

# Monads in Functional Programming using Scala

Marco Mantovani

January 31, 2021

# Table of contents

- 1 Preamble: Type classes
- 2 Functors
- 3 Monads
- 4 Sitography and Source codes

# Preamble: Type classes

When we want to extend an existing class with a new interface (and so, possibly, with new functionalities) we have four choice:

- 1 Extend the class itself, but often we can't modify it (it could be taken from an API)

# Preamble: Type classes

When we want to extend an existing class with a new interface (and so, possibly, with new functionalities) we have four choice:

- 1 Extend the class itself, but often we can't modify it (it could be taken from an API)
- 2 Create a new class that extends the original class and the new interface (not always possible)

# Preamble: Type classes

When we want to extend an existing class with a new interface (and so, possibly, with new functionalities) we have four choice:

- ❶ Extend the class itself, but often we can't modify it (it could be taken from an API)
- ❷ Create a new class that extends the original class and the new interface (not always possible)
- ❸ Create a wrap class that contains the original class and extends the new interface

# Preamble: Type classes

When we want to extend an existing class with a new interface (and so, possibly, with new functionalities) we have four choices:

- ❶ Extend the class itself, but often we can't modify it (it could be taken from an API)
- ❷ Create a new class that extends the original class and the new interface (not always possible)
- ❸ Create a wrap class that contains the original class and extends the new interface
- ❹ Using type classes

The last choice is a classical pattern of FP.

# Preamble: Type classes - Example (1)

Let's present type classes with an example. We have this scala interface

```
1 trait Summable[T] {  
2     def sumElements(list: List[T]): T  
3 }
```

and we want that Int and String class implement it.

# Preamble: Type classes - Example (1)

Let's present type classes with an example. We have this scala interface

```
1 trait Summable[T] {  
2     def sumElements(list: List[T]): T  
3 }
```

and we want that `Int` and `String` class implement it.

Of course we cannot change `Int` or `String` classes directly (they are into scala's library), so the first option is discarded.



# Preamble: Type classes - Example (1)

Let's present type classes with an example. We have this scala interface

```
1 trait Summable[T] {  
2     def sumElements(list: List[T]): T  
3 }
```

and we want that `Int` and `String` class implement it.

Of course we cannot change `Int` or `String` classes directly (they are into scala's library), so the first option is discarded.

We could create a new classes `MyInt` and `MyStr` that extends respectively `Int` and `String` but that classes are final, so we cannot work in this way.

# Preamble: Type classes - Example (1)

Let's present type classes with an example. We have this scala interface

```
1 trait Summable[T] {  
2   def sumElements(list: List[T]): T  
3 }
```

and we want that `Int` and `String` class implement it.

Of course we cannot change `Int` or `String` classes directly (they are into scala's library), so the first option is discarded.

We could create a new classes `MyInt` and `MyStr` that extends respectively `Int` and `String` but that classes are final, so we cannot work in this way.

Surely we can create a wrap classes `MyInt` and `MyStr`, but in this way we have now two definition of `Int` and of `String` and it could be messy.

## Preamble: Type classes - Example (2)

Let's try type classes! We start creating two implicit objects that implement the interface as follow:

```
1 implicit object IntSummable extends Summable[Int] {  
2   def sumElements(list: List[Int]): Int = list.sum  
3 }  
4 implicit object StringSummable extends Summable[String] {  
5   def sumElements(list: List[String]): String =  
6     list.mkString(" ")  
7 }
```

## Preamble: Type classes - Example (2)

Let's try type classes! We start creating two implicit objects that implement the interface as follow:

```
1 implicit object IntSummable extends Summable[Int] {  
2   def sumElements(list: List[Int]): Int = list.sum  
3 }  
4 implicit object StringSummable extends Summable[String] {  
5   def sumElements(list: List[String]): String =  
6     list.mkString(" ")  
7 }
```

In this way we can now define a generic method that use sumElements

```
1 def sumAll[T](l: List[T])(implicit summable: Summable[T]): T  
  =  
2   summable.sumElements(l)
```

## Preamble: Type classes - Example (3)

The advantage of this approach is that we can use `sum` method only with `Int` and `String` (all the other types will create a compile error) and, thanks to `implicit`, we don't have to pass summable object to `sum`.

```
1 sumAll(List(1,2,3)) // 6
2 sumAll(List("Scala ", "is ", "awesome")) // "Scala is
   awesome"
3 sumAll(List(true, true, false)) // COMPILE ERROR
```

## Preamble: Type classes - Example (4)

Through extension method we could also use `sumAll` as method in `List` of `summable`:

```
1 implicit class ListSummable[A](l: List[A])(implicit summable
   : Summable[A]) {
2   def sumAll: A = summable.sumElements(l)
3 }
```

## Preamble: Type classes - Example (4)

Through extension method we could also use `sumAll` as method in `List` of `summable`:

```
1 implicit class ListSummable[A](l: List[A])(implicit summable
   : Summable[A]) {
2   def sumAll: A = summable.sumElements(l)
3 }
```

```
1 List(1,2,3).sumAll //6
2 List("Scala ", "is ", "awesome").sumAll //"Scala is awesome"
3 List(true, true, false).sumAll // COMPILE ERROR
```

Let's start with map function:

```
1 def fmap[A, B](fa: F[A])(f: A => B): F[B]
```

## Definition

Every class that have the map method above which respects the following laws is a functor:

- 1 **Identity:** given the identity function *id* and a functor *obj* we have that  $map(obj)(id) = obj$ .



Let's start with map function:

```
1 def fmap[A, B](fa: F[A])(f: A => B): F[B]
```

## Definition

Every class that have the map method above which respects the following laws is a functor:

- 1 **Identity:** given the identity function  $id$  and a functor  $obj$  we have that  $map(obj)(id) = obj$ .
- 2 **Composition:** given two functions  $f$  and  $g$  and a functor  $obj$  we have that  $map(map(obj)(f))(g) = map(obj)(g \circ f)$

We could test these laws through PBT ScalaCheck.

# Functors - Implementation (1)

We'll use type classes to implement functors.

Functors can be represented with the following interface

```
1 trait Functor[F[_]] {  
2   def fmap[A, B](fa: F[A])(f: A => B): F[B]  
3 }
```

## Functors - Implementation (2)

Let's extend List and Option classes to be functors.

```
1 implicit object listFunctor: Functor[List] {  
2   def fmap[A, B](fa: List[A])(f: A => B): List[B] = fa.map(f  
3   )  
4 }  
5 implicit object optionFunctor: Functor[Option] {  
6   def fmap[A, B](fa: Option[A])(f: A => B): Option[B] =  
7   fa match {  
8     case None => None  
9     case Some(x) => Some(f(x))  
10  }  
11 }
```

The first case is easy because List class already have map function (to be honestly, Option have map method too).

# Functors - Implementation (3)

As before, we can create a generic method that apply map:

```
1 def applyMap[F[_], A, B](a: F[A])(f: A => B)(implicit
    functor: Functor[F]): F[B] = {
2     functor.fmap(a)(f)
3 }
4 applyMap(List(1, 2, 3))(_ + 1) //List(2, 3, 4)
5 applyMap(Some(1) : Option[Int])(_ + 1) // Some(2)
6 applyMap(Set(true, true, false))(!_ ) // COMPILE ERROR
```

Through extension method we could also use applyMap as method in functors:

```
1 implicit class FunctorMethod[F[_], A, B](a: F[A])(implicit
    functor: Functor[F]) {
2     def applyMap(f: A => B): F[B] = functor.fmap(a)(f)
3 }
4 List(1, 2, 3).applyMap(_ + 1) //List(2, 3, 4)
5 (Some(1):Option[Int]).applyMap(_ + 1) // Some(2)
6 Set(true, true, false).applyMap(!_ ) // COMPILE ERROR
```

# Why functors are useful?

Functors allow to avoid boilerplate code. Take Option type. Normally before applying a function on Option we should check if that value is either Some or None through pattern matching:

```
1 val a : Option[Int] = Some(20)
2 def f(a : Int) : Int = if (a % 2 == 0) a / 2 else a - 1
3 val ris = (a match {
4   case None => None
5   case Some(b) => Some(f(b))
6 }) //Some(10)
```

Functors reduce code:

```
1 val a : Option[Int] = Some(10)
2 def f(a : Int) : Int = if (a % 2 == 0) a / 2 else a - 1
3 val ris = a.applyMap(f) //Some(10)
```

Again, let's start from `bind` (also called `flatMap`) and `unit` methods:

```
1 def bind[A, B](fa: M[A])(f: A => M[B]): M[B]
2 def unit[A](a : A) : M[A]
```

## Definition

Every class that have `bind` and `unit` methods above which respect the following rules is a monad:

- 1 **Left identity:** given a function  $f$  and an object  $obj$  we have that  $bind(unit(obj))(f) = f(obj)$ .

Again, let's start from `bind` (also called `flatMap`) and `unit` methods:

```
1 def bind[A, B](fa: M[A])(f: A => M[B]): M[B]
2 def unit[A](a : A) : M[A]
```

## Definition

Every class that have `bind` and `unit` methods above which respect the following rules is a monad:

- 1 **Left identity:** given a function  $f$  and an object  $obj$  we have that  $bind(unit(obj))(f) = f(obj)$ .
- 2 **Right identity:** given a monad  $obj$  we have that  $bind(obj)(unit) = obj$

Again, let's start from `bind` (also called `flatMap`) and `unit` methods:

```
1 def bind[A, B](fa: M[A])(f: A => M[B]): M[B]
2 def unit[A](a : A) : M[A]
```

## Definition

Every class that have `bind` and `unit` methods above which respect the following rules is a monad:

- 1 **Left identity:** given a function  $f$  and an object  $obj$  we have that  $bind(unit(obj))(f) = f(obj)$ .
- 2 **Right identity:** given a monad  $obj$  we have that  $bind(obj)(unit) = obj$
- 3 **Associativity:** given two functions  $f$  and  $g$  and a monad  $obj$  we have that  $bind(bind(obj)(f))(g) = bind(obj)(x \Rightarrow flatMap(f(x))(g))$

We could test these laws through PBT ScalaCheck.



# Monads - Implementation (1)

We'll use type classes to implement monads.

Monads can be represented with the following interface

```
1 trait Monad[M[_]] {  
2   def bind[A, B](fa: M[A])(f: A => M[B]): M[B]  
3   def unit[A](a : A) : M[A]  
4 }
```

# Monads - Implementation (2)

Let's extend List and Option classes to be monads.

```
1 implicit object ListMonad extends Monad[List] {  
2   override def bind[A, B](fa: List[A])(f: A => List[B]):  
3     List[B] =  
4     fa.flatMap(f)  
5 }  
6 implicit object OptionMonad extends Monad[Option] {  
7   override def bind[A, B](fa: Option[A])(f: A => Option[B]):  
8     Option[B] =  
9     fa match {  
10      case None => None  
11      case Some(a) => f(a)  
12    }  
13   override def unit[A](a: A): Option[A] = Some(a)  
14 }
```

The first case is easy because List class already has flatMap function (to be honestly, Option has flatMap method too).

# Monads - Implementation (3)

As before, we can create a generic method that apply bind:

```
1 def bind[M[_], A, B](a: M[A])(f: A => M[B])(implicit monad:
    Monad[M]): M[B] = {
2   monad.bind(a)(f)
3 }
4 bind(List(1,2,3))((x : Int) => List(x + 1)) //List(2, 3, 4)
5 bind(Some(1) : Option[Int])((x : Int) => Some(x + 1)) //
    Some(2)
6 bind(Set(true, true, false))((x : Boolean) => Set(!x)) //
    COMPILE ERROR
```

# Monads - Implementation (4)

Through extension method we could also use bind as method in monads:

```
1 implicit class MonadMethods[M[_], A, B](a: M[A])(implicit
    monad: Monad[M]) {
2     def bind(f: A => M[B]): M[B] = monad.bind(a)(f)
3     def unit : A => M[A] = monad.unit
4 }
5 List(1,2,3).bind((x : Int) => List(x + 1)) //List(2, 3, 4)
6 (Some(1) : Option[Int]).bind((x : Int) => Some(x + 1)) //
    Some(2)
7 Set(true, true, false).applyBind((x : Boolean) => Set(!x))
    // COMPILE ERROR
```

# Differences between Functors and Monads (1)

Why should we use monads instead of functor?

Let's start from a simple example: concatenation of map function.

```
1 val a : Option[Int] = Some(20)
2 val temp = a.applyMap(_ + 1) //Some(2)
3 temp.applyMap(_ + 1) // Some(3)
```

That works nicely. We start from an Option of Int, after the first step we have an Option of Int on which we apply again the same map function and we finally get an Option of Int.

## Differences between Functors and Monads (2)

But, let's try with a new function:

```
1 val a : Option[Int] = Some(20)
2 def f(x:Int) : Option[Int] = if (x == 0) None else Some(2 /
  x)
3 val temp1 = a.applyMap(f) // return Some(Some(0))
4 temp1.applyMap(f) // COMPILER ERROR
```

Here we stuck because, after the first step, we get an Option of Option of Int. We could manage this problem and adjust the second function to work with Option of Option of Int, but in this way we get an Option of Option of Option of Int and we could continue getting more and more layer of Option.

# Differences between Functors and Monads (2)

But, let's try with a new function:

```
1 val a : Option[Int] = Some(20)
2 def f(x:Int) : Option[Int] = if (x == 0) None else Some(2 /
  x)
3 val temp1 = a.applyMap(f) // return Some(Some(0))
4 temp1.applyMap(f) // COMPILE ERROR
```

Here we stuck because, after the first step, we get an Option of Option of Int. We could manage this problem and adjust the second function to work with Option of Option of Int, but in this way we get an Option of Option of Option of Int and we could continue getting more and more layer of Option. Here comes in help monads!

```
1 val temp = a.bind(f) // return Some(0)
2 temp.bind(f) //None
```

We could continue to apply bind endless, without change the return type.

## Sitography

- Rock the JVM - Type classes
- Rock the JVM - Functor
- dcapwell - Functor
- adit.io - Functor & Monad

## Source codes

- Scala sources are available at my GitHub.



# The end

Thanks!