9 Lambdas und Streams

Dieses Kapitel stellt sowohl Lambda-Ausdrücke (kurz ***Lambdas***) als auch das damit engverbundene Stream-API einführend vor. Beides in Kombination ermöglicht es, Lösungen oftmals elegant zu formulieren.

9.1 Einstieg in Lambdas

Das Sprachkonstrukt Lambda kommt aus der ***funktionalen Programmierung***. Vereinfacht gesprochen ist ein ***Lambda*** ein Behälter für Sourcecode, der vielfältig eingesetzt werden kann.

9.1.1 Syntax von Lambdas

Lambdas ähneln Methoden, besitzen im Gegensatz dazu jedoch keinen Namen. Zudem findet sich keine explizite Angabe eines Rückgabetyps oder potenziell ausgelöster Exceptions. Damit ergibt sich eine ziemlich kurze, auf das Wesentliche reduzierte Schreibweise mit folgender Syntax:

```
(Parameterliste) -> { Ausdruck oder Anweisungen }
```

Ein paar einfache Beispiele für Lambdas sind die Addition von zwei Zahlen vom Typ `int`, die Multiplikation eines `long`-Werts mit dem Faktor 2 oder eine parameterlose Funktion zur Ausgabe eines Textes auf der Konsole. Diese Aktionen kann man als Lambdas wie folgt schreiben:

```
(int x, int y) -> { return x + y; }
(long x) -> { return x * 2; }
() -> { String msg = "Lambda"; System.out.println("Hello " + msg); }
```

Tatsächlich sehen diese Anweisungen recht unspektakulär aus, und insbesondere wird klar, dass ein Lambda lediglich ein Stück ausführbarer Sourcecode ist, der

- keinen Namen besitzt, sondern lediglich Funktionalität, und dabei
- keine explizite Angabe eines Rückgabetyps und
- keine Deklaration von Exceptions erfordert und erlaubt.[1]

[1] Das gilt für sogenannte Checked Exceptions (vgl. Abschnitt 11.4). Solche vom Basistyp `RuntimeException` sind erlaubt.

Lambdas im Java-Typsystem

Wir haben bisher gesehen, dass sich einfache Berechnungen mithilfe von Lambdas ausdrücken lassen. Wie können wir diese aber nutzen und aufrufen? Versuchen wir zunächst, einen Lambda einer `java.lang.Object`-Referenz zuzuweisen, so wie es mit jedem anderen Objekt in Java möglich ist:

```
// Compile-Error: incompatible types: Object is not a functional interface
Object greeter = () -> { System.out.println("Hello Lambda"); };
```

Die gezeigte Zuweisung wird nicht unterstützt und führt zu einem Kompilierfehler. Die Fehlermeldung gibt einen Hinweis auf inkompatible Typen und verweist darauf, dass `Object` kein Functional Interface ist. Aber was ist ein Functional Interface?

Besonderheit: Lambdas im Java-Typsystem

Bis JDK 8 konnte in Java jede Referenz auf den Basistyp `Object` abgebildet werden. Mit Lambdas existiert nun ein Sprachelement, das nicht direkt dem Basistyp `Object` zugewiesen werden kann, sondern nur an Functional Interfaces.

9.1.2 Functional Interfaces und SAM-Typen

Ein ***Functional Interface*** repräsentiert ein Interface mit genau einer abstrakten Methode. Ein solches wird auch ***SAM-Typ*** genannt, wobei SAM für Single Abstract Method steht. Diese Art von Interfaces gibt es nicht erst seit Java 8 im JDK, sondern schon seit Langem und vielfach – wobei es früher für sie aber keine Bezeichnung gab. Vertreter der SAM-Typen und Functional Interfaces sind etwa `Runnable`, `FileFilter`, `FilenameFilter`, `ActionListener`, `EventHandler` usw.

```
@FunctionalInterface
public interface Runnable
{
    public abstract void run();
}
```

Das Listing zeigt die Markierung mit der Annotation `@FunctionalInterface` aus dem Package `java.lang`. Damit wird ein Interface explizit als Functional Interface gekennzeichnet. Die Angabe der Annotation ist optional: Jedes Interface mit genau nur einer abstrakten Methode (SAM-Typ) stellt auch ohne explizite Kennzeichnung ein Functional Interface dar. Wenn die Annotation angegeben wird, kann der Compiler eine Fehlermeldung produzieren, falls es (versehentlich) mehrere abstrakte Methoden gibt.

Implementierung von Functional Interfaces

Ein SAM-Typ bzw. Functional Interface lässt sich durch eine anonyme innere Klasse implementieren. Seit JDK 8 sind Lambdas zu bevorzugen. Voraussetzung dafür ist, dass mit dem Lambda die abstrakte Methode des Functional Interface erfüllt werden kann, d. h., dass die Anzahl der Parameter übereinstimmt sowie deren Typen und der Rückgabetyp kompatibel sind. Betrachten wir zur Verdeutlichung zunächst ein allgemeines, etwas abstraktes Modell zur Transformation von bisherigen Realisierungen eines SAM-Typs mithilfe einer anonymen inneren Klasse in einen Lambda-Ausdruck:

```
// SAM-Typ als anonyme innere Klasse
new SAMTypeAnonymousClass()
{
    public void samTypeMethod(METHOD-PARAMETERS)
    {
        METHOD-BODY
    }
}

// SAM-Typ als Lambda
(METHOD-PARAMETERS) -> { METHOD-BODY }
```

Bei kurzen Methodenimplementierungen, wie sie für SAM-Typen häufig vorkommen, ist das Verhältnis von Nutzcode zu Boilerplate-Code (auch Noise genannt) bislang recht schlecht. Wenn man für derartige Realisierungen Lambdas einsetzt, so kann man mit einer Zeile das ausdrücken, was sonst fünf oder mehr Zeilen benötigt. Nachfolgend wird dies für das Interface `Comparator<T>` verdeutlicht.

Beispiel: `Comparator<T>` Die Vorteile von Lambdas lassen sich für den Typ `Comparator<T>` gut demonstrieren. Mit einem `Comparator<T>` wird ein Vergleich von zwei Instanzen vom Typ `T` realisiert, indem man die abstrakte Methode `int compare(T, T)` passend implementiert. Der Rückgabewert bestimmt die Reihenfolge der Werte. Wollte man zwei Strings nach deren Länge sortieren, so entsteht herkömmlicherweise einiges an Sourcecode:

```
Comparator<String> compareByLength = new Comparator<>()
{
    @Override
    public int compare(final String str1, final String str2)
    {
        return Integer.compare(str1.length(), str2.length());
    }
};
```

Wenn man Lambdas nutzt, lässt sich der Komparator knackig wie folgt schreiben:

```
Comparator<String> compareByLength = (final String str1, final String str2) ->
{
    return Integer.compare(str1.length(), str2.length());
};
```

Type Inference und Kurzformen der Syntax

Die Syntax von Lambdas besitzt einige Besonderheiten, um den Sourcecode prägnant formulieren zu können. Durch die sogenannte ***Type Inference*** ermittelt der Compiler die passenden Typen aus dem Einsatzkontext und es ist dadurch möglich, auf die Angabe der Typen für die Parameter im Sourcecode zu verzichten. Den vorherigen Komparator schreibt man ohne Typangabe bei den Parametern des Lambdas wie folgt:

```
Comparator<String> compareByLength = (str1, str2) ->
{
    return Integer.compare(str1.length(), str2.length());
};
```

Eine weitere Verkürzung in der Schreibweise eines Lambdas erreicht man durch folgende Regeln: Falls das auszuführende Stück Sourcecode ein Ausdruck ist, können die geschweiften Klammern um die Anweisungen entfallen. Ebenfalls kann dann das Schlüsselwort `return` weggelassen werden und der Rückgabewert entspricht dem Ergebnis des Ausdrucks. Außerdem gilt: Existiert lediglich ein Eingabeparameter, so sind die runden Klammern um den Parameter optional. Damit ergibt sich für die Ausdrücke

```
(int x, int y) -> { return x + y; }
(long x) -> { return x * 2; }
```

folgende Kurzschreibweise:

```
(x, y) -> x + y
x -> x * 2
```

Neben dem offensichtlichen Vorteil einer kompakten Schreibweise ist etwas anderes viel entscheidender: Lambdas können flexibler als streng typisierte Methoden genutzt werden. Für die gezeigten Berechnungen ist ein Einsatz überall dort möglich, wo für die Parameter die Operatoren + bzw. * definiert sind, also für die Typen `int`, `float`, `double` usw. Anders formuliert: ***Alles, was hergeleitet werden kann (und soll), darf in der Syntax weggelassen werden***. Als Beispiel betrachten wir folgende `ActionListener`-Implementierung, die schrittweise vereinfacht wird:

```
// Alter Stil
button.addActionListener(new ActionListener()
{
    @Override
    public void actionPerformed(final ActionEvent e)
    {
        System.out.println("button clicked (old way)");
    }
});
```

Diese herkömmliche Realisierung mithilfe einer anonymen inneren Klasse lässt sich als Lambda und mit Type Inference deutlich kürzer schreiben:

```
// Lambda-Variante mit Type Inference
button.addActionListener((event) -> { System.out.println("button clicked!"); });
```

Nutzt man zusätzlich die Regeln zur Schreibweisenabkürzung, so entsteht Folgendes:

```
// Lambda-Kurzschreibweise
button.addActionListener(event -> System.out.println("button clicked!"));
```

Lambdas als Parameter und als Rückgabewerte

Wir haben mittlerweile ein wenig Gespür für Lambdas entwickelt und wissen, dass man Lambdas anstelle einer anonymen inneren Klasse zur Realisierung eines SAM-Typs nutzen kann. Ebenso lassen sich Lambdas als Methodenparameter und als Rückgabe einer Methode verwenden, um Aufrufe lesbar zu gestalten.

Als Beispiel schauen wir uns das Sortieren einer Liste von Namen gemäß deren Länge an. Das können wir mit folgenden zwei Varianten eines Lambdas für das Interface `Comparator<T>` schreiben:

```
jshell> var names = Arrays.asList("Andy", "Michael", "Max", "Stefan")
names ==> [Andy, Michael, Max, Stefan]

jshell> // Lambda als Methodenparameter

jshell> names.sort((str1, str2) -> Integer.compare(str1.length(), str2.length())
      )

jshell> names
names ==> [Max, Andy, Stefan, Michael]

jshell> public static Comparator<String> compareByLength()
   ...> {
   ...>    return (str1, str2) -> Integer.compare(str1.length(), str2.length());
   ...> }
|  created method compareByLength()

jshell> // Alternativ Lambda als Rückgabe einer Methode und andere Sortierung

jshell> names.sort(compareByLength().reversed())

jshell> names
names ==> [Michael, Stefan, Andy, Max]
```

9.2 Methodenreferenzen

Neben Lambdas kann der Einsatz von Methodenreferenzen dazu beitragen, die Lesbarkeit des Sourcecodes zu erhöhen. Das Sprachfeature der Methodenreferenzen besitzt die Syntax `Klasse::Methodenname` und verweist auf ...

- eine Methode – `System.out::println`, `Person::getName`, ...
- einen Konstruktor – `ArrayList::new`, `Person[]::new`, ...

Das wirkt recht unspektakulär. Eine Methodenreferenz lässt sich aber zur Vereinfachung der Schreibweise anstelle eines Lambdas nutzen.

Betrachten wir folgendes Beispiel einer Konsolenausgabe:

```
jshell> List<String> names = List.of("Max", "Andy", "Michael")
names ==> [Max, Andy, Michael]

jshell> names.forEach(it -> System.out.println(it)) // Lambda
Max
Andy
Michael

jshell> names.forEach(System.out::println) // Methodenreferenz
Max
Andy
Michael
```

Wie man sieht, verbessert sich die Lesbarkeit durch den Einsatz der Methodenreferenz. Allerdings könnte man sich noch folgende Fragen zu der Ersetzung stellen: Methoden erhalten oftmals Parameter – wie auch im Listing. Wie werden diese für Methodenreferenzen übergeben? Wie ist die Reihenfolge bei mehreren? Die Antwort darauf ist, dass diese Informationen vom Compiler ermittelt und automatisch beim jeweiligen Methodenaufruf übergeben werden.

Ergänzend zu dieser Ausführung möchte ich an ein paar Beispielen zeigen, wie sich Methodenreferenzen auf Lambdas bzw. andersherum abbilden lassen. Dabei gibt es vier verschiedene Varianten, die in Tabelle 9-1 dargestellt sind.

Tabelle 9-1 *Methodenreferenzen*

Referenz auf ...	Als Methodenreferenz	Als Lambda
Statische Methode	`String::valueOf`	`obj -> String.valueOf(obj)`

Instanzmethode eines Typs	`Object::toString`
	
	`String::compareTo`
		`obj -> obj.toString()`
		
		`(str1, str2) -> str1.compareTo(str2)`

Instanzmethode eines Objekts	`person::getName`	`() -> person.getName()`
Konstruktor	`ArrayList::new`	`() -> new ArrayList<>()`

9.3 Externe vs. interne Iteration

Mittlerweile haben wir Lambdas ein paarmal in Aktion erlebt. Vor allem beim ***Durchlaufen einer Collection***, auch ***Iteration*** genannt, unterscheidet man die externe und die interne Iteration. Von ***externer Iteration*** spricht man, wenn der Vorgang der Iteration vom Aufrufer kontrolliert wird. Dagegen wird bei der ***internen Iteration*** das Durchlaufen durch die Collection-Klasse gekapselt und dort intern realisiert. Implementierungsdetails bleiben so verborgen, allerdings sind auch die Möglichkeiten zur Einflussnahme

durch den Aufrufer begrenzt. Betrachten wir nachfolgend einige Beispiele für die externe und interne Iteration.

9.3.1 Externe Iteration

Nehmen wir an, wir wollten alle Elemente einer Collection auf der Konsole ausgeben. Herkömmlicherweise könnte man dies wie folgt implementieren:

```
final List<String> names = Arrays.asList("Andi", "Mike", "Ralph", "Stefan" );

// Klassische Variante mit Iterator ...
final Iterator<String> it = names.iterator();
while (it.hasNext())
{
    System.out.println(it.next());
}

// ... oder alternativ mit indiziertem Zugriff
for (int i = 0; i < names.size(); i++)
{
    System.out.println(names.get(i));
}

// JDK-5-Schreibweise mit "for-each"
for (final String name : names)
{
    System.out.println(name);
}
```

Dieses Beispiel verdeutlicht die iterative und sequenzielle Abarbeitung sowohl für die Variante mit Iterator als auch für den danach gezeigten indizierten Zugriff. Die Variante mit der sogenannten for-each-Schleife zeigt den sequenziellen Charakter weniger klar. In allen drei Fällen spricht man von ***externer Iteration***, weil die ***Traversierung im Applikationscode programmiert*** wird.

9.3.2 Interne Iteration

Wir haben bei der Beschreibung von Collections verschiedene Iterationsvarianten besprochen, unter anderem auch die mit `forEach()` und einem Lambda. Das Besondere daran ist, dass man die in der internen Iteration auszuführende Funktionalität übergibt. Dazu bieten sich sowohl Lambdas als auch Methodenreferenzen an:

```
// Interne Iteration in zwei Varianten
names.forEach(name -> System.out.println(name));
names.forEach(System.out::println);
```

Die im Listing gezeigte Form wird ***interne Iteration*** genannt, weil die Iteration nicht vom Entwickler selbst programmiert werden muss, sondern diese ***in der Collection realisiert*** wird. Man übergibt nur die auszuführende Aktion.

9.3.3 Das Interface `Predicate<T>`

Das funktionale Interface `java.util.function.Predicate<T>` erlaubt es, sogenannte ***Prädikate*** zu formulieren. Das sind boolesche Bedingungen, die durch Aufruf der im Interface definierten Methode `boolean test(T)` ausgewertet werden. Das Interface `Predicate<T>` ist wie folgt definiert (gekürzt):

```
@FunctionalInterface
public interface Predicate<T>
{
   boolean test(T t);

    // ...
}
```

Im folgenden Listing sind mithilfe von Lambdas und Methodenreferenzen einfache Prüfungen auf den Wert `null`, einen Leerstring oder eine Mindestlänge von 5 Zeichen kurz und knackig formuliert:

```
jshell> Predicate<String> isNull = str -> str == null
isNull ==> $Lambda$66/0x0000000800c3ce90@32d992b2

jshell> Predicate<String> isEmpty = String::isEmpty
isEmpty ==> $Lambda$67/0x0000000800c3d2e8@5dfcfece

jshell> Predicate<String> fiveOrMoreChars = str -> str.length() >= 5
fiveOrMoreChars ==> $Lambda$68/0x0000000800c3dd58@100fc185
```

Wir sehen die ziemlich kryptische Protokollierung der Definition der Lambdas in der JShell. Für uns ist das aber nicht weiter von Relevanz, da wir nur den sprechenden Namen der Variablen nutzen.

Probieren wir das Ganze doch einmal aus und lernen dabei gleich mit `negate()` und `not()` noch zwei Varianten der Negation kennen:

```
jshell> String name = "Moin"
name ==> "Moin"

jshell> isNull.test(name)
$283 ==> false

jshell> isEmpty.test(name)
$284 ==> false

jshell> fiveOrMoreChars.negate().test(name)
$286 ==> true

jshell> Predicate.not(fiveOrMoreChars).test(name)
$287 ==> true
```

Die Aufrufe mit `test()` wirken noch etwas ungelenk. Tatsächlich sollte man die Methoden aus dem Functional Interface wohl eher selten selbst aufrufen. Im Kontext des Stream-APIs muss man dies nicht, sondern es geschieht versteckt im Framework. Dann ergibt sich deutlich lesbarer Sourcecode.

Beispiel: `Predicate<T>` in Aktion

Ein weiteres Beispiel für den sinnvollen Einsatz von `Predicate<T>` ist die Methode `removeIf()`, mit der man in Listen entsprechende Elemente löschen kann.

```
jshell> var cities = List.of("Kiel", "Köln", "Aachen", "Zürich", "Bern",
   ...>                      "Bremen", "Hamburg", "Lübeck", "Luzern")
cities ==> [Kiel, Köln, Aachen, Zürich, Bern, Bremen, Hamburg, Lübeck, Luzern]

jshell> var mutableCities = new ArrayList<>(cities)
mutableCities ==> [Kiel, Köln, Aachen, Zürich, Bern, Bremen, Hamburg, Lübeck,
      Luzern]

jshell> mutableCities.removeIf(Predicate.not(fiveOrMoreChars))
$293 ==> true

jshell> mutableCities
mutableCities ==> [Aachen, Zürich, Bremen, Hamburg, Lübeck, Luzern]
```

Im Listing sehen wir den Aufruf der Methode `removeIf(Predicate<E>)`, die Elemente aus einer Collection entfernt, die der übergebenen Bedingung entsprechen. Hier werden alle Städte mit weniger als 5 Zeichen aus der Liste gelöscht.

9.4 Streams im Überblick

Beim Konzept der ***Streams*** spielt das Interface `java.util.stream.Stream<T>` eine Schlüsselrolle. Streams sind eine Abstraktion für ***Folgen von Verarbeitungsschritten auf Daten***. Darüber hinaus ähneln Streams sowohl Collections als auch Iteratoren, wobei Streams keine Speicherung der Daten vornehmen und nur einmal traversiert werden können. Als weitere Analogie kann die Abarbeitung als Fließband betrachtet werden. Dabei unterscheidet man zwischen diesen drei Typen von Operationen: ***Create*** (Erzeugung), ***Intermediate*** (Berechnung) und ***Terminal*** (Ergebnisermittlung). Nachfolgend ist dies schematisch dargestellt:

$$\underbrace{Quelle \Rightarrow STREAM}_{Create} \Rightarrow \underbrace{OP_1 \Rightarrow OP_2 \Rightarrow ... \Rightarrow OP_{n-1}}_{Intermediate} \Rightarrow \underbrace{OP_n \Rightarrow Ergebnis}_{Terminal}$$

Einführendes Beispiel

Das folgende Listing zeigt die Operationen, ohne auf Details einzugehen. Hier geht es zunächst nur darum, einen ersten Eindruck für Streams und die Verarbeitung damit zu bekommen. Dazu schauen wir uns eine Liste von Personen an, die auf alle Erwachsenen gefiltert und als `List<Person>` zurückgegeben wird:

```
List<Person> adults = persons.stream().                    // Create
                              filter(Person::isAdult).     // Intermediate
                              collect(Collectors.toList()); // Terminal
```

Neben all diesen (noch unbekannten) Implementierungsneuerungen erkennt man sehr schön, dass sich Konzepte und das »Was« viel klarer erkennen lassen und nicht das »Wie« (die Details der Implementierung der Funktionalität) im Vordergrund steht.

9.4.1 Streams erzeugen – Create Operations

Nach dem ersten Beispiel zu Streams wollen wir unsere Kenntnisse vertiefen. In den folgenden Abschnitten stelle ich einige Varianten zur Erzeugung von Streams vor.

Streams für Arrays und Collections

Für Arrays oder Collections erzeugt die Methode `stream()` ein `Stream`-Objekt:

```
final String[] namesData = { "Karl", "Ralph", "Andi", "Andy", "Mike" };
final List<String> names = Arrays.asList(namesData);

final Stream<String> streamFromArray = Arrays.stream(namesData);
final Stream<String> streamFromList = names.stream();
```

Als Besonderheit können Collections eine sequenzielle sowie eine parallele Variante eines Streams liefern:

```
final Stream<String> sequentialStream = names.stream();
final Stream<String> parallelStream = names.parallelStream();
```

Für Arrays bietet die Utility-Klasse `java.util.Arrays` dagegen nur Zugriff auf eine sequenzielle Variante. Um hier auf Parallelverarbeitung zu wechseln, kann man die Methode `parallel()` auf dem zu parallelisierenden Stream aufrufen. Für das obige Array müsste man somit Folgendes schreiben:

```
final Stream<String> parallelArrayStream = Arrays.stream(namesData).parallel();
```

Streams für vordefinierte Wertebereiche

Teilweise soll über Streams ein fixer, vordefinierter Wertebereich abgebildet und bearbeitet werden. Dazu gibt es spezielle Methoden, etwa `of()`, `range()` und `chars()`:

```
final Stream<String> names = Stream.of("Tim", "Andy", "Mike");   // String
final Stream<Integer> integers = Stream.of(1, 4, 7, 7, 9, 7, 2); // Integer

final IntStream values = IntStream.range(0, 100);                // int
final IntStream chars = "This is a test".chars();                // int
```

Im Listing kommt neben dem generischen Interface `Stream<T>` auch das für den primitiven Datentyp `int` spezifische Interface `java.util.stream.IntStream` zum Einsatz. Die Verarbeitung erfolgt in dieser Art von Streams mit Werten primitiver Typen und nicht wie bei `Stream<Integer>` mit `Integer`-Objekten. Zudem gibt es es zur Verarbeitung der primitiven Typen `long` und `double` die Klassen `LongStream` und `DoubleStream` aus dem Package `java.util.stream`.