



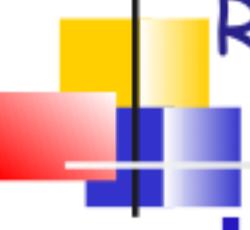
## Chapitre 4 : Manipuler des données

Les Blocs

Les Containers

Les Itérateurs

Les Modules

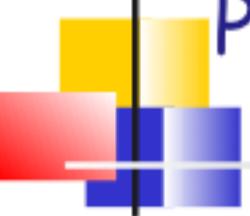


# Résumé des épisodes précédents

---

Dans le premier cours : les concepts fondamentaux la POO (et de Ruby) :

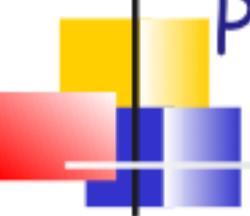
- • le système est composé d'objets
  - • les objets interagissent en s'envoyant des messages
    - l'envoi d'un message provoque l'exécution d'une méthode dans le contexte de l'objet receveur
  - tous les objets sont instances d'une classe
- Le cours n°2 a donné lieu à une présentation de la syntaxe Ruby.
  - Dans le cours n°3 nous avons abordé l'héritage qui est le troisième principe fondateur de la POO (et de Ruby).
  - Dans le cours 4 nous présentons les "outils" évolués pour la manipulation des données, un autre des apports des langages objets.



# Plan

---

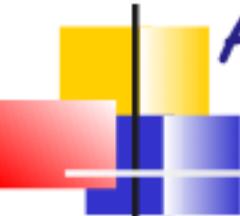
- A-Les Blocs
- B-Les Containers
  - 1-Les Collections
  - 2-Les Tableaux
  - 3-Les Chaînes de caractères
  - 4-Les Tables de Hashages
  - 5-Les Intervalles
- C-Les Itérateurs



# Plan

---

- D-Organisation du code
  - 1-Héritage Multiple en Poo
  - 2-Les Modules
  - 3-Les Mixins
  - 4-Retour sur les Collections
  - 5-Le Module Enumerable
- E-Exemples



## A-Les Blocs

---

- Un Bloc Ruby est similaire à un bloc de code en C, ie. une suite d'expression entre {}.
- Un Bloc C

```
{  
    printf("\nJe suis ");  
    printf("un bloc \n");  
    printf("et j'aime ");  
    printf("ça");  
}
```

- Un Bloc Ruby

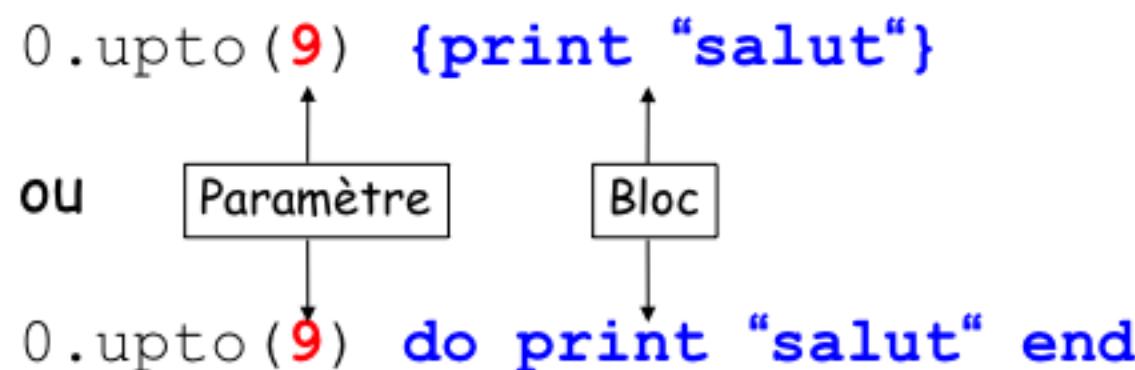
```
{  
    puts "Hello"  
    puts "I am a block"  
    puts "and I love that"  
}
```

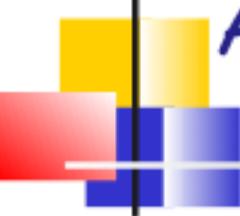
(On peut également délimiter les blocs en utilisant do ...end)

- En fait, c'est beaucoup plus que cela,

## A-Les Blocs

- Un bloc ne peut apparaître dans le code qu'à la suite d'un appel de méthode.
  - Par exemple : un bloc peut être utilisé dans une structure de répétition. Il apparaît à la suite des paramètres de la méthode.





## A-Les Blocs

---

- Un bloc peut aussi être utilisé avec des méthodes quelconques.
  - Le bloc est écrit sur la même ligne que le dernier paramètre de la méthode.
  - Le code du bloc n'est pas évalué immédiatement, Ruby conserve le contexte d'utilisation du bloc (les variables locales, l'object courant, etc.) et entre dans la méthode.
  - Dans la méthode le code du bloc peut être évalué par un appel à la méthode `yield`

# A-Les Blocs

Dans la méthode le bloc peut être évalué par un appel à la méthode **yield**

- Exemple

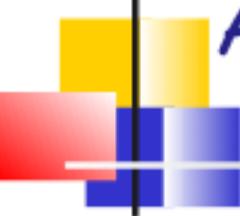
```
def nFois (unNombre)
    unNombre.times {yield}
end
```

```
nFois(3) { puts "Bonjour" }
```

## Résultat

```
Bonjour  
Bonjour  
Bonjour
```

Le bloc commence sur la même ligne que le dernier paramètre de la méthode.



## A-Les Blocs

---

- Les Blocs peuvent recevoir des paramètres
  - Dans la définition du bloc ( {.....} ), ils sont déclarés entre | |
  - Ils peuvent être en nombre quelconques.
  - Lors de la demande d'évaluation par **yield**, il faut alors fournir le nombre correct de paramètres.
- Les Blocs ont une valeur de retour qui peut être utilisée, c'est la valeur de la dernière expression du bloc

# A-Les Blocs

- Exemples

- Dans les itérations

```
1.upto(9) { |para| print para, " " }
```

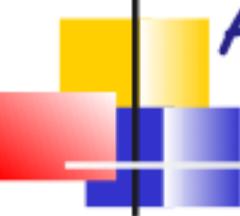
- Dans une méthode

```
def fibUpTo(max)
    i1, i2 = 1, 1
    while i1 <= max
        yield i1
        i1, i2 = i2, i1+i2
    end
end
```

```
fibUpTo(1000) { |f| print f, " " }
```

Lors de l'appel le paramètre formel **f** est lié à **i1**

Résultat : 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987

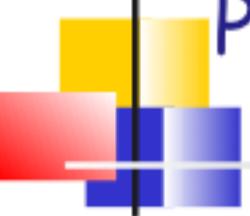


## A-Les Blocs

---

- Il est possible de stocker un bloc dans une variable, on obtient ce qu'on appelle un objet procédure ou **closure**
  - Cet objet peut par exemple être passé en paramètre à une fonction

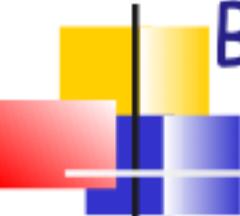
```
irb(main):001:0> p=lambda { |s| print s }
=> #<Proc:0x028178d8@(irb):1>
irb(main):002:0> p.class
=> Proc
irb(main):003:0> p.call ("Toto")
Toto=> nil
irb(main):004:0> p.call (1234)
1234=> nil
irb(main):005:0>
```



# Plan

---

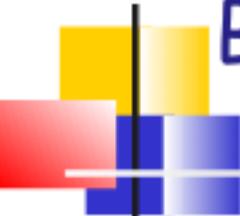
- A-Les Blocs
- **B-Les Containers**
  - 1-Les Collections
  - 2-Les Tableaux
  - 3-Les Chaînes de caractères
  - 4-Les Tables de Hashages
  - 5-Les Intervalles
- C-Les Itérateurs



## B-Les Containers

---

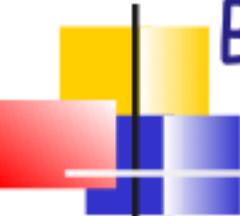
- Un **Container** en programmation est un objet qui peut contenir d'autres objets.
  - On utilise souvent le terme de **Collection**
  - Exemples : Liste, Pile, Interface graphique.
- Certains containers sont dits :
  - Hétérogènes :
    - panel dans une interface peut contenir des objets de natures différentes (bouton, zone de texte, ...)
  - Homogènes :
    - Pile d'entiers, Chaîne de caractères.



## B1-Les Collections

---

- Les Collections fournissent des services pour manipuler les données qu'elles contiennent :
  - Ajouter, supprimer, etc.
  - Permettent d'implémenter des TDA
- Exemples de Collections “Standards”
  - Le Sac (Bag)
    - fourre-tout, pour mettre des objets en vrac ; la plus simple des collections. Pas de tri, ni d'accès par index, ordre des éléments imprévisible. Doublons possibles
  - L'ensemble (Set)
    - ~ un ensemble mathématique, Pas de doublons



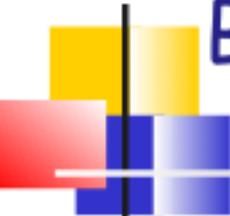
## B1-Les Collections

---

- En Ruby

- Une Chaine de Caractère (**String**) peut être considérée comme une collection homogène contenant des caractères.
  - Dans une moindre mesure(\*), les instances de la classe Intervalle (**Range**) aussi, collection homogène contenant des éléments entiers
- Les Tableaux (**Array**) ou Tables de Hashage(**Hash**) peuvent aussi être considérés comme des collections hétérogènes.

(\*) Un objet de la classe Range ne "contient" pas vraiment les données qu'il représente, seulement les bornes



## B2-Les Tableaux

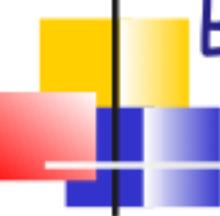
---

- La classe `Array` est utilisée pour représenter une collection d'items
  - Exemple

```
num = [ "zéro", "un", "deux", "trois", "quatre" ]  
      => ["zéro", "un", "deux", "trois", "quatre"]  
num.class => Array
```

- la méthode `class` indique que `num` est un tableau (`Array`). Comme dans les langages traditionnels on peut accéder de façon individuelle aux éléments du tableau.

```
num[0]    =>  
num[1]    =>  
num[4]    =>
```



## B2-Les Tableaux

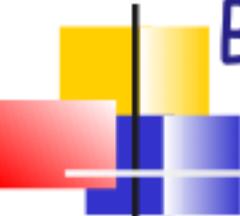
---

- La classe `Array` est utilisée pour représenter une collection d'items
  - Exemple

```
num = [ "zéro", "un", "deux", "trois", "quatre" ]
      => ["zéro", "un", "deux", "trois", "quatre"]
num.class => Array
```

- la méthode `class` indique que `num` est un tableau (`Array`). Comme dans les langages traditionnels on peut accéder de façon individuelle aux éléments du tableau.

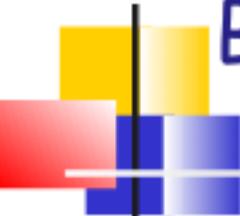
```
num[0]    => "zéro"
num[1]    => "un"
num[4]    => "quatre"
```



## B2-Les Tableaux

---

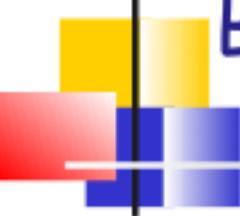
- Pour ajouter des données supplémentaires :
  - `num[5] = "cinq" => "cinq"`  
`num => ["zéro", "un", "deux", "trois", "quatre", "cinq"]`
  - Il peut y avoir des “trous”
    - `num[8] = "huit" => "huit"`  
`num => ["zéro", "un", "deux", "trois", "quatre", "cinq", nil, nil, "huit"]`
      - De lui même Ruby initialise les cases vides.
  - Les entrées dans le tableau sont stoquées séquentiellement à partir de 0.
    - Il peut contenir un nombre quelconque d'objets.
    - La taille n'a pas à être précisée au préalable.
  - Le Tableau Ruby en fait est une Liste



## B2-Les Tableaux

---

- Les objets stockés dans les tableaux gardent leurs propriétés et restent manipulable de la même façon.  
num[3].class =>  
num[3].upcase =>  
num[3].reverse =>
- Dans un tableau on peut stocker n'importe quel type d'objet
  - En même temps  
=> Array = Container hétérogène



## B2-Les Tableaux

---

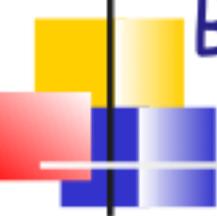
- Les objets stockés dans les tableaux gardent leurs propriétés et restent manipulable de la même façon.

`num[3].class => String`

`num[3].upcase => "TROIS"`

`num[3].reverse => "siort"`

- Dans un tableau on peut stocker n'importe quel type d'objet
  - En même temps  
=> `Array = Container hétérogène`



## B2-Les Tableaux

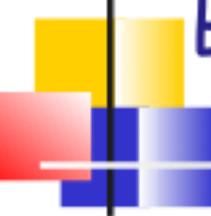
---

- Exemple

```
adr = [ 284, "Silver Sprint Rd" ]  
=> [284, "Silver Sprint Rd"]
```

- On peut en particulier stocker des tableaux dans un tableau

```
adrs = [ [ 284, "Silver Sprint Rd" ]  
        , [ 344, "Onta. Dr" ] ]  
adrs[0]      => [284, "Silver Sprint Rd"]  
adrs[1]      => [344, "Onta. Dr"]  
adrs[0][0]    => 284  
adrs[0][1]    => "Silver Sprint Rd"
```



## B2-Les Tableaux

---

### ■ Création

- `Array.new([taille=0 [,remplissage=nil]])`

`Array.new` =>

`Array.new(2)` =>

`Array.new(3, "truc")` =>

### ■ Manipulations ensemblistes

- Intersection &

`[1,2,3,4] & [1,3,5]` =>

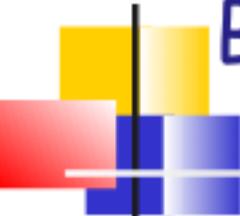
- Union |

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

- Concaténation

`["ta"] * 3` =>

`["truc", "machin"] * "-" =>`



## B2-Les Tableaux

---

### ■ Création

- `Array.new([taille=0 [,remplissage=nil]])`

`Array.new`      => `[]`

`Array.new(2)`    => `[nil, nil]`

`Array.new(3,"truc")` => `["truc", "truc", "truc"]`

### ■ Manipulations ensemblistes

- Intersection &

`[1,2,3,4] & [1,3,5]`      => `[1,3]`

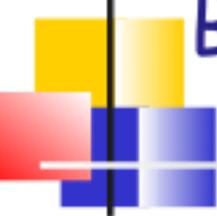
- Union |

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

- Concaténation

`["ta"] * 3`                => `["ta", "ta", "ta"]`

`["truc", "machin"] * "-" => "truc-machin"`



## B2-Les Tableaux

---

- Concaténation (suite)

- Addition

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

- Soustraction

`[1,2,3,4]-[1,3,5]` =>

- Accès

`[1,2,3].at(1)` =>

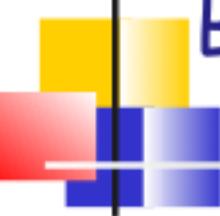
- Test

- `a.empty?`

- Retourne true si le tableau est vide

- `a.include?(ele)` ou `a.member?(ele)`

- Retourne true si le tableau contient ele



## B2-Les Tableaux

---

- Concaténation (suite)

- Addition

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

- Soustraction

`[1,2,3,4]-[1,3,5]` => `[2, 4]`

- Accès

`[1,2,3].at(1)` => `2`

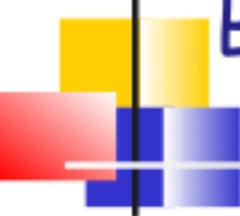
- Test

- `a.empty?`

- Retourne true si le tableau est vide

- `a.include?(ele)` ou `a.member?(ele)`

- Retourne true si le tableau contient ele



## B2-Les Tableaux

---

### ■ Manipulation d'éléments

- `a.first et a.last`
  - Retourne le premier / dernier élément
- `a.push (obj) ou a<<obj`
  - Ajoute obj à la fin du tableau
- `a.pop`
  - Supprime et Retourne le dernier élément
- `a.shift`
  - Supprime et Retourne le premier élément du tableau
- `a.unshift (obj)`
  - Ajoute obj au début du tableau

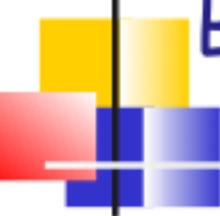


## B2-Les Tableaux

---

### ■ Suppression

- `a.clear` Supprime tous les éléments du tableau
- `a.uniq ou a.uniq!` Supprime les doublons
- `a.compact ou a.compact!`
  - Supprime tous les éléments nil du tableau
- `a.delete(e)`
  - Supprime tous les éléments correspondant à e
- `a.delete(e) {|el| ...}`
  - Si aucun élément supprimé renvoie l'évaluation du bloc
- `a.delete_at(n)`
  - Supprime l'élément situé à l'indice n dans le tableau
- `a.delete_if { |x| ...}`
  - Supprime les élts pour lesquels le bloc renvoie true



## B2-Les Tableaux

---

- Autres méthodes

- `unTableau.sort`
- `unTableau.reverse`
- `unTableau.length`

- Affichage

- `puts unTableau`

- Conversion

- `unTableau.to_s`
- `unTableau.join([s=$,])`  
["truc", "muche"].join =>  
["truc", "muche"].join(" ") =>



## B2-Les Tableaux

---

### ■ Autres méthodes

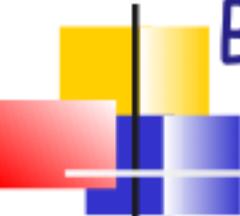
- `unTableau.sort`
- `unTableau.reverse`
- `unTableau.length`

### ■ Affichage

- `puts unTableau`

### ■ Conversion

- `unTableau.to_s`
- `unTableau.join([s=$,])`  
    `["truc", "muche"].join => "trucmuche"`  
    `["truc", "muche"].join(" ") => "truc muche"`



## B2-Les Tableaux

---

### ■ Exemple de Tri

- ad=[ [ 285, "Onta. Dr"], [17, "Queb. St"],  
[ 39, "Main St" ] ]

ad

=>[[285, "Onta. Dr"], [17, "Queb. St"],  
[39, "Main St"]]

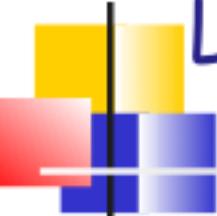
ad.sort

=>[[17, "Queb. St"], [39, "Main St"],  
[285, "Onta. Dr"]]

- ads =[[20, "Onta. Dr"], [20, "Main St"]]

ads.sort

=>[[20, "Main St"], [20, "Onta. Dr"]]



# Les Tableaux

---

## ■ Autres exemples

```
nom = [ "Nel", "Tom", "Lisa", "Ben"]
```

=>

```
nom.sort =>
```

```
nom =>
```

```
nom.reverse =>
```

```
nom.length =>
```

```
nom-["Ben"] =>
```

```
nom+=["Luc"] =>
```

```
nom-=["Nel", "Lisa"] =>
```

```
nom*=2 =>
```



# Les Tableaux

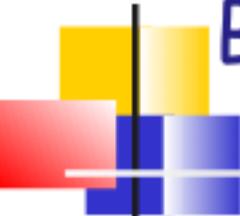
---

## ■ Autres exemples

```
nom = [ "Nel", "Tom", "Lisa", "Ben"]
       => ["Nel", "Tom", "Lisa", "Ben"]

nom.sort      => ["Ben", "Lisa", "Nel", "Tom"]
nom           => ["Nel", "Tom", "Lisa", "Ben"]
nom.reverse   => ["Ben", "Lisa", "Tom", "Nel"]
nom.length    => 4

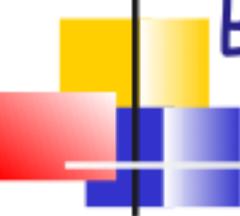
nom-["Ben"]   => ["Nel", "Tom", "Lisa"]
nom+["Luc"]   => ["Nel", "Tom", "Lisa", "Ben", "Luc"]
nom-["Nel", "Lisa"] => ["Tom", "Ben", "Luc"]
nom*=2 => ["Tom", "Ben", "Luc", "Tom", "Ben", "Luc"]
```



## B3-Les Chaînes de caractères

---

- Une chaîne de caractères peut être assimilée à un container homogène de caractères.
  - Sous Ruby on travaille souvent sur les chaines après les avoir converties en container Tableau
    - **String => Array => Traitement => String**
  - Exemple
    - **Conversion String => Array**  
s="Jacoboni"  
s.split("//") =>
    - **Conversion Array => String**  
a= ["i","l"," ","e","s","t"," ","f","o","r","t"]  
a.join =>



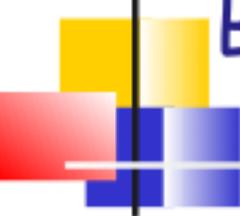
## B3-Les Chaînes de caractères

---

- Une chaîne de caractères peut être assimilée à un container homogène de caractères.
  - Sous Ruby on travaille souvent sur les chaines après les avoir converties en container Tableau
    - **String => Array => Traitement => String**
  - Exemple
    - Conversion String => Array

```
s="Jacoboni"
s.split("//)    => ["J", "a", "c", "o", "b", "o", "n", "i"]
```
    - Conversion Array => String

```
a= ["i", "l", " ", "e", "s", "t", " ", "f", "o", "r", "t"]
a.join      => "il est fort"
```

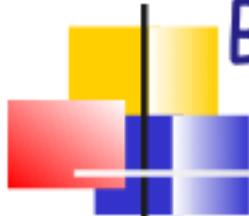


## B4-Les Tables de Hashages

---

- Une Table de Hashages permet de représenter des collections de paires clé-valeur.
  - Ou en quelque sorte des collections indicées par un type d'objet quelconque
- Méthode de création

```
h1=Hash.new([defaut = nil])  
      h1[30] => nil  
h2=Hash.new(15)  
      h2[30] => 15  
h3={1,2,3,4,5,6} =>  
      h3[1] => ; h3[10] =>
```



## B4-Les Tables de Hashages

---

- Une Table de Hashage permet de représenter des collections de paires clé-valeur.
  - Ou en quelque sorte des collections indexées par un type d'objet quelconque
- Méthode de création

```
h1=Hash.new([defaut = nil])
    h1[30] => nil
h2=Hash.new(15)
    h2[30] => 15
h3={1,2,3,4,5,6} => {5=>6, 1=>2, 3=>4}
    h3[1] => 2 ; h3[10] => nil
```



## B4-Les Tables de Hashages

---

- `h.clear`, supprime toutes les paires clé-valeur de `h`
- `h.delete(clé)`
  - Supprime la paire clé-valeur pour la clé indiquée
- `h.delete_if { |clé, valeur| ... }`
  - Supprime les paires clé-valeur pour lesquelles le bloc renvoie `true`

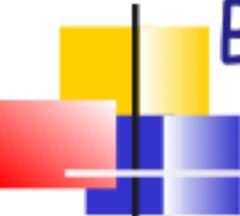
```
h={1=>2, 2=>4}
h.delete_if { |k, v| k % 2 ==0 }
h => {1=>2}
```
- `h.to_a`
  - Renvoie un tableau de tableaux `[clé, val]` de `h`
- `h.size`
  - Renvoie le nombre de paires de `h`



## B4-Les Tables de Hashages

---

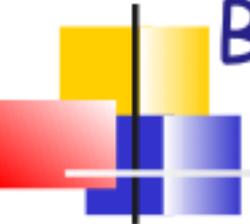
- `h.value?(valeur)`
  - Renvoie true si la valeur apparaît dans h
- `h.values`
  - Renvoie un tableau contenant toutes les valeurs de h
- `h.key?(cle)`
  - Renvoie true si la clé apparaît dans h
- `h.keys`
  - Renvoie un tableau contenant toutes les clés de h
- `h.rehash`
  - Reconstruit le hachage. Si un hachage n'est pas reconstruit après la modification d'une de ses clés, cette clé ne sera plus accessible.



## B4-Les Tables de Hashages

---

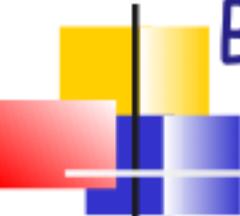
- `h.shift`
  - Supprime et retourne la première paire (clé, valeur)
- `h.sort ou h.sort {|a, b| ... }`
  - Produit un tableau trié
- `h.invert`
  - Inverse les clé et les valeurs. Attention si une valeur est présente plusieurs fois, une est prise au hasard.
- `h.index(valeur)`
  - Renvoie la clé de valeur ou nil si absente
- `h.fetch(clé[, default=nil])`
  - Renvoie la valeur associé à clé
- `h.fetch(clé) {|clé| ... }`
  - Si clé est absente, renvoie la valeur de retour du bloc



## B4-Les Tables de Hashages

---

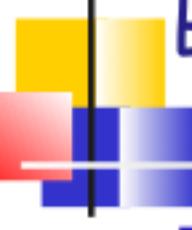
- `h.each { |cle, valeur| ... }`
  - Exécute le bloc pour chaque paire clé-valeur dans un ordre non spécifié
- `h.each_key { |cle| ... }`
  - Exécute le bloc pour chaque clé
- `h.each_value { |valeur| ... }`
  - Exécute le bloc pour chaque valeur



## B5-Les Intervalles

---

- La classe `Range` permet de représenter des intervalles à l'aide des opérateurs de créations `..` ou `...` ou encore `Range.new`
  - Les Intervalles peuvent être considérés comme des collections
    - Même si les objets intervalles “ne contiennent” pas vraiment les objets qu’ils représentent. (seulement les bornes)
  - Les méthodes
    - `begin` ou `first`, `end` ou `last`, `size` ou `length`
    - `==`



## B5-Les Intervalles

---

- ===

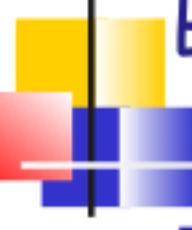
```
(1..10) === 5          =>  
(1..10) === 15         =>  
(1..10) === 3.14159   =>  
('a'..'j') === 'c'    =>  
('a'..'j') === 'z'    =>
```

- r1=1..12

```
r1.begin => ; r1.end      =>  
r1.each { |x| print x, ":"}  
=>
```

- r2=1...12

```
r2.begin => ; r2.end      =>  
r2.each { |x| print x, ":"}  
=>
```



## B5-Les Intervalles

---

■ ===

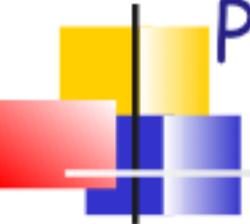
```
(1..10) === 5          => true
(1..10) === 15         => false
(1..10) === 3.14159   => true
('a'..'j') === 'c'     => true
('a'..'j') === 'z'     => false
```

■ r1=1..12

```
r1.begin => 1 ; r1.end      => 12
r1.each { |x| print x, ":"}
=> 1:2:3:4:5:6:7:8:9:10:11:12:
```

■ r2=1...12

```
r2.begin => 1 ; r2.end      => 12
r2.each { |x| print x, ":"}
=> 1:2:3:4:5:6:7:8:9:10:11:
```



# Plan

---

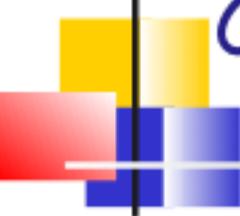
- A-Les Blocs
- B-Les Containers
  - 1-Les Collections
  - 2-Les Tableaux
  - 3-Les Chaînes de caractères
  - 4-Les Tables de Hashages
  - 5-Les Intervalles
- C-Les Itérateurs



## C-Les Itérateurs

---

- Un itérateur est une méthode qui utilise un bloc ou un objet de la classe Proc
- Dans le code le Bloc est placé immédiatement après l'invocation de la méthode.
- Les Itérateurs sont utilisés pour produire des structures de contrôles définies par les utilisateurs eux-mêmes, et en particulier les boucles.

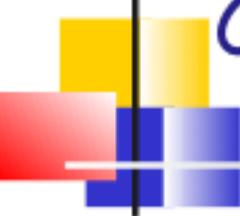


## C-Les Itérateurs

---

- Les itérateurs ne sont pas un concept propre à ruby. Ils sont courants dans les LOO.
  - Ils sont aussi utilisés dans des langages du type Lisp, bien qu'ils ne s'y appellent pas comme cela..
  - Le verbe *itérer* signifie faire la même chose plusieurs fois, un itérateur est donc quelque chose qui fait la même chose plusieurs fois.
    - Quand on écrit des programmes, on a besoin de boucles. En C, on les code avec des `for` ou des `while`.

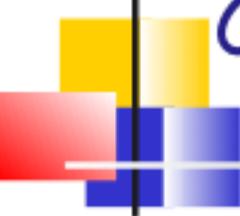
```
char *str;
for (str = "abcdefg"; *str != '\0'; str++) {
    /* ici on traite un caractère */
}
```



## C-Les Itérateurs

---

- La syntaxe C du `for(...)` fournit une abstraction pour créer une boucle.
  - Mais le test de `*str` vis à vis d'un caractère nul requiert du programmeur une connaissance de la représentation interne des chaînes.
    - le C est un langage de bas niveau.
- Les langages de plus haut niveau sont caractérisés par leur support plus souple des itérations.
  - Exemple : un programme Shell Unix



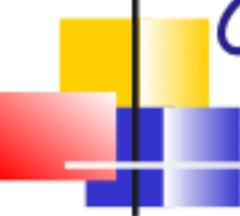
## C-Exemple : Un shell Unix

---

- Considérons le script shell sh suivant :

```
#!/bin/sh
for i in *.[ch]; do
# ... ici on ferait quelque chose sur chaque fichier
done
```

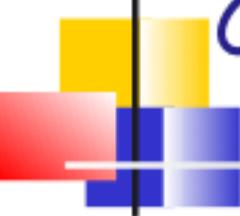
- Tous les fichiers sources C et header du répertoire courant sont traités, et la commande shell traite les détails relatifs à la collecte et à la substitution des noms de fichiers les uns après les autres.
- Voilà quelque chose de plus haut niveau que le C, non ?



## C-Les Itérateurs

---

- Mais il y a plus:
  - c'est bien de trouver dans un langage de quoi faire des itérations sur les types d'objets prédéfinis, mais c'est décevant s'il faut revenir aux boucles de bas niveau pour ses propres types d'objets.
    - C'est pourquoi tous les langages orientés objet fournissent ce qu'il faut pour les itérations.
- Certains fournissent une classe spéciale pour cela, ruby quand à lui permet de définir des itérateurs directement.



## C-Les Itérateurs

---

- Le type **string** (**chaîne**) en ruby offre des itérateurs très utiles:

```
"abc".each_byte{|c| printf "<%c>", c}; print "\n"
```

- **each\_byte** est un itérateur pour chacun des caractères de la chaîne.
  - Chaque caractère est placé successivement dans la variable locale **c**.
  - Ceci peut être traduit par quelque chose qui ressemble à du C ...

```
s="abc"; i=0
while i<s.length
    printf "<%c>", s[i]; i+=1
end;
print "\n"
```

<a><b><c>



## Mais, l'Itérateur each\_byte

---

- est à la fois conceptuellement plus simple et plus enclin à continuer de fonctionner même si la classe String devait être radicalement modifiée dans le futur.
  - Un des bénéfices des itérateurs est qu'ils tendent à être robustes face à de tels changements, ce qui est d'ailleurs une caractéristique du code sain en général.
- Un autre itérateur de String est each\_line.

```
"a\nb\nc\n".each_line{|l| print l}
```

- Tout ce qui ferait le gros du travail en C (chercher les délimiteurs de ligne, générer des souschaînes, etc.) est facilement réalisé avec ces itérateurs.

## C-Les Itérateurs

■ `yield` apparaît parfois dans la définition d'un itérateur.

- Rappel : `yield` passe le contrôle au bloc de code qui est associé à l'itérateur.
- L'exemple suivant définit un itérateur `repeat`, qui répète un bloc de code le nombre de fois indiqué en argument.

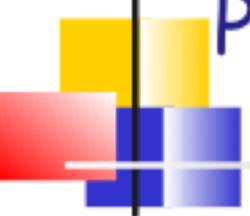
```
def repeat(num)
    while num > 0
        yield
        num -= 1
    end
end
```

### Utilisation

```
repeat(3) { print "foo " }
```

### Résultat

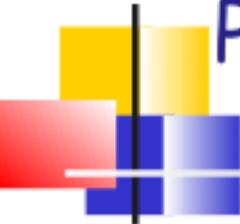
```
foo foo foo
```



# Plan

---

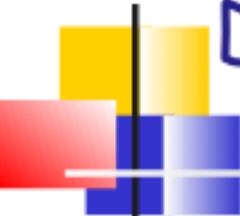
- A-Les Blocs
- B-Les Containers
  - 1-Les Collections
  - 2-Les Chaînes de caractères
  - 3-Les Tableaux
  - 4-Les Tables de Hashages
  - 5-Les Intervalles
- C-Les Itérateurs



# Plan

---

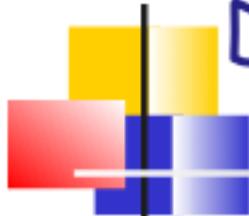
- D-Organisation du code
  - 1-Héritage Multiple en Poo
  - 2-Les Modules
  - 3-Les Mixins
  - 4-Retour sur les Collections
  - 5-Le Module Enumerable
- E-Exemples



## D1-Héritage Multiple en Poo

---

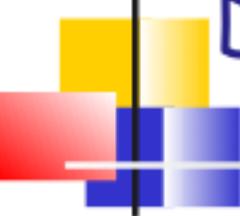
- Certains langages à objets comme C++, Eiffel, offrent le concept d'héritage multiple.
  - Une sous-classe peut hériter de deux ou de plusieurs super-classes
    - Ceci pose un certain nombre de problèmes non triviaux,
      - Au niveau de la définition du langage
      - Au niveau de son implémentation.
    - Le graphe d'héritage d'une application n'a plus la forme d'un arbre, mais d'un graphe
      - Il est possible qu'une classe hérite d'une même super-classe par deux, ou plusieurs, chemins distincts.



## D1-Héritage Multiple

---

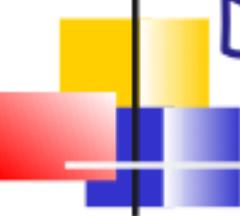
- Ruby comme Smalltalk et Java ne supporte pas l'héritage multiple.
  - Ruby propose néanmoins une forme d'héritage multiple basée sur les Mixins
    - Modules de définitions de méthodes inclus dans une classe
    - Cela permet de définir une classe qui hérite d'une autre et inclue un ou plusieurs modules.



## D2-Les Modules

---

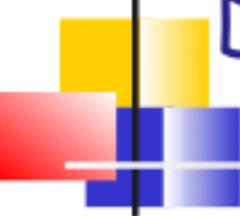
- Différences entre Classes et Modules Ruby
  - Les Modules sont des collections de méthodes et de constantes. Elles ne peuvent pas générer d'instances au contraire des classes.
    - Les Classes sont des modules un peu plus spécialisé, d'ailleurs la classe Class hérite de la classe Module
  - Une classe peut hériter d'une autre classe mais pas d'un module.
  - Un module ne peut pas hériter d'un autre module ou d'une classe



## D2-Les Modules

---

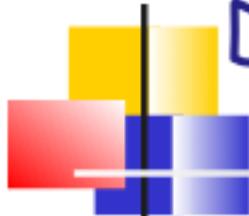
- Les Modules peuvent être “mixed in” (incorporés) dans les classes ou autres modules
  - Les constantes et méthodes du module sont alors “ajoutées” à la classe ou module augmentant les fonctionnalité de la classe ou Module
  - Les Classes, toutefois ne peuvent pas être “incorporées à (“mixed in”) autre chose



## D3-les Mixins

---

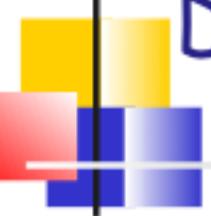
- Un mixin est un module que l'on utilise pour ajouter des fonctionnalités à une classe.
  - Souvent les Mixins fournissent un ensemble de méthode définissant un protocol commun à plusieurs classes.
- Par exemple, le module Comparable fournit un grand nombre d'opérateurs de comparaisons (<, <=, >, between? etc.).
  - Il définit ces opérateurs à partir de l'utilisation d'une méthode général ( $<=>$ ) de comparaison qui doit être définie dans la classe qui l'inclut.



## D3-Exemple d'utilisation : Mixin

---

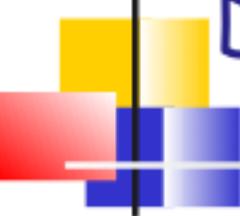
- On veut créer une classe (`MesAnim`) pour laquelle les comparaisons sont fondées sur le nombre de membres qu'un animal possède
- Il suffit que `MesAnim` définitse sa propre sémantique pour l'opérateur `<=>`, et déclare le module `Comparable` comme Mixin avec 'include'.



## D3-Exemple d'utilisation : Mixin

---

```
class MesAnim
  include Comparable      # C'est ici qu'il est 'Mix in'
  attr :membres
  def MesAnim.naitre(nom, membres)
    new(nom, membres)
  end
  def initialize(nom, membres)
    @nom, @membres = nom, membres
  end
  def <=>(o)
    return @membres <=> o.membres()
  end
  def inspect
    @nom
  end
end
```



## D3-Exemple d'utilisation : Mixin

---

- Toutes les méthodes de Comparable deviennent alors des méthodes de la classe MesAnim et un grand nombre de fonctionnalités sont ajoutées d'un coup.
  - Comme ce module est utilisé par de nombreuses classes, notre classe utilise la même sémantique que toutes les autres pour les opérations du même type.

```
class MesAnim
  include Comparable # C'est ici qu'il est 'Mix in'
  attr :membres
  def initialize(nom, membres)
    @nom, @membres = nom, membres
  end
  def <=>(o)
    return @membres <=> o.membres()
  end
  def inspect
    "@nom
    end
end
```



## D3-Exemple d'utilisation : Mixin

---

- Et on peut donc écrire des choses comme

```
c = MesAnim.naitre('chat', 4)
```

```
s = MesAnim.naitre('serpent', 0)
```

```
p = MesAnim.naitre('perroquet', 2)
```

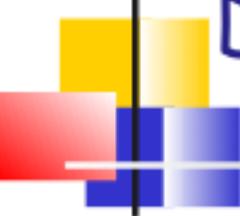
```
puts c < s          #=> false
```

```
puts s < c          #=> true
```

```
puts p >= s         #=> true
```

```
puts p.between?(s,c) #=> true
```

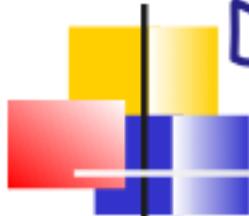
```
p [p, s, c].sort      #=> [serpent, perroquet, chat]
```



## D4-Retour sur les Collections

---

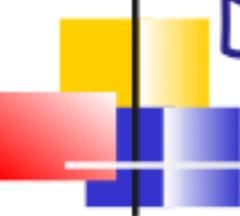
- Les Classes Collections “Implémentent” toutes en Ruby le Mixin “**Enumerable**”.
  - Dans toutes ces classes les méthodes :  
collect, detect, each\_with\_index, entries,  
find, find\_all, grep, include?, map, max,  
member?, min, reject, select, sort, to\_a  
Sont utilisables de la même façon
- Exemple
  - tab=[1,2,3,4,5,6]
  - tab.select { |el| el % 2 == 0} =>
  - tab.reject { |el| el % 2 == 0} =>



## D4-Retour sur les Collections

---

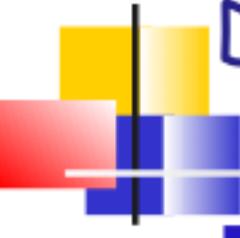
- Les Classes Collections “Implémentent” toutes en Ruby le Mixin “**Enumerable**”.
  - Dans toutes ces classes les méthodes :  
collect, detect, each\_with\_index, entries,  
find, find\_all, grep, include?, map, max,  
member?, min, reject, select, sort, to\_a  
Sont utilisables de la même façon
- Exemple
  - tab=[1,2,3,4,5,6]
  - tab.select { |el| el % 2 == 0} => [2,4,6]
  - tab.reject { |el| el % 2 == 0} => [1,3,5]



## D4-Retour sur les Collections

---

- Pour pouvoir disposer dans vos classes containers des facilités de parcours offertes par Ruby,
  - Il suffit que la classe "implémente" le Mixin Enumerable et définisse une méthode each ()
    - collect, detect, each\_with\_index, entries, find, find\_all, grep, include?, map, member?, reject, select, to\_a
  - Si la classe dispose en plus de la méthode <=> de comparaison, vous aurez accès également à toutes les méthodes de recherche ordonnée
    - min, max, sort.



## D5-Le Module Enumerable

---

e.collect{ |x| ... } ou e.map{ |x| ... }

- Renvoie un tableau contenant les résultats de l'exécution du bloc sur chaque élément de e

- Exemple sur un tableau

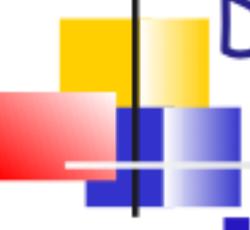
```
[1,2,3,4].collect { |x| x*x}  
=>
```

- Exemple sur une table de Hashage

```
h={1,2,3,4,5,6}           =>  
h.collect { |cle,val| cle*cle}  =>
```

- Exemple sur une chaîne

```
s="Jacoboni est un grand Homme"  
s.split("//).collect { |c| c.upcase }.join  
=>
```



## D5-Le Module Enumerable

---

`e.collect{ |x| ... } ou e.map{ |x| ... }`

- Renvoie un tableau contenant les résultats de l'exécution du bloc sur chaque élément de e

- Exemple sur un tableau

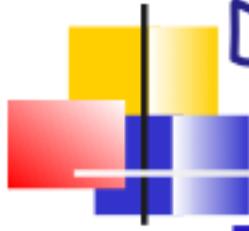
```
[1,2,3,4].collect { |x| x*x}  
=> [1,4,9,16]
```

- Exemple sur une table de Hashage

```
h={1,2,3,4,5,6}           => {5=>6, 1=>2, 3=>4}  
h.collect { |cle,val| cle*cle}  => [25,1,9]
```

- Exemple sur une chaîne

```
s="Jacoboni est un grand Homme"  
s.split("//).collect { |c| c.upcase }.join  
=> "JACOBONI EST UN GRAND HOMME"
```



## D5-Le Module Enumerable

---

- `e.detect{|x| ...}` ou `e.find{|x| ...}`

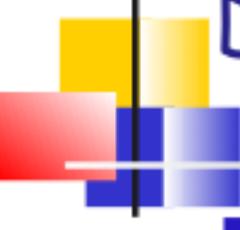
- Renvoie le 1er élément pour lequel le bloc renvoie true

```
# detecter le 1er élément débutant par m  
["truc", "machin"].detect{|s| ^m =~ s}  
=>
```

- `e.select{|x| ...}`

- Renvoie un tableau des éléments pour lesquels le bloc renvoie true

```
["truc", "machin", "bidule"].select{|s| /.*u =~ s}  
=>
```



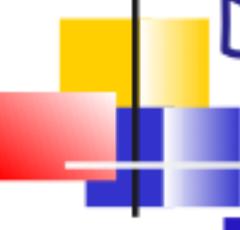
## D5-Le Module Enumerable

---

- `e.detect{|x| ...}` ou `e.find{|x| ...}`
  - Renvoie le 1er élément pour lequel le bloc renvoie `true`

```
# detecter le 1er élément débutant par m
["truc", "machin"].detect{|s| ^m =~ s}
=> "machin"
```
- `e.select{|x| ...}`
  - Renvoie un tableau des éléments pour lesquels le bloc renvoie `true`

```
["truc", "machin", "bidule"].select{|s| /.*u =~ s}
=> ["truc", "bidule"]
```



## D5-Le Module Enumerable

---

### La fabuleuse méthode inject

- permet d'accumuler une valeur entre les membres d'une collection.

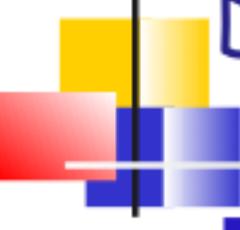
- Exemples:

```
# cumuler tous les éléments
```

```
[1,3,5,7].inject(0) { |sum, element|  
  sum+element }  
# => 16
```

```
# faire le produit, de tous les éléments
```

```
[1,3,5,7].inject(1) { |product, element|  
  product*element }  
# => 105
```



## D5-Le Module Enumerable

---

### La fabuleuse méthode inject (suite)

- Si on ne fournit pas d'argument le premier élément de la collection est utilisé pour l'initialisation

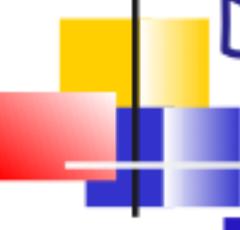
- Exemples:

```
# cumuler tous les éléments
[1,3,5,7].inject {|sum, element|
  sum+element }
```

# => 16

```
# faire le produit, de tous les éléments
[1,3,5,7].inject {|product, element|
  product*element }
```

# => 105



## D5-Le Module Enumerable

---

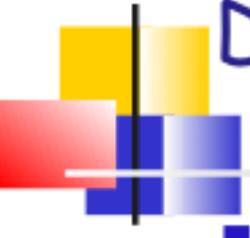
### La fabuleuse méthode inject (fin)

- vous pouvez également donner le nom de la méthode que vous souhaitez appliquer aux éléments successifs de la collection.

```
[1,3,5,7].inject(:+)  
# => 16
```

```
[1,3,5,7].inject(:*)  
# => 105
```

- Ces exemples fonctionnent parce que, en Ruby, addition et la multiplication sont simplement des méthodes sur les nombres, et `:+` est le symbole correspondant à la méthode `+`.



## D5-Le Module Enumerable

---

- e.each\_with\_index { |x, i| ... }

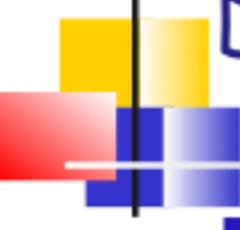
- Exécute le bloc pour chaque élément de e

```
["truc","machin"].each_with_index { |x,i|
    printf "%d : %s\n", i, x
}
0 : truc
1 : machin
```

- e.entries ou e.to\_a

- Renvoie un tableau contenant les éléments qui lui sont passés par e.each

```
{1,2,3,4,5,6}.entries
=> [[5, 6], [1, 2], [3, 4]]
```



## D5-Le Module Enumerable

---

- e.reject { |x| ... }

- Renvoie un tableau des éléments pour lesquels le bloc renvoie false.

```
["truc", "machin", "bidule"].reject{|s| /.*u/ =~ s}  
=>
```

- e.min et e.max

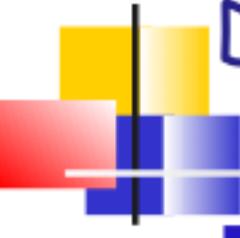
- Renvoie la valeur min (max)

```
[1,2,3,4].max    =>  
{1,2,3,4}.min    =>
```

- e.sort

- Renvoie un tableau des éléments de e triés

- Un bloc peut être utilisé pour effectuer les comparaisons.



## D5-Le Module Enumerable

---

- e.reject { |x| ... }

- Renvoie un tableau des éléments pour lesquels le bloc renvoie false.

```
["truc", "machin", "bidule"].reject{|s| /.*u/=~s}  
=> ["machin"]
```

- e.min et e.max

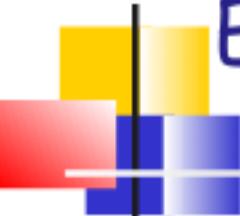
- Renvoie la valeur min (max)

```
[1,2,3,4].max => 4  
{1,2,3,4}.min => [1,2]
```

- e.sort

- Renvoie un tableau des éléments de e triés

- Un bloc peut être utilisé pour effectuer les comparaisons.



## E-Des Exemples

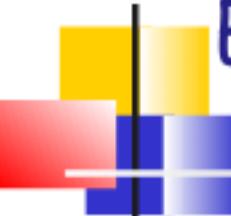
---

- Ajouter à la classe Array un itérateur qui permet de parcourir le tableau à l'envers :
  - Exemple d'utilisation

```
a=[1,2,3,4,5,6]
a.enversEach {|elt| print elt,":"}
6:5:4:3:2:1:
```

- EnversEach

```
class Array
  def enversEach
    copie=self.reverse
    copie.each {|elt| yield elt}
  end
end
```



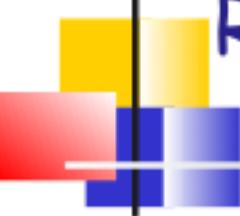
## E-Des Exemples

---

- Pour sélectionner par index dans un Array
  - Sélection de valeur si son index satisfait le bloc.
  - Exemple d'Utilisation
- **selectIndice**

```
[60, 62, 65, 70, 72, 73, 75].selectIndice { |i| i%2==0}  
=> [60, 65, 72, 75]
```

```
class Array  
  def selectIndice  
    resultat=[]  
    self.each with_index do |ele,idx|  
      resultat.push ele if yield idx  
    end  
    resultat  
  end  
end
```



# Reprise des Exos du TD2

---

## ■ La Factorielle

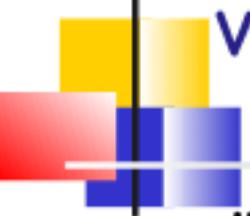
```
# Version 2 en utilisant un itérateur sur
# la collection générée par 1..self
def maFactoV2
    facto=1
    (1..self).each do |i|
        facto*=i
    end
    return facto
end
```



## voyelleEnMajuscule

---

```
class String
  # Version destructrice, la chaîne
  # receveuse est modifiée
  def voyelleEnMajuscule!
    for i in 0..self.length-1 do
      c=self[i]
      if "aeiouy".index(c) !=nil
        self[i]=c.chr.upcase
      end
    end
    return self
  end
```



## voyelleEnMajuscule

---

### # Version non destructrice

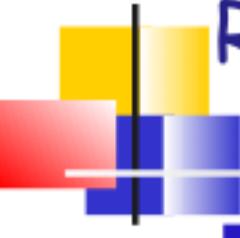
```
def voyelleEnMajuscule
    self.scan(/./).collect { |c|
        if "aeiouy".index(c) != nil
            c.upcase
        else c
        end
    }
end
```



## voyelleEnMajuscule

---

```
# une autre méthode rusée (non destructrice)
# presque la même renvoie un tableau
def voyelleEnMajuscule1
    return self.split("//).collect { |c|
        if c =~ /[aeiouy]/
            c.upcase
        else
            c
        end
    }
end
```

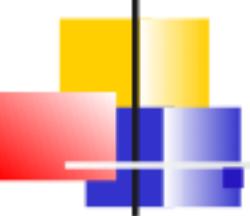


## Reprise de l'Exercice du TP1

---

Des maths (si peu !!). Ajoutez à la classe Integer de Ruby les trois méthodes suivantes :

- `listeDesDiviseurs()` qui retourne dans un tableau la liste des diviseurs de a.
  - Exemple pour  $a=6$  on obtient la liste  $(1, 2, 3, 6)$
- `listeDesNonsDiviseurs()` qui retourne dans un tableau la liste des non diviseurs de a.
  - Exemple pour  $a=6$  on obtient la liste  $(4, 5)$
- `RapDivNonDiv()` calcule et retourne le rationnel représentant le rapport entre la somme des diviseurs de a (inférieurs à a) et la somme des nonsDiviseurs de a.
  - pour  $a = 6$  le résultat est  $6/9$  :
    - 6 a pour diviseurs inférieurs à 6 : 1, 2, 3 et la somme de ceux-ci vaut 6
    - 6 a pour non diviseurs inférieurs à 6 : 4 et 5 et la somme de ceux ci vaut 9

- 
- On souhaite manipuler ce rapport sous la forme d'un nombre rationnel, donc de la forme  $a/b$  où  $a/b$  est une fraction irréductible ( $a/b$  est irréductible si elle n'est plus simplifiable). Dans l'exemple précédent on préfèrera donc afficher  $2/3$  plutôt que  $6/9$   
Pour cela on vous demande de donner le code de la classe NR qui fournit quelques opérations simples sur les nombres rationnels. Les opérations retourneront toujours les résultats sous forme de fractions irréductibles.
  - La classe NR permettra de réaliser les opérations suivantes :
    - Construire un rationnel irréductible associé au nombre rationnel  $a/b$ .
    - NR.simplifier(6, 9) permet de créer un rationnel dont le numérateur est 2 et le dénominateur 3
    - ajouter, multiplier, soustraire et diviser par un rationnel
    - afficher un rationnel et déterminer si deux rationnels sont égaux.

## ##### Programme de Test #####

```
puts "\n\tTest de la classe Diviseur"
puts "\t===="
print "Les Diviseurs de 1 à 10 : "
p Diviseur.listeDesDiviseurs(10)
print "Les Nombres non diviseurs de 1 à 10 : "
p Diviseur.listeDesNonsDiviseurs(10)
print "Le rapport des diviseurs de 1 à 10 : "
puts Diviseur.rapDivNonDiv(10)

puts "\n\tTest de la classe Simplifieur"
puts "\t===="
a=NR.simplifier(12/4)
b=NR.simplifier(12/6)
c=NR.simplifier(12/8)
d= NR.simplifier(12/12)
puts "(#{a}) + (#{b}) = #{e= a+b}" # Sucre Syntaxique e=a.+b
puts "(#{e}) - (#{c}) = #{f= e-c}" # Sucre Syntaxique f=e.-c
puts "(#{f}) / (#{d}) = #{g= f/d}" # Sucre Syntaxique g=f./d
puts "(#{e}) / (#{c}) = #{g / g}" # Sucre Syntaxique g./g
```