple receivers/implementations)
- > What are the polymorphic call sites? (methods called with different receiver/argument types)
- > ...

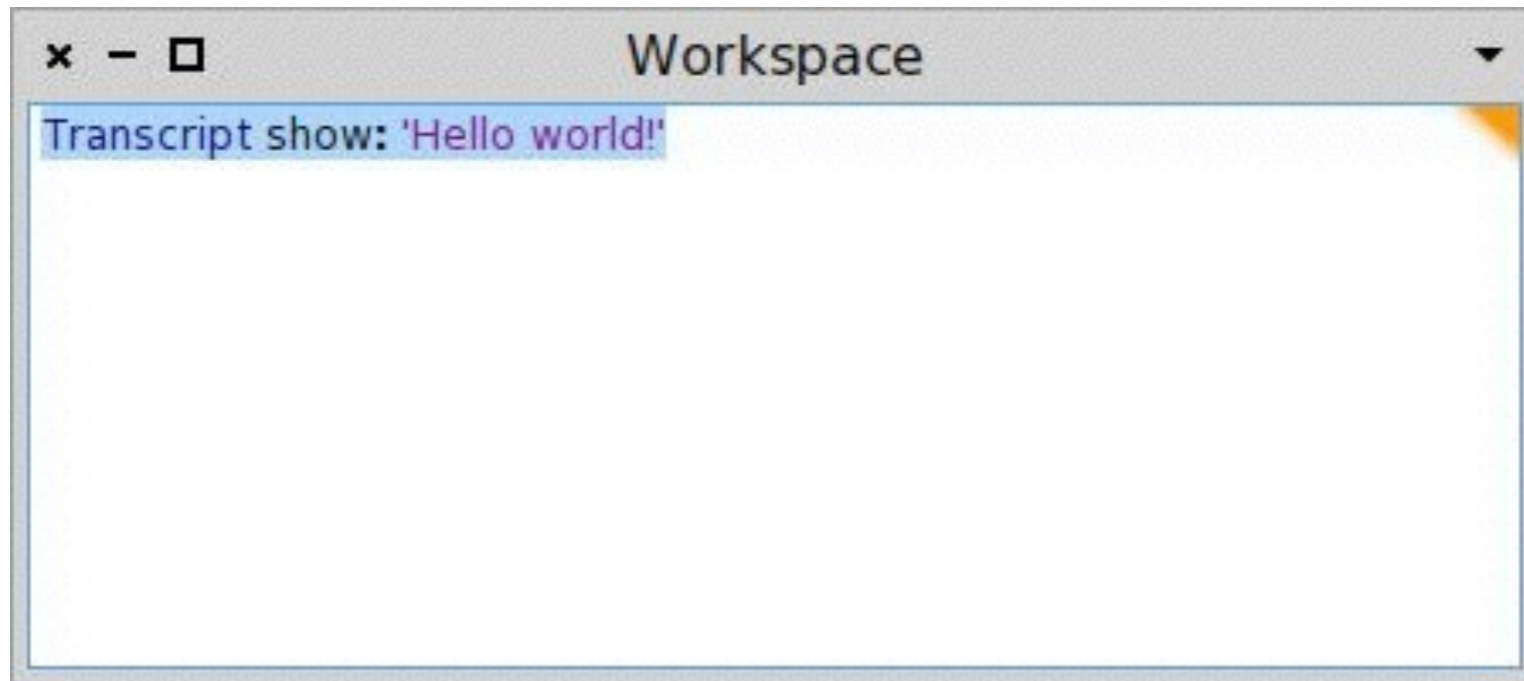
# Pharo — a modern Smalltalk



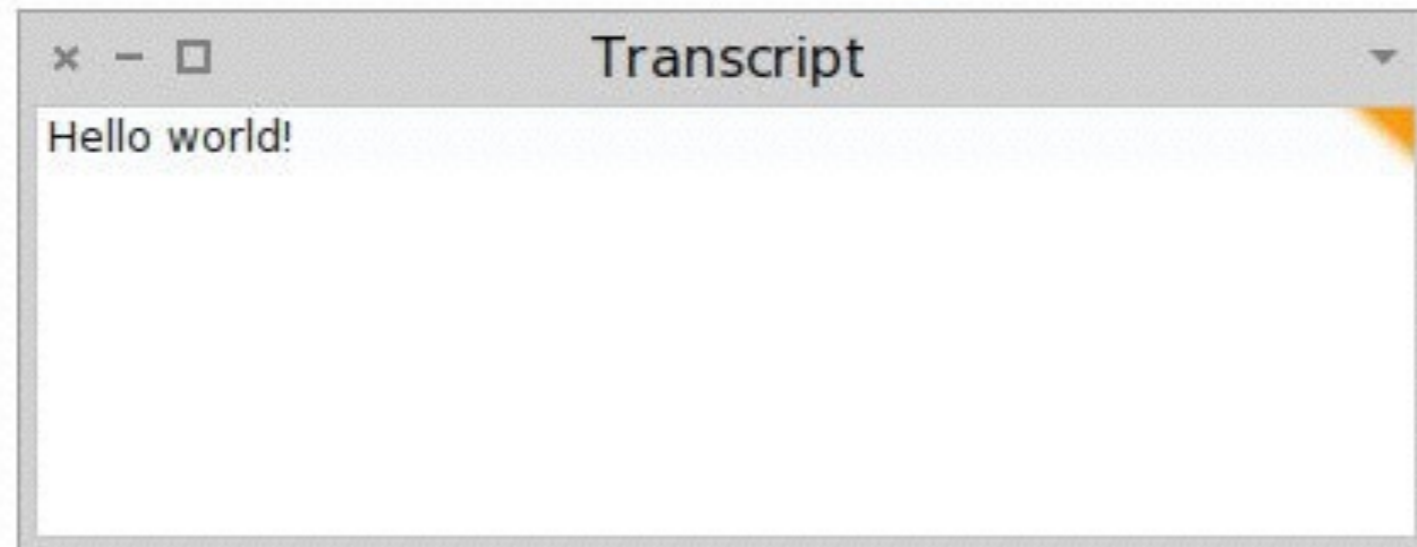
<http://pharobyexample.org/>

<http://pharo.org/download>

# The Workspace and the Transcript



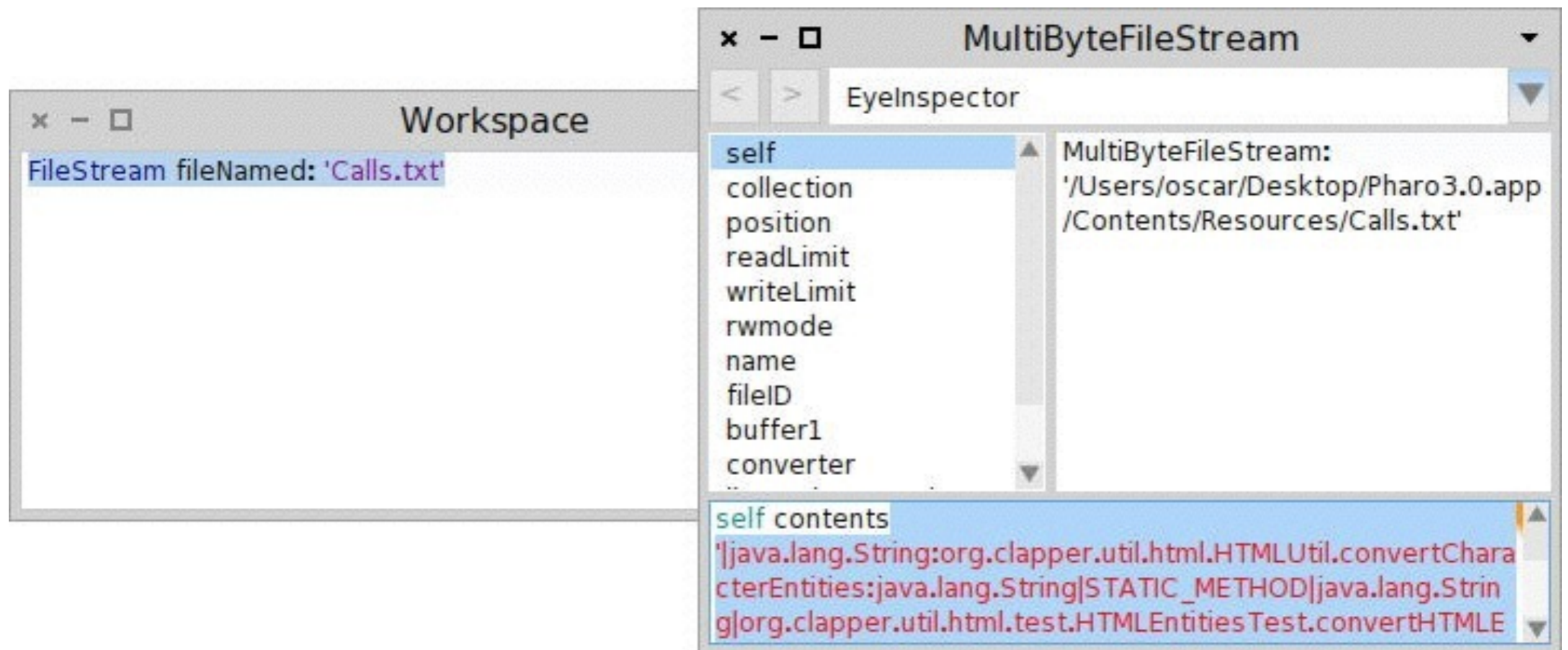
The Workspace is a place to evaluate arbitrary Smalltalk expressions



The Transcript is a place to print diagnostic messages

# Accessing a file from a Workspace

We can open a FileStream object on the Calls.txt file and extract its contents using an *Inspector*



*We should encapsulate this data in a ClassGraph object*

# Navigating to “implementors” or “senders”

“Categories”

“Protocols”

Methods

Classes

The screenshot displays an IDE interface with the following components:

- Workspace:** Shows a file named 'Calls.txt'.
- Class Browser:** Lists various packages and classes. The 'Kernel' package is selected, and the 'FileStream' class is highlighted.
- Class Side:** Shows the class hierarchy for 'FileStream', including subclasses like 'StandardFileStream' and 'MultiByteFileStream', and protocols like 'initialize-release' and 'stdio'.
- Methods:** Lists methods for the 'FileStream' class, including 'detectFile:do:', 'fileName:', 'forceNewFileName:', 'fullName:', 'isAFileName:', 'new', 'newFileName:', 'oldFileName:', 'oldFileOrNoneNamed:', and 'readOnlyFileName:'.
- Source Code:** Shows the implementation of the 'fileName:' method: `fileName: fileName` and `^self concreteStream fileName: (self fullName: fileName)`.

Source code

# Creating a new class

NB: A symbol



```
Object subclass: #CallGraph
  instanceVariableNames: ''
  classVariableNames: ''
  category: 'CallGraph'
```

To create a new class, send a message to its superclass in the system browser

NB: Be sure to write a *class comment!*

# Defining methods

Convention to  
indicate class name

“Selector” (method name)

argument

```
CallGraph>>from: aString  
calls := Character cr split: aString
```

method body

```
CallGraph>>calls  
^ calls
```

An accessor method

**NB: always put methods in a well-named “protocol”**



# How many calls are there in the call graph?

```
| cg |  
cg := CallGraph new from: (FileStream fileName: 'Calls.txt') contents.  
cg calls size 2476
```

*Let's improve the instantiation interface*

# Factory methods and other “static” methods are defined on the *class side*

```
CallGraph class>>fromFile: fileName  
^ self new from: (FileStream fileName: fileName) contents
```

```
(CallGraph fromFile: 'Calls.txt') calls size. 2476
```

Let's turn this into a test!

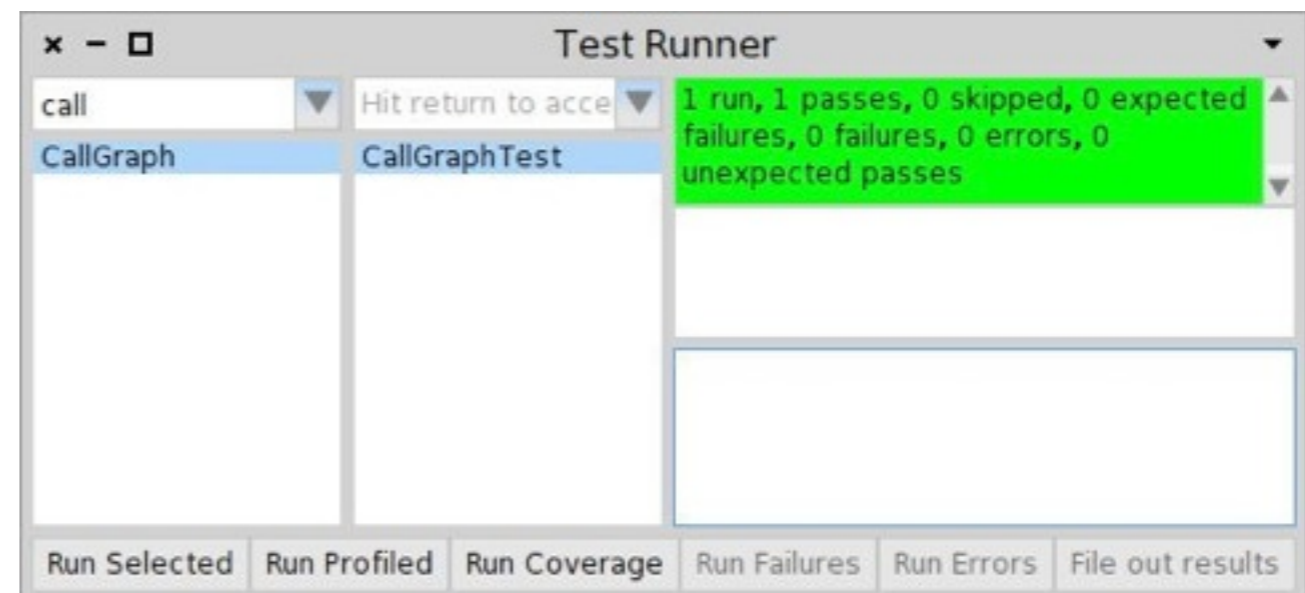
# Creating a simple test

a 5-line excerpt from Calls.txt

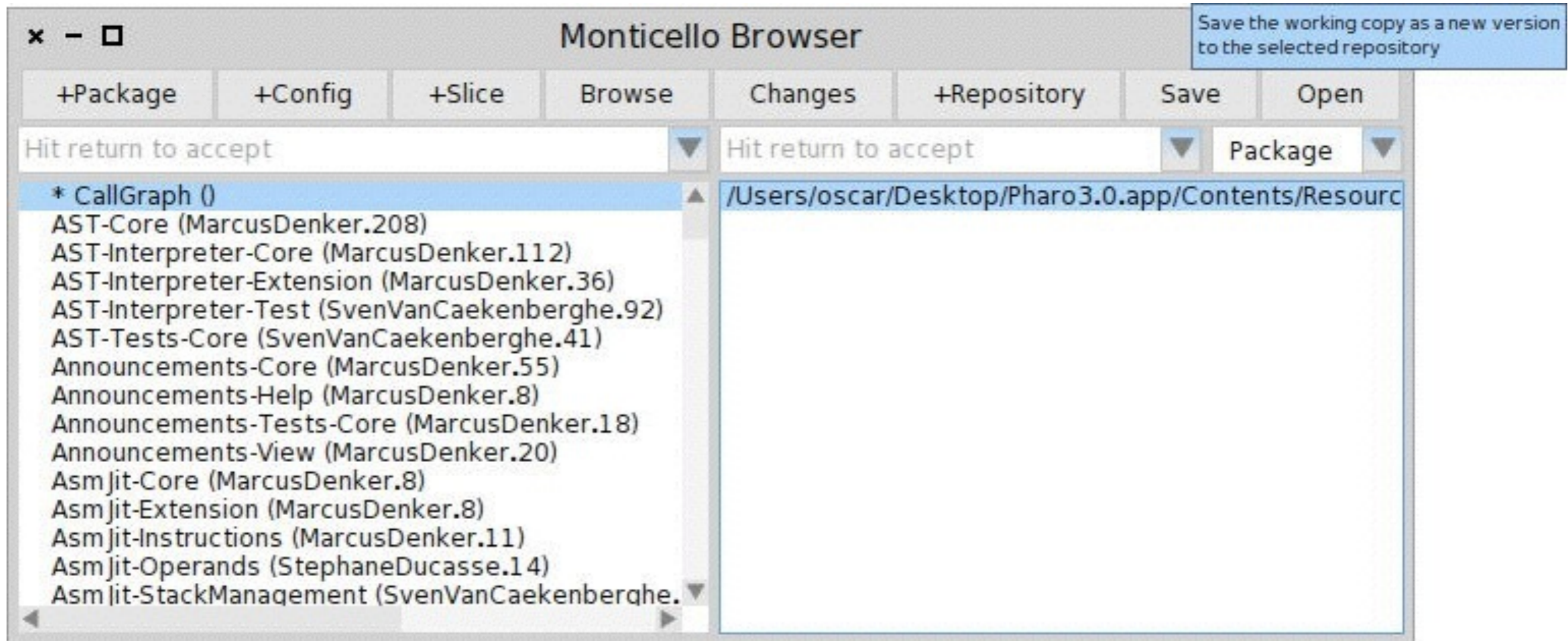
```
CallGraph class>>example  
^ self new from: '|java.lang.String:...'
```

```
TestCase subclass: #CallGraphTest  
instanceVariableNames: ''  
classVariableNames: ''  
category: 'CallGraph'
```

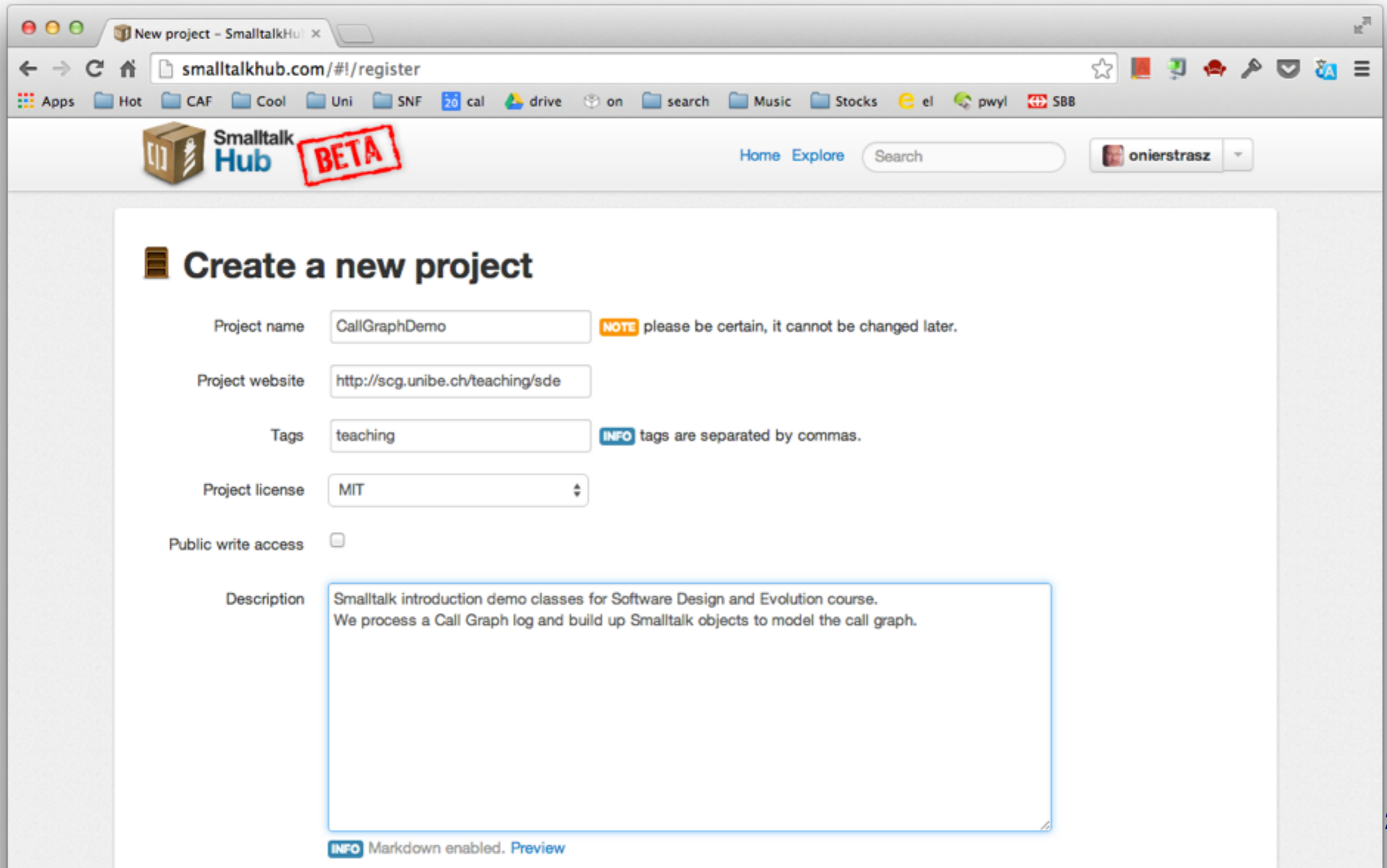
```
CallGraphTest>>testNumberOfCalls  
self assert: CallGraph example calls size equals: 5
```



# Monticello is a version control system for Smalltalk



# Smalltalkhub is a web site for sharing monticello projects



The screenshot shows a web browser window with the URL `smalltalkhub.com/#!/register`. The page title is "New project - SmalltalkHub". The browser's address bar shows the URL and various navigation icons. Below the browser window, the SmalltalkHub logo is visible, along with a "BETA" badge. The main content area is titled "Create a new project" and contains a form with the following fields:

- Project name:** CallGraphDemo. A note indicates: "NOTE please be certain, it cannot be changed later."
- Project website:** `http://scg.unibe.ch/teaching/sde`
- Tags:** teaching. An info note states: "INFO tags are separated by commas."
- Project license:** MIT (selected from a dropdown menu)
- Public write access:**
- Description:** Smalltalk introduction demo classes for Software Design and Evolution course. We process a Call Graph log and build up Smalltalk objects to model the call graph.

At the bottom of the form, there is an "INFO" note: "Markdown enabled. [Preview](#)"

# GitFileTree provides git integration

The screenshot shows a web browser displaying the GitHub repository page for `onierstrasz / callGraphs`. The page shows the file `CallGraph.class / instance / getMethod..st` with 6 lines of code. A configuration browser window is overlaid on the page, showing a list of dependencies. The `GitFileTree` dependency is highlighted. A second configuration browser window is also overlaid, showing the details for the `CallGraph` dependency.

**Repository: master@git@github.com:onierstrasz/callGraphs**

Hit return to accept

- GitFileTree (ThierryGoubier.27)
- Glorp (SvenVanCaekenberghe.38)
- GlorpDBX (EstebanM)
- GraphET (TudorGirb)
- Gravatar (TorstenBe)
- Grease (StephanEgg)
- HP35 (SvenVanCaek)
- INIFile (TorstenBerg)
- Iliad (HernanMorales)
- InstanceEncoder (He)
- JSON (PaulDeBruicke)
- Kendrick (SergeStin)
- KomHttpServer (Her)
- Load4s (HernanMoral)

Install Stable Vers

**CallGraph**

Hit return to accept

**CallGraph-OscarNierstrasz.1**

Name: CallGraph-OscarNierstrasz.1  
Author: OscarNierstrasz  
Time: 9 September 2014, 11:28:03 am  
UUID: aeca1702-b2ca-547f-ac4a-ad0a712c4b93  
Ancestors:

```
1 instance creation
2 getMethod: signature
3   | fields methodName |
4   fields := $. split: signature.
5   methodName := fields at: 2.
6   ^ methods at: methodName ifAbsentPut:
```

# Modeling Calls, Methods and Classes

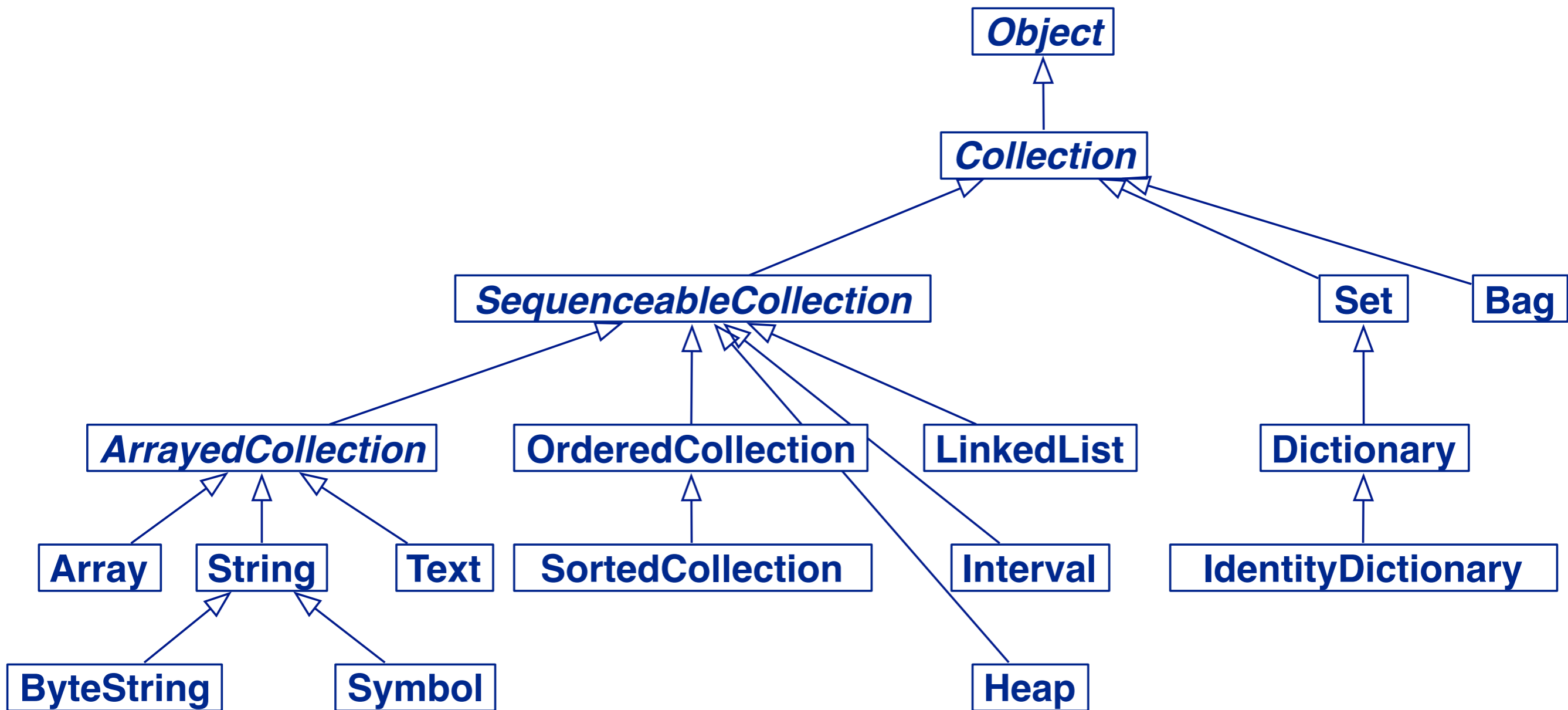
We want to build up a Call object for each line of the log

```
CallGraph>>from: aString  
  calls := (Character cr split: aString)  
           collect: [ :each | self createCall: each ]
```

```
'hello' collect: [ :each | each uppercase ] 'HELLO'
```

Let's look at Collections first ...

# Collections



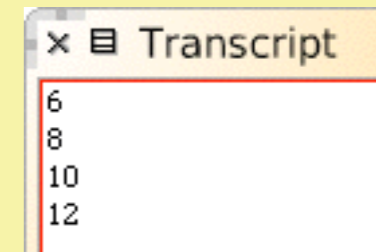
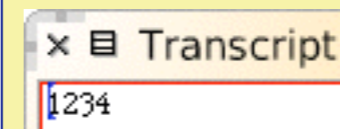
*Resist the temptation to program your own collections!*



# Common messages

```
#(1 2 3 4) includes: 5
#(1 2 3 4) size
#(1 2 3 4) isEmpty
#(1 2 3 4) contains: [:some | some < 0 ]
#(1 2 3 4) do:
  [:each | Transcript show: each ]
#(1 2 3 4) with: #(5 6 7 8)
  do: [:x : y | Transcript show: x+y; cr]
#(1 2 3 4) select: [:each | each odd ]
#(1 2 3 4) reject: [:each | each odd ]
#(1 2 3 4) detect: [:each | each odd ]
#(1 2 3 4) collect: [:each | each even ]
#(1 2 3 4) inject: 0
  into: [:sum :each | sum + each]
```

```
false
4
false
false
```

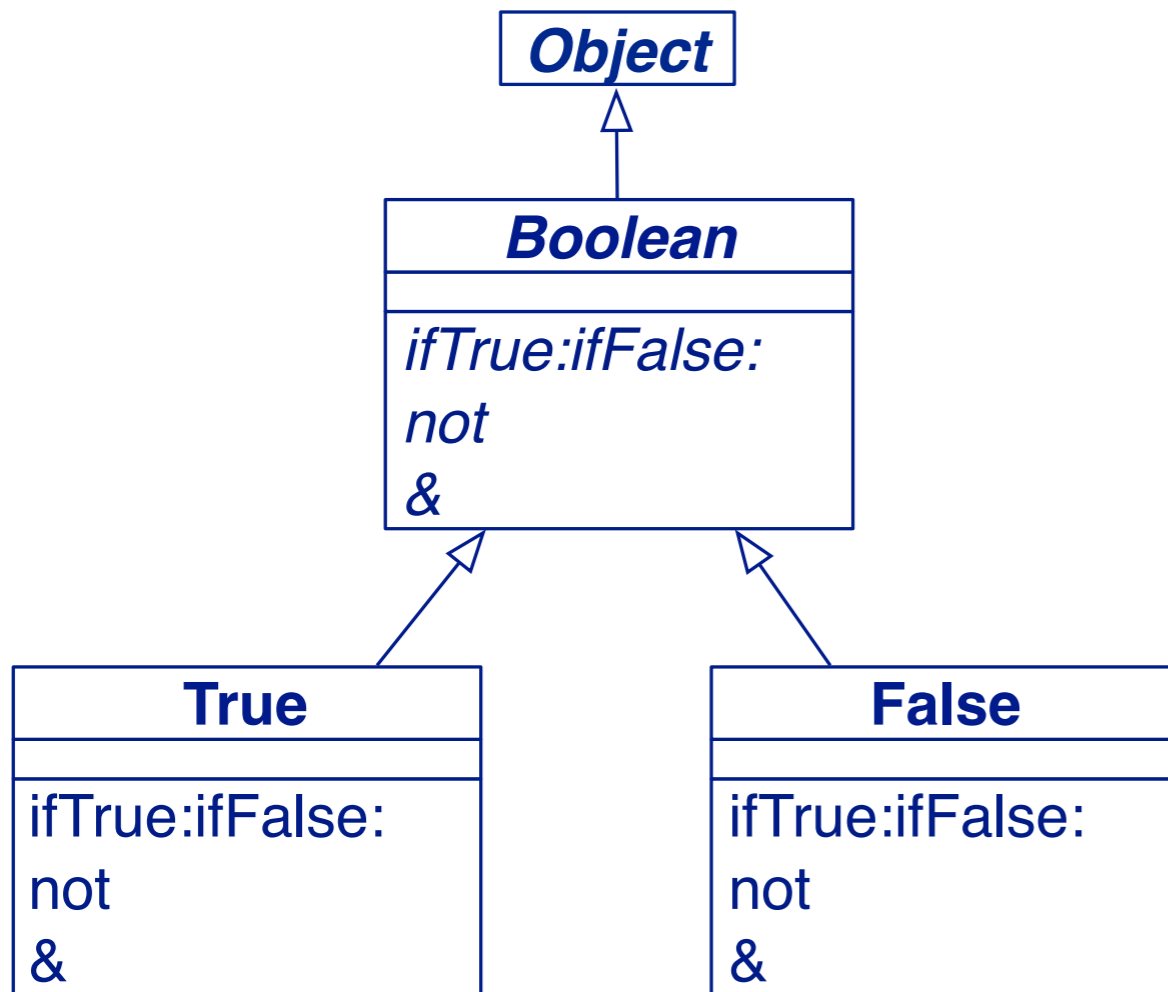


```
#(1 3)
#(2 4)
1
{false.true.false.true}

10
```

# Conditionals

```
(11 factorial + 1) isPrime ifTrue: [ 'yes' ] ifFalse: [ 'no' ]  
'yes'
```



- > All control constructs in Smalltalk are implemented by message passing
  - No keywords
  - Open, extensible
  - Built up from Booleans and Blocks

# Creating Calls, Methods and Classes

temporary (local) variables

```
CallGraph>>createCall: callString  
| fields callee |  
fields := $| split: callString.  
self assert: fields size = 5.  
self assert: (fields at: 1) size = 0.  
callee := self getMethod: (fields at: 2).  
^ Call new callee: callee  
"TODO -- handle the remaining fields!"
```

assertions (not tests)

a comment

```
CallGraph>>initialize  
super initialize.  
methods := Dictionary new
```

cache the methods!

```
CallGraph>>getMethod: signature  
| fields methodName |  
fields := $: split: signature.  
methodName := fields at: 2.  
^ methods at: signature  
ifAbsentPut: [ JMethod new name: methodName ]
```

```
CallGraph>> methods  
^ methods
```

# The debugger is your friend!

```
(CallGraph fromFile: 'Calls.txt') methods size.
```



# Using the debugger

The screenshot shows a debugger window titled "AssertionFailure: Assertion failed". The stack trace lists several frames, with "CallGraph createCall:" selected. Below the stack trace are buttons: Proceed, Restart, Into, Over, Through, Full Stack, Run to here, and Where is?.

```
createCall: callString
| fields callee |
fields := $| split: callString.
self assert: fields size = 5.
self assert: (fields at: 1) size = 0.
callee := self getMethod: (fields at: 2).
^ Call new callee: callee
"TODO -- handle the remaining fields!"
```

At the bottom, the "EyelInspector" shows the state of the selected frame:

- self: a CallGraph
- calls: [empty]
- methods: [empty]
- thisContext: CallGraph>>createCall:
- stackTop: [empty]
- all temp vars: [empty]
- callString: [empty]
- fields: [empty]
- callee: [empty]

The debugger reveals the false assumption that each log line is a complete entry

# Duck Typing

```
CallGraph>>from: aString
  calls := ((Character cr split: aString)
    select: #notEmpty)
    collect: [ :each | self createCall: each ]
```

Behaves like:

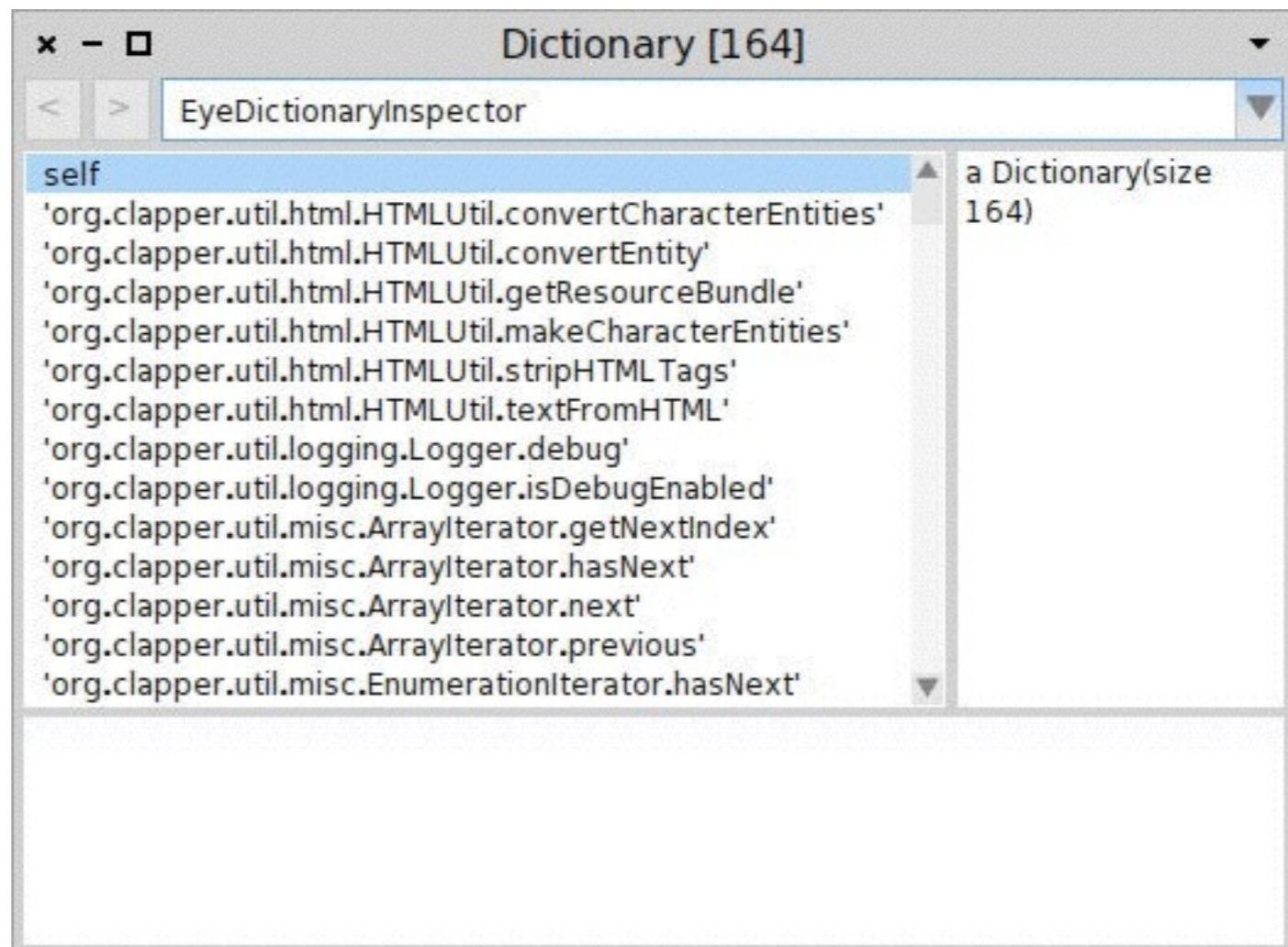
```
CallGraph>>from: aString
  calls := ((Character cr split: aString)
    select: [:each | each notEmpty])
    collect: [ :each | self createCall: each ]
```

since symbols also understand value:

# Number of methods

```
CallGraphTest>>testNumberOfMethods  
self assert: CallGraph example methods size equals: 5
```

```
(CallGraph fromFile: 'Calls.txt') methods size. 168
```



## To do ...

- > Model classes (introduce `JClass` class)
- > Model argument and return types of methods
- > Track which methods are static
- > Determine which methods are polymorphic



# Queries

```
(CallGraph fromFile: 'Calls.txt') methods size. 168
```

```
(CallGraph fromFile: 'Calls.txt') classes size. 209
```

```
((CallGraph fromFile: 'Calls.txt') methods  
  select: [ :m | m calls size > 1 ]) size. 141
```

```
((CallGraph fromFile: 'Calls.txt') methods  
  select: #isPolymorphic) size. 10
```

# What you should know!

- > What's the difference between a *method*, a *selector* and a *message*?
- > What are *categories* and *protocols*? What are they for?
- > How do you create a new class in Smalltalk?
- > What's the difference between `CallGraph` and `CallGraph class`?
- > What are “class side” methods for?
- > How is a block like a lambda?
- > What's the difference between a string and a symbol?

# Can you answer these questions?

- > Can a class access the fields of one of its instances?
- > Can you name something that is not an object in Smalltalk?
- > What happens to existing instances of a class if you add new fields at run time?
- > What will happen if you change the implementation of core classes (like Booleans or Strings)?
- > What's the difference between `self` and `super`?



## Attribution-ShareAlike 3.0

### You are free:

- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

### Under the following conditions:



**Attribution.** You must attribute the work in the manner specified by the author or licensor.



**Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

**Your fair use and other rights are in no way affected by the above.**

<http://creativecommons.org/licenses/by-sa/3.0/>