Uncovering the Power of Option, Either and Try

Scala for Beginners Series

Thursday, February 11, 2021

Why Scala?

- ✓ Write clean, concise, powerful code — less boilerplate
- ✓ Fast development, prototyping
- ✓ Easier to write robust, fault-tolerant applications
- ✓ Wizards use it

Functional Programming

…a programming paradigm where programs are constructed by applying and composing functions. It is a declarative programming paradigm in which function definitions are trees of expressions that map values to other values, *rather than a sequence of imperative statements which update the running state of the program*.

Wikipedia

```
int x = 0;
while (data.ready()) {
    int a = data.next();
    if (a > 0) x += a * a;
}
```

```
data.filter(_ > 0)
     .map(a => a * a)
     .sum
```

Quick advice for beginners: dump “var”

Scala has two types of variable declarations:

```
var x: Int = 1 // 1
x += 3         // now 4
```

```
val x: Int = 1
x += 3
```

- `error: reassignment to val`

*If you really want to learn Scala, **don’t let yourself <u>ever</u> use var**. It will seem incredibly hard at first. It may even seem impossible. (But it isn’t!) Eventually, your brain will make the paradigm shift.*

One hint: in some seemingly sticky situations, you’ll have to use recursion. It may seem weird, but don’t fight it. You can indeed evict the evil vars!

Writing robust, fault tolerant code

Java’s `NullPointerException` is about as popular and notorious as the old BSOD.

How does Scala avoid it?

- All functions must return something (whatever type they declare)
- Variables (`val`s) don’t get defined with uninitialized state
- Functional programmers consider “unhappy paths”

Option[T]

This is the easiest typeclass to understand:

```
val name: Option[String] = Some("Murray")
val name: Option[String] = None
```

It lets you consider cases where you may or may not have a value. Consider looking up a name from a user database. What if the user ID can’t be found?

```
def lookupName(id: String): Option[String]
```

Option is found throughout Scala library

```
scala> val users = Map(1 -> "Alley", 2 -> "Sam")
scala> users.get(1)
val res0: Option[String] = Some(Alley)

scala> users.get(3)
val res1: Option[String] = None

scala> users(3)
java.util.NoSuchElementException: key not found: 3
  at scala.collection.immutable.Map$Map2.apply(Map.scala:298)
  ... 32 elided
```

Try[T]

This is next-most-natural way to handle things:

```
val name: Try[String] = Success("Murray")
val name: Try[String] = Failure(new Exception(…))
```

If you're working with Java code or APIs where exceptions can be thrown (even simple `IOExceptions`) you can wrap it in a Try call.

```
Try(db.lookup(name))
```

Either[L,R]

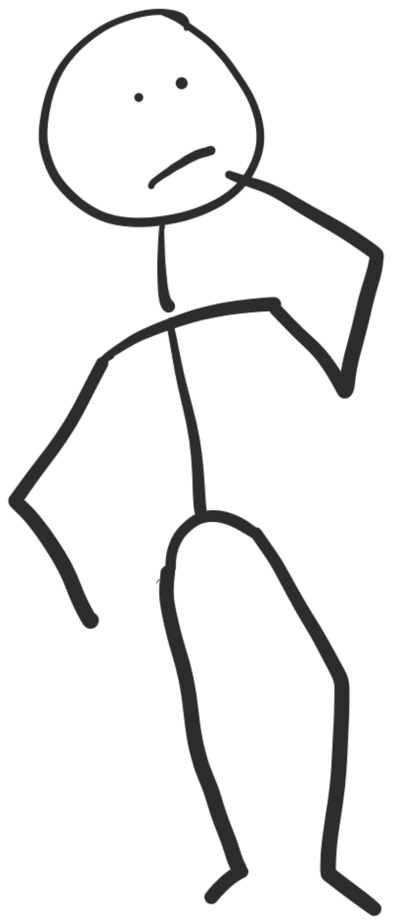
Strictly, an Either typeclass lets you have one of two types.

```
    def getAge(a: Int): Either[String,Int] =
      if (a < 16)
        Left("Renter is not old enought to drive.")
      else Right(a)
```

```
scala> getAge(10)
val res0: Either[String,Int] = Left(Renter is not old enought to drive.)
scala> getAge(20)
val res1: Either[String,Int] = Right(20)
```

Conventionally, consider L to be the “unhappy path” and R to be the “happy path”. (Pneumonic: think happy as what’s “right”)

I think I
get the gist
of this.



It can't be too
hard to use...

Let’s try to apply Option to a typical situation...

```
scala> val users = Map(1 -> "Alley", 2 -> "Sam")

scala> val greeting = if (users.contains(1)) { "Hello " + users(1) }
val greeting: Any = Hello Alley

scala> val greeting = if (users.contains(1)) { "Hello " +
                          users.get(1) } else "Don't know"
val greeting: String = Hello Some(Alley)

scala> val greeting = if (users.contains(1)) { "Hello " +
                          users(1) } else "Don't know"
val greeting: String = Hello Alley

scala> val greeting = if (users.contains(1)) { Some("Hello " +
                          users(1)) } else None
val greeting: Option[String] = Some(Hello Alley)
```

Let’s try something more realistic

```
case class Contact(id: Int, name: String, phone: Option[String],
                   email: Option[String], friendIds: List[Int]) {
  import Contact._

  /*
   * Send an email to friends, inviting to join
   * @return List of successfully sent emails
   */
  def inviteFriends: List[Int] = ???

}


object Contact {

  def sendEmail(email: String): Boolean = ???

  def dbLookup(id: Int): Try[Contact] = ???

}
```

Imperative (non-functional) approach

```
def inviteFriends(friendIds: List[Int]): List[Int] = {

  var successes: List[Int] = List.empty

  for (id <- friendIds) {
    // lookup id from database
    // if you can find it and it’s defined, send an email
    // if it was successful, add the id to the successes list
  }

  successes
}
```

Functional approach

Approach:

Start with a list of friend IDs to try

End up with a list of friend IDs from successful emails

Functional strategy:

Apply an operation (lookup & send) to each item in the list, converting it into a success.

```
friendIds.map(id => lookupAndSendWithSuccess(id))
```

This is sort of what we want. We really want to shorten the list to only have successful deliveries, but let's not worry about that yet…

First "functional" iteration

```
def inviteFriends: List[Int] = {

  val successes = friendIds.map { id =>
    dbLookup(id) match {
      case Failure(_) => None
      case Success(friend) => {
        friend.email match {
          case Some(email) => if (sendEmail(email)) Some(id) else None
          case None => None
        }
      }
    }
  }
  ??? // success is now of type List[Option[Int]]
}
```

Unhappy 1

Happy Path 1

Happy Path 2

Unhappy 2

Happy Path 3

Unhappy 3

Yuck!

You said Scala was pretty, clean and succinct!

Can I just keep doing imperative programming?

Please!

“Map” to the rescue!

Let’s just write code snippets that focus on the “happy paths”

`Option[T].map(f: T => U)` becomes `Option[U]`

`Try[T].map(f: T => U)` becomes `Trt[U]`

`Either[L,R].map(f: R => S)` becomes `Either[L,S]`

“Map” to the rescue!

Let's just write code snippets that focus on the “happy paths”

```
friend.email match {
  case Some(email) => if (sendEmail(email)) Some(id) else None
  case None => None
}
```

```
friend.email.map(e => sendEmail(e))
```

We get to think of e as a String (email) rather than an Option[String]

```
friend.email.map(sendEmail)
```

To apply a function that only takes the one parameter, you can simplify by just supplying the function name.

Second “functional” iteration

```
def inviteFriends: List[Int] = {

  val successes = friendIds.map { id =>
    dbLookup(id).map { contact =>
      contact.email.map { address =>
        sendEmail(address)
      }
    }
  }

  ??? // successes is now of type List[Try[Option[Boolean]]]
}
```

This still doesn't look clean!

Can I just keep doing imperative programming?

How do I work with a List of Try[Option[Boolean]] ??

Boxes within boxes! (Turtles all the way down)

Let’s start with 3 “safe” functions, each of which may or may not yield a successful transformation.

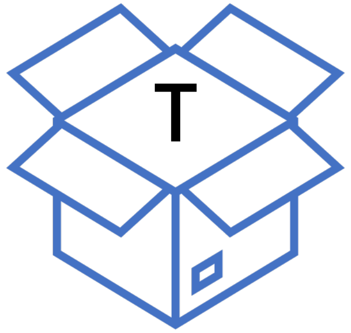
```
f: A => Option[B]
g: B => Option[C]
h: C => Option[D]
```

We want to apply these in order, like f(x).map(g).map(h), but we don’t want Option[Option[Option[D]]], we just want Option[D]

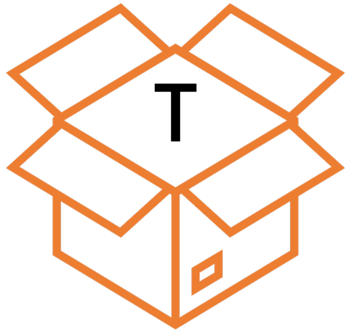
“FlatMap” to the rescue!

```
scala> def getName(id: Int): Option[String] = users.get(id)
def getName(id: Int): Option[String]

scala> val id: Option[Int] = Some(2)
val id: Option[Int] = Some(2)

scala> id.flatMap(getName)
val res3: Option[String] = Some(Sam)

scala> val id: Option[Int] = Some(3)
val id: Option[Int] = Some(3)

scala> id.flatMap(getName)
val res2: Option[String] = None
```

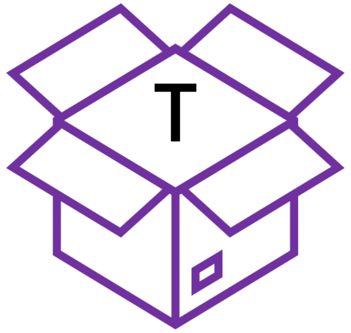
Think of Option, Try and Either as “Boxes”



Option[T]
Some[T] or
None



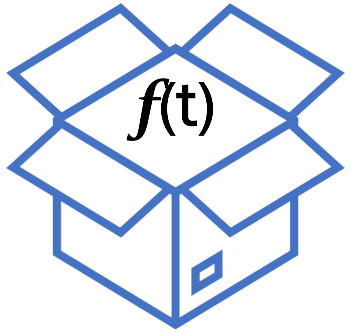
Try[T]
Success[T] or
Failure[E]



Either[S,T]
Right[T] or
Left[S]

All use "map" to apply functions to their contents.

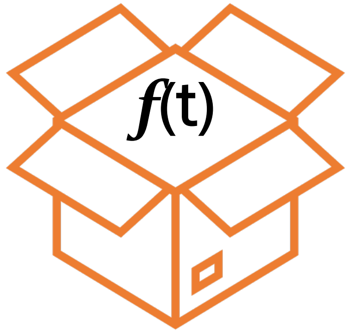
(The functions are ignored if the boxes are empty)



Option[T]

Some[T] or

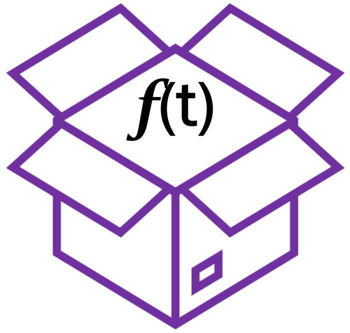
None



Try[T]

Success[T] or

Failure[E]



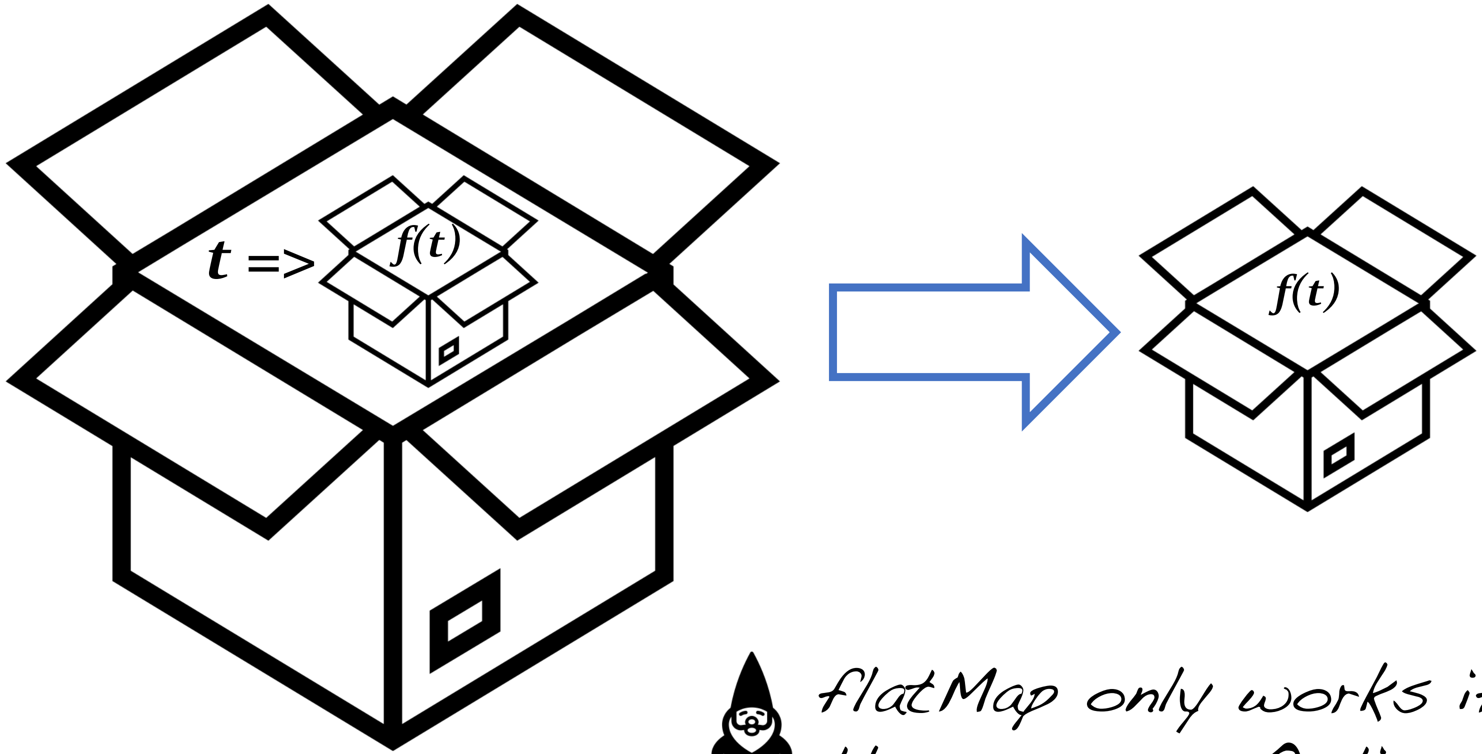
Either[S,T]

Right[T] or

Left[S]

All use "flatMap" for functions that would result in nested boxes.

(The functions are ignored if the boxes are empty)



flatMap only works if the boxes are the same, so Options inside Options

For multiple flatMap to work, keep the boxes the same!

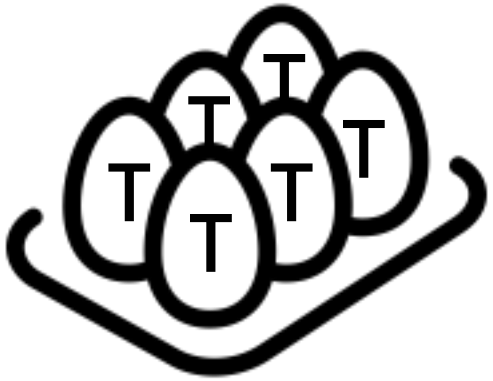
```
case class User(id: Int, email: Option[String])
def findUser(id: Int): Option[User]
def sendEmail(email: User): Option[String] // None if email fails

val sent = findUser(id).flatMap(_.email).flatMap(sendEmail)
send will be of type Option[String]
```

Trick #1: Quick Convert Try[T] to Option[T]

```
case class User(id: Int, email: Option[String])
def findUser(id: Int): Try[User]
def sendEmail(user: User): Option[Int] // None if email fails

val sent = findUser(id).toOption
                       .flatMap(_.email).flatMap(sendEmail)
```

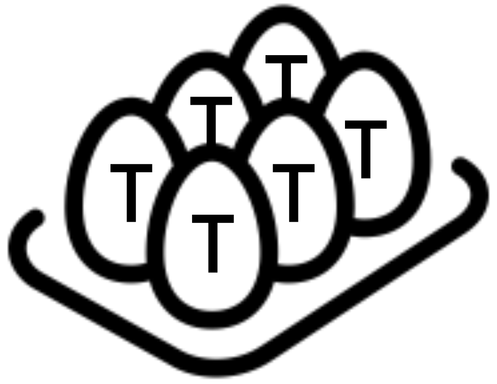
Remember how we mapped on List?

List is just another type of box, except it doesn't hold zero or one item of type T…



… it holds zero or more items of type T.

Remember how we mapped on List?



”map” applies a function to each of the items

If your function generates its own lists, flatMap “flattens” them!

```
scala> val myList = List(3,2,1)
val myList: List[Int] = List(3, 2, 1)

scala> myList.map(x => List.fill(x)(x))
val res0: List[List[Int]] = List(List(3, 3, 3), List(2, 2), List(1))

scala> myList.flatMap(x => List.fill(x)(x))
val res1: List[Int] = List(3, 3, 3, 2, 2, 1)
```

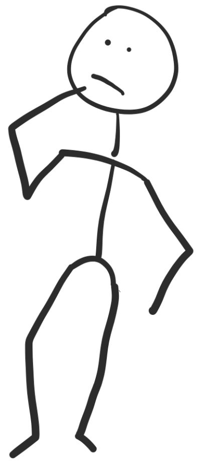
Options are also Lists!

Remember how we have to keep the ”boxes” the same for flatMap to work? Well, if you have an Option[T], it doubles as a List[T] with either zero (None) or one (Some(t)) elements!

```
val friends: List[Contact] = friendIds.flatMap(id => dbLookup(id).toOption)
val knownEmails: List[String] = friends.flatMap(_.email)
```

We're almost done!

```
def inviteFriends: List[Int] =
  friendIds
    .flatMap(id => dbLookup(id).toOption
      .flatMap(contact => contact.email)
      .filter(sendEmail).map(_ => id))
```



That's not so bad, and I do have all the built-in safety stuff...

but I'm still not sold.

Introducing For-comprehensions

```scala
def inviteFriends: List[Int] =
  friendIds
    .flatMap(id => dbLookup(id).toOption
      .flatMap(contact => contact.email)
      .filter(sendEmail).map(_ => id))
```

```scala
def inviteFriends: List[Int] = {
  for {
    id <- friendIds
    contact <- dbLookup(id).toOption
    e <- contact.email
    status = sendEmail(e)
    if status
  } yield(id)
}
```

Looking at the syntax in-depth

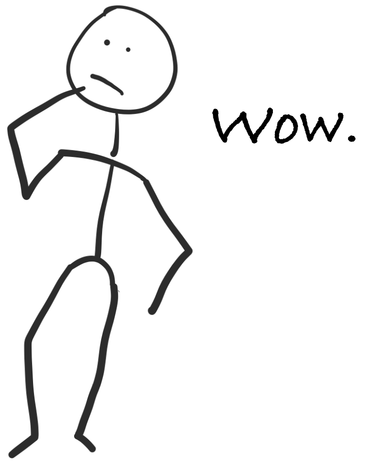
```scala
def inviteFriends: List[Int] = {
  for {
    id      <- friendIds
    contact <- dbLookup(id).toOption
    e       <- contact.email
    status   = sendEmail(e)
    if status
  } yield(id)
}
```

Whatever type of Box we start with dictates the rest of the Boxes

These are flatMap operations
<-

A simple map (not flatMap) is represented by =

"if" statements translate to .filter statements



Summary

- If you’re going to learn FP, start with banishing “var”
- Everything is a composition of functions
- Rather than iterating on lists, use “map” and “filter” and just transform them through your various stages
- Handle potential ‘issues’ with Option, Try or Either
- Operate on their contents by using ”map” and focusing on the happy path
- FlatMap keeps you from boxing (nesting) your boxes
- for-comprehensions create super-clean syntax for all of this

Unveiling my Evil Master Plan…

You just learned about Monads

- A monad is a kind of typeclass that “boxes” any other type
- It has .map that applies functions to the contents of the boxes
- It has .flatMap for functions that create their own boxes in order to avoid nesting problems
- Option, Try, Either and Future are good examples of Monads
- They isolate “pure functions” from problematic contexts like asynchronous operations, exception handling, missing data, etc.