Ruby

Strings
Colecciones
Enumerables
Excepciones

Strings: sustitución

```
x, y, z = 12, 36, 72

puts "The value of x is #{ x }."

puts "The sum of x and y is #{ x + y }."

puts "The average was #{ (x + y + z)/3 }."
```

Strings

```
name = 'Bond\nJames Bond'

puts "Hello #{name}"

name = "Bond\nJames Bond"

puts "Hello #{name}"
```

Arrays

- Colecciones ordenadas en base a la posición
- Cada elemento se accede en base a su posición
- El primer elemento está en la posición cero
- Un índice negativo indica que es relativo al final del array

```
a = [1,2,3,4]
v = Array.new(10)
puts v.length   # => 10
puts v.size     # => 10 (en Array es un alias de length)
puts v.count    # => 10 (no usarlo sin parámetros)
puts v.count(1) # => 0
puts v          # => 10 líneas en blanco
p v             # => [nil, nil, nil, nil, nil, nil, nil, nil, nil, nil]
```

Arrays (continuación)

```
v[0] = 1
v[1] = 2
v[10] = 10
puts v.count(1)
puts v.count{|e| e>1}
p v
puts v.count{|e| e.nil?}
puts v.count{|e| ! e.nil? && e > 1}
puts v.length
v[-2] = 9
p v
w = v.compact
v.select{|e| ! e.nil? && e > 1}
```

Arrays

```
names = Array.new(3, 'Juan')  # => ["Juan", "Juan", "Juan"]
names[2] = 'Pedro'
q = names.clone.reverse       # => q = ["Pedro", "Juan", "Juan"], names no cambia
names.delete(4)             # => no hace nada
r = names.delete('Juan')  # r = "Juan", names = ["Pedro"]
names[1] = 'Alf'
names.insert(-2, "Ana", "Mario")     # => names = ["Pedro", "Ana", "Mario", "Alf"]

# backup va a ser un alias de names ("apuntan" a lo mismo)
backup = names.insert(-1, "Fin")  # => names = ["Pedro", "Ana", "Mario", "Alf", "Fin"]

backup[5] = "Juan"                          # agrega en names
backup = names.clone
backup[6] = "Azul"                          # no agrega en names
```

Arrays

```
# pop: elimina el último o últimos elementos del Array
a = [ "a", "b", "c", "d" ]
a.pop     #=> "d"
a.pop(2)  #=> ["b", "c"]
a         #=> ["a"]
a.include?("a")   #=> true (usa == con cada elemento)

# push
a = [ "a", "b", "c" ]
a.push("d", "e", "f") #=> ["a", "b", "c", "d", "e", "f"]

# << es similar a push, pero aplica a un solo parámetro
a = [ "a", "b", "c" ]
a << "d" << "e" << "f"
```

Arrays

```
# vector de vectores
a = [[1,2,3], [4,5,6]]
p a.size                                    # => 2
p a                                         # [[1, 2, 3], [4, 5, 6]]
p a[0]                                      # [1, 2, 3]
p a[0][0]                                   # 1
a[2] = "String"
p a                                         # [[1, 2, 3], [4, 5, 6], "String"]

a[0] == a[1]                          # false
a[0] == a[4]                          # false
p a[1] == a[-2]                       # true

p a[-4]                                     # nil
```

Arrays

```
# Dado un vector, creamos una copia
# reemplazando cada nil por cero
```

```
# Calculamos la suma y el producto
# de los elementos de un vector
```

map, reduce

```
n = [1,nil,2,nil,3,nil,4,nil]
m = n.map {|e| e.nil? ? 0 : e}
p n   # [1, nil, 2, nil, 3, nil, 4, nil]
p m   # [1, 0, 2, 0, 3, 0, 4, 0]

sum = m.reduce(0, :+)
total = m.reduce{ |sum, n| sum + n }
mult = m.reduce{ |prod, n| prod * if n != 0 then n else 1 end}

longest = backup.inject do          # inject es alias de reduce
  |w, word| w.length > word.length ? w : word
end
longest = backup.reduce{|w, word| w.length > word.length ? w : word}

n.reduce{ |m,e| p e }
n.reduce{ |m,e| p m }
```

Enumerable

```
class Array
   include Enumerable
...
```

```
module Enumerable
     def to_a(...
          def sort(...                    # retorna un array ordenado
     def map(...
          def reduce(...
     def max(...
     def minmax(...
```

```
class Set
   include Enumerable
...
```

```
class Hash
   include Enumerable
...
```

Set

Colección no ordenada de valores no duplicados

```
require 'set'

s1 = Set[1, 2]                                  #=> #<Set: {1, 2}>
s2 = [1, 2].to_set                          #=> #<Set: {1, 2}>
s1 == s2                                        #=> true
s1.add("one")                                   #=> #<Set: {1, 2, "one"}>
s1.add("one")                               #=> #<Set: {1, 2, "one"}>
s1<<"two"                                       #=> #<Set: {1, 2, "one",
"two"}>
s1.merge([2, 6])                        #=> #<Set: {1, 2, "one", "two", 6}>
s1.subset?(s2)                                  #=> false
s2.subset?(s1)                              #=> true
Set[1, 3, 5] - Set[1, 5]                    #=> #<Set: {3}>
```

Comparable

Este módulo es utilizado por clases cuyos elementos pueden ordenarse

La clase debe definir el comportamiento del operador <=>

```
class Date
 include Comparable
 ...
 def <=>(other)
   c = year <=> other.year
   if c == 0
     c = month <=> other.month
   end
   if c == 0
     return day <=> other.day
   else
     return c
   end
 end
```

```
class Date
 include Comparable
 ...

 def <=>(other)
   year * 10000 + month * 100 + day
<=> other.year * 10000 +
other.month * 100 + other.day

 end
```

Comparable

```
b = [3, 2, 4, 5, 6, 1, 8, 3]
b_asc = b.sort                          # b_asc = [1, 2, 3, 3, 4, 5, 6, 8]
b_des = b.sort{|a,b| b <=> a}  # b_des = [8, 6, 5, 4, 3, 3, 2, 1]

set = Set.new(b)

c = set.sort                                        # c = [1, 2, 3, 4, 5, 6, 8]

d = [3, "2"].sort# in `sort': comparison of String with 3 failed
(ArgumentError)
```

Comparable

```
class Foo
 def initialize number
   @number = number
 end

 def to_s
   @number.to_s
 end
end

f1 = Foo.new 10
f2 = Foo.new 20
```

```
puts f1                    # 10
puts f2                    # 20

f1 <=> f2          # nil
[f1, f2].max               #
nil
p [f2, f1].sort
```

Hash

- Colección no ordenada de pares clave-valor
- Cada valor se “indexa” por una clave que es única
- Si se accede por una clave inexistente, retorna *nil*

```
months = Hash.new          # Un hash vacío
months[1]                  # => nil

months = Hash.new('month')   # "month" es el valor por defecto
months[1]                            # "month"
```

Hash

```
months = Hash[1 => 'Enero', 2 => 'Febrero']

months[3] = 'Marzo'
months[5] = 'Mayo'
months['Jan'] = 'Enero'
p months[1]       # => "Enero"
p months[2]       # => "Febrero"
p months[3]       # => "Marzo"
p months[4]       # => nil

k = months.keys   # k = [1, 2, 3, 5, "Jan"]
```

```
values = {:name => 'Juan', :last_name => 'García' }
p values[:name]             # => "Juan"
p values[:nombre]           # => nil
```

Hash: principales métodos

- clear
- default = obj
- delete(key)
- each { |key,value| block }
- each_key { |key| block }
- empty?
- key?
- value?

Hash: principales métodos

- index(value)
- keys
- length
- store(key, value)
- values

==, eql?, equal?

- *obj*.eql?(other) es usado por una tabla de hash para determinar equivalencia entre dos elementos.
- *obj*.hash es usado por Hash para indexar un elemento.

```
1 == 1.0      # => true
1.eql? 1.0    # => false

p months[1]           # => "Enero"
p months[1.0]         # => nil
```

Determinar la salida del siguiente programa

```ruby
class Point
 attr_accessor :x,:y
 def initialize(x,y)
   @x, @y = x, y
 end
 def ==(other)
   @x==other.x && @y==other.y
 end
 def hash
   [@x, @y].hash
 end
 def eql?(other)
   self==other
 end
end
```

```ruby
my_map = Hash.new
p1 = Point.new(0,0)
p2 = Point.new(1,1)
p3 = Point.new(2,2)
my_map.store(p1, '00')
my_map.store(p2, '11')
my_map.store(p3, '22')

puts my_map.key?(p1)
p1.x=2
puts my_map.key?(p1)
```

Determinar la salida del siguiente programa

```ruby
class Point
 ...
 def hash
   @x + @y
 end
end

my_map = Hash.new
p1 = Point.new(0,2)
p2 = Point.new(1,1)
p3 = Point.new(2,0)
```

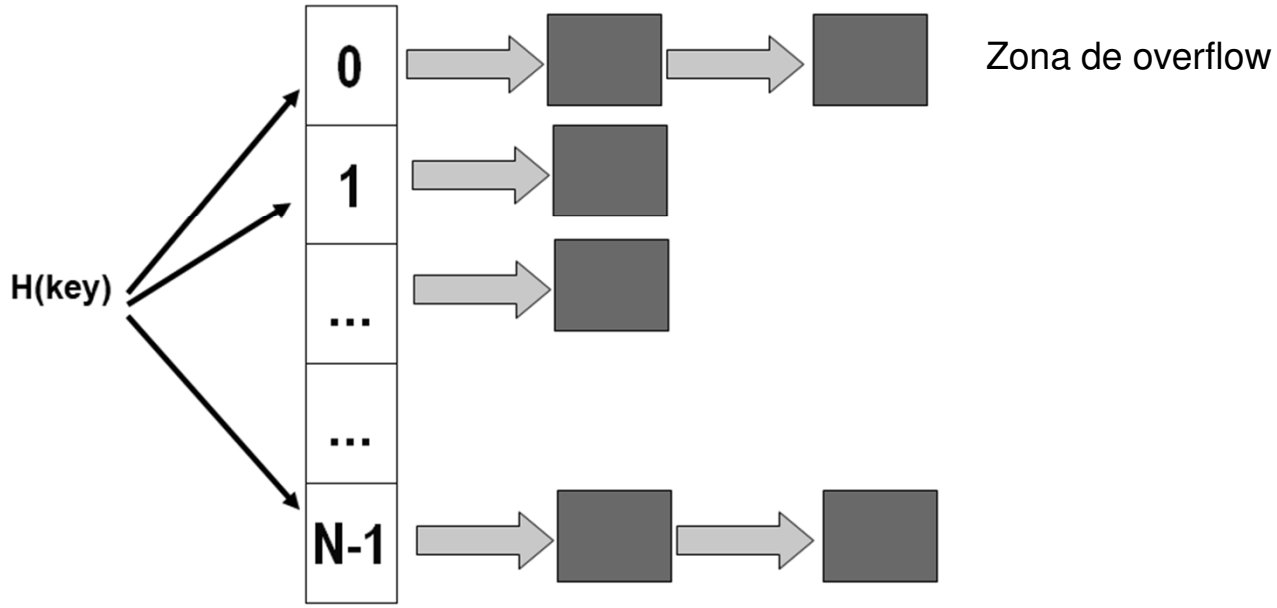
```ruby
my_map.store(p1, '02')
my_map.store(p2, '11')
my_map.store(p3, '20')

puts my_map[p1]
puts my_map[p2]
puts my_map[p3]

my_map.store(Point.new(1.0,1.0), "1.0, 1.0")

puts my_map[p2]
```

Hashing

- Estrategia para almacenamiento y recupero de información asociada a una clave única
- En casos ideales todas las operaciones son de O(1)
- En el peor caso es de O(N)
- Los elementos son pares (clave, valor) y se almacenan en un arreglo de N posiciones
- A cada clave se le aplica una función matemática (hash) que genera un valor numérico. El índice del arreglo será H(key) = key.hash %N
- Si dos claves “hashean” en la misma posición se produce una colisión

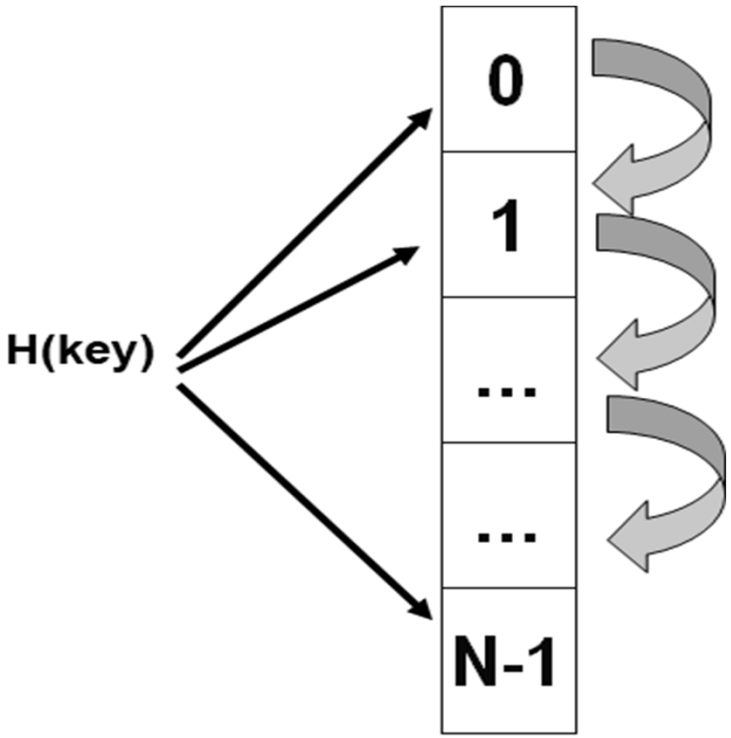
Resolución de colisiones con hashing abierto

- **H(Key) => i i en [0, N-1]**



Hashing cerrado: *open addressing*

Si el lugar está ocupado, se aplica una función H', H'', etc hasta obtener una posición libre



Date y Time

Puede no funcionar para manejar fechas anteriores a 1970 o posteriores a 2038

```
time = Time.new    # tambien Time.now
puts 'Fecha y hora actual : ' + time.inspect    # alias de to_s
puts time.year     # => Año
puts time.month    # => 1..12
puts time.day      # => 1..31
puts time.wday     # => 0..6 ( 0 es domingo)
puts time.yday     # => 1..365
puts time.hour     # => 0..23
puts time.min      # => minutos
puts time.sec      # => segundos
puts time.usec     # => microseconds
puts time.zone     # => Argentina Standard Time (por ejemplo)
```

Investigar constructor y otros métodos

Date y Time

```
require 'date'

t1 = Date.new(2018, 3, 15)
t2 = DateTime.new(2018, 3, 15)
puts t1      # => 2018-03-15
puts t2      # => 2018-03-15T00:00:00+00:00
```

Rangos

- Secuencias
- Condiciones
- Intervalos

Rangos: secuencias

```
(1..5)          #==> 1, 2, 3, 4, 5
(1...5)         #==> 1, 2, 3, 4
('a'..'d')      #==> 'a', 'b', 'c', 'd'

range = ('bar'..'bau').to_a

puts "#{range}"     # => ["bar", "bas", "bat", "bau"]
```

Rangos: secuencias

```
range1 = (1..10).to_a
range2 = ('bar'..'bau').to_a

puts "#{range1}"    # => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
                    # puts range1 imprime cada elemento en una línea
puts "#{range2}"    # => ["bar", "bas", "bat", "bau"]

puts (1..10).include? 5       # => true
puts (1...10).include? 10     # => false

r = (1..10).reject { |i| i.even? }      # r = [1, 3, 5, 7, 9]
r = range1.reject { |i| i.even? }       # r = [1, 3, 5, 7, 9]

# imprime del 1 al 10
(1..10).each do |i|
 puts i
end
```

Rangos: secuencias

```
p ('ba'..'baz').size                  # => nil
p ('ba'..'baz').to_a.size             # => 1352

# imprime los impares del 1 al 9
(1..10).step(2).each do |i|
 puts i
end

(2..10).step(0) { |v| puts v }
                    # step can't be 0 (ArgumentError)
```

Rangos: condiciones

```
result = case score
          when 0..40 then
            'Fail'
          when 41..60 then
            'Pass'
          when 61..70 then
            'Pass with Merit'
          when 71..100 then
            'Pass with Distinction'
          else
            'Invalid Score'
        end

puts result
```

Rangos: intervalos

```
if (4..10).include? x
 # xxx
end


if (4..10) === x
 # xxx
end

xxx  if x.between?(4,10)
```

Iteradores

- Son métodos soportados por las colecciones (array, hash, set)
- Los iteradores retornan los elementos de una colección, uno después de otro
- Los más usados son each y map (equivalente a collect)

```
collection.each do |variable|
   code
end
```

Iteradores

collect y map son lo mismo. Es de mejor estilo usar map que collect

```
class Person
 attr_accessor :name, :surname
 def initialize(name,surname)
   @name, @surname = name, surname
 end
end

bonds = []
bonds << Person.new("Daniel","Craig")
bonds << Person.new("Pierce","Brosnan")
bonds << Person.new("Roger","Moore")
bonds << Person.new("Sean","Connery")

p bonds.map { |p| p.surname }
```

Iteradores

Un Enumerator puede ser usado como un iterador externo. El método next retorna el siguiente valor o lanza la excepción StopIteration si no hay más elementos.

```
a = arr.each
p a.class      # Enumerator
p a            # #<Enumerator: [1, 2, 3]:each>
p a.next       # 1
p a.next       # 2
p a.next       # 3
p a.next       # => iteration reached an end (StopIteration)

a = arr.each
a.take(2)      # take retorna un array con los primeros n elementos
a.next         # 1
```

Iteradores

Es posible crear iteradores sobre nuestras clases. Para ello debemos definir un método que nos defina la sucesión

```
seq = Enumerator.new do |y|    # y es un "yielder" object
 a = 1
 loop do
   y << a       # << es un alias del método "yield"
   a += 1
 end
end

p seq.take(10)    # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Iteradores

```
give_me_a_name = Enumerator.new do |y|    # y es un "yielder" object
 a = b = 1
 loop do
   y << a       # << es un alias del método "yield"
   a, b = b, a + b
 end
end

p give_me_a_name.take(10)
```

Ejemplo: LinkedList

```
class Node
 attr_accessor :val, :next

 def initialize(val, next_node)
   @val = val
   @next = next_node
 end
end

class LinkedList
 @header

 ...
```

Ejemplo: LinkedList

```
 ...

 def iterator
   e = Enumerator.new do |y|
     c = @head
     loop do
       y << c.val
       c = c.next
     end
   end
   e
 end
end
```

Excepciones

- Una excepción es un objeto especial, instancia de alguna clase que extienda Exception
- Su existencia indica que algo salió mal
- Un programa finaliza si se produce una excepción no manejada
- Ruby permite manejar (capturar) las excepciones
- Un “exception handler” es un bloque de código que se ejecuta si ocurre una excepción

Excepciones

rescue permite capturar el flujo de ejecución cuando se lanza una excepción

```
begin
  c = a / b
rescue ZeroDivisionError
    puts 'División por cero'
rescue NoMethodError
        puts 'No son números'
end
```

Excepciones: raise

1) raise sin parámetros relanza la excepción actual, o RuntimeError si no había excepción actual

```
begin
 c = a / 0
rescue
 # guardamos el error en un log, pero dejamos que otro lo maneje
 ...
 raise   # => Lanza ZeroDivisionError
end
```

Excepciones: raise

2) Lanzar RuntimeException con un mensaje de error

```
begin
 raise 'Testing...'
rescue Exception => e
 puts e.message     # => Testing...
 puts e.class       # => RuntimeError
end
```

Excepciones: raise

3) Lanzar una instancia específica, con un mensaje de error

```
def foo(arg)
 raise ArgumentError, 'arg can't be nil' if arg.nil?
 ...
end

foo nil    # => in `foo': arg can't be nil (ArgumentError)
```

Excepciones: ejemplo

```
class LinkedList

def iterator
 e = Enumerator.new do |y|
   c = @head
   loop do
     raise StopIteration if c.nil?
     y << c.val
     c = c.next
   end
 end
 e
end
```

```
l = LinkedList.new

it = l.iterator
begin
 it.next
rescue StopIteration
 ...
end
```

Excepciones: crear propias

Sólo se debe extender Exception o alguna clase que extienda Exception

```
class MyException < StandardError

end
```