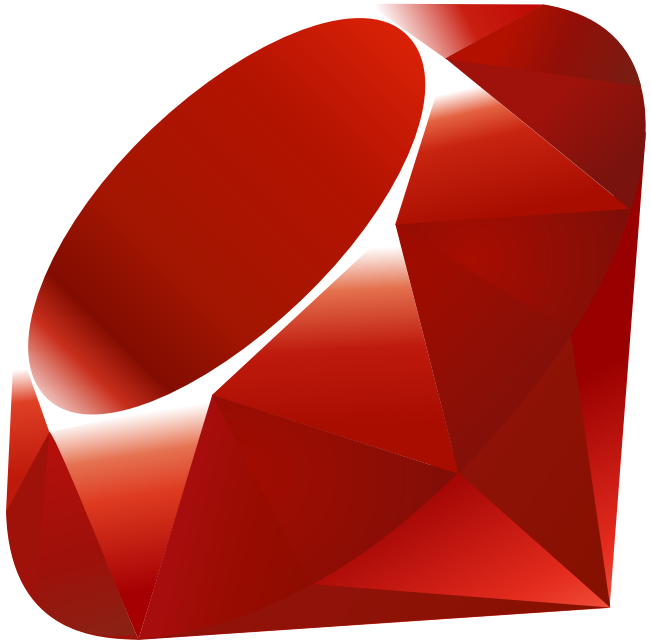
Ruby

A programmer's best friend

https://www.ruby-lang.org/en/about/



Ruby en quelques mots

- Écrit en 1995 par Yukihiro Matsumoto (Matz)
- Langage ~~Orienté~~ Objet
- Principe de *la moindre surprise*
- Rapide à coder
- Lent à exécuter car interprété (comme Python)
    - Plusieurs interpréteurs
    - Peu de vérification syntaxique avant exécution
- Multi plateformes

Type de langage

- Ramasse miette (vs gestion de la mémoire)
- Exceptions (vs retour d'erreur)
- Héritage simple uniquement mais mixins via les *Modules*
- Visibilité des méthodes contrôlable
- Taille des entiers illimitée (vs types CPU)
- Threads mais 😱 Great Interpreter Lock 😱
- Langage objet
- Duck typing

Services web et sites

Langage Objet

- Langage cohérent
  - Toute variable est une référence à un objet
  - Pas de type primitif, donc pas d'opérateurs "magiques" qu'on ne peut exprimer dans le langage
- Uniquement objet: Aucun type primitif
  - `-1.abs` vaut `1`
- Les fonctions existent... Mais sont des méthodes !
- Partie programmation fonctionnelle

Convention de coding

- Principe de la *moindre surprise* : plusieurs façons de coder la même chose
- On code court, lisible
- Pas sensible aux espaces
- Langage à `do` `end` plutôt qu'à accolades
- On ne pré-déclare pas les variables sauf pour en définir la portée
- Conditionnellement parlant: **`nil` et `false` sont faux, tout le reste est vrai**
- Les méthodes dont le nom finit par `?` retournent un boolean
- Les méthodes dont le nom finit par `!` sont desctructives

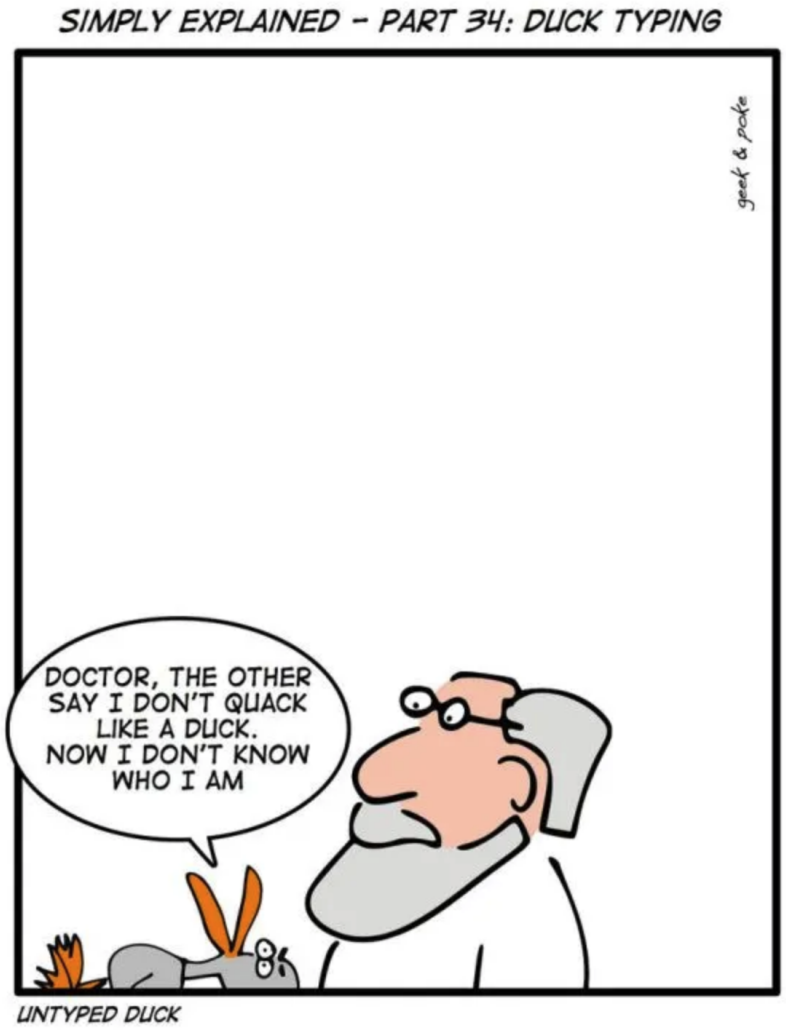
Coding case

- Nom de classe ou de module : CamelCase :
  - `ClasseComplexe`
- Nom de méthode, de variable, de fichier : snake_case
  - `methode_qui_fait_des_trucs()`
  - `variable_qui_varie`
  - `le_nom_de_ma_classe.rb`
- Nom de constantes : UPPER_SNAKE_CASE
  - `MATH_PI`

Duck typing

- On ne type jamais explicitement une variable
- Tout objet a bien un type unique

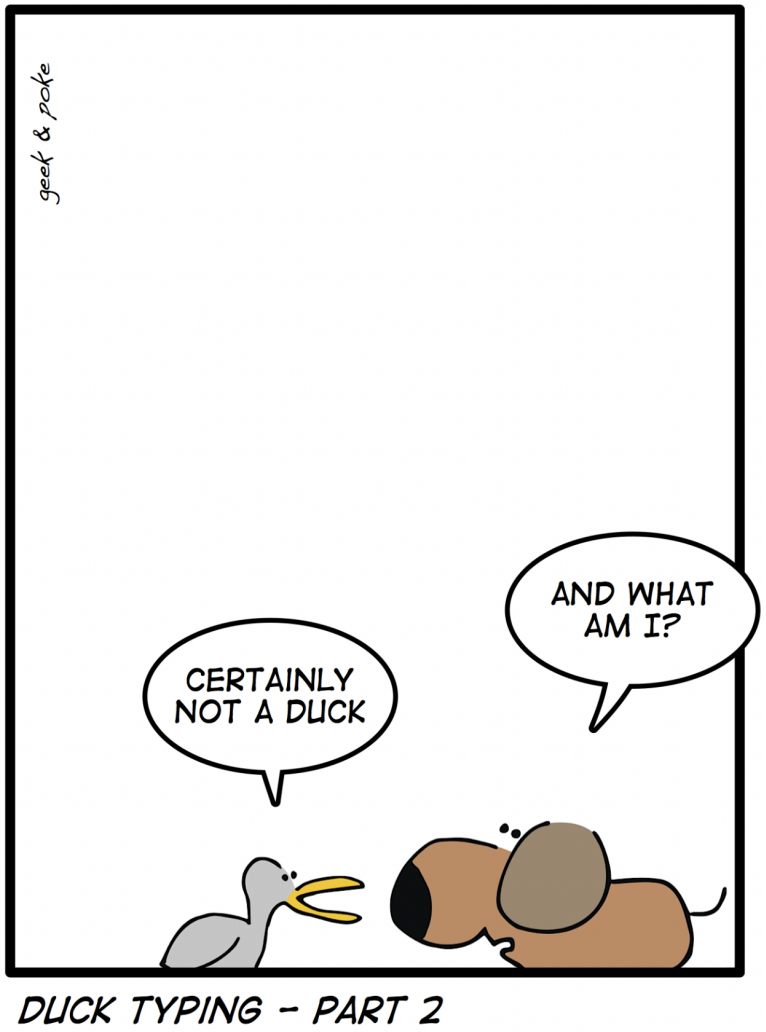
If it looks like a duck, swims like a duck, quacks like a duck, it is probably a duck.



Duck Typing - Variables

- Une forme de typage dynamique
- Les objets ont tous un type unique lié à leur création
- Mais les variables ne sont que des **références sans type**

```ruby
something = 1
puts(something.class) # Integer
something = 'Canard'
puts(something.class) # String
something = String.new('Lapin')
puts(something.class) # String
```

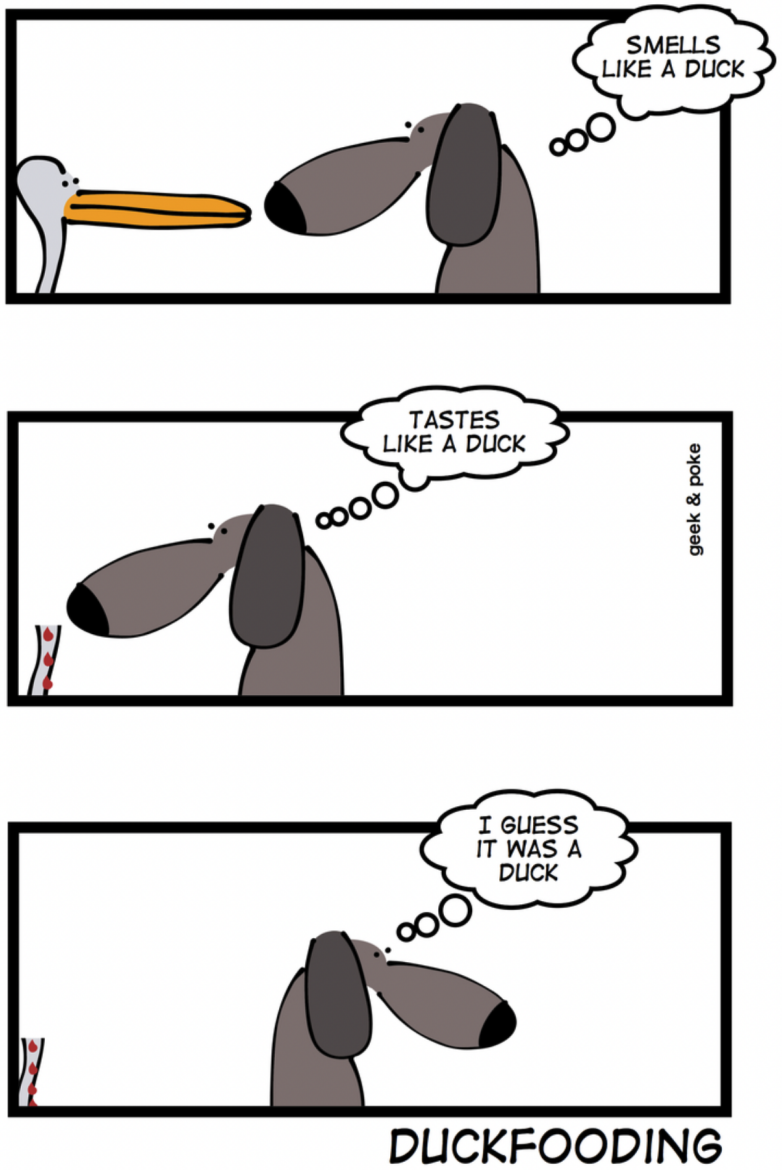


Duck Typing - Fonctions

- Les fonctions ne déclarent pas le type des objets passés

```ruby
def sum(a, b) # Pas d'info sur a et b
    a + b     # Retour automatique
end

something = sum(1, 3) # 4
something = sum('Baby ', 'Shark') # 'Baby Shark'
```



Fonctions - Pas de return ⛔

- La dernière valeur évaluée est retournée

```ruby
def pair(int)
    (int % 2) == 0
end

# if pair(666)
#     puts 'perdu'
# end

puts 'perdu' if pair(666)
```

- `return` est toujours possible mais gardé pour les early returns

Fonctions - Parenthèses optionelles

```ruby
def greet(nom="Joueur", numero=1)
    puts("Bonjour Joueur #{nom} n#{numero}")
end
```

⚠️ **Sans** paramètres, on préfère la notation **sans** parenthèses.

⚠️ **Avec** paramètres, on préfère la notation **avec** parenthèses.

```ruby
greet()                      # Possible mais non conventionnel
greet                        # Conventionnel
greet('Seong Gi-Hun', 456) # Conventionnel
greet 'Seong Gi-Hun', 456  # Possible mais non conventionnel
```

String - Basiques

- Gère UTF *8* , 16, 32, ...
- `'chaine simple'` : sans évaluation
- `"Chaîne évaluée"` : évaluation de blocs et appel à `to_s`

```
word = 'Hiéroglyphe'
puts ("Le mot #{word} a #{word.length} caractères")

sixteen = '16'
i = sixteen.to_i + '4'.to_i

puts('bla' * 3)
```

String - Modifier avec la casse

- `.upcase` `.downcase` `.capitalize`

```ruby
mot = 'bonjour'
puts mot.capitalize   # "Bonjour"
puts mot              # "bonjour"
puts mot.capitalize!  # "Bonjour"
puts mot              # "Bonjour"
```

⚠️ Rappel : par convention, les méthodes dont le nom finit par `!` sont destructives

String - Couper des chaînes

- `.split`

```
phrase = "J'aime les ours!"
phrase.split              # ["J'aime", "les", "ours!"]
phrase.split('e')         # => ["J'aim", " l", "s ours!"]
phrase.split(/[ ,!'-]/)  # => ["J", "aime", "les", "ours"]
```

- `/toto/` dénote une expression rationnelle

```
/^abc.+/.match?("abcdef")
```

- L'inverse de `.split(separator)` est `.join(separator)`

Symboles - String allégée

- Flyweight pattern 🪰
- Un symbole est le nom de quelque-chose pour Ruby.
- Un symbole s’écrit `:nom`, où "nom" est le nom du symbole
- Tous les symboles de même nom sont le même symbole :

```
etat = :pause   # p etat.object_id => 209908
etat = :pause   # p etat.object_id => 209908
etat = "pause"  # p etat.object_id => 240
etat = "pause"  # p etat.object_id => 260
```

- Ruby utilise des symboles pour identifier ses classes, méthodes, variables ...

Array (Liste) - Le conteneur de base

- Type vecteur : Redimensionnement dynamique

```ruby
fruits = ['pomme', 'poire', 'anis']

fruits.each do |fruit|
    puts fruit
end

puts("#{fruits.reverse} - #{fruits.size} éléments")

better_fruits = Array.new       # On pourrait remplacer par []
better_fruits[0] = 'calva'      # Affectation directe
better_fruits.push('williams') # Ajout à la fin
puts better_fruits.inspect      # ["calva", "williams"]
```

Array (Liste) - Parcours

```ruby
def minmax(enumerable)
  min = max = enumerable.first
  enumerable.each do |element|
    if element < min                    # if en version longue
      min = element
    end
    max = element if element > max      # if en version courte
  end
  return min, max                       # Return multiple dans un array
end

tableau = []
10.times { tableau << rand(100) }
min, max = minmax(tableau)
puts "#{min} <= elements <= #{max}"
```

Array (Liste) - Quelques méthodes

- Plein de méthodes héritées du module `Enumerable` 👁️

```ruby
numbers = [8, 1, 2, 2, 4, 1, 123]
puts numbers.uniq          # [8, 1, 2, 4, 123]
puts numbers.sort          # [1, 1, 2, 2, 4, 8, 123]
puts numbers.reverse       # [123, 1, 4, 2, 2, 1, 8]
puts numbers.min           # 1
puts numbers.sort.last     # 123
puts numbers.uniq!.sort!   # [1, 2, 4, 8, 123]
puts numbers               # [1, 2, 4, 8, 123]
```

Array (Liste) - Encore des méthodes !

```ruby
mots = %w[je fais des mots]
mots # ["je", "fais", "des", "mots"]

mots.include?('des')  # true
mots.all? { |mot| mot.size >= 2 } # true
mots.any? { |mot| mot[0] == 'f' } # true

mots.join(' ') # "je fais des mots"

mots.sort_by{ |mot| mot.length } # ["je", "des", "fais", "mots"]
mots.sort_by(&:length) # ["je", "des", "fais", "mots"]
```

- Les deux dernières lignes sont équivalentes 🤔

Array (Liste) - Map/Reduce

Transforme un `Array` en passant tous ses éléments à un *bloc* et en renvoyant un nouvel `Array` avec les nouvelles valeurs.

```ruby
mots = %w[je fais des mots]
mots.map { |mot| mot.capitalize } # ["Je", "Fais", "Des", "Mots"]

mots.map { |mot| mot.size } # [2, 4, 3, 4]
mots.map(&:size)            # [2, 4, 3, 4] - Equivalent
mots.map(&:size).max        # 4 - C'est bien un array

[[1,2],[[3]]].flatten # [1, 2, 3]

mots.map{ |mot| mot.split('') }.flatten.sort.uniq.join # ??
```

Array (Liste) - Map/Reduce

Transforme un `Array` en passant tous ses éléments à un *bloc* et en conservant un accumulateur tout du long pour le *réduire* à un élément. 👉

```ruby
nombres = []
5.upto(7) { |i| nombres.push(i) }
puts nombres # [5, 6, 7]

nombres.reduce(0){ |accumulateur, value| accumulateur + value } # 18
nombres.reduce(10){ |accumulateur, value| accumulateur + value } # 28

nombres.inject(10, :+)    # 28
nombres.inject(:+)        # 18
```

Range - Représenter un interval simplement

- Type de base `Range`
- Représente l'interval de valeurs entières entre les bornes (incluses)
- Enumerable !
- `to_a` pour dénormaliser et transformer en `Array`

```
r1 = Range.new(10, 15)
r2 = 10..15 # équivalent
r1.each { |i| puts i }
r1.to_a # [10, 11, 12, 13, 14, 15]
```

Hash (Dictionnaires) - Basiques

```ruby
mythos = { 'deficit_retraite' => 'consulter COR' }
mythos['voisin europeens'] = "pour une fois qu'on est premiers"
mythos['voisin europeens'] += ' et #slowlife' # accès en écriture
mythos[1200] = 'mensonge des ministres'       # clefs hétérogènes

mythos.each do |clef, valeur|
    puts "#{clef} => #{valeur}"
end

puts mythos.keys   # ["deficit_retraite", "voisin europeens", 1200]
mythos.include?('deficit_retraite') # true
```

⚠️ Rappel : par convention, les méthodes dont le nom finit par `?` retournent un boolean

Hash (Dictionnaires) - Valeur par défaut

- Possibilité de passer une valeur par défaut à la construction

```ruby
animaux = ["chat", "ours", "cheval", "ours"]

compte = Hash.new(0) # Attention, pour les objets, c'est toujours le
                     # même qui risque d'être utilisé

animaux.each do |animal|
  compte[animal] += 1
end

puts compte.inspect # {"chat" => 1, "cheval" => 1, "ours" => 2}
puts compte.to_a # [["chat", 1], ["cheval", 1], ["ours", 2]]
puts compte["martien"] # 0
```

Lecture de fichier

```ruby
# 1ère manière : Entièrement
lines = File.readlines('/etc/passwd') # Array de lignes
lines.first                           # "root:x:0:0::/root:/bin/bash\n"
lines.first.chomp                     # "root:x:0:0::/root:/bin/bash"

# 2ème manière : Par ligne
File.open('/etc/passwd').each_line do |line|
    puts "#{line.size}) #{line}"
end

# 3ème manière : Par char
compteur = Hash.new(0)
File.open('/etc/passwd').each_char do |char|
    compteur[char.downcase] += 1
end
```

ARGV - Arguments d'un programme

- `ARGV` : un `Array` contenant les arguments

```ruby
def grep(motif, file)
  File.new(file).each_line do |line|
    puts line if line.include?(motif)
  end
end

ARGV.each do |arg|
    grep('zsh', arg)
end

# S'appelle de cette façon: ruby cherche_zsh.rb /etc/passwd
```