Ruby

Un lenguaje simple, natural y productivo

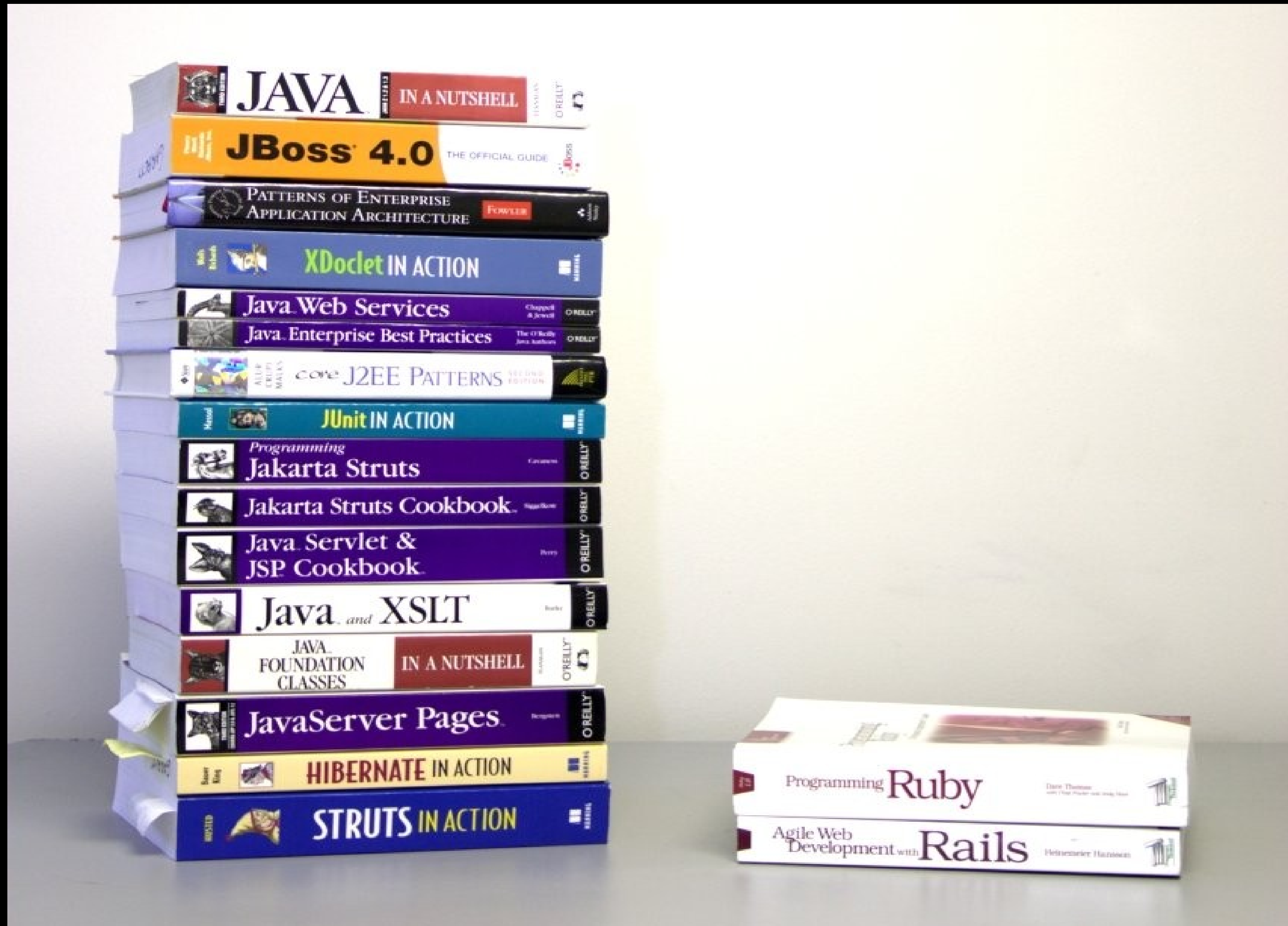
por: *Gastón Ramos*
www.gastonramos.com.ar

Qué es Ruby?

Un lenguaje de programación **libre** dinámico y enfocado en la simplicidad y productividad. Su elegante sintaxis se siente natural al leerla y fácil al escribirla.

```
["a,", "b,", "c"].empty?
3.between?(4, 5)
5.times{ puts "hola mundo" }
```

Ruby es simple y fácil de aprender



Creado por "Matz" en el año 1995

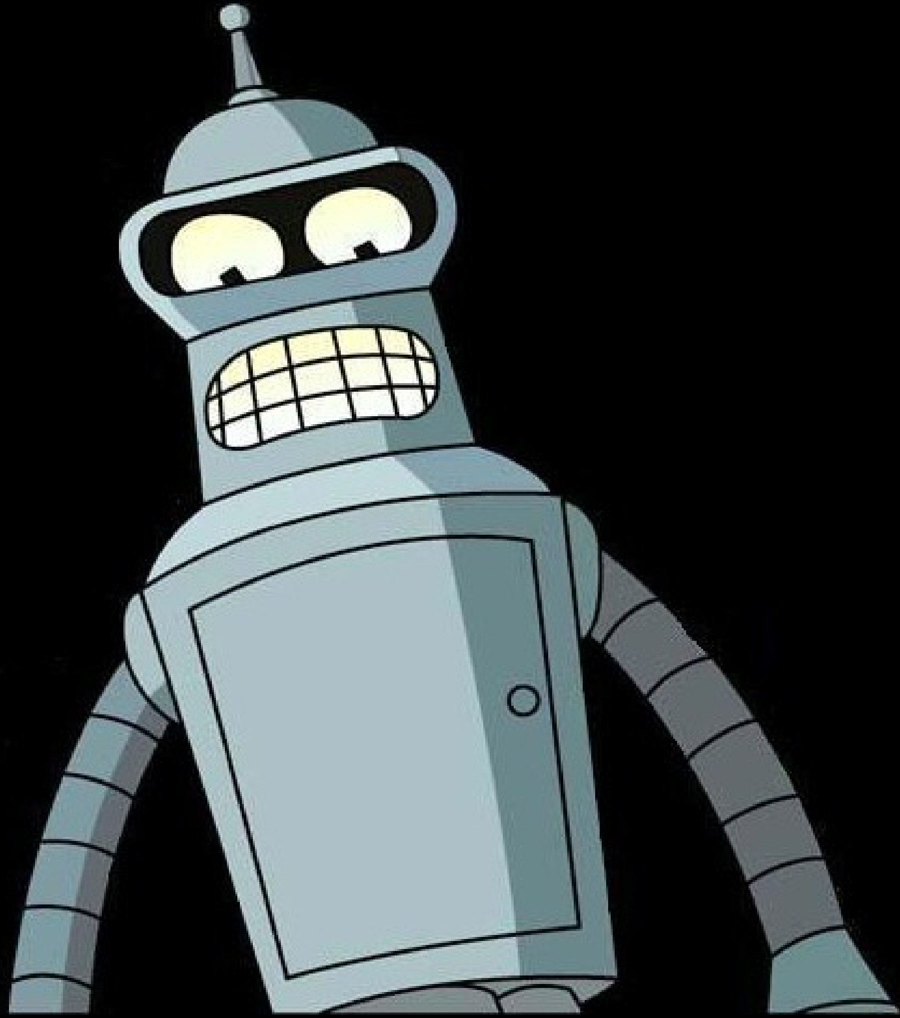
“...tratando de hacer que Ruby sea natural, no simple”

Matz mezcló partes de sus lenguajes favoritos (Perl, Smalltalk, Eiffel, Ada, y Lisp) para formar un nuevo lenguaje que incorporara tanto la programación funcional como la programación imperativa

Se hizo popular con Ruby on Rails

En el año 2005 se hace pública la primer versión de Ror un framework web desarrollado con Ruby. Hasta el momento no había documentación de Ruby en inglés.

Diseñado y pensando para las personas no para las máquinas...



“Often people, especially computer engineers, focus on the machines. They think, "By doing this, the machine will run faster. By doing this, the machine will run more effectively. By doing this, the machine will something something something." They are focusing on machines. But in fact we need to focus on humans, on how humans care about doing programming or operating the application of the machines. We are the masters. They are the slaves."

Ruby en el mundo real:

Simulaciones:

- NASA Langley Research Center
- Motorola

Modelado 3d:

- Google SketchUp

Robótica:

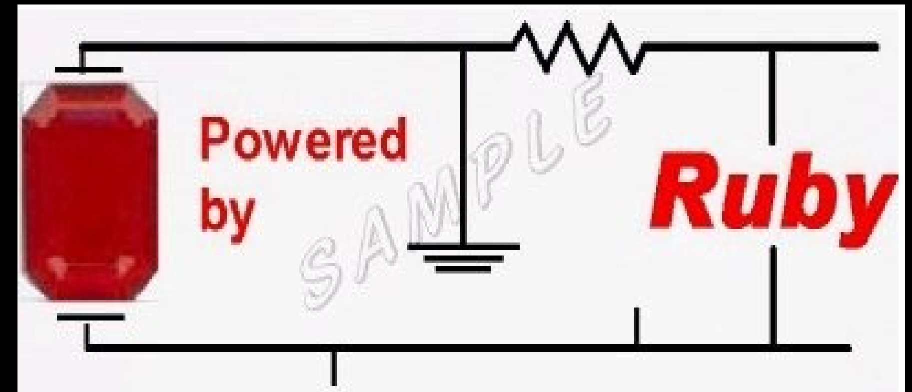
- MORPHA project

Networking:

- Open Domain Server

Aplicaciones web:

- Basecamp, Twitter, Assembla...



Características de Ruby:

- Ruby es un lenguaje interpretado, para ejectutar programas en ruby no hace falta compilarlos.
- Las variables tienen tipado dinámico. No hay que preocuparse por el tipo de datos de las variables.
- Las variables no se declaran.
- Tiene una sintaxis elegante.
- El manejo de la memoria es automático. Los objetos que ya no son referenciados desde ningún lado son automáticamente recolectados por el garbage collector.
- Todo es un objeto.

Caraterísticas de Ruby:

- Clases, Herencia y métodos. Intencionalemente no posee herencia múltiple.

- Metodos singleton. (Se puede definir un método para una instancia de un objeto en particular)
- Mix-in con módulos, para compartir implementaciones (Hack para lograr herencia mútliple).
- Iteradores y Closures.
- Procesamiento de texto y expresiones regulares.
- Manejo de excepciones.
- Acceso directo al sistema operativo.

Todo es un objeto

En ruby todas las cosas son un objeto.

Los resultados de las operaciones con estos objetos siempre retornan otro objeto.

Incluso las definiciones de clases son también instancias de la clase Class.

```
>> nil.class
=> NilClass
```

```
>>5.object_id
=> 11
>>5.odd?
 => true
```

Strings

Los strings son secuencias de bytes (8-bit). Normalmente contienen caracteres imprimibles. Son objetos de la clase String.

```
a = "Esto es un string"

b = %Q{Esto también es un string}

c = <<FIN

   Este es un string multiline
   FIN

   variable = "otra cosa"
d = "Otro string #{variable} "
=> "Otro string otra cosa"
```

Strings

```
"pasar a mayúsculas".upcase! #=> "PASAR A MAYÚSCULAS"

"capitalizar un string".capitalize! #=> "Capitalizar un string"

"invertir un string".reverse!  #=> "gnirts nu ritrevni"

"hello".tr('aeiou', '*') #=> "h*ll*"

"hello".gsub(/[aeiou]/, '*') #=> "h*ll*"

"InterCambiar Mayusc. y Minus.".swapcase! #=> "iNTERcAMBIAR mAYUSC. Y mINUS."
```

Colecciones. Qué es un Array?

Colecciones. Qué es un Array?

Es una lista de elementos indexada por números enteros que comienzan desde cero (cómo en c).

En ruby los arrays son dinámicos, crecen a medida que lo necesitan sin la intervención del programador.

En ruby los arrays son Heterogéneos de manera que podemos guardar cualquier tipo de datos. (Guardan referencias a los objetos más que los objetos en sí mismos).

La clase Array viene con un conjunto de funciones muy útiles, para buscar, eliminar, concatenar y manipular los arrays.

Como crear un array

```
>> a = Array.new
   => []

>> a = ["a", "z", 2, 4]
   => ["a", "z", 2, 4]

>> Array.new(2)
   => [nil, nil]

>> Array.new(5, "a")
   => ["a", "a", "a", "a", "a"]

>> squares = Array.new(5) {|i| i * i }
   => [0, 1, 4, 9, 16]
```

Métodos útiles de Array

```
p a[2] # => 3.14

  p a.reverse # => [[4, 5], 2, 1, 3.14, 'hi', 1]
  p a.flatten.uniq # =>  [1, 'hi', 3.14, 2, 4, 5]

  a.empty? # => false

  a.include? "hi"  # => true

[1, 2, 4, 5].map{|m| m * 2} # => [2, 4, 8, 10]

[1, 7, 3, 2, 4].sort # => [1, 2, 3, 4, 7]

  [1, 2, 4, 5].each{|e| puts e} # => 1 2 4 5
```

Más métodos útiles con arrays

```
[1, 2, 3, 4, 5].find_all{|v| v % 2 == 0} # => [2, 4]

[1, 2, 3, 4, 5].delete_if{|v| v % 2 == 0} # => [1, 3, 5]

(1..5).partition{|v| v % 2 == 0}  # => [[2, 4], [1, 3, 5]]


["que tal", "como", "que pasa", "si"].grep(/que/)

=> ["que tal", "que pasa"]
```

El método **Inject**

```
[1,3,5].inject(10) {|sum, element| sum + element} # => 19
```

F O L D

Operaciones con Arrays

```
# Intersección
>> [ 1, 1, 3, 5 ] & [ 1, 2, 3 ]   #=> [ 1, 3 ]

# Repetición
>>  [ 1, 2, 3 ] * 3    #=> [ 1, 2, 3, 1, 2, 3, 1, 2, 3 ]

# Concatenación
>> [ 1, 2, 3 ] + [ 4, 5 ]    #=> [ 1, 2, 3, 4, 5 ]

# Diferencia
>>  [ 1, 1, 2, 2, 3, 3, 4, 5 ] - [ 1, 2, 4 ]  #=>  [ 3, 3, 5 ]

# Agregar un elemento
[ 1, 2 ] << "c" << "d" << [ 3, 4 ] #=> [ 1, 2, "c", "d", [ 3, 4 ] ]
```

Colecciones (Hashes)

Son también conocidos como "arrays asociativos" o "diccionarios". Podemos pensarlos como una entidad que establece una asociación entre un índice x y un dato y

Hay dos diferencias fundamentales entre los Hashes y los arrays:

1. El índice de los arrays es siempre un entero. En lo hashes puede ser cualquier objeto.

– Los arrays tiene un orden, los hashes no.

Como crear un Hash

```
>> Hash["a", 100, "b", 200]
   => {"a"=>100, "b"=>200}

>> Hash["a" => 100, "b" => 200]
   => {"a"=>100, "b"=>200}

h = Hash.new("Go Fish")
h["a"] = 100
   h["a"] #=> 100
   h["c"] #=> "Go Fish"
>> h = Hash.new { |hash, key| hash[key] = "Go Fish: #{key}" }
   => {}
```

Iteradores

Un iterador es un método que puede invocar un bloque de código.

- Un bloque de código es una forma de agrupar sentencias.
- El bloque debe aparecer en la misma línea que la llamada al método.
- El bloque no se ejecuta cuando se encuentra.
- Ruby recuerda el contexto en el cual el bloque aparece.

Iteradores

```
def threeTimes
    yield
    yield
    yield
  end

  threeTimes { puts "Hello" }
```

Clases

```
class Persona
  attr_reader :nombre, :edad

  def initialize(nombre, edad)
    @nombre, @edad = nombre, edad
  end
end
```

```
# En ruby el último valor evaluado es el que se retorna.
```

```
p = Persona.new("Gastón", 31)
p.nombre  # =>  "Gastón"
```

El Constructor (de un objeto)

En Ruby no hay un "constructor" real como en C++ o en Java. El concepto está por que los objetos se tienen que instanciar e inicializarce.

En Ruby, una clase tiene un método new, que se usa para instanciar nuevos objetos. Este método new llama al método definido por el usuario initialize, que inicializa los atributos del objeto y retorna una referencia al nuevo objeto.

Creando atributos de instancia

```
class Persona

  def nombre
    @nombre
  end

  def nombre=(x)
     @nombre = x
  end

  def edad
    @edad
  end

end
```

En Ruby los atributos de instancia llevan el prefijo "@".

En algunos lenguajes OO, típicamente tenemos que crear 'setters'y 'getters'

Atajos para crear accesores.

Ruby posee un "atajo" para crearlos muy fácilmente mediante un truco de metaprograming, pero se pueden crear a mano también.

```
class Persona
  attr :nombre, true   # Crea @name, name=
  attr :edad           # Crea @edad, edad
end

class Persona
  attr_reader :a1, :a2        # Crea @a1, a1, @a2, a2
  attr_writer :b1, :b2        # Crea @b1, b1=, @b2, b2=
  attr_accessor :c1, :c2    # Crea @c1, c1, c1=, @c2, c2, c2=
end
```

Atributos y métodos de clase

Un método o un atributo no siempre está relacionado con una instancia de una clase a veces está re acionado con la clase en sí misma.

```
class ReproductorDeMusica
    @@duracion_cancion_actual = 70

    def ReproductorDeMusica.detectar_hardware
        # .....
    end

   def play
       # ....
   end
end
```

Heredando de una Superclase

```
class Persona
   attr_accessor :nombre, :edad

   def initialize(nombre, edad)
      @nombre, @edad = nombre, edad
   end
end

class Estudiante < Persona
  attr_accessor :idnum, :horas

  def initialize(nombre, edad, idnum, horas)
     super(nombre, edad)
     @idnum = idnum
     @horas  = horas
  end
end
```

Herencia y herencia múltiple

EL propósito de la herencia es aumentar la funcionalidad. Desde un modelo más general a uno más particular. Muchos lenguajes como C++ implementan Herencia Múltiple (MI) Ruby como java no permite la herencia múltiple directa, pero tiene una forma muy elegante de resolverlo, y es con los módulos.

Módulos y Mixins

Hay dos razones básicas para usar módulos en Ruby:

1. Manejo de espacio de nombres, tendremos pocas colisiones de nombres is guardamos las constantes y métodos dentro un módulo.
2. Y la más interesante, podemos usar un módulo como un mixin.

```
modules MiModulo
   def metodo_a
       puts "Este es el método a"
   end
end

class MiClase
  include MiModulo
end
x = MiClase.new
x.metodo_a #=> "Este es el método a"
```

Módulos y Mixins, de clase de instancia

```
module Jornada

  module InstanceMethods
    def saludar
      puts "hola #{@evento}"
    end
  end

  module ClassMethods
    def saludar_a_todos
      puts "hola a todos"
    end
  end

end
```

```
class Junin
  attr_accessor :evento
  include Jornada::InstanceMethods
  extend  Jornada::ClassMethods
end

>> Junin.saludar_a_todos
=> "hola a todos"

>> c =  Junin.new
>> c.evento = "VJSL"
>> c.saludar
=> "hola VJSL"
```

Módulos con métodos de clase y de instancia

```
module Jornada
    def self.included(klass)
      klass.class_eval do
        klass.extend(ClassMethods)
        include Jornada::InstanceMethods
      end
    end

  module InstanceMethods
      def saludar
          puts "hola #{@nombre}"
      end
  end

  module ClassMethods
      def saludar
          puts "hola a todos"
      end
  end
end
```

```
class Junin
   attr_accesor :evento
   include Jornada
end

>> Junin.saludar_a_todos
=> "hola a todos"

>> c =  Junin.new
>> c.evento = "VJSL"
>> c.saludar
=> "hola VJSL"
```

Excepciones

Las excepciones nos permiten empaquetar información acerca de un error dentro de un objeto. El objeto de excepción se propaga hasta que hasta que el sistema encuentra el código que maneja la excepción.

El "paquete" que contiene información acerca de una excepción es un objeto de la clase Exception o algunos de sus hijos.

```
raise Exception, "Fatal error!"
```

Excepciones

```
begin
   eval(string)
rescue SyntaxError, NameError => boom
   puts "esto no compila: " + boom
rescue TypeError
   puts "nil !!!"
ensure
  puts "ejecutando..."
end
```

Clases abiertas

En Ruby todas las clases están "abiertas" y podemos modificarlas a nuestro gusto.

```
class String
  def sin_vocales
    self.tr('aeiou', "*")
  end
end

"hola que onda".sin_vocales
# => "h*l* q** *nd*"
```

FIN

Bibliografía:

- http://wikipedia.org/
- Libro: "Pragmatic Ruby" (Dave Thomas)
- Libro: "The ruby way" (Hal Fulton)
- http://www.ruby-lang.org/es

http://gastonramos.com.ar (Mi sitio)
http://rubyargentina.soveran.com/ (Ruby Argentina)
http://rubylit.com.ar (Grupo de Usuarios de Ruby del Litoral)

ramos.gaston AT gmail DOT com