Scala-Helsinki

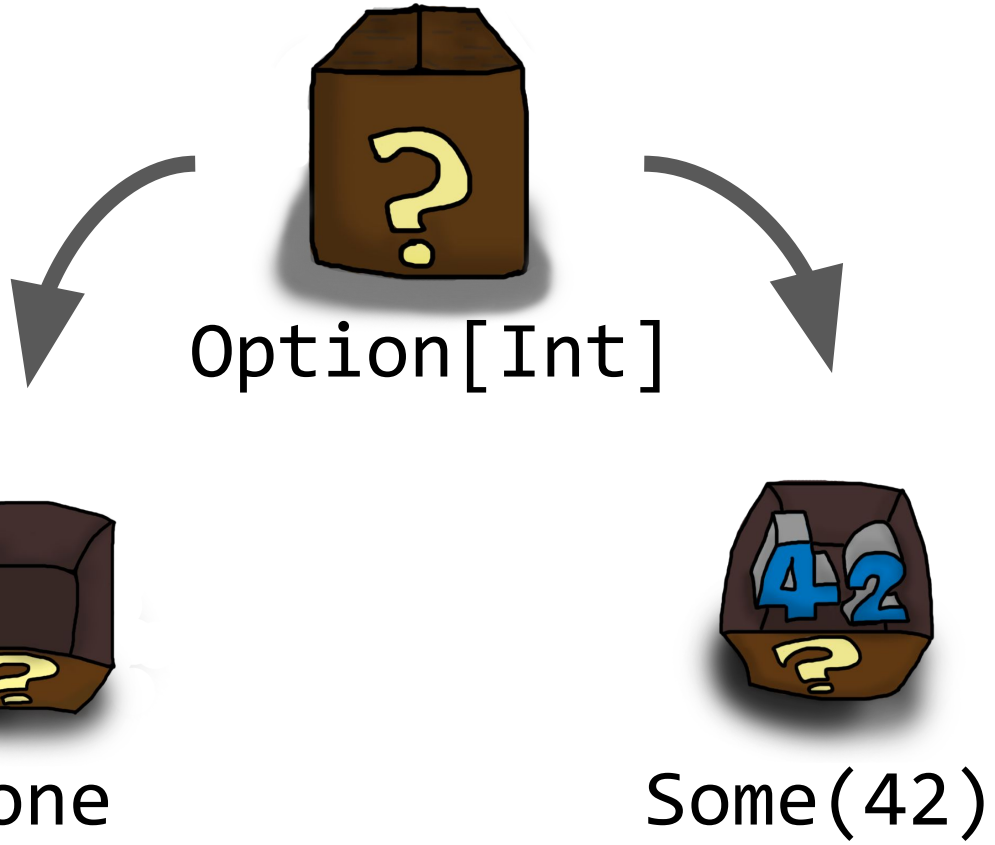
Meetup
29/09/2016

Hands-on session on `Option`, `Future` and use of those with `Actors`

So what is an **`Option`**?

- Represents a value that can be missing
- It's a parametric type (i.e. `Option[T]`)
- Main subtypes: `Some` and `None`
- Rich API for manipulating and combining `Options` (e.g. `map`, `flatMap`)
- Resemblance with type `Try[T]`
- `API docs:` http://www.scala-lang.org/api/current/#scala.Option

What does it look like?



So what is a `Future`?

- Represents a value that will be available at a later time (or more accurately, a "computation")
- It's a parametric type (i.e. `Future[T]`)
- Since the computation may fail, the contained value is a Try[T] (i.e. Success or Failure)
- Rich API for manipulating and combining `Futures`
- API docs: http://www.scala-lang.org/api/current/#scala.concurrent.Future

What does it look like?

Future[T]

Do
something
else
meanwhile...

Try[T]

Time

And what is an **`Actor`**?

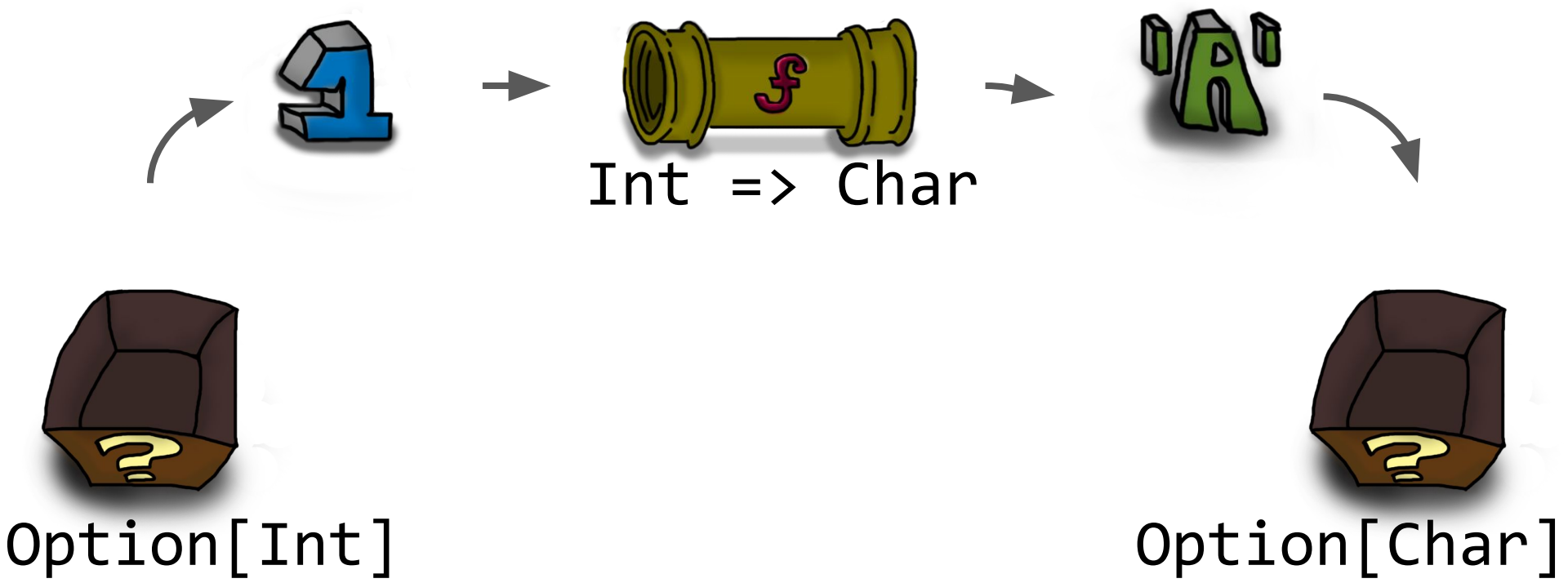
- a model for concurrent computation
- the basic unit of computation in this model
- strong analogy with real-life processes
- easy to reason about (most of the time :))
- share nothing - the only way to mutate state is by sending immutable messages between actors
- Docs: http://akka.io/docs/

What does it look like?

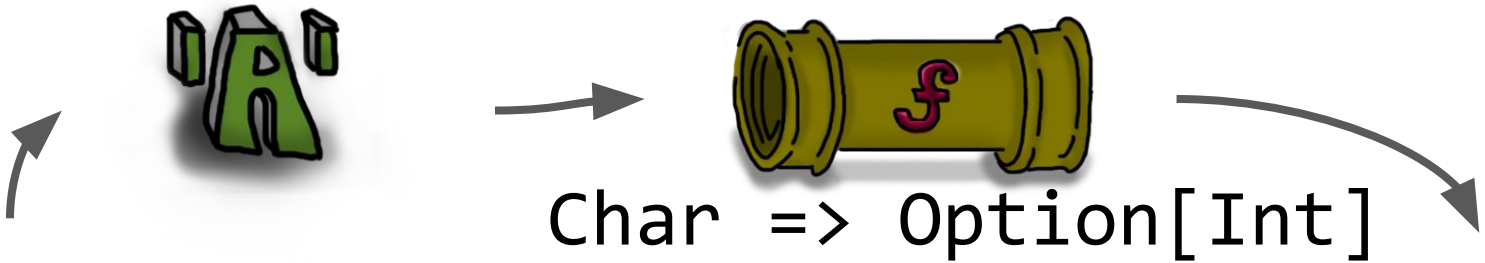
Common operations

- **isEmpty** (Option), **isCompleted** (Future)
- **map**: transforms the "contained" value
- **flatMap**: transforms the "contained" value and wraps it into another box
- **filter**: discards the "contained" value if the given predicate is false

How map works?



How `flatMap` works?



Option[Char]

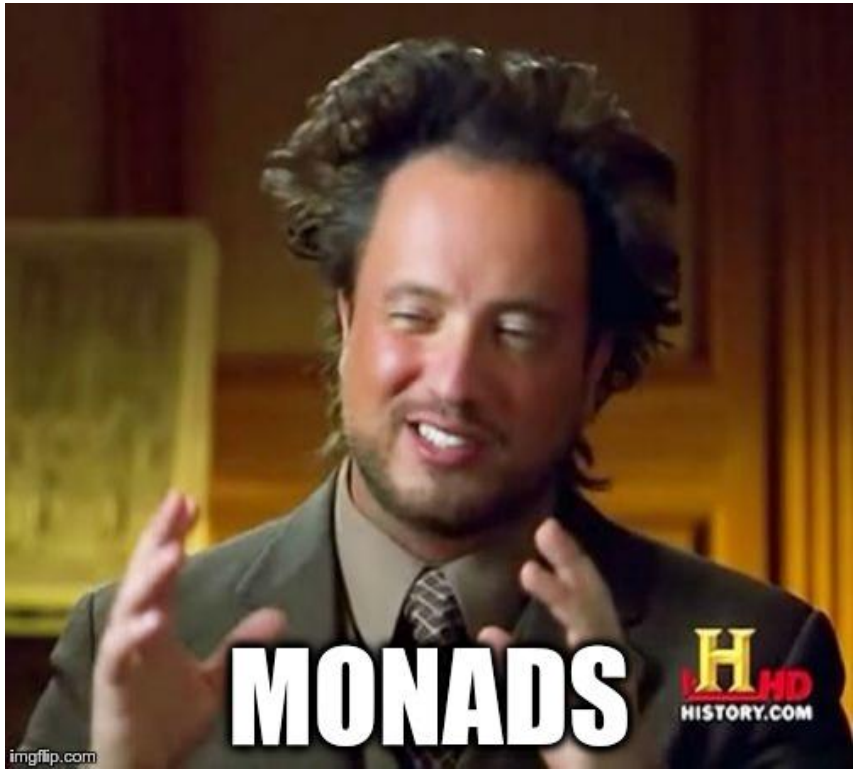
Option[Int]

Hands on!

https://github.com/ale64bit/meetup

Example #1: `Future+Option` with for-comprehension

- powerful abstractions for representing and combining computations
- simple enough to reason about
- not isolated; can interact with each other
- for-comprehension can be decomposed in a sequence of map+flatMap operations
- along with most other components in Scala build-in library (notably, collections), they present monadic characteristics...



Example #2: Guess the types after `getOrElse`

- `getOrElse` is practical, although dangerous
- Scala is full of unexpected effects
- the type system is powerful, but good knowledge is required to not run into those unwanted effects
- there are millions of ways of doing the same thing: Scala can be horrible for team-working

Example #3: Futures as workers

- `Futures` can be leveraged as workers to accomplish several concurrent computations
- the `ExecutionContext` nature is key for a performant system (don't use the default one!)
- `blocking` pattern can be useful as well to mark blocking computations

BONUS QUESTION: why worker numbers are always odd?

Example #4: Futures and Actors

- ask and pipe patterns allow interaction with Futures
- care should be taken when using the Actor's dispatcher as ExecutionContext
- ask should be used with care as well; beware of timeout hell and the temporary Actor who actually "asks"
- actors are low-level primitives
- as before, many different ways to accomplish the same things

Thanks!
Questions?