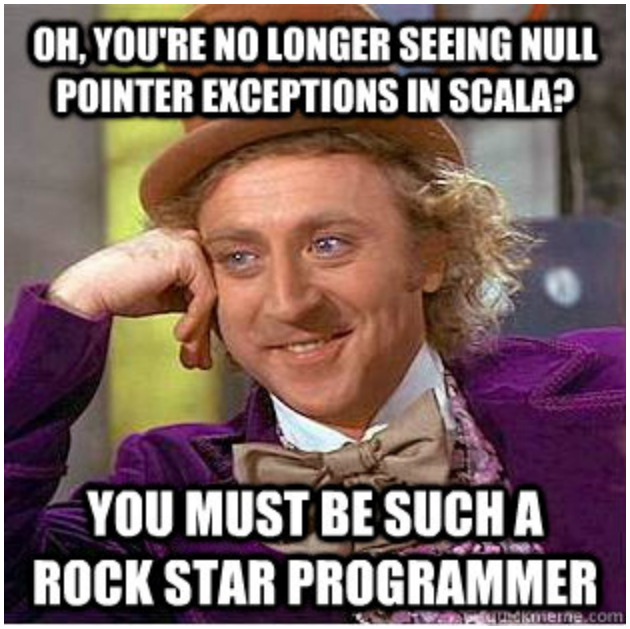
scalaz "For the Rest of Us" Adam Rosien

arosien@box.com && adam@rosien.net

```
@arosien #scalasv #scalaz
```


box


OH, YOU'RE NO LONGER SEEING NULL POINTER EXCEPTIONS IN SCALA?
YOU MUST BE SUCH A ROCK STAR PROGRAMMER


A? YOU CAN'T HANDLE AN A
BUT HERE'S AN M[A]

But `scalaz` is *AWESOME*.

This talk is specifically *not* about:

- Monads

- Applicative Functors

- Category theory

- Other really cool stuff you should learn about (eventually)

This talk *is* about every-day situations where `scalaz` can:

- Reduce syntactical noise

- Add type-safety with minimal "extra work"

- Provide useful types that solve many classes of problems

Getting Started

In `build.sbt`:

```
libraryDependencies +=
```

```
2   "org.scalaz" %% "scalaz-core" % "6.0.4"
```

Then:

```
1 import scalaz._
```

```
2 import Scalaz._
```

3

```
4 // profit
```

Memoization

Memoization

The goal: cache the result of a expensive computation.

```
1 def expensive(foo: Foo): Bar = ...
```

2

```
3 val f: Foo
```

4

```
5 expensive(f) // $$$
```

```
6 expensive(f) // $$$
```

```
7 ...
```

Memoization

Typically you might use a `mutable.Map` to cache results:

```
1 val cache = collection.mutable.Map[Foo, Bar]()
```

2

```
3 cache.getOrElseUpdate(f, expensive(f)) // $$$
```

```
4 cache.getOrElseUpdate(f, expensive(f)) // 1¢
```

Memoization

You can try to make it look like a regular function, avoiding the `getOrElseUpdate()` call:

```
1 val cache: Foo => Bar =
```

```
2   collection.mutable.Map[Foo, Bar]()
```

```
3       .withDefault(expensive _)
```

4

```
5 cache(f) // $$$ (miss & NO fill)
```

```
6 cache(f) // $$$ (miss & NO fill)
```

But it doesn't actually cache.

Memoization

In `scalaz`:

```
1 def expensive(foo: Foo): Bar = ...
```

2

```
3 // Memo[Foo, Bar]
```

```
4 val memo = immutableHashMapMemo {
```

```
5   foo: Foo => expensive(foo)
```

```
6 }
```

7

```
8 val f: Foo
```

9

```
10 memo(f) // $$$ (cache miss & fill)
```

```
11 memo(f) // 1¢  (cache hit)
```

Memoization

Many memoization strategies:

```
1 immutableHashMapMemo[K, V]
```

2

```
3 mutableHashMapMemo[K, V]
```

4

```
5 // remove + gc unreferenced entries
```

6 weakHashMapMemo[K, V]

7

```
8 // fixed size, K = Int
```

```
9 arrayMemo[V](size: Int)
```

Style

Style

Remove the need for temporary variables:

```
1 val f: A => B
```

```
2 val g: B => C
```

3

```
4 // using temps:
```

```
5 val a: A = ...
```

```
6 val b = f(a)
```

```
7 val c = g(b)
```

8

```
 9 // or via composition, which is a bit ugly:
```

```
10 val c = g(f(a))
```

11

```
12 // "unix-pipey"!
```

```
13 val c = a |> f |> g
```

Style

When you just can't stand all that (keyboard) typing:

```
val p: Boolean
```

2

```
3 // ternary-operator-ish
```

```
4 p ? "yes" | "no" // if (p) "yes" else "no"
```

5

```
6 val o: Option[String]
```

7

```
8 o | "meh"         // o.getOrElse("meh")
```

Style

More legible (and more type-safe):

```scala
// scala
```

```
2 Some("foo")  // Some[String]
```

```
3 None          // None.type
```

4

```
5 // scalaz
```

```
6 "foo".some   // Option[String]
```

```
7 none          // Option[Nothing], oops!
```

```
8 none[String] // Option[String]
```

Style

More legible (and more type-safe):

```scala
// scala
```

```
2 Right(42)   // Right[Nothing, Int], oops!
```

```
3 Left("meh") // Left[String, Nothing], oops!
```

```
4 Right[String, Int](42)   // verbose
```

```
5 Left[String, Int]("meh") // verbose
```

6

```
7 // scalaz
```

```
8 42.right[String] // Either[String, Int]
```

```
9 "meh".left[Int]  // Either[String, Int]
```

Validation

Validation

The Java way, eww:

```
1 def fetch(uri: URI): String
```

2

```
3 def meh(t: Throwable) = ...
```

```
4 def gotIt(s: String)  = ...
```

5

```
6 try {
```

```
7   val result = fetch(...)
```

```
8   gotIt(result)
```

```
 9 } catch {
```

```
10    case e: meh(e)
```

```
11 }
```

Validation

In Scala we represent the two cases in one type, `Either`:

```
1 def fetch(uri: URI): Either[Throwable, String]
```

2

```
3 def meh(t: Throwable) = ...
```

```
4 def gotIt(s: String)  = ...
```

5

```
6 val result = fetch(...)
```

Validation

And handle it in multiple ways. Via pattern match:

```
result match {
```

```
2     case Left(l)  =>
```

```
3       // Left is a fail, right?
```

```
4       meh(l)
```

```
5   case Right(r) =>
```

```
6       // Right must be right?
```

```
7       gotIt(r)
```

```
8 }
```

Validation

Via `fold()`:

```
result.fold(
```

```
2     l => meh(l),
```

```
3     r => gotIt(r))
```

Validation

Via for-comprehension:

```
1 for {
```

2

```
3 }
```

Validation

TODO: JUST A LIST OF TOPICS, WILL BE DELETED

- better names: Failure/Success vs. Left/Right

- monadic without the need for left/right projection, defaults to right

- accumulates errors via Semigroup append |+|

- ValidationNEL[X, A] alias for Validation[NonEmptyList[X], A]

- NonEmptyList has a Semigroup so you get error accumulation "for free"

- multi-level case class validation

- companion object apply() pattern

- Validation is Applicative so you can combine multiple Validations where failures are accumulated

Lenses

Lenses

TODO: WHEN? WHY?

Lenses

```
1 case class Foo(name: String, factor: Int)
```

2

```
3 case class FooNode(
```

```
4     value: Foo,
```

```
5   children: Seq[FooNode] = Seq())
```

Lenses

```
1 val tree =
```

```
2    FooNode(
```

```
3     Foo("root", 11),
```

```
4       Seq(
```

```
5         FooNode(Foo("child1", 1)),
```

```
6       FooNode(Foo("child2", 2)))) // <-- * 4
```

Task: Create a new tree where the *second child's* `factor` is multiplied by 4.

Lenses

Let's try all at once:

```
1 val secondTimes4: FooNode => FooNode =
```

```
2     node => node.copy(children = {
```

```
3       val second = node.children(1)
```

```
4       node.children.updated(
```

```
5       1,
```

```
6       second.copy(
```

```
7           value = second.value.copy(
```

```
8           factor = second.value.factor * 4)))
```

```
9     })
```

Eww.

Lenses

```
Lens[Thing, View](
```

```
2     get: Thing => View,
```

```
3     set: (Thing, View) => Thing)
```

4

```
5 val thing: Thing
```

```
6 val lens:  Lens[Thing, View] = ...
```

7

```
8 val view: View = lens(thing) // apply = get
```

9

```
10 // "set" a transformed View
```

```
11 val thing2: Thing = lens.mod(thing, v: View => ...)
```

12

```
13 // Lots of other operations on a Lens...
```

Lenses

```
1 val second: Lens[FooNode, FooNode] =
```

```
2   Lens(
```

```
3     _.children(1),
```

```
4       (node, c2) => node.copy(
```

```
5         children = node.children.updated(1, c2)))
```

6

```
7 val value: Lens[FooNode, Foo] =
```

```
8   Lens(
```

```
 9       _.value,
```

```
10      (node, value) => node.copy(value = value))
```

11

```
12 val factor: Lens[Foo, Int] =
```

```
13    Lens(
```

```
14      _.factor,
```

```
15      (foo, fac) => foo.copy(factor = fac))
```

Lenses

Lenses *compose*:

```
1 val secondFactor =
```

```
2    second andThen value andThen factor
```

Lenses

```
1 /* FooNode(
```

```
2       Foo("root", 11),
```

```
3       Seq(
```

```
4         FooNode(Foo("child1", 1)),
```

```
5         FooNode(Foo("child2", 2))))
```

6 ^

7 ^

```
8                                   ^  */
```

```
9 secondFactor(tree)        // 2
```

Lenses

```
1 /* FooNode(
```

```
2       Foo("root", 11),
```

```
3       Seq(
```

```
4         FooNode(Foo("child1", 1)),
```

```
 5         FooNode(Foo("child2", 2))))
```

6 ^

```
7                                   ^  */
```

```
 8 secondFactor.mod(tree,   _ * 4)
```

```
 9 /* FooNode(                    ^
```

```
10      Foo("root", 11),        ^
```

```
11        Seq(                          ^
```

```
12          FooNode(Foo("child1", ^)),
```

```
13          FooNode(Foo("child2", 8))))
```

```
14  */
```

Dependency Injection

TODO: these are just notes

- Reader monad, really just function composition

- Why this doesn't work without scalaz: no map/flatMap for Function1

Thanks!

Credits, sources and references:

- scalaz homepage, scalaz 6.0.4 source cross-reference

- jrwest/learn_you_a_scalaz

- debasishg/tryscalaz

- Runar Oli, Dead-Simple Dependency Injection

- Tony Morris, Dependency Injection Without the Gymnastics

scalaz "For the Rest of Us" Adam Rosien

arosien@box.com && adam@rosien.net

`@arosien #scalasv #scalaz`