eBook Gratuit

APPRENEZ Ruby Language

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#ruby

Table des matières

À propos

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

Chapitre 1: Démarrer avec Ruby Language

Remarques

Ruby est un langage interprété, dynamique, orienté objet et multi-plateforme, conçu pour être simpliste et productif. Il a été créé par Yukihiro Matsumoto (Matz) en 1995.

> Selon son créateur, Ruby a été influencé par Perl , Smalltalk , Eiffel , Ada et Lisp . Il prend en charge plusieurs paradigmes de programmation, notamment fonctionnels, orientés objet et impératifs. Il dispose également d'un système de type dynamique et d'une gestion automatique de la mémoire.

Versions

Version	Date de sortie
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

Bonjour tout le monde de l'IRB

Vous pouvez également utiliser le shell interactif Ruby (IRB) pour exécuter immédiatement les instructions Ruby que vous avez précédemment écrites dans le fichier Ruby.

Commencez une session IRB en tapant:

```
$ irb
```

Puis entrez la commande suivante:

```
puts "Hello World"
```

Cela se traduit par la sortie de console suivante (y compris newline):

```
Hello World
```

Si vous ne voulez pas commencer une nouvelle ligne, vous pouvez utiliser `print` :

```
print "Hello World"
```

Bonjour tout le monde avec tk

Tk est l'interface graphique standard pour Ruby. Il fournit une interface graphique multi-plateforme pour les programmes Ruby.

Exemple de code:

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

Le résultat:

Explication étape par étape:

```
require "tk"
```

Chargez le paquet tk.

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

Définir un widget avec le titre `Hello World`

```
Tk.mainloop
```

Démarrez la boucle principale et affichez le widget.

Bonjour le monde

> Cet exemple suppose que Ruby est installé.

Placez les éléments suivants dans un fichier nommé `hello.rb` :

```
puts 'Hello World'
```

À partir de la ligne de commande, tapez la commande suivante pour exécuter le code Ruby à partir du fichier source:

```
$ ruby hello.rb
```

Cela devrait sortir:

```
Hello World
```

La sortie sera immédiatement affichée sur la console. Les fichiers source Ruby n'ont pas besoin d'être compilés avant d'être exécutés. L'interpréteur Ruby compile et exécute le fichier Ruby à l'exécution.

Hello World sans fichiers sources

Exécutez la commande ci-dessous dans un shell après avoir installé Ruby. Cela montre comment vous pouvez exécuter des programmes Ruby simples sans créer de fichier Ruby:

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

Vous pouvez également envoyer un programme Ruby à l'entrée standard de l'interpréteur. Une façon de faire est d'utiliser un document ici dans votre commande shell:

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

Hello World en tant que fichier auto-exécutable - en utilisant Shebang (systèmes d'exploitation de type Unix uniquement)

Vous pouvez ajouter une directive interpréteur (shebang) à votre script. Créez un fichier appelé `hello_world.rb` qui contient:

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

puts 'Hello World!'
```

Attribuez au script des autorisations exécutables. Voici comment faire cela dans Unix:

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

Maintenant, vous n'avez pas besoin d'appeler l'interpréteur Ruby explicitement pour exécuter votre script.

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

Ma première méthode

Vue d'ensemble

Créez un nouveau fichier nommé `my_first_method.rb`

Placez le code suivant dans le fichier:

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

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

Maintenant, à partir d'une ligne de commande, exécutez les opérations suivantes:

```
ruby my_first_method.rb
```

Le résultat devrait être:

```
Hello world!
```

Explication

- `def` est un mot - clé qui nous dit que nous `def` -ining une méthode - dans ce cas, `hello_world` est le nom de notre méthode.
- `puts "Hello world!"` `puts` (ou pipes à la console) la chaine `Hello world!`
- `end` est un mot-clé qui signifie que nous `hello_world` notre définition de la méthode `hello_world`
- comme la méthode `hello_world` n'accepte aucun argument, vous pouvez omettre la parenthèse en appelant la méthode

Lire Démarrer avec Ruby Language en ligne: https://riptutorial.com/fr/ruby/topic/195/demarrer-avec-ruby-language

Chapitre 2: Applications en ligne de commande

Examples

Comment écrire un outil en ligne de commande pour obtenir la météo par code postal

Ce sera un tutoriel relativement complet sur la façon d'écrire un outil de ligne de commande pour imprimer la météo à partir du code postal fourni avec l'outil de ligne de commande. La première étape consiste à écrire le programme en ruby pour effectuer cette action. Commençons par écrire une méthode `weather(zip_code)` (cette méthode nécessite le gem `yahoo_weatherman` . Si vous ne possédez pas cette gem, vous pouvez l'installer en tapant `gem install yahoo_weatherman` partir de la ligne de commande)

```
require 'yahoo_weatherman'

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

Nous avons maintenant une méthode très basique qui donne la météo lorsqu'un code postal lui est fourni. Maintenant, nous devons en faire un outil de ligne de commande. Très rapidement, voyons comment un outil en ligne de commande est appelé depuis le shell et les variables associées. Lorsqu'un outil est appelé comme cet `tool argument other_argument` , dans ruby il y a une variable `ARGV` qui est un tableau égal à `['argument', 'other_argument']` . Maintenant, laissez-nous implémenter cela dans notre application

```
#!/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])
```

Bien! Nous avons maintenant une application en ligne de commande qui peut être exécutée. Remarquez la ligne *she-bang* au début du fichier (`#!/usr/bin/ruby`). Cela permet au fichier de devenir un exécutable. Nous pouvons enregistrer ce fichier comme `weather` . (**Remarque** : ne sauvegardez pas ceci comme `weather.rb` , il n'y a pas besoin de l'extension de fichier et le she-bang indique ce que vous avez besoin de dire qu'il s'agit d'un fichier ruby). Maintenant, nous pouvons exécuter ces commandes dans le shell (ne tapez pas le `$`).

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

Après avoir testé que cela fonctionne, nous pouvons maintenant créer un lien symétrique avec `/usr/bin/local/` en exécutant cette commande

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

Maintenant, le `weather` peut être appelé sur la ligne de commande, quel que soit le répertoire dans lequel vous vous trouvez.

Lire Applications en ligne de commande en ligne:
https://riptutorial.com/fr/ruby/topic/7679/applications-en-ligne-de-commande

Chapitre 3: Arguments de mots clés

Remarques

Les arguments de mots-clés ont été introduits dans Ruby 2.0 et améliorés dans Ruby 2.1 avec l'ajout d'arguments de mots-clés *requis* .

Une méthode simple avec un argument mot-clé ressemble à la suivante:

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

say
# => "Hello World"

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

Pour rappel, la même méthode sans argument mot-clé aurait été:

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

say
# => "Hello World"

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

2.0

Vous pouvez simuler un argument de mot clé dans les versions précédentes de Ruby en utilisant un paramètre Hash. C'est encore une pratique très courante, en particulier dans les bibliothèques qui offrent une compatibilité avec les versions antérieures à la version 2.0 de Ruby:

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

say
# => "Hello World"

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

Examples

Utilisation des arguments de mots clés

Vous définissez un argument de mot clé dans une méthode en spécifiant le nom dans la définition de la méthode:

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

say
# => "Hello World"

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

Vous pouvez définir plusieurs arguments de mot-clé, l'ordre de définition n'est pas pertinent:

```
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>"
```

Les arguments de mots clés peuvent être mélangés avec des arguments positionnels:

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

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

Le mélange de l'argument mot-clé avec l'argument positionnel était une approche très courante avant Ruby 2.1, car il n'était pas possible de définir les arguments de mot clé requis .

De plus, dans Ruby <2.0, il était très courant d'ajouter un `Hash` à la fin d'une définition de méthode à utiliser pour les arguments facultatifs. La syntaxe est très similaire aux arguments de mot-clé, au point que les arguments facultatifs via `Hash` sont compatibles avec les arguments de mots-clés 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>"
```

Notez que si vous tentez de transmettre un argument de mot-clé non défini, une erreur se produira:

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

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

Arguments de mots clés requis

2.1

Les arguments de mots clés requis ont été introduits dans Ruby 2.1, en tant qu'amélioration des arguments de mots clés.

Pour définir un argument de mot-clé comme requis, déclarez simplement l'argument sans valeur par défaut.

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

say
# => ArgumentError: missing keyword: message

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

Vous pouvez également mélanger les arguments de mots clés requis et non obligatoires:

```
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>"
```

Utilisation d'arguments de mots clés arbitraires avec l'opérateur splat

Vous pouvez définir une méthode pour accepter un nombre arbitraire d'arguments par mot clé en utilisant l'opérateur *double splat* (`**`):

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

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

Les arguments sont capturés dans un `Hash` . Vous pouvez manipuler le `Hash` , par exemple pour extraire les arguments souhaités.

```
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
```

L'utilisation d'un opérateur splat avec des arguments de mot-clé empêchera la validation des arguments, la méthode ne déclenchera jamais une `ArgumentError` en cas de mot-clé inconnu.

En ce qui concerne l'opérateur splat standard, vous pouvez reconvertir un `Hash` en arguments de mots-clés pour une méthode:

```
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
```

Ceci est généralement utilisé lorsque vous devez manipuler des arguments entrants et les transmettre à une méthode sous-jacente:

```
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
```

Lire Arguments de mots clés en ligne: https://riptutorial.com/fr/ruby/topic/5253/arguments-de-mots-cles

Chapitre 4: Blocs et Procs et Lambdas

Syntaxe

- Proc.new (*bloc*)
- lambda {| args | code}
- -> (arg1, arg2) {code}
- object.to_proc
- {| single_arg | code}
- do | arg, (clé, valeur) | fin du *code*

Remarques

Faites attention à la priorité des opérateurs lorsque vous avez une ligne avec plusieurs méthodes enchaînées, comme:

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

Au lieu d'imprimer quelque chose comme `abCDeFg` , comme on pouvait s'y attendre, il imprime quelque chose comme `#<Enumerator:0x00000000af42b28>` - c'est parce que `do ... end` a priorité moindre que les méthodes, ce qui signifie que `gsub` ne voit que l' `/./` arguments , et non l'argument de bloc. Il retourne un énumérateur. Le bloc finit par passer à `puts` , qui l'ignore et affiche simplement le résultat de `gsub(/./)` .

Pour résoudre ce problème, `gsub` appel `gsub` entre parenthèses ou utilisez plutôt `{ ... }` .

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"
```

Nous avons copié la méthode `call_the_block` du dernier exemple. Ici, vous pouvez voir qu'un processus est fait en appelant la méthode `proc` avec un bloc. Vous pouvez également voir que les blocs, comme les méthodes, ont des retours implicites, ce qui signifie que procs (et lambdas) le

font aussi. Dans la définition de `its_a` , vous pouvez voir que les blocs peuvent prendre des arguments splat aussi bien que des arguments normaux; ils sont également capables de prendre des arguments par défaut, mais je ne pouvais pas penser à un moyen de travailler avec. Enfin, vous pouvez voir qu'il est possible d'utiliser plusieurs syntaxes pour appeler une méthode - soit la méthode d' `call` , soit le `[]` opérateur

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)
```

Vous pouvez également utiliser `->` pour créer et `.()` Pour appeler 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
```

Ici, vous pouvez voir qu'un lambda est presque identique à un proc. Cependant, il y a plusieurs mises en garde:

- L'arité des arguments d'un lambda est appliquée; transmettre le nombre d'arguments incorrect à un lambda, déclenche une erreur `ArgumentError` . Ils peuvent toujours avoir des paramètres par défaut, des paramètres de splat, etc.

- `return` depuis un lambda retourne du lambda, tout en `return` d'un proc retourne hors de la portée englobante:

```
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"
```

Objets en tant qu'arguments de bloc aux méthodes

Mettre un `&` (esperluette) devant un argument le transmettra comme bloc de la méthode. Les objets seront convertis en un `Proc` utilisant la méthode `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)
```

Ceci est un modèle courant dans Ruby et de nombreuses classes standard le fournissent.

Par exemple, `Symbol` implémente `to_proc` en s'envoyant à l'argument:

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

Cela active l'idiome utile `&:symbol` , couramment utilisé avec les objets `Enumerable` :

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

Blocs

Les blocs sont des morceaux de code entre accolades `{}` (généralement pour les blocs à une seule ligne) ou `do..end` (utilisés pour les blocs à plusieurs lignes).

```
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
```

Remarque: les accolades ont une priorité plus élevée que `do..end`

Céder

Les blocs peuvent être utilisés dans les méthodes et les fonctions en utilisant le mot `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
```

Attention, si `yield` est appelé sans bloc, il `LocalJumpError` une `LocalJumpError` . `block_given?` fournit à cette fin une autre méthode appelée `block_given?` cela vous permet de vérifier si un bloc a été passé avant d'appeler le rendement

```
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` peut également fournir des arguments au bloc

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

Bien qu'il s'agisse d'un exemple simple, le `yield` peut être très utile pour permettre un accès direct aux variables d'instance ou aux évaluations dans le contexte d'un autre objet. Par exemple:

```
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>
```

Comme vous pouvez le constater, l'utilisation de `yield` de cette manière rend le code plus lisible que l'appel continu de `app.configuration.#method_name` . Au lieu de cela, vous pouvez effectuer toute la configuration à l'intérieur du bloc en conservant le code contenu.

Les variables

Les variables pour les blocs sont locales au bloc (similaire aux variables des fonctions), elles meurent lorsque le bloc est exécuté.

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

Les blocs ne peuvent pas être enregistrés, ils meurent une fois exécutés. Pour enregistrer des blocs, vous devez utiliser `procs` et `lambdas` .

Conversion en Proc

Les objets qui répondent à `to_proc` peuvent être convertis en procs avec l'opérateur `&` (qui leur permettra également d'être transmis en tant que blocs).

La classe Symbol définit `#to_proc` afin qu'elle tente d'appeler la méthode correspondante sur l'objet qu'elle reçoit en paramètre.

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

Les objets méthode définissent également `#to_proc` .

```
output = method( :p )

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

Application partielle et currying

Techniquement, Ruby n'a pas de fonctions, mais des méthodes. Cependant, une méthode Ruby se comporte presque de manière identique aux fonctions dans un autre langage:

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

Cette méthode / fonction normale prend un paramètre `n` , le double et renvoie la valeur. Définissons maintenant une fonction d'ordre supérieur (ou méthode):

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

Au lieu de renvoyer un nombre, `triple` renvoie une méthode. Vous pouvez le tester à l'aide d' Interactive Ruby 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 vous voulez réellement obtenir le nombre triplé, vous devez appeler (ou "réduire") le lambda:

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

Ou plus concis:

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

Currying et applications partielles

Cela n'est pas utile pour définir des fonctionnalités très basiques, mais cela est utile si vous voulez avoir des méthodes / fonctions qui ne sont pas appelées ou réduites instantanément. Par exemple, supposons que vous souhaitiez définir des méthodes qui ajoutent un nombre à un nombre spécifique (par exemple, `add_one(2) = 3`). Si vous deviez en définir une tonne, vous pourriez faire:

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

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

Cependant, vous pourriez aussi faire ceci:

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

En utilisant le lambda calcul, on peut dire que `add` est `(λa.(λb.(a+b)))` . Le curry est une façon d' *appliquer partiellement* `add` . Donc `add.curry.(1)` , est `(λa.(λb.(a+b)))(1)` qui peut être réduit à `(λb.(1+b))` . L'application partielle signifie que nous avons passé un argument à `add` mais que l'autre argument a été fourni ultérieurement. La sortie est une méthode spécialisée.

Des exemples plus utiles de currying

Disons que nous avons une très grande formule générale, que si nous spécifions certains arguments, nous pouvons en tirer des formules spécifiques. Considérez cette formule:

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

Cette formule est conçue pour fonctionner en trois dimensions, mais supposons que nous ne souhaitons que cette formule en ce qui concerne y et z. Disons aussi que pour ignorer x, nous voulons lui donner la valeur pi / 2. Faisons d'abord la formule générale:

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

Maintenant, utilisons le curry pour obtenir notre formule `yz` :

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

Ensuite, appeler le lambda stocké dans `f_yz` :

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

C'est assez simple, mais disons que nous voulons obtenir la formule pour `xz` . Comment pouvons-nous régler `y Math::PI/2` si ce n'est pas le dernier argument? Eh bien, c'est un peu plus compliqué:

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

Dans ce cas, nous devons fournir des espaces réservés pour le paramètre que nous ne pré-remplissons pas. Pour plus de cohérence, nous pourrions écrire `f_xy` comme ceci:

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

Voici comment fonctionne le calcul lambda pour `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)))))
```

Maintenant, regardons `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))))
```

Pour plus d'informations sur le lambda, essayez ceci .

Lire Blocs et Procs et Lambdas en ligne: https://riptutorial.com/fr/ruby/topic/474/blocs-et-procs-et-lambdas

Chapitre 5: C Extensions

Examples

Votre première extension

Les extensions C sont composées de deux pièces générales:

1. Le code C lui-même.
2. Le fichier de configuration d'extension.

Pour commencer avec votre première extension, placez-le dans un fichier nommé `extconf.rb` :

```
require 'mkmf'

create_makefile('hello_c')
```

Quelques points à souligner:

D'abord, le nom `hello_c` est ce que la sortie de votre extension compilée va être nommée. Ce sera ce que vous utiliserez en même temps que `require` .

Deuxièmement, le fichier `extconf.rb` peut en fait être nommé n'importe quoi, il est juste traditionnellement utilisé pour construire des gemmes avec du code natif, le fichier qui va en fait compiler l'extension est le Makefile généré lors de l'exécution de `ruby extconf.rb` . Le fichier `.c` par défaut généré compile tous les fichiers `.c` répertoire en cours.

Placez ce qui suit dans un fichier nommé `hello.c` et exécutez `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);
}
```

Une ventilation du code:

Le nom `Init_hello_c` doit correspondre au nom défini dans votre fichier `extconf.rb` , sinon, lors du chargement dynamique de l'extension, Ruby ne pourra pas trouver le symbole permettant d'amorcer votre extension.

L'appel à `rb_define_module` est en `rb_define_module` créer un module Ruby nommé `HelloC` lequel

nous allons `HelloC` nos fonctions C sous l'espace de noms.

Enfin, l'appel à `rb_define_singleton_method` crée une méthode de niveau module directement liée au module `HelloC` que nous pouvons appeler depuis ruby avec `HelloC.world` .

Après avoir compilé l'extension avec l'appel à `make` nous pouvons exécuter le code dans notre extension C.

Allumez une console!

```
irb(main):001:0> require './hello_c'
=> true
irb(main):002:0> HelloC.world
Hello World!
=> nil
```

Travailler avec des structures C

Pour pouvoir travailler avec des structures C en tant qu'objets Ruby, vous devez les envelopper avec des appels à `Data_Wrap_Struct` et `Data_Get_Struct` .

`Data_Wrap_Struct` une structure de données C dans un objet Ruby. Il prend un pointeur sur votre structure de données, ainsi que quelques pointeurs vers des fonctions de rappel, et renvoie une valeur. La macro `Data_Get_Struct` prend cette valeur et vous renvoie un pointeur sur votre structure de données C.

Voici un exemple 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);
}
```

Et l' `extconf.rb` :

```
require 'mkmf'

create_makefile('example')
```

Après avoir compilé l'extension:

```
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
```

Écrire en ligne C - RubyInLine

RubyInline est un framework qui vous permet d'intégrer d'autres langages dans votre code Ruby. Il définit la méthode inline du module #, qui renvoie un objet générateur. Vous transmettez au générateur une chaîne contenant du code écrit dans une langue autre que Ruby, et le générateur la transforme en quelque chose que vous pouvez appeler depuis Ruby.

Lorsqu'il est donné du code C ou C ++ (les deux langages pris en charge dans l'installation par défaut de RubyInline), les objets du générateur écrivent une petite extension sur le disque, la compilent et la chargent. Vous n'avez pas à gérer la compilation vous-même, mais vous pouvez voir le code généré et les extensions compilées dans le sous-répertoire .ruby_inline de votre répertoire personnel.

Intégrez le code C directement dans votre programme Ruby:

- RubyInline (disponible en tant que gem rubyinline) crée automatiquement une extension

  RubyInline ne fonctionnera pas depuis 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 fonction C `copy_file` existe maintenant comme méthode d'instance 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 }
```

Lire C Extensions en ligne: https://riptutorial.com/fr/ruby/topic/5009/c-extensions

Chapitre 6: Casting (conversion de type)

Examples

Casting à une chaîne

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

En règle générale, `String()` appelle simplement `#to_s` .

Les méthodes `Kernel#sprintf` et `String#%` se comportent comme C:

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

Casting à un entier

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

Une chaîne prend la valeur de tout entier à son début, mais ne prend pas les entiers ailleurs:

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

Cependant, il y a une différence lorsque la chaîne n'est pas un nombre entier valide:

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

Casting à un flotteur

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

Cependant, il y a une différence lorsque la chaîne n'est pas un `Float` valide:

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

Flotteurs et Entiers

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

Puisque nous divisons deux entiers, le résultat est un entier. Pour résoudre ce problème, nous devons lancer au moins l'un de ceux-ci sur Float:

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

Alternativement, `fdiv` peut être utilisé pour retourner le résultat en virgule flottante de la division sans lancer explicitement l'autre opérande:

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

Lire Casting (conversion de type) en ligne: https://riptutorial.com/fr/ruby/topic/219/casting--conversion-de-type-

Chapitre 7: Catching Exceptions avec Begin / Rescue

Examples

Un bloc de gestion des erreurs de base

Faisons une fonction pour diviser deux nombres, cela fait très confiance à ses entrées:

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

Cela fonctionnera très bien pour beaucoup d'entrées:

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

Mais pas tout

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

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

Nous pouvons réécrire la fonction en encapsulant l'opération de division risquée dans un bloc `begin... end` pour vérifier les erreurs et utiliser une clause `rescue` pour générer un message et renvoyer `nil` s'il y a un problème.

```
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
```

Enregistrement de l'erreur

Vous pouvez enregistrer l'erreur si vous souhaitez l'utiliser dans la clause 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>'
```

Vérification des erreurs différentes

Si vous voulez faire des choses différentes en fonction du type d'erreur, utilisez plusieurs clauses de `rescue` , chacune avec un type d'erreur différent en tant qu'argument.

```
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 vous souhaitez enregistrer l'erreur à utiliser dans le bloc de `rescue` :

```
rescue ZeroDivisionError => e
```

Utilisez une clause de `rescue` sans argument pour intercepter les erreurs d'un type non spécifié dans une autre clause 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)
```

Dans ce cas, essayer de diviser `nil` par 2 n'est pas un `ZeroDivisionError` ou un `TypeError` , donc il est géré par la clause de `rescue` par défaut, qui imprime un message pour nous informer qu'il s'agit d'une `NoMethodError` .

Réessayer

Dans une clause de `rescue` , vous pouvez `retry` pour exécuter la clause `begin` , probablement après avoir modifié les circonstances à l'origine de l'erreur.

```
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 nous transmettons des paramètres dont nous savons qu’ils provoquent une `TypeError` , la

clause `begin` est exécutée (marquée ici en imprimant "About to divide") et l'erreur est interceptée comme précédemment, et `nil` est renvoyé:

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

Mais si nous transmettons des paramètres qui provoquent une `ZeroDivisionError` , la clause `begin` est exécutée, l'erreur est interceptée, le diviseur passe de 0 à 1, puis `retry` l'exécution du bloc de `begin` (à partir du haut), maintenant avec un différent `y` . La deuxième fois, il n'y a pas d'erreur et la fonction renvoie une valeur.

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

Vérification de l'absence de toute erreur

Vous pouvez utiliser une clause `else` pour le code qui sera exécuté si aucune erreur n'est générée.

```
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 clause `else` ne s'exécute pas si une erreur transfère le contrôle à l'une des clauses de `rescue` :

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

Mais si aucune erreur n'est déclenchée, la clause `else` s'exécute:

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

Notez que la clause `else` ne sera pas exécutée *si vous revenez de la clause* `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
```

Code qui doit toujours s'exécuter

Utilisez une clause `ensure` si vous souhaitez toujours exécuter du code.

```
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 clause `ensure` sera exécutée en cas d'erreur:

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

Et quand il n'y a pas d'erreur:

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

La clause assure est utile lorsque vous voulez vous assurer, par exemple, que les fichiers sont fermés.

Notez que, contrairement à la clause `else` , la clause `ensure` *est exécutée* avant que la clause `begin`

ou `rescue` renvoie une valeur. Si la clause `ensure` a un `return` qui remplace la valeur de `return` de toute autre clause!

Lire Catching Exceptions avec Begin / Rescue en ligne:
https://riptutorial.com/fr/ruby/topic/7327/catching-exceptions-avec-begin---rescue

Chapitre 8: Chargement des fichiers source

Examples

Les fichiers requis ne doivent être chargés qu'une seule fois

La méthode kernel # require ne chargera les fichiers qu'une seule fois (plusieurs appels à `require` auront pour résultat que le code de ce fichier sera évalué une seule fois). Il cherchera votre ruby `$LOAD_PATH` pour trouver le fichier requis si le paramètre n'est pas un chemin absolu. Les extensions comme `.rb` , `.so` , `.o` ou `.dll` sont facultatives. Les chemins relatifs seront résolus dans le répertoire de travail en cours du processus.

```
require 'awesome_print'
```

Le noyau # require_relative vous permet de charger des fichiers relatifs au fichier dans lequel `require_relative` est appelé.

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

Chargement automatique des fichiers source

La méthode `Kernel#autoload` enregistre le nom du fichier à charger (en utilisant `Kernel::require`) la première fois que ce module (qui peut être un String ou un symbole) est accédé.

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

La méthode Kernel # autoload? renvoie le nom du fichier à charger si name est enregistré en tant que `autoload` .

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

Chargement de fichiers optionnels

Lorsque les fichiers ne sont pas disponibles, la famille `require` lance une `LoadError` . Ceci est un exemple qui illustre le chargement de modules optionnels uniquement s'ils existent.

```
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
```

Chargement de fichiers à plusieurs reprises

La méthode de chargement Noyau # évalue le code dans le fichier donné. Le chemin de recherche sera construit comme `require` . Il réévaluera ce code à chaque appel suivant, contrairement à la `require` . Il n'y a pas de `load_relative` .

```
load `somefile`
```

Chargement de plusieurs fichiers

Vous pouvez utiliser n'importe quelle technique Ruby pour créer dynamiquement une liste de fichiers à charger. Illustration de la mise en forme pour les fichiers commençant par `test` , chargés dans l'ordre alphabétique.

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

    require_relative source

end
```

Lire Chargement des fichiers source en ligne: https://riptutorial.com/fr/ruby/topic/3166/chargement-des-fichiers-source

Chapitre 9: Classe Singleton

Syntaxe

- singleton_class = class << objet; fin de soi

Remarques

Les classes singleton n'ont qu'une seule instance: leur objet correspondant. Cela peut être vérifié en interrogeant le `ObjectSpace` de Ruby:

```
instances = ObjectSpace.each_object object.singleton_class

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

En utilisant `<` , ils peuvent également être vérifiés pour être des sous-classes de la classe réelle de l'objet:

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

Les références:

- Trois contextes implicites dans Ruby

Examples

introduction

Ruby a trois types d'objets:

- Classes et modules qui sont des instances de classe Class ou class Module.
- Instances de classes.
- Classes Singleton.

Chaque objet a une classe qui contient ses méthodes:

```
class Example
end

object = Example.new

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

Les objets eux-mêmes ne peuvent pas contenir de méthodes, seule leur classe le peut. Mais avec les classes singleton, il est possible d'ajouter des méthodes à n'importe quel objet, y compris d'autres classes singleton.

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

`foo` est défini sur la classe singleton de l' `object` . Autres `Example` exemples ne peuvent pas répondre à `foo` .

Ruby crée des classes singleton à la demande. Leur accès ou leur ajout de méthodes force Ruby à les créer.

Accéder à la classe Singleton

Il y a deux façons d'obtenir la classe singleton d'un objet

- Méthode `singleton_class` .
- Réouverture de la classe singleton d'un objet et retour `self` .

```
object.singleton_class
```

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

Accès aux variables d'instance / classe dans les classes Singleton

Les classes Singleton partagent leurs variables d'instance / classe avec leur objet.

```
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
```

Les blocs se rapprochent de la cible de leurs variables instance / classe. L'accès aux variables d'instance ou de classe à l'aide d'un bloc dans `class_eval` ou `instance_eval` n'est pas possible. Passer une chaîne à `class_eval` ou utiliser `class_variable_get` fonctionne autour du problème.

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

class Example
  @@foo = :example

  Foo.define_singleton_method :foo do
    @@foo
  end
end

Foo.foo #=> :example
```

Héritage de la classe Singleton

Le sous-classement comprend également les sous-classes de la classe 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>
```

L'extension ou l'inclusion d'un module n'élargit pas la classe

Singleton

```
module ExampleModule
end

def ExampleModule.foo
  :foo
end

class Example
  extend ExampleModule
  include ExampleModule
end

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

Propagation des messages avec la classe Singleton

Les instances ne contiennent jamais une méthode qu'elles ne transportent que des données. Cependant, nous pouvons définir une classe singleton pour tout objet incluant une instance d'une classe.

Lorsqu'un message est transmis à un objet (la méthode est appelée), Ruby vérifie tout d'abord si une classe singleton est définie pour cet objet et si elle peut répondre à ce message, sinon Ruby vérifie la chaîne des ancêtres de la classe

```
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
```

Réouverture (correction de singe) Classes Singleton

Il existe trois façons de rouvrir une classe Singleton

- Utiliser `class_eval` sur une classe singleton.
- Utilisation de la `class <<` block
- Utilisation de `def` pour définir une méthode directement sur la classe singleton de l'objet

```
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
```

Chaque objet a une classe singleton à laquelle vous pouvez accéder

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

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

Classes Singleton

Tous les objets sont des instances d'une classe. Cependant, ce n'est pas toute la vérité. Dans

Ruby, chaque objet a également une *classe singleton* quelque peu cachée.

C'est ce qui permet de définir des méthodes sur des objets individuels. La classe singleton se situe entre l'objet lui-même et sa classe réelle, de sorte que toutes les méthodes définies sur cet objet sont disponibles pour cet objet, et cet objet uniquement.

```
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>>
```

L'exemple ci-dessus aurait pu être écrit en utilisant `define_singleton_method` :

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

Ce qui revient à définir la méthode sur `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
```

Avant l'existence de `singleton_class` dans le cadre de l'API principale de Ruby, les classes singleton étaient connues sous le nom de *métaclasses* et pouvaient être accédées via l'idiome suivant:

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

Lire Classe Singleton en ligne: https://riptutorial.com/fr/ruby/topic/4277/classe-singleton

Chapitre 10: Commandes du système d'exploitation ou du shell

Introduction

Il existe plusieurs façons d'interagir avec le système d'exploitation. Depuis Ruby, vous pouvez exécuter des commandes ou des sous-processus shell / système.

Remarques

Exec:
Les fonctionnalités d'Exec sont très limitées et, une fois exécutées, elles quitteront le programme Ruby et exécuteront la commande.

La commande système:
La commande System s'exécute dans un sous-shell au lieu de remplacer le processus en cours et renvoie true ou nill. La commande système est, comme backticks, une opération de blocage dans laquelle l'application principale attend que le résultat de l'opération du système soit terminé. Ici, l'opération principale n'a jamais à se soucier de capturer une exception déclenchée par le processus enfant.

La sortie de la fonction système sera toujours vraie ou nulle selon que le script a été exécuté ou non. Par conséquent, chaque erreur lors de l'exécution du script ne sera pas transmise à notre application. L'opération principale n'a jamais à s'inquiéter de la capture d'une exception générée par le processus enfant. Dans ce cas, la sortie est nulle car le processus enfant a déclenché une exception.
C'est une opération de blocage où le programme Ruby attendra que l'opération de la commande se termine avant de continuer.
L'opération du système utilise fork pour bifurquer le processus en cours, puis exécute l'opération donnée en utilisant exec.

Les backticks (`):
Le caractère backtick est généralement situé sous la touche d'échappement sur le clavier.
Backticks s'exécute dans un sous-shell au lieu de remplacer le processus en cours et renvoie le résultat de la commande.
Ici, nous pouvons obtenir la sortie de la commande mais le programme se bloquera lorsqu'une exception est générée.
S'il existe une exception dans le sous-processus, cette exception est donnée au processus principal et le processus principal peut se terminer si l'exception n'est pas gérée. C'est une opération de blocage où le programme Ruby attendra que l'opération de la commande se termine avant de continuer.
L'opération du système utilise fork pour bifurquer le processus en cours, puis exécute l'opération donnée en utilisant exec.

IO.popen:
IO.popen s'exécute dans un sous-processus. Ici, l'entrée standard de sous-processus et la sortie standard sont connectées à l'objet IO.

Popen3:
Popen3 vous permet d'accéder à l'entrée standard, à la sortie standard et à l'erreur standard. L'entrée et la sortie standard du sous-processus seront renvoyées dans les objets IO.

$? (identique à $ CHILD_STATUS)
Peut être utilisé avec les opérations backticks, system () ou% x {} et donnera le statut de la dernière commande exécutée par le système.
Cela peut être utile pour accéder aux `exitstatus` et `pid` .

```
$?.exitstatus
```

Examples

Méthodes recommandées pour exécuter le code shell en Ruby:

Open3.popen3 ou Open3.capture3:
Open3 utilise en fait simplement la commande spawn de Ruby, mais vous offre une API bien meilleure.

Open3.popen3

Popen3 s'exécute dans un sous-processus et renvoie stdin, stdout, stderr et wait_thr.

```
require 'open3'
stdin, stdout, stderr, wait_thr = Open3.popen3("sleep 5s && ls")
puts "#{stdout.read} #{stderr.read} #{wait_thr.value.exitstatus}"
```

ou

```
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
```

va sortir: **stdout est: stderr est: fatal: Pas un dépôt git (ou l'un des répertoires parents): .git**

ou

```
require 'open3'
cmd = 'ping www.google.com'
Open3.popen3(cmd) do |stdin, stdout, stderr, wait_thr|
  while line = stdout.gets
    puts line
  end
```

```
end
```

va sortir:

En cliquant sur www.google.com [216.58.223.36] avec 32 octets de données:
Réponse de 216.58.223.36: octets = 32 fois = 16ms TTL = 54
Réponse de 216.58.223.36: octets = 32 fois = 10ms TTL = 54
Réponse de 216.58.223.36: octets = 32 heure = 21ms TTL = 54
Réponse de 216.58.223.36: octets = 32 fois = 29ms TTL = 54
Statistiques de ping pour 216.58.223.36:
Paquets: Envoyé = 4, Reçu = 4, Perdu = 0 (perte de 0%),
Temps d'aller-retour approximatifs en milli-secondes:
Minimum = 10ms, Maximum = 29ms, Moyenne = 19ms

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
```

ou

```
Open3.capture3('/some/binary with some args')
```

Non recommandé en raison des frais généraux supplémentaires et du potentiel d'injection de coquille.

Si la commande lit à partir de stdin et que vous voulez lui donner des données:

```
Open3.capture3('my_funky_command', stdin_data: 'read from stdin')
```

Exécutez la commande avec un répertoire de travail différent, en utilisant chdir:

```
Open3.capture3('my_funky_command', chdir: '/some/directory')
```

Des moyens classiques pour exécuter du code shell en Ruby:

Exec:

```
exec 'echo "hello world"'
```

ou

```
exec ('echo "hello world"')
```

La commande système:

```
system 'echo "hello world"'
```

Produira "hello world" dans la fenêtre de commande.

ou

```
system ('echo "hello world"')
```

La commande système peut renvoyer un true si la commande a réussi ou non.

```
result = system 'echo "hello world"'
puts result  # will return a true in the command window
```

Les backticks (`):

`echo "hello world"` Affiche "hello world" dans la fenêtre de commande.

Vous pouvez également attraper le résultat.

```
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 }
```

Lire Commandes du système d'exploitation ou du shell en ligne:
https://riptutorial.com/fr/ruby/topic/10921/commandes-du-systeme-d-exploitation-ou-du-shell

Chapitre 11: commentaires

Examples

Commentaires sur lignes simples et multiples

Les commentaires sont des annotations lisibles par le programmeur qui sont ignorées lors de l'exécution. Leur but est de faciliter la compréhension du code source.

Commentaires sur une seule ligne

Le caractère `#` est utilisé pour ajouter des commentaires sur une seule ligne.

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

Une fois exécuté, le programme ci-dessus affichera `Hello World!`

Commentaires multilignes

Des commentaires sur plusieurs lignes peuvent être ajoutés à l'aide de la syntaxe `=begin` et `=end` (également connue sous le nom de marqueurs de bloc de commentaires) comme suit:

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

Une fois exécuté, le programme ci-dessus affichera `Hello World!`

Lire commentaires en ligne: https://riptutorial.com/fr/ruby/topic/3464/commentaires

Chapitre 12: Comparable

Syntaxe

- `include Comparable`
- mettre en œuvre l'opérateur de vaisseau spatial (`<=>`)

Paramètres

Paramètre	Détails
autre	L'instance à comparer à `self`

Remarques

`x <=> y` devrait renvoyer un nombre négatif si `x < y` , zéro si `x == y` et un nombre positif si `x > y` .

Examples

Rectangle comparable par zone

`Comparable` est l'un des modules les plus populaires de Ruby. Son but est de fournir des méthodes de comparaison de commodité.

Pour l'utiliser, vous devez `include Comparable` et définir l'opérateur spatial (`<=>`):

```
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
```

Lire Comparable en ligne: https://riptutorial.com/fr/ruby/topic/1485/comparable

Chapitre 13: Constantes spéciales en rubis

Examples

__FICHIER__

Est le chemin relatif du fichier à partir du répertoire d'exécution actuel
Supposons que nous ayons cette structure de répertoire: /home/stackoverflow/script.rb
script.rb contient:

```
puts __FILE__
```

Si vous êtes dans / home / stackoverflow et exécutez le script comme `ruby script.rb` alors `__FILE__` affichera `script.rb` Si vous êtes à l'intérieur / home, cela produira `stackoverflow/script.rb`

Très utile pour obtenir le chemin du script dans les versions antérieures à 2.0 où `__dir__` n'existe pas.

Remarque `__FILE__` n'est pas égal à `__dir__`

__dir__

`__dir__` n'est pas une constante mais une fonction
`__dir__` est égal à `File.dirname(File.realpath(__FILE__))`

$ PROGRAM_NAME ou $ 0

Contient le nom du script en cours d'exécution.
Est le même que `__FILE__` si vous exécutez ce script.

$$

Le numéro de processus du Ruby exécutant ce script

1 $, 2 $, etc.

Contient le sous-motif du jeu de parenthèses correspondant dans le dernier motif réussi correspondant, sans compter les motifs correspondant à des blocs imbriqués déjà sortis ou nuls si le dernier motif correspondant a échoué. Ces variables sont toutes en lecture seule.

ARGV ou $ *

Arguments de ligne de commande donnés pour le script. Les options pour l'interpréteur Ruby sont déjà supprimées.

STDIN

L'entrée standard La valeur par défaut pour $ stdin

STDOUT

La sortie standard La valeur par défaut pour $ stdout

STDERR

La sortie d'erreur standard. La valeur par défaut pour $ stderr

$ stderr

La sortie d'erreur standard actuelle.

$ stdout

La sortie standard actuelle

$ stdin

L'entrée standard actuelle

ENV

L'objet de type hachage contient les variables d'environnement actuelles. La définition d'une valeur dans ENV modifie l'environnement pour les processus enfants.

Lire Constantes spéciales en rubis en ligne: https://riptutorial.com/fr/ruby/topic/4037/constantes-speciales-en-rubis

Chapitre 14: Cordes

Syntaxe

- 'Une chaîne' // crée une chaîne via un littéral entre guillemets simples
- "Une chaîne" // crée une chaîne via un littéral entre guillemets
- String.new ("Une chaîne")
- % q (Une chaîne) // syntaxe alternative pour créer des chaînes entre guillemets simples
- % Q (Une chaîne) // syntaxe alternative pour créer des chaînes entre guillemets

Examples

Différence entre les littéraux de chaîne entre guillemets simples et entre guillemets

La principale différence est que les littéraux de `String` guillemets doubles prennent en charge les interpolations de chaînes et l'ensemble complet des séquences d'échappement.

Par exemple, ils peuvent inclure des expressions Ruby arbitraires par interpolation:

```
# 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
```

Les chaînes entre guillemets prennent également en charge l' ensemble des séquences d'échappement, y compris `"\n"` , `"\t"` ...

```
puts 'Hello\nWorld'
# Hello\nWorld

puts "Hello\nWorld"
# Hello
# World
```

... tandis que les chaînes entre guillemets simples *ne* prennent *pas en* charge les séquences d'échappement, en mettant en évidence le jeu minimal nécessaire pour les chaînes entre guillemets simples: guillemets simples et barres obliques inverses, respectivement `'\''` et `'\\'` .

Créer une chaîne

Ruby propose plusieurs méthodes pour créer un objet `String` . La méthode la plus courante consiste à utiliser des guillemets simples ou doubles pour créer un " littéral de chaîne ":

```
s1 = 'Hello'
s2 = "Hello"
```

La principale différence est que les littéraux de chaîne entre guillemets sont un peu plus flexibles car ils supportent l'interpolation et certaines séquences d'échappement de barre oblique inverse.

Il existe également plusieurs autres manières possibles de créer un littéral de chaîne en utilisant des délimiteurs de chaîne arbitraires. Un délimiteur de chaîne arbitraire est un `%` suivi d'une paire de délimiteurs correspondants:

```
%(A string)
%{A string}
%<A string>
%|A string|
%!A string!
```

Enfin, vous pouvez utiliser les séquences `%q` et `%Q` , équivalentes à `'` et `"` ":

```
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` séquences `%q` et `%Q` sont utiles lorsque la chaîne contient des guillemets simples, des guillemets doubles ou un mélange des deux. De cette façon, vous n'avez pas besoin d'échapper au contenu:

```
%Q(<a href="/profile">User's profile<a>)
```

Vous pouvez utiliser plusieurs délimiteurs différents, à condition qu'il y ait une paire correspondante:

```
%q(A string)
%q{A string}
%q<A string>
%q|A string|
%q!A string!
```

Concaténation de chaînes

Concaténer des chaînes avec l'opérateur `+` :

```
s1 = "Hello"
s2 = " "
s3 = "World"

puts s1 + s2 + s3
# => Hello World
```

```
s = s1 + s2 + s3
puts s
# => Hello World
```

Ou avec le `<<` opérateur:

```
s = 'Hello'
s << ' '
s << 'World'
puts s
# => Hello World
```

Notez que l'opérateur `<<` modifie l'objet sur le côté gauche.

Vous pouvez également multiplier les chaînes, par exemple

```
"wow" * 3
# => "wowwowwow"
```

Interpolation de chaîne

Le délimiteur à double guillemets `"` et la séquence `%Q` prennent en charge l'interpolation de chaînes à l'aide de `#{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
```

Manipulation de cas

```
"string".upcase     # => "STRING"
"STRING".downcase   # => "string"
"String".swapcase   # => "sTRING"
"string".capitalize # => "String"
```

Ces quatre méthodes ne modifient pas le récepteur d'origine. Par exemple,

```
str = "Hello"
str.upcase # => "HELLO"
puts str   # => "Hello"
```

Il existe quatre méthodes similaires qui effectuent les mêmes actions mais modifient le récepteur d'origine.

```
"string".upcase!     # => "STRING"
"STRING".downcase!   # => "string"
"String".swapcase!   # => "sTRING"
"string".capitalize! # => "String"
```

Par exemple,

```
str = "Hello"
str.upcase!  # => "HELLO"
puts str     # => "HELLO"
```

Remarques:

- avant Ruby 2.4, ces méthodes ne gèrent pas Unicode.

Fractionner une chaîne

`String#split` divise une `String` en un `Array` basé sur un délimiteur.

```
"alpha,beta".split(",")
# => ["alpha", "beta"]
```

Une `String` vide donne un `Array` vide:

```
"".split(",")
# => []
```

Un délimiteur sans correspondance génère un `Array` contenant un seul élément:

```
"alpha,beta".split(".")
# => ["alpha,beta"]
```

Vous pouvez également diviser une chaîne en utilisant des expressions régulières:

```
"alpha, beta,gamma".split(/, ?/)
# => ["alpha", "beta", "gamma"]
```

Le délimiteur est facultatif, par défaut une chaîne est divisée sur des espaces:

```
"alpha beta".split
# => ["alpha", "beta"]
```

Joindre des chaînes

`Array#join` joint un `Array` dans une `String` , en fonction d'un délimiteur:

```
["alpha", "beta"].join(",")
# => "alpha,beta"
```

Le délimiteur est facultatif et sa valeur par défaut est une `String` vide.

```
["alpha", "beta"].join
# => "alphabeta"
```

Un `Array` vide génère une `String` vide, quel que soit le délimiteur utilisé.

```
[].join(",")
# => ""
```

Cordes multilignes

Le moyen le plus simple de créer une chaîne multiligne consiste à utiliser plusieurs lignes entre guillemets:

```
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."
```

Le principal problème de cette technique est que si la chaîne contient une citation, elle va casser la syntaxe de la chaîne. Pour contourner le problème, vous pouvez utiliser un heredoc à la place:

```
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 prend en charge les documents de style shell avec `<<EOT` , mais le texte de fin doit démarrer la ligne. Cela gâche l'indentation du code, il n'y a donc pas beaucoup de raisons d'utiliser ce style. Malheureusement, la chaîne aura des indentations en fonction de la façon dont le code lui-même est en retrait.

Ruby 2.3 résout le problème en introduisant `<<~` qui supprime les espaces de début excédentaires:

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
```

Les chaînes de pourcentage fonctionnent également pour créer des chaînes multilignes:

```
%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
)
```

Il existe plusieurs moyens d'éviter les séquences d'interpolation et d'échappement:

- Guillemet simple au lieu de guillemet double: `'\n is a carriage return.'`
- Minuscule `q` dans une chaîne de pourcentage: `%q[#{not-a-variable}]`
- Citation unique de la chaîne de terminal dans un heredoc:

```
<<-'CODE'
   puts 'Hello world!'
CODE
```

Chaînes formatées

Ruby peut injecter un tableau de valeurs dans une chaîne en remplaçant les espaces réservés par les valeurs du tableau fourni.

```
"Hello %s, my name is %s!" % ['World', 'br3nt']
# => Hello World, my name is br3nt!
```

Les `['Hello', 'br3nt']` sont représentés par deux `%s` et les valeurs sont fournies par le tableau `['Hello', 'br3nt']` . L'opérateur `%` indique à la chaîne d'injecter les valeurs du tableau.

Remplacements de caractères de chaîne

La méthode `tr` renvoie une copie d'une chaîne où les caractères du premier argument sont remplacés par les caractères du second argument.

```
"string".tr('r', 'l') # => "stling"
```

Pour remplacer uniquement la première occurrence d'un motif avec une autre expression, utilisez la méthode `sub`

```
"string ring".sub('r', 'l') # => "stling ring"
```

Si vous souhaitez remplacer *toutes les* occurrences d'un motif par cette expression, utilisez `gsub`

```
"string ring".gsub('r','l') # => "stling ling"
```

Pour supprimer des caractères, transmettez une chaîne vide pour le deuxième paramètre

Vous pouvez également utiliser des expressions régulières dans toutes ces méthodes.

Il est important de noter que ces méthodes ne renverront qu'une nouvelle copie d'une chaîne et ne modifieront pas la chaîne en place. Pour ce faire, vous devez utiliser le `tr!` , `sub!` et `gsub!` méthodes respectivement.

Comprendre les données dans une chaîne

Dans Ruby, une chaîne n'est qu'une séquence d' octets avec le nom d'un codage (tel que `UTF-8` , `US-ASCII` , `ASCII-8BIT`) qui spécifie comment interpréter ces octets en tant que caractères.

Les chaînes Ruby peuvent être utilisées pour contenir du texte (essentiellement une séquence de caractères), auquel cas le codage UTF-8 est généralement utilisé.

```
"abc".bytes  # => [97, 98, 99]
"abc".encoding.name  # => "UTF-8"
```

Les chaînes Ruby peuvent également être utilisées pour contenir des données binaires (une séquence d'octets), auquel cas le codage ASCII-8BIT est généralement utilisé.

```
[42].pack("i").encoding  # => "ASCII-8BIT"
```

Il est possible que la séquence d'octets d'une chaîne ne corresponde pas à l'encodage, ce qui entraîne des erreurs si vous essayez d'utiliser la chaîne.

```
"\xFF \xFF".valid_encoding? # => false
"\xFF \xFF".split(' ')      # ArgumentError: invalid byte sequence in UTF-8
```

Substitution de cordes

```
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"
```

Voir String `%` docs et Kernel :: sprintf pour plus de détails.

La chaîne commence par

Pour trouver si une chaîne commence par un motif, le `start_with?` la méthode est pratique

```
str = "zebras are cool"
str.start_with?("zebras")      => true
```

Vous pouvez également vérifier la position du motif avec l' `index`

```
str = "zebras are cool"
```

```
str.index("zebras").zero?      => true
```

La chaîne se termine par

Pour trouver si une chaîne se termine par un motif, `end_with?` la méthode est pratique

```
str = "I like pineapples"
str.end_with?("pineaaples")      => false
```

Chaînes de positionnement

En Ruby, les chaînes peuvent être justifiées à gauche, justifiées à droite ou centrées

Pour justifier la chaîne, utilisez la méthode `ljust` . Cela prend deux paramètres, un entier représentant le nombre de caractères de la nouvelle chaîne et une chaîne représentant le motif à remplir.

Si l'entier est supérieur à la longueur de la chaîne d'origine, la nouvelle chaîne sera justifiée à gauche avec le paramètre de chaîne facultatif prenant l'espace restant. Si le paramètre de chaîne n'est pas donné, la chaîne sera remplie d'espaces.

```
str ="abcd"
str.ljust(4)           => "abcd"
str.ljust(10)          => "abcd      "
```

Pour justifier correctement une chaîne, utilisez la méthode `rjust` . Cela prend deux paramètres, un entier représentant le nombre de caractères de la nouvelle chaîne et une chaîne représentant le motif à remplir.

Si le nombre entier est supérieur à la longueur de la chaîne d'origine, la nouvelle chaîne sera justifiée à droite avec le paramètre de chaîne facultatif prenant l'espace restant. Si le paramètre de chaîne n'est pas donné, la chaîne sera remplie d'espaces.

```
str = "abcd"
str.rjust(4)           => "abcd"
str.rjust(10)          => "      abcd"
```

Pour centrer une chaîne, utilisez la méthode `center` . Cela prend deux paramètres, un entier représentant la largeur de la nouvelle chaîne et une chaîne avec laquelle la chaîne d'origine sera complétée. La chaîne sera alignée sur le centre.

```
str = "abcd"
str.center(4)           => "abcd"
str.center(10)          => "   abcd   "
```

Lire Cordes en ligne: https://riptutorial.com/fr/ruby/topic/834/cordes

Chapitre 15: Création / gestion de gemmes

Examples

Fichiers Gemspec

Chaque gem a un fichier au format `<gem name>.gemspec` qui contient des métadonnées sur la gemme et ses fichiers. Le format d'un gemspec est le suivant:

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

Les champs requis par RubyGems sont:

Author `author = string` ou `authors = array`

Utilisez `author =` s'il n'y a qu'un seul auteur et `authors =` lorsqu'il y en a plusieurs. Pour les `authors=` utilisez un tableau qui énumère les noms des auteurs.

```
files = array
```

Ici, `array` est une liste de tous les fichiers du joyau. Cela peut également être utilisé avec la fonction `Dir[]` , par exemple si tous vos fichiers sont dans le répertoire `/lib/` , alors vous pouvez utiliser `files = Dir["/lib/"]` .

```
name = string
```

Ici, string est juste le nom de votre gemme. Rubygems recommande quelques règles à suivre pour nommer votre gem.

1. Utilisez des traits de soulignement, AUCUN ESPACE
2. Utilisez uniquement des lettres minuscules
3. Utilisez hypens pour l'extension gem (par exemple, si votre gem est nommé `example` pour une extension, vous le nommeriez `example-extension`) de sorte que lorsque l'extension est requise, elle peut être requise comme `require "example/extension"` .

RubyGems ajoute également: «Si vous publiez un joyau sur rubygems.org, il peut être supprimé si le nom est répréhensible, viole la propriété intellectuelle ou si le contenu de la pierre satisfait à ces critères. Vous pouvez signaler un tel joyau sur le site de support RubyGems.

```
platform=
```

Je ne sais pas

```
require_paths=
```

Je ne sais pas

```
summary= string
```

String est un condensé du but des gemmes et de tout ce que vous souhaitez partager à propos du joyau.

```
version= string
```

Le numéro de version actuel de la gem

Les champs recommandés sont:

```
email = string
```

Une adresse e-mail qui sera associée à la gem

```
homepage= string
```

Le site où vit la gemme.

Soit `license=` ou `licenses=`

Je ne sais pas

Construire un joyau

Une fois que vous avez créé votre gem pour le publier, vous devez suivre quelques étapes:

1. Construisez votre gem avec `gem build <gem name>.gemspec` (le fichier gemspec doit exister)
2. Créez un compte RubyGems si vous n'en avez pas déjà un ici
3. Vérifiez pour vous assurer qu'il n'existe pas de gemmes qui partagent votre nom de gemmes
4. Publiez votre gem avec `gem publish <gem name>.<gem version number>.gem`

Les dépendances

Pour répertorier l'arbre de dépendance:

```
gem dependency
```

Pour lister les gemmes qui dépendent d'un gem spécifique (bundler par exemple)

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

Lire Création / gestion de gemmes en ligne: https://riptutorial.com/fr/ruby/topic/4092/creation---

gestion-de-gemmes

Chapitre 16: DateTime

Syntaxe

- DateTime.new (année, mois, jour, heure, minute, seconde)

Remarques

Avant d'utiliser DateTime, vous devez `require 'date'`

Examples

DateTime de la chaîne

`DateTime.parse` est une méthode très utile qui construit un DateTime à partir d'une chaîne en devinant son format.

```
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)
```

Remarque: Il existe de nombreux autres formats que `parse` reconnaît.

Nouveau

```
DateTime.new(2014,10,14)
# => #<DateTime: 2014-10-14T00:00:00+00:00 ((2456945j,0s,0n),+0s,2299161j)>
```

Heure actuelle:

```
DateTime.now
# => #<DateTime: 2016-08-04T00:43:58-03:00 ((2457605j,13438s,667386397n),-10800s,2299161j)>
```

Notez qu'il donne l'heure actuelle dans votre fuseau horaire

Ajouter / soustraire des jours à DateTime

`DateTime` + `Fixnum` (quantité en jours)

```
DateTime.new(2015,12,30,23,0) + 1
# => #<DateTime: 2015-12-31T23:00:00+00:00 ((2457388j,82800s,0n),+0s,2299161j)>
```

`DateTime` + `Float` (quantité en jours)

```
DateTime.new(2015,12,30,23,0) + 2.5
# => #<DateTime: 2016-01-02T11:00:00+00:00 ((2457390j,39600s,0n),+0s,2299161j)>
```

`DateTime` + `Rational` (quantité de jours)

```
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` (quantité en jours)

```
DateTime.new(2015,12,30,23,0) - 1
# => #<DateTime: 2015-12-29T23:00:00+00:00 ((2457388j,82800s,0n),+0s,2299161j)>
```

`DateTime` - `Float` (quantité en jours)

```
DateTime.new(2015,12,30,23,0) - 2.5
# => #<DateTime: 2015-12-28T11:00:00+00:00 ((2457385j,39600s,0n),+0s,2299161j)>
```

`DateTime` - `Rational` (nombre de jours)

```
DateTime.new(2015,12,30,23,0) - Rational(1,2)
# => #<DateTime: 2015-12-30T11:00:00+00:00 ((2457387j,39600s,0n),+0s,2299161j)>
```

Lire DateTime en ligne: https://riptutorial.com/fr/ruby/topic/5696/datetime

Chapitre 17: Démarrer avec Hanami

Introduction

Ma mission ici est de contribuer avec la communauté pour aider les nouvelles personnes qui veulent en apprendre davantage sur ce cadre incroyable - Hanami.

Mais comment ça va marcher?

Des didacticiels courts et faciles à suivre illustrant des exemples relatifs à Hanami et suivant les prochains didacticiels, nous verrons comment tester notre application et créer une API REST simple.

Commençons!

Examples

A propos de Hanami

Outre le fait que Hanami soit un cadre léger et rapide, l'un des points qui retient le plus l'attention est le concept d' **architecture propre** , qui nous montre que le cadre n'est pas notre application, comme l'a dit Robert Martin auparavant.

Hanami arquitecture design nous offre l'utilisation de **Container** , dans chaque Container nous avons notre application indépendamment du framework. Cela signifie que nous pouvons récupérer notre code et le placer dans un framework Rails par exemple.

Hanami est un framework MVC?

L'idée des frameworks MVC est de construire une structure en suivant le modèle -> contrôleur -> vue. Hanami suit le modèle | Controller -> View -> Template. Le résultat est une application plus inusitée, suivant les principes **SOLID** et beaucoup plus propre.

- Liens importants.

Hanami http://hanamirb.org/

Robert Martin - Clean Arquitecture https://www.youtube.com/watch?v=WpkDN78P884

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

Principes SOLIDES http://practicingruby.com/articles/solid-design-principles

Comment installer Hanami?

- **Étape 1:** Installation du bijou Hanami.

```
$ gem install hanami
```

- **Étape 2** : générer un nouveau paramètre de projet RSpec en tant que structure de test.

  Ouvrez une ligne de commande ou un terminal. Pour générer une nouvelle application hanami, utilisez hanami new suivi du nom de votre application et du paramètre de test rspec.

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

Obs. Par défaut, Hanami définit Minitest comme structure de test.

Cela créera une application hanami appelée myapp dans un répertoire myapp et installera les dépendances gem déjà mentionnées dans Gemfile en utilisant une installation groupée.

Pour basculer vers ce répertoire, utilisez la commande cd, qui signifie change directory.

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

Le répertoire myapp contient un certain nombre de fichiers et de dossiers générés automatiquement qui constituent la structure d'une application Hanami. Voici une liste des fichiers et dossiers créés par défaut:

- **Gemfile** définit nos dépendances Rubygems (en utilisant Bundler).
- **Rakefile** décrit nos tâches Rake.
- **Les applications** contiennent une ou plusieurs applications Web compatibles avec Rack. Ici, nous pouvons trouver la première application Hanami générée appelée Web. C'est l'endroit où nous trouvons nos contrôleurs, vues, itinéraires et modèles.
- **config** contient des fichiers de configuration.
- **config.ru** est pour les serveurs Rack.
- **La base de** données contient notre schéma de base de données et ses migrations.
- **lib** contient notre logique métier et notre modèle de domaine, y compris les entités et les référentiels.
- **public** contiendra des éléments statiques compilés.
- **spec** contient nos tests.
- **Liens importants**

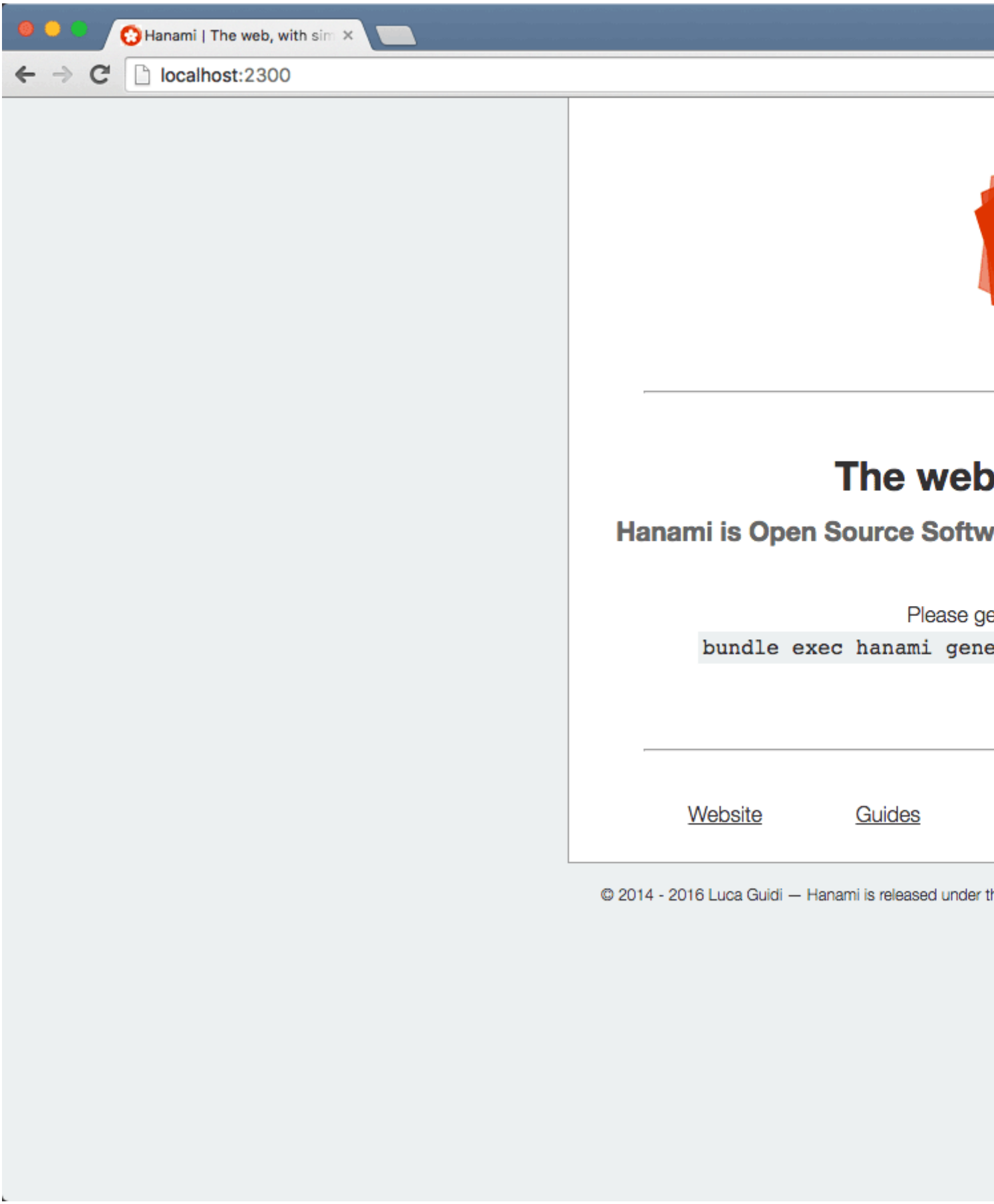
Hanami bijou https://github.com/hanami/hanami

Fonctionnaire de Hanami Mise en route http://hanamirb.org/guides/getting-started/

Comment démarrer le serveur?

- **Étape 1:** Pour démarrer le serveur, tapez simplement la commande ci-dessous, puis vous verrez la page de démarrage.

```
$ bundle exec hanami server
```



Lire Démarrer avec Hanami en ligne: https://riptutorial.com/fr/ruby/topic/9676/demarrer-avec-hanami

Chapitre 18: Des classes

Syntaxe

- nom du cours
- #some code décrivant le comportement de la classe
- fin

Remarques

Les noms de classe dans Ruby sont des constantes, donc la première lettre devrait être une majuscule.

```
class Cat # correct
end

class dog # wrong, throws an error
end
```

Examples

Créer une classe

Vous pouvez définir une nouvelle classe en utilisant le mot `class` clé `class` .

```
class MyClass
end
```

Une fois défini, vous pouvez créer une nouvelle instance en utilisant la méthode `.new`

```
somevar = MyClass.new
# => #<MyClass:0x007fe2b8aa4a18>
```

Constructeur

Une classe ne peut avoir qu'un seul constructeur, c'est-à-dire une méthode appelée `initialize` .
La méthode est automatiquement appelée lorsqu'une nouvelle instance de la classe est créée.

```
class Customer
  def initialize(name)
     @name = name.capitalize
  end
end

sarah = Customer.new('sarah')
sarah.name #=> 'Sarah'
```

Variables de classe et d'instance

Il existe plusieurs types de variables spécifiques qu'une classe peut utiliser pour partager plus facilement des données.

Variables d'instance, précédées de `@` . Ils sont utiles si vous souhaitez utiliser la même variable dans différentes méthodes.

```
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 classe, précédée de `@@` . Ils contiennent les mêmes valeurs sur toutes les instances d'une classe.

```
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, précédées de `$` . Ceux-ci sont disponibles partout dans le programme, alors assurez-vous de les utiliser à bon escient.

```
$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
```

Accès aux variables d'instance avec les getters et les setters

Nous avons trois méthodes:

1. **`attr_reader`** : permet de `read` la variable en dehors de la classe.
2. **`attr_writer`** : permet de modifier la variable en dehors de la classe.
3. **`attr_accessor`** : combine les deux méthodes.

```
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
```

Notez que les paramètres sont des symboles. Cela fonctionne en créant une méthode.

```
class Cat
  attr_accessor :breed
end
```

Est fondamentalement la même chose que:

```
class Cat
  def breed
    @breed
  end
  def breed= value
    @breed = value
  end
end
```

Niveaux d'accès

Ruby a trois niveaux d'accès. Ils sont `public` , `private` et `protected` .

Les méthodes qui suivent les mots-clés `private` ou `protected` sont définies comme telles. Les méthodes qui précèdent sont implicitement `public` méthodes `public` .

Méthodes publiques

Une méthode publique doit décrire le comportement de l'objet en cours de création. Ces méthodes peuvent être appelées en dehors de la portée de l'objet créé.

```
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
```

Ces méthodes sont des méthodes publiques ruby, elles décrivent le comportement d’initialisation d’un nouveau chat et le comportement de la méthode speak.

mot-clé `public` est inutile, mais peut être utilisé pour échapper `private` ou `protected`

```
def MyClass
  def first_public_method
  end

  private

  def private_method
  end

  public

  def second_public_method
  end
end
```

Méthodes Privées

Les méthodes privées ne sont pas accessibles de l'extérieur de l'objet. Ils sont utilisés en interne par l'objet. En utilisant à nouveau l'exemple de chat:

```
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">
```

Comme vous pouvez le voir dans l'exemple ci-dessus, l'objet Cat nouvellement créé a accès à la méthode `calculate_cat_age` interne. Nous affectons l' `age` variable au résultat de l'exécution de la méthode `calculate_cat_age` privée qui imprime le nom et l'âge du chat sur la console.

Lorsque nous essayons d'appeler la méthode `calculate_cat_age` dehors de l'objet `my_cat` , nous recevons une `NoMethodError` car elle est privée. Trouver?

Méthodes protégées

Les méthodes protégées sont très similaires aux méthodes privées. On ne peut pas y accéder en dehors de l'instance d'objet de la même manière que les méthodes privées ne peuvent pas l'être. Cependant, en utilisant la méthode `self` ruby, les méthodes protégées peuvent être appelées

dans le contexte d'un objet du même type.

```
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>
```

Vous pouvez voir que nous avons ajouté un paramètre age à la classe cat et créé trois nouveaux objets chat avec le nom et l'âge. Nous allons appeler la méthode `own_age` protected pour comparer l'âge de nos objets de chat.

```
cat1 == cat2
=> false

cat1 == cat3
=> true
```

Regardez cela, nous avons pu récupérer l'âge de cat1 en utilisant la méthode protégée `self.own_age` et la comparer à l'âge de `cat2.own_age` appelant `cat2.own_age` intérieur de cat1.

Types de méthodes de classe

Les classes ont 3 types de méthodes: les méthodes instance, singleton et class.

Méthodes d'instance

Ce sont des méthodes qui peuvent être appelées à partir d'une `instance` de la classe.

```
class Thing
  def somemethod
    puts "something"
  end
end

foo = Thing.new # create an instance of the class
foo.somemethod # => something
```

Méthode de classe

Ce sont des méthodes statiques, c'est-à-dire qu'elles peuvent être invoquées sur la classe et non sur une instanciation de cette classe.

```
class Thing
  def Thing.hello(name)
    puts "Hello, #{name}!"
  end
end
```

Cela équivaut à utiliser `self` à la place du nom de la classe. Le code suivant est équivalent au code ci-dessus:

```
class Thing
  def self.hello(name)
    puts "Hello, #{name}!"
  end
end
```

Invoquer la méthode en écrivant

```
Thing.hello("John Doe")  # prints: "Hello, John Doe!"
```

Méthodes Singleton

Ceux-ci ne sont disponibles que pour des instances spécifiques de la classe, mais pas pour tous.

```
# 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>
```

Les méthodes `singleton` et `class` sont toutes deux appelées `eigenclass` . Fondamentalement, Ruby crée une classe anonyme qui contient de telles méthodes afin de ne pas interférer avec les instances créées.

Une autre façon de faire est d' `class <<` constructeur de `class <<` . Par exemple:

```
# 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"
```

Création de classe dynamique

Les classes peuvent être créées dynamiquement via l'utilisation de `Class.new` .

```
# create a new class dynamically
MyClass = Class.new

# instantiate an object of type MyClass
my_class = MyClass.new
```

Dans l'exemple ci-dessus, une nouvelle classe est créée et affectée à la constante `MyClass` . Cette classe peut être instanciée et utilisée comme toute autre classe.

La méthode `Class.new` accepte une `Class` qui deviendra la super-classe de la classe créée dynamiquement.

```
# 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
```

La méthode `Class.new` accepte également un bloc. Le contexte du bloc est la classe nouvellement créée. Cela permet de définir des méthodes.

```
Duck =
  Class.new do
    def quack
      'Quack!!'
    end
  end

# instantiate an object of type Duck
duck = Duck.new
duck.quack # 'Quack!!'
```

Nouveau, allouer et initialiser

Dans de nombreux langages, de nouvelles instances d'une classe sont créées à l'aide d'un `new` mot-clé spécial. Dans Ruby, `new` est également utilisé pour créer des instances d'une classe, mais ce n'est pas un mot clé; au lieu de cela, c'est une méthode statique / classe, pas différente de toute autre méthode statique / classe. La définition est à peu près ceci:

```
class MyClass
   def self.new(*args)
     obj = allocate
     obj.initialize(*args) # oversimplied; initialize is actually private
     obj
   end
end
```

`allocate` effectue la véritable «magie» de la création d'une instance non initialisée de la classe

Notez également que la valeur de retour de `initialize` est ignorée et que obj est renvoyé à la place. Cela rend immédiatement clair pourquoi vous pouvez coder votre méthode initialize sans se soucier de retourner l' `self` à la fin.

La `new` méthode «normale» que toutes les classes obtiennent de `Class` fonctionne comme ci-dessus, mais il est possible de la redéfinir comme vous le souhaitez ou de définir des alternatives qui fonctionnent différemment. Par exemple:

```
class MyClass
  def self.extraNew(*args)
    obj = allocate
    obj.pre_initialize(:foo)
    obj.initialize(*args)
    obj.post_initialize(:bar)
    obj
  end
end
```

Lire Des classes en ligne: https://riptutorial.com/fr/ruby/topic/264/des-classes

Chapitre 19: Des exceptions

Remarques

Une *exception* est un objet qui représente l'occurrence d'une condition exceptionnelle. En d'autres termes, cela indique que quelque chose s'est mal passé.

Dans Ruby, les *exceptions* sont souvent appelées *erreurs* . En effet, la classe `Exception` base existe en tant qu'élément d'objet d'exception de niveau supérieur, mais les exceptions d'exécution définies par l'utilisateur sont généralement `StandardError` ou des descendants.

Examples

Lever une exception

Pour générer une exception, utilisez le `Kernel#raise` passant la classe et / ou le message d'exception:

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

Vous pouvez également simplement transmettre un message d'erreur. Dans ce cas, le message est enveloppé dans une `RuntimeError` :

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

Voici un exemple:

```
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"
```

Création d'un type d'exception personnalisé

Une exception personnalisée est toute classe qui étend `Exception` ou une sous-classe d' `Exception` .

En général, vous devez toujours étendre `StandardError` ou un descendant. La famille des `Exception` concerne généralement les erreurs de machine virtuelle ou de système, les récupérer peut empêcher une interruption forcée de fonctionner comme prévu.

```
# 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
```

Il est courant de nommer des exceptions en ajoutant le suffixe d' `Error` à la fin:

- `ConnectionError`
- `DontPanicError`

Toutefois, lorsque l'erreur est explicite, vous n'avez pas besoin d'ajouter le suffixe d' `Error` car il serait redondant:

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

Gérer une exception

Utilisez le bloc `begin/rescue` pour intercepter (sauver) une exception et la gérer:

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

Une clause de `rescue` est analogue à un bloc `catch` dans une accolade comme C # ou Java.

Un `rescue` brutal comme celui-ci sauve `StandardError` .

Remarque: veillez à ne pas intercepter `Exception` place de `StandardError` par défaut. La classe `Exception` inclut `SystemExit` et `NoMemoryError` et d'autres exceptions graves que vous ne souhaitez généralement pas intercepter. Toujours envisager d'attraper `StandardError` (valeur par défaut) à la place.

Vous pouvez également spécifier la classe d'exception à récupérer:

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

Cette clause de secours n'acceptera aucune exception qui n'est pas une `CustomError` .

Vous pouvez également stocker l'exception dans une variable spécifique:

```
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 vous ne parvenez pas à gérer une exception, vous pouvez l'augmenter à tout moment dans un bloc de secours.

```
begin
   #here goes your code
rescue => e
    #failed to handle
    raise e
end
```

Si vous voulez retenter votre `begin` bloc, appelez `retry` :

```
begin
   #here goes your code
rescue StandardError => e
   #for some reason you want to retry you code
   retry
end
```

Vous pouvez être bloqué dans une boucle si vous attrapez une exception dans chaque tentative. Pour éviter cela, limitez votre `retry_count` de `retry_count` à un certain nombre d'essais.

```
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
```

Vous pouvez également fournir un `else` bloc ou un bloc d' `ensure` . Un bloc `else` sera exécuté lorsque le bloc de `begin` termine sans qu'une exception soit lancée. Un bloc d' `ensure` sera toujours exécuté. Un bloc d' `ensure` est analogue à un bloc `finally` dans une accolade comme C # ou 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 vous vous trouvez dans un bloc `def` , `module` ou `class` , il n'est pas nécessaire d'utiliser l'instruction begin.

```
def foo
    ...
rescue
    ...
end
```

Gestion de plusieurs exceptions

Vous pouvez gérer plusieurs erreurs dans la même déclaration de `rescue` :

```
begin
  # an execution that may fail
rescue FirstError, SecondError => e
  # do something if a FirstError or SecondError occurs
end
```

Vous pouvez également ajouter plusieurs déclarations 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
```

L'ordre des blocs de `rescue` est pertinent: le premier match est celui exécuté. Par conséquent, si vous placez `StandardError` comme première condition et que toutes vos exceptions héritent de `StandardError` , les autres instructions de `rescue` ne seront jamais exécutées.

```
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
```

Certains blocs ont une gestion implicite des exceptions comme `def` , `class` et `module` . Ces blocs vous permettent d'ignorer l'instruction `begin` .

```
def foo
    ...
rescue CustomError
    ...
ensure
```

```
    ...
end
```

Ajout d'informations à des exceptions (personnalisées)

Il peut être utile d'inclure des informations supplémentaires avec une exception, par exemple à des fins de journalisation ou pour permettre une gestion conditionnelle lorsque l'exception est interceptée:

```
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
```

En soulevant l'exception:

```
raise CustomError.new(true)
```

Récupérer l'exception et accéder aux informations supplémentaires fournies:

```
begin
  # do stuff
rescue CustomError => e
  retry if e.safe_to_retry
end
```

Lire Des exceptions en ligne: https://riptutorial.com/fr/ruby/topic/940/des-exceptions

Chapitre 20: Des symboles

Syntaxe

- :symbole
- :'symbole'
- :"symbole"
- "symbole" .to_sym
- % s {symbole}

Remarques

Avantages de l'utilisation de symboles sur des chaînes:

1. Un symbole Ruby est un objet avec une comparaison O (1)

Pour comparer deux chaînes, nous devons potentiellement examiner chaque caractère. Pour deux chaînes de longueur N, cela nécessitera des comparaisons N + 1

```
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"
```

Mais puisque chaque apparition de: foobar fait référence au même objet, nous pouvons comparer des symboles en examinant les ID d'objet. Nous pouvons le faire avec une seule comparaison. (O (1))

```
def symbol_compare sym1, sym2
  sym1.object_id == sym2.object_id
end
symbol_compare :foobar, :foobar
```

2. Un symbole Ruby est une étiquette dans une énumération de forme libre

En C ++, nous pouvons utiliser des «énumérations» pour représenter des familles de constantes associées:

```
enum BugStatus { OPEN, CLOSED };
BugStatus original_status = OPEN;
BugStatus current_status  = CLOSED;
```

Mais comme Ruby est un langage dynamique, nous ne nous soucions pas de la déclaration d'un type BugStatus ou du suivi des valeurs légales. Au lieu de cela, nous représentons les valeurs d'énumération sous forme de symboles:

```
original_status = :open
current_status  = :closed
```

3. Un symbole Ruby est un nom unique et constant

En Ruby, on peut changer le contenu d'une chaîne:

```
"foobar"[0] = ?b # "boo"
```

Mais nous ne pouvons pas changer le contenu d'un symbole:

```
:foobar[0]  = ?b # Raises an error
```

4. Un symbole Ruby est le mot-clé d'un argument de mot-clé

Lorsque vous transmettez des arguments à une fonction Ruby, nous spécifions les mots-clés à l'aide de symboles:

```
# Build a URL for 'bug' using Rails.
url_for :controller => 'bug',
        :action => 'show',
        :id => bug.id
```

5. Un symbole Ruby est un excellent choix pour une clé de hachage

En règle générale, nous utiliserons des symboles pour représenter les clés d'une table de hachage:

```
options = {}
options[:auto_save]     = true
options[:show_comments] = false
```

Examples

Créer un symbole

La manière la plus courante de créer un objet `Symbol` est de préfixer l'identificateur de chaîne avec un signe deux-points:

```
:a_symbol       # => :a_symbol
:a_symbol.class # => Symbol
```

Voici quelques moyens de définir un `Symbol` , en combinaison avec un littéral `String` :

```
:"a_symbol"
"a_symbol".to_sym
```

Les symboles ont également une séquence `%s` qui prend en charge des délimiteurs arbitraires similaires à la façon dont `%q` et `%Q` fonctionnent pour les chaînes:

```
%s(a_symbol)
%s{a_symbol}
```

Le `%s` est particulièrement utile pour créer un symbole à partir d'une entrée contenant des espaces blancs:

```
%s{a symbol} # => :"a symbol"
```

Bien que certains symboles intéressants (`:/` `:[]` `:^` , etc.) puissent être créés avec certains identificateurs de chaîne, notez que les symboles ne peuvent pas être créés avec un identificateur numérique:

```
:1 # => syntax error, unexpected tINTEGER, ...
:0.3 # => syntax error, unexpected tFLOAT, ...
```

Les symboles peuvent se terminer par un seul `?` ou `!` sans avoir besoin d'utiliser un littéral de chaîne comme identifiant du symbole:

```
:hello?  # :"hello?" is not necessary.
:world!  # :"world!" is not necessary.
```

Notez que toutes ces différentes méthodes de création de symboles renverront le même objet:

```
:symbol.object_id == "symbol".to_sym.object_id
:symbol.object_id == %s{symbol}.object_id
```

Depuis Ruby 2.0, il existe un raccourci pour créer un tableau de symboles à partir de mots:

```
%i(numerator denominator) == [:numerator, :denominator]
```

Conversion d'une chaîne en symbole

Donné une `String` :

```
s = "something"
```

il existe plusieurs façons de le convertir en `Symbol` :

```
s.to_sym
# => :something
:"#{s}"
# => :something
```

Conversion d'un symbole en chaîne

Donné un `Symbol` :

```
s = :something
```

Le moyen le plus simple de le convertir en `String` est d'utiliser la méthode `Symbol#to_s` :

```
s.to_s
# => "something"
```

Une autre façon de procéder consiste à utiliser la méthode `Symbol#id2name` , alias de la méthode `Symbol#to_s` . Mais c'est une méthode unique à la classe `Symbol` :

```
s.id2name
# => "something"
```

Lire Des symboles en ligne: https://riptutorial.com/fr/ruby/topic/873/des-symboles

Chapitre 21: Design Patterns and Idioms in Ruby

Examples

Singleton

Ruby Standard Library possède un module Singleton qui implémente le modèle Singleton. La première étape de la création d'une classe Singleton consiste à exiger et à inclure le module **`Singleton`** dans une classe:

```
require 'singleton'

class Logger
  include Singleton
end
```

Si vous essayez d'instancier cette classe comme vous le feriez normalement avec une classe normale, une exception `NoMethodError` est `NoMethodError` . Le constructeur est rendu privé pour empêcher que d'autres instances soient créées accidentellement:

```
Logger.new

#=> NoMethodError: private method `new' called for AppConfig:Class
```

Pour accéder à l'instance de cette classe, nous devons utiliser l' `instance()` :

```
first, second = Logger.instance, Logger.instance
first == second

#=> true
```

Exemple de Logger

```
require 'singleton'


class Logger
  include Singleton

  def initialize
    @log = File.open("log.txt", "a")
  end

  def log(msg)
    @log.puts(msg)
  end
end
```

Pour utiliser l'objet `Logger` :

```
Logger.instance.log('message 2')
```

Sans Singleton inclure

Les implémentations singleton ci-dessus peuvent également être effectuées sans l'inclusion du module Singleton. Cela peut être réalisé avec les éléments suivants:

```
class Logger
  def self.instance
    @instance ||= new
  end
end
```

qui est une notation abrégée pour les éléments suivants:

```
class Logger
  def self.instance
    @instance = @instance || Logger.new
  end
end
```

Cependant, gardez à l'esprit que le module Singleton est testé et optimisé, ce qui en fait la meilleure option pour implémenter votre singleton.

Observateur

Le modèle d'observateur est un modèle de conception logicielle dans lequel un objet (appelé `subject`) conserve une liste de ses dépendants (appelés `observers`) et les avertit automatiquement de tout changement d'état, généralement en appelant l'une de ses méthodes.

Ruby fournit un mécanisme simple pour implémenter le modèle de conception Observer. Le module `Observable` fournit la logique pour informer l'abonné de tout changement dans l'objet Observable.

Pour que cela fonctionne, l'observable doit affirmer qu'il a changé et en informer les observateurs.

Les objets observant doivent implémenter une méthode `update()` , qui sera le rappel de l'observateur.

Implémentons une petite discussion, où les utilisateurs peuvent s'abonner aux utilisateurs et quand l'un d'eux écrit quelque chose, les abonnés sont avertis.

```
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
```

Produire la sortie suivante:

```
# Computer says: No
# Computer says: No
```

Nous avons déclenché la méthode `write` à la classe Moderator deux fois, en avertissant ses abonnés, dans ce cas-ci un seul.

Plus nous ajoutons d'abonnés, plus les changements se propageront.

Motif Décorateur

Le motif de décorateur ajoute un comportement aux objets sans affecter les autres objets de la même classe. Le motif de décorateur est une alternative utile à la création de sous-classes.

Créez un module pour chaque décorateur. Cette approche est plus flexible que l'héritage, car vous pouvez combiner et associer des responsabilités dans davantage de combinaisons. De plus, la transparence permet aux décorateurs d’être imbriqués récursivement, ce qui permet un nombre illimité de responsabilités.

Supposons que la classe Pizza a une méthode de coût qui renvoie 300:

```
class Pizza
  def cost
    300
  end
end
```

Représenter la pizza avec une couche supplémentaire de fromage et le coût augmente de 50. L'approche la plus simple consiste à créer une sous-classe `PizzaWithCheese` qui renvoie 350 dans la méthode du coût.

```
class PizzaWithCheese < Pizza
  def cost
    350
  end
end
```

Ensuite, nous devons représenter une grande pizza qui ajoute 100 au coût d'une pizza normale. Nous pouvons représenter cela en utilisant une sous-classe LargePizza de Pizza.

```
class LargePizza < Pizza
  def cost
    400
  end
end
```

Nous pourrions également avoir un ExtraLargePizza qui ajoute un coût supplémentaire de 15 à notre LargePizza. Si nous considérions que ces types de pizza pouvaient être servis avec du fromage, nous devions ajouter les sous-classes LargePizzaWithChese et ExtraLargePizzaWithCheese.

Pour simplifier l'approche, utilisez des modules pour ajouter dynamiquement un comportement à la classe Pizza:

Module + extend + super décorateur: ->

```
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
```

Procuration

L'objet proxy est souvent utilisé pour garantir un accès sécurisé à un autre objet, quelle logique métier interne nous ne voulons pas polluer avec les exigences de sécurité.

Supposons que nous souhaitons garantir que seul l'utilisateur des autorisations spécifiques puisse accéder aux ressources.

Définition du proxy: (il garantit que seuls les utilisateurs capables de voir les réservations pourront utiliser le service reservation_service)

```
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
```

Modèles et service de réservation:

```
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
```

Service client:

```
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
```

Tester:

```
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)
```

AVANTAGES

- **nous évitons tout changement dans `ReservationService` lorsque les restrictions d'accès sont modifiées.**
- **Nous ne `date_from` pas les données liées à l'entreprise (`date_from` , `date_to` , `reservations_count`) avec les concepts non liés au domaine (autorisations utilisateur) en service.**

- **Consumer (`StatsService`) est également exempt de logique liée aux autorisations**

CAVEATS

- L'interface proxy est toujours exactement identique à l'objet qu'elle cache, de sorte que l'utilisateur qui consomme le service encapsulé par un proxy n'était même pas au courant de la présence de proxy.

Lire Design Patterns and Idioms in Ruby en ligne: https://riptutorial.com/fr/ruby/topic/2081/design-patterns-and-idioms-in-ruby

Chapitre 22: Enumerable en Ruby

Introduction

Module enumberable, un ensemble de méthodes est disponible pour faire le parcours, le tri, la recherche, etc. dans la collection (Array, Hashes, Set, HashMap).

Examples

Module énumérable

```
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.
```

Résumé: Le meilleur moyen de transformer la collection consiste à utiliser le module Enumerable pour compacter le code maladroit.

Lire Enumerable en Ruby en ligne: https://riptutorial.com/fr/ruby/topic/10786/enumerable-en-ruby

Chapitre 23: Énumérateurs

Introduction

Un `Enumerator` est un objet qui implémente une itération de manière contrôlée.

Au lieu de boucler jusqu'à ce qu'une condition soit satisfaite, l'objet *énumère les* valeurs nécessaires. L'exécution de la boucle est suspendue jusqu'à ce que le propriétaire de l'objet demande la valeur suivante.

Les énumérateurs permettent des flux de valeurs infinis.

Paramètres

Paramètre	Détails
`yield`	Répond au `yield` , qui est aliasé comme `<<` . Céder à cet objet implémente une itération.

Examples

Enumérateurs personnalisés

Créons un `Enumerator` pour les nombres de Fibonacci .

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

Nous pouvons maintenant utiliser n'importe `Enumerable` méthode `Enumerable` avec `fibonacci` :

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

Méthodes existantes

Si une méthode d'itération telle que `each` est appelée sans bloc, un `Enumerator` doit être renvoyé.

Cela peut être fait en utilisant la méthode `enum_for` :

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

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

Cela permet au programmeur de composer des opérations `Enumerable` :

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

Rembobinage

Utilisez `rewind` pour redémarrer l'énumérateur.

```
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
```

Lire Énumérateurs en ligne: https://riptutorial.com/fr/ruby/topic/4985/enumerateurs

Chapitre 24: ERB

Introduction

ERB (Embedded Ruby) est utilisé pour insérer des variables Ruby dans des modèles, par exemple HTML et YAML. ERB est une classe Ruby qui accepte du texte et évalue et remplace le code Ruby par un balisage ERB.

Syntaxe

- <% number = rand (10)%> ce code sera évalué
- <% = number%> ce code sera évalué et inséré dans la sortie
- <% # commentaire texte%> ce commentaire ne sera pas évalué

Remarques

Conventions:

- ERB en tant que modèle: Abstenez-vous de la logique métier en code d'accompagnement, et gardez vos modèles ERB propres et lisibles pour les personnes sans connaissances Ruby.
- Ajouter des fichiers avec `.erb` : par exemple `.js.erb` , `.html.erb` , `.css.erb` , etc.

Examples

Analyse ERB

Cet exemple est un texte filtré d'une session `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>
```

Lire ERB en ligne: https://riptutorial.com/fr/ruby/topic/8145/erb

Chapitre 25: Évaluation dynamique

Syntaxe

- eval "source"
- eval "source", liaison
- eval "source", proc
- binding.eval "source" # égal à `eval "source", binding`

Paramètres

Paramètre	Détails
`"source"`	Tout code source Ruby
`binding`	Une instance de classe `Binding`
`proc`	Une instance de classe `Proc`

Examples

Évaluation d'instance

La méthode `instance_eval` est disponible sur tous les objets. Il évalue le code dans le contexte du récepteur:

```
object = Object.new

object.instance_eval do
  @variable = :value
end

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

`instance_eval` définit lui- `self` `object` pour la durée du bloc de code:

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

Le récepteur est également transmis au bloc comme seul argument:

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

La méthode `instance_exec` diffère à cet égard: elle transmet ses arguments au bloc à la place.

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

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

Evaluer une chaîne

Toute `String` peut être évaluée au moment de l'exécution.

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

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

Evaluer à l'intérieur d'une liaison

Ruby conserve la trace des variables locales et `self` variables `self` via un objet appelé binding. Nous pouvons obtenir une liaison avec une portée en appelant la `Kernel#binding` et évaluer la chaîne à l'intérieur d'une liaison via `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
```

Création dynamique de méthodes à partir de chaînes

Ruby propose `define_method` comme méthode privée sur les modules et les classes pour la définition de nouvelles méthodes d'instance. Cependant, le «corps» de la méthode doit être un `Proc` ou une autre méthode existante.

Une façon de créer une méthode à partir de données de chaîne brutes consiste à utiliser `eval` pour créer un Proc à partir du code:

```
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
```

Lire Évaluation dynamique en ligne: https://riptutorial.com/fr/ruby/topic/5048/evaluation-dynamique

Chapitre 26: Expressions régulières et opérations basées sur les regex

Examples

Groupes, nommés et autres.

Ruby étend la syntaxe de groupe standard `(...)` avec un groupe nommé `(?<name>...)` . Cela permet l'extraction par nom au lieu d'avoir à compter le nombre de groupes que vous avez.

```
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
```

L'index de la correspondance est compté en fonction de l'ordre des parenthèses de gauche (la totalité de l'expression rationnelle étant le premier groupe à l'index 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"
```

= ~ opérateur

```
if /hay/ =~ 'haystack'
  puts "There is hay in the word haystack"
end
```

Note: La commande **est importante** . Bien que la `'haystack' =~ /hay/` soit dans la plupart des cas équivalente, les effets secondaires peuvent différer:

- Les chaînes capturées à partir de groupes de capture nommés ne sont affectées à des variables locales que lorsque `Regexp#=~` est appelée (`regexp =~ str`);
- Le bon opérande pouvant être un objet arbitraire, pour `regexp =~ str` il sera appelé soit `Regexp#=~` soit `String#=~` .

Notez que cela ne renvoie pas une valeur true / false, mais renvoie soit l'index de la correspondance s'il est trouvé, soit nul s'il n'est pas trouvé. Parce que tous les entiers en ruby sont vrais (y compris 0) et nil est faux, cela fonctionne. Si vous voulez une valeur booléenne, utilisez `#===` comme indiqué dans un autre exemple .

Quantificateurs

Quantifiers permet de spécifier le nombre de chaînes répétées.

- Zéro ou un:

```
/a?/
```

- Zéro ou beaucoup:

```
/a*/
```

- Un ou plusieurs:

```
/a+/
```

- Nombre exact:

```
/a{2,4}/ # Two, three or four
/a{2,}/  # Two or more
/a{,4}/  # Less than four (including zero)
```

Par défaut, les quantificateurs sont gourmands , ce qui signifie qu'ils prennent autant de caractères qu'ils le peuvent tout en effectuant une correspondance. Normalement, cela ne se remarque pas:

```
/(?<site>.*) Stack Exchange/ =~ 'Motor Vehicle Maintenance & Repair Stack Exchange'
```

Le `site` groupe de capture nommé sera défini sur "Maintenance et réparation des véhicules automobiles" comme prévu. Mais si «Stack Exchange» est une partie facultative de la chaîne (car il pourrait s'agir de «Stack Overflow» à la place), la solution naïve ne fonctionnera pas comme prévu:

```
/(?<site>.*)( Stack Exchange)?/
```

Cette version correspondra toujours, mais la capture nommée inclura "Stack Exchange" puisque `*` mange avidement ces caractères. La solution consiste à ajouter un autre point d'interrogation pour

rendre le `*` paresseux:

```
/(?<site>.*?)( Stack Exchange)?/
```

Ajout `?` à tout quantificateur le rendra paresseux.

Classes de caractères

Décrit les plages de symboles

Vous pouvez énumérer explicitement les symboles

```
/[abc]/ # 'a' or 'b' or 'c'
```

Ou utiliser des gammes

```
/[a-z]/ # from 'a' to 'z'
```

Il est possible de combiner des plages et des symboles simples

```
/[a-cz]/ # 'a' or 'b' or 'c' or 'z'
```

Le trait d'union (`-`) est traité comme charachter

```
/[-a-c]/ # '-' or 'a' or 'b' or 'c'
```

Les classes peuvent être négatives lorsque des symboles précèdent avec `^`

```
/[^a-c]/ # Not 'a', 'b' or 'c'
```

Il existe des raccourcis pour les classes répandues et les charachters spéciaux, ainsi que des fins de ligne

```
^  # 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` se comprendra simplement comme nouvelle ligne

Pour échapper à n'importe quel charachter réservé, tel que `/` ou `[]` et d'autres, utilisez une barre

oblique inverse (barre oblique à gauche)

```
\\ # => \
\[\] # => []
```

Expressions régulières dans les déclarations de cas

Vous pouvez tester si une chaîne correspond à plusieurs expressions régulières à l'aide d'une instruction switch.

Exemple

```
case "Ruby is #1!"
when /\APython/
    puts "Boooo."
when /\ARuby/
    puts "You are right."
else
    puts "Sorry, I didn't understand that."
end
```

Cela fonctionne parce que les instructions de cas sont vérifiées pour l'égalité en utilisant l'opérateur `===` , pas l'opérateur `==` . Quand une regex est sur le côté gauche d'une comparaison utilisant `===` , elle testera une chaîne pour voir si elle correspond.

Définir une expression rationnelle

Une expression régulière peut être créée de trois manières différentes dans Ruby.

- en utilisant des barres obliques: `/ /`
- en utilisant `%r{}`
- en utilisant `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
```

rencontre? - Résultat booléen

Retourne `true` ou `false` , qui indique si l'expression rationnelle est mise en correspondance ou non sans mettre `$~` jour `$~` et d'autres variables associées. Si le second paramètre est présent, il

spécifie la position dans la chaîne pour commencer la recherche.

```
/R.../.match?("Ruby")    #=> true
/R.../.match?("Ruby", 1) #=> false
/P.../.match?("Ruby")    #=> false
```

Ruby 2.4+

Utilisation rapide commune

Les expressions régulières sont souvent utilisées dans les méthodes en tant que paramètres pour vérifier si d'autres chaînes sont présentes ou pour rechercher et / ou remplacer des chaînes.

Vous verrez souvent ce qui suit:

```
string = "My not so long string"
string[/so/] # gives so
string[/present/] # gives nil
string[/present/].nil? # gives true
```

Donc, vous pouvez simplement l'utiliser comme une vérification si une chaîne contient une sous-chaîne

```
puts "found" if string[/so/]
```

Plus avancé mais toujours court et rapide: recherchez un groupe spécifique en utilisant le deuxième paramètre, 2 est le second dans cet exemple car la numérotation commence à 1 et non à 0, un groupe est ce qui est entre parenthèses.

```
string[/(n.t).+(l.ng)/, 2] # gives long
```

Aussi souvent utilisé: rechercher et remplacer par `sub` ou `gsub` , `\1` donne le premier groupe trouvé, `\2` le second:

```
string.gsub(/(n.t).+(l.ng)/, '\1 very \2') # My not very long string
```

Le dernier résultat est mémorisé et peut être utilisé sur les lignes suivantes

```
$2 # gives long
```

Lire Expressions régulières et opérations basées sur les regex en ligne: https://riptutorial.com/fr/ruby/topic/1357/expressions-regulieres-et-operations-basees-sur-les-regex

Chapitre 27: Fil

Examples

Sémantique de fil de base

Un nouveau thread séparé de l'exécution du thread principal peut être créé à l'aide de `Thread.new` .

```
thr = Thread.new {
  sleep 1 # 1 second sleep of sub thread
  puts "Whats the big deal"
}
```

Cela démarrera automatiquement l'exécution du nouveau thread.

Pour geler l'exécution du thread principal, jusqu'à ce que le nouveau thread s'arrête, utilisez `join` :

```
thr.join #=> ... "Whats the big deal"
```

Notez que le thread peut déjà être terminé lorsque vous appelez join, auquel cas l'exécution se poursuivra normalement. Si un sous-thread n'est jamais joint et que le thread principal se termine, le sous-thread n'exécutera aucun code restant.

Accéder aux ressources partagées

Utilisez un mutex pour synchroniser l'accès à une variable accessible à partir de plusieurs threads:

```
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
```

Sinon, la valeur du `counter` actuellement visible pour un thread peut être modifiée par un autre thread.

Exemple **sans** `Mutex` (voir par exemple `Thread 0` , où `Before` et `After` diffèrent de plus 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
```

Exemple **avec** `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
```

Comment tuer un fil

Vous appelez utiliser `Thread.kill` ou `Thread.terminate` :

```
thr = Thread.new { ... }
Thread.kill(thr)
```

Terminer un fil

Un thread se termine s'il atteint la fin de son bloc de code. La meilleure façon de terminer un thread tôt est de le convaincre d'atteindre la fin de son bloc de code. De cette façon, le thread peut exécuter du code de nettoyage avant de mourir.

Ce thread exécute une boucle alors que la variable d'instance continue est vraie. Définissez cette variable sur false et le thread mourra d'une mort naturelle:

```
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
```

Lire Fil en ligne: https://riptutorial.com/fr/ruby/topic/995/fil

Chapitre 28: Flux de contrôle

Examples

si, elsif, sinon et fin

Ruby propose les expressions `if` et `else` attendues pour la logique de branchement, terminées par le mot clé `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
```

Dans Ruby, les instructions `if` sont des expressions dont l'évaluation donne une valeur et le résultat peut être affecté à une variable:

```
status = if age < 18
           :minor
         else
           :adult
         end
```

Ruby propose également des opérateurs ternaires de style C (voir ici pour plus de détails) pouvant être exprimés comme suit:

```
some_statement ? if_true : if_false
```

Cela signifie que l'exemple ci-dessus utilisant if-else peut aussi être écrit comme

```
status = age < 18 ? :minor : :adult
```

En outre, Ruby propose le mot-clé `elsif` qui accepte une expression pour activer une logique de branchement supplémentaire:

```
label = if shirt_size == :s
          'small'
        elsif shirt_size == :m
          'medium'
        elsif shirt_size == :l
          'large'
        else
          'unknown size'
        end
```

Si aucune des conditions d'une chaîne `if` / `elsif` n'est vraie et qu'il n'y a pas de clause `else` , l'expression est nulle. Cela peut être utile dans l'interpolation de chaînes, car `nil.to_s` est la chaîne vide:

```
"user#{'s' if @users.size != 1}"
```

Valeurs véridiques et fausses

Dans Ruby, il y a exactement deux valeurs qui sont considérées comme "falsy", et qui retourneront false si elles sont testées comme condition pour une expression `if` . Elles sont:

- `nil`
- booléen `false`

Toutes les autres valeurs sont considérées comme "véridiques", notamment:

- `0` - zéro numérique (entier ou autre)
- `""` - Chaînes vides
- `"\n"` - Chaînes ne contenant que des espaces
- `[]` - Tableaux vides
- `{}` - Les hashes vides

Prenez, par exemple, le code suivant:

```
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", {})
```

Va sortir:

```
false is falsy
nil is falsy
0 is truthy
empty string is truthy
\n is truthy
empty array is truthy
empty hash is truthy
```

tandis que, jusqu'à

A `while` la boucle exécute le bloc tandis que la condition donnée est satisfaite:

```
i = 0
while i < 5
  puts "Iteration ##{i}"
  i +=1
end
```

Un `until` la boucle exécute le bloc alors que le conditionnel est faux:

```
i = 0
until i == 5
  puts "Iteration ##{i}"
  i +=1
end
```

Inline si / sauf

Un schéma courant consiste à utiliser un inline ou à la fin `if` ou à `unless` :

```
puts "x is less than 5" if x < 5
```

Ceci est connu comme un *modificateur* conditionnel, et est un moyen pratique d'ajouter un code de garde simple et des retours anticipés:

```
def save_to_file(data, filename)
  raise "no filename given" if filename.empty?
  return false unless data.valid?

  File.write(filename, data)
end
```

Il n'est pas possible d'ajouter une clause `else` à ces modificateurs. En outre , il est généralement recommandé de ne pas utiliser les modificateurs conditionnels dans la logique principale - Pour le code complexe on devrait utiliser la normale `if` , `elsif` , d' `else` à la place.

sauf si

Une déclaration commune est `if !(some condition)` . Ruby offre l'alternative de la déclaration `unless` .

La structure est exactement la même qu'une instruction `if` , sauf que la condition est négative. En outre, la déclaration `unless` ne prend pas en charge `elsif` , mais prend en charge `else` :

```
# Prints not inclusive
unless 'hellow'.include?('all')
  puts 'not inclusive'
end
```

Déclaration de cas

Ruby utilise le mot-clé `case` pour les instructions switch.

Selon les Ruby Docs :

> Les instructions de cas sont constituées d'une condition facultative, qui se trouve dans la position d'un argument par rapport à la `case` , et de zéro ou plusieurs `when` clauses. La première clause `when` qui correspond à la condition (ou à l'évaluation de la vérité booléenne, si la condition est nulle) «gagne» et sa strophe de code est exécutée. La valeur de l'instruction case est la valeur de la clause `when` , ou `nil` s'il n'y en a pas.
>
> Une instruction de cas peut se terminer par une clause `else` . Chaque `when` qu'une instruction peut avoir plusieurs valeurs candidates, séparées par des virgules.

Exemple:

```
case x
when 1,2,3
  puts "1, 2, or 3"
when 10
  puts "10"
else
  puts "Some other number"
end
```

Version plus courte:

```
case x
when 1,2,3 then puts "1, 2, or 3"
when 10 then puts "10"
else puts "Some other number"
end
```

La valeur de la clause `case` correspond à chaque clause `when` à l'aide de la méthode `===` (non `==`). Par conséquent, il peut être utilisé avec différents types d'objets.

Une déclaration de `case` peut être utilisée avec des plages :

```
case 17
when 13..19
  puts "teenager"
end
```

Une déclaration de `case` peut être utilisée avec une expression rationnelle :

```
case "google"
when /oo/
  puts "word contains oo"
end
```

Une instruction de `case` peut être utilisée avec un Proc ou un lambda:

```
case 44
when -> (n) { n.even? or n < 0 }
  puts "even or less than zero"
```

```
end
```

Une déclaration de `case` peut être utilisée avec les classes :

```
case x
when Integer
  puts "It's an integer"
when String
  puts "It's a string"
end
```

En implémentant la méthode `===` , vous pouvez créer vos propres classes de correspondance:

```
class Empty
  def self.===(object)
    !object or "" == object
  end
end

case ""
when Empty
  puts "name was empty"
else
  puts "name is not empty"
end
```

Une instruction de `case` peut être utilisée sans valeur à comparer:

```
case
when ENV['A'] == 'Y'
  puts 'A'
when ENV['B'] == 'Y'
  puts 'B'
else
  puts 'Neither A nor B'
end
```

Une instruction de `case` a une valeur, vous pouvez donc l'utiliser comme argument de méthode ou dans une affectation:

```
description = case 16
              when 13..19 then "teenager"
              else ""
              end
```

Contrôle de boucle avec break, next et redo

Le flux d'exécution d'un bloc Ruby peut être contrôlé avec les instructions `break` , `next` et `redo` .

break

La déclaration de `break` sortira immédiatement du bloc. Toutes les instructions restantes dans le bloc seront ignorées et l'itération se terminera comme suit:

```
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

L'instruction `next` retournera immédiatement en haut du bloc et poursuivra l'itération suivante. Toutes les instructions restantes dans le bloc seront ignorées:

```
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

L'instruction `redo` retourne immédiatement en haut du bloc et réessaie la même itération. Toutes les instructions restantes dans le bloc seront ignorées:

```
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

Outre les boucles, ces instructions fonctionnent avec les méthodes d'itération Enumerable, telles que `each` et la `map` :

```
[1, 2, 3].each do |item|
  next if item.even?
  puts "Item: #{item}"
end

# Item: 1
# Item: 3
```

Bloquer les valeurs de résultat

Dans les instructions `break` et `next` , une valeur peut être fournie et sera utilisée comme valeur de résultat de bloc:

```
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
```

jeter, attraper

Contrairement à de nombreux autres langages de programmation, les mots clés `throw` et `catch` ne sont pas liés à la gestion des exceptions dans Ruby.

Dans Ruby, `throw` et `catch` agissent un peu comme des étiquettes dans d'autres langues. Ils sont utilisés pour modifier le flux de contrôle, mais ne sont pas liés à un concept "d'erreur" comme les exceptions.

```
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"
```

Contrôle du flux avec des instructions logiques

Bien que cela puisse sembler contre-intuitif, vous pouvez utiliser des opérateurs logiques pour déterminer si une instruction est exécutée ou non. Par exemple:

```
File.exist?(filename) or STDERR.puts "#{filename} does not exist!"
```

Cela vérifie si le fichier existe et n'imprime le message d'erreur que si ce n'est pas le cas. L'instruction `or` est paresseuse, ce qui signifie qu'elle cessera de s'exécuter une fois qu'il sera certain que sa valeur est vraie ou fausse. Dès que le premier terme est trouvé vrai, il n'est pas nécessaire de vérifier la valeur de l'autre terme. Mais si le premier terme est faux, il doit vérifier le second terme.

Une utilisation courante consiste à définir une valeur par défaut:

```
glass = glass or 'full' # Optimist!
```

Cela définit la valeur du `glass` à «plein» si ce n'est pas déjà fait. Plus précisément, vous pouvez utiliser la version symbolique de `or` :

```
glass ||= 'empty' # Pessimist.
```

Il est également possible d'exécuter la deuxième instruction uniquement si la première est fausse:

```
File.exist?(filename) and puts "#{filename} found!"
```

Encore une fois, `and` est paresseux donc il n'exécutera la deuxième instruction que si nécessaire pour arriver à une valeur.

L'opérateur `or` a une priorité inférieure à `and` . De même, `||` a une préséance inférieure à `&&` . Les formes de symbole ont une priorité plus élevée que les formes de mot. Ceci est pratique pour savoir quand vous voulez mélanger cette technique avec l'affectation:

```
a = 1 and b = 2
#=> a==1
#=> b==2
```

```
a = 1 && b = 2; puts a, b
#=> a==2
#=> b==2
```

Notez que le Guide de style Ruby recommande :

> Les mots clés `and` et `or` sont interdits. La lisibilité minimale ajoutée ne vaut tout simplement pas la forte probabilité d'introduire des bogues subtils. Pour les expressions booléennes, utilisez toujours `&&` et `||` au lieu. Pour le contrôle de flux, utilisez `if` et `unless` ; `&&` et `||` sont également acceptables mais moins clairs.

commence, fin

Le bloc de `begin` est une structure de contrôle regroupant plusieurs instructions.

```
begin
  a = 7
  b = 6
  a * b
end
```

Un bloc de `begin` renvoie la valeur de la dernière instruction du bloc. L'exemple suivant renverra `3` .

```
begin
  1
  2
  3
end
```

Le bloc `begin` est utile pour l'affectation conditionnelle à l'aide de l'opérateur `||=` où plusieurs instructions peuvent être nécessaires pour renvoyer un résultat.

```
circumference ||=
  begin
    radius = 7
    tau = Math::PI * 2
    tau * radius
  end
```

Il peut également être combiné avec d'autres structures de bloc telles que le `rescue` , `ensure` , tant `while` , `if` , `unless` , etc. pour fournir un meilleur contrôle du déroulement du programme.

`Begin` blocs de `do ... end` ne sont pas des blocs de code, comme `{ ... }` ou `do ... end` ; ils ne peuvent pas être transmis aux fonctions.

retour vs suivant: retour non local dans un bloc

Considérez cet extrait *cassé* :

```
def foo
  bar = [1, 2, 3, 4].map do |x|
    return 0 if x.even?
    x
  end
  puts 'baz'
  bar
end
foo # => 0
```

On pourrait s'attendre à ce que le `return` donne une valeur pour le tableau de résultats de bloc de `map` . Donc, la valeur de retour de `foo` serait `[1, 0, 3, 0]` . Au lieu de cela, **`return` renvoie une valeur de la méthode `foo`** . Notez que `baz` n'est pas imprimé, ce qui signifie que l'exécution n'a jamais atteint cette ligne.

`next` avec une valeur fait le tour. Il agit comme un `return` niveau du bloc.

```
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 l'absence de `return` , la valeur renvoyée par le bloc est la valeur de sa dernière expression.

Opérateur d'attribution Or-Equals / Conditional (|| =)

Ruby possède un opérateur or-equals qui permet d'attribuer une valeur à une variable si et seulement si cette variable est `nil` ou `false` .

```
||= # this is the operator that achieves this.
```

cet opérateur avec les doubles tuyaux représentant ou et le signe égal représentant l'attribution d'une valeur. Vous pensez peut-être que cela représente quelque chose comme ceci:

```
x = x || y
```

Cet exemple ci-dessus n'est pas correct. L'opérateur or-equals représente en fait ceci:

```
x || x = y
```

Si `x` évalué à `nil` ou à `false` `x` se voit attribuer la valeur de `y` et reste inchangé dans le cas contraire.

Voici un cas d'utilisation pratique de l'opérateur ou-equals. Imaginez que vous ayez une partie de votre code qui devrait envoyer un courrier électronique à un utilisateur. Que faites-vous si pour quelle raison il n'y a pas de courrier électronique pour cet utilisateur? Vous pourriez écrire quelque chose comme ceci:

```
if user_email.nil?
   user_email = "error@yourapp.com"
end
```

En utilisant l'opérateur ou-equals, nous pouvons couper tout ce morceau de code, fournissant un contrôle et des fonctionnalités propres et clairs.

```
user_email ||= "error@yourapp.com"
```

Dans les cas où `false` est une valeur valide, il faut veiller à ne pas la remplacer accidentellement:

```
has_been_run = false
has_been_run ||= true
#=> true

has_been_run = false
has_been_run = true if has_been_run.nil?
#=> false
```

Opérateur ternaire

Ruby a un opérateur ternaire (`?:` :), Qui renvoie une valeur sur deux si une condition est évaluée comme vraie:

```
conditional ? value_if_truthy : value_if_falsy

value = true
value ? "true" : "false"
#=> "true"

value = false
value ? "true" : "false"
#=> "false"
```

c'est la même chose qu'écrire `if a then b else c end` , bien que le ternaire soit préféré

Exemples:

```
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
```

Opérateur Flip-Flop

L'opérateur flip flop `..` est utilisé entre deux conditions dans une instruction conditionnelle:

```
(1..5).select do |e|
  e if (e == 2) .. (e == 4)
end
# => [2, 3, 4]
```

La condition est `false` *jusqu'à* ce *que* la première partie devienne `true` . Ensuite , il évalue à `true` *jusqu'à ce que* la seconde partie est `true` . Après cela, il redevient `false` .

Cet exemple illustre ce qui est sélectionné:

```
[1, 2, 2, 3, 4, 4, 5].select do |e|
  e if (e == 2) .. (e == 4)
end
# => [2, 2, 3, 4]
```

L'opérateur flip-flop ne fonctionne que dans l'opérateur ifs (y compris `unless`) et l'opérateur ternaire. Sinon, il est considéré comme l'opérateur de la gamme.

```
(1..5).select do |e|
  (e == 2) .. (e == 4)
end
# => ArgumentError: bad value for range
```

Il peut passer de `false` à `true` et inversement plusieurs fois:

```
((1..5).to_a * 2).select do |e|
  e if (e == 2) .. (e == 4)
end
# => [2, 3, 4, 2, 3, 4]
```

Lire Flux de contrôle en ligne: https://riptutorial.com/fr/ruby/topic/640/flux-de-controle

Chapitre 29: Gamme

Examples

Gammes comme séquences

L'utilisation la plus importante des gammes est d'exprimer une séquence

Syntaxe:

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

ou

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

La valeur `end` plus importante doit être supérieure au `begin` , sinon elle ne renverra rien.

Exemples:

```
(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]
```

Itérer sur une gamme

Vous pouvez facilement faire quelque chose pour chaque élément d'une plage.

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

Intervalle entre les dates

```
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"
```

Lire Gamme en ligne: https://riptutorial.com/fr/ruby/topic/3427/gamme

Chapitre 30: Générer un nombre aléatoire

Introduction

Comment générer un nombre aléatoire en Ruby.

Remarques

Alias de Random :: DEFAULT.rand. Cela utilise un générateur de nombres pseudo-aléatoires qui se rapproche du vrai hasard

Examples

6 faces dé

```
   # Roll a 6 sided die, rand(6) returns a number from 0 to 5 inclusive
   dice_roll_result = 1 + rand(6)
```

Générer un nombre aléatoire à partir d'une plage (inclus)

```
# ruby 1.92
lower_limit = 1
upper_limit = 6
Random.new.rand(lower_limit..upper_limit) # Change your range operator to suit your needs
```

Lire Générer un nombre aléatoire en ligne: https://riptutorial.com/fr/ruby/topic/9626/generer-un-nombre-aleatoire

Chapitre 31: Hash

Introduction

Un hachage est une collection de clés uniques de type dictionnaire et leurs valeurs. Aussi appelés tableaux associatifs, ils sont similaires aux tableaux, mais lorsqu'un tableau utilise des entiers comme index, un hachage vous permet d'utiliser n'importe quel type d'objet. Vous récupérez ou créez une nouvelle entrée dans un hachage en vous référant à sa clé.

Syntaxe

- {first_name: "Noel", second_name: "Edmonds"}
- {: first_name => "Noel",: second_name => "Edmonds"}
- {"Prénom" => "Noel", "Second Name" => "Edmonds"}
- {first_key => first_value, second_key => second_value}

Remarques

Hash in Ruby mappe les clés aux valeurs en utilisant une table de hachage.

Tout objet hashable peut être utilisé comme clé. Cependant, il est très courant d'utiliser un `Symbol` car il est généralement plus efficace dans plusieurs versions de Ruby, en raison de l'allocation d'objet réduite.

```
{ key1: "foo", key2: "baz"  }
```

Examples

Créer un hash

Un hachage dans Ruby est un objet qui implémente une table de hachage , mappant les clés aux valeurs. Ruby prend en charge une syntaxe littérale spécifique pour la définition des hachages à l'aide de `{}` :

```
my_hash = {}  # an empty hash
grades = { 'Mark' => 15, 'Jimmy' => 10, 'Jack' => 10 }
```

Un hachage peut également être créé en utilisant la `new` méthode standard:

```
my_hash = Hash.new  # any empty hash
my_hash = {}        # any empty hash
```

Les hachages peuvent avoir des valeurs de tout type, y compris des types complexes tels que des tableaux, des objets et d'autres hachages:

```
mapping = { 'Mark' => 15, 'Jimmy' => [3,4], 'Nika' => {'a' => 3, 'b' => 5} }
mapping['Mark']   # => 15
mapping['Jimmy']  # => [3, 4]
mapping['Nika']   # => {"a"=>3, "b"=>5}
```

Les clés peuvent également être de tout type, y compris les plus complexes:

```
mapping = { 'Mark' => 15, 5 => 10, [1, 2] => 9 }
mapping['Mark']  # => 15
mapping[[1, 2]]  # => 9
```

Les symboles sont couramment utilisés comme clés de hachage et Ruby 1.9 introduit une nouvelle syntaxe spécifiquement pour raccourcir ce processus. Les hachages suivants sont équivalents:

```
# 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 }
```

Le hachage suivant (valide dans toutes les versions de Ruby) est *différent* , car toutes les clés sont des chaînes:

```
grades = { "Mark" => 15, "Jimmy" => 10, "Jack" => 10 }
```

Bien que les deux versions de syntaxe puissent être mélangées, les opérations suivantes sont déconseillées.

```
mapping = { :length => 45, width: 10 }
```

Avec Ruby 2.2+, il existe une syntaxe alternative pour créer un hachage avec des clés de symbole (le plus utile si le symbole contient des espaces):

```
grades = { "Jimmy Choo": 10, :"Jack Sparrow": 10 }
# => { :"Jimmy Choo" => 10, :"Jack Sparrow" => 10}
```

Accès aux valeurs

Les valeurs individuelles d'un hachage sont lues et écrites à l'aide des méthodes `[]` et `[]=` :

```
my_hash = { length: 4, width: 5 }

my_hash[:length] #=> => 4

my_hash[:height] = 9

my_hash #=> {:length => 4, :width => 5, :height => 9 }
```

Par défaut, l'accès à une clé qui n'a pas été ajoutée au hachage renvoie `nil` , ce qui signifie qu'il est toujours prudent d'essayer de rechercher la valeur d'une clé:

```
my_hash = {}

my_hash[:age] # => nil
```

Les hachages peuvent également contenir des clés dans des chaînes. Si vous essayez d'y accéder normalement, il retournera simplement un `nil` , au lieu de cela vous y accéderez par leurs clés de chaîne:

```
my_hash = { "name" => "user" }

my_hash[:name]    # => nil
my_hash["name"]   # => user
```

Pour les situations où des clés sont attendues ou requises, les méthodes de hachage ont une méthode d' `fetch` qui génère une exception lors de l'accès à une clé qui n'existe pas:

```
my_hash = {}

my_hash.fetch(:age) #=> KeyError: key not found: :age
```

`fetch` accepte une valeur par défaut comme deuxième argument, qui est renvoyé si la clé n'a pas été définie précédemment:

```
my_hash =  {}
my_hash.fetch(:age, 45) #=> => 45
```

`fetch` peut également accepter un bloc qui est renvoyé si la clé n'a pas déjà été définie:

```
my_hash = {}
my_hash.fetch(:age) { 21 } #=> 21

my_hash.fetch(:age) do |k|
  puts "Could not find #{k}"
end

#=> Could not find age
```

Les hachages prennent également en charge une méthode de `store` tant qu'alias pour `[]=` :

```
my_hash = {}

my_hash.store(:age, 45)

my_hash #=> { :age => 45 }
```

Vous pouvez également obtenir toutes les valeurs d'un hachage en utilisant la méthode `values` :

```
my_hash = { length: 4, width: 5 }
```

```
my_hash.values #=> [4, 5]
```

Note: Ceci est uniquement pour Ruby 2.3+ `#dig` est pratique pour les `Hash` imbriqués. Extrait la valeur imbriquée spécifiée par la séquence d'objets idx en appelant dig à chaque étape, renvoyant nil si une étape intermédiaire est nulle.

```
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
```

Définition des valeurs par défaut

Par défaut, tenter de rechercher la valeur d'une clé qui n'existe pas renverra `nil` . Vous pouvez éventuellement spécifier une autre valeur à renvoyer (ou une action à effectuer) lorsque le hachage est accessible avec une clé inexistante. Bien que cela soit appelé "la valeur par défaut", il ne doit pas nécessairement s'agir d'une valeur unique. il pourrait s'agir, par exemple, d'une valeur calculée telle que la longueur de la clé.

La valeur par défaut d'un hachage peut être transmise à son constructeur:

```
h = Hash.new(0)

h[:hi] = 1
puts h[:hi]  # => 1
puts h[:bye] # => 0 returns default value instead of nil
```

Un défaut peut également être spécifié sur un hachage déjà construit:

```
my_hash = { human: 2, animal: 1 }
my_hash.default = 0
my_hash[:plant] # => 0
```

Il est important de noter que la **valeur par défaut n'est pas copiée à** chaque fois qu'une nouvelle clé est utilisée, ce qui peut donner des résultats surprenants lorsque la valeur par défaut est un type de référence:

```
# 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']
```

Pour contourner ce problème, le constructeur Hash accepte un bloc qui est exécuté à chaque

accès à une nouvelle clé, et la valeur renvoyée est utilisée par défaut:

```
authors = Hash.new { [] }

# Note that we're using += instead of <<, see below
authors[:homer] += ['The Odyssey']
authors[:plato] # => []

authors # => {:homer=>["The Odyssey"]}
```

Notez que ci-dessus, nous avons dû utiliser + = au lieu de << car la valeur par défaut n'est pas automatiquement attribuée au hachage; using << aurait ajouté au tableau, mais les auteurs [: homer] seraient restés indéfinis:

```
authors[:homer] << 'The Odyssey' # ['The Odyssey']
authors[:homer] # => []
authors # => {}
```

Pour pouvoir attribuer des valeurs par défaut à l'accès, ainsi que pour calculer des valeurs par défaut plus sophistiquées, le bloc par défaut est transmis à la fois au hachage et à la clé:

```
authors = Hash.new { |hash, key| hash[key] = [] }

authors[:homer] << 'The Odyssey'
authors[:plato] # => []

authors # => {:homer=>["The Odyssey"], :plato=>[]}
```

Vous pouvez également utiliser un bloc par défaut pour effectuer une action et / ou renvoyer une valeur dépendant de la clé (ou d'autres données):

```
chars = Hash.new { |hash,key| key.length }

chars[:test] # => 4
```

Vous pouvez même créer des hachages plus complexes:

```
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"}}
```

Pour définir la valeur par défaut sur Proc sur un hachage *déjà existant* , utilisez `default_proc=` :

```
authors = {}
authors.default_proc = proc { [] }

authors[:homer] += ['The Odyssey']
authors[:plato] # => []

authors # {:homer=>["The Odyssey"]}
```

Création automatique d'un hachage profond

Hash a une valeur par défaut pour les clés qui sont demandées mais qui n'existent pas (nil):

```
a = {}
p a[ :b ] # => nil
```

Lors de la création d'un nouveau Hash, vous pouvez spécifier la valeur par défaut:

```
b = Hash.new 'puppy'
p b[ :b ]            # => 'puppy'
```

Hash.new prend également un bloc, ce qui vous permet de créer automatiquement des hachages imbriqués, tels que le comportement d'autovivification de Perl ou `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 } } }
```

Modification des clés et des valeurs

Vous pouvez créer un nouveau hachage avec les clés ou les valeurs modifiées, vous pouvez également ajouter ou supprimer des clés en utilisant inject (AKA, réduire). Par exemple, pour produire un hachage avec des clés et des valeurs majuscules:

```
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 est un enumerable, en substance une collection de paires clé / valeur. Par conséquent, il existe des méthodes telles que `each` , `map` et `inject` .

Pour chaque paire clé / valeur du hachage, le bloc donné est évalué, la valeur de mémo lors de la première exécution est la valeur initiale transmise pour `inject` , dans notre cas, un hachage vide, `{}` . La valeur de `memo` pour les évaluations ultérieures est la valeur renvoyée de l'évaluation des blocs précédents, c'est pourquoi nous modifions le `memo` en définissant une clé avec une valeur, puis renvoyons le `memo` à la fin. La valeur de retour de l'évaluation des blocs finaux est la valeur de retour de `inject` , dans notre `memo` cas.

Pour éviter d'avoir à fournir la valeur finale, vous pouvez utiliser each_with_object à la place:

```
new_fruit = fruit.each_with_object({}) { |(k,v), memo| memo[k.to_s] = v.upcase }
```

Ou même carte :

1.8

```
new_fruit = Hash[fruit.map{ |k,v| [k.to_s, v.upcase] }]
```

(Voir cette réponse pour plus de détails, y compris comment manipuler les hachages en place.)

Itérer sur un hachage

Un `Hash` inclut le module `Enumerable` , qui fournit plusieurs méthodes d'itération, telles que: `Enumerable#each` , `Enumerable#each_pair` , `Enumerable#each_key` et `Enumerable#each_value` .

`.each` et `.each_pair` chaque paire clé-valeur:

```
h = { "first_name" => "John", "last_name" => "Doe" }
h.each do |key, value|
    puts "#{key} = #{value}"
end

# => first_name = John
#    last_name = Doe
```

`.each_key` itère `.each_key` sur les clés:

```
h = { "first_name" => "John", "last_name" => "Doe" }
h.each_key do |key|
  puts key
end

# => first_name
#    last_name
```

`.each_value` itère `.each_value` sur les valeurs:

```
h = { "first_name" => "John", "last_name" => "Doe" }
h.each_value do |value|
    puts value
end

# => John
#    Doe
```

`.each_with_index` les éléments et fournit l'index de l'itération:

```
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
```

Conversion vers et depuis les tableaux

Les hachages peuvent être librement convertis vers et depuis les tableaux. La conversion d'un

hachage de paires clé / valeur en un tableau produira un tableau contenant des tableaux imbriqués pour la paire:

```
{ :a => 1, :b => 2 }.to_a # => [[:a, 1], [:b, 2]]
```

Dans le sens inverse, un hachage peut être créé à partir d'un tableau du même format:

```
[[:x, 3], [:y, 4]].to_h # => { :x => 3, :y => 4 }
```

De même, les Hash peuvent être initialisés en utilisant `Hash[]` et une liste de clés et de valeurs en alternance:

```
Hash[:a, 1, :b, 2] # => { :a => 1, :b => 2 }
```

Ou à partir d'un tableau de tableaux avec deux valeurs chacun:

```
Hash[ [[:x, 3], [:y, 4]] ] # => { :x => 3, :y => 4 }
```

Les hachages peuvent être reconvertis en un tableau de clés et de valeurs en alternance en utilisant `flatten()` :

```
{ :a => 1, :b => 2 }.flatten # => [:a, 1, :b, 2]
```

La conversion facile vers et depuis un tableau permet à `Hash` de bien fonctionner avec de nombreuses méthodes `Enumerable` telles que `collect` et `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 }
```

Obtenir toutes les clés ou valeurs du hachage

```
{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>
```

Fonction de hachage prioritaire

Les hashes Ruby utilisent les méthodes `hash` et `eql?` pour effectuer l'opération de hachage et attribuer des objets stockés dans le hachage à des bacs de hachage internes. L'implémentation par défaut du `hash` dans Ruby est la fonction de hachage de murmure sur tous les champs membres de l'objet haché . Pour contourner ce comportement, il est possible de remplacer le `hash` et `eql?` méthodes.

Comme avec les autres implémentations de hachage, deux objets a et b seront hachés dans le

même `a.hash == b.hash` si `a.hash == b.hash` et seront réputés identiques si `a.eql?(b)` . Ainsi, lors de la réimplémentation du `hash` et de l' `eql?` il faut veiller à ce que si `a` et `b` sont égaux sous `eql?` ils doivent retourner la même valeur de `hash` . Sinon, cela pourrait entraîner des doublons dans un hachage. À l'inverse, un mauvais choix dans l'implémentation du `hash` peut conduire de nombreux objets à partager le même compartiment de hachage, détruisant ainsi le temps de recherche O (1) et provoquant l'appel de `eql?` O (n) `eql?` sur tous les objets.

Dans l'exemple ci-dessous, seule l'instance de la classe `A` est stockée en tant que clé, car elle a été ajoutée en premier:

```
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
```

Filtrage des hashs

`select` renvoie un nouveau `hash` avec des paires clé-valeur pour lesquelles le bloc est évalué comme `true` .

```
{ :a => 1, :b => 2, :c => 3 }.select { |k, v| k != :a && v.even? } # => { :b => 2 }
```

Lorsque vous n'avez pas besoin de la *clé* ou de la *valeur* dans un bloc de filtrage, la convention est d'utiliser un _ à cet endroit:

```
{ :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` renvoie un nouveau `hash` avec des paires clé-valeur pour lesquelles le bloc est évalué à `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 }
```

Définir les opérations sur les hachages

- **Intersection des Hashs**

  Pour obtenir l'intersection de deux hachages, retournez les clés partagées dont les valeurs sont égales:

  ```
  hash1 = { :a => 1, :b => 2 }
  hash2 = { :b => 2, :c => 3 }
  hash1.select { |k, v| (hash2.include?(k) && hash2[k] == v) } # => { :b => 2 }
  ```

- **Union des fusions:**

  les clés d'un hachage sont uniques, si une clé apparaît dans les deux hachages à fusionner, celle du hachage sur lequel la `merge` est appelée est écrasée:

  ```
  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 }
  ```

Lire Hash en ligne: https://riptutorial.com/fr/ruby/topic/288/hash

Chapitre 32: Héritage

Syntaxe

- class SubClass <SuperClass

Examples

Refactoring des classes existantes pour utiliser l'héritage

Disons que nous avons deux classes, `Cat` et `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
```

La méthode de `eat` est exactement la même dans ces deux classes. Bien que cela fonctionne, il est difficile à maintenir. Le problème s'aggravera s'il y a plus d'animaux avec la même méthode de `eat` . L'héritage peut résoudre ce problème.

```
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
```

Nous avons créé une nouvelle classe, `Animal` , et avons déplacé notre méthode de `eat` à cette classe. Nous avons ensuite fait en sorte que `Cat` and `Dog` hérite de cette nouvelle superclasse commune. Cela supprime le besoin de répéter le code

Héritage multiple

L'héritage multiple est une fonctionnalité qui permet à une classe d'hériter de plusieurs classes (c.-à-d. Plusieurs parents). Ruby ne prend pas en charge l'héritage multiple. Il ne prend en charge que l'héritage simple (c.-à-d. Que la classe ne peut avoir qu'un seul parent), mais vous pouvez utiliser la *composition* pour créer des classes plus complexes à l'aide de modules .

Des sous-classes

L'héritage permet aux classes de définir un comportement spécifique basé sur une classe existante.

```
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!'
```

Dans cet exemple:

- `Dog` hérite de l' `Animal` , ce qui en fait une *sous - classe* .
- `Dog` acquiert à la fois les méthodes `say_hello` et `eat` de `Animal` .
- `Dog` remplace la méthode `say_hello` avec différentes fonctionnalités.

Mixins

Les mixins sont un excellent moyen de réaliser quelque chose de similaire à l'héritage multiple.

Cela nous permet d'hériter ou plutôt d'inclure des méthodes définies dans un module dans une classe. Ces méthodes peuvent être incluses en tant que méthodes d'instance ou de classe. L'exemple ci-dessous illustre cette conception.

```
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"
```

Qu'est-ce qui est hérité?

Les méthodes sont héritées

```
class A
  def boo; p 'boo' end
end

class B < A; end

b = B.new
b.boo # => 'boo'
```

Les méthodes de classe sont héritées

```
class A
  def self.boo; p 'boo' end
end
```

```
class B < A; end

p B.boo # => 'boo'
```

Les constantes sont héritées

```
class A
  WOO = 1
end

class B < A; end

p B::WOO # => 1
```

Mais méfiez-vous, ils peuvent être remplacés:

```
class B
  WOO = WOO + 1
end

p B::WOO # => 2
```

Les variables d'instance sont héritées:

```
class A
  attr_accessor :ho
  def initialize
    @ho = 'haha'
  end
end

class B < A; end

b = B.new
p b.ho # => 'haha'
```

Attention, si vous remplacez les méthodes qui initialisent les variables d'instance sans appeler `super` , elles seront nulles. En continuant d'en haut:

```
class C < A
  def initialize; end
 end

c = C.new
p c.ho    # => nil
```

Les variables d'instance de classe ne sont pas héritées:

```
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
```

Les variables de classe ne sont pas vraiment héritées

Ils sont partagés entre la classe de base et toutes les sous-classes sous la forme d'une 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
```

Donc en continuant d'en haut:

```
class C < A
  def initialize
    @@foo = -10
    p @@foo
  end
end

a = C.new # => -10
b = B.new # => -9
```

Lire Héritage en ligne: https://riptutorial.com/fr/ruby/topic/625/heritage

Chapitre 33: Installation

Examples

Linux - Compiler à partir des sources

`De cette façon, vous obtiendrez le rubis le plus récent mais il a ses inconvénients. Faire comme ce ruby ne sera géré par aucune application.

!! N'oubliez pas de chagé la version pour qu'elle corresponde à votre !!

1. vous devez télécharger une archive contenant un lien sur un site officiel (https://www.ruby-lang.org/fr/downloads/)
2. Extraire l'archive
3. Installer

```
$> 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
```

Cela installera ruby dans `/usr/local` . Si vous n'êtes pas satisfait de cet emplacement, vous pouvez passer un argument à `./configure --prefix=DIR` où `DIR` est le répertoire dans lequel vous souhaitez installer ruby.

Linux: installation à l'aide d'un gestionnaire de packages

Probablement le choix le plus facile, mais attention, la version n'est pas toujours la plus récente. Ouvrez simplement le terminal et tapez (en fonction de votre distribution)

dans Debian ou Ubuntu en utilisant apt

```
$> sudo apt install ruby
```

dans CentOS, openSUSE ou Fedora

```
$> sudo yum install ruby
```

Vous pouvez utiliser l'option `-y` pour que vous ne soyez pas invité à accepter l'installation, mais à mon avis, il est recommandé de toujours vérifier ce que le gestionnaire de paquets tente d'installer.

Windows - Installation à l'aide du programme d'installation

La méthode la plus simple pour installer ruby sur Windows consiste à aller sur

http://rubyinstaller.org/ et à télécharger un exécutable que vous allez installer.

Vous n'avez pas à définir presque tout, mais il y aura une fenêtre importante. Il comportera une case à cocher indiquant *Ajouter un exécutable ruby à votre PATH* . Confirmez qu'il est **coché** , sinon cochez-le, sinon vous ne pourrez pas lancer ruby et définir vous-même la variable PATH.

Ensuite, allez ensuite jusqu'à ce qu'il installe et c'est ça.

Gemmes

Dans cet exemple, nous utiliserons «nokogiri» comme exemple de gem. 'nokogiri' peut plus tard être remplacé par un autre nom de gemme.

Pour travailler avec des gemmes, nous utilisons un outil de ligne de commande appelé `gem` suivi d'une option comme `install` ou `update` , puis les noms des gems que nous voulons installer, mais ce n'est pas tout.

Installez des gemmes:

```
$> gem install nokogiri
```

Mais ce n’est pas la seule chose dont nous avons besoin. Nous pouvons également spécifier la version, la source à partir de laquelle installer ou rechercher des gemmes. Commençons par quelques cas d'utilisation de base (UC) et vous pourrez plus tard poster une demande de mise à jour.

Liste de toutes les gemmes installées:

```
$> gem list
```

Désinstallation des gems:

```
$> gem uninstall nokogiri
```

Si nous avons plus de version du gem nokogiri, nous serons invités à spécifier celui que nous voulons désinstaller. Nous allons obtenir une liste qui est ordonnée et numérotée et nous écrivons simplement le numéro.

Mise à jour des gemmes

```
$> gem update nokogiri
```

ou si nous voulons les mettre à jour tous

```
$> gem update
```

Comman `gem` a beaucoup plus d'utilisations et d'options à explorer. Pour plus d'informations, veuillez consulter la documentation officielle. Si quelque chose n'est pas clair, envoyez une

demande et je l'ajouterai.

Linux - Dépannage de l'installation de gem

Première UC dans l'exemple **Gems** `$> gem install nokogiri` peut avoir un problème pour installer des gemmes parce que nous n'avons pas les permissions pour cela. Cela peut être réglé de plus d'une manière.

Première solution UC a:

U peut utiliser `sudo` . Cela va installer la gem pour tous les utilisateurs. Cette méthode devrait être mal vue. Cela ne devrait être utilisé qu'avec le bijou que vous savez utilisable par tous les utilisateurs. Habituellement, dans la vraie vie, vous ne voulez pas qu'un utilisateur ait accès à `sudo` .

```
$> sudo gem install nokogiri
```

Première solution UC b

U peut utiliser l'option `--user-install` qui installe les gemmes dans le dossier gem de votre utilisateur (habituellement à `~/.gem`)

```
&> gem install nokogiri --user-install
```

Première solution UC c

Vous pouvez définir GEM_HOME et GEM_PATH, qui feront ensuite que la commande `gem install` installera toutes les gemmes dans un dossier que vous spécifiez. Je peux vous en donner un exemple (la manière habituelle)

- Tout d'abord, vous devez ouvrir .bashrc. Utilisez nano ou votre éditeur de texte préféré.

```
$> nano ~/.bashrc
```

- Ensuite, à la fin de ce fichier, écrivez

```
export GEM_HOME=$HOME/.gem
export GEM_PATH=$HOME/.gem
```

- Maintenant, vous devrez redémarrer le terminal ou écrire . `~/.bashrc` pour recharger la configuration. Cela vous permettra d'utiliser `gem isntall nokogiri` et il installera ces gems dans le dossier que vous avez spécifié.

Installation de Ruby MacOS

La bonne nouvelle est que Apple inclut un interprète Ruby. Malheureusement, ce n'est pas une version récente:

```
$ /usr/bin/ruby -v
ruby 2.0.0p648 (2015-12-16 revision 53162) [universal.x86_64-darwin16]
```

Si vous avez installé Homebrew , vous pouvez obtenir le dernier Ruby avec:

```
$ brew install ruby

$ /usr/local/bin/ruby -v
ruby 2.4.1p111 (2017-03-22 revision 58053) [x86_64-darwin16]
```

(Il est probable que vous verrez une version plus récente si vous essayez ceci.)

Pour récupérer la version brassée sans utiliser le chemin complet, vous devez ajouter `/usr/local/bin` au début de votre variable d'environnement `$PATH` :

```
export PATH=/usr/local/bin:$PATH
```

L'ajout de cette ligne à `~/.bash_profile` garantit que vous obtiendrez cette version après le redémarrage de votre système:

```
$ type ruby
ruby is /usr/local/bin/ruby
```

Homebrew installera `gem` pour l' installation de Gems . Il est également possible de construire à partir de la source si vous en avez besoin. Homebrew comprend également cette option:

```
$ brew install ruby --build-from-source
```

Lire Installation en ligne: https://riptutorial.com/fr/ruby/topic/8095/installation

Chapitre 34: instance_eval

Syntaxe

- object.instance_eval 'code'
- object.instance_eval 'code', 'nomfichier'
- object.instance_eval 'code', 'nomfichier', 'numéro de ligne'
- object.instance_eval {code}
- object.instance_eval {| récepteur | code}

Paramètres

Paramètre	Détails
`string`	Contient le code source Ruby à évaluer.
`filename`	Nom du fichier à utiliser pour le rapport d'erreur.
`lineno`	Numéro de ligne à utiliser pour le rapport d'erreur.
`block`	Le bloc de code à évaluer.
`obj`	Le récepteur est transmis au bloc comme seul argument.

Examples

Évaluation d'instance

La méthode `instance_eval` est disponible sur tous les objets. Il évalue le code dans le contexte du récepteur:

```
object = Object.new

object.instance_eval do
  @variable = :value
end

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

`instance_eval` définit lui- `self object` pour la durée du bloc de code:

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

Le récepteur est également transmis au bloc comme seul argument:

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

La méthode `instance_exec` diffère à cet égard: elle transmet ses arguments au bloc à la place.

```
object.instance_exec :@variable do |name|
  instance_variable_get name # => :value
end
```

Mise en œuvre avec

De nombreux langages disposent d'une instruction `with` qui permet aux programmeurs d'omettre le récepteur d'appels de méthode.

`with` peut être facilement émulé dans Ruby en utilisant `instance_eval` :

```
def with(object, &block)
  object.instance_eval &block
end
```

La méthode `with` peut être utilisée pour exécuter de manière transparente des méthodes sur des objets:

```
hash = Hash.new

with hash do
  store :key, :value
  has_key? :key       # => true
  values              # => [:value]
end
```

Lire instance_eval en ligne: https://riptutorial.com/fr/ruby/topic/5049/instance-eval

Chapitre 35: Introspection

Examples

Voir les méthodes d'un objet

Inspection d'un objet

Vous pouvez trouver les méthodes publiques auxquelles un objet peut répondre en utilisant les `methods` ou les `methods public_methods` , qui renvoient un tableau de symboles:

```
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]
```

Pour une liste plus ciblée, vous pouvez supprimer des méthodes communes à tous les objets, par exemple

```
p (f.methods - Object.methods).sort
#=> [:bar,:yay]
```

Vous pouvez également transmettre `false` aux `methods` ou `public_methods` :

```
p f.methods(false) # public and protected singleton methods of `f`
#=> [:yay]

p f.public_methods(false)
#=> [:yay, :bar]
```

Vous pouvez trouver les méthodes privées et protégées d'un objet en utilisant `private_methods` et `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
#=> []
```

Comme avec les `methods` et `public_methods` , vous pouvez transmettre `false` à `private_methods` et `protected_methods` pour supprimer les méthodes héritées.

Inspection d'une classe ou d'un module

Outre les `methods` , `public_methods` , `protected_methods` et `private_methods` , classes et modules exposent `instance_methods` , `public_instance_methods` , `protected_instance_methods` et `private_instance_methods` pour déterminer les méthodes exposées pour les objets qui héritent de la classe ou du module. Comme ci-dessus, vous pouvez transmettre `false` à ces méthodes pour exclure les méthodes héritées:

```
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]
```

Enfin, si vous oubliez les noms de la plupart d'entre eux à l'avenir, vous pouvez trouver toutes ces méthodes en utilisant des `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]
```

Afficher les variables d'instance d'un objet

Il est possible d'interroger un objet sur ses variables d' `instance_variables` utilisant

`instance_variables` , `instance_variable_defined?` , et `instance_variable_get` , et modifiez-les en utilisant `instance_variable_set` et `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                #=> []
```

Les noms des variables d'instance incluent le symbole `@` . Vous obtiendrez une erreur si vous l'omettez:

```
f.instance_variable_defined?(:jim)
#=> NameError: `jim' is not allowed as an instance variable name
```

Afficher les variables globales et locales

Le `Kernel` expose des méthodes pour obtenir la liste des `global_variables` et des `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]
```

Contrairement aux variables d'instance, il n'existe aucune méthode spécifique pour obtenir, définir ou supprimer des variables globales ou locales. Rechercher une telle fonctionnalité est généralement un signe que votre code doit être réécrit pour utiliser un hachage pour stocker les valeurs. Cependant, si vous devez modifier des variables globales ou locales par nom, vous pouvez utiliser `eval` avec une chaîne:

```
var = "$demo"
eval(var)           #=> "in progress"
eval("#{var} = 17")
p $demo             #=> 17
```

Par défaut, `eval` évaluera vos variables dans la portée actuelle. Pour évaluer les variables locales dans une portée différente, vous devez capturer la *liaison* où les variables locales existent.

```
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
```

Dans ce qui précède, `test_1` n'a pas passé une liaison aux `local_variable_get` , et donc le `eval` a été exécuté dans le cadre de cette méthode, où une variable locale nommée `foo` a été réglée à `:inside` l' `:inside` .

Afficher les variables de classe

Les classes et les modules ont les mêmes méthodes pour l'introspection des variables d'instance que tout autre objet. La classe et les modules ont également des méthodes similaires pour interroger les variables de classe (`@@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
```

Semblable aux variables d'instance, le nom des variables de classe doit commencer par `@@` , ou vous obtiendrez une erreur:

```
p Bar.class_variable_defined?( :instances )
```

```
#=> NameError: `instances' is not allowed as a class variable name
```

Lire Introspection en ligne: https://riptutorial.com/fr/ruby/topic/6227/introspection

Chapitre 36: Introspection en rubis

Introduction

Qu'est-ce que l'introspection?

L'introspection cherche à connaître l'intérieur. C'est une définition simple de l'introspection.

Dans la programmation et Ruby en général… l'introspection est la capacité de regarder l'objet, la classe… au moment de l'exécution pour en savoir plus.

Examples

Permet de voir quelques exemples

Exemple:

```
s = "Hello"  # s is a string
```

Ensuite, nous découvrons quelque chose à propos de l'art. Commençons:

Donc, vous voulez savoir quelle est la classe de s au moment de l'exécution?

```
irb(main):055:0* s.class
=> String
```

Ohh bien. Mais quelles sont les méthodes 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__]
```

Vous voulez savoir si s est une instance de String?

```
irb(main):017:0*
irb(main):018:0* s.instance_of?(String)
=> true
```

Quelles sont les méthodes publiques 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__]
```

et méthodes privées ....

```
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]
```

Oui, avez-vous un nom de méthode supérieur. Vous voulez obtenir la version majuscule de s? Essayons:

```
irb(main):044:0> s.respond_to?(:upper)
=> false
```

Regardez non, la bonne méthode est upcase permet de vérifier:

```
irb(main):047:0*
irb(main):048:0* s.respond_to?(:upcase)
=> true
```

Introspection de classe

Permet de suivre la définition de la classe

```
class A
   def a; end
end

module B
   def b; end
end

class C < A
   include B
   def c; end
end
```

Quelles sont les méthodes d'instance de `C` ?

```
C.instance_methods # [:c, :b, :a, :to_json, :instance_of?...]
```

Quelles sont les méthodes d'instance qui ne déclarent que sur `C` ?

```
C.instance_methods(false) # [:c]
```

Quels sont les ancêtres de la classe `C` ?

```
C.ancestors # [C, B, A, Object,...]
```

Superclass de `C` ?

```
C.superclass # A
```

Lire Introspection en rubis en ligne: https://riptutorial.com/fr/ruby/topic/8752/introspection-en-rubis

Chapitre 37: IRB

Introduction

IRB signifie "Interactive Ruby Shell". Fondamentalement, il vous permet d'exécuter des commandes de ruby en temps réel (comme le fait le shell normal). IRB est un outil indispensable pour traiter l'API Ruby. Fonctionne comme un script rb classique. Utilisez-le pour des commandes courtes et faciles. Une des fonctions IRB intéressantes est que lorsque vous appuyez sur la touche tab lorsque vous tapez une méthode, cela vous donne des conseils sur ce que vous pouvez utiliser (ce n'est pas un IntelliSense).

Paramètres

Option	Détails
-F	Supprimer la lecture de ~ / .irbrc
-m	Mode Bc (charge mathn, fraction ou matrice disponible)
-ré	Définissez $ DEBUG sur true (identique à `ruby -d ')
-r module de chargement	Identique à `ruby -r '
-Je chemin	Spécifiez le répertoire $ LOAD_PATH
-U	Identique à `ruby -U`
-E enc	Identique au `ruby -E`
-w	Identique à `ruby -w`
-W [niveau = 2]	Identique à `ruby -W`
--inspecter	Utilisez `inspect 'pour la sortie (par défaut sauf pour le mode bc)
--noinspect	Ne pas utiliser inspect pour la sortie
--readline	Utiliser le module d'extension Readline
--noreadline	N'utilisez pas le module d'extension Readline
--prompt prompt-mode	Changer le mode d'invite. Les modes d'invite prédéfinis sont `default', "simple", `xmp' and` inf-ruby".
--inf-ruby-mode	Utilisez l'invite appropriée pour inf-ruby-mode sur emacs. Supprime --readline.

Option	Détails
--simple-prompt	Mode invite simple
--poprompt	Pas de mode invite
--traceur	Trace d'affichage pour chaque exécution de commandes.
--back-trace-limit n	Afficher les backtraces top n et tail n. La valeur par défaut est 16.
--irb_debug n	Définir le niveau de débogage interne sur n (pas pour une utilisation courante)
-v, --version	Imprimer la version de l'irb

Examples

Utilisation de base

IRB signifie "Interactive Ruby Shell", nous permettant d'exécuter des expressions ruby à partir de l'entrée standard.

Pour commencer, tapez `irb` dans votre shell. Vous pouvez écrire n'importe quoi en Ruby, à partir d'expressions simples:

```
$ irb
2.1.4 :001 > 2+2
=> 4
```

aux cas complexes comme les méthodes:

```
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
```

Démarrage d'une session IRB dans un script Ruby

A partir de Ruby 2.4.0, vous pouvez démarrer une session IRB interactive dans n'importe quel script Ruby en utilisant ces lignes:

```
require 'irb'
binding.irb
```

Cela démarrera une REPL IBR où vous aurez la valeur attendue pour vous- `self` et vous pourrez accéder à toutes les variables locales et variables d'instance qui sont dans la portée. Tapez Ctrl +

D ou `quit` pour reprendre votre programme Ruby.

Cela peut être très utile pour le débogage.

Lire IRB en ligne: https://riptutorial.com/fr/ruby/topic/4800/irb

Chapitre 38: Itération

Examples

Chaque

Ruby possède de nombreux types d'énumérateurs, mais le premier et le plus simple type d'énumérateur pour commencer est `each` . Nous imprimerons `even` ou `odd` pour chaque nombre entre `1` et `10` pour montrer comment `each` fonctionne.

Fondamentalement, il y a deux façons de faire passer les soi-disant `blocks` . Un `block` est un morceau de code passé qui sera exécuté par la méthode appelée. `each` méthode prend un `block` qu'elle appelle pour chaque élément de la collection d'objets sur laquelle elle a été appelée.

Il existe deux manières de transmettre un bloc à une méthode:

Méthode 1: Inline

```
(1..10).each { |i| puts i.even? ? 'even' : 'odd' }
```

C'est un moyen très compressé et *ruby* de résoudre ce problème. Décomposons cela pièce par pièce.

1. `(1..10)` est une plage de `1` à `10` inclus. Si nous voulions qu'il soit exclusif de `1` à `10` , nous écririons `(1...10)` .
2. `.each` est un énumérateur qui énumère `each` élément de l'objet sur lequel il agit. Dans ce cas, il agit sur `each` nombre de la plage.
3. `{ |i| puts i.even? ? 'even' : 'odd' }` est le bloc de `each` instruction, qui peut encore être décomposé.
    1. `|i|` cela signifie que chaque élément de la plage est représenté dans le bloc par l'identifiant `i` .
    2. `puts` est une méthode de sortie dans Ruby qui a un saut de ligne automatique après chaque impression. (Nous pouvons utiliser `print` si nous ne voulons pas le saut de ligne automatique)
    3. `i.even?` vérifie si `i` pair. Nous aurions aussi pu utiliser `i % 2 == 0` ; Cependant, il est préférable d'utiliser des méthodes intégrées.
    4. `? "even" : "odd"` c'est l'opérateur ternaire de ruby. La manière dont un opérateur ternaire est construit est l' `expression ? a : b` . C'est court pour

    ```
    if expression
      a
    else
      b
    end
    ```

Pour un code supérieur à une ligne, le `block` doit être transmis en tant que `multiline block` .

Méthode 2: multiligne

```
(1..10).each do |i|
  if i.even?
    puts 'even'
  else
    puts 'odd'
  end
end
```

Dans un `multiline block` le `do` remplace le support d'ouverture et la `end` remplace le crochet de fermeture du style en `inline` .

Ruby prend également en charge reverse_each. Il va parcourir le tableau en arrière.

```
@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"
```

Implémentation en classe

`Enumerable` est le module le plus populaire de Ruby. Son but est de vous fournir des méthodes itérables telles que la `map` , la `select` , la `reduce` , etc. Les classes qui utilisent `Enumerable` incluent `Array` , `Hash` , `Range` . Pour l'utiliser, vous devez `include Enumerable` et implémenter `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]
```

Carte

Renvoie l'objet modifié, mais l'objet d'origine reste tel qu'il était. Par exemple:

```
arr = [1, 2, 3]
arr.map { |i| i + 1 } # => [2, 3, 4]
arr # => [1, 2, 3]
```

`map!` change l'objet d'origine:

```
arr = [1, 2, 3]
arr.map! { |i| i + 1 } # => [2, 3, 4]
arr # => [2, 3, 4]
```

Remarque: vous pouvez également utiliser `collect` pour faire la même chose.

Itérer sur des objets complexes

Tableaux

Vous pouvez itérer sur les tableaux imbriqués:

```
[[1, 2], [3, 4]].each { |(a, b)| p "a: #{ a }", "b: #{ b }" }
```

La syntaxe suivante est également autorisée:

```
[[1, 2], [3, 4]].each { |a, b| "a: #{ a }", "b: #{ b }" }
```

Produira:

```
"a: 1"
"b: 2"
"a: 3"
"b: 4"
```

Hash

Vous pouvez parcourir les paires clé-valeur:

```
{a: 1, b: 2, c: 3}.each { |pair| p "pair: #{ pair }" }
```

Produira:

```
"pair: [:a, 1]"
"pair: [:b, 2]"
"pair: [:c, 3]"
```

Vous pouvez itérer simultanément les clés et les valeurs:

```
{a: 1, b: 2, c: 3}.each { |(k, v)| p "k: #{ k }", "v: #{ k }" }
```

Produira:

```
"k: a"
"v: a"
"k: b"
"v: b"
"k: c"
"v: c"
```

Pour itérateur

Cela itère de 4 à 13 (inclus).

```
for i in 4..13
    puts "this is #{i}.th number"
end
```

Nous pouvons également parcourir les tableaux en utilisant pour

```
names = ['Siva', 'Charan', 'Naresh', 'Manish']

for name in names
    puts name
end
```

Itération avec index

Parfois, vous voulez connaître la position (**index**) de l'élément en cours lors d'une itération sur un énumérateur. À cette fin, Ruby fournit la méthode `with_index` . Il peut être appliqué à tous les enquêteurs. Fondamentalement, en ajoutant `with_index` à une énumération, vous pouvez énumérer cette énumération. Index est transmis à un bloc comme deuxième argument.

```
[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` a un argument optionnel - le premier index qui est `0` par défaut:

```
[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]
```

Il existe une méthode spécifique `each_with_index` . La seule différence entre cela et `each.with_index` est que vous ne pouvez pas passer un argument à cela, donc le premier index est `0` tout le temps.

```
[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]
```

Lire Itération en ligne: https://riptutorial.com/fr/ruby/topic/1159/iteration

Chapitre 39: JSON avec Ruby

Examples

Utiliser JSON avec Ruby

JSON (JavaScript Object Notation) est un format d'échange de données léger. De nombreuses applications Web l'utilisent pour envoyer et recevoir des données.

Dans Ruby, vous pouvez simplement travailler avec JSON.

Au début, vous devez `require 'json'` , puis vous pouvez analyser une chaîne JSON via la commande `JSON.parse()` .

```
require 'json'

j = '{"a": 1, "b": 2}'
puts JSON.parse(j)
>> {"a"=>1, "b"=>2}
```

Ce qui se passe ici, c'est que l'analyseur génère un Ruby Hash à partir du JSON.

À l'inverse, la génération de JSON à partir d'un hash Ruby est aussi simple que l'analyse. La méthode de choix est `to_json` :

```
require 'json'

hash = { 'a' => 1, 'b' => 2 }
json = hash.to_json
puts json
>> {"a":1,"b":2}
```

Utiliser des symboles

Vous pouvez utiliser JSON avec les symboles Ruby. Avec l'option symbolize_names pour l'analyseur, les clés dans le hachage résultant seront des symboles au lieu de chaînes.

```
json = '{ "a": 1, "b": 2 }'
puts JSON.parse(json, symbolize_names: true)
>> {:a=>1, :b=>2}
```

Lire JSON avec Ruby en ligne: https://riptutorial.com/fr/ruby/topic/5853/json-avec-ruby

Chapitre 40: La destruction

Examples

Vue d'ensemble

La majeure partie de la magie de la déstructuration utilise l'opérateur splat (`*`).

Exemple	Résultat / *commentaire*
`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` *équivalent à* `.first`
`*, z = [0,1,2,3]`	`z=3` *équivalent à* `.last`

Arguments de bloc de destruction

```
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"]
```

Lire La destruction en ligne: https://riptutorial.com/fr/ruby/topic/4739/la-destruction

Chapitre 41: Le débogage

Examples

Passer du code avec Pry et Byebug

Tout d'abord, vous devez installer `pry-byebug` gem. Exécutez cette commande:

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

Ajoutez cette ligne en haut de votre fichier `.rb` :

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

Ensuite, insérez cette ligne là où vous voulez un point d'arrêt:

```
binding.pry
```

Un exemple `hello.rb` :

```
require 'pry-byebug'

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

Lorsque vous exécutez le fichier `hello.rb` , le programme se mettra en pause sur cette ligne. Vous pouvez ensuite parcourir votre code avec la commande `step` . Tapez le nom d'une variable pour en connaître la valeur. Quittez le débogueur avec `exit-program` ou `!!!` .

Lire Le débogage en ligne: https://riptutorial.com/fr/ruby/topic/7691/le-debogage

Chapitre 42: Les constantes

Syntaxe

- MY_CONSTANT_NAME = "ma valeur"

Remarques

Les constantes sont utiles dans Ruby lorsque vous avez des valeurs que vous ne souhaitez pas modifier par erreur dans un programme, telles que les clés API.

Examples

Définir une constante

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

Le nom de la constante commence par une majuscule. Tout ce qui commence par une majuscule est considéré comme `constant` dans Ruby. Donc `class` et `module` sont également constants. La meilleure pratique consiste à utiliser toutes les lettres majuscules pour déclarer une constante.

Modifier une constante

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

Le code ci-dessus génère un avertissement, car vous devez utiliser des variables si vous souhaitez modifier leurs valeurs. Cependant, il est possible de changer une lettre à la fois dans une constante sans avertissement, comme ceci:

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

Maintenant, après avoir changé la deuxième lettre de `MY_CONSTANT` , il devient `"Hullo, world"` .

Les constantes ne peuvent pas être définies dans les méthodes

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

Le code ci-dessus génère une erreur: `SyntaxError: (irb):2: dynamic constant assignment` .

Définir et modifier des constantes dans une classe

```
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` peut être modifiée avec le code suivant:

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

Lire Les constantes en ligne: https://riptutorial.com/fr/ruby/topic/4093/les-constantes

Chapitre 43: Les méthodes

Introduction

Les fonctions de Ruby fournissent un code organisé et réutilisable pour préparer un ensemble d'actions. Les fonctions simplifient le processus de codage, empêchent la logique redondante et facilitent le suivi du code. Cette rubrique décrit la déclaration et l'utilisation des fonctions, des arguments, des paramètres, des instructions de rendement et de la portée dans Ruby.

Remarques

Une **méthode** est un bloc de code nommé, associé à un ou plusieurs objets et généralement identifié par une liste de paramètres en plus du nom.

```
def hello(name)
  "Hello, #{name}"
end
```

Un appel de méthode spécifie le nom de la méthode, l'objet sur lequel il doit être appelé (parfois appelé récepteur) et zéro ou plusieurs valeurs d'argument affectées aux paramètres de la méthode nommée. La valeur de la dernière expression évaluée dans la méthode devient la valeur de l'expression d'invocation de la méthode.

```
hello("World")
# => "Hello, World"
```

Lorsque le récepteur n'est pas explicite, il est `self` .

```
self
# => main

self.hello("World")
# => "Hello, World"
```

Comme expliqué dans le manuel *Ruby Programming Language* , de nombreux langages distinguent les fonctions sans objet associé et les méthodes appelées sur un objet récepteur. Ruby étant un langage purement orienté objet, toutes les méthodes sont des méthodes vraies et sont associées à au moins un objet.

Vue d'ensemble des paramètres de la méthode

Type	Signature de méthode	Exemple d'appel	Les devoirs
R Évalué	`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]`

Type	Signature de méthode	Exemple d'appel	Les devoirs
D efault	`def fn(a=0,b=1)`	`fn(2,3)`	`a=2, b=3`
K eyword	`def fn(a:0,b:1)`	`fn(a:2,b:3)`	`a=2, b=3`

Ces types d'arguments peuvent être combinés de presque toutes les manières possibles pour créer des fonctions variadiques. Le nombre minimum d'arguments de la fonction sera égal à la quantité d'arguments requis dans la signature. Les arguments supplémentaires seront d'abord affectés aux paramètres par défaut, puis au paramètre `*rest` .

Type	Signature de méthode	Exemple d'appel	Les devoirs
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

Paramètre unique requis

```
def say_hello_to(name)
    puts "Hello #{name}"
end
```

```
say_hello_to('Charles')    # Hello Charles
```

Plusieurs paramètres requis

```
def greet(greeting, name)
    puts "#{greeting} #{name}"
end
```

```
greet('Hi', 'Sophie')    # Hi Sophie
```

Paramètres par défaut

```
def make_animal_sound(sound = 'Cuack')
    puts sound
end
```

```
make_animal_sound('Mooo') # Mooo
make_animal_sound         # Cuack
```

Il est possible d'inclure des valeurs par défaut pour plusieurs arguments:

```
def make_animal_sound(sound = 'Cuack', volume = 11)
    play_sound(sound, volume)
end

make_animal_sound('Mooo') # Spinal Tap cow
```

Cependant, il n'est pas possible de fournir le second sans fournir le premier. Au lieu d'utiliser des paramètres de position, essayez les paramètres de mot clé:

```
def make_animal_sound(sound: 'Cuack', volume: 11)
    play_sound(sound, volume)
end

make_animal_sound(volume: 1) # Duck whisper
```

Ou un paramètre de hachage qui stocke les options:

```
def make_animal_sound(options = {})
    options[:sound]  ||= 'Cuak'
    options[:volume] ||= 11
    play_sound(sound, volume)
end

make_animal_sound(:sound => 'Mooo')
```

Les valeurs de paramètres par défaut peuvent être définies par n'importe quelle expression ruby. L'expression s'exécutera dans le contexte de la méthode, vous pouvez même déclarer des variables locales ici. Notez que vous ne passerez pas en revue les codes. Gracieuseté de caius pour l' avoir signalé .

```
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)
```

Paramètre (s) facultatif (opérateur 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!
```

Notez que `welcome_guests(['Rob', 'Sally', 'Lucas'])` affichera `Welcome ["Rob", "Sally", "Lucas"]!` Au lieu de cela, si vous avez une liste, vous pouvez faire `welcome_guests(*['Rob', 'Sally', 'Lucas'])` et cela fonctionnera comme `welcome_guests('Rob', 'Sally', 'Lucas')` .

Mélange de paramètres facultatif par défaut requis

```
def my_mix(name,valid=true, *opt)
    puts name
    puts valid
    puts opt
end
```

Appelez comme suit:

```
my_mix('me')
# 'me'
# true
# []

my_mix('me', false)
# 'me'
# false
# []

my_mix('me', true, 5, 7)
# 'me'
# true
# [5,7]
```

Les définitions de méthode sont des expressions

Définir une méthode dans Ruby 2.x renvoie un symbole représentant le nom:

```
class Example
  puts def hello
  end
end

#=> :hello
```

Cela permet des techniques de métaprogrammation intéressantes. Par exemple, les méthodes peuvent être encapsulées par d'autres méthodes:

```
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.
```

Capture d'arguments de mots clés non déclarés (double splat)

L'opérateur `**` fonctionne de la même manière que l'opérateur `*` mais il s'applique aux paramètres de mot-clé.

```
def options(required_key:, optional_key: nil, **other_options)
  other_options
end

options(required_key: 'Done!', foo: 'Foo!', bar: 'Bar!')
#> { :foo => "Foo!", :bar => "Bar!" }
```

Dans l'exemple ci-dessus, si `**other_options` n'est pas utilisé, un `ArgumentError: unknown keyword: foo, bar` erreur de `ArgumentError: unknown keyword: foo, bar` serait `**other_options` .

```
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
```

C'est pratique lorsque vous avez un hachage d'options que vous souhaitez transmettre à une méthode et que vous ne souhaitez pas filtrer les clés.

```
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!" }
```

Il est également possible de *déballer* un hash en utilisant l'opérateur `**` . Cela vous permet de fournir un mot-clé directement à une méthode en plus des valeurs d'autres hachages:

```
my_hash = { foo: 'Foo!', bar: 'Bar!' }

options(required_key: true, **my_hash)
#> { :foo => "Foo!", :bar => "Bar!" }
```

Céder aux blocs

Vous pouvez envoyer un bloc à votre méthode et il peut appeler ce bloc plusieurs fois. Cela peut être fait en envoyant un proc / lambda ou autre, mais c'est plus facile et plus rapide avec le `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
```

Notez que le `{ puts ... }` n'est pas entre parenthèses, il vient implicitement après. Cela signifie également que nous ne pouvons avoir qu'un seul bloc de `yield` . Nous pouvons transmettre des arguments au `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
```

Avec le rendement, nous pouvons facilement créer des itérateurs ou des fonctions fonctionnant sur d'autres codes:

```
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
```

En fait, c'est avec le `yield` que des choses comme `foreach` , `each` et les `times` sont généralement implémentées dans les classes.

Si vous voulez savoir si un bloc vous a été donné ou non, utilisez `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
```

Cet exemple suppose que la classe `Employees` possède une liste `@employees` pouvant être itérée avec `each` pour obtenir des objets ayant des noms d'employés à l'aide de la méthode `name` . Si on nous donne un bloc, alors nous `yield` le nom au bloc, sinon nous le pousserons simplement dans un tableau que nous renverrons.

Tuple Arguments

Une méthode peut prendre un paramètre de tableau et le déstructurer immédiatement en variables locales nommées. Trouvé sur le 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"
```

Définir une méthode

Les méthodes sont définies avec le mot-clé `def` , suivi du *nom* de la *méthode* et d'une liste facultative de *noms* de *paramètre* entre parenthèses. Le code Ruby entre `def` et `end` représente le

corps de la méthode.

```
def hello(name)
  "Hello, #{name}"
end
```

Un appel de méthode spécifie le nom de la méthode, l'objet sur lequel il doit être appelé (parfois appelé récepteur) et zéro ou plusieurs valeurs d'argument affectées aux paramètres de la méthode nommée.

```
hello("World")
# => "Hello, World"
```

Lorsque le récepteur n'est pas explicite, il est `self` .

Les noms de paramètre peuvent être utilisés comme variables dans le corps de la méthode et les valeurs de ces paramètres nommés proviennent des arguments d'une invocation de méthode.

```
hello("World")
# => "Hello, World"
hello("All")
# => "Hello, All"
```

Utiliser une fonction comme un bloc

De nombreuses fonctions de Ruby acceptent un bloc comme argument. Par exemple:

```
[0, 1, 2].map {|i| i + 1}
 => [1, 2, 3]
```

Si vous avez déjà une fonction qui fait ce que vous voulez, vous pouvez la transformer en un bloc en utilisant `&method(:fn)` :

```
def inc(num)
   num + 1
end

[0, 1, 2].map &method(:inc)
 => [1, 2, 3]
```

Lire Les méthodes en ligne: https://riptutorial.com/fr/ruby/topic/997/les-methodes

Chapitre 44: Les opérateurs

Remarques

Les opérateurs sont des méthodes

La plupart des opérateurs ne sont en fait que des méthodes, donc `x + y` appelle la méthode `+` de `x` avec l'argument `y` , qui serait écrit `x.+(y)` . Si vous écrivez une méthode qui a une signification sémantique pour un opérateur donné, vous pouvez implémenter votre variante dans la classe.

Comme un exemple stupide:

```
# 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
```

Quand utiliser `&&` contre `and` , `||` vs `or`

Notez qu'il y a deux façons d'exprimer les booléens, soit `&&` ou `and` , et `||` ou `or` - ils sont souvent interchangeables, mais pas toujours. Nous nous référerons à ceux-ci en tant que variantes "caractère" et "mot".

Les variantes de caractères ont une *priorité* plus élevée, ce qui évite les erreurs inattendues.

Les variantes de mot étaient à l'origine destinées aux *opérateurs de flux de contrôle* plutôt qu'aux opérateurs booléens. Autrement dit, ils ont été conçus pour être utilisés dans des instructions de méthode chaînées:

```
raise 'an error' and return
```

Bien qu'ils *puissent* être utilisés comme opérateurs booléens, leur priorité inférieure les rend imprévisibles.

Deuxièmement, beaucoup de rubyistes préfèrent la variante de caractère lors de la création d'une expression booléenne (celle qui est évaluée comme `true` ou `false`) telle que `x.nil? || x.empty?` . En revanche, les variantes de mots sont préférables dans les cas où une *série de méthodes* sont en cours d'évaluation, et une autre peut échouer. Par exemple, un idiome commun utilisant la variante de mot pour les méthodes renvoyant `nil` en cas d'échec pourrait ressembler à ceci:

```
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

Priorité et méthodes de l'opérateur

Du plus haut au plus bas, c'est le tableau de priorité pour Ruby. Les opérations de haute priorité ont lieu avant les opérations de faible priorité.

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 + et unary - sont pour `+obj` , `-obj` ou `-(some_expression)` .

Modifier-if, modifier-sauf, etc. sont pour les versions modificateur de ces mots-clés. Par exemple, il s'agit d'un modificateur sauf si l'expression:

```
a += 1 unless a.zero?
```

Les opérateurs avec un ✓ peuvent être définis comme des méthodes. La plupart des méthodes sont nommées exactement comme l'opérateur est nommé, par exemple:

```
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] Les méthodes Bracket Lookup et Bracket Set (`[]` et `[]=`) ont leurs arguments définis après le nom, par exemple:

```
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] Les opérateurs "unaire plus" et "unaire moins" sont définis comme des méthodes nommées `+@` et `-@` , par exemple

```
class Foo
  def -@
    puts "unary minus"
  end
  def +@
    puts "unary plus"
  end
end

f = Foo.new
+f               #=> "unary plus"
-f               #=> "unary minus"
```

[3] Dans les premières versions de Ruby, l'opérateur d'inégalité `!=` Et l'opérateur sans correspondance `!~` Ne pouvaient pas être définis comme des méthodes. Au lieu de cela, la méthode pour l'opérateur d'égalité correspondant `==` ou l'opérateur correspondant `=~` été invoquée, et le résultat de cette méthode a été booléen inversé par Ruby.

Si vous ne définissez pas vos propres opérateurs `!=` Ou `!~` , Le comportement ci-dessus est toujours vrai. Cependant, à partir de Ruby 1.9.1, ces deux opérateurs peuvent également être définis comme des méthodes:

```
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"
```

Opérateur d'égalité de cas (===)

Aussi connu sous le nom de *triple equals* .

Cet opérateur ne teste pas l'égalité, mais teste plutôt si l'opérande droit a une relation IS A avec l'opérande gauche. En tant que tel, l' *opérateur d'égalité de* noms populaire est trompeur.

Cette réponse SO le décrit ainsi: la meilleure façon de décrire `a === b` est "si j'ai un tiroir étiqueté `a` , est-il judicieux d'y mettre `b` ?" En d'autres termes, l'ensemble `a` inclut-il le membre `b` ?

Exemples (source)

```
(1..5) === 3            # => true
(1..5) === 6            # => false

Integer === 42          # => true
Integer === 'fourtytwo' # => false

/ell/ === 'Hello'       # => true
/ell/ === 'Foobar'      # => false
```

Classes qui remplacent `===`

De nombreuses classes remplacent `===` pour fournir une sémantique significative dans les instructions de cas. Certains d'entre eux sont:

```
╔═══════════════════╦═══════════════════════╗
║      Class        ║     Synonym for       ║
╠═══════════════════╬═══════════════════════╣
║ Array             ║ ==                    ║
║                   ║                       ║
║ Date              ║ ==                    ║
║                   ║                       ║
║ Module            ║ is_a?                 ║
```

```
╟                  ╫                      ╢
╟ Object           ╫ ==                   ╢
╟                  ╫                      ╢
╟ Range            ╫ include?             ╢
╟                  ╫                      ╢
╟ Regexp           ╫ =~                   ╢
╟                  ╫                      ╢
╟ String           ╫ ==                   ╢
╚══════════════════╩══════════════════════╝
```

Pratique recommandée

L'utilisation explicite de l'opérateur d'égalité de casse `===` doit être évitée. Il ne teste pas l'égalité, mais plutôt la *subsomption* , et son utilisation peut être source de confusion. Le code est plus clair et plus facile à comprendre lorsque la méthode synonyme est utilisée à la place.

```
# Bad
Integer === 42
(1..5) === 3
/ell/ === 'Hello'

# Good, uses synonym method
42.is_a?(Integer)
(1..5).include?(3)
/ell/ =~ 'Hello'
```

Opérateur de navigation sécurisé

Ruby 2.3.0 a ajouté l' *opérateur de navigation sécurisé* , `&.` . Cet opérateur est destiné à raccourcir le paradigme de l' `object && object.property && object.property.method` dans les instructions conditionnelles.

Par exemple, vous avez un objet `House` avec une propriété `address` et vous voulez trouver le `street_name` partir de l' `address` . Pour programmer cela en toute sécurité afin d'éviter les erreurs nuls dans les anciennes versions de Ruby, vous utiliseriez le code suivant:

```
if house && house.address && house.address.street_name
  house.address.street_name
end
```

L'opérateur de navigation sécurisé réduit cette condition. Au lieu de cela, vous pouvez écrire:

```
if house&.address&.street_name
  house.address.street_name
end
```

Mise en garde:
L'opérateur de navigation sécurisé n'a pas *exactement* le même comportement que le conditionnel chaîné. En utilisant le conditionnel chaîné (premier exemple), le bloc `if` ne serait pas exécuté si, par exemple, l' `address` était `false` . L'opérateur de navigation sécurisé ne reconnaît que des valeurs `nil` , mais autorise des valeurs telles que `false` . Si l' `address` est `false` , l'utilisation du SNO

entraînera une erreur:

```
house&.address&.street_name
# => undefined method `address' for false:FalseClass
```

Lire Les opérateurs en ligne: https://riptutorial.com/fr/ruby/topic/3764/les-operateurs

Chapitre 45: Les opérateurs

Examples

Opérateurs de comparaison

Opérateur	La description
`==`	`true` si les deux valeurs sont égales.
`!=`	`true` si les deux valeurs *ne* sont *pas* égales.
`<`	`true` si la valeur de l'opérande de gauche est *inférieure à* la valeur de droite.
`>`	`true` si la valeur de l'opérande de gauche est *supérieure à* la valeur de droite.
`>=`	`true` si la valeur de l'opérande de gauche est *supérieure* **ou** *égale à* la valeur de droite.
`<=`	`true` si la valeur de l'opérande de gauche est *inférieure* **ou** *égale à* la valeur de droite.
`<=>`	`0` si la valeur de l'opérande à gauche est *égale à* la valeur à droite, `1` si la valeur de l'opérande à gauche est *supérieure à* la valeur à droite, `-1` si la valeur de l'opérande de gauche est *inférieure à* la valeur de droite.

Opérateurs d'affectation

Affectation simple

`=` est une affectation simple. Il crée une nouvelle variable locale si la variable n'a pas déjà été référencée.

```
x = 3
y = 4 + 5
puts "x is #{x}, y is #{y}"
```

Cela va sortir:

```
x is 3, y is 9
```

Affectation parallèle

Des variables peuvent également être affectées en parallèle, par exemple `x, y = 3, 9`. Ceci est

particulièrement utile pour échanger des valeurs:

```
x, y = 3, 9
x, y = y, x
puts "x is #{x}, y is #{y}"
```

Cela va sortir:

```
x is 9, y is 3
```

Affectation abrégée

Il est possible de mélanger les opérateurs et l'affectation. Par exemple:

```
x = 1
y = 2
puts "x is #{x}, y is #{y}"

x += y
puts "x is now #{x}"
```

Affiche la sortie suivante:

```
x is 1, y is 2
x is now 3
```

Diverses opérations peuvent être utilisées en affectation abrégée:

Opérateur	La description	Exemple	Équivalent à
`+=`	Ajoute et réaffecte la variable	`x += y`	`x = x + y`
`-=`	Soustrait et réaffecte la variable	`x -= y`	`x = x - y`
`*=`	Multiplie et réaffecte la variable	`x *= y`	`x = x * y`
`/=`	Divise et réaffecte la variable	`x /= y`	`x = x / y`
`%=`	Divise, prend le reste et réaffecte la variable	`x %= y`	`x = x % y`
`**=`	Calcule l'exposant et réaffecte la variable	`x **= y`	`x = x ** y`

Lire Les opérateurs en ligne: https://riptutorial.com/fr/ruby/topic/3766/les-operateurs

Chapitre 46: Message passant

Examples

introduction

Dans la *conception orientée objet* , les objets *reçoivent des* messages et y *répondent* . En Ruby, l'envoi d'un message *appelle une méthode* et le résultat de cette méthode est la réponse.

Dans Ruby, le passage de messages est dynamique. Lorsqu'un message arrive plutôt que de savoir exactement comment y répondre, Ruby utilise un ensemble de règles prédéfini pour trouver une méthode capable d'y répondre. Nous pouvons utiliser ces règles pour interrompre et répondre au message, l'envoyer à un autre objet ou le modifier parmi d'autres actions.

Chaque fois qu'un objet reçoit un message, Ruby vérifie:

1. Si cet objet a une classe singleton et qu'il peut répondre à ce message.
2. Recherche la classe de cet objet puis la chaîne des ancêtres de la classe.
3. Un par un vérifie si une méthode est disponible sur cet ancêtre et remonte la chaîne.

Message passant par la chaîne d'héritage

```
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
```

Pour trouver une méthode adaptée à `SubExample#subexample_method` Ruby examine d'abord la chaîne d'ancêtres de `SubExample`

```
SubExample.ancestors # => [SubExample, Example, Object, Kernel, BasicObject]
```

Cela commence par `SubExample` . Si nous envoyons `subexample_method` message sous-`subexample_method` , Ruby choisit celui qui est disponible sous SubExample et ignore `Example#subexample_method` .

```
s.subexample_method # => :subexample
```

Après `SubExample` il vérifie `Example` . Si nous envoyons `example_method` Ruby vérifie si `SubExample` peut y répondre ou non, et comme Ruby ne peut pas monter la chaîne et examine `Example` .

```
s.example_method # => :example
```

Après que Ruby ait vérifié toutes les méthodes définies, il exécute `method_missing` pour voir s'il peut répondre ou non. Si nous envoyons `missing_subexample_method` Ruby ne sera pas en mesure de trouver une méthode définie dans `SubExample` afin de passer à `Example` . Il ne peut pas non plus trouver une méthode définie sur `Example` ou toute autre classe supérieure dans la chaîne. Ruby recommence et exécute `method_missing` . `method_missing` de `SubExample` peut répondre à `missing_subexample_method` .

```
s.missing_subexample_method # => :subexample
```

Cependant, si une méthode est définie, Ruby utilise une version définie même si elle est plus élevée dans la chaîne. Par exemple, si nous envoyons `not_missed_method` même si `method_missing` de `SubExample` peut y répondre, Ruby se place sur `SubExample` car il ne possède pas de méthode définie avec ce nom et examine `Example` qui en a un.

```
s.not_missed_method # => :example
```

Message passant par la composition du module

Ruby monte sur la chaîne des ancêtres d'un objet. Cette chaîne peut contenir à la fois des modules et des classes. Les mêmes règles concernant le déplacement de la chaîne s’appliquent également aux modules.

```
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
```

Interruption des messages

Il existe deux manières d'interrompre les messages.

- Utilisez `method_missing` pour interrompre tout message non défini.
- Définir une méthode en milieu de chaîne pour intercepter le message

Après avoir interrompu les messages, il est possible de:

- Leur réponds.
- Envoyez-les ailleurs.
- Modifier le message ou son résultat.

Interrompre via `method_missing` et répondre au message:

```
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
```

Interception du message et modification:

```
class Example
  def initialize title, body
  end
end

class SubExample < Example
end
```

Imaginons maintenant que nos données sont "title: body" et que nous devons les séparer avant d'appeler `Example` . Nous pouvons définir l' `initialize` sur `SubExample` .

```
class SubExample < Example
  def initialize raw_data
    processed_data = raw_data.split ":"

    super processed_data[0], processed_data[1]
  end
end
```

Intercepter un message et l'envoyer à un autre objet:

```
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
```

Lire Message passant en ligne: https://riptutorial.com/fr/ruby/topic/5083/message-passant

Chapitre 47: Métaprogrammation

Introduction

La métaprogrammation peut être décrite de deux manières:

"Les programmes informatiques qui écrivent ou manipulent d'autres programmes (ou eux-mêmes) en tant que leurs données, ou qui effectuent une partie du travail au moment de la compilation qui serait effectuée au moment de l'exécution".

Plus simplement: la **métaprogrammation consiste à écrire du code qui écrit du code pendant l'exécution pour vous simplifier la vie** .

Examples

Implémenter "avec" en utilisant l'évaluation d'instance

De nombreux langages disposent d'une instruction `with` qui permet aux programmeurs d'omettre le récepteur d'appels de méthode.

`with` peut être facilement émulé dans Ruby en utilisant `instance_eval` :

```
def with(object, &block)
  object.instance_eval &block
end
```

La méthode `with` peut être utilisée pour exécuter de manière transparente des méthodes sur des objets:

```
hash = Hash.new

with hash do
  store :key, :value
  has_key? :key       # => true
  values              # => [:value]
end
```

Définition dynamique de méthodes

Avec Ruby, vous pouvez modifier la structure du programme en temps d'exécution. Une façon de le faire est de définir des méthodes dynamiquement en utilisant la méthode `method_missing` .

Disons que nous voulons pouvoir tester si un nombre est supérieur à un autre nombre avec la syntaxe `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
```

Une chose importante à retenir lors de l'utilisation de `method_missing` que vous `method_missing` également remplacer `respond_to?` méthode:

```
class Numeric
   def respond_to?(method_name, include_all = false)
     method_name.to_s.match(/^is_greater_than_(\d+)\?$/) || super
   end
end
```

Oublier de le faire conduit à une situation incohérente lorsque vous pouvez appeler `600.is_greater_than_123` avec succès, mais que `600.respond_to(:is_greater_than_123)` renvoie false.

Définir des méthodes sur des instances

Dans Ruby, vous pouvez ajouter des méthodes aux instances existantes de n'importe quelle classe. Cela vous permet d'ajouter un comportement et une instance à une classe sans modifier le comportement des autres instances de cette classe.

```
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}
```

méthode send ()

`send()` est utilisé pour transmettre un message à l' `object` . `send()` est une méthode d'instance de la classe `Object` . Le premier argument dans `send()` est le message que vous envoyez à l'objet, c'est-à-dire le nom d'une méthode. Il peut s'agir d'une `string` ou d'un `symbol` mais les **symboles** sont préférés. Ensuite, les arguments doivent passer en méthode, ce seront les arguments restants

dans `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.
```

Voici l'exemple plus descriptif

```
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
```

Remarque: `send()` n'est plus recommandé. Utilisez `__send__()` qui a le pouvoir d'appeler des méthodes privées, ou (recommandé) `public_send()`

Lire Métaprogrammation en ligne: https://riptutorial.com/fr/ruby/topic/5023/metaprogrammation

Chapitre 48: method_missing

Paramètres

Paramètre	Détails
méthode	Le nom de la méthode qui a été appelée (dans l'exemple ci-dessus, c'est `:say_moo` , notez qu'il s'agit d'un symbole.
* args	Les arguments transmis à cette méthode. Peut être n'importe quel nombre, ou aucun
&bloc	Le bloc de la méthode appelée, il peut s'agir d'un bloc `do` ou d'un bloc `{ }`

Remarques

Appelez toujours super, au bas de cette fonction. Cela évite les pannes silencieuses lorsque quelque chose est appelé et que vous n'obtenez pas d'erreur.

Par exemple, cette méthode_missing va causer des problèmes:

```
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` est un bon outil à utiliser lorsque cela est approprié, mais vous devez tenir compte de deux coûts. Premièrement, `method_missing` est moins efficace - ruby doit rechercher la classe et tous ses ancêtres avant de pouvoir `method_missing` cette approche; Cette pénalité de performance peut être triviale dans un cas simple, mais peut s'ajouter. Deuxièmement et plus largement, il s'agit d'une forme de méta-programmation dotée d'une grande puissance qui implique la responsabilité de garantir la sécurité de l'implémentation, de gérer correctement les entrées malveillantes, les entrées inattendues, etc.

Vous devriez également écraser `respond_to_missing?` ainsi:

```
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

Attraper des appels à une méthode non définie

```
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"
```

Utiliser la méthode manquante

```
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'
```

Utiliser avec bloc

```
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"
```

Utiliser avec paramètre

```
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"
```

Lire method_missing en ligne: https://riptutorial.com/fr/ruby/topic/1076/method-missing

Chapitre 49: Modificateurs d'accès Ruby

Introduction

Contrôle d'accès (portée) aux différentes méthodes, membres de données, méthodes d'initialisation.

Examples

Variables d'instance et variables de classe

Relevons d'abord quelles sont les **variables d'instance:** elles se comportent plus comme des propriétés pour un objet. Ils sont initialisés sur une création d'objet. Les variables d'instance sont accessibles via des méthodes d'instance. Par objet a des variables par instance. Les variables d'instance ne sont pas partagées entre les objets.

La classe de séquence a les variables d'instance @from, @to et @by.

```
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 classe Traiter la variable de classe de la même manière que les variables statiques de Java, qui sont partagées entre les différents objets de cette classe. Les variables de classe sont stockées dans la mémoire de tas.

```
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
```

Partagé entre objet et objet1.

En comparant les variables d'instance et de classe de Ruby contre 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
    }
}
```

Contrôles d'accès

Comparaison des contrôles d'accès de Java avec Ruby: Si la méthode est déclarée privée en Java, elle ne peut être accédée que par d'autres méthodes de la même classe. Si une méthode est déclarée protégée, elle peut être accédée par d'autres classes qui existent dans le même package, ainsi que par des sous-classes de la classe dans un package différent. Lorsqu'une méthode est publique, elle est visible par tous. En Java, le concept de visibilité du contrôle d'accès dépend de l'emplacement de ces classes dans la hiérarchie héritage / package.

Alors que dans Ruby, la hiérarchie d'héritage ou le package / module ne correspond pas. Il s'agit de quel objet est le récepteur d'une méthode .

Pour une méthode privée dans Ruby , il ne peut jamais être appelé avec un récepteur explicite. Nous pouvons (seulement) appeler la méthode privée avec un récepteur implicite.

Cela signifie également que nous pouvons appeler une méthode privée depuis une classe dans laquelle elle est déclarée, ainsi que toutes les sous-classes de cette classe.

```
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.
```

La méthode protégée peut être appelée avec un récepteur implicite, comme si elle était privée. De plus, la méthode protégée peut également être appelée par un récepteur explicite (uniquement) si le destinataire est "self" ou "un objet de la même classe".

```
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
```

Envisager des méthodes publiques avec une visibilité maximale

Résumé

1. **Public:** Les méthodes publiques ont une visibilité maximale
2. **Protected: la méthode protégée** peut être appelée avec un récepteur implicite, comme si c'était privé. De plus, la méthode protégée peut également être appelée par un récepteur explicite (uniquement) si le destinataire est "self" ou "un objet de la même classe".
3. **Privé: pour une méthode privée dans Ruby** , il ne peut jamais être appelé avec un récepteur explicite. Nous pouvons (seulement) appeler la méthode privée avec un récepteur implicite. Cela signifie également que nous pouvons appeler une méthode privée depuis une classe dans laquelle elle est déclarée, ainsi que toutes les sous-classes de cette classe.

Lire Modificateurs d'accès Ruby en ligne: https://riptutorial.com/fr/ruby/topic/10797/modificateurs-d-acces-ruby

Chapitre 50: Modules

Syntaxe

- Déclaration

```
module Name;

    any ruby expressions;

end
```

Remarques

Les noms de module dans Ruby sont des constantes, ils doivent donc commencer par une majuscule.

```
module foo; end # Syntax error: class/module name must be CONSTANT
```

Examples

Un mixin simple avec include

```
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
```

Maintenant, `Bar` est un mélange de ses propres méthodes et des méthodes de `SomeMixin` .

Notez que la façon dont un mixin est utilisé dans une classe dépend de la manière dont il est ajouté:

- le mot `include` clé `include` évalue le code du module dans le contexte de la classe (par exemple, les définitions de méthode seront des méthodes sur les instances de la classe),

- `extend` évalue le code du module dans le contexte de la classe singleton de l'objet (les méthodes sont disponibles directement sur l'objet étendu).

Module comme espace de noms

Les modules peuvent contenir d'autres modules et classes:

```
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
```

Un mixin simple avec extension

Un mixin est juste un module qui peut être ajouté à une classe. une façon de le faire est avec la méthode d'extension. La méthode `extend` ajoute des méthodes de mixin en tant que méthodes de classe.

```
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
```

Composition des modules et des classes

Vous pouvez utiliser des modules pour créer des classes plus complexes via la *composition* . La directive `include ModuleName` incorpore les méthodes d'un module dans une classe.

```
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` contient désormais des méthodes provenant de `Foo` et de `Bar` en plus de ses propres méthodes.

```
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!'
```

Lire Modules en ligne: https://riptutorial.com/fr/ruby/topic/4039/modules

Chapitre 51: Nombres

Remarques

Hiérarchie des nombres

Ruby comprend plusieurs classes intégrées pour représenter les nombres:

```
Numeric
  Integer
    Fixnum    # 1
    Bignum    # 10000000000000000000
  Float       # 1.0
  Complex     # (1+0i)
  Rational    # Rational(2, 3) == 2/3
  BigDecimal  # not loaded by default
```

Les plus courants sont:

- `Fixnum` pour représenter, par exemple, les entiers positifs et négatifs
- `Float` pour représenter des nombres à virgule flottante

`BigDecimal` est le seul non chargé par défaut. Vous pouvez le charger avec:

```
require "bigdecimal"
```

Notez que dans Ruby 2.4+, `Fixnum` et `Bignum` sont unifiés; *tous les* entiers ne sont plus que des membres de la classe `Integer` . Pour la compatibilité ascendante, `Fixnum == Bignum == Integer` .

Examples

Créer un entier

```
0       # creates the Fixnum 0
123     # creates the Fixnum 123
1_000   # creates the Fixnum 1000. You can use _ as separator for readability
```

Par défaut, la notation est la base 10. Cependant, il existe d'autres notations intégrées pour différentes 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
```

Conversion d'une chaîne en entier

Vous pouvez utiliser la méthode `Integer` pour convertir une `String` en un `Integer` :

```
Integer("123")      # => 123
Integer("0xFF")     # => 255
Integer("0b100")    # => 4
Integer("0555")     # => 365
```

Vous pouvez également passer un paramètre de base à la méthode `Integer` pour convertir des nombres à partir d'une certaine base

```
Integer('10', 5)    # => 5
Integer('74', 8)    # => 60
Integer('NUM', 36)  # => 30910
```

Notez que la méthode déclenche une `ArgumentError` si le paramètre ne peut pas être converti:

```
Integer("hello")
# raises ArgumentError: invalid value for Integer(): "hello"
Integer("23-hello")
# raises ArgumentError: invalid value for Integer(): "23-hello"
```

Vous pouvez également utiliser la méthode `String#to_i` . Cependant, cette méthode est légèrement plus permissive et a un comportement différent de `Integer` :

```
"23".to_i         # => 23
"23-hello".to_i   # => 23
"hello".to_i      # => 0
```

`String#to_i` accepte un argument, la base pour interpréter le nombre comme `String#to_i` :

```
"10".to_i(2) # => 2
"10".to_i(3) # => 3
"A".to_i(16) # => 10
```

Conversion d'un nombre en chaîne

Fixnum # to_s prend un argument de base facultatif et représente le nombre donné dans cette base:

```
2.to_s(2)   # => "10"
3.to_s(2)   # => "11"
3.to_s(3)   # => "10"
10.to_s(16) # => "a"
```

Si aucun argument n'est fourni, alors il représente le nombre en base 10

```
2.to_s # => "2"
10423.to_s # => "10423"
```

Division de deux nombres

Lorsque vous divisez deux nombres, faites attention au type que vous voulez en retour. Notez que la division de **deux entiers appellera la division entière** . Si votre objectif est d'exécuter la division float, au moins un des paramètres doit être de type `float` .

Division entière:

```
3 / 2 # => 1
```

Division flottante

```
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
```

Nombres rationnels

`Rational` représente un nombre rationnel comme numérateur et dénominateur:

```
r1 = Rational(2, 3)
r2 = 2.5.to_r
r3 = r1 + r2
r3.numerator   # => 19
r3.denominator # => 6
Rational(2, 4) # => (1/2)
```

Autres façons de créer un Rational

```
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)
```

Nombres complexes

```
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)
```

Numéros pairs et impairs

Le `even?` méthode peut être utilisée pour déterminer si un nombre est pair

```
4.even?      # => true
5.even?      # => false
```

L' `odd?` méthode peut être utilisée pour déterminer si un nombre est impair

```
4.odd?       # => false
5.odd?       # => true
```

Chiffres d'arrondi

La méthode `round` arrondit un nombre supérieur si le premier chiffre après sa décimale est supérieur ou égal à 5 et s'élève si ce chiffre est inférieur ou égal à 4. Cela prend un argument optionnel pour la précision que vous recherchez.

```
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
```

Les nombres à virgule flottante peuvent également être arrondis à l'entier inférieur inférieur au nombre avec la méthode d' `floor`

```
4.9999999999999.floor # => 4
```

Ils peuvent également être arrondis au nombre entier le plus faible supérieur au nombre utilisant la méthode `ceil`

```
4.0000000000001.ceil  # => 5
```

Lire Nombres en ligne: https://riptutorial.com/fr/ruby/topic/1083/nombres

Chapitre 52: Opérateur Splat (*)

Examples

Tableaux de contrainte dans la liste de paramètres

Supposons que vous ayez un tableau:

```
pair = ['Jack','Jill']
```

Et une méthode qui prend deux arguments:

```
def print_pair (a, b)
  puts "#{a} and #{b} are a good couple!"
end
```

Vous pourriez penser que vous pourriez simplement passer le tableau:

```
print_pair(pair) # wrong number of arguments (1 for 2) (ArgumentError)
```

Puisque le tableau n'est qu'un argument, pas deux, Ruby lance une exception. Vous *pouvez* retirer chaque élément individuellement:

```
print_pair(pair[0], pair[1])
```

Ou vous pouvez utiliser l'opérateur splat pour vous épargner des efforts:

```
print_pair(*pair)
```

Nombre variable d'arguments

L'opérateur splat supprime des éléments individuels d'un tableau et les transforme en une liste. Ceci est le plus couramment utilisé pour créer une méthode qui accepte un nombre variable d'arguments:

```
# 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')
```

Notez qu'un tableau ne compte que pour un élément de la liste. Vous devez donc nous aussi appeler l'opérateur splat du côté appel si vous souhaitez transmettre un tableau:

```
bonaparte = ['Napoleon','Joséphine','Marie Louise']
print_spouses(*bonaparte)
```

Lire Opérateur Splat (*) en ligne: https://riptutorial.com/fr/ruby/topic/9862/operateur-splat----

Chapitre 53: Opérations sur les fichiers et les E / S

Paramètres

Drapeau	Sens
"r"	Lecture seule, commence au début du fichier (mode par défaut).
"r +"	Lecture-écriture, commence au début du fichier.
"w"	Write-only, tronque le fichier existant à la longueur zéro ou crée un nouveau fichier pour l'écriture.
"w +"	Lecture-écriture, tronque le fichier existant à une longueur nulle ou crée un nouveau fichier pour la lecture et l'écriture.
"une"	Écriture seule, commence à la fin du fichier si le fichier existe, sinon crée un nouveau fichier pour l'écriture.
"a +"	Lecture-écriture, commence à la fin du fichier si le fichier existe, sinon crée un nouveau fichier pour la lecture et l'écriture.
"b"	Mode fichier binaire. Supprime la conversion EOL <-> CRLF sous Windows. Et définit le codage externe sur ASCII-8BIT sauf si explicitement spécifié. (Cet indicateur ne peut apparaître qu'avec les indicateurs ci-dessus. Par exemple, `File.new("test.txt", "rb")` ouvrirait `test.txt` en mode `read-only` en tant que fichier `binary` .)
"t"	Mode fichier texte. (Cet indicateur ne peut apparaître qu'avec les indicateurs ci-dessus. Par exemple, `File.new("test.txt", "wt")` ouvrirait `test.txt` en mode `write-only` en tant que fichier `text` .)

Examples

Ecrire une chaîne dans un fichier

Une chaîne peut être écrite dans un fichier avec une instance de la classe `File` .

```
file = File.new('tmp.txt', 'w')
file.write("NaNaNaNa\n")
file.write('Batman!\n')
file.close
```

La classe `File` offre également un raccourci pour les opérations `new` et `close` avec la méthode `open` .

```
File.open('tmp.txt', 'w') do |f|
  f.write("NaNaNaNa\n")
  f.write('Batman!\n')
end
```

Pour les opérations d'écriture simples, une chaîne peut également être écrite directement dans un fichier avec `File.write` . **Notez que cela écrasera le fichier par défaut.**

```
File.write('tmp.txt', "NaNaNaNa\n" * 4 + 'Batman!\n')
```

Pour spécifier un mode différent sur `File.write` , transmettez-le comme valeur d'un `mode` appelé clé dans un hachage comme autre paramètre.

```
File.write('tmp.txt', "NaNaNaNa\n" * 4 + 'Batman!\n', { mode: 'a'})
```

Ouvrir et fermer un fichier

Ouvrez et fermez manuellement un fichier.

```
# 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
```

Ferme automatiquement un fichier en utilisant un bloc.

```
f = File.open("test.txt", "r") do |f|
  # do something with file f
  puts f.read # for example, read it
end
```

obtenir un seul caractère d'entrée

Contrairement à `gets.chomp` cela n'attendra pas une nouvelle ligne.

La première partie du stdlib doit être incluse

```
require 'io/console'
```

Ensuite, une méthode d'assistance peut être écrite:

```
def get_char
  input = STDIN.getch
  control_c_code = "\u0003"
```

```
  exit(1) if input == control_c_code
  input
end
```

Il est important de sortir si `control+c` bouton `control+c` est enfoncé.

Lecture 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
```

Lecture des arguments avec ARGV

```
number1 = ARGV[0]
number2 = ARGV[1]
puts number1.to_i + number2.to_i
## run with: $ ruby a_plus_b.rb 1 2
```

Lire Opérations sur les fichiers et les E / S en ligne:
https://riptutorial.com/fr/ruby/topic/4310/operations-sur-les-fichiers-et-les-e---s

Chapitre 54: OptionParser

Introduction

OptionParser peut être utilisé pour analyser les options de ligne de commande à partir d' `ARGV` .

Examples

Options de ligne de commande obligatoires et facultatives

Il est relativement facile d'analyser la ligne de commande à la main si vous ne recherchez rien de trop complexe:

```
# 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
```

Mais lorsque vos options commenceront à se compliquer, vous devrez probablement utiliser un analyseur d'options tel que 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!
```

Il y a aussi une `parse` non destructive, mais c'est beaucoup moins utile si vous prévoyez d'utiliser le reste de ce que `ARGV` .

La classe OptionParser ne dispose d'aucun moyen pour appliquer des arguments obligatoires (tels que `--site` dans ce cas). Cependant, vous pouvez faire votre propre vérification après avoir exécuté l' `parse!` :

```
# Slightly more sophisticated error checking
if options[:site].nil? or ARGV.length == 0
  abort(optparse.help)
end
```

Pour un gestionnaire d'options obligatoire plus générique, consultez cette réponse . Au cas où ce ne serait pas clair, toutes les options sont facultatives, à moins que vous ne fassiez le maximum pour les rendre obligatoires.

Les valeurs par défaut

Avec `OptionsParser` , il est très facile de définir des valeurs par défaut. Il suffit de pré-remplir le hachage pour stocker les options dans:

```
options = {
  :directory => ENV['HOME']
}
```

Lorsque vous définissez l'analyseur, il remplace le paramètre par défaut si un utilisateur fournit une valeur:

```
OptionParser.new do |opts|
  opts.on("-d", "--directory HOME", "Directory to use") do |d|
    options[:directory] = d
  end
end
```

Longues descriptions

Parfois, votre description peut être assez longue. Par exemple `irb -h` répertorie l'argument qui se lit comme suit:

```
  --context-mode n  Set n[0-3] to method to create Binding Object,
                    when new workspace was created
```

Ce n'est pas immédiatement clair comment soutenir cela. La plupart des solutions nécessitent un ajustement pour que l'indentation des deuxièmes lignes et des lignes suivantes s'aligne sur la première. Heureusement, la `on` méthode prend en charge de multiples lignes de description en les ajoutant comme arguments séparés:

```
  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
```

Vous pouvez ajouter autant de lignes de description que vous le souhaitez pour expliquer pleinement l’option.

Lire OptionParser en ligne: https://riptutorial.com/fr/ruby/topic/9860/optionparser

Chapitre 55: Portée et visibilité variables

Syntaxe

- $ global_variable
- @@ class_variable
- @instance_variable
- variable locale

Remarques

Les variables de classe sont partagées dans la hiérarchie des classes. Cela peut entraîner un comportement surprenant.

```
class A
  @@variable = :x

  def self.variable
    @@variable
  end
end

class B < A
  @@variable = :y
end

A.variable  # :y
```

Les classes sont des objets, donc les variables d'instance peuvent être utilisées pour fournir un état spécifique à chaque classe.

```
class A
  @variable = :x

  def self.variable
    @variable
  end
end

class B < A
  @variable = :y
end

A.variable  # :x
```

Examples

Variables locales

Les variables locales (contrairement aux autres classes de variables) n'ont pas de préfixe

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

Son étendue dépend de l'endroit où il a été déclaré, il ne peut pas être utilisé en dehors du périmètre "déclaration conteneurs". Par exemple, si une variable locale est déclarée dans une méthode, elle ne peut être utilisée que dans cette méthode.

```
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'
```

Bien sûr, les variables locales ne se limitent pas aux méthodes, en règle générale , vous pouvez dire que, dès que vous déclarez une variable à l' intérieur d' un `do ... end` bloc ou wrapped accolades `{}` , il sera local et à SCOPED le bloc dans lequel il a été déclaré

```
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'
```

Cependant, les variables locales déclarées dans `if` ou les blocs de `case` peuvent être utilisés dans la parent-scope:

```
if true
    usable = "yay"
end

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

Bien que les variables locales ne puissent pas être utilisées en dehors de son bloc de déclaration, elles seront transmises à des blocs:

```
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"]
```

Mais pas aux définitions de méthode / classe / module

```
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'
```

Les variables utilisées pour les arguments de bloc sont (naturellement) locales au bloc, mais occulteront les variables précédemment définies, sans les écraser.

```
overshadowed = "sunlight"

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

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

Variables de classe

Les variables de classe ont une portée étendue, elles peuvent être déclarées n'importe où dans la classe. Une variable sera considérée comme une variable de classe avec le préfixe `@@`

```
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"
```

Les variables de classe sont partagées entre les classes associées et peuvent être écrasées à partir d'une classe enfant

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

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

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

Ce comportement est indésirable la plupart du temps et peut être contourné en utilisant des variables d'instance de niveau classe.

Les variables de classe définies dans un module n'écrasent pas leurs variables de classe incluant les classes:

```
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

Les variables globales ont une portée mondiale et peuvent donc être utilisées partout. Leur portée ne dépend pas de l'endroit où ils sont définis. Une variable sera considérée comme globale lorsqu'elle est préfixée par un signe `$` .

```
$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?"
```

Comme une variable globale peut être définie partout et sera visible partout, l'appel d'une variable globale "non définie" renverra zéro au lieu de générer une erreur.

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

Bien que les variables globales soient faciles à utiliser, leur utilisation est fortement déconseillée en faveur des constantes.

Variables d'instance

Les variables d'instance ont une portée étendue à l'objet, elles peuvent être déclarées n'importe où dans l'objet, cependant une variable d'instance déclarée au niveau de la classe ne sera visible que dans l'objet de classe. Une variable sera considérée comme une variable d'instance avec le préfixe `@` . Les variables d’instance permettent de définir et d’obtenir des attributs d’objet et renvoient zéro si elles ne sont pas définies.

```
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"
```

La variable d'instance déclarée au niveau de la classe n'est pas accessible au niveau de l'objet:

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

Cependant, nous avons utilisé la variable d'instance `@base_sound` pour instancier le son quand aucun son n'est transmis à la nouvelle méthode:

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

Les variables d'instance peuvent être déclarées n'importe où dans l'objet, même à l'intérieur d'un bloc:

```
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
```

Les variables d'instance ne sont **pas** partagées entre les instances de la même classe

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

Cela peut être utilisé pour créer des variables de niveau classe, qui ne seront pas écrasées par une classe enfant, car les classes sont également des objets dans 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"
```

Lire Portée et visibilité variables en ligne: https://riptutorial.com/fr/ruby/topic/4094/portee-et-visibilite-variables

Chapitre 56: Queue

Syntaxe

- q = Queue.new
- q.push objet
- q << objet # identique à #push
- q.pop # => objet

Examples

Plusieurs travailleurs un seul évier

Nous voulons collecter des données créées par plusieurs travailleurs.

Nous créons d'abord une file d'attente:

```
sink = Queue.new
```

Ensuite, 16 travailleurs ont tous généré un nombre aléatoire et l'ont poussé dans l'évier:

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

Et pour obtenir les données, convertissez une file d'attente en un tableau:

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

One Source Multiple Workers

Nous voulons traiter les données en parallèle.

Remplissons la source avec quelques données:

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

Ensuite, créez des travailleurs pour traiter les données:

```
(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)
```

Une source - Pipeline of Work - Un évier

Nous souhaitons traiter les données en parallèle et les acheminer sur la ligne à traiter par les autres travailleurs.

Puisque les travailleurs consomment et produisent des données, nous devons créer deux files d'attente:

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

La première vague de travailleurs lit un élément de `first_input_source` , traite l'élément et écrit les résultats dans `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 deuxième vague de travailleurs utilise `first_output_sink` comme source d’entrée et lit, puis écrit dans un autre `first_output_sink` de sortie:

```
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
```

Maintenant `second_output_sink` est le `second_output_sink` , convertissons-le en un tableau:

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

Pousser des données dans une file d'attente - #push

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

- Il n'y a pas de marée haute, les files d'attente peuvent croître à l'infini.
- `#push` ne bloque jamais

Extraction de données d'une file d'attente - #pop

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

- `#pop` bloquera jusqu'à ce que certaines données soient disponibles.
- `#pop` peut être utilisé pour la synchronisation.

Synchronisation - Après un point dans le temps

```
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)
```

Conversion d'une file d'attente en un tableau

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

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

Ou un seul paquebot :

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

Fusion de deux files d'attente

- Pour éviter le blocage à l'infini, la lecture des files d'attente ne devrait pas avoir lieu lorsque la fusion de threads se produit.
- Pour éviter la synchronisation ou l'attente infinie d'une des files d'attente alors que d'autres ont des données, la lecture des files d'attente ne devrait pas avoir lieu sur le même thread.

Commençons par définir et alimenter deux files d'attente:

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

Nous devrions créer une autre file d'attente et y insérer des données provenant d'autres threads:

```
merged = Queue.new

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

Si vous savez que vous pouvez consommer complètement les deux files d'attente (la vitesse de consommation est supérieure à la production, vous ne manquerez pas de RAM), l'approche est plus simple:

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

Lire Queue en ligne: https://riptutorial.com/fr/ruby/topic/4666/queue

Chapitre 57: Raffinements

Remarques

Les raffinements ont une portée lexicale, ce qui signifie qu'ils sont actifs à partir du moment où ils sont activés (avec le mot-clé `using`) jusqu'à ce que le contrôle change. Généralement, le contrôle est modifié à la fin d'un module, d'une classe ou d'un fichier.

Examples

Patch de singe à portée limitée

Le principal problème de la correction des singes est qu'elle pollue la portée globale. Votre code de travail est à la merci de tous les modules que vous utilisez sans marcher sur les pieds les uns des autres. La solution Ruby à cela est des améliorations, qui sont essentiellement des patchs de singe dans une portée limitée.

```
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'
```

Modules à double usage (améliorations ou correctifs globaux)

Il est recommandé de définir les correctifs à l'aide de Refinements, mais il est parfois utile de les charger globalement (par exemple, en développement ou en test).

Supposons, par exemple, que vous souhaitiez démarrer une console, que vous ayez besoin de votre bibliothèque et que vous ayez les méthodes corrigées disponibles dans la portée globale. Vous ne pouvez pas le faire avec des améliorations car l' `using` doit être appelée dans une définition de classe / module. Mais il est possible d'écrire le code de manière à ce qu'il ait un double objectif:

```
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
```

Raffinements dynamiques

Les raffinements ont des limitations spéciales.

`refine` ne peut être utilisé que dans une portée de module, mais peut être programmé en utilisant `send :refine` .

`using` est plus limitée. Il ne peut être appelé que dans une définition de classe / module. Cependant, il peut accepter une variable pointant vers un module et être appelé en boucle.

Un exemple montrant ces concepts:

```
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
```

Puisque `using` est si statique, il est possible de générer un ordre de chargement si les fichiers de raffinement ne sont pas chargés en premier. Un moyen de résoudre ce problème consiste à envelopper la définition de classe / module corrigée dans un processus. Par exemple:

```
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
```

L'appel du proc crée la classe corrigée `Foo::Bar` . Cela peut être retardé jusqu'à ce que tout le code ait été chargé.

Lire Raffinements en ligne: https://riptutorial.com/fr/ruby/topic/6563/raffinements

Chapitre 58: rbenv

Examples

1. Installez et gérez les versions de Ruby avec rbenv

La méthode la plus simple pour installer et gérer différentes versions de Ruby avec rbenv consiste à utiliser le plug-in ruby-build.

D'abord, clonez le dépôt rbenv dans votre répertoire personnel:

```
$ git clone https://github.com/rbenv/rbenv.git ~/.rbenv
```

Ensuite, clonez le plugin ruby-build:

```
$ git clone https://github.com/rbenv/ruby-build.git ~/.rbenv/plugins/ruby-build
```

Assurez-vous que rbenv est initialisé dans votre session shell, en l'ajoutant à votre `.bash_profile` ou `.zshrc` :

```
type rbenv > /dev/null
if [ "$?" = "0" ]; then
    eval "$(rbenv init -)"
fi
```

(Ceci vérifie en premier lieu si `rbenv` est disponible et l'initialise).

Vous devrez probablement redémarrer votre session shell - ou simplement ouvrir une nouvelle fenêtre Terminal.

Remarque: Si vous utilisez OSX, vous devrez également installer les outils de ligne de commande Mac OS avec:

```
$ xcode-select --install
```

Vous pouvez également installer `rbenv` utilisant Homebrew au lieu de construire à partir de la source:

```
$ brew update
$ brew install rbenv
```

Suivez ensuite les instructions données par:

```
$ rbenv init
```

Installez une nouvelle version de Ruby:

Liste des versions disponibles avec:

```
$ rbenv install --list
```

Choisissez une version et installez-la avec:

```
$ rbenv install 2.2.0
```

Marquez la version installée comme version globale - c.-à-d. Celle que votre système utilise par défaut:

```
$ rbenv global 2.2.0
```

Vérifiez votre version globale:

```
$ rbenv global
=> 2.2.0
```

Vous pouvez spécifier une version de projet locale avec:

```
$ rbenv local 2.1.2
=> (Creates a .ruby-version file at the current directory with the specified version)
```

Notes de bas de page:

[1]: Comprendre PATH

Désinstallation d'un Ruby

Il existe deux manières de désinstaller une version particulière de Ruby. Le plus simple est de supprimer simplement le répertoire de `~/.rbenv/versions` :

```
$ rm -rf ~/.rbenv/versions/2.1.0
```

Vous pouvez également utiliser la commande uninstall, qui fait exactement la même chose:

```
$ rbenv uninstall 2.1.0
```

Si cette version est utilisée quelque part, vous devrez mettre à jour votre version globale ou locale. Pour revenir à la version la plus récente de votre chemin (généralement la valeur par défaut fournie par votre système), utilisez:

```
$ rbenv global system
```

Lire rbenv en ligne: https://riptutorial.com/fr/ruby/topic/4096/rbenv

Chapitre 59: Récepteurs implicites et compréhension de soi

Examples

Il y a toujours un récepteur implicite

Dans Ruby, il y a toujours un récepteur implicite pour tous les appels de méthode. Le langage garde une référence au récepteur implicite actuel stocké dans la variable `self` . Certains mots-clés de langage tels que `class` et `module` changeront ce que le `self` indique. Comprendre ces comportements est très utile pour maîtriser la langue.

Par exemple, lorsque vous ouvrez `irb` première fois

```
irb(main):001:0> self
=> main
```

Dans ce cas, l'objet `main` est le récepteur implicite (voir http://stackoverflow.com/a/917842/417872 pour plus d'informations sur `main`).

Vous pouvez définir des méthodes sur le récepteur implicite en utilisant le mot-clé `def` . Par exemple:

```
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"
```

Cela a défini la méthode foo sur l'instance de l'objet principal s'exécutant dans votre repl.

Notez que les variables locales sont recherchées avant les noms de méthode, de sorte que si vous définissez une variable locale avec le même nom, sa référence remplacera la référence de méthode. Suite de l'exemple précédent:

```
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>
```

La `method` méthode peut toujours trouver la méthode `foo` car elle ne vérifie pas les variables

locales, alors que la référence normale `foo` fait.

Les mots clés changent le récepteur implicite

Lorsque vous définissez une classe ou un module, le récepteur implicite devient une référence à la classe elle-même. Par exemple:

```
puts "I am #{self}"
class Example
  puts "I am #{self}"
end
```

L'exécution du code ci-dessus imprimera:

```
"I am main"
"I am Example"
```

Quand utiliser soi-même?

La plupart du code Ruby utilise le récepteur implicite, donc les programmeurs qui sont nouveaux à Ruby sont souvent confus au sujet de quand utiliser `self` - `self` . La réponse pratique est que le `self` est utilisé de deux manières principales:

1. Pour changer le récepteur.

Habituellement, le comportement de `def` dans une classe ou un module consiste à créer des méthodes d'instance. Self peut être utilisé pour définir des méthodes sur la classe à la place.

```
class Foo
  def bar
    1
  end

  def self.bar
    2
  end
end

Foo.new.bar #=> 1
Foo.bar #=> 2
```

2. Désambiguïser le récepteur

Lorsque les variables locales peuvent avoir le même nom qu'une méthode, un récepteur explicite peut être amené à désambiguïser.

Exemples:

```
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
```

L'autre cas courant nécessitant la désambiguïsation implique des méthodes qui aboutissent au signe égal. Par exemple:

```
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
```

Lire Récepteurs implicites et compréhension de soi en ligne:
https://riptutorial.com/fr/ruby/topic/5856/recepteurs-implicites-et-comprehension-de-soi

Chapitre 60: Récursion en Ruby

Examples

Fonction récursive

Commençons par un algorithme simple pour voir comment la récursion pourrait être implémentée dans Ruby.

Une boulangerie a des produits à vendre. Les produits sont en packs. Il dessert les commandes en packs uniquement. L'emballage commence à partir de la taille de l'emballage la plus grande, puis les quantités restantes sont remplies par les emballages suivants.

Par exemple, si une commande de 16 est reçue, la boulangerie alloue 2 du pack 5 et 2 du pack 3. 2 *5* + *2* 3 = 16. Voyons comment cela est implémenté en récursivité. "allocate" est la fonction récursive ici.

```
#!/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}"
```

Le résultat est:

> La combinaison de paquets est: [3, 3, 5, 5]

Récursion de la queue

De nombreux algorithmes récursifs peuvent être exprimés en utilisant une itération. Par exemple, la plus grande fonction de dénominateur commun peut être écrite récursivement :

```
def gdc (x, y)
  return x if y == 0
  return gdc(y, x%y)
end
```

ou itérativement:

```
def gdc_iter (x, y)
  while y != 0 do
    x, y = y, x%y
  end

  return x
end
```

Les deux algorithmes sont équivalents en théorie, mais la version récursive risque de générer une erreur SystemStackError . Cependant, comme la méthode récursive se termine par un appel à elle-même, elle pourrait être optimisée pour éviter un débordement de pile. Une autre façon de le dire: l'algorithme récursif peut générer le même code machine que l'itératif *si* le compilateur sait rechercher l'appel à la méthode récursive à la fin de la méthode. Ruby ne fait pas l'optimisation des appels de queue par défaut, mais vous pouvez l' activer avec :

```
RubyVM::InstructionSequence.compile_option = {
  tailcall_optimization: true,
  trace_instruction: false
}
```

Outre l'activation de l'optimisation des appels, vous devez également désactiver le suivi des instructions. Malheureusement, ces options ne s'appliquent qu'au moment de la compilation, vous devez donc soit `require` la méthode récursive à partir d'un autre fichier, soit `eval` la définition de la méthode:

```
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
```

Enfin, l'appel de retour final doit renvoyer la méthode et *uniquement la méthode* . Cela signifie que vous devrez réécrire la fonction factorielle standard:

```
def fact(x)
  return 1 if x <= 1
  return x*fact(x-1)
end
```

À quelque chose comme:

```
 def fact(x, acc=1)
   return acc if x <= 1
   return fact(x-1, x*acc)
 end
```

Cette version transmet la somme accumulée via un second argument (optionnel) par défaut 1.

Lectures complémentaires: Optimisation de l'appel de queue dans Ruby et Tailin 'Ruby .

Lire Récursion en Ruby en ligne: https://riptutorial.com/fr/ruby/topic/7986/recursion-en-ruby

Chapitre 61: Ruby Version Manager

Examples

Comment créer gemset

Pour créer un gemset, nous devons créer un fichier `.rvmrc` .

Syntaxe:

```
$ rvm --rvmrc --create <ruby-version>@<gemsetname>
```

Exemple:

```
$ rvm --rvmrc --create ruby-2.2.2@myblog
```

La ligne ci-dessus créera un fichier `.rvmrc` dans le répertoire racine de l'application.

Pour obtenir la liste des gemsets disponibles, utilisez la commande suivante:

```
$ rvm list gemsets
```

Installer Ruby avec RVM

Le *Ruby Version Manager* est un outil de ligne de commande permettant d'installer et de gérer simplement différentes versions de Ruby.

- `rvm istall 2.3.1` par exemple installe Ruby version 2.3.1 sur votre machine.
- Avec la `rvm list` vous pouvez voir quelles versions sont installées et lesquelles sont réellement configurées pour être utilisées.

  ```
  user@dev:~$ rvm list

  rvm rubies

  =* ruby-2.3.1 [ x86_64 ]

  # => - current
  # =* - current && default
  #  * - default
  ```

- Avec `rvm use 2.3.0` vous pouvez changer entre les versions installées.

Lire Ruby Version Manager en ligne: https://riptutorial.com/fr/ruby/topic/4040/ruby-version-manager

Chapitre 62: Singe en Ruby

Introduction

Monkey Patching est un moyen de modifier et d'étendre les classes dans Ruby. Fondamentalement, vous pouvez modifier des classes déjà définies dans Ruby, en ajoutant de nouvelles méthodes et même en modifiant des méthodes précédemment définies.

Remarques

La correction de singe est souvent utilisée pour modifier le comportement du code Ruby existant, à partir de gemmes par exemple.

Par exemple, voir cet aperçu.

Il peut également être utilisé pour étendre les classes ruby existantes, comme Rails le fait avec ActiveSupport, en voici un exemple .

Examples

Changer n'importe quelle méthode

```
def hello
  puts "Hello readers"
end

hello # => "Hello readers"

def hello
  puts "Hell riders"
end

hello # => "Hell riders"
```

Modification d'une méthode Ruby existante

```
puts "Hello readers".reverse # => "sredaer olleH"

class String
  def reverse
    "Hell riders"
  end
end

puts "Hello readers".reverse # => "Hell riders"
```

Changer une méthode avec des paramètres

Vous pouvez accéder au même contexte que la méthode que vous remplacez.

```
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 □"
```

Extension d'une classe existante

```
class String
  def fancy
    "~~~{#{self}}~~~"
  end
end

puts "Dorian".fancy # => "~~~{Dorian}~~~"
```

Safe Monkey Patching avec Refinements

Depuis Ruby 2.0, Ruby permet d'avoir des patchs Monkey plus sûrs avec des améliorations. Fondamentalement, il permet de limiter le code Monkey Patched à s'appliquer uniquement lorsque cela est demandé.

Nous commençons par créer un raffinement dans un module:

```
module RefiningString
  refine String do
    def reverse
      "Hell riders"
    end
  end
end
```

Ensuite, nous pouvons décider où l'utiliser:

```
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"
```

Lire Singe en Ruby en ligne: https://riptutorial.com/fr/ruby/topic/6043/singe-en-ruby

Chapitre 63: Singe en Ruby

Examples

Singe patcher une classe

La correction de singe est la modification de classes ou d'objets en dehors de la classe elle-même.

Parfois, il est utile d'ajouter des fonctionnalités personnalisées.

Exemple: remplacer la classe de chaîne pour fournir une analyse à booléen

```
class String
  def to_b
    self =~ (/^(true|TRUE|True|1)$/i) ? true : false
  end
end
```

Comme vous pouvez le voir, nous ajoutons la méthode `to_b()` à la classe String, afin que nous puissions analyser n'importe quelle chaîne à une valeur booléenne.

```
>>'true'.to_b
=> true
>>'foo bar'.to_b
=> false
```

Singe patcher un objet

Tout comme le patchage des classes, vous pouvez également patcher des objets individuels. La différence est que seule cette instance peut utiliser la nouvelle méthode.

Exemple: remplacer un objet chaîne pour fournir une analyse à booléen

```
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)
```

Lire Singe en Ruby en ligne: https://riptutorial.com/fr/ruby/topic/6228/singe-en-ruby

Chapitre 64: Singe en Ruby

Remarques

Le patch de singe, bien que pratique, présente des pièges qui ne sont pas immédiatement évidents. Plus particulièrement, un patch comme celui-là dans l'environnement pollue la portée globale. Si deux modules ajoutent tous deux des `Hash#symbolize` , seul le dernier module requis applique effectivement sa modification; le reste est effacé.

De plus, s'il y a une erreur dans une méthode patchée, le stacktrace pointe simplement vers la classe corrigée. Cela implique qu'il y a un bogue dans la classe `Hash` elle-même (ce qui existe maintenant).

Enfin, Ruby étant très flexible avec les conteneurs à conserver, une méthode qui semble très simple lorsque vous écrivez a beaucoup de fonctionnalités indéfinies. Par exemple, la création de `Array#sum` convient à un tableau de nombres, mais se casse à partir du tableau d'une classe personnalisée.

Une alternative plus sûre est celle des raffinements, disponibles en Ruby> = 2.0.

Examples

Ajout de fonctionnalités

Vous pouvez ajouter une méthode à n'importe quelle classe de Ruby, que ce soit une version intégrée ou non. L'objet appelant est référencé à `self` aide de `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'
```

Lire Singe en Ruby en ligne: https://riptutorial.com/fr/ruby/topic/6616/singe-en-ruby

Chapitre 65: Struct

Syntaxe

- Structure = Struct.new: attribut

Examples

Créer de nouvelles structures pour les données

`Struct` définit de nouvelles classes avec les attributs et méthodes d'accesseur spécifiés.

```
Person = Struct.new :first_name, :last_name
```

Vous pouvez ensuite instancier des objets et les utiliser:

```
person = Person.new 'John', 'Doe'
# => #<struct Person first_name="John", last_name="Doe">

person.first_name
# => "John"

person.last_name
# => "Doe"
```

Personnalisation d'une classe de structure

```
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!"
```

Recherche d'attribut

Vous pouvez accéder aux attributs et aux symboles en tant que clés. Les index numériques fonctionnent également.

```
Person = Struct.new :name
alice = Person.new 'Alice'

alice['name']  # => "Alice"
alice[:name]   # => "Alice"
alice[0]       # => "Alice"
```

Lire Struct en ligne: https://riptutorial.com/fr/ruby/topic/5016/struct

Chapitre 66: Tableaux

Syntaxe

- a = [] # utilisant un littéral de tableau
- a = Array.new # équivalent à l'utilisation d'un littéral
- a = Array.new (5) # crée un tableau avec 5 éléments de valeur nulle.
- a = Array.new (5, 0) # crée un tableau avec 5 éléments avec une valeur par défaut de 0.

Examples

#carte

`#map` , fourni par Enumerable, crée un tableau en invoquant un bloc sur chaque élément et en collectant les résultats:

```
[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]
```

Le tableau d'origine n'est pas modifié. un nouveau tableau est renvoyé contenant les valeurs transformées dans le même ordre que les valeurs source. `map!` peut être utilisé si vous souhaitez modifier le tableau d'origine.

Dans la méthode `map` , vous pouvez appeler la méthode ou utiliser proc pour tous les éléments du tableau.

```
# 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` est synonyme de `collect` .

Créer un tableau avec le constructeur littéral []

Les tableaux peuvent être créés en joignant une liste d'éléments entre crochets (`[` et `]`). Les éléments de tableau dans cette notation sont séparés par des virgules:

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

Les tableaux peuvent contenir n'importe quel type d'objet dans n'importe quelle combinaison sans

restriction de type:

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

Créer un tableau de chaînes

Des tableaux de chaînes peuvent être créés à l'aide de la syntaxe de pourcentage de ruby:

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

C'est fonctionnellement équivalent à définir le tableau comme suit:

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

Au lieu de `%w()` vous pouvez utiliser d'autres paires de délimiteurs: `%w{...}` , `%w[...]` ou `%w<...>` .

Il est également possible d'utiliser des délimiteurs arbitraires non alphanumériques, tels que: `%w!...!` , `%w#...#` ou `%w@...@` .

`%W` peut être utilisé à la place de `%w` pour incorporer l'interpolation de chaîne. Considérer ce qui suit:

```
var = 'hello'

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

Plusieurs mots peuvent être interprétés en échappant à l'espace avec un \.

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

Créer un tableau de symboles

2.0

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

Crée le tableau `[:one, :two, :three, :four]` .

Au lieu de `%i(...)` , vous pouvez utiliser `%i{...}` ou `%i[...]` ou `%i!...!`

De plus, si vous souhaitez utiliser l'interpolation, vous pouvez le faire avec `%I`

2.0

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

Crée les tableaux: `array_one = [:hello, :goodbye, :world] array_two = [:"\#{a}", :"\#{b}", :world] array_one = [:hello, :goodbye, :world]` et `array_two = [:"\#{a}", :"\#{b}", :world]`

Créer un tableau avec Array :: new

Un tableau vide (`[]`) peut être créé avec la méthode de classe `Array::new` , `Array::new` :

```
Array.new
```

Pour définir la longueur du tableau, passez un argument numérique:

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

Il existe deux manières de remplir un tableau avec des valeurs par défaut:

- Passer une valeur immuable en deuxième argument.
- Passez un bloc qui obtient l'index actuel et génère des valeurs 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"]
```

Manipulation des éléments du tableau

Ajouter des éléments:

```
[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]]
```

Suppression d'éléments:

```
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
```

Combinaison de tableaux:

```
[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]
```

Vous pouvez également multiplier les tableaux, par exemple

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

Union des tableaux, intersection et différence

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

Union (`|`) contient des éléments des deux tableaux, avec les doublons supprimés:

```
x | y
```

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

Intersection (`&`) contient des éléments présents à la fois dans le premier et le second tableau:

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

Difference (`-`) contient des éléments présents dans le premier tableau et non présents dans le second tableau:

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

Filtrage de tableaux

Souvent, nous ne voulons opérer que sur des éléments d'un tableau qui remplissent une condition spécifique:

Sélectionner

Renvoie les éléments correspondant à une condition spécifique

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

Rejeter

Renvoie les éléments qui ne correspondent pas à une condition spécifique

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

Les deux `#select` et `#reject` renvoient un tableau, ils peuvent donc être chaînés:

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

Injecter, réduire

Injecter et réduire sont des noms différents pour la même chose. Dans d'autres langues, ces fonctions sont souvent appelées des plis (comme foldl ou foldr). Ces méthodes sont disponibles sur chaque objet Enumerable.

Inject prend une fonction à deux arguments et l'applique à toutes les paires d'éléments du tableau.

Pour le tableau `[1, 2, 3]` nous pouvons ajouter tous ces éléments avec la valeur de départ de

zéro en spécifiant une valeur de départ et bloquer comme suit:

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

Ici, nous passons à la fonction une valeur de départ et un bloc qui dit d'ajouter toutes les valeurs ensemble. Le bloc est d'abord exécuté avec `0` comme `a` et `1` car `b` prend alors le résultat de la prochaine `a` , nous ajoutons alors `1` à la deuxième valeur `2` . Ensuite, nous prenons le résultat de cela (`3`) et ajoutons cela au dernier élément de la liste (également `3`) en nous donnant notre résultat (`6`).

Si nous omettons le premier argument, il définira `a` comme étant le premier élément de la liste. L'exemple ci-dessus est donc identique à:

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

De plus, au lieu de passer un bloc avec une fonction, nous pouvons passer une fonction nommée en tant que symbole, soit avec une valeur de départ, soit sans symbole. Avec ceci, l'exemple ci-dessus pourrait être écrit comme:

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

ou en omettant la valeur de départ:

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

Accès aux éléments

Vous pouvez accéder aux éléments d'un tableau par leurs index. La numérotation des index de tableaux commence à `0` .

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

Vous pouvez recadrer un tableau à l'aide de la plage

```
%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)
```

Cela retourne un nouveau tableau, mais n'affecte pas l'original. Ruby prend également en charge l'utilisation d'indices négatifs.

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

Vous pouvez combiner des indices positifs et négatifs

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

Autres méthodes utiles

Utilisez d' `first` pour accéder au premier élément d'un tableau:

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

Ou d' `first(n)` pour accéder aux `n` premiers éléments retournés dans un tableau:

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

De même pour le `last` et le `last(n)` :

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

Utilisez `sample` pour accéder à un élément aléatoire dans un tableau:

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

Ou `sample(n)` :

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

Tableau bidimensionnel

En utilisant le constructeur `Array::new` , vous pouvez initialiser un tableau avec une taille donnée et un nouveau tableau dans chacun de ses emplacements. Les tableaux internes peuvent également recevoir une taille et une valeur initiale.

Par exemple, pour créer un tableau 3x4 de zéros:

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

Le tableau généré ci-dessus ressemble à ceci quand il est imprimé avec `p` :

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

Vous pouvez lire ou écrire des éléments comme ceci:

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

Les tableaux et l'opérateur splat (*)

L'opérateur `*` peut être utilisé pour décompresser les variables et les tableaux afin qu'ils puissent

être transmis en tant qu'arguments individuels à une méthode.

Cela peut être utilisé pour envelopper un objet unique dans un tableau s'il ne l'est pas déjà:

```
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)
#> []
```

Dans l'exemple ci-dessus, la méthode `wrap_in_array` accepte un argument, `value` .

Si `value` est un `Array` , ses éléments sont décompressés et un nouveau tableau contenant ces éléments est créé.

Si `value` est un objet unique, un nouveau tableau contenant cet objet unique est créé.

Si `value` est `nil` , un tableau vide est renvoyé.

L'opérateur splat est particulièrement utile lorsqu'il est utilisé comme argument dans les méthodes dans certains cas. Par exemple, il permet `nil` , des valeurs uniques et des tableaux à traiter de manière cohérente:

```
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
```

Décomposition

Tout tableau peut être rapidement **décomposé** en affectant ses éléments à plusieurs variables. Un exemple simple:

```
arr = [1, 2, 3]
# ---
```

```
a = arr[0]
b = arr[1]
c = arr[2]
# --- or, the same
a, b, c = arr
```

Précédant une variable avec l'opérateur *splat* (`*`) y place un tableau de tous les éléments qui n'ont pas été capturés par d'autres variables. S'il n'en reste plus, un tableau vide est attribué. Une seule splat peut être utilisée dans une seule tâche:

```
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 décomposition est *sûre* et ne génère jamais d'erreurs. `nil` s sont assignés là où il n'y a pas assez d'éléments, correspondant au comportement de l'opérateur `[]` lors de l'accès à un index hors limites:

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

La décomposition tente d'appeler implicitement `to_ary` sur l'objet affecté. En implémentant cette méthode dans votre type, vous avez la possibilité de la décomposer:

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

Si l'objet décomposé ne répond pas `respond_to? to_ary` , il est traité comme un tableau à élément unique:

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

La décomposition peut également être **imbriquée** en utilisant une expression de décomposition délimitée par `()` à la place de ce qui serait sinon un seul élément:

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

C'est effectivement l'opposé de *splat* .

En fait, toute expression de décomposition peut être délimitée par `()` . Mais pour le premier niveau, la décomposition est facultative.

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

Bordure: *un identifiant unique* ne peut pas être utilisé comme motif de déstructuration, externe ou imbriqué:

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

Lors de l'affectation d' **un littéral de tableau** à une expression de déstructuration, external `[]` peut être omis:

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

Ceci est connu sous le nom d' **affectation parallèle** , mais il utilise la même décomposition sous le capot. Ceci est particulièrement pratique pour échanger des valeurs de variables sans employer de variables temporaires supplémentaires:

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

Les valeurs sont capturées lors de la construction du côté droit de l'affectation. L'utilisation des mêmes variables comme source et destination est donc relativement sûre.

Transformez un tableau multidimensionnel en un tableau unidimensionnel (aplati)

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

Si vous avez un tableau multidimensionnel et que vous devez en faire un tableau *simple* (à une dimension), vous pouvez utiliser la méthode **`#flatten`** .

Obtenir des éléments de tableau uniques

Si vous avez besoin de lire des éléments de tableau en ***évitant les répétitions*** , utilisez la méthode **`#uniq`** :

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

Au lieu de cela, si vous souhaitez supprimer tous les éléments dupliqués d'un tableau, vous pouvez utiliser **`#uniq!`** méthode:

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

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

Alors que la sortie est la même, **`#uniq!`** stocke également le nouveau tableau:

```
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]
```

Obtenez toutes les combinaisons / permutations d'un tableau

La méthode de `permutation` , lorsqu'elle est appelée avec un bloc, produit un tableau à deux dimensions constitué de toutes les séquences ordonnées d'une collection de nombres.

Si cette méthode est appelée sans bloc, elle retournera un `enumerator` . Pour convertir en tableau, appelez la méthode `to_a` .

Exemple	Résultat
`[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`	`[]` -> Pas de permutations de longueur 4

Par contre, la méthode de `combination` , lorsqu'elle est appelée avec un bloc, produit un tableau à deux dimensions composé de toutes les séquences d'une collection de nombres. Contrairement à la permutation, l'ordre est ignoré dans les combinaisons. Par exemple, `[1,2,3]` est identique à `[3,2,1]`

Exemple	Résultat
`[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`	`[]` -> Aucune combinaison de longueur 4

L'appel de la méthode de combinaison seule aboutira à un énumérateur. Pour obtenir un tableau, appelez la méthode `to_a` .

Les méthodes `repeated_combination` et `repeated_permutation` sont similaires, sauf que le même élément peut être répété plusieurs fois.

Par exemple, les séquences `[1,1]` , `[1,3,3,1]` , `[3,3,3]` ne seraient pas valides dans les combinaisons et les permutations régulières.

Exemple	# 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

Créer un tableau de chiffres ou de lettres consécutifs

Cela peut être facilement accompli en appelant `Enumerable#to_a` sur un objet `Range` :

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

`(a..b)` signifie qu'il inclura tous les nombres entre a et b. Pour exclure le dernier numéro, utilisez `a...b`

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

ou

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

Un raccourci pratique pour créer un tableau est `[*a..b]`

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

Supprimer tous les éléments nil d'un tableau avec #compact

Si un tableau contient un ou plusieurs éléments `nil` et qu'ils doivent être supprimés, `Array#compact` ou `Array#compact!` des méthodes peuvent être utilisées, comme ci-dessous.

```
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]
```

Enfin, notez que si `#compact` ou `#compact!` sont appelés sur un tableau sans éléments `nil` , ceux-ci seront nuls.

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

array.compact # => nil

array.compact! # => nil
```

Créer un tableau de nombres

La manière normale de créer un tableau de nombres:

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

Les objets de plage peuvent être largement utilisés pour créer un tableau de nombres:

```
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` méthodes `#step` et `#map` nous permettent d'imposer des conditions sur la gamme des nombres:

```
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]
```

Toutes les méthodes ci-dessus chargent les chiffres avec impatience. Si vous devez les charger paresseusement:

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

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

Lancer à Array à partir de n'importe quel objet

Pour obtenir Array à partir de n'importe quel objet, utilisez `Kernel#Array` .

Ce qui suit est un exemple:

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

Par exemple, vous pouvez remplacer la méthode `join_as_string` par le code suivant

```
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)         #=> ""
```

au code suivant.

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

Lire Tableaux en ligne: https://riptutorial.com/fr/ruby/topic/253/tableaux

Chapitre 67: Tableaux multidimensionnels

Introduction

Les tableaux multidimensionnels dans Ruby ne sont que des tableaux dont les éléments sont d'autres tableaux.

Le seul problème est que, puisque les tableaux Ruby peuvent contenir des éléments de types mixtes, vous devez être sûr que le tableau que vous manipulez est composé d'autres tableaux et non, par exemple, de tableaux et de chaînes.

Examples

Initialisation d'un tableau 2D

Rappelons d'abord comment initialiser un tableau de nombres entiers 1D:

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

Étant un tableau 2D simplement un tableau de tableaux, vous pouvez l'initialiser comme ceci:

```
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]
]
```

Initialisation d'un tableau 3D

Vous pouvez aller plus loin et ajouter une troisième couche de tableaux. Les règles ne changent pas:

```
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']
  ],
  [
    []
  ]
]
```

Accéder à un tableau imbriqué

Accéder au 3ème élément du premier sous-tableau:

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

Aplatissement du tableau

Étant donné un tableau multidimensionnel:

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

l'opération d'aplatissement consiste à décomposer tous les enfants du tableau dans le tableau racine:

```
my_array.flatten

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

Lire Tableaux multidimensionnels en ligne: https://riptutorial.com/fr/ruby/topic/10608/tableaux-multidimensionnels

Chapitre 68: Temps

Syntaxe

- `Time.now`
- `Time.new([year], [month], [day], [hour], [min], [sec], [utc_offset])`

Examples

Comment utiliser la méthode strftime

Convertir un temps en chaîne est une chose assez courante à faire en Ruby. `strftime` est la méthode utilisée pour convertir le temps en chaîne.

Voici quelques exemples:

```
Time.now.strftime("%Y-%m-d %H:%M:S") #=> "2016-07-27 08:45:42"
```

Cela peut être simplifié davantage

```
Time.now.strftime("%F %X")  #=> "2016-07-27 08:45:42"
```

Créer des objets temporels

Obtenez l'heure actuelle:

```
Time.now
Time.new # is equivalent if used with no parameters
```

Obtenez une heure précise:

```
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
```

Pour convertir un temps en époque, vous pouvez utiliser la méthode `to_i` :

```
Time.now.to_i # => 1478633386
```

Vous pouvez également reconvertir de l'époque en heure en utilisant la méthode `at` :

```
Time.at(1478633386) # => 2016-11-08 17:29:46 -0200
```

Lire Temps en ligne: https://riptutorial.com/fr/ruby/topic/4346/temps

Chapitre 69: Test de l'API JSON Pure RSpec

Examples

Tester l'objet Serializer et le présenter au contrôleur

Disons que vous voulez construire votre API pour respecter la spécification jsonapi.org et que le résultat devrait ressembler à ceci :

```
{
  "article": {
    "id": "305",
    "type": "articles",
    "attributes": {
      "title": "Asking Alexandria"
    }
  }
}
```

L'objet Test for Serializer peut ressembler à ceci:

```
# 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
```

L'objet sérialiseur peut ressembler à ceci:

```
# 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
```

Lorsque nous exécutons nos spécifications "sérialiseurs", tout passe.

C'est assez ennuyeux. Introduisons une faute de frappe dans notre sérialiseur d'article: Au lieu de `type: "articles"` , retournons le `type: "events"` et relançons nos tests.

```
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)>'
```

Une fois le test effectué, il est assez facile de détecter l'erreur.

Une fois l'erreur corrigée (corrigez le type d' `article`), vous pouvez le présenter à Controller comme ceci:

```
# 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
```

Cet exemple est basé sur l'article: http://www.eq8.eu/blogs/30-pure-rspec-json-api-testing

Lire Test de l'API JSON Pure RSpec en ligne: https://riptutorial.com/fr/ruby/topic/7842/test-de-l-api-json-pure-rspec

Chapitre 70: Utilisation de gemme

Examples

Installer des gemmes rubis

Ce guide suppose que Ruby est déjà installé. Si vous utilisez Ruby < `1.9` vous devrez installer manuellement RubyGems car il ne sera pas inclus en mode natif .

Pour installer un bijou ruby, entrez la commande:

```
gem install [gemname]
```

Si vous travaillez sur un projet avec une liste de dépendances de gem, celles-ci seront répertoriées dans un fichier nommé `Gemfile` . Pour installer un nouveau bijou dans le projet, ajoutez la ligne de code suivante dans le `Gemfile` :

```
gem 'gemname'
```

Ce `Gemfile` est utilisé par la gemme Bundler pour installer les dépendances `Gemfile` par votre projet, cela signifie toutefois que vous devrez d'abord installer Bundler en exécutant (si vous ne l'avez pas déjà fait):

```
gem install bundler
```

Enregistrez le fichier, puis exécutez la commande:

```
bundle install
```

Spécifier les versions

Le numéro de version peut être spécifié sur la commande live, avec l'indicateur `-v` , par exemple:

```
gem install gemname -v 3.14
```

Lorsque vous spécifiez des numéros de version dans un `Gemfile` , vous disposez de plusieurs options:

- Aucune version spécifiée (`gem 'gemname')` - Installation de la *dernière* version compatible avec les autres gems du `Gemfile` .
- Version exacte spécifiée (`gem 'gemname', '3.14'`) - Ne tentera que d'installer la version `3.14` (et échouera si cela est incompatible avec d'autres gems du `Gemfile`).
- Numéro de version minimal **optimiste** (`gem 'gemname', '>=3.14'`) - Ne tentera que d'installer la *dernière* version compatible avec les autres gems du `Gemfile` , et échouera si aucune

version supérieure ou égale à `3.14` n'est compatible. L'opérateur `>` peut également être utilisé.
- Numéro de version minimum **pessimiste** (`gem 'gemname', '~>3.14'`) - Cela équivaut à utiliser `gem 'gemname', '>=3.14', '<4'` . En d'autres termes, seul le nombre après la *dernière période* est autorisé à augmenter.

En guise de meilleure pratique : vous pouvez utiliser l'une des bibliothèques de gestion de version Ruby, telles que rbenv ou rvm . Grâce à ces bibliothèques, vous pouvez installer différentes versions des runtime et des gems Ruby en conséquence. Ainsi, lorsque vous travaillez dans un projet, cela sera particulièrement utile car la plupart des projets sont codés sur une version connue de Ruby.

Installation de Gem depuis github / fileystem

Vous pouvez installer une gem depuis github ou système de fichiers. Si la gem a été extraite de git ou déjà d'une certaine manière sur le système de fichiers, vous pouvez l'installer en utilisant

```
gem install --local path_to_gem/filename.gem
```

Installation de gem depuis github. Télécharger les sources de github

```
mkdir newgem
cd newgem
git clone https://urltogem.git
```

Construire la gemme

```
gem build GEMNAME.gemspec
gem install gemname-version.gem
```

Vérifier si une gemme requise est installée depuis le code

Pour vérifier si une gemme requise est installée, à partir de votre code, vous pouvez utiliser les éléments suivants (en utilisant nokogiri comme exemple):

```
begin
  found_gem = Gem::Specification.find_by_name('nokogiri')
  require 'nokogiri'
  ....
  <the rest of your code>
rescue Gem::LoadError
end
```

Cependant, cela peut être étendu à une fonction qui peut être utilisée pour configurer des fonctionnalités dans votre code.

```
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
```

Vous pouvez maintenant vérifier si la gemme requise est installée et imprimer un message d'erreur.

```
if gem_installed?('nokogiri')
  require 'nokogiri'
else
  printf "nokogiri gem required\n"
  exit 1
end
```

ou

```
if gem_installed?('nokogiri')
  require 'nokogiri'
else
  require 'REXML'
end
```

Utiliser un Gemfile et un Bundler

Un `Gemfile` est le moyen standard d'organiser les dépendances dans votre application. Un Gemfile de base ressemblera à ceci:

```
source 'https://rubygems.org'

gem 'rack'
gem 'sinatra'
gem 'uglifier'
```

Vous pouvez spécifier les versions du gem que vous voulez comme suit:

```
# 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'
```

Vous pouvez également extraire des gemmes directement d'un repo 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'
```

Vous pouvez également regrouper des gemmes en fonction de leur utilisation. Par exemple:

```
group :development, :test do
    # This gem is only available in dev and test, not production.
    gem 'byebug'
end
```

Vous pouvez spécifier la plate-forme sur laquelle certaines gems doivent s'exécuter si votre application doit pouvoir s'exécuter sur plusieurs plates-formes. Par exemple:

```
platform :jruby do
  gem 'activerecord-jdbc-adapter'
  gem 'jdbc-postgres'
end

platform :ruby do
  gem 'pg'
end
```

Pour installer toutes les gemmes d'un Gemfile, procédez comme suit:

```
gem install bundler
bundle install
```

Bundler / inline (bundler v1.10 et versions ultérieures)

Parfois, vous devez créer un script pour quelqu'un, mais vous n'êtes pas sûr de ce qu'il a sur sa machine. Y a-t-il tout ce dont votre script a besoin? Ne pas s'inquiéter. Bundler a une grande fonction appelée en ligne.

Il fournit une méthode de `gemfile` et avant que le script ne soit exécuté, il télécharge et requiert toutes les gems nécessaires. Un petit exemple:

```
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
```

Lire Utilisation de gemme en ligne: https://riptutorial.com/fr/ruby/topic/1540/utilisation-de-gemme

Chapitre 71: Variables d'environnement

Syntaxe

- ENV [nom_variable]
- ENV.fetch (nom_variable, valeur par défaut)

Remarques

Permet d'obtenir le chemin du profil utilisateur de manière dynamique pour les scripts sous Windows

Examples

Exemple pour obtenir le chemin du profil utilisateur

```
# 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')
```

Lire Variables d'environnement en ligne: https://riptutorial.com/fr/ruby/topic/4276/variables-d-environnement

Chapitre 72: Vérité

Remarques

En règle générale, évitez d'utiliser des doubles négations dans le code. Rubocop dit que les doubles négations sont inutilement complexes et peuvent souvent être remplacées par quelque chose de plus lisible.

Au lieu d'écrire

```
def user_exists?
    !!user
end
```

utilisation

```
def user_exists?
    !user.nil?
end
```

Examples

Tous les objets peuvent être convertis en booléens en Ruby

Utilisez la syntaxe à double négation pour vérifier la véracité des valeurs. Toutes les valeurs correspondent à un booléen, quel que soit leur type.

```
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
```

Toutes les valeurs sauf `nil` et `false` sont véridiques.

```
irb(main):006:0> !!nil
=> false
irb(main):007:0> !!false
=> false
```

La véracité d'une valeur peut être utilisée dans des constructions if-else

Vous n'avez pas besoin d'utiliser la double négation dans les instructions if-else.

```
if 'hello'
    puts 'hey!'
else
    puts 'bye!'
end
```

Le code ci-dessus imprime "hey!" sur l'écran.

Lire Vérité en ligne: https://riptutorial.com/fr/ruby/topic/5852/verite

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec 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	Applications en ligne de commande	Eli Sadoff
3	Arguments de mots clés	giniouxe, mnoronha, Simone Carletti
4	Blocs et Procs et Lambdas	br3nt, coreyward, Eli Sadoff, engineersmnky, Jasper, Kathryn, Lukas Baliak, Marc-Andre, Matheus Moreira, meagar, Mhmd, nus, Pooyan Khosravi, QPaysTaxes, Simone Carletti
5	C Extensions	Austin Vern Songer, photoionized
6	Casting (conversion de type)	giniouxe, Jon Wood, meagar, Mhmd, Nakilon
7	Catching Exceptions avec Begin / Rescue	Sean Redmond, stevendaniels
8	Chargement des fichiers source	mnoronha, nus
9	Classe Singleton	Geoffroy, giniouxe, Matheus Moreira, MegaTom, nus, Pooyan Khosravi
10	Commandes du système d'exploitation ou du shell	Roan Fourie
11	commentaires	giniouxe, Jeremy, Rahul Singh, Robert Harvey
12	Comparable	giniouxe, ndn, sandstrom, sonna
13	Constantes spéciales en rubis	giniouxe, mnoronha, Redithion

14	Cordes	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
15	Création / gestion de gemmes	manasouza, thesecretmaster
16	DateTime	Austin Vern Songer, Redithion
17	Démarrer avec Hanami	Mauricio Junior
18	Des classes	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
19	Des exceptions	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
20	Des symboles	Artur Tsuda, Arun Kumar M, Nick Podratz, Owen, pjrebsch, Pooyan Khosravi, Simone Carletti, Tom Lord, walid
21	Design Patterns and Idioms in Ruby	4444, alexunger, Ali MasudianPour, Divya Sharma, djaszczurowski, Lucas Costa, user1213904
22	Enumerable en Ruby	Neha Chopra
23	Énumérateurs	errm, Matheus Moreira
24	ERB	amingilani
25	Évaluation dynamique	Matheus Moreira, MegaTom, Phrogz, Pooyan Khosravi, Simone Carletti
26	Expressions régulières et opérations basées sur les regex	Addison, Elenian, giniouxe, Jon Ericson, moertel, mudasobwa, Nick Roz, peter, Redithion, Saša Zejnilović, Scudelletti, Shelvacu
27	Fil	Austin Vern Songer, Maxim Fedotov, MegaTom, moertel, Simone Carletti, Surya
28	Flux de contrôle	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
29	Gamme	DawnPaladin, Rahul Singh, Yonatha Almeida
30	Générer un nombre aléatoire	user1821961
31	Hash	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
32	Héritage	br3nt, Gaelan, Kirti Thorat, Lynn, MegaTom, mlabarca, nus, Pascal Fabig, Pragash, RamenChef, Simone Carletti, thesecretmaster, Vasfed
33	Installation	Kathryn, Saša Zejnilović
34	instance_eval	Matheus Moreira
35	Introspection	Felix, giniouxe, Justin Chadwell, MegaTom, mnoronha, Phrogz
36	Introspection en rubis	Engr. Hasanuzzaman Sumon, suhao399
37	IRB	David Grayson, Maxim Fedotov, Saša Zejnilović
38	Itération	Charan Kumar Borra, Chris, Eli Sadoff, giniouxe, JCorcuera, Maxim Pontyushenko, MegaTom, ndn, Nick Roz, Ozgur Akyazi, Qstreet, SajithP, Simone Carletti
39	JSON avec Ruby	Alu
40	La destruction	Austin Vern Songer, Zaz
41	Le débogage	DawnPaladin, ogirginc

42	Les constantes	Engr. Hasanuzzaman Sumon, mahatmanich, user2367593
43	Les méthodes	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
44	Les opérateurs	ArtOfCode, Jonathan, nus, Phrogz, Tom Harrison Jr
45	Message passant	Pooyan Khosravi
46	Métaprogrammation	C dot StrifeVII, giniouxe, Matheus Moreira, MegaTom, meta, Phrogz, Pooyan Khosravi, Simon Soriano, Sourabh Upadhyay
47	method_missing	Adam Sanderson, Artur Tsuda, mnoronha, Nick Roz, Tom Harrison Jr, Yule
48	Modificateurs d'accès Ruby	Neha Chopra
49	Modules	giniouxe, Lynn, MegaTom, mrcasals, nus, RamenChef, Vasfed
50	Nombres	alexunger, Eli Sadoff, ndn, Redithion, Richard Hamilton, Simone Carletti, Steve, Tom Lord, wirefox
51	Opérateur Splat (*)	Kathryn
52	Opérations sur les fichiers et les E / S	Doodad, KARASZI István, Martin Velez, max pleaner, Milo P, mnoronha, Nuno Silva, thesecretmaster
53	OptionParser	Kathryn
54	Portée et visibilité variables	Matheus Moreira, Ninigi, Sandeep Tuniki
55	Queue	Pooyan Khosravi
56	Raffinements	max pleaner, xavdid
57	rbenv	Kathryn, Vidur
58	Récepteurs implicites et compréhension de soi	Andrew
59	Récursion en Ruby	jphager2, Kathryn, SajithP
60	Ruby Version Manager	Alu, giniouxe, Hardik Kanjariya ツ

61	Singe en Ruby	Dorian, paradoja, RamenChef
62	Struct	Matheus Moreira
63	Tableaux	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
64	Tableaux multidimensionnels	Francesco Boffa
65	Temps	giniouxe, Lucas Costa, MegaTom, stevendaniels
66	Test de l'API JSON Pure RSpec	equivalent8, RamenChef
67	Utilisation de gemme	Anthony Staunton, Brian, Inanc Gumus, mnoronha, MZaragoza, NateSHolland, Saša Zejnilović, SidOfc, Simone Carletti, thesecretmaster, Tom Lord, user1489580
68	Variables d'environnement	Lucas Costa, mnoronha, snonov
69	Vérité	giniouxe, Umang Raghuvanshi