Learning Object-Oriented Programming, Design and TDD with Pharo

Stéphane Ducasse

March 11, 2019

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 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 keywords, and there is no syntax 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 (sub-)classes are created by sending a message to their superclass.

1.1 Syntactic elements

Expressions are composed of the following building blocks:

1. The six reserved keywords, or *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

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

Syntax	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`	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>`	virtual machine primitive or annotation
`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`	expression separator (`.`)
`Transcript show: 'hello'; cr`	message cascade (`;`)

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 argument, 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. `Processor` is a global variable, an instance of the class `ProcessScheduler`.

The receiver. `self` is a keyword that refers to the object inside which the current method is executing. We call it "the receiver" because this object

has received the message that caused the method to execute. `self` is called a "pseudo-variable" since we cannot 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. 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 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 arrays. are defined by `#( )`, surrounding space-separated literals. Everything within the parentheses must be a compile-time constant. For example, `#(27 (true false) abc)` is a literal array of three elements: the integer 27, the compile-time array containing the two booleans, and the symbol `#abc`. (Note that this is the same as `#(27 #(true false) #abc)`.)

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

Comments. are enclosed in double quotes " ". "hello" is a comment, not a string, and is ignored by the Pharo compiler. Comments may span multiple lines.

Local variable definitions. Vertical bars `| |` enclose the declaration of one or more local variables in a method (and also in a block).

Assignment. `:=` assigns an object to a variable.

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.

Primitives. `< primitive: ... >` denotes an invocation of a virtual machine primitive. For example, `< primitive: 1 >` is the VM primitive for `SmallInteger`. Any code following the primitive is executed only if the primitive fails. The same syntax is also used for method annotations (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 message with an argument and whose selector looks like mathematical expressions (for example: `+`) sent to a receiver, and taking a single argument. In `3 + 4`, the receiver is 3, the message selector is `+`, and the argument is 4.

Keyword messages. They consist of multiple 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.

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

Sequences of statements. A period or full-stop (`.`) is the statement separator. Putting a period between two expressions turns them into independent statements.

Cascades. Semicolons (`;`) can be used to send a cascade of messages to a single receiver. In `Transcript show: 'hello'; cr` we first send the keyword message `show: 'hello'` to the receiver `Transcript`, and then we send the unary message `cr` to the same receiver.

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

1.2 Pseudo-variables

In Pharo, there are 6 reserved keywords, or 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 the Boolean classes `True` and `False`. See Chapter : Basic Classes for more details.
- `self` always refers to the receiver of the currently executing method.
- `super` also refers to the receiver of the current method, 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 uses `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 initialized to `nil`.

- `thisContext` is a pseudo-variable that represents the top frame of the execution stack. `thisContext` is normally not of interest to most programmers, but it is essential for implementing development tools like the debugger, and it is also used to implement exception handling and continuations.

1.3 Message sends

There are three kinds of messages in Pharo. This distinction has been made to reduce the number of mandatory parentheses.

1. *Unary* messages take no argument. `1 factorial` sends the message `factorial` to the object 1.
2. *Binary* messages take exactly one argument. `1 + 2` sends the message `+` with argument 2 to the object 1.
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.

Unary messages.

Unary message selectors consist of alphanumeric characters, and start with a lower case letter.

Binary messages.

Binary message selectors consist of one or more characters from the following set:

```
+ - / \ * ~ < > = @ % | & ? ,
```

Keyword message selectors.

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, so:

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

First we send `factorial` to 3, then we send `+ 6` to 1, and finally we send `raisedTo: 7` to 2. Recall that we use the notation expression `-->` to show the result of evaluating an expression.

Precedence aside, execution is strictly from left to right, so:

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

return 9 and not 7. Parentheses must be used to alter the order of evaluation:

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

Periods and semi-colons.

Message sends may be composed with periods and semi-colons. A period separated sequence of expressions causes each expression in the series to be evaluated as a *statement*, one after the other.

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

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

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:

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

This has precisely the same effect as the previous example.

1.4 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` 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.

```
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 arguments (none in this example)
2. comments (these 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 that expression. A method that terminates without explicitly returning some expression will implicitly return `self`.

Arguments 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.

1.5 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 take parameters, each of which is declared with a leading colon. A vertical bar separates the parameter declaration(s) 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 sent `value:value:`, 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. (A block with a large number of parameters is often a sign of a design problem.)

Blocks may also declare local variables, which are surrounded by vertical bars, just like local variable declarations in a method. Locals are declared after any arguments:

```
[ :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.

1.6 Some conditionals

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.

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'
```

1.7 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 iteration:

```
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 '
```

1.8 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 define 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 model).

```
(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 satisfying (or not) the boolean block condition.

`detect:` returns the first element satisfying 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.

1.9 Primitives and pragmas

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.

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:`).

Primitives are invoked with the syntax `<primitive: aNumber>`. A method that invokes such a primitive may also include Pharo code, which will be executed only if the primitive fails.

Here we see the code for `SmallInteger>>+`. If the primitive fails, the expression `super + aNumber` will be executed and 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
```

In Pharo, the angle bracket syntax is also used for method annotations called pragmas.

1.10 Chapter summary

- Pharo has only six reserved identifiers (also called 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. Use `{ ... }` to define a dynamic array.

```
#(1+2) size
>>> 3
{1+2} size
>>>1
```

- 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 are declared with 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 that conditionally evaluate blocks.

Bibliography