Ausdrucksstark

Lambda-Ausdrücke in Java 8

Markus Günther, Martin Lehmann

Das JDK 8 erweitert die Java-Sprache erneut um moderne Sprach-Features. Mit dem Java Specification Request 335 werden sogenannte Lambda-Ausdrücke eingeführt, die insbesondere in der Welt der funktionalen Programmiersprachen auch als anonyme Funktionen bekannt sind. Der Artikel illustriert sowohl die technischen Details dieser Spracherweiterung als auch die idiomatische Benutzung von Lambdas.

Java ist eine objektorientierte Programmiersprache. Mit dem Boom funktionaler Programmiersprachen wie Scala oder Clojure wurde der Wunsch lauter, auch in Java funktionale Konzepte anzubieten. Der JSR 335, bekannt als Project Lambda (vgl. [Lambda] und [JEP126]), hat bereits seit 2009 zum Ziel, Java um das Sprach-Feature „Lambda Expressions" zu erweitern. Zielversion ist nun endlich Java 8. Lambdas sind damit *das* Sprach-Feature der nächsten Java-Version und erstmals seit den Generics in Java 5 eine signifikante Spracherweiterung.

Grund genug, einen detaillierten Blick auf das Thema zu werfen und zu zeigen, wie Entwickler durch die Verwendung von Lambdas profitieren können.

Einführung in Lambda-Ausdrücke

In Java sind Methoden eindeutig Klassen zugewiesen. Um Funktionalität aufzurufen, erzeugen wir Objektinstanzen und rufen deren Methoden mit *Daten* als Parameter auf. Manchmal möchten wir aber die in den Methoden benutzte *Funktionalität* parametrieren. In anderen Sprachen ist das als Entwurfsmuster „Code as Data" nutzbar, indem auszuführende Funktionalität mit übergeben wird.

Bisher lösen wir in Java dieses Problem mit Callbacks. Die neuen Lambda-Ausdrücke (vgl. [LambdaKal]) können durch geringen syntaktischen Ballast die Ausdrucksfähigkeit erhöhen. Hier vier erste Beispiele für Lambda-Ausdrücke:

```
// Beispiel 1
x -> x + 1

// Beispiel 2
(Integer i) -> list.add(i)

// Beispiel 3
(Integer a, Integer b) -> {
  if (a < b) return a + b;
  return a;
}

// Beispiel 4
() -> System.out.println("Hallo Lambda!")
```

Die ersten drei Lambda-Ausdrücke sind parametriert (durch die Variablen `x`, `i`, `a`, `b`), gegebenenfalls durch Angabe eines expliziten Typs (Beispiel 2). Rechts des `->`-Operators steht der auszuführende Code (ggf. in einem Block wie in Beispiel 3). Das `return` kann implizit (Beispiel 1) oder explizit (Beispiel 3) erfolgen.

Lambdas sind nur eines der neuen Features

Ein funktionales Konzept wie das der Lambda-Ausdrücke nachträglich in eine objektorientierte Sprache zu integrieren, ist eine große Herausforderung, da die Umsetzung konsistent und verständlich zu gestalten ist, bei gleichzeitig größtmöglicher Abwärtskompatibilität. Damit sich Lambdas nahtlos in Java einfügen und wirklichen Nutzen stiften, wird Java 8 neben Lambda-Ausdrücken weitere JSR335-Techniken mitbringen:

- Funktionale Interfaces,
- Referenzen auf Methoden,
- eine verbesserte Typinferenz, um einen Lambda-Ausdruck mit einem funktionalen Interface oder einer Methodenreferenz zu verknüpfen,
- Default-Methoden (auch bekannt als Virtual Extension Methods) für Default-Implementierungen in Interfaces,
- Nutzung dieser Features in erweiterten Java-Collections.

Zum Zeitpunkt der Artikelerstellung hatten wir Zugriff auf den Early Access Build 1.8.0ea-b76 von Februar 2013. In der finalen Version von Java 8 können sich API-Abweichungen ergeben, grundlegende Konzeptänderungen sind aber nicht zu erwarten.

Funktionale Interfaces: Lambdas statt Anonymous Inner Classes

Java setzt zur Realisierung von Lambda-Ausdrücken auf funktionale Interfaces, auch Single-Abstract-Method-Typen (SAM-Typen) genannt. Ein funktionales Interface darf mehrere Methodendeklarationen beinhalten, solange *genau eine abstrakt* ist. Warum betonen wir hier abstrakt, sind denn nicht alle Methoden eines Interfaces abstrakt? Ja, das galt bisher und ändert sich aber mit Java 8 durch die Einführung der Default-Methoden: Dadurch kann ein Interface Default-Implementierungen und somit nicht-abstrakte Methoden enthalten! Die abstrakte Methode ist der Kontext für einen Lambda-Ausdruck.

Viele Interfaces innerhalb des JDK deklarieren nur eine einzige Methode, wie beispielsweise `Runnable`, `Comparator<T>` oder `ActionListener`: Ein `ActionListener` codiert Callback-Logik, die unabhängig vom Definitionszeitpunkt ausgeführt wird. Das geschieht in der Regel in Form einer anonymen inneren Klasse, deren Instanz man einem Handler übergibt:

```
ActionListener l = new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent e) {
    // Auszuführender Code
  }
};
handler.addActionListener(l);
```

Aus Entwicklerperspektive ist es mühsam, einen `ActionListener` als eigene Klasse zu schreiben. Solche funktionalen Interfaces können wir in Java 8 innerhalb eines Lambda-Ausdrucks prägnanter so benutzen:

```
ActionListener l = (e) -> { /* Auszuführender Code */ };
handler.addActionListener(l);
```

Wird die Instanz nicht noch anderweitig gebraucht, auch noch kürzer so:

```
handler.addActionListener(
  (e) -> { /* Auszuführender Code */ };
);
```

Die wenigen Transformationsschritte von einer anonymen inneren Klasse zu einem äquivalenten Lambda-Ausdruck illust-

rieren wir am Beispiel eines `Comparator` auf ganzen Zahlen. Zunächst die klassische Variante mit einem `Comparator` als anonyme innere Klasse:

```
Comparator<Integer> anonymousInnerClass = new Comparator<Integer>(){
  @Override
  public int compare(Integer a, Integer b) {
    return a - b;
  }
};
```

Ein äquivalenter Lambda-Ausdruck, der kompakter und funktionaler ist, sieht so aus:

```
Comparator<Integer> lambdaExpression1 = (Integer a, Integer b) -> {
  return a - b;
};
```

Obiger Lambda-Ausdruck codiert die Operation innerhalb einer Blockstruktur mit expliziter Rückgabe des Ergebnisses. Das Beispiel kann man weiter auf einen einzeiligen Ausdruck kürzen (mit implizitem `return`):

```
Comparator<Integer> lambdaExpression2=(Integer a, Integer b) -> a - b;
```

Da ein funktionales Interface nur genau eine abstrakte Methode deklariert, kann der Compiler den Kontext des Lambda-Ausdrucks eindeutig ermitteln und die Typen anhand von Methodenergebnis und Parameter eindeutig definieren (Typsicherheit). Um die Anwendbarkeit eines funktionalen Interfaces innerhalb eines Lambda-Ausdrucks zu überprüfen, muss der Compiler lediglich wissen, ob die genannten Typen zueinander kompatibel sind. Die Namen von Interface-Klasse und -Methode spielen keine Rolle.

Durch die in Java 8 erweiterte Typinferenz ist die Angabe des Parametertyps nicht mehr notwendig. `Comparator.compare` sagt bereits eindeutig, welche Argumenttypen sie erwartet. Noch kürzer ist somit Variante 3:

```
Comparator<Integer> lambdaExpression3 = (a, b) -> a - b;
```

Lambdas können auf Variablen außerhalb ihres Scopes zugreifen
Diese Variablen sind entweder explizit `final` deklariert oder durch den Compiler als „effectively final" erkennbar. Das heißt, dass keine erneute Zuweisung an die Variable stattfindet und somit der Compiler automatisch erkennen kann, dass die Variable `final` benutzt wird (selbst wenn sie nicht als solche deklariert ist):

```
// Zugriff im Lambda-Ausdruck auf c möglich
void someMethod () {
  int c = 5; // effectively final, wenn c nicht verändert wird
  Function<Integer, Integer> myAddFct = (a) -> a + c;
  ...
}
```

Lambdas führen keine neuen Namen oder Kontexte ein
Ein `this` bezieht sich in einem Lambda-Ausdruck auf die umschließende Klasse, in einer anonymen inneren Klasse aber auf sich selbst.

Package java.util.function

Theoretisch könnten wir funktionale Interfaces wie beispielsweise ein `Runnable` überall da nutzen, wo Ergebnis- und Parametertypen passen. Allerdings wäre es der Lesbarkeit des Codes wenig zuträglich, wenn wir an Stellen ein `Runnable` verwenden, an denen `Runnable` als Kontext eines Lambda-Ausdrucks *syntaktisch* passt, *semantisch* aber ein anderes Interface erforderlich ist. Unterschiedliche Konzepte sollen durch unterschiedliche funktionale Interfaces repräsentiert werden.

Dabei helfen die neuen funktionalen Interfaces im Package `java.util.function`. Das Interface `Function<T,R>` ermöglicht eine Transformation der Eingabedaten, wendet also eine Funktion auf Eingabetyp `T` an und liefert ein Ergebnis vom Typ `R`:

```
Function<Integer, Integer> identity = (n) -> n;
identity.apply(3); // liefert 3
```

Das Interface `Consumer<T>` arbeitet analog zu `Function`, allerdings ohne Ergebnisrückgabe. Man verlässt damit die rein funktionale Programmierung, wenn ein solcher `Consumer` den Programmzustand durch Seiteneffekte verändert:

```
Consumer<Person> consumer = (p) -> System.out.println(p);
consumer.accept(somePerson);
```

Das Interface `Predicate<T>` unterzieht ein Objekt vom Typ `T` einem logischen Testkriterium. Die Rückgabe eines `Predicate` ist ein Boolesches Testergebnis. Ein typisches Nutzungsszenario ist die Filterung einer Collection:

```
Predicate<Integer> isEven = (n) -> n % 2 == 0;
isEven.test(2); // liefert true
```

Das Interface `Supplier<T>` deklariert eine `get`-Methode mit Rückgabetyp `T`:

```
Supplier<Integer> rndRange100 = () -> (int)(Math.random() * 100);
rndRange100.get();
```

Das Package `java.util.function` enthält weitere Interfaces, die sich konzeptionell nur geringfügig von den genannten unterscheiden und als syntaktischer Zucker zu verstehen sind.

Eigene funktionale Interfaces und Methodenreferenzen

Funktionale Interfaces legen die Typkonvention für Lambda-Ausdrücke fest. Ein eigenes funktionales Interface für Prädikate können wir wie folgt definieren:

```
// Mit der Annotation @FunctionalInterface überprüft der Compiler
// explizit, ob das Interface *genau eine* abstrakte Methode besitzt
@FunctionalInterface
interface MyPredicate<T> {
  boolean evaluate(T t);
}
```

`MyPredicate` benutzen wir, um aus einer Collection von `Person`-Objekten die erste `Person` zurückzuliefern, die dem im übergebenen `MyPredicate` codierten Kriterium entspricht:

```
public Person firstMatch (Collection<Person> persons,
    MyPredicate<Person> predicate) {
  for (Person p: persons) {
    if (predicate.evaluate(p)) {
      return p;
    }
  }
}
```

Damit können wir die erste volljährige `Person` in einer `Person`-Liste suchen:

```
// Nutzung von firstMatch mit Lambda-Ausdruck
firstMatch(myPersonList, (Person p) -> p.getAge() >= 18)
```

Bestehende Implementierungen, wie beispielsweise die nachstehenden Checks der Klasse `Person`, kann man wiederverwenden, ohne sie in einen Lambda-Ausdruck verpacken zu müssen:

```
public static boolean atLeastEighteen(Person p) {
  return p.getAge() >= 18;
}
public static boolean isMale(Person p) {
  return p.getGender().equals(Person.GENDER_MALE);
}
```

Wir nutzen eine Methodenreferenz darauf so:

```
// Nutzung von firstMatch mit Methodenreferenz
firstMatch(myPersonList, Person::atLeastEighteen)
```

Das geht nur bei Typkompatibilität mit dem funktionalen Interface. Wir können nur eine Methode referenzieren, deren Signatur einen Booleschen Rückgabewert und einen Parameter vom Typ `Person` vorsieht. Dies ist bei `atLeastEighteen` der Fall, aber auch bei `isMale`:

```
// Nutzung von firstMatch mit anderer Methodenreferenz
firstMatch(myPersonList, Person::isMale)
```

Evolution von Interfaces

Mit der Definition von Interfaces wird ein Vertrag für alle Clients festgelegt. Solche Verträge machen Abhängigkeiten explizit, gestalten aber auch Änderungen aufwendig, wenn sich Anforderungen ändern und neue oder modifizierte Funktionalität einzubauen ist.

Vor dieser Herausforderung standen die JDK-Implementierer: Die Java-Collections sind alt und sollen in Java 8 mit den neuen, oben vorgestellten Lambda-Ausdrücken angereichert werden. Gleichzeitig sollen aber ihre Schnittstellen abwärtskompatibel bleiben. Ein solcher Spagat ist mit bisherigen Java-Bordmitteln nicht möglich und erfordert eine weitere signifikante Java-8-Sprachänderung: den Einbau von Default-Implementierungen in Interfaces. Diese Funktionalität ist auch völlig unabhängig von Lambda und Collections in Java 8 nutzbar und wird wohl so manche abstrakte Basisklasse ersetzen.

Ein Beispiel-Interface `MyInterface` sieht zunächst so aus:

```
// Interface, alt
public interface MyInterface {
  void foo();
}
```

Nun möchten wir `MyInterface` um die Methode `bar()` erweitern:

```
// Interface, neu
public interface MyInterface {
  void foo();
  int bar(); // neue Methode bricht bestehende Implementierungen
}
```

Müssen nun nicht alle Implementierungen nachziehen? Wir unterscheiden Laufzeit- von Quelltext-Sicht:

- Laufzeit-Sicht: Das neue `MyInterface` können wir (alleine) kompilieren und dann seine alte class-Datei in einer Laufzeitumgebung durch die neue class-Datei ersetzen (z. B. in einer jar-Datei). Dies würde die Binärkompatibilität der Implementierungen *in der Laufzeitumgebung* nicht verletzen, das heißt, wir können dort zunächst alle Implementierungen von `MyInterface` *ohne Neukompilieren* unverändert lassen.
- Quelltext-Sicht: Sobald wir eine Implementierung im Quelltext anfassen, müssen wir den neuen Vertrag von `MyInterface` erfüllen, also auch `bar()` implementieren. Unsere Interface-Änderung ist nicht mehr abwärtskompatibel und wir müssen bestehenden Code ändern, testen und ausliefern.

Default-Methoden

Es gibt diverse Lösungsstrategien, um eine abwärtskompatible Evolution von Interfaces zu ermöglichen. Allen Ansätzen ist gemein, dass sie Default-Implementierung für neue Funktionalität anbieten (vgl. [Götz11a]). Default-Methoden ermöglichen Interface-Evolution in Java 8. Dieser Ansatz basiert darauf, einen Methodenaufruf zur Laufzeit umzuschreiben.

Das Schlüsselwort `default` zeigt eine Default-Implementierung im Interface an:

```
// Interface, neu – mit Default-Implementierung
public interface MyInterface {
  public void foo();
  public default int bar() {
    /* ... Default-Implementierung im Interface ... */
  }
}
```

Da eine Klasse mehrere Interfaces implementieren darf, kann es auch mehrere Default-Methoden mit der gleichen Methodensignatur in der Vererbungshierarchie geben. Der Compiler muss in der Lage sein, aus diesen Methodendeklarationen auszuwählen, und geht dabei nach folgendem Schema vor:

- Eine Methodendeklaration in einer *Oberklasse hat immer Vorrang* über Default-Methoden in Interfaces mit gleicher Methodensignatur.
- Gibt es keine solche Oberklasse, so wird das *spezifischste Interface* ausgewählt, das eine Default-Implementierung für diese Methode bereitstellt.

Im nachstehenden Beispiel definiert Interface `A` eine Default-Implementierung der Methode `sayHello`. Interface `B` erbt von `A` und überschreibt ebenfalls die Default-Implementierung. `C1` implementiert beide Interfaces `A` und `B`:

```
public interface A {
  default void sayHello() { System.out.println("Hallo aus A"); }
}
public interface B extends A {
  default void sayHello() { System.out.println("Hallo aus B"); }
}
public class C1 implements A, B {
  public static void main(String[] args) {
    C1 c = new C1();
    c.sayHello(); // Ausgabe: Hallo aus B
  }
}
```

Ausgehend von Klasse `C1` sucht der Compiler entlang der Vererbungshierarchie nach den spezifischsten Interfaces, die `sayHello` bereitstellen. Er prüft zuerst, ob eine Default-Implementierung in den Supertypen von `C1` vorzufinden ist, und wird in `B` fündig. Da `B` auf dieser Ebene der Vererbungshierarchie das einzige Interface ist, in dem `sayHello` implementiert ist, ist `B` das spezifischste Interface. Der Compiler weiß somit, dass `c.sayHello` auf die Default-Implementierung in Interface `B` zurückgreift.

Nun ist die Methodenresolution nicht immer eindeutig möglich. Im nachstehenden Beispiel implementieren die Interfaces `D` und `E` eine Default-Methode mit der gleichen Methodensignatur:

```
public interface D {
  default void sayHello() { System.out.println("Hallo aus D"); }
}
public interface E {
  default void sayHello() { System.out.println("Hallo aus E"); }
}
public class C2 implements D, E { // Kompilierfehler!
  public static void main(String[] args) {
    C2 c = new C2();
    c.sayHello();
  }
}
```

Aus Sicht der Klasse `C2` ist die spezifischste Default-Methode nicht identifizierbar, da `D` und `E` auf der *gleichen Ebene* der Vererbungshierarchie liegen. Die Klasse `C2` kompiliert in diesem Szenario nicht. Der Entwickler kann diese Doppeldeutigkeit in `C2` auflösen, wenn die Default-Implementierung gezielt überschrieben und die gewünschte Default-Implementierung mit `super` aufgerufen wird:

```
public class C2 implements D, E {
  @Override
  public void sayHello() {
    D.super.sayHello();
  }
  ...
}
```

Ist das nicht Mehrfachvererbung?

Die *Mehrfachvererbung von Typen* ist in Java seit jeher möglich, da von einer Klasse abgeleitet werden kann und zusätzliche Interfaces implementiert werden können, ohne dass dies ein Problem darstellt. Wirklich problematisch wäre nur die *Mehrfachvererbung von Zustand*, wie es C++ erlaubt. Durch Default-Methoden wird eine Mehrfachvererbung von Zustand nicht möglich und ist unkritisch.

Dennoch: Wir haben es bei Default-Methoden mit der Mehrfachvererbung von Verhalten zu tun. Da Default-Methoden als virtuelle Methoden behandelt werden, kann es zu überraschenden Resultaten kommen, wie nachfolgendes Beispiel illustriert:

```
public interface A {
  default void m(int i, int j) {
    int result = calculate(i, j);
    System.out.println("A.m(i="+i+",j="+j+") := " + result);
  }
  default int calculate(int i, int j) {
    System.out.println("A.calculate aufgerufen");
    return i + j;
  }
}
public interface B extends A {
  default int calculate(int i, int j) {
    System.out.println("B.calculate aufgerufen");
    return i * j;
  }
}
public interface C extends A {
  default void m(int i, int j) {
    int result = calculate(i, j);
    System.out.println("C.m(i="+i+",j="+j+") := " + result);
  }
}
public class D implements B, C {
  public static void main(String[] args) {
    D d = new D();
    d.m(3, 4); // Ergebnis ist 12, nicht 7
  }
}
```

Interface `A` implementiert zwei Default-Methoden `m` und `calculate`. Die Interfaces `B` und `C` erben beide von `A`, überschreiben aber jeweils nur *eine* dieser Methoden. Eine Klasse `D`, die beide Interfaces `B` und `C` implementiert und `m(3, 4)` aufruft, gelangt – womöglich überraschend – zu dem Ergebnis `12` statt `7`. Es wird aus `C` die Multiplikation in `B`, nicht die Addition in `A` aufgerufen. Das liegt daran, dass die spezifischste Implementierung für `m` aus `C` ist, die spezifischste Implementierung für `calculate` aber aus `B` genutzt wird, auch wenn `C` und `B` völlig getrennt sind. Das Beispiel mag pathologisch erscheinen, kann aber im Alltag schnell zu Problemen führen, wenn der Entwickler sich nicht über Vererbungshierarchien in fremdem Programmierschnittstellencode im Klaren ist.

Lambda-Ausdrücke und die neuen Collections

Die Nutzung von Lambda-Ausdrücken zusammen mit den neuen Java-Collections möchten wir abschließend am Beispiel einer Schulverwaltung erläutern, die fachliche Informationen zu Lehrern, Schulklassen, Schülern und geschriebenen Tests mit Bewertungen enthält. Möchten wir für einen Schüler den Notenschnitt berechnen, so sähe ein typischer Vor-Java-8-Code wohl etwa so aus:

```
public float averageGrade() {
  // summiere alle Noten aus allen Tests des Schuelers
  float sum = 0.0f;
  for (Examination exam : this.exams) {
    sum += exam.getGrade();
  }
  // bilde den Notenschnitt
  return sum / this.exams.size();
}
```

Obige Iteration ist extern, also in der Verantwortung des Entwicklers. Wir können die Iteration internalisieren, sprich in die neuen Collections verlagern, wo eine effiziente Umsetzung möglich ist, insbesondere als interne Parallelverarbeitung mittels Fork-Join (vgl. [GüLe12]).

Wir setzen nachstehend die neue Stream-Programmierschnittstelle des JDK 8 (`java.util.stream`) ein, die unter anderem die Higher-Order-Funktionen Map, Reduce und Filter bereitstellt (vgl. [CHOF]). `Stream.map` wendet die angegebene Operation auf jedes einzelne Element an. `Stream.filter` filtert einzelne Elemente des Streams anhand eines Kriteriums. `Stream.reduce` reduziert die Listenoperationen auf ein einzelnes Ergebnis. Ein Java-8-konformes Beispiel sähe demnach so aus:

```
1: public float averageGrade() {
2:   return this.exams
3:   .stream()
4:   .map((Function<Examination, Float>) (e) -> e.getGrade())
5:   .reduce(0.0f, (a, b) -> a + b) / this.exams.size();
6: }
```

Um die Vorzüge der Stream-Programmierschnittstelle zu nutzen, konvertieren wir die ursprüngliche Liste `this.exams` zunächst in `Stream<Examination>` (Zeile 3):

- `Stream.map` ist ein Transformator, der über alle Elemente des Streams iteriert und die übergebene `Function<T,R>` auf jedes Element anwendet (Zeile 4). Ergebnis dieser Operation ist ein `Stream<R>` (Ergebnistyp `R` der übergebenen Funktion). In unserem Beispiel extrahieren wir die Note mit `Examination.getGrade` und erhalten so einen `Stream<Float>`. Bei der Umwandlung des parametrierten Typs eines Streams kann der Compiler den Zieltypen nicht inferieren, wir müssen also den Lambda-Ausdruck explizit mit Typinformationen angeben.
- Die Liste aller Noten reduzieren wir mit `reduce` zu einem Ergebniswert (Zeile 5). `reduce` verknüpft paarweise zwei Elemente des Streams zu einer Zwischenlösung, die wiederum mit dem folgenden Element verknüpft wird, bis der Stream erschöpft ist. Das liefert die Summe über alle Noten, die wir danach durch die Anzahl der Bewertungen teilen.

Ein zweites Beispiel zeigt die Verwendung von Methodenreferenzen anstelle eines Lambda-Ausdrucks: `SchoolClass.topPupilsOfClass` liefert die nach dem Notendurchschnitt sortierte Liste der `k` besten Schüler einer Klasse:

```
1: public List<Pupil> topPupilsOfClass(int k) {
2:   return this.pupils
3:   .stream()
4:   .sorted(Pupil::compareByGrade)
5:   .limit(k)
6:   .collect(Collectors.toCollection(ArrayList<Pupil>::new));
7: }
```

`Stream<T>.sorted` erwartet einen `Comparator` als Parameter (Zeile 4). `Comparator` ist ein funktionales Interface mit der Methode `Comparator<T>.compare(T o1, T o2)`. Die Methodensignatur von `Pupil.compareByGrade` matcht die Methodensignatur eines `Comparator<Pupil>`. Der Aufruf von `Stream<T>.limit` holt aus dem Stream maximal `k` Elemente (Zeile 5). Streams arbeiten lazy und werden durch eine abschließende Operation (hier: `Stream<T>.collect`), die Elemente des Streams konsumiert, erst realisiert. `collect` nutzt einen übergebenen `Collector`, um den `Stream<Pupil>` wieder in eine `List<Pupil>` zu überführen (Zeile 6).

Fazit

Insgesamt ziehen wir ein positives Fazit und können feststellen, dass sich die neuen Sprach-Features von Java 8 in alltäglichen Anwendungsfällen gut anfühlen. Entwickler können mit Lambda-Ausdrücken deutlich ausdrucksstärkeren Code formulieren. Default-Methoden in Interfaces werden auch über die neuen Collections hinaus nützlich sein. Da die neuen funktionalen Sprach-Features in eine zwanzig Jahre alte Programmiersprache eingebaut werden, ist aber nicht überraschend, dass sich an einigen Stellen insbesondere Lambda-Ausdrücke nicht immer nahtlos einfügen. Für eine ausführlichere Diskussion zu den Änderungen in existierenden Programmierschnittstellen des JDK empfehlen wir die Lektüre von [Götz12], [Götz11b] und [LambdaFAQ].

Literatur

[CHOF] Common Higher Order Functions, `http://c2.com/cgi/wiki?CommonHigherOrderFunctions`

[Götz11a] B. Goetz, Interface evolution via virtual extension methods, 4th draft, Juni 2011 `http://cr.openjdk.java.net/~briangoetz/lambda/Defender%20Methods%20v4.pdf`

[Götz11b] B. Goetz, State of the Lambda, 4th edition, Dezember 2011, `http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-4.html`

[Götz12] B. Goetz, State of the Lambda: Libraries Edition, April 2012, `http://cr.openjdk.java.net/~briangoetz/lambda/collections-overview.html`

[GüLe12] M. Günther, M. Lehmann, Java 7: Das Fork-Join-Framework für mehr Performance, in: JavaSPEKTRUM, 05/2012

[JEP126] JDK Enhancement Proposal 126: Lambda Expressions & Virtual Extension Methods, `http://openjdk.java.net/jeps/126`

[JSR335] Java Specification Request 335: Lambda Expressions for the JavaTM Programming Language, `http://jcp.org/en/jsr/detail?id=335`

[Lambda] Project Lambda, `http://openjdk.java.net/projects/lambda/`

[LambdaFAQ] Maurice Naftalin's Lambda FAQ, `http://www.lambdafaq.org`

[LambdaKal] Lambda-Kalkül, `http://de.wikipedia.org/wiki/Lambda-Kalk%C3%BCl`

Markus Günther ist Senior Software Engineer bei der Accelerated Solutions GmbH und beschäftigt sich dort vornehmlich mit der Konzeption und Entwicklung Java-basierter Systeme.
E-Mail: guenther@accso.de

Martin Lehmann ist Cheftechnologe bei der Accelerated Solutions GmbH und in einem großen Java-Projekt für ein Konsortium internationaler Wetterdienste tätig.
E-Mail: lehmann@accso.de