ToiletPaper #157

For-Comprehensions und Monaden in Scala

Autor: Aleksandar Stoimenov

✕ Problem

Auf den ersten Blick sehen Scala *For-Comprehensions* wie übliche *For-Schleifen* aus.
In folgendem Beispiel wollen wir eine *Address* erstellen, jedoch nur wenn sowohl *Street* als auch *Number* und *ZipCode* vorhanden sind.

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

Der Typ *Option[A]* in Scala hat zwei Ausprägungen – *Some[A]* und *None*.
Wir könnten *flatMap* und *map* nutzen, um das zu implementieren.

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

✓ Lösung

Alternativ könnten wir auch eine *For-Comprehension* nutzen.

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

Die zweite Implementierung ist kompakter und lesbarer.
Nur wenn alle *street* und *number* und *zipCode* Instanzen von *Some* sind, bekommen wir *Some[Address]* als Ergebnis. Ansonsten ist das Ergebnis *None*.

✚ Weiterführende Aspekte

In einer solch verschachtelten *For-Comprehension* mit yield kann jeder Datentyp verwendet werden, der die Funktionen *map* und *flatMap* implementiert. Dies tun beispielsweise alle Monaden. Wer sich an dieser Stelle die Frage stellt, was eine Monade ist: Die Beantwortung dieser Frage würde das ToiletPaper sprengen.

Sicher ist jedoch: Mit *For-Comprehensions* und Monaden können Operationen verkettet werden, die hintereinander ausgeführt werden sollen – wobei eine Operation als Input den Output von vorherigen Operationen benutzen kann. Die konkrete Implementierung der Monade definiert, was an der Grenze zwischen den Operationen genau passiert.

Das Verhalten einer *For-Comprehension* hängt davon ab, welche Monade benutzt wird. Andere häufig benutzte Monaden in Scala sind z. B. List[A], Either[A, B], Try[A] und Future[A].

Bei Listen ist das Verhalten ähnlich wie beim Iterieren.

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

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