Ruby e Introducción a las aplicaciones Web

Lo básico para comenzar en la plataforma Ruby

- Ruby vistazo rápido y detalle
- ¿Qué es una aplicación web?
- Protocolo HTTP
- Lenguaje HTML
- Algo de CSS

Vistazo rápido

```
ruby file_name.rb => Para correr programas
irb => Para acceder a consola
```

Asignación variables

```ruby
# this is a comment
a = 1
b = 2
c, d = 10, 20
```

Clases de datos

```ruby
a, b, c = 1, 2.3, 'some string'
a.class
# => Integer
b.class
# => Float
c.class
# = String
```

Casting

```ruby
a.to_f
# => 1.0

a.to_s
# => '1'
```

Inicialización de arreglos

```
# Arrays
days_of_week =  Array.new
# => []
days_of_week =  Array.new(7)
# => [nil, nil, nil, nil, nil, nil, nil]
days_of_week =  Array.new(7, 'some day')
# => ['some day', 'some day', 'some day', 'some day', 'some day', 'some day', 'some day']

days_of_week = []
days_of_week = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
```

Operaciones sobre arreglos

```
days_of_week.push('Saturday')
# => ['Monday', 'Tuesday', 'Wednesday',
#  'Thursday', 'Friday', 'Saturday']
days_of_week.push 'Sunday'
#  => ['Monday', 'Tuesday', 'Wednesday',
#  'Thursday', 'Friday', 'Saturday', 'Sunday']
```

```
first_half = ['Mon', 'Tue', 'Wed']
second_half = ['Thu', 'Fri', 'Sat', 'Sun']
days = first_half + second_half
# => ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']

days = first_half.concat(second_half)
# => ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']

days1 = ['Mon', 'Tue', 'Wed']
days1 << 'Thu' << 'Fri' << 'Sat' << 'Sun'
# => ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
```

Control de flujo

sentencias if / unless

```
if 10 < 20 then
  print '10 is less than 20'
end

if 10 < 20
  print '10 is less than 20'
end

print '10 is less than 20' if 10 < 20

unless 20 > 10
  print "this line won't be printed"
else
  print '20 is more than 10'
end
```

else y elsif

```
customer_name = 'Alice'

if customer_name == 'Fred'
  print 'Hello Fred!'
else
  print "You're not Fred! Where's Fred?"
end
```

```
if customer_name == 'Fred'
  print 'Hello Fred!'
elsif customer_name == 'Alice'
  print 'Hello Alice!'
elsif customer_name == 'Robert'
  print 'Hello Bob!'
end
```

operador ternario

```
customer_name == 'Fred' ? 'Hello Fred' : 'Who are you?'
```

Loops

```ruby
i = 0
while i < 5
    puts i
    i += 1
end

i = 0
until i == 5
    puts i
    i += 1
end

for i in 0..4
  puts i
end

5.times { |i| puts i }
```

Definición de métodos

```ruby
def hello_world()
  puts "I'm a ruby programmer now"
end
```

Símbolos (:Symbol)

Parecido a un string pero inmutable
Se define con semicolon inicial :
Mucho más eficiente que un string

caso string

```
city = 'santiago'
city.object_id
# => 300
city2 = 'santiago'
city2.object_id
# => 320
```

caso symbol

```
city = :santiago
city.object_id
# => 2168668
city2 = :santiago
city2.object_id
# => 2168668
```

VS

Diccionarios (Hashes)

Un hash es un objeto que almacena pares de key y value
Normalmente la llave es un symbol o string pero puede ser otra cosa

```ruby
months = Hash.new
months = {}
# => {}

hash_with_strings = { 'key1' => 'value1', 'key2' => 'value2' }
# => {"key"=>"value1", "key2"=>"value2"}

hash_with_symbols = { :key1 => 'value1', :key2 => 'value2' }
hash_with_symbols = { key1: 'value1', key2: 'value2' }
# => {:key1=>"value1", :key2=>"value2"}
```

```ruby
hash_with_strings['key1']
=> "value1"
hash_with_strings[:key1]
=> nil

hash_with_symbols[:key1]
=> "value1"
```

```ruby
months = { [1,"jan"] => "January" }
months[[1,"jan"]]
# => => "January"
```

Built-in methods para hashes

```ruby
my_pet = { name: 'Milah', age: 3, breed: 'Golden', vaccinated: true }
# ⇒ {:name⇒"Milah", :age⇒3, :breed⇒"Golden", :vaccinated⇒true}

my_pet.empty?
# ⇒ false
my_pet.clear
# ⇒ {}
my_pet.empty?
# ⇒ true
```

```ruby
my_pet = { name: 'Milah', age: 3, breed: 'Golden', vaccinated: true }

my_pet.delete(:vaccinated)
# ⇒ {:name⇒"Milah", :age⇒3, :breed⇒"Golden"}

my_pet[:favorite_toy] = 'a sheep'
# ⇒ {:name⇒"Milah", :age⇒3, :breed⇒"Golden", :favorite_toy⇒"a sheep"}
```

Built-in methods para hashes

```ruby
my_pet.has_key?(:breed)
my_pet.include?(:breed)
my_pet.key?(:breed)
# ⇒ true

my_pet.has_value?('Milah')
my_pet.value?('Milah')
# ⇒ true

my_pet.has_value?('Chiguagua')
# ⇒ false
```

```ruby
my_pet.inspect
# ⇒ "{:name⇒\"Milah\", :age⇒3, :breed⇒\"Golden\", :favorite_toy⇒\"a sheep\"}"

my_pet.length
my_pet.size
# ⇒ 4
```

```ruby
my_pet.keys
# ⇒ [:name, :age, :breed, :favorite_toy]
my_pet.values
# ⇒ ["Milah", 3, "Golden", "a sheep"]

for some_key in my_pet.keys do
  puts some_key
end
# ⇒  name
#    age
#    breed
#    favorite_toy
```

Built-in methods para hashes

```ruby
my_pet.each { |key, value| puts "the #{key} of my pet is #{value}" }
# ⇒  the name of my pet is Milah
#    the age of my pet is 3
#    the breed of my pet is Golden
#    the favorite_toy of my pet is a sheep
```

Bloques

Un trozo de código
Puede estar encerrado en paréntesis {} o un bloque do..end
Se ejecuta cuando se invoca, no cuando el intérprete lo lee
yield se usa para invocarlo

```ruby
def do_something
  puts 'start of the method'
  yield
  yield
  puts 'end of the method'
end

do_something {puts 'inside the method'}
# =>  start of the method
#     inside the method
#     inside the method
#     end of the method
```

Parámetros en Bloques

Un bloque recibe parámetros especificados entre ||

```ruby
def duplicate
  yield('first yield', 2)
  yield('second yield', 5)
end

duplicate {|str,num| puts str + ': ' + (2 * num).to_s}
# =>  first yield: 4
#     second yield: 10
my_pet.each { |key, value| puts "the #{key} of my pet is #{value}" }
# =>  the name of my pet is Milah
#     the age of my pet is 3
#     the breed of my pet is Golden
#     the favorite_toy of my pet is a sheep
```

Procs y lambdas

Un bloque no es un objeto, es un trozo de código. Pero se puede volver objeto usando Proc o lambda
Para invocar se usa el método .call

```ruby
youLike = Proc.new do |something_yummy|
  puts "¡I really like #{something_yummy}!"
end

youLike.call 'chocolate'
# => ¡I really like chocolate!
youLike.call 'ruby'
# => ¡I really like ruby!
```

```ruby
aBlock = lambda { |x| puts x }
# => <Proc:0x0000000000f2ce98 (irb):1 (lambda)>
aBlock.call 'Hello!'
# => Hello!
```

Los procs pueden ser pasados como parámetros a un método.
Dentro del método se deben invocar con el método .call

```ruby
def my_method aProc
  puts 'inside my method'
  aProc.call('Milah')
  puts 'end of my method'
end

my_proc = Proc.new do |name|
  puts "my pet's name is #{name}"
end

my_method my_proc
# =>  inside my method
#     my pet's name is Milah
#     end of my method
```

Classes

Se definen con la palabra reservada class

Nombre de la clase debe partir con Mayúscula (por ser constante)

El método initialize es el constructor

Atributos son llamados variables de instancia y se definen con @ inicial

```ruby
class BankAccount

  def initialize(account_name, account_number)
    @account_name = account_name
    @account_number = account_number
    @balance = 0.0
  end

  def withdraw amount
    @balance -= amount
  end

  def deposit amount
    @balance += amount
  end

end

account = BankAccount.new('Cuenta corriente', '123')
puts account.deposit(100)
puts account.withdraw(30)
```

Para acceder a las variables de instancia desde fuera de la clase se debe crear un método

Estos métodos son llamados getters

```ruby
class BankAccount

  def initialize(account_name, account_number)
    @account_name = account_name
    @account_number = account_number
    @balance = 0.0
  end

  def withdraw amount
    @balance -= amount
  end

  def deposit amount
    @balance += amount
  end

  def balance
    @balance
  end

  def account_name
    @account_name
  end

  def account_number
    @account_number
  end

end

account = BankAccount.new('Cuenta corriente', '123')
puts account.balance
puts account.account_name
puts account.account_number
```

attr_accesor,
attr_reader, attr_writer
nos facilita crear los
getters y setters

El bloque private permite
definir métodos que no
son accesibles fuera de
la clase

```ruby
class BankAccount
  attr_accessor :account_number, :account_name
  attr_reader :balance

  def initialize(account_name, account_number)
    @account_name = account_name
    @account_number = account_number
    @balance = 0.0
  end

  def withdraw amount
    if check_funds amount
      @balance -= amount
    end
  end

  def deposit amount
    @balance += amount
  end

  private

  def check_funds amount
    @balance - amount >= 0 ? true : false
  end
end
```

Subclases

Una clase A es una subclase de una clase B: class A < B

Todos los métodos y variables de B también estarán presentes en la clase A

La sentencia require B es necesaria si la clase B no está definida en el mismo archivo

```ruby
require 'BankAccount'

class NewBankAccount < BankAccount

  def customerPhone
    @customerPhone
  end

  def customerPhone=( value )
    @customerPhone = value
  end

end
```

Scope de Constantes

Define en qué parte del programa una constante es accesible

```
MY_CONSTANTS = 'Some value'
```

Nombre se escribe con mayúscula

Valor no puede cambiarse una vez asignado.

- Constantes declaradas dentro una clase o módulo serán accesible en cualquier lugar de esa clase o módulo
- Constantes declaradas fuera de una clase o de un módulo son consideradas globales

Scope de una variable

Tipos de scope: local, global

local

```ruby
_my_local_variable = 'some value'
another_local_variable = 'some value'
```

global $

```ruby
$a_global_variable = 'Do not use me'
```

```ruby
5.times do |number|
  puts number
end

puts number
```

arroja error

Scope de una variable

Tipos de scope: instance y class

instance @

```ruby
def withdraw amount
  if check_funds amount
    @balance -= amount
  end
end

def deposit amount
  @balance += amount
end
```

class @@

```ruby
@@a_class_variable = 'Accessible for all instances'
```

```ruby
class MyCustomMath
  @@pi = 3.1

  # We initialize our @value
  # attribute with value
  def initialize value
    @value = value
  end

  # This method sums value to
  # our current value stored
  # in @value
  def sum value
    @value += value
  end

  # This method calculate the
  # area of a circle with the
  # radius = @value
  def circleArea
    @@pi * (@value * @value)
  end

end
```

```ruby
my_number = MyCustomMath.new(2)
my_other_number = MyCustomMath.new(5)

puts my_number.sum(3)
# # ⇒ 5
puts my_other_number.sum(3)
# # ⇒ 8


puts my_number.circleArea()
# ⇒ 77.5
puts my_other_number.circleArea()
# ⇒ 198.4
```

```ruby
class MyCustomMath
  @@pi = 3.1

  # We initialize our @value
  # attribute with value
  def initialize value
    @value = value
  end

  # This method sums value to
  # our current value stored
  # in @value
  def sum value
    @value += value
  end

  # This method calculate the
  # area of a circle with the
  # radius = @value
  def circleArea
    @@pi * (@value * @value)
  end

  # This method change the
  # class variable @@pi
  def change_pi new_pi_value
    @@pi = new_pi_value
  end
end
```

```ruby
puts my_number.circleArea()
# => 77.5
puts my_other_number.circleArea()
# => 198.4

# Una instancia cambia el valor de pi
my_number.change_pi 3.14

puts my_number.circleArea()
# => 78.5

# Cambia para todas las instancias
puts my_other_number.circleArea()
# => 200.96
```