



Empezando con Ruby

Juan Sebastian Velez Posada



/jvelezpo

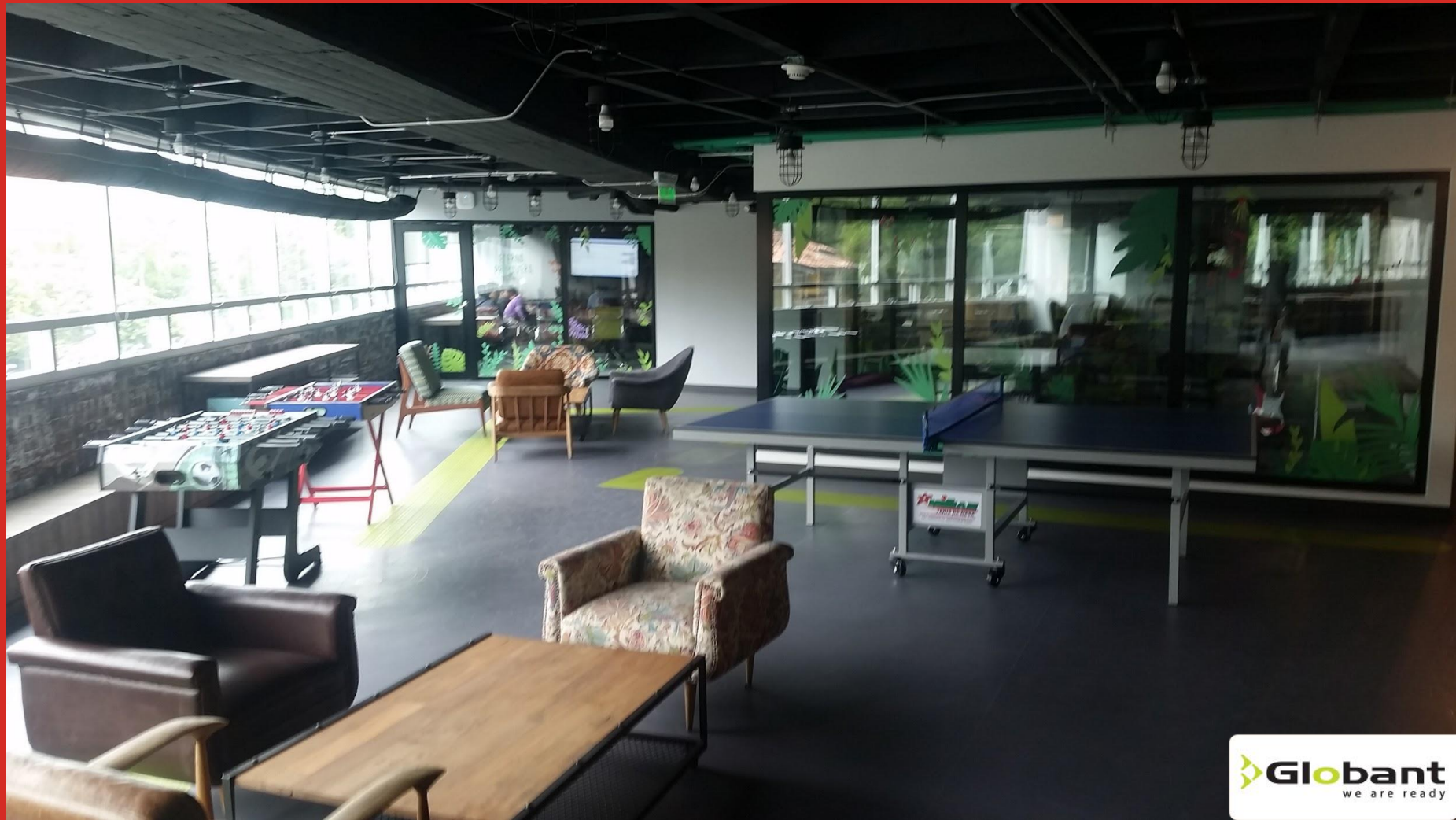


/rderoldan1









Temario

Esta presentación mostrará los siguientes temas:

1. Tipos de datos
2. Expresiones y Operadores
3. Estructuras de Control
4. Métodos

****TIP**

Se recomienda leer y aplicar las buenas practicas en la forma de programación especificadas en esta URL <https://github.com/bbatsov/ruby-style-guide>

Primeros pasos con 'irb'



```
1. irb (ruby)
~% irb
>> puts 'hello world'
hello world
=> nil
>> _
```

Consola interactiva de ruby, como programador permitirá validar conceptos, dado a que se obtiene una respuesta de la máquina de manera inmediata.

Ruby is..

- Interpreted
- Object-oriented
 - Everything is an object
 - Every operation is a method call on some object
- Dynamically typed: objects have types, but variables don't
- Dynamic
 - add, modify code at runtime (metaprogramming)
 - ask objects about themselves (reflection)
 - in a sense all programming is metaprogramming

Naming conventions

- ClassNames use UpperCamelCase

```
class FriendFinder ... end
```

- methods & variables use snake_case

```
def learn_conventions ... end
def faculty_member? ... end
def charge_credit_card! ... end
```

- CONSTANTS (scoped) & **\$GLOBALS (not scoped)**
TEST_MODE = true **\$TEST_MODE = true**

- symbols: like immutable string whose value is itself

```
favorite_framework = :rails
:rails.to_s() == "rails"
"rails".to_sym() == :rails
:rails == "rails" # => false
```

Tipos de dato - Comentarios

```
#  
# Comentarios de una linea  
#  
  
=begin  
  Comentario de múltiples lineas  
=end
```

Se recomienda usar siempre comentarios de una sola línea.

Tipos de dato - Números

```
# Enteros= FixNum(31 Bits), BigNum(+31 Bits)
0
1234
123456789876543

# Decimales
0.0
1.82
6.02e23
```

Todos los números descienden del Objeto 'Numeric' y son tratados 'transparentemente' por el lenguaje.

Tipos de dato - Texto

```
#Strings  
'Hello World'  
"Hello World at #{Time.now}"  
string = <<END  
Documento de texto,  
Varias Lineas  
END
```

Según buenas prácticas, se debe usar strings de “comilla simple” siempre que no se necesite interpolación.

Tipos de dato - Arreglos

```
# Arreglos
[1,2,3] # Arreglo de números
[[1,2],[3,4],[5,6]] # Arreglo de arreglos
[Date.today, 1, 'Hello World'] # Arreglo de varios tipos de datos
%w{primera segunda tercero} # Arreglo de strings
```

Los arreglos contienen múltiples tipos de datos, pueden ser cambiados en cualquier momento. El arreglo puede cambiar de tamaño en cualquier momento

Tipos de dato - Hashes

```
# Hashes  
{ "key1" => 1, "key2" => 2}  
{ :key1 => 1, :key2 => 2}  
{ key1: 1, key2: 2} # Introducido en Ruby 2.1
```

Estructuras asociativas, relaciona una llave con un valor, se recomienda en la llave usar símbolos, ya que ocupan menos espacio en memoria. El valor puede ser cualquier tipo de dato.

Variables, Arrays, Hashes

- There are no declarations!
 - local variables must be assigned before use
 - instance & class variables == `nil` until assigned

• OK: `x = 3; x = 'foo'`

- **Wrong: Integer x=3**

• Array: `x = [1, 'two', :three]`
 `x[1] == 'two' ; x.length==3`

• Hash: `w = {'a'=>1, :b=>[2, 3]}`
 `w[:b][0] == 2`
 `w.keys == ['a', :b]`

Methods

- Everything (except fixnums) is pass-by-reference

```
def foo(x,y)
  return [x,y+1]
end
```

```
def foo(x,y=0)      # y is optional, 0 if omitted
  [x,y+1]           # last exp returned as result
end
```

```
def foo(x,y=0) ; [x,y+1] ; end
```

- Call with: `a,b = foo(x,y)`
or `a,b = foo(x)` when optional arg used

Basic Constructs

- Statements end with ' ; ' or newline, but can span line if parsing is unambiguous

✓ `raise("Boom!") unless
 ship_stable`

✗ `raise("Boom!")
 unless (ship_stable)`

- Basic Comparisons & Booleans:

`== != < > =~ !~ true false nil`

- The usual control flow

```
if cond (or unless cond)  
  statements  
[ elsif cond  
  statements ]  
[else  
  statements]  
end
```

```
while cond (or until cond)  
  statements  
end  
1.upto(10) do |i| ... end  
10.times do...end  
collection.each do |elt|...end
```

Strings & Regular Expressions

(try rubular.com for your regex needs!)

```
"string", %Q{string}, 'string', %q{string}
```

```
a=41 ; "The answer is #{a+1}"
```

- match a string against a regexp:

```
"fox@berkeley.EDU" =~ /(.*)(.*)\.edu$/i
```

```
/(.*)@(.*)\.edu$/i =~ "fox@berkeley.EDU"
```

– If no match, value is false

– If match, value is non-false, and \$1...\$n capture parenthesized groups (\$1 == 'fox', \$2 == 'berkeley')

```
/(.*)$/i    or    %r{(.*)$}i    or    Regexp.new('(.*)$', Regexp::IGNORECASE)
```


Question

```
rx = { :fox=>/^arm/, 'fox' => [ %r{AN(DO)$}, /an(do)/i ] }
```

Which expression will evaluate to non-nil?

"armando" =~ rx{:fox}

rx[:fox][1] =~ "ARMANDO"

rx['fox'][1] =~ "ARMANDO"

"armando" =~ rx['fox', 1]

Modern OO Languages

- Objects
- Attributes (properties), getters & setters
- Methods
- Operator overloading
- Interfaces
- “Boxing” and “unboxing” primitive types

...is there a smaller set of mechanisms that captures most or all of these?

Todo es un objeto

```
class Persona
  attr_accessor :nombres, :apellidos, :edad

  def initialize(nombres, apellidos, edad)
    @nombres, @apellidos, @edad = nombres, apellidos, edad
  end

  def nombre_completo
    "#{@nombres} #{@apellidos}"
  end
end

persona = Persona.new('Ruben', 'Espinosa', 23)
puts "Hola #{@persona.nombre_completo}, usted tiene #{@persona.edad} años"

# => Hola Ruben Espinosa, usted tiene 23 años
```

Permite abstraer entidades de la realidad, como “Persona”, “Estudiante”, “Clase” y realizar las relaciones entre ellos de manera más flexible.

Everything is an object; (almost) everything is a method call

- Even lowly integers and nil are true objects:

```
57.methods
```

```
57.heinz_varieties
```

```
nil.respond_to?(:to_s)
```

- Rewrite each of these as calls to send:

```
- Example: my_str.length      => my_str.send(:length)
1 + 2                        1.send(:+, 2)
my_array[4]                  my_array.send(:[], 4)
my_array[3] = "foo"          my_array.send(:[]=, 3, "foo")
if (x == 3) ....             if (x.send(:==, 3)) ...
my_func(z)                   self.send(:my_func, z)
```

- in particular, things like “implicit conversion” on comparison is not in the type system, but in the instance methods

REMEMBER!

- a.b means: call method b on object a
 - a is the receiver to which you send the method call, assuming a will respond to that method

✗ does not mean: b is an instance variable of a

✗ does not mean: a is some kind of data structure that has b as a member

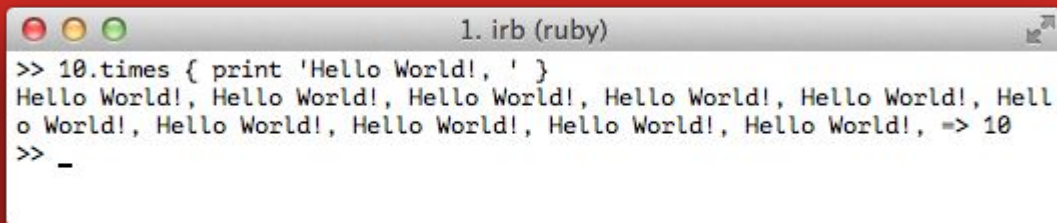
Understanding this distinction will save you from much grief and confusion

Example: every operation is a method call

```
y = [1,2]
y = y + ["foo", :bar] # => [1,2,"foo",:bar]
y << 5                # => [1,2,"foo",:bar,5]
y << [6,7]            # => [1,2,"foo",:bar,5,[6,7]]
```

- “<<” destructively modifies its receiver, “+” does not
 - destructive methods often have names ending in “!”
- Remember! These are nearly all instance methods of Array
 - not language operators!
- So `5+3`, `"a"+"b"`, and `[a,b]+[b,c]` are all different methods named `'+'`
 - `Numeric#+`, `String#+`, and `Array#+`, to be specific

Ruby, ingles para computadores?



```
1. irb (ruby)
>> 10.times { print 'Hello World!, ' }
Hello World!, Hello World!, Hello World!, Hello World!, Hello World!, Hello World!, Hello World!, Hello World!, Hello World!, Hello World!, => 10
>> _
```

Es importante, en términos de mantenibilidad del código que sea entendible para los programadores, esté, es uno de las ventajas de Ruby.

Hashes & Poetry Mode

```
h = {"stupid" => 1, :example=> "foo" }  
h.has_key?("stupid")           # => true  
h["not a key"]                 # => nil  
h[:example]                    # => "foo"  
h.delete(:example)             # => "foo"
```

- Ruby idiom: “poetry mode”

- using hashes to pass “keyword-like” arguments
- omit hash braces when last argument to function is hash
- omitting parens around function arguments

```
link_to("Edit",{:controller=>'students', :action=>'edit'})  
link_to "Edit", :controller=>'students', :action=>'edit'  
link_to 'Edit', controller: 'students', action: 'edit'
```

- When in doubt, parenthesize defensively

Poetry mode in action

```
a.should(be.send(:>=, 7))
```

```
a.should(be() >= 7)
```

```
a.should be >= 7
```

```
(redirect_to(login_page)) and return() unless logged_in?
```

```
redirect_to login_page and return unless logged_in?
```

Question

```
def foo(arg, hash1, hash2)
  ...
end
```

Which is **NOT** a legal call to foo():

```
foo a, { :x=>1, :y=>2 }, :z=>3
```

```
foo(a, :x=>1, :y=>2, :z=>3)
```

```
foo(a, { :x=>1, :y=>2 }, { :z=>3 })
```

```
foo a, { :x=>1, :y=>2 }, { :z=>3 }
```

Operadores - Variables

Variables de clase

`@@variable`

Variables de instancia

`@variable`

Variables globales

`$variable`

Variables locales

`variable`

Operadores - Constantes

```
#Constantes  
TASA_CAMBIO = 2000
```

Son espacios reservados para valores que nunca deben cambiar durante la ejecución del programa.

Operadores - Asignación

```
# Asignaciones  
x = 1 # Asigna a x el valor 1  
x += 1 # Incrementa en uno el valor de x y asigna el valor de x  
x,y = 1, 2 # Asigna a x el valor y en y el valor 2
```

Una asignación específica que uno ó más valores de la izquierda van a tener almacenado un valor de la derecha, usado en variables, constantes, atributos, arrays y hashes.

Operadores - Asignaciones abreviadas

Tabla de asignaciones abreviadas y sus equivalentes

<code>x += y</code>	<code>x = x + y</code>
<code>x -= y</code>	<code>x = x - y</code>
<code>x *= y</code>	<code>x = x * y</code>
<code>x /= y</code>	<code>x = x / y</code>
<code>x %= y</code>	<code>x = x % y</code>
<code>x **= y</code>	<code>x = x ** y</code>
<code>x &&= y</code>	<code>x = x && y</code>
<code>x = y</code>	<code>x = x y</code>
<code>x &= y</code>	<code>x = x & y</code>
<code>x = y</code>	<code>x = x y</code>
<code>x ^= y</code>	<code>x = x ^ y</code>
<code>x <<= y</code>	<code>x = x << y</code>
<code>x >>= y</code>	<code>x = x >> y</code>

Operadores - Aritmética

```
# Operadores  
1 + 1 # => 2  
[1] + [2] # => [1,2]  
'Hello' + ' Word' # => 'Hello World'  
  
1 - 2 # => 1  
[1,2] - [1] # => [2]  
  
1 * 3 # => 3  
[1] * 2 # => [1,1]  
'Hello World' * 2 # => 'Hello World Hello World'  
  
1 == 1 # => true  
1 != 1 # => false  
'hello' == 'hello' # => true  
'hello' != 'hello' # => false  
[1] == [1] # => true  
[1] != [1] # => false
```

Operadores - Aritmética

! ~ +	Boolean NOT, bitwise complement, unary plus ^a
**	Exponentiation
-	Unary minus (define with -@)
* / %	Multiplication, division, modulo (remainder)
+ -	Addition (or concatenation), subtraction
<< >>	Bitwise shift-left (or append), bitwise shift-right
&	Bitwise AND
^	Bitwise OR, bitwise XOR
< <= >= >	Ordering
== === != =~ !~ <=>	Equality, pattern matching, comparison ^b
&&	Boolean AND
	Boolean OR
.. ...	Range creation and Boolean flip-flops
?:	Conditional
rescue	Exception-handling modifier
=	Assignment
**= *= /= %= += -=	
<<= >>=	
&&= &= = = ^=	

Estructuras de control - Condicionales

```
# Condicionales
```

```
if 1 == 1
  expresion
end
```

```
if 1 == 1
  expresion1
else
  expresion2
end
```

```
if 1 == 1
  expresion1
elsif 2 == 2
  expresion2
end
```

```
# Condicionales
```

```
unless 1 == 1
  expresion
end
```

```
unless 1 == 1
  expresion1
else
  expresion2
end
```

```
# Condicionales
```

```
case variable
  when 1
    "uno"
  when 2 then "dos"
  else
    "Desconocido"
end
```

Estructuras de control - Ciclos

```
#ciclos
x = 10
while x >= 10 do
  puts "Hello World"
end

x = 0
until x > 10 do
  puts "Hello World"
end
```

```
#ciclos
array = [1,2,3,4]
for item in array
  puts item
end

array = [1,2,3,4]
array.each do |item|
  puts item
end
```

Métodos

Por convención, los métodos deben ser llamados en minúscula, separados por guión bajo (si aplica)

```
#metodos
def numero_positivo(numero)
  if numero >= 0
    puts 'Positivo'
  else
    puts 'Negativo'
  end
end
```

Los métodos son utilidades que permiten encapsular porciones de código con fines específicos. El valor de retorno por defecto es la última sentencia, aunque se puede especificar con la sentencia 'return'.

Métodos - Parámetros

```
#metodos
def numero_positivo(numero = 0)
  if numero >= 0
    puts 'Positivo'
  else
    puts 'Negativo'
  end
end

numero_positivo(10) # => "Positivo"
numero_positivo(-10) # => "Negativo"
numero_positivo # => "Positivo"
```

Al definir un método, se puede especificar un valor por defecto, así, en caso de no enviar el parámetro, éste tomará el valor definido

Homework

<https://github.com/jvelezpo/hw-ruby-intro>