

Foncteurs, en théorie et en pratique



Moi, moi, moi !

ebiznext

- Animateur data [@Ebiznext](#)
- [Scala Nantes UG](#)
- [Passionné de FP](#)
- [@mmenestret](#)
- [geekocephale.com](#)

Agenda

- Une intuition
- En pratique ...
- ... en théorie ...
- ... retour à la pratique !
- Résumons
- Plus, toujours plus !

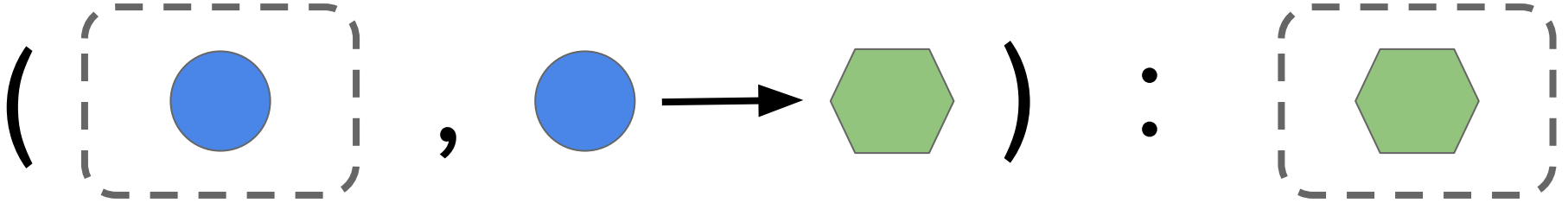
Une intuition



- **Conteneur**
- Dont le **contenu** peut être **transformé**, de façon générique, sans altérer le conteneur
- Mais en respectant **des lois**
 - Identité
 - Composition

La fonction map





ebiznext



Un pattern commun

`map(Option[●], ● → ●) == Option[●]`

Un pattern commun

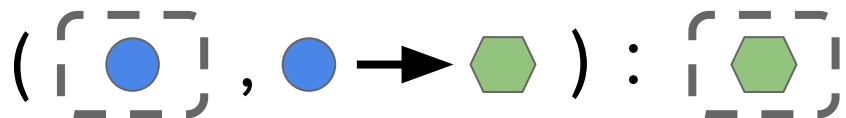
`map(List[,  → ) == List[]`

Un pattern commun

```
case class Container[A](value: A)
```

```
map( Container[●], ● → ● ) == Container[●]
```

Un pattern commun



$(F[A], A \Rightarrow B) : F[B]$



```
def map[A, B](fa: F[A])(f: A => B): F[B]
```

Un pattern commun



[Functors, Applicatives, And Monads In Pictures](#)

Un foncteur **F** est un conteneur dont le contenu peut être **transformé** sans altérer le conteneur

En pratique ...

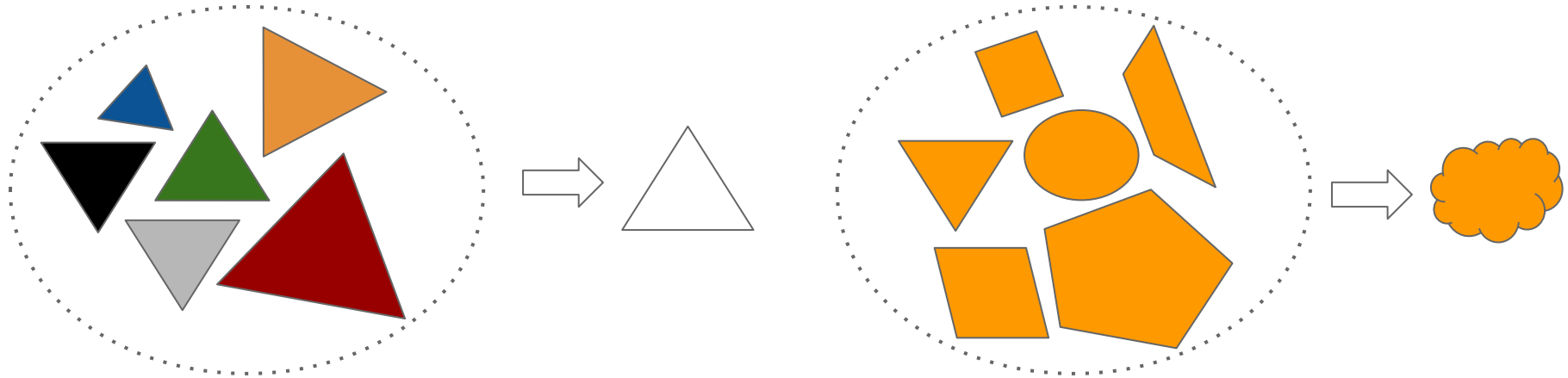


Les foncteurs en Scala

*Les foncteurs sont encodés par une **type class** de **constructeur de type***

Type class - Définition

*Un **ensemble** (une classe) des **types** ayant un **contrat commun***



Type class - Définition

Une **type class** est encodée, en Scala, par:

- Une **trait** avec un **paramètre de type**
- Une **instance implicite** pour chaque type
- Les **preuves** du respect des lois

Type class - Exemple

```
trait Show[A] {  
  def show(a: A): String  
}
```


Type class - Example

```
implicit val intShow: Show[Int] = new Show[Int] {  
  def show(a: Int): String = s"I'm an Int: $a"  
}
```

Type class Foncteur - Trait

```
trait Functor[F[_]] {  
    def map[A, B](fa: F[A])(f: A => B): F[B]  
}
```

Type class Foncteur - Trait

```
trait Functor[F[_]] {  
    def map[A, B](fa: F[A])(f: A => B): F[B]  
}
```

Constructeur de type

Un **constructeur de type** permet de créer de nouveaux **types**

- *List[_] + Int* → *List[Int]*
- *Map[_] + String + Int* → *Map[String, Int]*
- *Container[_] + Boolean* → *Container[Boolean]*

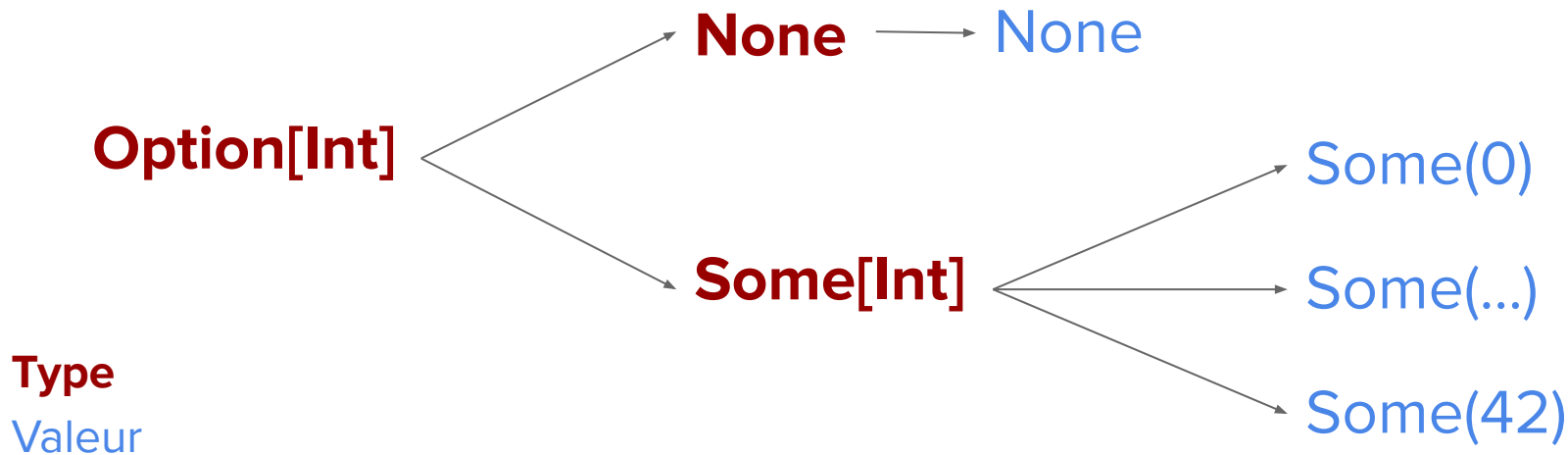
Type class Foncteur - Trait

```
trait Functor[F[_]] {  
  def map[A, B](fa: F[A])(f: A => B): F[B]  
}
```

Type class Foncteur - Instance

À quoi ressemble l'instance d'Option ?

Pour rappel:



Type class Foncteur - Instance

```
implicit val oFunctor = new Functor[Option] {  
  def map[A, B](fa: Option[A])(f: A => B): Option[B] =  
    fa match {  
      case Some(x) => Some( f(x) )  
      case None => None  
    }  
}
```

Type class Foncteur - Instance

```
case class Container[A](value: A)
```

```
implicit val cFunctor = new Functor[Container] {  
  def map[A, B](fa: Container[A])(f: A => B): Container[B] = {  
    val contenuTransforme = f(fa.value)  
    Container(contenuTransforme)  
  }  
}
```


Type class Foncteur - Lois

Pour tout **c** de type **Container[_]**:

- *Identité*: **map(c, identity) == c**
- *Composition associative*:
map(map(c, f), g) == map(c, f andThen g)
- Tester les lois avec Cats et Discipline

... en théorie ...



**UN FONCTEUR N'EST QU'UN
MAPPING ENTRE 2 CATÉGORIES
PRÉSERVANT LEURS STRUCTURES**

ebiznext



**QUEL EST
LE PROBLÈME ?**

Théorie des catégories

ebiznext

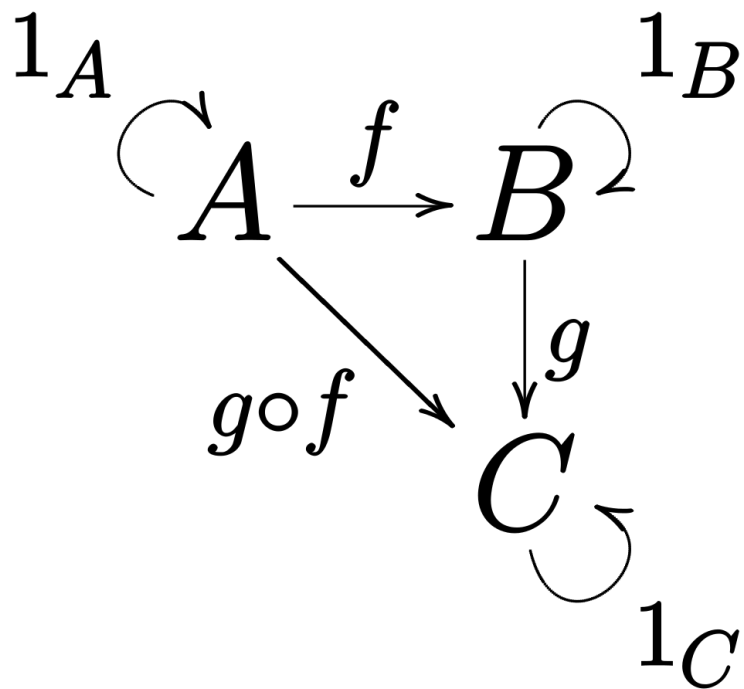
“L’étude des **relations** entre les **choses**”

Théorie des catégories

Une catégorie est composée:

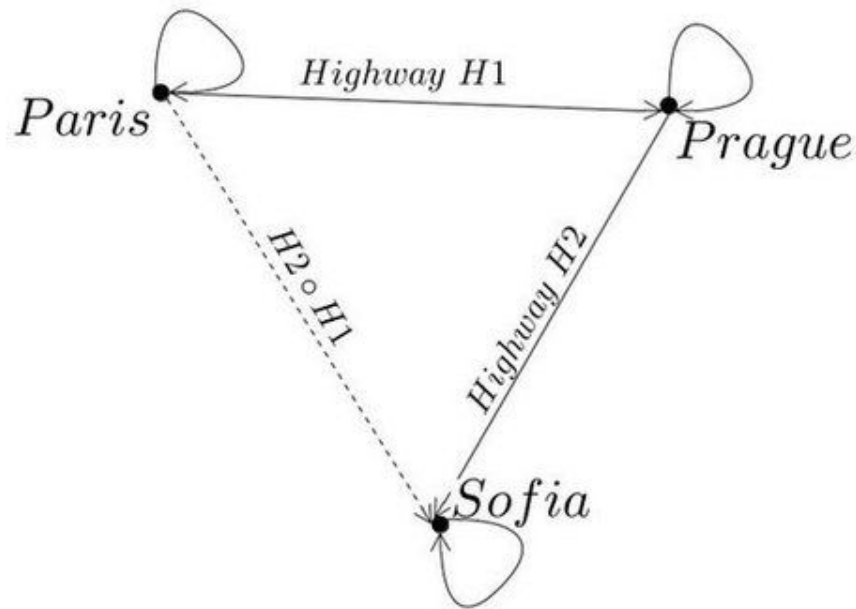
- **D'objets** (les choses)
- **De morphismes** (les relations)
 - Devant respecter 2 lois:
 - **Identité**
 - **Composition associative**

Théorie des catégories



- **Objets:** A , B , C
- **Des morphismes:** f , g , $g \circ f$, ...
 1. $g \circ f$ est la **composition** de f et g . Elle doit exister !
 2. **1_A , 1_B et 1_C** sont les morphismes identité de **A , B et C** . Ils doivent exister !

Théorie des catégories



Un réseau routier forme une catégorie.
Voir [Functional Programming and Category Theory](#)

Quel rapport avec Scala ?

Une catégorie importante, **S** (comme **S**cala):

- **Objets**: Les **types** de Scala
- **Morphismes**: Les **fonctions** de Scala
 - **Identité**: la fonction identité
 - **Composition associative**: la composition de fonctions est associative

Quel rapport avec Scala ?

- **Objets:** Int, List[Boolean], Unit, ...
- **Morphismes:** Int => String, List[Int] => Int, ...
 - **Identité**
 - **def** identity[A](a: A): A = a
 - **Composition associative**
 - **f** andThen (**g** andThen **h**)
≡
(**f** andThen **g**) andThen **h**

Quel rapport avec Scala ?

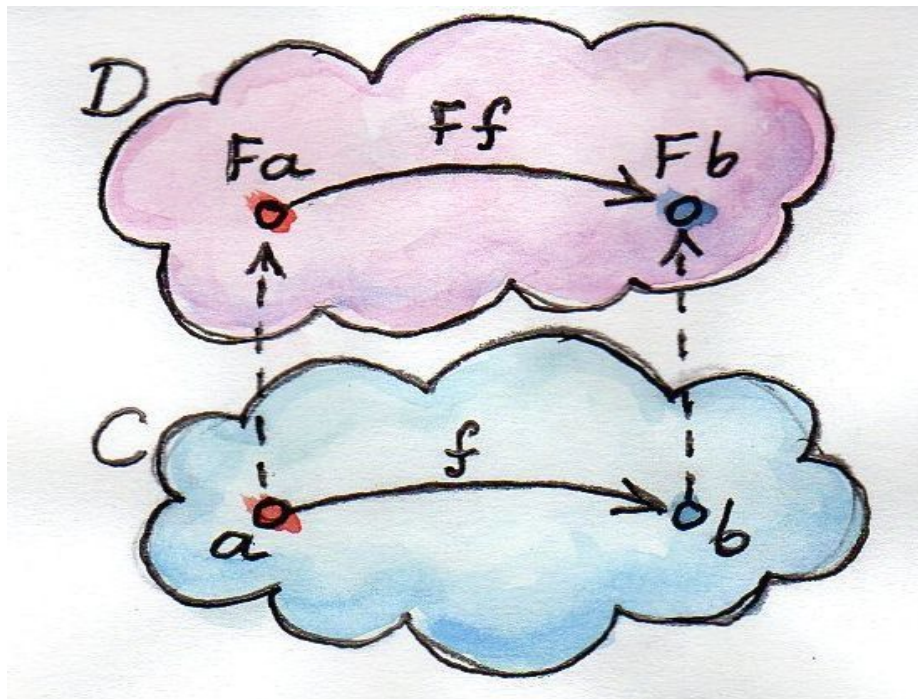
La théorie des catégories nous intéresse puisque **\mathcal{S}** est une catégorie et que nous développons dans **\mathcal{S}** !

Retour à nos foncteurs !

“Un **mapping** entre 2 catégories qui préserve leurs structures”

- **Objets** de **C** → **objets** de **C'**
- **Morphismes** de **C** → **morphismes** de **C'**
- **Préserve la structure** (les morphismes & les compositions)

Retour à nos foncteurs !

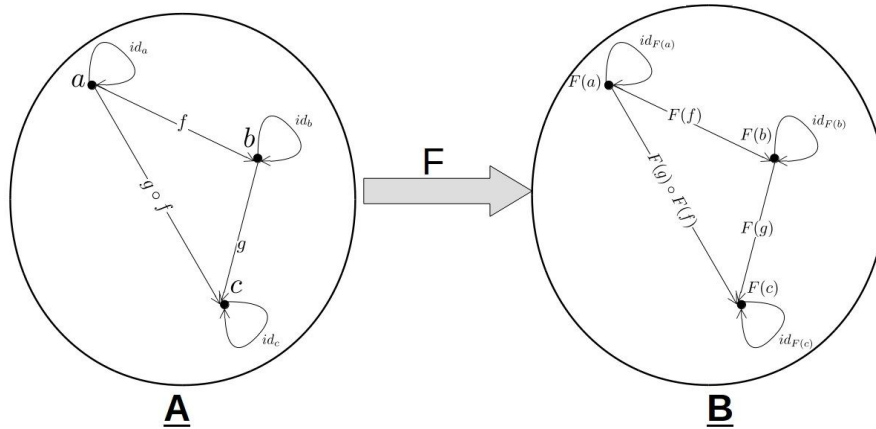


Les **endofoncteurs** mappent une **catégorie** à elle même.

Les endofoncteurs dans **S**:

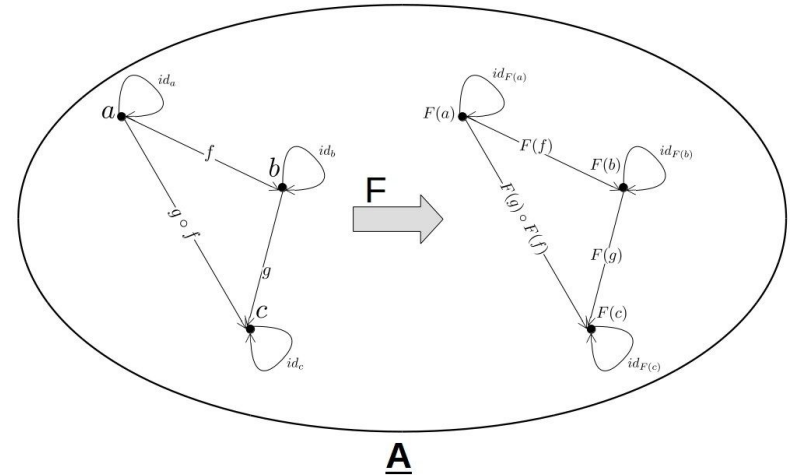
- **Mappent** les types *Scala* vers des types *Scala*
- **Mappent** les **fonctions** *Scala* vers des **fonctions** *Scala*

Endofoncteurs



Un foncteur de A dans B .

Voir [Functional Programming and Category Theory](#)



Un endofoncteur.

Voir [Functional Programming and Category Theory](#)

Les foncteurs en Scala

Objets: Le constructeur de type **Option[_]** mappe un type en un autre

Objets dans S	Objets dans S' (S' == S)
Int	Option[Int]
String	Option[String]
A	Option[A]

Les foncteurs en Scala

Morphismes

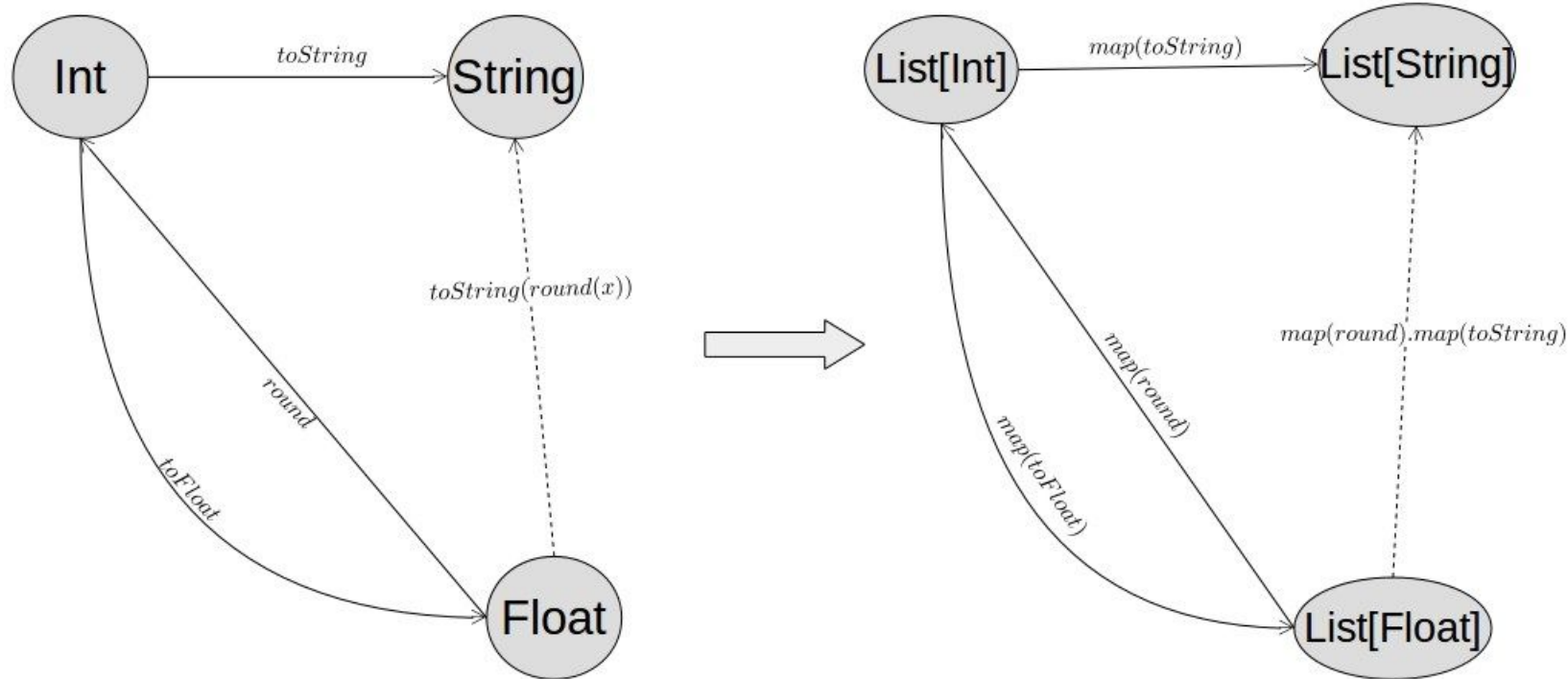
- **def** map[A, B](fa: F[A])(f: A => B): F[B]
- map(_)(f) avec **f**: **A** => **B**, donne une nouvelle fonction **F[A] → F[B]** !

Les foncteurs en Scala

Morphismes: La fonction **map** mappe une fonction $A \rightarrow B$ en une autre $F[A] \rightarrow F[B]$

Morphismes de S	Morphismes de S' ($S' == S$)
$\text{Int} \rightarrow \text{String}$	$\text{Option}[\text{Int}] \rightarrow \text{Option}[\text{String}]$
$A \rightarrow A$ (identity)	$\text{Option}[A] \rightarrow \text{Option}[A]$
$A \rightarrow B$	$\text{Option}[A] \rightarrow \text{Option}[B]$

Les foncteurs en Scala



Le foncteur List.

Voir [Functional Programming and Category Theory](#)

**... retour à la
pratique !**



Bénéfices - Abstraction

Les foncteurs abstraient et les contraintes libèrent:

- Améliorent la réutilisabilité du code
- Réduit les opportunités d'erreur

```
def cantDoMore[F[_]: Functor, A](fa: F[A]) = fa.map(...)
```

Il est possible de représenter des **valeurs** accompagnée d'un effet:

- **Option[A]:** **A + l'effet d'absence possible de valeur**
- **Try[A]:** **A + l'effet expression d'un échec**
- **IO[A]:** **A + effet de bord**
- ...

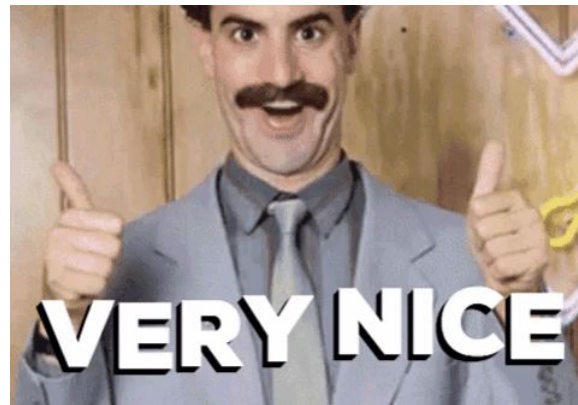
Les foncteurs permettent de transformer, de façon générique, une valeur avec effet sans en modifier l'effet

Résumons




Foncteurs

- **Conteneur** pouvant être **mappé**
- **Encodé** par une **type class**
- **Théorie**: un mapping entre 2 catégories qui en préserve les structures
- **Endofoncteurs** sont des mapping de **S** dans **S**
- **$S(cala)$** est une catégorie (types et fonctions)
- **Super pouvoirs** (abstraction & transformation générique)



**Plus, toujours
plus !**




Comment faire ?

Injecter, dans un conteneur, un contenu **de façon générique** ?

→  Les foncteurs ne proposent **que la fonction map** qui transforme le contenu d'un **conteneur existant**

Comment faire ?

Appliquer à une fonction à **plus d'un paramètre**, à **des arguments wrappés dans des conteneurs** ? Combiner plusieurs conteneurs ?

→  **map** ne peut appliquer qu'une **fonction à 1 paramètre** à **un conteneur**

Les foncteurs applicatifs

ebiznext

Enrichissent nos foncteurs:

```
trait Applicative[F[_]] extends Functor[F] {  
  def pure[A](a: A): F[A]  
  def ap[A, B](ff: F[A => B])(fa: F[A]): F[B]  
}
```

Les foncteurs applicatifs

ebiznext

La fonction **pure** permet de **lifter**, d'insérer, une valeur dans un conteneur de façon **purement générique**.

```
Applicative[Option].pure(42) // Some(42)
```

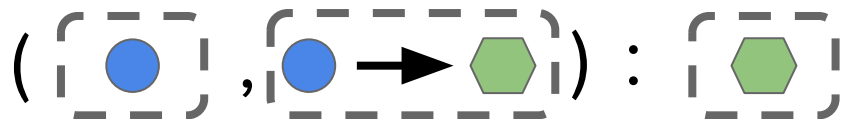
Les foncteurs applicatifs

ebiznext

L'utilisation de la fonction **ap** est moins évidente, décortiquons:

```
def map[A, B](fa: F[A])(f: A => B): F[B]
```

```
def ap[A, B] (fa: F[A])(f: F[A => B]): F[B]
```



🌈 Les foncteurs applicatifs 🌈

Comment **appliquer** une **fonction** à **2 paramètres** ?

Soit: $f: (A, B) \Rightarrow C$ $fa: F[A]$ $fb: F[B]$

1. `val ff = fa.map(a => f(a, _))`
→ $F[B \Rightarrow C]$, maintenant, comment l'appliquer à fb ?
2. `ap(ff)(fb)`
→ $F[C]$, sans `ap` on serait bloqué !

Les foncteurs applicatifs

ebiznext

Simplifions nous la vie !

```
trait Applicative[F[_]] extends Functor[F] {  
  def map2[A, B, C](fa: F[A], fb: F[B])(f: (A, B) => C): F[C]  
  def map3[A, B, C, D](fa: F[A], fb: F[B], fc: F[C])(f: (A, B, C) => D): F[D]  
  ...  
}
```

```
val res: F[C] = map2(fa, fb)(f)  
val res2: F[C] = (fa, fb).mapN(f)
```



Les foncteurs applicatifs

ebiznext

Devoirs à la maison:

1. Comment appliquer une fonction à **N paramètres**, à **N arguments** wrappées dans $F[_]$ **uniquement** avec **map** et **ap** ?
2. **Applicative** est plus “puissant” que **Foncteur**, prouvez le en **ré-implémentant map** en utilisant uniquement **ap** et **pure** !

Et ensuite ?

ebiznext



Further reading

- [Constraints Liberate, Liberties Constrain](#)
- [Scala with Cats - Functor](#)
- [Functors - Bartosz Milewski](#)
- [Functors, Applicatives, And Monads In Pictures](#)
- [Category Theory & Programming](#)
- [Functors - FP Foundation](#)
- [Categories and Functors](#)

Merci !
Questions (faciles) ?

