Hash en `ruby`

Objetivos:

1. Símbolos
2. Hashes
3. Operaciones `map reduce` 4.1 Map y collect 4.2 Select y reject 4.3 Inject 4.4 Group_by

Símbolos

> Similares a los strings, preceden de un `:`. A diferencia de los strings, no son mutables (difíciles de cambiar).

Este ejemplo, no se puede **concatenar** símbolos, por lo que arroja un error.

```ruby
puts 'hola' + 'hola'
# holahola
puts :hola + :hola
# `<main>': undefined method `+' for :hola:Symbol (NoMethodError)
```

Los símbolos se utilizan en lugar de los strings cuando tenemos estados o una configuración. Los strings son más flexibles, y los símbolos son más rápidos.

Prueba de velocidad

```ruby
require 'benchmark'

str = Benchmark.measure do
    30_000_000.times do
        'prueba'
    end
end.total

sym = Benchmark.measure do
    30_000_000.times do
        :prueba
    end
end.total

puts 'String: ' + str.to_s
puts 'Symbol: ' + sym.to_s
```

Hash

> Tipo de contenedor que se indexan por una **clave**, a diferencia del array que se indexa por un índice. La sintaxis del hash tiene los siguientes componentes:
>
> $$\text{Valor} \mapsto \text{objeto}$$

```ruby
a = {'clave1' => 'valor1', 'clave2' => 'valor2'}
```

Con array

```ruby
products = ['Producto1', 'Producto2', 'Product3']
prices = [100, 150, 210]

price_indez = products.index('Producto2')
puts prices[price_index]
```

Con hash

```ruby
products = { 'Producto1' => 100, 'Producto2' => 150, 'Product3' => 210}
puts products['Producto2']
```

Uso básico de hash

```ruby
products = ['Producto1', 'Producto2', 'Product3']
prices = [100, 150, 210]

search = gets.chomp
puts prices[products.index(search)]
```

Claves como hash o símbolos

Formas de crear hash

```ruby
datos = {'foo' => 'bar', 'cow' => 'moo', 'cat' => 'meow', 'dog' => 'woof'}

a = Hash.new
```

```ruby
colors = {'red' => 'ff0000', 'green' => '00ff00', 'blue' => '0000ff'}
colors2 = {:red => 'ff0000', :green => '00ff00', :blue => '0000ff'}

colors2[:red]
```

Otra forma (via índices)

```ruby
hash = {a:5, b:'hola'}
puts hash[:a] #5
```

Iterando un hash

> Continuamos el hash con un `.each`, de manera similar a un loop en array. Pero a diferencia del array, acá se necesitan dos iteradores (para la clave y el valor)

```ruby
edades = {'Oscar': 27, 'Javier': 31, 'Francisca': 32, 'Alejandro': 19}

edades.each { |key, value| puts key }
edades.each { |key, value| puts value}
edades.each { |key, value| puts "#{key} tiene #{value} años"
```

Agregando y borrando elementos

Agregando

```ruby
hash = {}
hash['nuevo'] = 'valor'
hash[:otro_nuevo] = 'valor'

puts hash
```

Borrando

```ruby
edades = {'Oscar': 27, 'Javier': 31, 'Francisca': 32, 'Alejandro': 19}
edades.delete(:Javier)
puts edades
```

Convertir un array en hash

```ruby
array = [[1, 2], [3, 4], [5, 6], [7, 8]]

print array.to_h
```

`Object_id`

> Con `object_id` podemos saber si dos variables referencian al mismo objeto. Devuelve un identificador único del objeto.

```ruby
a = 2
puts a.object_id

a = 3
puts a.object_id
```

Si dos variables tienen el mismo `object_id`, es peligroso hacer cambios

Caso Seguro

```ruby
a = 'hola'
b = 'hola'
puts a.object_id == b.object_id
# false
```

Caso Inseguro

```ruby
a = 'hola'
b = a
puts a.object_id == b.object_id
# true
```

Con los símbolos, `object_id` no presenta el problema, dado que éstos son *inmutables*.

Caso Seguro

```ruby
a = :hola
b = :hola
puts a.object_id == b.object_id
# true
```

Caso Seguro

```ruby
a = :hola
b = a
puts a.object_id == b.object_id
# true
```

Mutabilidad en hash y array

En Arrays

```ruby
a = [9, 4, 6, 2.0, 'hola']
b = a

a.object_id == b.object_id
# true

puts b
# a = [9, 4, 6, 2.0, 'hola']

b[0] = 8
# a y b cambian [8, 4, 6, 2.0, 'hola']

puts a[8]
# 8
```

En Hashes

```ruby
datos = {'usuario': 'gonzalo', 'password':'secreto'}
otros_datos = datos

otros_datos['usuario'] = 'nueva_password'
# agregar nuevo valor a la clave 'usuario'

datos['usuario']
# nueva_password
```

Map y collect

Bloques

Se definen entre brackets, do y end. Son útiles para flexibilizar meétodos.

```ruby
a = [1, 2, 3, 1]

a.each { |x| x + 1}

a.each do |x|
    x + 1
end
```

Ejemplos

Código	Def
`.each (iterar)`	Bloque con espeficaciones de qué hacer
`.sort_by (ordenar)`	Bloque con criterio de ordenamiento
`.group_by (agrupar)`	Bloque con criterio de agrupamiento
`.reject (filtrar)`	Bloque con criterio para filtrar

Map y collect

> `.map` devuelve un array con el resutado de aplicar la operación especificada a cada elemento. Esto, a diferencia del `each`, que modifica elemento por elemento.

Forma normal

```ruby
a = [1, 2, 3, 4, 5, 6, 7]
b = a.map do |e|
    e*2
end
```

Forma express

```ruby
a = [1, 2, 3, 4, 5, 6, 7]
b = a.map { |e| e*2 }
```

Select y reject

> `select` devuelve un array con todos los elementos que cumplen la condición. `reject` devuelve un array con todos los elementos que no cumplen la condición.

```ruby
a = [1, 2, 3, 4, 5, 6, 7]
b = a.select { |x| x % 2 == 0 }
# se seleccionan los pares. Se devuelve
# 2, 4, 6
```

```ruby
b = a.reject { |x| x % 2 == 0 }
# [1, 3, 5, 7]
```

Ejercicio de map y select

```ruby
nombres = ['Violeta', 'Andino', 'Clemente', 'Javiera', 'Paula', 'Pia', 'Ray']
```

```ruby
nombres = ['Violeta', 'Andino', 'Clemente', 'Javiera', 'Paula', 'Pia', 'Ray']
nombres.each do |nombre|
    puts nombre.length
end
```

```ruby
nombres = ['Violeta', 'Andino', 'Clemente', 'Javiera', 'Paula', 'Pia', 'Ray']
largo = nombres.map do |nombre|
    nombre.length
end
print largo
```

```ruby
nombres = ['Violeta', 'Andino', 'Clemente', 'Javiera', 'Paula', 'Pia', 'Ray']
nombres_largos = nombres.select { |e| x.length > 5 }
puts nombres_largos
```

Inject

> Método para reducir

```ruby
b = a.inject(0) { |sum, x| sum + x}

    # donde inject(0) es el valor inicial.
    # sum es el acumulador.
    # x es el iterador
```

Group_by

> Agrupar datos o elementos por el criterio que deseemos.

Código	Def
`a.group_by { |ele| ele.class }`	Agrupar por tipo de dato
`a.group_by { |ele| ele}`	Agrupar por elemento
`a.group_by { |ele| ele.even?}`	Agrupar por condición

```ruby
a = [1, 2, 5, 1, 6, 2]
print a.group_by{|x| x.even?}

# Devuelve un hash con la siguiente estructura
# {false=>[1, 5, 1], true=>[2, 6, 2]}

print a.group_by{|x| x}
# Devuelve un hash con la estructura
# {1=>[1, 2], 2=>[2, 2], 5=>[5], 6=>[6]}
```