Pharo Syntax

The conceptual model behind Pharo is simple: everything is an object and everything happens by sending messages to objects. Objects are instances of a class, an object that defines their behaviour — the implementation of methods that handle messages — as well as their layout — the slots for the data they encapsulate. Classes are organised in a single inheritance tree. That's it.

Pharo syntax is simple and elegant. Let's look at an example.

```
'hello' reversed
```

Selecting this piece of code and looking at the result of evaluating it — something you can do everywhere when working in Pharo — will yield

```
‘olleh’
```

as result.

What happened is that the message *reversed* was sent to a string, resulting in a new string with the characters reversed.

The literal string ‘hello’ is called the *receiver* while *reversed* is called the message *selector (*often just *message*).

Here is another example:

```
'hello' asUppercase
```

evaluates to

```
'HELLO'
```

These kinds of messages are called *unary messages* because they take no arguments.

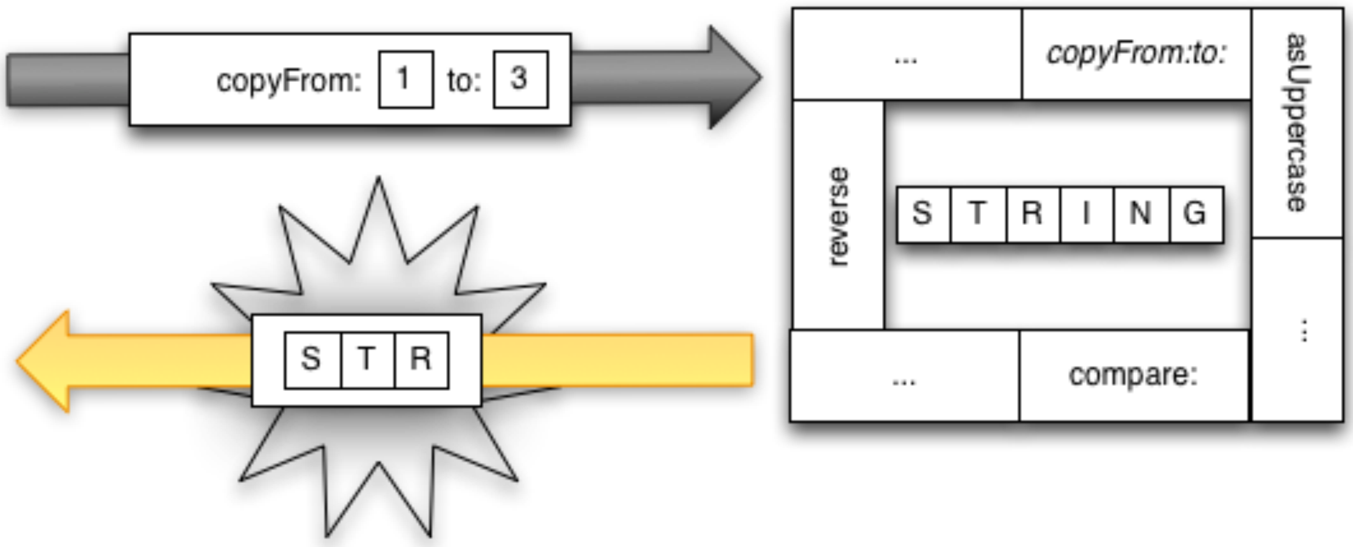
The general form for messages that take arguments uses interpolated keywords ending in colons, like this.

```
'STRING' copyFrom: 1 to: 3
```

which results in a new string

```
'STR'
```

Note how natural *keyword messages* read and how they are self documenting with well-chosen names.



Sending a message is the only way you can interact with an object

The third and last kind of messages are *binary messages*. These are used for arithmetic operations, among others.

```
1 + 2
```

Here, the message + is sent to the integer 1 with 2 as argument.

The simpler messages take precedence over the more complex ones, so unary messages are evaluated first, then binary message and finally keyword messages.

Parenthesis can be used to control the order of evaluation.

```
'string' asUppercase copyFrom: −1 + 2 to: 6 − 3
```

The above evaluates to ‘STR’ while the expression below evaluates to ‘RTS’.

```
('string' asUppercase first: 9 / 3) reversed
```

Note how you can send messages to the result of an expression: *copyFrom:to:* and *first:* were sent to the result of *asUppercase*. This is called *chaining*.

To refer to a class, a capitalised name is used. Classes are an important entry point to create objects. Of course, this is done by sending messages to the class object.

```
String new
```

Returns a new, empty string, ‘’. Some classes understand many messages, like the class Float that knows about *pi*.

Multiple *statements* are separated using a dot, just like sentences. Local variables are declared by writing their lowercase name in between vertical bars.

```
| array |
array := Array new: 3.
array at: 1 put: true.
array at: 2 put: false.
array
```

The result of the code above is a new array. Indexing is one-based, like normal humans count.

```
#(true false nil)
```

The literal array syntax is #(…). The first element is the boolean constant *true*, the second its counterpart *false*. Uninitialised elements remain *nil*, the undefined object constant. There are only six reserved keywords, the constants *true*, *false* and *nil* are three of them.

As you can see, *assignment* is done using := while the last statement defines the result for the whole program.

Often you'll be sending multiple messages to the same object, to the same receiver. To make this easier, there is some syntactic sugar called a *message cascade* using a semicolon. Our previous code could have been written as follows.

```
(Array new: 3)
  at: 1 put: true;
  at: 2 put: false;
  yourself
```

The 3 indented messages form a cascade, they are all being sent to the same object, the new array. The last message, *yourself*, is particularly useful in a cascade: it returns the object it is sent to. This is necessary because the result of the before last message send was not the object itself — the *at:put:* message returns the value assigned.

Square brackets are used to specify *blocks* (also known as closures or lambdas), pieces of code to be executed later on.

```
| adder |
adder := [ :x | x + 1 ].
adder value: 100
```

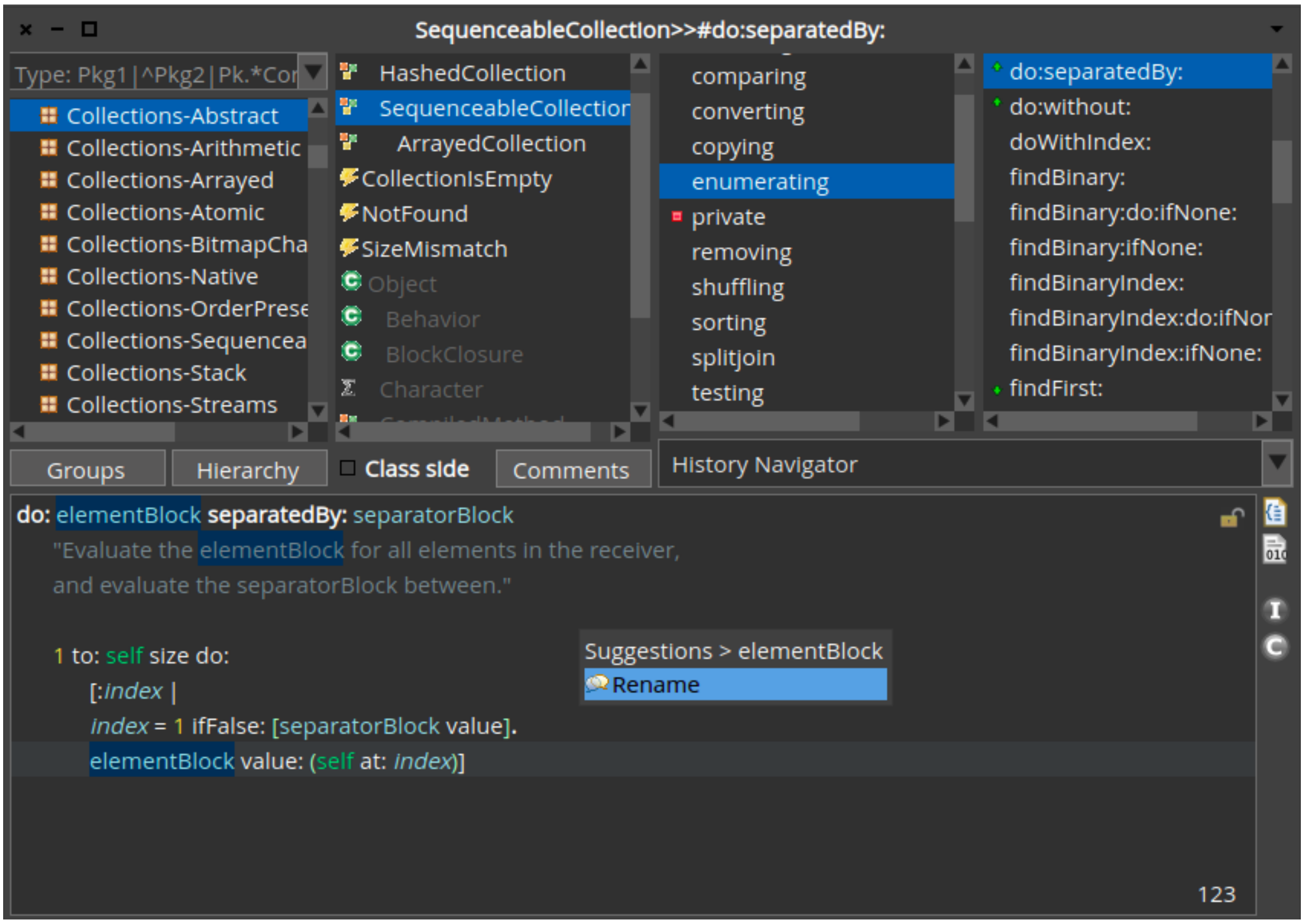
The *adder* local variable is assigned a one argument block. The code inside the block names the variables it accepts and the statements to be executed when it is evaluated. Evaluating a block is done by sending a message, *value:* with an actual object as argument. The argument gets bound to the variable and the block is executed, resulting in 101.

Blocks are used to express all control structures, from standard conditionals and loops to the exotic application specific ones, using the normal messaging syntax.

Here is an actual piece of code from Pharo, the implementation of the method *do:separatedBy:* on the class SequenceableCollection.

```
do: elementBlock separatedBy: separatorBlock
  “Evaluate the elementBlock for all elements in the receiver,
   and evaluate the separatorBlock between.”

  1 to: self size do: [ :index |
    index = 1 ifFalse: [ separatorBlock value ].
    elementBlock value: (self at: index) ]
```



```
do: elementBlock separatedBy: separatorBlock
	"Evaluate the elementBlock for all elements in the receiver,
	and evaluate the separatorBlock between."

	1 to: self size do:
		[:index |
		index = 1 ifFalse: [separatorBlock value].
		elementBlock value: (self at: index)]
```

Selection of a method argument & suggested refactoring in a code browser

Methods are entered one by one in a code browser, like the one shown in the screenshot. The first line specifies the method name, the selector, with names for all arguments. Comments are surrounded by double quotes.

Inside a method, *self* refers to the object itself, the receiver. The equivalent of a basic for loop is *to:do:* which takes a block as its last argument. This block is executed for each index. The conditional, *ifFalse:*, is written as a message sent to a boolean, true or false, the result of =, comparing two objects for equality and takes a block as its argument.

In a method, the receiver (self) is the default return value of the whole method. Using a caret we can return something else or even return earlier. Here is the code of the method *allSatisfy:* on the class Collection.

```
allSatisfy: aBlock
  "Evaluate aBlock with the elements of the receiver.
   If aBlock returns false for any element return false.
   Otherwise return true.”

  self do: [ :each |
    (aBlock value: each) ifFalse: [ ^ false ] ].
  ^ true
```

Using *do:* we iterate over all elements in the collection. For each element we evaluate block for a boolean value and act accordingly. As soon as we get a false value, we back out and return an overall false value. If every evaluation gave us true, we passed the whole test and can return true as overal result.

The following is then true.

```
#(2 4 8 16 32) allSatisfy: [ :each | each even ]
```

But the following is false.

```
#(1 2 3 4 5 6) allSatisfy: [ :each | each odd ]
```

You now know enough to read 95% of Pharo code. Remember, it is all just messages being sent to objects.

More advanced syntax not explained:

- character literals such as $A
- symbols such as #foo
- integer radix notation such as 16rFF
- dynamic arrays like { 1+2. 3+4 }
- the *super* pseudo variable
- instance and class variables
- the *thisContext* pseudo variable
- method pragmas such as <Example>

Written by Sven Van Caekenberghe