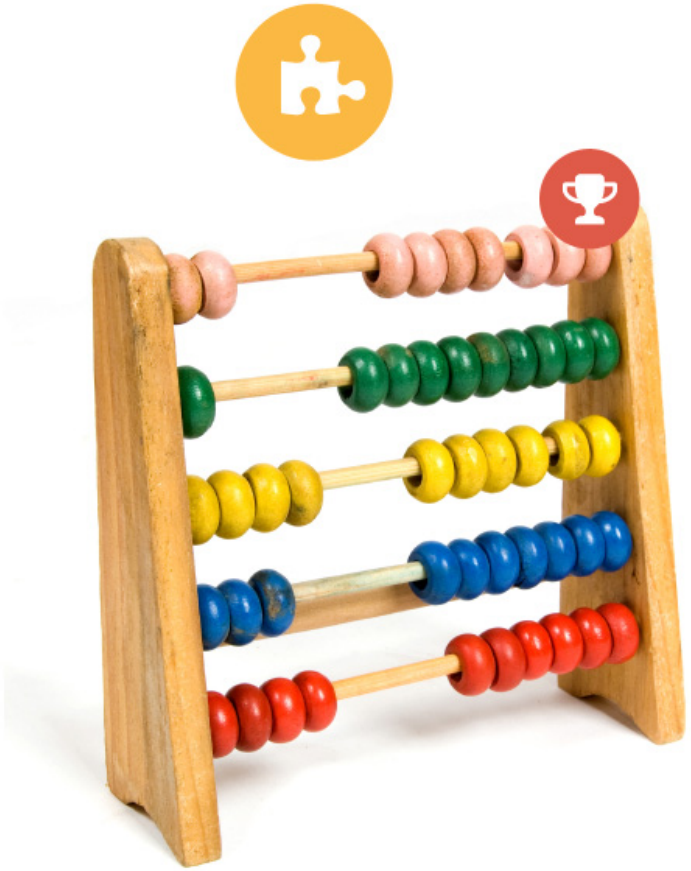
Programación en Lenguaje Ruby

Módulos

Delio Tolivia

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



Índice

Módulos

Módulos como Namespaces

Operador ::

Método extend

Callbacks de include y extend

Módulos

- Los módulos sirven para crear grupos de métodos que luego se podrán incluir o mezclar con las clases que se necesite. Los módulos solamente tienen métodos a diferencia de las clases que además tienen atributos.

- Los módulos no pueden ser instanciados por lo que sus métodos no pueden ser llamados directamente. Deben incluirse en otra clase para que les puedan usar en las instancias de estas (no sirve para métodos de clase).

- Para incluir un módulo en una clase se utiliza el método ***include*** que recibe como parámetro el nombre del módulo.

- Los módulos son instancias de la clase ***Module***. A su vez ***Module*** es la superclase de ***Class*** por lo que las clases a su vez son módulos

```
module WarmUp
  def push_ups
    "Phew, I need a break!"
  end
end
```

```
class Gym
  include WarmUp

  def preacher_curls
    "I'm building my biceps."
  end
end

class Dojo
  include WarmUp

  def tai_kyo_kyu
    "Look at my stance!"
  end
end
```

Módulos como Namespaces

- Los Namespaces son una forma de reunir de forma “lógica” objetos relacionados. Los módulos se pueden utilizar de esta manera para evitar conflictos por ejemplo entre distintas clases al repetirse nombres.

```
module Perimeter
  class Array
    def initialize
      @size = 400
    end
  end
end

our_array = Perimeter::Array.new
ruby_array = Array.new

p our_array.class # Devolverá Perimeter::Array
p ruby_array.class # Devolverá Array
```

Operador :: (I)

- En el ejemplo anterior utilizamos el operador ***::*** para referirnos a una clase dentro de un módulo. También puede ser utilizado para buscar constantes y no solamente clases.

```
module Dojo
  A = 4
  module Kata
        B = 8
    module Roulette
      class ScopeIn
        def push
          15
        end
      end
    end
  end
end

A = 16
B = 23
C = 42
```

```
puts "A - #{A}"    #Mostrará 16
puts "Dojo::A - #{Dojo::A}" #Mostrará 4

puts "B - #{B}" #Mostrará 23
puts "Dojo::Kata::B - #{Dojo::Kata::B}" #Mostrará 8


puts "C - #{C}" Mostrará 42
puts "Dojo::Kata::Roulette::ScopeIn.new.push –
      #{Dojo::Kata::Roulette::ScopeIn.new.push}" #Mostrará 15
```

Operador :: (II)

- Si utilizamos el operador *::* sin un nombre a su izquierda devolverá el elemento de su derecha del nivel más alto.

```
module Kata
  A = 5
  module Dojo
    B = 9
    A = 7

    class ScopeIn
      def push
        ::A
      end
    end
  end
end

A=10
```

Kata::Dojo::Scopein.new.push #Devolverá 10

Método extend (I)

- El método ***extend*** permite añadir métodos a un objeto igual que hace ***include*** pero además nos permite realizarlo a nivel de un objeto.

```
module Foo
  def module_method
    puts "Module Method invoked"
  end
end

class Bar
end

bar=Bar.new
bar.extend Foo
bar.module_method # ”Module Method invoked”
```

Método extend (II)

- Otra función más utilizada para el método ***extend*** es añadir los métodos de un módulo como métodos de clase

```
module Foo
  def say_hi
    puts "Hi!"
  end
end

class Bar
end

Bar.extend Foo
Bar.say_hi # ”Hi!”
```

Callbacks de include y extend (I)

- El método ***included*** es un callback de ***include***, es decir se llamará automáticamente cuando se incluye el módulo con ***include***

```
module Foo
  def self.included(base)
    puts "Class #{base} has been extended with module #{self} !"
  end
end

class Bar
  include Foo
end

# Class Bar has been extended with module Foo !
```

- Lo mismo ocurre con el método ***extended*** como callback de ***extend***