## Pharo: a Live Programming Environment

Pharo comes with an integrated development environment. Pharo is a *live programming environment*: you can modify your objects and your code while your program is executing. All Pharo tools are implemented in Pharo:

- a code browser with refactorings;
- a debugger, a workspace, and inspectors;
- the compiler itself and much, much more.

Code can be inspected and evaluated directly in the image, using simple key combinations and menus (open the contextual menu on any selected text to see available options).

The 5 Panes Pharo Code Browser
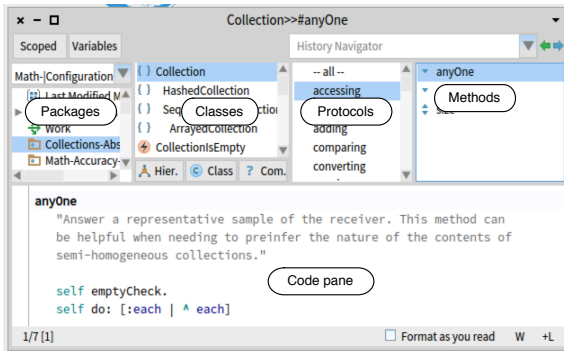
- The *packages* pane shows all the packages of the system.
- The *classes* pane shows the class hierarchy of the selected package; the *class side* checkbox allows for getting the methods of the metaclass.
- The *protocols* pane groups the methods of the selected class to ease navigation. When a protocol name starts with a *, methods of this protocol belong to a different package (*e.g.,* the *Fuel* protocol groups methods that belong to the *Fuel* package);
- The *methods* pane lists the methods of the selected protocol; icons are clickable and trigger special actions;
- The *source code* pane shows the source code of the selected method.

Defining a class

To add a class or edit a class, edit the proposed template! The following expression defines the class `Counter` as a subclass of `Object`. It defines two instance variables `count` and `initialValue` inside the package `MyCounter`.

```
Object subclass: #Counter
    instanceVariableNames: 'count initialValue'
    classVariableNames: ''
    package: 'MyCounter'
```

The method `initialize` is automatically invoked when a new instance is created by sending the message `new` to the class i.e., `Counter new`.

```
Counter >> initialize
    super initialize.
    count := 0.
```

`Counter >> initialize` is a *notation* to indicate that the following text is the content of the method `initialize` in the class `Counter`.

Methods

Methods are public and virtual. There are always looked up in the class of the receiver. By default a method returns `self`. Class methods follow the same dynamic lookup than instance methods. Method `factorial` defined in class Integer.

```
Integer >> factorial
    "Answer the factorial of the receiver."
    self = 0 ifTrue: [^ 1].
    self > 0 ifTrue: [^ self * (self - 1) factorial].
```

Unit testing

A test must be implemented in a method whose name starts with `test` and in a class that inherits from `TestCase`.

```
OrderedCollectionTest >> testAdd
    | added |
    added := collection add: 'foo'.
    self assert: (collection includes: 'foo').
```

The second line declares the variable `added`. The message `assert:` expects a true value.

A simple, uniform and powerful model

Pharo has a simple dynamically-typed object model:

- everything is an object — instance of a class;
- classes are objects too and there is single inheritance between classes;
- traits are groups of methods that can be reused orthogonally to inheritance;
- instance variables are protected; methods are public and virtual; and blocks are lexical closures.

Less is more: No type declarations, no primitive objects, no generic types, no modifiers, no operators, no inner classes, no constructors, and no static methods. There are not needed!

An innovative, open-source Smalltalk-inspired language and system for live programming

http://www.pharo.org

Pharo is both an *object-oriented*, *dynamically-typed* general-purpose language and its own programming environment. The language has a simple and expressive syntax which can be learned in a few minutes. Concepts in Pharo are *very consistent*:

- Everything is an object: buttons, colors, arrays, numbers, classes, methods... *Everything!*
- A small number of rules, no exceptions!

Main Web Sites

Code hosting http://smalltalkhub.com
Questions https://pharoproject.slack.com/
Blog http://pharoweekly.wordpress.com
Contributors http://pharo.org/about
Topics http://topics.pharo.org
Consortium http://consortium.pharo.org
Association http://association.pharo.org

PharoBooks

Pharo books are available at: http://books.pharo.org
Pharo By Example, Deep into Pharo, Enterprise Pharo: a Web Perspective, Numerical Methods in Pharo, TinyBlog Tutorial, Dynamic Web Development in Seaside (http://book.seaside.st)

More books http://stephane.ducasse.free.fr/FreeBooks

Minimal Syntax

Six reserved words only	
`nil`	the undefined object
`true,false`	the boolean objects
`self`	the receiver of the current message
`super`	the receiver but for accessing overridden methods
`thisContext`	the current method or block activation

Minimal Syntax (II)

Object constructors & reserved syntactic constructs

"comment"	
'string'	collection of characters
#symbol	unique string
$a, Character space	the character a and a space
12 2r1100 16rC	twelve (decimal, binary, hexa)
3.14 1.2e3	floating-point numbers
#(abc 123)	literal array with the symbol #abc and the number 123
{foo . 3+2}	dynamic array built from 2 expressions
#[123 21 255]	byte array
| foo bar |	declaration of two temporary variables
var := *expr*	assignment
exp1. *exp2*	period - expression separator
;	semicolon - message cascade
[:p | *expr*]	code block with a parameter
<unary>	method annotation
<key:'*any*'wrd:#*lit*>	with any literal arguments
^ *expr*	caret - returns a result from a method

Message Sending

When we send a message to an object (the *receiver*), the corresponding method is selected and executed, and the method answers an object. Messages syntax mimics natural languages, with a subject, a verb, and complements.

Pharo	Java
aColor r: 0.2 g: 0.3 b: 0	aColor.setRGB(0.2,0.3,0)
d at: '1' put: 'Chocolate'.	d.put("1", "Chocolate");

Three Types of Messages: Unary, Binary, and Keyword

A **unary message** is one with no arguments.

Array new.	⤳ anArray
#(4 2 1) size.	⤳ 3

new is an unary message sent to classes (classes are objects).

A **binary message** takes only one argument and is named by one or more symbol characters from +, −, *, =, <, >, ...

---

3 + 4	⤳ 7
'Hello' , ' World'	⤳ 'Hello World'

The + message is sent to the object 3 with 4 as argument. The string 'Hello' receives the message , (comma) with ' World' as the argument.

A **keyword message** can take one or more arguments that are inserted in the message name.

'Pharo' allButFirst: 2.	⤳ 'aro'
[:x | x + 2] value: 7	⤳ 9
3 to: 10 by: 2.	⤳ (3 to: 10 by: 2)

The second line executes a block. The third example sends to:by: to 3, with arguments 10 and 2; this returns an interval containing 3, 5, 7, and 9.

Message Precedence

Parentheses > unary > binary > keyword, and finally from left to right.

    (10 between: 1 and: 2+4*3) not

Messages + and * are sent first, then between:and: is sent, and not. The rule suffers no exception: operators are just binary messages with *no notion of mathematical precedence*. 2+4*3 reads left-to-right and gives 18, not 14!

Cascade: Sending Muliple Messages to the Same Object

Using ; (a cascade) multiple messages are sent to the result of the same expression. Here ; arrives after add: 1, so messages add: 2 and add: 3 are sent to add: 1's receiver: a collection.

```
OrderedCollection new
    add: 1;
    add: 2;
    add: 3.
```

The whole message cascade value is the value of the last message sent (the symbol #g). To return the receiver of the message cascade instead (*i.e.,* the collection), send yourself as the last message of the cascade.

Blocks

Blocks are objects containing code that is executed on demand. They are the basis for control structures: conditionals & loops.

```
2 = 2
    ifTrue: [ Error signal: 'Help' ].
```

Send the message ifTrue: to the boolean true (computed from 2 = 2) with a block as argument. Because the boolean is

---

true, the block is executed and an exception is signaled.

```
#('Hello World' $!)
    do: [ :e | Transcript show: e ]
```

Send the message do: to an array. This executes the block once for each element, passing it via the e parameter. As a result, Hello World! is printed.

Common Constructs

Conditionals

condition	if (condition)
ifTrue: [action]	{ action(); }
ifFalse: [anotherAction]	else { anotherAction(); }
[condition] whileTrue:	while (condition) { action();
[action. anotherAction]	anotherAction(); }

Loops/Iterators

1 to: 11 do: [:i |	for(int i=1; i<11; i++){
Transcript show: i ; cr]	System.out.println(i); }
| names |	String [] names ={"A", "B", "C"};
names := #('A' 'B' 'C').	for(String name : names) {
names do: [:each |	System.out.print(name);
Transcript show: each, ',']	System.out.print(","); }

Collections start at 1. aCol at:i accesses element at i and aCol at:i put:value sets element at i to value.

Collections

#(4 2 1) at: 3	⤳ 1
#(4 2 1) copy at: 3 put: 6	⤳ #(4 2 6)
{4 . 2 . 1} at: 3 put: 6	⤳ #(4 2 6)
(Array new: 2) add: 4; add: 2 ; yourself	⤳ #(4 2)
Set new add: 4; add: 4 ; yourself	⤳ aSet
Dictionary new	
at: #a put: 'Alpha' ; yourself	⤳ aDictionary

Files and Streams

work := FileSystem disk workingDirectory.
stream := (work / 'foo.txt') writeStream.
stream nextPutAll: 'Hello World'.
stream close.
stream := (work / 'foo.txt') readStream.

stream contents.	⤳ 'Hello World'

stream close.