

# Gestion d'erreur fonctionnelle

---

AVEC SCALA ET CATS

# Objectifs

---

Se simplifier l'existence avec la programmation fonctionnelle

En se basant sur des notions théoriques efficaces

Mais sans faire de la théorie

Pour le développement d'applications

En partant d'un niveau basique

Avec un cas réaliste et plutôt complexe à la fin

# Scala en 3 slides (1)

---

```
abstract class Option[A]
case class Some[A](value: A) extends Option[A]
case class None[A]() extends Option[A]

// data Option a = Some a | None

def maybeAnInt: Option[Int] = ???
val something: Option[String] = Some("thing")

something match {
  case Some(x) => println(x)
  case None() => println("nothing")
}
```

# Scala en 3 slides (2)

---

```
def f(x: Int): String = ???
```

```
Some[Int](5).map(f) // => Some(f(5))
```

```
None.map(f)         // => None
```

```
def g(x: Int): Option[String] = ???
```

```
Some(3).map(g) : Option[Option[String]]
```

```
Some(4).flatMap(g) // => g(4)
```

```
None.flatMap(g)    // => None
```

# Scala en 3 slides (3)

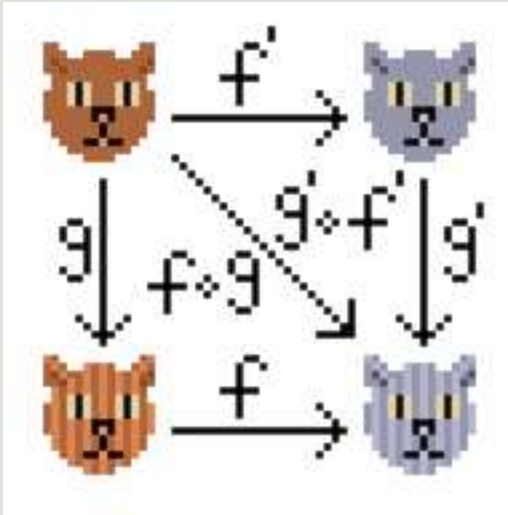
---

```
def f: Option[Int]           = ???
def g(x: Int): Option[String] = ???
def h(s: String): Double     = ???

val forComprehension: Option[Double] =
  for {
    x <- f
    y <- g(x)
  } yield h(y)

val desugared: Option[Double] =
  f.flatMap(x => g(x).map(y => h(y)))
```

# cats



Projet Typelevel @ <https://github.com/typelevel/cats>

Abstractions « mathématiques »

Descend de Haskell

Ça fait un peu peur

On va le regarder comme un outil

```
import cats.data._
import cats.instances.all._
import cats.syntax.all._
```

# Les erreurs dans une application

---

« Fail fast »

...Mais pas toujours 

Composition de plein de formats

- Exceptions
- null !
- Option
- Either
- Future

# Option

---

```
case class User(name: String, address: Option[String])

def getUserById(id: Long): Option[User] = ???

def getUser1: Option[User] = Some(User("Thomas", Some("Lille")))

for {
  user    <- getUser1
  address <- user.address
} yield address

// Some("Lille")
```



# Option (2)

---

```
case class User(name: String, address: Option[String])

def getUserById(id: Long): Option[User] = ???

def getUser2: Option[User] = Some(User("Other", None))

for {
  user    <- getUser2
  address <- user.address
} yield address

// None
```



# Either

---

```
val anError: Either[String, Int] = Left("OMG error!")
val aSuccess: Either[String, Int] = Right(1)

Some(12).toRight("Error message") // Right(12)
None.toRight("Error message")     // Left("error message")

Either.catchNonFatal(throw new Exception("!")) : Either[Throwable, Int]
```

# Either

---

```
val anError: Either[String, Int] = Left("OMG error!")
val aSuccess: Either[String, Int] = Right(1)

def f(x: Int): String = ???
aSuccess.map(f) // Right(f(1))           : Either[String, String]
anError.map(f)  // Left("OMG error!")     : Either[String, String]

def g(x: Int): Either[String, String] = ???
aSuccess.flatMap(g) // g(1)
anError.flatMap(g)  // Left("OMG error!")
```

# Either (fail-fast)

---

```
case class User(name: String, address: Option[String])

def getUser1: Option[User] = Some(User("Thomas", Some("Lille")))

for {
  user    <- getUser1.toRight("User 1 not found")
  address <- user.address.toRight("User 1 has no address")
} yield address

// Right("Lille")
```

# Either (fail-fast)

---

```
case class User(name: String, address: Option[String])

def getUser2: Option[User] = Some(User("Other", None))

for {
  user    <- getUser2.toRight("User 2 not found")
  address <- user.address.toRight("User 2 has no address")
} yield address

// Left("User 2 has no address")
```



# Either (formulaire)

```
case class Form(name: String, phone: String, email: String)

def validateName(name: String): Either[String, Unit] = ???
def validatePhone(phone: String): Either[String, Unit] = ???
def validateEmail(email: String): Either[String, Unit] = ???

val form = Form("bad name", "bad phone", "bad email")

for {
  _ <- validateName(form.name)
  _ <- validatePhone(form.phone)
  _ <- validateEmail(form.email)
} yield form

// Left("bad name")
```



# Validated

---

```
Validated.Valid("Some value")
Validated.Invalid("Some error")

"Some value".valid
"Some error".invalid

def f(x: Int): String = ???

13.valid.map(f)      // Valid(f(13))
"31".invalid.map(f) // Invalid("31")

// no flatMap

val anEither: Either[String, Int]      = Right(24)
val toValidated: Validated[String, Int] = anEither.toValidated
val backToEither: Either[String, Int]  = toValidated.toEither
```

# Validated (formulaire)

---

```
case class Form(name: String, phone: String, email: String)

def validateName(name: String): Either[String, Unit] = ???
def validatePhone(phone: String): Either[String, Unit] = ???
def validateEmail(email: String): Either[String, Unit] = ???

val form = Form("bad name", "bad phone", "bad email")

(validateName(form.name).toValidated |@|
 validateEmail(form.name).toValidated |@|
 validatePhone(form.phone).toValidated)
.map((_, _, _) => form)
```



Idl



# |@| - L'opérateur « cri »

```
def f : (Int, String) => Double = ???

(1.valid[String] |@| "foo".valid).map(f)      // Valid(f(1, "foo"))
("nope".invalid |@| "bar".valid).map(f)       // Invalid("nope")
(1.valid[String] |@| "nope".invalid).map(f)    // Invalid("nope")
("nope".invalid |@| "nope".invalid).map(f)     // Invalid("nopenope")

import cats.Cartesian._
map2(1.valid[String], "foo".valid)(f)
```

# Validated (formulaire)

---

```
case class Form(name: String, phone: String, email: String)

def validateName(name: String): Either[String, Unit] = ???
def validatePhone(phone: String): Either[String, Unit] = ???
def validateEmail(email: String): Either[String, Unit] = ???

val form = Form("bad name", "bad phone", "bad email")

(validateName(form.name).toValidated |@|
 validateEmail(form.name).toValidated |@|
 validatePhone(form.phone).toValidated)
  .map((_, _, _) => form)

// Invalid("bad namebad phonebad email")
```

# Validated (formulaire bis)

```
val nonEmptyList = NonEmptyList("alice", List("bob"))

def anEither: Either[String, Int] = ???
anEither.toValidatedNel: Validated[NonEmptyList[String], Int]

val form = Form("bad name", "bad phone", "bad email")

(validateName(form.name).toValidatedNel |@|
 validateEmail(form.name).toValidatedNel |@|
 validatePhone(form.phone).toValidatedNel)
  .map((_, _, _) => form)

// Invalid(NonEmptyList("bad name", "bad phone", "bad email"))
```



# Validated (formulaire ter)

---

```
val nonEmptyList = NonEmptyList("alice", List("bob"))

def anEither: Either[String, Int] = ???
anEither.toValidatedNel: Validated[NonEmptyList[String], Int]

val form = Form("bad name", "bad phone", "bad email")

validateName(form.name).toValidatedNel *>
  validateEmail(form.name).toValidatedNel *>
  validatePhone(form.phone).toValidatedNel *>
  form.validNel

// Invalid(NonEmptyList("bad name", "bad phone", "bad email"))
```

# En pratique

---

Résultat : `Either[Error, X]`

- `Error` = `String` ou un type plus adapté

Enchaînement de

- Étapes pouvant échouer (`X => Either[Error, Y]`)
- Étapes « normales » (`X => Y`)

`flatMap`

`map`

Mais aussi

- Transformation de l'erreur (`Error => String`)
- Récupération d'erreur (`Error => X`)
- ... Pouvant à son tour échouer (`Error => Either[Error, X]`)
- Appels asynchrones

`leftMap`

`handleError`

`handleErrorWith`

...

# Future (intro)

---

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

def f : Future[Int] = ??? // un Int, un jour, peut-être

f.map(x => x + 1) : Future[Int]

def g : Int => Future[String] = ???
f.flatMap(g) : Future[String]

f.onComplete(???)
import scala.concurrent.Await
import scala.concurrent.duration.Duration
Await.result(f, Duration.Inf)
```



# Future et Either

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

def callHttpApi: Future[String] = ???
def uncertainComputation(s: String): Either[String, Int] = ???
def callOtherApi(x: Int): Future[Int] = ???
def composeUserMessage(x: Int): String = ???
```

```
callHttpApi
  .map(uncertainComputation)
  .flatMap {
    case Left(error) => Future.successful(Left(error))
    case Right(value) => callOtherApi(value).map(Right(_))
  }
  .map(x => x.map(composeUserMessage))
```



# EitherT

---

```
type Result[X] = Future[Either[String, X]]

// EitherT[Future, String, X]

val fromResult = EitherT(??? : Result[Int])
fromResult.value : Result[Int]

val successNow = EitherT.pure[Future, String, Int](4)
val eitherNow   = EitherT.fromEither[Future](Either.left("error"))

val successLater = EitherT.liftT[Future, String, Int](Future(5))
val errorAsLeft  = Future.failed(new Exception("!")).attemptT.leftMap(_.toString)
// Left("java.lang.Exception: !") – succès immédiat
```

# EitherT

```
def callHttpApi: Future[String] = ???
def uncertainComputation(s: String): Either[String, Int] = ???
def callOtherApi(x: Int): Future[Int] = ???
def composeUserMessage(x: Int): String = ???

for {
  step1 <- callHttpApi.attemptT.leftMap(_.toString)
  step2 <- EitherT.fromEither[Future](uncertainComputation(step1))
  step3 <- callOtherApi(step2).attemptT.leftMap(_.toString)
  step4 = composeUserMessage(step3)
} yield step4
```



# Combinateurs

---

`EitherT[Future, MyError, A]`

Récupération d'erreurs

- `attemptT : EitherT[Future, MyError, A]`
- `handleError(f : MyError => A)`
- `handleErrorWith(f : MyError => EitherT[Future, MyError, A])`
- `leftMap(f : MyError => String)`

Transformations

- `transform(Either[MyError, A] => Either[String, B]) : EitherT[Future, String, B]`

# Combinateurs

---

`EitherT[Future, MyError, A]`

## Chainage

- `flatMap(f : A => EitherT[Future, MyError, B])`
- `flatMapF(f : A => Future[Either[MyError, B]])`
- `subflatMap(f : A => Either[MyError, B])`
- `semiflatMap(f : A => Future[B])`
- `map(f: A => B)`

## Séquences :

- `sequence : List[EitherT[Future, MyError, A]] => EitherT[Future, MyError, List[A]]`
- `traverse : List[A] => (A => EitherT[Future, MyError, B]) => EitherT[Future, MyError, List[B]]`

# Récap

---

On peut mixer gestion d'erreur et asynchronie et c'est à peu près confortable

## **Recommandations**

Tout représenter comme EitherT

Toujours attemptT

Utiliser son propre type pour les erreurs

# Références

---

Scala <https://www.scala-lang.org>

Cats <https://github.com/typelevel/cats>

Sources <https://github.com/chwthewke/lillefp4>

## **Inspirations <3**

Eugene Yokota – herding cats, stacking Future and Either

<http://eed3si9n.com/herding-cats/stacking-future-and-either.html>

Brendan McAdams – A Skeptic's look into scalaz's gateway drugs

[https://www.youtube.com/watch?v=BPYz19z\\_3s8](https://www.youtube.com/watch?v=BPYz19z_3s8)

Daniela Sfregola – Easy and efficient data validation with cats

<https://www.youtube.com/watch?v=OkTfcyFohS0>