Ausgabe 03/2020

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

D: 4,90 EUR | CH: 9,80 CHF | A: 5,60 EUR | Benelux: 5,80 EUR

4 191978 304903 03

Java aktuell

iJUG Verbund
www.ijug.eu

Progressive Web Apps
PWAs und ihre Vorteile in der Praxis

Flutter
Von der Entscheidung bis zur Entwicklung

Geteilte Hologramme
Shared Augmented Reality im Überblick

DIE ZUKUNFT IST MOBILE

adesso business. people. technology.

Von Anfang an Teil des Java-Teams!

Entwickelt bereits kluge IT-Lösungen bei adesso:
Ihr neuer Kollege Kristof Hierath | Software Engineer

SOFTWARE DEVELOPMENT@adesso

Sie wollen dort einsteigen, wo Zukunft programmiert wird? Dann sind Sie mit einem Start in unserem Java-Team bei adesso genau richtig. Gemeinsam setzen wir herausfordernde Projekte für unsere Kunden um. Dafür brauchen wir Menschen, die Lust haben, ihr Wissen, ihre Talente und ihre Fähigkeiten einzubringen.

Planen und realisieren Sie in interdisziplinären Projektteams anspruchsvolle Anwendungen und Unternehmensportale auf Basis von Java/JavaScript-basierten Technologien als

→ **(Senior) Software Engineer (w/m/d) Java**
→ **Software Architekt (w/m/d) Java**
→ **(Technischer) Projektleiter Softwareentwicklung (w/m/d) Java**

CHANCENGEBER – WAS ADESSO AUSMACHT

Kontinuierlicher Austausch, Teamgeist und ein respektvoller, anerkennender Umgang sorgen für ein Arbeitsklima, das verbindet. So belegen wir nach 2016 auch 2018 den 1. Platz beim Wettbewerb „Deutschlands Beste Arbeitgeber in der ITK"!

Mehr als 650 Software Engineers Java bei adesso, über 120 Schulungen und Weiterbildungen – zum Beispiel in Angular2 oder Spring Boot – sowie ein Laptop und ein Smartphone ab dem ersten Tag warten auf Sie!

GREAT PLACE TO WORK® 2018 Beste Arbeitgeber™ ITK

IHRE BENEFITS – WIR HABEN EINE MENGE ZU BIETEN:

- Welcome Days
- Choose your own Device
- Weiterbildung
- Events: fachlich und mit Spaß
- Sportförderung
- Mitarbeiterprämien
- Auszeitprogramm

Es wird Ihnen bei uns gefallen! Mehr Informationen auf **www.karriere.adesso.de**.
Olivia Slotta aus dem Recruiting-Team freut sich auf Ihre Kontaktaufnahme:
adesso SE | Leona Demiri | T +49 231 7000-7100 | jobs@adesso.de

Funktionale Programmierung in Kotlin

Stefan López Romero, MaibornWolff GmbH

Dieser Artikel gibt eine Einführung in die funktionale Programmierung mit Kotlin. Wir lernen die Grundbausteine und Mechanismen dieses Programmierparadigmas kennen. Unter anderem sehen wir, wie man Funktionen höherer Ordnung programmiert, diese kombiniert und „curryfiziert". Kotlin ist dabei unser Vehikel, um leichtgewichtig in die funktionale Programmierung einzusteigen. Die Sprache bietet im Vergleich zu Java echte Funktionstypen sowie viele syntaktische Verbesserungen und Vereinfachungen.

Was ist funktionale Programmierung?

Funktionale Programmierung ist ein deklaratives Programmierparadigma, das ausschließlich sogenannte pure Functions verwendet. Um zu verstehen, was eine *pure Function* ist, benötigen wir den Begriff der *referenziellen Transparenz*:

Ein Ausdruck ist referenziell transparent, wenn er durch seinen Rückgabewert ersetzt werden kann, ohne das Verhalten des Programms zu verändern [1].

Damit können wir nun formal definieren, was wir unter einer pure Function verstehen:

Eine Funktion f ist pur, wenn der Ausdruck f(x) für alle referenziell transparenten x referenziell transparent ist [1].

Diese etwas abstrakte Definition besagt, dass eine Funktion pur ist, wenn sowohl der Parameter x als auch die Funktionsauswertung f(x) durch ihr Ergebnis ersetzt werden können, ohne dass dies das Verhalten des Programms ändert. In *Listing 1* sehen wir ein Beispiel dafür.

Die Funktion `plus` ist pur, denn man kann `a.plus(2)` durch das Ergebnis `4` ersetzen, ohne dass dies etwas am weiteren Ablauf des Programmes ändert. In *Listing 2* sehen wir dagegen eine Funktion, die diese Eigenschaft nicht erfüllt.

Würden wir hier die zweite Zeile durch ihr Ergebnis „`Tic Tac`" ersetzten, wäre `c` nicht mehr „`Tic Tac Toe`", sondern „`Tic Toe`". Dies liegt daran, dass die `append`-Funktion einen Seiteneffekt hat. Neben der Rückgabe eines zusammengesetzten Strings wird zusätzlich der Zustand der Instanz verändert. Seiteneffekte sorgen dafür, dass Code schwerer zu verstehen und somit auch fehleranfälliger ist. Wir können nicht wie in *Listing 1* nur eine Zeile betrachten, um das Ergebnis einer Funktion zu kennen, sondern müssen alle Zustandsänderungen, die auf der Instanz passieren, im Auge behalten. Im Beispiel ist dies noch überschaubar. Liegen die besagten Zustandsänderungen jedoch weit auseinander oder sind sie auf verschiedene Funktionen verteilt, ist es oft nur noch schwer möglich, den Überblick zu behalten.

Hier punktet die funktionale Programmierung. Aufgrund der referenziellen Transparenz sind in pure Functions keine Seiteneffekte erlaubt. Funktionaler Code ist dadurch leichter zu verstehen, zu testen und zu debuggen. Zudem führt die Seiteneffektfreiheit zu aussagekräftigeren Funktionssignaturen: Eine Funktion nimmt einen oder mehrere Parameter entgegen, führt einen Algorithmus aus und gibt immer ein Ergebnis zurück. Daneben kann und darf nichts passieren. Aufgrund dieses Aspekts kann man funktionale Programmierung auch als Programmierung ohne Seiteneffekte bezeichnen.

```
val a = 2
val b = a.plus(2)  //=4
val c = a.plus(3)  //=5
```

Listing 1: Pure Function

```
val a = StringBuilder("Tic")
val b = a.append(" Tac").toString() //="Tic Tac"
val c = a.append(" Toe").toString() //="Tic Tac Toe"
```

Listing 2: Impure Function

Funktionen und Datentypen als Bausteine

Unsere Bausteine, um Programme zu erstellen, sind neben pure Functions (im weiteren Verlauf einfach als „Funktionen" bezeichnet) Datentypen. Ein Datentyp in der funktionalen Programmierung ist ein Name, der einer Menge möglicher Werte gegeben wird. Dieser Datentyp kann als Ein- und Ausgabe einer Funktion verwendet werden. Funktionen agieren dabei als Transformatoren zwischen den Datentypen. Sie wandeln Informationen von einem oder mehreren Eingabetypen in einen Ausgabetyp. Datentypen können in Kotlin am einfachsten mithilfe von Data Classes erstellt werden. In *Listing 3* sehen wir, wie eine Data Class `Name` definiert wird, die aus den beiden Strings `firstName` und `lastName` zusammengesetzt ist.

Funktionen als Werte

In funktionalen Sprachen sind Funktionen sogenannte *first-class citizens*. Das bedeutet, dass Funktionen (wie auch Werten) Variablen zugewiesen, als Argumente übergeben oder von einer anderen Funktion zurückgegeben werden können. Da Kotlin statisch typisiert ist, müssen auch Funktionen einen Typ haben. In *Listing 3* sehen wir ein Beispiel für die Deklaration eines Funktionstyps. Der Bezeichner `sayHello` ist vom Typ `(Name) -> String`. Dies bedeutet, dass sich hinter dem Namen `sayHello` eine Funktion verbirgt, die ein Argument des Datentyps `Name` entgegennimmt und einen `String` zurückgibt. Nach der Deklaration wird der Funktionstyp im Beispiel durch einen Lambda-Ausdruck instanziiert. Ein Lambda-Ausdruck ist eine Kurzschreibweise für eine anonyme Funktion *[2]* und immer von geschweiften Klammern umgeben. Innerhalb der Klammern folgen die Parameter-Deklarationen, die optional mit Typangaben versehen werden können. Der Body der Funktion steht nach dem Pfeil. Das Ergebnis des letzten Ausdrucks innerhalb des Lambda-Body ist zugleich der Rückgabewert. Eine so definierte Funktion nennen wir aufgrund ihrer Zuweisung zu einem Wert *value Function*.

Die Instanziierung durch einen Lambda-Ausdruck ist nur eine Möglichkeit der Funktionsdefinition. In *Listing 4* sehen wir, wie das auch durch eine anonyme Funktion sayHelloAno möglich ist.

```
data class Name(val firstName: String, val lastName: String)

val sayHello: (Name) -> String = {
    name -> "Hello, ${name.firstName}  ${name.lastName}"
}
```

Listing 3: Datentypen und Funktionen

Neben den bisher kennengelernten value Functions gibt es noch eine zweite Schreibweise für Funktionen in Kotlin. Wie wir anhand von `sayHelloFun` sehen, können Funktionen auch mit dem Schlüsselwort `fun` eingeleitet werden. Danach folgen der Bezeichner, die Argumente und am Ende der Rückgabewert. Der Body der Funktion steht nach dem Gleichheitszeichen. Diese Art von Funktionen bezeichnen wir als `fun` Functions. Hier stellt sich die Frage: Wieso gibt es überhaupt zwei Schreibweisen für Funktionen? `fun` Functions sind effizienter und können mit generischen Typen (Generics) umgehen. Deshalb sollte man für Funktionen, die nicht als Wert übergeben werden, diese Schreibweise verwenden. `fun` Functions können durch Funktionsreferenzen wie in `sayHelloRef` zu value Functions konvertiert werden.

Funktionskomposition

Funktionen und Datentypen sind also unsere Bausteine für funktionale Programmierung. Bleibt noch die Frage: Wie können wir diese Bausteine miteinander verbinden, sodass aus einzelnen kleinen, wiederverwendbaren Funktionen ein Programm wird, das eine bestimmte Aufgabe erfüllt? Durch das Fehlen von Seiteneffekten können Funktionen einzig und allein durch eine Verkettung von Funktionsaufrufen miteinander interagieren. In *Listing 5* sehen wir ein Beispiel dafür. Die Funktion `parseInt` liest einen String ein und wandelt diesen in einen Integer-Wert um. Die Funktion `multBy4` multipliziert einen gegebenen Integer-Wert mit vier. Wollen wir in der `main`-Funktion eine als String angegebene Zahl mit vier multiplizieren, so verketten wir beide Funktionen, sodass der Rückgabewert der einen Funktion als Eingabe für die nächste Funktion fungiert.

Wie bereits erwähnt können Funktionen auch andere Funktionen als deren Parameter oder Rückgabewert verwenden. Solche Funktionen werden Funktionen höherer Ordnung genannt. Machen wir uns diese Eigenschaft zunutze, so können wir die Verkettung von Funktionen abstrahieren und so wiederverwendbar machen. Der Mechanismus hierfür wird Funktionskomposition genannt und ist folgendermaßen definiert:

Seien A, B, C beliebige Mengen und g: A → B sowie f: B → C Funktionen, so heißt die Funktion ***f ° g: A → C , (f ° g)(x) = f(g(x))*** *Komposition von f und g.*

Auf unser Beispiel übertragen bedeutet dies: Wir können eine Funktion `compose` definieren, die zwei Funktionen `f` und `g` mit `f:(Int)-> Int` und `g: (String)-> Int` entgegennimmt und eine Funktion `(String)-> Int` zurückgibt. *Listing 6* zeigt die Implementierung dieser Funktion. Im Body der `compose`-Funktion sehen wir, wie die beiden gegebenen Funktionen per `f(g(x))` miteinander verkettet werden. Durch diese Abstraktion können wir jetzt ganz einfach eine neue Funktion `parseAndMult` durch Kombination der zwei vorhin definierten Funktionen erstellen *(siehe Listing 6)*.

Leider ist unsere `compose`-Funktion auf bestimmte Datentypen beschränkt. Aber auch dies können wir abstrahieren. In *Listing 7* sehen wir eine durch generische Typen noch allgemeinere Version dieser Funktion. Wie bereits erwähnt ist die Verwendung von Generics nur mit `fun` Functions möglich, deshalb wählen wir jetzt diese Schreibweise. Da wir keinerlei Angaben zu den generischen Typen `A`, `B` und `C` machen, sind sie durch jegliche Typen ersetzbar. Funktionen höherer Ordnung, die mit generischen Typen arbeiten, werden polymorphe Funktionen höherer Ordnung genannt. Am Aufruf der `compose`-Funktion ändert sich trotz der generischen Typen nichts.

```
val sayHelloAno: (Name) -> String = fun(name:Name) =
    "Hello ${name.firstName}  ${name.lastName}"

fun sayHelloFun(name: Name) : String =
    "Hello ${name.firstName}  ${name.lastName}"

val sayHelloRef: (Name) -> String = ::sayHelloFun
```

Listing 4: Weitere Möglichkeiten der Funktionsdefinition

```
val parseInt: (String) -> Int = {
    s -> s.toInt()
}
val multBy4: (Int) -> Int = {
        x -> x * 4
}
fun main() {
    val result = multBy4(parseInt("2"))
}
```

Listing 5: Funktionsverkettung

```
val compose: (f:(Int) -> Int, g:(String) -> Int) -> (String) -> Int = {
        f, g -> {
            x -> f(g(x))
    }
}
fun main() {
    val parseAndMult = compose(multBy4, parseInt)
    val result = parseAndMult("4")

}
```

Listing 6: Funktionskomposition

```
fun <A, B, C> compose(f: (B) -> C, g: (A) -> B): (A) -> C = {
        x -> f(g(x))
}
```

Listing 7: Polymorphe Funktion höherer Ordnung

```
fun <A, B, C> curry(f:(A, B)-> C) : (A) -> (B) -> C = {
    a -> {
        b -> f(a, b)
    }
}
```

Listing 8: Currying

```
fun calcTax(rate: Double, value: Double): Double {
    return value * rate
}

fun main() {
    val calcVat = curry(::calcTax)(0.19)
    val vat = calcVat(149.99)
}
```

Listing 9: Partielle Funktionsanwendung

Anhand der Signatur der zu komponierenden Funktionen kann Kotlins Typ-Inferenz den Funktionstyp `(String) -> Int` als Rückgabetyp selbstständig ermitteln.

Currying

In vielen funktionalen Sprachen kann eine Funktion nur ein Argument verarbeiten. Dies ist jedoch kein Nachteil, da man mithilfe eines Verfahrens namens Currying aus jeder Funktion mit mehreren Parametern eine gleichwertige Sequenz von Funktionen mit einem Parameter machen kann. Der Name dieses Verfahrens geht auf den Mathematiker Haskell Brooks Curry *[3]* zurück. Als Beispiel nehmen wir eine Funktion, die zwei Argumente annimmt – eines vom Typ A und eines vom Typ B – und ein Ergebnis vom Typ C erzeugt. Durch Currying wird diese in eine Funktion übersetzt, die ein einziges Argument vom Typ A annimmt und als Ergebnis eine Funktion von B nach C erzeugt. Wir konsumieren also immer nur ein Argument und geben Funktionen zurück, die die weiteren Argumente verarbeiten. Wie man dieses Verfahren in Kotlin-Code übersetzt, sehen wir in *Listing 8*. Die Funktion `curry` wandelt jede zweistellige Funktion in eine Funktion in Curry-Schreibweise um.

Diese Umformung einer Funktion bietet uns mehr Flexibilität. Es ist nun nicht mehr zwingend notwendig, dass wir beide Argumente der Funktion sofort angeben, sondern wir können erst mal ein Argument belegen und die Angabe des zweiten auf später verschieben. Dies können wir uns zum Beispiel in Anwendungsfällen zunutze machen, bei denen ein Parameter eine Konstante ist. In *Listing 9* sehen wir ein Beispiel dafür: Die Funktion `calcTax` berechnet den Steuerbetrag zu den in den Parametern angegebenen Steuersatz und Wert. Wir wollen diese Funktion zur Berechnung der Mehrwertsteuer verwenden, die in unserem Anwendungsfall konstant 19 Prozent ist. Mithilfe unserer `curry`-Funktion wandeln wir die `calcTax`-Funktion in die Curry-Schreibweise um. Hierdurch können wir den Steuersatz fest auf 0.19 setzen und erhalten eine neue Funktion, die nur noch den Wert als Parameter erwartet. Diesen Mechanismus, bei dem nicht alle Parameter einer Funktion sofort belegt werden, nennt man partielle Funktionsanwendung.

Zusammenfassung und Ausblick

Funktionen als *first-class citizens* bieten uns sehr viel Spielraum in der Software-Entwicklung. Wir können Abstraktionen für wiederkehrende Probleme erstellen, ohne hierfür komplexe Klassenhierarchien aufbauen zu müssen. Durch die Seiteneffektfreiheit wirkt eine Funktion nur lokal und kann keinen äußeren Zustand verändern, was das Testen und die Fehlersuche sehr vereinfacht. Jedoch ist jede Ein- oder Ausgabe, etwa das Lesen von der Konsole oder Schreiben in eine Datei, auch ein Seiteneffekt und somit nicht erlaubt. Hier fragt sich der eine oder die andere bestimmt, wie man mit dieser Einschränkung in der Praxis umgehen kann: In der funktionalen Programmierung geht es nicht darum, alle Seiteneffekte zu verbieten. Vielmehr will man bewusst mit diesen umgehen und funktionalen von nicht-funktionalem Code trennen. Inhaltlich zeige ich in diesem Artikel nur den grundlegendsten Teil der funktionalen Programmierung. Jedem, der tiefer in dieses Thema mit Kotlin einsteigen will, sei das Buch Functional Programming in Kotlin *[4]* sowie die Library Arrow *[5]* empfohlen.

Quellen

[1] Paul Chiusano, Rúnar Bjarnason (2015): *Functional Programming in Scala*. Mannning Publications Co, Shelter Island NY

[2] *https://en.wikipedia.org/wiki/Anonymous_function*

[3] *https://de.wikipedia.org/wiki/Haskell_Brooks_Curry*

[4] Marco Vermeulen, Rúnar Bjarnason, and Paul Chiusano (2020): *Functional Programming in Kotlin*. Manning Publications Co, Shelter Island NY

[5] *https://arrow-kt.io/*

Stefan López Romero

MaibornWolff GmbH

stefan.lopez@maibornwolff.de

Stefan López Romero ist IT-Architekt bei MaibornWolff. Der Diplom-Informatiker beschäftigt sich seit vielen Jahren mit funktionaler Programmierung in verschiedenen Sprachen und versucht diese, wo immer es geht, im Projektalltag anzuwenden.

Java aktuell

Mehr Informationen zum Magazin und Abo unter:

https://www.ijug.eu/de/java-aktuell

iJUG Verbund
www.ijug.eu

Eventpartner: AOUG SOUG swiss oracle user group iJUG Verbund ORACLE