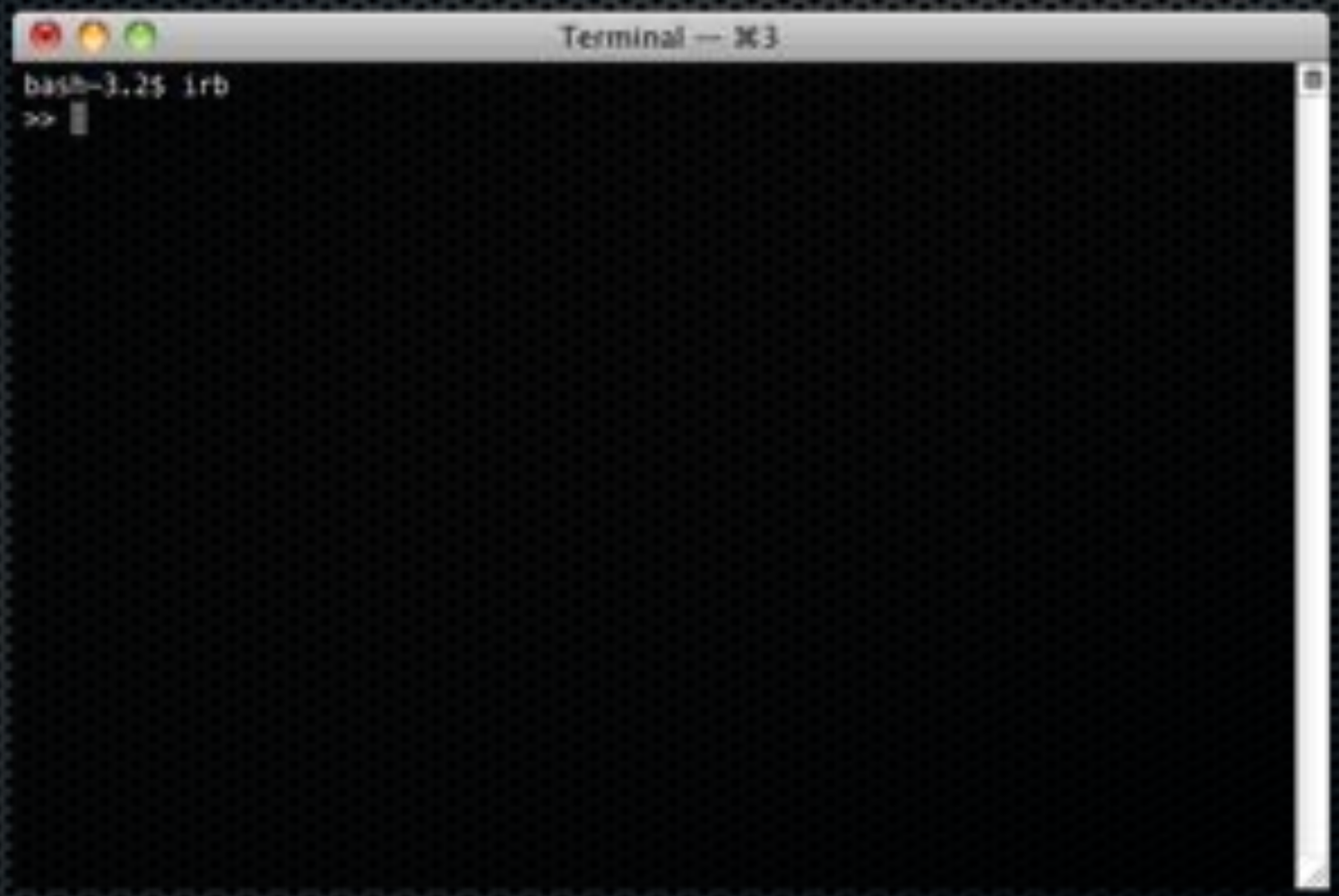
Ruby

Le serment

“Je m’engage à interrompre cette présentation avec des remarques et des questions, quelles que soient mes connaissances en programmation.”

irb

L’arme secrète des Rubyistes



Le REPL

- Ruby fourni un outil Read-Eval-Print-Loop appelé *irb*
  - en gros, on lui envoie du Ruby et il donne le résultat
- C’est un excellent outil pour apprendre le langage
  - Ça devient un outil très puissant pour manipuler des données quand on est dans un framework
- faites vous du bien et jouez beaucoup avec irb

Utiliser irb

- taper irb pour le lancer
- Vous entrez des expressions Ruby
- irb répond avec le résultat
- _ contient le dernier résultat
- On sort avec exit ou ^D

```
$ irb
>> 1 + 2
=> 3
>> "pizza".capitalize
=> "Pizza"
>> %w[G U L P].reverse
=> ["P", "L", "U", "G"]
>> _.join("-")
=> "P-L-U-G"
>> exit
```

Ruby est généraliste

Multi-plateformes

- fonctionne sous Linux et variantes d'Unix
  - y compris Mac OS X où il est installé par défaut
- fonctionne aussi sous Windows, mais le reste de l'écosystème est moins compatible

Scripts et admin système

- très proche de Perl et Python dans les cas d'usage
- on trouve des micro-scripts très spécialisés
- mais aussi des programmes complets et autonomes

Propice aux framework

- Sa souplesse et sa modularité ont permis de créer de nombreux framework :
  - **Ruby on Rails**, **Merb**, … (frameworks web complets)
  - **Sinatra**, **Ramaze**, … (micro-frameworks web)
  - **Anemone** (web spider), **Test::Unit**, **RSpec**, **Cucumber** (frameworks de tests)

Multi-implémentations

- **MRI** (Matz Ruby Interpreter) l'officiel <= 1.8
- **YARV** (Yet Another Ruby Virtual machine) >= 1.9
- JRuby, MacRuby, IronRuby, BlueRuby, …
- Rubinius, MagLev, …

Une véritable spec ISO est en cours de finalisation

Ruby est orienté humain

La syntaxe de Ruby est conçue pour faciliter l'**expression de la pensée** du développeur.

Très proche de l'anglais parlé.

Les noms sont signifiants et cohérents

Concis et précis

- pas de caractères inutiles, si possible
  - fin de lignes
  - parenthèses

```
puts "Qui commande les pizzas ?"

puts "On est 15 à en vouloir"
```

Lisible

- les commandes s’enchaînent comme les mots dans une phrase

```
3.times { print "Pizza !"}

exit unless "menu".include? "nu"



['Pizza', 'Ordis'].each do |truc|
  puts truc
end
```

Expressif et prévisible

- les méthodes sont prévisibles
- il y a des conventions de nommage cohérentes entre les classes et librairies

```ruby
door.check_if_opened
# vs
door.open?

friend.set_first_name('Vincent')
# vs
friend.first_name = 'Vincent'


# Array
['olive','anchois'].include? 'olive'

# Range
(0..9).include? 3

# String
"PLUG".include? "LUG"

# IPAddr
net1 = IPAddr.new "192.168.2.0/24"
net2 = IPAddr.new "192.168.2.1"
net1.include? net2
```

Les types de données

Les fondamentaux de Ruby

Types et Structures

- Ruby a des types communs à la plupart des langages de programmation : **String**, **Integer**, **Float**, …
- Ruby a 2 structures principales : **Array** et **Hash**
  - Ces structures sont très souples et peuvent servir de files, piles, ensembles, …
- Ruby a d'autres types, comme **Time**
- Et tout ça, ce sont des objets en Ruby

String	“\tplein de texte\n#{1 + 2}” ‘moins (\\ ou \’)’
Integer	255 0377 0xFF 0b111111111
Float	0.00003 3.0e-5
Array	[“Pizza”, “Bière”, “Chips”, “Canapé”] %w[Pizza Bières Chips Canapé]
Hash	{“nom” => “PLUG”, “membres” => 42}
Symbol	:lieu_reunions
Regexp	/\AP(?:rovence)?L(?:inux)?U(?:ser)?G(?:roup)\z/
Time	Time.now Time.local(2010, 4, 9) Time.utc(2010, 4, 9)

Manipuler des **String**s

- **String** fourni :
  - changement de casse
  - nettoyage d’espaces
  - édition courante
  - indexation
  - ...

```
>> space = "\textra space   \n"
=> "\textra space   \n"
>> space.strip
=> "extra space"
>> space.rstrip
=> "\textra space"

>> "James".delete("aeiou")
=> "Jms"

>> date = "Avril 2010"
=> "Avril 2010"
>> date[0..4]
=> "Avril"
>> date[-2..-1]
=> "10"
>> date[/\d+/]
=> "2010"
```

Manipuler des **Array**s

- **Array** fourni :
  - ajout d'éléments
  - retrait d'éléments
  - indexation
  - union, intersections
  - ...

```
>> a = [0, 1]
=> [0, 1]
>> a << 2 << 3 << 4
=> [0, 1, 2, 3, 4]
>> a.pop
=> 4
>> a
=> [0, 1, 2, 3]

>> a[1]
=> 1
>> a[1..-1]
=> [1, 2, 3]

>> a & [0, 2, 4, 6]
=> [0, 2]
>> a | [42]
=> [0, 1, 2, 3, 42]
```

Manipuler des **Hash**es

- **Hash** fourni :
  - stockage clé/valeur
  - ajout/retrait de clés
  - indexation
  - requêtage
  - ...

```
>> h = {:a => 1, :b => 2}
=> {:a=>1, :b=>2}
>> h[:b]
=> 2
>> h[:c] = 3
=> 3
>> h
=> {:a=>1, :b=>2, :c=>3}

>> h.keys
=> [:a, :b, :c]
>> h.values
=> [1, 2, 3]

>> h.include? :c
=> true
>> h.include? :d
=> false
```

Conversions de types

- Ruby a plein de méthodes de conversion
  - **String**s vers **Integer**s ou **Float**s
  - Nombres en **String** dans une base donnée
  - **String**s vers **Array**s, et retour
  - ...

```
>> pi = "3.1415"
=> "3.1415"
>> pi.to_f
=> 3.1415
>> pi.to_i
=> 3

>> num = 42
=> 42
>> num.to_s
=> "42"
>> num.to_s(16)
=> "2a"

>> aperos = "pizza,bieres,chips"
=> "pizza, bieres, chips"
>> aperos.split(",")
=> ["pizza", "bieres", "chips"]
>> aperos.split(",", 2)
=> ["pizza", "bieres, chips"]
>> aperos.split(",").join(" | ")
=> "pizza | bieres | chips"
```

Le **Duck typing**

- Si un object “marche” comme un canard
- et si il “parle” comme un canard
- alors c’est un canard

```ruby
# On vérifie si ces objets implémentent la méthode to_str()
# à ne pas confondre avec to_s() qui fait du transtypage

puts ('Une chaîne'.respond_to? :to_str)
# => true
puts (Exception.new.respond_to? :to_str)
# => true
puts (4.respond_to? :to_str)
# => false
```

Le contrôle de flux

Logique conditionnelle

La déclaration **if**

- Ruby utilise les conditionnelles **if**/**elsif**/**else**
- Le code est executé si la condition donne **true**
- En Ruby, **false** et **nil** sont **false** et tout le reste est **true** (**0**, **“”**, etc.)

```ruby
num = rand(10)
print "#{num}:   "
if num == 7
  puts "Chanceux!"
elsif num <= 3
  puts "Un peu bas."
else
  puts "Un nombre chiant."
end
# >> 2:  Un peu bas.
```

Quand ça tourne mal

- Ruby “lève” des erreurs (appelées **Exception**s) quand ça tourne mal
- Les objets Error on un type, un message, et un backtrace
- Par defaut, l’execution s’arrête si elle n’est pas capturée

```
>> 42 / 0
ZeroDivisionError: divided by 0
  from (irb):1:in `/'
  from (irb):1
  from :0
```

Gestion des **Exceptions**

- Il faut encadrer le code incertain avec **begin ... end**
- Ajouter la clause **rescue** pour les types d'erreur qu'on veut gérer

```
>> begin
>>   n = 42 / 0
>> rescue ZeroDivisionError => error
>>   n = 0
>> end
=> 0
>> n
=> 0
```

Objets et Méthodes

En Ruby, (presque) tout est un objet

Tout est un **Object**

- À part de très très rares exceptions, tout est un **Objet**
- Même un nombre est un **Objet** et on peut appeler des méthodes dessus

```
>> -42.abs
=> 42
>> 3.times { puts "Pizza" }
Pizza
Pizza
Pizza
=> 3
```

Les méthodes d’un objet

- Visibilité :
  - **publiques**
  - **protégées**
  - **privées**
- Portée
  - de **classe**
  - d’**instance**
  - de **Singleton**

```ruby
class UneClasse
  def an_instance_method
    puts "instance"
    a_protected_method
  end

  def self.a_class_method
    puts "classe"
  end

  def call_for_private
    private_method
  end

  def indirect_private(other)
    other.private_method
  end

  protected
  def a_protected_method
    puts "protegee"
  end

  private
  def private_method
    puts "privee"
  end
end
```

```ruby
UneClasse.new.an_instance_method
# => "instance"
#    "protegee"

UneClasse.a_class_method
# => "methode de classe"

UneClasse.new.a_protected_method
# => Erreur :
#    protected method 'a_protected_method'
#    called for <#UneClasse:0x80125478>

UneClasse.new.call_for_private
# => "privee"

other = UneClasse.new
UneClasse.new.indirect_private(other)
# => Erreur :
#    private method 'a_private_method'
#    called for <#UneClasse:0x80125478>
```

Les variables d'un objet

- Portée
  - de **classe**
  - d'**instance**
  - **locales**

```ruby
class Name
  def initialize(first = nil)
    self.first = first
  end

  def first=(first)
    @first = first
  end

  def first
    @first
  end
end
```

```ruby
jlecour   = Name.new("Jérémy")
mrmoins   = Name.new
mrmoins.first = "Christophe"
puts jlecour.first
puts mrmoins.first
# >> Jérémy
# >> Christophe
```

Variables d’instance

Elles sont privées, stockées par objet

Héritage simple

- Une **Class**e peut déclarer un seul parent
- Un enfant hérite du comportement de ses parents
- **Object** en Ruby est le parent le plus élevé pour toutes les **Class**es

```ruby
class Parent
  def salut
    @salutation ||= "Coucou"
  end
end
class Enfant < Parent
  def initialize
    @salutation = "Yo!"
  end
end
puts Parent.new.salut
puts Enfant.new.salut
# >> Coucou!
# >> Yo!
```

Méthodes dangereuses et questions

- Ruby a des conventions de nommage
- Les méthodes dangereuses finissent par **!**
- Les méthodes interrogatives (qui renvoient **true** ou **false**) finissent par **?**

```
>> s = "string"
=> "string"
>> s.upcase
=> "STRING"
>> s
=> "string"
>> s.upcase!
=> "STRING"
>> s
=> "STRING"

>> 0.zero?
=> true
>> 0.0.zero?
=> true
>> 0.00001.zero?
=> false
```

“Mixins”

Le moyen spécial de Ruby pour partager des méthodes

Modules

- Ruby n'a pas d'héritage multiple
- À la place on peut "mixer" un **Module** de méthodes "dans" dans une **Class**e
- On appèle ces **Module**s des "mixins"

```ruby
module Netstring
  def to_netstring(*args)
    str = to_s(*args)
    "#{str.length}:#{str},"
  end
end

class String
  include Netstring
end
class Integer < Numeric
  include Netstring
end

p "Pizza".to_netstring
p 42.to_netstring(2)
# >> "5:Pizza,"
# >> "6:101010,"
```

Blocs et Itérateurs

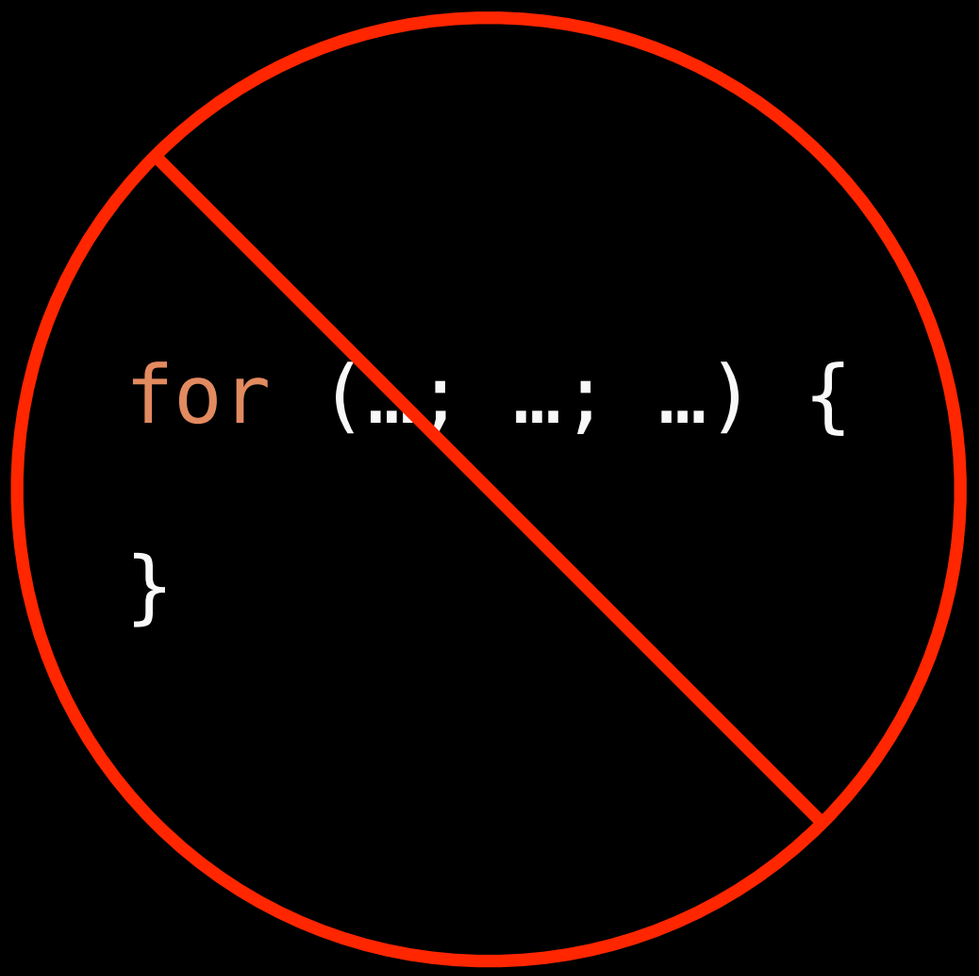
Les Rubyistes ne supportent pas les boucles

Blocs

- En Ruby, on peut passer un bloc (du code) à une méthode
  - Le code du bloc est dans **{ ... }** ou **do ... end**
- Cette méthode peut executer le code fourni avec **yield**

```ruby
def jusqu_a_succes
  loop do
    break if yield == :succes
  end
end

jusqu_a_succes {
  puts "Appelé."
  :succes if rand(3).zero?
}
# >> Appelé.
# >> Appelé.
# >> Appelé.
# >> Appelé.
```



Les Rubyistes ne “bouclent” pas

On “itere” plutôt

L’itérateur **each()**

- On laisse Ruby s’occuper des index
- **each()** appellera le bloc une fois pour chaque élément de la collection

```ruby
name = %w[Pizza Bières Chips]
name.each do |word|
  puts word.reverse
end
# >> azziP
# >> serèiB
# >> spihC
```

The **map()** Iterator

- **map()** est utilisé pour transformer une collection
- Chaque élément de la collection est passé au bloc et le résultat remplace l'élément dans une nouvelle collection

```ruby
nums = *1..5
p nums
p nums.map { |n| n ** 2 }
# >> [1, 2, 3, 4, 5]
# >> [1, 4, 9, 16, 25]
```

L’itérateur **select()**

- **select()** sert à filtrer une collection
- Chaque élément est passé dans le bloc et si le résultat donne **true** l’élément est placé dans une nouvelle collection

```ruby
nums = *1..10
p nums
p nums.select { |n| n % 2 == 0 }
# >> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
# >> [2, 4, 6, 8, 10]
```

Et bien plus encore !

On a à peine égratigné la surface de Ruby

Ruby est un Langage riche

- Plus de 140 méthodes sur **String** et plus de 70 sur **Array**
- Conversion mathématiques automatiques
- Un puissant **case** (conditions multi-branches)
- Comportements personnalisés par objet
- Plus de 30 itérateurs
- Puissantes capacités de réflexion

Questions ?

Merci à **James Edward Gray II (@jeg2)** pour m’avoir autorisé à utiliser ses slides comme base de départ.