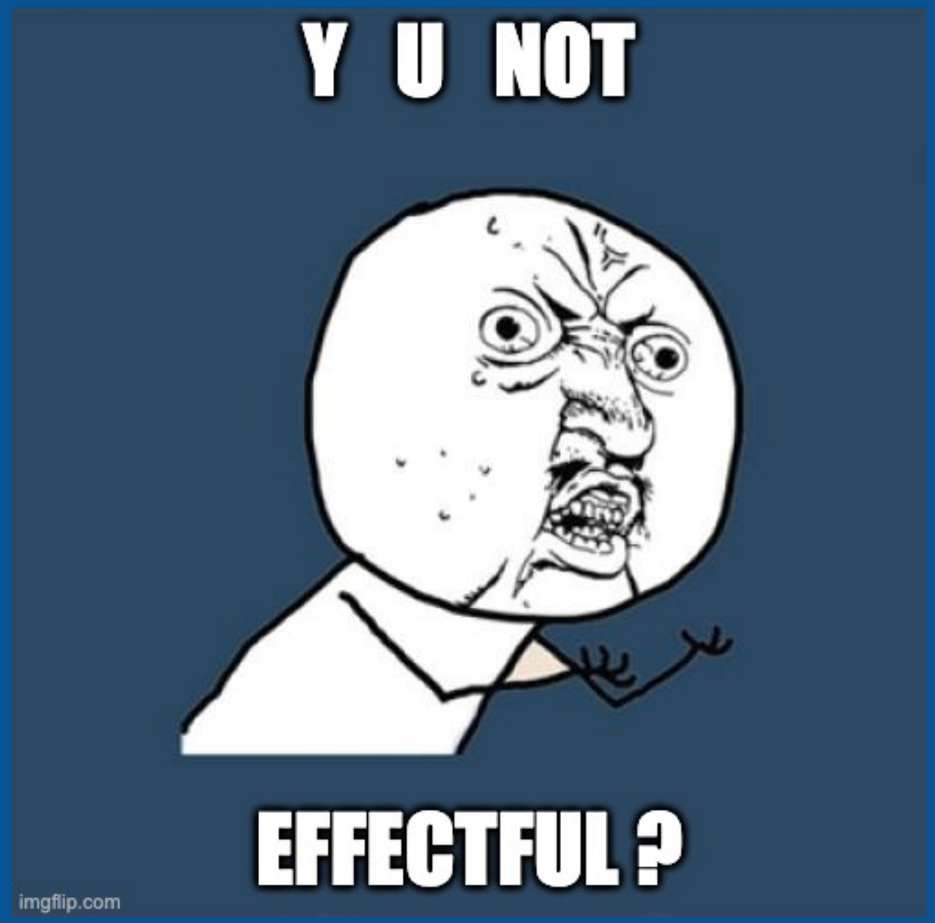
A presentation with memes

WHAT IS ZIO?

ZIO IS A FUNCTIONAL LIBRARY FOR ASYNCHRONOUS AND CONCURRENT PROGRAMMING CREATED BY JOHN DE GOES

WHAT IS WRONG WITH FUTURE[_] ?

FUTURE CANNOT WRAP SIDE EFFECTS.



For example, the following code

```
val myFuture = for {
   _ <- Future{println("Hello")}
   _ <- Future{println("Hello")}
} yield ()
```

cannot be refactored into

```
lazy val myEffect = Future{println("Hello")}

val myFuture = for {
   _ <- myEffect
   _ <- myEffect
} yield ()
```

(Hello is only displayed once on the screen.)

ALWAYS NEEDS TO HAVE AN EXECUTIONCONTEXT

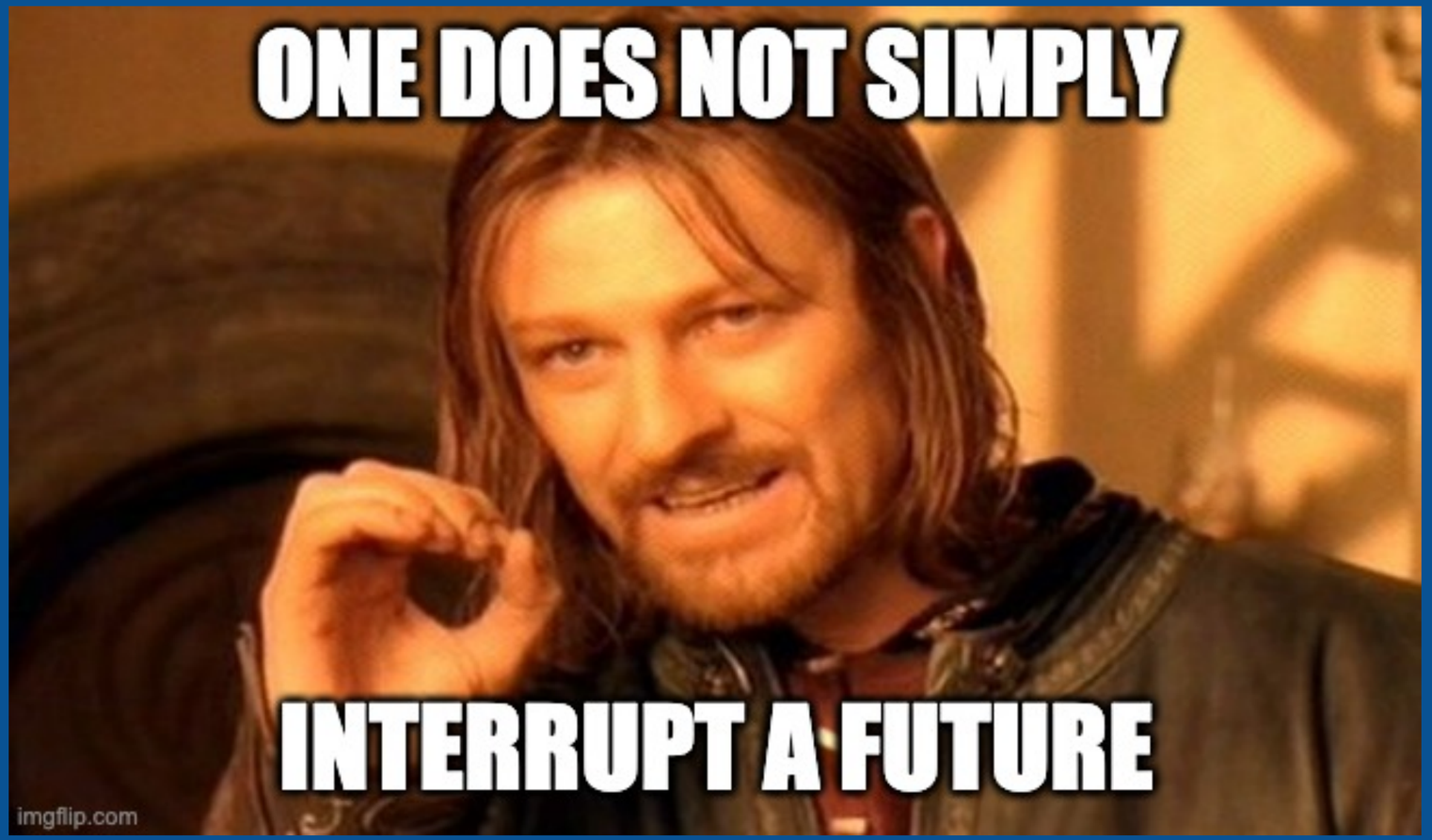
AND THIS IMPLICIT IS EVERYWHERE!

Most methods of Future use this and very often we end up writing a lot of boilerplate just for that.

```
def map[S](f (T) => S)(implicit executor: ExecutionContext): Future[S]

def foreach[U](f (T) => U)(implicit executor: ExecutionContext): Unit
```

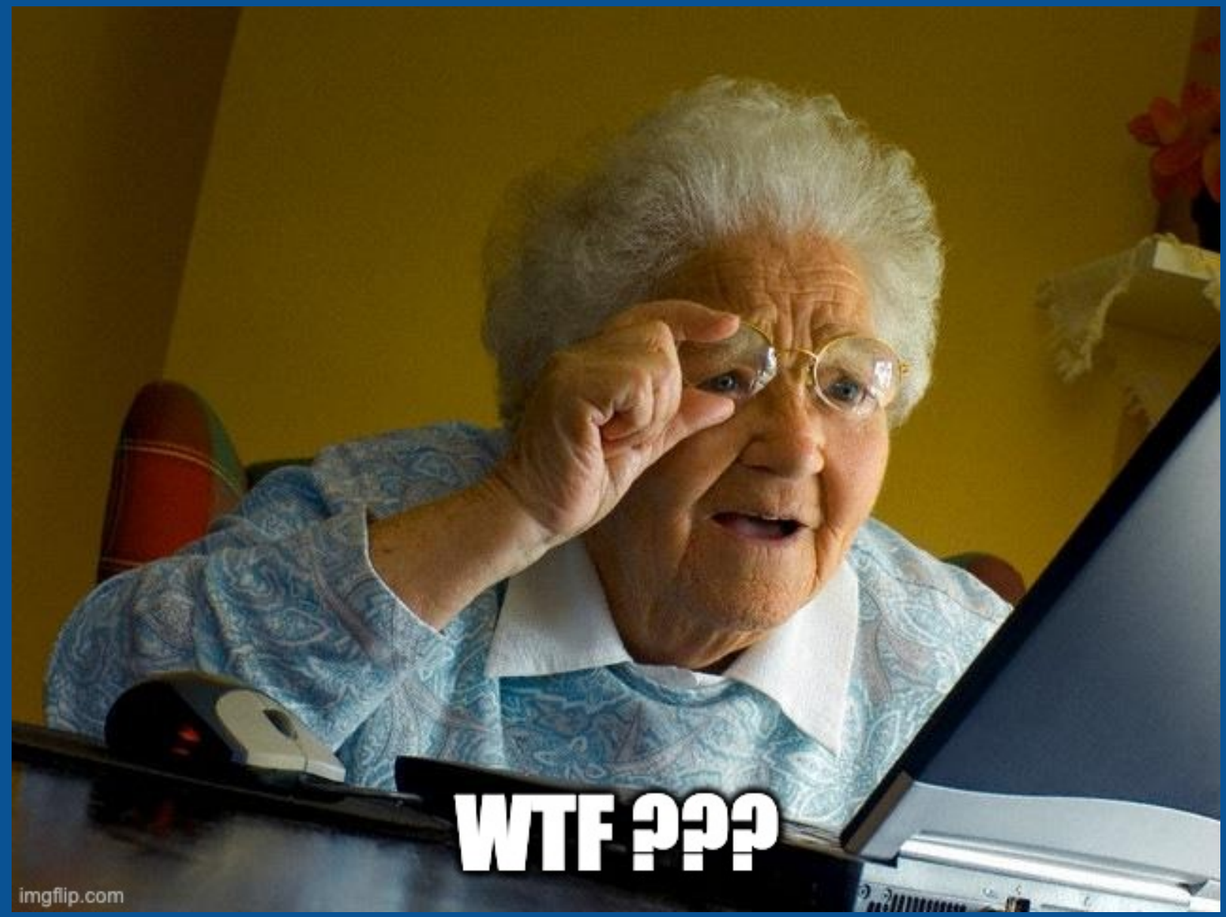
FUTURE IS UN-INTERRUPTABLE



```scala
val future: Future[Response] = ???
val executed: Response = Await.result(future, 60.seconds)
```

Let's imagine that your Future is writing to a database.
If it takes longer than 60 seconds, the process will keep on running in the background, potentially consuming resources.

FUTURE IS HARD TO TROUBLESHOOT



```
PostgresException: Syntax error at or near 42
    at example$.getConnection(example.scala:43)
    at example$.$anonfun$asyncDbCall$1(example.scala:23)
    at scala.concurrent.Future$.$anonfun$apply$1(Future.scala:658)
    at scala.util.Success.$anonfun$map$1(Try.scala:255)
    at scala.util.Success.map(Try.scala:213)
    at scala.concurrent.Future.$anonfun$map$1(Future.scala:292)
    at scala.concurrent.impl.Promise.liftedTree1$1(Promise.scala:33)
    at scala.concurrent.impl.Promise.$anonfun$transform$1(Promise.scala:33)
    at scala.concurrent.impl.CallbackRunnable.run(Promise.scala:64)
    at java.util.concurrent.ForkJoinTask$RunnableExecuteAction.exec(ForkJoinTask.java:1402)
    at java.util.concurrent.ForkJoinTask.doExec(ForkJoinTask.java:289)
    at java.util.concurrent.ForkJoinPool$WorkQueue.runTask(ForkJoinPool.java:1056)
    at java.util.concurrent.ForkJoinPool.runWorker(ForkJoinPool.java:1692)
    at java.util.concurrent.ForkJoinWorkerThread.run(ForkJoinWorkerThread.java:157)
```

We see a lot from the implementation from Future not so much about our code 😩

THE DETAILS

ONE EFFECT TYPE, MULTIPLE ALIASES

THE WHOLE LIBRARY IS BASED AROUND ONE DATATYPE: THE ZIO CLASS.

A ZIO object is the description of a process.

It can be an atomic action (initialize a variable for example) or composed of multiple sub-processes ("Read from input", "Process the data", "Write to output" can all be wrapped into "ETL process" for example).

```
ZIO[R, E, A]
```

Where:

» R is the Environment Type

» E is the Failure Type

» A is the Success Type

This can be summarized by:

```
R => Either[E, A]
```

When you think about it, a Future[A] can be represented as the following type

```
ZIO[ExecutionContext, Throwable, A]
```

We need an ExecutionContext to run it and it returns either a value of type A if everything went fine or a Throwable if there was an Exception.

THERE ARE MULTIPLE ALIASES

```
type UIO[A] = ZIO[Any, Nothing, A]

type Task[A] = ZIO[Any, Throwable, A]

type RIO[R, A] = ZIO[R, Throwable, A]

type IO[E, A] = ZIO[Any, E, A]

And many others!
```

USING ZIO

BASIC OPERATIONS

```scala
import zio._

// You can map
val answerToLife: UIO[Int] = IO.succeed(21).map(_ * 2)

// You can use your Future
val myFuture: Future[A] = ???
val myZioFuture: IO[Throwable, A] = ZIO.fromFuture(myFuture)

// You can chain your calls together
val consoleDependentChain = getStrLn
    .flatMap(input => putStrnLn(s"Read $input from line"))
```

BASIC OPERATIONS

For-comprehension makes our functional code look more "procedural"

```
import zio._

val program =
   for {
       _    <- putStrLn("Hello! What is your name?")
       name <- getStrLn
       _    <- putStrLn("Nice to meet you, $name !")
   } yield ()
```

HAVING FUN WITH FIBERS

WHAT ARE FIBERS?

“[...] Fibers are a lightweight mechanism for concurrency.
You can fork any IO[E, A] to immediately yield an UIO[Fiber[E, A]].”

-- Official documentation

From the documentation:

```
import zio._

val analyzed =
  for {
    fiber1   <- analyzeData(data).fork  // IO[E, Analysis]
    fiber2   <- validateData(data).fork // IO[E, Boolean]
    // Do other stuff
    valid    <- fiber2.join
    _        <- if (!valid) fiber1.interrupt
                else IO.unit
    analyzed <- fiber1.join
  } yield analyzed
```

QUICK DEMO

TO GO FURTHER

WEBSITE

» The official website

YOUTUBE VIDEOS

» John De Goes - Upgrade Your Future

» Fabio Labella—How do Fibers Work? A Peek Under the Hood

TO GO FURTHER (CONTINUED)

ONLINE CHATS

» The official Discord channel

» The official Gitter channel

TRAININGS FOR THE GENERAL PUBLIC

» John De Goes's Patreon page - The Monad and Spartan tiers are worth the price!