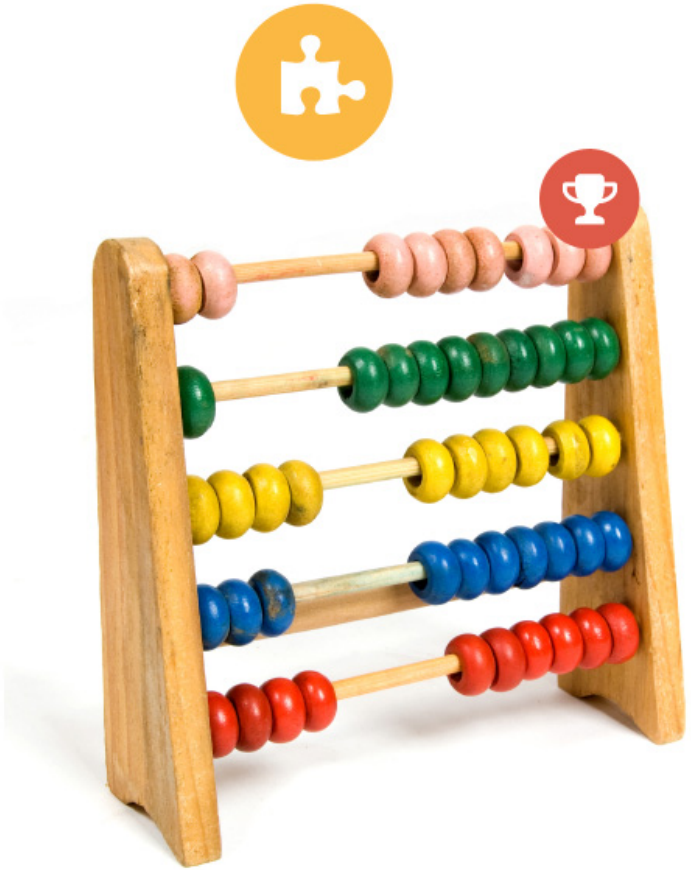
Programación en Lenguaje Ruby

Lambdas y Bloques

Delio Tolivia

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



Índice

Lambdas

Métodos de clase en variables

Bloques

Lambdas (I)

- Las lambda son funciones anónimas sin nombre. Su valor de retorno es el de la ultima instrucción que ejecuten. En Ruby son objetos y por tanto pueden ser asignadas a una variable y tienen sus propios métodos como ***call***

```
2.0.0-p648 > l = lambda { "Do or do not" }
 => #<Proc:0x007fc8439a5250@(irb):31 (lambda)>
2.0.0-p648 > puts l.call
Do or do not
 => nil
```

- También pueden recibir argumentos entre barras verticales | | y se pueden definir con { } o con do…end (esta ultima versión es la recomendada para cuando tienen más de una linea de código)

```
l = lambda do |string|
  if string == "try"
    return "There's no such thing"
  else
    return "Do or do not."
  end
end
```

Lambdas (II)

- Hay una sintaxis alternativa para crear lambdas con el uso de ->

```
short = ->(a, b) { a + b }
puts short.call(2, 3)

long = lambda { |a, b| a + b }
puts long.call(2, 3)
```

Ejercicio

- Hacer el ejercicio 13 del pdf de ejercicios

Métodos de clase en variable

- Las lambda son objetos de la clase ***Proc***. Los métodos de una clase son simplemente un bloque de código asociado a un objeto que accede a los atributos del mismo. Podemos guardar en una variable un método de una clase de la siguiente forma

```
2.0.0-p648 > class Calculator
2.0.0-p648 ?>   def add(a, b)
2.0.0-p648 ?>     return a + b
2.0.0-p648 ?>     end
2.0.0-p648 ?>   end
 => nil
2.0.0-p648 > addition_method = Calculator.new.method("add")
 => #<Method: Calculator#add>
2.0.0-p648 > addition = addition_method.to_proc
 => #<Proc:0x007fc8439564c0 (lambda)>
2.0.0-p648 > puts addition.call(5,6)
11
 => nil
```

Bloques (I)

- Los bloques son trozos de código que no pueden almacenarse en variables y que no son objetos. Son más rápidos que las lambda pero no tan versátiles.

```
def demonstrate_block(number)
  yield(number) # yield ejecuta un bloque que se pase "implicitamente" fuera de la lista de argumentos
end

puts demonstrate_block(1) { |number| number + 1 } # Pasamos un argumento y un bloque
```

- Si nuestra función espera un bloque para ser ejecutado con ***yield*** y no se lo pasamos de forma implícita nos dará un error con una excepción de tipo ***LocalJumpError.*** Para que esto no ocurra tenemos el método ***block_given?*** que nos permite asegurarnos de solo ejecutar el ***yield*** si hemos recibido un bloque

```
def demonstrate_block(number)
  yield(number) if block_given?
end
```

Bloques (II)

- Los bloques también pueden ser pasados de forma explicita como un argumento a la función

```
def calculation_with_explicit_block_passing(a, b, operation)
  operation.call(a, b)
end

addition = lambda { |a, b| a + b }
calculation_with_explicit_block_passing(5, 5, addition)
```

- De todas formas el uso de ***yield*** es mucho más rápido. Si medimos el tiempo que tarda esta función en comparación con la misma pasando el bloque de forma implícita podríamos ver lo siguiente:

```
Rehearsal ----------------------------------------------
explicit   0.000000   0.000000   0.000000 (  0.000202)
implicit   0.000000   0.000000   0.000000 (  0.000128)
------------------------------------- total: 0.000000sec
```

Bloques (III)

- Ruby nos permite pasar los bloques de explícitos a implícitos y viceversa con el uso de & (al definirlo como explicito debe de ser el último parámetro para que funcione)

```
def calculation(a, b, &block) # Aqui tenemos un bloque explicito
 block.call(a, b)
end

puts calculation(5, 5) { |a, b| a + b } # Podemos pasarlo como implicito#


def calculation(a, b)
  yield(a, b) # esperamos un bloque implicito
end

addition = lambda {|x, y| x + y}
puts calculation(5, 5, &addition) #Pasamos el bloque implicito
```

Ejercicio

- Hacer el ejercicio 16 del pdf de ejercicios