Scala is niet moeilijk

Scala maakt Java development eenvoudiger

Zes jaar geleden kwam ik met Scala in aanraking toen een klant besloot om over te gaan van Java naar Scala. Sindsdien ben ik aan deze taal verknocht. Enerzijds vanwege de veel leesbaardere code aangezien veel boilerplate code weggelaten kan worden. Anderzijds biedt Scala ook veel mogelijkheden die Java niet biedt aangezien Scala zowel OO als FP concepten verenigd. Veel Java ontwikkelaars lijken een vooringenomen standpunt te hebben als het gaat om Scala, namelijk 'Scala is moeilijk en langzaam'. Hierdoor lijkt nu in Nederland de gekke situatie te ontstaan dat klanten Scala willen gebruiken maar niet de ontwikkelaars kunnen vinden, omdat Java ontwikkelaars geen Scala willen leren. Met de huidige snelheid van de technologische ontwikkelingen kan je niet in je comfort zone blijven zitten. Je moet verder kijken naar hoe je je klant het beste kan helpen om zijn doelen te realiseren.

Ontwikkeling Java

Java heeft zich de afgelopen tien jaar maar zeer langzaam ontwikkeld. En dat in een decennium waarin op technologisch gebied waanzinnig veel gebeurd is. Twintig jaar geleden had niemand nog gedacht aan 'deployen in de cloud', non-blocking microservices, concurrency en multi-core processoren. Java biedt ontwikkelaars niet de tools en libraries om ons daarmee te helpen.

Scala biedt wél features om je daarmee te helpen. Vanwege de uitstekende compatibiliteit met Java, zou je Scala ook niet als een andere taal moeten zien, maar als een futuristische Java versie waarmee je nu al aan de slag kunt en waarmee je, met een vergelijkbare syntax en met behoud van bestaande code, wel de applicaties kan ontwikkelen die op dit moment gevraagd worden.

Scala wordt vaak als 'moeilijk' en 'complex' gezien, omdat heel veel mogelijk is met de taal. Ook wordt het als moeilijk gezien, omdat er, vanwege de Functional Programming basis, veel met complexe wiskundige termen wordt gestrooid. Daar komt bij dat de meeste cursussen zich in het begin meer richtten op de FP en wiskundige aspecten in plaats van de Java/OO developer.

Te moeilijk?

Echter, Scala hoeft helemaal niet zo ingewikkeld te zijn. Scala's language specification is bijvoorbeeld een stuk kleiner dan die van Java, terwijl toch alle Java features ondersteund worden (en meer). Door alleen al gebruik te maken van een paar Scala features, kan je als Java developer een stuk productiever zijn. Rod Johnson heeft in een keynote op ScalaDays in 2013 al eens gezegd dat er twee groepen Scala developers zullen zijn, waar ik me wel in kan vinden. Zo heb je de groep die Scala zeer pragmatisch gebruiken om op een Java manier (enterprise) applicaties te bouwen in minder code, meer leesbare code en minder bugs. Daarnaast heb je de andere groep die helemaal los gaat met Scala om fantastische generieke en typesafe libraries te maken voor de eerste groep.

Ik behoor tot de eerste groep en op mijn huidige project gebruikten we de afgelopen 2,5 jaar Scala ook op deze manier. Toch vind ik het ook leuk om te zien en te leren hoe Scala door die andere, tweede groep meer functioneel gebruikt wordt. Deze kennis komt ook terug in de applicatiecode, wanneer we denken dat dit de code nog simpeler en leesbaarder kan maken. Natuurlijk is het mogelijk om onleesbare Scala code te schrijven, maar goede Scala code is beter te lezen dan Java code en het is de verantwoordelijkheid van het team om door middel van reviews te bewaken dat de Scala code ook leesbaar en begrijpelijk blijft.

Handige features

Voorbeelden zeggen meer dan duizend argumenten. Hieronder vind je een greep uit de vele Scala features die je helpen betere code te schrijven:

REPL en Worksheet: Met de Scala REPL en worksheets in een IDE is het mogelijk om snel wat uit te proberen. En hier kan je natuurlijk ook gewoon Java in gebruiken. Na het installeren van Scala start je de REPL met *scala* op de command-line. De REPL is gebruikt voor alle onderstaande voorbeelden.

Joost den Boer is een senior software developer, ZZP-er en Scala fan. Hij organiseert Brabant-Scala en '073 developer beer chat' meet-ups. Ook deelt hij Scala kennis en ervaring via conferenties, workshops en cursussen.

Case classes: Reduceer een POJO tot een enkele expressie met behulp van een *case class*. Methodes als *toString* en *copy* alsmede een correcte *equals* en *hashCode* implementatie krijg je cadeau. Met *case class* reduceer je je Hibernate entiteit of DTO tot een paar regels (zie **Listing 1**).

```
> case class User(firstName: String, lastName: String, age: Int)
defined class User
```

Listing 1

Named argument: Bij het aanroepen van een methode kan je de naam van het argument gebruiken waardoor het in code duidelijker is waaraan je een waarde toekent (zie **Listing 2**).

```
> new User(lastName = "Klaassen", firstName = "Jan", age = 22)
res1: User = User(Jan, Klaassen, 22)
```

Listing 2

Default argument: Een default argument krijgt de default waarde als het argument niet ingevuld wordt in de aanroep. Het argument wordt dus optioneel. Hierdoor is het niet meer nodig allerlei overloaded methodes te maken, wat de code vereenvoudigd (zie **Listing 3**).

```
> def add(x: Int, y: Int = 2) = x + y
add: (x: Int, y: Int)Int
> add(1)
res2: Int = 3
```

Listing 3

Type inference: Betere type inference waardoor het type vaak weggelaten kan worden, zoals bij variabelen of methode return type (zie **Listing 4**).

```
> val user = User("Jan", "Klaassen", 31)
user: User = User(Jan,Klaassen,31)
```

Listing 4

Geen null's, NullPointerException's en checked exceptions: Door gebruik te maken van return-types die meerdere states kunnen aannemen, is het veel duidelijker voor gebruikers van een methode wat de mogelijke resultaten kunnen zijn. Er is geen reden meer om *null* te gebruiken. Scala biedt hiervoor diverse classes zoals Option, Try, Either, Future.

Immutability: Default is alles immutable: de Collection API, case classes, variabelen gedefinieerd met het *val* keyword.

```
> s"User name ${user.firstName}"
res4: String = User name Jan
```

Listing 5

String interpolatie: String interpolatie biedt een veel simpelere manier van string formatteren waarbij de geformatteerde string duidelijk leesbaar blijft (zie **Listing 5**).

```
> val json = s"""{ "name": "Jan", "age": ${user.age} }"""
json: String = { "name": "Jan", "age": 31 }
```

Listing 6

Raw strings: Met 'Raw' strings hoeft er niets meer ge-escaped te worden. Zeer handig voor regular expressies of json content, maar ook voor strings die over meerdere regels gaan en het kan gecombineerd worden met string interpolatie (zie **Listing 6**).

```
> val timeRegex = """(\d+):(\d+)""".r
regex: scala.util.matching.Regex = (\d+):(\d+)
```

Listing 7

Reguliere expressies: worden een stuk eenvoudiger met raw strings en pattern matching. De toevoeging '.r' maakt van een string een Regex object (zie **Listing 7**).

```
> def switch[A](a: A) = a match {
     | case User(firstName, lastName, age) => s"User with name $firstName $lastName"
     | case timeRegex(hour, minute) => s"$hour hour $minute"
     | case other => s"Unknown $other"
     | }
switch: [A](a: A)String

> switch(user)
res5: String = User with name Jan Klaassen
> switch("10:14")
res6: String = 10 hour 14
> switch(10)
res7: String = Unknown 10
```

Listing 8

BIJ SCALA BLIJFT HET MOGELIJK OM JE VERDER TE VERDIEPEN IN DE TAAL

```
> val adultFirstNames = users.filter(_.age > 18).map(_.firstName)
adultFirstNames: List[String] = List()
```

Listing 9

```
> object UserPrinter {
     | def print(user: User) = println(s"User ${user.firstName}")
     | }
defined object UserPrinter

> UserPrinter.print(user)
User Jan
```

Listing 10

```
> def print(b: Boolean, one: => Unit, two: => Unit) = if (b) one else two
print: (b: Boolean, one: => Unit, two: => Unit)Unit
> print(true, println("one"), println("two"))
```

Listing 11

```
> type QueryResult[A] = Future[Or[A, String]]
defined type alias QueryResult
> def find(id: Int): QueryResult[User] = ???
find: (id: Int)QueryResult[User]
```

Listing 12

```
> def returnsTuple() = (1, "some string", true)
returnsTuple: (Int, String, Boolean) = (1, some string, true)
```

Listing 13

```
> val (i, s, b) = returnsTuple
i: Int = 1
s: String = some string
b: Boolean = true
```

Listing 14

Pattern matching: Scala's *match* is een geavanceerde *switch* expressie waar ook objecten en reguliere expressies in gebruikt kunnen worden. Deze kunnen direct 'uitgepakt' worden, waardoor de values daarin direct bruikbaar zijn (zie **Listing 8**).

Lambdas en Collection API: Sinds Java 8 hebben Java developers al kennis kunnen maken met streams en lambdas. Scala's support hiervoor in de Collection API maakt dit een stuk eenvoudiger. Behalve dat geen *stream()* en *collect(..)* nodig is, is de Collection API veel uitgebreider waardoor je bepaalde functies niet zelf telkens opnieuw hoeft te implementeren (zie **Listing 9**).

Singleton: Native support voor echte singleton objecten met het object keyword. Elke *case class* krijgt automatisch ook een eigen object welke het *companion object* genoemd wordt. Via dit *companion object* kan je een instantie van een class maken zonder het *new* keyword te gebruiken (zie **Listing 10**).

Call-by-name: Een call-by-name argument biedt de mogelijkheid om een expressie door te geven zonder dat deze direct geëvalueerd wordt (zie **Listing 11**).

Type alias: geeft de mogelijkheid om een alias te maken van een complex type waardoor code beter te begrijpen wordt (zie **Listing 12**).

Tuples: Soms wil je gewoon even wat values bij elkaar voegen zonder dat je daar direct een class voor wilt maken. Bijvoorbeeld als return value van een methode (zie **Listing 13**).

Meestal assign je de Tuple value niet aan een variabele, maar splits je de Tuple direct weer naar de individuele waardes door middel van pattern matching (zie **Listing 14**).

Build tools: Scala heeft met Sbt zijn eigen build tool, maar Maven en Gradle kunnen ook gebruikt worden. Zowel Sbt als Gradle hebben het voordeel ten opzichte van Maven dat je gemakkelijk de build kan aanpassen voor je eigen proces zonder daar een plug-in voor te moeten schrijven.

Scala heeft nog veel meer features, zoals een beter generics type systeem, value classes, extension methods, collections factory methodes, currying, methodes in methodes, etc. En zoals je ziet, hoef je helemaal geen Functional Programming of categorietheorie te kennen om deze features te kunnen gebruiken.

Tot slot

Scala bestaat al ruim twaalf jaar en er is inmiddels een schat aan bronnen beschikbaar om Scala te leren. Naast boeken zijn er veel online resources. De 'Twitter Scala School' is nog steeds één van m'n favorieten. Scala-Exercises is vrij nieuw om Scala of een library te leren door middel van kleine opgaven. Of volg een online of klassikale cursus zoals bij Vijfhart. Bezoek de 'Amsterdam Scala' of 'Brabant Scala' meet-up groep of één van de vele Scala conferenties, vooral in Europa. En natuurlijk kom je Scala ook tegen op Java-, Reactive- en Functional Programming conferenties.

Rod Johnson zei in 2013 al: "Scala is my most loved language since C", dus trek je stoute schoenen aan en probeer Scala een keer uit! ■

REFERENTIES

Scala Exercises: **https://www.scala-exercises.org**
Twitter Scala School: **https://twitter.github.io/scala_school/**
Scala Center: **https://scala.epfl.ch**
Functional programming in Scala: **https://www.coursera.org/specializations/scala**
Scala cursus: **https://www.vijfhart.nl/opleidingen/scala-essentials/**
Rod Johnson keynote ScalaDays 2013: **https://www.youtube.com/watch?v=DBu6zmrZ_50**