Lambda-Ausdrücke in Java

Software Entwicklung 1

Annette Bieniusa, Mathias Weber, Peter Zeller

Mit Version 8 haben eine Reihe von Ideen aus der funktionalen Programmierung auch in Java Einzug gehalten. So wurde die Sprache um Lambda-Ausdrücke erweitert; diese heben die scharfe Trennung von Daten und Programmlogik auf. So ist es nun auch in Java möglich, Berechnung über Lambda-Ausdrücke zu parametrisieren und Funktionen höherer Ordnung zu verwenden. Durch die elegante und prägnante Schreibweise wird eine Vielzahl von typischen Berechnungen vereinfacht; darüberhinaus bieten sich Möglichkeiten zur Optimierung durch den Compiler und die Laufzeitumgebung (JVM). Die Einbettung von Lambda-Ausdrücken in Java 8 ist technisch recht komplex. Wir beschränken uns hier auf typische Verwendungsmustern, die wir anhand von Beispielen erläutern. Im Anhang zu diesem Kapitel finden Sie eine detaillierte Beschreibung, die auf die Implementierung eingeht.

1 Syntax von Lambda-Ausdrücken

Lambda-Ausdrücke in Java sind quasi Methoden ohne Namen. Sie bestehen aus einer Liste von formalen Parametern, einem Pfeil `->` und einem Funktionsrumpf. Im Gegensatz zu Methoden werden der Rückgabetyp und Exceptions nicht spezifiziert, sondern vom Compiler inferiert.
Die Syntax für Lambda-Ausdrücke ist sehr vielseitig; die folgende Tabelle zeigt typische Beispiele.

```
(int x)          ->  x + 1   //Parameter mit expliziter Typangabe
(x)              ->  x + 1   //Parameter ohne explizite Typangabe
x                ->  x + 1   //Parameter ohne explizite Typangabe
(int x, int y)   ->  x + y   //zwei Parameter mit expliziter Typangabe
(x, y)           ->  x + y   //zwei Parameter ohne explizite Typangabe
()               ->  5       //leere Parameterliste
```

Der Funktionsrumpf besteht entweder aus einem Ausdruck (wie in der Tabelle) oder aus einem Anweisungsblock mit abschließender `return`-Anweisung.[1]

[1] Der Anweisungsblock muss immer mit geschweiften Klammern umschlossen sein, auch wenn er nur aus einer `return`-Anweisung besteht: `x -> { return x+1; }`

```
(x, y) -> x * y
(x, y) -> { ... /* weitere Anweisungen */
            return x * y; }
```

Regeln:

- Bei einem Parameter ohne Typangabe können die Klammern weggelassen werden.
- Hat die Parameterliste Typangaben, so muss die Parameterliste in Klammern stehen.
- Die Parameternamen in Lambda-Ausdrücken dürfen nicht bereits als Variablen in der umschließenden Methode definiert sein.
- Im Funktionsrumpf kann auf Variablen einer umschließenden Methode zugreifen, diese müssen aber `final` sein bzw. dürfen nicht geändert werden.

2 Beispiele: Listen von Integern bearbeiten

Wir zeigen hier zunächst die Verwendung von Lambda-Ausdrücken an Beispielen. In den folgenden Abschnitten erläutern wir dann die Funktionsweise.

```
public static void listLambdas(List<Integer> list) {
    // Sammelt alle Elemente der Liste um 1 inkrementiert
    // in einer neuen Liste
    List<Integer> incremented = new ArrayList<Integer>();
    for(int n : list) {
      incremented.add(n+1);
    }

    // Variante mit Lambdas
    List<Integer> incremented2 = list.stream()
            .map(x -> x + 1)
            .collect(Collectors.toList());

    // Skaliert die Element in der Liste um den Faktor s
    // und addiert dazu jeweils t;
    // sammelt das Ergebnis in einer neuen Liste
    final int s = 2;
    final int t = 1;
    List<Integer> scaled = list.stream()
            .map(x -> s * x + t)
            .collect(Collectors.toList());

    // Filtert alle geraden Elemente aus der Liste
    // und sammelt sie in einer neuen Liste
    List<Integer> even = list.stream()
            .filter(x -> x % 2 == 0)
            .collect(Collectors.toList());


    // Summiert die Elemente in der Liste
```

```
    int summe_reduce = 0;
    for(int n : list) {
      summe_reduce += n;
    }
    // Variante mit Lambdas
    int summe_reduce2 = list.stream()
            .reduce(0, (a, b) -> a + b);
}
```

3 Streams

Ein ***Datenstrom*** (engl. *Stream*) ist eine (potentiell unendliche) Folge von Daten gleicher Art. Er wird von einer oder mehrerer Quellen mit Daten versorgt und erlaubt es, diese Daten der Reihe nach aus dem Strom zu entnehmen. Wir haben Datenströme bereits im Zusammenhang mit I/O eingeführt.
Die Ströme aus Java 8 verwenden das Stream-Konzept, um Datenverarbeitung deklarativ zu definieren. Im Fokus steht dabei, *was* mit den Elementen eines Streams passiert; offengelassen ist dabei zunächst, *wie* die Verarbeitung passiert (z.B. in welcher Reihenfolge).

```
public static void listStream(List<String> list) {
    Stream<String> s1 = list.stream();
    Stream<Integer> s2 = s1.map(x -> Integer.valueOf(x));
    Stream<Integer> s3 = s2.filter(x -> x > 0);
    long i = s3.count();
    System.out.println("i = " + i);
}
```

Bei Verarbeiten eines Datenstroms werden die Elemente dem Strom entnommen (vgl. `read()` in Kapitel 17. Die intermediären Operationen (engl. *intermediate operations*), wie `map` oder `filter`, bearbeiten die einzelnen Datenelemente und erzeugen aus dem Ergebnis einen neuen Datenstrom. Wie in den Beispielen gesehen, können intermediären Operatoren daher hintereinander geschaltet werden. Mittels einer terminalen Operation (engl. *terminal operation*) wird ein aggregiertes Ergebnis eines Datenstroms erzeugt. Dabei kann der Strom entweder zu einem einzelnen Wert reduziert werden (z.B. `count`), oder aber wiederum in eine Collections, ein Array, etc. umgewandelt werden.
In diesem Abschnitt stellen wir verschiedene Möglichkeiten vor Streams zu erzeugen, die Datenelemente zu verarbeiten und in einer terminalen Operation zu aggregieren.

Erzeugen von Streams Mit der Methode `stream()` wird ein (sequentieller) Stream erzeugt. Als Quelle können dienen:

- Collections

```
List<String> list = ....;
Stream<String> liststream = list.stream();

Map<Integer, String> map = ...;
Stream<Map.Entry<Integer, String>> entrystream = map.entrySet().
    stream();
```

```
Stream<Integer> keystream = map.keySet().stream();
```

- Arrays

```
String[] ar = ....;
Stream<String> arrstream = Arrays.stream(ar);
```

- Generatorfunktionen

```
Stream<Double> random = Stream.generate(() -> Math.random());
Stream<Double> seq    = Stream.iterate(1, x -> x + 1);
```

- I/O-Kanäle

```
try (Stream<String> filestream = Files.lines(Paths.get("data.txt")))
  {
    .... // Verwendung von filestream
  }
```

3.1 Operationen auf Streams

map wendet eine Funktion auf alle Elemente im Stream an und liefert den Ergebnis-Stream.

```
Stream<String> words    = ...
Stream<Integer> numbers = ...


// wandelt alle Strings in einem Stream in Kleinbuchstaben um:
words.map(s -> s.toLowerCase())
// Quadriert alle Zahlen im Stream:
numbers.map(x -> x*x)
```

filter filtert alle Elemente eines Streams, sodass nur die Elemente übrig bleiben, die eine gegebene Bedingung erfüllen.

```
// Liefert alle positiven Zahlen aus dem Stream:
numbers.filter(x -> x > 0)
// liefert die Strings, die das Wort "toll" enthalten
words.filter(s -> s.contains("toll"))
```

reduce wendet eine zweistellige Funktion auf die Elemente des Streams an und reduziert diese so zu einem einzelnen Wert. Wir haben sie in Kapitel 25 unter dem Namen `fold` kennengelernt. Dabei gibt es drei Varianten der Funktion:
Die erste Variante nimmt nur eine zweistellige Funktion und wendet sie paarweise auf die Elemente im Stream an. Wenn der Stream leer ist, gibt `reduce` einen leeren `Optional<T>` zurück, ansonsten einen `Optional<T>` mit dem Ergebnis vom Typ `T`. Mit der Methode `isPresent()` kann geschaut werden, ob ein `Optional<T>` ein Ergebnis hat. Mit `get()` kann gegebenenfalls das Ergebnis extrahiert werden. Da die Reihenfolge nicht definiert ist, sollte die Reduktionsfunktion assoziativ sein, damit das Ergebnis deterministisch ist.

```
// Aufsummieren der Zahlen im Stream:
numbers.reduce((x,y) -> x+y)
```

Frage 1: Gegeben eine Integer-Liste `list` mit Einträgen `3, 17, 9, 1, 0, -3`. Was liefern folgende Ausdrücke?

```
list.stream()
  .filter(x -> x < 10 && x > 0)
  .reduce((x,y) -> {if (x > y)
      then return x
      else return y;})

list.stream()
  .map(x -> 2 * x)
  .filter(x -> x % 2 == 1)
  .reduce((x,y) -> x + y)
```

Um den undefinierten Fall für den leeren Stream zu vermeiden, gibt es eine zweite Variante von `reduce`, welche zusätzlich einen Startwert nimmt. Diese Funktion liefert dann nicht mehr einen `Optional<T>`, sondern direkt das Ergebnis der Funktion. Dieser Startwert muss eine Identität im Bezug auf die gegebene Funktion sein.

```
// Aufsummieren der Zahlen im Stream:
numbers.reduce(0, (x,y) -> x+y);
// Aufmultiplizieren der Zahlen im Stream:
numbers.reduce(1, (x,y) -> x*y);
```

In manchen Fällen soll der Ergebnistyp von `reduce` sich vom Element-Typ des `Streams` unterscheiden. Für diesen Fall gibt es eine dritte Variante von `reduce` mit drei Parametern, auf die wir hier nicht näher eingehen.

collect gleicht der `reduce` Methode, verwendet allerdings eine andere Vorgehensweise zum Berechnen des Ergebnisses. Während `reduce` in jedem Schritt einen neuen Wert berechnet, verwendet `collect` veränderbare Objekte, um das Ergebnis aufzubauen.

```
// Stream von Strings in einer ArrayList speichern:
words.collect(
  () -> new ArrayList<String>(),
  (list, s) -> list.add(s),
  (list1, list2) -> list1.addAll(list2)
)
```

Diese drei Funktionen lassen sich auch in einem `Collector` zusammenfassen. Die `collect`-Methode wird häufig mit den vordefinierten `Collector`-Objekten aufgerufen, die in der Klasse `Collectors` definiert sind.

```
// Stream von Strings in einer Liste speichern:
List<String> l = words1.collect(Collectors.toList());
// Stream von Strings in einer Menge speichern:
Set<String> s = words2.collect(Collectors.toSet());
// Elemente durch Komma getrennt als String:
String str = words3.collect(Collectors.joining(", "));
```

forEach ruft eine Funktion ohne Ergebnis für jedes Element im Stream auf. Im Gegensatz zu `map` erzwingt `forEach` die Auswertung des Streams und liefert kein Ergebnis.

```
// Alle Strings im Stream ausgeben:
words.forEach(w -> {
    System.out.println(w);
});
```

anyMatch prüft, ob es ein Element im Stream gibt, welches die gegebene Bedingung erfüllt.

```
// gibt es einen leeren String im Stream?
boolean hasEmpty = words.anyMatch(s -> s.isEmpty());
// gibt es eine negative Zahl im Stream?
boolean hasNeg = numbers.anyMatch(x -> x < 0);
```

allMatch prüft, ob alle Elemente im Stream die gegebene Bedingung erfüllen.

```
// enthalten alle Strings im Stream ein "e"?
boolean allHaveE = words.allMatch(s -> s.contains("e"));
// sind alle Zahlen im Stream positiv?
boolean hasNeg = numbers.allMatch(x -> x >= 0);
```

3.2 Vergleichen und Sortieren mit Lambdas

In Kapitel 14 (Maps) haben wir bereits das Interface `Comparator` kennengelernt, welches zwei Objekte vergleichen kann. Dieses Interface kann zum Sortieren oder zum Verwalten von sortierten Datenstrukturen verwendet werden.

Mit Hilfe von Lambda-Ausdrücken lassen sich seit Java 8 `Comparator`-Objekte deutlich kompakter erstellen, ohne dass eine eigene Klasse implementiert werden muss. Beispielsweise lässt sich ein `Comparator` für Strings, welcher die Strings nach Ihrer Länge vergleicht wie folgt definieren:

```
Comparator<String> nachLaenge = (x, y) -> {
    if (x.length() > y.length()) {
        return 1;
    } else if (x.length() < y.length()) {
        return -1;
    } else {
        return x.compareTo(y);
    }
};
```

Wir betrachten zunächst zwei Methoden zum Sortieren, mit denen `Comparator` wie `nachLaenge` verwendet werden können:

sort ist eine statische Methode der Klasse `java.util.Collections`. Sie nimmt eine Liste und einen `Comparator` und sortiert die gegebene Liste aufsteigend.

```
// Liste nach Länge der Strings sortieren:
List<String> list = Arrays.asList("aaaa", "b", "ccc", "dd", "e");
Collections.sort(list, nachLaenge);
System.out.println(list);
// Ausgabe: [b, dd, ccc, aaaa]
```

sorted ist eine Methode auf Streams. Sie nimmt einen `Comparator` und liefert einen `Stream`, in dem die Werte sortiert sind.

```
List<String> list = Arrays.asList("aaaa", "b", "ccc", "dd", "e");
List<String> sorted = list.stream()
        .sorted(nachLaenge)
        .collect(Collectors.toList());
System.out.println(list); // Ausgabe: [aaaa, b, ccc, dd, e]
System.out.println(sorted); // Ausgabe: [b, dd, ccc, aaaa]
```

Neben der Möglichkeit, einen `Comparator` über eine Klasse oder einen Lambda-Ausdruck zu definieren, gibt es noch einige Methoden zum Definieren von oft verwendeten `Comparator`n:

naturalOrder ist eine statische Methode von `Comparator`, welche die natürliche Ordnung auf dem gegebenen Typ darstellt. Diese Methode ist nur für Subtypen von `Comparable` anwendbar.

```
// Für Strings ist die natürliche Ordnung die lexikographische
Comparator<String> c1 = Comparator.naturalOrder();
```

reverseOrder ist eine Methode von `Comparator` und liefert die Ordnung in umgekehrter Reihenfolge.

```
List<String> list = Arrays.asList("Amsel", "Bär", "Ameise", "Schimpanse");
Comparator<String> lexikographisch = Comparator.naturalOrder();
Collections.sort(list, lexikographisch);
System.out.println(list); // [Ameise, Amsel, Bär, Schimpanse]
Collections.sort(list, lexikographisch.reversed());
System.out.println(list); // [Schimpanse, Bär, Amsel, Ameise]
```

comparing ist eine statische Methode von `Comparator`. Sie nimmt eine Funktion `f` und liefert die Ordnung, die zwei Werte `x` und `y` vergleicht, indem sie `f(x)` und `f(y)` vergleicht. Um `f(x)` und `f(y)` zu vergleichenn kann ein weiterer `Comparator` angegeben oder die natürliche Ordnung verwendet werden.

```
// Strings nach ihrer Länge vergleichen:
Comparator<String> nachLaenge = Comparator.comparing(s -> s.length());
```

thenComparing ist eine Methode auf `Comparator`, welche die aktuelle Ordnung nimmt und diese erweitert, so dass bei gleichen Elementen eine weitere Ordnung verwendet wird. Die zweite Ordnung kann entweder ein `Comparator` sein, oder wie bei `comparing` eine Auswahl-Funktion mit einem optionalen `Comparator`.

```
// Strings erst nach ihrer Länge
// und dann nach ihrer natürlichen Ordnung vergleichen:
Comparator<String> nachLaenge = Comparator
        .comparing((String s) -> s.length())
        .thenComparing(Comparator.naturalOrder());
```

Das folgende Beispiel greift Aufgabe 2 von Blatt 9 auf und zeigt, wie mit den gezeigten Methoden kompliziertere Ordnungen definiert werden können:

```
// Orte erst nach ihrer Postleitzahl und dann nach ihrem Namen
    sortieren:
Comparator<Ort> ortVergleicher =
    Comparator
        .comparing((Ort o) -> o.getPostleitzahl())
        .thenComparing(o -> o.getName());

// Addressen erst nach ihrem Ort mit vergleichen (mit dem
    ortVergleicher)
// Bei gleichem Ort nach Straße und dann nach Hausnummer ordnen:
Comparator<Addresse> addressVergleicher =
    Comparator
        .comparing((Addresse a) -> a.getOrt(), ortVergleicher)
        .thenComparing(a -> a.getStrasse())
        .thenComparing(a -> a.getHausNummer());
```

Hierbei ist zu beachten, dass beim ersten Lambda-Ausdruck der Parametertyp angegeben werden **muss**, da Java hier den Typ nicht automatisch bestimmen kann.

4 Auswertung von Streamoperationen und Seiteneffekte

Die Kombination von Streams und Lambdas ermöglicht deklaratives Programmieren in Java: Wir spezifizieren, *was* mit den Elemente eines Streams passiert; offengelassen ist dabei zunächst, *wie* die Verarbeitung passiert. Der direkte Vergleich zwischen den verschiedenen Möglichkeiten der Iteration über die Elemente einer Liste zeigt den Unterschied.

```
9     public static void listIteration(Collection<Integer> coll) {
10        // Variante 1: Iteration mit Iterator
11        Iterator<Integer> it = coll.iterator();
12        while (it.hasNext()) {
13            int i = it.next();
14            System.out.print(i + " ");
15        }
16        // Variante 2: Explizites Iterieren
17        for (int i : coll) {
18            System.out.print(i + " ");
19        }
20        // Variante 3: Implizites Iterieren
21        coll.stream().forEach(i -> System.out.print(i + " "));
22    }
```

Variante 1 verwendet Iteratoren, die geordnete Collections (wie Listen) in einer festlegten Reihenfolge die einzelnen Elemente der Liste ablaufen. Variante 2 ist lediglich eine syntaktische Variation dieses Iteratorschemas. In der letzten Variante unter der Verwendung von `forEach` ist die Reihenfolge der Iteration (auch bei geordneten Collections) nicht festgelegt. Dies kann zu unerwartetem Verhalten des Programms führen! Warum ist dies so?

Funktionen sind in der rein-funktionalen Programmierung frei von Seiteneffekten. Sie abstrahieren von Ausdrücken und modifizieren typischerweise den Programmzustand nicht (kein Verändern von Attributen, globalen Variablen, Konsole, ...). Das Ergebnis (d.h. der Effekt) der Funktionsauswertung hängt allein von den Parametern ab; bei gleichen Parameterwerten liefert eine Funktion immer das gleiche Ergebnis.

Methoden wie in Java abstrahieren hingegen von Anweisungssequenzen. Sie können Parameter haben; diese beeinflußen die Haupt- und Seiteneffekte, die bei der Ausführung der Methode verursacht werden.

Die Konzepte “Methode” und “Funktion” werden in Java nicht scharf von einander getrennt. Methoden, die keine Seiteneffekte verursachen, können als Funktionen aufgefasst werden; andererseits wird nicht erzwungen, dass Lambda-Ausdrücke (die ja quasi anonyme Methoden sind) seiteneffektfrei sein müssen.

Als Programmierer müssen Sie daher darauf achten, Streams möglichst seiteneffektfrei zu verarbeiten. Bei der gewollten Verwendung von Seiteneffekten, wie dem Ausgeben von Elementen auf Konsole, sollten geordnete oder sortierte Streams verwendet werden, um eine deterministische Programmausführung zu garantieren.

```
Arrays.asList("Amsel", "Drossel", "Affe", "Dromedar").stream()
    .filter(s -> s.startsWith("D"))
    .map(s-> s.toUpperCase())
    .forEachOrdered(s -> System.out.println(s));
```

Die Auswertung von intermediären Operatoren erfolgt *lazy*, d.h. die Auswertung erfolgt nur dann, wenn das Ergebnis zwingend erforderlich ist. Dies erlaubt es beispielsweise mit unendlichen Datenströmen zu arbeiten.

```
List<Integer> seq = Stream
  .iterate(1,x -> x+1)
  .limit(5)   // Nehme nur die ersten 5 Elemente aus dem Stream
  .collect(Collectors.toList());
```

Die Verarbeitung von Datenströmen muss außerdem durch eine terminale Operation abgeschlossen werden, da erst durch diese terminale Operation die Verarbeitung des Stroms erzwungen wird.

Frage 2: Im folgenden Beispiel wurden `println`-Anweisungen in die `map`- und `filter`-Operationen eingebunden, um die Ausführung nachzuvollziehen. Was ist die Ausgabe des Programms?

```
List<String> list = Arrays.asList("Apfel", "Birne", "Kiwi");
String res = list.stream()
        .map(o -> {
            System.out.println("map " + o);
            return o.toLowerCase();
        })
        .filter(o -> {
            System.out.println("filter " + o);
            return o.startsWith("b");
        })
        .findFirst()
        .get();
System.out.println("res = " + res);
```

```
Arrays.asList("a", "b", "c").stream()
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    });

// Liefert keine Ausgabe, da terminale Operation fehlt!
```

Streams können nicht "wiederverwendet" werden.

```
List<String> l = Arrays.asList("a", "b", "c");

Stream<String> s1 = l.stream();
Stream<String> s2 = s1.map(s -> "!");
Stream<String> s3 = s1.map(s -> "?");
// IllegalStateException: stream has already been operated upon or
  closed
```

Nach der terminalen Operation ist der Stream auch nicht mehr verfügbar.

5 Zusammenfassung und Ausblick

Lambda-Ausdrücke erlauben es knapp und prägnant in Java zu programmieren. Syntaktische Erweiterungen wie Methodenreferenzen (hier nicht weitervertieft) führen dazu, dass Programme in Java immer stärker ihren prozeduralen Charakter verlieren:

```
// Orte erst nach ihrer Postleitzahl und dann nach ihrem Namen sortieren:
Comparator<Ort> ortVergleicher =
    Comparator
        .comparing(Ort::getPostleitzahl)
        .thenComparing(Ort::getName);

// Addressen erst nach ihrem Ort mit vergleichen (mit dem ortVergleicher);
// bei gleichem Ort nach Straße und dann nach Hausnummer ordnen:
Comparator<Addresse> addressVergleicher =
    Comparator
        .comparing(Addresse::getOrt, ortVergleicher)
        .thenComparing(Addresse::getStrasse)
        .thenComparing(Addresse::getHausNummer);
```

Die Einführung von Streams in Java 8 hatte als weiteres wichtiges Ziel die Parallelisierung von Berechnungen. Moderne Prozessoren haben eine Mehrkern-Architektur (engl. *multi-core processors*). Berechnungen, die unabhängig voneinander sind, können auf verschiedenen Prozessoren gleichzeitig durchgeführt werden und so die gesamte Berechnungszeit reduziert werden.
Java 8 bietet mit parallelen Streams eine vergleichsweise einfache Möglichkeit Berechnungen zu parallelisieren. Auch hier sind Seiteneffekte zu vermeiden, da die Reihenfolge der Effekte bei gleichzeitiger Ausführung nichtdeterministisch ist und es zu Resourcenkonflikten kommen kann. Die Aufteilung von Berechnungen in unabhängige Teilberechnungen und die Synchronisierung von Prozessen ist nicht-trivial; diese Thematik wird vertieft in der Veranstaltung SE3 behandelt.

Anhang: Behind the Scenes: Wie funktionieren Lambda-Ausdrücke in Java?

Bei der Einführung von Lambda-Ausdrücken in Java hat man nach Möglichkeiten gesucht, diese Konstrukte in das existierende Typsystem zu integrieren. Man entschied letztendlich die Lambdas auf Interface-Typen, die nur eine einzige Methode erfordern, abzubilden (***funktionale Interfaces***, engl. *functional interface types* oder auch *single abstract method types*). Dieser Typ wird vom Programmierer nicht spezifiziert, sondern vom Compiler aus dem Kontext abgeleitet.
Hier eine Auswahl der funktionalen Interfaces (aus dem Paket `java.util.function`):

Interface	Lambda-Ausdruck	Auswertungsmethode	Bemerkung
`Function<T,R>`	`x -> x+1;`	`apply(T x)`	ein Parameter vom Typ `T`, Rückgabewert vom Typ `R`
`BiFunction<T,U,R>`	`(x,y) -> x*y;`	`apply(T x, U y)`	zwei Parameter vom Typ `T` und `U`, Rückgabewert vom Typ `R`
`BinaryOperator<T>`	`(x,y) -> x-y;`	`apply(T x, T y)`	zwei Parameter und Rückgabewert vom Typ `T`
`Predicate<T>`	`x -> x<0;`	`test(T x)`	ein Parameter vom Typ `T`, Rückgabewert vom Typ `boolean`
`Consumer<T>`	`x -> return;`	`accept(T x)`	ein Parameter, kein Rückgabewert

Ein Lambda-Ausdruck kann über eine Variable des entsprechenden Funktionstypen referenziert werden, um dann später aufgerufen und verwendet zu werden.

```
1  import java.util.function.*;
2
3  public class Funktionen {
4
5      public static void functionReferences() {
6          int a = 3;
7          int b = 4;
8
9          Function<Integer, Integer> f = (x) -> x * x - 1;
10         BiFunction<Integer, Integer, Integer> g =
11           (x, y) -> x * x + 2 * x - 1;
12
13         System.out.println("f(" + a + ") = " + f.apply(a));
14         System.out.println("g(" + a + "," + b + ") = " + g.apply(a, b));
15     }
16
17 }
```

Im Funktionsrumpf hat man Zugriff auf die Parameter und lokale Variablen des Lambda-Ausdrucks. Auch auf die unveränderlichen Variablen des Kontexts kann man zugreifen. Folgendes Beispiel zeigt, warum diese Variablen nicht mehr verändert werden dürfen:

```
public void example() {
    List<Function<Integer, Integer>> functions = new ArrayList<>();

    for (int i=0; i<10; i++) {
```

```
            int y = i;
            // aktueller Wert von y wird in der Funktion gespeichert
            functions.add(x -> x * y);
        }

        for (Function<Integer, Integer> f : functions) {
            System.out.println(f.apply(f.apply(2)));
        }
    }
```

Würde `y` nach der Verwendung im Lambda-Ausdruck verändert, wäre nicht klar, ob sich der Wert im Lambda-Ausdruck auch ändern soll.[2] Ebenso darf auf die Attribute der umschließenden Klasse zugegriffen werden (ähnlich wie bei inneren Klassen). Attribute dürfen verändert werden, da lediglich die unveränderbare Referenz `this` gespeichert wird.

Operationen auf Streams

Für die Stream-Operationen ergeben sich damit die folgenden Typen:

```
Stream<T> filter(Predicate<? super T> p)
Stream<R> map(Function<? super T,? extends R> mapper)
T reduce(T start, BinaryOperator<T> f)
R collect(Collector<? super T,A,R> c)
void forEach(Consumer<? super T> a)
```

Um die Verwendung der Funktionen flexibler zu gestalten, wurden hier ***Wildcard-Typen*** (`?`) mit Einschränkungen (`? super T` und `? extends T`) verwendet. Für `? super T` kann jeder Supertyp von T eingesetzt werden, für `? extends T` jeder Subtyp von T. Dies erlaubt es zum Beispiel, bei `map` eine Funktion anzugeben, die einen allgemeineren Typen erwartet und einen spezielleren Typen zurückgibt. Analoges gilt für die anderen Operationen.

```
List<String> list = Arrays.asList("a", "b", "c");
Stream<String> stream = list.stream();
Function<Object, Integer> f = o -> o.hashCode();
Stream<Number> numberStream = stream.map(f);
```

Ein Collector ist parametrisiert über die Elemente, die in der Reduktion verwendet werden (`T`), den Typ der Akkumulation bei der Reduktion (`A`, oft offengelassen), sowie dem Ergebnistyp der Reduktionsoperation (`R`). So ist beispielsweise der Collector `Collectors.toList()` vom Typ `Collector<T,?,List<T>>`.

Anwendungsbeispiel: Klausurergebnisse

Im folgenden Beispiel sehen wir die bisher vorgestellten und weitere intermediäre und terminale Operationen.

[2] Dies ist in anderen Programmiersprachen anders gelöst, z.B. bei Closures in JavaScript.

```java
import java.util.*;
import java.util.stream.*;

public class ExamResults {
    private class Submission {
        String name;
        int[] points;
        int sum;

        Submission(String name, int[] points) {
            this.name = name;
            // Erstelle Kopie des Punkte-Arrays -> Defensives Vorgehen
            this.points = Arrays.copyOf(points, points.length);
            // Summiert die Punktezahl
            this.sum = Arrays.stream(points).sum();
        }
    }

    private List<Submission> results = new ArrayList<Submission>();

    // Fuegt eine neues Klausurergebnis hinzu
    public void addSubmission(String name, int[] points) {
        results.add(new Submission(name, points));
    }

    // Ermittelt die Namen aller, die bestanden haben
    // D.h. deren Gesamtpunkzahl groesser als ein Schwellwert ist
    public List<String> getPassed(final int threshold) {
        return results.stream()
                .filter(s -> s.sum >= threshold)
                .map(s -> s.name)
                .collect(Collectors.toList());
    }

    // Ermittelt den Mittelwert der Klausurergebnisse
    // Löst eine RuntimeException aus, wenn die List der
    Klausurergebnisse leer ist
    public double getAverage() {
      try {
        return results.stream()
                .mapToInt(s -> s.sum)
                .average()
                .getAsDouble();
      } catch (NoSuchElementException e) {
        throw new RuntimeException("Mittelwert kann nicht ermittelt
    werden");
      }
    }

    // Liefert eine Liste mit den Namen der Klausurteilnehmer,
    // aufsteigend sortiert nach erreichter Gesamtpunktzahl,
    // bei Punktegleichstand nach Name
    public List<String> getSorted() {
      Comparator<Submission> nachErgebnis = Comparator
      .comparing((Submission s) -> s.sum)
      .thenComparing((Submission s) -> s.name);
```

```
55
56          return results.stream()
57                  .sorted(nachErgebnis)
58                  .map((Submission s) -> s.name)
59                  .collect(Collectors.toList());
60      }
61  }
```

Frage 3: Implementieren Sie zum Vergleich die Methode `getPassed`, `getAverage` und `getSorted` ohne die Verwendung von Streams und Lambda-Ausdrücken!