Introduction to Ruby for YNelson Users

April 24, 2014

For `YNelson` users, basic Ruby syntax is necessary. This document is a primer to Ruby for `YNelson` users. This documents should be used in the same way as `YNelson` tutorial (Introduction to `YNelson`) – that is, get an *irb* session running, and type all the examples in by yourself. You might also wish to install `YNelson` by by typing "`gem install y_nelson`" from your command line. Since many of the code samples here rely heavily on each other, you may consider to do this document in one session. On the other hand, if you happen to be a Ruby hacker, you do not need to continue reading this document at all. This document is not a replacement for a Ruby textbook. For those, who want more thorough introduction to the language, I recommend the document http://www.rubyist.net/~slagell/ruby/index.html, or one of the many Ruby textbooks.

Variables and constants

In Ruby, everything[1] is an *object*. Objects can be assigned to *variables* or *constants*. Ruby constants must always start with capital letter. Variables starting with small letter are *local variables*. (Other types of variables are *instance variables*, *class variables* and *global constants*; this is not important at the moment.)

```
alpha = 1
#=> 1
beta = [1, 2]
#=> [1, 2]
Gamma = { x: 1, y: 2, z: 3 }
#=> {:x=>1, :y=>2, :z=>3}
```

You can check this using `defined?` operator:

```
defined? alpha
#=> "local-variable"
defined? Gamma
#=> "constant"
```

Code lines and comments

Comments are denoted by `#` sign. Anything on the line following the `#` sign is ignored:

```
puts “Hello world!” # this line prints the words “Hello world!”
```

Ruby lines can be written with or without semicolons:

[1]Almost everything. Non-object include eg. variables or argument fields.

```
a = "with";
b = "without"
puts [ a, b ].join " or "
```

Semicolon is compulsory only when two or more logical lines are crammed together like this:

```
a = “Hello“; b = “world!”; puts a + ’ ’ + b
```

Methods

Different classes respond to different *methods*, and respond to them differently:

```
beta.size
#=> 2
Gamma.size
#=> 3
Gamma.keys
#=> [:x, :y, :z]
Gamma.values
#=> [1, 2, 3]
beta.keys
#=> NoMethodError: undefined method ‘keys’ for [1, 2]:Array
```

Methods can be defined by `def` keyword:

```
def average( a, b )
  ( a + b ).to_f / 2
end
#=> nil
average( 2, 3 )
#=> 2.5
```

In the code example above, ’`to_f`’ method performs conversion of an integer into a floating point number, which is not important.

Classes

Every object belongs to some *class* (object type):

```
alpha.class
#=> Fixnum
beta.class
#=> Array
Gamma.class
#=> Hash
```

New classes can be defined with `class` keyword. The methods defined inside the class will become the *instance methods* of that class:

```
class Dog
  def speak!
    puts "Bow wow!"
  end
end
#=> nil
Spot = Dog.new
#=> #<Dog:0x9c214ac>
Spot.speak!
#=> Bow wow!
class Cat
  def speak!
    puts "Meow"
  end
end
#=> nil
Lisa = Cat.new
#=> #<Cat:0x98efb80>
Lisa.speak!
#=> Meow
```

These two classes now represent respectively dogs and cats in your irb session. In the code above, you could notice '`new`' method, used to create instances from the defined classes, and '`puts`' method, used to simply print characters on the screen.

Strings, Symbols, Arrays and Hashes

For `YPetri` users, it will be especially necessary to learn more about *strings*, *symbols*, *arrays*, *hashes*, and how to define and read *closures* (aka. *anonymous functions*). Strings and symbols are among the most basic Ruby objects, while arrays and hashes are important in understanding *argument passing* to methods and closures. **Understanding argument passing and closure writing is essential in using YPetri DSL.**

Strings

A string is simply a sequence of characters, which can be defined using single or double quotes (`'` or `"`):

```
my_string = 'Hello world!'
#=> "Hello world!"
my_string.class
#=> String
```

Strings are mutable (can be changed):

```
my_string.object_id
#=> 81571950
7.times do my_string.chop! end
#=> 7
my_string
#=> "Hello"
my_string.object_id
#=> 81571950
```

Above, you can newly notice `times` method, `do ... end` block, and `chop!` method that removes the last character from `my_string` 7 times, until only `"Hello"` remains. But the important thing is that as `object_id` method shows, `my_string` is still the same object (same *object id*), although the contents is changed.

```
my_string << "Spot!"
#=> "Hello Spot!"
my_string.object_id
#=> 81571950
```

Again, << operator changed the contents, but the object id remained the same.

Symbols

Unlike strings, symbols are immutable – they never change. They are written with colon (`:`):

```
:Spot.class
#=> Symbol
```

Arrays

As seen earlier, they can be defined with square brackets `[]`. Square brackets are also used to address the array elements, counting from 0.

```
my_array = [ Spot, Lisa ]
#=> [#<Dog:0x9c214ac>, #<Cat:0x98efb80>]
my_array[0]
#=> #<Dog:0x9c214ac>
```

Negative numbers can be used to address the elements from the end of the array:

```
my_array[-1]
#=> #<Cat:0x98efb80>
my_array[-2]
#=> #<Dog:0x9c214ac>
```

Hashes

As for hashes, there are two ways of defining them. The first way uses *Ruby rocket* (`=>`):

```
h1 = { Spot => "dog", Lisa => "cat" }
#=> {#<Dog:0x9c214ac>=>"dog", #<Cat:0x98efb80>=>"cat"}
h1[ Lisa ]
#=> "cat"
h1[ Spot ]
#=> "dog"
```

The second way is possible only when the keys are symbols. It is done by shifting the colon to the right side of the symbol:

```
h2 = { dog: Spot, cat: Lisa }
#=> {:dog=>#<Dog:0x9c214ac>, :cat=>#<Cat:0x98efb80>}
h2[:dog]
#=> #<Dog:0x9c214ac>
```

Code blocks and Closures

Code blocks, or simply *blocks*, are pieces of code enclosed by `do` / `end` pair, or by curly brackets `{}`. Code blocks can be passed to methods:

```
[1, 2, 3, 4].map { |n| n + 3 }
#=> [4, 5, 6, 7]
my_array.each do |member| member.speak! end
#=> Bow wow!
    Meow
```

In the first case, '`map`' method was passed a block specifying addition of 3. In the second case, '`each`' method was passed a block calling `speak!` method on the array elements. Please note the pipe, or vertical line charecters (|), that delimit the block arguments (both blocks above happen to have only one argument). Code blocks can be understood as anonymous functions – a way of specifying an operation, when one does not want to write a method for it. Their semantics corresponds to *lambda calculus*.

Return values

Code blocks (and actually, all Ruby statements) have return value. With code blocks, the return value will typically be the last statement:

```
[1, 2, 3, 4].map { |v|
  v + 3    # this value will be ignored
  v - 1    # last value of the block will be returned
}
#=> [0, 1, 2, 3]
```

Closures

A block packaged for future use is called a *closure*. Ruby closures come in two flavors: `proc` and `lambda`. They are created by passing a block to the `proc` / `lambda` keyword:

```
my_proc = proc do |organism| organism.speak! end
#=> #<Proc:0x952674c@(irb):136>
my_lambda = lambda do |organism| organism.speak! end
#=> #<Proc:0x942faf0@(irb):137 (lambda)>
```

Once defined, they can be reused in code. Notice the ampersand (&) indicating block reuse:

```
my_array.each &my_proc
#=> Bow wow!
    Meow
my_array.each &my_lambda
#=> Bow wow!
    Meow
```

Closures can also be called alone, a little bit like methods:

```
my_proc.call( Spot )
#=> Bow wow!
my_lambda.call( Lisa )
#=> Meow
```

Instead of `call` keyword, you can just use dot before the parenthesis to call closures:

```
my_proc.( Lisa )
#=> Meow
my_lambda.( Spot )
#=> Bow wow!
```

Differences between `proc` and `lambda` closures are minor. For `YNelson` users, the most noticeable difference will be, that `proc` less finicky about its arguments than `lambda`:

```
my_proc.( Lisa, "garbage" )
#=> Meow
my_lambda.( Lisa, "garbage" )
#=> ArgumentError: wrong number of arguments (2 for 1)
```

Finally, let us notice the alternative syntax for defining lambdas:

```
my_lambda = lambda do |organism| organism.speak! end
my_lambda = lambda { |organism| oranism.speak! }
my_lambda = -> organism do organism.speak! end
my_lambda = -> orgnism { organism.speak! }
```

All of the four above statements define exactly the same thing.

Passing arguments

Earlier, we have defined method `average`, expecting two arguments. If wrong number of arguments is supplied, `ArgumentError` will ensue:

```
average( 3, 5 )
#=> 4
average( 3, 5, 8 )
#=> ArgumentError: wrong number of arguments (3 for 2)
```

Obviously, this is not a very nice behavior when it comes to averages. It is a general situation, that when calling more advanced methods, we need to modify their behavior, or pass more complicated structures to them. This is seen eg. with `YNelson::Transition` constructors, and will be further encountered in `YCell` and `YChem` DSLs. Furthermore, `YNelson` users have to be able to write their own closures, because that is how *functions* of *functional transitions* are specified. In other words, `YNelson` **users have to master argument passing from both user and programmer side**. There is no way around this. With functional Petri nets, one cannot avoid writing functions. It is possible to avoid using `YNelson`, but it is not possible to avoid learning to write functions. Every simulator of functional Petri nets brings with itself some sort of function language, which one has to learn. With `YNelson`, this is the language of Ruby closures.

Optional arguments

Arguments with prescribed default value are optional. Let us write an improved `average` method that can accept either 2 or 3 arguments:

```
def average( a, b, c=nil )
  # If c argument was not given, it will default to nil
  if c == nil then
    ( a + b ).to_f / 2
```

```
  else
    ( a + b + c ).to_f / 3
  end
end
#=> nil
average( 3, 5 )
#=> 4
average( 3, 5, 8 )
#=> 5.333333333333333
average( 1, 2, 3, 4 )
#=> ArgumentError: wrong number of arguments (4 for 3)
```

The default value for `c` argument is prescribed using single equals sign (`=`). Apart from that, you can notice `if` ... `then` ... `else` ... `end` statement, which needs no explanation, equality test (double equals sign, `==`), used to test whether `c` contains `:pochi` symbol (indicating missing value), and comment character (octothorpe aka. sharp, `#`). Comment character `#` causes all characters until the end of the line to be ignored by Ruby. All code lines, exception the obvious ones, should have comments.

Variable-length argument lists

We will now improve our `average` method, so that it can calculate averages of any number of arguments. For this, we will use asterisk (`*`) syntactic modifier, also known as *splash*. The asterisk will cause a method to collect the arguments into an array. Let's try it out first:

```
def examine_arguments( x, *aa )
  puts "x is a #{x.class}."
  puts "aa is #{aa.class} of #{aa.size} elements."
end
#=> nil
```

Method examine arguments takes one normal argument (`x`), and collects the rest of the arguments into an array (`aa`), thanks to the splash modifier. (Apart from that, you can notice string interpolation using `#{ ... }` notation in the above code.) Then it prints the class of `x`, class of `aa` (which should be an array), and the number of elements after `x`.

```
examine_arguments( 1 )
#=> x is a Fixnum.
    aa is Array of 0 elements.
    nil
examine_arguments( :hello, nil, 3, 5, "garbage" )
#=> x is a Symbol.
    aa is Array of 4 elements.
    nil
```

With this, we can go on to define our improved average method:

```
def average( *aa )
  aa.reduce( :+ ).to_f / aa.size
end
#=> nil
average 3, 5, 7, 11
#=> 6.5
```

You can also newly notice `reduce( :+ )` method, used to calculate the sum of the `aa` array. To also practice closures, let us define a lambda doing the same as the `average` method above:

```
avg = lambda { |*aa| aa.reduce( :+ ).to_f / aa.size }
#=> #<Proc:0x9dbd220@(irb):208 (lambda)>
avg.( 11, 7, 5, 3 )
#=> 6.5
```

Named arguments

The main purpose of named arguments is to make the interface (or DSL) easier to remember, and the code easier to read. Easy-to-read code is a crucial requirement for scalable development. In Ruby methods, named arguments can be specified as hash pairs in the method call:

```
def density( x: 1, y: 1, z: 1, weight: 1 )
  weight.to_f / ( x * y * z )
end
#=> nil
density( x: 2, y: 2, z: 2, weight: 10 )
#=> 1.25
```

The above method calculates mean density of boxes of certain height, width, length and weight. Double splash (`**`) can be used to collect all the options in a hash. Let's use it to define a closure that does exactly the same thing as the method `density` we have just defined, in a slightly different way:

```
dens_closure = -> **nn do
  nn[:weight].to_f / ( nn[:x] * nn[:y] * nn[:z] ) end
#=> #<Proc:0x9a5d60c@(irb):241 (lambda)>
dens_closure.( x: 2, y: 2, z: 2, weight: 10 )
#=> 1.25
```

Above, note the alternative syntax for lambdas: `-> arg do ... end` is the same as `lambda do |arg| ... end`. Having hereby introduced the named arguments, let us notice hash-collecting behavior for square bracket (`[]`) array constructor syntax.

Hash-collecting behavior of square brackets

In more complicated method argument structures, it can be advantageous to take use of the hash-collecting by square brackets. It is normal for curly braces to create hashes:

```
h = { x: 2, y: 3, z: 4 }
#=> {:x=>2, :y=>3, :z=>4}
h.class
#=> Hash
```

However, square brackets, that generally create arrays, are also able to collect hashes just like the argument fields with named arguments:

```
a0 = [ 1, 2, 3 ]
#=> [1, 2, 3]
a0.class
#=> Array
```

```
a1 = [ 1, 2, 3, x: 2, y: 3, z: 4 ]
#=> [1, 2, 3, {:x=>2, :y=>3, :z=>4}]
a1.class
#=> Array
a1.map &:class
#=> [Fixnum, Fixnum, Fixnum, Hash]
a1[-1]
#=> {:x=>2, :y=>3, :z=>4}
```

In other words, if there are any trailing `key/value` pairs inside square brackets, they will be collected into a hash, which will become the last element of the array. This possibility to mix ordered elements with `key/value` pairs is used eg. in `YCell` `enzyme` constructor method.

Arity

Every closure and every method has arity, which is basically the number of input arguments. (Closures with 0 arguments are null*ary*, with 1 argument un*ary*, with 2 arguments bin*ary*, with 3 arguments tern*ary* etc. – therefrom *arity*.)

```
doubler = lambda { |a| a * 2 }
#=> #<Proc:0xa19b5b8@(irb):1 (lambda)>
doubler.call( 3 )
#=> 6
doubler.arity
#=> 1
adder = -> p, q { p + q }
#=> #<Proc:0xa27d940@(irb):6 (lambda)>
adder.call( 5, 6 )
#=> 11
adder.arity
#=> 2
scaler = -> number, p, q { number * (q / p) }
#=> #<Proc:0xa2825e4@(irb):7 (lambda)>
scaler.call( 10, 2, 3 )
#=> 15
scaler.arity
#=> 3
constant_function = -> { 42 }
#=> #<Proc:0xa2825e4@(irb):7 (lambda)>
constant_function.call
#=> 42
constant_function.arity
#=> 0
```

Closures / methods with variable length arguments indicate this by reporting negative arity:

```
summation = -> *array { array.reduce( :+ ) }
#=> #<Proc:0xa296ddc@(irb):9 (lambda)>
summation.call( 1, 2, 3, 4 )
#=> 10
summation.arity
#=> -1
```

```
array_scale = -> *a, coeff { a.map { |e| e * coeff } }
#=> #<Proc:0xa2a9edc@(irb):12 (lambda)>
array_scale.call( 1, 2, 3, 4, 7 )
#=> [7, 14, 21, 28]
array_scale.arity
#=> -2
```

Return value

The last statement in a closure or method becomes the return value. In methods and lambda-type closures, return statement can also be used explicitly:

```
divider = -> u, v {
            if v == 0 then
              return :division_by_zero # explicit return
            end
            u / v # implicit return - last statement
          }
#=> #<Proc:0xa21e878@(irb):15 (lambda)>
divider.call( 15, 3 )
#=> 5
divider.call( 15, 0 )
#=> :division_by_zero
experimental_closure = proc {
                         42     # ignored
                         41     # returned
                       }
#=> #<Proc:0xa249460@(irb):28>
experimental_closure.call
#=> 41
experimental_lambda = lambda {
                        1          # ignored
                        return 3   # returned
                        7          # never executed
                      }
#=> #<Proc:0xa3200dc@(irb):38 (lambda)>
experimental_lambda.call
#=> 3
```

Return value arity

It is possible to return more than one value[2]. For example:

```
mult_table = -> number {
                         [1, 2, 3, 4, 5].map { |e| e * number }
                       }
#=> #<Proc:0xa36a0d8@(irb):55 (lambda)>
```

This method returns 5 values. We can receive them by using a simultaneous assignment statement:

[2]Technically, methods and closures always return exactly 1 object – multiple values are returned via a single array object. But pragmatically, and especially with respect to `YPetri`, the notion of return value arity is useful.

```
by_one, by_two, by_three, by_four, by_five = mult_table.call( 7 )
#=> [7, 14, 21, 28, 35]
by_one
#=> 7
by_two
#=> 14
by_five
#=> 35
```

Or we can simply collect them in an array:

```
collection = mult_table.( 3 )
#=> [3, 6, 9, 12, 15]
```

In `YNelson`, it sometimes becomes necessary to write closures with higher return arity (returning more than one value). This is normally done by returning an array. Also, lambda return statement can be used to return multiple values:

```
constant_vector = -> { return 1, 2, 3 }
#=> #<Proc:0xa3cb338@(irb):72 (lambda)>
x, y, z = constant_vector.call
#=> [1, 2, 3]
x
#=> 1
y
#=> 2
z
#=> 3
```

`YSupport` library

Finally, having introduced the basic Ruby syntax, let us mention `YSupport` gem (gem = published Ruby library), that collects the assets (modules, classes, methods...) of general concern in use by `YPetri`/`YNelson`. Of these, a particular mention goes to `NameMagic`, widely used in `YPetri`, `YNelson` and `SY` (physical units) libraries.

`NameMagic`

In software engineering, *magic* is a technical term for irregular side effects of language expressions. The problem that `NameMagic` solves is, that objects (such as chemical species encoded in `YNelson`) are frequently named, and naming them is an annoying chore. Consider a simple case:

```
class Student
  attr_accessor :name
  def initialize name: nil
    @name = name
  end
end
```

Now, to create named `Student` instances, one has to mention `:name` named argument in the constructor, and frequently, the same name has to be mentioned twice, such as when assigning to constants or variables:

```
richard = Student.new( name: "Richard" )
richard.name
#=> "Richard"
```

In Ruby, we can notice that some objects have built-in capability to be named simply by constant assignment:

```
foo = Class.new
foo.name
#=> nil
Car = foo
foo.name
#=> "Car"
```

Magically, upon assigning `Car = foo`, the object referred to by the `foo` variable received an attribute `name`, with value set to "`Car`". This standard behavior is termed *constant magic*. `NameMagic` mixin (part of `YSupport`) extends this standard behavior to any chosen object, and also takes care of keeping the instance registry and doing general naming related chores for its includers:

```
require 'y_support/name_magic'
class Chemical
  include NameMagic
end
NaCl = Chemical.new
NaCl.name
#=> "NaCl"
```

It might seem like a small thing, but in a big file full of complicated statements, it really matters whether you have to write each time "`NaCl = Chemical.new( name:  NaCl )`", or just "`NaCl = Chemical.new`". `NameMagic` is a part of `YSupport` library accompanying `YPetri` and `YNelson`. You can install `YSupport` from the command line by "`gem install y_support`".

Other essential concepts

There are a few more essential concepts of Ruby that `YNelson` users should be familiar with, such as namespaces and parametrized subclassing. Code examples in this section are slightly more complicated, and also, they make use of `YSupport` gem. Install `YSupport` by typing `gem install y_support` in your command line before studying code examples in this section.

Namespaces

In Ruby, namespaces are known as modules (objects of `Module` class). These objects are containers for constants and method definitions. For example, let us imagine that we want to define constants, classes and methods related to the game of chess. We could simply define them in the command line, without any considerations, We could do it directly, but that way, all of them would be defined in the root of Ruby namespace – on `Object` class. The reason why this is not a good idea is the same as the reason why it is not a good idea to put all your files in the root of your filesystem. Chess-related terms such as `Field` or `Piece` could collide with concepts from other domains not related to chess. For that reason, we will collect all the chess-related assets into a single namespace:

```
module Chess
  class Board < Array
    SIZE = 8          # standard chessboard
```

```
    class Field      # chessboard field
      attr_accessor :contents
    end
    def self.new     # constructs 8 × 8 array of arrays
      super( SIZE, Array.new( SIZE ) { Field.new } )
    end
    # chessboard is defined here
  end

  Piece = Class.new         # chess piece
  Pawn = Class.new Piece    # chess pawn
  Knight = Class.new Piece  # chess knight
  Rook = Class.new Piece    # chess rook
  # etc.
end
```

We then access the contents of the namespace in the way similar to the way we address the files in the filesystem:

```
Chess::Board        # namespace Chess, constant Board
Chess::Piece        # namespace Chess, constant Piece
Chess::Pawn         # namespace Chess, constant Pawn
Chess::Board::SIZE  # namespace Chess::Board, constant SIZE
Chess::Board::Field # namespace Chess::Board, constant Field
# etc.
```

Let us note that in the above example, `Board`, `Piece`, `Pawn` are merely constants of the namespace `Chess`. Similarly, in `YPetri`, when talking about `YPetri::Place`, `YPetri::Transition` or `YPetri::Net`, it means constants `Place`, `Transition` and `Net` belonging to the module `YPetri` and containing the relevant class objects. But each of these classes is a namespace of its own, that can have constants defined on it. For example, `YPetri::Simulation` has constants `YPetri::Simulation::PlaceRepresentation` and `YPetri::Simulation::TransitionRepresentation`, representing copies of the net's places and transitions when executed inside a `Simulation` instance.

Parametrized subclassing

One of the core techniques used in `YNelson` / `YPetri` domain model is parametrized subclassing. Literature on the topic does exist [Roberts et al., 1996], but again, the concept is best explained on examples:

```
require 'y_support/all'
class Human
  include NameMagic  # allows humans to be named easily
end
```

Humans generally live in settlements. Let us create class `Village` representing settlements.

```
class Village
  include NameMagic  # allows villages to be named easily
end
```

At this point, we are standing in front of the problem of making humans associated with their settlements. One way to do it is to make each `Human` instance remember which settlement they belong to. This approach, which you can certainly imagine well even without demonstration, is in common use. But we have a more powerful approach at our disposal – subclassing. This is how we can define a subclass of humans living in London:

```
London = Village.new
class Londoner < Human             # using < sign
  def self.settlement; London end # let the class know its city
end
John = Londoner.new
John.class.settlement #=> London
```

To make it easier to ask humans about their settlement, let's reopen class `Human` and delegate method `#settlement` to the class:

```
class Human
  def settlement; self.class.settlement end
end
```

Alternative syntax for subclassing is this:

```
Dublin = Village.new
Dubliner = Class.new Human do     # using argument to Class.new
  def self.settlement; Dublin end
end
Finnegan = Dubliner.new
Finnegan.class.settlement #=> Dublin
```

Simply, each settlement has its own class of humans – its inhabitants. But since there are many settlements, it is inconvenient to manually define the inhabitant class for each of them. We therefore make each village automatically construct its own subclass of `Human` and parametrize it with `settlement` attribute. `YSupport` supports parametrized subclassing with method `#param_class`, and makes it easy to construct a PS of `Human` for each `Village` istance.

```
class Village              # reopening the class defined earlier
  def initialize           # defining a constructor
    param_class( { Human: Human }, with: { settlement: self } )
  end
end
```

Each village has now its own PS of `Human`.

```
Stockholm, Riga, Canberra = 3.times.map { Village.new }
Stockholm.Human                # class of Stockholm citizens
Riga.Human                     # class of Riga citizens
Canberra.Human                 # class of Vilnius citizens
Stockholm.Human == Riga.Human #=> false
Fred = Stockholm.Human.new     # Stockholm citizen constructor
Fred.class.settlement          #=> Stockholm
```

We say that PS of `Human` class *depends* on `Village`. The advantage is that instances of the PS of `Human` don't need to be explicitly told which village do they belong to, and have easy access to the assets of its owner `Village` instance. The concept of parametrized subclassing is actually very simple.

Convenience methods

Convenience methods are methods in which the consistency of the behavior is traded for syntax sweetness. Convenience methods may do entirely different things for different argument sets. For example, in `YNelson`, (you can install it by "`gem install y_nelson`", if you haven't done so yet), `Place#marking` without arguments simply returns the place's marking. But with arguments, it can be used to define a guard:

```
require 'y_nelson' and include YNelson
A = Place marking: 42
A.marking # Simply returns its marking
# But with different arguments, same method can be used to
# define a guard.
A.marking "must never be above 100" do |m| m <= 100 end
A.marking = 99 # no problem
A.marking = 101 # YPetri::GuardError is raised
```

Depending on circumstances, use of convenience methods in reusable code may or may not be bad practice.

References

Don Roberts, Ralph Johnson, et al. Evolving frameworks: A pattern language for developing object-oriented frameworks. *Pattern languages of program design*, 3: 471-486, 1996.