



Pourquoi la programmation fonctionnelle ?



#1 Les effets de bord

#1 Les effets de bord

- Mutation d'une variable
- Ecriture dans la console
- Ecriture dans un fichier de log
- Communication avec une base de données
- Lancement d'une exception
- ...

#1 Les effets de bord

// la signature de la méthode n'indique pas qu'une exception pourrait être lancée.

```
public Integer diviser(Integer nombre1, Integer nombre2) {
```

```
    // déclenche une exception si nombre2 vaut 0
```

```
    return nombre1 / nombre2;
```

```
}
```

#1 Les effets de bord

```
// la signature de la méthode n'indique pas qu'une exception pourrait être lancée.  
public Integer diviser(Integer nombre1, Integer nombre2) {  
    // déclenche une exception si nombre2 vaut 0  
    return nombre1 / nombre2;  
}
```

Pour l'utilisateur d'une telle méthode, il y a clairement une rupture du flux d'exécution.

#1 Les effets de bord

```
// la signature de la méthode n'indique pas qu'une exception pourrait être lancée.  
public Integer diviser(Integer nombre1, Integer nombre2) {  
    // déclenche une exception si nombre2 vaut 0  
    return nombre1 / nombre2;  
}
```

*La programmation fonctionnelle préconise d'utiliser une structure **expressive** avec un typage exprimant le fait que cette méthode peut produire une erreur.*

#1 Les effets de bord

```
// la signature de la méthode n'indique pas qu'une exception pourrait être lancée.  
public Integer diviser(Integer nombre1, Integer nombre2) {  
    // déclenche une exception si nombre2 vaut 0  
    return nombre1 / nombre2;  
}
```


*Pour ce cas de figure, Vavr fournit la structure **Try**.*

#1 Les effets de bord

```
// la signature de la méthode indique deux cas de sortie possibles :  
// = Success(resultat)  
// = Failure(exception)  
public Try<Integer> diviser(Integer nombre1, Integer nombre2)  
{  
  
    // l'exception est interceptée par la structure  
    return Try.of(() -> nombre1 / nombre2);  
}
```




#2 Transparence référentielle



Une fonction ou une expression est dite transparente référentielle si son invocation peut être remplacée par une valeur sans affecter le fonctionnement du programme.



Pour une fonction, les mêmes entrées produisent
TOUJOURS les mêmes sorties.



// n'est pas transparente référentielle
`Math.random();`

// est transparente référentielle
`Math.max(1,2);`



Pureté d'une fonction



Une fonction est dite *pure* si toutes les expressions qui la compose sont référentielles transparentes.



#3 Penser en valeurs



Les valeurs les plus intéressantes sont celles qui
sont **immuables**



Elles ne produisent pas d'effets de bord dans un
contexte Multi-threads.



Leurs méthodes *equals* et *hashCode* sont
stables dans le temps




Elles n'ont pas besoin d'être clonées.



Structures de données fonctionnelle



Une structure de données fonctionnelle est
immuable et persistante.



Une structure de données est dite **persistante** quand elle **conserve ses précédentes versions** lorsqu'elle est modifiée.



Java 8 et la programmation fonctionnelle

Quelques défauts de Java 8

- Peu de structures fonctionnelles
- Pas de mémorisation, lifting sur les fonctions
- Pas de tuples
- Optional pas sérialisable, pas itérable
- Les exceptions...
- Des API non fonctionnelles...

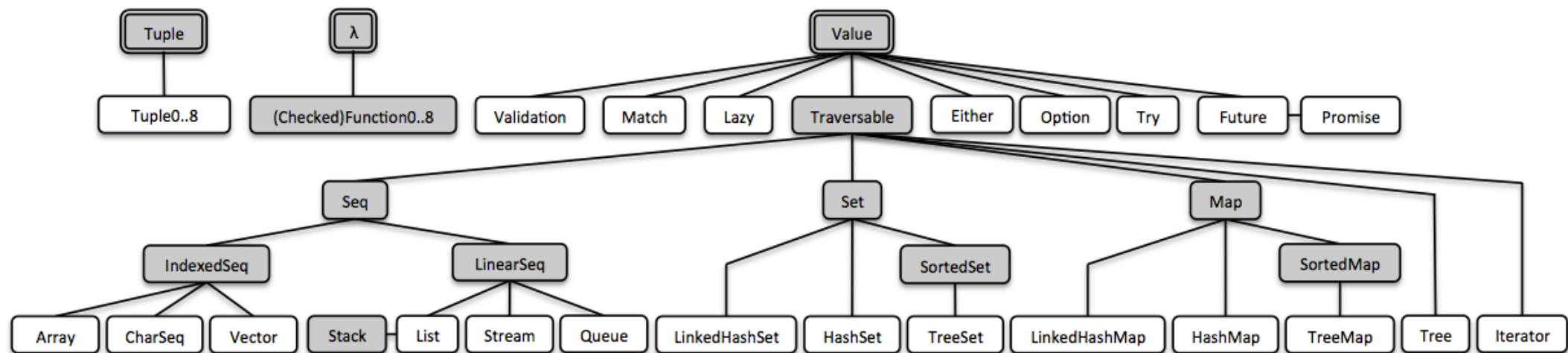


Vavr



Vavr ?

Vavr (anciennement appelé Javaslang) est une librairie Java 8+ fournissant des structures facilitant la programmation fonctionnelle.



Tuples

```
Tuple2<Integer, Integer> t1 = Tuple.of(12, 18);
```

```
assert t1._1 == 12;
```

```
assert t1._2 == 18;
```

```
Tuple3<String, Integer, Integer> t1 = Tuple.of("hello", 12, 18);
```

Fonctions

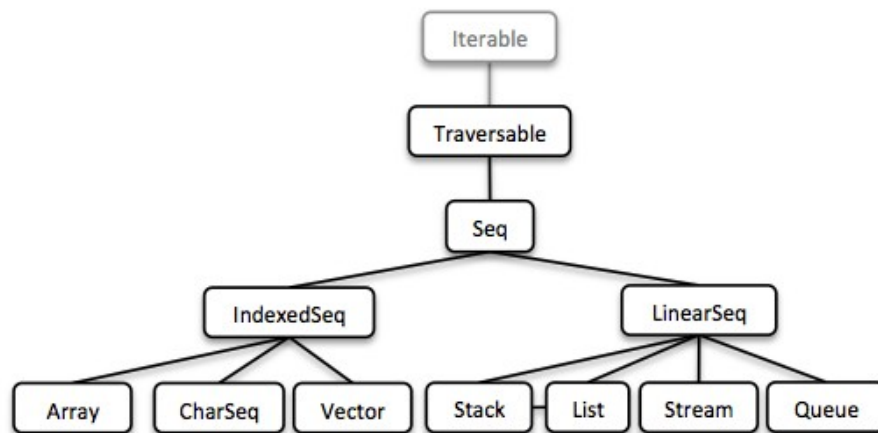
- **Function0...8**
- **CheckedFunction0...8**
- **Composition** : `fonction1.andThen(fonction2)`
- **Lifting** : Encapsule les erreurs et transforme le résultat en `Optional`
- **Application partielle** : `F(x1, x2, x3)` s'utilise `F(x1)(x2, x3)`
- **Currying** : `F(x1, x2, x3)` s'utilise `F'(x1)(x2)(x3)`
- **Memoization** : du cache de résultat

Values

- (Toutes itérables)
- **Validation** : gestion des erreurs de validation
- **Either** : Encapsule 2 résultats <KO, OK>
- **Lazy** : Evaluation tardive.
- **Match** : Pattern Matching
- **Option** : un Optional plus fonctionnel
- **Try** : catch d'exception et produit un Success ou un Failure
- **Future & Promesse** : alternative à CompletableFuture

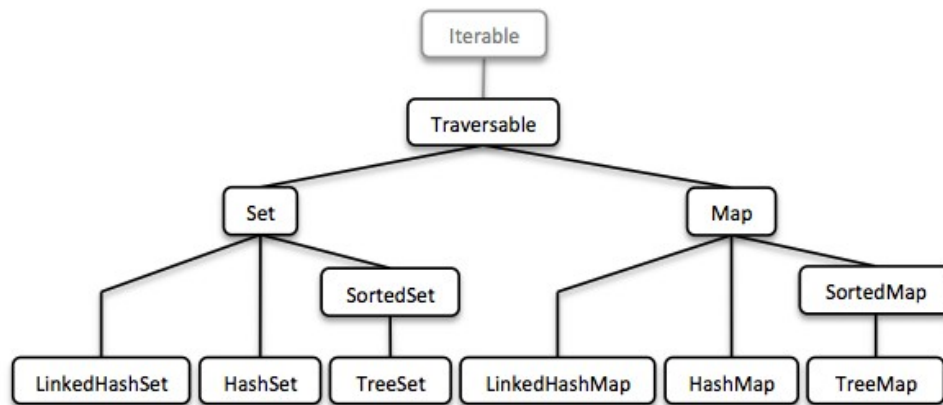
Collections - Seq

- Toutes immuables et persistantes
- API plus agréable que les streams Java 8



Collections - Set & Map

- Toutes immuables et persistantes
- API plus agréable que les streams Java 8





http://www.vavr.io/vavr-docs/#_usage_guide