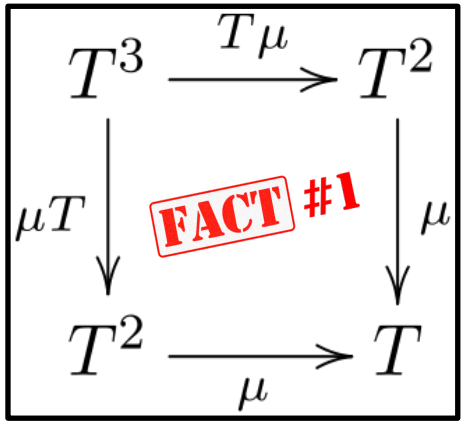
# MONAD FACT #1

Scala **for comprehensions** require a **monad** to be defined in terms of **unit**, **map** and **flatMap** rather than simply in terms of **unit** and **flatMap**





slides by  @philip_schwarz

slideshare https://www.slideshare.net/pjschwarz

@philip_schwarz

One way to define a **monad** is with the following trait

```
trait Monad[F[_]] {
  def flatMap[A,B](ma: F[A])(f: A => F[B]): F[B]
  def unit[A](a: => A): F[A]
}
```

If we define a data structure, e.g. an **Option**

```
sealed trait Option[+A]
case object None extends Option[Nothing]
case class Some[+A](get: A) extends Option[A]
```

we can then create a **monad** instance for **Option** by providing implementations for **unit** and **flatMap**

```
val optionMonad = new Monad[Option] {

  def unit[A](a: => A): Option[A] = Some(a)

  def flatMap[A, B](ma: Option[A])(f: A => Option[B]): Option[B] = {
    ma match {
      case Some(a) => f(a)
      case None => None
    }
  }
}
```

Here is an example of using the **Option monad** instance

```
def greeting(maybeGreeting: Option[String],
             maybeName:     Option[String],
             maybeSurname:  Option[String]): Option[String] =
  optionMonad.flatMap(maybeGreeting) { greeting =>
    optionMonad.flatMap(maybeName) { name =>
      optionMonad.flatMap(maybeSurname) { surname =>
        optionMonad.unit(s"$greeting $name $surname!")
      }
    }
  }
```

```
assert(
  greeting(maybeGreeting = Some("Hello"), maybeName = Some("Fred"), maybeSurname = Some("Smith"))
  == Some("Hello Fred Smith!"))

assert(
  greeting(maybeGreeting = Some("Hello"), maybeName = None, maybeSurname = Some("Smith"))
  == None)
```

We just saw how to define the **Option monad** by defining **unit** and **flatMap** functions that operate on **Option**.

Another way of defining the **Option monad** is by adding **map** and **flatMap** functions to **Option**.

@philip_schwarz

```
sealed trait Option[+A] {

  def map[B](f: A => B): Option[B] =
    this match {
      case None => None
      case Some(a) => Some(f(a))
    }

  def flatMap[B](f: A => Option[B]): Option[B] =
    this match {
      case None => None
      case Some(a) => f(a)
    }
}
case object None extends Option[Nothing]
case class Some[+A](get: A) extends Option[A]
```

If we do that, then we can take advantage of a great **Scala** feature that allows code that uses the **monad** to be **much easier to understand** than woud otherwise be the case. Here is how the **greeting** example looks like using this feature.

```
def greeting(maybeGreeting: Option[String],
             maybeName:     Option[String],
             maybeSurname:  Option[String]): Option[String] =
  for {
    greeting <- maybeGreeting
    name     <- maybeName
    surname  <- maybeSurname
  } yield s"$greeting $name $surname!"
```

```
assert(greeting(maybeGreeting = Some("Hello"), maybeName = Some("Fred"), maybeSurname = Some("Smith"))
       == Some("Hello Fred Smith!"))
assert(greeting(maybeGreeting = Some("Hello"), maybeName = None, maybeSurname = Some("Smith"))
       == None)
```

The feature in question is the ability to write an easy-to-understand **for comprehension**, whose **syntactic sugar** is translated by the **Scala** compiler into a harder-to-understand **desugared** chain of calls to **map** and **flatMap**

```
for {
    greeting <- maybeGreeting
    name     <- maybeName
    surname  <- maybeSurname
} yield s"$greeting $name $surname!”
```

desugars to

```
maybeGreeting flatMap { greeting =>
  maybeName flatMap { name =>
    maybeSurname map { surname =>
      s"$greeting $name $surname!"
    }
  }
}
```

@philip_schwarz

On the first slide we defined a **monad** simply in terms of **unit** and **flatMap**.

Later on, in order to take advantage of **for comprehensions**, we redefined a **monad** in terms of **unit**, **map** and **flatMap**. “We did not define a **unit** function!”, I hear you say. Well, although it is not called **unit**, we did define it: it is called **Some**, because creating a ‘defined’ **Option**, e.g. one with a value of 5, is done by evaluating **Some**(5) (to get hold of the ‘undefined’ **Option** we use **None**).

So why is it that in the first approach we can simply define a **monad** in terms of **unit** and **flatMap**

```
def flatMap[A,B](ma: F[A])(f: A => F[B]): F[B]
def unit[A](a: => A): F[A]
```

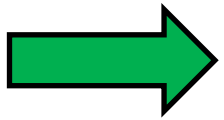
whereas in the second approach we have to define a **monad** in terms of **unit**, **map** and **flatMap**?

```
def map[B](f: A => B): Option[B] =
def flatMap[B](f: A => Option[B]): Option[B] =
case class Some[+A](get: A) extends Option[A] // Some acts as unit function
```

The answer to that question is that is that in **Scala** there is no **Monad** trait that **monads** can implement and that the compiler is aware of, so the compiler does not know what the **unit** function of a **monad** is because for one thing, the function can have an arbitrary name. In this case the **unit** function is called **Some**, in the case of the **Either monad** it is called **Right**, in the case of the **List monad** it is called **List**, etc.

If there were some convention by which the compiler could figure out what the **unit** function of a **monad** is, then rather than desugaring a **for comprehension** as follows:

```
for {
    greeting <- maybeGreeting
    name     <- maybeName
    surname  <- maybeSurname
} yield s"$greeting $name $surname!”
```



desugars to

```
maybeGreeting flatMap { greeting =>
    maybeName flatMap { name =>
      maybeSurname map { surname =>
        s"$greeting $name $surname!"
      }
    }
  }
```

It could desugar it to something like this:

```
maybeGreeting flatMap { greeting =>
    maybeName flatMap { name =>
      maybeSurname flatMap { surname =>
        Some(s"$greeting $name $surname!")
      }
    }
  }
```

```
maybeGreeting flatMap { greeting =>
    maybeName flatMap { name =>
      maybeSurname flatMap { surname =>
        monadInstance.unit(s"$greeting $name $surname!")
      }
    }
  }
```

See the following for the list of all available slide decks in the **MONAD FACT** series

slideshare https://www.slideshare.net/pjschwarz/the-monad-fact-slide-deck-series