Programmation fonctionnelle en Golo

Philippe Charrière

https://github.com/k33g/q/issues

@k33g

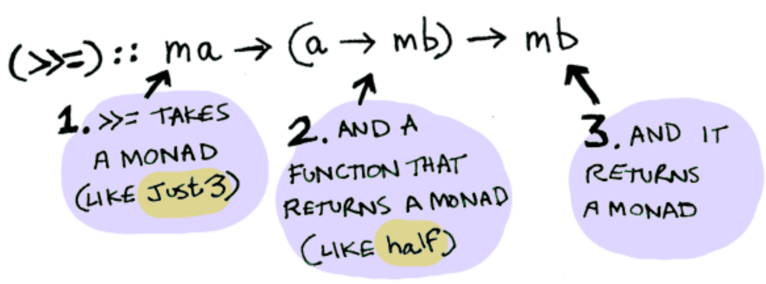
@k33g_org

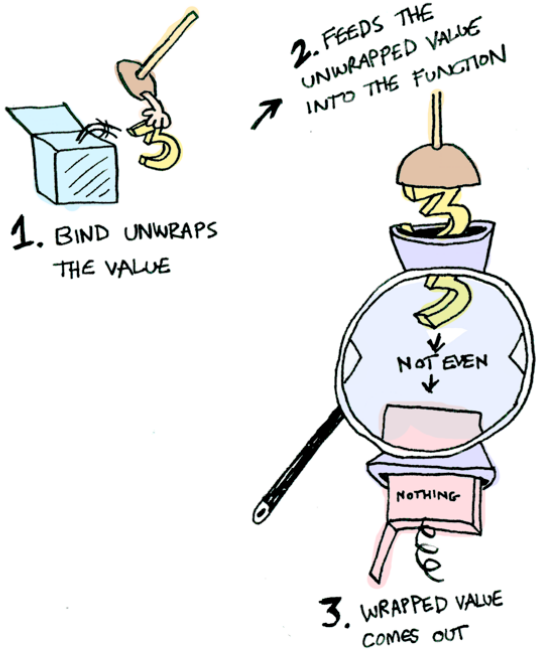
GitHub

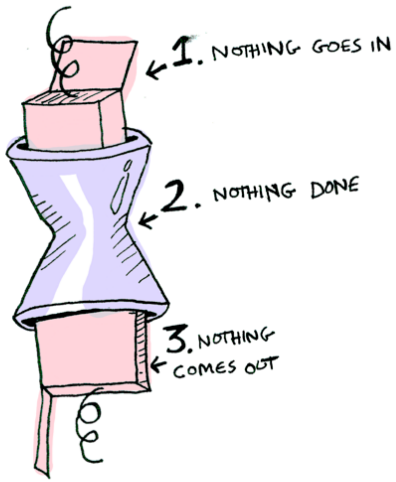
mix-IT

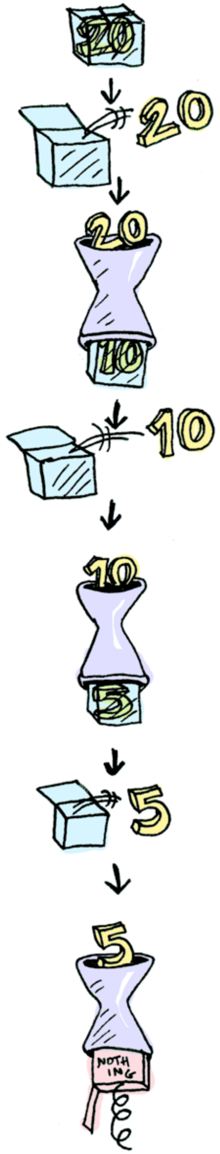
Pourquoi ce talk? … Pour qui?

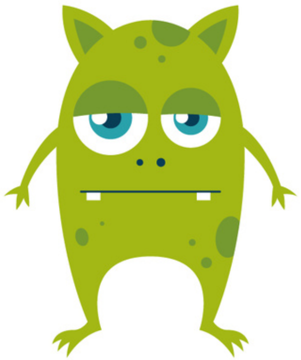
http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html











Précautions d’usage

> vocabulaire(s)
> implémentations incomplètes
> échanges de points de vue:
https://github.com/k33g/web.2.day.golo/issues

Programmation fonctionnelle?

Définition

> Coder avec uniquement des fonctions pures
> = fonctions sans “side effect”
> > toujours retourner le même résultat
> > ne pas modifier de variable
> > ne pas propager d’exception ou s’arrêter lors d’une erreur
> > pas de print, pas de input
> > pas de lecture / écriture de fichier
> > ...

développeurs fonctionnels

Nous allons voir

> 2, 3 trucs utiles
> Container
> Functor
> Monad
> Maybe
> Either
> Applicative Functor
> Validation

Golo

???

Golo ?

http://golo-lang.org/

1 langage **dynamique** pour la **JVM**

à base **d'Invokedynamic**

facile à utiliser

facile à modifier

& c'est un projet **Eclipse**

Golo ?

Créé par **@jponge** (aka “le doc”)

équipe **DynaMid** du labo citilab

Golo ?

Léger 708 kb

Golo **<3** Java

Golo est **rapide**

dans un contexte dynamique #notroll

& en environnement contraint

Golo: modèle objet

simple et efficace,

proche de celui de Go,

composition plutôt que héritage,

approche mixte objet / fonctionnelle

Quelques structures caractéristiques

Closures

```
let add = |a, b| {
  return a + b
}

let multiply = |a, b| -> a * b
```

DynamicObjects

```
let human =
  DynamicObject()
    : firstName("Bob")
    : lastName("Morane")
    : define("hello", |this| {
        return "Hello " + this: firstName() + " " + this: lastName()
      })
```

Structures

```
struct Human = {
  firstName,
  lastName
}

augment Human {
  function hello = |this| -> println("Hello " + this: firstName() +
     this: lastName())
  function hello = |this, message| -> println(message)
}
```

To be continued…

Prog. Fonctionnelle
Quelques bases utiles

“**currying** is a way of constructing functions that allows partial application of a function’s arguments”

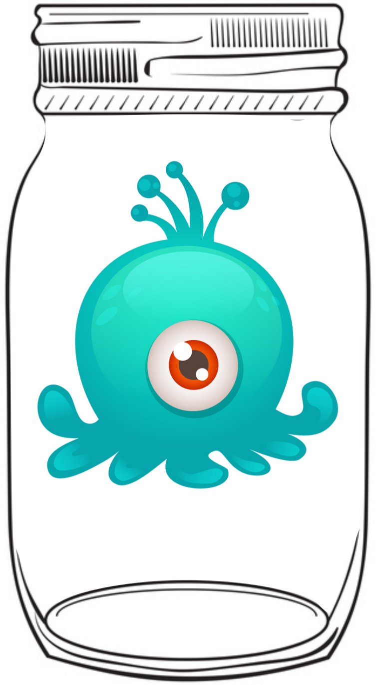
```
let addition = |a,b| -> a + b

# currying addition
let add = |a| -> |b| -> a + b
```

```
add(40)(2)

let incrementBy1 = add(1)
let incrementBy2 = add(2)

incrementBy1(41) # 42

let arr1 = [100, 200, 300, 400, 500]

arr1: map(|x| -> x + 1) # [101, 201, 301, 401, 501]

arr1: map(incrementBy1) # [101, 201, 301, 401, 501]
```

“Function **composition** is the act of pipelining the result of one function, to the input of another, creating an entirely new function.”

```
let compose2 = |f, g| {
  return |x| {
    return f(g(x))
  }
}

# |f, g| -> |x| -> f(g(x))

let compose2 = |f, g, h| {
  return |x| {
    return f(g(h(x)))
  }
}
```

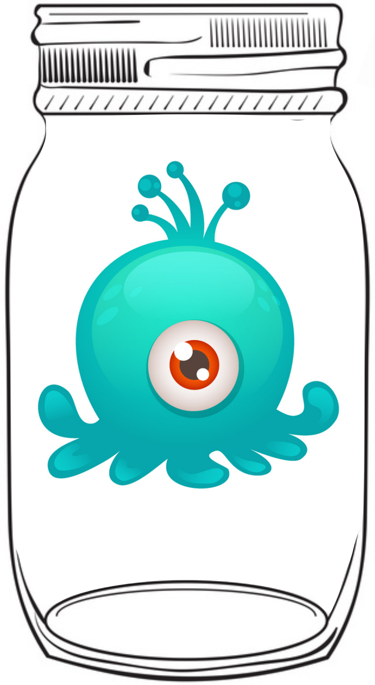
Container

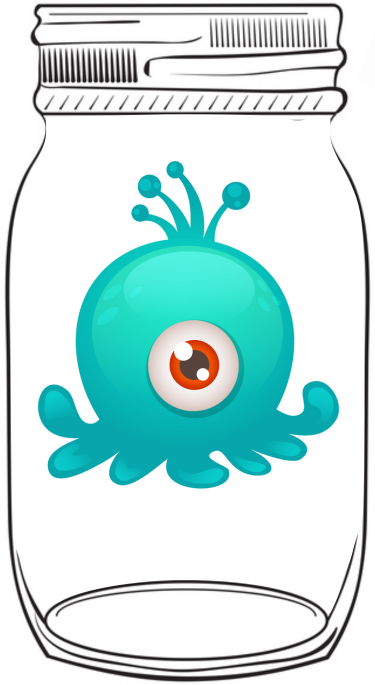


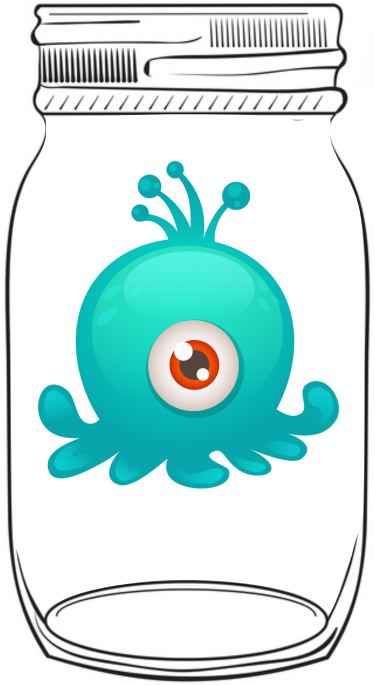
```
struct Container = {
  _value # private variable
}

augment Container {
  # kind of constructor
  function of = |this, value| -> Container(value)
  function value = |this| -> this: _value()
}
```

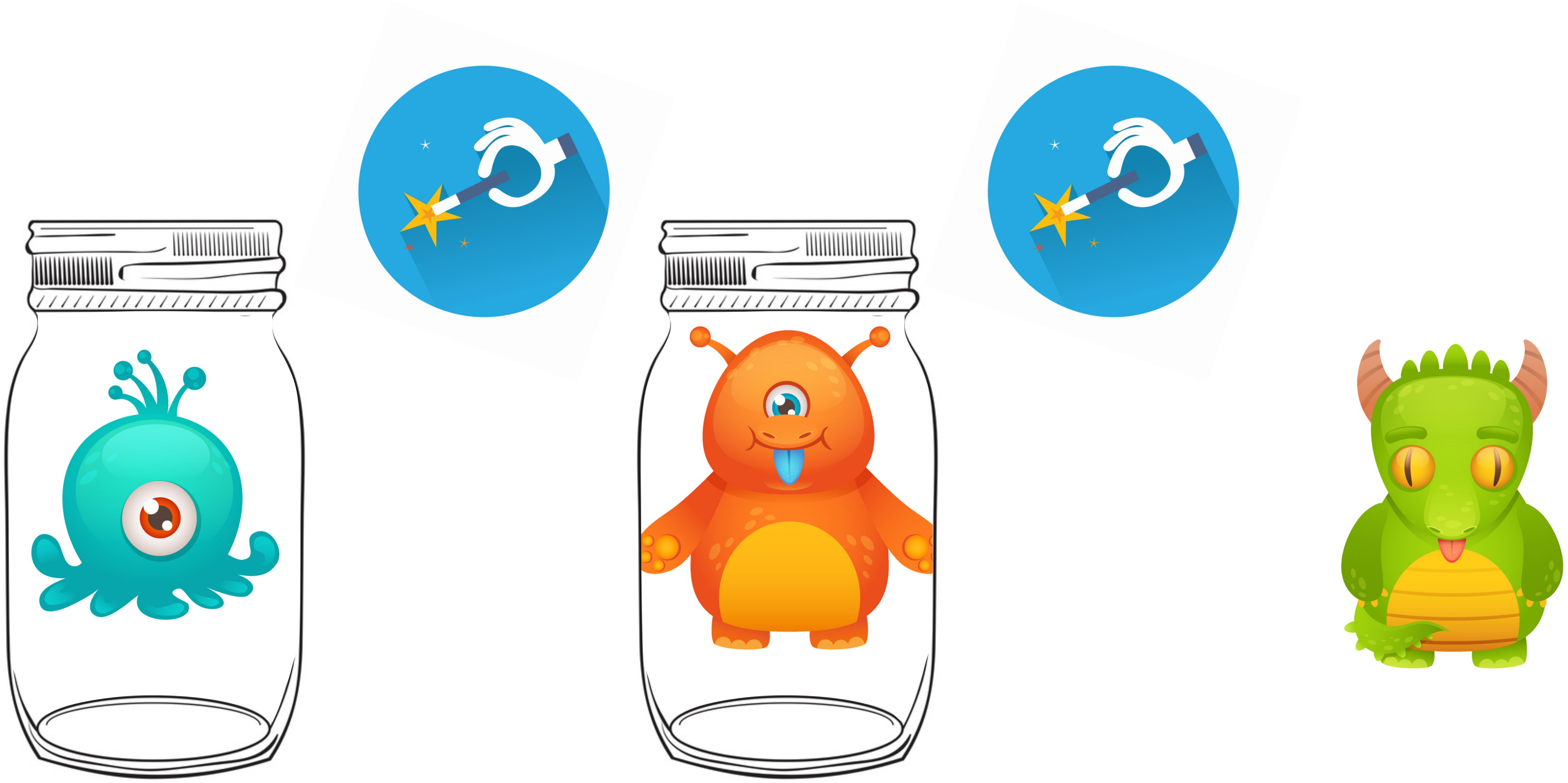
```
function main = |args| {

 let bob = Container("Bob Morane")
 println(bob: value())

 let john = Container(): of("John Doe")
 println(john: value())

}
```

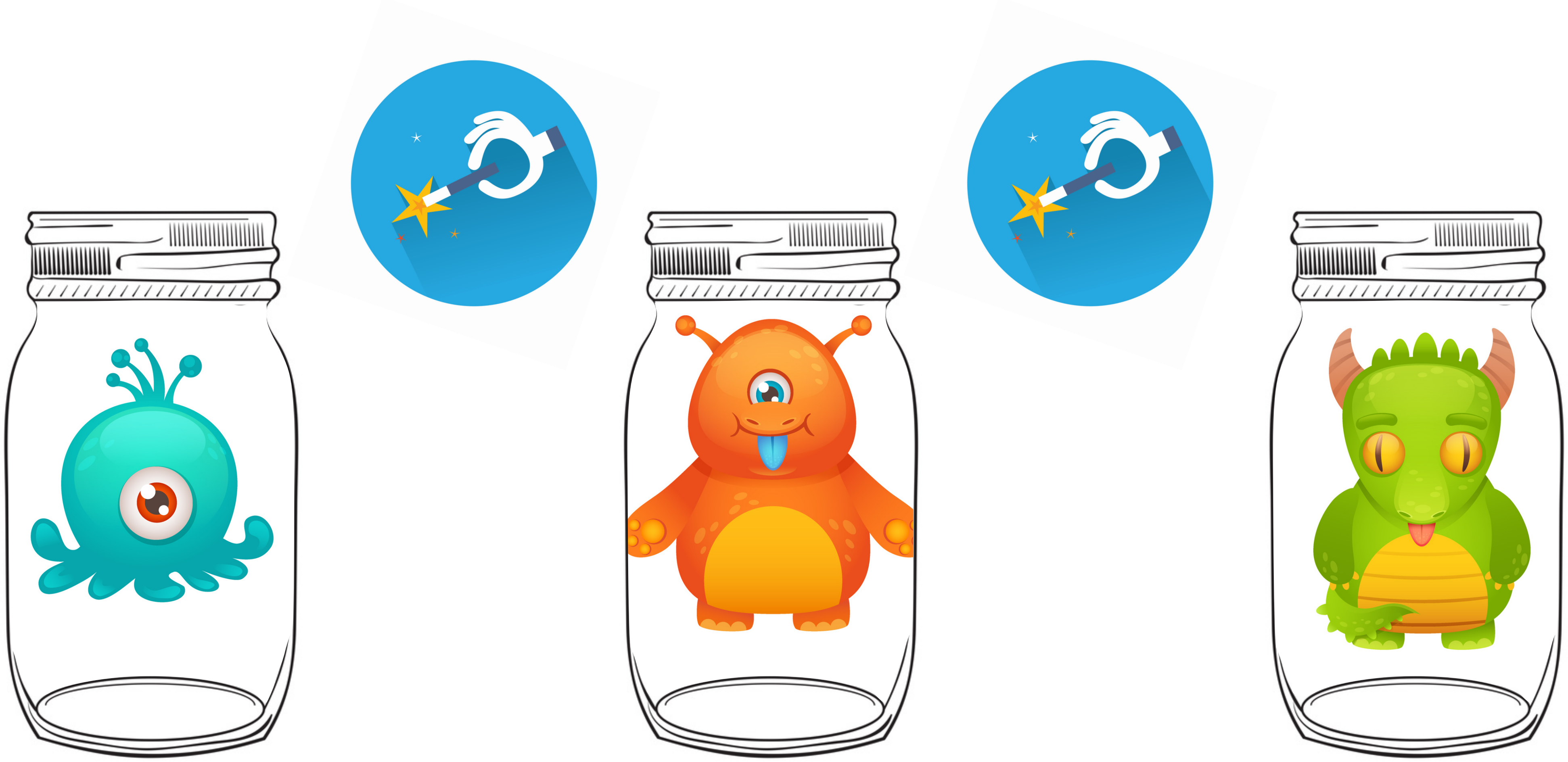
Functor



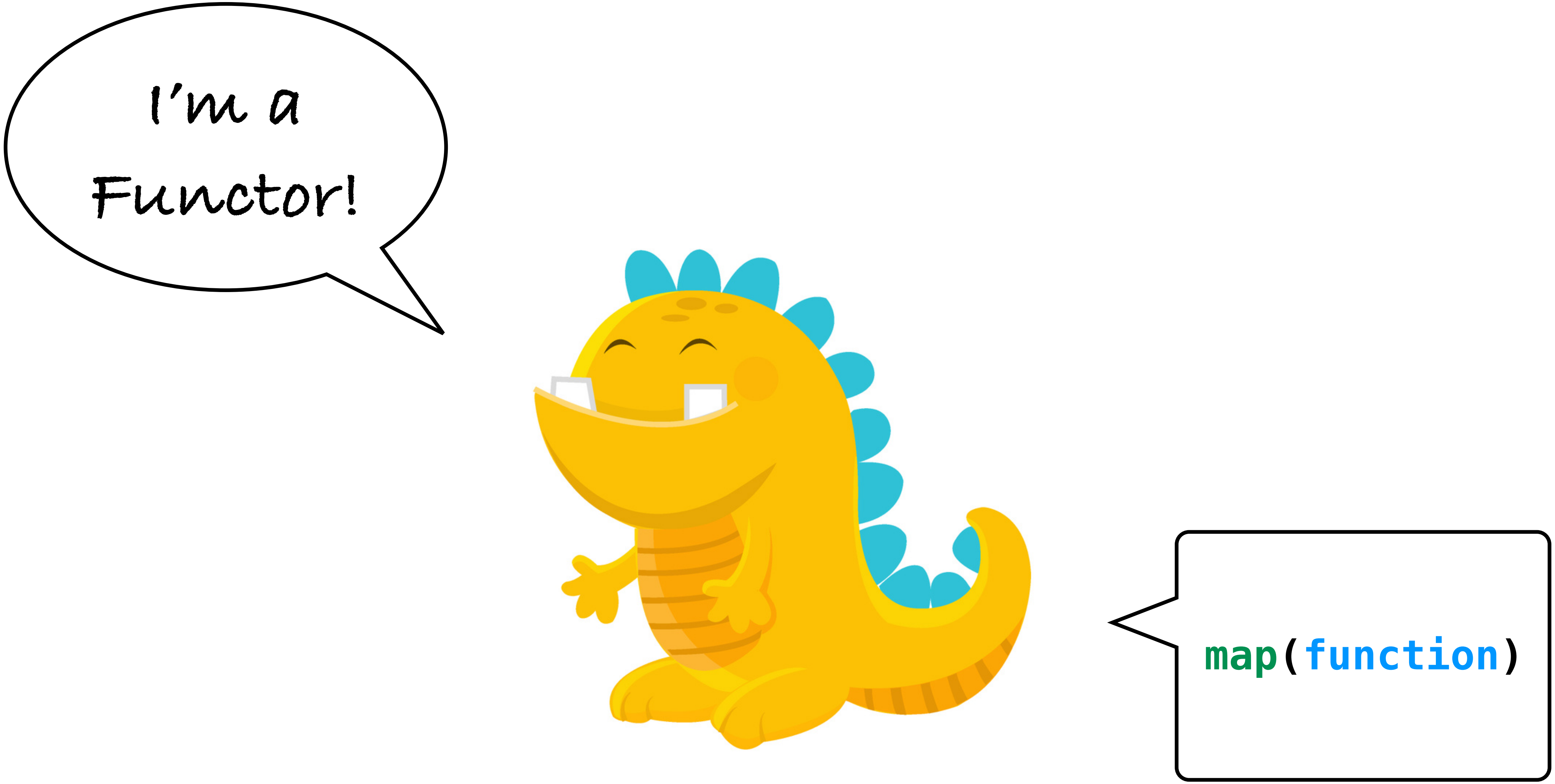








```
struct Functor = {
  _value # private variable
}

augment Functor {
  # kind of constructor
  function of = |this, value| -> Functor(value)
  function value = |this| -> this: _value()

  function map = |this, fn| -> Functor(): of(fn(this: _value()))
}
```

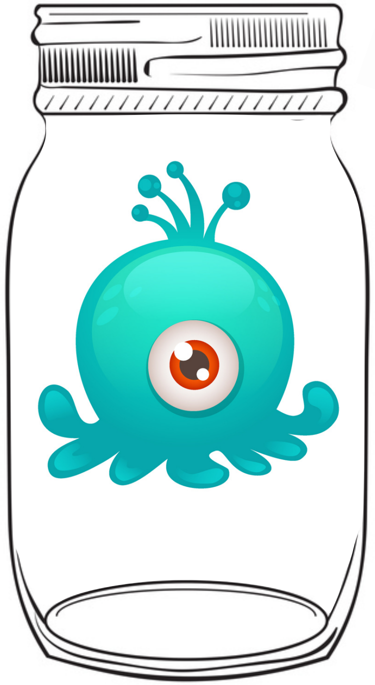

I'm a Functor!
map(function)

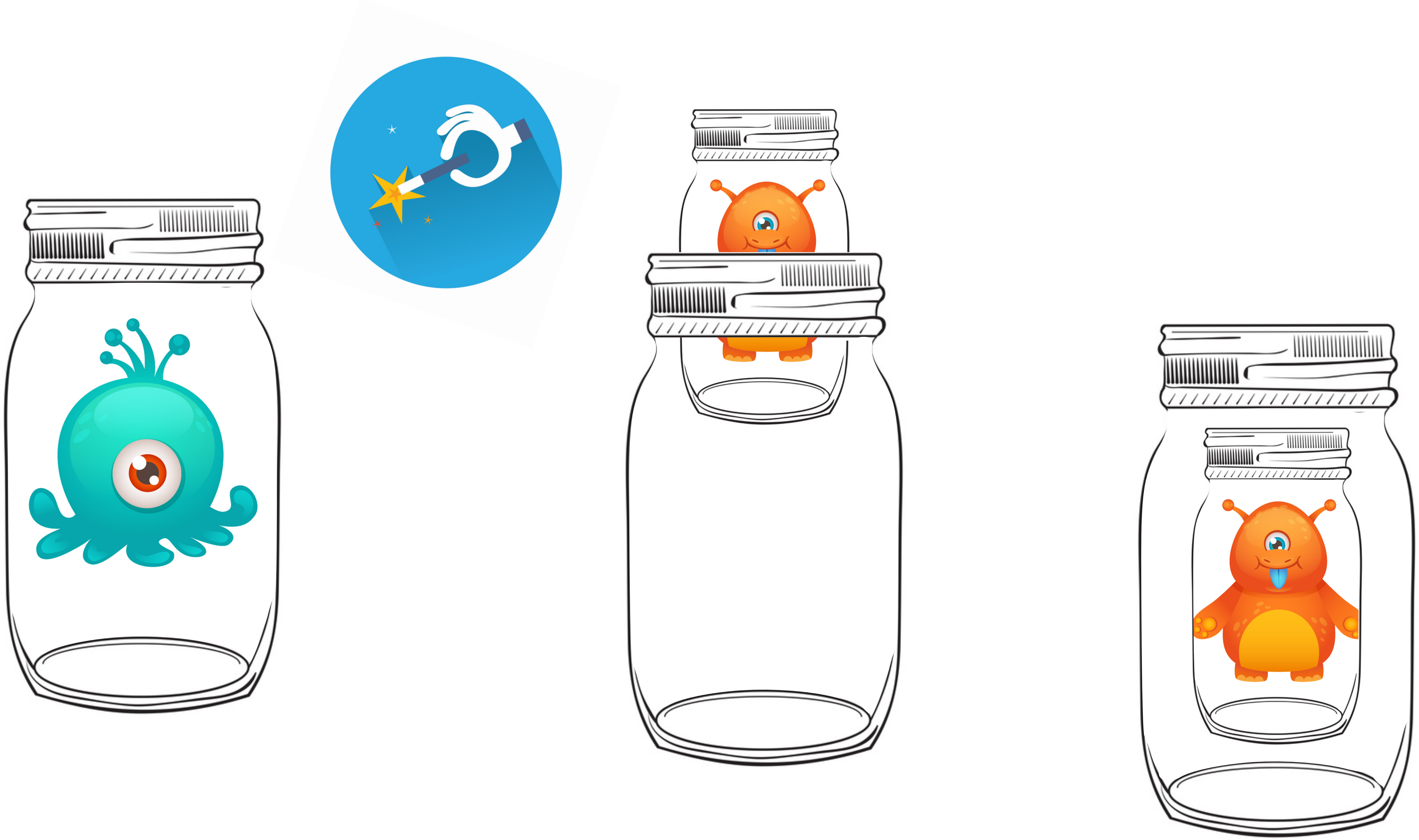
```
let panda = Functor(): of("🐼")

let addLapinouBuddy = |me| -> me + "🐰"

let buddies = panda: map(addLapinouBuddy)

println(buddies: value())

let addCatBuddy = |me| -> me + "🐱"

println(
  buddies
    : map(addCatBuddy)
    : map(addLapinouBuddy)
    : value()
)
```

“un peu” plus sérieusement

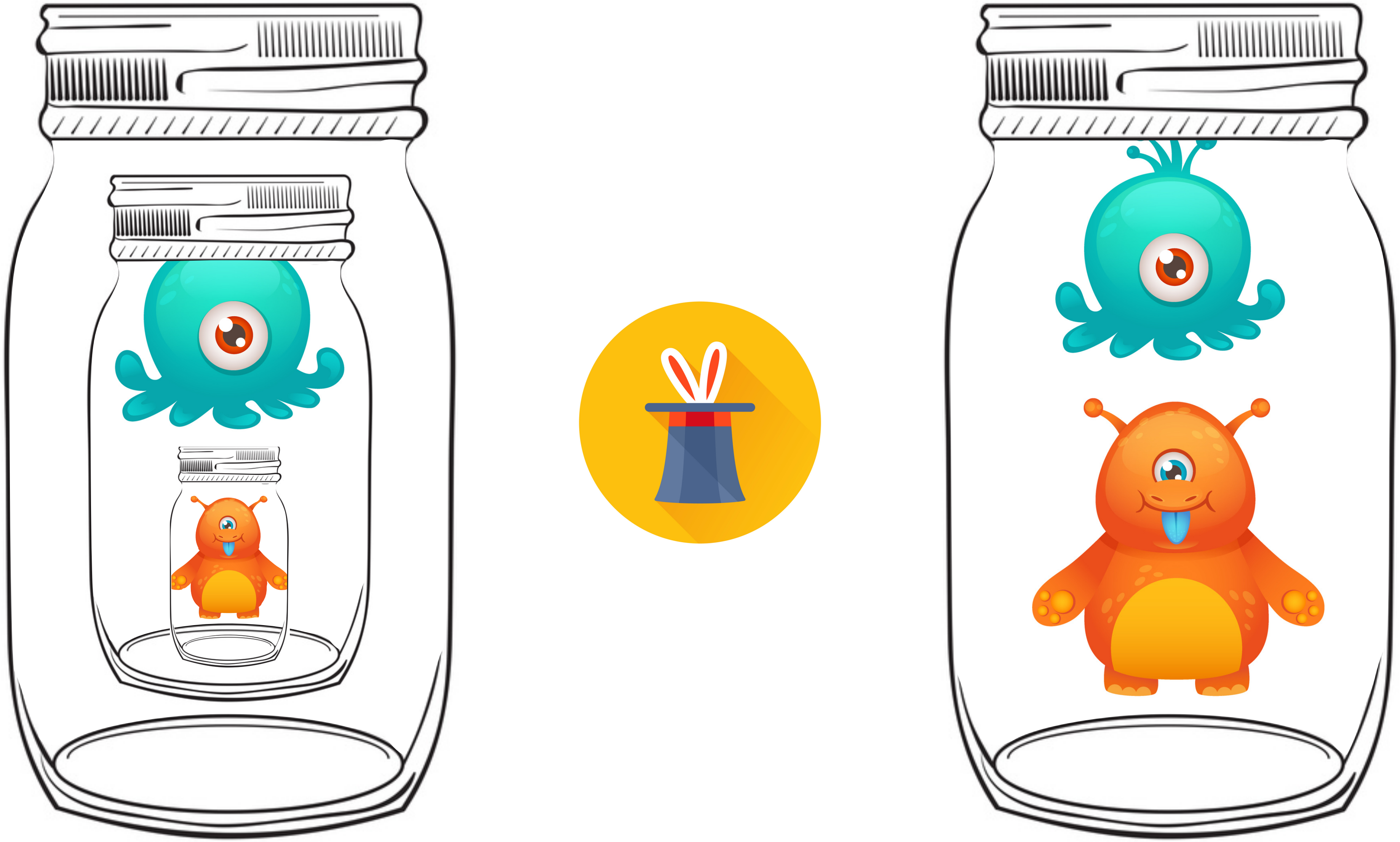
```
let addOne = |value| -> value + 1
let multiplyBy5 = |value| -> value * 5
let divideByThree = |value| -> value / 3

let a = Functor(): of(23.2)

let b = a
   : map(addOne)
   : map(addOne)
   : map(multiplyBy5)
   : map(divideByThree)

println(b: value())
```

Monad



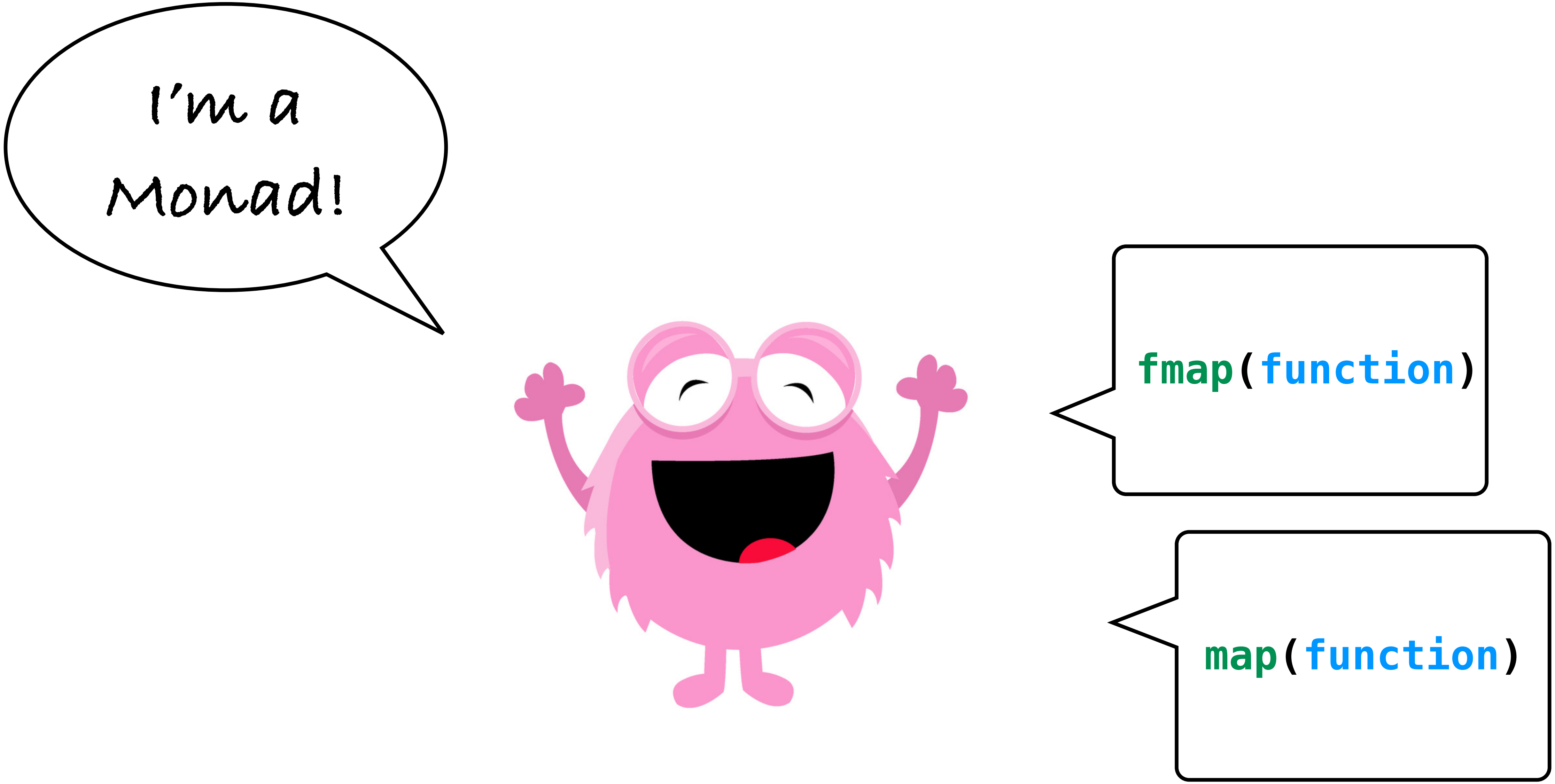


```
let panda = Functor(): of("🐼")
let addTigrouBuddy = |me| -> Functor(): of(me + "🐯")

let buddies = panda: map(addTigrouBuddy)

println(buddies: value()) # struct Functor{}
println(buddies: value(): value()) # 🐼🐯
```



```
struct Monad = {
  _value # private variable
}
augment Monad {
  # kind of constructor
  function of = |this, value| -> Monad(value)
  function value = |this| -> this: _value()
  function map = |this, fn| -> Monad(): of(fn(this: _value()))

  # Now, I'm a Monad!
  function fmap = |this, fn| -> fn(this: _value()) # // flatMap, bind
}
```

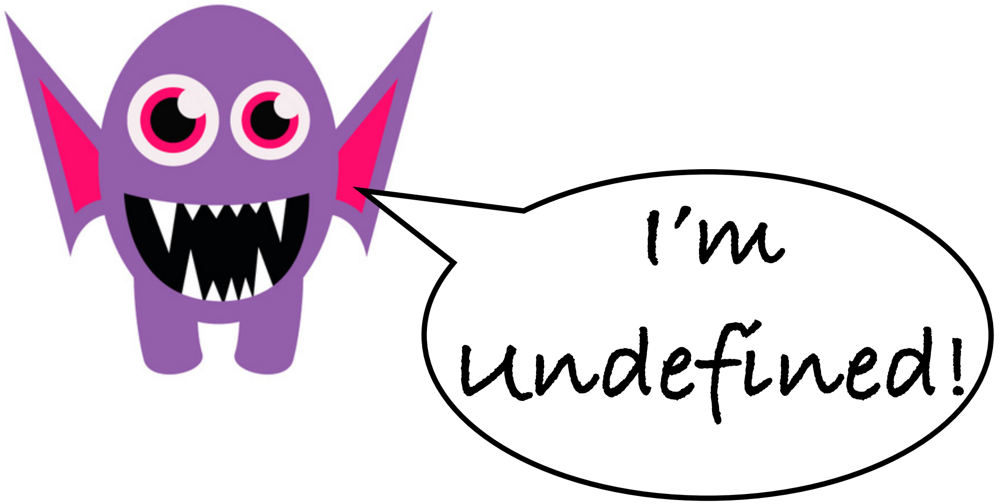
```
let panda = Monad(): of("🐼")
let addTigrouBuddy = |me| -> Monad(): of(me + "🐯")


let buddies = panda: fmap(addTigrouBuddy)

println(buddies: value()) # 🐼🐯

println(
  panda
    : fmap(addTigrouBuddy)
    : fmap(addTigrouBuddy)
    : fmap(addTigrouBuddy)
    : fmap(addTigrouBuddy)
    : value()  # 🐼🐯🐯🐯🐯
)
```



I'm a Monad!
fmap(function)
map(function)

Maybe

Pain points

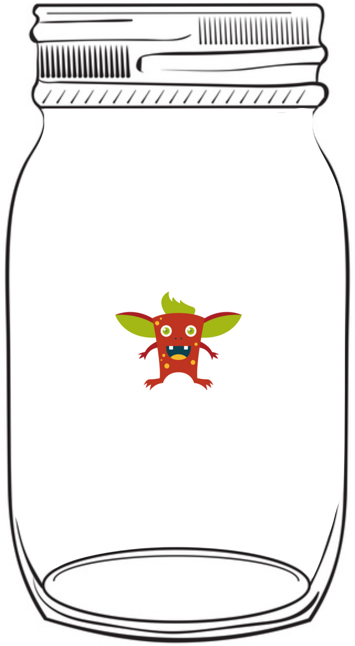


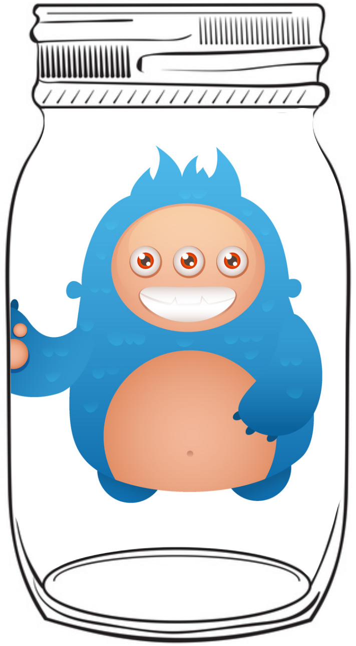
```
let bob =
  DynamicObject()
    : id("bob")
    : address(
        DynamicObject()
          : email("bob@github.com")
          : country("US")
      )

let john =
  DynamicObject()
    : id("john")
    : address(
        DynamicObject()
          : country("US")
      )

let jane =
  DynamicObject()
    : id("jane")

let buddies = [bob, jane, john]
```

```
let getMailById = |id, buddies| {
  let email =
    buddies: find(|buddy| -> buddy: id(): equals(id))
      : address()
      : email()

  if email is null {
    return "no email"
  }
  return email
}

println( getMailById("bob", buddies) )
println( getMailById("john", buddies) )
println( getMailById("jane", buddies) )
```

Maybe

None **Some**





```
struct Maybe = {
  _err # private variable
}
augment Maybe {
  function some = |this, value| -> Some(value)
  function none = |this, err| -> None()
  function fromNullable = |this, value| {
    if value is null {
      return None()
    } else {
      return Some(value)
    }
  }
}
```

```
struct None = {
  _value # private variable
}
augment None {
  function value = |this| -> null
  function map = |this, fn| -> this
  function map = |this| -> this
  function fmap = |this, fn| -> null
  function fmap = |this| -> null
  function getOrElse = |this, value| -> value
  function isNone = |this| -> true
  function isSome = |this| -> false
}
```

```
struct Some = {
  _value # private variable
}
augment Some {
  function of = |this, value| -> Some(value)
  function value = |this| -> this: _value()
  function map = |this, fn| {
    let res = fn(this: _value())
    if res is null {
      return None()
    }
    return Some(res)
  }
  function fmap = |this, fn| -> fn(this: _value())
  function getOrElse = |this, value| -> this: _value()
  function isNone = |this| -> false
  function isSome = |this| -> true
}
```

“functional” refactoring

```
let bob =
  DynamicObject()
    : id("bob")
    : address(
        DynamicObject()
          : email("bob@github.com")
          : country("US")
      )

let john =
  DynamicObject()
    : id("john")
    : address(
        DynamicObject()
          : country("US")
      )

let jane =
  DynamicObject()
    : id("jane")

let buddies = [bob, jane, john]
```

```
let getMailById = |id, buddies| ->
  Maybe()
    : fromNullable(
        buddies: find(
         |buddy| -> buddy: id(): equals(id)
        )
      )
    : map(|buddy| -> buddy: address())
    : map(|address| -> address: email())
    : getOrElse("no email!")
```

Maybe

`null`, “expliqué” par le type: `None()`

pas d’information sur l’erreur

I'm a Maybe!

J'ai 2 sous types

None
Some

fmap(function)

map(function)

Either

Either

Left **Right**

```
struct Either = {
  _value # private variable
}
augment Either {
  function right = |this, value| -> Right(value)
  function left = |this, err| -> Left(err)
}
```

```
struct Left = {
  _err # private variable
}
augment Left {
  function value = |this| -> this: _err()
  function map = |this, fn| -> this
  function map = |this| -> this
  function fmap = |this, fn| -> this
  function fmap = |this| -> this
  function getOrElse = |this, value| -> value

  function cata = |this, leftFn, rightFn| -> leftFn(this: _err())
}
```

```
struct Right = {
  _value # private variable
}
augment Right {
  function value = |this| -> this: _value()
  function map = |this, fn| {
    let res = fn(this: _value())
    if res is null {
      return Left(res)
    }
    return Right(res)
  }
  function getOrElse = |this, value| -> this: _value()

  function cata = |this, leftFn, rightFn| -> rightFn(this: _value())
}
```

utilisation

```
function getMonthName = |mo| {
  let months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul",
      "Aug", "Sep", "Oct", "Nov", "Dec"]
  try {
    return months: get(mo - 1)
  } catch(err) {
    raise("Invalid Month Number")
  }
}

function getDayName = |da| {
  let days = ["Mo", "Tu", "We", "Th", "Fr", "Sa", "Su"]
  try {
    return days: get(da - 1)
  } catch(err) {
    raise("Invalid Day Number")
  }
}
```

```
function main = |args| {
  println( getMonthName(1) )
  println( getMonthName(42) )
  println( getDayName(5) )
}
```

on “cache” les exceptions

```
let giveMeMonthName = |monthNumber| {
  try {
    return Either(): right(getMonthName(monthNumber))
  } catch (e) {
    return Either(): left(e: message())
  }
}

let giveMeDayName = |dayNumber| {
  try {
    return Either(): right(getDayName(dayNumber))
  } catch (e) {
    return Either(): left(e: message())
  }
}

println( giveMeMonthName(1): getOrElse("oups") )
println( giveMeMonthName(15): getOrElse("oups") )
println( giveMeDayName(42): getOrElse("oups") )
```

catamorphisme

```
# les println c'est mal
giveMeMonthName(12): cata(
  leftFn = |err| -> println(err),
  rightFn = |value| -> println(value)
)

giveMeMonthName(42): cata(
  leftFn = |err| -> println(err),
  rightFn = |value| -> println(value)
)
```

mais...

```
let checkMonthAndDay = |monthNumber, dayNumber| {
  try {
    let month = getMonthName(monthNumber)
    let day = getDayName(dayNumber)
    return Either(): right([month, day])
  } catch (e) {
    return Either(): left(e: message())
  }
}

# les println c'est mal
checkMonthAndDay(42, 42): cata(
  leftFn = |err| -> println(err),
  rightFn = |value| -> println(value)
)
```

On ne trace pas toutes les erreurs

I'm an Either!

J'ai 2 sous types

Left
Right

```
fmap(function)
```

```
map(function)
```

```
cata(
  left function,
  right function
)
```

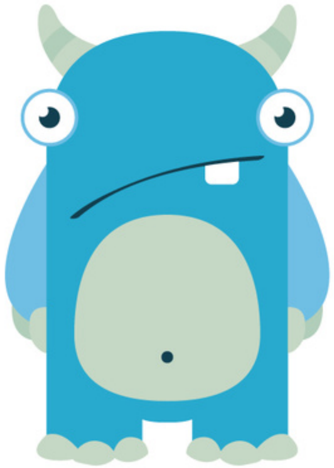
Validation

Applicative Functor ???

c'est un  avec
une méthode `ap()`

Méthode ap( )

“ap is a function that can apply the function contents of one functor to the value contents of another”



Validation

```
struct Validation = {
  _value # private variable
}
augment Validation {
  function success = |this, value| -> Success(value)
  function fail = |this, err| -> Fail(err)
}
```

```
struct Success = {
  _value # private variable
}
augment Success {
  function value = |this| -> this: _value()
  function map = |this, fn| -> Success(fn(this: _value()))
  function isSuccess = |this| -> true
  function isFail = |this| -> false

  function ap = |this, otherContainer| {
    if otherContainer: isFail() {
      return otherContainer
    } else {
      return otherContainer: map(this: _value())
    }
  }

  function cata = |this, failureFn, successFn| -> successFn(this: _value())
}
```

```
struct Fail = {
  _value # private variable
}
augment Fail {
  function value = |this| -> this: _value()
  function map = |this| -> this
  function map = |this, fn| -> this
  function isSuccess = |this| -> false
  function isFail = |this| -> true

  function ap = |this, otherContainer| {
    if otherContainer: isFail() {
      let clone = this: _value(): clone()
      clone: addAll(otherContainer: value(): clone())
      return Fail(clone)
    } else { return this }
  }

  function cata = |this, failureFn, successFn| -> failureFn(this: _value())
}
```

exemple(s)

```
let checkMonth = |monthNumber| {
  try {
    return Validation(): success(getMonthName(monthNumber))
  } catch (e) {
    return Validation(): fail(vector[e: message()])
  }
}

let checkDay = |dayNumber| {
  try {
    return Validation(): success(getDayName(dayNumber))
  } catch (e) {
    return Validation(): fail(vector[e: message()])
  }
}
```

```
Validation(): success(
  |day| ->
    |month| ->
      "Day:" + day + " Month:" + month
)
  : ap(checkDay(42))
  : ap(checkMonth(42))
  : cata(
    failureFn = |err| -> println(err),
    successFn = |value| -> println(value)
  )
```

I'm a validation!

J'ai 2 sous types

Fail
Success

```
ap(validation)
```

```
map(function)
```

```
cata(
  failureFn function,
  successFn function
)
```

En fait Golo propose déjà des “goodies” fonctionnels

```
import gololang.Errors
```

Some, None, either

```
let toInt = |value| {
  try {
    return Some(java.lang.Integer.parseInt(value))
  } catch(e) {
    return None()
  }
}

let result = toInt("42"): either(|value| {
  println("Succeed!")
  return value
}, {
  println("Failed!")
  return 42
})
```

Some, None, either, orElse

```
let result2 = toInt("Quarente-deux"): map(|value| {
  println("Succeed!")
  return value
}): orElse(42) #  valeur par défaut
```

Some, None, either, orElse, orElseGet

```
let result3 = toInt("Quarante-deux"): map(|value| {
  println("Succeed!")
  return value
}): orElseGet({
  println("Failed!")
  return 42
})
```

Result & Error

```
let toInt = |value| {
  try {
    return Result(java.lang.Integer.parseInt(value))
  } catch(e) {
    return Error(e: getMessage())
  }
}
```

Encore mieux

Avec les decorators

```
@option
function toInt = |value| -> java.lang.Integer.parseInt(value)

@result
function toIntBis = |value| -> java.lang.Integer.parseInt(value)
```

Une petite dernière pour la route

null-safe methods invocation with Elvis

```
buddies: get(2)?: address()?: email() orIfNull "no email"
buddies: get(0)?: address()?: email() orIfNull "no email"
buddies: get(1)?: address()?: email() orIfNull "no email"
```

Ressources

Lectures

Mostly Adequate Guide (Brian Lonsdorf): https://www.gitbook.com/book/drboolean/mostly-adequate-guide/details

Functional Programming in Java (Pierre-Yves Saumont): https://www.manning.com/books/functional-programming-in-java

Functional Programming in JavaScript (Luis Atencio): https://www.manning.com/books/functional-programming-in-javascript

Functional Programming in Scala (Paul Chiusano and Rúnar Bjarnason): https://www.manning.com/books/functional-programming-in-scala

Le code source de Golo:

https://github.com/eclipse/golo-lang/blob/master/src/main/golo/errors.golo

https://github.com/eclipse/golo-lang/blob/master/src/main/java/gololang/error/Result.java

Talks

Gérer les erreurs avec l'aide du système de types de Scala !
(David Sferruzza):
https://www.youtube.com/watch?v=TwJQKrZ23Vs

TDD, comme dans Type-Directed Development
(Clément Delafargue):
https://www.youtube.com/watch?v=XhcgCF0xXRs

Remerciements

Merci 

> Loïc Descotte @loic_d
> Julien Ponge @jponge
> Thierry Chantier @titimoby

> @web2day

Questions

https://github.com/k33g/web.2.day.golo/issues