

DESAFIO **LATAM**

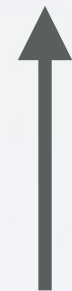
HOY APRENDEREMOS:

- **Símbolos**
- **Hashes (diccionarios)**
- **Operaciones map reduce**
 - Map y Collect
 - Select y Reject
 - Inject
 - Group_by

SÍMBOLOS

LOS SÍMBOLOS SON OTRO TIPO DE LITERAL

:hola



Empieza con :

¿PARA QUÉ SIRVEN?

- Son como los strings, pero no son mutables, eso quiere decir que no los podemos cambiar por error, además tienen una ventaja a nivel de memoria

UN EJEMPLO

```
puts 'hola' + 'hola' #holahola
```

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

¿PARA QUÉ SE OCUPAN?

- Se ocupan en lugar de los strings cuando tenemos estados o una configuración.
- Un semáforo podría estar en "rojo", "amarillo" o "verde", en este caso sería mejor ocupar los símbolos :rojo, :amarillo, :verde
- Y las ocuparemos en los hashes

¿QUÉ ES MEJOR, STRING O SÍMBOLO?

- Tienen distintos usos, los strings son más flexibles, los símbolos son más rápidos

PRUEBA DE VELOCIDAD

```
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 { }

```
a = {"clave1" => "valor1", "clave2" => "valor2"}
```

Los hash son otro tipo de contenedor que en lugar de indexarse por un índice, si indexan por una clave.

La sintaxis de los hash es clave: valor, donde la clave puede ser un string o símbolo y el valor puede ser cualquier objeto, ya sea un string (o sea cualquier tipo de dato inclusive otro hash)“

¿PARA QUÉ SIRVEN LOS HASHES?

Son otra forma de almacenar valores

EJEMPLO

Con arrays

```
products = ['Producto1', 'Producto2',  
            'Producto3']  
prices = [100, 150, 210]
```

Cómo podríamos obtener el precio
del producto2?

```
# 1) Obtenemos el índice del producto  
price_index =  
products.index("Producto2")
```

```
# 2) Utilizamos el índice en el  
segundo arreglo  
puts prices[price_index]
```

Con hash

```
products = {'product1' => 100,  
            'product2' => 150,  
            'product3' => 210}  
puts products['product2']
```

HASH { }

Creados con { }

```
data = {"foo" => "bar", "cow" => "moo"}
```

Creados con Clase Hash

```
a = Hash.new
```

La clave en los hashes debe ser única,
al igual que el índice en los array.

```
hash_vacio = {}
```

HASH { }

```
colors = {"red" => "ff0000", "green" => "00ff00", "blue" => "0000ff"}  
colors["red"]
```

LA CLAVE DE UN HASH PUEDE SER UN STRING O UN SÍMBOLO

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

```
colors2[:red]
```

*Se suele utilizar **símbolos** como claves

ACCEDIENDO A UN HASH

A través del índice

Los índices en el hash se llaman keys (claves)

```
hash = {a:5, b:"ho1a"}  
puts hash[:a] #5
```

hay que tener cuidado de referirse correctamente a la clave entendiendo que puede ser símbolo

STRING O SÍMBOLO

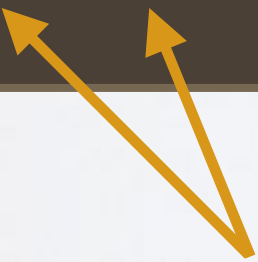
El índice de un diccionario puede ser un string
o un símbolo

```
hash = {"a"=>"hola", "b"=>"string"}  
puts hash["a"] #hola
```

ITERANDO UN HASH

Para iterar un hash se **.each** al igual que en un array

```
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" }
```



Se necesitan 2 iteradores, uno para las claves y el otro para los valores

AGREGANDO UN ELEMENTO A UN HASH

```
hash = {}  
hash["nuevo"] = "valor"  
hash[:otro_nuevo] = "valor"  
  
puts hash
```

BORRANDO UN ELEMENTO

```
notas = { Javier: 8, Julian: 6 }  
notas.delete(:Julian)  
puts notas
```

UN ARRAY SE PUEDE CONVERTIR EN HASH

```
array = [[1, 2], [3, 4], [5, 6]]  
print array.to_h
```

pero debe tener esta estructura

UN HASH SE PUEDE CONVERTIR EN ARRAY

```
# De Hash a Array  
hash = { 'Camila' => [4, 7, 3], 'Francisco' => [8, 1, 5] }  
print hash.to_a
```

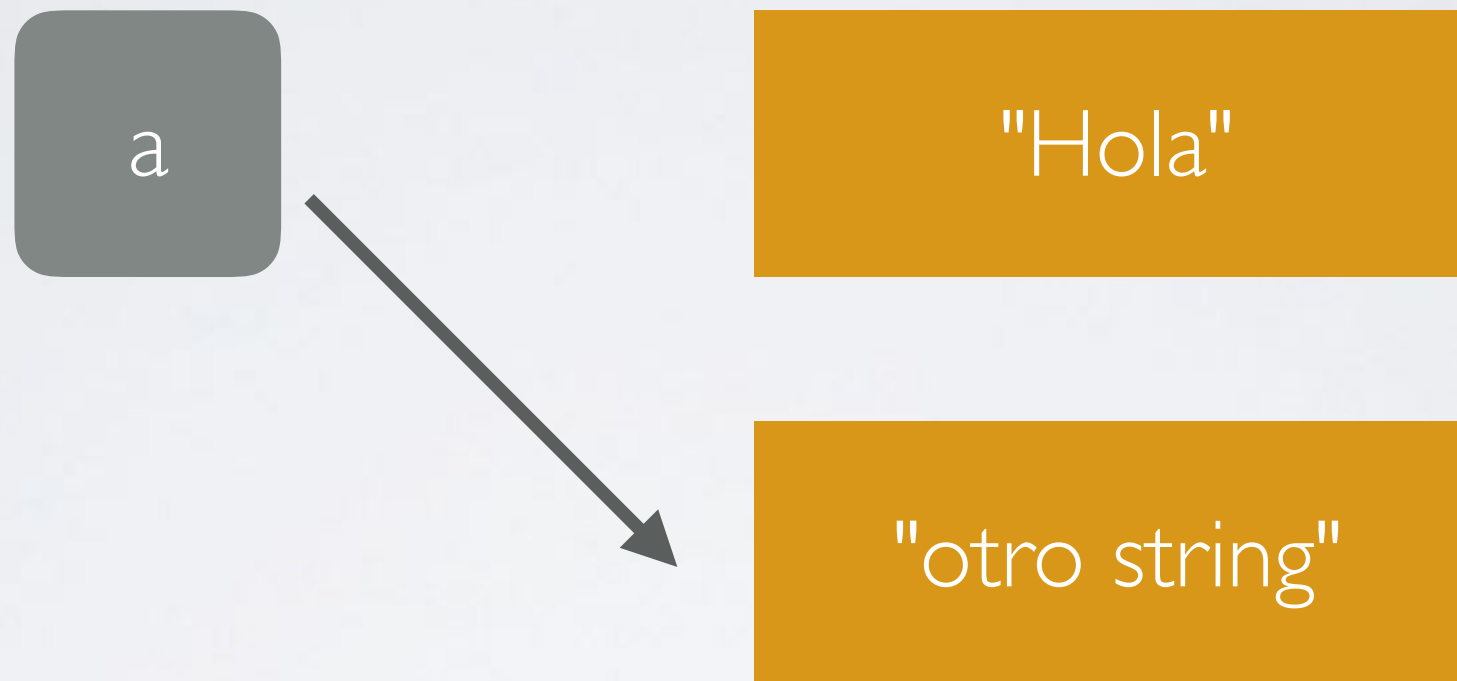
VARIABLES Y REFERENCIAS

ASIGNANDO UN OBJETO A UNA VARIABLE



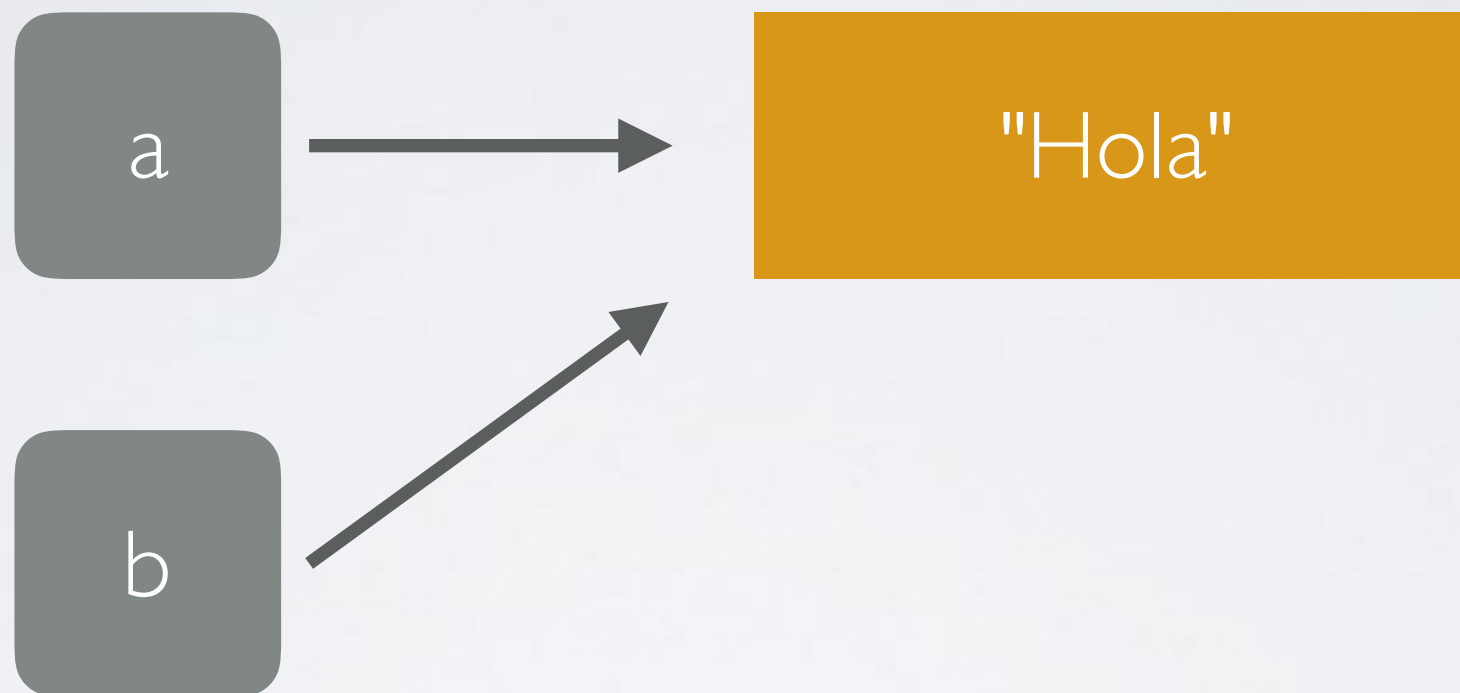
```
a = 'hola'  
puts a.object_id
```


ASIGNANDO UN OBJETO A UNA VARIABLE



```
a = 'hola'  
puts a.object_id  
  
a = 'otro string'  
puts a.object_id
```

VEAMOS UN EJEMPLO CON DOS VARIABLES



```
a = 'hola'
puts a.object_id # 70134114924760

b = a
puts b.object_id # 70134114924760
puts b == a # true
```

MUTABLE VS IMMUTABLE

- Un objeto mutable es aquel que puede cambiar una vez creado, un objeto inmutable es aquel que no puede cambiar.

Por ejemplo podemos cambiar un string con `.uppercase!`

***Esto suena muy técnico para ser necesario, pero las implicancias son muy importantes**

VEAMOS UN EJEMPLO CON DOS VARIABLES



```
a = 'hola'  
b = a
```

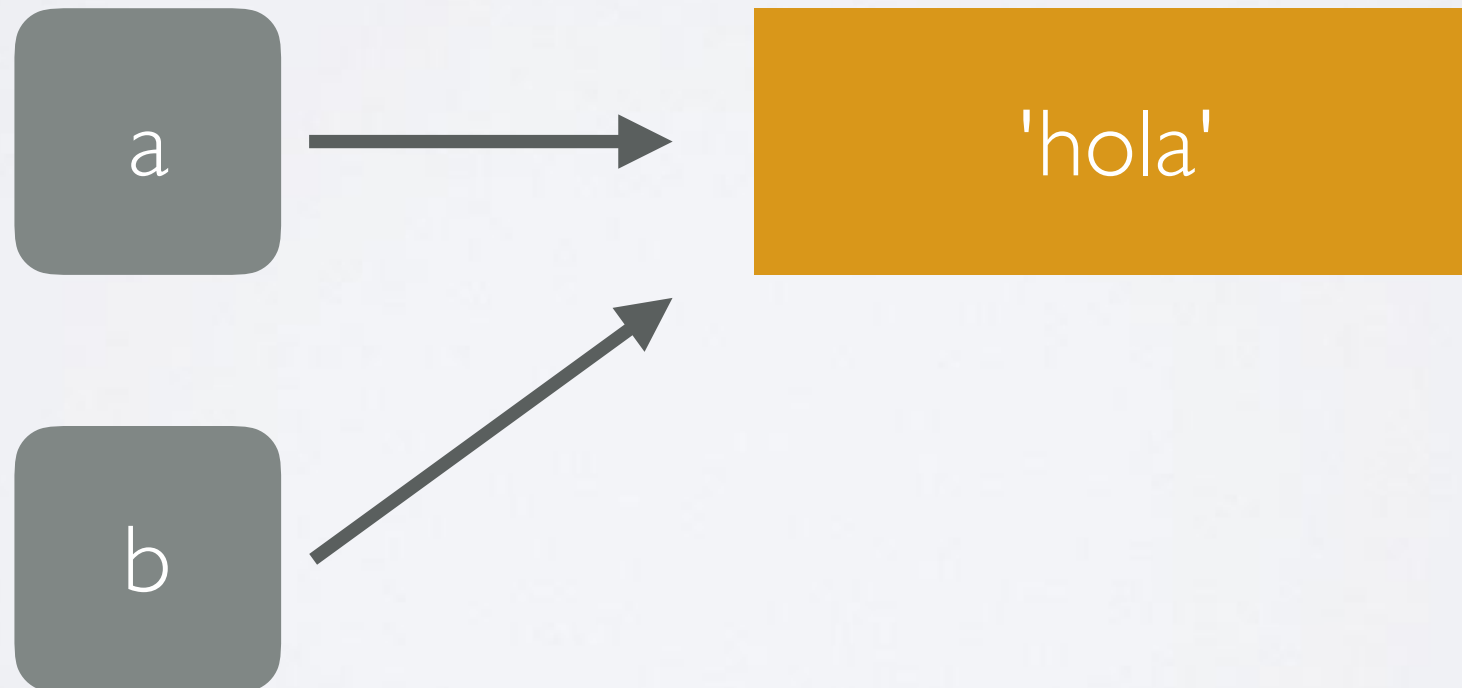
VEAMOS UN EJEMPLO CON DOS VARIABLES



```
a = 'hola'  
b = a  
a.upper()
```

NO CONFUNDIRSE

```
a = 'hola'  
b = a
```



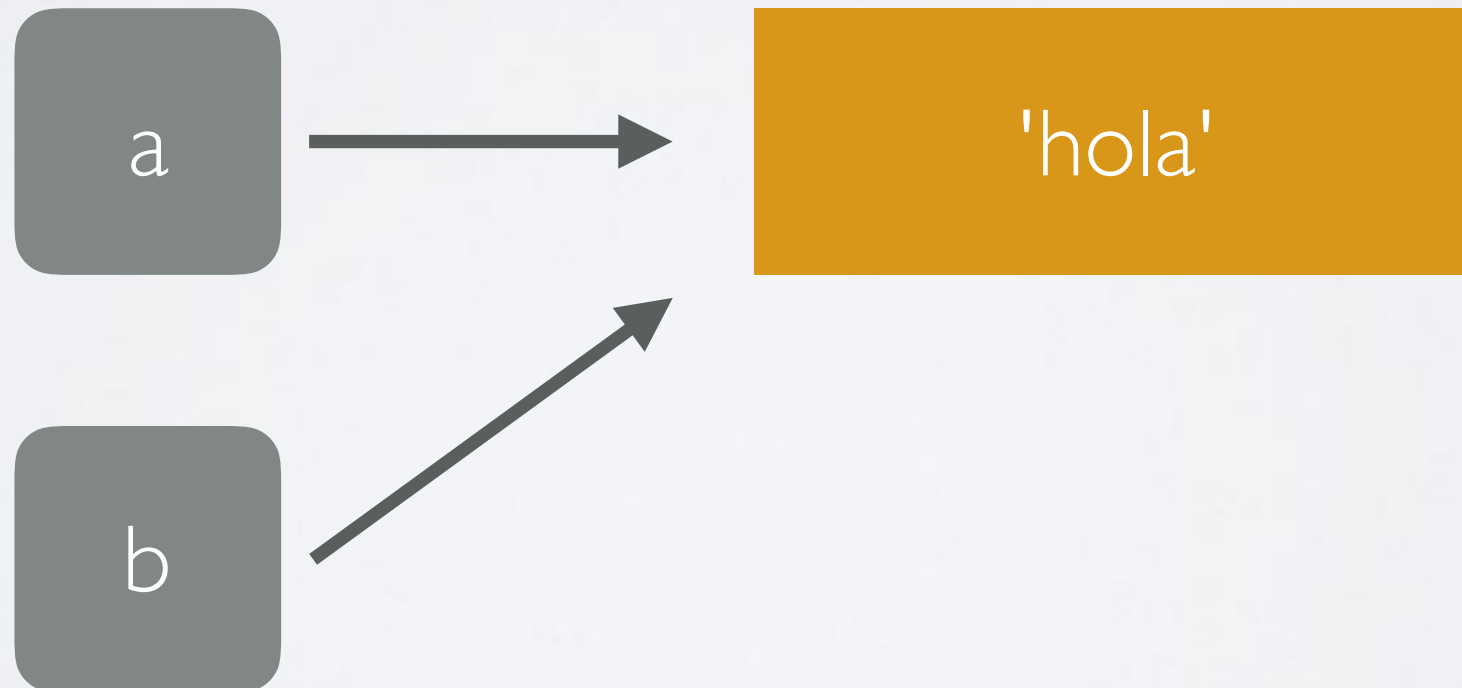
NO CONFUNDIRSE

```
a = 'hola'  
b = a  
b = 'otra cosa'
```



Y QUÉ SUCEDE CON +=

```
a = 'hola'  
b = a
```



Y QUÉ SUCEDE CON +=

```
a = 'hola'  
b = a  
b = b + 'otro string'
```



APRENDIMOS DOS LECCIONES MUY IMPORTANTES

- 1) Cambiar el estado no es lo mismo que asignar un objeto nuevo
- 2) Cambiar el estado de un objeto referenciado por más de una variable es peligroso

¿CÓMO PODEMOS SABER SI DOS VARIABLES REFERENCIAN AL MISMO OBJETO?

con `.object_id`

```
a = 2  
puts a.object_id # 5  
  
a = 3  
puts a.object_id # 7
```

SI DOS VARIABLES TIENEN EL MISMO OBJECT_ID ES PELIGROSO HACER UN CAMBIO

Caso Seguro

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

Caso Inseguro

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

OBJECT_ID Y LOS SÍMBOLOS

Caso Seguro

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

Caso Seguro

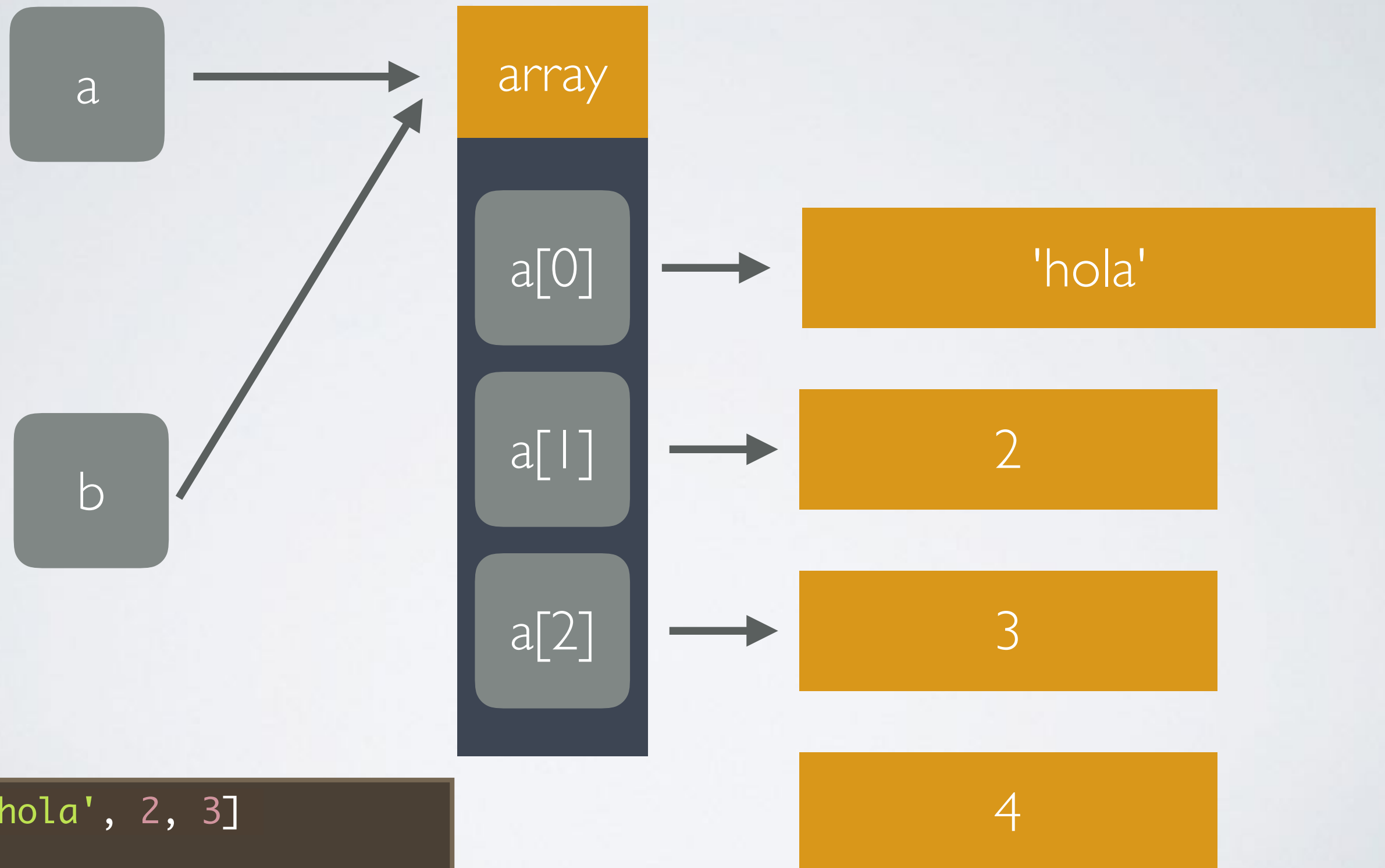
```
a = :hola
b = a
puts a.object_id == b.object_id
# true
```

El mismo object_id en los símbolos no es un problema porque son inmutable, o sea no los podemos modificar solo podemos asignar un símbolo nuevo

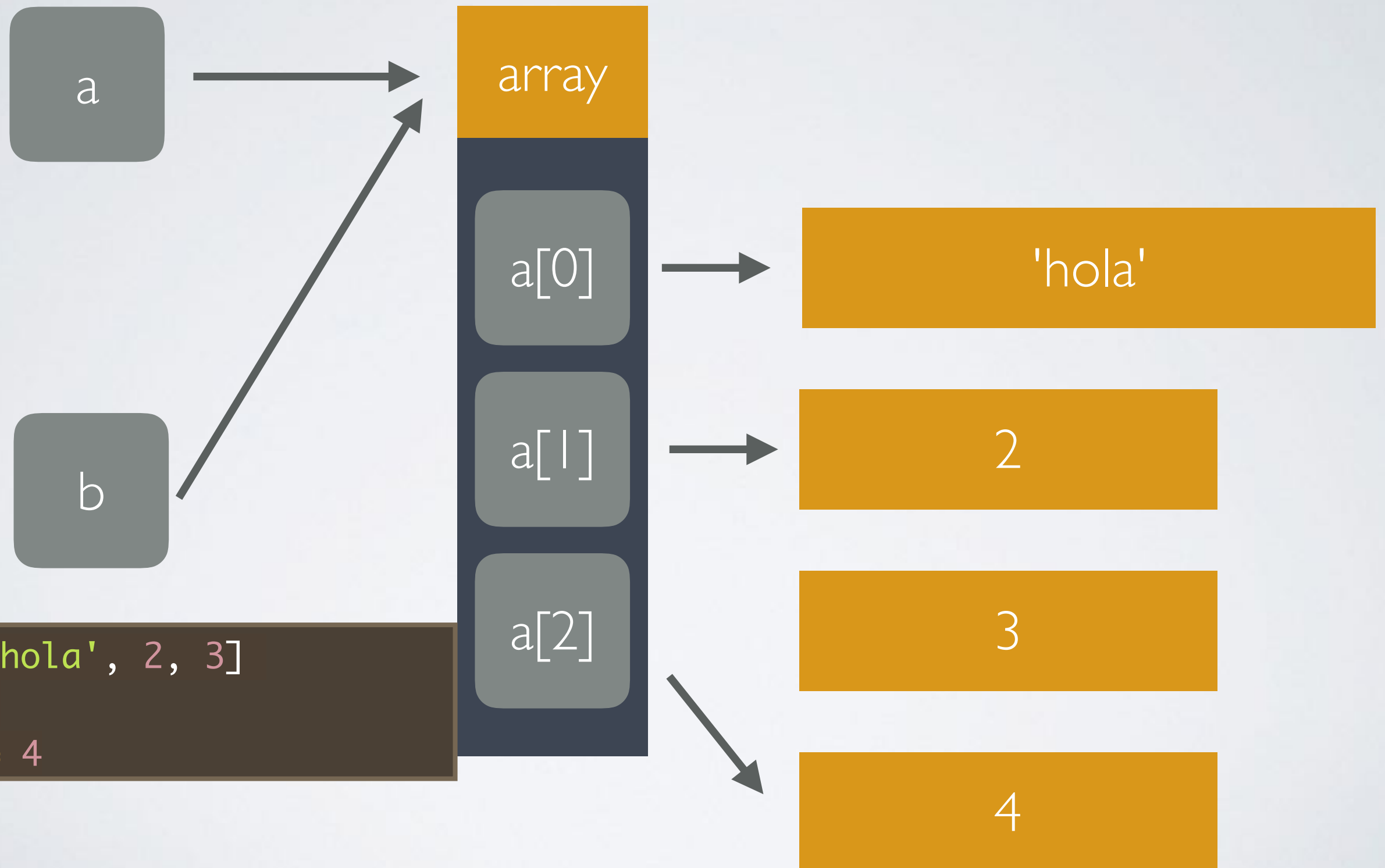
EN ESTE ASPECTO LOS ARRAYS SON COMO LOS STRINGS

```
a = [9,4,6,2.0,"hola"]  
b = a  
# => a.object_id == b.object_id # true  
  
puts b  
# => [9,4,6,2.0,"hola"]  
  
b[0] = 8  
# cambian a y b => [8,4,6,2.0,"hola"]  
  
puts a[0]  
# => 8
```

ARRAYS



ARRAYS



LOS HASH TAMBIÉN SON MUTABLES

```
datos = {"usuario": "gonzalo", "password": "secreto"}  
otros_datos = datos  
  
otros_datos["usuario"] = "nuevo_password"  
# => Agrego nuevo valor a la clave "usuario".  
  
datos["usuario"]  
# => nuevo_password
```

ES MUY FÁCIL EQUIVOCARSE Y CAMBIARLOS DENTRO DE UN MÉTODO POR ERROR

```
def metodo1(arreglo)
  arreglo[0] = 8
end

arr = [1, 2, 3, 4, 5]
metodo1(arr)
print arr #[8, 2, 3, 4, 5]
```

SI EN ALGÚN MOMENTO NECESITAMOS
HACER UNA COPIA PARA EVITAR PROBLEMAS
LO PODEMOS HACER CON CLONE

```
arr1 = [1, 2, 3]  
arr2 = arr1.clone  
  
arr1[4] = 'cambio'  
  
print arr2 # [1, 2, 3]
```

MORALEJA

SI DOS VARIABLES TIENEN EL MISMO OBJECT_ID ES
PELIGROSO HACER UN CAMBIO

PARA APRENDER MAS SOBRE ARRAYS
Y HASHES NECESITAMOS
PROFUNDIZAR EN BLOQUES Y PROCS

UN BLOQUE ES UN CONJUNTO DE INSTRUCCIONES

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

# bloque con brackets
a.each {|x| x + 1}

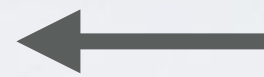
# bloque con do
a.each do |x|
  x + 1
end
```

Los bloques se declaran entre brackets, **{ }** o, utilizando **do** y **end**.

SON ÚTILES PARA HACER
FLEXIBLE LOS MÉTODOS

MÉTODOS QUE RECIBEN BLOQUES HAY MUCHOS

`.each (iterar)`



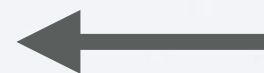
bloque con especificaciones
de que hacer en cada
iteración

`.sort_by (ordenar)`



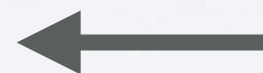
bloque con criterio de
ordenamiento

`.group_by (agrupar)`



bloque con criterio de
agrupamiento

`.reject (filtrar)`



bloque con criterio
para filtrar

VEAMOS UNO QUE YA CONOCEMOS

.each

```
arr1 = ['Arroz', 'Leche', 'Café']  
arr1.each do |item|  
  puts "Debo comprar #{item}"  
end
```

TAMBIÉN PODEMOS OCUPAR LAS LLAVES

```
arr1 = ['Arroz', 'Leche', 'Café']  
arr1.each { |item| puts "Debo comprar #{item}" }
```

MAP Y COLLECT

Forma normal

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

Forma express

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

.map devuelve un array con el resultado de aplicar la operación especificada a cada elemento

* collect y map son sinónimos

SELECT || REJECT

select

```
a = [1,2,3,4,5,6,7]
b = a.select{ |x| x % 2 == 0 }
# seleccionamos todos los
pares
# => [2,4,6]
```

reject

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

select devuelve un array con todos los elementos que cumplen la condición **reject** devuelve un array con los elementos que no la cumplen

* reject y select hacen lo contrario

EJERCICIO

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

1. Iterar por el arreglo y mostrar la cantidad de caracteres de cada uno de los nombres
2. Utilizando un map para generar un array con la cantidad de caracteres de cada nombre
3. Utilizando select generar un nuevo array con todos los nombres que tengan más de 5 letras

INJECT

inject

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

Valor inicial

acumulador

iterador

GROUP_BY

Podemos agrupar por cualquier criterio que queramos

```
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

EJERCICIOS

- Dado el arreglo $a = [1,2,3,4,5]$, cree otro arreglo que contenga los elementos pares de a , utilizando los métodos `.for`, `.each` y `.map`. (por separado)
- Dado el arreglo $c = [1,12,3,45, 21]$, cree otro arreglo que contenga todos los elementos de c que sean menores a 15, utilizando los métodos `each` y `.map` (por separado).
- Haga un arreglo que contenga el nombre de sus compañeros como elementos, luego seleccione en otro arreglo todos los nombres que comienzan con la letra P(o elija) y en otro arreglo que queden todos los nombres que NO comienzan con la letra P.