Pharo 9 by Example

Stéphane Ducasse, Sebastijan Kaplar, Gordana Rakic and Quentin Ducasse

July 27, 2021

Layout and typography based on the `sbabook` LaTeX class by Damien Pollet.

Contents

Illustrations

CHAPTER 1

Syntax in a nutshell

Pharo adopts a syntax very close to that of its ancestor, Smalltalk. The syntax is designed so that program text can be read aloud as though it were a kind of pidgin English. The following method of the class `Week` shows an example of the syntax. It checks whether `DayNames` already contains the argument, i.e., if this argument represents a correct day name. If this is the case, it will assign it to the class variable `StartDay`.

```
startDay: aSymbol

   (DayNames includes: aSymbol)
      ifTrue: [ StartDay := aSymbol ]
      ifFalse: [ self error: aSymbol, ' is not a recognised day
    name' ]
```

Pharo's syntax is minimal. Essentially there is syntax only for sending messages (i.e., expressions). Expressions are built up from a very small number of primitive elements (message sends, assignments, closures, returns...). There are only 6 reserved keywords, i.e., pseudo-variables, and there are no dedicated syntax constructs for control structures or declaring new classes. Instead, nearly everything is achieved by sending messages to objects. For instance, instead of an if-then-else control structure, conditionals are expressed as messages (such as `ifTrue:`) sent to Boolean objects. New subclasses are created by sending a message to their superclass.

1.1 Syntactic elements

Expressions are composed of the following building blocks:

1. The six *pseudo-variables*: `self`, `super`, `nil`, `true`, `false`, and `thisContext`
2. Constant expressions for *literal* objects including numbers, characters, strings, symbols and arrays
3. Variable declarations
4. Assignments
5. Block closures
6. Messages
7. Method returns

We can see examples of the various syntactic elements in the table below.

Syntax expression	**What it represents**
`startPoint`	a variable name
`Transcript`	a global variable name
self	pseudo-variable
`1`	decimal integer
`2r101`	binary integer
`1.5`	floating point number
`2.4e7`	number in exponential notation
`$a`	the character `'a'`
`'Hello'`	the string `'Hello'`
`#Hello`	the symbol `#Hello`
`#(1 2 3)`	a literal array
`{ 1 . 2 . 1 + 2 }`	a dynamic array
`"a comment"`	a comment
`| x y |`	declaration of variables `x` and `y`
`x := 1`	assign `1` to `x`
`[:x | x + 2 ]`	a block that evaluates to `x + 2`
`<primitive: 1>`	a method annotation (here primitive)
`3 factorial`	unary message `factorial`
`3 + 4`	binary message `+`
`2 raisedTo: 6 modulo: 10`	keyword message `raisedTo:modulo:`
`^ true`	return the value `true`
`x := 2 . x := x + x`	two expressions separated by separator (`.`)
`Transcript show: 'hello'; cr`	two cascade messages separated by (`;`)

Local variables. `startPoint` is a variable name, or identifier. By convention, identifiers are composed of words in ”camelCase” (i.e., each word except the first starting with an upper case letter). The first letter of an instance variable, method or block parameters, or temporary variable must be lower case. This indicates to the reader that the variable has a private scope.

Shared variables. Identifiers that start with upper case letters are global variables, class variables, pool dictionaries or class names. `Transcript` is a global variable, an instance of the class `ThreadSafeTranscript`.

The message receiver. `self` is a pseudo-variable that refers to the object that receives the message (that led to the execution of the method using `self`). It gives us a way send messages to it. We call `self` ”the receiver” because this object will receive the message that causes the method to be executed. Finally, `self` is called a ”pseudo-variable” since we cannot directly change its values or assign to it.

Integers. In addition to ordinary decimal integers like 42, Pharo also provides a radix notation. `2r101` is 101 in radix 2 (i.e., binary), which is equal to decimal 5.

Floating point numbers. Such numbers can be specified with their base-ten exponent: `2.4e7` is `2.4 x 10^7`.

Characters. A dollar sign introduces a literal character: `$a` is the literal for the character `'a'`. Instances of special, non-printing characters can be obtained by sending appropriately named messages to the `Character` class, such as `Character space` and `Character tab`.

Strings. Single quotes `' '` are used to define a literal string. If you want a string with a single quote inside, just double the quote, as in `'G''day'`.

Symbols. Symbols are like Strings, in that they contain a sequence of characters. However, unlike a string, a literal symbol is guaranteed to be globally unique. There is only one `Symbol` object `#Hello` but there may be multiple `String` objects with the value `'Hello'`.

Compile-time literal arrays. They are defined by `#( )`, surrounding space-separated literals. Everything within the parentheses must be a compile-time constant. For example, `#(27 (true) abc 1+2)` is a literal array of 6 elements: the integer 27, the compile-time array containing the object `true` (non-changeable Boolean), the symbol `#abc`, the integer 1, the symbol + and the integer 2. Note that this is the same as `#(27 #(true) #abc 1 #+ 2)`.

Run-time dynamic arrays. Curly braces `{ }` define a dynamic array whose elements are expressions, separated by periods, and evaluated at run-time. So `{ 1. 2. 1 + 2 }` defines an array with elements 1, 2, and 3 the result of evaluating `1+2`.

Comments. They are enclosed in double quotes ” ”. ”hello” is a comment, not a string, and is ignored by the Pharo compiler. Comments may span multiple lines but they cannot be nested.

Local variable definitions. Vertical bars `| |` enclose the declaration of one or more local variables before the beginning of a method or a block body.

Assignment. The two characters `:=` specify that a variable refers to an object.

Blocks. Square brackets `[ ]` define a block, also known as a block closure or a lexical closure, which is a first-class object representing a function. As we shall see, blocks may take arguments (`[:i | ...]`) and can have local variables (`[| x | ...]`). Blocks also close over their definition environment, i.e., they can refer to variables that where reachable at the time of their definition.

Pragmas and primitives. `< primitive: ... >` is a method annotation. This specific one denotes the invocation of a virtual machine (VM) primitive. In the case of a primitive the code following it, it either to explain what the primitive is doing (for essential primitives) or is executed only if the primitive fails (for optional primitive). The same syntax of a message within `< >` is also used for other kinds of method annotations also called pragmas.

Unary messages. These consist of a single word (like `factorial`) sent to a receiver (like `3`). In `3 factorial`, `3` is the receiver, and `factorial` is the message selector.

Binary messages. These are messages sent to a receiver with a single argument, and whose selector looks like mathematical operator (for example: `+`). In `3 + 4`, the receiver is `3`, the message selector is `+`, and the argument is `4`.

Keyword messages. Their selectors consist of one or more keywords (like `raisedTo: modulo:`), each ending with a colon and taking a single argument. In the expression `2 raisedTo: 6 modulo: 10`, the message selector `raisedTo:modulo:` takes the two arguments `6` and `10`, one following each colon. We send the message to the receiver `2`.

Sequences of statements. A period or full-stop (`.`) is the statement separator. Putting a period between two expressions turns them into independent statements like in `x := 2. x := x + x`. Here we first assign value `2` to the variable `x`, and then duplicate its value by assigning a value of `x + x` to it.

Cascades. Semicolons (`;`) are used to send a cascade of messages to a single receiver. In `stream nextPutAll: 'Hello World'; close` we first send the keyword message `nextPutAll: 'Hello World'` to the receiver `stream`, and then we send the unary message `close` to the same receiver.

Method return. `^` is used to *return* a value from a method.

The basic classes `Number`, `Character`, `String` and `Boolean` are described in Chapter : Basic Classes.

1.2 Pseudo-variables

In Pharo, there are 6 pseudo-variables: `nil`, `true`, `false`, `self`, `super`, and `thisContext`. They are called pseudo-variables because they are predefined and cannot be assigned to. `true`, `false`, and `nil` are constants, while the values of `self`, `super`, and `thisContext` vary dynamically as code is executed.

- `true` and `false` are the unique instances of classes `True` and `False` which are the subclasses of class `Boolean`. See Chapter : Basic Classes for more details.
- `self` always refers to the receiver of the message and denotes the object in which the corresponding method will be executed. Therefore, the value of `self` dynamically changes during the program execution, but can not be assigned in the code.
- `super` also refers to the receiver of the message too, but when you send a message to `super`, the method-lookup changes so that it starts from the superclass of the class containing the method that sends message to `super`. For further details see Chapter : The Pharo Object Model.
- `nil` is the undefined object. It is the unique instance of the class `UndefinedObject`. Instance variables, class variables and local variables are, by default, initialized to `nil`.
- `thisContext` is a pseudo-variable that represents the top frame of the execution stack and gives access to the current execution point. `thisContext` is normally not of interest to most programmers, but it is essential for implementing development tools such as the debugger, and it is also used to implement exception handling and continuations.

1.3 Messages and message sends

As we described, there are three kinds of messages in Pharo with predefined precedence. This distinction has been made to reduce the number of mandatory parentheses.

Here we give a brief overview on message kinds and ways for sending and executing them, while more detailed description is provided in Chapter : Understanding messages.

1. *Unary* messages take no argument. `1 factorial` sends the message `factorial` to the object 1. Unary message selectors consist of alphanumeric characters, and start with a lower case letter.
2. *Binary* messages take exactly one argument. `1 + 2` sends the message `+` with argument 2 to the object 1. Binary message selectors consist of one or more characters from the following set: `+ - / * ~ < > = @ % | & ? ,`
3. *Keyword* messages take an arbitrary number of arguments. `2 raisedTo: 6 modulo: 10` sends the message consisting of the message selector `raisedTo:modulo:` and the arguments 6 and 10 to the object 2. Keyword message selectors consist of a series of alphanumeric keywords, where each keyword starts with a lower-case letter and ends with a colon.

Message precedence

Unary messages have the highest precedence, then binary messages, and finally keyword messages, while brackets can be used to change the evaluation order.

Thus, in the following example we first send `factorial` to 3 which will give us result 6. Afterwards we send `+ 6` to 1 which gives the result 7, and finally we send `raisedTo: 7` to 2.

```
2 raisedTo: 1 + 3 factorial
>>> 128
```

Precedence aside, for the messages of the same kind, execution is strictly from left to right. Hence, as we have two binary messages, the following example return 9 and not 7.

```
1 + 2 * 3
>>> 9
```

Parentheses must be used to alter the order of evaluation as follows:

```
1 + (2 * 3)
>>> 7
```

1.4 Sequences and cascades

All expressions may be composed in sequences separated by period, while message sends may be also composed in cascades by semi-colons. A period separated sequence of expressions causes each expression in the series to be evaluated as a separate *statement*, one after the other.

```
Transcript cr.
Transcript show: 'hello world'.
Transcript cr
```

This will send `cr` to the `Transcript` object, then send to Transcript the message `show: 'hello world'`, and finally send it another `cr`, again.

When a series of messages is being sent to the *same* receiver, then this can be expressed more succinctly as a *cascade*. The receiver is specified just once, and the sequence of messages is separated by semi-colons as follows:

```
Transcript
  cr;
  show: 'hello world';
  cr
```

This cascade has precisely the same effect as the sequence in the previous example.

1.5 Method syntax

Whereas expressions may be evaluated anywhere in Pharo (for example, in a playground, in a debugger, or in a browser), methods are normally defined in a browser window, or in the debugger. Methods can also be filed in from an external medium, but this is not the usual way to program in Pharo.

Programs are developed one method at a time, in the context of a given class. A class is defined by sending a message to an existing class, asking it to create a subclass, so there is no special syntax required for defining classes.

Here is the method `lineCount` defined in the class `String`. The usual *convention* is to refer to methods as `ClassName>>methodName`. Here the method is then `String>>lineCount`. Note that `ClassName>>methodName` is not part of the Pharo syntax just a convention used in books to clearly define a method within a class in which it is defined.

```
String >> lineCount
  "Answer the number of lines represented by the receiver, where
    every cr adds one line."

  | cr count |
  cr := Character cr.
  count := 1 min: self size.
  self do: [:c | c == cr ifTrue: [count := count + 1]].
  ^ count
```

Syntactically, a method consists of:

1. the method pattern, containing the name (i.e., `lineCount`) and any parameters (none in this example),
2. comments which may occur anywhere, but the convention is to put one at the top that explains what the method does,
3. declarations of local variables (i.e., `cr` and `count`), and
4. any number of expressions separated by dots (here there are four).

The execution of any expression preceded by a ^ (a caret or upper arrow, which is `Shift-6` for most keyboards) will cause the method to exit at that point, returning the value of the expression that follows the ^. A method that terminates without explicitly returning value of some expression will implicitly return `self` object.

Parameters and local variables should always start with lower case letters. Names starting with upper-case letters are assumed to be global variables. Class names, like `Character`, for example, are simply global variables referring to the object representing that class.

1.6 Block syntax

Blocks (lexical closures) provide a mechanism to defer the execution of expressions. A block is essentially an anonymous function with a definition context. A block is executed by sending it the message `value`. The block answers the value of the last expression in its body, unless there is an explicit return (with ^) in which case it returns the value of the returned expression.

```
[ 1 + 2 ] value
>>> 3
```

```
[ 3 = 3 ifTrue: [ ^ 33 ]. 44 ] value
>>> 33
```

Blocks may have parameters each of which is declared with a leading colon. A vertical bar separates the parameters declaration from the body of the block. To evaluate a block with one parameter, you must send it the message `value:` with one argument. A two-parameter block must be evaluated by sending `value:value:` with two arguments, and so on, up to 4 arguments.

```
[ :x | 1 + x ] value: 2
>>> 3
```

```
[ :x :y | x + y ] value: 1 value: 2
>>> 3
```

If you have a block with more than four parameters, you must use `valueWithArguments:` and pass the arguments in an array. However, a block with a large number of parameters is often a sign of a design problem.

In blocks there may be also declared local variables, surrounded by vertical bars, just like local variable declarations in a method. Local variables are declared after arguments and vertical bar separator, and before the block body. In the following example, `x y` are parameters, and `z` is local variable.

```
[ :x :y |
  | z |
  z := x + y.
  z ] value: 1 value: 2
>>> 3
```

Blocks are actually lexical closures, since they can refer to variables of the surrounding environment. The following block refers to the variable `x` of its enclosing environment:

```
| x |
x := 1.
[ :y | x + y ] value: 2
>>> 3
```

Blocks are instances of the class `BlockClosure`. This means that they are objects, so they can be assigned to variables and passed as arguments just like any other object.

1.7 Conditionals and loops

Pharo offers no special syntax for control constructs. Instead, these are typically expressed by sending messages to booleans, numbers and collections, with blocks as arguments.

Some conditionals

Conditionals are expressed by sending one of the messages `ifTrue:`, `ifFalse:` or `ifTrue:ifFalse:` to the result of a boolean expression. See Chapter : Basic Classes, for more about booleans.

```
(17 * 13 > 220)
  ifTrue: [ 'bigger' ]
  ifFalse: [ 'smaller' ]
>>>'bigger'
```

Some loops

Loops are typically expressed by sending messages to blocks, integers or collections. Since the exit condition for a loop may be repeatedly evaluated, it should be a block rather than a boolean value. Here is an example of a very procedural loop:

```
n := 1.
[ n < 1000 ] whileTrue: [ n := n*2 ].
n
>>> 1024
```

`whileFalse:` reverses the exit condition.

```
n := 1.
[ n > 1000 ] whileFalse: [ n := n*2 ].
n
>>> 1024
```

`timesRepeat:` offers a simple way to implement a fixed number of iterations through the loop body:

```
n := 1.
10 timesRepeat: [ n := n*2 ].
n
>>> 1024
```

We can also send the message `to:do:` to a number which then acts as the initial value of a loop counter. The two arguments are the upper bound, and a block that takes the current value of the loop counter as its argument:

```
result := String new.
1 to: 10 do: [:n | result := result, n printString, ' '].
result
>>> '1 2 3 4 5 6 7 8 9 10 '
```

High-order iterators

Collections comprise a large number of different classes, many of which support the same protocol. The most important messages for iterating over collections include `do:`, `collect:`, `select:`, `reject:`, `detect:` and `inject:into:`. These messages represent high-level iterators that allow one to write very compact code.

An **Interval** is a collection that lets one iterate over a sequence of numbers from the starting point to the end. `1 to: 10` represents the interval from 1 to 10. Since it is a collection, we can send the message `do:` to it. The argument is a block that is evaluated for each element of the collection.

```
result := String new.
(1 to: 10) do: [:n | result := result, n printString, ' '].
result
>>> '1 2 3 4 5 6 7 8 9 10 '
```

`collect:` builds a new collection of the same size, transforming each element. You can think of `collect:` as the Map in the MapReduce programming.

```
(1 to:10) collect: [ :each | each * each ]
>>> #(1 4 9 16 25 36 49 64 81 100)
```

`select:` and `reject:` build new collections, each containing a subset of the elements of the iterated collection that satisfies, or not, respectively, the boolean block condition.

`detect:` returns the first element in the collection that satisfies the condition.

Don't forget that strings are also collections (of characters), so you can iterate over all the characters.

```
'hello there' select: [ :char | char isVowel ]
>>> 'eoee'
```

```
'hello there' reject: [ :char | char isVowel ]
>>> 'hll thr'
```

```
'hello there' detect: [ :char | char isVowel ]
>>> $e
```

Finally, you should be aware that collections also support a functional-style fold operator in the `inject:into:` method. You can also think of it as the Reduce in the MapReduce programming model. This lets you generate a cumulative result using an expression that starts with a seed value and injects each element of the collection. Sums and products are typical examples.

```
(1 to: 10) inject: 0 into: [ :sum :each | sum + each]
>>> 55
```

This is equivalent to `0+1+2+3+4+5+6+7+8+9+10`.

More about collections can be found in Chapter : Collections.

1.8 Method annotations: Primitives and pragmas

In Pharo methods can be annotated too. Method annotation are delimitated by < and >. There are used for two main scenarios: execution specific metadata for the primitives of the language and metadata.

Primitives

In Pharo everything is an object, and everything happens by sending messages. Nevertheless, at certain points we hit rock bottom. Certain objects can only get work done by invoking virtual machine primitives. Such primitives are essential primitives since they cannot be expressed in Pharo.

For example, the following are all implemented as primitives: memory allocation (`new`, `new:`), bit manipulation (`bitAnd:`, `bitOr:`, `bitShift:`), pointer and integer arithmetic (+, -, <, >, *, /, =, ==...), and array access (`at:`, `at:put:`).

When a method with a primitive is executed, the primitive code is executed in place of the method. A method using such a primitive may include additional Pharo code, which will be executed only if the primitive fails (for the case the primitive is an optional one).

In the following example, we see the code for `SmallInteger>>+`. If the primitive fails, the expression `super + aNumber` will be evaluated and its value returned.

```
+ aNumber
  "Primitive. Add the receiver to the argument and answer with the
    result
  if it is a SmallInteger. Fail if the argument or the result is not
    a
  SmallInteger Essential No Lookup. See Object documentation
    whatIsAPrimitive."

  <primitive: 1>
  ^ super + aNumber
```

Pragmas

In Pharo, the angle bracket syntax is also used for method annotations called pragmas. Once a method has been annotated with a pragma, the annotations can be collected using a collection (see the class `PragmaCollector`).

1.9 Chapter summary

- Pharo has only six reserved identifiers known as pseudo-variables: `true`, `false`, `nil`, `self`, `super`, and `thisContext`.
- There are five kinds of literal objects: numbers (5, 2.5, 1.9e15, 2r111), characters ($a), strings (`'hello'`), symbols (`#hello`), and arrays (`#('hello' #hi)` or `{ 1 . 2 . 1 + 2 }`)
- Strings are delimited by single quotes, comments by double quotes. To get a quote inside a string, double it.
- Unlike strings, symbols are guaranteed to be globally unique.
- Use `#( ... )` to define a literal array at compile time. Use `{ ... }` to define a dynamic array at runtime. Note that `#(1+2) size >>> 3`, but `{12+3} size >>> 1`. To observe why, compare `#(12+3) inspect` and `{1+2} inspect`.
- There are three kinds of messages: unary (e.g., `1 asString`, `Array new`), binary (e.g., `3 + 4`, `'hi', ' there'`), and keyword (e.g., `'hi' at: 2 put: $o`)
- A cascaded message send is a sequence of messages sent to the same target, separated by semi-colons: `OrderedCollection new add: #calvin; add: #hobbes; size >>> 2`
- Local variables declarations are delimited by vertical bars. Use `:=` for assignment. `|x| x := 1`
- Expressions consist of message sends, cascades and assignments, evaluated left to right (and optionally grouped with parentheses). Statements are expressions separated by periods.
- Block closures are expressions enclosed in square brackets. Blocks may take arguments and can contain temporary variables. The expressions in the block are not evaluated until you send the block a value message with the correct number of arguments. `[ :x | x + 2 ] value: 4`
- There is no dedicated syntax for control constructs, just messages whose sends conditionally evaluate blocks.

Bibliography