Pharo Syntax in a Nutshell

Damien Cassou, Stéphane Ducasse and Luc Fabresse

W1S05

Pharo

http://www.pharo.org

Getting a Feel About Syntax

Give you the general feel to get started:

- Overview of syntactical elements and constructs
- Three kinds of messages to minimize parentheses
- Overview of block syntax

This lecture is an **overview**
No stress if you do not get it right now!
We will repeat in future lectures

The Complete Syntax on a Postcard

No need to understand everything! But "everything" is on this screen :)

```
exampleWithNumber: x
 "This method illustrates the complete syntax."
 <aMethodAnnotation>

 | y |
 true & false not & (nil isNil)
  ifFalse: [ self halt ].
 y := self size + super size.
 #($a #a 'a' 1 1.0)
  do: [ :each | Transcript
        show: (each class name);
        show: (each printString);
        show: ' ' ].
 ^ x < y
```

Hello World

```
'Hello World' asMorph openInWindow
```

We send the message asMorph to a string and obtain a graphical element that we open in a window by sending it the message openInWorld

Getting the Pharo Logo from the Web

```
(ZnEasy getPng: 'http://pharo.org/web/files/pharo.png')
  asMorph openInWindow
```

- ZnEasy designates a class
  - Class names start with an uppercase character
- Message getPng: is sent to the ZnEasy class with a string as argument
  - getPng: is a keyword message
- 'http://pharo.org/web/files/pharo.png' is a string
- Messages asMorph and openInWindow are from left to right

Syntactic Elements

comment	"a comment"
character	$c $# $@
string	'lulu' 'l"idiot'
symbol (unique string)	#mac #+
literal array	#(12 23 36)
integer	1, 2r101
real	1.5 6.03e-34,4, 2.4e7
boolean	true, false (instances of True and False)
undefined	nil (instance of UndefinedObject)
point	10@120

Essential Constructs

- Temporary variable declaration: | var |
- Variable assignment: var := aValue
- Separator: message . message
- Return: ^ expression
- Block (lexical closures, a.k.a anonymous method)

```
[ :x | x + 2 ] value: 5
> 7
```

Essence of Pharo Computation

- Objects (created using messages)
- Messages
- Blocks (anonymous methods)

Three Kinds of Messages to Minimize Parentheses

- Unary message
  - Syntax: receiver selector
  - 9 squared
  - Date today
- Binary message
  - Syntax: receiver selector argument
  - 1+2
  - 3@4
- Keyword message
  - Syntax: receiver key1: arg1 key2: arg2
  - 2 between: 10 and: 20

Message Precedence

(Msg) > Unary > Binary > Keywords

- First we execute ()
- Then unary, then binary and finally keyword messages

This order minimizes () needs
But let us start with messages

Sending an Unary Message

receiver selector

Example

10000 factorial

We send the message factorial to the object 10000

Sending a Binary Message

```
receiver selector argument
```

Example

```
1 + 3
```

We send the message `+` to the object `1` with the object `3` as argument

Sending a Keyword Message

```
receiver keyword1: arg1 keyword2: arg2
```

equivalent to C like syntax

```
receiver.keyword1keyword2(arg1, arg2)
```

Example: Sending an HTTP Request

```
ZnClient new
url: 'https://en.wikipedia.org/w/index.php';
queryAt: 'title' put: 'Pharo';
queryAt: 'action'  put: 'edit';
get
```

- new is a unary message sent to a class
- queryAt:put: is a keyword message
- get is a unary message
- ; (called a cascade) sends all messages to the same receiver

Messages are Everywhere!

- Conditionals
- Loops
- Iterators
- Concurrency

Conditionals are also Message Sends

```
factorial
  "Answer the factorial of the receiver."
  self = 0 ifTrue: [ ^ 1 ].
  self > 0 ifTrue: [ ^ self * (self – 1) factorial ].
  self error: 'Not valid for negative integers'
```

- ifTrue: is sent to an object, a boolean!
- ifFalse:ifTrue:, ifTrue:ifFalse: and ifFalse: also exist

You can read their implementation, this is not magic!

Loops are also Message Sends

```
1 to: 4 do: [ :i| Transcript << i ]
> 1
> 2
> 3
> 4
```

- to:do: is a message sent to an integer
- Many other messages implement loops: timesRepeat:, to:by:do:, whileTrue:, whileFalse:, ...

With Iterators

We ask the collection to perform the iteration on itself

```
#(1 2 −4 −86)
  do: [ :each | Transcript show: each abs printString ; cr ]
> 1
> 2
> 4
> 86
```

Blocks Look like Functions

fct(x) = x*x+3

```
fct := [ :x | x * x + 3 ]
```

fct(2)

```
fct value: 2
```

Blocks

- Kind of anonymous methods

```
[ :each | Transcript show: each abs printString ; cr ]
```

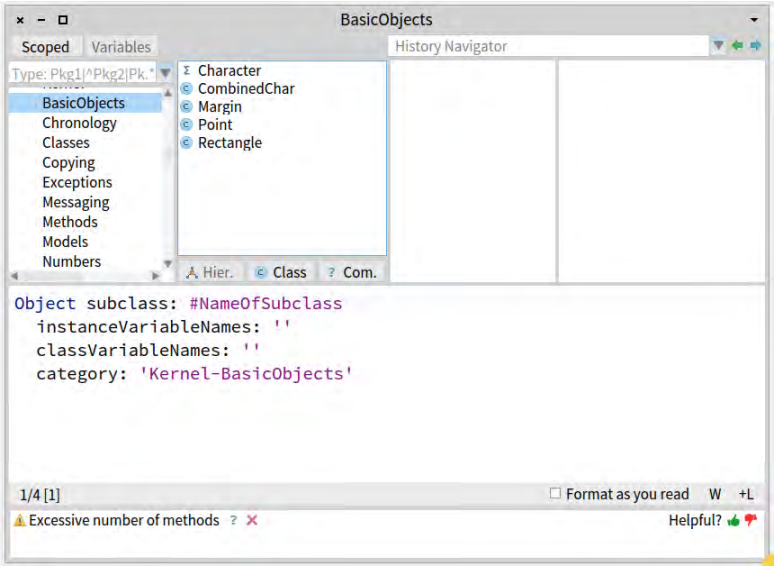
- Are lexical closures
- Are plain objects:
  - can be passed as method arguments
  - can be stored in variables
  - can be returned

Block Usage

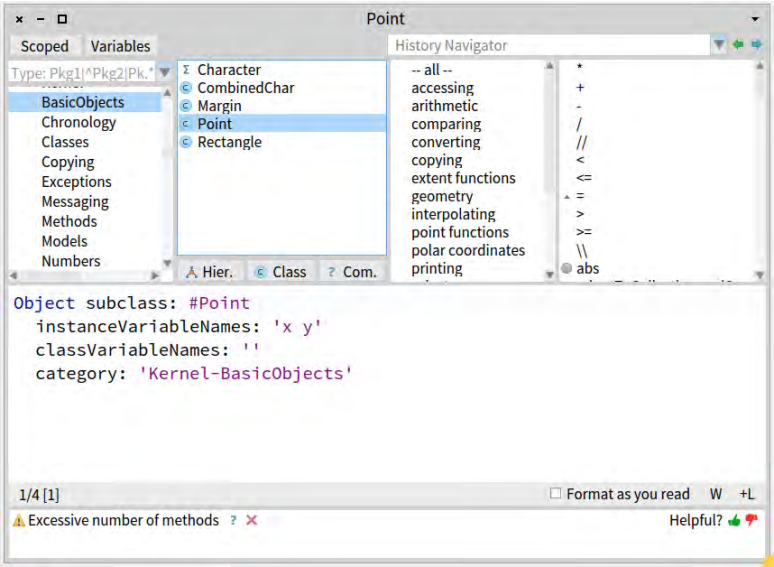
```
#(1 2 −4 −86)
  do: [ :each | Transcript show: each abs printString ; cr ]
> 1
> 2
> 4
> 86
```

- [ ] delimits the block
- :each is the block argument
- each will take the value of each element of the array

Class Definition Template



Class Definition within the IDE



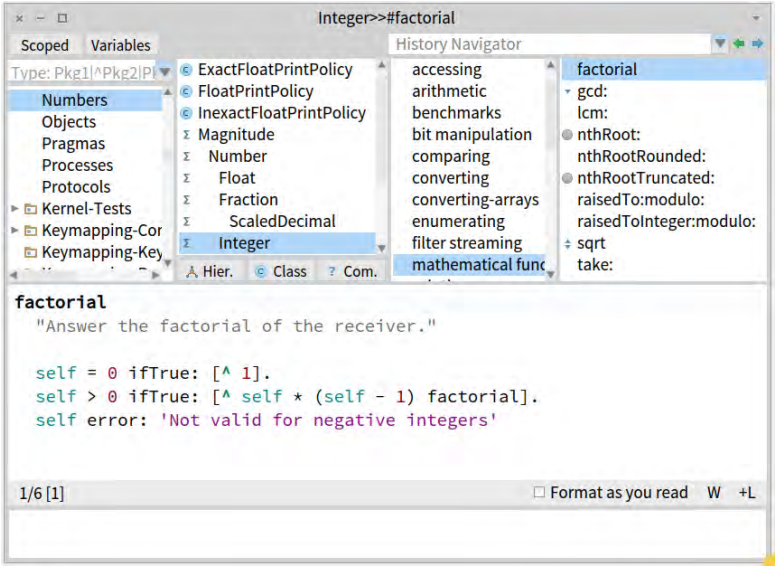
Method Definition

- Methods are public
- Methods are virtual (*i.e.,* looked up at runtime)
- By default return self

```
messageSelectorAndArgumentNames
  "comment stating purpose of message"

  | temporary variable names |
  statements
```

Method Definition Example



Messages Summary

3 kinds of messages:

- Unary: Node new
- Binary: 1+2, 3@4
- Keywords: 2 between: 10 and: 20

Message Priority:

- (Msg) > unary > binary > keyword
- Same-Level messages: from left to right

Conclusion

- Compact syntax
- Few constructs but really expressive
- Mainly messages and closures
- Three kinds of messages
- Support for Domain Specific Languages

A course by

and

in collaboration with