JAVA FONCTIONNEL

LES COURS DU MODULE « JAVA FONCTIONNEL » DU SEMESTRE 3

KILLIAN REINE

Licence 2
orientation Informatique

2024-2025

TABLE DES MATIÈRES

PARTIE 1

BASES DU LANGAGE HASKELL

1.1) Paradigme impératif, et rappels

Rappel de cours 1

- ➢ Un langage impératif est un type de langage de programmation où **un programme est essentiellement une liste d'instructions** qui décrivent pas à pas les actions à réaliser par l'ordinateur.
- ➢ Une instruction est un ordre donné à l'ordinateur.
- ➢ Une variable permet de stocker une valeur.
- ➢ L'affectation consiste à associer une information à une variable.
- ➢ L'état d'un programme désigne l'ensemble des valeurs des variables et structures de données à un moment donné dans l'exécution du programme.
  Une instruction permet de passer d'un état e_1 à un état e_2. Il peut arriver que celle ci **ne change pas l'état** ($e_1 = e_2$), c'est le cas lorsqu'on affiche la valeur d'une variable à l'écran. On appelle ceci des effets de bord.
- ➢ La trace d'exécution désigne la succession des états durant l'exécution du programme.

Exemple
Vous connaissez déjà pas mal de langages impératifs : Python, Processing, Shell, Assembleur, ...

DÉFINITION *(donnée)*

Les données sont l'objet de l'existence de l'informatique (étymologiquement, informatique vient de la contraction de « information » et de « automatique ». Il s'agit donc de pouvoir traiter automatiquement des informations).
On dit alors que l'on peut voir une donnée comme une **information structurée**.

Rappel de cours 2

Les données sont **typées**. Nous avons déjà vus les nombres, les booléens, les chaînes de caractères, ...

Remarque | Une **fonction** est un outil qui permet de transformer ou manipuler des données.

1.2) Paradigme fonctionnel

❑ **Origine**
Travail d'*Alonzo Church* dans les années 1932, en définissant le λ-calcul.

❑ **Idée principale**
Les **fonctions sont des données** comme les autres, on doit alors pouvoir les manipuler, les passer en paramètre à d'autres fonctions, ...

❑ **Conséquences**
On a dû trouver une façon de considérer une fonction comme une donnée. Comme les autres données, une fonction n'a pas de nom propre. Le « nom » d'une fonction correspond en réalité au nom de la variable dans laquelle elle est enregistrée.

Exemple mathématiques
Pour définir une fonction quelconque :

$$\begin{array}{rcl} f:\mathbb{R} & \rightarrow & \mathbb{R} \\ x & \mapsto & x+1 \end{array}$$

Ici, on a définit la fonction $f(x) = x+1$.
Utlisation : $f(2) = 3$.

Notions introduites par Church
Définition d'une fonction

$$\lambda x.(x+1)$$

Exemple d'utilisation avec $x = 2$.

$$(\lambda x.(x+1))(2)$$

Exemples plus complexes

✍ Passage d'une fonction en paramètre : $\lambda(f,x).(f(x+1))$

✍ Définir une fonction renvoyant une fonction : $\lambda x.(\lambda y.(x+y))$
$(\lambda x.(\lambda y.(x+y)))(1) = \lambda y.(1+y)$

❑ **Seconde idée**
Un programme est une composition de fonctions. Il n'y a donc plus de notions d'états.

❑ **Conclusion**

❍ Il n'y a plus de variable au sens où on l'entend. Désormais, **une variable est une donnée immuable** (non-modifiable), comme en mathématiques, on construit des objets à partir d'autres données.

❍ Il n'y a plus de boucle, on doit passer par l'appel de fonction (récursivité).

❍ Il n'y a `if then else` classique, désormais en programmation fonctionnelle, le `if then else` à valeur d'expression, comme le ternaire de Java par exemple.

1.3) Langage fonctionnel - Haskell

1.3.1) Opérations de base et inférence de type

L'addition / Soustraction de deux nombres
Soient `a` et `b` deux nombres.
Lorsque l'on veut les additionner en Haskell, il suffit d'écrire :

```
ghci> a + b
ghci> a - b
```

Le produit / Quotient de deux nombres
Lorsque l'on veut multiplier deux nombres entre eux, il suffit décrire :

```
ghci> a*b
ghci> a / b
```

Puissance d'un nombre
Lorsque l'on veut multiplier `n` fois le nombre `a` (« le mettre à la puissance `n` ») :

```
ghci> a^n
```

Remarque

Pour noter un nombre négatif en haskell, on utilise la notation `-a`, cette dernière possède quelques subtilités d'utilisations.
Exemple

```
ghci> -3
-3
ghci> -3 + 8
5
ghci> 8 + - 3
ERREUR
```

Le − peut servir pour dire qu'un nombre est négatif mais est aussi utilisé pour effectuer des soustractions, il faut alors mettre des parenthèses.

```
ghci> 8 + (-3)
5
```

Affichicher la signature de type d'une expression
Pour afficher la signature de type d'une expression, il suffit d'écrire la comande `:t`

```
ghci> :t expression
```

Exemples

- L'expression `succ a` permet de rajouter 1 au nombre `a`. (=incrémenter `a`)

```
ghci> succ 6
7
ghci> :t succ
succ :: Enum a => a -> a
```

Quelques explications s'imposent pour comprendre la signification de la « signature de type » de la fonction `succ`.

➢ `Enum a` signifie que la fonction `succ` prend en paramètre une valeur `a` dont le type doit faire partie de la classe `Enum`.

Rappel de cours 3

> La classe `Enum` englobe les types `Int`, `Char`, `Bool`, ...

➢ `a -> a` signifie alors que la fonction `succ` prend un élément de type `a` *(appartenant donc à la classe Enum)* et renvoi un élément du même type que `a`.

- L'expression `tail` permet d'afficher une liste sans son premier élément.

```
ghci> tail [1, 2, 3, 4]
[2, 3, 4]
ghci> :t tail
tail :: GHC.Stack.Types.HasCallStack => [a] -> [a]
```

Alors, `tail` prend en argument une liste d'élément `a` appartenant à une classe de type particulière et elle renvoie une liste d'éléments du même type que celle de `a`. *On ne se préoccupe pas de l'expression « GHC.Stack.Types.HasCallStack »*

- Chaine de caractères

```
ghci> :t "bonjour"
"bonjour" :: String
```

Signifie simplement que le type de "bonjour" est contenu dans la classe `String`.

- Liste d'entier, de caractères

```
ghci> :t [1, 2, 3]
[1, 2, 3] :: Num a => [a]
ghci> :t ['1', 'c', 'd']
['1', 'c', 'd'] :: [Char]
```

➢ `[1, 2, 3]` est une liste d'élément d'un certain type `a`. En Haskell, tout est homogènre, c'est à dire que les éléments d'une liste doivent tous être du même type.
D'où la contrainte `Num a` qui signifie que `a` est d'un type contenu dans la classe `Num` (= type numérique) comme `Int, Integer, Foat, Double`.
➢ `['1', 'c', 'd']` est simplement une liste d'élément de type contenu dans la classe `Char`.

1.3.2) Écrire des fonctions sur le terminal de commandes

Contexte
Les fonctions suivantes sont codées dirrectement dans le terminal.
Notre objectif vas être de coder une fonction `double`, et une autre `somme` que l'on améliorera pour diverses utilisations.

(1) La fonction `double`
Le principe est le suivant : *la fonction prend en paramètre un nombre et renvoi son double.*

```
ghci> double x = 2*x
ghci> double 6
12
```

Nous pouvons compliquer l'utilisation de la fonction double. Maintenant, il faut utiliser cette fonction pour renvoyer le double de `(8+1)*(6+1)`.

```
ghci> double (8+1)*(6+1)
126
```

Pourquoi au lieu de faire `8+1` et `6+1` nous n'utilisons pas la méthode `succ` vue précédemment qui permet d'incrémenter un nombre de 1.

```
ghci> double succ 8 * succ 6
ERREUR
```

Ici, vous obtenez une erreur car votre fonction `double` ne prend qu'un paramètre. Or les deux appels à la méthode `succ` sont interprétés comme deux appels différents donc deux paramètres. Il faut donc mettre des parenthèses.

```
ghci> double (succ 8 * succ 6)
126
```

Un petit mot sur le type de la fonction `double`.

```
ghci> :t double
double :: Num a => a -> a
```

Alors, la fonction `double` prend en paramètre `a` d'un type cointenu dans la classe `Num` et renvoie un élément du même type.

(2) La fonction `somme`

Ici le principe reste simple : *Prendre deux nombres et les additionner entre eux*

```
ghci> somme x y = x+y
ghci> somme 3 5
8
ghci> somme succ 1 8
ERREUR
ghci> somme (succ 1) 8
10
ghci> somme (succ 1) succ 7 - ou somme succ 1 (succ 7)
ERREUR
ghci> somme (succ 1) (succ 7)
10
- Utilisation plus avancée
ghci> somme (succ 1 + 2 * head [5, 6, 8]) (succ 3*6+45)
81
```

Structure de la dernière instruction

$$\texttt{somme}\ \underbrace{\texttt{(succ 1 + 2 * head [5, 6, 8])}}_{\text{paramètre 1}}\ \underbrace{\texttt{(succ 3 * 6 + 45)}}_{\text{paramètre 2}}$$

Voici son fonctionnement

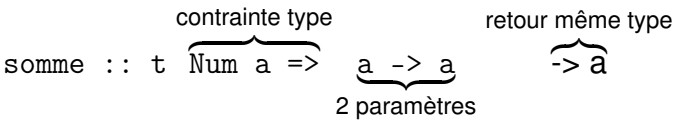
- `succ 1 + 2 * head [5, 6, 8]`
  On incrémente 1 et on récupère le premier élément de la liste 5
  On a : $2+2*5=2+10=12$
- `succ 3*6+45`
  On incrémente 3 de 1 puis on calcul $4*6=24$ on ajoute 45.
  on a alors : $24+45=69$
- Enfin c'est comme si on appelait `somme 12 69` qui donne 81.

$$\texttt{somme (succ 1 + 2 * head [5, 6, 8]) (succ 3*6+45)} \Longleftrightarrow \texttt{somme 12 69}$$

Un petit mot concernat le type de la fonction `somme`

```
ghci> :t somme
somme :: Num a => a -> a -> a
```

Il y a une contrainte de type sur le.s paramètre.s (`Num a =>`) puisqu'il doit appartenir à la classe `Num`. Ensuite `a -> a -> a` signifie que la fnction prend deux paramètres du même type et renvoi un résultat du même type que les paramètres.
Structure du type



1.3.3) Test en Haskell

Structure générale

```
if a then b else c
```

où

- `a` est le test effectué
- `b` l'instruction a effectué en cas de réalisation du test
- `c` l'instuction a effectué en cas d'échec du test

Remarque

- ➙ L'instruction `if` possède un type, celui de `b` et `c`
- ➙ Cela sous-entend donc que `b` et `c` sont de même type et qu'ils doivent toujours être définis.
  Contrairement à python, C ou autre, la condition `else` est alors obligatoire.

Exemple
On souhaite savoir si un nombre est pair, si c'est le cas on renvoi « nombre pair » sinon « nombre impair ».

```
ghci> nombrePair x = if mod x 2 == 0 then "nombre pair" else "nombre impair"
ghci> nombrePair 63
"nombre impair"
```

Dans notre cas, on s'apperçoit bien que `b` et `c` sont définis et sont de même type `String`.
Qu'en est-il du type de ce test ?

```
ghci> :t nombrePair
nombrePair :: Integral a => a -> String
```

Ici, ma condition prend un élément d'un type contenu dans la classe `Integral`, elle ne prend qu'un paramètre et renvoi une chaine de caractères.

1.3.4) Définition de fonctions par filtrage

Remarque

Définir une fonction par filtrage n'est pas possible directement via le terminal il faut laors passer par un fichier, donc par un programme.

DÉFINITION *(Pattern matching)*

Le **Pattern matching** (ou filtrage) en Haskell permet de définir une fonction, effectuant des opérations différentes selon les arguments (=paramètres) qui lui ont été donnés.
Chaque « partern » permet alors de décrire les valeurs de l'argument ainsi, la fonction applique le code associé à ce dernier.

Exemple

1. **Fonction** `Zero`
   Codons une fonction par filtrage qui permet de renvoyez "Zéro" si le nombre entré est 0 sinon renvoyer "Non-zéro".

```haskell
zero :: (Eq a, Num a) => a -> String
zero 0 = "Zéro"
zero n = "Pas zéro"

main :: IO ()
main = do
    print (zero 0)
    print (zero 5)
```

```
"Zéro"
"Pas zéro"
```

   Explications

   - J'ai spécifier la signature de type de ma fonction
     Elle prend en argument un élément d'un type appartenant sois à la classe `Eq`, sois à la classe `Num`, et elle renvoie une chaine de caractères `String`.
   - J'ai spécifier le retour des différents cas :
     ➙ Si $n = 0$, renvoyer "Zéro"
     ➙ Si $n \neq 0$, renvoyé "Pas zéro"
   - J'ai terminé en définissant ce que fera le programme principal, ici il testera la fonction `zero`.

2. **Fonction** `discriminant`
   Codons une fonction qui permet de savoir quel est le nombres de racines d'un polynôme de degré 2 en utilisant son discriminant.

```haskell
discriminant :: (Ord a, Num a) => a -> String
discriminant delta
    | delta > 0     = "Deux racines réelles"
    | delta < 0     = "Aucune racine réelle"
    | otherwise     = "Une racine réelle"

-- Utilisations possible --
print (discriminant 0)
print (discriminant 5)
print (discriminant (-3))
```

```
"Une racine réelle"
"Deux racines réelles"
"Aucune racine réelle"
```

1.3.5) Un peu de récursivité

Rappel de cours 4

> Une fonction est dite récursive lorsqu'elle s'appelle elle-même.

Exemple

1. **Fonction** `u`
   Ici, on code une suite mathématique définie par récurrence, elle renverra 0 si $n = 0$ et $2u_{n-1}+3$ si ce n'est pas le cas.

$$\forall n \in \mathbb{N},\ u_n = \begin{cases} u_0 & = 0 \\ u_{n+1} & = 2(u_{n-1})+3 \end{cases}$$

   (a) VERSION 1
   On défini la fonction de manière récursive.

```haskell
u :: Int -> Int
u 0 = 1
u n = 2*u(n-1)+3

-- Utilisation --
print (u 4)
```

```
125
```

   (b) VERSION 2
   On défini la fonction avec un opérateur ternaire `if then else`.

```haskell
u n = if n == 0 then 0 else 2*u(n-1)+3
```

2. **Fonction** `sommeREP`
   On souhaite calculer la somme des éléments 0 à n notée $\sum_{i=0}^{n}$.

```haskell
sommeREP :: Int -> Int
sommeREP 0 = 0
sommeREP n = n + sommeREP (n - 1)

main :: IO ()
main = do
    -- let pour définir une variable --
    let n = 5
    print (sommeREP n)
```

```
15
```

1.3.6) Les listes en Haskell

DÉFINITION *(liste)*

En haskell, une **liste** permet de stocker un certain nombre de variables du même type.
Structure générale :

```
[elem1, elem2, elm3, ..., elemn]
```

1. **Liste vide**
   La liste vide est donnée sous la forme `[ ]`
2. **Liste en extension**
   La liste en extension est donnée sous la forme `[elem1, elem2, elm3, ..., elemn]`.
3. **Liste en adjonction entête**
   Permet de rajouter un élément en tête de liste, `elem : liste`
4. **Concaténation**
   Pour ajouter deux listes entre-elles, il suffit d'utiliser l'oppérateur ++.

   ```
   liste1 ++ liste2
   ```

5. **Accès à l'élément** i on utilise l'instruction `liste !! i`
6. `head` permet de retourner le premier élément d'une liste.
7. `tail` permet d'afficher une liste sans son premier élément.

Remarque

- `head[ ]` déclanche une erreur, on ne peut pas récupérer le premier élément d'une liste vide.
- `tail[ ]` déclanche une erreur, on ne peut enlever un élément d'une liste qui est déjà vide.

```
ghci> head[ ]
ERREUR
ghci> tail[ ]
ERREUR
```

8. **Récupérer la taille d'une liste** avec `length`

Exemples d'utilisations dans le terminal

```
ghci> 1 : [2, 3]
[1, 2, 3]
ghci> [1, 2, 3, 4] ++ [5, 6, 7]
[1, 2, 3, 4, 5, 6, 7]
ghci> [8, 4, 9, 5]!!2
9
ghci> [ ]!!0
ERREUR
ghci> head[1, 2, 3]
1
ghci> tail[1, 2, 3]
[2, 3]
ghci> length[45, 78, 12, 54]
4
length[ ]
0
```

1.3.7) Traitement récursif sur les listes

Objectifs

- Somme d'éléments d'une liste
- Ajouter 1 aux élément d'une liste
- Doubler chaque élément d'une liste

Faire la somme des éléments d'une liste

1. **Identification des différents cas**

   Rappel de cours 5

   > Une liste est une stucture permettant de stocker des données du même type. Elle peut être définie avec des valeurs ou alors vide.

   Deux cas se distinguent alors :

   - ❏ Le cas d'une **liste vide**, on renverra alors `0`
   - ❏ Le cas d'une **liste non-vide**

2. **Le programme**

```
1 sommeListe :: Num a => [a] -> a
2 sommeListe [] = 0
3 sommeListe (x:y) = x + sommeListe y
4
5 print (sommeListe [])
6 print (sommeListe [1, 5, 8])
```

```
15
```

3. **Explication de l'algorithme**

   Le cas de base est lorsque la liste est vide, on renvoie alors 0.
   Pour les autres cas :

   - ➙ `x` représente le *premier élément de la liste*
   - ➙ `y` représente le *reste de la liste*

   On ajoute alors à la somme récursive les éléments restant dans `y`.
   Voici ce que fait réellement le programme.
   Soit `L = [2, 3, 4]` la liste étudiée.

```
sommeListe [2, 3, 4] = 2 + sommeListe [3, 4]
sommeListe [2, 3, 4] = 2 + (3 + sommeListe [4])
sommeListe [2, 3, 4] = 2 + (3 + (4 + sommeListe [ ])
sommeListe [2, 3, 4] = 2 + (3 + (4 + 0))
sommeListe [2, 3, 4] = 2 + 3 + 4 + 0 = 9
```

Ajouter 1 à chacun des éléments de la liste

1. **Objectif**

   Accéder à chacun des éléments d'une liste puis les incrémenter de 1.

```
increment [1, 2, 3, 4] = [2, 3, 4, 5]
```

   Voilà le résultat que nous voulons obtenir.

2. **Le programme**

```
1 increment :: Num a => [a] -> [a]
2 increment [ ] = [ ]
3 increment (x:y) = (x+1) : increment y
4
5 print (increment [])
6 print (increment [1, 5, 8])
```

```
[2, 6, 9]
```

3. **Explications de l'algorithme**
   De la même manière que pour la fonction précédente, `x` représente le premier élément de la liste et `y` le reste.
   Lorsque l'on appelle la fonction, on récupère le premier élément, on lui ajoute 1 et on appelle récursivement la fonction sur l'élément suivant et ainsi de suite.
   Voici ce que fait réellement le programme.

```
increment [1, 2 ,3] = (1+1) : increment [2, 3]
increment [1, 2 ,3] = (1+1) : ((2+1) : increment [3])
increment [1, 2 ,3] = (1+1) : ((2+1) : ((3+1) increment [ ]))
increment [1, 2 ,3] = (1+1) : (2+1) : (3+1) : [ ]
increment [1, 2, 3] = 2 : 3 : 4 : [ ]
increment [1, 2, 3] = [2, 3, 4]
```

Doubler chaque éléments d'une liste
ici, exactement la même chose que pour ajouter 1, sauf que là, on met $\times 2$.

```
1 doubler :: Num a => [a] -> [a]
2 doubler [ ] = [ ]
3 doubler (x:y) = (x*2) : doubler y
4
5 print (doubler [])
6 print (doubler [1, 5, 8])
```

```
[2, 10, 16]
```

Exercice

Tours de Hanoi.
Ajouter au cours lors de la prochaine MAJ

Pour aller plus loin sur les listes

- ➙ `last [...]` permet de retourner le dernier élément d'une liste.
- ➙ `init [...]` permet de retourner la liste **privé de son dernier élément**.
- ➙ `take n [...]` premet de prendre les n premiers éléments d'une liste.
- ➙ `drop n [...]` permet de supprimer les n premiers éléments d'une liste.

1.3.8) Fonction d'ordre supérieur

DÉFINITION *(fonction d'ordre supérieur)*

On appelle **fonction d'ordre supérieur** une fonction paramétrée par au moins une autre fonction.

1. **La fonction** `map`

Remarque Les traitements récursifs vus précédemments (`incrementer`, `doubler`) sont similaires, à l'opération. On vas alors passer cette opération en paramètre en plus de la liste, c'est la **fonction** `map`.

(a) Fonction `traitement`

La fonction `traitement` permet de prendre deux arguments `f` une fonction prenant un élément de type `a` et renvoie une liste d'éléments de type `b` ainsi qu'une liste d'élément de type `a`.

La fonction effectue donc la fonction `f` sur les éléments d'une liste.

```
1 traitement :: (a -> b) -> [a] -> [b]
2 -- cas d'une liste vide --
3 traitement f [ ] = [ ]
4 -- cas d'une liste non-vide --
5 traitement f (x:y) = f x : traitement f y
```

(b) Réécriture de `increment` **avec la fonction** `traitement`

```
1 incrementIntermediaire :: [Int] -> [Int]
2 incrementIntermediaire xs = traitement (\x -> x + 1) y
```

(c) Différence avec `map`

```
1 incrementMap :: [Int] -> [Int]
2 incrementMap y = map (+1) y
```

DÉFINITION *(map)*

La fonction `map` est une *fonction d'ordre suppérieure* qui permet d'appliquer une fonction à chacun élément d'une liste.

Signature de type

```
1 map :: (a -> b) -> [a] -> [b]
```

La fonction `map` prend deux arguments :

- ❑ `(a -> b)` Une fonction qui prend elle-même un argument de type `a` et renvoi un résultat de type `b`.
- ❑ `[a]` Une liste d'éléments de type `a`.

Et, elle renvoie une liste d'élément de type `b`.

Exemple
On veut prendre une liste d'entiers, puis on veut les multiplier tous par 2.

```
map (\ x -> x*2) [1, 2, 3, 4]
[2, 4, 6, 8]
```

Dans notre exemple, la signature de type est donnée par :

```
1 map :: (Int -> Int) -> [Int] -> [Int]
```

2. La fonction `filter`

DÉFINITION *(filter)*

La fonction d'ordre supérieure `filter` est utiliser pour « filtrer » les éléments d'une liste à l'aide d'une condition.
Cette dernière renvoie alors une liste d'élément qui respectent la condition établie.

Remarque La fonction `filter` permet de sélectionner les éléments d'une liste selon une condition, opération.

`filter (condition) liste`

Signature de type

```
1 filter :: (a -> Bool) -> [a] -> [a]
```

La fonction `filter` prend deux arguments :

- `(a -> Bool)` Une fonction dite « de filtrage » qui prend elle-même un argument de type `a` et renvoi un résultat de type `Bool`.
- `[a]` Une liste d'éléments de type `a`.

Et, elle renvoie la listedes éléments de type `a` respectant la condition de filtrage.

Exemple
Récupérer les nombres pairs d'une liste
Version terminal

```
filter (\x -> mod x 2 == 0) [3, 4, 7, 8, 12]
[4, 8, 12]
```

Version programme externe

```
1 -- Signature de type de la fonction --
2 estPair :: Int -> Bool
3 -- Corps de la fonction --
4 estPair x = x mod 2 == 0
5
6 -- Utilisation --
7 result = filter estPair [1, 7, 6, 4, 2, 0, 78, 41, 1]
8 print result
```

```
[6, 4, 2, 0, 78]
```

La fonction `filter` est donc appelée pour chaque élément de la liste. Si cette dernière renvoie `True`, l'élément est conservé dans la liste de retour. Sinon, l'élément est enlevé.

Exemple (avec les caractères)
On souhaite récupérer les voyelles dans une chaine de caractère.
Version 1 - Brutale

```
filter (\c -> c `elem` "aeiouAEIOU") "Bonjour vous"
oouou
```

Version 2 - Programme externe

```
1 estVoyelle :: Char -> Bool
2 estVoyelle c = c `elem` "aeiouAEIOU"
3
4 main = print (filter estVoyelle "bonjour vous")
```

```
oouou
```

1.3.9) Forme curryfiée

DÉFINITION *(currification)*

La **currification** en haskell est une technique qui permet de transformer uen focntion qui prend plusieurs arguments en **une suite de fonctions** ne prenant qu'un unique argument.

Prenons un exemple concrêt
Considérons la fonction `addition` suivante :

```
1 add :: Int -> Int -> Int
2 add x y = x + y
```

Cette dernière prend donc deux arguments de type entiers et renvoie la somme de ces deux derniers.
Méthode de currification

- ❏ Ne considérons que des fonctions ne prenant qu'un seul argument.
- ❏ La première fonction prend `x` comme argument et elle renvoie une autre fonction qui attend `y`
- ❏ La nouvelle fonction retourne alors `x+y`

```
1 -- On reprend la fonction initiale --
2 add :: Int -> Int -> Int
3 add x y = x + y
4
5 main = do
6     let fctIntermed = add 3
7     print (fctIntermed 5)
```

Explications du fonctionnement

1. On utilise `let` pour créer une variable
2. `fctIntermed` est une fonction qui appelle `add 3`
3. Grâce à la currification, lorsque l'on appelle `add 3`, nous fournissons un premier argument, on obtient alors une nouvelle fonction qui attend un second argument.
4. De manière générale on a :

```
1 let fctIntermed = 3 -- Revient à 3 + y --
```

5. Le fait ensuite d'appeler cette fonction intermédiaire et de lui donner `5` en argument fait que :

`fctIntermed 5 = 3+ 5` donc `8`

1.3.10) Pliage de liste, `foldl` et `foldl1`

DÉFINITION *(fold, foldl, foldr)*

La fonction `fold` (ou ses équivalents `foldl` ou encore `foldr` est utilisée pour résuire une liste à une seule valeur en appliquant une fonction de manière itérative.

Remarque | `foldl` signifie pliage « à gauche », `foldr` quant à lui pliage « à droite ».

Prenons un exemple concrêt
Exemple
Considérons la fonction suivante qui calcule la somme des éléments d'une liste

```
1 somme :: [Int] -> Int
2 somme liste = foldl (+) 0 liste
```

Fonctionnement de la fonction
On considère qu'elle prend une liste `L` de `n` éléments.

1. `(+)` c'est l'opération que l'on souhaite appliquer aux éléments de la liste.
   Ici c'est l'addition.
2. `0` c'est la valeur initiale du résultat
3. `foldl` signifie que l'on parcours la liste de gauche à droite.
4. `foldl` parcours alors toute la liste en ajoutant au résultat chaque élément de cette dernière un à un. Sachant que le résultat est initialisé à `0`.

Quelques exemples
Exemple 2, produit des éléments d'une liste

```
1 produit :: [Int] -> Int
2 produit liste = foldl (*) 0 liste
```

Exemple 3, concaténation de chaine
`foldl` fonctionne aussi sur les chaines de caractères.

```
1 concatenation :: [String] -> String
2 concatenation liste = foldl (++) "" liste
```

Remarque | Ne vous trompez pas entre `foldl` et `foldr`, si vous utilisez `foldr`, vous parcourrez la liste de droite à gauche, c'est à dire du dernier élément de la liste au premier.

DÉFINITION *(foldl1)*

`foldl1` est une variante de `foldl` qui est utilisée pour réduire une liste à une seule valeur en utilisant une **fonction binaire**.

Remarque | Contrairement à `fold`, `foldl1` n'a pas nécessairement besoin de valeur initiale. Le premier élément de la liste passée en argument sert de valeur initiale du résultat.

Signature de type

```
1 foldl1 :: (a -> a -> a) -> [a] -> a
```

La fonction `foldl1` prend deux arguments :

- ❏ `(a -> a -> a)` Une fonction qui prend elle-même deux arguments de type `a` et renvoi un résultat de type `Bool`.
- ❏ `[a]` Une liste d'éléments de type `a`.

Elle renvoie une valeur de type `a` pour résultat.
Exemple, maximum d'une liste

```haskell
maximumListe :: [Int] -> Int
maximumListe liste = foldl1 max liste

main :: IO ()
main = do
    -- Utilise foldl1 pour trouver le maximum --
    let resultat = maximumListe [4, 6, 10, 2, 8]
    print resultat  -- Affiche 10 --
```

Remarque

`fold1` échouera si la liste donnée est vide car il n'aura pas de valeur de retour. On a rappelé précédemment que la valeur de retour initiale est le premier élément de la liste passée en paramètre. Si il n'y a pas d'éléments dans la liste, `foldl1` ne renvoi rien donc une erreur.

1.3.11) Le type *maybe*

DÉFINITION *(maybe)*

En haskell, le type `maybe` est un type de donnée qui permet de représenter une valeur qui peut être présente ou absente.

Définition du type `maybe`

```haskell
data maybe a = Nothin | Just a
```

- ❏ `Nothing` représente l'absence de valeur
- ❏ `Just a` représente la présence d'une valeur de type `a`.

Exemple, une division bancale
On souhaite faire une fonction qui divise deux nombres mais ne peut pas diviser par 0.

```haskell
division :: Int -> Int -> Maybe Double
division _ 0 = Nothing
division x y = Just( fromIntegral x / fromIntegral y)

main :: IO ()
main = do
    print (division 10 2)  -- Affiche Just 5.0 --
    print (division 10 0)  -- Affiche Nothing --
```

Remarque

`fromIntegral x` permet de convertir un entier x en un nomble à virgule flottante.
`Maybe` peut être utiliser avec `foldl` et ses variante, puis avec `fmap`, `map`, `>>=`.

1.3.12) Une variante de *map*, le *fmap*

DÉFINITION *(fmap)*

La fonction `fmap` en haskell permet d'appliquer une fonction à une valeur à l'intérieur d'une structure de données qui est instance de la classe de type `Functor`.
Inclu des types comme les listes, `Maybe`, ...

Signature de type

```
1 fmap :: Functor f => (a -> b) -> f a -> f b
```

`fmap` prend deux arguments :

- `Functor f` impose que `f` doit être une structure de données appartenant à l'instance `Functor`
- `(a -> b)` une fonction qui prend en argument une valeur de type `a` et renvoi une valeur de type `b`.
- `f a` une structure de données contenant des éléments de type `a`

elle renvoie `f b`, une structure de données contenant des éléments de type `b`.

Exemple (1) sur les listes

```
1 fmap (+1) [1, 2, 3]
```

```
[2, 3, 4]
```

Exemple (2) avec `Maybe`

```
1 fmap (*2) Just 5
2 fmap (*2) Nothing
```

```
Just 10
Nothing
```

Remarque Pour le second exemple `fmap (2*) Nothing` ne prend même pas la peine de traiter la fonction puisque `Nothing` est passé en argument, rien est à transformer.

Rappel de cours 6

Un **foncteur** est une strucure de données qui peut être transformée à l'aide d'une fonction.
En haskell, un `functor` est défini par une classe de type `Functor`, qui permet d'appliquer une fonction à des valeurs contenues dans la structure de données sans la détruire.

1.3.13) Les monades

DÉFINITION *(monade)*

(1) Définition brute
Une **monade** est une abstraction qui représente une computation qui peut être enchaînée.

(2) Définition avec Haskell
En haskell, uen monade est définie par la classe de type `Monad`, qui fournit une interface pour travailler avec des valeurs encapsulées dans un contexte.

(3) Définition intuitive
Une monade permet de composer des opérations tout en gardant votre code propre et fonctionnel, comme un conteneur qui vous aide à gérer ce qui se trouve à l'intérieur.

Définition de la classe `Monad`

```
1 classe Applicative m => Monad m where
2     return :: a -> m a
3     (>>=)  :: m a -> (a -> m b) -> m b
```

Explications :

- `return` est une fonction qui prend une valeur et une place dans un contexte monadique.
  Exemple
  Pour une monade `Maybe`, `return 5` renverra `Just 5`
- (>>=) est l'opérateur de liaison qui prend une valeur encapsulée de type `m a` et une fonction qui transforme cette valeur en type `m b`.

Exemple avec `Maybe`

```
1 safeDivide :: Int -> Int -> Maybe Int
2 safeDivide _ 0 = Nothing
3 safeDivide x y = Just (x div y)
4
5 result :: Maybe Int
6 result = Just 10 >>= \x -> safeDivide x 2
7 print result
```

```
Just 5
```

Ici, >>= permet d'enchainer la division sécurisée, tout en gérant le cas où la division par 0 se produirais.

1.3.14) L'opérateur >>=

DÉFINITION *(>>=)*

L'opérateur >>= est l'opérateur de liaison pour les monades. Il permet de chainer les calculs monadiques ensemble.

Signature de la >>=

```
1 (>>=) :: Monad m => m a -> (a -> m b) -> m b
```

L'opérateur >>= :

- ❏ `Monad m =>` contraint l'opérateur >>=, en gros, l'opérateur >>= fonctionne pour tout `m` qui est une monade.
- ❏ `m a`, la monade d'entrée contient des éléments de type `a`.
- ❏ `(a -> m b)` une fonction prenant une valeur de type `a` extraitre de la monade `m a`, elle renvoie une nouvelle valeur monadique de typ `b m b`.

L'opérateur >>= retourne une monade de type `m b`. En gros, on part d'une monade de type `a` et on retourne une autre monade de type `b`.

Exemple

```
1 safeDivide :: Int -> Int -> Maybe Int
2 safeDivide _ 0 = Nothing
3 safeDivide x y = Just (x div y)
4
5 result :: Maybe Int
6 result = Just 10 >>= (\x -> safeDivide x 2) >>= (\y -> safeDiv y 0)
7 print result
```

Fonctionnement du programme

- `Just 10 >>= (-> safeDivide x 2)`
  Ici dans un contexte monadique `Just 10` est extrait alors $x = 10$
  On applique la fonction `safeDiv x 2` qui renvoie `5`.
- `Just 5 >>= (-> safeDiv y 0)`
  Toujours dans un contexte monadique, on extrait `Just 5` d'où `y=5`
  On applique la fonction `safeDiv y 0` qui renvoie `Nothing` cat le cas d'une division par `0` n'est pas défini.

```
Nothing
```

PARTIE 2

LES ASPECTS FONCTIONNELS DE JAVA

2.1) Rétrospective sur l'héritage en Java

Pour rappel Java est un langage de programmation orienté objet (POO), la notion d'héritage est donc un aspect important du langage. C'est pour cette raison que des rappels s'imposent :

DÉFINITION *(héritage)*

L'**héritage** permet de réutiliser du code déjà existant et de créer des relations hiérarchiques entre les classes.

Soit $\mathscr{M}$ et $\mathscr{F}$ deux classes distinctes.
On considère alors qu'une classe $\mathscr{F}$ dite « classe fille » peut hériter des attributs et des méthodes d'une autre classe $\mathscr{M}$ dite « classe mère ». En Java, pour spécifier qu'une classe hérite d'une autre on utilisera le mot clé `extends`.

Exemple concret
Prenons un exemple assez ludique, sur une table nous disposons de deux fruits, une **pomme** et une **orange**. Nous sommes tous d'accord pour dire que ces derniers n'ont pas la même couleur, le même goût donc qu'ils sont différents. Pourtant ce sont tous les deux des fruits malgré leurs caractéristiques différentes.

```
1  public classe Fruit {
2      private String nom;
3      private String couleur;
4      private double poids;
5
6      public Fruit(String nom, String couleur, double poids) {
7          this.nom = nom;
8          this.couleur = couleur;
9          this.poids = poids;
10     }
11
12     public String getNom() {
13         return nom;
14     }
15
16     public String getCouleur() {
17         return couleur;
18     }
19
```

```
20      public double getPoids() {
21          return poids;
22      }
23
24      public void afficherInfo() {
25          System.out.println("Nom: " + nom);
26          System.out.println("Couleur: " + couleur);
27          System.out.println("Poids: " + poids + " grammes");
28      }
29
30      public double calculerPrix() {
31          return 0.0;
32      }
33  }
```

Nous avons donc définit au dessus une classe `Fruit`, elle possède les attributs `Nom`, `Poids`, `couleur` et `forme` ainsi que deux méthodes principales permettant d'afficher les informations du fruits et de calculer son prix.

```
1   public classe Pomme extends Fruit {
2       private String variete;
3
4       public Pomme(String nom, String couleur, double poids, String variete) {
5           super(nom, couleur, poids);
6           this.variete = variete;
7       }
8
9       public String getVariete() {
10          return variete;
11      }
12
13      public void setVariete(String variete) {
14          this.variete = variete;
15      }
16
17      @Override
18      public void afficherInfo() {
19          super.afficherInfo();
20          System.out.println("Variété: " + variete);
21      }
22
23      @Override
24      public double calculerPrix() {
25          /* Prix fictif : 2 euros par kilo */
26          return getPoids() * 0.002;
27      }
28  }
```

Le mot clé **extends** lors de la définition de la classe `Pomme` signifie que celle-ci hérite de la classe `Fruit`, ce qui implique que :
Dans la suite de l'exemple, les classes `Fruit` *et* `Pomme` *seront représentées par les lettres* F *et* P *respectivement*.

- En héritant de F la classe P hérite de tous les attributs et les méthodes de F. Ceci évite la répétition de code, entre autre, la classe P n'a pas besoin de redéfinit les attributs `nom`, `couleur` et `poids` car ils sont présents dans la classe F.
- La classe P peut donc utiliser les méthodes et les attributs de la classe F, si ces derniers possèdent une visibilité adéquat (`public`, ou `protected`).
- La classe P peut ajouter ces propres méthodes et attributs en plus de ceux hérités de la classe F, ici ajout de l'attribut `variete` et redéfinition des méthodes de `Fruit`.

- Le mot clé `super` permet d'appeler une méthode de la classe F.

Voici comment utiliser ces classes.

```java
public class Main {
    public static void main(String[] args) {
        Pomme golden = new Pomme("Golden", "Jaune", 150, "Golden");
        golden.afficherInfo();
        System.out.println("Prix: " + golden.calculerPrix() + " euros");
    }
}
```

```
Nom: Golden
Couleur: Jaune
Poids: 150.0 grammes
Variété: Golden
Prix: 0.3 euros
```

Remarque

Utilisation de `@override`
Le mot clé `@override` utilisé avant deux méthodes de la classe P permet de dire au compilateur que l'on **redéfinit des méthodes** de la classe parent (ici c'est la classe F).
Par exemple, en prenant la mathode `afficherInfo( )` de la classe `Pomme`, on redéfinit la méthode de la classe mère, c'est à dire que lorsque l'on appellera cette méthode avec un objet de type P, on exécutera la méthode qui se trouve dans cette classe fille.

2.2) Les interfaces en Java

DÉFINITION *(Interfaces)*

Les **interfaces** en Java peuvent être présentées comme étant un contrat qui définit un ensemble de méthode abstraite que qu'une classe doit implémenter.
En gros toute classe qui implémente une interface doit contenir le code de toutes les fonctions de cette dernière.
Une classe peut implémenter plusieurs interfaces.

Rappel de cours 7

On appelle **méthode abstraite** une méthode de classe définie sans corps.
De manière générale :

```
visibilite typeRetour nomMethode (parametre1, ..., parametreN) ;
```

Cette définition est aussi appelée « signature » de la fonction.

Exemple, les outils sur Minecraft

- **Un peu de contexte pour comprendre le problème**
  Minecraft est un jeu bac à sable où les outils sont des éléments plus qu'important puisqu'ils permettent au joueur de miner, couper du bois, ou encore creuser.
  Chaque outil a une utilité et un fonctionnement bien spécifique, mais chacun d'entre eu doit pouvoir intéragir avec le monde.

Rappels :

- ❏ Une **Hache** (`Axe`) permet de couper du bois.
- ❏ Une **Pioche** (`Pickaxe`) permet de miner de la pierre, des minerais.
- ❏ Une **Pelle** (`shovel`) permet quant à elle de creuser et récupérer du sable, de la terre, du gravier

- **Implémentation sans interfaces**

```java
class Pickaxe {
    public void useTool() {
        System.out.println("Miner un bloc avec une pioche...");
    }
}

class Axe {
    public void useTool() {
        System.out.println("Couper du bois avec une hache...");
    }
}

public class MinecraftWithoutInterface {
    public static void main(String[] args) {
        Pickaxe pickaxe = new Pickaxe();
        Axe axe = new Axe();

        useSpecificTool(pickaxe);
        useSpecificTool(axe);
    }

    public static void useSpecificTool(Object tool) {
        if (tool instanceof Pickaxe) {
            Pickaxe pickTool = (Pickaxe) tool;
            pickTool.useTool();
        } else if (tool instanceof Axe) {
            Axe axeTool = (Axe) tool;
            axeTool.useTool();
        } else {
            System.out.println("Outil inconnu.");
        }
    }
}
```

Maintenant j'aimerais rajouter une classe `Shovel` qui contiendra les spécificités de la Pelle. Mais on remarque déjà qu'il y a quelques problèmes.

- ⇨ Dans la classe principale `MinecraftWithoutInterface`, la méthode `useSpecificTool` qui permet d'afficher quelle outil on veut utiliser est obligée de faire des `cast` afin de savoir à quel classe appartient l'objet passé en argument.
  En gros là on a deux classes donc deux cast, mais plus on rajoutera de class (`Shovel`, `Sword`, ...) plus il y aura de cast pour savoir quel objet on manipule.
- ⇨ Si vous êtes à plusieurs développeurs à travailler sur l'implémentation des différents outils, chacun d'entre vous allez devoir toucher à la méthode principale, alors cela augmente considérablement les risques de bugs.
- ⇨ **Manque de polymorphisme**
  Tous les outils doivent être traités différemment car pas le même type, pas toujours le même comportement.

- **Implémentation avec interface**

```java
interface Tool {
    void useTool(); /* Méthode commune à tous les outils */
}

class Pickaxe implements Tool {
    @Override
    public void useTool() {
        System.out.println("Miner un bloc avec une pioche...");
    }
}

class Axe implements Tool {
    @Override
    public void useTool() {
        System.out.println("Couper du bois avec une hache...");
    }
}

public class MinecraftWithInterface {
    public static void main(String[] args) {
        Tool pickaxe = new Pickaxe();
        Tool axe = new Axe();

        useTool(pickaxe);
        useTool(axe);
    }

    public static void useTool(Tool tool) {
        tool.useTool();
    }
}
```

Explications

- ⇨ Création de l'interface `Tool` qui définit le signature de la méthode `useTool( )`.
  Ainsi toutes les classes qui implémenteront cette interface devront définir le corps de la méthode `useTool( )`.
- ⇨ Ainsi, les objets associés à l'interface `Tool` peuvent être manipulés de la même manière avec la méthode `useTool( )` **sans vérifier le type**.
- ⇨ **Extensibilité du code**
  On rappelle que je voulais ajouter la classe `Shovel` pour définir le comportement d'une Pelle.
  Ainsi on ajoutera :

```java
class Shovel implements Tool {
    @Override
    public void useTool() {
        System.out.println("Digging with a shovel...");
    }
}
```

- ⇨ Ainsi, même si les développeurs doivent ajouter d'autres outils, puisque chacun est indépendant mais qu'ils implémentent l'interface `Tool`, il n'y aura pas besoin de changer la méthode principale `useTool(Tool tool)`.
- ⇨ Le code devient alors bien plus clair.

Exercice

Dans Minecraft, le joueur peut miner différents types de minerais, comme le fer, l'or, ou le diamant. Chaque minerai a des caractéristiques spécifiques, mais ils partagent aussi un comportement commun : être minés et donner un matériau.

1. Créer l'interface `Ore` qui définit deux méthodes :
   - **A.** `mine( )` Affichera quel minerai le joueur mine
   - **B.** `getMaterial( )` Affiche quel matériau on obtient après minage (ex. 'diamond', 'Iron Ingot').
2. Implémenter trois classes `IronOre`, `GoldOre`, `DiamondOre`
3. Créer une classe principale qui utilise ce que vous venez de coder.

2.3) Généricité en Java

2.3.1) Rappels de cours sur la généricité

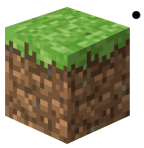
DÉFINITION *(généricité)*

La **généricité** permet de créer des classes, des interfaces et des méthodes capables de fonctionner avec différents types de données tout en garantissant la sécurité des types lors de la compilation.
Cela évite des cast, donc des conversions et des erreurs de types.

Avantages à utiliser la généricité :

- ⇨ Au lieu de créer des classes et des méthodes distinctes pour différents types, on utilise une unique version générique unique.
- ⇨ Le erreurs de types sont détectées lors de la compilation.
- ⇨ Le code devient plus clair et lisible.

Exemple, (suite) les outils sur Minecraft



- **Un peu de contexte pour comprendre le problème**
  On a vu précédemment que l'on pouvais définir une interface `Tool` et l'implémenter pour différentes classes d'outils, comme `Pickaxe`, `Axe`, `Shovel`. Cependant, si on veut gérer une collection (liste) d'outils, la généricité peut devenir bien utile.

- **Exemple sans généricité**
  On reprend l'interface `Tool` et les différentes classes outils créées précédemment.
  Création d'une classe `ToolBox` qui possède un attribut `tools` de type `List`, un constructeur, une méthode pour ajouter un outil à la liste et une autre méthode permettant de parcourir toute la liste et d'utiliser tous les outils.

```
1  import java.util.ArrayList;
2  import java.util.List;
3
4  interface Tool {
5      void useTool();
6  }
7
8  class Pickaxe implements Tool {
9      @Override
```

```
10      public void useTool() {
11          System.out.println("Miner avec une pioche...");
12      }
13  }
14
15  class Axe implements Tool {
16      @Override
17      public void useTool() {
18          System.out.println("Couper du bois avec une hache...");
19      }
20  }
21
22  class Shovel implements Tool {
23      @Override
24      public void useTool() {
25          System.out.println("Creuser avec une pelle...");
26      }
27  }
28
29  class ToolBox {
30      private List tools;
31
32      public ToolBox() {
33          this.tools = new ArrayList();
34      }
35
36      public void addTool(Object tool) {
37          tools.add(tool);
38      }
39
40      public void useAllTools() {
41          for (Object tool : tools) {
42              if (tool instanceof Tool) {
43                  ((Tool) tool).useTool();
44              }
45          }
46      }
47  }
48
49  public class MinecraftToolBox {
50      public static void main(String[] args) {
51          ToolBox toolBox = new ToolBox();
52          toolBox.addTool(new Pickaxe());
53          toolBox.addTool(new Axe());
54          toolBox.addTool(new Shovel());
55
56          toolBox.useAllTools(); /* Utilisation des outils */
57      }
58  }
```

Problèmes qui vont se poser

- ⇨ Lors de l'ajout d'un outil dans la liste, on ne vérifie pas si le type de l'objet que l'on ajoute est de type `Tool`, cela pourra provoquer des erreurs lors de l'exécution.
- ⇨ Dans la méthode `useAllTools()` il faut faire un cast sur chaque élément de la liste d'outils ce qui alourdit le code et peut provoquer des erreurs si le type d'un des objets n'est pas correct.

- **Exemple avec généricité**

```java
import java.util.ArrayList;
import java.util.List;

interface Tool {
    void useTool();
}

class Pickaxe implements Tool {
    @Override
    public void useTool() {
        System.out.println("Miner avec une pioche...");
    }
}

class Axe implements Tool {
    @Override
    public void useTool() {
        System.out.println("Couper du bois avec une hache...");
    }
}

class Shovel implements Tool {
    @Override
    public void useTool() {
        System.out.println("Creuser avec une pelle...");
    }
}

class ToolBox<T extends Tool> {
    private List<T> tools;

    public ToolBox() {
        this.tools = new ArrayList<>();
    }

    public void addTool(T tool) {
        tools.add(tool);
    }

    public void useAllTools() {
        for (T tool : tools) {
            tool.useTool();
        }
    }
}

public class MinecraftToolBox {
    public static void main(String[] args) {
        ToolBox<Tool> toolBox = new ToolBox<>();

        toolBox.addTool(new Pickaxe());
        toolBox.addTool(new Axe());
        toolBox.addTool(new Shovel());

        toolBox.useAllTools();
    }
}
```

Après exécution...

```
Miner avec une pioche...
Couper du bois avec une hache...
Creuser avec une pelle...
```

Avantages à la généricité

- ➪ La classe `ToolBox` impose qu'elle n'accepte qye des objets `T` de type Tool, ainsi tout objet ne faisant pas partie de la classe `Tool` provoquera une erreur lors de la compilation.
- ➪ Puisque `ToolBox` garantit que les objets `T` soient de type `Tool` alors la liste `Tools` contiendra obligatoirement des objets de type `Tool`, donc plus besoin de cast.
- ➪ Si plus tard on souhaite implémenter d'autres outils (comme `Mass`, `Hoe`), la classe `ToolBox` pourra les accepter tant qu'ils implémentent la classe `Tool`, sans pour autant devoir modifier la classe `ToolBox`.
  $\Longrightarrow$ **code plus flexible**.

Exercice

Créer une nouvelle boîte d'outils `PickaceBox` qui devra contenir uniquement des objets qui implémentent `Pickaxe`.

2.3.2) Les *ArrayList*, classe générique

DÉFINITION *(ArrayList)*

ArrayList est une classe de la bibliothèque standard qui permet en fait de stocker des éléments dans un tableau dynamique.
Du coup, une ArrayList peut s'adapter dynamiquement à la taille des éléments qu'elle contient, en ajouter, en supprimer sans avoir besoin de redimentionner manuellement le tableau (comme en C...).

Application : Utiliser les `ArrayList` avec des types génériques pour créer des collections de types spécifique en préservant la sécurité de type à la compilation.

2.3.2.1) *Exemple simple aux ArrayList*

```java
import java.util.ArrayList;

public class ArrayListExample {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();

        list.add("Apple");
        list.add("Banana");
        list.add("Cherry");

        for (String item : list) {
            System.out.println(item);
        }
    }
}
```

DÉFINITION *(classe générique)*

Les **classes génériques** permettent de définir des collection paramétrées par type.

Remarque | En gros, au lieu de définit un liste d'`Object` qui peut contenir n'importe quel type, vous pouvez donner le type exact des objets que la liste contiendra, comme ça on évite les cast inutiles et les erreurs de types.

2.3.2.2) *Méthodes associées aux* `ArrayList`

Évidemment, la classe `ArrayList` permet d'introduire un certain nombre de méthodes qui lui sont associés. *Ici, ce sont les méthodes principales*

Ajouter un élément `.add( )`

Soit `ArrayList<T> L = new ArrayList< >` une liste dynamique contenant des éléments de type `T`, deux façons d'ajouter un élément :

- ❏ Ajouter un élément en **fin de liste**
- ❏ Ajouter un élément **à l'indice** `i`

```
1 L.add(T element); /* Ajoute en fin de liste */
2 L.add(int i, T element); /* Ajoute à l'indice i */
```

Accéder aux éléments `.get( )`

Soit `ArrayList<T> L = new ArrayList< >` une liste dynamique contenant des éléments de type `T`, alors on peut accéder à l'élément d'indice `i` :

```
1 L.get(int i); /* J'accède à le i-ème élément */
```

Modifier les éléments `.set( )`

Soit `ArrayList<T> L = new ArrayList< >` une liste dynamique contenant des éléments de type `T`, alors on peut modifier l'élément d'indice `i` :

```
1 L.set(int i, T nouvElem); /* Je modifie le i-ème élément */
```

Supprimer des éléments `.remove( )`

Soit `ArrayList<T> L = new ArrayList< >` une liste dynamique contenant des éléments de type `T`, deux façons de supprimer un élément :

- ❏ Supprimer un objet `T truc` souhaité
- ❏ Supprimer un élément **à l'indice** `i`

```
1 L.remove(T truc); /* Supprime le premier élément correspondant */
2 L.remove(int i); /* Supprime le i-ième élément */
```

Liste vide ? `.isempty( ), .contains( ), .indexOf( )`

Soit `ArrayList<T> L = new ArrayList< >` une liste dynamique contenant des éléments de type `T`. :

- ❏ Vérifier si la liste est vide
- ❏ Vérifier si la liste contient `T truc`
- ❏ Retourner l'indice de l'élément `T truc` si trouvé

```
1 L.isEmpty( ); /* La liste est-elle vide ? */
2 L.contains(T truc); /* La liste contient elle truc ? */
3 L.indexOf(T truc); /* Renvoi l'indice de truc si il existe */
```

Remarque || Si `L.indexOf(...)` ne trouve pas l'objet dans la liste, alors il renverra `-1`.

Taille de la liste `.size( )`

Soit `ArrayList<T> L = new ArrayList< >` une liste dynamique contenant des éléments de type `T`, alors on peut obtenir la taille de `L` :

```
1 L.size( );
```

Itération et parcours `.iterator( ), .forEach( )`

Soit `ArrayList<T> L = new ArrayList< >` une liste dynamique contenant des éléments de type `T`, alors on peut parcourir un tableau :

```
1 L.iterator( ); /* Retourne un itérateur pour parcourir les éléments */
2 L.forEach(Consumer<? super T> action);
3 /* Applique une action à chaque élément */
```

Nettoyer une liste `.clear( )`

Soit `ArrayList<T> L = new ArrayList< >` une liste dynamique contenant des éléments de type `T`, alors on peut nettoyer (= vider) la liste :

```
1 L.clear( );
```

2.3.3) *L'interface générique* `Collection<E>`

DÉFINITION *(L'interface `Collection`)*

`Collection<E>` esy une **interface générique**, elle représente un ensemble d'objet (appelés éléments).

Remarque | Puisque la classe `ArrayList<E>` implémente justement `Collection<E>` alors elle possède toute les méthodes de cette dernières.
La signature de certaines des méthodes décrites au dessus appartiennent à `Colection<E>` comme `add(E elem)` par exemple.

Remarque | Un type simple ne peut pas être utilisé pour un type générique, on utilise alors les **types *Wrapper*** à la place (`Integer` au lieu de `Int` par exemple).

La classe générique `ArrayList<E>` implémente l'interface générique `List<E>`.

2.3.4) Les *HashSet*, une autre collection générique

DÉFINITION *(HashSet)*

`HashSet` est une implémentation de l'interface `Set<E>`. Ce dernier est utilisé pour stocker des éléments unique.
En gros, il ne **permet pas de stocker des doublons**.
Pour gérer ses données, `HashSet` utilise une table de hachage en interne (HashMap).

Caractéristiques principales

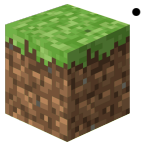
- Les éléments d'un `HashSet` sont uniques.
  Si vous souhaitez ajouter un élément déjà présent, ce dernier ne sera pas ajouté
- Les données ne sont pas ordonnées.
- Si on ajoute/supprime un élément, l'ordre peut changer.

Rappel de cours 8

Puisque `HashSet` implémente `Set<E>` et hérite de `Collection<E>` ainsi, elle propose les méthodes suivantes :

- `add(E elem)`, `remove(Object o)`, `clear( )`
- `isEmpty( )`, `contains(Object o)`
- *et d'autres...*

Exemple, Partie spéciale sur Minecraft



- Dans Minecraft, un joueur a un inventaire, mais on veut s'assurer que certains types d'objets ne peuvent pas être dupliqués. Par exemple, dans une partie spéciale, un joueur ne peut avoir qu'un seul exemplaire de chaque outil rare (comme une « épée de diamant » , une « pioche enchantée », etc.).

```
1  import java.util.HashSet;
2
3  public class MinecraftInventory {
4      public static void main(String[] args) {
5          HashSet<String> inventory = new HashSet<>();
6
7          inventory.add("Épée de diamant");
8          inventory.add("Pioche enchantée");
9          inventory.add("Arc puissant");
10
11         System.out.println("Inventaire : " + inventory);
12
13         boolean ajoutEpee = inventory.add("Épée de diamant");
14         if (!ajoutEpee) {
15             System.out.println("Tu as déjà une Épée de diamant
16             dans l'inventaire !");
17         }
18
19         if (inventory.contains("Pioche enchantée")) {
20             System.out.println("Tu as une Pioche enchantée !");
21         } else {
```

```
22                System.out.println("Tu n'as pas encore de Pioche enchantée.");
23            }
24
25            inventory.remove("Arc puissant");
26            System.out.println("Après suppression, inventaire : " + inventory);
27
28            System.out.println("Nombre d'outils dans l'inventaire : " +
29            inventory.size());
30        }
31    }
```

Remarque | On tente d'ajouter (ligne 13) l'item « Épée de diamant » dans `inventory`, mais cette dernière est déjà dans l'inventaire alors, la valeur de `ajoutEpee` sera fausse.

```
Inventaire : ["Épée de diamant", "Arc puissant", "Pioche enchantée"]
Tu as déjà une Épée de diamant dans l'inventaire !
Tu as une Pioche enchantée !
Après suppression, inventaire : ["Épée de diamant", "Pioche enchantée"]
Nombre d'outils dans l'inventaire : 2
```

2.3.5) Zoom sur la boucle « for each »

En Java, la boucle `for-each` est une boucle améliorée qui est utilisée pour parcourir les éléments d'une collection ou d'un tableau de manière simple et lisible.
Cette boucle fonctionne avec toute classe qui implémente l'interface `Iterable<E>` comme `ArrayList<E>` et `HashSet<E>` vus précédemment.

Exemple, liste d'outils

```
1  import java.util.ArrayList;
2
3  public class ForEachExample {
4      public static void main(String[] args) {
5          /* Création d'une collection */
6          ArrayList<String> tools = new ArrayList<>();
7          tools.add("Épée");
8          tools.add("Pioche");
9          tools.add("Arc");
10
11         /* Parcourir les éléments avec for-each */
12         for (String tool : tools) {
13             System.out.println(tool);
14         }
15     }
16 }
```

```
Épée
Pioche
Arc
```

Structure générale de la boucle for-each

```
for (Type truc : collection) { ... }
```

Remarque | Une interface peut contenir le corps des méthodes si ces dernières sont déclarées avec les mots clé `static` ou `final`, mais seront étudiées plus tard.

2.4) Les lambda-expressions

DÉFINITION *(lamba-expression)*

Les **expressions lambda** ont été introduites en Java 8, elles permettent de simplifier la création d'objets pour les interfaces fonctionnelles (interfaces avec une seule méthode abstraite).
Rendant le code plus concis.
Autrement dit, on peut voir une lambda expression comme étant une méthode anonyme utilisée pour définir une fonction en ligne.

Syntaxe générale des lambdas expressions

```
(paramètres) -> { corps de la méthode }
```

Exemple, tri d'une liste
On a une liste de chaine de caractères (`String`) et on souhaite la trier par ordre alphabétique.

- **Sans utiliser d'expression lambda**

```java
import java.util.*;

public class LambdaExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Steve", "Alex", "Herobrine");

        Collections.sort(names, new Comparator<String>() {
            @Override
            public int compare(String a, String b) {
                return a.compareTo(b);
            }
        });
        System.out.println(names);
    }
}
```

- **En utilisant les expressions lambda**

```java
import java.util.*;

public class LambdaExample {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Steve", "Alex", "Herobrine");

        Collections.sort(names, (a, b) → a.compareTo(b));
        System.out.println(names);
    }
}
```

```
[Alex, Herobrine, Steve]
```

`Collections.sort(names, (a, b) -> a.compareTo(b));`
On utilise uhne méthode statique de la classe `Collections` qui trie les éléments d'une liste prenant en paramètre, la liste à trier, ainsi que la règle de tri.

Rappel de cours 9

La méthode `.compareTo( )`
Elle fait partie de `String` et compare de chaine lexicographiquement :

- Si $a < b$ renvoi un nombre négatif.
- Si $b < a$ renvoi un nombre positif.
- Si $a = b$ renvoi 0.

Remarque

Dans Haskell, le système de type est beaucoup plus flexible et intelligent en ce qui concerne l'inférence des types. Lorsque vous écrivez une lambda (ou une fonction anonyme), le compilateur peut déduire automatiquement le type des paramètres à partir du contexte, sans que vous ayez besoin de spécifier explicitement les types. Cela rend les lambdas en Haskell très concises et le langage généralement plus flexible.

2.5) 4 interfaces fonctionnelles de base

2.5.1) L'interface *Function<T, R>*

DÉFINITION *(Function<T, R>)*

L'interface `Function<T, R>` fait partie de l'**API des fonctions lambda** introduites en Java 8. Elle est utilisée pour représenter une fonction qui prend un argument de type `T` et renvoie un résultat de type `R`.

Signature de l'interface Function

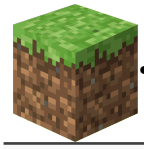
```java
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
}
```

Explications :

❑ **Type générique** `T` **et** `R`
- ➪ Le type `T` représente le type de l'argument que reçoit la fonction.
  On rappel que `T` doit faire partie des types Wrapper.
- ➪ Le type `R` est le type du résultat retourné par la fonction.

❑ **Méthode** `apply(T t)`
Méthode principale de l'interface, qui définit le comportement de la fonction.
En gros, quand vous créez une instance de l'interface `Function` vous devez aussi définir la méthode `apply` pour spécifier ce que la fonction doit faire avec l'argument `T`.

Exemple, Création d'un plugin sur Minecraft



- Vous souhaitez créer un plugin Minecraft où on veut vérifier si un joueur a atteint son objectif.

```java
import java.util.function.Function;

public class MinecraftFunctionExample {
    public static void main(String[] args) {
```

```
5           Function<Player, String> checkGoal = (Player player) → {
6               if (player.getPoints() >= 100) {
7                   return player.getName() + " a atteint l'objectif !";
8               } else {
9                   return player.getName() + " doit encore travailler
10                  pour atteindre l'objectif.";
11              }
12          };
13          Player player1 = new Player("Alex", 120);
14          Player player2 = new Player("Steve", 80);
15
16          System.out.println(checkGoal.apply(player1));
17          System.out.println(checkGoal.apply(player2));
18      }
19  }
20
21  class Player {
22      private String name;
23      private int points;
24
25      public Player(String name, int points) {
26          this.name = name;
27          this.points = points;
28      }
29
30      public String getName() {
31          return name;
32      }
33
34      public int getPoints() {
35          return points;
36      }
37  }
```

Explications :

- La fonction `main`
  - ⇨ Création de `checkGoal`
    `checkGoal` est une variable qui permet de vérifier si le joueur a atteint son objectif ou non. `checkGoal` est une variable de type `Function<Player, String>`, c'est à dire que la fonction prend en argument un objet de type `Player` et renvoi un argument de type `String`.
  - ⇨ Création de deux objets issus de la classe `Player` et vérifions si ils ont atteint l'objectif.

```
Alex a atteint l'objectif !
Steve doit encore travailler pour atteindre l'objectif.
```

2.5.2) Linterface *Predicate<T>*

DÉFINITION *(Predicate)*

L'interfacte `Predicate<T>` a elle aussi été introduite depuis Java 8. C'est une fonction qui évalue une condition (= prédicat) sur un objet de type `T` et retourne un booléen.

Signature de l'interface `Predicate<T>`

```java
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}
```

La fonction `test` reçoit un argument de type `T` et renvoi un booléen.
Vous le savez retourne `true` si vrai, `false` sinon.

En plus de la méthode `test`, l'interface fournit d'autre méthodes pour composer plusieurs prédicats :

- ❏ `and(Predicate<? super T> other)`
  Combine deux prédicats avec la relation logique **ET**.
- ❏ `or(Predicate<? super T> other)`
  Combine deux prédicats avec la relation logique **OU** (inclusid).
- ❏ `negate()`
  Retourne un prédicat qui est en fait l'inverse logique du prédicat courant.
- ❏ ...

Exemple (1)

```java
import java.util.function.Predicate;

public class SimplePredicateExample {
    public static void main(String[] args) {
        Predicate<Integer> isEven = n → n % 2 == 0;

        System.out.println(isEven.test(4));
        System.out.println(isEven.test(7));
    }
}
```

```
true
false
```

Exemple (2), combinaison de prédicats

```java
import java.util.function.Predicate;

public class SimplePredicateComposition {
    public static void main(String[] args) {
        Predicate<Integer> isEven = n → n % 2 == 0;
        Predicate<Integer> greaterThanFive = n → n > 5;

        Predicate<Integer> evenAndGreaterFive = isEven.and(greaterThanFive);

        System.out.println(evenAndGreaterFive.test(8));
        System.out.println(evenAndGreaterFive.test(4));
        System.out.println(evenAndGreaterFive.test(7));
    }
}
```

Ici on a créé deux prédicats, un qui vérifie si le nombre est pair, un autre si il est plus grand que 5. Ensuite, on a créer un prédicat « fusion » qui utilise les deux créés précédemment.

```
true
false
false
```

2.5.3) L'interface `Consumer<T>`

DÉFINITION *(Consumer)*

Introduite avec Java 8,
L'interface `Consumer<T>` et elle est en fait représentée par une opération qui prend un argument de type `T` et ne retourne rien.

Remarque | `Consumer<T>` est idéale pour effectuer des opérations comme afficher des données, modifier un objet, effectuer une opération sans retourner de valeur.

Signature de l'interface `Consumer<T>`

```java
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}
```

La fonction `accept` reçoit un argument de type `T` et ne renvoi rien.

En plus de la méthode `accept(T t)`, `Consumer` fournit une méthode par défaut qui permet de combiner plusieurs `Consumer` :

`andThen(Consumer<? super T> after>`

Exemple (1)

```java
import java.util.Arrays;
import java.util.List;
import java.util.function.Consumer;

public class ConsumerExample {
    public static void main(String[] args) {
        Consumer<String> print = s → System.out.println(s);
        List<String> names = Arrays.asList("Steve", "Alex", "Notch");

        names.forEach(print);
    }
}
```

Dans le main,

- Création d'un `Consumer<String>` qui permet d'afficher une donnée.
- Création d'une `List` de trois éléments de type `String`
- Parcours de la liste en effectuant le `Consumer` sur chaque élément

```
Steve
Alex
Notch
```

Exemple (2), combinaison de deux `Consumer`

```java
import java.util.function.Consumer;

public class ConsumerChainingExample {
    public static void main(String[] args) {
        Consumer<String> print = s → System.out.println("Valeur : " + s);

        Consumer<String> printLength = s → System.out.println("Longueur : " +
                                                              s.length());

        Consumer<String> combined = print.andThen(printLength);

        combined.accept("Minecraft");
    }
}
```

Deux `Consumer` :

- Un pour afficher une valeur
- Un pour afficher la taille de la chaîne de caractère entrée en paramètre

Ainsi, on combine les deux `Consumer`, le troisième vas donc afficher la valeur de la chaine de caractère puis sa longueur.

```
Valeur : Minecraft
Longueur : 9
```

2.5.4) L'interface *Supplier<R>*

DÉFINITION *(Supplier)*

L'interface `Supplier<R>` a été introduite en Java 8, elle représente un « fournisseur de résultat », en gros elle ne prend aucun argument mais renvoi une valeur de type `R`.

Signature de l'interface `Supplier<R>`

```java
@FunctionalInterface
public interface Supplier<R> {
    R get();
}
```

La fonction ne prend aucun argument mais renvoi une valeur de type `R`.

Exemple (1)

```java
import java.util.function.Supplier;

public class SimpleSupplierExample {
    public static void main(String[] args) {
        Supplier<String> messageSupplier = () → "Bienvenue dans Minecraft!";

        System.out.println(messageSupplier.get());
    }
}
```

```
Bienvenue dans Minecraft !
```

Exemple (2), un peu plus compliqué

```java
import java.util.ArrayList;
import java.util.List;
import java.util.function.Supplier;

public class ListSupplierExample {
    public static void main(String[] args) {
        Supplier<List<String>> listSupplier = ArrayList::new;

        List<String> newList = listSupplier.get();
        newList.add("Minecraft");
        newList.add("Java");

        System.out.println("Liste : " + newList);
    }
}
```

```
Liste : [Minecraft, Java]
```

2.6) La classe `Optional`

Rappel de cours 10

> En Haskell, on pouvais représenter le fait qu'une fonction retourne un résultat ou pas avec le type `Maybe`, la clase `Optional` en java permet de faire la même chose.

DÉFINITION *(Optional)*

Introduite avec Java 8,
La classe `Optional` est utilisée pour **représenter un résultat absent ou présent**.
Permettant de mieux gérer les cas d'une valeur `null` et de manipulation de `null`.
Comme les `NullPointerException`.

Définition de la classe `Optional`

```java
public final class Optional<T> {
    private Optional() { /* private constructor */ }
}
```

`Optional` est une classe générique, ce qui signifie qu'elle peut contenir n'importe quel type d'objet. Elle contient une valeur de type `T`, ou peut être vide (sans valeur).

2.6.1) Construction d'un `Optional`

3 façons vues en cours

- ❏ Contruction d'un `Optional` représentant l'**absence de données** (`Nothing` en Haskell) : `Optional.empty( )`
- ❏ Construire un `Optional` qui **encapsule une donnée** : `Optional.of(donnee)`
- ❏ Contruire un `Optional` vide ou qui encapsule une donnée à partir d'une référence qui peut être `null` : `Optional.ofNullable(reference)`

2.6.2) Récupérer la valeur d'un *Optional*

Il est possible de récupérer une donnée encapsulée dans un `Optional` en utilisant **la méthode** `get( )`. Mais, si l'`Optional` est vide, `get` renverra une erreur.
Ainsi, on introduit la méthode `orElse(defaultValue)` qui s'applique à tout les `Optional` et renvoie la donnée encapsulée dans le `Optional` si il n'est pas vide. Sinon renverra `defaultValue`.

Exemple, récupérer la donnée d'un `Optional`

```java
import java.util.Optional;

public class SimpleOptionalExample {
    public static void main(String[] args) {
        Optional<String> playerName = Optional.of("Steve");
        String name = playerName.orElse("Joueur inconnu");
        System.out.println(name);
        Optional<String> emptyName = Optional.empty();
        String defaultName = emptyName.orElse("Joueur inconnu");
        System.out.println(defaultName);
    }
}
```

```
Steve
Joueur inconnu
```

2.6.3) Déterminer si un *Optional* est vide

Pour savoir si un `Optional` est vide ou non, deux méthodes possibles `isEmpty( )` et `isPresent( )`.

Exemple

```java
import java.util.Optional;

public class OptionalEmptyCheck {
    public static void main(String[] args) {
        Optional<String> playerName = Optional.of("Steve");
        if (playerName.isEmpty()) {
            System.out.println("L'Optional est vide.");
        } else {
            System.out.println("L'Optional contient une valeur : "
            + playerName.get());
        }
        Optional<String> emptyName = Optional.empty();
        if (!emptyName.isPresent()) {
            System.out.println("L'Optional est vide.");
        } else {
            System.out.println("L'Optional contient une valeur : "
            + emptyName.get());
        }
    }
}
```

```
L'Optional contient une valeur : Steve
L'Optional est vide.
```

2.6.4) Utilisation de *map* et *filter* sur un *Optinal*

2.6.4.1) *Optional* est un foncteur : Méthode *map*

Rappel de cours 11

En Haskell Maybe est un foncteur, on pouvait alors utiliser fmap.

```
Functor f => (a -> b) -> f a -> f b
```

En Java, Optional présente les mêmes propriétés avec la méthode map.

```
Optional<A> :
    Optional<B> map(Function<A, B>)
```

Rappel de cours 12

On se souvient que map permet de transformer la valeur dans un Optional si elle est présente.

Exemple

```
import java.util.Optional;

public class OptionalMapExample {
    public static void main(String[] args) {
        Optional<String> playerName = Optional.of("Steve");

        Optional<String> uppercaseName = playerName.map(String::toUpperCase);

        uppercaseName.ifPresent(System.out::println);

        Optional<String> emptyName = Optional.empty();

        Optional<String> result = emptyName.map(String::toUpperCase);
        result.ifPresentOrElse(
            System.out::println,
            () → System.out.println("Aucun nom disponible")
        );
    }
}
```

```
STEVE
Aucun nom disponible
```

2.6.4.2) *Filter* sur un *Optional*

La méthode filter permet d'obtenir un nouvel Optional à partir d'un Optional en suivant le fonctionnement suivant :

- ➪ Si Optional vide alors on obtient Optional vide
- ➪ Si la donnée de l'Optional ne respecte pas le prédicat passé en paramètre de filter, on obtient un Optional vide
- ➪ Si la donnée de l'Optional respecte le prédicat passé en paramètre de filter, on obtient le même Optional qu'au début

Exemple

```java
import java.util.Optional;

public class OptionalFilterExample {
    public static void main(String[] args) {
        Optional<String> playerName = Optional.of("Steve");

        Optional<String> longName = playerName.filter(name → name.length() > 4);

        longName.ifPresent(System.out::println);

        Optional<String> shortName = playerName.filter(name → name.length() > 5);

        shortName.ifPresentOrElse(
            System.out::println,
            () → System.out.println("Nom trop court")
        );
    }
}
```

```
Steve
Nom trop court
```

2.6.5) Utiliser un `Optional` pour éviter de multiples tests à `null`

Exemple

```java
public class Player {
    private String name;

    public Player(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
}

public class NullCheckExample {
    public static void main(String[] args) {
        Player player = null;

        if (player != null) {
            String name = player.getName();
            if (name != null) {
                System.out.println(name.toUpperCase());
            } else {
                System.out.println("Nom du joueur non disponible.");
            }
        } else {
            System.out.println("Aucun joueur.");
        }
    }
}
```

Ici, sans utiliser d'`Optional`, on est obliger de faire des tests à répétition afin de savoir si le joueur existe, si il a un nom, ...

Cela rend le code bien plus lourd et difficile à maintenir.
La solution est donc d'utiliser `Optional`.

```java
import java.util.Optional;

public class Player {
    private String name;

    public Player(String name) {
        this.name = name;
    }

    public Optional<String> getName() {
        return Optional.ofNullable(name); /* Retourne un Optional */
    }
}

public class OptionalExample {
    public static void main(String[] args) {
        Player player = new Player("Steve");

        String result = player.getName()
                .map(String::toUpperCase)
                .orElse("Nom du joueur non disponible");
        System.out.println(result);

        Player unnamedPlayer = new Player(null);

        String result2 = unnamedPlayer.getName()
                .map(String::toUpperCase)
                .orElse("Nom du joueur non disponible");
        System.out.println(result2);

        Optional<Player> noPlayer = Optional.empty();

        String result3 = noPlayer.flatMap(Player::getName)
                .map(String::toUpperCase)
                .orElse("Aucun joueur.");
        System.out.println(result3);
    }
}
```

Explications

On souhaite vérifier si un joueur existe, et si il possède un nom, alors on l'affichera en majuscule.

- **CAS 1**
  `player` est un joueur qui s'appelle `Steve`
  `player.getName( )` renverra alors `Optional("Steve")` ainsi lorsque l'on applique `map` dessus avec `toUpperCase` on vas alors obtenir `Optional("STEVE")`.
- **CAS 2**
  `unnamedPlayer` est un joueur sans nom
  Alors `unnamedPlayer.getName( )` renvoie `Optionnal(null)` le `map` ne peux donc s'effectuer, on renvoi un message d'erreur.
- **CAS 3**
  `noPlayer` est un `Optional` vide
  Ici `flatMap( )` ne s'exécute pas et renvoi dirrectement `Aucun joueur`

```
STEVE
Nom du joueur non disponible
Aucun joueur
```

Remarque | | On utilise `flatMap()` dans le cas n°3 car `player` est déjà un `Optional`.

2.6.6) Modifier une donnée encapsulée dans un *Optional*

Rappel de cours 13

Contrairement aux langages de programmation fonctionnels comme Haskell où les données sont des objets immuables. Il peut arriver que l'on souhaite modifier une donnée encapsulée dans un `Optional`.

Remarque | | Une donnée encapsulée dans un `Optional` n'est pas directement modifiable. Pour se faire, il faudra appliquer une méthode sur l'`Optional` visé et réencapsuler le résultat dans un nouvel `Optional`.

2.6.6.1) *Méthode 1 - Utiliser ifPresent*

Si l'`Optional` encapsule une donnée immuable, il est possible de modifier cet objet par un accès explicite via `isPresent( )`.

Exemple

```java
import java.util.Optional;

public class OptionalModifyObjectExample {
    public static void main(String[] args) {
        Optional<Player> optionalPlayer = Optional.of(new Player("Steve"));
        /* Optinal A */

        optionalPlayer.ifPresent(player → player.setName("Alex"));
        /* Optional B */

        optionalPlayer.ifPresent(player → System.out.println(player.getName()));
        /* Optional C */
    }
}

class Player {
    private String name;
    public Player(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Explications
Dans le `main`, on encapsule un objet `Player`. On souhaite modifier "Steve" en "Alex" alors on utilise la méthode `isPresent( )`.
En gros, si il y a une donnée (ici un joueur) dans l'`Optional` alors je change son nom en "Alex", enfin si le joueur est toujours présent j'affiche son nom. (vérification)

```
Alex
```

2.6.6.2) *Méthode 2 - Utiliser `map( )` pas vue en cours*

Pour modifier une donnée encapsulée dans un `Optional` on peut aussi utiliser `map`. Ainsi en utilisant `map` on vas alors créer un nouvel `Optional` qui contiendra la valeur modifiée de l'ancien si elle existe. Dans le cas contraire, on retourne alors un `Optional` vide.

Exemple

```java
import java.util.Optional;

public class OptionalModifyExample {
    public static void main(String[] args) {
        Optional<String> playerName = Optional.of("Steve");
        Optional<String> modifiedName = playerName.map(name → name + " le grand");
        System.out.println(modifiedName.orElse("Aucun joueur modifier."));

        Optional<String> emptyName = Optional.empty();
        Optional<String> result = emptyName.map(name → name + " le grand");
        System.out.println(result.orElse("Aucun joueur à modifier."));
    }
}
```

```
Steve le grand
Aucun joueur à modifier.
```

2.6.7) *`Optional` est une monade*

Rappel de cours 14

Signature de type de l'opérateur >>=

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

Avec `map` on aurait eu

```
maybe a -> (a -> maybe b) -> maybe (maybe b)
```

Pour `Optional` c'est `flatMap` qui joue le rôle de >>=.
Typage simplifié

```
Optional<T> :
    Optional<U> flatMap(Function<T, Optional<U>> mapper)
```

Typage réel

```
Optional<T> :
    Optional<U> flatmap(Function< ? super T, ? extends Optional< ? extends U>>)
```

2.7) Gestion des Flux

2.7.1) Généralités sur les flux

Rappel de cours 15

En Haskell on a vu que l'on pouvait traiter des masses de données d'un type précis grâce aux listes. En Java c'est aussi possible grâce aux flux `Stream`.

DÉFINITION *(flux)*

Les **flux** (`Stream`) permettent de traiter des données de manières séquentielle ou parallèle. En général on les utilise pour lire ou écrire des données, faire des transformations, ou appliquer des calculs sur des collections.

Remarque Tout comme le type `list` est paramétré par un type, le type `Stream` est en fait un type paramétré : `Stream<T>`.

Pourquoi `Stream<T>` et pas `List<E>` ?

- ❑ Assurer la rétro-compatibilté

  DÉFINITION *(rétro-compatibilité)*

  La **rétro-compatibilité** (ou compatibilité ascendante) désigne la capacité d'un système, d'un logiciel ou d'un matériel plus récent à fonctionner avec des versions antérieures.

- ❑ Le type `List` n'est pas conçu pour l'évaluation paresseuse

  DÉFINITION *(évaluation paresseuse)*

  L'**évaluation paresseuse** en Java consiste à ne pas évaluer une expression ou exécuter un calcul tant que son résultat n'est pas réellement nécessaire. Cela permet d'optimiser les performances en évitant des calculs inutiles.

  Exemple
  L'utilisation des opérateurs logiques && et ||, où Java évalue uniquement ce qui est nécessaire :
  - ⇨ Avec && : Si la première condition est false, la seconde n'est pas évaluée, car le résultat sera forcément false.
  - ⇨ Avec || : Si la première condition est true, la seconde n'est pas évaluée, car le résultat sera forcément true.

2.7.2) Création de flux

Puisque `Stream<T>` est une interface et pas une classe, alorx il n'y a donc pas de constructeur.
Voici quelques méthodes pour créer un flux :

- ❑ La méthode `stream( )` de l'interface `Collection`.
- ❑ La méthode `stream(T[] tableau)` de la classe `Arrays` pour les tableaux.

- La méthode d'instance `of( )` de l'inetrface `Stream` pour un tableau ou un ensemble fini connu d'éléments.
- La méthode `empty( )` de `Stream` qui créer un flux vide.

Exemple du cours

```java
import java.util.Arrays;
import java.util.List;
import java.util.stream.Stream;

public class Main1 {
  public static void main(String[] args) {
    List<String> mots = List.of("Bonjour", "tout", "le", "monde");

    Stream<String> fluxMots = mots.stream();

    Stream<String> fluxMots2 = Stream.of("Bonjour", "tout", "le", "monde");

    String[] tableau = {"Bonjour", "tout", "le", "monde"};
    Stream<String> fluxMots3 = Arrays.stream(tableau);

    Stream<String> fluxVide = Stream.empty();
  }
}
```

Remarque

Il existe d'autres nombreuses classes qui permettent de créer des flux.

- `stream( )` de la classe `Optional` (flux vide, ou à 1 élément suivant le contenu de l'`Optional`)
- `lines(Path p)` de la classe `java.nio.file.Files` (flux des lignes d'un fichier)
- `list(Path dir)` de la classe `java.nio.file.Files` (flux des fichier d'un dossier)

Un flux est destiné à être utilisé pour produire une autre donnée, ou un effet de bord, grâce à une méthode dite « terminale ».
Ainsi, pour le contenu d'une flux, on peut utiliser la méthode suivante :

```java
void forEach(Consumer< ? Super T> traitement)
```

Exemple, afficher les éléments contenus dans un flux

```java
import java.util.List;
import java.util.stream.Stream;

public class Main2 {
  public static void main(String[] args) {
    List<String> mots = List.of("Bonjour", "tout", "le", "monde");
    Stream<String> fluxMots = mots.stream();
    fluxMots.forEach(mot → System.out.println(mot));
  }
}
```

```
Bonjour
tout
le
monde
```

2.7.3) Un flux est un foncteur

Remarque | Comme les listes en Haskell, les flux Java sont des foncteurs, ainsi on y retrouve la méthode `map`.

```
Stream<T>
    <R> Stream<R> map(Function<? super T, extends R> mapper)
```

ou dans un premier temps...

```
Stream<T>
    <R> Stream<R> map(Function<T,R> mapper)
```

Exemple

```
import java.util.List;
import java.util.stream.Stream;

public class Main1 {
  public static void main(String[] args) {
    List<String> mots = List.of("Bonjour", "tout", "le", "monde");
    Stream<String> fluxMots = mots.stream();
    mots.stream()
        .map(mot → mot.length())
        .forEach(lg → System.out.println(lg));
    System.out.println("-------------");
    mots.stream()
        .map(mot → mot.length())
        .map(nb → 2 * nb)
        .forEach(lg → System.out.println(lg));
    System.out.println("-------------");
    mots.stream()
        .map(mot → mot.length())
        .filter(lg → lg >= 5)
        .map(nb → 2 * nb)
        .forEach(lg → System.out.println(lg));
  }
}
```

Explications

- On affiche la longueur de chaque mot
- On récupère la longueur de chaque mot et on la multiplie par 2 puis on l'affiche
- On récupère la longueur de chaque mot, on regarde si elle est supérieure ou égale à 5 si c'est le cas on la garde, on la multiplie par 2 puis on l'affiche

```
7
4
2
5
-------
14
8
4
10
-------
14
10
```

Remarque

Dans l'exemple ci-dessus, on a aligné les points, ce n'est pas obligatoire mais c'est une convention qui **est à respecter**.
Vous avez remarqué que l'on a utilisé `filter` sur les flux

```
Stream<T>
    Stream<T> filter(Predicate<? super T> predicate)
```

Exemple

```
import java.util.Arrays;
import java.util.stream.IntStream;
import java.util.stream.Stream;

public class Main1 {
  public static void main(String[] args) {
    int[] tab = {2, 4, 1, 3, 9, 10, 6,7};

    Arrays.stream(tab) // génère un IntStream
          .filter(n → n % 2 == 0)
            .filter(x → x+1)
          .forEach(nb → System.out.println(nb));
  }
}
```

```
3
5
11
7
```

Remarque

On remarque ainsi que `map` et `filter`renvoient à chaque fois un nouveau flux.
Il est donc possible ,de chaîner les traitement sans écrire de boucles, ce qui rend l'écriture du code bien plus lisible.
De plus imaginons que dans notre exemple du dessus si la liste ne contenait que des nombres impair alors, l'évaluation paresseuse évite d'exécuter le second filter sur un flux vide.

2.7.4) Propriétés générales sur les flux en Java

- ❏ Un **flux à usage unique**
  En gros après avoir utilisé un flux pour un traitement, si on veut on faire un autre, il faut donc recréer un flux
- ❏ Un **flux est évalué paresseusement** (notion expliquée au dessus)
- ❏ Il existe d'autre classes de flux plus efficaces pour certains types simples `DoubleStream`, `IntStream` et `LongStream`.

Remarque

- La méthode `stream` de `Arrays` crée un flux de types simples si le tableau passé en paramètre contient des valeurs de type simple.
- **Attention :** sur ces classes de flux spécifiques, le map renvoie aussi un flux du même type simple

2.7.5) Analogie avec les listes de Haskell

- La méthode `Optional<T> findFirst( )` permet de **récupérer le premier élément d'un flux** (si il n'est pas vide).
  Méthode `head` en Haskell
- La méthode `Stream<T> limit(long n)` permer de **garder les n-premiers éléments d'un flux**
  Méthode `take` en Haskell
- La méthode `Stream<T> skip(long n)` permet de **sauter les n premier élément d'un flux**
  Méthode `drop` en Haskell
- La méthode `Stream<T> takeWhile(Predicate<? super T> predicate)` permet de **garder les premiers éléments du flux tant qu'ils vérifient le prédicat**.
  `takeWhile` en Haskell
- La méthode `Stream<T> dropWhile(Predicate<? super T> predicate)` permet de **jeter les premiers éléments du flux tant qu'ils vérifient le prédicat**.
  `dropWhile` en Haskell

Remarque | Toutes ces méthodes **sauf** `findFirst( )` renvoient des flux, on peut alors les chaîner pour faire différents traitements.

Exemple du cours

```java
import java.util.List;
import java.util.Optional;

public class Main4 {
  public static void main(String[] args) {
    List<String> mots = List.of("Bonjour", "tout", "le", "monde");
    System.out.println("-------------");
    long nbElements = mots.stream()
                      .filter(mot → mot.contains("o"))
                      .count();
    System.out.println(nbElements);

    System.out.println("-------------");
    boolean tous0 = mots.stream()
                .allMatch(mot →mot.contains("o"));
    System.out.println("contiennent tous un o : " + tous0);

    System.out.println("-------------");
    boolean un0 = mots.stream()
          .anyMatch(mot →mot.contains("o"));
    System.out.println("au moins 1 mot contient un o : " + un0);

    System.out.println("-------------");
    boolean zero0 = mots.stream()
                .noneMatch(mot →mot.contains("o"));
    System.out.println("aucun mot ne contient un o : " + zero0);

    Optional<String> max = mots.stream()
                        .max((s1,s2) → s1.compareToIgnoreCase(s2));
    System.out.println(max.get());
  }
}
```

```
-------
3
-------
contiennent tous un o : false
-------
au moins 1 mot contient un o : true
-------
aucun mot ne contient un o : false
tout
```

2.7.6) Méthodes terminales

Jusqu'à maintenant la seule méthode terminale que l'on connaissait c'est `forEach`, mais il y en a d'autre.

Remarque | Une **méthode terminale** est une méthode qui ne renvoie pas un flux. Son opposé, la **méthode non-terminale** elle renvoie un flux, *voir les méthodes de la partie précédante*.

- `long count( )` renvoi le nombre d'élément d'un flux
- `boolean allMatch(Predicate<? super T> predicate)` tout les éléments d'un flux vérifient le prédicat ?
- `boolean anyMatch(Predicate<? super T> predicate)` au moins un élément du flux vérifit le prédicat ?
- `boolean noneMatch(Predicate<? super T> predicate)` Aucuns éléments du flux ne vérifient le prédicat ?
- `Optional<T> max(Comparator<? super T> comparator)` quel est le plus grand élément du flux ?
- `Optional<T> min(Comparator<? super T> comparator)` quel est le plus petit élément du flux ?

Exemple du cours

```java
import java.util.List;
import java.util.Optional;

public class Main4 {
  public static void main(String[] args) {
    List<String> mots = List.of("Bonjour", "tout", "le", "monde");
    System.out.println("-------------");
    long nbElements = mots.stream()
                      .filter(mot → mot.contains("o"))
                      .count();
    System.out.println(nbElements);

    System.out.println("-------------");
    boolean tous0 = mots.stream()
                .allMatch(mot →mot.contains("o"));
    System.out.println("contiennent tous un o : " + tous0);

    System.out.println("-------------");
    boolean un0 = mots.stream()
          .anyMatch(mot →mot.contains("o"));
    System.out.println("au moins 1 mot contient un o : " + un0);

    System.out.println("-------------");
```

```
24      boolean zeroO = mots.stream()
25                   .noneMatch(mot →mot.contains("o"));
26      System.out.println("aucun mot ne contient un o : " + zeroO);
27
28      Optional<String> max = mots.stream()
29                           .max((s1,s2) → s1.compareToIgnoreCase(s2));
30      System.out.println(max.get());
31    }
32  }
```

Il y a aussi des méthodes terminales qui permettent de stocker les éléments d'un flux dans un tableau ou dans une liste immuable.

- `Object[] toArray( )` stocker les éléments du flux dans un tableau de type `Object`.
- `<A> A[] to Array(IntFunction<A[]> createurTableau)` créer un tableau de type `A` à n cases. Il faut fournir un expression lambda pour déterminer n.
- `List[T] toList( )` stocker les éléments d'un flux dans une liste immuable.

Exemple

```
1  import java.util.*;
2  import java.util.stream.*;
3
4  public class ExempleStream {
5      public static void main(String[] args) {
6          /* Exemple 1 : Object[] toArray() */
7          Stream<String> flux1 = Stream.of("A", "B", "C");
8          Object[] tableau1 = flux1.toArray();
9          System.out.println("Exemple 1 : " + Arrays.toString(tableau1));
10
11         /* Exemple 2 : <A> A[] toArray(IntFunction<A[]> createurTableau) */
12         Stream<Integer> flux2 = Stream.of(1, 2, 3);
13         Integer[] tableau2 = flux2.toArray(Integer[]::new);
14         System.out.println("Exemple 2 : " + Arrays.toString(tableau2));
15
16         /* Exemple 3 : List<T> toList() */
17         Stream<String> flux3 = Stream.of("X", "Y", "Z");
18         List<String> liste = flux3.toList();
19         System.out.println("Exemple 3 : " + liste);
20
21         /* Test immuabilité de la liste */
22         liste.add("W");
23     }
24 }
```

```
Exemple 1 : [A, B, C]
Exemple 2 : [1, 2, 3]
Exemple 3 : [X, Y, Z]
UnsupportedOperationException
```

2.7.7) Flux infinis et évaluation paresseuse

Rappel de cours 16

En Haskell, on peut définir des listes infinies grâce à l'évaluation paresseuse :

- `[0..]` liste des entiers naturels

- `[0, 2, ..]` liste des pairs

On peut faire la même chose avec les flux en Java.

2.7.7.1) *La méthode* `iterate`

La méthode d'interface `iterate(T seed, UnaryOperator<T> f)` permet de générer un flux infini :

- `seed`, est la première valeur
- f la fonction qui détermine à partir de `seed` la valeur suivante

Remarque | Puisque l'évaluation est paresseuse, lors de la création d'un flux infini, seuls les éléments utilisé au moment de la méthode terminale seront calculés.

Exemple du cours

```java
import java.util.stream.Stream;

public class Main5ter {
  public static void main(String[] args) {
    Stream.iterate(0,  n→ n + 2)
          .map(n → n * n)
          .takeWhile(n → n <= 5000)
          .forEach(n → System.out.println(n));
  }
}
```

```
0
4
16 ...
```

2.7.8) *Tris sur les fulx avec la méthode* `sorted( )`

Remarque | La méthode `.sort( )` sans paramètre utilise en fait la méthode `compareTo` du type de données du flux.

Exemple du cours

```java
import java.util.List;

public class Main7 {
  public static void main(String[] args) {
    List<String> mots = List.of("Bonjour", "tout", "le", "monde");
    mots.stream()
        .sorted()
        .forEach(m → System.out.println(m));
  }
}
```

```
Bonjour
le
monde
tout
```

Ici le tri se fait dans l'ordre lexicographique.