Ruby

Variables
Constantes
Métodos
Operadores
Sentencias
Loops

Características

- Interpretado
- Tipado dinámico
- Todo es un objeto
- *Garbage collector*
- Open Source
- Manejo de excepciones
- Incorpora aspectos de programación funcional
    - tratar funciones como valores
    - crear y retornar funciones

Definición de variables

```
variable_name = value
```

```
month = 10
jan, feb, mar = 'January', 'February', 'March'
month = "Marzo"

birth_date = Date.new(10,3,1998)

success = true

amount = 100.50
tax = amount * 1.21

puts abc          # => undefined local variable or method 'abc'
```

Definición de constantes

```
CONST_NAME = value
```

```ruby
MONTH = 10
ENERO = 'January'

BIRTH_DATE = Date.new(10,3,1998)
BIRTH_DATE.month = 7
BIRTH_DATE = Date.new(19,6,2001)   # => warning: already initialized
                                        constant BIRTH_DATE
BIRTH_DATE.month=9
puts BIRTH_DATE    # => 19/9/2001

def method
 A = 1             # => dynamic constant assignment (SyntaxError)
end

puts Abc           # => uninitialized constant Abc (NameError)
```

Definición de métodos

```
def method_name(params)
   # code
end
```

```
def hello world
 puts 'Hello, world!'
end

hello_world

hello_world()
```

Definición de métodos

```
def hello(name)
 puts 'Hello, ' + name
end

hello 'Carlitos'

hello('Carlitos')

hello(nil)
```

Hello, Carlitos

in `+': no implicit conversion of nil into String (TypeError)

Métodos: valores por defecto

```
def hello(name= 'John Doe')
 puts 'Hello, ' + name
end

hello
```

Hello, John Doe

```
hello(nil)
```

in `+': no implicit conversion of nil into String (TypeError)

```
hello(10)
```

in `+': no implicit conversion of Integer into String (TypeError)

Métodos: paréntesis opcionales

```
def hello(name = 'Natalia', lastname = 'Natalia')
 puts "Hello, #{name} #{lastname}"
end

hello 'Mary', 'Smith'

hello( 'Mary', 'Smith') # más claro

hello('Mary')

hello
hello()
```

Métodos: argumentos variables

```
def foo(a, b, *v, c)

end

foo(1, 2, 3, 4, 5, 6) # a=1, b=2, v= [3,4,5], c = 6
foo(1,2,3)  # a=1, b=2, v= [], c = 3

foo(1,2)      => in 'foo': wrong number of arguments
                 (given 2, expected 3+) (ArgumentError)
```

Métodos: valor de retorno

Si el valor se retorna con la última instrucción, omitir el return

```
# bad
def some_method(some_arr)
  return return_value
end

# good
def some_method(some_arr)
  return_value
end
```

Métodos

Las siguientes expresiones son equivalentes, ya que es opcional encerrar los parámetros entre paréntesis

```
obj.meth obj2.meth param
```

```
obj.meth (obj2.meth param)
```

Por claridad, encerrar los parámetros entre paréntesis

Métodos de instancia y de clase

```
class Human
 # Class method (a.k.a. static method)
 def self.classification
   'Mammal'
 end

 # Instance constructor
 def initialize(first_name, last_name)
   @my_first_name = first_name
   @my_last_name  = last_name
 end

 # Instance method
 def full_name
   "#{@my_first_name} #{@my_last_name}"
 end
end
```

https://gist.github.com/JamesMGreene/54d77bf9c786fb34b498

Métodos de instancia y de clase

Determinar la salida

```
class Father
 def self.default_make
   'Father'
 end

 def father_one
   self.class.default_make
 end

 def father_two
   Father.default_make
 end
end

class Son < Father
 def self.default_make
   'Son'
 end
end
```

```
a=Son.new
puts a.father_one
puts a.father_two
```

Son
Father

Métodos de instancia y de clase

```ruby
class Father
 def self.default_make
   'Father'
 end

 def father_one
   self.class.default_make
 end

 def father_two
   Father.default_make
 end
end

class Son < Father
 def self.default_make
   'Son'
 end
end
```

Determinar la salida

```ruby
a=Father.new
puts a.father_one
puts a.father_two
```

```
Father
Father
```

```ruby
a=Father.new
puts a.default_make
```

```
undefined method `default_make' for
#<Father:0x00007f9a0a035628> (NoMethodError)
```

Métodos privados

```
class Father
 ...

  private def aux_method
      # private code
  end

  def some_method(other)
    other.aux_method
  end

end


father = Father.new
father.aux_method
```

```
a = Father.new
b = Father.new

b.some_method a
```

private method `aux_method' called for #<Father:0x000000000473b7f8> (NoMethodError)

private method `aux_method'
called for #<Father:0x00000000045e0890> (NoMethodError)

Métodos privados

```
class Father
 ...

  private def priv_method
      # private code
  end

end

class Son < Father
 def self.default_make
   'Son'
 end

 def some_method
   priv_method
  end
end
```

Métodos “*protected*”

```ruby
class Father

 ...

 protected def protected_method
      # code
 end

 def father_method(other)
   other.protected_method
 end

end


father = Father.new
father.protected_method
```

```ruby
class Son < Father

 ...

 def some_method
   protected_method
 end

end
```

protected method `protected_method' called for #<Father:0x00000000045cbb20> (NoMethodError)

is_a, kind_of, instance_of

```ruby
class Point
 attr_accessor :x,:y
 def initialize(x,y)
   @x, @y = x, y
 end
end
```

```ruby
class Square
 attr_accessor :topLeft,:side
 def initialize(topLeft, side)
   raise 'Invalid top left point' until topLeft.instance_of? Point
   raise 'Invalid side' until side > 0
   @topLeft=topLeft
   @side=side
 end
end
```

```ruby
class Rectangle
 attr_accessor :topLeft,:height, :width
 def initialize(topLeft, height, width)
   raise 'Invalid top left point' until topLeft.instance_of? Point
   raise 'Invalid height' until height > 0
   raise 'Invalid width' until width > 0
   @topLeft=topLeft
   @height=height
   @width=width
 end
end
```

is_a, kind_of, instance_of

```ruby
class Circle
 attr_accessor :center, :radius
 def initialize(center, radius)
   raise 'Invalid center point' until center.is_a? Point
   raise 'Invalid radius' until radius > 0
   @center=center
   @radius=radius
 end
end
```

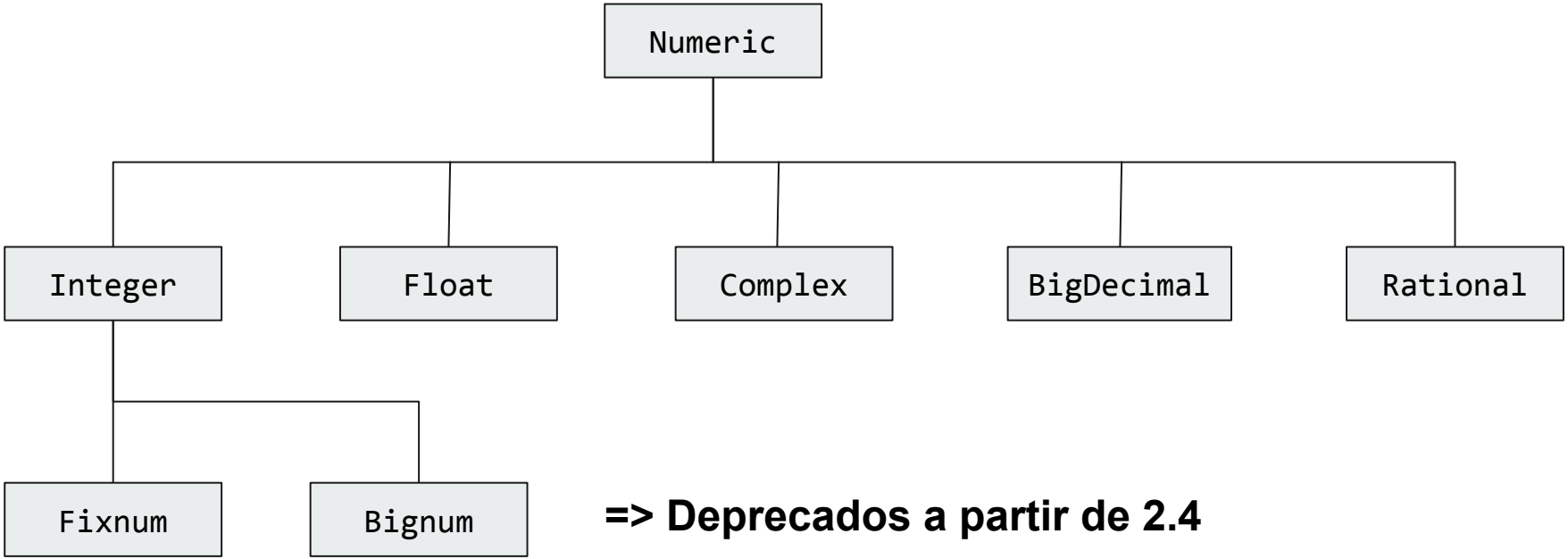
```ruby
class Figure
 PI = 3.14159

 def self.area(shape)
   if shape.is_a? Square
     return shape.side * shape.side
   end
   if shape.is_a? Rectangle
     return shape.height * shape.width
   end
```

```ruby
   if shape.is_a? Circle
     return PI * shape.radius * shape.radius
   end
   nil
 end
end
```

Valores lógicos

- `true` es la única instancia de la clase ***`TrueClass`*** y representa el valor verdadero en expresiones booleanas.
- en forma similar se implementan `false` y ***`FalseClass`***
- Las clases proveen operadores para que `true` y `false` se puedan utilizar en expresiones lógicas
- Ejercicio: implementar los operadores (métodos) &&, ||, !

Números



=> Deprecados a partir de 2.4

Números

```
100.is_a?(Numeric)              #  => true
100.instance_of?(Numeric)       #  => false
100.instance_of?(Integer)       #  => true
100.instance_of?(Fixnum)        #  => true
100.instance_of?(Bignum)        #  => true
100 + 10                        #  => El objeto 100 recibe un mensaje
                                      a través del método +

100 < 150                       #  => true
100 <=> 150                     #  => -1
```

Números

```
a = 1.5
a.is_a? Numeric          # true
a.is_a? BigDecimal       # => uninitialized constant BigDecimal

require 'bigdecimal'

a = 1.5
a.is_a? BigDecimal       # false
a.is_a? Float            # true
```

Strings

```
'12345'.size                # => 5
'12345ñ'.ascii_only?        # => false
'abc'.capitalize            # => Abc
'abc'.chars                 # => ["a", "b", "c"]
'abcde'.include?('cd')      # => true
'abcde'.include?('dc')      # => false
'abcde'.to_s                # => "abcde"
'abcdeabcde'.delete('a')    # => "bcdebcde"
'abcdeabcde'.delete('^a')   # => "aa"
'abcde'.reverse             # => "edcba"
'|' * 5                     # => "|||||"
'abced'.chars.max               # => "e"
```

Strings

Comillas simples vs comillas dobles

```
name = 'World'          # forma abreviada de name = String.new("World")

puts "Hello #{name}"

puts 'Hello #{name}'
```

Hello World

Hello #{name}

Operadores

Estos “operadores” pueden tratarse como métodos, y por lo tanto se pueden sobreescribir (excepto ! = y ! ~)

::	Resolución de constantes
[ ] [ ]=	Referenciar y setear elemento
**	Exponente
! ~ + -	Not, complemento, más y menos unarios
* / %	Multiplicación, división, módulo
+ -	Suma y resta binarios
<< >>	Shift
&	And a nivel de bits
| ^	OR, XOR a nivel de bits
<= < > >=	Comparación
<=> == === != =~ !~	Igualdad y coincidencia de patrones

Operadores

```
class POO02
 AUTHOR    = "ITBA"
 TITLE     = "Introducción a Ruby"
end

puts POO02::AUTHOR     #  => ITBA
p POO02::AUTHOR       #  => "ITBA"
```

Operadores: <=> (“*spaceship*”)

Necesario para poder ordenar una colección de objetos. Se espera que retorne -1, 0 ó 1.
Para Object retorna nil

```
class Foo
 def initialize number
   @number = number
 end

 def to_s
   @number.to_s
 end
end
```

```
f1 = Foo.new 10
f2 = Foo.new 20


f1 <=> f2    # nil
```

Ejemplo basado en https://ruby-doc.org/core-2.4.0/Numeric.html

```ruby
class Tally < Numeric
 def initialize(string)
   @string = string
 end

 def to_s
   @string
 end

 def to_i
   @string.size
 end
```

```ruby
 def <=>(other)
   to_i <=> other.to_i
 end

 def +(other)
   self.class.new('|' * (to_i + other.to_i))
 end

 def -(other)
   self.class.new('|' * (to_i - other.to_i))
 end
```

Ejemplo basado en https://ruby-doc.org/core-2.4.0/Numeric.html

```
  def *(other)
    self.class.new('|' * (to_i * other.to_i))
  end

  def /(other)
    self.class.new('|' * (to_i / other.to_i))
  end
end
```

```
tally = Tally.new('||')
puts tally * 2              #=> "||||"
puts tally > 1              #=> true
puts tally / 2              #=> "|"

Tally.new('||') <=> Tally.new('||||')   # -1
```

==, ===, equal?

- **`equal?`** a.equal?(b) si a y b son el mismo objeto (a y b referencian la misma instancia)
- **`==`** a==b si a y b representan el mismo objeto. Por defecto lo mismo que equal? pero se puede sobreescribir
- **`===`** “case equality”. Para Object es lo mismo que ==, pero se sobreescribe para proveer semántica para sentencia `case`
- **`eql?`** lo vemos más adelante

==, ===, equal?

```ruby
class Person
 @id
 def id
   @id
 end
 def initialize(id)
   @id=id
 end
 def ==(other)
   return self.id == other.id
 end
end
```

```ruby
a = Person.new(100)
b = Person.new(100)

a == b          # => true
a.equal?(b)     # => false
b = a
a == b          # => true
a.equal?(b)     # => true
```

Operadores

&&	AND lógico
||	OR lógico
.. ...	Rango (inclusivo y exclusivo)
? :	condicional
= %= { /= -= += |= &= >>= <<= *= &&= ||= **=	Asignación
defined?	
not	negación lógica
or and	composición lógica

Operadores

```
true && false                # => false
true || false                # => true
1 > 3 || 2 < 5               # => true
(1 > 3 || 2 < 5).class       # => TrueClass
4 && 5                       # => 5
5 && 4                       # => 4
5 || 4                       # => 5

false &&                     # => false

true ||                      # => true

(nil &&  ).nil?              # => true
```

Rangos

```
(1..5).class      # => Range
(1..5).max        # => 5
(1...5).max       # => 4
(1..5).min        # => 1
(2...5).min       # => 2
(1...1).min       # => nil

(1..10) === 5        # => true
(1..10) === 15       # => false
(1..10) === 3.14159  # => true
('a'..'j') === 'c'   # => true
('a'..'j') === 'z'   # => false
```

==, ===, equal?

```
5 == (1..10)              # false
5 === (1..10)             # false
5.equal?(1..10)           # false

(1..10) === 5             # true
(1..10) == 5              # false
(1..10).equal? 5          # false
```

Operadores

```
a=7
defined? a                  # => local-variable
defined? b                  # =>
(defined?  ).class          # => String
defined? a.to_s             # => method
defined? a.foo              # =>
A=7
defined? A                  # => constant
defined? nil                # => nil
defined? true               # => true
defined? false              # => false
defined? TRUE               # => constant    ( TRUE = true )
```

Operadores: ejemplo

```
class Integer
 def factorial
   self <= 1 ? 1 : self * (self - 1).factorial
 end
end
```

Operadores: ejemplo

```
class Integer
 def factorial
   self <= 1 ? 1 : self * (self - 1).factorial
 end
 def factorial_iterative
   f = 1; for i in 1..self; f *= i; end;
   f
 end
 alias :factorial :factorial_iterative
end
```

Operadores and, or

Más que operadores lógicos son de control de flujo

```
age2 = (age = 17 and age + 1)          # => age vale 17, age2 18
age >= 18 and puts 'Mayor de edad'     # => no imprime nada
age >= 18 && puts 'Mayor de edad'      # => syntax error
age >= 18 or raise 'No apto para menores'

f = file.zip and  f.backup  and logger.info('Backup done')

b = true and false  # ( b = true ) and false
b = true && false   # b = (true && false)
```

Salvo excepciones, aconsejamos no usarlos

Sentencia if...else

```
if conditional [then]
   code...
[elsif conditional [then]
   code...]
[else
   code...]
end
```

Verdadero salvo para false y nil

if...else

```
if a == 0
 puts 'a is zero'
elsif a == 1
 puts 'a is one'
else
 puts 'a is some other value'
end
```

if...else

```
a = 1

if a == 0
 puts 'a is zero'
elsif a == 1
 puts 'a is one'
elsif a >= 1
 puts 'a is greater than or equal to one'
else
 puts 'a is some other value'
end
```

if...else

```
if a == 0
 puts 'a is zero'
elsif a >= 1
 puts 'a is greater than or equal to one'
elsif a == 1
 puts 'a is one'          # nunca se ejecuta
else
 puts 'a is some other value'
end
```

Modificadores if, unless

```
a+=1 if a == 0

puts 'positivo' if a > 0            # bien
puts 'non-zero' if not a == 0       # feo
puts 'non-zero' if a != 0           # ok
puts 'non-zero' unless a == 0       # bien
puts 'non-zero' unless a.zero?      # mejor

avg = sum / qty unless qty == 0     # => avg es el promedio o nil
```

Modificadores if, unless

sentencia if de una sola proposición vs modificador if

```
# bad
if some_condition
 do_something
end

# good
do_something if some_condition
```

Expresión case

Opción 1: comparar un objeto contra múltiples patrones. Los patrones son “*matcheados*” usando el método === (que en Object es un alias de ==)

```
case a
 when 1, 2 then
   puts 'a is one or two'
 when 3 then
   puts 'a is three '
 else
   puts 'I don't know what a is'
end
```

Expresión case

El valor devuelto es el último valor evaluado en la expresión

```
puts (
   case a
     when 1, 2 then
       'a is one or two'
     when 3 then
       'a is three '
     else
       'I don't know what a is'
   end
)
```

Expresión case

Opción 1: comparar un objeto contra múltiples patrones

```
case '12345'
 when /^1/, '2'
   puts 'the string starts with one or is two'
 when /^2/
   puts 'the string starts with two'
end
```

Expresión case

Opción 2: similar a if-elseif

```
case
 when a == 1, a == 2
   puts 'a is one or two'
 when a == 3
   puts 'a is three'
 else
   puts 'I don't know what a is'
end
```

Blocks

- Un bloque está formado por “pedazos” de código
- Tiene un nombre asociado
- Se encierra entre llaves
- Se invoca desde un método que tenga el mismo nombre
- Para invocarlo se usa la sentencia *yield*

Blocks: ejemplo

```
def test
 puts 'You are in the method'
 yield
 puts 'You are again back to the method'
 yield
end

# Invocamos al método test y le "pasamos" un bloque
test {puts 'You are in the block'}
```

Blocks: parámetros

```
def test
 yield 5
 puts 'You are in the method test'
 yield 100
end

test {|i| puts "You are in the block #{i}"}
```

Blocks: BEGIN y END

```ruby
BEGIN {
 puts 'Primero muestra esto'
}

END {
 puts 'Cuarto lugar'
}

END {
 puts 'Tercer lugar'
}

puts 'Segundo muestra esto'
```

Se recomienda no usar END.

```ruby
# bad
END { puts 'Goodbye!' }

# good
at_exit { puts 'Goodbye!' }
```

Blocks como parámetros

```
(1..10).each{puts "\n"}

10.times do
 puts "\n"
end


10.times {puts "\n"}

(1..10).each{|i| puts i}
```

Loops

while: ejecuta mientras una condición es verdadera (cualquier valor excepto **`false`** o **`nil`**)

```
a=1
while a <= 10   [do]
 puts a
 a += 1
end
```

Loops

until: ejecuta mientras una condición es **`false`** o **`nil`**

```
a=1
until a == 10
 puts a
 a += 1
end
```

Loops

for: itera sobre los valores de un conjunto de datos

```
for a in 1...10
 puts a
end

s=0
for v in [1,20,5,3]
 s+=v
end
```

el for ha perdido popularidad en Ruby

Imperativo vs OO + funcional

Escribir un programa ineficiente que imprima los pares entre 1 y 100

```
a=1
while a <= 100
 if a % 2 == 0
   puts a
 end
 a = a + 1
end
```

```
for a in 1..100
 puts a if a.even?
end
```

```
(1..100).each { |v| puts v if v.even?}
```

Loops

while y until se pueden usar como modificadores

```
code until conditional
```

Usar esta forma si code es una sola línea

```
begin
 code
end until conditional
```

Loops

while y until se pueden usar como modificadores

```
# bad
while some_condition
  do_something
end

# good
do_something while some_condition
```

Loops: break

break se utiliza para salir del ciclo en forma temprana

```
values.each do |value|
 break if value.even?
 # ...
end

while true do
 puts a
 a += 1
 break if a > 10
end
```

Otros: next, redo

Módulos

- Brindan una forma de agrupar métodos, constantes, clases
- Proveen un ambiente con un nombre
- Permiten el uso de *mixins*: agregar funcionalidad a clases

Módulos: require

Archivo trig.rb

```
module Trig
 PI = 3.141592654
 def Trig.sin(x)
    # ...
 end
 def Trig.cos(x)
    # ...
 end
end
```

```
require 'trig.rb'

y = Trig.sin(Trig::PI/2)
```

Módulos: include

```ruby
module A
 def a1
 end
 def a2
 end
end

module B
 def b1
 end
 def b2
 end
end
```

```ruby
class Sample
 include A
 include B
 def s1
 end
end

samp = Sample.new
samp.a1
samp.a2
samp.b1
samp.b2
samp.s1
```

Módulos: include

```ruby
module A
 def a1
 end
end

module B
 def b1
 end
 def a1
 end
end
```

```ruby
class Sample
 include A
 include B
 def s1
 end
end

samp = Sample.new
samp.a1   # es B.a1
samp.b1
samp.s1
```