Métodos en Ruby

> Análogo a una función en otros lenguajes, pero no estrictamente similar.

```ruby
def saludar                                 # Nombre del método, precedido de def
    puts "hola"
    puts "yo"
    puts "soy"                              # Cuerpo del método
    puts "un"
    puts "conjunto"
    puts "de"
    puts "instrucciones"
    puts "agrupadas"
end
```

Llamando este ejemplo en terminal:

```ruby
ruby saludar.rb                         # Llamada del método
```

En ruby los paréntesis son optativos para declarar métodos, utilizándose más para casos con más de 1 parámetro a ingresar en el método. El método puede terminar con `!` y `?`, dependiendo del caso.

Paso de parámetros

> Poder ingresar valores externos al método.

```ruby
def saludar(nombre)
    puts "hola #{nombre}"
end

saludar("Anacleto")
#Hola Anacleto
saludar("Eulalia")
#Hola Eulalia
```

Se puede especificar más de un parámetro separándolos con comas `,`

```ruby
def sumar(a, b)
    puts a + b
end

sumar(2, 4)
# 6
```

Se pueden agregar valores por defecto

```ruby
def incrementar(valor, incremento = 1)
    puts valor + incremento
end

incrementar(2)
# 3, incremento constante en 1
incrementar(5, 2)
# 7, 5 más un incremento especificado de 2.
```

Ejercicio 1

> Crear un método que muestre un saludo. Debe reconocer un parámetro "bye", donde entrega bye bye en la pantalla.

```ruby
def saludo(cadena)
    if cadena == 'bye'
        puts 'bye bye'
    else
        puts 'wena choro'
    end
end

saludo()
# wena choro
saludo(cadena  = 'loro')
# wena choro
saludo(cadena = 'bye')
# bye bye
```

Retorno

> Hacer que el método devuelva un valor post-ejecución.

```ruby
def suma(a, b)
    return a + b
end

puts suma(5, 6)
# 11
```

Como lenguaje objeto orientado, se prefiere devolver un valor dado que nos permite volver a trabajar este. Todos los métodos de ruby se devuelven a la última línea.

```ruby
suma1 = suma(7, 3)
puts suma1
# 10
```

Suma pares

> Crear un método que devuelva la suma de los números pares entre 1 y `n`, donde `n` es un parámetro.

```ruby
def suma_pares(n)
    (1..n).each do |i|
        suma += i if i.even?
    end
    return suma
end


puts suma_pares(10)
# 30
```

Validación porcentajes

> Crear un método que reciba un porcentaje. Si el porcentaje está entre 0 y 100, TRUE. else FALSE

```ruby
def validar_porcentaje(porcentaje)
    if porcentaje >= 0 && porcentaje <= 100
        return true
    else
        return false
    end
end

puts validar_porcentaje(60)
# true
puts validar_porcentaje(12312)
# false
puts validar_porcentaje(sqrt(-1))
# go fsck yourself.
```

Scope de una variable

> Hay que definir el alcance de las variables. Estas difieren entre aquellas que son **locales** y **globales**. Las variables locales pueden vivir en un bloque `{}` o `do ... end` o un método `def ... end`. El scope está asociado al bloque.

Scope de una variable dentro de un método

> Si utilizo una variable definida dentro de un método fuera de éste último, la variable no tendrá valor y arrojará un error. Cada método tiene su propio juego de variables locales.

```ruby
def foo()
    j = 5
end

def bar()
    j = 7
end

j = 3
puts j
# 3
foo()
# 3
bar()
# 3
```

`j` no cambia en este caso.

Stacktrace

> Stack: jerarquización y agrupación de métodos. Afectan el comportamiento de las variables.

Cuando `ruby` muestra errores en nuestro código, el stacktrace nos indica en qué método está el error y qué metodo lo llamó.

Ejemplo:

```ruby
def bar()
    error
end

def foo()
    bar
end

foo
```

el error en el terminal sería:

```
$ ruby example.rb
example.rb:2:in `bar': undefined local variable or method `error' for main:Object(NameError)
        from example.rb:6:in `foo'
        from example.rb:9:in `<main>'
```

Require

> Importar código externo. `require_relative` en base al camino relativo del archivo ejecutor. `require` para gems y otras librerías.

```ruby
# second.rb

puts "hola"

def suma(a, b)
    a + b
end
```

```ruby
# main.rb
require_relative 'second'

suma(2, 3)
# 5. Desde second.rb
```

Arrays

> Definición: contenedores de múltiples datos y diversos tipos.

```ruby
a = [1, "3", "string", :simbolo]

#averiguar clase
a.class
# Array
```

Formas de crear arrays:

1. Creadas con `[]`:

```
    a = [1, 5, "9", 4, "hola"]

    a = []
    ```

2. Creadas con clase `Array`:

```ruby
    a = Array.new(3, 0)

    #prompts zero three times

    a = Array.new(4, "muu")
    #prompts muu four times
    ```

Se puede ingresar a los arrays por medio de los índices, partiendo en 0.

```ruby
a = [1, 2, 3, 4,  "hola", :simbolo]
puts a[0]
# 1
puts[4]
# hola
```

`puts` imprime los arreglos hacia abajo. `print` imprime los arreglos al lado

Iteraciones

```ruby
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
for element in a do
    puts element
end

a.each do |i|
    puts i + 1
end

a.each { |i| puts i + 1 }
```

Método para para saber si un elemento está en un arreglo

```ruby
def check_if_exists?(array, match)
    array.each do |i|
        return true if i == match
    end
    false
end

a  = [1, 2, 3, 4, 'hola', :simbolo]
```

Each*with*index

> Iterando en un array con index.

```ruby
a = ["ich", "bin", "eine", "arreglo"]
a.each_with_index do |value, index|
    puts "#{index}-> #{value}"
end

#0 -> ich
#1 -> bin
#2 -> eine
#3 -> arreglo
```

```ruby
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
a.each_with_index do |ele, i|
    if ele % 3 == 0
        puts "foo #{ele}"
    end
    if ele % 4 == 0
        puts "bar #{i}"
    end
end
```

Métodos de arreglo

No es válido (no se agregan scalars and vectors)

```ruby
    [3, 6, 3] + 19
```

Válido (arreglos concatenados)

```
    [3, 6, 3] + [19]
```

Para ver los métodos, seguimos un objeto con `.methods`

```
def promedio(arreglo)
    suma = 0
    arreglo.each do |ele|
        suma += ele
    end
    suma / arreglo.count.to_f
end

a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
putpromedio(a)
```