EBook Gratis

APRENDIZAJE Ruby Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#ruby

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: ruby-language

It is an unofficial and free Ruby Language ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Ruby Language.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con Ruby Language

Observaciones

Ruby es un lenguaje multiplataforma de código abierto, orientado a objetos dinámico, diseñado para ser simplista y productivo. Fue creado por Yukihiro Matsumoto (Matz) en 1995.

> Según su creador, Ruby fue influenciado por Perl , Smalltalk , Eiffel , Ada y Lisp . Admite múltiples paradigmas de programación, incluyendo funcional, orientado a objetos e imperativo. También cuenta con un sistema de tipo dinámico y gestión automática de memoria.

Versiones

Versión	Fecha de lanzamiento
2.4	2016-12-25
2.3	2015-12-25
2.2	2014-12-25
2.1	2013-12-25
2.0	2013-02-24
1.9	2007-12-25
1.8	2003-08-04
1.6.8	2002-12-24

Examples

Hola mundo de IRB

Alternativamente, puede usar el Ruby Interactive Shell (IRB) para ejecutar inmediatamente las declaraciones de Ruby que escribió anteriormente en el archivo de Ruby.

Comience una sesión de IRB escribiendo:

```
$ irb
```

Luego ingrese el siguiente comando:

```
puts "Hello World"
```

Esto da como resultado la siguiente salida de consola (incluida la nueva línea):

```
Hello World
```

Si no desea iniciar una nueva línea, puede usar `print` :

```
print "Hello World"
```

Hola mundo con tk

Tk es la interfaz gráfica de usuario (GUI) estándar para Ruby. Proporciona una interfaz gráfica de usuario multiplataforma para los programas de Ruby.

Código de ejemplo:

```
require "tk"
TkRoot.new{ title "Hello World!" }
Tk.mainloop
```

El resultado:

Explicación paso a paso:

```
require "tk"
```

Cargue el paquete tk.

```
TkRoot.new{ title "Hello World!" }
```

Define un widget con el título `Hello World`

```
Tk.mainloop
```

Iniciar el bucle principal y mostrar el widget.

Hola Mundo

> Este ejemplo asume que Ruby está instalado.

Coloque lo siguiente en un archivo llamado `hello.rb` :

```
puts 'Hello World'
```

Desde la línea de comandos, escriba el siguiente comando para ejecutar el código Ruby desde el archivo fuente:

```
$ ruby hello.rb
```

Esto debería dar salida:

```
Hello World
```

La salida se mostrará inmediatamente a la consola. Los archivos fuente de Ruby no necesitan ser compilados antes de ser ejecutados. El intérprete de Ruby compila y ejecuta el archivo de Ruby en tiempo de ejecución.

Hola mundo sin archivos fuente

Ejecute el siguiente comando en un shell después de instalar Ruby. Esto muestra cómo puede ejecutar programas Ruby simples sin crear un archivo Ruby:

```
ruby -e 'puts "Hello World"'
```

También puede alimentar un programa Ruby a la entrada estándar del intérprete. Una forma de hacerlo es usar un documento aquí en su comando de shell:

```
ruby <<END
puts "Hello World"
END
```

Hello World como archivo autoejecutable: usando Shebang (solo sistemas operativos similares a Unix)

Puede agregar una directiva de intérprete (shebang) a su script. Cree un archivo llamado `hello_world.rb` que contenga:

```
#!/usr/bin/env ruby

puts 'Hello World!'
```

Dar los permisos ejecutables de script. Aquí es cómo hacer eso en Unix:

```
$ chmod u+x hello_world.rb
```

Ahora no necesita llamar explícitamente al intérprete de Ruby para ejecutar su script.

```
$ ./hello_world.rb
```

Mi primer metodo

Visión general

Crea un nuevo archivo llamado `my_first_method.rb`

Coloque el siguiente código dentro del archivo:

```
def hello_world
  puts "Hello world!"
end

hello_world() # or just 'hello_world' (without parenthesis)
```

Ahora, desde una línea de comando, ejecuta lo siguiente:

```
ruby my_first_method.rb
```

La salida debe ser:

```
Hello world!
```

Explicación

- `def` es una palabra clave que nos dice que estamos `def` -ining un método - en este caso, `hello_world` es el nombre de nuestro método.
- `puts "Hello world!"` `puts` (o canaliza a la consola) la cadena `Hello world!`
- `end` es una palabra clave que significa que estamos terminando nuestra definición del método `hello_world`
- Como el método `hello_world` no acepta ningún argumento, puede omitir el paréntesis invocando el método.

Lea Empezando con Ruby Language en línea:
https://riptutorial.com/es/ruby/topic/195/empezando-con-ruby-language

Capítulo 2: Alcance variable y visibilidad

Sintaxis

- $ global_variable
- @@ class_variable
- @Instancia variable
- variable local

Observaciones

Las variables de clase se comparten en la jerarquía de clases. Esto puede resultar en un comportamiento sorprendente.

```
class A
  @@variable = :x

  def self.variable
    @@variable
  end
end

class B < A
  @@variable = :y
end

A.variable  # :y
```

Las clases son objetos, por lo que las variables de instancia se pueden usar para proporcionar un estado que sea específico para cada clase.

```
class A
  @variable = :x

  def self.variable
    @variable
  end
end

class B < A
  @variable = :y
end

A.variable  # :x
```

Examples

Variables locales

Las variables locales (a diferencia de las otras clases de variables) no tienen ningún prefijo

```
local_variable = "local"
p local_variable
# => local
```

Su alcance depende de donde se ha declarado, no se puede usar fuera del alcance de "contenedores de declaración". Por ejemplo, si una variable local se declara en un método, solo se puede usar dentro de ese método.

```
def some_method
    method_scope_var = "hi there"
    p method_scope_var
end

some_method
# hi there
# => hi there

method_scope_var
# NameError: undefined local variable or method `method_scope_var'
```

Por supuesto, las variables locales no se limitan a los métodos, como regla de oro podría decir que, tan pronto como declare una variable dentro de un bloque `do ... end` o envuelto entre llaves `{}` , será local y estará dentro del alcance de El bloque ha sido declarado en.

```
2.times do |n|
    local_var = n + 1
    p local_var
end
# 1
# 2
# => 2

local_var
# NameError: undefined local variable or method `local_var'
```

Sin embargo, las variables locales declaradas en `if` o los bloques de `case` se pueden usar en el ámbito principal:

```
if true
    usable = "yay"
end

p usable
# yay
# => "yay"
```

Si bien las variables locales no pueden utilizarse fuera de su bloque de declaración, se transmitirán a los bloques:

```
my_variable = "foo"
```

```
my_variable.split("").each_with_index do |char, i|
    puts "The character in string '#{my_variable}' at index #{i} is #{char}"
end
# The character in string 'foo' at index 0 is f
# The character in string 'foo' at index 1 is o
# The character in string 'foo' at index 2 is o
# => ["f", "o", "o"]
```

Pero no a las definiciones de método / clase / módulo

```
my_variable = "foo"

def some_method
    puts "you can't use the local variable in here, see? #{my_variable}"
end

some_method
# NameError: undefined local variable or method `my_variable'
```

Las variables utilizadas para los argumentos de bloque son (por supuesto) locales al bloque, pero eclipsarán las variables previamente definidas, sin sobrescribirlas.

```
overshadowed = "sunlight"

["darkness"].each do |overshadowed|
    p overshadowed
end
# darkness
# => ["darkness"]

p overshadowed
# "sunlight"
# => "sunlight"
```

Variables de clase

Las variables de clase tienen un amplio alcance de clase, se pueden declarar en cualquier parte de la clase. Una variable se considerará una variable de clase cuando se prefija con `@@`

```
class Dinosaur
    @@classification = "Like a Reptile, but like a bird"

    def self.classification
        @@classification
    end

    def classification
        @@classification
    end
end

dino = Dinosaur.new
dino.classification
# => "Like a Reptile, but like a bird"

Dinosaur.classification
```

```
# => "Like a Reptile, but like a bird"
```

Las variables de clase se comparten entre clases relacionadas y se pueden sobrescribir de una clase secundaria

```
class TRex < Dinosaur
    @@classification = "Big teeth bird!"
end

TRex.classification
# => "Big teeth bird!"

Dinosaur.classification
# => "Big teeth bird!"
```

Este comportamiento no es deseado la mayor parte del tiempo y se puede sortear mediante el uso de variables de instancia de nivel de clase.

Las variables de clase definidas dentro de un módulo no sobrescribirán sus variables de clase incluidas las clases:

```
module SomethingStrange
    @@classification = "Something Strange"
end

class DuckDinosaur < Dinosaur
    include SomethingStrange
end

DuckDinosaur.class_variables
# => [:@@classification]
SomethingStrange.class_variables
# => [:@@classification]

DuckDinosaur.classification
# => "Big teeth bird!"
```

Variables globales

Las variables globales tienen un alcance global y, por lo tanto, se pueden utilizar en todas partes. Su alcance no depende de donde se definan. Una variable se considerará global, cuando se prefija con un signo `$` .

```
$i_am_global = "omg"

class Dinosaur
    def instance_method
       p "global vars can be used everywhere. See? #{$i_am_global}, #{$another_global_var}"
    end

    def self.class_method
       $another_global_var = "srsly?"
       p "global vars can be used everywhere. See? #{$i_am_global}"
    end
```

```
end

Dinosaur.class_method
# "global vars can be used everywhere. See? omg"
# => "global vars can be used everywhere. See? omg"

dinosaur = Dinosaur.new
dinosaur.instance_method
# "global vars can be used everywhere. See? omg, srsly?"
# => "global vars can be used everywhere. See? omg, srsly?"
```

Dado que una variable global se puede definir en todas partes y será visible en todas partes, llamar a una variable global "indefinida" devolverá nil en lugar de generar un error.

```
p $undefined_var
# nil
# => nil
```

Si bien las variables globales son fáciles de usar, su uso está fuertemente desaconsejado a favor de las constantes.

Variables de instancia

Las variables de instancia tienen un amplio alcance de objeto, se pueden declarar en cualquier parte del objeto, sin embargo, una variable de instancia declarada en el nivel de clase, solo será visible en el objeto de clase. Una variable se considerará una variable de instancia cuando se prefija con `@` . Las variables de instancia se utilizan para establecer y obtener atributos de un objeto y devolverán cero si no se definen.

```
class Dinosaur
    @base_sound = "rawrr"

    def initialize(sound = nil)
        @sound = sound || self.class.base_sound
    end

    def speak
        @sound
    end

    def try_to_speak
        @base_sound
    end

    def count_and_store_sound_length
        @sound.chars.each_with_index do |char, i|
            @sound_length = i + 1
            p "#{char}: #{sound_length}"
        end
    end

    def sound_length
        @sound_length
    end
```

```
    def self.base_sound
        @base_sound
    end
end

dino_1 = Dinosaur.new
dino_2 = Dinosaur.new "grrr"

Dinosaur.base_sound
# => "rawrr"
dino_2.speak
# => "grrr"
```

No se puede acceder a la variable de instancia declarada en el nivel de clase en el nivel de objeto:

```
dino_1.try_to_speak
# => nil
```

Sin embargo, usamos la variable de instancia `@base_sound` para crear una instancia del sonido cuando no se pasa ningún sonido al nuevo método:

```
dino_1.speak
# => "rawwr"
```

Las variables de instancia se pueden declarar en cualquier parte del objeto, incluso dentro de un bloque:

```
dino_1.count_and_store_sound_length
# "r: 1"
# "a: 2"
# "w: 3"
# "r: 4"
# "r: 5"
# => ["r", "a", "w", "r", "r"]

dino_1.sound_length
# => 5
```

Las variables de instancia **no se** comparten entre instancias de la misma clase

```
dino_2.sound_length
# => nil
```

Esto se puede usar para crear variables de nivel de clase, que no serán sobrescritas por una clase hija, ya que las clases también son objetos en Ruby.

```
class DuckDuckDinosaur < Dinosaur
    @base_sound = "quack quack"
end

duck_dino = DuckDuckDinosaur.new
duck_dino.speak
```

```
# => "quack quack"
DuckDuckDinosaur.base_sound
# => "quack quack"
Dinosaur.base_sound
# => "rawrr"
```

Lea Alcance variable y visibilidad en línea: https://riptutorial.com/es/ruby/topic/4094/alcance-variable-y-visibilidad

Capítulo 3: Aplicaciones de línea de comandos

Examples

Cómo escribir una herramienta de línea de comandos para obtener el clima por código postal

Este será un tutorial relativamente completo sobre cómo escribir una herramienta de línea de comandos para imprimir el clima desde el código postal provisto a la herramienta de línea de comandos. El primer paso es escribir el programa en ruby para hacer esta acción. Comencemos escribiendo un método `weather(zip_code)` (este método requiere la gema `yahoo_weatherman` . Si no tiene esta gema, puede instalarla escribiendo `gem install yahoo_weatherman` desde la línea de comandos)

```
require 'yahoo_weatherman'

def weather(zip_code)
  client = Weatherman::Client.new
  client.lookup_by_location(zip_code).condition['temp']
end
```

Ahora tenemos un método muy básico que da el clima cuando se le proporciona un código postal. Ahora necesitamos convertir esto en una herramienta de línea de comandos. Rápidamente repasemos cómo se llama una herramienta de línea de comandos desde el shell y las variables asociadas. Cuando se llama a una herramienta como esta, la `tool argument other_argument` , en ruby hay una variable `ARGV` que es una matriz igual a `['argument', 'other_argument']` . Ahora vamos a implementar esto en nuestra aplicación.

```
#!/usr/bin/ruby
require 'yahoo_weatherman'

def weather(zip_code)
  client = Weatherman::Client.new
  client.lookup_by_location(zip_code).condition['temp']
end

puts weather(ARGV[0])
```

¡Bueno! Ahora tenemos una aplicación de línea de comandos que se puede ejecutar. Observe la línea de *she-bang* al principio del archivo (`#!/usr/bin/ruby`). Esto permite que el archivo se convierta en un ejecutable. Podemos guardar este archivo como `weather` . (**Nota** : no guarde esto como `weather.rb` , no hay necesidad de la extensión de archivo y she-bang le dice a cualquier cosa que necesite que le diga que este es un archivo ruby). Ahora podemos ejecutar estos comandos en el shell (no escriba los `$`).

```
$ chmod a+x weather
$ ./weather [ZIPCODE]
```

Después de probar que esto funciona, ahora podemos vincularlo con el `/usr/bin/local/` ejecutando este comando

```
$ sudo ln -s weather /usr/local/bin/weather
```

Ahora se puede consultar el `weather` en la línea de comandos sin importar el directorio en el que se encuentre.

Lea Aplicaciones de línea de comandos en línea:
https://riptutorial.com/es/ruby/topic/7679/aplicaciones-de-linea-de-comandos

Capítulo 4: Argumentos de palabras clave

Observaciones

Los argumentos de palabras clave se introdujeron en Ruby 2.0 y se mejoraron en Ruby 2.1 con la adición de *los* argumentos de palabras clave *requeridos* .

Un método simple con un argumento de palabra clave se parece al siguiente:

```
def say(message: "Hello World")
  puts message
end

say
# => "Hello World"

say message: "Today is Monday"
# => "Today is Monday"
```

Como recordatorio, el mismo método sin argumento de palabra clave habría sido:

```
def say(message = "Hello World")
  puts message
end

say
# => "Hello World"

say "Today is Monday"
# => "Today is Monday"
```

2.0

Puede simular argumentos de palabras clave en versiones anteriores de Ruby usando un parámetro Hash. Esta sigue siendo una práctica muy común, especialmente en las bibliotecas que proporcionan compatibilidad con las versiones de Ruby anteriores a 2.0:

```
def say(options = {})
  message = options.fetch(:message, "Hello World")
  puts
end

say
# => "Hello World"

say message: "Today is Monday"
# => "Today is Monday"
```

Examples

Usando argumentos de palabras clave

Para definir un argumento de palabra clave en un método, especifique el nombre en la definición del método:

```
def say(message: "Hello World")
  puts message
end

say
# => "Hello World"

say message: "Today is Monday"
# => "Today is Monday"
```

Puede definir múltiples argumentos de palabras clave, el orden de definición es irrelevante:

```
def say(message: "Hello World", before: "<p>", after: "</p>")
  puts "#{before}#{message}#{after}"
end

say
# => "<p>Hello World</p>"

say message: "Today is Monday"
# => "<p>Today is Monday</p>"

say after: "</p><hr>", message: "Today is Monday"
# => "<p>Today is Monday</p><hr>"
```

Los argumentos de palabras clave se pueden mezclar con argumentos posicionales:

```
def say(message, before: "<p>", after: "</p>")
  puts "#{before}#{message}#{after}"
end

say "Hello World", before: "<span>", after: "</span>"
# => "<span>Hello World</span>"
```

Mezclar el argumento de palabras clave con un argumento posicional era un enfoque muy común antes de Ruby 2.1, porque no era posible definir los argumentos de palabras clave requeridos .

Además, en Ruby <2.0, era muy común agregar un `Hash` al final de una definición de método para usar para argumentos opcionales. La sintaxis es muy similar a los argumentos de palabras clave, hasta el punto en que los argumentos opcionales a través de `Hash` son compatibles con los argumentos de palabras clave de Ruby 2.

```
def say(message, options = {})
  before = option.fetch(:before, "<p>")
  after  = option.fetch(:after, "</p>")
  puts "#{before}#{message}#{after}"
end

# The method call is syntactically equivalent to the keyword argument one
```

```
say "Hello World", before: "<span>", after: "</span>"
# => "<span>Hello World</span>"
```

Tenga en cuenta que al intentar pasar un argumento de palabra clave no definido se generará un error:

```
def say(message: "Hello World")
  puts message
end

say foo: "Hello"
# => ArgumentError: unknown keyword: foo
```

Argumentos de palabras clave requeridos

2.1

Los argumentos de palabras clave requeridos se introdujeron en Ruby 2.1, como una mejora de los argumentos de palabras clave.

Para definir un argumento de palabra clave según sea necesario, simplemente declare el argumento sin un valor predeterminado.

```
def say(message:)
  puts message
end

say
# => ArgumentError: missing keyword: message

say message: "Hello World"
# => "Hello World"
```

También puede mezclar argumentos de palabras clave requeridos y no requeridos:

```
def say(before: "<p>", message:, after: "</p>")
  puts "#{before}#{message}#{after}"
end

say
# => ArgumentError: missing keyword: message

say message: "Hello World"
# => "<p>Hello World</p>"

say message: "Hello World", before: "<span>", after: "</span>"
# => "<span>Hello World</span>"
```

Usando argumentos arbitrarios de palabras clave con el operador splat

Puede definir un método para aceptar un número arbitrario de argumentos de palabras clave utilizando el operador de *doble splat* (`**`):

```
def say(**args)
  puts args
end

say foo: "1", bar: "2"
# {:foo=>"1", :bar=>"2"}
```

Los argumentos son capturados en un `Hash` . Puede manipular el `Hash` , por ejemplo, para extraer los argumentos deseados.

```
def say(**args)
  puts args[:message] || "Message not found"
end

say foo: "1", bar: "2", message: "Hello World"
# Hello World

say foo: "1", bar: "2"
# Message not found
```

El uso del operador splat con argumentos de palabras clave evitará la validación de los argumentos de palabras clave, el método nunca generará un `ArgumentError` en caso de palabras clave desconocidas.

En cuanto al operador splat estándar, puede volver a convertir un `Hash` en argumentos de palabras clave para un método:

```
def say(message: nil, before: "<p>", after: "</p>")
  puts "#{before}#{message}#{after}"
end

args = { message: "Hello World", after: "</p><hr>" }
say(**args)
# <p>Hello World</p><hr>

args = { message: "Hello World", foo: "1" }
say(**args)
# => ArgumentError: unknown keyword: foo
```

Generalmente se usa cuando necesita manipular los argumentos entrantes y pasarlos a un método subyacente:

```
def inner(foo:, bar:)
  puts foo, bar
end

def outer(something, foo: nil, bar: nil, baz: nil)
  puts something
  params = {}
  params[:foo] = foo || "Default foo"
  params[:bar] = bar || "Default bar"
  inner(**params)
end

outer "Hello:", foo: "Custom foo"
```

```
# Hello:
# Custom foo
# Default bar
```

Lea Argumentos de palabras clave en línea: https://riptutorial.com/es/ruby/topic/5253/argumentos-de-palabras-clave

Capítulo 5: Arrays

Sintaxis

- a = [] # usando literal de matriz
- a = Array.new # equivalente a usar literal
- a = Array.new (5) # crea una matriz con 5 elementos con valor nil.
- a = Array.new (5, 0) # crea una matriz con 5 elementos con un valor predeterminado de 0.

Examples

#mapa

`#map` , proporcionado por Enumerable, crea una matriz invocando un bloque en cada elemento y recolectando los resultados:

```
[1, 2, 3].map { |i| i * 3 }
# => [3, 6, 9]

['1', '2', '3', '4', '5'].map { |i| i.to_i }
# => [1, 2, 3, 4, 5]
```

La matriz original no se modifica; se devuelve una nueva matriz que contiene los valores transformados en el mismo orden que los valores de origen. `map!` Se puede utilizar si desea modificar la matriz original.

En el método de `map` , puede llamar al método o usar proc para todos los elementos de la matriz.

```
# call to_i method on all elements
%w(1 2 3 4 5 6 7 8 9 10).map(&:to_i)
# => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# using proc (lambda) on all elements
%w(1 2 3 4 5 6 7 8 9 10).map(&->(i){ i.to_i * 2})
# => [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

`map` es sinónimo de `collect` .

Creando un Array con el constructor literal []

Las matrices se pueden crear al incluir una lista de elementos entre corchetes (`[` y `]`). Los elementos de matriz en esta notación están separados por comas:

```
array = [1, 2, 3, 4]
```

Las matrices pueden contener cualquier tipo de objetos en cualquier combinación sin restricciones de tipo:

```
array = [1, 'b', nil, [3, 4]]
```

Crear matriz de cuerdas

Las matrices de cadenas se pueden crear utilizando la sintaxis de cadena de porcentaje de ruby:

```
array = %w(one two three four)
```

Esto es funcionalmente equivalente a definir la matriz como:

```
array = ['one', 'two', 'three', 'four']
```

En lugar de `%w()` puede usar otros pares de delimitadores coincidentes: `%w{...}` , `%w[...]` o `%w<...>` .

También es posible utilizar delimitadores no alfanuméricos arbitrarios, tales como: `%w!...!` , `%w#...#` o `%w@...@` .

Se puede usar `%W` lugar de `%w` para incorporar la interpolación de cadenas. Considera lo siguiente:

```
var = 'hello'

%w(#{var}) # => ["\#{var}"]
%W(#{var}) # => ["hello"]
```

Se pueden interpretar varias palabras al escapar del espacio con una \.

```
%w(Colorado California New\ York) # => ["Colorado", "California", "New York"]
```

Crear matriz de símbolos

2.0

```
array = %i(one two three four)
```

Crea la matriz `[:one, :two, :three, :four]` .

En lugar de `%i(...)` , puede usar `%i{...}` o `%i[...]` o `%i!...!`

Además, si desea usar interpolación, puede hacer esto con `%I`

2.0

```
a = 'hello'
b = 'goodbye'
array_one = %I(#{a} #{b} world)
array_two = %i(#{a} #{b} world)
```

Crea las matrices: `array_one = [:hello, :goodbye, :world] array_two = [:"\#{a}", :"\#{b}",`

`:world] array_one = [:hello, :goodbye, :world]` y `array_two = [:"\#{a}", :"\#{b}", :world]`

Crear Array con Array :: nuevo

Se puede crear una matriz vacía (`[]`) con el método de clase de la `Array::new` , `Array::new` :

```
Array.new
```

Para establecer la longitud de la matriz, pase un argumento numérico:

```
Array.new 3 #=> [nil, nil, nil]
```

Hay dos formas de rellenar una matriz con valores predeterminados:

- Pase un valor inmutable como segundo argumento.
- Pasa un bloque que obtiene el índice actual y genera valores mutables.

```
Array.new 3, :x #=> [:x, :x, :x]

Array.new(3) { |i| i.to_s } #=> ["0", "1", "2"]

a = Array.new 3, "X"            # Not recommended.
a[1].replace "C"                # a => ["C", "C", "C"]

b = Array.new(3) { "X" }        # The recommended way.
b[1].replace "C"                # b => ["X", "C", "X"]
```

Manipular elementos de matriz

Añadiendo elementos:

```
[1, 2, 3] << 4
# => [1, 2, 3, 4]

[1, 2, 3].push(4)
# => [1, 2, 3, 4]

[1, 2, 3].unshift(4)
# => [4, 1, 2, 3]

[1, 2, 3] << [4, 5]
# => [1, 2, 3, [4, 5]]
```

Eliminando elementos:

```
array = [1, 2, 3, 4]
array.pop
# => 4
array
# => [1, 2, 3]

array = [1, 2, 3, 4]
array.shift
```

```
# => 1
array
# => [2, 3, 4]

array = [1, 2, 3, 4]
array.delete(1)
# => 1
array
# => [2, 3, 4]

array = [1,2,3,4,5,6]
array.delete_at(2) // delete from index 2
# => 3
array
# => [1,2,4,5,6]


array = [1, 2, 2, 2, 3]
array - [2]
# => [1, 3]    # removed all the 2s
array - [2, 3, 4]
# => [1]       # the 4 did nothing
```

Combinando matrices:

```
[1, 2, 3] + [4, 5, 6]
# => [1, 2, 3, 4, 5, 6]

[1, 2, 3].concat([4, 5, 6])
# => [1, 2, 3, 4, 5, 6]

[1, 2, 3, 4, 5, 6] - [2, 3]
# => [1, 4, 5, 6]

[1, 2, 3] | [2, 3, 4]
# => [1, 2, 3, 4]

[1, 2, 3] & [3, 4]
# => [3]
```

También puede multiplicar matrices, por ejemplo

```
[1, 2, 3] * 2
# => [1, 2, 3, 1, 2, 3]
```

Arreglos de unión, intersección y diferencia.

```
x = [5, 5, 1, 3]
y = [5, 2, 4, 3]
```

Union (|) contiene elementos de ambas matrices, con duplicados eliminados:

```
x | y
=> [5, 1, 3, 2, 4]
```

La intersección (`&`) contiene elementos que están presentes tanto en la primera como en la segunda matriz:

```
x & y
=> [5, 3]
```

La diferencia (`-`) contiene elementos que están presentes en la primera matriz y no están presentes en la segunda matriz:

```
x - y
=> [1]
```

Matrices de filtrado

A menudo queremos operar solo con elementos de una matriz que cumplan una condición específica:

Seleccionar

Devolverá elementos que coincidan con una condición específica.

```
array = [1, 2, 3, 4, 5, 6]
array.select { |number| number > 3 } # => [4, 5, 6]
```

Rechazar

Devolverá elementos que no coincidan con una condición específica.

```
array = [1, 2, 3, 4, 5, 6]
array.reject { |number| number > 3 } # => [1, 2, 3]
```

Tanto `#select` como `#reject` devuelven una matriz, por lo que se pueden encadenar:

```
array = [1, 2, 3, 4, 5, 6]
array.select { |number| number > 3 }.reject { |number| number < 5 }
  # => [5, 6]
```

Inyectar, reducir

Inyectar y reducir son nombres diferentes para la misma cosa. En otros idiomas, estas funciones a menudo se llaman pliegues (como foldl o foldr). Estos métodos están disponibles en cada objeto Enumerable.

Inject toma una función de dos argumentos y la aplica a todos los pares de elementos en el Array.

Para la matriz `[1, 2, 3]` podemos agregar todo esto junto con el valor de inicio de cero especificando un valor de inicio y un bloque de este modo:

```
[1,2,3].reduce(0) {|a,b| a + b} # => 6
```

Aquí pasamos a la función un valor inicial y un bloque que dice que se agreguen todos los valores. El bloque se ejecuta primero con `0` como `a` y `1` como `b` , luego toma el resultado de eso como el siguiente `a` por lo que estamos agregando `1` al segundo valor `2` . Luego tomamos el resultado de eso (`3`) y lo agregamos al elemento final de la lista (también `3`) que nos da el resultado (`6`).

Si omitimos el primer argumento, se establece `a` a ser el primer elemento de la lista, por lo que el ejemplo anterior es lo mismo que:

```
[1,2,3].reduce {|a,b| a + b} # => 6
```

Además, en lugar de pasar un bloque con una función, podemos pasar una función nombrada como un símbolo, ya sea con un valor de inicio o sin él. Con esto, el ejemplo anterior podría escribirse como:

```
[1,2,3].reduce(0, :+) # => 6
```

u omitiendo el valor inicial:

```
[1,2,3].reduce(:+) # => 6
```

Elementos de acceso

Puede acceder a los elementos de una matriz por sus índices. La numeración del índice de matriz comienza en `0` .

```
%w(a b c)[0] # => 'a'
%w(a b c)[1] # => 'b'
```

Puedes recortar una matriz usando el rango

```
%w(a b c d)[1..2] # => ['b', 'c'] (indices from 1 to 2, including the 2)
%w(a b c d)[1...2] # => ['b'] (indices from 1 to 2, excluding the 2)
```

Esto devuelve una nueva matriz, pero no afecta al original. Ruby también apoya el uso de índices negativos.

```
%w(a b c)[-1] # => 'c'
%w(a b c)[-2] # => 'b'
```

También puedes combinar índices negativos y positivos.

```
%w(a b c d e)[1...-1] # => ['b', 'c', 'd']
```

Otros metodos utiles

Utilice `first` para acceder al primer elemento de una matriz:

```
[1, 2, 3, 4].first # => 1
```

O `first(n)` para acceder a los primeros `n` elementos devueltos en una matriz:

```
[1, 2, 3, 4].first(2) # => [1, 2]
```

Del mismo modo para el `last` y el `last(n)` :

```
[1, 2, 3, 4].last    # => 4
[1, 2, 3, 4].last(2) # => [3, 4]
```

Utilice la `sample` para acceder a un elemento aleatorio en una matriz:

```
[1, 2, 3, 4].sample  # => 3
[1, 2, 3, 4].sample  # => 1
```

O `sample(n)` :

```
[1, 2, 3, 4].sample(2) # => [2, 1]
[1, 2, 3, 4].sample(2) # => [3, 4]
```

Matriz bidimensional

Al usar `Array::new` constructor, puede inicializar una matriz con un tamaño dado y una nueva matriz en cada una de sus ranuras. Las matrices internas también pueden tener un tamaño y un valor inicial.

Por ejemplo, para crear una matriz de ceros 3x4:

```
array = Array.new(3) { Array.new(4) { 0 } }
```

La matriz generada anteriormente se ve así cuando se imprime con `p` :

```
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

Puedes leer o escribir en elementos como este:

```
x = array[0][1]
array[2][3] = 2
```

Arrays y el operador splat (*)

El operador `*` se puede usar para desempaquetar variables y matrices de modo que puedan pasarse como argumentos individuales a un método.

Esto se puede usar para envolver un solo objeto en una matriz si aún no lo está:

```
def wrap_in_array(value)
  [*value]
end

wrap_in_array(1)
#> [1]

wrap_in_array([1, 2, 3])
#> [1, 2, 3]

wrap_in_array(nil)
#> []
```

En el ejemplo anterior, el método `wrap_in_array` acepta un argumento, `value` .

Si el `value` es una `Array` , sus elementos se descomprimen y se crea una nueva matriz que contiene esos elementos.

Si el `value` es un solo objeto, se crea una nueva matriz que contiene ese único objeto.

Si el `value` es `nil` , se devuelve una matriz vacía.

El operador splat es particularmente útil cuando se usa como argumento en métodos en algunos casos. Por ejemplo, permite que los valores y matrices individuales y `nil` se manejen de manera consistente:

```
def list(*values)
  values.each do |value|
    # do something with value
    puts value
  end
end

list(100)
#> 100

list([100, 200])
#> 100
#> 200

list(nil)
# nothing is outputted
```

Descomposición

Cualquier matriz se puede **descomponer** rápidamente asignando sus elementos a múltiples variables. Un ejemplo simple:

```
arr = [1, 2, 3]
# ---
a = arr[0]
b = arr[1]
```

```
c = arr[2]
# --- or, the same
a, b, c = arr
```

Al preceder una variable con el operador *splat* (`*`), se coloca una matriz de todos los elementos que no han sido capturados por otras variables. Si no queda ninguno, se asigna una matriz vacía. Solo se puede usar un splat en una sola tarea:

```
a, *b = arr       # a = 1; b = [2, 3]
a, *b, c = arr    # a = 1; b = [2]; c = 3
a, b, c, *d = arr # a = 1; b = 2; c = 3; d = []
a, *b, *c = arr   # SyntaxError: unexpected *
```

La descomposición es *segura* y nunca plantea errores. `nil` s se asignan donde no hay elementos suficientes, que coinciden con el comportamiento del operador `[]` cuando se accede a un índice fuera de límites:

```
arr[9000] # => nil
a, b, c, d = arr # a = 1; b = 2; c = 3; d = nil
```

La descomposición intenta llamar `to_ary` implícita en la cesión de los objetos. Al implementar este método en su tipo, obtiene la capacidad de descomponerlo:

```
class Foo
  def to_ary
    [1, 2]
  end
end
a, b = Foo.new # a = 1; b = 2
```

Si el objeto que está siendo descompuesto no `respond_to? to_ary` , se trata como una matriz de un solo elemento:

```
1.respond_to?(:to_ary) # => false
a, b = 1 # a = 1; b = nil
```

La descomposición también se puede **anidar** utilizando una expresión de descomposición delimitada por `()` en lugar de lo que de otro modo sería un elemento único:

```
arr = [1, [2, 3, 4], 5, 6]
a, (b, *c), *d = arr # a = 1; b = 2; c = [3, 4]; d = [5, 6]
#   ^^^^^
```

Esto es efectivamente lo contrario de *splat* .

En realidad, cualquier expresión de descomposición puede ser delimitada por `()` . Pero para el primer nivel la descomposición es opcional.

```
a, b = [1, 2]
(a, b) = [1, 2] # the same thing
```

Caso perimetral: *un solo identificador* no se puede usar como patrón de desestructuración, ya sea externo o anidado:

```
(a) = [1] # SyntaxError
a, (b) = [1, [2]] # SyntaxError
```

Cuando se asigna **un literal de matriz** a una expresión de desestructuración, se puede omitir `[]` externo:

```
a, b = [1, 2]
a, b =  1, 2  # exactly the same
```

Esto se conoce como **asignación paralela** , pero utiliza la misma descomposición debajo del capó. Esto es particularmente útil para intercambiar valores de variables sin emplear variables temporales adicionales:

```
t = a; a = b; b = t # an obvious way
a, b = b, a         # an idiomatic way
(a, b) = [b, a]     # ...and how it works
```

Los valores se capturan cuando se construye el lado derecho de la asignación, por lo que usar las mismas variables como origen y destino es relativamente seguro.

Convierta una matriz multidimensional en una matriz unidimensional (aplanada)

```
[1, 2, [[3, 4], [5]], 6].flatten  # => [1, 2, 3, 4, 5, 6]
```

Si tiene una matriz multidimensional y necesita convertirla en una matriz *simple* (es decir, unidimensional), puede usar el método **`#flatten`** .

Obtener elementos de matriz únicos

En caso de que necesite leer una matriz ***evitando las repeticiones*** , use el método **`#uniq`** :

```
a = [1, 1, 2, 3, 4, 4, 5]
a.uniq
#=> [1, 2, 3, 4, 5]
```

En su lugar, si desea eliminar todos los elementos duplicados de una matriz, puede usar **`#uniq!`** método:

```
a = [1, 1, 2, 3, 4, 4, 5]
a.uniq!
#=> [1, 2, 3, 4, 5]
```

Mientras que la salida es la misma, **`#uniq!`** También almacena la nueva matriz:

```
a = [1, 1, 2, 3, 4, 4, 5]
a.uniq
#=> [1, 2, 3, 4, 5]
a
#=> [1, 1, 2, 3, 4, 4, 5]

a = [1, 1, 2, 3, 4, 4, 5]
a.uniq!
#=> [1, 2, 3, 4, 5]
a
#=> [1, 2, 3, 4, 5]
```

Obtener todas las combinaciones / permutaciones de una matriz

El método de `permutation` , cuando se llama con un bloque, produce una matriz bidimensional que consiste en todas las secuencias ordenadas de una colección de números.

Si este método se llama sin un bloque, devolverá un `enumerator` . Para convertir a una matriz, llame al método `to_a` .

Ejemplo	Resultado
`[1,2,3].permutation`	`#<Enumerator: [1,2,3]:permutation`
`[1,2,3].permutation.to_a`	`[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]`
`[1,2,3].permutation(2).to_a`	`[[1,2],[1,3],[2,1],[2,3],[3,1],[3,2]]`
`[1,2,3].permutation(4).to_a`	`[]` -> Sin permutaciones de longitud 4

El método de `combination` , por otro lado, cuando se llama con un bloque produce una matriz bidimensional que consta de todas las secuencias de una colección de números. A diferencia de la permutación, el orden se ignora en las combinaciones. Por ejemplo, `[1,2,3]` es lo mismo que `[3,2,1]`

Ejemplo	Resultado
`[1,2,3].combination(1)`	`#<Enumerator: [1,2,3]:combination`
`[1,2,3].combination(1).to_a`	`[[1],[2],[3]]`
`[1,2,3].combination(3).to_a`	`[[1,2,3]]`
`[1,2,3].combination(4).to_a`	`[]` -> Sin combinaciones de longitud 4

Llamar al método de combinación por sí mismo resultará en un enumerador. Para obtener una matriz, llame al método `to_a` .

Los métodos de `repeated_permutation` `repeated_combination` y de `repeated_permutation` son similares, excepto que el mismo elemento puede repetirse varias veces.

Por ejemplo, las secuencias `[1,1]` , `[1,3,3,1]` , `[3,3,3]` no serían válidas en combinaciones y

permutaciones regulares.

Ejemplo	# Combos
`[1,2,3].combination(3).to_a.length`	1
`[1,2,3].repeated_combination(3).to_a.length`	6
`[1,2,3,4,5].combination(5).to_a.length`	1
`[1,2,3].repeated_combination(5).to_a.length`	126

Crea una matriz de números o letras consecutivas

Esto se puede lograr fácilmente llamando a `Enumerable#to_a` en un objeto `Range` :

```
(1..10).to_a    #=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

`(a..b)` significa que incluirá todos los números entre ay b. Para excluir el último número, use `a...b`

```
a_range = 1...5
a_range.to_a       #=> [1, 2, 3, 4]
```

o

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

Un atajo conveniente para crear una matriz es `[*a..b]`

```
[*1..10]           #=> [1,2,3,4,5,6,7,8,9,10]
[*'a'..'f']        #=> ["a", "b", "c", "d", "e", "f"]
```

Eliminar todos los elementos nil de una matriz con #compact

Si resulta que una matriz tiene uno o más elementos `nil` y estos deben eliminarse, ¡ `Array#compact` o `Array#compact!` Se pueden utilizar métodos, como se muestra a continuación.

```
array = [ 1, nil, 'hello', nil, '5', 33]

array.compact # => [ 1, 'hello', '5', 33]

#notice that the method returns a new copy of the array with nil removed,
#without affecting the original

array = [ 1, nil, 'hello', nil, '5', 33]

#If you need the original array modified, you can either reassign it

array = array.compact # => [ 1, 'hello', '5', 33]
```

```
array = [ 1, 'hello', '5', 33]

#Or you can use the much more elegant 'bang' version of the method

array = [ 1, nil, 'hello', nil, '5', 33]

array.compact # => [ 1, 'hello', '5', 33]

array = [ 1, 'hello', '5', 33]
```

Finalmente, note que si `#compact` o `#compact!` se invocan en una matriz sin elementos `nil` , estos devolverán nil.

```
array = [ 'foo', 4, 'life']

array.compact # => nil

array.compact! # => nil
```

Crear matriz de números

La forma normal de crear una matriz de números:

```
numbers = [1, 2, 3, 4, 5]
```

Los objetos de rango pueden usarse ampliamente para crear una matriz de números:

```
numbers = Array(1..10) # => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

numbers = (1..10).to_a # => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

`#step` métodos `#step` y `#map` nos permiten imponer condiciones en el rango de números:

```
odd_numbers = (1..10).step(2).to_a # => [1, 3, 5, 7, 9]

even_numbers = 2.step(10, 2).to_a # => [2, 4, 6, 8, 10]

squared_numbers = (1..10).map { |number| number * number } # => [1, 4, 9, 16, 25, 36, 49, 64,
81, 100]
```

Todos los métodos anteriores cargan los números con impaciencia. Si tienes que cargarlos perezosamente:

```
number_generator = (1..100).lazy # => #<Enumerator::Lazy: 1..100>

number_generator.first(10) # => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Cast to Array desde cualquier objeto

Para obtener Array de cualquier objeto, use `Kernel#Array` .

Lo siguiente es un ejemplo:

```
Array('something') #=> ["something"]
Array([2, 1, 5])   #=> [2, 1, 5]
Array(1)           #=> [1]
Array(2..4)        #=> [2, 3, 4]
Array([])          #=> []
Array(nil)         #=> []
```

Por ejemplo, podría reemplazar el método `join_as_string` del siguiente código

```
def join_as_string(arg)
  if arg.instance_of?(Array)
    arg.join(',')
  elsif arg.instance_of?(Range)
    arg.to_a.join(',')
  else
    arg.to_s
  end
end

join_as_string('something') #=> "something"
join_as_string([2, 1, 5])   #=> "2,1,5"
join_as_string(1)           #=> "1"
join_as_string(2..4)        #=> "2,3,4"
join_as_string([])          #=> ""
join_as_string(nil)         #=> ""
```

al siguiente código.

```
def join_as_string(arg)
  Array(arg).join(',')
end
```

Lea Arrays en línea: https://riptutorial.com/es/ruby/topic/253/arrays

Capítulo 6: Arreglos Multidimensionales

Introducción

Las matrices multidimensionales en Ruby son matrices cuyos elementos son otras matrices.

El único problema es que dado que los arreglos de Ruby pueden contener elementos de tipos mixtos, debe estar seguro de que el arreglo que está manipulando está efectivamente compuesto de otros arreglos y no, por ejemplo, arreglos y cadenas.

Examples

Inicializando una matriz 2D

Primero recapitulemos cómo inicializar una matriz de números enteros de rubíes 1D:

```
my_array = [1, 1, 2, 3, 5, 8, 13]
```

Al ser una matriz 2D simplemente una matriz de matrices, puede inicializarla de la siguiente manera:

```
my_array = [
  [1, 1, 2, 3, 5, 8, 13],
  [1, 4, 9, 16, 25, 36, 49, 64, 81],
  [2, 3, 5, 7, 11, 13, 17]
]
```

Inicializando una matriz 3D

Puedes ir un nivel más abajo y agregar una tercera capa de arreglos. Las reglas no cambian:

```
my_array = [
  [
    [1, 1, 2, 3, 5, 8, 13],
    [1, 4, 9, 16, 25, 36, 49, 64, 81],
    [2, 3, 5, 7, 11, 13, 17]
  ],
  [
    ['a', 'b', 'c', 'd', 'e'],
    ['z', 'y', 'x', 'w', 'v']
  ],
  [
    []
  ]
]
```

Accediendo a una matriz anidada

Accediendo al 3er elemento del primer subarray:

```
my_array[1][2]
```

Aplanamiento de matrices

Dada una matriz multidimensional:

```
my_array = [[1, 2], ['a', 'b']]
```

la operación de aplanamiento es descomponer todos los elementos secundarios de la matriz en la matriz raíz:

```
my_array.flatten

# [1, 2, 'a', 'b']
```

Lea Arreglos Multidimensionales en línea: https://riptutorial.com/es/ruby/topic/10608/arreglos-multidimensionales

Capítulo 7: Atrapar excepciones con Begin / Rescue

Examples

Un bloque de manejo de error básico

Hagamos una función para dividir dos números, eso es muy confiado en su entrada:

```
def divide(x, y)
  return x/y
end
```

Esto funcionará bien para muchas entradas:

```
> puts divide(10, 2)
5
```

Pero no todos

```
> puts divide(10, 0)
ZeroDivisionError: divided by 0

> puts divide(10, 'a')
TypeError: String can't be coerced into Fixnum
```

Podemos reescribir la función envolviendo la operación de división de riesgo en un bloque de `begin... end` para verificar errores, y usar una cláusula de `rescue` para emitir un mensaje y devolver `nil` si hay un problema.

```
def divide(x, y)
  begin
    return x/y
  rescue
    puts "There was an error"
    return nil
  end
end

> puts divide(10, 0)
There was an error

> puts divide(10, 'a')
There was an error
```

Guardando el error

Puede guardar el error si desea utilizarlo en la cláusula de `rescue`

```
def divide(x, y)
  begin
    x/y
  rescue => e
    puts "There was a %s (%s)" % [e.class, e.message]
    puts e.backtrace
  end
end

> divide(10, 0)
There was a ZeroDivisionError (divided by 0)
       from (irb):10:in `/'
       from (irb):10
       from /Users/username/.rbenv/versions/2.3.1/bin/irb:11:in `<main>'

> divide(10, 'a')
There was a TypeError (String can't be coerced into Fixnum)
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/workspace.rb:87:in `eval'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/workspace.rb:87:in `evaluate'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/context.rb:380:in `evaluate'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:489:in `block (2 levels) in
eval_input'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:623:in `signal_status'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:486:in `block in eval_input'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/ruby-lex.rb:246:in `block (2 levels)
in each_top_level_statement'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/ruby-lex.rb:232:in `loop'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/ruby-lex.rb:232:in `block in
each_top_level_statement'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/ruby-lex.rb:231:in `catch'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb/ruby-lex.rb:231:in
`each_top_level_statement'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:485:in `eval_input'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:395:in `block in start'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:394:in `catch'
/Users/username/.rbenv/versions/2.3.1/lib/ruby/2.3.0/irb.rb:394:in `start'
/Users/username/.rbenv/versions/2.3.1/bin/irb:11:in `<main>'
```

Comprobación de errores diferentes

Si desea hacer cosas diferentes según el tipo de error, use varias cláusulas de `rescue` , cada una con un tipo de error diferente como argumento.

```
def divide(x, y)
  begin
    return x/y
  rescue ZeroDivisionError
    puts "Don't divide by zero!"
    return nil
  rescue TypeError
    puts "Division only works on numbers!"
    return nil
  end
end

> divide(10, 0)
Don't divide by zero!

> divide(10, 'a')
```

```
Division only works on numbers!
```

Si desea guardar el error para utilizarlo en el bloque de `rescue` :

```
rescue ZeroDivisionError => e
```

Utilice una cláusula de `rescue` sin argumentos para detectar errores de un tipo no especificado en otra cláusula de `rescue` .

```
def divide(x, y)
  begin
    return x/y
  rescue ZeroDivisionError
    puts "Don't divide by zero!"
    return nil
  rescue TypeError
    puts "Division only works on numbers!"
    return nil
  rescue => e
    puts "Don't do that (%s)" % [e.class]
    return nil
  end
end

> divide(nil, 2)
Don't do that (NoMethodError)
```

En este caso, tratar de dividir `nil` por 2 no es un `ZeroDivisionError` o un `TypeError` , por lo que se maneja con la cláusula de `rescue` predeterminada, que imprime un mensaje que nos permite saber que era un `NoMethodError` .

Reintentando

En una cláusula de `rescue` , puede usar `retry` para ejecutar la cláusula de `begin` nuevamente, probablemente después de cambiar la circunstancia que causó el error.

```
def divide(x, y)
  begin
    puts "About to divide..."
    return x/y
  rescue ZeroDivisionError
    puts "Don't divide by zero!"
    y = 1
    retry
  rescue TypeError
    puts "Division only works on numbers!"
    return nil
  rescue => e
    puts "Don't do that (%s)" % [e.class]
    return nil
  end
end
```

Si pasamos parámetros que sabemos que causarán un `TypeError` , la cláusula de `begin` se ejecuta

(marcada aquí imprimiendo "Acerca de la división") y el error se detecta como antes, y se devuelve `nil` :

```
> divide(10, 'a')
About to divide...
Division only works on numbers!
 => nil
```

Pero si pasamos parámetros que causarán un `ZeroDivisionError` , se ejecuta la cláusula `begin` , se captura el error, el divisor cambia de 0 a 1, y luego el `retry` hace que el bloque `begin` se ejecute nuevamente (desde arriba), ahora con un diferente `y` La segunda vez no hay ningún error y la función devuelve un valor.

```
> divide(10, 0)
About to divide...     # First time, 10 ÷ 0
Don't divide by zero!
About to divide...     # Second time 10 ÷ 1
=> 10
```

Comprobando si no se produjo ningún error

Puede usar una cláusula `else` para el código que se ejecutará si no se produce ningún error.

```
def divide(x, y)
  begin
    z = x/y
  rescue ZeroDivisionError
    puts "Don't divide by zero!"
  rescue TypeError
    puts "Division only works on numbers!"
    return nil
  rescue => e
    puts "Don't do that (%s)" % [e.class]
    return nil
  else
    puts "This code will run if there is no error."
    return z
  end
end
```

La cláusula `else` no se ejecuta si hay un error que transfiere el control a una de las cláusulas de `rescue` :

```
> divide(10,0)
Don't divide by zero!
=> nil
```

Pero si no se produce ningún error, la cláusula `else` ejecuta:

```
> divide(10,2)
This code will run if there is no error.
=> 5
```

Tenga en cuenta que la cláusula `else` no se ejecutará *si regresa de la cláusula* `begin`

```
def divide(x, y)
  begin
    z = x/y
    return z                  # Will keep the else clause from running!
  rescue ZeroDivisionError
    puts "Don't divide by zero!"
  else
    puts "This code will run if there is no error."
    return z
  end
end

> divide(10,2)
=> 5
```

Código que siempre debe correr

Utilice un `ensure` cláusula si hay código que siempre se desea ejecutar.

```
def divide(x, y)
  begin
    z = x/y
    return z
  rescue ZeroDivisionError
    puts "Don't divide by zero!"
  rescue TypeError
    puts "Division only works on numbers!"
    return nil
  rescue => e
    puts "Don't do that (%s)" % [e.class]
    return nil
  ensure
    puts "This code ALWAYS runs."
  end
end
```

La `ensure` cláusula se ejecutará cuando hay un error:

```
> divide(10, 0)
Don't divide by zero!   # rescue clause
This code ALWAYS runs.   # ensure clause
=> nil
```

Y cuando no hay error:

```
> divide(10, 2)
This code ALWAYS runs.   # ensure clause
=> 5
```

La cláusula de garantía es útil cuando quiere asegurarse, por ejemplo, de que los archivos están cerrados.

Tenga en cuenta que, a diferencia de la `else` cláusula, el `ensure` cláusula *se ejecuta* antes del

`begin` o `rescue` cláusula devuelve un valor. Si el `ensure` cláusula tiene un `return` que va a anular el `return` valor de cualquier otra cláusula!

Lea Atrapar excepciones con Begin / Rescue en línea:
https://riptutorial.com/es/ruby/topic/7327/atrapar-excepciones-con-begin---rescue

Capítulo 8: Bloques y Procs y Lambdas

Sintaxis

- Proc.nuevo (*bloque*)
- lambda {| args | código}
- -> (arg1, arg2) {código}
- object.to_proc
- {| single_arg | código}
- do | arg, (clave, valor) | *código* final

Observaciones

Tenga cuidado con la prioridad del operador cuando tenga una línea con múltiples métodos encadenados, como:

```
str = "abcdefg"
puts str.gsub(/./) do |match|
  rand(2).zero? ? match.upcase : match.downcase
end
```

En lugar de imprimir algo como `abCDeFg` , como es de esperar, imprime algo como `#<Enumerator:0x00000000af42b28>` - esto es porque `do ... end` tiene menor precedencia que los métodos, lo que significa que `gsub` solo ve el argumento `/./` , y no el argumento del bloque. Devuelve un enumerador. El bloque termina pasado a las `puts` , lo que lo ignora y solo muestra el resultado de `gsub(/./)` .

Para solucionar este problema, envuelva la llamada `gsub` entre paréntesis o use `{ ... }` lugar.

Examples

Proc

```
def call_the_block(&calling); calling.call; end

its_a = proc do |*args|
  puts "It's a..." unless args.empty?
  "beautiful day"
end

puts its_a       #=> "beautiful day"
puts its_a.call  #=> "beautiful day"
puts its_a[1, 2] #=> "It's a..." "beautiful day"
```

Hemos copiado el método `call_the_block` del último ejemplo. Aquí, puede ver que un proceso se realiza llamando al método `proc` con un bloque. También puede ver que los bloques, como los métodos, tienen retornos implícitos, lo que significa que procs (y lambdas) también lo tienen. En la

definición de `its_a` , puede ver que los bloques pueden tomar argumentos splat así como los normales; también son capaces de tomar argumentos predeterminados, pero no se me ocurre una forma de trabajar en eso. Por último, puede ver que es posible usar varias sintaxis para llamar a un método, ya sea el método de `call` o el `[]` operador.

Lambdas

```
# lambda using the arrow syntax
hello_world = -> { 'Hello World!' }
hello_world[]
# 'Hello World!'

# lambda using the arrow syntax accepting 1 argument
hello_world = ->(name) { "Hello #{name}!" }
hello_world['Sven']
# "Hello Sven!"

the_thing = lambda do |magic, ohai, dere|
  puts "magic! #{magic}"
  puts "ohai #{dere}"
  puts "#{ohai} means hello"
end

the_thing.call(1, 2, 3)
# magic! 1
# ohai 3
# 2 means hello

the_thing.call(1, 2)
# ArgumentError: wrong number of arguments (2 for 3)

the_thing[1, 2, 3, 4]
# ArgumentError: wrong number of arguments (4 for 3)
```

También puede usar `->` para crear y `.()` Para llamar a lambda

```
the_thing = ->(magic, ohai, dere) {
  puts "magic! #{magic}"
  puts "ohai #{dere}"
  puts "#{ohai} means hello"
}

the_thing.(1, 2, 3)
# => magic! 1
# => ohai 3
# => 2 means hello
```

Aquí puedes ver que un lambda es casi lo mismo que un proc. Sin embargo, hay varias advertencias:

- La aridad de los argumentos de un lambda se hace cumplir; pasar el número incorrecto de argumentos a un lambda, generará un `ArgumentError` . Todavía pueden tener parámetros predeterminados, parámetros splat, etc.
- `return` ing desde una lambda devuelve desde la lambda, mientras `return` ing desde una proc

devuelve fuera del alcance adjunto:

```
def try_proc
  x = Proc.new {
    return # Return from try_proc
  }
  x.call
  puts "After x.call" # this line is never reached
end

def try_lambda
  y = -> {
    return # return from y
  }
  y.call
  puts "After y.call" # this line is not skipped
end

try_proc # No output
try_lambda # Outputs "After y.call"
```

Objetos como argumentos de bloque a métodos.

Poner un `&` (ampersand) delante de un argumento lo pasará como el bloque del método. Los objetos se convertirán en un `Proc` utilizando el método `to_proc` .

```
class Greeter
  def to_proc
    Proc.new do |item|
      puts "Hello, #{item}"
    end
  end
end

greet = Greeter.new

%w(world life).each(&greet)
```

Este es un patrón común en Ruby y muchas clases estándar lo proporcionan.

Por ejemplo, `Symbol` s implementa `to_proc` enviándose ellos mismos al argumento:

```
# Example implementation
class Symbol
  def to_proc
    Proc.new do |receiver|
      receiver.send self
    end
  end
end
```

Esto habilita el lenguaje útil `&:symbol` , comúnmente utilizado con objetos `Enumerable` :

```
letter_counts = %w(just some words).map(&:length)  # [4, 4, 5]
```

Bloques

Los bloques son fragmentos de código encerrados entre llaves `{}` (generalmente para bloques de una sola línea) o `do..end` (utilizados para bloques de varias líneas).

```
5.times { puts "Hello world" } # recommended style for single line blocks

5.times do
    print "Hello "
    puts "world"
end   # recommended style for multi-line blocks

5.times {
    print "hello "
    puts "world" } # does not throw an error but is not recommended
```

Nota: los frenos tienen mayor prioridad que `do..end`

Flexible

Los bloques se pueden usar dentro de los métodos y funciones usando la palabra `yield` :

```
def block_caller
    puts "some code"
    yield
    puts "other code"
end
block_caller { puts "My own block" } # the block is passed as an argument to the method.
#some code
#My own block
#other code
```

Tenga cuidado, si se llama a `yield` sin un bloque, se generará un `LocalJumpError` . Para este propósito ruby proporciona otro método llamado `block_given?` esto le permite verificar si se pasó un bloqueo antes de llamar a rendimiento

```
def block_caller
  puts "some code"
  if block_given?
    yield
  else
    puts "default"
  end
  puts "other code"
end
block_caller
# some code
# default
# other code
block_caller { puts "not defaulted"}
# some code
# not defaulted
# other code
```

`yield` puede ofrecer argumentos al bloque también.

```
def yield_n(n)
  p = yield n if block_given?
  p || n
end
yield_n(12) {|n| n + 7 }
#=> 19
yield_n(4)
#=> 4
```

Si bien este es un ejemplo simple, el `yield` puede ser muy útil para permitir el acceso directo a las variables de la instancia o evaluaciones dentro del contexto de otro objeto. Por ejemplo:

```
class Application
  def configuration
    @configuration ||= Configuration.new
    block_given? ? yield(@configuration) : @configuration
  end
end
class Configuration; end

app = Application.new
app.configuration do |config|
  puts config.class.name
end
# Configuration
#=> nil
app.configuration
#=> #<Configuration:0x2bf1d30>
```

Como puede ver, usar el `yield` de esta manera hace que el código sea más legible que llamar continuamente a `app.configuration.#method_name` . En su lugar, puede realizar toda la configuración dentro del bloque manteniendo el código contenido.

Variables

Las variables para bloques son locales al bloque (similares a las variables de funciones), mueren cuando se ejecuta el bloque.

```
my_variable = 8
3.times do |x|
    my_variable = x
    puts my_variable
end
puts my_variable
#=> 0
# 1
# 2
# 8
```

Los bloques no se pueden salvar, mueren una vez ejecutados. Para guardar bloques necesitas usar `procs` y `lambdas` .

Convertir a Proc

Los objetos que responden a `to_proc` se pueden convertir a procs con el operador `&` (lo que también permitirá que se pasen como bloques).

El símbolo de la clase define `#to_proc` por lo que intenta llamar al método correspondiente en el objeto que recibe como parámetro.

```
p [ 'rabbit', 'grass' ].map( &:upcase ) # => ["RABBIT", "GRASS"]
```

Los objetos de método también definen `#to_proc` .

```
output = method( :p )

[ 'rabbit', 'grass' ].map( &output ) # => "rabbit\ngrass"
```

Aplicación parcial y curry

Técnicamente, Ruby no tiene funciones, sino métodos. Sin embargo, un método de Ruby se comporta de manera casi idéntica a las funciones en otro idioma:

```
def double(n)
  n * 2
end
```

Este método / función normal toma un parámetro `n` , lo duplica y devuelve el valor. Ahora definamos una función de orden superior (o método):

```
def triple(n)
  lambda {3 * n}
end
```

En lugar de devolver un número, el `triple` devuelve un método. Puedes probarlo usando el Ruby Interactive Shell :

```
$ irb --simple-prompt
>> def double(n)
>>   n * 2
>> end
=> :double
>> def triple(n)
>>   lambda {3 * n}
>> end
=> :triple
>> double(2)
=> 4
>> triple(2)
=> #<Proc:0x007fd07f07bdc0@(irb):7 (lambda)>
```

Si realmente desea obtener el número triplicado, debe llamar (o "reducir") la lambda:

```
triple_two = triple(2)
triple_two.call # => 6
```

O más concisamente:

```
triple(2).call
```

Aplicaciones al curry y parciales.

Esto no es útil en términos de definir una funcionalidad muy básica, pero es útil si desea tener métodos / funciones que no sean llamados o reducidos instantáneamente. Por ejemplo, supongamos que desea definir métodos que agreguen un número por un número específico (por ejemplo, `add_one(2) = 3`). Si tuvieras que definir una tonelada de estos, podrías hacer:

```
def add_one(n)
  n + 1
end

def add_two(n)
  n + 2
end
```

Sin embargo, también podría hacer esto:

```
add = -> (a, b) { a + b }
add_one = add.curry.(1)
add_two = add.curry.(2)
```

Usando el cálculo lambda podemos decir que `add` es `(λa.(λb.(a+b)))` . Currying es una forma de *aplicar parcialmente* `add` . Entonces `add.curry.(1)` , es `(λa.(λb.(a+b)))(1)` que puede reducirse a `(λb.(1+b))` . La aplicación parcial significa que pasamos un argumento para `add` pero dejamos el otro argumento para proporcionarlo más adelante. La salida es un método especializado.

Ejemplos más útiles de curry

Digamos que tenemos una fórmula general realmente grande, que si le especificamos ciertos argumentos, podemos obtener fórmulas específicas de ella. Considera esta fórmula:

```
f(x, y, z) = sin(x\*y)*sin(y\*z)*sin(z\*x)
```

Esta fórmula está hecha para trabajar en tres dimensiones, pero digamos que solo queremos esta fórmula con respecto a y y z. Digamos también que para ignorar x, queremos establecer su valor en pi / 2. Primero hagamos la fórmula general:

```
f = ->(x, y, z) {Math.sin(x*y) * Math.sin(y*z) * Math.sin(z*x)}
```

Ahora, usemos el curry para obtener nuestra fórmula `yz` :

```
f_yz = f.curry.(Math::PI/2)
```

Luego para llamar a la lambda almacenada en `f_yz` :

```
f_xy.call(some_value_x, some_value_y)
```

Esto es bastante simple, pero digamos que queremos obtener la fórmula para `xz` . ¿Cómo podemos configurar `y` en `Math::PI/2` si no es el último argumento? Bueno, es un poco más complicado:

```
f_xz = -> (x,z) {f.curry.(x, Math::PI/2, z)}
```

En este caso, debemos proporcionar marcadores de posición para el parámetro que no estamos rellenando previamente. Por coherencia podríamos escribir `f_xy` así:

```
f_xy = -> (x,y) {f.curry.(x, y, Math::PI/2)}
```

Así es como funciona el cálculo lambda para `f_yz` :

```
f = (λx.(λy.(λz.(sin(x*y) * sin(y*z) * sin(z*x))))
f_yz = (λx.(λy.(λz.(sin(x*y) * sin(y*z) * sin(z*x)))) (π/2) # Reduce =>
f_yz = (λy.(λz.(sin((π/2)*y) * sin(y*z) * sin(z*(π/2)))))
```

Ahora veamos `f_xz`

```
f = (λx.(λy.(λz.(sin(x*y) * sin(y*z) * sin(z*x))))
f_xz = (λx.(λy.(λz.(sin(x*y) * sin(y*z) * sin(z*x)))) (λt.t) (π/2)  # Reduce =>
f_xz = (λt.(λz.(sin(t*(π/2)) * sin((π/2)*z) * sin(z*t))))
```

Para más información sobre el cálculo lambda intente esto .

Lea Bloques y Procs y Lambdas en línea: https://riptutorial.com/es/ruby/topic/474/bloques-y-procs-y-lambdas

Capítulo 9: Cargando archivos de origen

Examples

Requieren que los archivos se carguen solo una vez

El método Kernel # require cargará los archivos solo una vez (varias llamadas `require` que el código en ese archivo sea evaluado solo una vez). `$LOAD_PATH` tu ruby `$LOAD_PATH` para encontrar el archivo requerido si el parámetro no es una ruta absoluta. Extensiones como `.rb` , `.so` , `.o` o `.dll` son opcionales. Las rutas relativas se resolverán en el directorio de trabajo actual del proceso.

```
require 'awesome_print'
```

El Kernel # require_relative le permite cargar archivos relativos al archivo en el que se llama a `require_relative` .

```
# will search in directory myproj relative to current source file.
#
require_relative 'myproj/version'
```

Carga automática de archivos fuente

El método `Kernel#autoload` registra el nombre de archivo a cargar (usando `Kernel::require`) la primera vez que se accede a ese módulo (que puede ser una cadena o un símbolo).

```
autoload :MyModule, '/usr/local/lib/modules/my_module.rb'
```

El método Kernel # autoload? devuelve el nombre de archivo que se cargará si el nombre se registra como `autoload` .

```
autoload? :MyModule  #=> '/usr/local/lib/modules/my_module.rb'
```

Cargando archivos opcionales

Cuando los archivos no están disponibles, la familia `require` lanzará un `LoadError` . Este es un ejemplo que ilustra la carga de módulos opcionales solo si existen.

```
module TidBits

@@unavailableModules = []

[
      { name: 'CoreExtend', file: 'core_extend/lib/core_extend'  } \
    , { name: 'Fs'        , file: 'fs/lib/fs'                    } \
    , { name: 'Options'   , file: 'options/lib/options'          } \
    , { name: 'Susu'      , file: 'susu/lib/susu'                } \
```

```
].each do |lib|

    begin

        require_relative lib[ :file ]

    rescue LoadError

        @@unavailableModules.push lib

    end

end

end # module TidBits
```

Cargando archivos repetidamente

El método de carga Kernel # evaluará el código en el archivo dado. La ruta de búsqueda se construirá como se `require` . Reevaluará ese código en cada llamada subsiguiente a diferencia del `require` . No hay `load_relative` .

```
load `somefile`
```

Cargando varios archivos

Puedes usar cualquier técnica de ruby para crear dinámicamente una lista de archivos para cargar. Ilustración de globbing para archivos que comienzan con `test` , cargados en orden alfabético.

```
Dir[ "#{ __dir__ }**/test*.rb" ) ].sort.each do |source|

    require_relative source

end
```

Lea Cargando archivos de origen en línea: https://riptutorial.com/es/ruby/topic/3166/cargando-archivos-de-origen

Capítulo 10: Casting (conversión de tipo)

Examples

Casting a una cadena

```
123.5.to_s    #=> "123.5"
String(123.5) #=> "123.5"
```

Normalmente, `String()` solo llamará `#to_s` .

Los métodos `Kernel#sprintf` y `String#%` comportan de manera similar a C:

```
sprintf("%s", 123.5) #=> "123.5"
"%s" % 123.5 #=> "123.5"
"%d" % 123.5 #=> "123"
"%.2f" % 123.5 #=> "123.50"
```

Casting a un entero

```
"123.50".to_i     #=> 123
Integer("123.50") #=> 123
```

Una cadena tomará el valor de cualquier entero al comienzo, pero no tomará enteros de ninguna otra parte:

```
"123-foo".to_i # => 123
"foo-123".to_i # => 0
```

Sin embargo, hay una diferencia cuando la cadena no es un entero válido:

```
"something".to_i     #=> 0
Integer("something") # ArgumentError: invalid value for Integer(): "something"
```

Casting a un flotador

```
"123.50".to_f   #=> 123.5
Float("123.50") #=> 123.5
```

Sin embargo, hay una diferencia cuando la cadena no es un `Float` válido:

```
"something".to_f   #=> 0.0
Float("something") # ArgumentError: invalid value for Float(): "something"
```

Flotadores y enteros

```
1/2 #=> 0
```

Como estamos dividiendo dos enteros, el resultado es un entero. Para resolver este problema, necesitamos lanzar al menos uno de esos a Float:

```
1.0 / 2       #=> 0.5
1.to_f / 2    #=> 0.5
1 / Float(2) #=> 0.5
```

Alternativamente, `fdiv` puede usarse para devolver el resultado de división de punto flotante sin lanzar explícitamente ningún operando:

```
1.fdiv 2 # => 0.5
```

Lea Casting (conversión de tipo) en línea: https://riptutorial.com/es/ruby/topic/219/casting--conversion-de-tipo-

Capítulo 11: Clase Singleton

Sintaxis

- singleton_class = clase << objeto; auto final

Observaciones

Las clases de Singleton solo tienen una instancia: su objeto correspondiente. Esto se puede verificar consultando el `ObjectSpace` de Ruby:

```
instances = ObjectSpace.each_object object.singleton_class

instances.count            # => 1
instances.include? object  # => true
```

Usando `<` , también se puede verificar que sean subclases de la clase real del objeto:

```
object.singleton_class < object.class  # => true
```

Referencias:

- Tres contextos implícitos en ruby

Examples

Introducción

Ruby tiene tres tipos de objetos:

- Clases y módulos que son instancias de clase Clase o clase Módulo.
- Instancias de las clases.
- Clases singleton.

Cada objeto tiene una clase que contiene sus métodos:

```
class Example
end

object = Example.new

object.class  # => Example
Example.class # => Class
Class.class   # => Class
```

Los objetos en sí mismos no pueden contener métodos, solo su clase puede. Pero con las clases

de singleton, es posible agregar métodos a cualquier objeto, incluidas otras clases de singleton.

```
def object.foo
  :foo
end
object.foo #=> :foo
```

`foo` se define en la clase de `object` singleton. Otros `Example` casos no pueden responder a `foo` .

Ruby crea clases de singleton bajo demanda. Acceder a ellos o agregarles métodos obliga a Ruby a crearlos.

Acceso a la clase Singleton

Hay dos formas de obtener la clase singleton de un objeto.

- método `singleton_class`
- Reapertura de la clase singleton de un objeto y regreso del `self` .

```
object.singleton_class
```

```
singleton_class = class << object
  self
end
```

Acceso a variables de instancia / clase en clases singleton

Las clases de Singleton comparten sus variables de instancia / clase con su objeto.

```
class Example
  @@foo = :example
end

def Example.foo
  class_variable_get :@@foo
end

Example.foo #=> :example
```

```
class Example
  def initialize
    @foo = 1
  end

  def foo
    @foo
  end
end

e = Example.new
```

```
e.instance_eval <<-BLOCK
  def self.increase_foo
    @foo += 1
  end
BLOCK

e.increase_foo
e.foo #=> 2
```

Los bloques se cierran alrededor de sus variables de instancia / clase de destino. Acceder a las variables de instancia o clase usando un bloque en `class_eval` o `instance_eval` no es posible. Pasar una cadena a `class_eval` o usar `class_variable_get` el problema.

```
class Foo
  @@foo = :foo
end

class Example
  @@foo = :example

  Foo.define_singleton_method :foo do
    @@foo
  end
end

Foo.foo #=> :example
```

Herencia de la clase Singleton

Subclasificación también Subclases Clase Singleton

```
class Example
end

Example.singleton_class #=> #<Class:Example>

def Example.foo
  :example
end

class SubExample < Example
end

SubExample.foo #=> :example

SubExample.singleton_class.superclass #=> #<Class:Example>
```

Extender o incluir un módulo no amplía la clase Singleton

```
module ExampleModule
end

def ExampleModule.foo
```

```
  :foo
end

class Example
  extend ExampleModule
  include ExampleModule
end

Example.foo #=> NoMethodError: undefined method
```

Propagación de mensajes con Singleton Class

Las instancias nunca contienen un método que solo llevan datos. Sin embargo, podemos definir una clase singleton para cualquier objeto, incluida una instancia de una clase.

Cuando se pasa un mensaje a un objeto (se llama método), Ruby primero verifica si una clase singleton está definida para ese objeto y si puede responder a ese mensaje, de lo contrario Ruby verifica la cadena de ancestros de la clase de la instancia y continúa con eso.

```
class Example
  def foo
    :example
  end
end

Example.new.foo #=> :example

module PrependedModule
  def foo
    :prepend
  end
end

class Example
  prepend PrependedModule
end

Example.ancestors #=> [Prepended, Example, Object, Kernel, BasicObject]
e = Example.new
e.foo #=> :prepended

def e.foo
  :singleton
end

e.foo #=> :singleton
```

Reapertura (parches de monos) Clases Singleton

Hay tres formas de reabrir una clase Singleton

- Usando `class_eval` en una clase de singleton.
- Usando la `class <<` bloque.
- Usando `def` para definir un método en la clase singleton del objeto directamente

```
class Example
end

Example.singleton_class.class_eval do
  def foo
    :foo
  end
end

Example.foo #=> :foo
```

```
class Example
end

class << Example
  def bar
    :bar
  end
end

Example.bar #=> :bar
```

```
class Example
end

def Example.baz
  :baz
end

Example.baz #=> :baz
```

Cada objeto tiene una clase singleton a la que puedes acceder

```
class Example
end
ex1 = Example.new
def ex1.foobar
  :foobar
end
ex1.foobar #=> :foobar

ex2 = Example.new
ex2.foobar #=> NoMethodError
```

Clases singleton

Todos los objetos son instancias de una clase. Sin embargo, esa no es toda la verdad. En Ruby, cada objeto también tiene una *clase singleton* algo oculta.

Esto es lo que permite definir métodos en objetos individuales. La clase singleton se encuentra entre el objeto en sí y su clase real, por lo que todos los métodos definidos en él están disponibles para ese objeto, y solo ese objeto.

```
object = Object.new

def object.exclusive_method
  'Only this object will respond to this method'
end

object.exclusive_method
# => "Only this object will respond to this method"

Object.new.exclusive_method rescue $!
# => #<NoMethodError: undefined method `exclusive_method' for #<Object:0xa17b77c>>
```

El ejemplo anterior podría haberse escrito usando `define_singleton_method` :

```
object.define_singleton_method :exclusive_method do
  "The method is actually defined in the object's singleton class"
end
```

Lo que equivale a definir el método en `singleton_class` `object` :

```
# send is used because define_method is private
object.singleton_class.send :define_method, :exclusive_method do
  "Now we're defining an instance method directly on the singleton class"
end
```

Antes de la existencia de `singleton_class` como parte de la API central de Ruby, las clases de singleton se conocían como *metaclases* y se podía acceder a ellas a través del siguiente idioma:

```
class << object
  self  # refers to object's singleton_class
end
```

Lea Clase Singleton en línea: https://riptutorial.com/es/ruby/topic/4277/clase-singleton

Capítulo 12: Cola

Sintaxis

- q = Queue.nuevo
- objeto q.push
- q << objeto # igual que #push
- q.pop # => objeto

Examples

Múltiples trabajadores un fregadero

Queremos recopilar datos creados por múltiples trabajadores.

Primero creamos una cola:

```
sink = Queue.new
```

Luego 16 trabajadores, todos generando un número aleatorio y empujándolo hacia el sumidero:

```
(1..16).to_a.map do
  Thread.new do
    sink << rand(1..100)
  end
end.map(&:join)
```

Y para obtener los datos, convierta una cola en una matriz:

```
data = [].tap { |a| a << sink.pop until sink.empty? }
```

Una fuente de trabajadores múltiples

Queremos procesar los datos en paralelo.

Vamos a llenar la fuente con algunos datos:

```
source = Queue.new
data = (1..100)
data.each { |e| source << e }
```

Luego crea algunos trabajadores para procesar los datos:

```
(1..16).to_a.map do
  Thread.new do
    until source.empty?
      item = source.pop
```

```
      sleep 0.5
      puts "Processed: #{item}"
    end
  end
end.map(&:join)
```

One Source - Pipeline of Work - One Sink

Queremos procesar datos en paralelo y empujarlos hacia abajo para que otros trabajadores los procesen.

Como los trabajadores consumen y producen datos, tenemos que crear dos colas:

```
first_input_source = Queue.new
first_output_sink  = Queue.new
100.times { |i| first_input_source << i }
```

La primera oleada de trabajadores lee un elemento de `first_input_source` , procesa el elemento y escribe los resultados en `first_output_sink` :

```
(1..16).to_a.map do
  Thread.new do
    loop do
      item = first_input_source.pop
      first_output_source << item ** 2
      first_output_source << item ** 3
    end
  end
end
```

La segunda ola de trabajadores usa `first_output_sink` como su fuente de entrada y lee, luego el proceso escribe en otro receptor de salida:

```
second_input_source = first_output_sink
second_output_sink  = Queue.new

(1..32).to_a.map do
  Thread.new do
    loop do
      item = second_input_source.pop
      second_output_sink << item * 2
      second_output_sink << item * 3
    end
  end
end
```

Ahora `second_output_sink` es el sumidero, vamos a convertirlo en una matriz:

```
sleep 5 # workaround in place of synchronization
sink = second_output_sink
[].tap { |a| a << sink.pop until sink.empty? }
```

Empujando datos en una cola - #push

```
q = Queue.new
q << "any object including another queue"
# or
q.push :data
```

- No hay marca de agua alta, las colas pueden crecer infinitamente.
- `#push` nunca bloquea

Extraer datos de una cola - #pop

```
q = Queue.new
q << :data
q.pop #=> :data
```

- `#pop` se bloqueará hasta que haya algunos datos disponibles.
- `#pop` se puede utilizar para la sincronización.

Sincronización - Después de un punto en el tiempo

```
syncer = Queue.new

a = Thread.new do
  syncer.pop
  puts "this happens at end"
end

b = Thread.new do
  puts "this happens first"
  STDOUT.flush
  syncer << :ok
end

[a, b].map(&:join)
```

Convertir una cola en una matriz

```
q = Queue.new
q << 1
q << 2

a = Array.new
a << q.pop until q.empty?
```

O un trazador de líneas :

```
[].tap { |array| array < queue.pop until queue.empty? }
```

Fusionando dos colas

- Para evitar el bloqueo infinito, la lectura de las colas no debería ocurrir en la combinación de hilos en la que está ocurriendo.

- Para evitar la sincronización o la espera infinita de una de las colas mientras que otra tiene datos, la lectura de las colas no debería ocurrir en el mismo hilo.

Empecemos definiendo y llenando dos colas:

```
q1 = Queue.new
q2 = Queue.new
(1..100).each { |e| q1 << e }
(101..200).each { |e| q2 << e }
```

Deberíamos crear otra cola e introducir datos de otros subprocesos en ella:

```
merged = Queue.new

[q1, q2].map do |q|
  Thread.new do
    loop do
      merged << q.pop
    end
  end
end
```

Si sabe que puede consumir ambas colas completamente (la velocidad de consumo es mayor que la producción, no se quedará sin RAM), hay un enfoque más simple:

```
merged = Queue.new
merged << q1.pop until q1.empty?
merged << q2.pop until q2.empty?
```

Lea Cola en línea: https://riptutorial.com/es/ruby/topic/4666/cola

Capítulo 13: Comentarios

Examples

Comentarios de líneas simples y múltiples

Los comentarios son anotaciones legibles por el programador que se ignoran en el tiempo de ejecución. Su propósito es hacer que el código fuente sea más fácil de entender.

Comentarios de una sola línea

El carácter `#` se usa para agregar comentarios de una sola línea.

```
#!/usr/bin/ruby -w
# This is a single line comment.
puts "Hello World!"
```

Cuando se ejecute, el programa anterior dará salida a `Hello World!`

Comentarios multilínea

Los comentarios de varias líneas se pueden agregar utilizando la sintaxis de `=begin` y `=end` (también conocidas como marcadores de bloque de comentarios) de la siguiente manera:

```
#!/usr/bin/ruby -w
=begin
This is a multiline comment.
Write as many line as you want.
=end
puts "Hello World!"
```

Cuando se ejecute, el programa anterior dará salida a `Hello World!`

Lea Comentarios en línea: https://riptutorial.com/es/ruby/topic/3464/comentarios

Capítulo 14: Comparable

Sintaxis

- `include Comparable`
- implementar el operador de la nave espacial (`<=>`)

Parámetros

Parámetro	Detalles
otro	La instancia a comparar con `self`

Observaciones

`x <=> y` debe devolver un número negativo si `x < y` , cero si `x == y` y un número positivo si `x > y` .

Examples

Rectángulo comparable por área

`Comparable` es uno de los módulos más populares en Ruby. Su finalidad es proporcionar métodos de comparación de conveniencia.

Para usarlo, debe `include Comparable` y definir el operador de la nave espacial (`<=>`):

```
class Rectangle
  include Comparable

  def initialize(a, b)
    @a = a
    @b = b
  end

  def area
    @a * @b
  end

  def <=>(other)
    area <=> other.area
  end
end

r1 = Rectangle.new(1, 1)
r2 = Rectangle.new(2, 2)
r3 = Rectangle.new(3, 3)

r2 >= r1 # => true
r2.between? r1, r3 # => true
```

```
r3.between? r1, r2 # => false
```

Lea Comparable en línea: https://riptutorial.com/es/ruby/topic/1485/comparable

Capítulo 15: Constantes

Sintaxis

- MY_CONSTANT_NAME = "mi valor"

Observaciones

Las constantes son útiles en Ruby cuando tiene valores que no desea que se cambien por error en un programa, como las claves API.

Examples

Define una constante

```
MY_CONSTANT = "Hello, world" # constant
Constant = 'This is also constant' # constant
my_variable = "Hello, venus" # not constatn
```

El nombre constante comienza con mayúscula. Todo lo que comienza con mayúscula se considera `constant` en Ruby. Así que la `class` y el `module` también son constantes. La mejor práctica es usar todas las letras mayúsculas para declarar constante.

Modificar una constante

```
MY_CONSTANT = "Hello, world"
MY_CONSTANT = "Hullo, world"
```

El código anterior genera una advertencia, ya que debería utilizar variables si desea cambiar sus valores. Sin embargo, es posible cambiar una letra a la vez en una constante sin advertencia, como esto:

```
MY_CONSTANT = "Hello, world"
MY_CONSTANT[1] = "u"
```

Ahora, después de cambiar la segunda letra de `MY_CONSTANT` , se convierte en `"Hullo, world"` .

Las constantes no se pueden definir en métodos.

```
def say_hi
  MESSAGE = "Hello"
  puts MESSAGE
end
```

El código anterior produce un error: `SyntaxError: (irb):2: dynamic constant assignment` .

Definir y cambiar constantes en una clase.

```
class Message
  DEFAULT_MESSAGE = "Hello, world"

  def speak(message = nil)
    if message
      puts message
    else
      puts DEFAULT_MESSAGE
    end
  end
end
```

La constante `DEFAULT_MESSAGE` se puede cambiar con el siguiente código:

```
Message::DEFAULT_MESSAGE = "Hullo, world"
```

Lea Constantes en línea: https://riptutorial.com/es/ruby/topic/4093/constantes

Capítulo 16: Constantes especiales en Ruby

Examples

__EXPEDIENTE__

Es la ruta relativa al archivo desde el directorio de ejecución actual.
Supongamos que tenemos esta estructura de directorios: /home/stackoverflow/script.rb
script.rb contiene:

```
puts __FILE__
```

Si está dentro de / home / stackoverflow y ejecuta el script como `ruby script.rb` entonces `__FILE__` generará `script.rb` Si está dentro de / home, generará `stackoverflow/script.rb`

Muy útil para obtener la ruta del script en versiones anteriores a 2.0 donde no existe `__dir__` .

Nota `__FILE__` no es igual a `__dir__`

__dir__

`__dir__` no es una constante sino una función
`__dir__` es igual a `File.dirname(File.realpath(__FILE__))`

$ PROGRAM_NAME o $ 0

Contiene el nombre del script que se está ejecutando.
Es lo mismo que `__FILE__` si está ejecutando ese script.

$$

El número de proceso del Ruby ejecutando este script.

$ 1, $ 2, etc.

Contiene el subpatrón del conjunto de paréntesis correspondiente en el último patrón exitoso coincidente, sin contar los patrones coincidentes en bloques anidados que ya se han salido, o nil si la última coincidencia de patrón falló. Estas variables son todas de solo lectura.

ARGV o $ *

Argumentos de línea de comando dados para el script. Las opciones para el intérprete de Ruby ya están eliminadas.

STDIN

La entrada estándar. El valor predeterminado para $ stdin

Repartir

La salida estándar. El valor predeterminado para $ stdout

STDERR

La salida de error estándar. El valor predeterminado para $ stderr

$ stderr

La salida de error estándar actual.

$ stdout

La salida estándar actual.

$ stdin

La entrada estándar actual

ENV

El objeto tipo hash contiene variables de entorno actuales. Establecer un valor en ENV cambia el entorno para los procesos secundarios.

Lea Constantes especiales en Ruby en línea: https://riptutorial.com/es/ruby/topic/4037/constantes-especiales-en-ruby

Capítulo 17: Creación / gestión de gemas

Examples

Archivos Gemspec

Cada gema tiene un archivo en el formato `<gem name>.gemspec` que contiene metadatos sobre la gema y sus archivos. El formato de una gemspec es el siguiente:

```
Gem::Specification.new do |s|
  # Details about gem. They are added in the format:
  s.<detail name> = <detail value>
end
```

Los campos requeridos por RubyGems son:

Cualquiera de los `author = string` o `authors = array`

Use `author =` si solo hay un autor, y `authors =` cuando hay múltiples. Para `authors=` utilizar una matriz que enumera los nombres de los autores.

```
files = array
```

Aquí la `array` es una lista de todos los archivos en la gema. Esto también se puede usar con la función `Dir[]` , por ejemplo, si todos sus archivos están en el directorio `/lib/` , entonces puede usar `files = Dir["/lib/"]` .

```
name = string
```

Aquí la cadena es solo el nombre de tu gema. Rubygems recomienda algunas reglas que debes seguir al nombrar tu gema.

1. Use guiones bajos, sin espacios
2. Use solo letras minúsculas
3. Utilice hypens para la extensión joya (por ejemplo, si su joya se llama `example` para una extensión que le nombrarlo `example-extension`) de manera que cuando el entonces extensión se requiere que se le puede exigir que `require "example/extension"` .

RubyGems también agrega "Si publicas una gema en rubygems.org, se puede eliminar si el nombre es objetable, viola la propiedad intelectual o el contenido de la gema cumple con estos criterios. Puedes reportar dicha gema en el sitio de soporte de RubyGems".

```
platform=
```

No lo sé

```
require_paths=
```

No lo sé

```
summary= string
```

String es un resumen del propósito de las gemas y cualquier cosa que quieras compartir sobre la gema.

```
version= string
```

El número de versión actual de la gema.

Los campos recomendados son:

```
email = string
```

Una dirección de correo electrónico que se asociará con la gema.

```
homepage= string
```

El sitio web donde vive la gema.

O `license=` o `licenses=`

No lo sé

Construyendo una gema

Una vez que hayas creado tu gema para publicarla, debes seguir algunos pasos:

1. Construye tu gema con `gem build <gem name>.gemspec` (el archivo gemspec debe existir)
2. Crea una cuenta de RubyGems si aún no tienes una aquí
3. Comprueba que no existan gemas que compartan el nombre de tus gemas.
4. Publica tu gema con `gem publish <gem name>.<gem version number>.gem`

Dependencias

Para listar el árbol de dependencias:

```
gem dependency
```

Para enumerar qué gemas dependen de una gema específica (agrupador, por ejemplo)

```
gem dependency bundler --reverse-dependencies
```

Lea Creación / gestión de gemas en línea: https://riptutorial.com/es/ruby/topic/4092/creacion---

gestion-de-gemas

Capítulo 18: Depuración

Examples

Pasando por el código con Pry y Byebug

Primero, necesitas instalar la gema `pry-byebug` . Ejecute este comando:

```
$ gem install pry-byebug
```

Agregue esta línea en la parte superior de su archivo `.rb` :

```
require 'pry-byebug'
```

Luego inserta esta línea donde quieras un punto de interrupción:

```
binding.pry
```

Un ejemplo de `hello.rb` :

```
require 'pry-byebug'

def hello_world
  puts "Hello"
  binding.pry # break point here
  puts "World"
end
```

Cuando ejecute el archivo `hello.rb` , el programa se detendrá en esa línea. A continuación, puede recorrer su código con el comando de `step` . Escriba el nombre de una variable para conocer su valor. Salga del depurador con `exit-program` o `!!!` .

Lea Depuración en línea: https://riptutorial.com/es/ruby/topic/7691/depuracion

Capítulo 19: Destruccion

Examples

Visión general

La mayor parte de la magia de la desestructuración utiliza el operador splat (`*`).

Ejemplo	Resultado / *comentario*
`a, b = [0,1]`	`a=0, b=1`
`a, *rest = [0,1,2,3]`	`a=0, rest=[1,2,3]`
`a, * = [0,1,2,3]`	`a=0` *Equivalente a* `.first`
`*, z = [0,1,2,3]`	`z=3` *Equivalente a* `.last`

La destrucción de los argumentos de bloque

```
triples = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

triples.each { |(first, second, third)| puts second }
# 2
# 5
# 8

triples.map { |(first, *rest)| rest.join(' ') } # => ["2 3", "5 6", "8 9"]
```

Lea Destruccion en línea: https://riptutorial.com/es/ruby/topic/4739/destruccion

Capítulo 20: Distancia

Examples

Gamas como secuencias

El uso más importante de los rangos es expresar una secuencia.

Sintaxis:

```
(begin..end) => this construct will include end value
(begin...end) => this construct will exclude end value
```

o

```
Range.new(begin,end,exclude_end) => exclude_end is by default false
```

El valor `end` más importante debe ser mayor al `begin` , de lo contrario no devolverá nada.

Ejemplos:

```
(10..1).to_a            #=> []
(1...3)                 #=> [1, 2]
(-6..-1).to_a           #=> [-6, -5, -4, -3, -2, -1]
('a'..'e').to_a         #=> ["a", "b", "c", "d", "e"]
('a'...'e').to_a        #=> ["a", "b", "c", "d"]
Range.new(1,3).to_a     #=> [1, 2, 3]
Range.new(1,3,true).to_a#=> [1, 2]
```

Iterando sobre un rango

Puedes hacer algo fácilmente para cada elemento en un rango.

```
(1..5).each do |i|
    print i
end
# 12345
```

Rango entre fechas

```
require 'date'

date1 = Date.parse "01/06/2016"
date2 = Date.parse "05/06/2016"

p "Period #{date1.strftime("%d/%m/%Y")} to #{date2.strftime("%d/%m/%Y")}"

(date1..date2).each do |date|
  p date.strftime("%d/%m/%Y")
```

```
end

# "01/06/2016"
# "02/06/2016"
# "03/06/2016"
# "04/06/2016"
# "05/06/2016"
```

Lea Distancia en línea: https://riptutorial.com/es/ruby/topic/3427/distancia

Capítulo 21: Empezando con Hanami

Introducción

Mi misión aquí es contribuir con la comunidad para ayudar a las nuevas personas que desean aprender sobre este marco increíble: Hanami.

Pero, ¿cómo va a funcionar?

Tutoriales cortos y sencillos que muestran ejemplos sobre Hanami y siguiendo los siguientes tutoriales, veremos cómo probar nuestra aplicación y crear una API REST simple.

¡Empecemos!

Examples

Acerca de Hanami

Además de que Hanami sea un marco liviano y rápido, uno de los puntos que más llama la atención es el concepto de **Arquitectura Limpia** donde nos muestra que el marco no es nuestra aplicación como Robert Martin dijo anteriormente.

Hanami arquitecture design nos ofrece el uso de **Container** , en cada Container tenemos nuestra aplicación independientemente del marco. Esto significa que podemos tomar nuestro código y ponerlo en un marco de Rails, por ejemplo.

¿Hanami es un framework MVC?

La idea de los marcos del MVC es construir una estructura siguiendo el Modelo -> Controlador -> Ver. Hanami sigue el modelo | Controlador -> Ver -> Plantilla. El resultado es una aplicación más sin saltos, siguiendo los principios de **SOLID** y mucho más limpio.

- Links importantes.

Hanami http://hanamirb.org/

Robert Martin - Arquitectura limpia https://www.youtube.com/watch?v=WpkDN78P884

Arquitectura limpia https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html

Principios Sólidos http://practicingruby.com/articles/solid-design-principles

¿Cómo instalar Hanami?

- **Paso 1:** Instalación de la gema Hanami.

```
$ gem install hanami
```

- **Paso 2** : Generar un nuevo proyecto configurando RSpec como marco de prueba.

  Abre una línea de comando o terminal. Para generar una nueva aplicación de hanami, use hanami new seguido del nombre de su aplicación y el parámetro de prueba rspec.

```
$ hanami new "myapp" --test=rspec
```

Obs. Por defecto, Hanami establece a Minitest como marco de prueba.

Esto creará una aplicación de hanami llamada myapp en un directorio de myapp e instalará las dependencias de gemas que ya se mencionaron en Gemfile usando la instalación de paquetes.

Para cambiar a este directorio, use el comando cd, que significa cambiar directorio.

```
$ cd my_app
$ bundle install
```

El directorio myapp tiene una serie de archivos y carpetas generados automáticamente que conforman la estructura de una aplicación Hanami. A continuación se muestra una lista de archivos y carpetas que se crean de forma predeterminada:

- **Gemfile** define nuestras dependencias de Rubygems (usando Bundler).
- **Rakefile** describe nuestras tareas de Rake.
- **aplicaciones** contiene una o más aplicaciones web compatibles con Rack. Aquí podemos encontrar la primera aplicación de Hanami generada llamada Web. Es el lugar donde encontramos nuestros controladores, vistas, rutas y plantillas.
- **config** contiene archivos de configuración.
- **config.ru** es para servidores Rack.
- **db** contiene nuestro esquema de base de datos y migraciones.
- **lib** contiene nuestra lógica de negocios y modelo de dominio, incluyendo entidades y repositorios.
- **El público** contendrá activos estáticos compilados.
- **Spec** contiene nuestras pruebas.
- **Links importantes.**

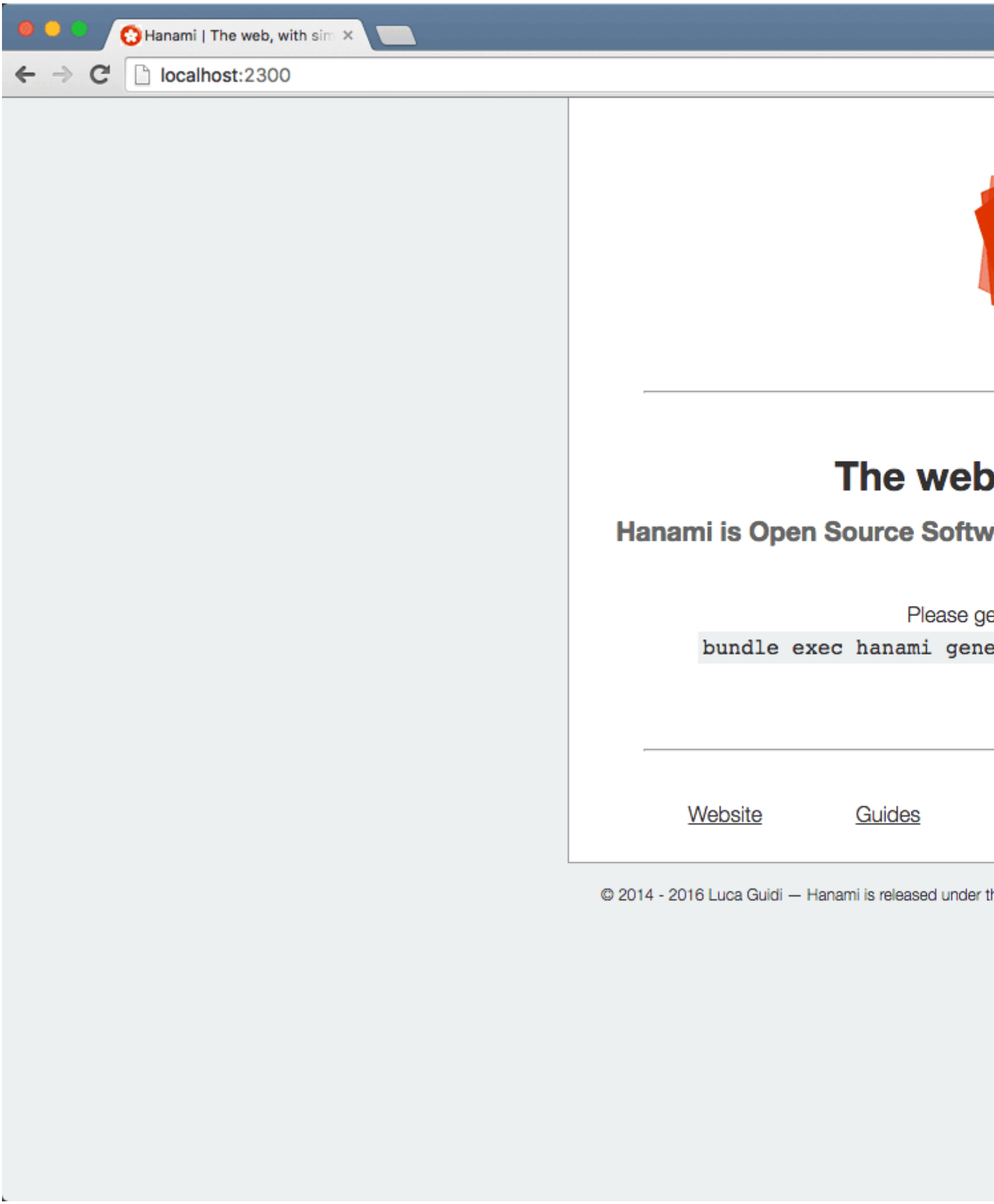
Hanami joy https://github.com/hanami/hanami

Hanami official Getting Started http://hanamirb.org/guides/getting-started/

¿Cómo iniciar el servidor?

- **Paso 1:** Para iniciar el servidor, simplemente escriba el siguiente comando y verá la página de inicio.

```
$ bundle exec hanami server
```



Lea Empezando con Hanami en línea: https://riptutorial.com/es/ruby/topic/9676/empezando-con-hanami

Capítulo 22: Enumerable en ruby

Introducción

Enumerable módulo, hay un conjunto de métodos disponibles para realizar recorridos, clasificación, búsqueda, etc. en toda la colección (Array, Hashes, Set, HashMap).

Examples

Módulo enumerable

```
1. For Loop:
CountriesName = ["India", "Canada", "America", "Iraq"]
for country in CountriesName
    puts country
end

2. Each Iterator:
Same set of work can be done with each loop which we did with for loop.
CountriesName = ["India", "Canada", "America", "Iraq"]
CountriesName.each do |country|
    puts country
end


Each iterator, iterate over every single element of the array.
each ---------- iterator
do ------------ start of the block
|country| ----- argument passed to the block
puts country----block



3. each_with_index Iterator:
each_with_index iterator provides the element for the current iteration and index of the
element in that specific collection.
CountriesName = ["India", "Canada", "America", "Iraq"]
CountriesName.each_with_index do |country, index|
    puts country + "  " + index.to_s
end


4. each_index Iterator:
Just to know the index at which the element is placed in the collection.
CountriesName = ["India", "Canada", "America", "Iraq"]
CountriesName.each_index do |index|
    puts index
end


5. map:
"map" acts as an iterator and also used to fetch the transformed copy of the array. To fetch
the new set of the array rather than introducing the change in the same specific array.
Let's deal with for loop first:
```

```
You have an array arr = [1,2,3,4,5]
You need to produce new set of array.
arr = [1,2,3,4,5]
newArr = []
for x in 0..arr.length-1
   newArr[x] = -arr[x]
end


The above mentioned array can be iterated and can produce new set of the array using map
method.
arr = [1,2,3,4,5]
newArr = arr.map do |x|
    -x
      end

puts arr
[1,2,3,4,5]

puts newArr
[-1, -2, -3, -4, -5]


map is returning the modified copy of the current value of the collection. arr has unaltered
value.


Difference between each and map:
1. map returned the modified value of the collection.

Let's see the example:
arr = [1,2,3,4,5]
newArr = arr.map do |x|
    puts x
    -x
      end

puts newArr
[-1, -2, -3, -4, -5]

map method is the iterator and also return the copy of transformed collection.


arr = [1,2,3,4,5]
newArr = arr.each do |x|
    puts x
    -x
      end

puts newArr
[1,2,3,4,5]

each block will throw the array because this is just the iterator.
Each iteration, doesn't actually alter each element in the iteration.


6. map!
map with bang changes the orginal collection and returned the modified collection not the copy
of the modified collection.
arr = [1,2,3,4,5]
arr.map! do |x|
```

```
    puts x
    -x
end
puts  arr
[-1, -2, -3, -4, -5]

7. Combining map and each_with_index
Here each_with_index will iterator over the collection and map will return the modified copy
of the collection.
CountriesName = ["India", "Canada", "America", "Iraq"]
newArray =
CountriesName.each_with_index.map do |value, index|
    puts "Value is #{value} and the index is #{index}"
    "Value is #{value} and the index is #{index}"
end

newArray =
CountriesName.each_with_index.map do |value, index|
    if ((index%2).eql?0)
        puts "Value is #{value} and the index is #{index}"
        "Value is #{value} and the index is #{index}"
    end
end

puts newArray
 ["Value is India and the index is 0", nil, "Value is America and the index is 2", nil]


8. select
MixedArray = [1, "India", 2, "Canada", "America", 4]
MixedArray.select do |value|
    (value.class).eql?Integer
end

select method fetches the result based on satifying certain condition.


9. inject methods
inject method reduces the collection to a certain final value.
Let's say you want to find out the sum of the collection.
With for loop how would it work
arr = [1,2,3,4,5]
sum = 0
for x in 0..arr.length-1
   sum = sum + arr[0]
end
puts sum
15

So above mentioned sum can be reduce by single method
arr = [1,2,3,4,5]
arr.inject(0) do |sum, x|
    puts x
    sum = sum + x
end
inject(0) - passing initial value sum = 0
If used inject with no argument sum = arr[0]
sum - After each iteration, total is equal to the return value at the end of the block.
x  - refers to the current iteration's element

inject method is also an iterator.
```

Resumen: la mejor manera de transformar la colección es hacer uso del módulo Enumerable para compactar el código torpe.

Lea Enumerable en ruby en línea: https://riptutorial.com/es/ruby/topic/10786/enumerable-en-ruby

Capítulo 23: Enumeradores

Introducción

Un `Enumerator` es un objeto que implementa la iteración de una manera controlada.

En lugar de realizar un bucle hasta que se cumpla alguna condición, el objeto *enumera los* valores según sea necesario. La ejecución del bucle se detiene hasta que el propietario del objeto solicita el siguiente valor.

Los enumeradores hacen posible la transmisión infinita de valores.

Parámetros

Parámetro	Detalles
`yield`	Responde al `yield` , que se presenta como `<<` . El ceder a este objeto implementa la iteración.

Examples

Enumeradores personalizados

Vamos a crear un `Enumerator` para los números de Fibonacci .

```
fibonacci = Enumerator.new do |yielder|
  a = b = 1
  loop do
    yielder << a
    a, b = b, a + b
  end
end
```

Ahora podemos usar cualquier método `Enumerable` con `fibonacci` :

```
fibonacci.take 10
# => [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Metodos existentes

Si se llama a un método de iteración como `each` sin un bloque, se debe devolver un `Enumerator` .

Esto se puede hacer usando el método `enum_for` :

```
def each
  return enum_for :each unless block_given?
```

```
  yield :x
  yield :y
  yield :z
end
```

Esto le permite al programador componer operaciones `Enumerable` :

```
each.drop(2).map(&:upcase).first
# => :Z
```

Rebobinado

Utilice `rewind` para reiniciar el enumerador.

```
N = Enumerator.new do |yielder|
  x = 0
  loop do
    yielder << x
    x += 1
  end
end

N.next
# => 0

N.next
# => 1

N.next
# => 2

N.rewind

N.next
# => 0
```

Lea Enumeradores en línea: https://riptutorial.com/es/ruby/topic/4985/enumeradores

Capítulo 24: ERB

Introducción

ERB significa Embedded Ruby, y se utiliza para insertar variables de Ruby dentro de plantillas, por ejemplo, HTML y YAML. ERB es una clase de Ruby que acepta texto, y evalúa y reemplaza el código de Ruby rodeado por el marcado ERB.

Sintaxis

- <% number = rand (10)%> este código será evaluado
- <% = número%> este código se evaluará e insertará en la salida
- <% # comentario texto%> este comentario no será evaluado

Observaciones

Convenciones:

- ERB como plantilla: abstraiga la lógica empresarial en el código auxiliar, y mantenga sus plantillas ERB limpias y legibles para las personas sin conocimiento de Ruby.
- `.erb` archivos con `.erb` : por ejemplo, `.js.erb` , `.html.erb` , `.css.erb` , etc.

Examples

Análisis de ERB

Este ejemplo es texto filtrado de una sesión `IRB` .

```
=> require 'erb'
=> input = <<-HEREDOC
<ul>
<% (0..10).each do |i| %>
    <%# This is a comment %>
    <li><%= i %> is <%= i.even? ? 'even' : 'odd' %>.</li>
<% end %>
</ul>
HEREDOC

=> parser = ERB.new(input)
=> output = parser.result
=> print output
<ul>


    <li>0 is even.</li>


    <li>1 is odd.</li>
```

```
    <li>2 is even.</li>

    <li>3 is odd.</li>

    <li>4 is even.</li>

    <li>5 is odd.</li>

    <li>6 is even.</li>

    <li>7 is odd.</li>

    <li>8 is even.</li>

    <li>9 is odd.</li>

    <li>10 is even.</li>
</ul>
```

Lea ERB en línea: https://riptutorial.com/es/ruby/topic/8145/erb

Capítulo 25: Evaluación dinámica

Sintaxis

- eval "fuente"
- eval "fuente", vinculante
- eval "fuente", proc
- binding.eval "source" # igual a `eval "source", binding`

Parámetros

Parámetro	Detalles
`"source"`	Cualquier código fuente de Ruby
`binding`	Una instancia de clase `Binding`
`proc`	Una instancia de la clase `Proc`

Examples

Evaluación de instancias

El método `instance_eval` está disponible en todos los objetos. Evalúa el código en el contexto del receptor:

```
object = Object.new

object.instance_eval do
  @variable = :value
end

object.instance_variable_get :@variable  # => :value
```

`instance_eval` establece `self` para `object` durante la duración del bloque de código:

```
object.instance_eval { self == object }  # => true
```

El receptor también se pasa al bloque como su único argumento:

```
object.instance_eval { |argument| argument == object }  # => true
```

El método `instance_exec` difiere a este respecto: en su lugar, pasa sus argumentos al bloque.

```
object.instance_exec :@variable do |name|
```

```
  instance_variable_get name  # => :value
end
```

Evaluando una cadena

Cualquier `String` puede ser evaluada en tiempo de ejecución.

```
class Example
  def self.foo
    :foo
  end
end

eval "Example.foo" #=> :foo
```

Evaluando dentro de un enlace

Ruby realiza un seguimiento de las variables locales y de la `self` variable a través de un objeto llamado enlace. Podemos obtener un enlace de un ámbito con `Kernel#binding` y evaluar una cadena dentro de un enlace a través de `Binding#eval` .

```
b = proc do
  local_variable = :local
  binding
end.call

b.eval "local_variable" #=> :local
```

```
def fake_class_eval klass, source = nil, &block
  class_binding = klass.send :eval, "binding"

  if block
    class_binding.local_variable_set :_fake_class_eval_block, block
    class_binding.eval "_fake_class_eval_block.call"
  else
    class_binding.eval source
  end
end

class Example
end

fake_class_eval Example, <<-BLOCK
  def self.foo
    :foo
  end
BLOCK

fake_class_eval Example do
  def bar
    :bar
  end
end

Example.foo #=> :foo
```

```
Example.new.bar #=> :bar
```

Creación dinámica de métodos a partir de cadenas

Ruby ofrece `define_method` como un método privado en módulos y clases para definir nuevos métodos de instancia. Sin embargo, el 'cuerpo' del método debe ser un `Proc` u otro método existente.

Una forma de crear un método a partir de datos de cadena sin procesar es utilizar `eval` para crear un Proc a partir del código:

```
xml = <<ENDXML
<methods>
  <method name="go">puts "I'm going!"</method>
  <method name="stop">7*6</method>
</methods>
ENDXML

class Foo
  def self.add_method(name,code)
    body = eval( "Proc.new{ #{code} }" )
    define_method(name,body)
  end
end

require 'nokogiri' # gem install nokogiri
doc = Nokogiri.XML(xml)
doc.xpath('//method').each do |meth|
  Foo.add_method( meth['name'], meth.text )
end

f = Foo.new
p Foo.instance_methods(false)  #=> [:go, :stop]
p f.public_methods(false)      #=> [:go, :stop]
f.go                           #=> "I'm going!"
p f.stop                       #=> 42
```

Lea Evaluación dinámica en línea: https://riptutorial.com/es/ruby/topic/5048/evaluacion-dinamica

Capítulo 26: Excepciones

Observaciones

Una *excepción* es un objeto que representa la ocurrencia de una condición excepcional. En otras palabras, indica que algo salió mal.

En Ruby, las *excepciones a* menudo se conocen como *errores* . Esto se debe a que la clase de `Exception` base existe como un elemento de objeto de excepción de nivel superior, pero las excepciones de ejecución definidas por el usuario generalmente son `StandardError` o descendientes.

Examples

Levantando una excepción

Para generar una excepción, use `Kernel#raise` pasando la clase de excepción y / o el mensaje:

```
raise StandardError # raises a StandardError.new
raise StandardError, "An error" # raises a StandardError.new("An error")
```

También puede simplemente pasar un mensaje de error. En este caso, el mensaje se envuelve en un `RuntimeError` :

```
raise "An error" # raises a RuntimeError.new("An error")
```

Aquí hay un ejemplo:

```
def hello(subject)
  raise ArgumentError, "`subject` is missing" if subject.to_s.empty?
  puts "Hello #{subject}"
end

hello # => ArgumentError: `subject` is missing
hello("Simone") # => "Hello Simone"
```

Creando un tipo de excepción personalizado

Una excepción personalizada es cualquier clase que amplíe `Exception` o una subclase de `Exception` .

En general, siempre debe extender `StandardError` o un descendiente. La familia de `Exception` suele ser por errores de máquinas virtuales o del sistema, al rescatarlos se puede evitar que una interrupción forzada funcione como se espera.

```
# Defines a new custom exception called FileNotFound
```

```
class FileNotFound < StandardError
end

def read_file(path)
  File.exist?(path) || raise(FileNotFound, "File #{path} not found")
  File.read(path)
end

read_file("missing.txt")  #=> raises FileNotFound.new("File `missing.txt` not found")
read_file("valid.txt")    #=> reads and returns the content of the file
```

Es común nombrar excepciones agregando el sufijo de `Error` al final:

- `ConnectionError`
- `DontPanicError`

Sin embargo, cuando el error se explica por sí mismo, no es necesario agregar el sufijo de `Error` porque sería redundante:

- `FileNotFound` VS `FileNotFoundError`
- `DatabaseExploded` VS `DatabaseExplodedError`

Manejando una excepción

Use el bloque de `begin/rescue` para capturar (rescatar) una excepción y manejarla:

```
begin
  # an execution that may fail
rescue
  # something to execute in case of failure
end
```

Una cláusula de `rescue` es análoga a un bloque `catch` en un lenguaje con llaves como C # o Java.

Un simple `rescue` como este rescata a `StandardError` .

Nota: tenga cuidado de evitar la captura de `Exception` lugar del `StandardError` predeterminado. La clase de `Exception` incluye `SystemExit` y `NoMemoryError` y otras excepciones serias que normalmente no desea capturar. Siempre considere capturar `StandardError` (el predeterminado) en su lugar.

También puede especificar la clase de excepción que debe ser rescatada:

```
begin
  # an excecution that may fail
rescue CustomError
  # something to execute in case of CustomError
  # or descendant
end
```

Esta cláusula de rescate no detectará ninguna excepción que no sea `CustomError` .

También puede almacenar la excepción en una variable específica:

```
begin
  # an excecution that may fail
rescue CustomError => error
  # error contains the exception
  puts error.message # provide human-readable details about what went wrong.
  puts error.backtrace.inspect # return an array of strings that represent the call stack
end
```

Si no pudo manejar una excepción, puede elevarla en cualquier momento en un bloque de rescate.

```
begin
   #here goes your code
rescue => e
    #failed to handle
    raise e
end
```

Si desea reintentar su bloque de `begin` , llame a `retry` :

```
begin
   #here goes your code
rescue StandardError => e
   #for some reason you want to retry you code
   retry
end
```

Puede quedar atrapado en un bucle si detecta una excepción en cada reintento. Para evitar esto, limite su `retry_count` a un cierto número de intentos.

```
retry_count = 0
begin
      # an excecution that may fail
rescue
    if retry_count < 5
        retry_count = retry_count + 1
        retry
    else
        #retry limit exceeds, do something else
    end
```

También puede proporcionar un bloque `else` o un bloque `ensure` . Se ejecutará un bloque `else` cuando el bloque de `begin` se complete sin una excepción lanzada. Siempre se ejecutará un bloque de `ensure` . Un bloque de `ensure` es análogo a un bloque `finally` en un lenguaje de llaves como C # o Java.

```
begin
  # an execution that may fail
rescue
  # something to execute in case of failure
else
  # something to execute in case of success
ensure
  # something to always execute
```

```
end
```

Si se encuentra dentro de una `def` , `module` o bloque de `class` , no es necesario utilizar la instrucción begin.

```
def foo
    ...
rescue
    ...
end
```

Manejando múltiples excepciones

Puedes manejar múltiples errores en la misma declaración de `rescue` :

```
begin
  # an execution that may fail
rescue FirstError, SecondError => e
  # do something if a FirstError or SecondError occurs
end
```

También puede agregar múltiples declaraciones de `rescue` :

```
begin
  # an execution that may fail
rescue FirstError => e
  # do something if a FirstError occurs
rescue SecondError => e
  # do something if a SecondError occurs
rescue => e
  # do something if a StandardError occurs
end
```

El orden de los bloques de `rescue` es relevante: la primera coincidencia es la ejecutada. Por lo tanto, si coloca `StandardError` como la primera condición y todas sus excepciones heredan de `StandardError` , las otras declaraciones de `rescue` nunca se ejecutarán.

```
begin
  # an execution that may fail
rescue => e
  # this will swallow all the errors
rescue FirstError => e
  # do something if a FirstError occurs
rescue SecondError => e
  # do something if a SecondError occurs
end
```

Algunos bloques tienen un manejo de excepciones implícito como `def` , `class` y `module` . Estos bloques le permiten omitir la instrucción `begin` .

```
def foo
    ...
```

```
rescue CustomError
    ...
ensure
    ...
end
```

Agregando información a excepciones (personalizadas)

Puede ser útil incluir información adicional con una excepción, por ejemplo, para fines de registro o para permitir el manejo condicional cuando se detecta la excepción:

```
class CustomError < StandardError
  attr_reader :safe_to_retry

  def initialize(safe_to_retry = false, message = 'Something went wrong')
    @safe_to_retry = safe_to_retry
    super(message)
  end
end
```

Levantando la excepción:

```
raise CustomError.new(true)
```

Capturando la excepción y accediendo a la información adicional proporcionada:

```
begin
  # do stuff
rescue CustomError => e
  retry if e.safe_to_retry
end
```

Lea Excepciones en línea: https://riptutorial.com/es/ruby/topic/940/excepciones

Capítulo 27: Expresiones regulares y operaciones basadas en expresiones regulares

Examples

Grupos, nombrados y otros.

Ruby extiende la sintaxis estándar del grupo `(...)` con un grupo nombrado, `(?<name>...)` . Esto permite la extracción por nombre en lugar de tener que contar cuántos grupos tiene.

```
name_reg = /h(i|ello), my name is (?<name>.*)/i #i means case insensitive

name_input = "Hi, my name is Zaphod Beeblebrox"

match_data = name_reg.match(name_input) #returns either a MatchData object or nil
match_data = name_input.match(name_reg) #works either way

if match_data.nil? #Always check for nil! Common error.
  puts "No match"
else
  match[0] #=> "Hi, my name is Zaphod Beeblebrox"
  match[1] #=> "i" #the first group, (i|ello)
  match[2] #=> "Zaphod Beeblebrox"
  #Because it was a named group, we can get it by name
  match[:name]  #=> "Zaphod Beeblebrox"
  match["name"] #=> "Zaphod Beeblebrox"
  puts "Hello #{match[:name]}!"
end
```

El índice de la coincidencia se cuenta según el orden de los paréntesis izquierdos (con la expresión regular completa como el primer grupo en el índice 0)

```
reg = /(((a)b)c)(d)/
match = reg.match 'abcd'
match[0] #=> "abcd"
match[1] #=> "abc"
match[2] #=> "ab"
match[3] #=> "a"
match[4] #=> "d"
```

= operador ~

```
if /hay/ =~ 'haystack'
  puts "There is hay in the word haystack"
end
```

Nota: El orden **es significativo** . Aunque `'haystack' =~ /hay/` es en la mayoría de los casos un

equivalente, los efectos secundarios pueden diferir:

- Las cadenas capturadas de los grupos de captura nombrados se asignan a las variables locales solo cuando se llama a `Regexp#=~` (`regexp =~ str`);
- Como el operando correcto podría ser un objeto arbitrario, para `regexp =~ str` se llamará `Regexp#=~` o `String#=~` .

Tenga en cuenta que esto no devuelve un valor verdadero / falso, en su lugar devuelve el índice de la coincidencia si se encuentra, o nil si no se encuentra. Debido a que todos los enteros en ruby son veraces (incluido 0) y nil es falsy, esto funciona. Si desea un valor booleano, use `#===` como se muestra en otro ejemplo .

Cuantificadores

Los cuantificadores permiten especificar el recuento de cadenas repetidas.

- Cero o uno:

```
/a?/
```

- Cero o muchos:

```
/a*/
```

- Uno o muchos:

```
/a+/
```

- Numero exacto:

```
/a{2,4}/ # Two, three or four
/a{2,}/  # Two or more
/a{,4}/  # Less than four (including zero)
```

Por defecto, los cuantificadores son codiciosos , lo que significa que toman la mayor cantidad de caracteres que pueden mientras hacen una coincidencia. Normalmente esto no se nota:

```
/(?<site>.*) Stack Exchange/ =~ 'Motor Vehicle Maintenance & Repair Stack Exchange'
```

El `site` grupo de captura nombrado se configurará en "Mantenimiento y reparación de vehículos de motor" como se esperaba. Pero si 'Stack Exchange' es una parte opcional de la cadena (porque podría ser 'Stack Overflow' en su lugar), la solución ingenua no funcionará como se esperaba:

```
/(?<site>.*)( Stack Exchange)?/
```

Esta versión aún coincidirá, pero la captura nombrada incluirá 'Intercambio de pila' ya que `*` come con avidez esos personajes. La solución es agregar otro signo de interrogación para que el `*`

perezoso:

```
/(?<site>.*?)( Stack Exchange)?/
```

`?` Anexando `?` a cualquier cuantificador lo hará perezoso.

Clases de personajes

Describe rangos de símbolos.

Puedes enumerar símbolos explícitamente

```
/[abc]/ # 'a' or 'b' or 'c'
```

O utilizar rangos

```
/[a-z]/ # from 'a' to 'z'
```

Es posible combinar rangos y símbolos únicos.

```
/[a-cz]/ # 'a' or 'b' or 'c' or 'z'
```

El guión inicial (`-`) se trata como un personaje.

```
/[-a-c]/ # '-' or 'a' or 'b' or 'c'
```

Las clases pueden ser negativas cuando los símbolos anteriores con `^`

```
/[^a-c]/ # Not 'a', 'b' or 'c'
```

Hay algunos accesos directos para clases generalizadas y caracteres especiales, además de finales de línea.

```
^  # Start of line
$  # End of line
\A # Start of string
\Z # End of string, excluding any new line at the end of string
\z # End of string
.  # Any single character
\s # Any whitespace character
\S # Any non-whitespace character
\d # Any digit
\D # Any non-digit
\w # Any word character (letter, number, underscore)
\W # Any non-word character
\b # Any word boundary
```

`\n` será entendido simplemente como nueva línea

Para escapar de cualquier personaje reservado, como `/` o `[]` y otros, utilice la barra invertida

(barra izquierda)

```
\\ # => \
\[\] # => []
```

Expresiones regulares en declaraciones de casos

Puede probar si una cadena coincide con varias expresiones regulares usando una instrucción switch.

Ejemplo

```
case "Ruby is #1!"
when /\APython/
    puts "Boooo."
when /\ARuby/
    puts "You are right."
else
    puts "Sorry, I didn't understand that."
end
```

Esto funciona porque se comprueba la igualdad de las declaraciones de casos utilizando el operador `===` , no el operador `==` . Cuando una expresión regular está en el lado izquierdo de una comparación utilizando `===` , probará una cadena para ver si coincide.

Definiendo un Regexp

Un Regexp se puede crear de tres maneras diferentes en Ruby.

- utilizando barras: `/ /`
- utilizando `%r{}`
- usando `Regex.new`

```
#The following forms are equivalent
regexp_slash = /hello/
regexp_bracket = %r{hello}
regexp_new = Regexp.new('hello')

string_to_match = "hello world!"

#All of these will return a truthy value
string_to_match =~ regexp_slash    # => 0
string_to_match =~ regexp_bracket  # => 0
string_to_match =~ regexp_new      # => 0
```

¿partido? - Resultado booleano

Devuelve `true` o `false` , que indica si la expresión regular coincide o no sin actualizar `$~` y otras variables relacionadas. Si el segundo parámetro está presente, especifica la posición en la

cadena para comenzar la búsqueda.

```
/R.../.match?("Ruby")    #=> true
/R.../.match?("Ruby", 1) #=> false
/P.../.match?("Ruby")    #=> false
```

Ruby 2.4+

Uso rápido común

Las expresiones regulares a menudo se usan en métodos como parámetros para verificar si hay otras cadenas presentes o para buscar y / o reemplazar cadenas.

A menudo verá lo siguiente:

```
string = "My not so long string"
string[/so/] # gives so
string[/present/] # gives nil
string[/present/].nil? # gives true
```

Así que simplemente puedes usar esto como una verificación si una cadena contiene una subcadena

```
puts "found" if string[/so/]
```

Más avanzado pero aún breve y rápido: busque un grupo específico utilizando el segundo parámetro, 2 es el segundo en este ejemplo porque la numeración comienza en 1 y no en 0, un grupo es lo que está entre paréntesis.

```
string[/(n.t).+(l.ng)/, 2] # gives long
```

También se usa a menudo: buscar y reemplazar con `sub` o `gsub` , `\1` da el primer grupo encontrado, `\2` el segundo:

```
string.gsub(/(n.t).+(l.ng)/, '\1 very \2') # My not very long string
```

El último resultado se recuerda y se puede utilizar en las siguientes líneas

```
$2 # gives long
```

Lea Expresiones regulares y operaciones basadas en expresiones regulares en línea: https://riptutorial.com/es/ruby/topic/1357/expresiones-regulares-y-operaciones-basadas-en-expresiones-regulares

Capítulo 28: Extensiones C

Examples

Tu primera extension

Las extensiones en C se componen de dos piezas generales:

1. El código C en sí.
2. El archivo de configuración de extensión.

Para comenzar con su primera extensión, coloque lo siguiente en un archivo llamado `extconf.rb` :

```
require 'mkmf'

create_makefile('hello_c')
```

Un par de cosas para señalar:

Primero, el nombre `hello_c` es el nombre de la salida de la extensión compilada. Será lo que uses junto con el `require` .

En segundo lugar, el archivo `extconf.rb` puede llamarse de cualquier manera, simplemente es lo que tradicionalmente se usa para construir gemas que tienen código nativo, el archivo que realmente compilará la extensión es el Makefile generado al ejecutar `ruby extconf.rb` . El archivo Makefile predeterminado que se genera compila todos los archivos `.c` en el directorio actual.

Coloque lo siguiente en un archivo llamado `hello.c` y ejecute `ruby extconf.rb && make`

```
#include <stdio.h>
#include "ruby.h"

VALUE world(VALUE self) {
  printf("Hello World!\n");
  return Qnil;
}

// The initialization method for this module
void Init_hello_c() {
  VALUE HelloC = rb_define_module("HelloC");
  rb_define_singleton_method(HelloC, "world", world, 0);
}
```

Un desglose del código:

El nombre `Init_hello_c` debe coincidir con el nombre definido en su archivo `extconf.rb` , de lo contrario, al cargar dinámicamente la extensión, Ruby no podrá encontrar el símbolo para iniciar su extensión.

La llamada a `rb_define_module` está creando un módulo Ruby llamado `HelloC` cual vamos a poner

un espacio de nombre bajo nuestras funciones en C.

Finalmente, la llamada a `rb_define_singleton_method` hace un método de nivel de módulo vinculado directamente al módulo `HelloC` que podemos invocar desde ruby con `HelloC.world` .

Después de haber compilado la extensión con la llamada para `make` que podamos ejecutar el código en nuestra extensión C.

¡Enciende una consola!

```
irb(main):001:0> require './hello_c'
=> true
irb(main):002:0> HelloC.world
Hello World!
=> nil
```

Trabajando con C Structs

Para poder trabajar con C structs como objetos Ruby, necesita envolverlos con llamadas a `Data_Wrap_Struct` y `Data_Get_Struct` .

`Data_Wrap_Struct` envuelve una estructura de datos C en un objeto Ruby. Toma un puntero a su estructura de datos, junto con algunos punteros a las funciones de devolución de llamada, y devuelve un valor. La macro `Data_Get_Struct` toma ese VALOR y le devuelve un puntero a su estructura de datos C.

Aquí hay un ejemplo simple:

```
#include <stdio.h>
#include <ruby.h>

typedef struct example_struct {
  char *name;
} example_struct;

void example_struct_free(example_struct * self) {
  if (self->name != NULL) {
    free(self->name);
  }
  ruby_xfree(self);
}

static VALUE rb_example_struct_alloc(VALUE klass) {
  return Data_Wrap_Struct(klass, NULL, example_struct_free,
ruby_xmalloc(sizeof(example_struct)));
}

static VALUE rb_example_struct_init(VALUE self, VALUE name) {
  example_struct* p;

  Check_Type(name, T_STRING);

  Data_Get_Struct(self, example_struct, p);
  p->name = (char *)malloc(RSTRING_LEN(name) + 1);
  memcpy(p->name, StringValuePtr(name), RSTRING_LEN(name) + 1);
```

```
  return self;
}

static VALUE rb_example_struct_name(VALUE self) {
  example_struct* p;
  Data_Get_Struct(self, example_struct, p);

  printf("%s\n", p->name);

  return Qnil;
}

void Init_example()
{
  VALUE mExample = rb_define_module("Example");
  VALUE cStruct = rb_define_class_under(mExample, "Struct", rb_cObject);

  rb_define_alloc_func(cStruct, rb_example_struct_alloc);
  rb_define_method(cStruct, "initialize", rb_example_struct_init, 1);
  rb_define_method(cStruct, "name", rb_example_struct_name, 0);
}
```

Y el `extconf.rb` :

```
require 'mkmf'

create_makefile('example')
```

Después de compilar la extensión:

```
irb(main):001:0> require './example'
=> true
irb(main):002:0> test_struct = Example::Struct.new("Test Struct")
=> #<Example::Struct:0x007fc741965068>
irb(main):003:0> test_struct.name
Test Struct
=> nil
```

Escritura en línea C - Ruby en línea

RubyInline es un marco que te permite incrustar otros idiomas dentro de tu código de Ruby. Define el método en línea del Módulo #, que devuelve un objeto constructor. Le pasas al constructor una cadena que contiene un código escrito en un idioma que no es Ruby, y el constructor lo transforma en algo que puedes llamar desde Ruby.

Cuando se le da un código C o C ++ (los dos idiomas admitidos en la instalación predeterminada de RubyInline), los objetos del constructor escriben una pequeña extensión en el disco, la compilan y la cargan. No tiene que lidiar con la compilación usted mismo, pero puede ver el código generado y las extensiones compiladas en el subdirectorio .ruby_inline de su directorio de inicio.

Incruste el código C directamente en su programa Ruby:

- RubyInline (disponible como la gema rubyinline) crea una extensión automáticamente

  RubyInline no funcionará desde dentro de irb

```
#!/usr/bin/ruby -w
    # copy.rb
    require 'rubygems'
    require 'inline'

    class Copier
    inline do |builder|
      builder.c <<END
    void copy_file(const char *source, const char *dest)
    {
      FILE *source_f = fopen(source, "r");
      if (!source_f)
      {
        rb_raise(rb_eIOError, "Could not open source : '%s'", source);
      }

      FILE *dest_f = fopen(dest, "w+");
      if (!dest_f)
      {
        rb_raise(rb_eIOError, "Could not open destination : '%s'", dest);
      }

      char buffer[1024];

      int nread = fread(buffer, 1, 1024, source_f);
      while (nread > 0)
      {
        fwrite(buffer, 1, nread, dest_f);
        nread = fread(buffer, 1, 1024, source_f);
      }
    }
    END
     end
    end
```

La función C `copy_file` ahora existe como un método de instancia de `Copier` :

```
open('source.txt', 'w') { |f| f << 'Some text.' }
Copier.new.copy_file('source.txt', 'dest.txt')
puts open('dest.txt') { |f| f.read }
```

Lea Extensiones C en línea: https://riptutorial.com/es/ruby/topic/5009/extensiones-c

Capítulo 29: Fecha y hora

Sintaxis

- DateTime.new (año, mes, día, hora, minuto, segundo)

Observaciones

Antes de utilizar DateTime, debe `require 'date'`

Examples

DateTime de cadena

`DateTime.parse` es un método muy útil que construye un DateTime a partir de una cadena, adivinando su formato.

```
DateTime.parse('Jun, 8 2016')
# => #<DateTime: 2016-06-08T00:00:00+00:00 ((2457548j,0s,0n),+0s,2299161j)>
DateTime.parse('201603082330')
# => #<DateTime: 2016-03-08T23:30:00+00:00 ((2457456j,84600s,0n),+0s,2299161j)>
DateTime.parse('04-11-2016 03:50')
# => #<DateTime: 2016-11-04T03:50:00+00:00 ((2457697j,13800s,0n),+0s,2299161j)>
DateTime.parse('04-11-2016 03:50 -0300')
# => #<DateTime: 2016-11-04T03:50:00-03:00 ((2457697j,24600s,0n),-10800s,2299161j)
```

Nota: Hay muchos otros formatos que `parse` reconoce.

Nuevo

```
DateTime.new(2014,10,14)
# => #<DateTime: 2014-10-14T00:00:00+00:00 ((2456945j,0s,0n),+0s,2299161j)>
```

Tiempo actual:

```
DateTime.now
# => #<DateTime: 2016-08-04T00:43:58-03:00 ((2457605j,13438s,667386397n),-10800s,2299161j)>
```

Tenga en cuenta que le da la hora actual en su zona horaria

Añadir / restar días a DateTime

`DateTime` + `Fixnum` (cantidad de días)

```
DateTime.new(2015,12,30,23,0) + 1
# => #<DateTime: 2015-12-31T23:00:00+00:00 ((2457388j,82800s,0n),+0s,2299161j)>
```

`DateTime` + `Float` (cantidad de días)

```
DateTime.new(2015,12,30,23,0) + 2.5
# => #<DateTime: 2016-01-02T11:00:00+00:00 ((2457390j,39600s,0n),+0s,2299161j)>
```

`DateTime` + `Rational` (cantidad de días)

```
DateTime.new(2015,12,30,23,0) + Rational(1,2)
# => #<DateTime: 2015-12-31T11:00:00+00:00 ((2457388j,39600s,0n),+0s,2299161j)>
```

`DateTime` - `Fixnum` (cantidad de días)

```
DateTime.new(2015,12,30,23,0) - 1
# => #<DateTime: 2015-12-29T23:00:00+00:00 ((2457388j,82800s,0n),+0s,2299161j)>
```

`DateTime` - `Float` (cantidad de días)

```
DateTime.new(2015,12,30,23,0) - 2.5
# => #<DateTime: 2015-12-28T11:00:00+00:00 ((2457385j,39600s,0n),+0s,2299161j)>
```

`DateTime` - `Rational` (cantidad de días)

```
DateTime.new(2015,12,30,23,0) - Rational(1,2)
# => #<DateTime: 2015-12-30T11:00:00+00:00 ((2457387j,39600s,0n),+0s,2299161j)>
```

Lea Fecha y hora en línea: https://riptutorial.com/es/ruby/topic/5696/fecha-y-hora

Capítulo 30: Flujo de control

Examples

si, elsif, else y end

Ruby ofrece las expresiones `if` y `else` esperadas para la lógica de bifurcación, terminadas por la palabra clave `end` :

```
# Simulate flipping a coin
result = [:heads, :tails].sample

if result == :heads
  puts 'The coin-toss came up "heads"'
else
  puts 'The coin-toss came up "tails"'
end
```

En Ruby, `if` las declaraciones son expresiones que se evalúan como un valor, y el resultado se puede asignar a una variable:

```
status = if age < 18
           :minor
         else
           :adult
         end
```

Ruby también ofrece operadores ternarios de estilo C (consulte aquí para obtener detalles) que se pueden expresar como:

```
some_statement ? if_true : if_false
```

Esto significa que el ejemplo anterior usando if-else también puede escribirse como

```
status = age < 18 ? :minor : :adult
```

Además, Ruby ofrece la palabra clave `elsif` que acepta una expresión para habilitar lógica de bifurcación adicional:

```
label = if shirt_size == :s
          'small'
        elsif shirt_size == :m
          'medium'
        elsif shirt_size == :l
          'large'
        else
          'unknown size'
        end
```

Si ninguna de las condiciones en una cadena `if` / `elsif` es verdadera, y no hay `else` cláusula, entonces la expresión se evalúa como nula. Esto puede ser útil dentro de la interpolación de cadenas, ya que `nil.to_s` es la cadena vacía:

```
"user#{'s' if @users.size != 1}"
```

Valores de verdad y falsedad.

En Ruby, hay exactamente dos valores que se consideran "falsos" y devolverán falso cuando se analicen como una condición para una expresión `if` . Son:

- `nil`
- booleano `false`

Todos los demás valores se consideran "veraces", incluidos:

- `0` - cero numérico (entero o no)
- `""` - Cuerdas vacías
- `"\n"` - Cadenas que contienen solo espacios en blanco
- `[]` - Arreglos vacíos
- `{}` - Hashes vacíos

Tomemos, por ejemplo, el siguiente código:

```
def check_truthy(var_name, var)
  is_truthy = var ? "truthy" : "falsy"
  puts "#{var_name} is #{is_truthy}"
end

check_truthy("false", false)
check_truthy("nil", nil)
check_truthy("0", 0)
check_truthy("empty string", "")
check_truthy("\\n", "\n")
check_truthy("empty array", [])
check_truthy("empty hash", {})
```

Saldrá:

```
false is falsy
nil is falsy
0 is truthy
empty string is truthy
\n is truthy
empty array is truthy
empty hash is truthy
```

mientras, hasta

A `while` bucle se ejecuta el bloque, mientras que se cumple la condición dada:

```
i = 0
while i < 5
  puts "Iteration ##{i}"
  i +=1
end
```

Un bucle `until` ejecuta el bloque mientras que el condicional es falso:

```
i = 0
until i == 5
  puts "Iteration ##{i}"
  i +=1
end
```

En línea si / a menos

Un patrón común es usar una línea o cola, `if` o a `unless` :

```
puts "x is less than 5" if x < 5
```

Esto se conoce como un *modificador* condicional y es una forma útil de agregar código de guarda simple y devoluciones tempranas:

```
def save_to_file(data, filename)
  raise "no filename given" if filename.empty?
  return false unless data.valid?

  File.write(filename, data)
end
```

No es posible agregar una cláusula `else` a estos modificadores. Además, generalmente no se recomienda usar modificadores condicionales dentro de la lógica principal. Para código complejo, uno debería usar normal `if` , `elsif` , `else` lugar.

a no ser que

Una declaración común es `if !(some condition)` . Ruby ofrece la alternativa de la declaración `unless` .

La estructura es exactamente la misma que una instrucción `if` , excepto que la condición es negativa. Además, la declaración `unless` no es compatible con `elsif` , pero sí es compatible con `else` :

```
# Prints not inclusive
unless 'hellow'.include?('all')
  puts 'not inclusive'
end
```

Declaración del caso

Ruby usa la palabra clave del `case` para las declaraciones de cambio.

Según los documentos de Ruby :

> Las declaraciones de casos consisten en una condición opcional, que está en la posición de un argumento a `case` , y cero o más `when` cláusulas. La primera cláusula `when` para que coincida con la condición (o para evaluar la verdad booleana, si la condición es nula) "gana" y se ejecuta su stanza de código. El valor de la declaración de caso es el valor de la cláusula `when` éxito, o `nil` si no existe tal cláusula.
>
> Una declaración de caso puede terminar con una cláusula `else` . Cada una `when` una declaración puede tener múltiples valores candidatos, separados por comas.

Ejemplo:

```
case x
when 1,2,3
  puts "1, 2, or 3"
when 10
  puts "10"
else
  puts "Some other number"
end
```

Versión más corta:

```
case x
when 1,2,3 then puts "1, 2, or 3"
when 10 then puts "10"
else puts "Some other number"
end
```

El valor del `case` cláusula se empareja con cada `when` cláusula usando el `===` método (no `==`). Por lo tanto, se puede utilizar con una variedad de diferentes tipos de objetos.

Una declaración de `case` se puede utilizar con rangos :

```
case 17
when 13..19
  puts "teenager"
end
```

Se puede usar una declaración de `case` con un Regexp :

```
case "google"
when /oo/
  puts "word contains oo"
end
```

Se puede usar una declaración de `case` con un Proc o lambda:

```
case 44
```

```
when -> (n) { n.even? or n < 0 }
  puts "even or less than zero"
end
```

Se puede usar una declaración de `case` con las Clases :

```
case x
when Integer
  puts "It's an integer"
when String
  puts "It's a string"
end
```

Al implementar el método `===` puede crear sus propias clases de coincidencia:

```
class Empty
  def self.===(object)
    !object or "" == object
  end
end

case ""
when Empty
  puts "name was empty"
else
  puts "name is not empty"
end
```

Se puede usar una declaración de `case` sin que coincida un valor con:

```
case
when ENV['A'] == 'Y'
  puts 'A'
when ENV['B'] == 'Y'
  puts 'B'
else
  puts 'Neither A nor B'
end
```

Una declaración de `case` tiene un valor, por lo que puede usarla como un argumento de método o en una asignación:

```
description = case 16
              when 13..19 then "teenager"
              else ""
              end
```

Control de bucle con ruptura, siguiente y rehacer

El flujo de ejecución de un bloque de Ruby se puede controlar con las declaraciones `break` , `next` y `redo` .

`break`

La sentencia `break` saldrá del bloque inmediatamente. Cualquier instrucción restante en el bloque se omitirá, y la iteración terminará:

```
actions = %w(run jump swim exit macarena)
index = 0

while index < actions.length
  action = actions[index]

  break if action == "exit"

  index += 1
  puts "Currently doing this action: #{action}"
end

# Currently doing this action: run
# Currently doing this action: jump
# Currently doing this action: swim
```

next

La `next` declaración volverá a la parte superior del bloque inmediatamente y continuará con la siguiente iteración. Cualquier instrucción restante en el bloque será omitida:

```
actions = %w(run jump swim rest macarena)
index = 0

while index < actions.length
  action = actions[index]
  index += 1

  next if action == "rest"

  puts "Currently doing this action: #{action}"
end

# Currently doing this action: run
# Currently doing this action: jump
# Currently doing this action: swim
# Currently doing this action: macarena
```

redo

La instrucción de `redo` volverá a la parte superior del bloque inmediatamente y volverá a intentar la misma iteración. Cualquier instrucción restante en el bloque será omitida:

```
actions = %w(run jump swim sleep macarena)
index = 0
repeat_count = 0

while index < actions.length
  action = actions[index]
  puts "Currently doing this action: #{action}"

  if action == "sleep"
    repeat_count += 1
```

```
    redo if repeat_count < 3
  end

  index += 1
end

# Currently doing this action: run
# Currently doing this action: jump
# Currently doing this action: swim
# Currently doing this action: sleep
# Currently doing this action: sleep
# Currently doing this action: sleep
# Currently doing this action: macarena
```

`Enumerable` enumerable

Además de los bucles, estas declaraciones funcionan con los métodos de iteración Enumerable, como `each` y el `map` :

```
[1, 2, 3].each do |item|
  next if item.even?
  puts "Item: #{item}"
end

# Item: 1
# Item: 3
```

Valores de bloque de resultados

Tanto en la declaración de `break` como en la `next` , se puede proporcionar un valor, y se usará como un valor de resultado de bloque:

```
even_value = for value in [1, 2, 3]
  break value if value.even?
end

puts "The first even value is: #{even_value}"

# The first even value is: 2
```

lanzar

A diferencia de muchos otros lenguajes de programación, las palabras clave de `throw` y `catch` no están relacionadas con el manejo de excepciones en Ruby.

En Ruby, `throw` y `catch` actúan un poco como etiquetas en otros idiomas. Se utilizan para cambiar el flujo de control, pero no están relacionados con un concepto de "error", como lo son las excepciones.

```
catch(:out) do
  catch(:nested) do
```

```
    puts "nested"
  end

  puts "before"
  throw :out
  puts "will not be executed"
end
puts "after"
# prints "nested", "before", "after"
```

Flujo de control con sentencias lógicas.

Si bien puede parecer contrario a la intuición, puede usar operadores lógicos para determinar si se ejecuta una declaración. Por ejemplo:

```
File.exist?(filename) or STDERR.puts "#{filename} does not exist!"
```

Esto verificará si el archivo existe y solo imprimirá el mensaje de error si no existe. La declaración `or` es perezosa, lo que significa que dejará de ejecutarse una vez que esté seguro de si el valor es verdadero o falso. Tan pronto como se determina que el primer término es verdadero, no hay necesidad de verificar el valor del otro término. Pero si el primer término es falso, debe verificar el segundo término.

Un uso común es establecer un valor predeterminado:

```
glass = glass or 'full' # Optimist!
```

Eso establece el valor del `glass` en 'lleno' si aún no está configurado. Más concisamente, puede utilizar la versión simbólica de `or` :

```
glass ||= 'empty' # Pessimist.
```

También es posible ejecutar la segunda instrucción solo si la primera es falsa:

```
File.exist?(filename) and puts "#{filename} found!"
```

De nuevo, `and` es perezoso, por lo que solo ejecutará la segunda instrucción si es necesario para llegar a un valor.

El operador `or` tiene menor prioridad que `and` . Del mismo modo, `||` tiene menor precedencia que `&&` . Las formas de los símbolos tienen mayor prioridad que las formas de las palabras. Esto es útil para saber cuándo desea mezclar esta técnica con la asignación:

```
a = 1 and b = 2
#=> a==1
#=> b==2
```

```
a = 1 && b = 2; puts a, b
#=> a==2
```

```
#=> b==2
```

Tenga en cuenta que la Guía de estilo Ruby recomienda :

> El `and` y `or` palabras clave están prohibidos. La legibilidad añadida mínima no vale la pena por la alta probabilidad de introducir errores sutiles. Para expresiones booleanas, use siempre `&&` y `||` en lugar. Para control de flujo, use `if` y a `unless` ; `&&` y `||` También son aceptables pero menos claras.

comenzar

El bloque de `begin` es una estructura de control que agrupa varias declaraciones.

```
begin
  a = 7
  b = 6
  a * b
end
```

Un bloque de `begin` devolverá el valor de la última instrucción en el bloque. El siguiente ejemplo devolverá `3` .

```
begin
  1
  2
  3
end
```

El bloque de `begin` es útil para la asignación condicional utilizando el operador `||=` donde se pueden requerir varias declaraciones para devolver un resultado.

```
circumference ||=
  begin
    radius = 7
    tau = Math::PI * 2
    tau * radius
  end
```

También se puede combinar con otras estructuras de bloque, como `rescue` , `ensure` , `while` `if` , a `unless` , etc. para proporcionar un mayor control del flujo del programa.

`Begin` bloques `Begin` no son bloques de código, como `{ ... }` o `do ... end` ; no pueden ser pasados a funciones.

retorno vs. siguiente: retorno no local en un bloque

Considere este fragmento *roto* :

```
def foo
  bar = [1, 2, 3, 4].map do |x|
    return 0 if x.even?
```

```
    x
  end
  puts 'baz'
  bar
end
foo # => 0
```

Se podría esperar que el `return` produzca un valor para la matriz de resultados de bloque del `map` . Entonces, el valor de retorno de `foo` sería `[1, 0, 3, 0]` . En su lugar, **`return` devuelve un valor del método `foo`** . Tenga en cuenta que `baz` no se imprime, lo que significa que la ejecución nunca llegó a esa línea.

`next` con un valor hace el truco. Actúa como un `return` nivel de bloque.

```
def foo
  bar = [1, 2, 3, 4].map do |x|
    next 0 if x.even?
    x
  end
  puts 'baz'
  bar
end
foo # baz
    # => [1, 0, 3, 0]
```

En ausencia de una `return` , el valor devuelto por el bloque es el valor de su última expresión.

Or-Equals / Operador de asignación condicional (|| =)

Ruby tiene un operador o-igual que permite que un valor se asigne a una variable si y solo si esa variable se evalúa como `nil` o `false` .

```
||= # this is the operator that achieves this.
```

este operador con los tubos dobles que representan o y el signo igual que representa la asignación de un valor. Puedes pensar que representa algo como esto:

```
x = x || y
```

Este ejemplo anterior no es correcto. El operador or-equals en realidad representa esto:

```
x || x = y
```

Si `x` evalúa como `nil` o `false` , a `x` se le asigna el valor de `y` , de lo contrario, no se modifica.

Aquí hay un caso práctico de uso del operador or-igual. Imagina que tienes una parte de tu código que se espera que envíe un correo electrónico a un usuario. ¿Qué debe hacer si por cualquier motivo no hay un correo electrónico para este usuario? Podrías escribir algo como esto:

```
if user_email.nil?
```

```
   user_email = "error@yourapp.com"
end
```

Usando el operador or-equals podemos cortar todo este fragmento de código, brindando un control y funcionalidad claros y claros.

```
user_email ||= "error@yourapp.com"
```

En los casos donde `false` es un valor válido, se debe tener cuidado de no anularlo accidentalmente:

```
has_been_run = false
has_been_run ||= true
#=> true

has_been_run = false
has_been_run = true if has_been_run.nil?
#=> false
```

Operador ternario

Ruby tiene un operador ternario (`?:` :), Que devuelve uno de los dos valores en función de si una condición se evalúa como verdadera:

```
conditional ? value_if_truthy : value_if_falsy

value = true
value ? "true" : "false"
#=> "true"

value = false
value ? "true" : "false"
#=> "false"
```

es lo mismo que escribir `if a then b else c end` , aunque se prefiere el ternario

Ejemplos:

```
puts (if 1 then 2 else 3 end) # => 2

puts 1 ? 2 : 3                # => 2

x = if 1 then 2 else 3 end
puts x                        # => 2
```

Operador de flip-flop

El operador flip flop `..` se usa entre dos condiciones en una declaración condicional:

```
(1..5).select do |e|
  e if (e == 2) .. (e == 4)
```

```
end
# => [2, 3, 4]
```

La condición se evalúa como `false` *hasta que* la primera parte se vuelva `true` . Luego se evalúa como `true` *hasta que* la segunda parte se vuelva `true` . Después de eso cambia a `false` otra vez.

Este ejemplo ilustra lo que se está seleccionando:

```
[1, 2, 2, 3, 4, 4, 5].select do |e|
  e if (e == 2) .. (e == 4)
end
# => [2, 2, 3, 4]
```

El operador de flip-flop solo trabaja dentro de ifs (incluyendo a `unless`) y el operador ternario. De lo contrario, se está considerando como el operador de rango.

```
(1..5).select do |e|
  (e == 2) .. (e == 4)
end
# => ArgumentError: bad value for range
```

Puede cambiar de `false` a `true` y hacia atrás varias veces:

```
((1..5).to_a * 2).select do |e|
  e if (e == 2) .. (e == 4)
end
# => [2, 3, 4, 2, 3, 4]
```

Lea Flujo de control en línea: https://riptutorial.com/es/ruby/topic/640/flujo-de-control

Capítulo 31: Generar un número aleatorio

Introducción

Cómo generar un número aleatorio en Ruby.

Observaciones

Alias de Random :: DEFAULT.rand. Esto utiliza un generador de números pseudoaleatorios que se aproxima a la aleatoriedad verdadera

Examples

6 caras mueren

```
   # Roll a 6 sided die, rand(6) returns a number from 0 to 5 inclusive
   dice_roll_result = 1 + rand(6)
```

Generar un número aleatorio desde un rango (inclusive)

```
# ruby 1.92
lower_limit = 1
upper_limit = 6
Random.new.rand(lower_limit..upper_limit) # Change your range operator to suit your needs
```

Lea Generar un número aleatorio en línea: https://riptutorial.com/es/ruby/topic/9626/generar-un-numero-aleatorio

Capítulo 32: Hashes

Introducción

A Hash es una colección similar a un diccionario de claves únicas y sus valores. También llamados arrays asociativos, son similares a Arrays, pero cuando un Array usa enteros como su índice, un Hash le permite usar cualquier tipo de objeto. Usted recupera o crea una nueva entrada en un Hash refiriéndose a su clave.

Sintaxis

- {first_name: "Noel", second_name: "Edmonds"}
- {: first_name => "Noel",: second_name => "Edmonds"}
- {"Nombre" => "Noel", "Segundo nombre" => "Edmonds"}
- {first_key => first_value, second_key => second_value}

Observaciones

Los hash en Ruby asignan claves a valores utilizando una tabla hash.

Cualquier objeto hashable se puede utilizar como claves. Sin embargo, es muy común usar un `Symbol` ya que generalmente es más eficiente en varias versiones de Ruby, debido a la reducción de la asignación de objetos.

```
{ key1: "foo", key2: "baz"  }
```

Examples

Creando un hash

Un hash en Ruby es un objeto que implementa una tabla hash , asignando claves a valores. Ruby admite una sintaxis literal específica para definir hashes utilizando `{}` :

```
my_hash = {}  # an empty hash
grades = { 'Mark' => 15, 'Jimmy' => 10, 'Jack' => 10 }
```

También se puede crear un hash usando el `new` método estándar:

```
my_hash = Hash.new  # any empty hash
my_hash = {}        # any empty hash
```

Los hash pueden tener valores de cualquier tipo, incluidos tipos complejos como matrices, objetos

y otros hashes:

```
mapping = { 'Mark' => 15, 'Jimmy' => [3,4], 'Nika' => {'a' => 3, 'b' => 5} }
mapping['Mark']   # => 15
mapping['Jimmy']  # => [3, 4]
mapping['Nika']   # => {"a"=>3, "b"=>5}
```

También las claves pueden ser de cualquier tipo, incluidas las complejas:

```
mapping = { 'Mark' => 15, 5 => 10, [1, 2] => 9 }
mapping['Mark']  # => 15
mapping[[1, 2]]  # => 9
```

Los símbolos se usan comúnmente como claves hash, y Ruby 1.9 introdujo una nueva sintaxis específicamente para acortar este proceso. Los siguientes hashes son equivalentes:

```
# Valid on all Ruby versions
grades = { :Mark => 15, :Jimmy => 10, :Jack => 10 }
# Valid in Ruby version 1.9+
grades = { Mark: 15, Jimmy: 10, Jack: 10 }
```

El siguiente hash (válido en todas las versiones de Ruby) es *diferente* , porque todas las claves son cadenas:

```
grades = { "Mark" => 15, "Jimmy" => 10, "Jack" => 10 }
```

Si bien se pueden mezclar ambas versiones de sintaxis, se desaconseja lo siguiente.

```
mapping = { :length => 45, width: 10 }
```

Con Ruby 2.2+, hay una sintaxis alternativa para crear un hash con teclas de símbolo (más útil si el símbolo contiene espacios):

```
grades = { "Jimmy Choo": 10, :"Jack Sparrow": 10 }
# => { :"Jimmy Choo" => 10, :"Jack Sparrow" => 10}
```

Valores de acceso

Los valores individuales de un hash se leen y escriben utilizando los métodos `[]` y `[]=` :

```
my_hash = { length: 4, width: 5 }

my_hash[:length] #=> => 4

my_hash[:height] = 9

my_hash #=> {:length => 4, :width => 5, :height => 9 }
```

De forma predeterminada, el acceso a una clave que no se ha agregado al hash devuelve `nil` , lo que significa que siempre es seguro intentar buscar el valor de una clave:

```
my_hash = {}

my_hash[:age] # => nil
```

Los hashes también pueden contener claves en cadenas. Si intenta acceder a ellos normalmente, solo devolverá un valor `nil` , en lugar de eso, acceda a ellos mediante sus claves de cadena:

```
my_hash = { "name" => "user" }

my_hash[:name]    # => nil
my_hash["name"]   # => user
```

Para situaciones en las que se espera o se requiere que existan claves, los hash tienen un método de `fetch` que generará una excepción al acceder a una clave que no existe:

```
my_hash = {}

my_hash.fetch(:age) #=> KeyError: key not found: :age
```

`fetch` acepta un valor predeterminado como su segundo argumento, que se devuelve si la clave no se ha establecido previamente:

```
my_hash =  {}
my_hash.fetch(:age, 45) #=> => 45
```

`fetch` también puede aceptar un bloque que se devuelve si la clave no se ha establecido previamente:

```
my_hash = {}
my_hash.fetch(:age) { 21 } #=> 21

my_hash.fetch(:age) do |k|
  puts "Could not find #{k}"
end

#=> Could not find age
```

Los elementos hash también admiten un método de `store` como un alias para `[]=` :

```
my_hash = {}

my_hash.store(:age, 45)

my_hash #=> { :age => 45 }
```

También puede obtener todos los valores de un hash utilizando el método de `values` :

```
my_hash = { length: 4, width: 5 }

my_hash.values #=> [4, 5]
```

Nota: esto es solo para Ruby `#dig` es útil para `Hash` anidados. Extrae el valor anidado especificado por la secuencia de objetos idx llamando a dig en cada paso, devolviendo nil si algún paso intermedio es nulo.

```
h = { foo: {bar: {baz: 1}}}

h.dig(:foo, :bar, :baz)   # => 1
h.dig(:foo, :zot, :xyz)   # => nil

g = { foo: [10, 11, 12] }
g.dig(:foo, 1)            # => 11
```

Configuración de valores predeterminados

De forma predeterminada, al intentar buscar el valor de una clave que no existe, se devolverá `nil` . Opcionalmente, puede especificar algún otro valor para devolver (o una acción a realizar) cuando se accede al hash con una clave que no existe. Si bien esto se conoce como "el valor predeterminado", no es necesario que sea un valor único; podría, por ejemplo, ser un valor computado como la longitud de la clave.

El valor predeterminado de un hash se puede pasar a su constructor:

```
h = Hash.new(0)

h[:hi] = 1
puts h[:hi]  # => 1
puts h[:bye] # => 0 returns default value instead of nil
```

Un valor predeterminado también se puede especificar en un hash ya construido:

```
my_hash = { human: 2, animal: 1 }
my_hash.default = 0
my_hash[:plant] # => 0
```

Es importante tener en cuenta que el **valor predeterminado no se copia** cada vez que se accede a una nueva clave, lo que puede generar resultados sorprendentes cuando el valor predeterminado es un tipo de referencia:

```
# Use an empty array as the default value
authors = Hash.new([])

# Append a book title
authors[:homer] << 'The Odyssey'

# All new keys map to a reference to the same array:
authors[:plato] # => ['The Odyssey']
```

Para evitar este problema, el constructor de hash acepta un bloque que se ejecuta cada vez que se accede a una nueva clave, y el valor devuelto se utiliza como predeterminado:

```
authors = Hash.new { [] }
```

```
# Note that we're using += instead of <<, see below
authors[:homer] += ['The Odyssey']
authors[:plato] # => []

authors # => {:homer=>["The Odyssey"]}
```

Tenga en cuenta que anteriormente tuvimos que usar + = en lugar de << porque el valor predeterminado no se asigna automáticamente al hash; el uso de << se habría agregado a la matriz, pero los autores [: homer] habrían permanecido indefinidos:

```
authors[:homer] << 'The Odyssey' # ['The Odyssey']
authors[:homer] # => []
authors # => {}
```

Para poder asignar valores predeterminados en el acceso, así como para calcular valores predeterminados más sofisticados, el bloque predeterminado se pasa el hash y la clave:

```
authors = Hash.new { |hash, key| hash[key] = [] }

authors[:homer] << 'The Odyssey'
authors[:plato] # => []

authors # => {:homer=>["The Odyssey"], :plato=>[]}
```

También puede usar un bloque predeterminado para realizar una acción y / o devolver un valor dependiente de la clave (o algún otro dato):

```
chars = Hash.new { |hash,key| key.length }

chars[:test] # => 4
```

Incluso puedes crear hashes más complejos:

```
page_views = Hash.new { |hash, key| hash[key] = { count: 0, url: key } }
page_views["http://example.com"][:count] += 1
page_views # => {"http://example.com"=>{:count=>1, :url=>"http://example.com"}}
```

Para establecer el valor predeterminado en un Proc en un hash *ya existente* , use `default_proc=` :

```
authors = {}
authors.default_proc = proc { [] }

authors[:homer] += ['The Odyssey']
authors[:plato] # => []

authors # {:homer=>["The Odyssey"]}
```

Creando automáticamente un Hash profundo

Hash tiene un valor predeterminado para las claves que se solicitan pero que no existen (nil):

```
a = {}
p a[ :b ] # => nil
```

Al crear un nuevo hash, se puede especificar el valor predeterminado:

```
b = Hash.new 'puppy'
p b[ :b ]            # => 'puppy'
```

Hash.new también toma un bloque, que le permite crear hashes anidados, como el comportamiento de autovivificación de Perl o `mkdir -p` :

```
# h is the hash you're creating, and k the key.
#
hash = Hash.new { |h, k| h[k] = Hash.new &h.default_proc }
hash[ :a ][ :b ][ :c ] = 3

p hash # => { a: { b: { c: 3 } } }
```

Modificación de claves y valores.

Puede crear un nuevo hash con las claves o los valores modificados, de hecho, también puede agregar o eliminar claves, usando inyectar (AKA, reducir). Por ejemplo, para producir un hash con claves de cadena y valores en mayúsculas:

```
fruit = { name: 'apple', color: 'green', shape: 'round' }
# => {:name=>"apple", :color=>"green", :shape=>"round"}

new_fruit = fruit.inject({}) { |memo, (k,v)| memo[k.to_s] = v.upcase; memo }

# => new_fruit is {"name"=>"APPLE", "color"=>"GREEN", "shape"=>"ROUND"}
```

Hash es un enumerable, en esencia, una colección de pares clave / valor. Por eso tiene métodos como `each` , `map` e `inject` .

Para cada par de clave / valor en el hash se evalúa el bloque dado, el valor de memo en la primera ejecución es el valor semilla que se pasa para `inject` , en nuestro caso un hash vacío, `{}` . El valor de `memo` para evaluaciones posteriores es el valor devuelto de la evaluación de bloques anterior, es por eso que modificamos `memo` estableciendo una clave con un valor y luego devolvemos `memo` al final. El valor de retorno de la evaluación final bloques es el valor de retorno de `inject` , en nuestro caso `memo` .

Para evitar tener que proporcionar el valor final, puede utilizar each_with_object en su lugar:

```
new_fruit = fruit.each_with_object({}) { |(k,v), memo| memo[k.to_s] = v.upcase }
```

O incluso mapa :

1.8

```
new_fruit = Hash[fruit.map{ |k,v| [k.to_s, v.upcase] }]
```

(Consulte esta respuesta para obtener más detalles, incluida la forma de manipular los hashes en su lugar).

Iterando sobre un hash

A `Hash` incluye el módulo `Enumerable`, que proporciona varios métodos de iteración, como: `Enumerable#each`, `Enumerable#each_pair`, `Enumerable#each_key`, y `Enumerable#each_value`.

`.each` y `.each_pair` sobre cada par clave-valor:

```
h = { "first_name" => "John", "last_name" => "Doe" }
h.each do |key, value|
    puts "#{key} = #{value}"
end

# => first_name = John
#    last_name = Doe
```

`.each_key` itera sobre las teclas solamente:

```
h = { "first_name" => "John", "last_name" => "Doe" }
h.each_key do |key|
  puts key
end

# => first_name
#    last_name
```

`.each_value` itera sobre los valores solamente:

```
h = { "first_name" => "John", "last_name" => "Doe" }
h.each_value do |value|
    puts value
end

# => John
#    Doe
```

`.each_with_index` itera sobre los elementos y proporciona el índice de la iteración:

```
h = { "first_name" => "John", "last_name" => "Doe" }
h.each_with_index do |(key, value), index|
    puts "index: #{index} | key: #{key} | value: #{value}"
end

# => index: 0 | key: first_name | value: John
#    index: 1 | key: last_name | value: Doe
```

Conversión ay desde matrices

Los hashes se pueden convertir libremente desde y hacia matrices. Convertir un hash de pares clave / valor en una matriz producirá una matriz que contiene matrices anidadas para la pareja:

```
{ :a => 1, :b => 2 }.to_a # => [[:a, 1], [:b, 2]]
```

En la dirección opuesta se puede crear un Hash a partir de una matriz del mismo formato:

```
[[:x, 3], [:y, 4]].to_h # => { :x => 3, :y => 4 }
```

De manera similar, Hashes puede inicializarse usando `Hash[]` y una lista de claves y valores alternos:

```
Hash[:a, 1, :b, 2] # => { :a => 1, :b => 2 }
```

O de una matriz de matrices con dos valores cada una:

```
Hash[ [[:x, 3], [:y, 4]] ] # => { :x => 3, :y => 4 }
```

Los hash se pueden convertir de nuevo a una matriz de claves y valores alternativos utilizando `flatten()` :

```
{ :a => 1, :b => 2 }.flatten # => [:a, 1, :b, 2]
```

La fácil conversión hacia y desde una matriz permite que `Hash` funcione bien con muchos métodos `Enumerable` , como `collect` y `zip` :

```
Hash[('a'..'z').collect{ |c| [c, c.upcase] }] # => { 'a' => 'A', 'b' => 'B', ... }

people = ['Alice', 'Bob', 'Eve']
height = [5.7, 6.0, 4.9]
Hash[people.zip(height)] # => { 'Alice' => 5.7, 'Bob' => '6.0', 'Eve' => 4.9 }
```

Obteniendo todas las claves o valores de hash.

```
{foo: 'bar', biz: 'baz'}.keys   # => [:foo, :biz]
{foo: 'bar', biz: 'baz'}.values # => ["bar", "baz"]
{foo: 'bar', biz: 'baz'}.to_a   # => [[:foo, "bar"], [:biz, "baz"]]
{foo: 'bar', biz: 'baz'}.each   #<Enumerator: {:foo=>"bar", :biz=>"baz"}:each>
```

Anulando la función hash

Rubí hashes utiliza los métodos `hash` y `eql?` para realizar la operación de hash y asignar objetos almacenados en el hash a los hash bins internos. La implementación predeterminada de `hash` en Ruby es la función hash murmur sobre todos los campos miembros del objeto hash . Para anular este comportamiento, es posible anular el `hash` y el `eql?` metodos

Al igual que con otras implementaciones de hash, dos objetos a y b se procesarán en el mismo grupo si `a.hash == b.hash` y se considerarán idénticos si `a.eql?(b)` . ¿Así, al reimplementar `hash` y `eql?` uno debe tener cuidado de asegurarse de que si `a` y `b` son iguales bajo `eql?` Deben devolver el mismo valor `hash` . De lo contrario, esto podría resultar en entradas duplicadas en un hash. Por el contrario, una mala elección en la implementación de `hash` podría llevar a muchos objetos a

compartir el mismo hash bucket, destruyendo efectivamente el tiempo de búsqueda O (1) y causando O (n) para llamar a `eql?` en todos los objetos.

En el siguiente ejemplo, solo la instancia de la clase `A` se almacena como una clave, ya que se agregó primero:

```
class A
  def initialize(hash_value)
    @hash_value = hash_value
  end
  def hash
    @hash_value # Return the value given externally
  end
  def eql?(b)
    self.hash == b.hash
  end
end

class B < A
end

a = A.new(1)
b = B.new(1)

h = {}
h[a] = 1
h[b] = 2

raise "error" unless h.size == 1
raise "error" unless h.include? b
raise "error" unless h.include? a
```

Filtrado de hashes

`select` devuelve un nuevo `hash` con pares clave-valor para los cuales el bloque se evalúa como `true` .

```
{ :a => 1, :b => 2, :c => 3 }.select { |k, v| k != :a && v.even? } # => { :b => 2 }
```

Cuando no necesite la *clave* o el *valor* en un bloque de filtro, la convención es usar un _ en ese lugar:

```
{ :a => 1, :b => 2, :c => 3 }.select { |_, v| v.even? } # => { :b => 2 }
{ :a => 1, :b => 2, :c => 3 }.select { |k, _| k == :c } # => { :c => 3 }
```

`reject` devuelve un nuevo `hash` con pares clave-valor para los cuales el bloque se evalúa como `false` :

```
{ :a => 1, :b => 2, :c => 3 }.reject { |_, v| v.even? } # => { :a => 1, :c => 3 }
{ :a => 1, :b => 2, :c => 3 }.reject { |k, _| k == :b } # => { :a => 1, :c => 3 }
```

Establecer operaciones en Hashes

- **Intersección de Hashes**

  Para obtener la intersección de dos hashes, devuelva las claves compartidas cuyos valores son iguales:

```
hash1 = { :a => 1, :b => 2 }
hash2 = { :b => 2, :c => 3 }
hash1.select { |k, v| (hash2.include?(k) && hash2[k] == v) } # => { :b => 2 }
```

- **Unión (fusión) de hashes:**

  las claves en un hash son únicas, si una clave se produce en ambos hashes que se fusionarán, se sobrescribe la del hash que se llama a `merge` :

```
hash1 = { :a => 1, :b => 2 }
hash2 = { :b => 4, :c => 3 }

hash1.merge(hash2) # => { :a => 1, :b => 4, :c => 3 }
hash2.merge(hash1) # => { :b => 2, :c => 3, :a => 1 }
```

Lea Hashes en línea: https://riptutorial.com/es/ruby/topic/288/hashes

Capítulo 33: Herencia

Sintaxis

- clase Subclase <SuperClase

Examples

Refactorizando las clases existentes para usar la herencia.

Digamos que tenemos dos clases, `Cat` y `Dog` .

```
class Cat
  def eat
    die unless has_food?
    self.food_amount -= 1
    self.hungry = false
  end
  def sound
    puts "Meow"
  end
end

class Dog
  def eat
    die unless has_food?
    self.food_amount -= 1
    self.hungry = false
  end
  def sound
    puts "Woof"
  end
end
```

El método de `eat` es exactamente el mismo en estas dos clases. Si bien esto funciona, es difícil de mantener. El problema empeorará si hay más animales con el mismo método de `eat` . La herencia puede resolver este problema.

```
class Animal
  def eat
    die unless has_food?
    self.food_amount -= 1
    self.hungry = false
  end
  # No sound method
end

class Cat < Animal
  def sound
    puts "Meow"
  end
end
```

```
class Dog < Animal
  def sound
    puts "Woof"
  end
end
```

Hemos creado una nueva clase, `Animal` , y hemos movido nuestro método de `eat` a esa clase. Luego, hicimos que `Cat` y `Dog` heredaran de esta nueva superclase común. Esto elimina la necesidad de repetir el código.

Herencia múltiple

La herencia múltiple es una característica que permite que una clase herede de varias clases (es decir, más de un padre). Ruby no admite herencia múltiple. Solo admite herencia simple (es decir, la clase solo puede tener un padre), pero puede usar la *composición* para crear clases más complejas utilizando módulos .

Subclases

La herencia permite que las clases definan un comportamiento específico basado en una clase existente.

```
class Animal
  def say_hello
    'Meep!'
  end

  def eat
    'Yumm!'
  end
end

class Dog < Animal
  def say_hello
    'Woof!'
  end
end

spot = Dog.new
spot.say_hello # 'Woof!'
spot.eat       # 'Yumm!'
```

En este ejemplo:

- `Dog` hereda de `Animal` , haciéndolo una *subclase* .
- `Dog` gana tanto el `say_hello` como el de `eat` de `Animal` .
- `Dog` reemplaza el método `say_hello` con diferentes funcionalidades.

Mixins

Los mixins son una forma hermosa de lograr algo similar a la herencia múltiple. Nos permite

heredar o, más bien, incluir métodos definidos en un módulo en una clase. Estos métodos pueden incluirse como métodos de instancia o de clase. El siguiente ejemplo muestra este diseño.

```
module SampleModule

  def self.included(base)
    base.extend ClassMethods
  end

  module ClassMethods

    def method_static
      puts "This is a static method"
    end

  end

  def insta_method
    puts "This is an instance method"
  end

end

class SampleClass
  include SampleModule
end

sc = SampleClass.new

sc.insta_method

prints "This is an instance method"

sc.class.method_static

prints "This is a static method"
```

¿Qué se hereda?

Los métodos son heredados

```
class A
  def boo; p 'boo' end
end

class B < A; end

b = B.new
b.boo # => 'boo'
```

Los métodos de clase son heredados.

```
class A
  def self.boo; p 'boo' end
end

class B < A; end
```

```
p B.boo # => 'boo'
```

Las constantes son heredadas

```
class A
  WOO = 1
end

class B < A; end

p B::WOO # => 1
```

Pero cuidado, pueden ser anulados:

```
class B
  WOO = WOO + 1
end

p B::WOO # => 2
```

Las variables de instancia son heredadas:

```
class A
  attr_accessor :ho
  def initialize
    @ho = 'haha'
  end
end

class B < A; end

b = B.new
p b.ho # => 'haha'
```

Tenga cuidado, si anula los métodos que inicializan las variables de instancia sin llamar a `super` , serán nulos. Continuando desde arriba:

```
class C < A
  def initialize; end
 end

c = C.new
p c.ho    # => nil
```

Las variables de instancia de clase no se heredan:

```
class A
    @foo = 'foo'
    class << self
        attr_accessor :foo
    end
end
```

```
class B < A; end

p B.foo # => nil

# The accessor is inherited, since it is a class method
#
B.foo = 'fob' # possible
```

Las variables de clase no son realmente heredadas

Se comparten entre la clase base y todas las subclases como 1 variable:

```
class A
    @@foo = 0
    def initialize
        @@foo  += 1
        p @@foo
    end
end

class B < A;end

a = A.new # => 1
b = B.new # => 2
```

Continuando desde arriba:

```
class C < A
  def initialize
    @@foo = -10
    p @@foo
  end
end

a = C.new # => -10
b = B.new # => -9
```

Lea Herencia en línea: https://riptutorial.com/es/ruby/topic/625/herencia

Capítulo 34: Hilo

Examples

Semántica de hilo básico

Se puede crear un nuevo hilo separado de la ejecución del hilo principal, usando `Thread.new` .

```
thr = Thread.new {
  sleep 1 # 1 second sleep of sub thread
  puts "Whats the big deal"
}
```

Esto iniciará automáticamente la ejecución del nuevo hilo.

Para congelar la ejecución del hilo principal, hasta que el nuevo hilo se detenga, use `join` :

```
thr.join #=> ... "Whats the big deal"
```

Tenga en cuenta que el subproceso ya puede haber finalizado cuando llama a unirse, en cuyo caso la ejecución continuará normalmente. Si un subproceso nunca se une y el subproceso principal se completa, el subproceso no ejecutará ningún código restante.

Acceso a recursos compartidos

Use un mutex para sincronizar el acceso a una variable a la que se accede desde varios subprocesos:

```
counter = 0
counter_mutex = Mutex.new

# Start three parallel threads and increment counter
3.times.map do |index|
  Thread.new do
    counter_mutex.synchronize { counter += 1 }
  end
end.each(&:join) # Wait for all threads to finish before killing the process
```

De lo contrario, el valor del `counter` actualmente visible para un hilo podría ser cambiado por otro hilo.

Ejemplo **sin** `Mutex` (ver, por ejemplo, el `Thread 0` , donde `Before` y `After` difieren en más de `1`):

```
2.2.0 :224 > counter = 0; 3.times.map { |i| Thread.new { puts "[Thread #{i}] Before:
#{counter}"; counter += 1; puts "[Thread #{i}] After: #{counter}"; } }.each(&:join)
[Thread 2] Before: 0
[Thread 0] Before: 0
[Thread 0] After: 2
[Thread 1] Before: 0
```

```
[Thread 1] After: 3
[Thread 2] After: 1
```

Ejemplo **con** `Mutex` :

```
2.2.0 :226 > mutex = Mutex.new; counter = 0; 3.times.map { |i| Thread.new { mutex.synchronize
{ puts "[Thread #{i}] Before: #{counter}"; counter += 1; puts "[Thread #{i}] After:
#{counter}"; } } }.each(&:join)
[Thread 2] Before: 0
[Thread 2] After: 1
[Thread 1] Before: 1
[Thread 1] After: 2
[Thread 0] Before: 2
[Thread 0] After: 3
```

Cómo matar un hilo

Usted llama usar `Thread.kill` o `Thread.terminate` :

```
thr = Thread.new { ... }
Thread.kill(thr)
```

Terminando un hilo

Un hilo termina si llega al final de su bloque de código. La mejor manera de terminar un hilo temprano es convencerlo de que llegue al final de su bloque de código. De esta manera, el hilo puede ejecutar código de limpieza antes de morir.

Este hilo ejecuta un bucle mientras que la variable de instancia continue es verdadera. Establece esta variable en falso, y el hilo morirá una muerte natural:

```
require 'thread'

class CounterThread < Thread
  def initialize
    @count = 0
    @continue = true

    super do
      @count += 1 while @continue
      puts "I counted up to #{@count} before I was cruelly stopped."
    end
  end

  def stop
    @continue = false
  end
end

counter = CounterThread.new
sleep 2
counter.stop
```

Lea Hilo en línea: https://riptutorial.com/es/ruby/topic/995/hilo

Capítulo 35: Hora

Sintaxis

- `Time.now`
- `Time.new([year], [month], [day], [hour], [min], [sec], [utc_offset])`

Examples

Cómo utilizar el método de tiempo de guerra

Convertir una hora en una cadena es algo muy común en Ruby. `strftime` es el método que se usaría para convertir el tiempo en una cadena.

Aquí hay unos ejemplos:

```
Time.now.strftime("%Y-%m-d %H:%M:S") #=> "2016-07-27 08:45:42"
```

Esto se puede simplificar aún más

```
Time.now.strftime("%F %X")  #=> "2016-07-27 08:45:42"
```

Creando objetos de tiempo

Obtener hora actual:

```
Time.now
Time.new # is equivalent if used with no parameters
```

Obtener tiempo específico:

```
Time.new(2010, 3, 10) #10 March 2010 (Midnight)
Time.new(2015, 5, 3, 10, 14) #10:14 AM on 3 May 2015
Time.new(2050, "May", 3, 21, 8, 16, "+10:00") #09:08:16 PM on 3 May 2050
```

Para convertir un tiempo a la época puedes usar el método `to_i` :

```
Time.now.to_i # => 1478633386
```

También puede convertir de nuevo de época a Tiempo usando el método `at` :

```
Time.at(1478633386) # => 2016-11-08 17:29:46 -0200
```

Lea Hora en línea: https://riptutorial.com/es/ruby/topic/4346/hora

Capítulo 36: Instalación

Examples

Linux - Compilación desde la fuente

De esta manera obtendrás el rubí más nuevo, pero tiene sus desventajas. Hacerlo de esta manera no será gestionado por ninguna aplicación.

!! Recuerda chagne la versión para que coincida con tu !!

1. necesita descargar un archivo comprimido para encontrar un enlace en un sitio web oficial (https://www.ruby-lang.org/en/downloads/)
2. Extraer el tarball
3. Instalar

```
$> wget https://cache.ruby-lang.org/pub/ruby/2.3/ruby-2.3.3.tar.gz
$> tar -xvzf ruby-2.3.3.tar.gz
$> cd ruby-2.3.3
$> ./configure
$> make
$> sudo make install
```

Esto instalará ruby en `/usr/local` . Si no está satisfecho con esta ubicación, puede pasar un argumento a `./configure --prefix=DIR` donde `DIR` es el directorio en el que desea instalar ruby.

Linux: instalación mediante un gestor de paquetes

Probablemente la opción más fácil, pero cuidado, la versión no siempre es la más nueva. Solo abre la terminal y escribe (dependiendo de tu distribución)

en Debian o Ubuntu usando apt

```
$> sudo apt install ruby
```

en CentOS, openSUSE o Fedora

```
$> sudo yum install ruby
```

Puede usar la opción `-y` para que no se le pida que acepte la instalación, pero en mi opinión, es una buena práctica verificar siempre qué es lo que el administrador de paquetes está intentando instalar.

Windows - Instalación mediante instalador

Probablemente la forma más fácil de configurar Ruby en Windows es ir a http://rubyinstaller.org/ y desde allí descargar un archivo ejecutable que instalarás.

No tiene que configurar casi nada, pero habrá una ventana importante. Tendrá una casilla de verificación que dice *Agregar ejecutable de ruby a su RUTA* . Confirme que esté **marcado** , si no lo hace, de lo contrario no podrá ejecutar ruby y tendrá que establecer la variable PATH por su cuenta.

Luego solo ve a continuación hasta que se instale y eso es todo.

Gemas

En este ejemplo usaremos 'nokogiri' como ejemplo de gema. 'nokogiri' puede ser reemplazado posteriormente por cualquier otro nombre de gema.

Para trabajar con gemas usamos una herramienta de línea de comandos llamada `gem` seguida de una opción como `install` o `update` y luego los nombres de las gemas que queremos instalar, pero eso no es todo.

Instala gemas:

```
$> gem install nokogiri
```

Pero eso no es lo único que necesitamos. También podemos especificar la versión, la fuente desde la cual instalar o buscar gemas. Comencemos con algunos casos de uso básicos (UC) y luego puede solicitar una actualización más tarde.

Listado de todas las gemas instaladas:

```
$> gem list
```

Desinstalando gemas:

```
$> gem uninstall nokogiri
```

Si tenemos más versiones de la gema nokogiri, se nos pedirá que especifiquemos cuál queremos desinstalar. Obtendremos una lista ordenada y numerada y solo escribiremos el número.

Actualizando gemas

```
$> gem update nokogiri
```

O si queremos actualizarlos todos.

```
$> gem update
```

La `gem` Comman tiene muchos más usos y opciones para explorar. Para más información, por favor consulte la documentación oficial. Si algo no está claro publica una solicitud y la añadiré.

Linux - Solucionar problemas de instalación de gem

First UC en el ejemplo **Gems** `$> gem install nokogiri` puede tener problemas para instalar gemas porque no tenemos los permisos para ello. Esto se puede resolver de más de una manera.

Primera solución UC a:

U puede usar `sudo` . Esto instalará la gema para todos los usuarios. Este método debe ser mal visto. Esto se debe usar solo con la gema que sabe que todos los usuarios podrán utilizar. Usualmente en la vida real no quieres que algún usuario tenga acceso a `sudo` .

```
$> sudo gem install nokogiri
```

Primera solución UC b

U puede usar la opción - `--user-install` que instala las gemas en su carpeta de gemas de usuarios (generalmente en `~/.gem`)

```
&> gem install nokogiri --user-install
```

Primera solución UC c

U puede configurar GEM_HOME y GEM_PATH que luego harán que el comando `gem install` instale todas las gemas en una carpeta que usted especifique. Te puedo dar un ejemplo de eso (la forma habitual)

- En primer lugar necesitas abrir .bashrc. Utilice nano o su editor de texto favorito.

```
$> nano ~/.bashrc
```

- Luego al final de este archivo escribe

```
export GEM_HOME=$HOME/.gem
export GEM_PATH=$HOME/.gem
```

- Ahora tendrás que reiniciar el terminal o escribir . `~/.bashrc` para volver a cargar la configuración. Esto te permitirá usar `gem isntall nokogiri` e instalará esas gemas en la carpeta que especificaste.

Instalando Ruby MacOS

Así que la buena noticia es que Apple amablemente incluye un intérprete Ruby. Desafortunadamente, tiende a no ser una versión reciente:

```
$ /usr/bin/ruby -v
ruby 2.0.0p648 (2015-12-16 revision 53162) [universal.x86_64-darwin16]
```

Si tienes Homebrew instalado , puedes obtener la última versión de Ruby con:

```
$ brew install ruby
```

```
$ /usr/local/bin/ruby -v
ruby 2.4.1p111 (2017-03-22 revision 58053) [x86_64-darwin16]
```

(Es probable que veas una versión más reciente si lo intentas).

Para elegir la versión elaborada sin utilizar la ruta completa, querrá agregar `/usr/local/bin` al inicio de su `$PATH` entorno `$PATH` :

```
export PATH=/usr/local/bin:$PATH
```

Agregar esa línea a `~/.bash_profile` asegura que obtendrás esta versión después de reiniciar tu sistema:

```
$ type ruby
ruby is /usr/local/bin/ruby
```

Homebrew instalará `gem` para instalar gemas . También es posible construir desde la fuente si lo necesitas. Homebrew también incluye esa opción:

```
$ brew install ruby --build-from-source
```

Lea Instalación en línea: https://riptutorial.com/es/ruby/topic/8095/instalacion

Capítulo 37: instancia_eval

Sintaxis

- object.instance_eval 'código'
- object.instance_eval 'código', 'nombre de archivo'
- object.instance_eval 'código', 'nombre de archivo', 'número de línea'
- object.instance_eval {code}
- object.instance_eval {| receiver | código}

Parámetros

Parámetro	Detalles
`string`	Contiene el código fuente de Ruby para ser evaluado.
`filename`	Nombre de archivo a usar para reportar errores.
`lineno`	Número de línea a utilizar para la notificación de errores.
`block`	El bloque de código a evaluar.
`obj`	El receptor se pasa al bloque como su único argumento.

Examples

Evaluación de instancias

El método `instance_eval` está disponible en todos los objetos. Evalúa el código en el contexto del receptor:

```
object = Object.new

object.instance_eval do
  @variable = :value
end

object.instance_variable_get :@variable # => :value
```

`instance_eval` establece `self` para `object` durante la duración del bloque de código:

```
object.instance_eval { self == object } # => true
```

El receptor también se pasa al bloque como su único argumento:

```
object.instance_eval { |argument| argument == object } # => true
```

El método `instance_exec` difiere a este respecto: en su lugar, pasa sus argumentos al bloque.

```
object.instance_exec :@variable do |name|
  instance_variable_get name # => :value
end
```

Implementando con

Muchos lenguajes cuentan `with` una declaración `with` que los programadores pueden omitir el receptor de llamadas a métodos.

`with` puede ser fácilmente emulado en Ruby usando `instance_eval` :

```
def with(object, &block)
  object.instance_eval &block
end
```

El método `with` se puede usar para ejecutar métodos sin problemas en objetos:

```
hash = Hash.new

with hash do
  store :key, :value
  has_key? :key       # => true
  values              # => [:value]
end
```

Lea instancia_eval en línea: https://riptutorial.com/es/ruby/topic/5049/instancia-eval

Capítulo 38: Instrumentos de cuerda

Sintaxis

- 'Una cadena' // crea una cadena a través de un literal entre comillas simples
- "Una cadena" // crea una cadena a través de un literal entre comillas dobles
- String.new ("Una cadena")
- % q (una cadena) // sintaxis alternativa para crear cadenas entre comillas simples
- % Q (una cadena) // sintaxis alternativa para crear cadenas entre comillas dobles

Examples

Diferencia entre literales de cadena entre comillas simples y comillas dobles

La principal diferencia es que los literales de `String` comillas dobles admiten interpolaciones de cadenas y el conjunto completo de secuencias de escape.

Por ejemplo, pueden incluir expresiones de Ruby arbitrarias a través de la interpolación:

```
# Single-quoted strings don't support interpolation
puts 'Now is #{Time.now}'
# Now is #{Time.now}

# Double-quoted strings support interpolation
puts "Now is #{Time.now}"
# Now is 2016-07-21 12:43:04 +0200
```

Las cadenas entre comillas dobles también admiten el conjunto completo de secuencias de escape, incluidos `"\n"` , `"\t"` ...

```
puts 'Hello\nWorld'
# Hello\nWorld

puts "Hello\nWorld"
# Hello
# World
```

... mientras que las cadenas de comillas simples *no* admiten secuencias de escape, muestra el conjunto mínimo necesario para que las cadenas de comillas simples sean útiles: comillas simples literales y barras invertidas, `'\''` y `'\\'` respectivamente.

Creando una cadena

Ruby proporciona varias formas de crear un objeto `String` . La forma más común es usar comillas simples o dobles para crear una " cadena literal ":

```
s1 = 'Hello'
```

```
s2 = "Hello"
```

La principal diferencia es que los literales de cadena entre comillas dobles son un poco más flexibles ya que admiten la interpolación y algunas secuencias de escape de barra invertida.

También hay varias otras formas posibles de crear un literal de cadena utilizando delimitadores de cadena arbitrarios. Un delimitador de cadena arbitrario es un `%` seguido por un par de delimitadores coincidentes:

```
%(A string)
%{A string}
%<A string>
%|A string|
%!A string!
```

Finalmente, puede usar las secuencias `%q` y `%Q` , que son equivalentes a `'` y `"` ":

```
puts %q(A string)
# A string
puts %q(Now is #{Time.now})
# Now is #{Time.now}

puts %Q(A string)
# A string
puts %Q(Now is #{Time.now})
# Now is 2016-07-21 12:47:45 +0200
```

`%Q` secuencias `%q` y `%Q` son útiles cuando la cadena contiene comillas simples, comillas dobles o una combinación de ambas. De esta manera, no necesitas escapar del contenido:

```
%Q(<a href="/profile">User's profile<a>)
```

Puede usar varios delimitadores diferentes, siempre que haya un par coincidente:

```
%q(A string)
%q{A string}
%q<A string>
%q|A string|
%q!A string!
```

Concatenación de cuerdas

Concatenar cadenas con el operador `+` :

```
s1 = "Hello"
s2 = " "
s3 = "World"

puts s1 + s2 + s3
# => Hello World

s = s1 + s2 + s3
```

```
puts s
# => Hello World
```

O con el operador `<<` :

```
s = 'Hello'
s << ' '
s << 'World'
puts s
# => Hello World
```

Tenga en cuenta que el operador `<<` modifica el objeto en el lado izquierdo.

También puedes multiplicar cadenas, por ejemplo

```
"wow" * 3
# => "wowwowwow"
```

Interpolación de cuerdas

El delimitador de doble comillas `"` y `%Q` secuencia `%Q` admiten la interpolación de cadenas usando `#{ruby_expression}` :

```
puts "Now is #{Time.now}"
# Now is Now is 2016-07-21 12:47:45 +0200

puts %Q(Now is #{Time.now})
# Now is Now is 2016-07-21 12:47:45 +0200
```

Manipulación de casos

```
"string".upcase     # => "STRING"
"STRING".downcase   # => "string"
"String".swapcase   # => "sTRING"
"string".capitalize # => "String"
```

Estos cuatro métodos no modifican el receptor original. Por ejemplo,

```
str = "Hello"
str.upcase # => "HELLO"
puts str   # => "Hello"
```

Hay cuatro métodos similares que realizan las mismas acciones pero modifican el receptor original.

```
"string".upcase!     # => "STRING"
"STRING".downcase!   # => "string"
"String".swapcase!   # => "sTRING"
"string".capitalize! # => "String"
```

Por ejemplo,

```
str = "Hello"
str.upcase!  # => "HELLO"
puts str     # => "HELLO"
```

Notas:

- Antes de Ruby 2.4, estos métodos no manejan Unicode.

Dividiendo una cadena

`String#split` divide una `String` en una `Array` , basada en un delimitador.

```
"alpha,beta".split(",")
# => ["alpha", "beta"]
```

Una `String` vacía da como resultado una `Array` vacía:

```
"".split(",")
# => []
```

Un delimitador no coincidente da como resultado una `Array` contiene un solo elemento:

```
"alpha,beta".split(".")
# => ["alpha,beta"]
```

También puedes dividir una cadena usando expresiones regulares:

```
"alpha, beta,gamma".split(/, ?/)
# => ["alpha", "beta", "gamma"]
```

El delimitador es opcional; por defecto, una cadena se divide en espacios en blanco:

```
"alpha beta".split
# => ["alpha", "beta"]
```

Unirse a cuerdas

`Array#join` une un `Array` en una `String` , basado en un delimitador:

```
["alpha", "beta"].join(",")
# => "alpha,beta"
```

El delimitador es opcional, y por defecto es una `String` vacía.

```
["alpha", "beta"].join
# => "alphabeta"
```

Una `Array` vacía da como resultado una `String` vacía, sin importar qué delimitador se use.

```
[].join(",")
# => ""
```

Cuerdas multilínea

La forma más fácil de crear una cadena multilínea es simplemente usar varias líneas entre comillas:

```
address = "Four score and seven years ago our fathers brought forth on this
continent, a new nation, conceived in Liberty, and dedicated to the
proposition that all men are created equal."
```

El principal problema con esa técnica es que si la cadena incluye una cita, romperá la sintaxis de la cadena. Para solucionar el problema, puede utilizar un heredoc en su lugar:

```
puts <<-RAVEN
  Once upon a midnight dreary, while I pondered, weak and weary,
  Over many a quaint and curious volume of forgotten lore—
      While I nodded, nearly napping, suddenly there came a tapping,
  As of some one gently rapping, rapping at my chamber door.
  "'Tis some visitor," I muttered, "tapping at my chamber door—
              Only this and nothing more."
  RAVEN
```

Ruby admite documentos de estilo shell aquí con `<<EOT` , pero el texto de terminación debe comenzar la línea. Eso complica la sangría del código, por lo que no hay muchas razones para usar ese estilo. Desafortunadamente, la cadena tendrá sangrías según la forma en que se sangra el código.

Ruby 2.3 resuelve el problema introduciendo `<<~` que elimina los espacios iniciales en exceso:

2.3

```
def build_email(address)
  return (<<~EMAIL)
    TO: #{address}

    To Whom It May Concern:

    Please stop playing the bagpipes at sunrise!

    Regards,
    Your neighbor
  EMAIL
end
```

Porcentaje de cadenas también trabajan para crear cadenas multilínea:

```
%q(
HAMLET        Do you see yonder cloud that's almost in shape of a camel?
```

```
POLONIUS        By the mass, and 'tis like a camel, indeed.
HAMLET        Methinks it is like a weasel.
POLONIUS        It is backed like a weasel.
HAMLET        Or like a whale?
POLONIUS        Very like a whale
)
```

Hay algunas formas de evitar la interpolación y las secuencias de escape:

- Comillas simples en lugar de comillas dobles: `'\n is a carriage return.'`
- Minúscula `q` en una cadena de porcentaje: `%q[#{not-a-variable}]`
- Cita simple la cadena terminal en un heredoc:

```
<<-'CODE'
   puts 'Hello world!'
CODE
```

Cuerdas formateadas

Ruby puede inyectar una matriz de valores en una cadena reemplazando cualquier marcador de posición con los valores de la matriz suministrada.

```
"Hello %s, my name is %s!" % ['World', 'br3nt']
# => Hello World, my name is br3nt!
```

Los marcadores de posición están representados por dos `%s` y los valores son proporcionados por la matriz `['Hello', 'br3nt']` . El operador `%` indica a la cadena que inyecte los valores de la matriz.

Reemplazos de caracteres de cadena

El método `tr` devuelve una copia de una cadena donde los caracteres del primer argumento son reemplazados por los caracteres del segundo argumento.

```
"string".tr('r', 'l') # => "stling"
```

Para reemplazar solo la primera aparición de un patrón con otra expresión, use el método `sub`

```
"string ring".sub('r', 'l') # => "stling ring"
```

Si desea reemplazar *todas las* apariciones de un patrón con esa expresión, use `gsub`

```
"string ring".gsub('r','l') # => "stling ling"
```

Para eliminar caracteres, pase una cadena vacía para el segundo parámetro

También puedes usar expresiones regulares en todos estos métodos.

Es importante tener en cuenta que estos métodos solo devolverán una nueva copia de una cadena y no la modificarán en su lugar. Para hacer eso, necesitas usar el `tr!` , `sub!` y `gsub!` métodos respectivamente.

Entendiendo los datos en una cadena

En Ruby, una cadena es solo una secuencia de bytes junto con el nombre de una codificación (como `UTF-8` , `US-ASCII` , `ASCII-8BIT`) que especifica cómo puede interpretar esos bytes como caracteres.

Las cadenas de Ruby se pueden usar para contener texto (básicamente una secuencia de caracteres), en cuyo caso se suele usar la codificación UTF-8.

```
"abc".bytes  # => [97, 98, 99]
"abc".encoding.name  # => "UTF-8"
```

Las cadenas de Ruby también se pueden usar para almacenar datos binarios (una secuencia de bytes), en cuyo caso se suele utilizar la codificación ASCII-8BIT.

```
[42].pack("i").encoding  # => "ASCII-8BIT"
```

Es posible que la secuencia de bytes en una cadena no coincida con la codificación, lo que resulta en errores si intenta usar la cadena.

```
"\xFF \xFF".valid_encoding? # => false
"\xFF \xFF".split(' ')      # ArgumentError: invalid byte sequence in UTF-8
```

Sustitución de cuerdas

```
p "This is %s" % "foo"
# => "This is foo"

p "%s %s %s" % ["foo", "bar", "baz"]
# => "foo bar baz"

p "%{foo} == %{foo}" % {:foo => "foo" }
# => "foo == foo"
```

Vea String `%` docs y Kernel :: sprintf para más detalles.

Cadena comienza con

Para encontrar si una cadena comienza con un patrón, el `start_with?` el método es útil

```
str = "zebras are cool"
str.start_with?("zebras")      => true
```

También puede comprobar la posición del patrón con el `index`

```
str = "zebras are cool"
str.index("zebras").zero?      => true
```

Cadena termina con

Para encontrar si una cadena termina con un patrón, el `end_with?` el método es útil

```
str = "I like pineapples"
str.end_with?("pineaaples")      => false
```

Cuerdas de posicionamiento

En Ruby, las cadenas pueden ser justificadas a la izquierda, justificadas a la derecha o centradas

Para justificar a la izquierda la cadena, use el método `ljust` . Esto incluye dos parámetros, un entero que representa el número de caracteres de la nueva cadena y una cadena, que representa el patrón a rellenar.

Si el número entero es mayor que la longitud de la cadena original, la nueva cadena se justificará a la izquierda con el parámetro de cadena opcional tomando el espacio restante. Si no se proporciona el parámetro de cadena, la cadena se rellenará con espacios.

```
str ="abcd"
str.ljust(4)          => "abcd"
str.ljust(10)         => "abcd      "
```

Para justificar a la derecha una cadena, use el método `rjust` . Esto incluye dos parámetros, un entero que representa el número de caracteres de la nueva cadena y una cadena, que representa el patrón a rellenar.

Si el número entero es mayor que la longitud de la cadena original, la nueva cadena se justificará a la derecha con el parámetro de cadena opcional que ocupará el espacio restante. Si no se proporciona el parámetro de cadena, la cadena se rellenará con espacios.

```
str = "abcd"
str.rjust(4)          => "abcd"
str.rjust(10)         => "      abcd"
```

Para centrar una cadena, usa el método `center` . Esto incluye dos parámetros, un entero que representa el ancho de la nueva cadena y una cadena con la que se rellenará la cadena original. La cuerda se alineará al centro.

```
str = "abcd"
str.center(4)          => "abcd"
str.center(10)         => "   abcd   "
```

Lea Instrumentos de cuerda en línea: https://riptutorial.com/es/ruby/topic/834/instrumentos-de-cuerda

Capítulo 39: Introspección

Examples

Ver los métodos de un objeto.

Inspeccionar un objeto

Puede encontrar los métodos públicos a los que un objeto puede responder utilizando los `methods` o los `methods public_methods` , que devuelven una matriz de símbolos:

```
class Foo
  def bar; 42; end
end
f = Foo.new
def f.yay; 17; end
p f.methods.sort
#=> [:!, :!=, :!~, :<=>, :==, :===, :=~, :__id__, :__send__, :bar, :class, :clone,
#=>  :define_singleton_method, :display, :dup, :enum_for, :eql?, :equal?, :extend,
#=>  :freeze, :frozen?, :hash, :inspect, :instance_eval, :instance_exec,
#=>  :instance_of?, :instance_variable_defined?, :instance_variable_get,
#=>  :instance_variable_set, :instance_variables, :is_a?, :itself, :kind_of?,
#=>  :method, :methods, :nil?, :object_id, :private_methods, :protected_methods,
#=>  :public_method, :public_methods, :public_send, :remove_instance_variable,
#=>  :respond_to?, :send, :singleton_class, :singleton_method, :singleton_methods,
#=>  :taint, :tainted?, :tap, :to_enum, :to_s, :trust, :untaint, :untrust,
#=>  :untrusted?, :yay]
```

Para una lista más específica, puede eliminar métodos comunes a todos los objetos, por ejemplo,

```
p (f.methods - Object.methods).sort
#=> [:bar,:yay]
```

Alternativamente, puede pasar `false` a `methods` o `public_methods` :

```
p f.methods(false) # public and protected singleton methods of `f`
#=> [:yay]

p f.public_methods(false)
#=> [:yay, :bar]
```

Puede encontrar los métodos privados y protegidos de un objeto utilizando `private_methods` y `protected_methods` :

```
p f.private_methods.sort
#=> [:Array, :Complex, :DelegateClass, :Float, :Hash, :Integer, :Rational, :String,
#=>  :__callee__, :__dir__, :__method__, :`, :abort, :at_exit, :autoload, :autoload?,
#=>  :binding, :block_given?, :caller, :caller_locations, :catch,
#=>  :default_src_encoding, :eval, :exec, :exit, :exit!, :fail, :fork, :format, :gem,
#=>  :gem_original_require, :gets, :global_variables, :initialize, :initialize_clone,
```

```
#=>  :initialize_copy, :initialize_dup, :irb_binding, :iterator?, :lambda, :load,
#=>  :local_variables, :loop, :method_missing, :open, :p, :print, :printf, :proc,
#=>  :putc, :puts, :raise, :rand, :readline, :readlines, :require, :require_relative,
#=>  :respond_to_missing?, :select, :set_trace_func, :singleton_method_added,
#=>  :singleton_method_removed, :singleton_method_undefined, :sleep, :spawn,
#=>  :sprintf, :srand, :syscall, :system, :test, :throw, :trace_var, :trap,
#=>  :untrace_var, :warn]

p f.protected_methods
#=> []
```

Al igual que con `methods` y `public_methods` , puede pasar `false` a `private_methods` y `protected_methods` a recortar los métodos heredados.

Inspeccionar una clase o módulo

Además de `methods` , `public_methods` , `protected_methods` , y `private_methods` , clases y módulos exponer `instance_methods` , `public_instance_methods` , `protected_instance_methods` , y `private_instance_methods` para determinar los métodos expuestos para los objetos que heredan de la clase o módulo. Como antes, puede pasar `false` a estos métodos para excluir métodos heredados:

```
p Foo.instance_methods.sort
#=> [:!, :!=, :!~, :<=>, :==, :===, :=~, :__id__, :__send__, :bar, :class,
#=>  :clone, :define_singleton_method, :display, :dup, :enum_for, :eql?,
#=>  :equal?, :extend, :freeze, :frozen?, :hash, :inspect, :instance_eval,
#=>  :instance_exec, :instance_of?, :instance_variable_defined?,
#=>  :instance_variable_get, :instance_variable_set, :instance_variables,
#=>  :is_a?, :itself, :kind_of?, :method, :methods, :nil?, :object_id,
#=>  :private_methods, :protected_methods, :public_method, :public_methods,
#=>  :public_send, :remove_instance_variable, :respond_to?, :send,
#=>  :singleton_class, :singleton_method, :singleton_methods, :taint,
#=>  :tainted?, :tap, :to_enum, :to_s, :trust, :untaint, :untrust, :untrusted?]

p Foo.instance_methods(false)
#=> [:bar]
```

Finalmente, si olvida los nombres de la mayoría de estos en el futuro, puede encontrar todos estos métodos utilizando `methods` :

```
p f.methods.grep(/methods/)
#=> [:private_methods, :methods, :protected_methods, :public_methods,
#=>  :singleton_methods]

p Foo.methods.grep(/methods/)
#=> [:public_instance_methods, :instance_methods, :private_instance_methods,
#=>  :protected_instance_methods, :private_methods, :methods,
#=>  :protected_methods, :public_methods, :singleton_methods]
```

Ver las variables de instancia de un objeto

Es posible consultar un objeto acerca de sus variables de `instance_variables` usando `instance_variables` , `instance_variable_defined?` y `instance_variable_get` , y `remove_instance_variable`

usando `instance_variable_set` y `remove_instance_variable` :

```
class Foo
  attr_reader :bar
  def initialize
    @bar = 42
  end
end
f = Foo.new
f.instance_variables                 #=> [:@bar]
f.instance_variable_defined?(:@baz) #=> false
f.instance_variable_defined?(:@bar) #=> true
f.instance_variable_get(:@bar)       #=> 42
f.instance_variable_set(:@bar, 17)    #=> 17
f.bar                                #=> 17
f.remove_instance_variable(:@bar)    #=> 17
f.bar                                #=> nil
f.instance_variables                 #=> []
```

Los nombres de las variables de instancia incluyen el símbolo `@` . Obtendrá un error si lo omite:

```
f.instance_variable_defined?(:jim)
#=> NameError: `jim' is not allowed as an instance variable name
```

Ver variables globales y locales

El `Kernel` expone métodos para obtener la lista de `global_variables` y `local_variables` :

```
cats  = 42
$demo = "in progress"
p global_variables.sort
#=> [:$!, :$", :$$, :$&, :$', :$*, :$+, :$,, :$-0, :$-F, :$-I, :$-K, :$-W, :$-a,
#=>  :$-d, :$-i, :$-l, :$-p, :$-v, :$-w, :$., :$/, :$0, :$1, :$2, :$3, :$4, :$5,
#=>  :$6, :$7, :$8, :$9, :$:, :$;, :$<, :$=, :$>, :$?, :$@, :$DEBUG, :$FILENAME,
#=>  :$KCODE, :$LOADED_FEATURES, :$LOAD_PATH, :$PROGRAM_NAME, :$SAFE, :$VERBOSE,
#=>  :$\, :$_, :$`, :$binding, :$demo, :$stderr, :$stdin, :$stdout, :$~]

p local_variables
#=> [:cats]
```

A diferencia de las variables de instancia, no hay métodos específicos para obtener, configurar o eliminar variables globales o locales. Buscar dicha funcionalidad suele ser una señal de que su código debe reescribirse para usar un Hash para almacenar los valores. Sin embargo, si debe modificar variables globales o locales por nombre, puede usar `eval` con una cadena:

```
var = "$demo"
eval(var)           #=> "in progress"
eval("#{var} = 17")
p $demo             #=> 17
```

Por defecto, `eval` evaluará sus variables en el alcance actual. Para evaluar las variables locales en un ámbito diferente, debe capturar el *enlace* donde existen las variables locales.

```
def local_variable_get(name, bound=nil)
  foo = :inside
  eval(name,bound)
end

def test_1
  foo = :outside
  p local_variable_get("foo")
end

def test_2
  foo = :outside
  p local_variable_get("foo",binding)
end

test_1 #=> :inside
test_2 #=> :outside
```

En lo anterior, `test_1` no pasó un enlace a `local_variable_get` , por lo que la `eval` se ejecutó dentro del contexto de ese método, donde una variable local llamada `foo` se estableció en `:inside` .

Ver variables de clase

Las clases y los módulos tienen los mismos métodos para introspeccionar variables de instancia que cualquier otro objeto. La clase y los módulos también tienen métodos similares para consultar las variables de la clase (`@@these_things`):

```
p Module.methods.grep(/class_variable/)
#=> [:class_variables, :class_variable_get, :remove_class_variable,
#=>  :class_variable_defined?, :class_variable_set]

class Foo
  @@instances = 0
  def initialize
    @@instances += 1
  end
end

class Bar < Foo; end

5.times{ Foo.new }
3.times{ Bar.new }
p Foo.class_variables                   #=> [:@@instances]
p Bar.class_variables                   #=> [:@@instances]
p Foo.class_variable_get(:@@instances)  #=> 8
p Bar.class_variable_get(:@@instances)  #=> 8
```

Similar a las variables de instancia, el nombre de las variables de clase debe comenzar con `@@` , o aparecerá un error:

```
p Bar.class_variable_defined?( :instances )
#=> NameError: `instances' is not allowed as a class variable name
```

Lea Introspección en línea: https://riptutorial.com/es/ruby/topic/6227/introspeccion

Capítulo 40: Introspección en rubí

Introducción

¿Qué es la introspección?

La introspección está mirando hacia adentro para saber sobre el interior. Esa es una definición simple de introspección.

En programación y Ruby en general ... la introspección es la capacidad de mirar un objeto, una clase ... en el tiempo de ejecución para saber sobre eso.

Examples

Veamos algunos ejemplos.

Ejemplo:

```
s = "Hello"  # s is a string
```

Entonces nos enteramos de algo sobre s. Vamos a empezar:

Entonces, ¿quieres saber cuál es la clase de s en el tiempo de ejecución?

```
irb(main):055:0* s.class
=> String
```

Ohh bien. ¿Pero cuáles son los métodos de s?

```
irb(main):002:0> s.methods
=> [:unicode_normalize, :include?, :to_c, :unicode_normalize!, :unicode_normalized?, :%, :*,
:+, :count, :partition, :unpack, :encode, :encode!, :next, :casecmp, :insert, :bytesize,
:match, :succ!, :next!, :upto, :index, :rindex, :replace, :clear, :chr, :+@, :-@, :setbyte,
:getbyte, :<=>, :<<, :scrub, :scrub!, :byteslice, :==, :===, :dump, :=~, :downcase, :[], :[]=,
:upcase, :downcase!, :capitalize, :swapcase, :upcase!, :oct, :empty?, :eql?, :hex, :chars,
:split, :capitalize!, :swapcase!, :concat, :codepoints, :reverse, :lines, :bytes, :prepend,
:scan, :ord, :reverse!, :center, :sub, :freeze, :inspect, :intern, :end_with?, :gsub, :chop,
:crypt, :gsub!, :start_with?, :rstrip, :sub!, :ljust, :length, :size, :strip!, :succ,
:rstrip!, :chomp, :strip, :rjust, :lstrip!, :tr!, :chomp!, :squeeze, :lstrip, :tr_s!, :to_str,
:to_sym, :chop!, :each_byte, :each_char, :each_codepoint, :to_s, :to_i, :tr_s, :delete,
:encoding, :force_encoding, :sum, :delete!, :squeeze!, :tr, :to_f, :valid_encoding?, :slice,
:slice!, :rpartition, :each_line, :b, :ascii_only?, :hash, :to_r, :<, :>, :<=, :>=, :between?,
:instance_of?, :public_send, :instance_variable_get, :instance_variable_set,
:instance_variable_defined?, :remove_instance_variable, :private_methods, :kind_of?,
:instance_variables, :tap, :is_a?, :extend, :to_enum, :enum_for, :!~, :respond_to?, :display,
:object_id, :send, :method, :public_method, :singleton_method, :define_singleton_method,
:nil?, :class, :singleton_class, :clone, :dup, :itself, :taint, :tainted?, :untaint, :untrust,
:trust, :untrusted?, :methods, :protected_methods, :frozen?, :public_methods,
:singleton_methods, :!, :!=, :__send__, :equal?, :instance_eval, :instance_exec, :__id__]
```

¿Quieres saber si s es una instancia de String?

```
irb(main):017:0*
irb(main):018:0* s.instance_of?(String)
=> true
```

¿Cuáles son los métodos públicos de s?

```
irb(main):026:0* s.public_methods
=> [:unicode_normalize, :include?, :to_c, :unicode_normalize!, :unicode_normalized?, :%, :*,
:+, :count, :partition, :unpack, :encode, :encode!, :next, :casecmp, :insert, :bytesize,
:match, :succ!, :next!, :upto, :index, :rindex, :replace, :clear, :chr, :+@, :-@, :setbyte,
:getbyte, :<=>, :<<, :scrub, :scrub!, :byteslice, :==, :===, :dump, :=~, :downcase, :[], :[]=,
:upcase, :downcase!, :capitalize, :swapcase, :upcase!, :oct, :empty?, :eql?, :hex, :chars,
:split, :capitalize!, :swapcase!, :concat, :codepoints, :reverse, :lines, :bytes, :prepend,
:scan, :ord, :reverse!, :center, :sub, :freeze, :inspect, :intern, :end_with?, :gsub, :chop,
:crypt, :gsub!, :start_with?, :rstrip, :sub!, :ljust, :length, :size, :strip!, :succ,
:rstrip!, :chomp, :strip, :rjust, :lstrip!, :tr!, :chomp!, :squeeze, :lstrip, :tr_s!, :to_str,
:to_sym, :chop!, :each_byte, :each_char, :each_codepoint, :to_s, :to_i, :tr_s, :delete,
:encoding, :force_encoding, :sum, :delete!, :squeeze!, :tr, :to_f, :valid_encoding?, :slice,
:slice!, :rpartition, :each_line, :b, :ascii_only?, :hash, :to_r, :<, :>, :<=, :>=, :between?,
:pretty_print, :pretty_print_cycle, :pretty_print_instance_variables, :pretty_print_inspect,
:instance_of?, :public_send, :instance_variable_get, :instance_variable_set,
:instance_variable_defined?, :remove_instance_variable, :private_methods, :kind_of?,
:instance_variables, :tap, :pretty_inspect, :is_a?, :extend, :to_enum, :enum_for, :!~,
:respond_to?, :display, :object_id, :send, :method, :public_method, :singleton_method,
:define_singleton_method, :nil?, :class, :singleton_class, :clone, :dup, :itself, :taint,
:tainted?, :untaint, :untrust, :trust, :untrusted?, :methods, :protected_methods, :frozen?,
:public_methods, :singleton_methods, :!, :!=, :__send__, :equal?, :instance_eval,
:instance_exec, :__id__]
```

y métodos privados ....

```
irb(main):030:0* s.private_methods
=> [:initialize, :initialize_copy, :DelegateClass, :default_src_encoding, :irb_binding,
:sprintf, :format, :Integer, :Float, :String, :Array, :Hash, :catch, :throw, :loop,
:block_given?, :Complex, :set_trace_func, :trace_var, :untrace_var, :at_exit, :Rational,
:caller, :caller_locations, :select, :test, :fork, :exit, :`, :gem_original_require, :sleep,
:pp, :respond_to_missing?, :load, :exec, :exit!, :system, :spawn, :abort, :syscall, :printf,
:open, :putc, :print, :readline, :puts, :p, :srand, :readlines, :gets, :rand, :proc, :lambda,
:trap, :initialize_clone, :initialize_dup, :gem, :require, :require_relative, :autoload,
:autoload?, :binding, :local_variables, :warn, :raise, :fail, :global_variables, :__method__,
:__callee__, :__dir__, :eval, :iterator?, :method_missing, :singleton_method_added,
:singleton_method_removed, :singleton_method_undefined]
```

Sí, tengo un nombre de método superior. ¿Quieres obtener la versión mayúscula de s? Intentemos:

```
irb(main):044:0> s.respond_to?(:upper)
=> false
```

Parece que no, el método correcto es upcase permite verificar:

```
irb(main):047:0*
irb(main):048:0* s.respond_to?(:upcase)
```

```
=> true
```

Introspección de clase

Veamos a continuación la definición de clase.

```
class A
   def a; end
end

module B
   def b; end
end

class C < A
   include B
   def c; end
end
```

¿Cuáles son los métodos de instancia de `C` ?

```
C.instance_methods # [:c, :b, :a, :to_json, :instance_of?...]
```

¿Cuáles son los métodos de instancia que declaran solo en `C` ?

```
C.instance_methods(false) # [:c]
```

¿Cuáles son los ancestros de la clase `C` ?

```
C.ancestors # [C, B, A, Object,...]
```

Superclase de `C` ?

```
C.superclass # A
```

Lea Introspección en rubí en línea: https://riptutorial.com/es/ruby/topic/8752/introspeccion-en-rubi

Capítulo 41: IRB

Introducción

IRB significa "Ruby Interactive Shell". Básicamente, te permite ejecutar comandos ruby en tiempo real (como lo hace el shell normal). IRB es una herramienta indispensable cuando se trata de API Ruby. Funciona como script rb clásico. Úsalo para comandos cortos y fáciles. Una de las buenas funciones de IRB es que cuando presiona la tecla tab mientras se escribe un método, se le dará un consejo sobre lo que puede usar (esto no es un IntelliSense)

Parámetros

Opción	Detalles
-F	Suprimir lectura de ~ / .irbrc
-metro	Modo Bc (carga mathn, fracción o matriz están disponibles)
-re	Establezca $ DEBUG en verdadero (igual que `ruby -d ')
-r módulo de carga	Igual que `ruby -r '
-I camino	Especifique el directorio $ LOAD_PATH
-U	Igual que el `ruby -U`
-E enc	Igual que el `ruby -E`
-w	Igual que `ruby -w`
-W [nivel = 2]	Igual que el `ruby -W`
--inspeccionar	Utilice `inspeccionar 'para la salida (por defecto, excepto para el modo bc)
--sospecho	No utilice inspeccionar para salida
--la línea de lectura	Utilice el módulo de extensión Readline
--noreadline	No use el módulo de extensión Readline
--prompt modo de solicitud	Cambiar el modo de aviso. Los modos de solicitud `default', ` son por `default', ` simple', `xmp' and` inf-ruby'
--inf-ruby-mode	Use el indicador apropiado para inf-ruby-mode en emacs. Suprime --la línea de lectura.

Opción	Detalles
--simple-prompt	Modo de solicitud simple
- no aparece	No hay modo de aviso
--trazador	Mostrar traza para cada ejecución de comandos.
--back-trace-limit n	Mostrar el trazo de vuelta top n y tail n. El valor predeterminado es 16.
--irb_debug n	Establezca el nivel de depuración interna en n (no para uso popular)
-v, --version	Imprime la versión de irb

Examples

Uso básico

IRB significa "Ruby Shell interactivo", que nos permite ejecutar expresiones de ruby desde la entrada estándar.

Para empezar, escribe `irb` en tu shell. Puedes escribir cualquier cosa en Ruby, desde expresiones simples:

```
$ irb
2.1.4 :001 > 2+2
=> 4
```

A casos complejos como métodos:

```
2.1.4 :001> def method
2.1.4 :002?>   puts "Hello World"
2.1.4 :003?> end
=> :method
2.1.4 :004 > method
Hello World
=> nil
```

Iniciar una sesión IRB dentro de un script Ruby

A partir de Ruby 2.4.0, puede iniciar una sesión IRB interactiva dentro de cualquier script de Ruby usando estas líneas:

```
require 'irb'
binding.irb
```

Esto iniciará un REPL de IBR donde tendrá el valor esperado para `self` y podrá acceder a todas las variables locales y variables de instancia que estén dentro del alcance. Escriba Ctrl + D o `quit`

para reanudar su programa Ruby.

Esto puede ser muy útil para la depuración.

Lea IRB en línea: https://riptutorial.com/es/ruby/topic/4800/irb

Capítulo 42: Iteración

Examples

Cada

Ruby tiene muchos tipos de enumeradores, pero el primer y más simple tipo de enumerador para comenzar es `each` . Imprimiremos `even` o `odd` para cada número entre `1` y `10` para mostrar cómo funciona `each` .

Básicamente hay dos formas de pasar los llamados `blocks` . Un `block` es un fragmento de código que se pasa y se ejecutará mediante el método al que se llama. `each` método toma un `block` que llama a cada elemento de la colección de objetos a los que fue llamado.

Hay dos formas de pasar un bloque a un método:

Método 1: en línea

```
(1..10).each { |i| puts i.even? ? 'even' : 'odd' }
```

Esta es una forma muy comprimida y *rubí* de resolver esto. Vamos a desglosar pieza por pieza.

1. `(1..10)` es un rango de `1` a `10` inclusive. Si quisiéramos que fuera de `1` a `10` exclusiva, escribiríamos `(1...10)` .
2. `.each` es un enumerador que enumera `each` elemento en el objeto sobre el que está actuando. En este caso, actúa sobre `each` número en el rango.
3. `{ |i| puts i.even? ? 'even' : 'odd' }` es el bloque para `each` declaración, que a su vez se puede desglosar.
    1. `|i|` esto significa que cada elemento en el rango está representado dentro del bloque por el identificador `i` .
    2. `puts` es un método de salida en Ruby que tiene un salto de línea automático cada vez que se imprime. (Podemos usar `print` si no queremos el salto de línea automático)
    3. `i.even?` comprueba si `i` par. También podríamos haber utilizado `i % 2 == 0` ; sin embargo, es preferible utilizar métodos incorporados.
    4. `? "even" : "odd"` es el operador ternario de ruby. ¿La forma en que se construye un operador ternario es la `expression ? a : b` . Esto es corto para

    ```
    if expression
      a
    else
      b
    end
    ```

Para el código más largo de una línea, el `block` debe pasar como un `multiline block` .

Método 2: multilínea

```
(1..10).each do |i|
  if i.even?
    puts 'even'
  else
    puts 'odd'
  end
end
```

En un `multiline block do` reemplaza el soporte de apertura y el `end` reemplaza el soporte de cierre del estilo en `inline` .

Ruby soporta reverse_each también. Se itera la matriz hacia atrás.

```
@arr = [1,2,3,4]
puts @arr.inspect # output is [1,2,3,4]

print "Reversed array elements["
@arr.reverse_each do |val|
        print " #{val} " # output is 4 3 2 1
end
print "]\n"
```

Implementación en una clase

`Enumerable` es el módulo más popular en Ruby. Su propósito es proporcionarle métodos iterables como `map` , `select` , `reduce` , etc. Las clases que usan `Enumerable` incluyen `Array` , `Hash` , `Range` . Para usarlo, debes `include Enumerable` e implementar `each` .

```
class NaturalNumbers
  include Enumerable

  def initialize(upper_limit)
    @upper_limit = upper_limit
  end

  def each(&block)
    0.upto(@upper_limit).each(&block)
  end
end

n = NaturalNumbers.new(6)

n.reduce(:+)                    # => 21
n.select(&:even?)               # => [0, 2, 4, 6]
n.map { |number| number ** 2 } # => [0, 1, 4, 9, 16, 25, 36]
```

Mapa

Devuelve el objeto modificado, pero el objeto original permanece como estaba. Por ejemplo:

```
arr = [1, 2, 3]
arr.map { |i| i + 1 } # => [2, 3, 4]
arr # => [1, 2, 3]
```

`map!` Cambia el objeto original:

```
arr = [1, 2, 3]
arr.map! { |i| i + 1 } # => [2, 3, 4]
arr # => [2, 3, 4]
```

Nota: también puedes usar la `collect` para hacer lo mismo.

Iterando sobre objetos complejos.

Arrays

Puede iterar sobre matrices anidadas:

```
[[1, 2], [3, 4]].each { |(a, b)| p "a: #{ a }", "b: #{ b }" }
```

La siguiente sintaxis también está permitida:

```
[[1, 2], [3, 4]].each { |a, b| "a: #{ a }", "b: #{ b }" }
```

Producirá:

```
"a: 1"
"b: 2"
"a: 3"
"b: 4"
```

Hashes

Puede iterar sobre pares clave-valor:

```
{a: 1, b: 2, c: 3}.each { |pair| p "pair: #{ pair }" }
```

Producirá:

```
"pair: [:a, 1]"
"pair: [:b, 2]"
"pair: [:c, 3]"
```

Puede iterar sobre claves y valores simultáneamente:

```
{a: 1, b: 2, c: 3}.each { |(k, v)| p "k: #{ k }", "v: #{ k }" }
```

Producirá:

```
"k: a"
"v: a"
"k: b"
"v: b"
"k: c"
"v: c"
```

Para iterador

Esto se repite de 4 a 13 (inclusive).

```
for i in 4..13
    puts "this is #{i}.th number"
end
```

También podemos iterar sobre matrices usando para

```
names = ['Siva', 'Charan', 'Naresh', 'Manish']

for name in names
    puts name
end
```

Iteración con índice

A veces desea saber la posición (**índice**) del elemento actual mientras se repite en un enumerador. Para tal fin, Ruby proporciona el método `with_index` . Se puede aplicar a todos los enumeradores. Básicamente, al agregar `with_index` a una enumeración, puede enumerar esa enumeración. El índice se pasa a un bloque como segundo argumento.

```
[2,3,4].map.with_index { |e, i| puts "Element of array number #{i} => #{e}" }
#Element of array number 0 => 2
#Element of array number 1 => 3
#Element of array number 2 => 4
#=> [nil, nil, nil]
```

`with_index` tiene un argumento opcional: el primer índice es `0` por defecto:

```
[2,3,4].map.with_index(1) { |e, i| puts "Element of array number #{i} => #{e}" }
#Element of array number 1 => 2
#Element of array number 2 => 3
#Element of array number 3 => 4
#=> [nil, nil, nil]
```

Hay un método específico `each_with_index` . La única diferencia entre él y cada uno. Con `each.with_index` es que no puede pasarle un argumento a eso, por lo que el primer índice es `0` todo el tiempo.

```
[2,3,4].each_with_index { |e, i| puts "Element of array number #{i} => #{e}" }
#Element of array number 0 => 2
#Element of array number 1 => 3
```

```
#Element of array number 2 => 4
#=> [2, 3, 4]
```

Lea Iteración en línea: https://riptutorial.com/es/ruby/topic/1159/iteracion

Capítulo 43: JSON con Ruby

Examples

Usando JSON con Ruby

JSON (JavaScript Object Notation) es un formato de intercambio de datos ligero. Muchas aplicaciones web lo utilizan para enviar y recibir datos.

En Ruby puedes simplemente trabajar con JSON.

Al principio, debe `require 'json'` , luego puede analizar una cadena JSON a través del `JSON.parse()` .

```
require 'json'

j = '{"a": 1, "b": 2}'
puts JSON.parse(j)
>> {"a"=>1, "b"=>2}
```

Lo que sucede aquí es que el analizador genera un Ruby Hash a partir del JSON.

Al revés, generar JSON a partir de un hash de Ruby es tan simple como analizar. El método de elección es `to_json` :

```
require 'json'

hash = { 'a' => 1, 'b' => 2 }
json = hash.to_json
puts json
>> {"a":1,"b":2}
```

Usando simbolos

Puedes usar JSON junto con los símbolos de Ruby. Con la opción symbolize_names para el analizador, las claves en el hash resultante serán símbolos en lugar de cadenas.

```
json = '{ "a": 1, "b": 2 }'
puts JSON.parse(json, symbolize_names: true)
>> {:a=>1, :b=>2}
```

Lea JSON con Ruby en línea: https://riptutorial.com/es/ruby/topic/5853/json-con-ruby

Capítulo 44: La verdad

Observaciones

Como regla general, evite usar negaciones dobles en el código. Rubocop dice que las negaciones dobles son innecesariamente complejas y que a menudo se pueden reemplazar con algo más legible.

En lugar de escribir

```
def user_exists?
    !!user
end
```

utilizar

```
def user_exists?
    !user.nil?
end
```

Examples

Todos los objetos se pueden convertir a booleanos en Ruby

Use la sintaxis de doble negación para verificar la veracidad de los valores. Todos los valores corresponden a un booleano, independientemente de su tipo.

```
irb(main):001:0> !!1234
=> true
irb(main):002:0> !!"Hello, world!"
(irb):2: warning: string literal in condition
=> true
irb(main):003:0> !!true
=> true
irb(main):005:0> !!{a:'b'}
=> true
```

Todos los valores excepto `nil` y `false` son veraces.

```
irb(main):006:0> !!nil
=> false
irb(main):007:0> !!false
=> false
```

La veracidad de un valor se puede usar en las construcciones if-else

No es necesario utilizar la doble negación en las sentencias if-else.

```
if 'hello'
    puts 'hey!'
else
    puts 'bye!'
end
```

El código de arriba imprime 'hey!' en la pantalla.

Lea La verdad en línea: https://riptutorial.com/es/ruby/topic/5852/la-verdad

Capítulo 45: Las clases

Sintaxis

- nombre de la clase
- # algún código que describe el comportamiento de la clase
- fin

Observaciones

Los nombres de las clases en Ruby son constantes, por lo que la primera letra debe ser mayúscula.

```
class Cat # correct
end

class dog # wrong, throws an error
end
```

Examples

Creando una clase

Puede definir una nueva clase utilizando la palabra clave de `class` .

```
class MyClass
end
```

Una vez definido, puede crear una nueva instancia usando el método `.new`

```
somevar = MyClass.new
# => #<MyClass:0x007fe2b8aa4a18>
```

Constructor

Una clase solo puede tener un constructor, es decir, un método llamado `initialize` . El método se invoca automáticamente cuando se crea una nueva instancia de la clase.

```
class Customer
  def initialize(name)
     @name = name.capitalize
  end
end

sarah = Customer.new('sarah')
sarah.name #=> 'Sarah'
```

Variables de clase e instancia

Hay varios tipos de variables especiales que una clase puede usar para compartir datos más fácilmente.

Variables de instancia, precedidas por `@` . Son útiles si desea utilizar la misma variable en diferentes métodos.

```
class Person
  def initialize(name, age)
    my_age = age # local variable, will be destroyed at end of constructor
    @name = name # instance variable, is only destroyed when the object is
  end

  def some_method
    puts "My name is #{@name}." # we can use @name with no problem
  end

  def another_method
    puts "My age is #{my_age}." # this will not work!
  end
end

mhmd = Person.new("Mark", 23)

mhmd.some_method #=> My name is Mark.
mhmd.another_method #=> throws an error
```

Variable de clase, precedida por `@@` . Contienen los mismos valores en todas las instancias de una clase.

```
class Person
  @@persons_created = 0 # class variable, available to all objects of this class
  def initialize(name)
    @name = name

    # modification of class variable persists across all objects of this class
    @@persons_created += 1
  end

  def how_many_persons
    puts "persons created so far: #{@@persons_created}"
  end
end

mark = Person.new("Mark")
mark.how_many_persons #=> persons created so far: 1
helen = Person.new("Helen")

mark.how_many_persons #=> persons created so far: 2
helen.how_many_persons #=> persons created so far: 2
# you could either ask mark or helen
```

Variables globales, precedidas por `$` . Estos están disponibles en cualquier parte del programa, así que asegúrese de usarlos sabiamente.

```
$total_animals = 0

class Cat
  def initialize
    $total_animals += 1
  end
end

class Dog
  def initialize
    $total_animals += 1
  end
end

bob = Cat.new()
puts $total_animals #=> 1
fred = Dog.new()
puts $total_animals #=> 2
```

Acceso a variables de instancia con captadores y definidores.

Tenemos tres métodos:

1. **`attr_reader`** : se utiliza para permitir `read` la variable fuera de la clase.
2. **`attr_writer`** : se utiliza para permitir la modificación de la variable fuera de la clase.
3. **`attr_accessor`** : combina ambos métodos.

```
class Cat
  attr_reader :age # you can read the age but you can never change it
  attr_writer :name # you can change name but you are not allowed to read
  attr_accessor :breed # you can both change the breed and read it

  def initialize(name, breed)
    @name = name
    @breed = breed
    @age = 2
  end
  def speak
    puts "I'm #{@name} and I am a #{@breed} cat"
  end
end

my_cat = Cat.new("Banjo", "birman")
# reading values:

my_cat.age  #=> 2
my_cat.breed #=> "birman"
my_cat.name #=> Error

# changing values

my_cat.age = 3 #=> Error
my_cat.breed = "sphynx"
my_cat.name = "Bilbo"

my_cat.speak #=> I'm Bilbo and I am a sphynx cat
```

Tenga en cuenta que los parámetros son símbolos. Esto funciona creando un método.

```
class Cat
  attr_accessor :breed
end
```

Es básicamente lo mismo que:

```
class Cat
  def breed
    @breed
  end
  def breed= value
    @breed = value
  end
end
```

Niveles de acceso

Ruby tiene tres niveles de acceso. Son `public` , `private` y `protected` .

Los métodos que siguen a las palabras clave `private` o `protected` se definen como tales. Los métodos que vienen antes de estos son métodos implícitamente `public` .

Métodos Públicos

Un método público debe describir el comportamiento del objeto que se está creando. Estos métodos se pueden llamar desde fuera del alcance del objeto creado.

```
class Cat
  def initialize(name)
    @name = name
  end

  def speak
    puts "I'm #{@name} and I'm 2 years old"
  end

  ...
end

new_cat = Cat.new("garfield")
#=> <Cat:0x2321868 @name="garfield">

new_cat.speak
#=> I'm garfield and I'm 2 years old
```

Estos métodos son métodos públicos de rubí, describen el comportamiento para inicializar un nuevo gato y el comportamiento del método de hablar.

`public` palabra clave `public` es necesaria, pero puede usarse para escapar de forma `private` o `protected`

```
def MyClass
  def first_public_method
  end

  private

  def private_method
  end

  public

  def second_public_method
  end
end
```

Métodos privados

Los métodos privados no son accesibles desde fuera del objeto. Son utilizados internamente por el objeto. Usando el ejemplo del gato otra vez:

```
class Cat
  def initialize(name)
    @name = name
  end

  def speak
    age = calculate_cat_age # here we call the private method
    puts "I'm #{@name} and I'm #{age} years old"
  end

  private
     def calculate_cat_age
       2 * 3 - 4
     end
end

my_cat = Cat.new("Bilbo")
my_cat.speak #=> I'm Bilbo and I'm 2 years old
my_cat.calculate_cat_age #=> NoMethodError: private method `calculate_cat_age' called for
#<Cat:0x2321868 @name="Bilbo">
```

Como puede ver en el ejemplo anterior, el objeto Cat recién creado tiene acceso al método `calculate_cat_age` internamente. Asignamos la `age` variable al resultado de ejecutar el método private `calculate_cat_age` que imprime el nombre y la edad del gato en la consola.

Cuando intentamos llamar al método `calculate_cat_age` desde fuera del objeto `my_cat` , recibimos un `NoMethodError` porque es privado. ¿Consíguelo?

Métodos protegidos

Los métodos protegidos son muy similares a los métodos privados. No se puede acceder a ellos fuera de la instancia del objeto de la misma manera que los métodos privados no pueden ser. Sin embargo, utilizando el método `self` ruby, los métodos protegidos pueden llamarse dentro del

contexto de un objeto del mismo tipo.

```
class Cat
  def initialize(name, age)
    @name = name
    @age = age
  end

  def speak
    puts "I'm #{@name} and I'm #{@age} years old"
  end

  # this == method allows us to compare two objects own ages.
  # if both Cat's have the same age they will be considered equal.
  def ==(other)
     self.own_age == other.own_age
  end

  protected
     def own_age
        self.age
     end
end

cat1 = Cat.new("ricky", 2)
=> #<Cat:0x007fe2b8aa4a18 @name="ricky", @age=2>

cat2 = Cat.new("lucy", 4)
=> #<Cat:0x008gfb7aa6v67 @name="lucy", @age=4>

cat3 = Cat.new("felix", 2)
=> #<Cat:0x009frbaa8V76 @name="felix", @age=2>
```

Puede ver que hemos agregado un parámetro de edad a la clase cat y que hemos creado tres nuevos objetos cat con el nombre y la edad. Vamos a llamar al método protegido `own_age` para comparar la edad de los objetos de nuestro gato.

```
cat1 == cat2
=> false

cat1 == cat3
=> true
```

Mire eso, pudimos recuperar la edad de cat1 utilizando el método `self.own_age` protected y compararla con la edad de cat2 llamando a `cat2.own_age` dentro de cat1.

Clases de métodos de clase

Las clases tienen 3 tipos de métodos: métodos de instancia, singleton y clase.

Métodos de instancia

Estos son métodos que se pueden llamar desde una `instance` de la clase.

```
class Thing
  def somemethod
    puts "something"
  end
end

foo = Thing.new # create an instance of the class
foo.somemethod # => something
```

Método de clase

Estos son métodos estáticos, es decir, pueden invocarse en la clase, y no en una instanciación de esa clase.

```
class Thing
  def Thing.hello(name)
    puts "Hello, #{name}!"
  end
end
```

Es equivalente a usar `self` en lugar del nombre de la clase. El siguiente código es equivalente al código anterior:

```
class Thing
  def self.hello(name)
    puts "Hello, #{name}!"
  end
end
```

Invoca el método escribiendo

```
Thing.hello("John Doe")  # prints: "Hello, John Doe!"
```

Métodos singleton

Estos solo están disponibles para instancias específicas de la clase, pero no para todos.

```
# create an empty class
class Thing
end

# two instances of the class
thing1 = Thing.new
thing2 = Thing.new

# create a singleton method
def thing1.makestuff
  puts "I belong to thing one"
end
```

```
thing1.makestuff # => prints: I belong to thing one
thing2.makestuff # NoMethodError: undefined method `makestuff' for #<Thing>
```

Tanto los métodos `singleton` como los de `class` se llaman `eigenclass` es. Básicamente, lo que hace Ruby es crear una clase anónima que contenga dichos métodos para que no interfiera con las instancias que se crean.

Otra forma de hacerlo es mediante la `class <<` constructor. Por ejemplo:

```
# a class method (same as the above example)
class Thing
  class << self # the anonymous class
    def hello(name)
      puts "Hello, #{name}!"
    end
  end
end

Thing.hello("sarah") # => Hello, sarah!

# singleton method

class Thing
end

thing1 = Thing.new

class << thing1
  def makestuff
    puts "I belong to thing one"
  end
end

thing1.makestuff # => prints: "I belong to thing one"
```

Creación dinámica de clases

Las clases se pueden crear dinámicamente mediante el uso de `Class.new` .

```
# create a new class dynamically
MyClass = Class.new

# instantiate an object of type MyClass
my_class = MyClass.new
```

En el ejemplo anterior, se crea una nueva clase y se asigna a la constante `MyClass` . Esta clase puede ser instanciada y utilizada como cualquier otra clase.

El método `Class.new` acepta una `Class` que se convertirá en la superclase de la clase creada dinámicamente.

```
# dynamically create a class that subclasses another
Staffy = Class.new(Dog)
```

```
# instantiate an object of type Staffy
lucky = Staffy.new
lucky.is_a?(Staffy) # true
lucky.is_a?(Dog)    # true
```

El método `Class.new` también acepta un bloque. El contexto del bloque es la clase recién creada. Esto permite definir métodos.

```
Duck =
  Class.new do
    def quack
      'Quack!!'
    end
  end

# instantiate an object of type Duck
duck = Duck.new
duck.quack # 'Quack!!'
```

Nuevo, asignar e inicializar

En muchos idiomas, las nuevas instancias de una clase se crean utilizando una `new` palabra clave especial. En Ruby, `new` también se usa para crear instancias de una clase, pero no es una palabra clave; en cambio, es un método estático / clase, no es diferente de cualquier otro método estático / clase. La definición es aproximadamente esta:

```
class MyClass
   def self.new(*args)
     obj = allocate
     obj.initialize(*args) # oversimplied; initialize is actually private
     obj
   end
end
```

`allocate` realiza la verdadera 'magia' de crear una instancia no inicializada de la clase

Tenga en cuenta también que el valor de retorno de `initialize` se descarta, y se devuelve obj en su lugar. Esto hace que sea inmediatamente claro por qué se puede codificar su método de inicialización sin tener que preocuparse acerca de regresar `self` al final.

El `new` método "normal" que todas las clases obtienen de la `Class` funciona como se indicó anteriormente, pero es posible redefinirlo como desee, o definir alternativas que funcionen de manera diferente. Por ejemplo:

```
class MyClass
  def self.extraNew(*args)
    obj = allocate
    obj.pre_initialize(:foo)
    obj.initialize(*args)
    obj.post_initialize(:bar)
    obj
  end
end
```

Lea Las clases en línea: https://riptutorial.com/es/ruby/topic/264/las-clases

Capítulo 46: Los operadores

Observaciones

Los operadores son métodos

La mayoría de los operadores son en realidad solo métodos, por lo que `x + y` está llamando al método `+` de `x` con el argumento `y` , que se escribiría `x.+(y)` . Si escribe un método propio que tenga un significado semántico de un operador determinado, puede implementar su variante en la clase.

Como un ejemplo tonto:

```
# A class that lets you operate on numbers by name.
class NamedInteger
  name_to_value = { 'one' => 1, 'two' => 2, ... }

  # define the plus method
  def + (left_addend, right_addend)
    name_to_value(left_addend) + name_to_value(right_addend)
  end

  ...
end
```

Cuándo usar `&&` vs. `and` , `||` vs. `or`

Tenga en cuenta que hay dos formas de expresar valores booleanos, ya sea `&&` o `and` , y `||` o `or` - a menudo son intercambiables, pero no siempre. Nos referiremos a ellos como variantes de "carácter" y "palabra".

Las variantes de caracteres tienen mayor *prioridad,* por lo que reducir la necesidad de paréntesis en declaraciones más complejas ayuda a evitar errores inesperados.

Las variantes de la palabra originalmente fueron pensadas como *operadores de flujo de control en* lugar de operadores booleanos. Es decir, fueron diseñados para ser utilizados en declaraciones de métodos encadenados:

```
raise 'an error' and return
```

Si bien *pueden* utilizarse como operadores booleanos, su menor prioridad los hace impredecibles.

En segundo lugar, muchos rubyists prefieren la variante de caracteres cuando crean una expresión booleana (una que se evalúa como `true` o `false`) como `x.nil? || x.empty?` . Por otro lado, las variantes de la palabra se prefieren en los casos en *que* se evalúan una *serie de métodos* , y uno puede fallar. Por ejemplo, una expresión idiomática común que usa la variante de la palabra para los métodos que devuelven `nil` en caso de falla podría ser:

```
def deliver_email
  # If the first fails, try the backup, and if that works, all good
  deliver_by_primary or deliver_by_backup and return
  # error handling code
end
```

Examples

Precedencia y métodos del operador

De mayor a menor, esta es la tabla de precedencia para Ruby. Las operaciones de alta precedencia suceden antes de las operaciones de baja precedencia.

Operators	Operations	Method?
.	Method call (e.g. foo.bar)	
[] []=	Bracket Lookup, Bracket Set	✓[1]
! ~ +	Boolean NOT, complement, unary plus	✓[2]
**	Exponentiation	✓
-	Unary minus	✓[2]
* / %	Multiplication, division, modulo	✓
+ -	Addition, subtraction	✓
<< >>	Bitwise shift	✓
&	Bitwise AND	✓
| ^	Bitwise OR, Bitwise XOR	✓
< <= >= >	Comparison	✓
<=> == != === =~ !~	Equality, pattern matching, comparison	✓[3]
&&	Boolean AND	
||	Boolean OR	
.. ...	Inclusive range, Exclusive range	
? :	Ternary operator	
rescue	Modifier rescue	
= += -=	Assignments	
defined?	Defined operator	
not	Boolean NOT	
or and	Boolean OR, Boolean AND	
if unless while until	Modifier if, unless, while, until	
{ }	Block with braces	
do end	Block with do end	

Unary + y unary - son para `+obj` , `-obj` o `-(some_expression)` .

Modificador-si, modificador-a menos, etc. son para las versiones modificadoras de esas palabras clave. Por ejemplo, esta es una expresión de modificador, a menos que:

```
a += 1 unless a.zero?
```

Los operadores con ✓ pueden definirse como métodos. La mayoría de los métodos se nombran exactamente como se nombra al operador, por ejemplo:

```
class Foo
```

```
  def **(x)
    puts "Raising to the power of #{x}"
  end
  def <<(y)
    puts "Shifting left by #{y}"
  end
  def !
    puts "Boolean negation"
  end
end

Foo.new ** 2     #=> "Raising to the power of 2"
Foo.new << 3     #=> "Shifting left by 3"
!Foo.new         #=> "Boolean negation"
```

[1] Los métodos de Búsqueda de corchetes y Conjunto de corchetes (`[]` y `[]=`) tienen sus argumentos definidos después del nombre, por ejemplo:

```
class Foo
  def [](x)
    puts "Looking up item #{x}"
  end
  def []=(x,y)
    puts "Setting item #{x} to #{y}"
  end
end

f = Foo.new
f[:cats] = 42    #=> "Setting item cats to 42"
f[17]            #=> "Looking up item 17"
```

[2] Los operadores "unary plus" y "unary minus" se definen como métodos denominados `+@` y `-@` , por ejemplo

```
class Foo
  def -@
    puts "unary minus"
  end
  def +@
    puts "unary plus"
  end
end

f = Foo.new
+f               #=> "unary plus"
-f               #=> "unary minus"
```

[3] En las primeras versiones de Ruby, el operador de desigualdad `!=` Y el operador no coincidente `!~` No se podían definir como métodos. En su lugar, se invocó el método para el operador de igualdad correspondiente `==` o el operador coincidente `=~` , y el resultado de ese método fue booleano invertido por Ruby.

Si no define sus propios operadores `!=` O `!~` , El comportamiento anterior sigue siendo cierto. Sin embargo, a partir de Ruby 1.9.1, esos dos operadores también pueden definirse como métodos:

```
class Foo
  def ==(x)
    puts "checking for EQUALITY with #{x}, returning false"
    false
  end
end

f = Foo.new
x = (f == 42)     #=> "checking for EQUALITY with 42, returning false"
puts x            #=> "false"
x = (f != 42)     #=> "checking for EQUALITY with 42, returning false"
puts x            #=> "true"

class Foo
  def !=(x)
    puts "Checking for INequality with #{x}"
  end
end

f != 42           #=> "checking for INequality with 42"
```

Operador de igualdad de casos (===)

También conocido como *triple igual* .

Este operador no prueba la igualdad, sino que prueba si el operando derecho tiene una relación IS A con el operando izquierdo. Como tal, el nombre popular *operador de igualdad de casos* es engañoso.

Esta respuesta SO lo describe así: la mejor manera de describir `a === b` es "si tengo un cajón con la etiqueta `a` , ¿tiene sentido poner `b` en él?" En otras palabras, ¿el conjunto `a` incluye al miembro `b` ?

Ejemplos (fuente)

```
(1..5) === 3            # => true
(1..5) === 6            # => false

Integer === 42          # => true
Integer === 'fourtytwo' # => false

/ell/ === 'Hello'       # => true
/ell/ === 'Foobar'      # => false
```

Clases que anulan `===`

Muchas clases anulan `===` para proporcionar semántica significativa en las declaraciones de casos. Algunos de ellos son:

```
╔═══════════════════╦═══════════════════════╗
║      Class        ║     Synonym for       ║
╠═══════════════════╬═══════════════════════╣
║ Array             ║ ==                    ║
║                   ║                       ║
║ Date              ║ ==                    ║
```

```
╠                  ╠                      ╠
╠ Module           ╠ is_a?                ╠
╠                  ╠                      ╠
╠ Object           ╠ ==                   ╠
╠                  ╠                      ╠
╠ Range            ╠ include?             ╠
╠                  ╠                      ╠
╠ Regexp           ╠ =~                   ╠
╠                  ╠                      ╠
╠ String           ╠ ==                   ╠
╚══════════════════╩══════════════════════╝
```

Práctica recomendada

Se debe evitar el uso explícito del operador de igualdad de casos `===` . No prueba la igualdad sino la *subsunción* , y su uso puede ser confuso. El código es más claro y más fácil de entender cuando se utiliza el método de sinónimos en su lugar.

```
# Bad
Integer === 42
(1..5) === 3
/ell/ === 'Hello'

# Good, uses synonym method
42.is_a?(Integer)
(1..5).include?(3)
/ell/ =~ 'Hello'
```

Operador de navegación segura

Ruby 2.3.0 agregó el *operador de navegación segura* , `&.` . Este operador está destinado a acortar el paradigma de `object && object.property && object.property.method` en sentencias condicionales.

Por ejemplo, tiene un objeto `House` con una propiedad de `address` y desea buscar el `street_name` la `street_name` en la `address` . Para programar esto de manera segura para evitar errores nulos en versiones anteriores de Ruby, usarías un código como este:

```
if house && house.address && house.address.street_name
  house.address.street_name
end
```

El operador de navegación segura acorta esta condición. En su lugar, puede escribir:

```
if house&.address&.street_name
  house.address.street_name
end
```

Precaución:
El operador de navegación segura no tiene *exactamente* el mismo comportamiento que el condicional encadenado. Usando el condicional encadenado (primer ejemplo), el bloque `if` no se ejecutaría si, por ejemplo, la `address` fuera `false` . El operador de navegación segura solo reconoce valores `nil` , pero permite valores como `false` . Si la `address` es `false` , el uso del SNO

producirá un error:

```
house&.address&.street_name
# => undefined method `address' for false:FalseClass
```

Lea Los operadores en línea: https://riptutorial.com/es/ruby/topic/3764/los-operadores

Capítulo 47: Los operadores

Examples

Operadores de comparación

Operador	Descripción
`==`	`true` si los dos valores son iguales.
`!=`	`true` si los dos valores *no* son iguales.
`<`	`true` si el valor del operando a la izquierda es *menor que* el valor a la derecha.
`>`	`true` si el valor del operando a la izquierda es *mayor que* el valor a la derecha.
`>=`	`true` si el valor del operando de la izquierda es *mayor* **o** *igual que* el valor de la derecha.
`<=`	`true` si el valor del operando de la izquierda es *menor* **o** *igual que* el valor de la derecha.
`<=>`	`0` si el valor del operando a la izquierda es *igual al* valor a la derecha, `1` si el valor del operando a la izquierda es *mayor que* el valor a la derecha, `-1` si el valor del operando a la izquierda es *menor que* el valor a la derecha.

Operadores de Asignación

Asignación simple

`=` es una tarea simple. Crea una nueva variable local si la variable no fue referenciada previamente.

```
x = 3
y = 4 + 5
puts "x is #{x}, y is #{y}"
```

Esto dará como resultado:

```
x is 3, y is 9
```

Asignación paralela

Las variables también se pueden asignar en paralelo, por ejemplo `x, y = 3, 9` . Esto es

especialmente útil para intercambiar valores:

```
x, y = 3, 9
x, y = y, x
puts "x is #{x}, y is #{y}"
```

Esto dará como resultado:

```
x is 9, y is 3
```

Asignación abreviada

Es posible mezclar operadores y asignación. Por ejemplo:

```
x = 1
y = 2
puts "x is #{x}, y is #{y}"

x += y
puts "x is now #{x}"
```

Muestra la siguiente salida:

```
x is 1, y is 2
x is now 3
```

Se pueden utilizar varias operaciones en la asignación abreviada:

Operador	Descripción	Ejemplo	Equivalente a
`+=`	Agrega y reasigna la variable.	`x += y`	`x = x + y`
`-=`	Resta y reasigna la variable.	`x -= y`	`x = x - y`
`*=`	Multiplica y reasigna la variable.	`x *= y`	`x = x * y`
`/=`	Divide y reasigna la variable.	`x /= y`	`x = x / y`
`%=`	Divide, toma el resto y reasigna la variable	`x %= y`	`x = x % y`
`**=`	Calcula el exponente y reasigna la variable.	`x **= y`	`x = x ** y`

Lea Los operadores en línea: https://riptutorial.com/es/ruby/topic/3766/los-operadores

Capítulo 48: Metaprogramacion

Introducción

La metaprogramación se puede describir de dos maneras:

"Programas de computadora que escriben o manipulan otros programas (o ellos mismos) como sus datos, o que hacen parte del trabajo en tiempo de compilación que de otra manera se haría en tiempo de ejecución".

En pocas palabras, la **metaprogramación es escribir código que escribe código durante el tiempo de ejecución para hacer su vida más fácil** .

Examples

Implementando "con" usando evaluación de instancia

Muchos lenguajes cuentan `with` una declaración `with` que los programadores pueden omitir el receptor de llamadas a métodos.

`with` puede ser fácilmente emulado en Ruby usando `instance_eval` :

```
def with(object, &block)
  object.instance_eval &block
end
```

El método `with` se puede usar para ejecutar métodos sin problemas en objetos:

```
hash = Hash.new

with hash do
  store :key, :value
  has_key? :key       # => true
  values              # => [:value]
end
```

Definiendo métodos dinámicamente.

Con Ruby puedes modificar la estructura del programa en tiempo de ejecución. Una forma de hacerlo, es definiendo métodos dinámicamente usando el método `method_missing` .

Digamos que queremos poder probar si un número es mayor que otro número con la sintaxis `777.is_greater_than_123?` .

```
# open Numeric class
class Numeric
  # override `method_missing`
  def method_missing(method_name,*args)
```

```
    # test if the method_name matches the syntax we want
    if method_name.to_s.match /^is_greater_than_(\d+)\?$/
      # capture the number in the method_name
      the_other_number = $1.to_i
      # return whether the number is greater than the other number or not
      self > the_other_number
    else
      # if the method_name doesn't match what we want, let the previous definition of
`method_missing` handle it
      super
    end
  end
end
```

¿Algo importante para recordar cuando se usa `method_missing` que también se debe anular `respond_to?` método:

```
class Numeric
   def respond_to?(method_name, include_all = false)
     method_name.to_s.match(/^is_greater_than_(\d+)\?$/) || super
   end
end
```

Olvidarse de hacerlo conduce a una situación inconsistente, cuando puede llamar con éxito `600.is_greater_than_123` , pero `600.respond_to(:is_greater_than_123)` devuelve falso.

Definiendo métodos en instancias.

En ruby puedes agregar métodos a instancias existentes de cualquier clase. Esto le permite agregar comportamiento y una instancia de una clase sin cambiar el comportamiento del resto de las instancias de esa clase.

```
class Example
  def method1(foo)
    puts foo
  end
end

#defines method2 on object exp
exp = Example.new
exp.define_method(:method2) {puts "Method2"}

#with method parameters
exp.define_method(:method3) {|name| puts name}
```

método de enviar ()

`send()` se usa para pasar un mensaje a un `object` . `send()` es un método de instancia de la clase `Object` . El primer argumento en `send()` es el mensaje que está enviando al objeto, es decir, el nombre de un método. Podría ser una `string` o un `symbol` pero se prefieren los **símbolos** . Luego, los argumentos que necesitan pasar en el método, esos serán los argumentos restantes en `send()` .

```
class Hello
  def hello(*args)
    puts 'Hello ' + args.join(' ')
  end
end
h = Hello.new
h.send :hello, 'gentle', 'readers'   #=> "Hello gentle readers"
# h.send(:hello, 'gentle', 'readers') #=> Here :hello is method and rest are the arguments to
method.
```

Aquí está el ejemplo más descriptivo.

```
class Account
  attr_accessor :name, :email, :notes, :address

  def assign_values(values)
    values.each_key do |k, v|
      # How send method would look a like
      # self.name = value[k]
      self.send("#{k}=", values[k])
    end
  end
end

user_info = {
  name: 'Matt',
  email: 'test@gms.com',
  address: '132 random st.',
  notes: "annoying customer"
}

account = Account.new
If attributes gets increase then we would messup the code
#--------- Bad way --------------
account.name = user_info[:name]
account.address = user_info[:address]
account.email = user_info[:email]
account.notes = user_info[:notes]

# --------- Meta Programing way --------------
account.assign_values(user_info) # With single line we can assign n number of attributes

puts account.inspect
```

Nota: `send()` sí ya no se recomienda. Use `__send__()` que tiene el poder de llamar a métodos privados, o (recomendado) `public_send()`

Lea Metaprogramacion en línea: https://riptutorial.com/es/ruby/topic/5023/metaprogramacion

Capítulo 49: método_missing

Parámetros

Parámetro	Detalles
método	El nombre del método que se ha llamado (en el ejemplo anterior es `:say_moo` , tenga en cuenta que este es un símbolo.
* args	Los argumentos pasaron a este método. Puede ser cualquier número, o ninguno
&bloquear	El bloque del método llamado, puede ser un bloque `do` o un bloque encerrado `{ }`

Observaciones

Siempre llama super, en la parte inferior de esta función. Esto guarda un fallo silencioso cuando se llama a algo y no recibe un error.

Por ejemplo, este method_missing va a causar problemas:

```
class Animal
  def method_missing(method, *args, &block)
    say, speak = method.to_s.split("_")
    if say == "say"
      speak
    end
  end
end

=> Animal.new.foobar
=> nil # This should really be raising an error
```

`method_missing` es una buena herramienta para usar cuando sea apropiado, pero tiene dos costos que debe considerar. Primero, `method_missing` es menos eficiente: Ruby debe buscar en la clase y en todos sus ancestros antes de poder recurrir a este enfoque. esta penalización de rendimiento puede ser trivial en un caso simple, pero puede sumar. En segundo lugar, y más ampliamente, esta es una forma de meta-programación que tiene un gran poder que viene con la responsabilidad de garantizar que la implementación sea segura, que maneje adecuadamente las entradas maliciosas, las entradas inesperadas, etc.

También deberías anular la `respond_to_missing?` al igual que:

```
class Animal
  def respond_to_missing?(method, include_private = false)
    method.to_s.start_with?("say_") || super
  end
```

```
end

=> Animal.new.respond_to?(:say_moo) # => true
```

Examples

Atrapar llamadas a un método indefinido

```
class Animal
  def method_missing(method, *args, &block)
    "Cannot call #{method} on Animal"
  end
end

=> Animal.new.say_moo
> "Cannot call say_moo on Animal"
```

Usando el método que falta

```
class Animal
  def method_missing(method, *args, &block)
    say, speak = method.to_s.split("_")
    if say == "say"
      speak
    else
      super
    end
  end
end

=> a = Animal.new
=> a.say_moo
=> "moo"
=> a.shout_moo
=> NoMethodError: undefined method `shout_moo'
```

Usar con bloque

```
class Animal
  def method_missing(method, *args, &block)
    if method.to_s == 'say'
      block.call
    else
      super
    end
  end
end

 => Animal.new.say{ 'moo' }
 => "moo"
```

Utilizar con parametro

```
class Animal
  def method_missing(method, *args, &block)
    say, speak = method.to_s.split("_")
    if say == "say" && speak
      return speak.upcase if args.first == "shout"
      speak
    else
      super
    end
  end
end

=> Animal.new.say_moo
=> "moo"
=> Animal.new.say_moo("shout")
=> "MOO"
```

Lea método_missing en línea: https://riptutorial.com/es/ruby/topic/1076/metodo-missing

Capítulo 50: Métodos

Introducción

Las funciones en Ruby proporcionan un código organizado y reutilizable para realizar un conjunto de acciones. Las funciones simplifican el proceso de codificación, evitan la lógica redundante y hacen que el código sea más fácil de seguir. Este tema describe la declaración y la utilización de funciones, argumentos, parámetros, declaraciones de rendimiento y alcance en Ruby.

Observaciones

Un **método** es un bloque de código con nombre, asociado con uno o más objetos y generalmente identificado por una lista de parámetros además del nombre.

```
def hello(name)
  "Hello, #{name}"
end
```

La invocación de un método especifica el nombre del método, el objeto sobre el que se invoca (a veces se denomina receptor) y cero o más valores de argumento que se asignan a los parámetros del método nombrado. El valor de la última expresión evaluada en el método se convierte en el valor de la expresión de invocación del método.

```
hello("World")
# => "Hello, World"
```

Cuando el receptor no es explícito, es `self` .

```
self
# => main

self.hello("World")
# => "Hello, World"
```

Como se explica en el libro de *lenguaje de programación Ruby* , muchos idiomas distinguen entre funciones, que no tienen un objeto asociado, y métodos, que se invocan en un objeto receptor. Debido a que Ruby es un lenguaje puramente orientado a objetos, todos los métodos son verdaderos métodos y están asociados con al menos un objeto.

Resumen de los parámetros del método

Tipo	Método Firma	Ejemplo de llamada	Asignaciones
R equipara	`def fn(a,b,c)`	`fn(2,3,5)`	`a=2, b=3, c=5`
V ariadic	`def fn(*rest)`	`fn(2,3,5)`	`rest=[2, 3, 5]`

Tipo	Método Firma	Ejemplo de llamada	Asignaciones
D efault	`def fn(a=0,b=1)`	`fn(2,3)`	`a=2, b=3`
K palabra clave	`def fn(a:0,b:1)`	`fn(a:2,b:3)`	`a=2, b=3`

Estos tipos de argumentos pueden combinarse en prácticamente cualquier forma que pueda imaginar para crear funciones variables. El número mínimo de argumentos para la función será igual a la cantidad de argumentos requeridos en la firma. Los argumentos adicionales se asignarán a los parámetros predeterminados primero, luego al `*rest` parámetros.

Tipo	Método Firma	Ejemplo de llamada	Asignaciones
R, D, V, R	`def fn(a,b=1,*mid,z)`	`fn(2,97)`	`a=2, b=1, mid=[], z=97`
		`fn(2,3,97)`	`a=2, b=3, mid=[], z=97`
		`fn(2,3,5,97)`	`a=2, b=3, mid=[5], z=97`
		`fn(2,3,5,7,97)`	`a=2, b=3, mid=[5,7], z=97`
R, K, K	`def fn(a,g:6,h:7)`	`fn(2)`	`a=2, g=6, h=7`
		`fn(2,h:19)`	`a=2, g=6, h=19`
		`fn(2,g:17,h:19)`	`a=2, g=17, h=19`
VK	`def fn(**ks)`	`fn(a:2,g:17,h:19)`	`ks={a:2, g:17, h:19}`
		`fn(four:4,five:5)`	`ks={four:4, five:5}`

Examples

Solo parámetro requerido

```
def say_hello_to(name)
    puts "Hello #{name}"
end
```

```
say_hello_to('Charles')    # Hello Charles
```

Múltiples parámetros requeridos

```
def greet(greeting, name)
    puts "#{greeting} #{name}"
end
```

```
greet('Hi', 'Sophie')    # Hi Sophie
```

Parámetros por defecto

```
def make_animal_sound(sound = 'Cuack')
    puts sound
end
```

```
make_animal_sound('Mooo') # Mooo
make_animal_sound         # Cuack
```

Es posible incluir valores predeterminados para múltiples argumentos:

```
def make_animal_sound(sound = 'Cuack', volume = 11)
    play_sound(sound, volume)
end

make_animal_sound('Mooo') # Spinal Tap cow
```

Sin embargo, no es posible suministrar el segundo sin suministrar también el primero. En lugar de usar parámetros posicionales, pruebe los parámetros de palabras clave:

```
def make_animal_sound(sound: 'Cuack', volume: 11)
    play_sound(sound, volume)
end

make_animal_sound(volume: 1) # Duck whisper
```

O un parámetro hash que almacena opciones:

```
def make_animal_sound(options = {})
    options[:sound]  ||= 'Cuak'
    options[:volume] ||= 11
    play_sound(sound, volume)
end

make_animal_sound(:sound => 'Mooo')
```

Los valores predeterminados de los parámetros pueden ser establecidos por cualquier expresión ruby. La expresión se ejecutará en el contexto del método, por lo que incluso puede declarar variables locales aquí. Tenga en cuenta, no obtendrá a través de la revisión de código. Cortesía de Caius por señalar esto .

```
def make_animal_sound( sound = ( raise 'TUU-too-TUU-too...' ) ); p sound; end

make_animal_sound 'blaaaa' # => 'blaaaa'
make_animal_sound          # => TUU-too-TUU-too... (RuntimeError)
```

Parámetros opcionales (operador splat)

```
def welcome_guests(*guests)
    guests.each { |guest| puts "Welcome #{guest}!" }
end
```

```
welcome_guests('Tom')    # Welcome Tom!
welcome_guests('Rob', 'Sally', 'Lucas') # Welcome Rob!
                                        # Welcome Sally!
                                        # Welcome Lucas!
```

Tenga en cuenta que `welcome_guests(['Rob', 'Sally', 'Lucas'])` emitirán `Welcome ["Rob", "Sally", "Lucas"]!`
En cambio, si tiene una lista, puede hacer `welcome_guests(*['Rob', 'Sally', 'Lucas'])` y eso funcionará como `welcome_guests('Rob', 'Sally', 'Lucas')` .

Mezcla opcional de parámetros por defecto requerida

```
def my_mix(name,valid=true, *opt)
    puts name
    puts valid
    puts opt
end
```

Llame de la siguiente manera:

```
my_mix('me')
# 'me'
# true
# []

my_mix('me', false)
# 'me'
# false
# []

my_mix('me', true, 5, 7)
# 'me'
# true
# [5,7]
```

Las definiciones de los métodos son expresiones

La definición de un método en Ruby 2.x devuelve un símbolo que representa el nombre:

```
class Example
  puts def hello
  end
end

#=> :hello
```

Esto permite interesantes técnicas de metaprogramación. Por ejemplo, los métodos pueden ser

envueltos por otros métodos:

```
class Class
  def logged(name)
    original_method = instance_method(name)
    define_method(name) do |*args|
      puts "Calling #{name} with #{args.inspect}."
      original_method.bind(self).call(*args)
      puts "Completed #{name}."
    end
  end
end

class Meal
  def initialize
    @food = []
  end

  logged def add(item)
    @food << item
  end
end

meal = Meal.new
meal.add "Coffee"
# Calling add with ["Coffee"].
# Completed add.
```

Capturando argumentos de palabras clave no declarados (doble splat)

El operador `**` funciona de manera similar al operador `*` , pero se aplica a los parámetros de palabras clave.

```
def options(required_key:, optional_key: nil, **other_options)
  other_options
end

options(required_key: 'Done!', foo: 'Foo!', bar: 'Bar!')
#> { :foo => "Foo!", :bar => "Bar!" }
```

En el ejemplo anterior, si no se utilizan las `**other_options` se `**other_options` un `ArgumentError: unknown keyword: foo, bar` error de `ArgumentError: unknown keyword: foo, bar` .

```
def without_double_splat(required_key:, optional_key: nil)
  # do nothing
end

without_double_splat(required_key: 'Done!', foo: 'Foo!', bar: 'Bar!')
#> ArgumentError: unknown keywords: foo, bar
```

Esto es útil cuando tiene un hash de opciones que quiere pasar a un método y no quiere filtrar las claves.

```
def options(required_key:, optional_key: nil, **other_options)
  other_options
```

```
end

my_hash = { required_key: true, foo: 'Foo!', bar: 'Bar!' }

options(my_hash)
#> { :foo => "Foo!", :bar => "Bar!" }
```

También es posible *desempaquetar* un hash usando el operador `**` . Esto le permite suministrar palabras clave directamente a un método además de los valores de otros hashes:

```
my_hash = { foo: 'Foo!', bar: 'Bar!' }

options(required_key: true, **my_hash)
#> { :foo => "Foo!", :bar => "Bar!" }
```

Cediendo a bloques

Puede enviar un bloque a su método y puede llamar a ese bloque varias veces. Esto se puede hacer enviando un proc / lambda o similar, pero es más fácil y más rápido con el `yield` :

```
def simple(arg1,arg2)
  puts "First we are here:  #{arg1}"
  yield
  puts "Finally we are here:  #{arg2}"
  yield
end
simple('start','end') { puts "Now we are inside the yield" }

#> First we are here:  start
#> Now we are inside the yield
#> Finally we are here:  end
#> Now we are inside the yield
```

Tenga en cuenta que `{ puts ... }` no está dentro de los paréntesis, implícitamente viene después. Esto también significa que solo podemos tener un bloque de `yield` . Podemos pasar argumentos al `yield` :

```
def simple(arg)
  puts "Before yield"
  yield(arg)
  puts "After yield"
end
simple('Dave') { |name| puts "My name is #{name}" }

#> Before yield
#> My name is Dave
#> After yield
```

Con el rendimiento podemos hacer fácilmente iteradores o cualquier función que funcione en otro código:

```
def countdown(num)
  num.times do |i|
```

```
    yield(num-i)
  end
end
```

```
countdown(5) { |i| puts "Call number #{i}" }
```

```
#> Call number 5
#> Call number 4
#> Call number 3
#> Call number 2
#> Call number 1
```

De hecho, es con `yield` que cosas como `foreach` , `each` y `each times` se implementan generalmente en las clases.

Si desea averiguar si le han dado un bloqueo o no, use `block_given?` :

```
class Employees
  def names
    ret = []
    @employees.each do |emp|
      if block_given?
        yield(emp.name)
      else
        ret.push(emp.name)
      end
    end

    ret
  end
end
```

Este ejemplo asume que la clase `Employees` tiene una lista de `@employees` que se puede iterar con `each` para obtener objetos que tienen nombres de empleados usando el método del `name` . Si nos dan un bloque, entonces vamos a `yield` el nombre al bloque, de lo contrario, simplemente empujar a una matriz que volvamos.

Argumentos de la tupla

Un método puede tomar un parámetro de matriz y destruirlo inmediatamente en variables locales nombradas. Encontrado en el blog de Mathias Meyer .

```
def feed( amount, (animal, food) )

    p "#{amount} #{animal}s chew some #{food}"

end

feed 3, [ 'rabbit', 'grass' ] # => "3 rabbits chew some grass"
```

Definiendo un método

Los métodos se definen con la palabra clave `def` , seguidos del *nombre* del *método* y una lista

opcional de *nombres* de *parámetros* entre paréntesis. El código de Ruby entre `def` y `end` representa el *cuerpo* del método.

```
def hello(name)
  "Hello, #{name}"
end
```

La invocación de un método especifica el nombre del método, el objeto sobre el que se invoca (a veces se denomina receptor) y cero o más valores de argumento que se asignan a los parámetros del método nombrado.

```
hello("World")
# => "Hello, World"
```

Cuando el receptor no es explícito, es `self` .

Los nombres de los parámetros se pueden usar como variables dentro del cuerpo del método, y los valores de estos parámetros nombrados provienen de los argumentos a la invocación de un método.

```
hello("World")
# => "Hello, World"
hello("All")
# => "Hello, All"
```

Usa una función como bloque.

Muchas funciones en Ruby aceptan un bloque como argumento. P.ej:

```
[0, 1, 2].map {|i| i + 1}
 => [1, 2, 3]
```

Si ya tiene una función que hace lo que quiere, puede convertirla en un bloque usando `&method(:fn)` :

```
def inc(num)
   num + 1
end

[0, 1, 2].map &method(:inc)
 => [1, 2, 3]
```

Lea Métodos en línea: https://riptutorial.com/es/ruby/topic/997/metodos

Capítulo 51: Modificadores de acceso Ruby

Introducción

Control de acceso (alcance) a varios métodos, miembros de datos, métodos de inicialización.

Examples

Variables de instancia y variables de clase

Primero repasemos cuáles son las **variables de instancia: se** comportan más como propiedades para un objeto. Se inicializan en la creación de un objeto. Las variables de instancia son accesibles a través de métodos de instancia. Por objeto tiene variables por instancia. Las variables de instancia no se comparten entre objetos.

La clase de secuencia tiene @from, @to y @by como variables de instancia.

```
class Sequence
    include Enumerable

    def initialize(from, to, by)
        @from = from
        @to = to
        @by = by
    end

    def each
        x = @from
        while x < @to
            yield x
            x = x + @by
        end
    end

    def *(factor)
        Sequence.new(@from*factor, @to*factor, @by*factor)
    end

    def +(offset)
        Sequence.new(@from+offset, @to+offset, @by+offset)
    end
end

object = Sequence.new(1,10,2)
object.each do |x|
    puts x
end

Output:
1
3
5
7
```

```
9

object1 = Sequence.new(1,10,3)
object1.each do |x|
    puts x
end

Output:
1
4
7
```

Variables de clase Trate la variable de clase igual que las variables estáticas de java, que se comparten entre los diversos objetos de esa clase. Las variables de clase se almacenan en la memoria del montón.

```
class Sequence
    include Enumerable
    @@count = 0
    def initialize(from, to, by)
        @from = from
        @to = to
        @by = by
        @@count = @@count + 1
    end

    def each
        x = @from
        while x < @to
            yield x
            x = x + @by
        end
    end

    def *(factor)
        Sequence.new(@from*factor, @to*factor, @by*factor)
    end

    def +(offset)
        Sequence.new(@from+offset, @to+offset, @by+offset)
    end

    def getCount
        @@count
    end
end

object = Sequence.new(1,10,2)
object.each do |x|
    puts x
end

Output:
1
3
5
7
9
```

```
object1 = Sequence.new(1,10,3)
object1.each do |x|
    puts x
end

Output:
1
4
7

puts object1.getCount
Output: 2
```

Compartido entre objeto y objeto1.

Comparando las variables de instancia y clase de Ruby con Java:

```
Class Sequence{
    int from, to, by;
    Sequence(from, to, by){// constructor method of Java is equivalent to initialize method of
ruby
        this.from = from;// this.from of java is equivalent to @from indicating
currentObject.from
        this.to = to;
        this.by = by;
    }
    public void each(){
        int x = this.from;//objects attributes are accessible in the context of the object.
        while x > this.to
            x = x + this.by
    }
}
```

Controles de acceso

Comparación de los controles de acceso de Java contra Ruby: si el método se declara privado en Java, solo se puede acceder a él mediante otros métodos dentro de la misma clase. Si un método se declara protegido, puede ser accedido por otras clases que existen dentro del mismo paquete, así como por subclases de la clase en un paquete diferente. Cuando un método es público es visible para todos. En Java, el concepto de visibilidad de control de acceso depende de dónde residen estas clases en la jerarquía de herencia / paquete.

Mientras que en Ruby, la jerarquía de herencia o el paquete / módulo no encajan. Se trata de qué objeto es el receptor de un método .

Para un método privado en Ruby , nunca se puede llamar con un receptor explícito. Podemos (solo) llamar al método privado con un receptor implícito.

Esto también significa que podemos llamar a un método privado desde una clase en la que se declara, así como a todas las subclases de esta clase.

```
class Test1
  def main_method
    method_private
```

```
  end

  private
  def method_private
    puts "Inside methodPrivate for #{self.class}"
  end
end

class Test2 < Test1
  def main_method
    method_private
  end
end

Test1.new.main_method
Test2.new.main_method

Inside methodPrivate for Test1
Inside methodPrivate for Test2

class Test3 < Test1
  def main_method
    self.method_private #We were trying to call a private method with an explicit receiver and
if called in the same class with self would fail.
  end
end

Test1.new.main_method
This will throw NoMethodError

You can never call the private method from outside the class hierarchy where it was defined.
```

El método protegido se puede llamar con un receptor implícito, como privado. Además, el método protegido también puede ser llamado por un receptor explícito (solo) si el receptor es "self" o "un objeto de la misma clase".

```
class Test1
  def main_method
    method_protected
  end

  protected
  def method_protected
    puts "InSide method_protected for #{self.class}"
  end
end

class Test2 < Test1
  def main_method
    method_protected # called by implicit receiver
  end
end

class Test3 < Test1
  def main_method
    self.method_protected # called by explicit receiver "an object of the same class"
  end
end
```

```
InSide method_protected for Test1
InSide method_protected for Test2
InSide method_protected for Test3


class Test4 < Test1
  def main_method
    Test2.new.method_protected # "Test2.new is the same type of object as self"
  end
end

Test4.new.main_method

class Test5
  def main_method
    Test2.new.method_protected
  end
end

Test5.new.main_method
This would fail as object Test5 is not subclass of Test1
```

Considerar métodos públicos con la máxima visibilidad.

Resumen

1. **Público: los** métodos públicos tienen máxima visibilidad.
2. **Protegido: el método protegido** se puede llamar con un receptor implícito, como privado. Además, el método protegido también puede ser llamado por un receptor explícito (solo) si el receptor es "self" o "un objeto de la misma clase".
3. **Privado: para un método privado en Ruby** , nunca se puede llamar con un receptor explícito. Podemos (solo) llamar al método privado con un receptor implícito. Esto también significa que podemos llamar a un método privado desde una clase en la que se declara, así como a todas las subclases de esta clase.

Lea Modificadores de acceso Ruby en línea:
https://riptutorial.com/es/ruby/topic/10797/modificadores-de-acceso-ruby

Capítulo 52: Módulos

Sintaxis

- Declaración

```
module Name;

    any ruby expressions;

end
```

Observaciones

Los nombres de los módulos en Ruby son constantes, por lo que deben comenzar con una letra mayúscula.

```
module foo; end # Syntax error: class/module name must be CONSTANT
```

Examples

Una mezcla simple con incluir

```
module SomeMixin
  def foo
    puts "foo!"
  end
end

class Bar
  include SomeMixin
  def baz
    puts "baz!"
  end
end

b = Bar.new
b.baz          # => "baz!"
b.foo          # => "foo!"
# works thanks to the mixin
```

Ahora `Bar` es una mezcla de sus propios métodos y los métodos de `SomeMixin` .

Tenga en cuenta que la forma en que se utiliza una mezcla en una clase depende de cómo se agrega:

- la palabra clave `include` evalúa el código del módulo en el contexto de la clase (por ejemplo, las definiciones de métodos serán métodos en instancias de la clase),

- `extend` evaluará el código del módulo en el contexto de la clase singleton del objeto (los métodos están disponibles directamente en el objeto extendido).

Módulo como espacio de nombres

Los módulos pueden contener otros módulos y clases:

```
module Namespace

    module Child

        class Foo; end

    end # module Child

    # Foo can now be accessed as:
    #
    Child::Foo

end # module Namespace

# Foo must now be accessed as:
#
Namespace::Child::Foo
```

Una mezcla simple con extender

Un mixin es solo un módulo que se puede agregar (mezclar) a una clase. Una forma de hacerlo es con el método extendido. El método de `extend` agrega métodos de la mezcla como métodos de clase.

```
module SomeMixin
  def foo
    puts "foo!"
  end
end

class Bar
  extend SomeMixin
  def baz
    puts "baz!"
  end
end

b = Bar.new
b.baz         # => "baz!"
b.foo         # NoMethodError, as the method was NOT added to the instance
Bar.foo       # => "foo!"
# works only on the class itself
```

Módulos y composición de clase

Puedes usar módulos para construir clases más complejas a través de la *composición* . La directiva `include ModuleName` incorpora los métodos de un módulo en una clase.

```
module Foo
  def foo_method
    puts 'foo_method called!'
  end
end

module Bar
  def bar_method
    puts 'bar_method called!'
  end
end

class Baz
  include Foo
  include Bar

  def baz_method
    puts 'baz_method called!'
  end
end
```

`Baz` ahora contiene métodos tanto de `Foo` como de `Bar` además de sus propios métodos.

```
new_baz = Baz.new
new_baz.baz_method #=> 'baz_method called!'
new_baz.bar_method #=> 'bar_method called!'
new_baz.foo_method #=> 'foo_method called!'
```

Lea Módulos en línea: https://riptutorial.com/es/ruby/topic/4039/modulos

Capítulo 53: Números

Observaciones

Jerarquía de números

Ruby incluye varias clases incorporadas para representar números:

```
Numeric
  Integer
    Fixnum    # 1
    Bignum    # 10000000000000000000
  Float       # 1.0
  Complex     # (1+0i)
  Rational    # Rational(2, 3) == 2/3
  BigDecimal  # not loaded by default
```

Los más comunes son:

- `Fixnum` para representar, por ejemplo, enteros positivos y negativos
- `Float` para representar números de punto flotante

`BigDecimal` es el único que no está cargado por defecto. Puedes cargarlo con:

```
require "bigdecimal"
```

Tenga en cuenta que en ruby 2.4+, `Fixnum` y `Bignum` están unificados; *todos los* enteros son ahora solo miembros de la clase `Integer` . Para compatibilidad con versiones anteriores, `Fixnum == Bignum == Integer` .

Examples

Creando un entero

```
0       # creates the Fixnum 0
123     # creates the Fixnum 123
1_000   # creates the Fixnum 1000. You can use _ as separator for readability
```

Por defecto, la notación es la base 10. Sin embargo, hay otras notaciones incorporadas para diferentes bases:

```
0xFF    # Hexadecimal representation of 255, starts with a 0x
0b100   # Binary representation of 4, starts with a 0b
0555    # Octal representation of 365, starts with a 0 and digits
```

Convertir una cadena a entero

Puede utilizar el método `Integer` para convertir una `String` en un `Integer` :

```
Integer("123")      # => 123
Integer("0xFF")     # => 255
Integer("0b100")    # => 4
Integer("0555")     # => 365
```

También puede pasar un parámetro base al método `Integer` para convertir números de una base determinada

```
Integer('10', 5)    # => 5
Integer('74', 8)    # => 60
Integer('NUM', 36)  # => 30910
```

Tenga en cuenta que el método genera un `ArgumentError` si el parámetro no se puede convertir:

```
Integer("hello")
# raises ArgumentError: invalid value for Integer(): "hello"
Integer("23-hello")
# raises ArgumentError: invalid value for Integer(): "23-hello"
```

También puede utilizar el método `String#to_i` . Sin embargo, este método es ligeramente más permisivo y tiene un comportamiento diferente al de `Integer` :

```
"23".to_i         # => 23
"23-hello".to_i   # => 23
"hello".to_i      # => 0
```

`String#to_i` acepta un argumento, la base para interpretar el número como:

```
"10".to_i(2) # => 2
"10".to_i(3) # => 3
"A".to_i(16) # => 10
```

Convertir un número en una cadena

Fixnum # to_s toma un argumento base opcional y representa el número dado en esa base:

```
2.to_s(2)   # => "10"
3.to_s(2)   # => "11"
3.to_s(3)   # => "10"
10.to_s(16) # => "a"
```

Si no se proporciona ningún argumento, entonces representa el número en base 10

```
2.to_s # => "2"
10423.to_s # => "10423"
```

Dividiendo dos numeros

Al dividir dos números, preste atención al tipo que desea a cambio. Tenga en cuenta que dividir **dos enteros invocará la división de enteros** . Si su objetivo es ejecutar la división de flotación, al menos uno de los parámetros debe ser de tipo `float` .

División entera:

```
3 / 2 # => 1
```

División de flotación

```
3 / 3.0 # => 1.0

16 / 2 / 2    # => 4
16 / 2 / 2.0  # => 4.0
16 / 2.0 / 2  # => 4.0
16.0 / 2 / 2  # => 4.0
```

Numeros racionales

`Rational` representa un número racional como numerador y denominador:

```
r1 = Rational(2, 3)
r2 = 2.5.to_r
r3 = r1 + r2
r3.numerator   # => 19
r3.denominator # => 6
Rational(2, 4) # => (1/2)
```

Otras formas de crear un racional

```
Rational('2/3')  # => (2/3)
Rational(3)      # => (3/1)
Rational(3, -5)  # => (-3/5)
Rational(0.2)    # => (3602879701896397/18014398509481984)
Rational('0.2')  # => (1/5)
0.2.to_r         # => (3602879701896397/18014398509481984)
0.2.rationalize  # => (1/5)
'1/4'.to_r       # => (1/4)
```

Números complejos

```
1i     # => (0+1i)
1.to_c # => (1+0i)
rectangular = Complex(2, 3)  # => (2+3i)
polar       = Complex('1@2') # => (-0.4161468365471424+0.9092974268256817i)

polar.rectangular # => [-0.4161468365471424, 0.9092974268256817]
rectangular.polar # => [3.605551275463989, 0.982793723247329]
rectangular + polar # => (1.5838531634528576+3.909297426825682i)
```

Números pares e impares

El `even?` El método puede usarse para determinar si un número es par

```
4.even?      # => true
5.even?      # => false
```

Lo `odd?` El método puede usarse para determinar si un número es impar.

```
4.odd?       # => false
5.odd?       # => true
```

Redondear numeros

La `round` método redondear un número hasta si el primer dígito después de su lugar decimal es 5 o superior y redondear hacia abajo si ese dígito es 4 o inferior. Esto incluye un argumento opcional para la precisión que está buscando.

```
4.89.round        # => 5
4.25.round        # => 4
3.141526.round(1) # => 3.1
3.141526.round(2) # => 3.14
3.141526.round(4) # => 3.1415
```

Los números de punto flotante también se pueden redondear al número entero más bajo que el número con el método de `floor`

```
4.9999999999999.floor # => 4
```

También se pueden redondear hasta el número entero más bajo que el número utilizando el método `ceil`

```
4.0000000000001.ceil  # => 5
```

Lea Números en línea: https://riptutorial.com/es/ruby/topic/1083/numeros

Capítulo 54: Operaciones de archivo y E / S

Parámetros

Bandera	Sentido
"r"	Solo lectura, comienza al principio del archivo (modo predeterminado).
"r +"	Lectura-escritura, comienza al principio del archivo.
"w"	Solo escritura, trunca el archivo existente a longitud cero o crea un nuevo archivo para escribir.
"w +"	Leer-escribir, trunca el archivo existente a longitud cero o crea un nuevo archivo para leer y escribir.
"una"	Solo escritura, comienza al final del archivo si el archivo existe, de lo contrario crea un nuevo archivo para escribir.
"a +"	Lectura-escritura, comienza al final del archivo si el archivo existe, de lo contrario crea un nuevo archivo para leer y escribir.
"segundo"	Modo de archivo binario. Suprime la conversión EOL <-> CRLF en Windows. Y establece la codificación externa en ASCII-8BIT a menos que se especifique explícitamente. (Este indicador solo puede aparecer junto con los indicadores anteriores. Por ejemplo, `File.new("test.txt", "rb")` abriría `test.txt` en modo de `read-only` como un archivo `binary`).
"t"	Modo de archivo de texto. (Este indicador solo puede aparecer junto con los indicadores anteriores. Por ejemplo, `File.new("test.txt", "wt")` abriría `test.txt` en modo de `write-only` como un archivo de `text`).

Examples

Escribir una cadena en un archivo

Se puede escribir una cadena en un archivo con una instancia de la clase de `File` .

```
file = File.new('tmp.txt', 'w')
file.write("NaNaNaNa\n")
file.write('Batman!\n')
file.close
```

La clase `File` también ofrece una forma abreviada de las operaciones `new` y `close` con el método `open` .

```
File.open('tmp.txt', 'w') do |f|
  f.write("NaNaNaNa\n")
  f.write('Batman!\n')
end
```

Para operaciones de escritura simples, una cadena también se puede escribir directamente en un archivo con `File.write` . **Tenga en cuenta que esto sobrescribirá el archivo de forma predeterminada.**

```
File.write('tmp.txt', "NaNaNaNa\n" * 4 + 'Batman!\n')
```

Para especificar un modo diferente en `File.write` , páselo como el valor de una clave llamada `mode` en un hash como otro parámetro.

```
File.write('tmp.txt', "NaNaNaNa\n" * 4 + 'Batman!\n', { mode: 'a'})
```

Abrir y cerrar un archivo

Abrir y cerrar manualmente un archivo.

```
# Using new method
f = File.new("test.txt", "r") # reading
f = File.new("test.txt", "w") # writing
f = File.new("test.txt", "a") # appending

# Using open method
f = open("test.txt", "r")

# Remember to close files
f.close
```

Cerrar automáticamente un archivo utilizando un bloque.

```
f = File.open("test.txt", "r") do |f|
  # do something with file f
  puts f.read # for example, read it
end
```

obtener una sola char de entrada

A diferencia de `gets.chomp` esto no esperará una nueva línea.

La primera parte del stdlib debe ser incluida

```
require 'io/console'
```

Entonces se puede escribir un método auxiliar:

```
def get_char
  input = STDIN.getch
```

```
  control_c_code = "\u0003"
  exit(1) if input == control_c_code
  input
end
```

Es importante salir si se presiona `control+c` .

Lectura de STDIN

```
# Get two numbers from STDIN, separated by a newline, and output the result
number1 = gets
number2 = gets
puts number1.to_i + number2.to_i
## run with: $ ruby a_plus_b.rb
## or:       $ echo -e "1\n2" | ruby a_plus_b.rb
```

Leyendo de argumentos con ARGV

```
number1 = ARGV[0]
number2 = ARGV[1]
puts number1.to_i + number2.to_i
## run with: $ ruby a_plus_b.rb 1 2
```

Lea Operaciones de archivo y E / S en línea:
https://riptutorial.com/es/ruby/topic/4310/operaciones-de-archivo-y-e---s

Capítulo 55: Operador Splat (*)

Examples

Coaccionar matrices en la lista de parámetros

Supongamos que tienes una matriz:

```
pair = ['Jack','Jill']
```

Y un método que toma dos argumentos:

```
def print_pair (a, b)
  puts "#{a} and #{b} are a good couple!"
end
```

Podría pensar que podría pasar la matriz:

```
print_pair(pair) # wrong number of arguments (1 for 2) (ArgumentError)
```

Dado que la matriz es solo un argumento, no dos, Ruby lanza una excepción. Se *podía* sacar cada elemento de forma individual:

```
print_pair(pair[0], pair[1])
```

O puede usar el operador splat para ahorrarse algo de esfuerzo:

```
print_pair(*pair)
```

Número variable de argumentos

El operador splat elimina elementos individuales de una matriz y los convierte en una lista. Esto se usa más comúnmente para crear un método que acepta un número variable de argumentos:

```
# First parameter is the subject and the following parameters are their spouses
def print_spouses(person, *spouses)
  spouses.each do |spouse|
    puts "#{person} married #{spouse}."
  end
end

print_spouses('Elizabeth', 'Conrad', 'Michael', 'Mike', 'Eddie', 'Richard', 'John', 'Larry')
```

Tenga en cuenta que una matriz solo cuenta como un elemento en la lista, por lo que también necesitará utilizar el operador splat en el lado de la llamada si tiene una matriz que desea aprobar:

```
bonaparte = ['Napoleon','Joséphine','Marie Louise']
print_spouses(*bonaparte)
```

Lea Operador Splat (*) en línea: https://riptutorial.com/es/ruby/topic/9862/operador-splat----

Capítulo 56: OptionParser

Introducción

OptionParser se puede usar para analizar las opciones de línea de comandos de `ARGV` .

Examples

Opciones de línea de comando obligatorias y opcionales

Es relativamente fácil analizar la línea de comandos a mano si no está buscando algo demasiado complejo:

```
# Naive error checking
abort('Usage: ' + $0 + ' site id ...') unless ARGV.length >= 2

# First item (site) is mandatory
site = ARGV.shift

ARGV.each do | id |
  # Do something interesting with each of the ids
end
```

Pero cuando sus opciones comienzan a complicarse, probablemente necesitará usar un analizador de opciones como, bueno, OptionParser :

```
require 'optparse'

# The actual options will be stored in this hash
options = {}

# Set up the options you are looking for
optparse = OptionParser.new do |opts|
  opts.banner = "Usage: #{$0} -s NAME id ..."

  opts.on("-s", "--site NAME", "Site name") do |s|
    options[:site] = s
  end

  opts.on( '-h', '--help', 'Display this screen' ) do
    puts opts
    exit
  end
end

# The parse! method also removes any options it finds from ARGV.
optparse.parse!
```

También hay un `parse` no destructivo, pero es mucho menos útil si planeas usar el resto de lo que está en `ARGV` .

La clase OptionParser no tiene una manera de imponer argumentos obligatorios (como `--site` en este caso). Sin embargo, usted puede hacer su propia comprobación después de ejecutar `parse!` :

```
# Slightly more sophisticated error checking
if options[:site].nil? or ARGV.length == 0
  abort(optparse.help)
end
```

Para un controlador de opciones obligatorio más genérico, vea esta respuesta . En caso de que no quede claro, todas las opciones son opcionales, a menos que haga un esfuerzo adicional para hacerlas obligatorias.

Valores predeterminados

Con `OptionsParser` , es muy fácil configurar los valores predeterminados. Simplemente rellene previamente el hash en el que almacena las opciones en:

```
options = {
  :directory => ENV['HOME']
}
```

Cuando defina el analizador, sobrescribirá el valor predeterminado si un usuario proporciona un valor:

```
OptionParser.new do |opts|
  opts.on("-d", "--directory HOME", "Directory to use") do |d|
    options[:directory] = d
  end
end
```

Descripciones largas

A veces tu descripción puede llegar a ser bastante larga. Por ejemplo, `irb -h` enumera en un argumento que dice:

```
  --context-mode n  Set n[0-3] to method to create Binding Object,
                    when new workspace was created
```

No está claro de inmediato cómo apoyar esto. La mayoría de las soluciones requieren ajustes para que la sangría de la segunda línea y las siguientes se alineen con la primera. Afortunadamente, el método `on` admite múltiples líneas de descripción al agregarlas como argumentos separados:

```
  opts.on("--context-mode n",
          "Set n[0-3] to method to create Binding Object,",
          "when new workspace was created") do |n|
    optons[:context_mode] = n
  end
```

Puede agregar tantas líneas de descripción como desee para explicar completamente la opción.

Lea OptionParser en línea: https://riptutorial.com/es/ruby/topic/9860/optionparser

Capítulo 57: Parches de mono en rubí

Introducción

Monkey Patching es una forma de modificar y extender clases en Ruby. Básicamente, puedes modificar las clases ya definidas en Ruby, agregar nuevos métodos e incluso modificar métodos previamente definidos.

Observaciones

Los parches de mono a menudo se usan para cambiar el comportamiento del código ruby existente, por ejemplo de gemas.

Por ejemplo, ver esta esencia .

También se puede usar para ampliar las clases de ruby existentes como Rails lo hace con ActiveSupport, aquí hay un ejemplo de eso .

Examples

Cambiando cualquier método

```
def hello
  puts "Hello readers"
end

hello # => "Hello readers"

def hello
  puts "Hell riders"
end

hello # => "Hell riders"
```

Cambiando un método de rubí existente

```
puts "Hello readers".reverse # => "sredaer olleH"

class String
  def reverse
    "Hell riders"
  end
end

puts "Hello readers".reverse # => "Hell riders"
```

Cambiar un método con parámetros

Puede acceder al mismo contexto exacto que el método que reemplaza.

```
class Boat
  def initialize(name)
    @name = name
  end

  def name
    @name
  end
end

puts Boat.new("Doat").name # => "Doat"

class Boat
  def name
    "□ #{@name} □"
  end
end

puts Boat.new("Moat").name # => "□ Moat □"
```

Extendiendo una clase existente

```
class String
  def fancy
    "~~~{#{self}}~~~"
  end
end

puts "Dorian".fancy # => "~~~{Dorian}~~~"
```

Parches Safe Monkey con Refinamientos

Desde Ruby 2.0, Ruby permite tener revisiones de mono más seguras con mejoras. Básicamente, permite limitar el código de Monkey Patched para que solo se aplique cuando se solicita.

Primero creamos un refinamiento en un módulo:

```
module RefiningString
  refine String do
    def reverse
      "Hell riders"
    end
  end
end
```

Entonces podemos decidir dónde usarlo:

```
class AClassWithoutMP
  def initialize(str)
    @str = str
  end

  def reverse
```

```
    @str.reverse
  end
end

class AClassWithMP
  using RefiningString

  def initialize(str)
    @str = str
  end

  def reverse
    str.reverse
  end
end

AClassWithoutMP.new("hello".reverse # => "olle"
AClassWithMP.new("hello").reverse # "Hell riders"
```

Lea Parches de mono en rubí en línea: https://riptutorial.com/es/ruby/topic/6043/parches-de-mono-en-rubi

Capítulo 58: Parches de mono en rubí

Examples

Mono parcheando una clase

El parche del mono es la modificación de clases u objetos fuera de la misma clase.

A veces es útil agregar funcionalidad personalizada.

Ejemplo: Omitir clase de cadena para proporcionar análisis a booleano

```
class String
  def to_b
    self =~ (/^(true|TRUE|True|1)$/i) ? true : false
  end
end
```

Como puede ver, agregamos el método `to_b()` a la clase String, por lo que podemos analizar cualquier cadena a un valor booleano.

```
>>'true'.to_b
=> true
>>'foo bar'.to_b
=> false
```

Mono parcheando un objeto

Al igual que el parcheo de clases, también puede parchear objetos individuales. La diferencia es que solo una instancia puede usar el nuevo método.

Ejemplo: anular un objeto de cadena para proporcionar análisis a booleano

```
s = 'true'
t = 'false'

def s.to_b
  self =~ /true/ ? true : false
end

>> s.to_b
=> true
>> t.to_b
=> undefined method `to_b' for "false":String (NoMethodError)
```

Lea Parches de mono en rubí en línea: https://riptutorial.com/es/ruby/topic/6228/parches-de-mono-en-rubi

Capítulo 59: Parches de mono en rubí

Observaciones

El parche del mono, aunque es conveniente, tiene algunas trampas que no son evidentes de inmediato. En particular, un parche como ese en el ejemplo contamina el alcance global. Si los dos módulos añaden `Hash#symbolize` , solo el último módulo requerido aplica su cambio; el resto son borrados.

Además, si hay un error en un método parcheado, el seguimiento de la pila simplemente apunta a la clase parcheada. Esto implica que hay un error en la clase `Hash` (que ahora existe).

Por último, dado que Ruby es muy flexible con los contenedores para guardar, un método que parece muy sencillo cuando lo escribes tiene muchas funciones no definidas. Por ejemplo, crear `Array#sum` es bueno para una matriz de números, pero se interrumpe cuando se le da una matriz de una clase personalizada.

Una alternativa más segura son los refinamientos, disponibles en Ruby> = 2.0.

Examples

Agregando Funcionalidad

Puedes agregar un método a cualquier clase en Ruby, ya sea integrado o no. Se hace referencia al objeto llamante usando `self` .

```
class Fixnum
  def plus_one
    self + 1
  end

  def plus(num)
    self + num
  end

  def concat_one
    self.to_s + '1'
  end
end

1.plus_one # => 2
3.plus(5) # => 8
6.concat_one # => '61'
```

Lea Parches de mono en rubí en línea: https://riptutorial.com/es/ruby/topic/6616/parches-de-mono-en-rubi

Capítulo 60: Paso de mensajes

Examples

Introducción

En el *diseño orientado a* objetos, los objetos *reciben* mensajes y *responden* a ellos. En Ruby, enviar un mensaje está *llamando a un método* y el resultado de ese método es la respuesta.

En Ruby el paso de mensajes es dinámico. Cuando llega un mensaje en lugar de saber exactamente cómo responder, Ruby usa un conjunto predefinido de reglas para encontrar un método que pueda responderle. Podemos usar estas reglas para interrumpir y responder el mensaje, enviarlo a otro objeto o modificarlo entre otras acciones.

Cada vez que un objeto recibe un mensaje Ruby comprueba:

1. Si este objeto tiene una clase singleton y puede responder a este mensaje.
2. Busca la clase de este objeto y luego la cadena de ancestros de la clase.
3. Uno por uno comprueba si un método está disponible en este antepasado y sube la cadena.

Mensaje que pasa a través de la cadena de herencia

```
class Example
  def example_method
    :example
  end

  def subexample_method
    :example
  end

  def not_missed_method
    :example
  end

  def method_missing name
    return :example if name == :missing_example_method
    return :example if name == :missing_subexample_method
    return :subexample if name == :not_missed_method
    super
  end
end

class SubExample < Example
  def subexample_method
    :subexample
  end

  def method_missing name
    return :subexample if name == :missing_subexample_method
    return :subexample if name == :not_missed_method
    super
```

```
  end
end

s = Subexample.new
```

Para encontrar un método adecuado para el `SubExample#subexample_method` Ruby primero mira la cadena de antepasados del `SubExample`

```
SubExample.ancestors # => [SubExample, Example, Object, Kernel, BasicObject]
```

Se inicia desde el `SubExample` . Si enviamos el mensaje `subexample_method` , Ruby elige el que está disponible, un SubExample e ignora el `Example#subexample_method` .

```
s.subexample_method # => :subexample
```

Después del `SubExample` verifica `Example` . Si enviamos `example_method` Ruby verifica si `SubExample` puede responderle o no, y como no puede, Ruby sube la cadena y mira en `Example` .

```
s.example_method # => :example
```

Después de que Ruby comprueba todos los métodos definidos, ejecuta `method_missing` para ver si puede responder o no. Si enviamos `missing_subexample_method` Ruby no podrá encontrar un método definido en el `SubExample` por lo que se mueve hacia el `Example` . Tampoco puede encontrar un método definido en `Example` o cualquier otra clase superior en cadena. Ruby comienza de nuevo y ejecuta `method_missing` . `method_missing` of `SubExample` puede responder a `missing_subexample_method` .

```
s.missing_subexample_method # => :subexample
```

Sin embargo, si se define un método, Ruby usa una versión definida aunque sea más alta en la cadena. Por ejemplo, si enviamos `not_missed_method` a pesar de que `method_missing` de `SubExample` puede responder, Ruby aparece en `SubExample` porque no tiene un método definido con ese nombre y busca en `Example` que tiene uno.

```
s.not_missed_method # => :example
```

Mensaje que pasa a través de la composición del módulo

Ruby se mueve hacia arriba en la cadena de antepasados de un objeto. Esta cadena puede contener tanto módulos como clases. Las mismas reglas sobre el ascenso de la cadena también se aplican a los módulos.

```
class Example
end

module Prepended
  def initialize *args
    return super :default if args.empty?
```

```
    super
  end
end

module FirstIncluded
  def foo
    :first
  end
end

module SecondIncluded
  def foo
    :second
  end
end

class SubExample < Example
  prepend Prepended
  include FirstIncluded
  include SecondIncluded

  def initialize data = :subexample
    puts data
  end
end

SubExample.ancestors # => [Prepended, SubExample, SecondIncluded, FirstIncluded, Example,
Object, Kernel, BasicObject]

s = SubExample.new # => :default
s.foo # => :second
```

Mensajes de interrupcion

Hay dos formas de interrumpir los mensajes.

- Use `method_missing` para interrumpir cualquier mensaje no definido.
- Defina un método en medio de una cadena para interceptar el mensaje

Después de interrumpir los mensajes, es posible:

- Responder a ellos.
- Envíalos a otro lugar.
- Modificar el mensaje o su resultado.

Interrupción a través de `method_missing` y respuesta al mensaje:

```
class Example
  def foo
    @foo
  end

  def method_missing name, data
    return super unless name.to_s =~ /=$/
    name = name.to_s.sub(/=$/, "")
    instance_variable_set "@#{name}", data
```

```
  end
end

e = Example.new

e.foo = :foo
e.foo # => :foo
```

Interceptando mensaje y modificándolo:

```
class Example
  def initialize title, body
  end
end

class SubExample < Example
end
```

Ahora imaginemos que nuestros datos son "título: cuerpo" y tenemos que dividirlos antes de llamar a `Example` . Podemos definir `initialize` en el `SubExample` .

```
class SubExample < Example
  def initialize raw_data
    processed_data = raw_data.split ":"

    super processed_data[0], processed_data[1]
  end
end
```

Interceptando mensaje y enviándolo a otro objeto:

```
class ObscureLogicProcessor
  def process data
    :ok
  end
end

class NormalLogicProcessor
  def process data
    :not_ok
  end
end

class WrapperProcessor < NormalLogicProcessor
  def process data
    return ObscureLogicProcessor.new.process data if data.obscure?

    super
  end
end
```

Lea Paso de mensajes en línea: https://riptutorial.com/es/ruby/topic/5083/paso-de-mensajes

Capítulo 61: Patrones de diseño y modismos en Ruby

Examples

Semifallo

Ruby Standard Library tiene un módulo Singleton que implementa el patrón Singleton. El primer paso para crear una clase Singleton es requerir e incluir el módulo **`Singleton`** en una clase:

```
require 'singleton'

class Logger
  include Singleton
end
```

Si intenta crear una instancia de esta clase como lo haría normalmente en una clase normal, se `NoMethodError` una excepción `NoMethodError` . El constructor se hace privado para evitar que otras instancias se creen accidentalmente:

```
Logger.new

#=> NoMethodError: private method `new' called for AppConfig:Class
```

Para acceder a la instancia de esta clase, necesitamos usar la `instance()` :

```
first, second = Logger.instance, Logger.instance
first == second

#=> true
```

Ejemplo de registrador

```
require 'singleton'


class Logger
  include Singleton

  def initialize
    @log = File.open("log.txt", "a")
  end

  def log(msg)
    @log.puts(msg)
  end
end
```

Para usar el objeto `Logger` :

```
Logger.instance.log('message 2')
```

Sin Singleton incluye

Las implementaciones singleton anteriores también se pueden realizar sin la inclusión del módulo Singleton. Esto se puede lograr con lo siguiente:

```
class Logger
  def self.instance
    @instance ||= new
  end
end
```

que es una notación abreviada para lo siguiente:

```
class Logger
  def self.instance
    @instance = @instance || Logger.new
  end
end
```

Sin embargo, tenga en cuenta que el módulo Singleton está probado y optimizado, por lo que es la mejor opción para implementar su Singleton.

Observador

El patrón de observador es un patrón de diseño de software en el que un objeto (`subject` llamado) mantiene una lista de sus dependientes (llamados `observers`) y les notifica automáticamente cualquier cambio de estado, generalmente llamando a uno de sus métodos.

Ruby proporciona un mecanismo simple para implementar el patrón de diseño Observer. El módulo `Observable` proporciona la lógica para notificar al suscriptor de cualquier cambio en el objeto Observable.

Para que esto funcione, el observable debe afirmar que ha cambiado y notificar a los observadores.

Los objetos que observan deben implementar un método `update()` , que será la devolución de llamada para el observador.

Implementemos un pequeño chat, donde los usuarios pueden suscribirse a los usuarios y cuando uno de ellos escribe algo, los suscriptores reciben una notificación.

```
require "observer"

class Moderator
  include Observable

  def initialize(name)
    @name = name
  end
```

```
  def write
    message = "Computer says: No"
    changed
    notify_observers(message)
  end
end

class Warner
  def initialize(moderator, limit)
    @limit = limit
    moderator.add_observer(self)
  end
end

class Subscriber < Warner
  def update(message)
    puts "#{message}"
  end
end

moderator = Moderator.new("Rupert")
Subscriber.new(moderator, 1)
moderator.write
moderator.write
```

Produciendo la siguiente salida:

```
# Computer says: No
# Computer says: No
```

Hemos activado el método de `write` en la clase Moderador dos veces, notificando a sus suscriptores, en este caso solo uno.

Cuantos más suscriptores agreguemos, más se propagarán los cambios.

Patrón decorador

El patrón de decorador agrega comportamiento a los objetos sin afectar a otros objetos de la misma clase. El patrón decorador es una alternativa útil para crear subclases.

Crea un módulo para cada decorador. Este enfoque es más flexible que la herencia porque puede combinar responsabilidades en más combinaciones. Además, debido a que la transparencia permite que los decoradores se aniden de forma recursiva, permite un número ilimitado de responsabilidades.

Supongamos que la clase de pizza tiene un método de costo que devuelve 300:

```
class Pizza
  def cost
    300
  end
end
```

Represente la pizza con una capa adicional de ráfaga de queso y el costo aumenta en 50. El enfoque más simple es crear una subclase de `PizzaWithCheese` que devuelva 350 en el método de costo.

```
class PizzaWithCheese < Pizza
  def cost
    350
  end
end
```

A continuación, debemos representar una pizza grande que agregue 100 al costo de una pizza normal. Podemos representarlo usando una subclase LargePizza de Pizza.

```
class LargePizza < Pizza
  def cost
    400
  end
end
```

También podríamos tener un ExtraLargePizza que agrega un costo adicional de 15 a nuestro LargePizza. Si tuviéramos que considerar que estos tipos de pizza podrían servirse con queso, tendríamos que agregar LargePizzaWithChese y ExtraLargePizzaWithCheese subclasses. Terminamos con un total de 6 clases.

Para simplificar el enfoque, use módulos para agregar dinámicamente el comportamiento a la clase Pizza:

Módulo + extender + super decorador: ->

```
class Pizza
  def cost
    300
  end
end

module CheesePizza
  def cost
    super + 50
  end
end

module LargePizza
  def cost
    super + 100
  end
end

pizza = Pizza.new         #=> cost = 300
pizza.extend(CheesePizza) #=> cost = 350
pizza.extend(LargePizza)  #=> cost = 450
pizza.cost                #=> cost = 450
```

Apoderado

El objeto proxy se usa a menudo para garantizar el acceso protegido a otro objeto, cuya lógica empresarial interna no queremos contaminar con los requisitos de seguridad.

Supongamos que queremos garantizar que solo los usuarios con permisos específicos puedan acceder a los recursos.

Definición de proxy: (garantiza que solo los usuarios que realmente puedan ver las reservas puedan reservar el servicio al consumidor)

```
class Proxy
  def initialize(current_user, reservation_service)
    @current_user = current_user
    @reservation_service = reservation_service
  end

  def highest_total_price_reservations(date_from, date_to, reservations_count)
    if @current_user.can_see_reservations?
      @reservation_service.highest_total_price_reservations(
        date_from,
        date_to,
        reservations_count
      )
    else
      []
    end
  end
end
```

Modelos y servicio de reserva:

```
class Reservation
  attr_reader :total_price, :date

  def initialize(date, total_price)
    @date = date
    @total_price = total_price
  end
end

class ReservationService
  def highest_total_price_reservations(date_from, date_to, reservations_count)
    # normally it would be read from database/external service
    reservations = [
      Reservation.new(Date.new(2014, 5, 15), 100),
      Reservation.new(Date.new(2017, 5, 15), 10),
      Reservation.new(Date.new(2017, 1, 15), 50)
    ]

    filtered_reservations = reservations.select do |reservation|
      reservation.date.between?(date_from, date_to)
    end

    filtered_reservations.take(reservations_count)
  end
end

class User
  attr_reader :name
```

```
  def initialize(can_see_reservations, name)
    @can_see_reservations = can_see_reservations
    @name = name
  end

  def can_see_reservations?
    @can_see_reservations
  end
end
```

Servicio al consumidor:

```
class StatsService
  def initialize(reservation_service)
    @reservation_service = reservation_service
  end

  def year_top_100_reservations_average_total_price(year)
    reservations = @reservation_service.highest_total_price_reservations(
      Date.new(year, 1, 1),
      Date.new(year, 12, 31),
      100
    )

    if reservations.length > 0
      sum = reservations.reduce(0) do |memo, reservation|
        memo + reservation.total_price
      end

      sum / reservations.length
    else
      0
    end
  end
end
```

Prueba:

```
def test(user, year)
  reservations_service = Proxy.new(user, ReservationService.new)
  stats_service = StatsService.new(reservations_service)
  average_price = stats_service.year_top_100_reservations_average_total_price(year)
  puts "#{user.name} will see: #{average_price}"
end

test(User.new(true, "John the Admin"), 2017)
test(User.new(false, "Guest"),         2017)
```

BENEFICIOS

- **estamos evitando cualquier cambio en `ReservationService` cuando se cambian las restricciones de acceso.**
- **no estamos mezclando datos relacionados con la empresa (`date_from` , `date_to` , `reservations_count`) con conceptos de dominio no relacionados (permisos de usuario) en servicio.**

- **El consumidor (`StatsService`) también está libre de la lógica relacionada con los permisos**

CUEVAS

- La interfaz de proxy es siempre exactamente la misma que el objeto que oculta, por lo que el usuario que consume el servicio envuelto por el proxy ni siquiera estaba al tanto de la presencia del proxy.

Lea Patrones de diseño y modismos en Ruby en línea:
https://riptutorial.com/es/ruby/topic/2081/patrones-de-diseno-y-modismos-en-ruby

Capítulo 62: Pruebas de la API JSON de RSpec pura

Examples

Probando el objeto Serializador y presentándolo al Controlador

Digamos que quieres construir tu API para cumplir con la especificación jsonapi.org y el resultado debería ser:

```
{
  "article": {
    "id": "305",
    "type": "articles",
    "attributes": {
      "title": "Asking Alexandria"
    }
  }
}
```

La prueba para el objeto Serializador puede verse así:

```
# spec/serializers/article_serializer_spec.rb

require 'rails_helper'

RSpec.describe ArticleSerializer do
  subject { described_class.new(article) }
  let(:article) { instance_double(Article, id: 678, title: "Bring Me The Horizon") }

  describe "#as_json" do
    let(:result) { subject.as_json }

    it 'root should be article Hash' do
      expect(result).to match({
        article: be_kind_of(Hash)
      })
    end

    context 'article hash' do
      let(:article_hash) { result.fetch(:article) }

      it 'should contain type and id' do
        expect(article_hash).to match({
          id: article.id.to_s,
          type: 'articles',
          attributes: be_kind_of(Hash)
        })
      end

      context 'attributes' do
        let(:article_hash_attributes) { article_hash.fetch(:attributes) }
```

```
        it do
          expect(article_hash_attributes).to match({
            title: /[Hh]orizon/,
          })
        end
      end
    end
  end
end
```

El objeto serializador puede verse así:

```
# app/serializers/article_serializer.rb

class ArticleSerializer
  attr_reader :article

  def initialize(article)
    @article = article
  end

  def as_json
    {
      article: {
        id: article.id.to_s,
        type: 'articles',
        attributes: {
          title: article.title
        }
      }
    }
  end
end
```

Cuando ejecutamos nuestras especificaciones de "serializadores" todo pasa.

Eso es bastante aburrido. Introduzcamos un error tipográfico en nuestro serializador de artículos: en lugar de `type: "articles"` volvamos a `type: "events"` y volver a ejecutar nuestras pruebas.

```
rspec spec/serializers/article_serializer_spec.rb

.F.

Failures:

  1) ArticleSerializer#as_json article hash should contain type and id
     Failure/Error:
       expect(article_hash).to match({
         id: article.id.to_s,
         type: 'articles',
         attributes: be_kind_of(Hash)
       })

       expected {:id=>"678", :type=>"event",
:attributes=>{:title=>"Bring Me The Horizon"}} to match {:id=>"678",
:type=>"articles", :attributes=>(be a kind of Hash)}
       Diff:
```

```
       @@ -1,4 +1,4 @@
       -:attributes => (be a kind of Hash),
       +:attributes => {:title=>"Bring Me The Horizon"},
        :id => "678",
       -:type => "articles",
       +:type => "events",

     # ./spec/serializers/article_serializer_spec.rb:20:in `block (4
levels) in <top (required)>'
```

Una vez que haya ejecutado la prueba, es bastante fácil detectar el error.

Una vez que solucione el error (corrija el tipo para que sea un `article`), puede presentarlo al Controlador de esta manera:

```
# app/controllers/v2/articles_controller.rb
module V2
  class ArticlesController < ApplicationController
    def show
      render json: serializer.as_json
    end

    private
      def article
        @article ||= Article.find(params[:id])
      end

      def serializer
        @serializer ||= ArticleSerializer.new(article)
      end
  end
end
```

Este ejemplo se basa en el artículo: http://www.eq8.eu/blogs/30-pure-rspec-json-api-testing

Lea Pruebas de la API JSON de RSpec pura en línea:
https://riptutorial.com/es/ruby/topic/7842/pruebas-de-la-api-json-de-rspec-pura

Capítulo 63: rbenv

Examples

1. Instalar y administrar versiones de Ruby con rbenv

La forma más sencilla de instalar y administrar varias versiones de Ruby con rbenv es usar el complemento ruby-build.

Primero clona el repositorio de rbenv en tu directorio de inicio:

```
$ git clone https://github.com/rbenv/rbenv.git ~/.rbenv
```

Luego clona el plugin ruby-build:

```
$ git clone https://github.com/rbenv/ruby-build.git ~/.rbenv/plugins/ruby-build
```

Asegúrese de que rbenv esté inicializado en su sesión de shell, agregando esto a su `.bash_profile` o `.zshrc` :

```
type rbenv > /dev/null
if [ "$?" = "0" ]; then
    eval "$(rbenv init -)"
fi
```

(Esto es esencialmente lo primero que se verifica si `rbenv` está disponible, y lo inicializa).

Probablemente tendrá que reiniciar su sesión de shell o simplemente abrir una nueva ventana de Terminal.

Nota: si está ejecutando OSX, también deberá instalar las herramientas de línea de comandos de Mac OS con:

```
$ xcode-select --install
```

También puede instalar `rbenv` utilizando Homebrew en lugar de `rbenv` desde la fuente:

```
$ brew update
$ brew install rbenv
```

Luego siga las instrucciones dadas por:

```
$ rbenv init
```

Instala una nueva versión de Ruby:

Listar las versiones disponibles con:

```
$ rbenv install --list
```

Elige una versión e instálala con:

```
$ rbenv install 2.2.0
```

Marque la versión instalada como la versión global, es decir, la que su sistema usa de forma predeterminada:

```
$ rbenv global 2.2.0
```

Comprueba cuál es tu versión global con:

```
$ rbenv global
=> 2.2.0
```

Puede especificar una versión de proyecto local con:

```
$ rbenv local 2.1.2
=> (Creates a .ruby-version file at the current directory with the specified version)
```

Notas al pie:

[1]: Entendiendo el PATH

Desinstalando un Ruby

Hay dos formas de desinstalar una versión particular de Ruby. Lo más fácil es simplemente eliminar el directorio de `~/.rbenv/versions` :

```
$ rm -rf ~/.rbenv/versions/2.1.0
```

Alternativamente, puede usar el comando de desinstalación, que hace exactamente lo mismo:

```
$ rbenv uninstall 2.1.0
```

Si esta versión está en uso en algún lugar, deberá actualizar su versión global o local. Para volver a la versión que está primero en su ruta (usualmente la predeterminada que proporciona su sistema) use:

```
$ rbenv global system
```

Lea rbenv en línea: https://riptutorial.com/es/ruby/topic/4096/rbenv

Capítulo 64: Receptores implícitos y comprensión del yo

Examples

Siempre hay un receptor implícito.

En Ruby, siempre hay un receptor implícito para todas las llamadas de método. El lenguaje mantiene una referencia al receptor implícito actual almacenado en la variable `self` . Ciertas palabras clave de lenguaje como `class` y `module` cambiarán a lo que apunta `self` . Comprender estos comportamientos es muy útil para dominar el idioma.

Por ejemplo, cuando abres por primera `irb`

```
irb(main):001:0> self
=> main
```

En este caso, el objeto `main` es el receptor implícito (consulte http://stackoverflow.com/a/917842/417872 para obtener más información acerca de `main`).

Puede definir métodos en el receptor implícito usando la palabra clave `def` . Por ejemplo:

```
irb(main):001:0> def foo(arg)
irb(main):002:1> arg.to_s
irb(main):003:1> end
=> :foo
irb(main):004:0> foo 1
=> "1"
```

Esto ha definido el método foo en la instancia del objeto principal que se ejecuta en su respuesta.

Tenga en cuenta que las variables locales se buscan antes que los nombres de los métodos, de modo que si define una variable local con el mismo nombre, su referencia reemplazará a la referencia del método. Continuando del ejemplo anterior:

```
irb(main):005:0> defined? foo
=> "method"
irb(main):006:0> foo = 1
=> 1
irb(main):007:0> defined? foo
=> "local-variable"
irb(main):008:0> foo
=> 1
irb(main):009:0> method :foo
=> #<Method: Object#foo>
```

El `method` método todavía puede encontrar el método `foo` porque no comprueba las variables locales, mientras que la referencia normal `foo` sí lo hace.

Las palabras clave cambian el receptor implícito.

Cuando define una clase o módulo, el receptor implícito se convierte en una referencia a la clase en sí. Por ejemplo:

```
puts "I am #{self}"
class Example
  puts "I am #{self}"
end
```

Ejecutando el código anterior se imprimirá:

```
"I am main"
"I am Example"
```

¿Cuándo usar uno mismo?

La mayoría del código de Ruby utiliza el receptor implícito, por lo que los programadores que son nuevos en Ruby a menudo se confunden acerca de cuándo usar `self` . La respuesta práctica es que el `self` se usa de dos maneras principales:

1. Para cambiar el receptor.

Normalmente, el comportamiento de `def` dentro de una clase o módulo es crear métodos de instancia. Self se puede usar para definir métodos en la clase.

```
class Foo
  def bar
    1
  end

  def self.bar
    2
  end
end

Foo.new.bar #=> 1
Foo.bar #=> 2
```

2. Desambiguar el receptor.

Cuando las variables locales pueden tener el mismo nombre que un método, es posible que se requiera un receptor explícito para desambiguar.

Ejemplos:

```
class Example
  def foo
    1
  end

  def bar
```

```
    foo + 1
  end

  def baz(foo)
    self.foo + foo # self.foo is the method, foo is the local variable
  end

  def qux
    bar = 2
    self.bar + bar # self.bar is the method, bar is the local variable
  end
end

Example.new.foo    #=> 1
Example.new.bar    #=> 2
Example.new.baz(2) #=> 3
Example.new.qux    #=> 4
```

El otro caso común que requiere desambiguación involucra métodos que terminan en el signo igual. Por ejemplo:

```
class Example
  def foo=(input)
    @foo = input
  end

  def get_foo
    @foo
  end

  def bar(input)
    foo = input # will create a local variable
  end

  def baz(input)
    self.foo = input # will call the method
  end
end

e = Example.new
e.get_foo #=> nil
e.foo = 1
e.get_foo #=> 1
e.bar(2)
e.get_foo #=> 1
e.baz(2)
e.get_foo #=> 2
```

Lea Receptores implícitos y comprensión del yo en línea:
https://riptutorial.com/es/ruby/topic/5856/receptores-implicitos-y-comprension-del-yo

Capítulo 65: Recursion en Ruby

Examples

Función recursiva

Comencemos con un algoritmo simple para ver cómo se podría implementar la recursión en Ruby.

Una panadería tiene productos para vender. Los productos están en paquetes. Atiende pedidos en paquetes únicamente. El empaque comienza desde el tamaño de paquete más grande y luego las cantidades restantes se llenan con los siguientes tamaños de paquete disponibles.

Por ejemplo, si se recibe un pedido de 16, la panadería asigna 2 de 5 paquetes y 2 de 3 paquetes. 2 *5* + *2* 3 = 16. Veamos cómo se implementa esto en la recursión. "asignar" es la función recursiva aquí.

```
#!/usr/bin/ruby

class Bakery
  attr_accessor :selected_packs

  def initialize
    @packs = [5,3] # pack sizes 5 and 3
    @selected_packs = []
  end

  def allocate(qty)
    remaining_qty = nil

    # ============================================
    # packs are allocated in large packs first order
    # to minimize the packaging space
    # ============================================
    @packs.each do |pack|
      remaining_qty = qty - pack

      if remaining_qty > 0
        ret_val = allocate(remaining_qty)
        if ret_val == 0
          @selected_packs << pack
          remaining_qty = 0
          break
        end
      elsif remaining_qty == 0
        @selected_packs << pack
        break
      end
    end

    remaining_qty
  end
end
```

```
bakery = Bakery.new
bakery.allocate(16)
puts "Pack combination is: #{bakery.selected_packs.inspect}"
```

La salida es:

> La combinación de paquetes es: [3, 3, 5, 5]

Recursion de cola

Muchos algoritmos recursivos pueden expresarse usando iteración. Por ejemplo, la función de denominador común más grande se puede escribir recursivamente :

```
def gdc (x, y)
  return x if y == 0
  return gdc(y, x%y)
end
```

o iterativamente:

```
def gdc_iter (x, y)
  while y != 0 do
    x, y = y, x%y
  end

  return x
end
```

Los dos algoritmos son equivalentes en teoría, pero la versión recursiva corre el riesgo de un SystemStackError . Sin embargo, dado que el método recursivo termina con una llamada a sí mismo, podría optimizarse para evitar un desbordamiento de pila. Otra forma de decirlo: el algoritmo recursivo puede resultar en el mismo código de máquina que el iterativo *si* el compilador sabe que debe buscar la llamada al método recursivo al final del método. Ruby no realiza la optimización de llamadas de cola de forma predeterminada, pero puede activarlo con :

```
RubyVM::InstructionSequence.compile_option = {
  tailcall_optimization: true,
  trace_instruction: false
}
```

Además de activar la optimización de llamada de cola, también debe desactivar el seguimiento de instrucciones. Desafortunadamente, estas opciones solo se aplican en el momento de la compilación, por lo que debe `require` el método recursivo de otro archivo o `eval` la definición del método:

```
RubyVM::InstructionSequence.new(<<-EOF).eval
  def me_myself_and_i
    me_myself_and_i
  end
EOF
me_myself_and_i # Infinite loop, not stack overflow
```

Finalmente, la última llamada devuelta debe devolver el método y *solo el método* . Eso significa que tendrás que volver a escribir la función factorial estándar:

```
def fact(x)
  return 1 if x <= 1
  return x*fact(x-1)
end
```

A algo como:

```
 def fact(x, acc=1)
   return acc if x <= 1
   return fact(x-1, x*acc)
 end
```

Esta versión pasa la suma acumulada a través de un segundo argumento (opcional) que por defecto es 1.

Lectura adicional: Optimización de llamadas de cola en Ruby y Tailin 'Ruby .

Lea Recursion en Ruby en línea: https://riptutorial.com/es/ruby/topic/7986/recursion-en-ruby

Capítulo 66: Refinamientos

Observaciones

Los refinamientos tienen un alcance léxico, lo que significa que están vigentes desde el momento en que se activan (con la palabra clave que `using`) hasta que el control cambia. Por lo general, el control se cambia al final de un módulo, clase o archivo.

Examples

Parches de mono con alcance limitado

El principal problema de Monkey Patching es que contamina el alcance global. Su código de trabajo está a la merced de todos los módulos que utiliza, no pisando los otros dedos de los pies. La solución de Ruby para esto son los refinamientos, que son básicamente parches de mono en un alcance limitado.

```
module Patches
  refine Fixnum do
    def plus_one
      self + 1
    end

    def plus(num)
      self + num
    end

    def concat_one
      self.to_s + '1'
    end
  end
end

class RefinementTest
  # has access to our patches
  using Patches

  def initialize
    puts 1.plus_one
    puts 3.concat_one
  end
end

# Main scope doesn't have changes

1.plus_one
# => undefined method `plus_one' for 1:Fixnum (NoMethodError)

RefinementTest.new
# => 2
# => '31'
```

Módulos de doble propósito (refinamientos o parches globales)

Es una buena práctica utilizar parches con Refinamientos, pero a veces es bueno cargarlos globalmente (por ejemplo, en desarrollo o pruebas).

Digamos, por ejemplo, que desea iniciar una consola, requerir su biblioteca y luego tener los métodos parcheados disponibles en el ámbito global. No podría hacer esto con los refinamientos porque el `using` debe llamarse en una definición de clase / módulo. Pero es posible escribir el código de tal manera que tenga un doble propósito:

```
module Patch
  def patched?; true; end
  refine String do
    include Patch
  end
end

# globally
String.include Patch
"".patched? # => true

# refinement
class LoadPatch
  using Patch
  "".patched? # => true
end
```

Refinamientos dinámicos

Los refinamientos tienen limitaciones especiales.

`refine` solo se puede utilizar en el alcance de un módulo, pero se puede programar usando `send :refine` .

`using` es más limitado. Solo se puede llamar en una definición de clase / módulo. Aún así, puede aceptar una variable que apunta a un módulo y puede invocarse en un bucle.

Un ejemplo que muestra estos conceptos:

```
module Patch
  def patched?; true; end
end

Patch.send(:refine, String) { include Patch }

patch_classes = [Patch]

class Patched
  patch_classes.each { |klass| using klass }
  "".patched? # => true
end
```

Como el `using` es muy estático, se puede emitir con orden de carga si los archivos de refinamiento

no se cargan primero. Una forma de abordar esto es envolver la definición de clase / módulo parcheada en un proceso. Por ejemplo:

```
module Patch
  refine String do
    def patched; true; end
  end
end

class Foo
end

# This is a proc since methods can't contain class definitions
create_patched_class = Proc.new do
  Foo.class_exec do
    class Bar
      using Patch
      def self.patched?; ''.patched == true; end
    end
  end
end
create_patched_class.call
Foo::Bar.patched? # => true
```

Al llamar al proceso se crea la clase parcheada `Foo::Bar` . Esto puede retrasarse hasta que todo el código se haya cargado.

Lea Refinamientos en línea: https://riptutorial.com/es/ruby/topic/6563/refinamientos

Capítulo 67: Ruby Version Manager

Examples

Cómo crear gemset

Para crear un gemset necesitamos crear un archivo `.rvmrc` .

Sintaxis:

```
$ rvm --rvmrc --create <ruby-version>@<gemsetname>
```

Ejemplo:

```
$ rvm --rvmrc --create ruby-2.2.2@myblog
```

La línea anterior creará un archivo `.rvmrc` en el directorio raíz de la aplicación.

Para obtener la lista de gemsets disponibles, use el siguiente comando:

```
$ rvm list gemsets
```

Instalando Ruby con RVM

Ruby Version Manager es una herramienta de línea de comandos para instalar y administrar diferentes versiones de Ruby.

- `rvm istall 2.3.1` por ejemplo, instala Ruby versión 2.3.1 en su máquina.
- Con `rvm list` puede ver qué versiones están instaladas y cuáles están configuradas para su uso.

  ```
  user@dev:~$ rvm list

  rvm rubies

  =* ruby-2.3.1 [ x86_64 ]

  # => - current
  # =* - current && default
  #  * - default
  ```

- Con `rvm use 2.3.0` puede cambiar entre las versiones instaladas.

Lea Ruby Version Manager en línea: https://riptutorial.com/es/ruby/topic/4040/ruby-version-manager

Capítulo 68: Simbolos

Sintaxis

- :símbolo
- :'símbolo'
- :"símbolo"
- "símbolo" .to_sym
- % s {símbolo}

Observaciones

Ventajas de usar símbolos sobre cadenas:

1. Un símbolo de Ruby es un objeto con comparación O (1)

Para comparar dos cadenas, potencialmente debemos mirar a cada personaje. Para dos cadenas de longitud N, esto requerirá N + 1 comparaciones

```
def string_compare str1, str2
  if str1.length != str2.length
        return false
  end
  for i in 0...str1.length
    return false if str1[i] != str2[i]
  end
  return true
end
string_compare "foobar", "foobar"
```

Pero dado que cada aparición de: foobar se refiere al mismo objeto, podemos comparar los símbolos mirando los ID de los objetos. Podemos hacer esto con una sola comparación. (O (1))

```
def symbol_compare sym1, sym2
  sym1.object_id == sym2.object_id
end
symbol_compare :foobar, :foobar
```

2. Un símbolo de Ruby es una etiqueta en una enumeración de forma libre

En C ++, podemos usar "enumeraciones" para representar familias de constantes relacionadas:

```
enum BugStatus { OPEN, CLOSED };
BugStatus original_status = OPEN;
BugStatus current_status  = CLOSED;
```

Pero debido a que Ruby es un lenguaje dinámico, no nos preocupamos por declarar un tipo de estado de error o por mantener un registro de los valores legales. En cambio, representamos los

valores de enumeración como símbolos:

```
original_status = :open
current_status  = :closed
```

3. Un símbolo de rubí es un nombre constante y único.

En Ruby, podemos cambiar el contenido de una cadena:

```
"foobar"[0] = ?b # "boo"
```

Pero no podemos cambiar los contenidos de un símbolo:

```
:foobar[0]  = ?b # Raises an error
```

4. Un símbolo de Ruby es la palabra clave para un argumento de palabra clave

Al pasar argumentos de palabras clave a una función Ruby, especificamos las palabras clave usando símbolos:

```
# Build a URL for 'bug' using Rails.
url_for :controller => 'bug',
        :action => 'show',
        :id => bug.id
```

5. Un símbolo de rubí es una excelente opción para una clave hash

Normalmente, usaremos símbolos para representar las claves de una tabla hash:

```
options = {}
options[:auto_save]     = true
options[:show_comments] = false
```

Examples

Creando un símbolo

La forma más común de crear un objeto de `Symbol` es prefijando el identificador de cadena con dos puntos:

```
:a_symbol       # => :a_symbol
:a_symbol.class # => Symbol
```

Aquí hay algunas formas alternativas de definir un `Symbol` , en combinación con un literal de `String` :

```
:"a_symbol"
"a_symbol".to_sym
```

Los símbolos también tienen una secuencia `%s` que admite delimitadores arbitrarios similares a cómo funcionan `%q` y `%Q` para cadenas:

```
%s(a_symbol)
%s{a_symbol}
```

El `%s` es particularmente útil para crear un símbolo a partir de una entrada que contiene espacios en blanco:

```
%s{a symbol} # => :"a symbol"
```

Mientras que algunos símbolos interesantes (`:/` `:[]` `:^` , etc.) se pueden crear con ciertos identificadores de cadena, tenga en cuenta que los símbolos no se pueden crear usando un identificador numérico:

```
:1 # => syntax error, unexpected tINTEGER, ...
:0.3 # => syntax error, unexpected tFLOAT, ...
```

Los símbolos pueden terminar con una sola `?` o `!` sin necesidad de usar una cadena literal como el identificador del símbolo:

```
:hello?  # :"hello?" is not necessary.
:world!  # :"world!" is not necessary.
```

Tenga en cuenta que todos estos métodos diferentes para crear símbolos devolverán el mismo objeto:

```
:symbol.object_id == "symbol".to_sym.object_id
:symbol.object_id == %s{symbol}.object_id
```

Desde Ruby 2.0 hay un atajo para crear una matriz de símbolos a partir de palabras:

```
%i(numerator denominator) == [:numerator, :denominator]
```

Convertir una cadena a un símbolo

Dada una `String` :

```
s = "something"
```

Hay varias formas de convertirlo en un `Symbol` :

```
s.to_sym
# => :something
:"#{s}"
# => :something
```

Convertir un símbolo en cadena

Dado un `Symbol` :

```
s = :something
```

La forma más sencilla de convertirlo en una `String` es mediante el uso del método `Symbol#to_s` :

```
s.to_s
# => "something"
```

Otra forma de hacerlo es utilizando el método `Symbol#id2name` , que es un alias para el método `Symbol#to_s` . Pero es un método que es exclusivo de la clase de `Symbol` :

```
s.id2name
# => "something"
```

Lea Simbolos en línea: https://riptutorial.com/es/ruby/topic/873/simbolos

Capítulo 69: Sistema operativo o comandos de shell

Introducción

Hay muchas formas de interactuar con el sistema operativo. Desde dentro de Ruby puede ejecutar comandos / sub-procesos de shell / sistema.

Observaciones

Exec:
Exec tiene una funcionalidad muy limitada y, cuando se ejecute, saldrá del programa Ruby y ejecutará el comando.

El Comando del Sistema:
El comando del sistema se ejecuta en un sub-shell en lugar de reemplazar el proceso actual y devuelve true o nill. El comando del sistema es, como backticks, una operación de bloqueo donde la aplicación principal espera hasta que se complete el resultado de la operación del sistema. Aquí, la operación principal nunca debe preocuparse por capturar una excepción generada desde el proceso hijo.

La salida de la función del sistema siempre será verdadera o nula dependiendo de si el script se ha ejecutado o no sin error. Por lo tanto, todos los errores al ejecutar el script no se pasarán a nuestra aplicación. La operación principal nunca debe preocuparse por capturar una excepción provocada por el proceso secundario. En este caso, la salida es nula porque el proceso hijo generó una excepción.
Esta es una operación de bloqueo en la que el programa Ruby esperará hasta que finalice la operación del comando antes de continuar.
La operación del sistema usa la bifurcación para bifurcar el proceso actual y luego ejecuta la operación dada usando exec.

Los backticks (`):
El carácter de comilla invertida generalmente se encuentra debajo de la tecla Escape del teclado. Backticks se ejecuta en un sub-shell en lugar de reemplazar el proceso actual y devuelve el resultado del comando.
Aquí podemos obtener la salida del comando, pero el programa se bloqueará cuando se genere una excepción.
Si hay una excepción en el subproceso, esa excepción se otorga al proceso principal y el proceso principal podría terminar si no se maneja la excepción. Esta es una operación de bloqueo en la que el programa Ruby esperará hasta que finalice la operación del comando antes de continuar.
La operación del sistema usa la bifurcación para bifurcar el proceso actual y luego ejecuta la operación dada usando exec.

IO.popen:

IO.popen se ejecuta en un subproceso. Aquí, la entrada estándar del subproceso y la salida estándar se conectan al objeto IO.

Popen3:
Popen3 le permite acceder a la entrada estándar, la salida estándar y el error estándar.
La entrada y salida estándar del subproceso se devolverán a los objetos de E / S.

PS (igual que $ CHILD_STATUS)
Se puede usar con las operaciones backticks, system () o% x {} y dará el estado del último comando ejecutado del sistema.
Esto podría ser útil para acceder a las `exitstatus` y `pid` .

```
$?.exitstatus
```

Examples

Formas recomendadas para ejecutar código de shell en Ruby:

Open3.popen3 o Open3.capture3:
Open3 en realidad solo usa el comando spawn de Ruby, pero te da una API mucho mejor.

Open3.popen3

Popen3 se ejecuta en un subproceso y devuelve stdin, stdout, stderr y wait_thr.

```
require 'open3'
stdin, stdout, stderr, wait_thr = Open3.popen3("sleep 5s && ls")
puts "#{stdout.read} #{stderr.read} #{wait_thr.value.exitstatus}"
```

o

```
require 'open3'
cmd = 'git push heroku master'
Open3.popen3(cmd) do |stdin, stdout, stderr, wait_thr|
  puts "stdout is:" + stdout.read
  puts "stderr is:" + stderr.read
end
```

dará salida: **stdout es: stderr es: fatal: no es un repositorio git (o cualquiera de los directorios principales): .git**

o

```
require 'open3'
cmd = 'ping www.google.com'
Open3.popen3(cmd) do |stdin, stdout, stderr, wait_thr|
  while line = stdout.gets
    puts line
  end
end
```

saldrá:

Hacer ping en www.google.com [216.58.223.36] con 32 bytes de datos:
Respuesta de 216.58.223.36: bytes = 32 tiempo = 16 ms TTL = 54
Respuesta de 216.58.223.36: bytes = 32 tiempo = 10 ms TTL = 54
Respuesta de 216.58.223.36: bytes = 32 tiempo = 21 ms TTL = 54
Respuesta de 216.58.223.36: bytes = 32 tiempo = 29ms TTL = 54
Estadísticas de ping para 216.58.223.36:
Paquetes: Enviados = 4, Recibidos = 4, Perdidos = 0 (0% de pérdida),
Tiempo aproximado de ida y vuelta en milisegundos:
Mínimo = 10 ms, Máximo = 29 ms, promedio = 19 ms

Open3.capture3:

```
require 'open3'

stdout, stderr, status = Open3.capture3('my_funky_command', 'and', 'some', 'argumants')
if status.success?
  # command completed successfully, do some more stuff
else
  raise "An error occured"
end
```

o

```
Open3.capture3('/some/binary with some args')
```

Sin embargo, no se recomienda, debido a la sobrecarga adicional y al potencial de inyecciones de shell.

Si el comando se lee desde la entrada estándar y desea alimentar algunos datos:

```
Open3.capture3('my_funky_command', stdin_data: 'read from stdin')
```

Ejecute el comando con un directorio de trabajo diferente, usando chdir:

```
Open3.capture3('my_funky_command', chdir: '/some/directory')
```

Formas clásicas de ejecutar código shell en Ruby:

Exec:

```
exec 'echo "hello world"'
```

o

```
exec ('echo "hello world"')
```

El Comando del Sistema:

```
system 'echo "hello world"'
```

Saldrá "hola mundo" en la ventana de comandos.

o

```
system ('echo "hello world"')
```

El comando del sistema puede devolver un verdadero si el comando fue exitoso o nulo cuando no lo fue.

```
result = system 'echo "hello world"'
puts result  # will return a true in the command window
```

Los backticks (`):

`echo "hello world"` Saldrá "hello world" en la ventana de comandos.

También puedes coger el resultado.

```
result = `echo "hello world"`
puts "We always code a " + result
```

IO.popen:

```
# Will get and return the current date from the system
IO.popen("date") { |f| puts f.gets }
```

Lea Sistema operativo o comandos de shell en línea:
https://riptutorial.com/es/ruby/topic/10921/sistema-operativo-o-comandos-de-shell

Capítulo 70: Struct

Sintaxis

- Estructura = Struct.new: atributo

Examples

Creando nuevas estructuras para datos.

`Struct` define nuevas clases con los atributos y métodos de acceso especificados.

```
Person = Struct.new :first_name, :last_name
```

Luego puedes instanciar objetos y usarlos:

```
person = Person.new 'John', 'Doe'
# => #<struct Person first_name="John", last_name="Doe">

person.first_name
# => "John"

person.last_name
# => "Doe"
```

Personalizar una clase de estructura

```
Person = Struct.new :name do
  def greet(someone)
    "Hello #{someone}! I am #{name}!"
  end
end

Person.new('Alice').greet 'Bob'
# => "Hello Bob! I am Alice!"
```

Búsqueda de atributos

A los atributos se puede acceder cadenas y símbolos como claves. Los índices numéricos también funcionan.

```
Person = Struct.new :name
alice = Person.new 'Alice'

alice['name']  # => "Alice"
alice[:name]   # => "Alice"
alice[0]       # => "Alice"
```

Lea Struct en línea: https://riptutorial.com/es/ruby/topic/5016/struct

Capítulo 71: Uso de gemas

Examples

Instalando gemas de rubí

Esta guía asume que ya tienes Ruby instalado. Si está utilizando Ruby < `1.9` , deberá instalar RubyGems manualmente, ya que no se incluirá de forma nativa .

Para instalar una gema rubí, ingrese el comando:

```
gem install [gemname]
```

Si está trabajando en un proyecto con una lista de dependencias de gemas, éstas se enumerarán en un archivo llamado `Gemfile` . Para instalar una nueva gema en el proyecto, agregue la siguiente línea de código en el `Gemfile` :

```
gem 'gemname'
```

La gema de Bundler utiliza este `Gemfile` para instalar las dependencias que requiere su proyecto; sin embargo, esto significa que tendrá que instalar primero Bundler ejecutando (si aún no lo ha hecho):

```
gem install bundler
```

Guarde el archivo y luego ejecute el comando:

```
bundle install
```

Especificando versiones

El número de versión se puede especificar en el comando live, con el indicador `-v` , como por ejemplo:

```
gem install gemname -v 3.14
```

Al especificar los números de versión en un `Gemfile` , tiene varias opciones disponibles:

- No se ha especificado ninguna versión (`gem 'gemname')` : instalará la *última* versión que sea compatible con otras gemas en el `Gemfile` .
- Versión exacta especificada (`gem 'gemname', '3.14'`): solo intentará instalar la versión `3.14` (y fallará si esto es incompatible con otras gemas en el `Gemfile`).
- Número de versión mínimo **optimista** (`gem 'gemname', '>=3.14'`): solo intentará instalar la *última* versión que sea compatible con otras gemas en el `Gemfile` , y falla si ninguna versión

mayor o igual a `3.14` es compatible. El operador `>` también se puede utilizar.
- Número de versión mínimo **pesimista** (`gem 'gemname', '~>3.14'`) - Esto es funcionalmente equivalente a usar `gem 'gemname', '>=3.14', '<4'` . En otras palabras, solo se permite aumentar el número después del *período final* .

Como práctica recomendada : es posible que desee utilizar una de las bibliotecas de administración de versiones de Ruby como rbenv o rvm . A través de estas bibliotecas, puede instalar diferentes versiones de tiempos de ejecución y gemas de Ruby en consecuencia. Por lo tanto, cuando trabaje en un proyecto, esto será especialmente útil porque la mayoría de los proyectos están codificados contra una versión conocida de Ruby.

Instalación de gemas desde github / filesystem

Puedes instalar una gema desde github o sistema de archivos. Si la gema se ha extraído de git o de alguna manera ya se encuentra en el sistema de archivos, puede instalarla usando

```
gem install --local path_to_gem/filename.gem
```

Instalación de gema de github. Descarga las fuentes de github

```
mkdir newgem
cd newgem
git clone https://urltogem.git
```

Construir la gema

```
gem build GEMNAME.gemspec
gem install gemname-version.gem
```

Comprobando si una gema requerida está instalada desde el código

Para verificar si una gema requerida está instalada, desde su código, puede usar lo siguiente (usando nokogiri como ejemplo):

```
begin
  found_gem = Gem::Specification.find_by_name('nokogiri')
  require 'nokogiri'
  ....
  <the rest of your code>
rescue Gem::LoadError
end
```

Sin embargo, esto se puede extender aún más a una función que se puede usar para configurar la funcionalidad dentro de su código.

```
def gem_installed?(gem_name)
  found_gem = false
  begin
```

```
    found_gem = Gem::Specification.find_by_name(gem_name)
  rescue Gem::LoadError
     return false
  else
    return true
  end
end
```

Ahora puede verificar si la gema requerida está instalada e imprimir un mensaje de error.

```
if gem_installed?('nokogiri')
  require 'nokogiri'
else
  printf "nokogiri gem required\n"
  exit 1
end
```

o

```
if gem_installed?('nokogiri')
  require 'nokogiri'
else
  require 'REXML'
end
```

Usando un Gemfile y Bundler

Un `Gemfile` es la forma estándar de organizar dependencias en su aplicación. Un Gemfile básico se verá así:

```
source 'https://rubygems.org'

gem 'rack'
gem 'sinatra'
gem 'uglifier'
```

Puede especificar las versiones de la gema que desee de la siguiente manera:

```
# Match except on point release. Use only 1.5.X
gem 'rack', '~>1.5.2'
# Use a specific version.
gem 'sinatra', '1.4.7'
# Use at least a version or anything greater.
gem 'uglifier', '>= 1.3.0'
```

También puedes sacar gemas directamente de un repositorio de git:

```
# pull a gem from github
gem 'sinatra', git: 'https://github.com/sinatra/sinatra.git'
# you can specify a sha
gem 'sinatra', git: 'https://github.com/sinatra/sinatra.git', sha:
'30d4fb468fd1d6373f82127d845b153f17b54c51'
# you can also specify a branch, though this is often unsafe
```

```
gem 'sinatra', git: 'https://github.com/sinatra/sinatra.git', branch: 'master'
```

También puedes agrupar gemas dependiendo de para qué se usan. Por ejemplo:

```
group :development, :test do
    # This gem is only available in dev and test, not production.
    gem 'byebug'
end
```

Puede especificar en qué plataforma deben ejecutarse ciertas gemas si su aplicación necesita poder ejecutarse en múltiples plataformas. Por ejemplo:

```
platform :jruby do
  gem 'activerecord-jdbc-adapter'
  gem 'jdbc-postgres'
end

platform :ruby do
  gem 'pg'
end
```

Para instalar todas las gemas de un Gemfile haz:

```
gem install bundler
bundle install
```

Bundler / inline (bundler v1.10 y posteriores)

A veces necesitas hacer un guión para alguien pero no estás seguro de lo que tiene en su máquina. ¿Hay todo lo que tu guión necesita? No es para preocuparse. Bundler tiene una gran función llamada en línea.

Proporciona un método `gemfile` y, antes de que se ejecute el script, se descarga y requiere todas las gemas necesarias. Un pequeño ejemplo:

```
require 'bundler/inline' #require only what you need

#Start the bundler and in it use the syntax you are already familiar with
gemfile(true) do
  source 'https://rubygems.org'
        gem 'nokogiri', '~> 1.6.8.1'
        gem 'ruby-graphviz'
end
```

Lea Uso de gemas en línea: https://riptutorial.com/es/ruby/topic/1540/uso-de-gemas

Capítulo 72: Variables de entorno

Sintaxis

- ENV [nombre_variable]
- ENV.fetch (variable_name, default_value)

Observaciones

Permite obtener la ruta del perfil de usuario de forma dinámica para las secuencias de comandos en Windows

Examples

Muestra para obtener la ruta del perfil de usuario

```
# will retrieve my home path
ENV['HOME'] # => "/Users/username"

# will try retrieve the 'FOO' environment variable. If failed, will get 'bar'
ENV.fetch('FOO', 'bar')
```

Lea Variables de entorno en línea: https://riptutorial.com/es/ruby/topic/4276/variables-de-entorno

Creditos

S. No	Capítulos	Contributors
1	Empezando con Ruby Language	alejosocorro, CalmBit, Community, ctietze, Darpan Chhatravala, David Grayson, DawnPaladin, Eli Sadoff, Jonathan_W, Jonathon Jones, Ken Y-N, knut, Lucas Costa, luissimo, Martin Velez, Mhmd, mnoronha, numbermaniac, peter, prcastro, RamenChef, Simone Carletti, smileart, Steve, Timo Schilling, Tom Lord, Tot Zam, Undo, Vishnu Y S, Wayne Conrad
2	Alcance variable y visibilidad	Matheus Moreira, Ninigi, Sandeep Tuniki
3	Aplicaciones de línea de comandos	Eli Sadoff
4	Argumentos de palabras clave	giniouxe, mnoronha, Simone Carletti
5	Arrays	Ajedi32, alebruck, Andrea Mazzarella, Andrey Deineko, Automatico, br3nt, Community, Dalton, daniero, David Grayson, davidhu2000, DawnPaladin, D-side, Eli Sadoff, Francesco Lupo Renzi, iGbanam, joshaidan, Katsuhiko Yoshida, knut, Lucas Costa, Lukas Baliak, lwassink, Masa Sakano, meagar, Mhmd, Mike H-R, MrTheWalrus, ndn, Nick Roz, nus, Pablo Torrecilla, Pooyan Khosravi, Richard Hamilton, Sagar Pandya, Saroj Sasmal, Shadoath, squadette, Steve, Tom Lord, Undo, Vasfed
6	Arreglos Multidimensionales	Francesco Boffa
7	Atrapar excepciones con Begin / Rescue	Sean Redmond, stevendaniels
8	Bloques y Procs y Lambdas	br3nt, coreyward, Eli Sadoff, engineersmnky, Jasper, Kathryn, Lukas Baliak, Marc-Andre, Matheus Moreira, meagar, Mhmd, nus, Pooyan Khosravi, QPaysTaxes, Simone Carletti
9	Cargando archivos de origen	mnoronha, nus
10	Casting (conversión de tipo)	giniouxe, Jon Wood, meagar, Mhmd, Nakilon
11	Clase Singleton	Geoffroy, giniouxe, Matheus Moreira, MegaTom, nus, Pooyan

		Khosravi
12	Cola	Pooyan Khosravi
13	Comentarios	giniouxe, Jeremy, Rahul Singh, Robert Harvey
14	Comparable	giniouxe, ndn, sandstrom, sonna
15	Constantes	Engr. Hasanuzzaman Sumon, mahatmanich, user2367593
16	Constantes especiales en Ruby	giniouxe, mnoronha, Redithion
17	Creación / gestión de gemas	manasouza, thesecretmaster
18	Depuración	DawnPaladin, ogirginc
19	Destruccion	Austin Vern Songer, Zaz
20	Distancia	DawnPaladin, Rahul Singh, Yonatha Almeida
21	Empezando con Hanami	Mauricio Junior
22	Enumerable en ruby	Neha Chopra
23	Enumeradores	errm, Matheus Moreira
24	ERB	amingilani
25	Evaluación dinámica	Matheus Moreira, MegaTom, Phrogz, Pooyan Khosravi, Simone Carletti
26	Excepciones	David Grayson, Eric Bouchut, hillary.fraley, iturgeon, kamaradclimber, Lomefin, Lucas Costa, Lukas Baliak, lwassink, Michael Kuhinica, moertel, Muhammad Abdullah, ndn, Robert Columbia, Simone Carletti, Steve, Vasfed, Wayne Conrad
27	Expresiones regulares y operaciones basadas en expresiones regulares	Addison, Elenian, giniouxe, Jon Ericson, moertel, mudasobwa, Nick Roz, peter, Redithion, Saša Zejnilović, Scudelletti, Shelvacu
28	Extensiones C	Austin Vern Songer, photoionized
29	Fecha y hora	Austin Vern Songer, Redithion
30	Flujo de control	alebruck, angelparras, br3nt, daniero, DarKy, David Grayson,

		dgilperez, Dimitry_N, D-side, Elenian, Francesco Lupo Renzi, giniouxe, JoeyB, jose_castro_arnaud, kannix, Kathryn, Lahiru, mahatmanich, meagar, MegaTom, Michael Gaskill, moertel, mudasobwa, Muhammad Abdullah, ndn, Nick Roz, Pablo Torrecilla, russt, Scudelletti, Simone Carletti, Steve, the Tin Man , theIV, Tom Lord, Vasfed, Ven, vgoff, Yule
31	Generar un número aleatorio	user1821961
32	Hashes	4444, Adam Sanderson, Arman Jon Villalobos, Atul Khanduri, Bo Jeanes, br3nt, C dot StrifeVII, Charlie Egan, Charlie Harding, Christoph Petschnig, Christopher Oezbek, Community, danielrsmith, David Grayson, dgilperez, divyum, Felix, G. Allen Morris III, gorn, iltempo, Ivan, Jeweller, jose_castro_arnaud, kabuko, Kathryn, kleaver, Konstantin Gredeskoul, Koraktor, Kris , Lucas Costa, Lukas Baliak, Marc-Andre, Martin Samami, Martin Velez, Matt, MattD, meagar, MegaTom, Mhmd, Michael Kuhinica, moertel, mrlee, MZaragoza, ndn, neontapir, New Alexandria, Nic Nilov, Nick Roz, nus, Old Pro, Owen, peter50216, pjam, PJSCopeland, Pooyan Khosravi, RamenChef , Richard Hamilton, Sid, Simone Carletti, spejamchr, spickermann, Steve, stevendaniels, the Tin Man, Tom Lord, Ven , wirefox, Zaz
33	Herencia	br3nt, Gaelan, Kirti Thorat, Lynn, MegaTom, mlabarca, nus, Pascal Fabig, Pragash, RamenChef, Simone Carletti, thesecretmaster, Vasfed
34	Hilo	Austin Vern Songer, Maxim Fedotov, MegaTom, moertel, Simone Carletti, Surya
35	Hora	giniouxe, Lucas Costa, MegaTom, stevendaniels
36	Instalación	Kathryn, Saša Zejnilović
37	instancia_eval	Matheus Moreira
38	Instrumentos de cuerda	AJ Gregory, br3nt, Charlie Egan, Community, David Grayson, davidhu2000, Jon Ericson, Julian Kohlman, Kathryn, Lucas Costa, Lukas Baliak, meagar, Muhammad Abdullah, NateW, Nick Roz, Phil Ross, Richard Hamilton, sandstrom, Sid, Simone Carletti, Steve, Vasfed, Velocibadgery, wjordan
39	Introspección	Felix, giniouxe, Justin Chadwell, MegaTom, mnoronha, Phrogz
40	Introspección en rubí	Engr. Hasanuzzaman Sumon, suhao399
41	IRB	David Grayson, Maxim Fedotov, Saša Zejnilović

42	Iteración	Charan Kumar Borra, Chris, Eli Sadoff, giniouxe, JCorcuera, Maxim Pontyushenko, MegaTom, ndn, Nick Roz, Ozgur Akyazi, Qstreet, SajithP, Simone Carletti
43	JSON con Ruby	Alu
44	La verdad	giniouxe, Umang Raghuvanshi
45	Las clases	br3nt, davidhu2000, Elenian, Eric Bouchut, giniouxe, JoeyB, Jon Wood, Justin Chadwell, Lukas Baliak, Martin Velez, MegaTom, Mhmd, Nick Roz, nus, philomory, Simone Carletti, spencer.sm, stevendaniels, thesecretmaster
46	Los operadores	ArtOfCode, Jonathan, nus, Phrogz, Tom Harrison Jr
47	Metaprogramacion	C dot StrifeVII, giniouxe, Matheus Moreira, MegaTom, meta, Phrogz, Pooyan Khosravi, Simon Soriano, Sourabh Upadhyay
48	método_missing	Adam Sanderson, Artur Tsuda, mnoronha, Nick Roz, Tom Harrison Jr, Yule
49	Métodos	Adam Sanderson, Artur Tsuda, br3nt, David Ljung Madison, fairchild, giniouxe, Kathryn, mahatmanich, Nick Podratz, Nick Roz, nus, Redithion, Simone Carletti, Szymon Włochowski, Thomas Gerot, Zaz
50	Modificadores de acceso Ruby	Neha Chopra
51	Módulos	giniouxe, Lynn, MegaTom, mrcasals, nus, RamenChef, Vasfed
52	Números	alexunger, Eli Sadoff, ndn, Redithion, Richard Hamilton, Simone Carletti, Steve, Tom Lord, wirefox
53	Operaciones de archivo y E / S	Doodad, KARASZI István, Martin Velez, max pleaner, Milo P, mnoronha, Nuno Silva, thesecretmaster
54	Operador Splat (*)	Kathryn
55	OptionParser	Kathryn
56	Parches de mono en rubí	Dorian, paradoja, RamenChef
57	Paso de mensajes	Pooyan Khosravi
58	Patrones de diseño y modismos en Ruby	4444, alexunger, Ali MasudianPour, Divya Sharma, djaszczurowski, Lucas Costa, user1213904
59	Pruebas de la API	equivalent8, RamenChef

	JSON de RSpec pura	
60	rbenv	Kathryn, Vidur
61	Receptores implícitos y comprensión del yo	Andrew
62	Recursion en Ruby	jphager2, Kathryn, SajithP
63	Refinamientos	max pleaner, xavdid
64	Ruby Version Manager	Alu, giniouxe, Hardik Kanjariya ツ
65	Simbolos	Artur Tsuda, Arun Kumar M, Nick Podratz, Owen, pjrebsch, Pooyan Khosravi, Simone Carletti, Tom Lord, walid
66	Sistema operativo o comandos de shell	Roan Fourie
67	Struct	Matheus Moreira
68	Uso de gemas	Anthony Staunton, Brian, Inanc Gumus, mnoronha, MZaragoza, NateSHolland, Saša Zejnilović, SidOfc, Simone Carletti, thesecretmaster, Tom Lord, user1489580
69	Variables de entorno	Lucas Costa, mnoronha, snonov