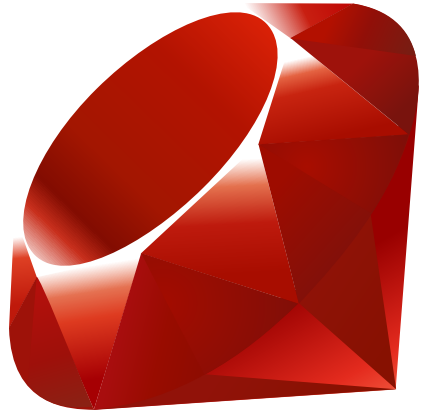
Ruby plus avancé

1. Fonctions (paramètres nommés)
2. Classes
3. Héritage
4. Modules multi-fonction
5. Exceptions



Fonctions - Paramètres nommés

Un paramètre nommé est identifié par son nom, non par sa position

```ruby
def affiche_statistiques(duree: 60, recus: 0, emis: 0)
    reception = recus.to_f / duree
    envoi = emis.to_f / duree
    puts "Recus: #{reception}b/s \t Emis: #{envoi}b/s"
end
```

L'ordre des arguments n'a plus d'importance, mais il faut tous les nommer.

```ruby
affiche_statistiques(duree: 48, recus: 144, emis: 63)
affiche_statistiques(duree: 48, emis: 03)
affiche_statistiques(emis: 03, duree: 48)
affiche_statistiques(48, 144, 63) # Error: wrong number of arguments
```

`puts` Affiche une chaine de caractère et saute une ligne

Fonctions - Paramètres nommés et positionnels ensemble

On peut aussi mélanger arguments normaux et nommés.

```ruby
def affiche(message, repetitions: 1, important: false)
    message.upcase! if important
    repetitions.times do
      puts message
    end
end

affiche("bonjour !")
affiche("bonjour !", repetitions: 2, important: true)
affiche("bonjour !", important: true)
```

⚠️ Les arguments nommés doivent être en dernier ⚠️

Classes - Une classe tout simple

```ruby
class User
  def initialize(name, favorite_emoji: nil) # Constructeur
    @name, @favorite_emoji = name, favorite_emoji
  end

  def name # Getter
    @name
  end

  def name=(name) # Setter
    @name = name
  end

  def to_s
    "User #{@name} (#{@favorite_emoji})"
  end
end
```

Classes - Une classe tout simple

```ruby
class User
  attr_accessor :favorite_emoji
  attr_reader :name

  def initialize(name, favorite_emoji: nil, age: nil) # Constructeur
    @name, @favorite_emoji, @age = name, favorite_emoji, age
  end

  def to_s
    "User #{@name} (#{@favorite_emoji})"
  end
end
```

`attr_reader(symbol, ...)` : génère une méthode pour lire un attribut (getter)

`attr_writer(symbol, ...)` : génère une méthode pour écrire un attribut (setter)

`attr_accessor(symbol, ...)` : génère un getter et un setter pour le mème attribut

Classes - Construire une instance

```ruby
alice = User.new('Alice', favorite_emoji: '😱')
bob = User.new('Bob')

puts alice.name
puts alice
```

- Comment alice va être affiché par `puts` ?

```
irb(main):052:0> puts alice
User Alice (😱)
```

- `@attribut` visible seulement depuis l'intérieur de la classe (et hérité !)
- Aucun accès possible à l'âge, comment l'ajouter ?

Classes - Méthodes et attributs de classe

```ruby
class Vehicle
  @@fleet = [] # Attribut de classe

  def initialize(identifier)
    @identifier = identifier
    @@fleet.push(self) # self: objet courant
  end

  def self.fleet # Méthode de classe
    @@fleet
  end
end

falcon = Vehicle.new('Falcon Millennium');
tie = Vehicle.new('Tie Fighter')

p Vehicle.fleet  # `p` affiche une version plus détaillée d'un objet (via #inspect),
                 # utile pour le débuggage
Vehicle.fleet.each { |vehicle| p vehicle }
```

Classes - Visibilité des méthodes

- `public` : visible depuis partout (par défaut)
- `private` : visible depuis l'instance uniquement (mais les methodes privés sont hérités)
- `protected` : comme `private` sauf pour les instances des classes filles, et autres instances de la même classe

Encapsulation en Ruby

- Seule l'instance à accès à ses méthodes privée (pas les autres instances de la même classe)
- Les autres méthodes de la classe peuvent accéder aux méthodes privée via l'objet courant

Classes - Visibilité des méthodes - Exemple trivial

```ruby
class Armoire
  attr_reader :contenu

  def initialize(taille_max = 10)  # Publique
    @taille_max = taille_max
    @contenu = []
  end

  def stocker(*objets) # Publique (par défaut)
    objets.each { |objet| ranger(objet) }
  end

  private # Les méthodes suivantes seront privées

  def ranger(objet) # Privée
    raise 'Trop gros !' if objet.size > @taille_max
    @contenu << objet
  end
end

Armoire.new.stocker('lingot', 'liasse de petites coupures')
```

Classes - Visibilité des méthodes - Exemple d'usage de protected

```ruby
class User
  def initialize(name)
    @name = name
  end

  def ==(other)
    @name == other.name # Utilise le reader
  end

  protected # Pourrait-on mettre private ???
  attr_reader :name # reader protected 🤔
end

user1 = User.new('Bilbo'); user2 = user1.dup # copie
p (user1 == user2)
```

Héritage en Ruby

- Pas d’héritage privé/protégé ou d’interface
- `initialize` pour le constructeur
- `<` pour héritage
- `super` appelle la méthode de même nom de la classe mère
  - Appelé sans argument, passe les arguments de la méthode courante
  - ⚠️ C'est différent de Java/C++/C#
- Les méthodes les plus spécialisées sont toujours appelées (toutes virtuelles)
- Pas de override
  - Car pas de surcharge explicite
- Pas d’équivalent au final de Java (pas dans la philosophie Ruby).

Héritage en Ruby - Exemple de sous-classe

```ruby
class Downcase < String
  def initialize(value='')
    super(value.downcase)
  end

  def ==(other)
    self.to_s == other.downcase # Pourquoi le to_s?
  end
end

test_string = Downcase.new('DéRiVaTion')
puts test_string # => dérivation
puts (test_string == "dériVation") # => true

test_string += '!!' # => "dérivation!!"
```

Modules - Comme namespace

Ranger son code dans une arborescence

```ruby
module Authentication
  class User
    def initialize(name, role)
      @name, @role = name, role
    end
    attr_reader :name, :role
  end

  module Roles
    BASIC_USER = :basic_user
    MODERATOR = :moderator
  end
end
```

Modules - Comme namespace (2)

- Les modules peuvent s'imbriquer (module défini dans module) - Arborescence
- Un module peut être défini sur plusieurs fichiers
- Notez l'usage des symboles dans l'exemple
- Créez des modules de code indépendants - Loosely coupled

Modules - Comme namespace (3)

À l'utilisation :

```ruby
role = Authentication::Roles::MODERATOR
user = Authentication::User.new("Alice", role)

def can_edit_post?(post, user)
  post.owner == user || user.role == :moderator
end
```

On peut inclure les modules pour éviter de taper le namespace... mais on s'interdira de le faire.

```ruby
include Authentication

role = Roles::MODERATOR
user = User.new("Alice")
```

Modules - Comme namespace (4)

Les modules peuvent porter des méthodes et attributs de classe.

```ruby
module PhoneHelpers
  @@sent_texts = 0

  def self.send_text_message(number, text)
    # ...
    @@sent_texts += 1
  end

  def self.sent_texts
    @@sent_texts
  end
end

PhoneHelpers.send_text_message("36303630", "Hello Santa")
PhoneHelpers.sent_texts # => 1
```

Modules - En tant que Mixins

Objectif : Génériciser du code et/ou obtenir des fonctionnalités automatiquement.

```ruby
module Flyable
  def fly
    puts "#{self} vole!"
  end
end

class Seaplane < Boat
  include Flyable
end

vehicle = Seaplane.new
vehicle.fly
```

Modules - En tant que Mixins (2) - Pour résumer

Le module définit :

- des méthodes
- des attributs
- des constantes

La classe inclut le module.

```
class Seaplane < Boat
  include Flyable
end
```

L'instance de l'objet récupère les **méthodes** du module, les **attributs** définis par le module et peut utiliser les **constantes** comme si elles étaient dans sa classe.

Modules - En tant que Mixins (3) - Etendre un seul objet

```ruby
class Car
  def initialize(color)
    @color = color
  end

  def to_s
    "Voiture #{color}"
  end
end

k2000 = Car.new('Noire')
taxi = Car.new('Blanche')

taxi.extend(Flyable)

taxi.fly # "Voiture blanche vole!"
k2000.fly # => NoMethodError
```

Modules - En tant que Mixins (4) - Et si je veux faire ça ?

```ruby
class Car
  include FleetManagement # vient avec notify_creation
                          # et ajoute une méthode de classe fleet
  def initialize(color)
    @color = color
    notify_creation
  end
end

c1 = Car.new('red')
c2 = Car.new('blue')

p Car.fleet
# [
#  #<Car:0x000000015016c398 @color="red">,
#  #<Car:0x000000015016c230 @color="blue">
# ]
```

Modules - En tant que Mixins (5) - Callback included

Quid des méthodes de classes ?

```ruby
module FleetManagement
  def notify_creation
    self.class.add_vehicle(self)
  end

  def self.included(base_class)
    base_class.extend(FleetManager) # la magie opère ici, on étend l'objet qui représente la classe
  end

  module FleetManager
    attr_accessor :fleet

    def add_vehicle(vehicle)
      @fleet = [] if @fleet.nil?
      @fleet.push(vehicle)
    end
  end
end
```

Modules - En tant que Mixins (6) - Monkey Patching

Si on part d'un module pour plus entendre parler d'Elon Musk :

```ruby
module MuskAvoidance
  def should_i_read?
    !self.match?(/elon musk/i)
  end
end
```

On peut alors ré-ouvrir une classe pour inclure le module (ou ajouter/surcharger des méthodes).

```ruby
class String
  include MuskAvoidance
end
# L'include nécessite que self ait une méthode `match?`
"Elon Musk a encore fait un truc".should_i_read? # => false
```

Modules - Conclusion

- Les modules sont facile à utiliser et servent à mettre des classes, constantes, méthodes, etc dans des espaces de nommage.
- Les modules peuvent servir à faire des mixins, qui ressemblent à :
  - du code pré-processé (macros) en C/C++
  - des interfaces avec des implémentations par défaut en Java/C#
  - des traits en Rust
- C'est un peu compliqué d'ajouter des méthodes de classe avec des modules. On verra les `ActiveSupport::Concern` qui simplifient cette tâche.

Lever une exception

- Les exceptions servent à gérer des erreurs rares et imprévues.

```ruby
raise "Une erreur s'est produite" # RuntimeError
raise ArgumentError, "Wrong parameters" # On peut passer le type d'exception
```

- Si l'argument passé est un `String`, `raise` lève une `RuntimeError`
- De nombreuses exceptions existent en Ruby (cf. doc)
- Les types d'erreurs les plus standards sont des sous-classes de `StandardError`

```ruby
raise StandardError, "Une erreur s'est produite"
```

Lever une exception et gérer une exception custom

```ruby
class ProblemBetweenChairAndKeybord < StandardError
end

def foo
  raise ProblemBetweenChairAndKeybord.new('Manque de café')
end

begin
  foo
rescue ProblemBetweenChairAndKeybord => error
  p err.message
end
```

- Notez la syntaxe pour gérer l'exception en fonction de son type et récupérer son instance.