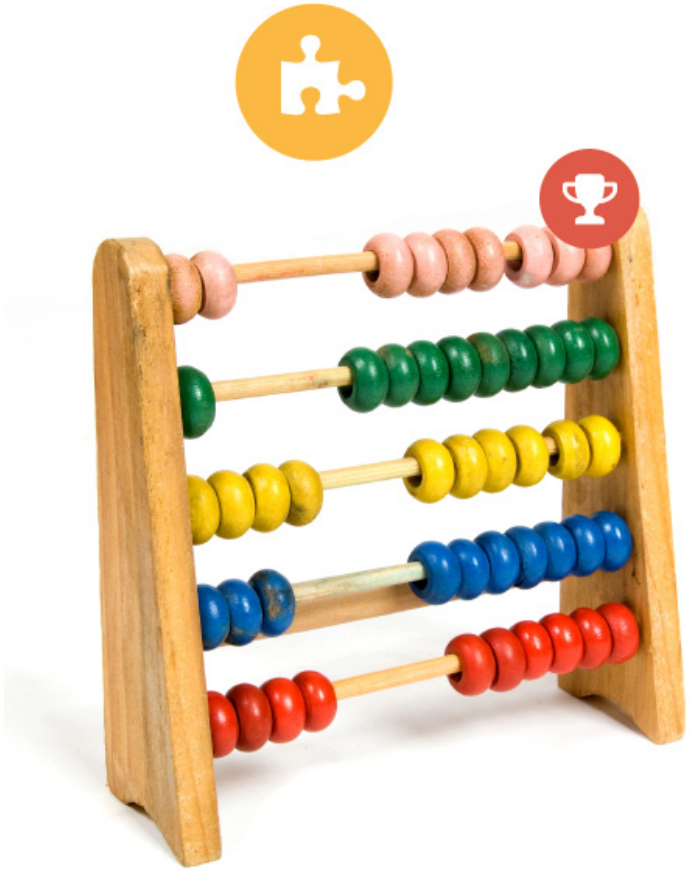
Programación en Lenguaje Ruby

Clases

Delio Tolivia

Gijón crece 2016 / 2019
AYTO. DE GIJÓN / CCOO / FADE / UGT
Ayuntamiento de Gijón
Gijón impulsa Empleo
dicampus TALENTO FORMACIÓN



Índice

Clases en Ruby

- Podemos saber la clase a la que pertenece un objeto con el método ***class***

```
2.0.0-p648 > 1.class
 => Fixnum
2.0.0-p648 > "".class
 => String
2.0.0-p648 > [].class
 => Array
2.0.0-p648 > {}.class
 => Hash
2.0.0-p648 :045 > 1.class.class
 => Class  #Las clases son objetos de la clase Class
```

- También tenemos el método ***is_a?*** (también ***kind_of?***) con el que podemos preguntar a un objeto si es de una clase en particular

```
2.0.0-p648 > 1.is_a?(Integer)
 => true
2.0.0-p648 > 1.is_a?(String)
 => false
```

Definir clases

- Para definir clases lo haremos de la siguiente manera

```
class Rectangle #Nombre de la clase
   def initialize(length, breadth) #Método de inicialización
                @length = length #Atributo
                @breadth = breadth #Atributo
   end

   def perimeter #Definicion de un método
                2 * (@length + @breadth)
   end
end
```

- Los métodos podrán aceptar parámetros y siempre retornan el valor de la ultima instrucción (podemos utilizar también la palabra reservada ***return***)

Creando objetos

- Una vez tenemos definida nuestra clase podemos crear un objeto con new que se inicializará según el método ***initialize*** que hemos definido

```
2.0.0-p648 > r=Rectangle.new(5,6)
 => #<Rectangle:0x007f99a20af7d8 @length=5, @breadth=6>
```

- Y podemos hacer llamadas al método ***perimeter***

```
2.0.0-p648 > r.perimeter
 => 22
```

Métodos (I)

- Los métodos pueden recibir varios argumentos y estos a su vez pueden tener valores por defecto de forma que si cuando llamamos el método no le pasamos dicho argumento por defecto toma ese valor

```
2.0.0-p648 > def add(n1, n2, n3 = 0)
2.0.0-p648 ?>     n1+ n2+ n3
2.0.0-p648 ?> end
 => nil
2.0.0-p648 > add(1,2)
 => 3
2.0.0-p648 > add(1,2,3)
 => 6
```

Métodos (II)

- Podemos indicar que nuestro método recibe un número variable de argumentos (a modo de array) con el operador *

```
2.0.0-p648 > def add(*numbers)
2.0.0-p648 ?>   numbers.inject(0) {|sum, number| sum+number}
2.0.0-p648 ?> end
 => nil
2.0.0-p648 > add(1)
 => 1
2.0.0-p648 > add(1,2)
 => 3
2.0.0-p648 > add(1,2,3)
 => 6
2.0.0-p648 > add(1,2,3,4)
 => 10
```

Métodos (III)

- El operador * también puede ser utilizado para pasar de un array a una lista de argumentos

```
2.0.0-p648 > def add(n1,n2,n3)
2.0.0-p648 ?>   n1+n2+n3
2.0.0-p648 ?>   end
 => nil
2.0.0-p648 > numbers_to_add = [1,2,3]
 => [1, 2, 3]
2.0.0-p648 > add(*numbers_to_add)
 => 6
2.0.0-p648 > add(numbers_to_add)
ArgumentError: wrong number of arguments (1 for 3)
```

Métodos (IV)

- También un argumento puede ser un hash

```
2.0.0-p648 > def add(n1, n2, options = {})
2.0.0-p648 ?>   sum = n1 + n2
2.0.0-p648 ?>   sum = sum.abs if options[:absolute]
2.0.0-p648 ?>   sum = sum.round(options[:precision]) if options[:round]
2.0.0-p648 ?>   sum
2.0.0-p648 ?> end
 => nil
2.0.0-p648 > add(1.0134, -5.568)
 => -4.5546
2.0.0-p648 > add(1.0134, -5.568, absolute:true)
 => 4.5546
2.0.0-p648 > add(1.0134, -5.568, absolute:true, round:true, precision:2)
 => 4.55
```

Métodos (V)

```
def add(*numbers)
  numbers.inject(0) { |sum, number| sum + number }
end

def subtract(*numbers)
  current_result = numbers.shift
  numbers.inject(current_result) { |current_result, number| current_result - number }
end

def calculate(*arguments)
  # if the last argument is a Hash, extract it
  # otherwise create an empty Hash
  options = arguments[-1].is_a?(Hash) ? arguments.pop : {}
  options[:add] = true if options.empty?
  return add(*arguments) if options[:add]
  return subtract(*arguments) if options[:subtract]
end
```

Ejercicio

- Realizar el “Ejercicio 1” del pdf de ejercicios.

Herencia

- En Ruby indicamos que estamos heredando de otra clase con el operador <

```
class Rectangle
  def initialize(length, breadth)
    @length = length
    @breadth = breadth
  end

  def perimeter
    2 * (@length + @breadth)
  end
end

class Square < Rectangle
  def initialize(length)
    @length=length
    @breadth = length
  end
end
```

Sobreescribir métodos (I)

- Podemos sobrescribir un método en una clase hija. La idea es que el método se va a comportar de forma distinta en nuestra clase hija que en la clase padre.

- Esto lo hemos hecho en el ejemplo anterior. Hemos sobrescrito el método ***initialize*** en nuestra clase hija ***Square*** sin modificar el comportamiento del método en la clase ***Rectangle***.

Sobreescribir métodos (II)

- Siempre podemos llamar al método sobrescrito de la clase padre mediante el uso de ***super***.

```
class Animal
  def move
    "I can move"
  end
end

class Bird < Animal
  def move
    super + " by flying"
  end
end

puts Animal.new.move # Devolvera “I can move”
puts Bird.new.move # Devolvera “I can move by flying”
```

Getters y setters (I)

- Ruby por defecto trata los atributos de una clase como privados. Para poder acceder a ellos desde fuera del objeto necesitamos los métodos que se conocen como getters y setters. Se pueden definir el getter con el nombre del atributo y el setter con el nombre seguido del símbolo =

```
class Item
  def initialize(item_name, quantity)
    @item_name = item_name
    @quantity = quantity
  end

  def quantity=(new_quantity)
    @quantity = new_quantity
  end

  def quantity
    @quantity
  end
end
```

```
item = Item.new("a",1)
item.quantity = 3
p item.quantity
```

Getters y setters (II)

- Ruby en caso de que nuestro getter y/o setter no haga nada mas que o bien devolver el valor directamente del atributo o bien darle el valor que recibimos nos ofrece unas palabras clave para generarlos de forma sencilla

```
class Item
  attr_writer :description # Crea el atributo y el setter
  attr_reader :color # Crea el atributo y el getter
  attr_accesor :size # Crea el atributo, el getter y el setter

  def initialize(description, color, size)
    @description = description
    @color = color
    @size = size
  end
end
```

Ejercicio

- Hacer el ejercicio 6 del pdf de ejercicios
- Hacer el ejercicio 12 del pdf de ejercicios

Métodos de clase

- Existen unos métodos particulares conocidos como métodos de clase. Estos métodos no son llamados sobre un objeto si no sobre la clase en si. Tienen la particularidad de que no pueden acceder a los atributos de la clase y solamente podrán acceder a las llamadas variables de clase. Podemos definirlos de dos formas

```
class Item
  def self.show
    puts "Class method show invoked"
  end
end
```

```
class Item
  class << self
    def show
      puts "Class method show invoked"
    end
  end
end
```

```
Item.show
```

Variables de clase (I)

- Las variables de clase se definen con el prefijo @@. Pueden ser accedidas por los metodos de clase y los métodos de instancia. Solamente deben usarse para almacenar información propia de la clase. Se suelen usar para almacenar información de configuración como nombre de la aplicación, versión, base de datos….

```
class Planet
  @@planets_count = 0

  def initialize(name)
    @name = name
    @@planets_count += 1
  end

  def self.planets_count
    @@planets_count
  end
end

Planet.new("earth"); Planet.new("uranus")

p Planet.planets_count #Devolvera 2
```

Variables de clase (II)

- Un problema con las variables de clase es cuando hacemos herencia. Al heredar si cualquiera de las clases hijas modifica el valor de una variable de clase de la clase padre este valor se modifica para todas ellas así como para la clase padre

```
class ApplicationConfiguration
  @@configuration = {}

  def self.set(property, value)
    @@configuration[property] = value
  end

  def self.get(property)
    @@configuration[property]
  end
end

class ERPApplicationConfiguration < ApplicationConfiguration
end

class WebApplicationConfiguration < ApplicationConfiguration
end
```

```
ERPApplicationConfiguration.set("name", "ERP Application")
WebApplicationConfiguration.set("name", "Web Application")

p ERPApplicationConfiguration.get("name") # Web Application
p WebApplicationConfiguration.get("name") # Web Application

p ApplicationConfiguration.get("name") # Web Application
```

Class instance variables

- En el ejemplo anterior vimos el problema de las variables de clase con la herencia. Para solventarlo disponemos de las variables de instancia de clase (class instance variables).

```
class Foo
  @foo_count = 0 # Se le da valor en la clase

  def self.increment_counter # Solo accedemos con métodos de clase
    @foo_count += 1
  end

  def self.current_count
    @foo_count
  end
end

class Bar < Foo
  @foo_count = 100
end
```

```
Foo.increment_counter
Bar.increment_counter
p Foo.current_count # Devolverá 1
p Bar.current_count # Devolverá 101
```

Igualdad de objetos (I)

- Para chequear la igualdad de objetos estándar (String, numeros, arrays, hash, etc.) se puede hacer mediante el operador **==**

puts [1,2] == [1,2]
puts [1,2] == [1,3]

puts "a" == "xyz”

Igualdad de objetos (II)

- En las clases que definamos el operador **==** no funcionará pues no compara el contenido de los objetos para comprobarlo. Por lo que debemos implementar el operador **==** (tener en cuenta que es igual que escribir ***a.==(b)***). De esta forma podemos también reimplementar cualquiera de los operadores.

```
class Item
  attr_reader :item_name, :qty

  def initialize(item_name, qty)
    @item_name = item_name
    @qty = qty
  end
  def to_s
    "Item (#{@item_name}, #{@qty})"
  end
  def ==(other_item)
    @item_name==other_item.item_name&&@qty==other_item.qty
  end
end
```

Igualdad de objetos (III)

- Existen algunos métodos de comparación como ***uniq*** de la clase ***Array*** (que debería retornar solamente los elementos únicos de un array) que no utilizan el método **==**. Estos métodos utilizan otros dos métodos para sus cálculos, son el método ***eql?*** Y el método ***hash***.

```
class Item
  attr_reader :item_name, :qty

  def initialize(item_name, qty)
    @item_name = item_name
    @qty = qty
  end

  def to_s
    "Item (#{@item_name}, #{@qty})"
  end
end
```

```
def eql?(other_item)
  @item_name == other_item.item_name && @qty == other_item.qty
end
```

```
def hash
  self.item_name.hash ^ self.qty.hash
end
```

- No confundir el método ***hash*** con los ***Hash*** que hemos visto antes. Este es un método que devuelve un ***hash code*** que es un entero único para cada objeto. Por lo que para comparar dos objetos simplemente hay que comparar estos enteros.

Mostrando objetos (I)

- Para mostrar el contenido de un objeto utilizamos el método ***puts*** y el método ***p***. La diferencia esta en que el método ***puts*** normalmente llama al método ***to_s*** del objeto mientras que el método ***p*** invoca el método ***inspect***.

- Por defecto ***puts*** nos muestra la dirección de memoria donde está el objeto mientras que ***p*** nos muestra esa información más el contenido de los atributos del objeto.

```
class Item
  def initialize(item_name, qty)
    @item_name = item_name
    @qty = qty
  end
end

item = Item.new("a",1)

puts item # #<Item:0x007f465c037a10>
p item # #<Item:0x007f465c037a10 @item_name="a", @qty=1>
```

Mostrando objetos (II)

- Si queremos que se muestre la información de una manera más comprensible podemos sobrescribir el método ***to_s***

```
class Item
  def initialize(item_name, qty)
    @item_name = item_name
    @qty = qty
  end
  def to_s
    "#{@item_name} #{@qty}"
  end
end

item = Item.new("a",1)

puts item # a 1
p item # a 1
```

- Aunque solamente hemos sobrescrito el método ***to_s, p*** se comporta como si llamase a ***to_s*** en vez de a ***inspect***. Ruby si sobrescribimos el método ***to_s*** hace esto por defecto. Si queremos que ***p*** se comporte de otra manera habría que sobrescribir ***inspect***

Serializando objetos (I)

- Cuando se quiere pasar información entre dos programas muchas veces se realiza mediante archivos. Para guardar nuestros objetos en dichos archivos y que desde el otro programa se puedan recuperar se pasan a un String con un formato definido, esto se define como serializar el objeto. Podemos hacerlo mediante el método ***to_s*** para generar el String y el método ***from_s*** para el paso inverso

```
class CerealBox
  attr_accessor :ounces, :contains_toy

  def initialize(ounces, contains_toy)
    @ounces = ounces
    @contains_toy = contains_toy
  end

  def self.to_boolean(str)
    str == 'true'
  end
end
```

```
def self.from_s(s)
    ounces = 0
    contains_toy = false
    s.each_line do |field|
      value = field.split(":")[1].strip
      ounces = value.to_i if field.include?("ounces")
      contains_toy = to_boolean(value) if field.include?("contains_toy")
    end
    CerealBox.new(ounces, contains_toy)
  end
```

```
def to_s
    "@ounces: #{@ounces}\n @contains_toy: #{@contains_toy}"
  end
```

Serializando objetos (II)

- La solución anterior es muy costosa de realizar y será diferente para cada clase. Por lo tanto se suelen utilizar formatos estándar como XML, JSON o YAML. Podemos producir cualquiera de ellos pero vamos a prestar especial atención a YAML pues Ruby tiene las herramientas para utilizarlo directamente con los métodos ***YAML::dump*** y ***YAML::load***

```
def self.deserialize(yaml_string)
  YAML::load(yaml_string)
end

def serialize
  YAML::dump(self)
end
```

```
yaml = our_object.serialize

our_new_object = Item.deserialize (yaml)
```