entwickler.press
shortcuts

Java 9 Streams

Sven Ruppert

Sven Ruppert

Java 9 Streams

ISBN: 978-3-86802-769-3

Ein Imprint der Software & Support Media GmbH

1 Funktionale Programmierung in Java 9

Seit Java 8 ist das Stream-API im JDK verfügbar und es erfreut sich großer Beliebtheit. Im Zusammenhang mit den ebenfalls in Java 8 dazugekommenen Lambdas ergibt sich ein recht großes Spektrum an Funktionalität und Struktur, was bei vielen Entwicklern dazu geführt hat, sich sowohl mit den Aspekten der funktionalen als auch der reaktiven Programmierung auseinanderzusetzen.

1.1 Was waren nochmal diese Streams?

Jeder ist in Java schon einmal mit irgendeiner Form von Streams konfrontiert worden. Aber was genau macht einen Stream nun im JDK 8/9 aus? Was bringen Streams dem Entwickler und wie werden sie verwendet? Wie kann man mit funktionalen Aspekten beginnen, ohne die Sprache zu wechseln, und wird für reaktive Programmierung wirklich ein großes Framework benötigt? Wir werden uns nun Schritt für Schritt diesen Themen zuwenden und zeigen, dass all die erforderlichen Dinge im Core JDK 9 enthalten sind.

Es gibt einige Dinge, die einen Stream von einer Datenstruktur unterscheiden:

Zusätzlich zu den Quelltextbeispielen im Text verwende ich auch die Sources des Open-Source-Projekts Functional Reactive with Core Java[1]. Diese befinden sich unter https://github.com/functional-reactive/functional-reactive-lib/.

- Streams sind keine Datenstruktur, was bedeutet, dass sie keinen Storage für die Daten selbst darstellen. Es handelt sich eher um eine Pipeline für Datenströme. In dieser Pipeline werden verschiedene Transformationen auf die Daten angewandt. In diesem speziellen Fall werden die Transformationen nicht auf den Daten der Quellstruktur selbst durchgeführt. Die zugrunde liegenden Datenstrukturen wie Arrays oder Listen werden also nicht verändert, aber bei den enthaltenen Elementen ist Vorsicht geboten. Ein Stream wrappt also die Datenstruktur, entnimmt ihr die Quelldaten und arbeitet auf diesen Elementen.
- Streams wurden für den Einsatz von Lambdas konzipiert. Es gibt also keine Streams ohne Lambdas, was kein Problem darstellt, da Streams und Lambdas seit Version 8 zusammen im JDK enthalten sind.
- Streams bieten keinen wahlfreien Zugriff per Index oder Ähnlichem auf die Quelldaten.

Der Zugriff auf das erste Element ist möglich, nicht jedoch auf beliebige nachfolgende Elemente. Das bedeutet, dass auch bei der Verarbeitung aller Daten erst einmal davon ausgegangen werden muss, dass wir immer nur Zugriff auf das gerade verarbeitete Element haben. Jede Stufe der Verarbeitung sollte demnach zustandslos sein.

- Streams bieten guten Support, um die Ergebnisse selbst wieder als z. B. Array oder List zur Verfügung zu stellen. Damit wird es einem Entwickler sehr einfach gemacht, zwischen diesen beiden Welten hin und her zu wechseln.
- Streams sind „lazy“ organisiert. Das bedeutet, dass die Elemente erst dann geholt werden, wenn die Operation auf ihnen angewandt werden soll. Besteht die Datenquelle aus tausend Elementen, dann dauert der erste Zugriff nicht tausend Zeiteinheiten, sondern eine Zeiteinheit (vorausgesetzt, der Zugriff auf ein Element ist im Zeitverbrauch linear).
- Streams sind parallel, jedoch nur wenn gewünscht. Streams kann man prinzipiell in zwei Hauptgruppen unterteilen: in serielle und parallele Implementierungen. Wenn also Operationen atomar und ohne Invarianten durchgeführt werden können, wird kein typischer Multi-Threaded-Code notwendig, um die im System schlafenden Cores zu verwenden.
- Streams sind ungebunden, da sie nicht wie Collections initial gefüllt werden. Damit können Streams auch unendlich sein. Es können Generatorfunktionen angegeben werden, die für die permanente Lieferung von Quelldaten sorgen, die erzeugt werden, wenn der Client Elemente des Streams konsumiert.
- Streams werden konsumiert. Wird ein Stream einmal vollständig konsumiert bzw. terminiert, so kann er kein zweites Mal konsumiert werden. Es muss also ein neuer Stream derselben Struktur erzeugt werden.

1.2 Functional Interfaces

Mit Java 8 bekamen wir die Functional Interfaces, die dann in Java 9 erweitert wurden. Diese sind ein oft verwendetes technisches Mittel, um funktionale Elemente in Java zu realisieren. Zur Wiederholung seien hier nochmals einige Kleinigkeiten hinsichtlich der Interfaces und der Änderungen in Java 9 dargestellt.

Wir haben ein Interface mit zwei Methoden. Eine davon, doMore(), ist statisch, die andere, doSomething(), hat eine default-Implementierung (Listing 1.1).

```
public interface DemoInterface {
  static void doMore() {
    System.out.println(DemoInterface.class.getSimpleName() + " : " + "doMore");
```

```
  }

  default void doSomething() {
    System.out.println(DemoInterface.class.getSimpleName() + " : " + "doSomething")
;
  }
}
```

Listing 1.1

Nun erstellen wir ein Interface mit dem Namen InterfaceA und erben von DemoInterface. Hier überschreiben wir die Methode doSomething() (Listing 1.2).

```
public interface InterfaceA extends DemoInterface {
  default void doSomething() {
    System.out.println(InterfaceA.class.getSimpleName() + " : " + "doSomething");
  }
}
```

Listing 1.2

Zuletzt erstellen wir ein Interface mit dem Namen InterfaceB, das nicht von DemoInterface erbt, aber auch eine Methode mit dem Namen doSomething und einer default-Implementierung hat.

Wenn wir nun den in Listing 1.3 gezeigten Quelltext schreiben und ausführen, was erhalten wir dann für Meldungen auf der Konsole?

```
public static void main(String[] args) {
  DemoInterface.doMore();
  new ImplA().doSomething();
  ImplA.doMore();
  new ImplB().doSomething();
}
```

Listing 1.3

Das Erste, was passiert, ist ein Abbruch der Übersetzung. Die Zeile ImplA.doMore() wird mit einem cannot find symbol "doMore()" durch den Compiler quittiert. Statische Methoden können also nur auf dem originalen Interface aufgerufen werden. Leider wird das gerne vergessen und erst später bei der Implementierung bemerkt. Wenn wir also die entsprechende Zeile auskommentieren, können wir den Quelltext ausführen und erhalten folgende Ausgabe:

```
DemoInterface : doMore
InterfaceA : doSomething
ImplB : doSomething
```

Es wird also immer die letzte Implementierung aus der Vererbungskette verwendet. Im Fall von InterfaceB kann der Compiler nicht selbst entscheiden, welche der beiden Implemetierungen die dominantere ist. Hier muss der Entwickler mittels Implementierung selbst die Entscheidung

treffen.

In Java 9 können wir nun Methoden zusätzlich mit der Sichtbarkeit private deklarieren bzw. implementieren. Eine reine Deklaration ist nicht möglich, da diese auch in erbenden Interfaces nicht mit einer Implementierung versehen werden könnte. Nun sind wir aber in der Lage, größere default-Implementierungen wieder sauber in mehrere Methoden innerhalb eines Interface aufzuteilen. In Java 8 musste das entweder ausgelagert werden oder führte zu unschönen Quelltextfragmenten.

Kommen wir nun zu den Functional Interfaces. Ihre Definition ist in Java 9 immer noch die gleiche wie zu Java-8-Zeiten. Es handelt sich um ein Functional Interface, wenn eine noch zu implementierende Methode vorhanden ist. Die Anzahl der Methoden mit default-Implementierung hat hier keinen Einfluss, ebenso wenig, ob die Annotation @FunctionalInterface vorhanden ist oder nicht.

1.3 Lambdas

Ich werde hier nicht auf alle möglichen Formen eingehen, sondern nur darstellen, dass überall dort, wo ein Functional Interface als Attribut übergeben werden kann, auch ein Lambda verwendet wird. Die Verwendung von Lambdas werden des Öfteren mit funktionaler Programmierung gleichgesetzt, aber ist das wirklich so? Sehen wir uns dazu das Beispiel in Listing 1.4 an.

```
public class Main {

  @FunctionalInterface
  public static interface Service {
    public Integer doWork(String value);
  }

  public static class ServiceImpl implements Service {
    @Override
    public Integer doWork(String value) {
      return Integer.valueOf(value);
    }
  }

  public static void main(String[] args) {
    Integer integer = new ServiceImpl().doWork("1");
    Service serviceA = (value)-> { return Integer.valueOf(value);};
    Service serviceB = Integer::valueOf;
    serviceA.doWork("1");
    serviceB.doWork("1");
  }
}
```

Listing 1.4

Hier haben wir ein Interface mit dem Namen Service, das einmal klassisch durch die Klasse ServiceImpl implementiert worden ist. Die Verwendung erfolgt nun, wie in Java üblich, indem eine Instanz der Klasse erzeugt wird, um nachfolgend die Methode aufzurufen.

Aber auch die beiden anderen Implementierungen, erst mittels Lambda und danach mittels Method Reference, können wie gewohnt verwendet werden. Nur haben wir dann einen etwas anderen Lebenszyklus bei der Initialisierung, und es kann kein Klassenattribut (Zustand) wie bei der ersten klassischen Implementierung geben.

Das kann man schon in den Bereich der funktionalen Aspekte aufnehmen: Ein Wert wird eingegeben, ein Wert wird ausgegeben und es gibt keinen Zustand, der einem etwas verderben kann. Die Unterschiede zwischen Lambda und Method Reference lasse ich hier bewusst unerwähnt.

Nun kann es ja sein, das die Methode mit dem Eingangswert "HoppelPoppel" aufgerufen wird. Was soll die Antwort der Methode sein? Entweder eine Exception oder ein Null? Beides widerspricht den grundsätzlichen Überlegungen der funktionalen Programmierung.

1.4 Optional<T>

Sehen wir uns die seit Java 8 verfügbare Klasse Optional<T> an. Sie dient dazu, eine Referenz zurückzugeben, selbst wenn der Wert, um den es letztendlich geht, nicht verfügbar ist. Anstelle von null oder dem Wert bekommt man nun eine Instanz der Klasse Optional. Hierzu schreiben wir das Interface wie in Listing 1.5 um (die klassische Implementierung lasse ich hier aus, sie ist im Git Repository unter https://github.com/functional-reactive/functional-reactive-lib/ zu finden):

```
@FunctionalInterface
public static interface Service {
  public Optional<Integer> doWork(String value);
}

public static class ServiceImpl implements Service {
  @Override
  public Optional<Integer> doWork(String value) {
    try {
      return Optional.of(Integer.valueOf(value));
    } catch (NumberFormatException e) {
      e.printStackTrace();
    }
    return Optional.empty();
  }
}

public static void main(String[] args) {
  Optional<Integer> integer = new ServiceImpl().doWork("1");
```

```
  Service serviceA = (value) -> {
    try {
      return Optional.of(Integer.valueOf(value));
    } catch (NumberFormatException e) {
      e.printStackTrace();
    }
    return Optional.empty();
  };

//SNIPP usage of the implementations
}
```

Listing 1.5

Es werden also immer Werte übergeben, bzw. es wird nie null als Rückgabewert verwendet. Auch wurde kein Wert aus dem Wertevorrat der Zielmenge, hier Integer, dazu verwendet, einen Fehler zu signalisieren. Die RuntimeException muss auch nicht mehr bei der Verwendung gefangen werden. Wir haben also ein wesentlich robusteres Stück Quelltext geschrieben.

Wenn wir uns nun die Verwendung der Instanz Optional ansehen, kommen wir einen Schritt weiter in Richtung funktionaler Aspekte. Wir können die Methode in eine Function<String, Optional<Integer>> umschreiben und das als Rückgabewert einer statischen Methode implementieren (Listing 1.6).

```
@FunctionalInterface
public static interface Service {
  public Optional<Integer> doWork(String value);
}

static Function<String, Optional<Integer>> service() {
  return (value) -> {
    try {
      return Optional.of(Integer.valueOf(value));
    } catch (NumberFormatException e) {
      e.printStackTrace();
    }
    return Optional.empty();
  };
}

public static void main(String[] args) {
  Optional<Integer> apply = service().apply("1");
}
```

Listing 1.6

Nun haben wir wieder eine Funktion, aber den Bezug zur fachlichen Ausprägung Service verloren. Bei Bedarf gibt es hier wieder die Möglichkeit, das mittels Vererbung in den fachlichen Namensbereich Service zu heben (Listing 1.7).

```
public class Main {

  @FunctionalInterface
  public static interface Service extends Function<String, Optional<Integer>> {
    public default Optional<Integer> doWork(String value) {
      return apply(value);
    }
  }

  static Service service() {
    return (value) -> {
      try {
        return Optional.of(Integer.valueOf(value));
      } catch (NumberFormatException e) {
        e.printStackTrace();
      }
      return Optional.empty();
    };
  }

  public static void main(String[] args) {

    Optional<Integer> applyA = service().apply("1");

    Optional<Integer> applyB = service().doWork("1");
  }
}
```

Listing 1.7

Dabei ist zu beachten, dass die Methode mittels default-Implementierung an die Methode apply delegiert und wir dadurch wieder ein Functional Interface erhalten. Der Rückgabewert kann jetzt unterschiedlich verarbeitet werden.

Nun haben wir ein Optional, in das ein Integer gepackt ist. Was ist damit zu tun? Lassen Sie uns die Möglichkeiten erkunden, die uns ein Optional bietet (Listing 1.8).

```
Optional<Integer> apply = service().doWork("1");

final Integer intA = apply.get();
//if(apply.isPresent()){
if (intA != null) {
  //do something useful
}
else {
  //do something ???
}
```

Listing 1.8

Natürlich können wir den enthaltenen Wert einfach dem Optional entnehmen. Danach sind wir aber wieder genau an dem Punkt angelangt, an dem wir begonnen haben. Wir müssen entweder

den Wert an sich auf null prüfen oder das Optional selbst mittels der Methode isPresent() daraufhin befragen, ob ein Wert enthalten ist. Das Ziel ist jedoch, diese direkten Kontrollstrukturen durch funktionale Aspekte zu ersetzen (Listing 1.9).

```
final Integer intB1 = apply.orElseGet(() -> - 1);
final Integer intB2 = apply.orElse(- 1);
final Integer intB3 = apply.orElseThrow(() -> new RuntimeException("panic ;-)"));
```

Listing 1.9

Ein Optional kann auch mit Alternativwerten versehen werden oder eine Exception werfen, wenn kein Wert vorhanden ist. Letzteres ist jedoch mit Bedacht zu tun, da wir erst zur Laufzeit wieder damit konfrontiert werden. Schließlich kann man eine Aktion auch nur mit dem Vorhandensein eines Werts verknüpfen (Listing 1.10).

```
apply.ifPresent(v -> {
  System.out.println("v = " + v);
});
```

Listing 1.10

Bei der Verwendung von Java 8 fiel schnell auf, dass es hierbei einige Unzulänglichkeiten gab. Zum Beispiel fehlte die Möglichkeit, zusätzlich zur Aktion beim Vorhandensein eines Werts auch eine Aktion zu definieren, wenn eben kein Wert vorhanden ist. Das wurde in Java 9 ermöglicht (Listing 1.11).

```
//since 9

apply.ifPresentOrElse(v -> {
  System.out.println("v = " + v);
} , () -> {
  System.out.println("value not present");
});
```

Listing 1.11

Auch wurde die Kombination ifPresent() und get() oft verwendet. Deswegen gibt es in Java 9 nun einen stream(), der einen Stream<T> vom Optional<T> liefert:

```
final Stream<Integer> stream = apply.stream();
```

Ebenso wurde eine Möglichkeit für eine Alternative eingebaut. Diesmal aber nicht für den Wert, sondern als Optional. Der Typ T kann dabei allerdings nicht geändert werden. Die Alternative wird geliefert, wenn die Methode isPresent() false liefert:

```
apply.or(() -> Optional.of(Integer.MAX_VALUE));
```

1.5 Zusammenfassung

Fasst man das alles zusammen, ergeben sich daraus verschiedene Ansatzpunkte, die man im Arbeitsalltag anwenden kann. Zum Beispiel kann man APIs mit einem Optional versehen, das an der Stelle der NullPointerException entgegenarbeitet.

Die Verwendung von Optional<T> führt auch dazu, dass man Kontrollstrukturen anders formulieren kann. Es sind keine imperativen Fallunterscheidungen nötig, was definitiv in Richtung funktionaler Entwicklung geht.

Aber wenn man ein wenig darüber nachdenkt, kann man mit der map()-Methode bei einem Optional auch noch etwas anderes anstellen (Listing 1.12).

```
String value = "";
service()
  .doWork(value)
  .or(() -> Optional.of(0))  // per definition
  .map((Function<Integer, Function<Integer, String>>) integer
    -> (valueToAdd) -> integer + valueToAdd + " was calculated")
  .ifPresentOrElse(
    fkt -> System.out.println("f(10) ==> = " + fkt.apply(10)),
    () -> System.out.println("value not present (useless here)"));
```

Listing 1.12

---

1 http://www.functional-reactive.org

2 Streams: Data in, Data out

Beginnen wir jetzt damit, uns die Schnittstelle zwischen Streams und der klassischen Java-Welt anzusehen.

2.1 Wo kommen die Quelldaten her?

Wenn man daran denkt, dass Streams nicht wie Collections ihre eigenen Daten halten, dann stellt sich die Frage, wo sie denn herkommen. Die gebräuchlichste Art, Streams zu erzeugen, ist die Verwendung von Methoden, die aus einer festen Anzahl von Elementen einen Stream erzeugen. Das sind die Methoden Stream.of(val1,val2,val3…) und Stream.of(array). Es gibt auch spezialisierte Streams für primitive Datentypen, die über spezifische Erzeugungsmethoden, zum Beispiel IntStream.range(int, int), verfügen. Zu den Methoden aus dem Bereich der Erzeugung aus einem festen Wertevorrat gehört auch die Methode, die aus einem String einen Stream erzeugt. Ein String ist nichts anderes als eine endliche Kette von chars. Sie kann als Wertevorrat übergeben werden:

```
final Stream<String> splitOf = Stream.of("A,B,C".split(","));
```

Streams können mithilfe der Methoden stream() und parallelStream()auch direkt aus Collections erzeugt werden. Den Unterschied von beiden Methoden werden wir uns zu einem späteren Zeitpunkt noch genauer ansehen. Auch Arrays bieten die Möglichkeit, direkt einen Stream aus ihnen zu erzeugen. Hierzu kann man die Methode Arrays.stream(..) verwenden.

Einige Klassen, die als Datenlieferanten dienen, haben Methoden, die einen Stream zurückliefern, zum Beispiel die Klasse Random mit der Methode Random.ints() oder BufferedReader.lines(). Genauso können Streams aus Streams erzeugt werden, was im Folgenden genauer dargestellt werden wird.

Nun fehlen noch zwei Möglichkeiten, Streams zu erzeugen. Mit einem Builder kann ein Stream programmatisch erzeugt werden:

```
final Stream<Pair> stream = Stream.<Pair>builder().add(new Pair()).build();
```

Die letzte Möglichkeit, einen Stream zu erzeugen, besteht darin, einen Generator zu verwenden. Das erfolgt über die Methode Streams.generate(..), in deren Argument die Methode eine Instanz der Klasse Supplier<T> bekommt (Listing 2.1).

```
Stream.generate(() -> {
  final Pair p = new Pair();
  p.id = random.nextInt(100);
```

```
  p.value = "Value + " + p.id;
  return p;
})
```

Listing 2.1

2.2 Wo gehen die Daten hin?

Nachdem wir jetzt wissen, woher die Daten kommen, stellt sich die Frage, wie die Daten aus dem Stream wiederzubekommen sind. Immerhin ist ja meist (nicht immer) angedacht, damit weiterzuarbeiten. Die einfachste Möglichkeit besteht darin, aus einem Stream mit der Methode stream.toArray() ein Array oder mittels stream.collect(Collectors.toList()) eine Liste zu erzeugen. Damit sind fast 90 Prozent der Einsatzgebiete beschrieben, aber es können auch Sets und Maps erzeugt werden.

Um als Ergebnis ein Set zu erhalten, kann mittels der Methode stream.collect(Collectors.toSet() ein HashSet angefordert werden. Ob es auch in den kommenden Versionen ein HashSet sein wird, ist nicht sicher. Maps hingegen werden mit stream.collect(Collectors.groupingBy(..)) erzeugt. Das Argument von groupingBy() sieht mindestens eine Funktion vor, mit der eine Gruppierung vorgenommen werden kann. Die Gruppierung stellt den Schlüssel in der Map dar, das Value ist dann eine Liste vom Typ der Elemente des Streams.

Eine für so manchen Entwickler etwas ungewohnte Möglichkeit besteht darin, den Stream in einem String auszugeben. Um das zu erreichen, wird in der Methode collect ein toStringJoiner verwendet, dessen Übergabeparameter ein Delimiter ist. Das Ergebnis ist dann eine Liste von durch diesen Delimiter konkatenierten toString()-Repräsentationen aller Elemente.

Nachfolgend einige Beispiele der gerade vorgestellten Methoden, ohne an dieser Stelle näher darauf einzugehen (Listing 2.2). Wir werden uns alles nach und nach im Detail ansehen.

```
public static void main(String[] args) {
  final List<Pair<String,String>> generateDemoValues
    = generateDemoValues();
  //Stream from Values
  final Stream<String> fromValues
    = Stream.of("A" , "B");
  //Stream from Array
  final String[] strings = {"A" , "B"};
  final Stream<String> fromArray = Stream.of(strings);
  //Stream from List
  final Stream<Pair<String,String>> fromList
    = generateDemoValues.stream();
  //Stream from String
  final Stream<String> abc = Stream.of("ABC");
  final Stream<IntStream> of = Stream.of("ABC".chars());
  final Stream<String> splitOf
    = Stream.of("A,B,C".split(","));
  //Stream from builder
```

```
  final Stream<String> builderPairStream =
    Stream.<String>builder().add("A").build();
  //Stream to Array
  final Pair<String, String>[] toArray =
    generateDemoValues.stream().toArray(Pair[]::new);
  //Stream to List
  final List<Pair> toList =
    generateDemoValues.stream().collect(Collectors.toList());
  //Stream to Set
  final Set<Pair> toSet =
    generateDemoValues.stream().collect(Collectors.toSet());
  //Stream to Map
  final Map<String, List<Pair<String,String>>> collectedToMap =
    generateDemoValues
    .stream()
    .collect(Collectors
      .groupingBy(Pair::getT1));
    System.out.println("collectedToMap.size() = " + collectedToMap.size());

  for (final Map.Entry<String, List<Pair<String, String>>> entry :
  collectedToMap.entrySet()) {
  System.out.println("entry = " + entry);
  }
}

private static List<Pair<String,String>> generateDemoValues() {
  return Arrays.asList(
    new Pair<>("A" , "A") ,
    new Pair<>("B" , "B") ,
    new Pair<>("C" , "C")
  );
}
```

Listing 2.2

2.3 Collectors

Wir haben bisher die Klasse Collectors verwendet, ohne sie uns genauer anzusehen. Hier finden sich eine Menge Methoden, die eine Implementierung des Interface java.util.stream.Collector liefern. Das Interface Collector selbst ist hinsichtlich der Anzahl der definierten Methoden noch recht überschaubar, die Implementierung kann jedoch ein wenig aufwendiger und komplexer werden. Zu Beginn werden wir damit vorlieb nehmen, die Service Method der Klasse Collectors zu verwenden.

2.3.1 toList und toSet

Die wohl gebräuchlichsten und meiner Meinung nach einfachsten Methoden haben wir schon kennen gelernt. Hierbei wird aus der Ergebnismenge des Streams eine Liste oder ein Set erzeugt. Auch hier gibt es noch einige Feinheiten, die man eventuell gebrauchen könnte.

Wenn wir die Methoden ohne die explizite Angabe einer Implementierung verwenden, bekommen wir im Fall einer Liste die Implementierung der ArrayList und bei einem Set die Implementierung HashSet. Wenn wir nun eine LinkedList haben wollen, gehen wir den in Listing 2.3 gezeigten Weg.

```
public static Stream<Pair<String, String>> nextStream() {
  return Stream.of(
    new Pair<>("A" , "A") ,
    new Pair<>("B" , "B") ,
    new Pair<>("C" , "C")
  );
}

public static void main(String[] args) {

  List<Pair<String,String>> listA = nextStream()
    .collect(Collectors.toCollection(()-> new LinkedList<>()));

  List<Pair<String,String>> listB = nextStream()
    .collect(Collectors.toCollection(LinkedList::new));
}
```

Listing 2.3

In Listing 2.3 wird mithilfe der Methode Collectors.toCollection(..) eine Factory eingesetzt, die dazu verwendet wird, die gewünschte Instanz einer Collection zu erzeugen. Dabei habe ich als Beispiel die LinkedList verwendet.

An dieser Stelle möchte ich auf die Interfaces und Klassen im Package utils hinweisen. Dorthin werde ich die öfter benötigten Methoden, zum Beispiel die Methode zum Erzeugen von Beispieldaten oder Streams, auslagern.
In den nachfolgenden Quelltexten werden diese Implementierungen im Allgemeinen nicht mehr gezeigt, um die Beispiele nicht unnötig redundant werden zu lassen.

2.3.2 toMap

Möchte man eine Map als Datenstruktur erhalten, kann man mit der Methode toMap(..) arbeiten. Diese gibt es in drei Ausprägungen.

Beginnen wir mit der ersten, die uns die Möglichkeit gibt, zwei Funktionen anzugeben. Die erste ist eine Funktion vom Datentyp auf den gewünschten Schlüssel der Map und die zweite eine Funktion vom Datentyp auf den gewünschte Wert zu dem gerade erzeugten Schlüssel. Somit sind Key und Value eine Funktion basierend auf dem Datentyp des Value, das sich an dieser Stelle gerade im Stream befindet. In Listing 2.4 bin ich sehr ausführlich vorgegangen und werde in Listing 2.5 die Implementierung mittels Lambdas und Methodenreferenzen ein wenig

kompakter gestalten.

```
final Map<String, String> map = nextStream()
  .collect(Collectors.toMap(new Function<Pair<String, String>, String>() {
    @Override
    public String apply(Pair<String, String> p) {
      return p.getT1();
    }
  } , new Function<Pair<String, String>, String>() {
    @Override
    public String apply(Pair<String, String> p) {
      return p.getT2().toLowerCase();
    }
  }));
System.out.println("map = " + map);
```

Listing 2.4

```
final Map<String, String> mapB = nextStream()
  .collect(Collectors.toMap(Pair::getT1 , p -> p.getT2().toLowerCase()));

System.out.println("mapB = " + mapB);
```

Listing 2.5

Wir bekommen von der Klasse HashMap eine Instanz zurückgeliefert. Wichtig ist dabei zu wissen, dass eine Fehlermeldung erzeugt wird, wenn wir einen Schlüssel zweimal erhalten. Wenn wir in unserem Fall die Methode nextStream() intern ändern und die Methode DemoData.nextStreamWithDuplicates()verwenden, bekommen wir bei der ersten Implementierung folgende Fehlermeldung:

```
IllegalStateException: Duplicate key B (attempted merging values b and b)
```

Sehen wir uns nun die zweite Methodensignatur von toMap(..) an. Hier können wir zusätzlich BinaryOperator<T> extends BiFunction<T,T,T> angeben, was es uns ermöglicht, im Fall eines Konflikts eine explizite Entscheidung zu treffen (Listing 2.6).

```
final Map<String, String> mapC = nextStream()
  .collect(Collectors.toMap(Pair::getT1 ,
    p -> p.getT2().toLowerCase() ,
    new BinaryOperator<String>() {
      @Override
      public String apply(String s1 , String s2) {
        return s1 + " - " + s2;
      }
    }

  ));
System.out.println("mapC = " + mapC);
```

Listing 2.6

Listing 2.7 zeigt das in kompakter Form und unter Verwendung von statischen Imports.

```
final Map<String, String> mapB = nextStream()
  .collect(toMap(Pair::getT1 ,
    p -> p.getT2().toLowerCase() ,
    (s1 , s2) -> s1 + " - " + s2));

System.out.println("mapB = " + mapB);
```

Listing 2.7

Schaut man sich nun die Ausgabe an, sieht man, dass die Values bei mehrfach auftretenden Schlüsseln einfach hintereinander gehängt werden:

```
mapB = {A=a, B=b - b, C=c - c - c}
```

Kommen wir nun zur dritten Variante, bei der zusätzlich noch ein Supplier angegeben werden kann, mit dem die zu verwendende Datenstruktur erzeugt wird. Hier kann an wie im nachfolgenden Fall eine TreeMap oder natürlich auch jede beliebige Implementierung einer Map angeben. Als Beispiel könnte eine verteilte Datenstruktur verwendet werden, wie sie im Projekt Hazelcast vorhanden ist (Listing 2.8).

```
final Map<String, String> mapA = nextStream()
  .collect(toMap(Pair::getT1 ,
    p -> p.getT2().toLowerCase() ,
    (s1 , s2) -> s1 + " - " + s2 ,
    TreeMap::new));

System.out.println("mapA = " + mapA);
```

Listing 2.8

Alle Implementierungen der Methode toMap(..) gibt es auch als parallele Version mit der Signatur toConcurrentMap(..).

2.3.3 groupingBy

Wir hatten hier den Fall, dass es im finalen Zustand eine eindeutige Zuordnung zwischen einem Schlüssel und einem Wert gab. Wenn es ein wenig schwächer formuliert werden darf, bekommen wir einen Schlüssel, der einer Menge von Werten zugeordnet wird. In diesem Fall verwenden wir die Methode groupingBy(..). Auch die gibt es wieder in mehrfacher Ausprägung und in der parallelen Version groupingByConcurrent(..) (Listing 2.9).

```
final Map<String, List<Pair<String, String>>> mapA = nextStream()
  .collect(Collectors.groupingBy(new Function<Pair<String, String>, String>() {
    @Override
```

```
    public String apply(Pair<String, String> p) {
      return p.getT1();
    }
  }));
System.out.println("mapA = " + mapA);
```

Listing 2.9

Der große Unterschied besteht hier darin, das wir eine Map bekommen, in der die Values nochmals in eine Liste verpackt worden sind. Damit erklärt sich auch die erste Signatur, bei der wir nur eine Funktion benötigen, um den Schlüssel zu identifizieren.

Als Defaultverhalten wird nun der Wert einfach passend zum Schlüssel in eine Liste gesetzt. Demnach kann es Duplikate geben, und eine explizite Entscheidung ist nicht notwendig. Listing 2.10 zeigt die kompakte Form.

```
final Map<String, List<Pair<String, String>>> mapB = nextStream()
  .collect(Collectors.groupingBy(Pair::getT1));
System.out.println("mapB = " + mapB);
```

Listing 2.10

In der zweiten Ausprägung können wir eine Implementierung des Interface Collector mit übergeben. Da wir bisher keine eigenen Collectors geschrieben haben, verwenden wir hier die uns schon bekannten. Im Beispiel in Listing 2.11 wollen wir in der Ergebnismenge, die zu einem Schlüssel gehört, keine Duplikate zulassen und verwenden anstelle einer Liste ein Set.

```
final Map<String, Set<Pair<String, String>>> mapA = nextStream()
  .collect(groupingBy(Pair::getT1, Collectors.toSet()));
System.out.println("mapA = " + mapA);
```

Listing 2.11

Bisher ist die Implementation der Map immer vom Typ HashMap gewesen. Auch hier gibt es wieder eine Möglichkeit, eine Map-Implementierung zu wählen, die dem Problem adäquat ist. Als Beispiel nehme ich wieder die TreeMap (Listing 2.12):

```
final Map<String, Set<Pair<String, String>>> mapA = nextStream()
  .collect(groupingBy(Pair::getT1, TreeMap::new, Collectors.toSet()));
System.out.println("mapA = " + mapA);
```

Listing 2.12

2.3.4 reducing

Soll aus den Werten im Stream ein einziger, finaler Wert ermittelt werden, kann man hier einen Collector verwenden, der nur einen einzelnen Wert zurückliefert. Um die Problematik mit den null-Werten zu umgehen, wird der Wert selbst in einem Optional ausgeliefert. Hier soll aus einer

Liste von Integer-Werten die Summe berechnet werden. Der Übergabeparameter ist ein BinaryOperator, bei dem der erste Wert immer das Ergebnis einer vorhergehenden Berechnung und der zweite Wert der aktuelle Wert ist (Listing 2.13).

```
final Optional<Integer> resultA = nextIntegerStream()
  .collect(Collectors.reducing(new BinaryOperator<Integer>() {
    @Override
    public Integer apply(Integer a , Integer b) {
      return a + b;
    }
  }));

System.out.println("resultA = " + resultA);

final Optional<Integer> resultB = nextIntegerStream()
  .collect(Collectors.reducing((a , b) -> a + b));
System.out.println("resultB = " + resultB);

final Optional<Integer> resultC = nextIntegerStream()
  .reduce((a , b) -> a + b);
System.out.println("resultC = " + resultC);
```

Listing 2.13

An dieser Stelle möchte ich schon einmal vorab darauf hinweisen, dass einige Methoden direkt auf die Ebene des Interface Stream gehoben wurden. In diesem Fall ist es die Methode reduce(..), die eine Instanz vom Typ BinaryOperator erwartet. Hier handelt es sich nicht um eine direkte Delegation an die Methode collect(..).

Möchte man mit der Aggregation von einem bestimmten Punkt aus beginnen, zum Beispiel ein OffSet verwenden, kann man den Startwert mit angeben. Dadurch ergibt sich auch der minimale Rückgabewert, der kein null ist, und demnach besteht auch nicht mehr die Notwendigkeit, das Ergebnis in ein Optional zu verpacken.

Was auf der einen Seite logisch erscheint, führt leider auf der anderen Seite zu einem unsymmetrischen API (Listing 2.14).

```
final Integer resultA = nextIntegerStream()
  .collect(reducing(
    0 ,
    new BinaryOperator<Integer>() {
      @Override
      public Integer apply(Integer a , Integer b) {
        return a + b;
      }
    }
  ));
System.out.println("resultA = " + resultA);

final Integer resultB = nextIntegerStream()
```

```
  .collect(reducing(
    0 ,
    (a , b) -> a + b
  ));
System.out.println("resultB = " + resultB);

final Integer resultC = nextIntegerStream()
  .reduce(0 , (a , b) -> a + b);
System.out.println("resultC = " + resultC);
```

Listing 2.14

Als dritte Version haben wir die Möglichkeit, eine Funktion mit zu übergeben, die den Wert vor der Berechnung transformiert (Listing 2.15). In unserem Fall wird einfach durch 10 geteilt und das Interimsergebnis gerundet: Math.round(10 / input).

```
final Integer resultA = nextIntegerStream()
  .collect(reducing(
    0 ,
    new Function<Integer, Integer>() {
      @Override
      public Integer apply(Integer input) {
        return Math.round(10 / input);
      }
    } ,
    new BinaryOperator<Integer>() {
      @Override
      public Integer apply(Integer a , Integer b) {
        return a + b;
      }
    }
  ));
System.out.println("resultA = " + resultA);

final Integer resultB = nextIntegerStream()
  .collect(reducing(
    0 ,
    input -> Math.round(10 / input) ,
    (a , b) -> a + b
  ));
System.out.println("resultB = " + resultB);
```

Listing 2.15

Auch hier gibt es wieder eine Version der Methode reduce auf der Ebene des Interface Stream. Nur ist die Methodensignatur ein wenig anders. Es wird nicht eine Funktion zur Transformation selbst angeboten, sondern man kann eine BiFunction übergeben, die den alten und den aktuellen Wert erhält, um eine Transformation vorzunehmen (Listing 2.16).

```
final Integer resultC = nextIntegerStream()
  .reduce(0 ,
```

```
    new BiFunction<Integer, Integer, Integer>() {
      @Override
      public Integer apply(Integer old , Integer next) {
        return old + Math.round(10 / next);
      }
    } ,
    (a , b) -> a + b
  );
System.out.println("resultC = " + resultC);

final Integer resultD = nextIntegerStream()
  .reduce(0 ,
    (old , next) -> old + Math.round(10 / next) ,
    (a , b) -> a + b
  );
System.out.println("resultD = " + resultD);
```

Listing 2.16

An dieser Stelle muss man ein wenig konzentrierter sein, damit ein semantisches Äquivalent erzeugt wird.

2.3.5 joining

Kommen wir nun zu den Methoden mit dem Namen joining(..). Wie der Name vermuten lässt, werden mit ihnen wieder die Werte aus dem Stream zusammengeführt. Die Besonderheit liegt dabei im Datentyp, es muss sich um einen Stream von CharSequence oder auch Stream<String> handeln und in einen String überführt werden. Die einfachste Version ohne Parameter bildet einen finalen String aus allen Elementen, die im Stream geliefert werden (Listing 2.17).

```
final String resultA = nextStringStreamWithDuplicates()
  .collect(Collectors.joining());
System.out.println("resultA = " + resultA);
```

Listing 2.17

Es gibt die Methode joining wieder in verschiedenen Ausprägungen. Die erste davon ist um die Angabe eines Delimiters erweitert worden. Somit werden die einzelnen Teile wieder zu einem finalen String zusammengeführt, nun jedoch getrennt um die angegebene Sequenz (Listing 2.18).

```
final String resultA = nextStringStreamWithDuplicates()
  .collect(Collectors.joining(","));
System.out.println("resultA = " + resultA);
```

Listing 2.18

Und als Letztes gibt es noch die Möglichkeit, ein Prefix und ein Suffix für den finalen String anzugeben (Listing 2.19).

```
final String resultA = nextStringStreamWithDuplicates()
  .collect(Collectors.joining(",", "PREFIX", "SUFFIX"));
System.out.println("resultA = " + resultA);
```

Listing 2.19

2.3.6 collectingAndThen

Mit dieser Methode werden wir in die Lage versetzt, nicht nur einen Collector anzugeben, sondern auch noch eine Funktion, um das Ergebnis des Collectors noch einmal beliebig zu transformieren (Listing 2.20).

```
final String resultA = nextIntegerStream()
  .collect(collectingAndThen(toList() , new Function<List<Integer>, String>() {
    @Override
    public String apply(List<Integer> v) {
      return v.toString().substring(7);
    }
  }));
System.out.println("resultA = " + resultA);

final String resultB
    = nextIntegerStream()
    .collect(collectingAndThen(toList() , v -> v.toString().substring(7)));
System.out.println("resultB = " + resultB);
```

Listing 2.20

2.3.7 averagingInt, averagingDouble und averagingLong

Hier ist vorgesehen, dass alle Werte aus dem Stream in einen Integer (Long oder Double) konvertiert werden. Als Ergebnis bekommt man einen Double-Wert, der den arithmetischen Mittelwert aller verarbeiteten Werte darstellt (Listing 2.21).

```
final Double valueA = nextIntegerStream()
  .collect(averagingInt(new ToIntFunction<Integer>() {
    @Override
    public int applyAsInt(Integer v) {
      return v;
    }
  }));
System.out.println("valueA = " + valueA);

final Double valueB = nextInteger
  Stream().collect(averagingInt(v -> v));
System.out.println("valueB = " + valueB);
```

Listing 2.21

2.3.8 summarizingInt, summarizingDouble und summarizingLong

Mit der Methode summarizingInt(..) erhalten wir einen Collector, der statistische Werte liefert,

die in einer Instanz der Klasse IntSummaryStatistics verpackt wurden. Ausgewertet werden alle Werte, die in der letzten Stufe des Streams ankommen und mit der übergebenen Funktion in den entsprechenden int-Wert transformiert wurden. Ermittelt werden die Anzahl der Elemente, die Summe aller Werte sowie der Mittel-, Minimal- und Maximalwert (Listing 2.22).

```
final IntSummaryStatistics statisticsA = nextIntegerStream()
  .collect(summarizingInt(new ToIntFunction<Integer>() {
    @Override
    public int applyAsInt(Integer v) {
      return v;
    }
  }));

System.out.println("statisticsA = " + statisticsA);

final IntSummaryStatistics statisticsB = nextIntegerStream()
  .collect(summarizingInt(v -> v));

System.out.println("statisticsB = " + statisticsB);
```

Listing 2.22

Diese Methode gibt es in Ausprägungen für Integer, Double und Long.

2.3.9 summingInt, summingDouble und summingLong

Mit dieser Methode erhalten wir einen Collector, der uns mit der übergebenen Funktion den jeweiligen Wert in einen Double transformiert und final die Summe aller Werte in Form eines Doubles liefert (Listing 2.23).

```
final Double sumA = nextIntegerStream().collect(summingDouble(new ToDoubleFunction<
Integer>() {
  @Override
  public double applyAsDouble(Integer v) {
    return v;
  }
}));
System.out.println("sumA = " + sumA);

final Double sumB = nextIntegerStream().collect(summingDouble (v -> v));
System.out.println("sumB = " + sumB);

//alternative
final Double sumC = nextIntegerStream().mapToDouble(v -> v).sum();
System.out.println("sumC = " + sumC);
```

Listing 2.23

2.3.10 counting

Wenn nur die Anzahl der Elemente von Interesse ist, die final im Collector ankommen, kann man mit der Methode counting() der dafür benötigte Collector erhalten (Listing 2.24).

```
final Long amountA = nextIntegerStream().collect(counting());

//alternative
final Long amountB = nextIntegerStream().count();
```

Listing 2.24

2.3.11 maxBy und minBy

Um einen Collector zu erhalten, der den Maximal- oder Minimalwert liefert, kann man der Methode maxBy(..) (minBy(..)) einen Comparator übergeben. Das Ergebnis wird in einem Optional verpackt ausgeliefert. Der in Listing 2.25 implementierte Comparator ist natürlich recht trivial.

```
final Optional<Pair<String, String>> resultA = nextStreamWithDuplicates()
  .collect(maxBy(new Comparator<Pair<String, String>>() {
    @Override
    public int compare(Pair<String, String> o1 , Pair<String, String> o2) {
      return 0;
    }
  }));
System.out.println("resultA = " + resultA);

final Optional<Pair<String, String>> resultB = nextStreamWithDuplicates()
  .collect(maxBy((o1 , o2) -> 0));
System.out.println("resultB = " + resultB);

final Optional<Pair<String, String>> resultC = nextStreamWithDuplicates().max((o1 ,
 o2) -> 0);
System.out.println("resultC = " + resultC);
```

Listing 2.25

2.3.12 mapping

Den hier vorgestellten Collector erhalten wir mit der Methode mapping(..). Dazu übergeben wir in Listing 2.26 als Argument eine Funktion, die den Wert in den gewünschten Typ und einen Collector in die gewünschte Datenstruktur transformiert, zum Beispiel eine Liste.

```
final List<Pair<String, String>> listA = nextIntegerStream()
  .collect(mapping(new Function<Integer, Pair<String, String>>() {
    @Override
    public Pair<String, String> apply(Integer v) {
      return new Pair<>("OK", String.valueOf(v));
    }
  } , toList()));
System.out.println("listA = " + listA);
```

```
final List<Pair<String, String>> listB = nextIntegerStream()
  .collect(mapping(v -> new Pair<>("OK", String.valueOf(v)) , toList()));
System.out.println("listB = " + listB);

final List<Pair<String, String>> listC = nextIntegerStream()
  .map(v -> new Pair<>("OK" , String.valueOf(v)))
  .collect(toList());
System.out.println("listC = " + listC);
```

Listing 2.26

2.3.13 partitioningBy

Mit der Methode partitioningBy erhalten wir einen Collector, der die Ergebnismenge in zwei Teile aufteilt, was auf Basis des übergebenen Predicates realisiert wird. Hier kann man entscheiden, welche Zielgruppe für ein Element vorgesehen ist. Die Methode partitioningBy kann zusätzlich noch einen weiteren Collector mit übergeben bekommen, der dann für die jeweilige Teilmenge angewandt wird. Hier wird im zweiten Beispiel ein Set als Datenstruktur für die beiden Teilmengen gewählt, um die Duplikate zu entfernen (Listing 2.27).

```
final Map<Boolean, List<Integer>> collectA
  = nextIntegerStreamWithDuplicates()
  .collect(partitioningBy(e -> e >= 5));

System.out.println("collectA = " + collectA);

final Map<Boolean, Set<Integer>> collectB = nextIntegerStreamWithDuplicates()
  .collect(partitioningBy(e -> e >= 5 , Collectors.toSet()));

System.out.println("collectB = " + collectB);
```

Listing 2.27

2.3.14 flatMapping

Nun kommen wir zu den neuen Methoden, die uns seit der Java-Version 9 zur Verfügung stehen. Mit der Methode flatMapping bekommen wir einen Collector, der es uns ermöglicht, aus einem Element einen Stream eines anderen Typs zu erzeugen. In unserem Beispiel (Listing 2.28) werden die Strings, die im Attribut der Instanz Pair<String,String> enthalten sind, in einen Stream<Character> transformiert. Ein nicht vorhandener Wert wird in einen leeren Stream überführt, sodass es keine NullPointerException geben wird. Der ebenfalls anzugebende Collector wird dann auf die resultierende Ergebnismenge angewandt.

```
final List<Character> resultA = nextStreamWithDuplicates()
      .collect(flatMapping(new Function<Pair<String, String>, Stream<Character>>()
{
  @Override
  public Stream<Character> apply(Pair<String, String> pair) {
    return pair.getT1().chars().mapToObj(value -> (char) value);
```

```
  }
} , toList()));

System.out.println("resultA " + resultA);

final List<Character> resultB = nextStreamWithDuplicates()
  .collect(flatMapping(pair -> pair.getT1().chars().mapToObj(value -> (char) value)
 , toList()));

System.out.println("resultB " + resultB);

final Set<Character> resultC = nextStreamWithDuplicates()
  .collect(flatMapping(pair -> pair.getT1().chars().mapToObj (value -> (char) value
) , toSet()));

System.out.println("resultC " + resultC);
```

Listing 2.28

2.3.15 filtering

Ebenfalls in Java 9 hinzugekommen ist die Möglichkeit, einen filternden Collector zu erzeugen. Hier werden aus der Menge der Elemente alle Elemente, die dem übergebenen Predicate entsprechen, herausgefiltert und dem ebenfalls übergebenen Collector dann in die gewünschte Datenstruktur übertragen (Listing 2.29).

```
final List<Integer> resultA = nextIntegerStreamWithDuplicates()
    .collect(filtering(new Predicate<Integer>() {
  @Override
  public boolean test(Integer v) {
    return v > 5;
  }
} , toList()));

System.out.println("resultA = " + resultA);

final List<Integer> resultB = nextIntegerStreamWithDuplicates()
  .collect(filtering(v -> v > 5 , toList()));

System.out.println("resultB = " + resultB);
```

Listing 2.29

2.3.16 collectingAndThen

Nun kann es sein, dass mit dem Ergebnis des Collectors eine weitere Transformation durchgeführt werden soll. Diese kann man in Form einer Funktion gleich mit angeben. Im Beispiel in Listing 2.30 wird aus der Menge der eineindeutigen Integerwerte (unifiziert mittels Set) die Summe berechnet.

```
final Integer resultA = nextIntegerStreamWithDuplicates()
```

```
  .collect(collectingAndThen(toSet() , new Function<Set<Integer>, Integer>() {
    @Override
    public Integer apply(Set<Integer> integers) {
      return integers.stream().collect(Collectors.summingInt (new ToIntFunction<Int
eger>() {
        @Override
        public int applyAsInt(Integer value) {
          return value;
        }
      }));
    }
  }));

System.out.println("resultA = " + resultA);

final Integer resultB = nextIntegerStreamWithDuplicates()
  .collect(collectingAndThen(toSet() , set -> (Integer) set.stream().mapToInt(value
 -> value).sum()));

System.out.println("resultB = " + resultB);
```

Listing 2.30

2.4 Kombinieren von Collectors

Mit den vorhandenen Collectors haben wir nun die Möglichkeit, die Elemente eines Streams wieder in eine Datenstruktur zu überführen. Die meisten der vorhandenen Collectors sind im Allgemeinen auch ausreichend.

Nun gibt es noch die Möglichkeit, Collectors selbst zu implementieren, die wir uns im nächsten Abschnitt ansehen werden, und die Möglichkeit, die bestehenden Collectors zu kombinieren. An einigen Stellen haben wir das schon gemacht, da wir durch die Signatur der beschriebenen Methoden dazu aufgefordert worden sind. Wenn man sich nun diese Möglichkeit ein wenig genauer ansieht, kann man doch einiges an Potenzial darin erkennen.

Gehen wir nachfolgendes Beispiel einmal durch. Wir wollen darin die Menge der Zufallszahlen von null bis inklusive neun wie folgt unterteilen:

- Als Erstes alle Zahlen in die Mengen größer vier und kleiner fünf aufteilen.
- Anschließend sollen die jeweiligen Mengen nochmals in gerade und ungerade Zahlen unterteilt werden.

Das können wir wie in Listing 2.31 angehen.

```
final Map<Boolean, List<Integer>> resultA = nextIntegerStreamWithDuplicates()
  .collect(partitioningBy(i -> i < 5));
System.out.println("resultA = " + resultA);
```

```
final Map<Boolean, List<Integer>> resultA_1 = resultA.get(true).stream().collect(pa
rtitioningBy(i -> i % 2 == 0));
final Map<Boolean, List<Integer>> resultA_2 = resultA.get(false).stream().collect(p
artitioningBy(i -> i % 2 == 0));
```

Listing 2.31

Hier erzeugen wir, wie zuvor erklärt, die ersten beiden Ergebnismengen und wenden anschließend die nächste Teilung an. Nun können wir auch die Collectors kombinieren, sodass wir den zweiten Collector dem ersten mitgeben (Listing 2.32).

```
final Map<Boolean, Map<Boolean, List<Integer>>> resultB = nextIntegerStreamWithDupl
icates()
  .collect(partitioningBy(i -> i < 5 , partitioningBy(i -> i%2 == 0)));
System.out.println("resultB = " + resultB);
```

Listing 2.32

Es gibt hier sicherlich mehrere Wege, die auszuprobieren sich lohnt.

Schnell wird ersichtlich, dass man aufpassen muss, dass dabei keine unübersichtliche Datenstruktur entsteht. Sehen wir uns hierzu ein weiteres Beispiel an. Wir definieren uns ein Auto mit den Attributen Marke, Farbe, gefahrene Kilometer und ob es modifiziert wurde (Listing 2.33).

```
public enum Brand {BMW, VW, PORSCHE, FIAT}

public enum Colour {RED, GREEN, WHITE, BLACK}

public static class Car extends Quad<Brand, Colour, Integer, Boolean> {
  public Car(Brand brand , Colour colour , Integer km , Boolean modified) {
    super(brand , colour , km , modified);
  }

  public Brand brand() {return getT1();}

  public Colour colour() {return getT2();}

  public Integer km() {return getT3();}

  public Boolean modified() {return getT4();}
}
```

Listing 2.33

Um über eine ausreichende Menge an Fahrzeugen zu verfügen, erzeugen wir uns einige nach dem Zufallsprinzip (Listing 2.34).

```
public static Stream<Car> randomCars(int amount) {
  final Random random = new Random();
```

```
  return IntStream
    .range(0 , amount)
    .mapToObj((i) -> new Car(
      Brand.values()[random.nextInt(4)] ,
      Colour.values()[random.nextInt(4)] ,
      random.nextInt(100_000) ,
      random.nextBoolean()
    ));
}
```

Listing 2.34

Nun können wir diese Menge an Fahrzeugen nach verschiedenen Kriterien sortieren bzw. unterteilen. Als Erstes erzeugen wir eine Gruppierung nach Marke und Farbe (Listing 2.35).

```
final Map<Brand, Map<Colour, List<Car>>> resultA = randomCars(100)
  .collect(
    groupingBy(Car::brand ,
      groupingBy(Car::colour)));
```

Listing 2.35

Um daraus wiederum eine Teilmenge zu bestimmen, kann man recht komfortabel durch die Datenstruktur navigieren. In Listing 2.36 zum Beispiel zu der Menge aller BMW mit der Farbe GREEN.

```
resultA.getOrDefault(BMW , emptyMap())
  .getOrDefault(GREEN , emptyList())
  .forEach(System.out::println);
```

Listing 2.36

Zu beachten ist, dass auch ohne Verwendung der Klasse Optional keine NullPointerException entstehen kann.

Als letztes Beispiel bestimmen wir in Listing 2.37 die durchschnittlichen km aller modifizierten Autos und gruppieren diese nach Marke und Farbe. Ausgeben wollen wir allerdings nur die Werte der Marke BMW.

```
final Map<Brand, Map<Colour, Double>> resultB = randomCars(100)
  .collect(filtering(Car::modified ,
    groupingBy(Car::brand ,
      groupingBy(Car::colour ,
        averagingDouble(Car::km)))));

resultB.getOrDefault(BMW, emptyMap())
  .forEach((colour , km) -> System.out.println("colour / avg [km] = " + colour + "
- " + km));
```

Listing 2.37

2.5 Eigene Collectors

Die allgemeinen Collectors reichen unter Umständen nicht aus, sodass man manchmal in die Verlegenheit kommt, selbst welche zu erstellen, die dem Problem angemessen sind. Hier gibt es prinzipiell zwei Wege.

2.5.1 Collectors.of()

Die Methode of(..) ermöglicht es, eigene Collectors zu definieren und erwartet die einzelnen Komponenten in Form von Lambdas. Es gibt hier zwei verschiedene Signaturen. Beginnen wir mit der umfangreicheren, da diese dem Verständnis zuträglicher ist.

Die Methodensignatur besteht aus fünf Elementen, bei denen in der Summe drei Typen mit definiert werden. Als Erstes geben wir den Typ an, aus dem die Elemente bestehen, die eingesammelt werden sollen. In diesem Beispiel werden wir Elemente vom Typ String verarbeiten. Der zweite Typ beschreibt die Datenstruktur, die intern verwendet wird, um die einzelnen Elemente, die der Stream liefert, zu verarbeiten. In unserem Fall ist es eine List<String>. Der letzte Typ, der definiert werden muss, beschreibt die finale Datenstruktur, also den Typ des Ergebnisses dieses Collectors. In unserem Fall ist es ein Set<String>. Kommen wir nun zu den Parametern, die die Methode of(..) erwartet.

- Nummer eins ist ein Supplier, hier vom Typ Supplier<List<String>>, der verwendet wird, um eine Instanz einer ArrayList<String> zu erzeugen. In dieser Instanz werden die zu verarbeitenden Elemente gesammelt.
- Nummer zwei ist ein BiConsumer (BiConsumer<List<String>, String>), der dazu dient, zu beschreiben, wie ein Element in die hier verwendete Instanz zum Sammeln der Elemente (ArrayList<String>) eingefügt wird.
- Nummer drei ist vom Typ BinaryOperator<List<String>> und wird Merger genannt. Die Aufgabe des Mergers ist es, zwei Teilergebnisse zusammenzufügen. Hier kann man entscheiden, ob man eines der beiden Elemente weiterverwenden möchte oder eine vollständig neue Instanz erzeugt. In unserem Beispiel wird einfach eine der beiden Listen weiter verwendet.
- Nummer vier kann man sich als transformierende Einheit vorstellen, die aus der Instanz, in der die Elemente gesammelt worden sind, den finalen Ausgangstyp erzeugt. Hier wird aus der Arbeitsliste ein Set erzeugt.
- Der letzte Parameter liefert Informationen über den Collector selbst. Hier kann man angeben, ob es sich um einen Collector handelt, der zum Beispiel auch parallel arbeiten kann. In unserem Fall ist es ein serieller, der keine Ansprüche an die Reihenfolge der Elemente stellt.

```
final Collector<String, List<String>, Set<String>> collectorA
  = Collector.<String, List<String>, Set<String>>of(
  (Supplier<List<String>>) () -> new ArrayList<>() ,
  (BiConsumer<List<String>, String>) (list , o) -> list.add(o) ,
  (BinaryOperator<List<String>>) (listA , listB) -> {
    listA.addAll(listB);
    return listA;
  } ,
  (Function<List<String>, Set<String>>) list -> {
    Set<String> set = new HashSet<>();
    set.addAll(list);
    return set;
  } ,
  Collector.Characteristics.UNORDERED
);

final Set<String> resultA = Stream.of("Hello" , "Collector").collect(collectorA);
```

Listing 2.38

Das gerade gezeigte Listing 2.38 kann man natürlich auch kürzer, also ohne all die Typinformationen, schreiben (Listing 2.39).

```
final Collector<String, List<String>, Set<String>> collectorB
  = Collector.of(
  ArrayList::new ,
  List::add ,
  (listA , listB) -> concat(listA.stream(), listB.stream()).collect(toList()),
  HashSet::new,
  Collector.Characteristics.UNORDERED
);
```

Listing 2.39

Nun stellt sich die Frage, warum man eigentlich die Möglichkeit hat, eine Datenstruktur für die interne Implementierung zu verwenden und dann noch einmal eine Transformation in eine finale Datenstruktur vorzunehmen. Es kann effizienter sein, für die einzelnen Schritte des Einsammelns eine dafür geeignetere Datenstruktur zu verwenden, auch wenn sie nicht der gewünschten Zieldatenstruktur entspricht. In diesem Fall kann man zwei verschiedene Typen intern verwenden, um so eine möglichst effiziente Implementierung zu realisieren.

Die zweite Version der Methode of(..) benötigt nur vier Parameter, es wird der Parameter Nummer vier der vorherigen Signatur weggelassen.

Somit ist die Ausgangsdatenstruktur die, die auch bei der Verarbeitung verwendet wird. Die Instanz, die man erhält, ist demnach die der letzten Merge-Phase. Das sieht man im Beispiel daran, dass der gelieferte Typ eine Liste und kein Set ist (Listing 2.40).

```
final Collector<String, List<String>, List<String>> collectorC = Collector.of(
  () -> new ArrayList<>() ,
```

```
  (list , value) -> list.add(value) ,
  (listA , listB) -> {
    listA.addAll(listB);
    return listA;
  } ,
  Collector.Characteristics.UNORDERED);
```

Listing 2.40

2.5.2 Interface „Collector“ implementieren

Natürlich kann man auch das Interface selbst implementieren. Auch hier sind alle Elemente anzugeben, wie vorher gezeigt (Listing 2.41).

```
final Collector<String, List<String>, Set<String>> collector = new Collector<>() {
  @Override
  public Supplier<List<String>> supplier() {
    return ArrayList::new;
  }

  @Override
  public BiConsumer<List<String>, String> accumulator() {
    return List::add;
  }

  @Override
  public BinaryOperator<List<String>> combiner() {
    return (listA , listB) -> concat(listA.stream(), listB.stream()).collect(toList
());
  }

  @Override
  public Function<List<String>, Set<String>> finisher() {
    return HashSet::new;
  }

  @Override
  public Set<Characteristics> characteristics() {
    return Set.of(Characteristics.UNORDERED);
  }
};

  Stream.of("Hello", "Collector", "Hello", "Collector")
    .collect(collector)
    .forEach(System.out::println);
```

Listing 2.41

2.5.3 Beispiel eines eigenen Collectors

Kommen wir zu einem weiteren Beispiel. Wir gehen davon aus, dass die Datenmenge, die vom Collector verarbeitet werden soll, größer ist als die Menge an verfügbarem Arbeitsspeicher. Was

für eine Datenstruktur kann man nun verwenden? Hier kann man auf eine Implementierung einer Map, wie zum Beispiel aus dem Projekt MapDB[1], zugreifen. In diesem Projekt wurden Datenstrukturen implementiert, die Daten für den Entwickler transparent auf die Festplatte auslagern. Es kann somit eine sehr große Menge an Elementen mit diesem Collector verarbeitet werden, ohne dass wir die hierfür notwendige Menge an Arbeitsspeicher vorhalten müssen.

Ebenfalls eignen sich Collectors, die man selbst definiert, dazu, auf eine Implementierung bestehender Builder zurückzugreifen. In einem solchen Fall ist die intern verwendete Datenstruktur die des Builders. Auch die finale Datenstruktur ist die vom Builder erzeugte Datenstruktur. Hier muss man beachten, ob und wie man die Merge-Phase implementieren möchte. Von Vorteil ist es auf jeden Fall, dass man hier auf bestehende Implementierungen zurückgreifen kann.

2.6 AutoCloseable

Wir kommen zu einer Besonderheit, die man bei der Verwendung von Streams nicht vergessen sollte. Streams implementieren das Interface AutoCloseable. Bei der Verwendung von Streams muss man also darauf achten, ob die Erzeugung eines Streams mittels try-catch-Block abgesichert werden muss. Hierzu zählen alle Streams, die Ressourcen öffnen und/oder verwenden, die selbst wieder geschlossen werden müssen, beispielsweise auf JDBC basierende Streams. Das Beispiel in Listing 2.42 habe ich von Stack Overflow übernommen.[2]

```
final Path p = Paths.get(System.getProperty("java.home"), "COPYRIGHT");
try(Stream<String> stream= Files.lines(p, StandardCharsets.ISO_8859_1)) {
  System.out.println(stream.filter(s->s.contains("Oracle")).count());
} catch (IOException e) {
  e.printStackTrace();
}
```

Listing 2.42

2.7 Zusammenfassung

Es ist sehr schön zu sehen, dass sich die Streams einfach in bestehenden Java-Code einbinden lassen. Mann muss keine unnötigen Wrapper oder Ähnliches schreiben. Die Integration ist damit mühelos in Altprojekten genauso möglich wie in neuen Projekten. Hat man sich ein wenig an das API gewöhnt, fallen einem sehr viele Stellen auf, an denen durch den Einsatz von Streams eine starke Codereduktion erreicht werden kann.

---

1 http://www.mapdb.org

2 https://stackoverflow.com/questions/38698182/close-java-8-stream/

3 Streams – Core Methods

Nachdem wir uns im vorigen Kapitel damit beschäftigt haben, wie die Daten in die Streams gelangen und wie sie wieder zu entnehmen sind, werden wir uns jetzt mit der Datentransformation beschäftigen. Es stehen unter anderem drei Basismethoden – forEach, match und find – zur Verfügung, mit denen man schnell und einfach die ersten Versuche unternehmen kann.

3.1 forEach – ein Lambda für jedes Element

Die Methode forEach(<lambda>) macht eigentlich genau das, was man vermutet: Sie wendet das als Argument übergebene Lambda auf jedes einzelne Element des Streams an. Diese Methode ist auch bei Iterable, List, Map und einigen anderen Klassen/Interfaces zu finden, was erfreulicherweise zu kürzeren sprachlichen Konstrukten führt. Listing 3.1 zeigt zuerst die Iteration in Pre JDK 8 Notation und danach unter Verwendung von forEach(<lambda>). In ihm sind lange und kurze Versionen zu sehen. Die langen Versionen verwenden die volle Notation inklusive der geschwungenen Klammern. Jeweils zuletzt wird eine Version mit der Verwendung von Method References angegeben, zu erkennen an den Doppelpunkten.

```
final List<Integer> list = List.of(1 , 2 , 3 , 4);

for (Integer integer : list) {
  System.out.println("integer = " + integer);
}

list.stream().forEach((Integer integer) -> {System.out.println(integer);});
list.stream().forEach(integer -> {System.out.println(integer);});
list.stream().forEach(integer -> System.out.println(integer));
list.stream().forEach(System.out::println);

list.parallelStream().forEach((Integer integer) -> {System.out.println(integer);});
list.parallelStream().forEach(integer -> {System.out.println(integer);});
list.parallelStream().forEach(integer -> System.out.println(integer));
list.parallelStream().forEach(System.out::println);

list.forEach((Integer integer) -> {System.out.println(integer);});
list.forEach(integer -> {System.out.println(integer);});
list.forEach(integer -> System.out.println(integer));
list.forEach(System.out::println);
```

Listing 3.1

In Listing 3.1 besteht ein kleiner, aber feiner Unterschied zwischen den beiden Versionen. Die

Variante, die den Stream mit der Methode parallelStream() erzeugt, lässt die Elemente parallel verarbeiten. Wir haben an der Stelle einen parallelen Stream.

Bei beiden ist sichergestellt, dass die übergebene Methodenreferenz nur einmal auf jedes Element angewandt wird. Nicht sichergestellt ist jedoch die Reihenfolge, in der die einzelnen Elemente abgearbeitet werden, wenn es sich um parallele Streams handelt. Ist die Reihenfolge von Bedeutung, muss die Methode forEachOrdered(<lambda>) verwendet werden. Hier wird bei der Abarbeitung die Reihenfolge, die in der Verarbeitungsstufe vor dem forEach(..) vorliegt, eingehalten. Diese Methode sollte nur verwendet werden, wenn es zwingend notwendig ist. Was hier allerdings auf den ersten Blick nicht ersichtlich ist: Es wird ein NotNullCheck durchgeführt, der im Fall von null mit einer NullPointerException quittiert wird. Jedes Element wird durch die consumer.accept-Methode konsumiert (Listing 3.2).

```
//class - ReferencePipeline
@Override
public void forEach(Consumer<? super P_OUT> action) {
  evaluate(ForEachOps.makeRef(action, false));
}
//class - AbstractPipeline
final <R> R evaluate(TerminalOp<E_OUT, R> terminalOp) {
  assert getOutputShape() == terminalOp.inputShape();
  if (linkedOrConsumed)
    throw new IllegalStateException(MSG_STREAM_LINKED);
  linkedOrConsumed = true;
  return isParallel()
    ? terminalOp.evaluateParallel(this,
    sourceSpliterator(terminalOp.getOpFlags()))
    : terminalOp.evaluateSequential(this,
    sourceSpliterator(terminalOp.getOpFlags()));
}
//class - ForEachOps
public static <T> TerminalOp<T, Void>
  makeRef(Consumer<? super T> action, boolean ordered) {
  Objects.requireNonNull(action); //throws NPE
  return new ForEachOp.OfRef<>(action, ordered);
}
//class – ForEachOps. OfRef
static final class OfRef<T> extends ForEachOp<T> {
  final Consumer<? super T> consumer;
//...
  @Override
  public void accept(T t) {
    consumer.accept(t);
  }
}
```

Listing 3.2

Bei der Verwendung von forEach(<lambda>) ist auch Folgendes zu beachten: Durch die

Methode accept im Consumer wird das Element konsumiert. Das bedeutet, dass forEach(<lambda>) nur einmal auf einen Stream angewandt werden kann. Man spricht in diesem Zusammenhang auch von einer terminalen Operation. Ist mehr als eine Operation auf das Element anzuwenden, kann dies natürlich innerhalb des übergebenden Lambdas geschehen. Das Argument der forEach(<lambda>)-Methode kann jedoch wiederverwendet werden, indem man eine Instanz vorhält und sie dann mehreren Streams übergibt (Listing 3.3).

```
final Consumer<? super Pair> consumer = System.out::println;
generateDemoValues.stream().forEachOrdered(consumer);
generateDemoValues.parallelStream().forEachOrdered(consumer);
```

Listing 3.3

Ebenfalls nicht gestattet ist die Manipulation von umgebenden Variablen. Wie das erfolgen kann, sehen wir uns im Zusammenhang mit den Methoden map und reduce an. Der größte Unterschied zu einer for-Schleife ist allerdings, dass nicht vorzeitig unterbrochen werden kann, weder mit break noch mit return.

3.2 map – Transformationen gefällig?

Die Methode map(<lambda>) erzeugt einen neuen Stream, bestehend aus der Summe aller Transformationen der Elemente des Quell-Streams. Auch hier ist das Argument wieder ein Lambda. Das bedeutet, dass der Ziel-Stream bis auf die funktionale Kopplung nichts mit dem Quell-Stream gemeinsam haben muss.

Im Beispiel in Listing 3.4 wird aus einem Stream<Integer> ein Stream<Float>, um anschließend in einen Stream<String> gemappt zu werden. Die Methode map(<lambda>) kann beliebig oft hintereinander angewandt werden, da das Ergebnis immer wieder ein neuer Stream ist.

```
final Stream<Integer> streamA = Stream.of(1 , 2 , 3 , 4 , 5);

final Stream<Float> streamB = streamA.map(new Function<Integer, Float>() {
  @Override
  public Float apply(Integer i) {
    return Float.valueOf(i);
  }
});

final Stream<String> streamC = streamB.map(v -> String.valueOf(v));

//all together
final Stream<String> stream = Stream
  .of(1 , 2 , 3 , 4 , 5)
  .map(Float::valueOf)
  .map(String::valueOf);
```

Listing 3.4

3.3 filter – wer darf es sein?

Wie die Methode map(<lambda>) erzeugt auch die Methode filter(<Lambda>) einen neuen Stream. Aus der Menge der Quellelemente werden die für die weiteren Schritte benötigten Elemente herausgefiltert. Die Methode filter(<Lambda>) kann mehrfach hintereinander angewandt werden, wobei mit jedem weiteren Aufruf die Menge weiter gefiltert wird. Es findet somit eine weitere Reduktion statt. Die Methode filter(<Lambda>) kann in beliebiger Kombination angewandt werden, z. B. map -> filter -> map –> filter -> filter (Listing 3.5).

```
final Stream<Integer> stream = Stream
  .of(1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9)
  .filter(v -> v % 2 == 0)
  .filter(v -> v > 2);
```

Listing 3.5

3.4 findFirst – was ist das erste Element?

Immer wieder gibt es eine Menge von Elementen, deren Reihenfolge nicht definiert und deren Anzahl unbestimmt ist, aber aus der genau ein Element mit bestimmten Eigenschaften zu entnehmen ist. Was in der Datenbank dank SQL in den meisten Fällen kein Problem darstellt, kann auf der imperativen Seite schon mal zu einem längeren Stück Quelltext führen.

Die Methode findFirst() liefert das erste Element aus dem Stream. Auf den ersten Blick eine triviale Methode, umso mehr freut man sich beim zweiten Blick. Der Rückgabewert ist ein Optional<T>, im Fall eines leeren Streams ein leeres Optional (Listing 3.6).

```
final Optional<String> first = Stream
  .of("AB" , "AAB" , "AAAB" , "AAAAB" , "AAAAAB")
  .findFirst();
```

Listing 3.6

Mit der Methode findFirst() wird der erste Treffer aus dem definierten Werteraum, basierend auf dem Stream-Inhalt, als Optional geliefert. Das bedeutet aber auch, dass es nicht zwingend der erste aus der Reihenfolge der Eingangswerte sein muss. Listing 3.6 liefert das erste Element aus dem Stream ("AB","AAB","AAAB","AAAAB","AAAAAB") zurück, das eine Zeichenfolge „AAA“ enthält, in unserem Beispiel also immer „AAAB“. Was aber passiert, wenn man das als ParallelStream definiert (Listing 3.7, zweite Version)? Es kann passieren, dass irgendein Value aus der Werteliste, das dem Kriterium entspricht, geliefert wird, da der Stream parallel abgearbeitet wird.

```
Stream
  .of("AB" , "AAB" , "AAAB" , "AAAAB" , "AAAAAB")
  .findFirst()
```

```
  .ifPresent(System.out::println);

Stream
  .of("AB" , "AAB" , "AAAB" , "AAAAB" , "AAAAAB")
  .parallel()
  .findFirst()
  .ifPresent(System.out::println);
```

Listing 3.7

Die Methode findFirst() gehört zu den terminalen Methoden. Das bedeutet, dass nach dem Aufruf von findFirst() keine weiteren Stream-Operationen durchgeführt werden können, der Stream wird terminiert. Mit dem Einsatz von findFirst() können recht komplexe Muster abgebildet werden, um einzelne Exemplare aus dem Stream zu erhalten. Da es sich bei den Streams um Pipelines handelt, werden immer nur so viele Elemente verarbeitet, wie zum Finden dieses einen Elements notwendig sind. Im Vergleich zu den Ausdrücken in der herkömmlichen Notation, sind die Ausdrücke mittels Streams meist um einiges kompakter. Der Einsatz von findFirst() eignet sich überall dort, wo eine deklarative mengenorientierte Beschreibung der einzelnen Entität nicht angewandt werden kann und demnach ein imperativer Weg notwendig ist.

3.5 reduce – bring es auf einen Nenner

Bei den bisherigen Betrachtungen wurden ausschließlich Transformationen betrachtet, die eine Abbildung von n auf m darstellten. Die Methode reduce((v1,v2)->) ermöglicht jedoch die Abbildung von n Elementen auf ein finales Element. Hierbei liegt das besondere Augenmerk auf dem unterschiedlichen Verhalten bei seriellen und parallelen Streams.

Alle bisher betrachteten Methoden waren nicht in der Lage, beispielsweise Elemente der Position n-1 in die Verwertung des Elements n mit einzubeziehen. Wie also kann man aufeinander aufbauende Werte erzeugen? Nehmen wir folgendes Beispiel: Dem Wert n soll immer der Wert n-1 angehängt werden. Die Eingangswerte sind die Zeichen der Kette „A, B, C, D, E“. Diese Elemente sollen verbunden werden. Listing 3.8 zeigt die erste Version mit seriellem und parallelem Stream. Die Methode reduce((v1,v2)->) bekommt ein Lambda mit zwei Parametern, V1 und V2, deren Inhalt aus dem Stream die Elemente n-1 und n sind. Bei beiden Versionen kommt dieselbe Zeichenfolge, „ABCDE“, verpackt in einem Optional heraus. Das ist verwunderlich, da doch die zweite Version ein paralleler Stream ist.

```
final List<String> demoValues
  = Arrays.asList("A" , "B" , "C" , "D" , "E");

demoValues
  .stream()
  .reduce(String::concat)
  .ifPresent(System.out::println);
```

```
demoValues
  .parallelStream()
  .reduce(String::concat)
  .ifPresent(System.out::println);
```

Listing 3.8

Ändern wir also die Implementierung aus Listing 3.8 zu der Implementierung wie in Listing 3.9, um ein wenig mehr zu sehen. Das Ergebnis ist nun für die serielle Version #ABCDE und für die parallele Version #A#B#C#D#E.

```
final List<String> demoValues
  = Arrays.asList("A" , "B" , "C" , "D" , "E");

final String reduceA = demoValues
  .stream()
  .reduce("#" , String::concat);
System.out.println("reduceA = " + reduceA);

final String reduceB = demoValues
  .parallelStream()
  .reduce("#" , String::concat);
System.out.println("reduceB = " + reduceB);
```

Listing 3.9

Nun stellt sich die Frage, in welchen Teilschritten die jeweiligen Ergebnisse produziert werden. Dazu erweitern wir ein wenig die Ausgabe, und zwar mit einem Postfix (Listing 3.10).

```
final List<String> demoValues
  = Arrays.asList("A", "B", "C", "D", "E");

final String reduceA = demoValues
  .stream()
  .reduce("X_" , (v1 , v2) -> {
    System.out.println("v1 -> " + v1);
    System.out.println("v2 -> " + v2);
    return v1.concat(v2) + "#";
  });
System.out.println(reduceA);

final String reduceB = demoValues
  .parallelStream()
  .reduce("X_" , (v1 , v2) -> {
    System.out.println("v1 -> " + v1);
    System.out.println("v2 -> " + v2);
    return v1.concat(v2) + "#";
  });
System.out.println(reduceB);
```

Listing 3.10

Nun kann man einfach die einzelnen Schritte erkennen, die im Fall des parallelen Streams erfolgen. Die serielle Version ergibt den String X_A#B#C#D#E#. Hier wird also das Präfix, dann jedes Element mit einem „#“ konkateniert ausgegeben, alles in der Reihenfolge, wie es im Quell-Stream vorhanden ist. Bei der parallelen Version sieht das Ergebnis gänzlich anders aus: X_A#X_B##X_C#X_D#X_E####. Hier lohnt es sich, die einzelnen Schritte (Listing 3.11) genauer anzusehen. Wichtig zu wissen ist, dass bei jedem Durchlauf das Ergebnis in der Reihenfolge der verarbeiteten Elemente anders aussehen kann.

```
v1 -> X_
v1 -> X_
v1 -> X_
v1 -> X_
v2 -> C
v1 -> X_
v2 -> B
v2 -> E
v2 -> D
v2 -> A
v1 -> X_A#
v2 -> X_B#
v1 -> X_D#
v2 -> X_E#
v1 -> X_C#
v2 -> X_D#X_E##
v1 -> X_A#X_B##
v2 -> X_C#X_D#X_E###
X_A#X_B##X_C#X_D#X_E####
```

Listing 3.11: „reduce“-Output, parallel

Verfolgt man die einzelnen Schritte, erkennt man die Aufteilung, die durch die Abarbeitung des parallelen Streams erfolgt. Dadurch werden unterschiedliche Teilkomponenten konkateniert, die Summe ist jedoch konstant, da der Wertevorrat gleich bleibt, genauso wie die Reihenfolge, mit der die Daten in den Stream gelangen. A und B bilden das erste Wertepaar, das ausgewertet wird, C und D sind das zweite Paar, dann folgt E. Die Teilergebnisse werden zusammengeführt. Deswegen sind an B zwei # und an E vier #.

Die Methode reduce ermöglicht es, die Werte aus dem Quell-Stream miteinander zu verarbeiten, um ein einzelnes Ergebnis zu erhalten. Hierbei ist es wichtig, die Unterscheidung zwischen der seriellen und parallelen Verarbeitung zu beachten. Die Ergebnisse können unterschiedlich sein, wobei es auf die jeweilige Reduktionstransformation ankommt. Bei trivialen Dingen, wie dem Auffinden eines Maximalwerts, kommt es nicht zu Randerscheinungen. Jedoch sollte man durchaus auch bei vermeintlich trivialen Transformationen den Test machen, ob das Ergebnis immer noch dem gewünschten Resultat entspricht.

Bei der Verwendung von Streams findet man viele grundlegende Funktionen, die im API schon enthalten sind und einem die Entwicklung von Basic Utilities ersparen. Diese werden wir uns

nun ansehen und anhand kleiner Beispiele ihre Verwendung zeigen.

3.6 limit/skip – bitte nicht alles

Streams können undefiniert lang sein. Das bedeutet, dass ein Stream im Extremfall kein Ende hat. Es ist also manchmal sinnvoll, Streams nur bis zu einer bestimmten Länge abzuarbeiten, oder nur eine bestimmte Menge von Ergebnissen zu sammeln, da der Rest nicht mehr für die nachfolgende Logik zu verwenden ist. Die Methode limit(count) ist genau dafür gedacht. Das nachfolgende Beispiel (Listing 3.12) zeigt, wie die initiale Menge reduziert und die Menge der Ergebnisse beschränkt werden kann. Es werden also immer ab der Stelle, an der die Methode limit(count) aufgerufen wird, die restlichen Schritte auf das angegebene Limit begrenzt.

```
final List<Integer> demoValues
  = Arrays.asList(1,2,3,4,5,6,7,8,9,10);
  //limit the input -> [1, 2, 3, 4]
  System.out.println(demoValues
    .stream().limit(4)
      .collect(Collectors.toList()));
  //limit the result -> [5, 6, 7, 8]
  System.out.println(demoValues
    .stream().filter((v)->v > 4)
    .limit(4)
    .collect(Collectors.toList()));
```

Listing 3.12

Mit der Methode skip(count) verhält es sich ein klein wenig anders. Hier ist auch eine Begrenzung des Streams vorhanden, jedoch handelt es sich hier um eine absolute Grenze. Der Counter gibt an, wie viele Elemente übersprungen werden. Das Ende ist allerdings offen. Die Begrenzung findet somit zu Beginn statt, indem n Elemente einfach übersprungen und nicht verarbeitet werden (Listing 3.13). Die Methode skip(counter) kann auch mehrfach und an verschiedenen Stellen im Gesamtkonstrukt vorkommen.

```
//jumping over the first 4 elements -> [5, 6, 7, 8, 9, 10]
System.out.println(demoValues
  .stream().skip(4)
  .collect(Collectors.toList()));
```

Listing 3.13

3.7 distinct – alles nur einmal, bitte

Aus dem Bereich SQL kennt man den Befehl DISTINCT, um eine Menge von Werten auf jeweils nur ein Exemplar eines Werts zu reduzieren, also das Erzeugen einer Unique-Menge. Genau dasselbe erledigt die Methode distinct(). Die Implementierung (Listing 3.14) selbst arbeitet in der Klasse DistinctOps auf einer ConcurrentHashMap, da diese Operation auch für

parallele Streams entwickelt worden ist. Die distinct-Menge ist dann das KeySet der HashMap. Das ausschlaggebende Element ist die hashCode- und equals-Implementierung der Elemente, die dort in die Unique-Menge überführt werden sollen. An dieser Stelle kann man das Verhalten und die Performance der distinct-Operation beeinflussen.

```
// [77, 79, 81, 95, 43, 10, 53, 48,
//  74, 68, 60, 86, 83, 24, 57, 28, 8,
//  85, 70, 66, 20, 14, 97, 73, 22,
//  36, 40, 39, 32, 19, 41, 67, 25, 88]
final Random random = new Random();
  System.out.println(
  Stream.generate(() -> random.nextInt(100))
    .limit(40)
    .distinct()
    .collect(Collectors.toList())
);
```

Listing 3.14

3.8 min/max – ganz klein, ganz groß

Die Methoden min(<Comparator>) und max(<Comparator>) liefern aus der Menge der Werte im Stream das Minimum bzw. das Maximum (Listing 3.15). Dieser Wert wird mittels Comparator ermittelt. Das hat zur Folge, dass alle Elemente durchlaufen werden müssen. Es kann also nicht auf unendlichen Streams ausgeführt werden. Die Definition des Comparators lässt entsprechend verschiedene Interpretationen zu, was ein Minimum und was ein Maximum ist. Gleichzeitig ist die Implementierung des Comparators eines der bestimmenden Glieder bei der Performance, da es auf alle Elemente angewandt wird. Es ist auf jeden Fall schneller als das Sortieren der Elemente mit anschließendem findFirst(), da die Komplexität von min/max bei O(n) liegt und die Komplexität des Sortierens bei O(n log n).

```
//find the maximum
System.out.println(demoValues
  .stream().max(Integer::compareTo));
//find the BUG ;-)
System.out.println(demoValues
  .stream().min((v1, v2) -> Integer.compare(v2, v1)));
```

Listing 3.15

3.9 allMatch, anyMatch, noneMatch und count

Die Methoden allMatch(<Predicate>), anyMatch(<Predicate>), noneMatch(<Predicate>) liefern ein Boolean zurück: allMatch, wenn die definierte Bedingung bei genau allen Elementen zutrifft; anyMatch, wenn einige Elemente der Bedingung entsprechen (mind. zwei); noneMatch, wenn kein einziges Element der Bedingung entspricht.

Sieht man sich die Laufzeit der einzelnen Methoden an, kann man feststellen, dass noneMatch(<Predicate>) auf den gesamten Wertevorrat angewandt werden muss. anyMatch(<Predicate>) und allMatch(<Predicate>) hingegen brechen ab, sobald das Ergebnis ableitbar ist (Listing 3.16). In unserem Fall bricht allMatch(<Predicate>) nach genau einem Vergleich ab, da dieser schon nicht passt. (Ungerade Zahl) nach anyMatch(<Predicate>) bricht nach genau zwei erfolgreichen Treffern ab, da die Bedingung any erfüllt ist (Listing 3.17).

```
// true, some are matching
System.out.println("anyMatch " + demoValues.stream()
  .map((e) -> {
    System.out.println("e = " + e);
    return e;
  })
  .anyMatch((v) -> v % 2 == 0));
//false, not all are matching
System.out.println("allMatch " + demoValues.stream()
  .map((e) -> {
    System.out.println("e = " + e);
    return e;
  })
  .allMatch((v) -> v % 2 == 0));
//false, not all are NOT matching
System.out.println("noneMatch " + demoValues.stream()
  .map((e) -> {
    System.out.println("e = " + e);
    return e;
  })
  .noneMatch((v) -> v % 2 == 0));
```

Listing 3.16

```
e = 1
e = 2
anyMatch true
e = 1
allMatch false
e = 1
e = 2
noneMatch false
e = 1
e = 2
e = 3
e = 4
e = 5
e = 6
e = 7
e = 8
e = 9
e = 10
```

Listing 3.17: „Match“-Output

Nun fehlt nur noch die Methode count(). Sie ist schnell erklärt, da sie die Anzahl der Elemente zurückliefert, die bis zu diesem Schritt in der Verarbeitung des Streams gekommen sind (Listing 3.18).

```
//5 matching the filter, 2,4,6,8,10
System.out.println("count " + demoValues.stream()
  .map((e) -> {
    System.out.println("e = " + e);
    return e;
  })
  .filter((v) -> v % 2 == 0)
  .count());
```

Listing 3.18

3.10 parallel/sequential – umschalten, wenn nötig

Zwei weitere Methoden, die wir uns hier ansehen werden, sind parallel() und sequential(). Diese Methoden, die wiederum einen Stream zurückliefern, können explizit in eine serielle oder parallele Version geschaltet werden. Sollte eine nachfolgende Operation nicht parallel durchführbar sein, kann das mit dem Methodenaufruf seriell geschehen. Es kann bei jedem Stream einzeln entschieden werden, ob parallel oder seriell gearbeitet wird (Listing 3.19).

```
System.out.println(demoValues.stream()  //seriell
  .map((m1) -> m1)
  .parallel()
  .map((m2) -> m2)
  .sequential() //seriell
  .collect(Collectors.toList()));
```

Listing 3.19

3.11 ofNullable

Mit der Methode Stream.of(..) kann man Streams erzeugen. Nachteilig allerdings war es bisher, dass man sicherstellen musste, dass keine Nullwerte in der Menge enthalten waren. Seit dem JDK 9 gibt es die Methode ofNullable(..). Bei einem Nullwert wird daraus ein leerer Stream erzeugt und es gibt demnach auch keine NullPointerException mehr.

3.12 dropWhile/takeWhile

Diese beiden Methoden wurden eingeführt, da sich zeigte, dass es immer wieder notwendig ist, eine bestimmte Menge von Elementen zu verarbeiten, bis ein bestimmter Wert erreicht worden ist (takeWhile()), oder sobald ein bestimmter Wert erreicht worden ist (dropWhile()). Das Erkennen der Grenze wird mittels Predicate realisiert. Listing 3.20 zeigt ein Beispiel unter der

Verwendung von takeWhile(). Die Ausgabe ist in diesem Fall zweimal 1234.

```
Stream
  .of(1,2,3,4,5,6,7,8,9)
  .takeWhile(new Predicate<Integer>() {
    @Override
    public boolean test(Integer integer) {
      return integer < 5;
    }
  }).forEach(System.out::print);

Stream
  .of(1,2,3,4,5,6,7,8,9)
  .takeWhile(i -> i < 5)
  .forEach(System.out::print);
```

Listing 3.20

3.13 static Method – iterate

Die statische Methode iterate() ermöglicht es, einen Stream zu erzeugen, der einen unendlichen Wertevorrat hat. Der Vorteil dieser Variante im Vergleich zum Supplier liegt darin, dass die Funktion zum Erzeugen des neuen Werts nicht selbst den vorherigen Wert kennen muss.

Um solch einen Stream zu erhalten, in unserem Beispiel einen Stream, der einfach Integerwerte liefert, die immer zum Wert n-1 zwei hinzu addieren, kann man Folgendes schreiben (Listing 3.21), wobei wir bei dem Wert drei beginnen.

```
Stream.iterate(3 , v -> v + 2)
  .limit(4)
  .forEach(System.out::println);
```

Listing 3.21

In Listing 3.21 ist eine Besonderheit zu beachten. Um den Stream ab einem bestimmten Wert zu terminieren, wurde mit limit(4) eine Stufe eingebaut. Ohne diese würden wir einen unendlichen Stream erhalten.

In Java 9 wurde eine zweite Methodensignatur hinzugefügt, in der wir ein Predicate mit angeben können, das zur Terminierung verwendet wird (Listing 3.22).

```
Stream.iterate(3 , v -> v < 10 , v -> v + 2)
  .forEach(System.out::println);
```

Listing 3.22

3.14 Zusammenfassung

Schon diese wenigen Basismethoden ermöglichen es nach sehr kurzer Einarbeitung, Streams

recht effizient und effektiv einzusetzen. Zur Übung kann ich empfehlen, bestehende Quelltexte in Konstrukte mit Streams umzuformen. Dabei wird sich zeigen, dass mit dieser Transformation eine starke Codereduktion einhergeht. An so mancher Stelle kann man dank der Streams auch Teilaufgaben parallelisieren, was zu einer höheren Auslastung der vorhandenen modernen CPU-Architekturen führt. Der Umbau wird sich lohnen.

4 StreamSupport

Kommen wir nun zur Klasse StreamSupport, die uns zu den Basiselementen eines Streams führt. Die hier vorgestellten Methoden werden für die Erzeugung eigener Streams verwendet. In der API-Dokumentation wird diese Klasse als low-level bezeichnet.

Da die einzelnen Methoden im Aufbau alle gleichförmig sind, werden wir uns an dieser Stelle lediglich die Methode zur Erzeugung eines Stream<T> ansehen. Die Methoden der Klasse StreamSupport sind allesamt statisch, können also direkt aufgerufen werden und sind vom Verhalten her Factory-Methoden. Es gibt hier zwei Ausprägungen (beide in Listing 4.1):

- Die erste Version hat zwei Parameter, eine Instanz vom Typ Spliterator und ein Boolean, um anzugeben, ob es sich um einen parallelen Stream handeln soll.
- Die zweite Variante erwartet einen Supplier für einen Spliterator, ein int zur Angabe der Charakteristiken und ebenfalls einen Boolean zur Angabe, ob es ein paralleler Stream werden soll.

```
public static <T> Stream<T> stream(Spliterator<T> spliterator, boolean parallel)

public static <T> Stream<T> stream(Supplier<? extends Spliterator<T>> supplier, int
 characteristics, boolean parallel)
```

Listing 4.1: „StreamSupport“ in JDK 9

4.1 Spliterators

Wie wir gerade gesehen haben, benötigen wir Instanzen vom Typ Spliterator. Auch hier gibt es eine Klasse, ähnlich der Klasse Collectors, die uns eine Reihe von Hilfsmethoden zur Erzeugung eines Spliterators anbietet. Diese Methoden gibt es auch zur Erzeugung der Varianten, die verwendet werden können, um einen IntStream, DoubleStream usw. zu erzeugen. Wir werden hier das Augenmerk auf die Erzeugung von regulären Streams werfen.

4.2 Klasse „Spliterators“

Kommen wir zur ersten Frage: Wozu benötigen wir diese Spliterators und was für eine Aufgabe sollen sie eigentlich übernehmen?

Die Aufgabe eines Spliterators ist recht einfach erklärt: Er soll Streams aufteilen können. Damit werden die Arbeitspakete erzeugt, die dann parallel abgearbeitet werden können. Für die Erzeugung von sequenziellen Streams werden Spliterators benötigt, die lediglich ein einziges

Arbeitspaket zurückliefern. Das hier angedeutete Verfahren stammt noch von Java 7, als der ForkJoinPool eingeführt worden ist. Die Implementierungen des Streams-API basieren auf der Verwendung genau dieses Default-ForkJoinPools. Um die Aufgaben zu verteilen, werden einzelne Arbeitspakete benötigt, die unabhängig voneinander in diesem Pool abgearbeitet werden können. Unabhängig meint nicht vollständig voneinander losgelöst, können doch Arbeitspakete im Ergebnis auf den Teilergebnissen der Unterpakete beruhen. Wir werden darauf nochmals bei der Besprechung der Implementierung von Spliterators eingehen.

emptySpliterator()

Diese Methode liefert einen Spliterator<T> zur Erzeugung von sequenziellen Streams.

iterator(..)

Mithilfe der Methode Iterator<T> iterator(Spliterator<? extends T> spliterator) kann man einen Spliterator in einen Iterator transformieren. Dieser Iterator kann dann wieder wie gewohnt verwendet werden.

spliterator(Collection, characteristics)

Um einen Spliterator zu erzeugen, erwartet diese Methode die Angabe einer Collection und eines int-Werts zur Angabe der Charakteristik. Die Collection kann jede beliebige Variante sein, zum Beispiel ein ArrayList:

```
Spliterator<T> spliterator(Collection<? extends T> c, int characteristics)
```

Was hat es nun mit diesen Charakteristiken auf sich? Hier wird ein int-Wert übergeben, der eine Bitmaske repräsentiert (Listing 4.2). Um diese zu erzeugen, sollte man die dafür vorgesehenen Konstanten aus dem Interface Spliterator verwenden (Tabelle 4.1).

CONCURRENT	Zeigt an, dass die verwendete Quelle concurrent modifiziert werden kann
DISTINCT	Die Elementquelle liefert keinen Wert mehr als einmal
SORTED	Die Elemente der Quelle sind sortiert
SIZED	Die Quelle der Elemente ist endlich, und die Anzahl aller Elemente ist bekannt
SUBSIZED	Die Anzahl einer Untermenge (nach dem Aufteilen) ist bekannt
IMMUTABLE	Die Quelle der Elemente kann nicht modifiziert werden
NONNULL	Die Quelle enthält keine Null-Elemente
ORDERED	Die Elemente sind in der Quelle strikt sortiert, Zugriff per Index ist möglich

Tabelle 4.1 Spliterator Characteristics

```
final Spliterator<String> spliterator = Spliterators
  .spliterator(
    asList("1" , "2" , "3") ,
    Spliterator.NONNULL);

out.println("spliterator.getExactSizeIfKnown() = " + spliterator.getExactSizeIfKnow
n());
out.println("spliterator.estimateSize() = " + spliterator.estimateSize());
```

Listing 4.2

Die Ausgabe ist hier in beiden Fällen der Wert 3. Das hat den Grund, dasS immer, wenn nicht CONCURRENT mit angegeben wird, die Werte für SIZED und SUBSIZED gesetzt werden (Listing 4.3).

```
public static <T> Spliterator<T> spliterator(Collection<? extends T> c, int charact
eristics) {
  return new IteratorSpliterator<>(Objects.requireNonNull(c), characteristics);
}

public IteratorSpliterator(Collection<? extends T> collection, int characteristics)
 {
  this.collection = collection;
  this.it = null;
  this.characteristics = (characteristics & Spliterator.CONCURRENT) == 0 ? characte
ristics |
   Spliterator.SIZED | Spliterator.SUBSIZED : characteristics;
}
```

Listing 4.3: JDK-9-Spliterators

spliterator(Iterator, size, characteristics)

Der hier erzeugt Spliterator basiert auf einem Iterator und der Angabe einer initialen Menge:

```
Spliterator<T> spliterator(Iterator<? extends T> iterator, long size, int character
istics)
```

Das nachfolgende Beispiel in Listing 4.4 liefert ebenfalls die beiden Werte 3, obwohl in der Liste vier Werte enthalten sind.

```
final List<String> list = asList("1" , "2" , "3", "4");

final Spliterator<String> spliterator = Spliterators
  .spliterator(list.iterator() , 3, Spliterator.NONNULL);

out.println("spliterator.getExactSizeIfKnown() = " + spliterator.getExactSizeIfKnow
n());
out.println("spliterator.estimateSize() = " + spliterator.estimateSize());
```

Listing 4.4

spliterator(Object[], characteristics)

Basis für diesen Spliterator ist das übergebene Array:

```
Spliterator<T> spliterator(Object[] array, int additionalCharacteristics)
```

spliterator(Object[], fromIndex, toIndex, characteristics)

Auch hier ist das übergebene Array die Quelle von Elementen, die vom Spliterator verwendet werden. Nur werden in diesem Fall die Elemente durch die Angabe der beiden Grenzen reduziert. Wir arbeiten demnach auf einer Teilmenge aus dem Array:

```
Spliterator<T> spliterator(Object[] array, int fromIndex, int toIndex, int addition
alCharacteristics)
```

spliteratorUnknownSize

Mithilfe dieser Methode werden Spliteratoren erzeugt, die auf einem Iterator basieren und keine Angabe über die Menge der Elemente haben. Es werden intern explizit die Werte für SIZED und SUBSIZED aus den übergebenen Charakteristiken entfernt:

```
Spliterator<T> spliteratorUnknownSize(Iterator<? extends T> iterator, int character
istics)
```

4.3 Interface „Spliterator“

Nachdem wir die Servicemethoden der Klasse Spliterators betrachtet haben, beginnen wir nun; eigene Spliterators zu schreiben. Das ist immer dann sinnvoll, wenn man zum Beispiel auf Datenstrukturen arbeitet, die einen effizienten Weg zur Partitionierung ermöglichen, um mittels parallelen Streams ein Speed-up bei der Verarbeitung zu erreichen.

Aber beginnen wir ganz am Anfang. Das Interface Spliterator definiert vier Methoden, die es zu implementieren gilt (Listing 4.5).

```
final Spliterator<String> spliterator = new Spliterator<>() {
  @Override
  public boolean tryAdvance(Consumer<? super String> action) {
    return false;
  }

  @Override
  public Spliterator<String> trySplit() {
    return null;
  }

  @Override
  public long estimateSize() {
```

```
    return 0;
  }

  @Override
  public int characteristics() {
    return 0;
  }
};
```

Listing 4.5

characteristics()

Der Rückgabewert ist die Kombination von int-Werten, basierend auf den Konstanten in Tabelle 4.1. Um die gewünschte Kombination zu erhalten, werden die jeweiligen Konstanten mit einem logischen ODER verknüpft zurückgeliefert.

estimateSize()

Hier wird eine Aussage über die Menge an Elementen zurückgegeben. Wenn es keine Aussage gibt, bzw. der Wert nicht bestimmbar ist, wird Long.MAX_VALUE als Rückgabewert gewählt. Nicht bestimmbar kann auch bedeuten, dass die Berechnung zu aufwendig ist und deswegen darauf verzichtet wird. Wie der Wert ermittelt wird, hängt natürlich von der Datenquelle selbst ab.

trySplit()

Wenn dieser Spliterator partitioniert werden kann, liefert diese Methode einen Spliterator, der eine Teilmenge bearbeitet. Diese Teilmenge wird vom ursprünglichen Spliterator nicht weiter berücksichtigt. Kann keine Teilmenge, kein dafür verantwortlicher Spliterator erzeugt werden, wird null zurückgeliefert.

tryAdvance(Consumer<? super String> action)

Mit dieser Methode kann man einen Consumer übergeben, der auf das nächste Element angewandt wird. Ist kein weiteres Element vorhanden, wird ein false zurückgeliefert, ansonsten ein true. Exceptions werden nach oben an die aufrufende Instanz durchgereicht.

4.4 Beispiel

Als Beispiel werden wir eine Enumeration in einen Stream umwandeln. Die Enumeration besteht aus einer endlichen und fest definierten Anzahl an Elementen eines Typs. Nun kann man zum Beispiel eine der Hilfsmethoden der Klasse Spliterators verwenden. In Listing 4.6 wird ein Spliterator basierend auf einem Iterator gebaut.

```
static <T> Function<Enumeration<T>, Stream<T>> enumToStream() {
  return (e) ->
```

```
    StreamSupport
      .stream(Spliterators.spliteratorUnknownSize(new Iterator<T>() {
        public T next() {
          return e.nextElement();
        }

        public boolean hasNext() {
          return e.hasMoreElements();
        }
      }, Spliterator.ORDERED), false);
}
```

Listing 4.6

Allerdings ist die Menge definiert, demnach müsste auch ein anderer Weg funktionieren. Definieren wir in Listing 4.7 hierfür erst einmal unsere Enumeration.

```
enum ELEMENTS {
  a, b, c, d, e, f, g, h, i, j, k
}
```

Listing 4.7

Als Nächstes implementieren wir das Interface Spliterator (Listing 4.8).

```
private static class ElementsSpliterator implements Spliterator<ELEMENTS> {

  private ELEMENTS[] elements;
  private int left = 0;
  private int right = 0;

  public ElementsSpliterator(ELEMENTS[] elements) {
    System.out.println("constructor elements = " + elements.length);
    System.out.println("constructor elements = " + Arrays.asList(elements));

    this.elements = elements;
    left = 0;
    right = elements.length -1;
    System.out.println("constructor elements = left = " + left + " right = " + righ
t);
  }

}
```

Listing 4.8

In Listing 4.8 gibt es einige System.out-Anweisungen, die lediglich erklärenden Zweck haben. In produktiven Quelltexten hat so etwas natürlich nichts zu suchen.

Wir erhalten das Array und die rechte und linke Grenze der betrachteten Daten darin. Die ersten beiden Methoden, die wir in Listing 4.9 implementieren, sind recht einfach und meines

Erachtens selbsterklärend.

```
@Override
public long estimateSize() {
  return elements.length;
}

@Override
public int characteristics() {
  return DISTINCT | SIZED | SUBSIZED | NONNULL;
}
```

Listing 4.9

Kommen wir zu der Methode, die sich damit beschäftigt, den Spliterator aufzuteilen (Listing 4.10).

```
@Override
public Spliterator<ELEMENTS> trySplit() {
  System.out.println("trySplit = " + Arrays.asList(elements));
  if (estimateSize() < 3) return null;
  final int lowerBorder = (int) estimateSize() / 2;
  System.out.println("lowerBorder = " + lowerBorder);
  left = 0;
  final ElementsSpliterator result = new ElementsSpliterator
    (copyOfRange(elements , lowerBorder , elements.length));
  elements = copyOfRange(elements , 0 , lowerBorder );
  right = elements.length -1;
  System.out.println("trySplit (after) = " + Arrays.asList(elements));
  return result;
}
```

Listing 4.10

Als Erstes wird eine Schranke definiert, ab der eine weitere Aufteilung keinen Sinn mehr ergibt. Hier ist das der Fall, wenn die Anzahl der Elemente kleiner/gleich 3 ist. Danach teilen wir die Menge der Elemente in zwei Teile. Aus dem oberen Teil erzeugen wir eine neue Instanz unserer Spliterator-Implementierung. Der untere Teil wird als neuer Wertevorrat gesetzt.

Kommen wir nun zum Konsumieren mittels tryAdvance. Hier gehen wir lediglich die noch vorhandenen Elemente durch und wenden den Consumer darauf an (Listing 4.11).

```
@Override
public boolean tryAdvance(Consumer<? super ELEMENTS> action) {
  System.out.print("tryAdvance -> ");
  if(left == right) return false;

  action.accept(elements[left]);
  left++;
  return left <= right;
}
```

Listing 4.11

4.5 Zusammenfassung

Sicherlich ist diese Implementierung nicht für die Produktion geeignet, zeigt jedoch deutlich, wie ein Spliterator arbeitet.

Nachfolgend noch ein Beispiel, um zu zeigen, wie der Spliterator funktioniert (Listing 4.12). Hier wird explizit eine Teilung provoziert und die beiden Teilmengen werden einzeln durchlaufen.

```
final ElementsSpliterator spliteratorA = new ElementsSpliterator(ELEMENTS.values())
;
final Spliterator<ELEMENTS> spliteratorB = spliteratorA.trySplit();

System.out.println("spliteratorA.estimateSize() = " + spliteratorA.estimateSize());
System.out.println("spliteratorB.estimateSize() = " + spliteratorB.estimateSize());

StreamSupport.stream(spliteratorA , false).forEach(System.out::println);
System.out.println("====== ");
StreamSupport.stream(spliteratorB , false).forEach(System.out::println);
```

Listing 4.12

Die Ausgabe auf der Kommandozeile ist dann wie in Listing 4.13 zu sehen.

```
constructor elements = 11
constructor elements = [a, b, c, d, e, f, g, h, i, j, k]
constructor elements = left = 0 right = 10
trySplit = [a, b, c, d, e, f, g, h, i, j, k]
lowerBorder = 5
constructor elements = 6
constructor elements = [f, g, h, i, j, k]
constructor elements = left = 0 right = 5
trySplit (after) = [a, b, c, d, e]
spliteratorA.estimateSize() = 5
spliteratorB.estimateSize() = 6
tryAdvance -> a
tryAdvance -> b
tryAdvance -> c
tryAdvance -> d
tryAdvance -> ======
tryAdvance -> f
tryAdvance -> g
tryAdvance -> h
tryAdvance -> i
tryAdvance -> j
tryAdvance ->
```

Listing 4.13

5 Kleinere Streams-Pattern-Beispiele

Im Folgenden zeige ich anhand von drei kleineren Beispielen, was mit Java 9 Streams unter anderem machbar ist.

5.1 Ist es eine Primzahl?

Als kleines Anfangsbeispiel hier eine Implementierung, mit der sich herausfinden lässt, ob es sich bei einer Zahl um eine Primzahl handelt. Die Methode isPrime2 ist ohne Streams realisiert, isPrime1 ist das mittels Stream realisierte Äquivalent. Bei dieser recht einfachen Implementierung ist der Unterschied zwischen den Implementierungen noch recht klein. Es zeigt aber schon die unterschiedliche Lesbarkeit, meines Erachtens nach zugunsten der Stream-Version (Listing 5.1).

```
public static void main(String[] args) {
  for(int i = 0; i<1_000_000; i++){
    final boolean b = isPrime1(i) != isPrime2(i);
    if(b)
      System.out.println("ungleiches Ergebnis =  " + i);
  }
}
public static boolean isPrime1(int n) {
  if (n <= 1) return false;
  if (n == 2) return true;
  return n >= 2 && IntStream
    .rangeClosed(2, (int) (Math.sqrt(n)))
    .allMatch((d) -> n % d != 0);
}
public static boolean isPrime2(int n) {
  if (n <= 1) return false;
  if (n == 2) return true;
  if (n % 2 == 0) return false;
  for (int i = 3; i <= Math.sqrt(n) + 1; i = i + 2) {
    if (n % i == 0) return false;
  }
  return true;
}
```

Listing 5.1

5.2 Fibonacci als Stream

Einem Stream kann man einen Supplier als Instanz übergeben. Das bedeutet auch, dass dort immer dieselbe Instanz verwendet wird, solange der Stream verwendet wird. Das kann man

verwenden, um z. B. aufeinander aufbauende Zahlenfolgen zu generieren. In Listing 5.2 erfolgt eine triviale Implementierung, um Fibonacci-Zahlen zu erzeugen.

```
public static void main(String[] args) {
  final Stream<Long> fibStream = makeFibStream(10);
  fibStream.forEachOrdered(System.out::println);
}
public static Stream<Long> makeFibStream() {
  return(Stream.generate(new FibonacciSupplier()));
}
public static Stream<Long> makeFibStream(int numFibs) {
  return(makeFibStream().limit(numFibs));
}
public static List<Long> makeFibList(int numFibs) {
  return(makeFibStream(numFibs)
    .collect(Collectors.toList()));
}
public static class FibonacciSupplier implements Supplier<Long> {
  private long previous = 0;
  private long current = 1;
  @Override
  public Long get() {
    long next = current + previous;
    previous = current;
    current = next;
    return(previous);
  }
}
```

Listing 5.2

Hier gibt es allerdings einige Nachteile, oder besser gesagt Dinge, die auf jeden Fall berücksichtigt werden müssen.

Wenn eine Supplier-Instanz erzeugt wird, muss sichergestellt sein, dass diese nicht zur Erzeugung von mehr als einem Stream verwendet wird. Wenn dem so ist, kann kann es zu Nebenläufigkeitsproblemen kommen, da beide Streams unabhängig voneinander arbeiten. Der nächste Punkt ist, dass Supplier die zusätzlichen Zustände intern halten auch bei der Erzeugung einer weiteren Streaminstanz. Auch hier kann es zu ungewollten Verhaltensmustern kommen. Demnach kann man als Regel formulieren, dass ein Supplier immer nur für einen Stream gelten sollte.

5.3 Matrix als Stream

Mit Streams kann man auch elegant auf einer n-dimensionalen Matrix arbeiten. Im folgenden Beispiel (Listing 5.3) soll in einer zweidimensionalen Matrix die Zahl 66 gesucht werden. Vereinfachend wird angenommen, dass sie nur einmal gefunden werden soll. Die Pre-Streams-Lösung basiert auf geschachtelten for-Schleifen mit einem Label an der äußersten Schleife.

Allgemein kann man folgende Transformationsregeln ableiten: Reine for-Schleifen können auf forEach abgebildet werden, wenn kein Abbruch beim Schleifendurchlauf erfolgen soll. Soll innerhalb einer for-Schleife eine Bedingung mittels if geprüft werden, dann gibt es zwei Varianten:

- if ohne else kann auf die Methode filter gemappt werden.
- if mit else wird auf die Map-Methode gemappt, innerhalb derer die Fallunterscheidung durchgeführt wird.

Es ist also stark abhängig vom Kontrollfluss, ob sich eine Transformation in Streams lohnen wird.

```
public static void main(String[] args) {
  final List<List<Integer>> matrix = new ArrayList<>();
  matrix.add(Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9));
  matrix.add(Arrays.asList(1,2,3,4,5,66,7,8,9));
  matrix.add(Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9));
  matrix.forEach(System.out::println);

  final Integer s = matrix.stream()
    .map(l -> l.stream()
      .filter(v -> v.equals(66))
      .findFirst().orElse(null))
    .filter(f->f != null)
    .findFirst().orElse(null);
  System.out.println("s = " + s);

  Integer result = null;
  endPos:
  for (final List<Integer> integers : matrix) {
    for (final Integer integer : integers) {
      if(integer.equals(66)){
        result = integer;
        break endPos;
      }
    }
  }
  System.out.println("result " + result);
}
```

Listing 5.3

5.4 Zusammenfassung

Bei der Verwendung von Streams stellen sich unter anderem folgende Fragen: Ist eine Nebenläufigkeit gewünscht oder nicht? Wenn ja, dann sind Streams unter Verwendung von parallelStream() in vielen Fällen ein einfacher und schneller Ansatz.

Soll die Verschachtelung des Kontrollflusses gemindert werden? Hier ist es abhängig von den Konstrukten innerhalb der Fallunterscheidungen selbst. Mit leichten Veränderungen lassen sich mittels Streams recht oft aussagekräftigere Konstrukte aufbauen, die auf lange Sicht zu einer besseren Wartbarkeit führen. Ob sich das bei Altprojekten immer lohnt, muss im Einzelfall entschieden werden.

Sind mathematische Funktionen abzubilden? Hier kann man in vielen Fällen mittels Stream schneller zum Erfolg kommen, ohne gleich andere funktionale Sprachen in das Projekt zu integrieren.

Alles in allem sind Streams eine sehr effektive Unterstützung bei der täglichen Arbeit mit Java. Gerade durch den generischen Ansatz kann bei typischen Geschäftsanwendungen eine Erleichterung realisiert werden. Die Einarbeitung in Streams sollte normalerweise innerhalb von zwei bis drei Arbeitstagen zu spürbaren Ergebnissen führen.

Probieren Sie es aus!

Der Autor

Sven Ruppert arbeitet seit 1996 mit Java und ist Developer Advocate bei Vaadin. In seiner Freizeit spricht er auf internationalen und nationalen Konferenzen, schreibt für IT-Magazine und für Tech-Portale.

Table of Contents