A Skeptic's Look at scalaz' Gateway Drugs

Brendan McAdams (@rit)

We are *not* using the "Zed" word.

We are ~~not~~ using the "Zed" word.
Ok, so I lied a little...

How I’ve Traditionally Seen scalaz

- In the past, I've seen scalaz as fairly intimidating
- People always spoke about it being more "pure"/"haskelly"/"mathy"
- I'll be the first to admit: I don't have a CS degree and sort of suck at math
- "What's wrong with what I have in standard Scala?!?"

The Reality about scalaz?

IT’S MAGIC

The Road to scalaz

- Once I got started, it was hard to stop
- The constructs are powerful and useful
- I am by *no means* an expert: just an excited amateur
- This is not a category theory or haskell talk: Let's be practical

The Road to scalaz

- I want you to learn:
  - "Hey! This stuff may be useful!"

- I am not going to teach you:
  - "A monad is a monoid in the category of endofunctors, what's the problem?"

The Road to scalaz

Problems to solve...

- Our API server was part of a larger Angular.js application: error passing was hard
  - Providing clear errors & validating input was a problem
  - 500s & generic exceptions complicate and frustrate frontend devs' debugging

Helping Developers Help Themselves

- An error occurred
  - API Received bad/invalid data? (e.g. JSON Failed to parse)
  - Database failed?
  - Hovercraft filled up with eels?
- What if multiple errors occurred?
- How do we communicate all of this effectively?

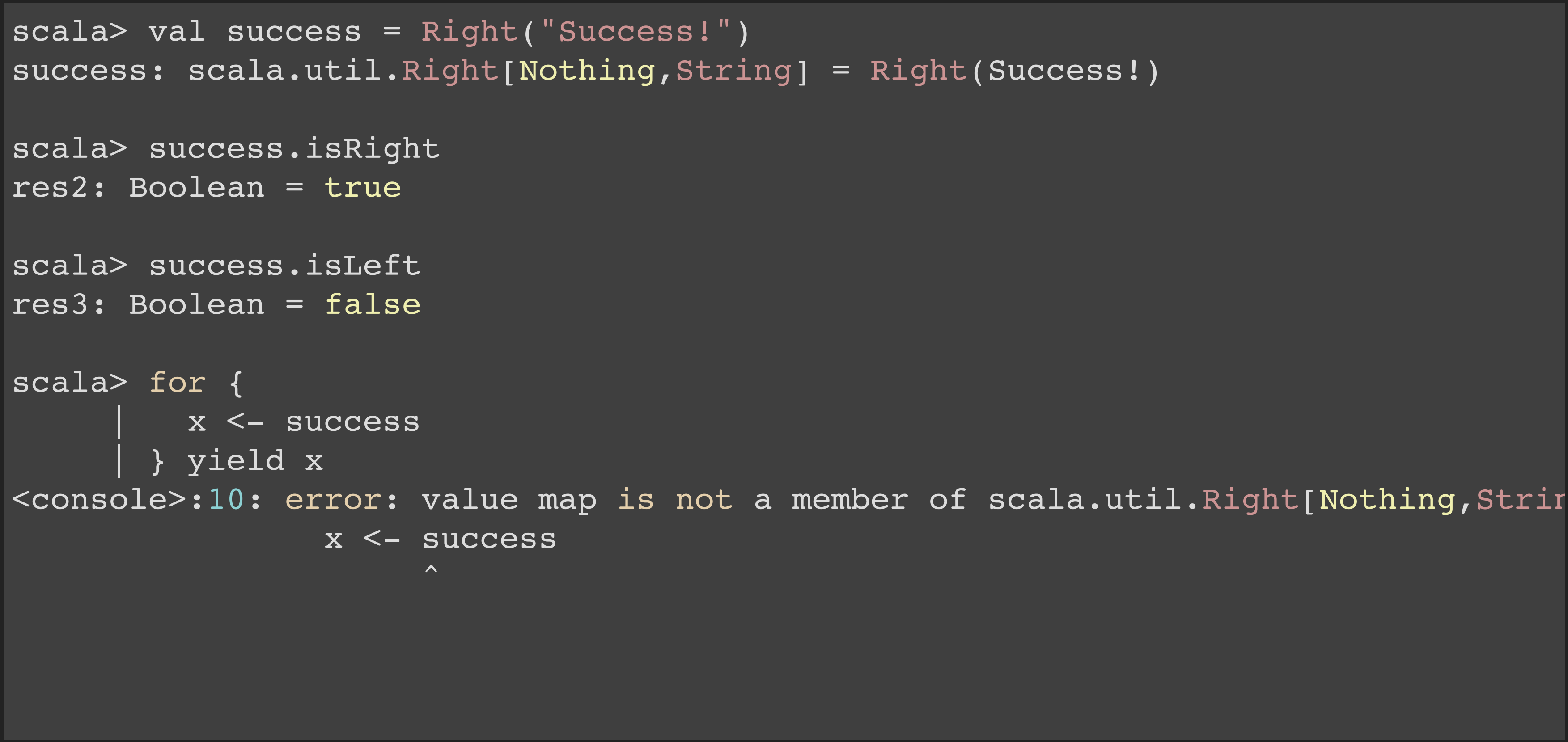
Scala’s Either: The Limitations

- Scala's builtin **Either** is a commonly used tool, allowing **Left** and **Right** projections

- By convention *Left* indicates an error, while *Right* indicates a success

- Good concept, but there are some limitations in interaction

Scala's Either: The Limitations



```
scala> val success = Right("Success!")
success: scala.util.Right[Nothing,String] = Right(Success!)

scala> success.isRight
res2: Boolean = true

scala> success.isLeft
res3: Boolean = false

scala> for {
     |   x <- success
     | } yield x
<console>:10: error: value map is not a member of scala.util.Right[Nothing,Strin
                x <- success
                     ^
```

Not a monad. Pain in the ass to extract.

Disjunctions as an Alternative

- scalaz' \/ (aka "Disjunction") is similar to "Either"
- By convention, the right is success and the left failure
  - The symbol -\/ is "left"
  - The symbol \/- is "right"

Disjunctions as an Alternative

- Disjunctions assume we prefer success (the right)
- This is also known as "Right Bias"
- for comprehensions, map, and flatMap statements unpack where "success" \/- continues, and "failure" -\/ aborts

Disjunctions as an Alternative

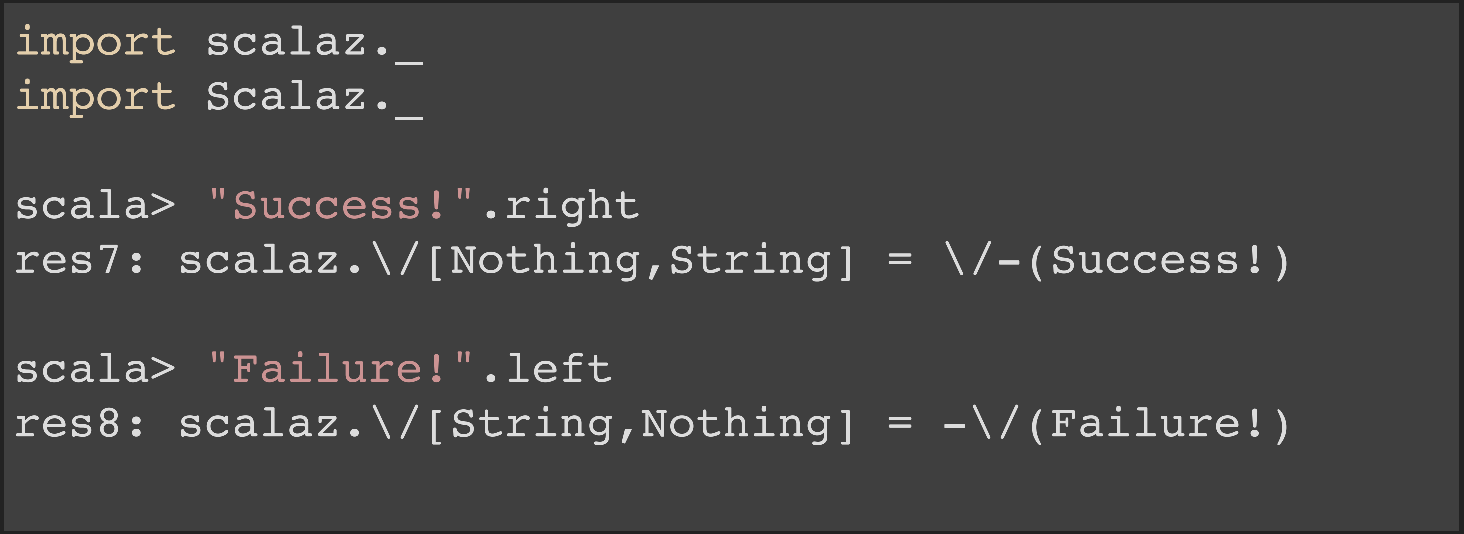
Best Practice

When declaring types, prefer infix notation, i.e.

```
def query(arg: String): Error \/ Success
```

over "standard" notation such as

```
def query(arg: String): \/[Error, Success]
```



```
import scalaz._
import Scalaz._

scala> "Success!".right
res7: scalaz.\/[Nothing,String] = \/-(Success!)

scala> "Failure!".left
res8: scalaz.\/[String,Nothing] = -\/(Failure!)
```

Postfix Operators (**.left** & **.right**) allow us to wrap an existing Scala value to a disjunction

```
import scalaz._
import Scalaz._

scala> \/.left("Failure!")
res10: scalaz.\/[String,Nothing] = -\/(Failure!)


scala> \/.right("Success!")
res12: scalaz.\/[Nothing,String] = \/-(Success!)
```

We can also invoke **.left** & **.right** methods on the Disjunction singleton for the same effect...

```
import scalaz._
import Scalaz._

scala> -\/("Failure!")
res9: scalaz.-\/[String] = -\/(Failure!)

scala> \/-("Success!")
res11: scalaz.\/-[String] = \/-(Success!)
```

... or go fully symbolic with specific constructors:

-\/ for left

\/- for right

Digression: Scala **Option**

- Scala Option is a commonly used container, having a **None** and a **Some** subtype

- Like \/ it also has a bias towards "success": **Some**

- Comprehension over it has issues with "undiagnosed aborts"

```
case class Address(city: String)

case class User(first: String,
                last: String,
                address: Option[Address])

case class DBObject(id: Long,
                    user: Option[User])

val brendan =
  Some(DBObject(1, Some(User("Brendan", "McAdams", None))))

val someOtherGuy =
  Some(DBObject(2, None))
```

```
for {
  dao <- brendan
  user <- dao.user
} yield user

/* res13: Option[User] = Some(User(Brendan,McAdams,None)) */

for {
  dao <- someOtherGuy
  user <- dao.user
} yield user

/* res14: Option[User] = None */
```

What went wrong?

\/ to the Rescue

- Comprehending over groups of Option leads to "silent failure"

- Luckily, scalaz includes implicits to help convert a **Option** to a Disjunction

- \/ right bias makes it easy to comprehend

- On a **left,** we'll get potentially useful information instead of **None**

```
None \/> "No object found"
/* res0: scalaz.\/[String,Nothing] = -\/(No object found) */

None toRightDisjunction "No object found"
/* res1: scalaz.\/[String,Nothing] = -\/(No object found) */

Some("My Hovercraft Is Full of Eels") \/> "No object found"
/* res2: scalaz.\/[String, String] = \/-(My Hovercraft Is Full of Eels) */

Some("I Will Not Buy This Record It Is Scratched")
  .toRightDisjunction("No object found")
/* res3: scalaz.\/[String, String] =
  \/-(I Will Not Buy This Record, It Is Scratched") */
```

```
for {
  dao <- brendan \/> "No user by that ID"
  user <- dao.user \/> "Join failed: no user object"
} yield user
/* res0: scalaz.\/[String,User] = \/-(User(Brendan,McAdams,None)) */

for {
  dao <- someOtherGuy \/> "No user by that ID"
  user <- dao.user \/> "Join failed: no user object"
} yield user
/* res1: scalaz.\/[String,User] = -\/(Join failed: no user object) */
```

Suddenly we have much more useful failure information.

But what if we want to do something beyond comprehensions?

Validation

- **Validation** *looks* similar to \/ at first glance
  - (And you can convert between them)
  - Subtypes are **Success** and **Failure**
- **Validation** is *not* a monad
- **Validation** is an *applicative functor*, and many can be chained together
- If any failure in the chain, failure wins: All errors get appended together

```
val brendanCA =
  DBObject(4,
    Some(User("Brendan", "McAdams",
      Some(Address("Sunnyvale"))))
  )


val cthulhu =
  DBObject(5,
    Some(User("Cthulhu", "Old One",
      Some(Address("R'lyeh"))))
  )

val noSuchPerson = DBObject(6, None)

val wanderingJoe =
  DBObject(7,
    Some(User("Wandering", "Joe", None))
  )
```

```scala
def validDBUser(dbObj: DBObject): Validation[String, User] = {
  dbObj.user match {

    case Some(user) =>
      Success(user)

    case None =>
      Failure(s"DBObject $dbObj does not contain a user object")

  }
}
```

```
validDBUser(brendanCA)
/* Success[User] */

validDBUser(cthulhu)
/* Success[User] */

validDBUser(noSuchPerson)
/* Failure("... does not contain a user object") */

validDBUser(wanderingJoe)
/* Success[User] */
```

```
def validAddress(user: Option[User]): Validation[String, Address] = {
  user match {

    case Some(User(_, _, Some(address))) if postOfficeValid(address) =>
      address.success

    case Some(User(_ , _, Some(address))) =>
      "Invalid address: Not recognized by postal service".failure

    case Some(User(_, _, None)) =>
      "User has no defined address".failure

    case None =>
      "No such user".failure

  }
}
```

```
validAddress(brendanCA.user)
/* Success(Address(Sunnyvale)) */

// let's assume R'Lyeh has no mail carrier
validAddress(cthulhu.user)
/* Failure(Invalid address: Not recognized by postal
service) */

validAddress(noSuchPerson.user)
/* Failure(No such user) */

validAddress(wanderingJoe.user)
/* Failure(User has no defined address) */
```

Sticking it all together

- scalaz has a number of *applicative* operators to combine **Validation** results

- *> and <* are two of the ones you'll run into first
  - *> takes the right hand value and discards the left
  - <* takes the left hand value and discards the right
  - Errors "win"

```
1.some *> 2.some
/* res10: Option[Int] = Some(2) */

1.some <* 2.some
/* res11: Option[Int] = Some(1) */

1.some <* None
/* res13: Option[Int] = None */

None *> 2.some
/* res14: Option[Int] = None */
```

BUT: with **Validation** it will chain together *all* errors that occur instead of short circuiting

```
validDBUser(brendanCA) *> validAddress(brendanCA.user)
/* res16: scalaz.Validation[String,Address] =
Success(Address(Sunnyvale)) */

validDBUser(cthulhu) *> validAddress(cthulhu.user)
/* res17: scalaz.Validation[String,Address] =
Failure(Invalid address: Not recognized by postal service) */

validDBUser(wanderingJoe) *> validAddress(wanderingJoe.user)
/* res19: scalaz.Validation[String,Address] =
Failure(User has no defined address) */

validDBUser(noSuchPerson) *> validAddress(noSuchPerson.user)
/* res18: scalaz.Validation[String,Address] =
  Failure(DBObject DBObject(6,None) does not contain a user objectNo such user)*/
```

Wait. WTF happened to that last one?

```
validDBUser(brendanCA) *> validAddress(brendanCA.user)
/* res16: scalaz.Validation[String,Address] =
Success(Address(Sunnyvale)) */

validDBUser(cthulhu) *> validAddress(cthulhu.user)
/* res17: scalaz.Validation[String,Address] =
Failure(Invalid address: Not recognized by postal service) */

validDBUser(wanderingJoe) *> validAddress(wanderingJoe.user)
/* res19: scalaz.Validation[String,Address] =
Failure(User has no defined address) */

validDBUser(noSuchPerson) *> validAddress(noSuchPerson.user)
/* res18: scalaz.Validation[String,Address] =
  Failure(DBObject DBObject(6,None) does not contain a user objectNo such user)*/
```

- The way *> is called on **Validation**, it appends all errors together...

- We'll need another tool if we want this to make sense

NonEmptyList

- **NonEmptyList** is a scalaz **List** that is guaranteed to have *at least one element*

- Commonly used with **Validation** to allow accrual of multiple error messages

- There's a type alias for **Validation[NonEmptyList[L], R]** of **ValidationNEL[L, R]**

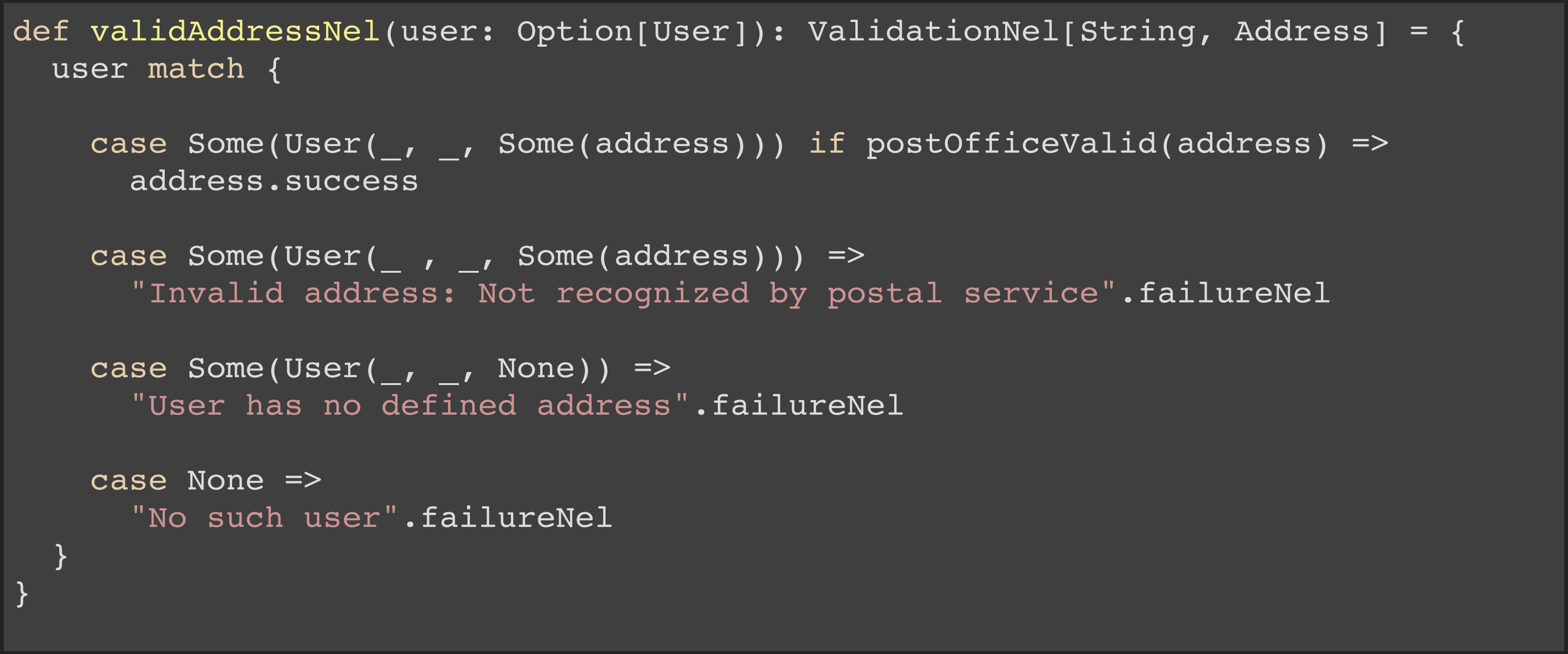
- Like a list, *append* allows elements to be added to the end

```scala
def validDBUserNel(dbObj: DBObject): Validation[NonEmptyList[String], User] = {
  dbObj.user match {

    case Some(user) =>
      Success(user)

    case None =>
      Failure(NonEmptyList(s"DBObject $dbObj does not contain a user object"))
  }
}
```

We can be explicit, and construct a **NonEmptyList** by hand



```scala
def validAddressNel(user: Option[User]): ValidationNel[String, Address] = {
  user match {

    case Some(User(_, _, Some(address))) if postOfficeValid(address) =>
      address.success

    case Some(User(_ , _, Some(address))) =>
      "Invalid address: Not recognized by postal service".failureNel

    case Some(User(_, _, None)) =>
      "User has no defined address".failureNel

    case None =>
      "No such user".failureNel
  }
}
```

Or we can use some helpers, calling **.failureNel**, and declaring a **ValidationNel** return type.

```
validDBUserNel(noSuchPerson) *> validAddressNel(noSuchPerson.user)
/* res20: scalaz.Validation[scalaz.NonEmptyList[String],Address] =
Failure(NonEmptyList(
  DBObject(6,None) does not contain a user object,
  No such user
))
*/
```

Now, we get a list of errors - instead of a globbed string

One Last Operator

- scalaz provides another useful applicative operator for us

- **|@|** combines all of the **Failure** *and* **Success** conditions

- To handle **Success**es we provide a **PartialFunction**

```
(validDBUserNel(brendanCA) |@| validAddressNel(brendanCA.user)) {
  case (user, address) =>
    s"User ${user.first} ${user.last} lives in ${address.city}"
}

// "User Brendan McAdams lives in Sunnyvale"
```

Our other users will return an **NEL** of errors, like with ***>**

```
(validDBUserNel(noSuchPerson) |@| validAddressNel(noSuchPerson.user)) {
  case (user, address) =>
    s"User ${user.first} ${user.last} lives in ${address.city}"
}

// Failure(
//   NonEmptyList(DBObject DBObject(6,None) does not contain a user object,
//                No such user))
```

noSuchPerson gets a combined list

One last *function:* Error Handling

- Dealing sanely with errors is always a challenge
- There are a few ways in the Scala world to avoid try/catch, such as **scala.util.Try**
- scalaz' \/ offers the Higher Order Function **fromTryCatchThrowable**, which catches any specific exception, and returns a Disjunction
- You specify your return type, the type of exception to catch, and your function body...

```
"foo".toInt

/* java.lang.NumberFormatException: For input string: "foo"
           at java.lang.NumberFormatException.forInputString ...
           at java.lang.Integer.parseInt(Integer.java:492)
           at java.lang.Integer.parseInt(Integer.java:527) */
```

Here's a great function to wrap...

```
\/.fromTryCatchThrowable[Int, NumberFormatException] {
  "foo".toInt
}

/* res9: scalaz.\/[NumberFormatException,Int] =
    -\/(java.lang.NumberFormatException:
        for input string: "foo") */
```

Note the reversed order of arguments: **Right** type, *then* **Left** type

```
\/.fromTryCatchThrowable[Int, Exception] {
  "foo".toInt
}

/* res9: scalaz.\/[NumberFormatException,Int] =
    -\/(java.lang.NumberFormatException:
        for input string: "foo") */
```

We can also be "less specific" in our exception type to catch more

```
\/.fromTryCatchThrowable[Int, java.sql.SQLException] {
  "foo".toInt
}

/*
java.lang.NumberFormatException: For input string: "foo"
  at java.lang.NumberFormatException.forInputString(NumberFormatException.j
  at java.lang.Integer.parseInt(Integer.java:580)
  at java.lang.Integer.parseInt(Integer.java:615)
...
*/
```

Our exception type *matters*: if an Exception doesn't match it will still be thrown

```
\/.fromTryCatchNonFatal[Int] {
  "foo".toInt
}
/* res14: scalaz.\/[Throwable,Int] =
    -\/(java.lang.NumberFormatException:
        For input string: "foo") */
```

There is also **\/.tryCatchNonFatal** which will
catch *anything* classified as **scala.util.control.NonFatal**

Final Thought: On Naming

- From the skeptical side, the common use of symbols gets... interesting
- Agreeing on names - at least within your own team - is important
- Although it is defined in the file "Either.scala", calling \/ "Either" gets confusing vs. Scala's builtin **Either**
- Here's a few of the names I've heard used in the community for **|@|** (There's also a unicode alias of ⊛)

Oink

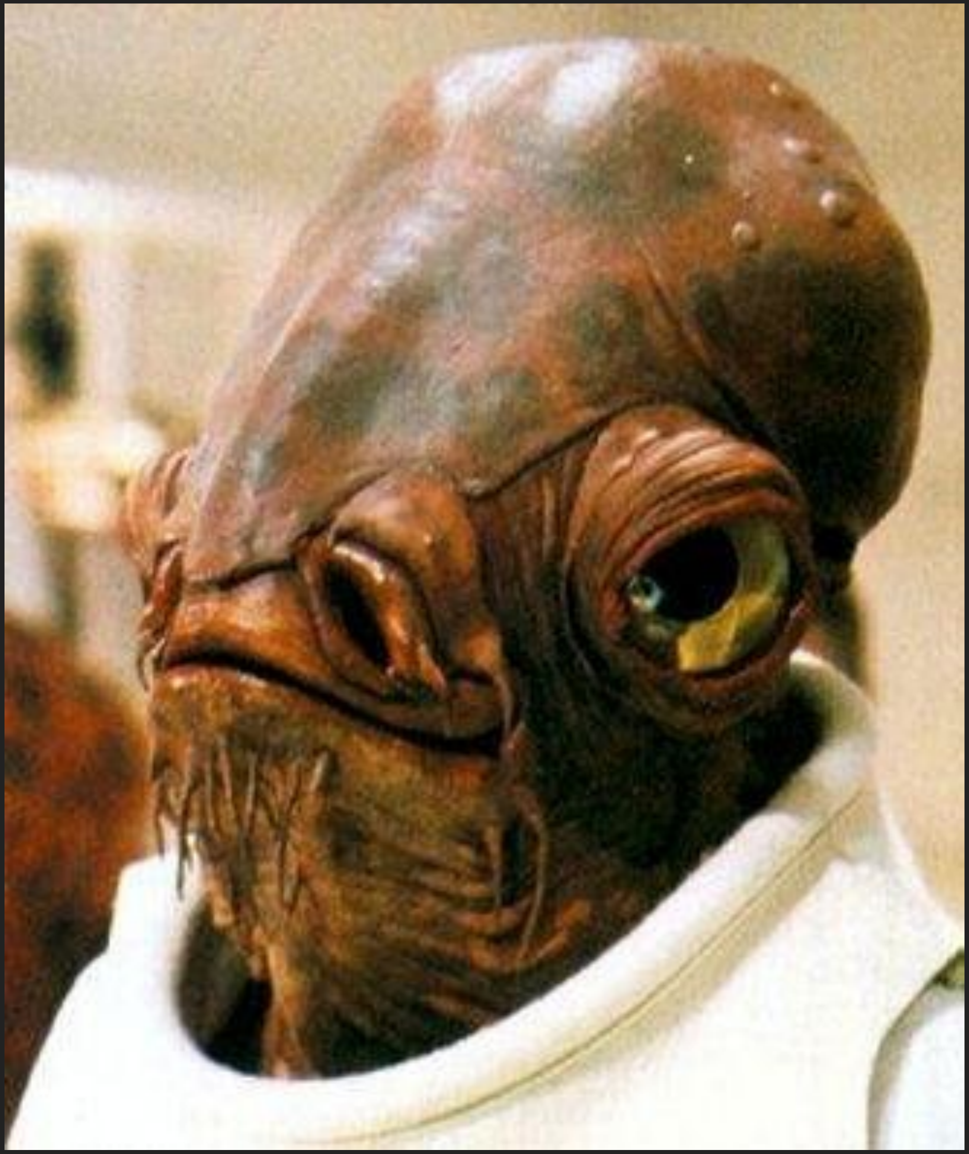
Cinnabon/Cinnamon Bun

Chelsea Bun / Pain aux Raisins

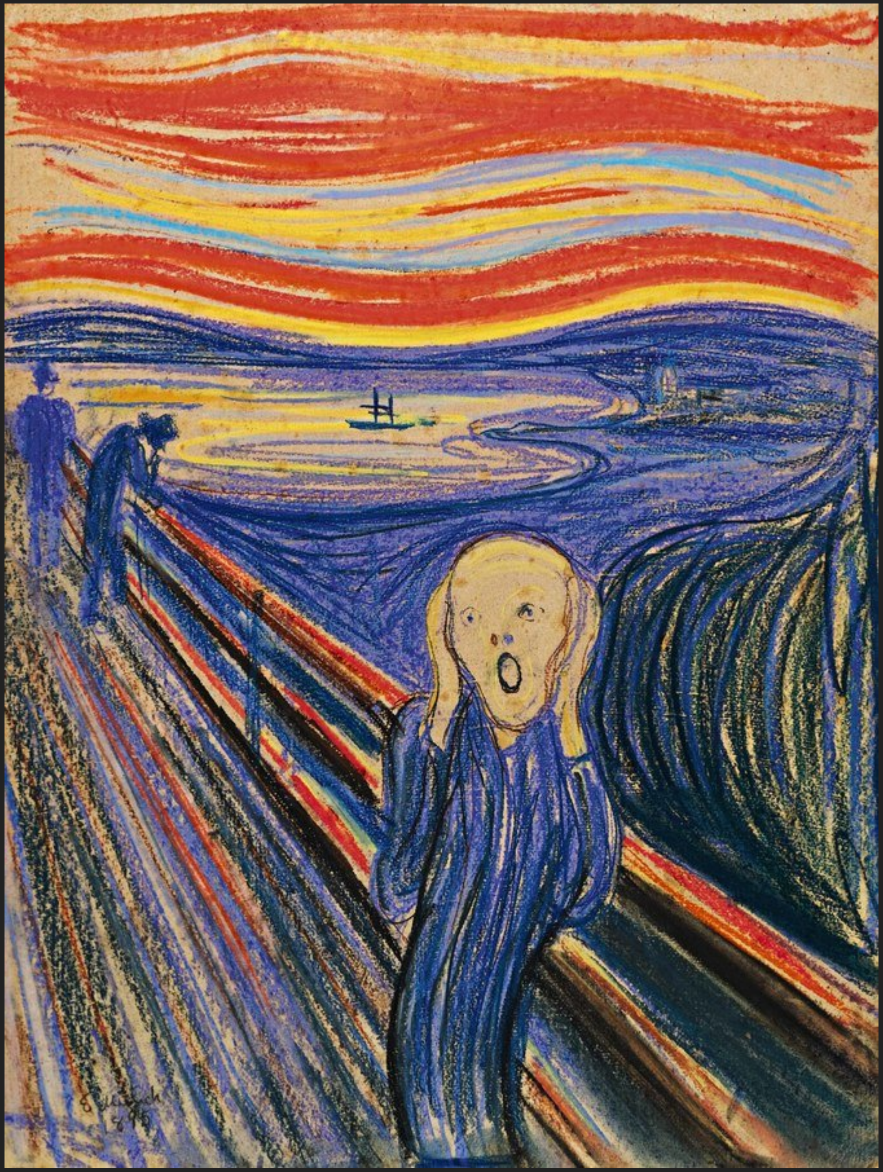
Tie Fighter

Princess Leia

Admiral Ackbar



Scream



Scream 2?

Home Alone

Pinkie Pie

WOULD YOU LIKE TO KNOW MORE?

Some Resources...

- Eugene Yokota's free website, "Learning Scalaz"
  - http://eed3si9n.com/learning-scalaz/

- Learn some Haskell! I really like "Learn You A Haskell For Great Good" by Miran Lipovača
  - http://learnyouahaskell.com

Questions?