ToiletPaper #157

jambit
WHERE INNOVATION WORKS

For-comprehensions and monads in Scala

Author: Aleksandar Stoimenov

✕ Problem

At first glance, Scala for-comprehensions look like common for-loops.
In the following example we want to create an address, but only if *street*, *number* and *zipCode* are available.

```
val maybeStreet: Option[String] = ???
val maybeNumber: Option[Int] = ???
val maybeZipCode: Option[Int] = ???
case class Address(street: String, number: Int, zipCode: Int)
```

The *Option[A]* type in Scala has two forms, *Some[A]* and *None*.
We could use *flatMap* and *map* in order to implement it.

```
val maybeAddress: Option[Address] = maybeStreet.flatMap(street =>
    maybeNumber.flatMap(number =>
        maybeZipCode.map(zipCode => Address(street, number, zipCode))))
```

✓ Solution

We could alternatively solve this with a for-comprehension.

```
val address: Option[Address] = for {
    street <- maybeStreet
    number <- maybeNumber
    zipCode <- maybeZipCode
  } yield Address(street, number, zipCode)
```

The second implementation is more compact and readable.
Only when all *street* and *number* and *zipCode* are instances of *Some*, we get *Some[Address]* as result. Otherwise, the result is *None*.

✚ Further Aspects

In such a nested *for-comprehension* with yield, any data type that implements the functions *map* and *flatMap* can be used. For example, all monads would be possible. If you are wondering now what a monad is: The answer to this question would go beyond the scope of this ToiletPaper.

But one thing is for sure: With *for-comprehensions* and monads, operations can be chained, which are to be executed one after the other. While doing so, an operation can use the output of previous operations as input. The concrete implementation of the monad defines what exactly happens at the boundary between the operations.

The behavior of a *for-comprehension* depends on which monad is used. Other commonly used monads in Scala include *List[A]*, *Either[A, B]*, *Try[A*], and *Future[A]*.

For lists, the behavior is similar to iteration.

```
val result: List[(Int, String)] = for {
  a <- List(1, 2, 3)
  b <- List("A", "B", "C")
  } yield (a, b)

// Result: List((1, "A"), (1, "B"), "....
```