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

Pharo's syntax overview

Pharo syntax is minimal: The complete syntax fits in one postcard. Pharo syntax is based on a couple of constructs such as *variable definition*, ' *assignment*, *return*, *sequence*, and two main construct used systematically: *messages* and *blocks*. This chapter provides some basic introduction to the key points. There will be not much more but the reader willing to get a deep and precise presentation of syntax should read the corresponding chapters in Pharo By Example available at http://books.pharobyexample.org

1.1 Minimal

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 a couple a special variables, 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.

Although Pharo's message syntax is simple, it is unconventional. This chapter offers some guidance to help you get acclimatized to this special syntax for sending messages. If you already feel comfortable with the syntax, you may choose to skip this chapter, or come back to it later.

1.2 Literal objects

In Pharo, most objects are created by sending the message `new` to a class. There are also some objects that are directly created by the compiler.

Comments

Comments are delimited by double quotes `"`.

```
"Hello"
```

Characters, Strings and Symbols

Characters short form creation is following the pattern dollar sign followed by the letter. The expression `$A` is the character `A`.

```
$A
>>> $A
```

Strings are delimited by single quotes `'`.

```
'Hello'
>>> 'Hello'
```

Strings are sequence of characters.

```
'Hello' first
>>> $A
```

We can concatenate strings using the message `,`.

```
'Hello ', 'Pharoers'
>>> 'Hello Pharoers'
```

Symbols are unique strings. They start with `#`.

```
#'Hello Pharoers'
>>> #'Hello Pharoers'
```

Booleans

There are two Booleans `true` and `false`.

```
true not
>>> false
```

Arrays

Arrays can be created as any other objects by sending the message `new` to the class `Array`. In addition, Pharo supports two other ways: `#( )` and `{ . }`.

The expression #(1 + 2 3) creates an array with four objects: 1, #+, 2, and 3.

```
#(1 + 2 3)
>>> #(1 + 2 3)
```

The expression {1 + 2} creates an array with one object resulting from the expression 1 + 2.

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

The expression {1 + 2 . 3 + 4} creates an array with two objects resulting from the execution of expressions separated by period.

```
{ 1 + 2 . 3 + 4}
>>> #(3 7)
```

1.3 Key expressions

In Pharo, except for a few constructs (:= ^ . ; # () {} [: |]), everything else is a message send. There is no operators or control flow constructs, just messages sent to objects. Therefore you can define a message named + in your class.

Variable definition and assignment

The expression | sum | declares a local variable. The expression sum := 0 changes the value of the variable sum to 0. The assignment construct is :=.

```
| sum |
sum := 0
```

Special variables

There are several special variables:

- nil is the is the undefined object. It is the default value of variables.
- 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.
- true and false are the unique instances of the Boolean classes True and False.
- self is the receiver of the message and executing method (See Chapter **??**).
- super is the receiver of the message and executing method (See Chapter **??**).

Returning an object

By default a method returns the receiver of the message. If we want to return a different object we use the construct `^`. The expression `^ sum` returns the object pointed by the variable `sum`.

```
| sum |
sum := 0.
sum := sum + 1.
^ sum
```

Sequence of expressions

`.` separates messages.

```
| a |
a := #(11 22 33).
a at: 2 put: 66.
a
>>>#(11 66 33)
```

1.4 Message structure

As most computation in Pharo is done by message passing, correctly identifying messages is key to avoiding future mistakes. The following terminology will help us:

- A message is composed of the message **selector** and the optional message arguments.
- A message is sent to a **receiver**.
- The combination of a message and its receiver is called a *message send* as shown in Figure 1-1.

A message is always sent to a receiver, which can be a single literal, a block or a variable or the result of evaluating another message.

1.5 Three kind of Messages

There are three kinds of message: *unary* (without any argument), *binary* (with two arguments) and *keywords* (with arguments).

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`.

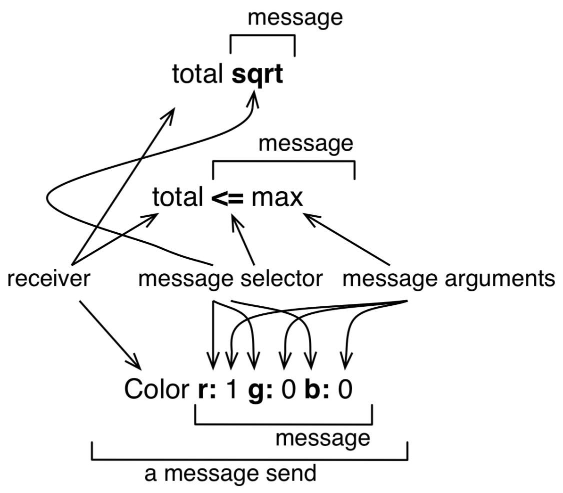

Figure 1-1 Message sends composed of a receiver, a method selector, and a set of arguments.

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`.

1.6 Unary messages

Unary message selectors consist of alphanumeric characters, and start with a lower case letter. Here is an example of unary message:

```
20 factorial
>>> 2432902008176640000
```

We send the unary message `factorial` to the number `20`. Unary means that there is no argument and that the selector is composed of letters.

1.7 Binary messages

Here is an example of binary message:

```
12 + 15
>>> 27
```

We send the binary message `+` to the number `12` with the argument `15`. A binary message is defined by a binary selector: a selector written without alphabetic characters, i.e., such `+-/*=~ < >`. Binary message selectors consist of one or more characters from the following set:

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

1.8 Keyword messages

Keyword messages are messages with arguments. Keyword message selectors consist of a series of alphanumeric keywords, where each keyword starts with a lower-case letter and ends with a colon.

For example to access the element of an array we send the message `at:` with as argument the index in the array. Arrays have their first element at index 1.

```
#(11 22 33) at: 2
>>>22
```

Another keyword-based message is the message `at:put:` to change the value of an array. Here we put the number 66 in the second position.

```
| a |
a := #(11 22 33).
a at: 2 put: 66.
a
>>>#(11 66 33)
```

Another example

```
1 between: 0 and: 3
>>> true
```

This message would expressed in C-like syntax as follows:

```
1.betweenAnd(0,3)
>>> true
```

1.9 Execution order and parentheses

The order in which messages are sent is determined by the type of message. There are just three types of messages: **unary**, **binary**, and **keyword messages**. Pharo distinguishes such three types of messages to minimize the number of parentheses.

- Unary messages are always sent first, then binary messages and finally keyword ones.
- As in most languages, parentheses can be used to change the order of execution.

These rules make Pharo code as easy to read as possible. And most of the time you do not have to think about the rules.

```
(4 + 4) * 5
>>> 40
```

```
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
```

1.10 Sending multiple messages to the same object

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.

1.11 Blocks

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
```

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.

1.12 Method syntax

Whereas expressions may be evaluated anywhere in Pharo (for example, in a playground, in a debugger, or in a browser), methods are defined in a class using a code browser, 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>>` is not part of the Pharo syntax just a convention used in books to clearly define a method. It means that when you define it you should just type the selector (here `lineCount`).

```
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.

1.13 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.

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

Note that the order can be reserved too.

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

1.14 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
```

The message `whileFalse:` reverses the exit condition.

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

The message `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'
```

1.15 Iterators

Collections comprise a large number of different classes, many of which support the same set of messages. 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 '
```

The message `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)
```

The messages `select:` and `reject:` build new collections, each containing a subset of the elements satisfying (or not) the boolean block condition.

The message `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
```

More about collections can be found in Pharo by Example.

1.16 Chapter summary

- Pharo has some reserved identifiers (also called pseudo-variables): `true`, `false`, `nil`, `self` and `super`.
- 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.
- 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:
- 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