Programmierzettel

Funktionsobjekte

Dominikus Herzberg

Version 0.3, 2017-03-03

Table of Contents

Funktionsobjekte

Funktionen sind ein Sprachkonzept der funktionalen Programmierung. Eine Funktion ist vergleichbar mit einer Methode. Aufgerufen mit Argumenten berechnet die Funktion einen Ergebniswert. Allerdings können Funktionen in Variablen gespeichert und als Argumente übergeben werden — das wiederum geht mit Methoden nicht, sondern erinnert an Eigenschaften von Objekten.

Mit Java 8 hat man die Idee einer Funktion in der Verquickung von Methode und Objekt durch sogenannte Funktionsobjekte realisiert. Damit hat man dem Bedarf nach funktionalen Ausdrucksmöglichkeiten in Java Rechnung getragen.

Funktionsobjekt

: Ein Funktionsobjekt ist eine Instanz eines funktionalen Interfaces.

Funktionales Interface

: Als funktional wird ein Interface dann bezeichnet, wenn es nur genau *eine* abstrakte Methode enthält (eine Methode ohne Implementierung). Daneben kann das Interface beliebig viele Methoden enthalten, die entweder `static` sind oder aber eine `default` -Implementierung haben.

Funktionale Methode

: Die abstrakte Methode des funktionalen Interfaces wird auch als "funktionale Methode" bezeichnet.

Jedes Interface, das die genannte Bedingung erfüllt, ist ein funktionales Interface. Java stellt jedoch für die funktionale Programmierung eine Reihe generischer funktionaler Interfaces zur Verfügung.

In der funktionalen Programmierung ist ein Lambda-Ausdruck ein Konstrukt zur Erstellung einer Funktion, einer anonymen, sprich namenlosen Funktion. Nicht anders ist es in Java. Mit einem Lambda-Ausdruck erstellt man in Java ein Funktionsobjekt.

Lambda-Ausdruck

: Ein Lambda-Ausdruck deklariert die Implementierung eines funktionalen Interfaces und evaluiert zu einem Funktionsobjekt, einer Instanz des funktionalen Interfaces.

Um jede Methode und jeden Konstruktor auch als Funktion verwenden zu können, hat Java außerdem die sogenannte Methodenreferenz eingeführt.

Methodenreferenz

: Eine Methodenreferenz verweist auf die Implementierung einer Methode oder eines Konstruktors und evaluiert zu einem Funktionsobjekt, das die Methode bzw. den Konstruktor als Implementierung hat.

Lambda-Ausdrücke (wie `n -> n + 1`)

Ein Lambda-Ausdruck deklariert (wie die Methode) einen oder mehr Parameter (`LambdaParameters`) und einen Rumpf (`LambdaBody`), der Pfeil `->` trennt beides voneinander. Der Rumpf ist ein Ausdruck oder ein vollwertiger Programmblock.

Syntax zu Lambda-Ausdrücken

```
LambdaExpression: LambdaParameters `->` LambdaBody

LambdaParameters: Identifier
                | `(` [FormalParameterList] `)`
                | `(` InferredFormalParameterList `)`

InferredFormalParameterList:
    Identifier {`,` Identifier}

LambdaBody: Expression
          | Block
```

Das folgende Beispiel definiert ein funktionales Interface, sprich ein Interface, das mithilfe eines Lambda-Ausdrucks implementiert werden kann.

```
jshell> @FunctionalInterface interface UnaryOperation { int operate(int i); }
|  created interface UnaryOperation
```

Die Annotation `@FunctionalInterface` verdeutlich die Aufgabe des Interfaces, und der Compiler prüft, ob die Bedingung an ein funktionales Interface erfüllt ist.

Der Lambda-Ausdruck deklariert eine Implementierung zu dem Interface. Die Evaluation des Lambda-Ausdrucks ergibt eines Instanz vom Typ des Interfaces.

```
jshell> UnaryOperation inc = n -> n + 1;
inc ==> $Lambda$21/540642172@6fc6f14e
```

Der Java-Compiler weiß mit Blick auf das Interface, dass mit `n` der Parameter `i` aus dem Interface gemeint ist und dass `n + 1` per implizitem `return` den Rückgabewert liefert.

Das Funktionsobjekt kann nun mit dem Namen der funktionalen Methode aufgerufen werden.

```
jshell> inc.operate(7)
$30 ==> 8
```

Zur Notation von Lambda-Ausdrücken

Implizite Typisierung der Parameter

Die Parameter eines Lambda-Ausdrucks sind entweder

- explizit typisiert (*explicitly typed*), Beispiel `int n -> n + 1`
- oder implizit typisiert (*implicitly typed*), Beispiel `n -> n + 1`.

Bei der impliziten Typisierung ist ausschließlich der Name des Parameters anzugeben. Der Typ der Parameter und des Rückgabewerts ergibt sich aus den Typangaben im funktionalen Interface.

Es ist üblich, Lambda-Ausdrücke *ohne* explizite Typangabe zu formulieren.

Notation der Parameter `(...)`

- Bei einem Parameter können die runden Klammern entfallen, statt `(n) -> n + 1` kann man `n -> n + 1` schreiben
- Ab zwei Parametern müssen die runden Klammern genutzt werden, die Parameter sind durch Kommas zu trennen, Beispiel `(x, y) -> x + y`
- Soll der Lambda-Ausdruck parameterlos sein, so sind leere Klammern links vom Pfeil zu verwenden, Beispiel `() -> "Hi"`

Da Lambda-Ausdrücke oftmals nur einen Parameter haben, wird die klammerlose Schreibweise bevorzugt.

Notation des Rumpfes `{...}`

Besteht der Rumpf aus einem einzigen Ausdruck, wie z.B. bei `n -> n + 1`, so können das `return` und die geschweiften Klammern für den Programmblock entfallen; vollständig müsstes es `n -> { return n + 1; }` heißen. Ist der Rumpf ein Programmblock, so ist die Verwendung der `return`-Anweisung die gleiche wie bei Methoden.

Ein Kurzrumpf ohne `return` und geschweifte Klammern ist üblich. Es gibt Befürworter, die emfehlen die Refaktorisierung (Umbau) des Codes, wenn der Rumpf nicht mit der Kurzform auskommt.

Beschränkter Zugriff auf den Lambda-Kontext

Im Rumpf eines Lambda-Ausdrucks kann auf Variablen des ihn umgebenden Programmkontextes zugegriffen werden. Allerdings müssen diese Variablen (*effectively*) *final* sein. Eine Variable ist *effectively final*, wenn sie zwar nicht als `final` ausgewiesen ist, aber so verwendet wird. (Der Grund ist übrigens der gleiche wie bei den lokalen Klassen: Zugriffe auf Variablen der umgebenden Methode(!) sind unsinnig, das Objekt soll ja aus diesem Kontext gelöst werden.)

Methodenreferenzen (wie `String::length`)

Methodenreferenzen evaluieren ebenfalls zu Funktionsobjekten. Nur — und das ist der Clou — wird das Funktionsobjekt aus der Implementierung einer vorhandenen Klassen- oder Instanzmethode oder gar aus einem Konstruktor erzeugt. Mit anderen Worten: Im Gegensatz zum Lambda-Ausdruck verweist die Methodenreferenz namentlich auf eine Implementierung, aus der sich außerdem die Typ-Anforderungen an das funktionale Interface ableiten, gegeben durch die Typsignatur (die Typen der Methoden-Parameter) und den Rückgabetyp.

Die Syntax zur Methodenreferenzen "ersetzt", wenn man so möchte, den Punkt als Zugriffsoperator zur Methode durch einen Doppelpunkt `::` und lässt die runden Klammern mit den Argumenten zum Methodenaufruf weg.

Syntax zur Methodenreferenz

```
MethodReference:
      ExpressionName `::` [TypeArguments] Identifier
    | ReferenceType `::` [TypeArguments] Identifier
    | Primary `::` [TypeArguments] Identifier
    | `super` `::` [TypeArguments] Identifier
    | TypeName `.` `super` `::` [TypeArguments] Identifier
    | ClassType `::` [TypeArguments] `new`
    | ArrayType `::` `new`
```

Ein Beispiel: Die Methode `length()` der Klasse `String` lässt sich per `String::length` referenzieren, was ein funktionales Interface vom Typ `String` zu `Integer` bedingt. Erstellen wir ein geeignetes funktionales Interface:

```
jshell> @FunctionalInterface interface S2I { Integer apply(String s); }
|  created interface S2I
```

Mit diesem Interface können wir ein Funktionsobjekt mit der Methodenreferenz erzeugen und schlußendlich über die `apply`-Methode des Interfaces aufrufen.

```
jshell> (S2I) String::length
$48 ==> $Lambda$26/120694604@369f73a2

jshell> $48.apply("Hello")
$49 ==> 5
```

Mit anderen Worten: "Hinter" der funktionalen Methode des funktionalen Interfaces ist die Methode der Methodenreferenz hinterlegt.

Die funktionalen Interfaces `Function`, `Comsumer`, `Supplier`

Java bietet im Paket `java.util.function` eine Reihe generischer funktionaler Interfaces an. Sie ersparen einem, diese Interfaces selbst anlegen zu müssen, und sie bieten durchgängig verwendbare funktionale Methoden an.

`Function<T,R>`

Das Interface `Function<T,R>` bietet mit der funktionalen Methode `apply()` die Vorlage für Funktionsobjekte, die *ein* Argument vom Typ `T` entgegen nehmen und einen Rückgabewert vom Typ `R` zurückliefern. Statt das Interface `S2I` zu nehmen, kann es nun heißen:

```
jshell> (Function<String,Integer>) String::length
$50 ==> $Lambda$27/2006034581@3a5ed7a6

jshell> $50.apply("Hello again!")
$51 ==> 12
```

`Consumer<T>`

Für Funktionsobjekte, deren Rückgabetyp als `void` deklariert ist, eignet sich das Interface `Consumer<T>` mit der funktionalen Methode `accept()`; `T` ist der Typ des Aufrufarguments. Ein Beispiel zur Methode `System.out.println()`:

```
jshell> (Consumer<String>) System.out::println
$52 ==> $Lambda$28/1740189450@2b9627bc

jshell> $52.accept("Print me!")
Print me!
```

`Supplier<T>`

Das funktionale Interface `Supplier<T>` mit der funktionalen Methode `get()` erwartet keine Argumente, sondern liefert eine Instanz vom Typ `T` zurück. Hier ein Beispiel zusammen mit einer Konstruktorreferenz `new`:

```
jshell> (Supplier<ArrayList<Integer>>) ArrayList<Integer>::new
$54 ==> $Lambda$29/1643691748@7b49cea0

jshell> $54.get()
$55 ==> []
```

Die funktionalen Interfaces `Predicate`, `UnaryOperator`

`Predicate<T>`

Das funktionale Interface `Predicate<T>` mit der funktionalen Methode `test()` erwartet als Input einen Wert vom Typ `T` und liefert ein boolsches Ergebnis zurück. Funktionen, die einen boolschen Wert liefern, nennt man generell "Prädikate".

```
jshell> (Predicate<String>) String::isEmpty
$56 ==> $Lambda$30/1846412426@5bc79255

jshell> $56.test("Leer")
$57 ==> false
```

`UnaryOperator<T>`

Das funktionale Interface `UnaryOperator<T>` erwartet einen Eingabewert vom Typ `T` und einen Rückgabewert vom Typ `T`. Die funktionale Methode ist `apply()`.

```
jshell> (UnaryOperator<Integer>) i -> i + 1
$1 ==> $Lambda$17/507084503@490ab905

jshell> $1.apply(8)
$4 ==> 9
```

Wir hätten uns gar nicht die Mühe machen müssen, ein eigenes Interface `UnaryOperation` zu definieren, siehe das Beispiel beim Lambda-Ausdruck.

Alle weiteren funktionalen Interfaces von `java.util.function` sind Variationen von `Function`, `Supplier`, `Consumer`, `Predicate` oder `Operator` und tragen diese Bezeichnung in ihrem Namen. Entscheidend sind auch die ergänzenden `default`- und `static`-Methoden, die diese funktionalen Interfaces mitbringen. Sie finden in vielen Standard-Bibliotheken von Java Verwendung.

Die Bedeutung von Lambda-Ausdrücken in Java

Modernes Java ist ohne Lambda-Ausdrücke und Methodenreferenzen nicht mehr denkbar. Viele APIs wurden angepasst, um insbesondere von Lambda-Ausdrücken zu profitieren. Ganz besonders gilt das für die Arbeit mit Strömen (*streams*), die dem imperativen Programmierstil einen datenflussorientierten hinzugefügt haben.