4.11 Stream-API

Zusammen mit Lambda-Ausdrücken, die auf der Sprache-Seite stehen, ist in Java 8 eine ganz neue Bibliothek implementiert worden, die ein einfaches Verarbeiten von Datenmengen möglich macht – ihr Name: Stream-API. Im Zentrum stehen Operationen zum Filtern, Abbilden und Reduzieren von Daten aus Sammlungen. Die API ist im funktionalen Stil, und Programme lesen sich damit sehr kompakt:


```
Object[] words = { " ", '3', null, "2", 1, "" };
Arrays.stream( words )
      .filter( Objects::nonNull )
      .map( Objects::toString )
      .map( String::trim )
      .filter( s -> ! s.isEmpty() )
      .map( Integer::parseInt )
      .sorted()
      .forEach( System.out::println );  // 1 2 3
```

Während die Klassen aus der Collection-API optimale Speicherformen für Daten realisieren, ist es Aufgabe der Stream-API, die Daten komfortabel zu erfragen. Gut ist hier zu erkennen, dass die Stream-API das *Was* betont, nicht das *Wie*. Das heißt, Durchläufe und Iterationen kommen im Code nicht vor, sondern die Fluent-API beschreibt deklarativ, wie das Ergebnis aussehen soll. Die Bibliothek realisiert schlussendlich das Wie. So kann eine Implementierung zum Beispiel entscheiden, ob die Abarbeitung sequenziell oder parallel erfolgt, ob die Reihenfolge eine Rolle spielen muss oder alle Daten zwecks Sortierung zwischengespeichert werden müssen usw.

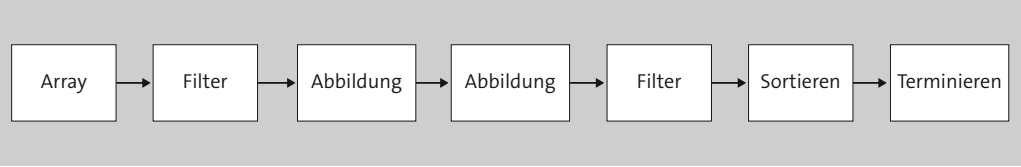

Abbildung 4.6 Pipeline-Prinzip bei Streams

Interne versus externe Iteration

Als Erstes fällt bei der Stream-API auf, dass die klassische Schleife fehlt. Normalerweise gibt es Schleifen, die durch Daten laufen und dann Abfragen auf den Elementen vornehmen. Traditionelle Schleifen sind immer sequenziell und laufen von Element zu Element, und zwar vom Anfang bis zum Ende. Das Gleiche gilt auch für einen Iterator. Die Stream-API verfolgt einen anderen Ansatz, hier wird die so genannte externe Iteration (durch Schleifen vom Entwickler gesteuert) durch eine interne Iteration (die Stream-API holt sich Daten) abgelöst. Wenn etwa `forEach(…)` nach Daten fragt, wird die Datenquelle abgezapft und ausgesaugt, aber erst dann. Der Vorteil ist, dass wir zwar bestimmen, welche Datenstruktur abgelaufen werden soll, aber wie das intern geschieht, kann die Implementierung selbst bestimmen und dahingehend optimieren.

Wenn wir selbst die Schleife schreiben, läuft die Verarbeitung immer Element für Element, während die interne Iteration auch von sich aus parallelisieren und Teilprobleme von mehreren Ausführungseinheiten berechnen lassen kann.

Hinweis

An verschiedenen Sammlungen hängt eine `forEach(…)`-Methode, die über alle Elemente läuft und sie verarbeitet. Das heißt jetzt nicht, dass die klassische `for`-Schleife – etwa über das erweiterte `for` – damit überflüssig wird. Neben der einfachen Schreibweise und dem einfachen Debuggen hat die übliche Schleife immer noch einige Vorteile. `forEach(…)` bekommt den auszuführenden Code in der Regel über einen Lambda-Ausdruck, und der hat Einschränkungen. So darf er etwa keine lokalen Variablen beschreiben (alle vom Lambda-Ausdruck adressierten lokalen Variablen sind effektiv final), und Lambda-Ausdrücke dürfen keine geprüften Ausnahmen auslösen – im Inneren einer Schleife ist das alles kein Thema. Im Übrigen gibt es für Schleifenabbrüche das `break`, das in Lambda-Ausdrücken nicht existiert (ein `return` im Lambda entspricht `continue`).

Was ist ein Stream?

Ein Strom ist eine Sequenz von Daten, aber keine Datenquelle an sich, die Daten wie eine Datenstruktur speichert. Die Daten vom Strom werden in einer Kette von nachgeschalteten Verarbeitungsschritten

- gefiltert (engl. *filter*),
- transformiert/abgebildet (engl. *map*) und
- komprimiert/reduziert (engl. *reduce*).

Die Verarbeitung entlang einer Kette nennt sich *Pipeline* und besteht aus drei Komponenten:

- Am Anfang steht eine Datenquelle, wie etwa ein Array, eine Datenstruktur oder ein Generator.
- Es folgen diverse Verarbeitungsschritte wie Filterungen oder Abbildungen; diese Veränderungen auf dem Weg nennen sich *intermediäre Operationen* (engl. *intermediate operations*). Ergebnis einer intermediären Operation ist wieder ein Stream.
- Am Schluss wird das Ergebnis eingesammelt, das Ergebnis ist kein Stream mehr.

Die eigentliche Datenstruktur wird nicht verändert, vielmehr steht am Ende der intermediären Operationen eine terminale Operation, die das Ergebnis erfragt. So eine terminale Operation ist etwa `forEach(…)`; sie steht am Ende der Kette, und der Strom bricht ab.

Alle intermediären Operationen sind »faul« (engl. *lazy*), weil sie die Berechnungen so lange hinausschieben, bis sie benötigt werden. Im Gegensatz zu den fortführenden Operationen stehen terminale Operationen, bei denen das Ergebnis vorliegen muss: Sie sind »begierig« (engl. *eager*). Im Prinzip wird alles so lange aufgeschoben, bis ein Wert gebraucht wird, das heißt, bis eine terminale Operation auf das Ergebnis wirklich zugreifen möchte.

Intermediäre Operationen können einen Zustand haben oder nicht. Eine Filteroperation zum Beispiel hat keinen Zustand, weil sie zur Erfüllung ihrer Aufgabe nur das aktuelle Element be-

trachten muss, nicht aber vorangehende. Eine Sortierungsoperation hat dagegen einen Status, sie »will«, dass alle anderen Elemente gespeichert werden, denn das aktuelle Element reicht für die Sortierung nicht aus, sondern auch alle vorangehenden werden benötigt.

Viele terminale Operationen reduzieren die durchlaufenden Daten auf einen Wert, anders als etwa `forEach(…)`. Dazu gehören etwa Methoden zum einfachen Zählen der Elemente oder zum Summieren; das nennen wir *reduzierende Operationen*. In der API gibt es für Standardreduktionen – wie Bildung der Summe, des Maximums, des Durchschnitts – vorgefertigte Methoden, doch sind allgemeine Reduktionen über eigene Funktionen möglich, etwa statt der Summe das Produkt.

4.11.1 Stream erzeugen

Das Paket `java.util.stream` deklariert diverse Typen rund um Streams. Im Mittelpunkt steht für Objektströme eine generisch deklarierte Schnittstelle `Stream`. Ein konkretes Exemplar wird immer von einer Datenquelle erzeugt, unter anderem stehen folgende Stream-Erzeuger zur Verfügung:

Typ	Rückgabe	Methode
`BufferedReader`	`Stream<String>`	`lines()`
`Collection`	`Stream<E>`	`stream()`
`Arrays`	`Stream<T>`	`stream(T[] array)`
	`Stream<T>`	`stream(T[] array, int start, int end)`
`Pattern`	`Stream<String>`	`splitAsStream(CharSequence input)`
`ZipFile`	`Stream<? extends ZipEntry>`	`stream()`
`JarFile`	`Stream<JarEntry>`	`stream()`
`Stream`	`Stream<T>`	`empty()` (statisch)
	`Stream<T>`	`of(T... values)` (statisch)
	`Stream<T>`	`of(T value)` (statisch)
	`Stream<T>`	`generate(Supplier<T> s)`(statisch)
	`Stream<T>`	`iterate(T seed, UnaryOperator<T> f)`(statisch)

Tabelle 4.9 Methoden, die Stream-Exemplare liefern

`Stream` hat einige Fabrikmethoden für neue Ströme (etwa `of(…)`), alles andere sind Objektmethoden anderer Klassen, die Ströme liefern. Die `of(…)`-Methoden sind als statische Schnittstellen-

methoden von `Stream` implementiert, und `of(T... values)` ist nur eine Fassade für `Arrays.stream(values)`. Die Methode `Stream.empty()` liefert wie erwartet einen Strom ohne Elemente, `generate(…)` produziert Elemente aus einem `Supplier`, `iterate(…)` aus einem Startwert und einer Funktion, die die nächsten Elemente produziert, beide Ströme sind immer unendlich.

Beispiele

Produziere einen Stream aus gegebenen Ganzzahlen, entferne die Vorzeichen, sortiere und platziere das Ergebnis in eine neue Liste:

```
System.out.println(
  Stream.of( -4, 1, -2, 3 )
        .map( Math::abs ).sorted().collect( Collectors.toList() )
); // [1, 2, 3, 4]
```

Erzeuge ein Feld von zehn Zufallszahlen nach Normalverteilung:

```
Random random = new Random();
double[] randoms = Stream.generate( random::nextGaussian )
                         .limit( 10 ).mapToDouble( e -> e ).toArray();
System.out.println( Arrays.toString( randoms ) );
```

Produziere Zufallszahlen, wobei das Vorzeichen in der Belegung von `sign` das Vorzeichen der Zufallszahlen bestimmt:

```
double sign = +1.234;
Stream.iterate( sign, seed -> Math.signum( seed ) * Math.random() )
      .limit( 10 ).forEach( System.out::println );
```

Erzeuge einen Strom von Fibonacci-Zahlen[16]:

```
class FibSupplier implements Supplier<BigInteger> {
  private final Queue<BigInteger> fibs =
      new LinkedList<>( Arrays.asList( BigInteger.ZERO, BigInteger.ONE ) );
  @Override public BigInteger get() {
    fibs.offer( fibs.remove().add( fibs.peek() ) );
    return fibs.peek();
  }
};
Stream.generate( new FibSupplier() ).limit( 1000 ).forEach( System.out::println );
```

Die Implementierung nutzt keinen Lambda-Ausdruck, sondern eine altmodische Klassenimplementierung, da wir uns für die Fibonacci-Folgen die letzten beiden Elemente merken müssen. Sie speichert eine `LinkedList`, die als `Queue` genutzt wird. Bei der Anfrage an ein neues Element nehmen wir das erste Element heraus, sodass das zweite nachrutscht, und addieren es mit dem Kopfelement, was die Fibonacci-Zahl ergibt. Die hängen wir im zweiten Schritt hinten an und geben sie zurück.

16 *https://de.wikipedia.org/wiki/Fibonacci-Folge*

Parallele oder sequenzielle Streams

Ein Stream kann parallel oder sequenziell sein, das heißt, es ist möglich, dass Threads gewisse Operationen nebenläufig durchführen, wie zum Beispiel die Suche nach einem Element. `Stream` liefert über `isParallel()` die Rückgabe `true`, wenn ein Stream Operationen nebenläufig durchführt.

Alle `Collection`-Datenstrukturen können mit der Methode `parallelStream()` einen potenziell nebenläufigen Stream liefern.

Typ	Rückgabe	Methode
`Collection`	`Stream<E>`	`parallelStream()`
`Collection`	`Stream<E>`	`stream()`

Tabelle 4.10 Parallele und nichtparallele Streams von jeder Collection erfragen

Jeder parallele Stream lässt sich mithilfe der `Stream`-Methode `sequencial()` in einen sequenziellen Stream konvertieren. Parallele Streams nutzen intern das Fork-&-Join-Framework, doch sollte nicht automatisch jeder Stream parallel sein, da das nicht zwingend zu einem Performance-Vorteil führt; wenn zum Beispiel keine Parallelisierung möglich ist, bringt es wenig, Threads einzusetzen.

4.11.2 Terminale Operationen

Wir haben gesehen, dass sich Operationen in intermediär und terminal unterscheiden lassen. Die Schnittstelle `Stream` bietet insgesamt 18 terminale Operationen, und die Rückgaben der Methoden sind etwa `void` oder ein Array. Bei den intermediären Operationen sind die Rückgaben allesamt neue `Stream`-Exemplare.

Wir schauen uns nacheinander die terminalen Operationen an.

Anzahl Elemente

Die vielleicht einfachste terminale Operation ist `count()`: Sie liefert ein `long` mit der Anzahl der Elemente im Strom.

```
interface java.util.stream.Stream<T>
extends BaseStream<T,Stream<T>>
```

- `long count()`

Beispiele

Wie viele Elemente hat ein Feld von Referenzen, von `null` einmal abgesehen?

```
Object[] array = { null, 1, null, 2, 3 };
long size = Stream.of( array )
                  .filter( Objects::nonNull ).count();
System.out.println( size );  // 3
```

Und jetzt alle – forEachXXX(...)

Die Stream-API bietet zwei `forEachXXX(…)`-Methoden zum Ablaufen der Ergebnisse:

```
interface java.util.stream.Stream<T>
extends BaseStream<T,Stream<T>>
```

- `void forEach(Consumer<? super T> action)`
- `void forEachOrdered(Consumer<? super T> action)`

Übergeben wird immer ein Codeblock vom Typ `Consumer`:

```
Consumer<A> { void apply(A a); }
```

Die funktionale Schnittstelle `Consumer` hat einen Parameter, und `forEachXXX(…)` übergibt beim Ablaufen Element für Element an den `Consumer`.

Das Ablaufen mit `forEach(…)` ist erst einmal ohne Reihenfolge, es kommt darauf an, wie der Stream das realisiert. Einen Unterschied macht es schon, ob der Stream parallel ist oder nicht, und hier liegt der Unterschied zwischen `forEach(…)` und `forEachOrdered(…)` – Letzteres ist im Prinzip eine Abkürzung für `sequential().forEach(…)`.

Beispiel

Im ersten Fall ist die Ausgabe der Werte ungeordnet (cedab), im zweiten Fall geordnet (abcde).

```
Character[] chars = { 'a', 'b', 'c', 'd', 'e' };
Arrays.asList( chars ).parallelStream().forEach( System.out::print );
System.out.println();
Arrays.asList( chars ).parallelStream().forEachOrdered( System.out::print );
```

Einzelne Elemente aus dem Strom holen

Produziert der Strom mehrere Daten, so erfragt `findFirst()` das erste Element im Strom, wohingegen `findAny()` irgendein Element vom Strom liefern kann. Letztere Methode ist insbesondere bei parallelen Operationen interessant, wobei es völlig offen ist, welches Element das ist. Beide

Methoden haben den Vorteil, dass sie so genannte abkürzende Operationen sind, das heißt, sie ersparen einem einiges an Arbeit.

Da ein Strom von Elementen leer sein kann, ist die Rückgabe der Methoden immer `Optional`:

```
interface java.util.stream.Stream<T>
extends BaseStream<T,Stream<T>>
```

- `Optional<T> findFirst()`
- `Optional<T> findAny()`

Beispiel

```
Predicate<Character> p = Character::isDigit;
System.out.println( Stream.of('9', 'a', '0').parallel().filter( p ).findFirst() );
System.out.println( Stream.of('9', 'a', '0').parallel().filter( p ).findAny() );
System.out.println( Stream.of('9', 'a', '0').filter( p ).findFirst() );
System.out.println( Stream.of('9', 'a', '0').filter( p ).findAny() );
```

Die Ausgabe könnte sein: Optional[9], Optional[0], Optional[9], Optional[9].

Hängen andere zustandsbehaftete Operationen dazwischen, ist eine Optimierung mit `findAny()` mitunter hinfällig. So wird `stream.sorted().findAny()` nicht die Sortierung umgehen können.

Existenz-Tests mit Prädikaten

Ob Elemente eines Stroms eine Bedingung erfüllen, zeigen drei Methoden:

```
interface java.util.stream.Stream<T>
extends BaseStream<T,Stream<T>>
```

- `boolean anyMatch(Predicate<? super T> predicate)`
- `boolean allMatch(Predicate<? super T> predicate)`
- `boolean noneMatch(Predicate<? super T> predicate)`

Die Bedingung wird immer als `Predicate` formuliert.

Beispiel

Teste in einem Stream mit zwei Strings, ob entweder alle, irgendein oder kein Element leer ist:

```
System.out.println( Stream.of("", "").allMatch( String::isEmpty ) );   // true
System.out.println( Stream.of("", "a").anyMatch( String::isEmpty ) );  // true
System.out.println( Stream.of("", "a").noneMatch( String::isEmpty ) ); // false
```

Strom reduzieren

Ein Strom kann aus beliebig vielen Elementen bestehen, die durch Reduktionsfunktionen auf einen Wert reduziert werden können. Ein bekanntes Beispiel ist die `Math.max(a, b)`-Methode, die zwei Werte auf das Maximum abbildet. Diese Maximum-/Minimum-Methoden gibt es auch im `Stream`:

```
interface java.util.stream.Stream<T>
extends BaseStream<T,Stream<T>>
```

- `Optional<T> min(Comparator<? super T> comparator)`
- `Optional<T> max(Comparator<? super T> comparator)`

Interessant ist, dass immer ein `Comparator` übergeben werden muss und die Methoden nie auf die natürliche Ordnung zählen. Hilfreich ist hier eine `Comparator`-Utility-Methode, die genau so einen natürlichen `Comparator` liefert. Die Methoden liefern ein `Optional.empty()`, wenn der Stream leer ist.

Beispiel

Was ist die größte Fließkommazahl im Stream?

```
System.out.println( Stream.of( 9, 3, 4, 11 ).max( Comparator.naturalOrder() ).get() );
```

Die Bestimmung des Minimums/Maximums sind nur zwei Beispiele einer Reduktion. Allgemein kann jedes Paar von Werten auf einen Wert reduziert werden. Das sieht im Prinzip für die Minimum-Funktion wie folgt aus:

Stream (4, 2, 3, 1)	Funktionsanwendung	Resultat (Minimum)
4	–	4
2	min(4, 2)	2
3	min(2, 3)	2
1	min(2, 1)	1

Tabelle 4.11 Minimum-Berechnung beim Stream durch Reduktion

Java greift für Reduktionen auf die funktionalen Schnittstellen `BiFunction<T,U,R>` bzw. `BinaryOperator<T>` zurück (`BinaryOperator<T>` ist durch Vererbung eine `BiFunction<T,T,T>`).

Der `Stream`-Typ deklariert drei `reduce(…)`-Methoden, die mit diesen funktionalen Schnittstellen arbeiten. Wichtig ist, dass diese Reduktionsfunktion assoziativ ist, also f(a, f(b, c)) = f(f(a, b), c) gilt, denn insbesondere bei nebenläufiger Verarbeitung können beliebige Paare gebildet und reduziert werden.

```
interface java.util.stream.Stream<T>
extends BaseStream<T,Stream<T>>
```

- `Optional<T> reduce(BinaryOperator<T> accumulator)`
  Reduziert alle Paare von Werten schrittweise mit dem `accumulator` auf einen Wert. Ist der `Stream` leer, liefert die Methode `Optional.empty()`. Gibt es nur ein Element, bildet das die Rückgabe.
- `T reduce(T identity, BinaryOperator<T> accumulator)`
  Reduziert die Werte, wobei das erste Element mit `identity` über den `accumulator` reduziert wird. Ist der Strom leer, bildet `identity` die Rückgabe.
- `<U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)`
  Die beiden anderen Methoden liefern als Ergebnistyp immer den Elementtyp `T` des Streams. Mit dieser Methode ist der Ergebnistyp nicht `T`, sondern `U`, da der `accumulator` die Typen `(U, T)` auf `U` abbildet. Der `combiner` ist *nur* dann nötig, wenn in der parallelen Verarbeitung zwei Ergebnisse zusammengelegt werden, sonst ist der `combiner` überflüssig. Da er aber nicht `null` sein darf, haben wir es mit dem unschönen Fall zu tun, irgendetwas übergeben zu müssen, auch wenn das bei der sequenziellen Verarbeitung unnötig ist.

Beispiel

Was ist das größte Element im Stream von positiven Zahlen?

```
System.out.println( Stream.of( 9, 3, 4, 11 ).reduce( 0, Math::max ) );  // 11
```

Anders als die `max(…)`-Methode vom `Stream` liefert `reduce(…)` direkt ein `Integer` und kein `Optional`, weil bei leerem Stream 0 – die Identität – zurückgegeben wird. Das ist problematisch, denn 0 ist nicht das größte Element eines leeren Streams.

Beispiel

Reduziere einen Strom von `Dimension`-Objekten auf die Summe der Flächen:

```
Dimension[] dims = { new Dimension( 10, 10 ), new Dimension( 100, 100 ) };
BiFunction<Integer, Dimension, Integer> accumulator =
    (area, dim) -> area + dim.height * dim.width;
BinaryOperator<Integer> combiner = Integer::sum;
System.out.println( Arrays.stream( dims ).reduce( 0, accumulator, combiner ) );
// 10100
```

Das Ergebnis einer Reduktion ist immer ein neuer Wert, wobei »Wert« natürlich alles sein kann, eine Zahl, ein String, ein Datum oder eine Datenstruktur, wenn es insbesondere darum geht, als Ergebnis zum Schluss alle Elemente in einer Datenstruktur zu haben.

Ergebnisse in einen Container schreiben, Teil 1: collect(…)

Während die `reduce(…)`-Methoden durch den Akkumulator immer neue Werte in jedem Verarbeitungsschritt produzieren, können die `Stream`-Methoden `collect(…)` etwas anders arbeiten. Vereinfacht gesagt erlauben sie es, Daten eines Streams in einen Container (Datenstruktur oder auch String) zu setzen und diesen Container zurückzugeben.

```
interface java.util.stream.Stream<T>
extends BaseStream<T,Stream<T>>
```

- `<R> R collect(Supplier<R> resultFactory, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)`
  Der `Supplier` baut den Container auf, und der `BiConsumer` bekommt zwei Argumente, den Container und das Element, und muss dann das Element dem Container hinzufügen. Der `combiner` ist wieder dafür verantwortlich, zwei Container zusammenzulegen, was aber nur bei Parallelverarbeitung nötig ist.

Beispiel

Vier Zahlen eines Streams sollen im `LinkedHashSet` landen:

```
LinkedHashSet<Integer> set =
    Stream.of( 2, 3, 1, 4 )
          .collect( LinkedHashSet::new, LinkedHashSet::add, LinkedHashSet::addAll );
System.out.println( set );  // [2, 3, 1, 4]
```

Zur allgemeinen Initialisierung eines `LinkedHashSet` ist das natürlich noch etwas zu viel Code, hier geht es einfacher, mit `Collection.addAll(…)` die Elemente hinzuzufügen.

Wie bei `reduce(…)` bekommt die `collect(…)`-Methode jedes Element, doch verbindet es dieses Element nicht zu einem Kombinat, sondern setzt es in Container. Daher sind auch die verwendeten Typen nicht `BinaryOperator`/`BiFunction` (bekommen etwas und liefern etwas zurück), sondern `BiConsumer` (bekommt etwas, liefert aber nicht zurück). Der Zustand sitzt damit im Container und nicht in den Zwischenverarbeitungsschritten.

Beispiel

Eine eigene Klasse macht diese Zustandshaltung nötig, wenn zum Beispiel bei einem Strom von Ganzzahlen am Ende die Frage steht, wie viele Zahlen positiv, negativ oder null waren. Der Kollektor sieht dann so aus:

```
class NegZeroPosCollector {
  public long neg, zero, pos;

  public void accept( int i ) {
    if ( i > 0 ) pos++; else if ( i < 0 ) neg++; else zero++;
```

```
  }

  public void combine( NegZeroPosCollector other ) {
    neg += other.neg; zero += other.zero; pos += other.pos;
  }
}
```

Bei der Nutzung muss der Kollektor aufgebaut und müssen Akkumulator und Kombinierer angegeben werden:

```
NegZeroPosCollector col = Stream.of( 1, -2, 4, 0, 4 )
                               .collect( NegZeroPosCollector::new,
                                         NegZeroPosCollector::accept,
                                         NegZeroPosCollector::combine );
System.out.printf( "-:%d, 0:%d, +:%d", col.neg, col.zero, col.pos ); // -:1, 0:1, +:3
```

Ergebnisse in einen Container schreiben, Teil 2: Collector und Collectors

Es gibt zwei Varianten der `collect(…)`-Methode, und die zweite ist:

```
interface java.util.stream.Stream<T>
extends BaseStream<T,Stream<T>>
```

- `<R> R collect(Collector<? super T,R> collector)`

Ein `Collector` fasst Supplier, Akkumulator und Kombinierer zusammen. Die statische `Collector.of(…)`-Methode baut einen `Collector` auf, den wir ebenso an `collect(Collector)` übergeben können – unser Beispiel von eben ist also mit folgendem identisch:

```
.collect( Collector.of( LinkedHashSet::new,
                        LinkedHashSet::add, LinkedHashSet::addAll) );
```

Vorteil vom `Collector` ist also, dass dieser drei Objekte zusammenfasst (genau genommen noch ein viertes, `Collector.Characteristics` kommt noch dazu). Die Schnittstelle deklariert zwei statische `of(…)`-Methoden und weitere Methoden, die jeweils die Funktionen vom `Collector` erfragen, also `accumulator()`, `characteristics()`, `combiner()`, `finisher()` und `supplier()`. Die `Collector.Characteristics` sind eine Aufzählung mit Eigenschaften so eines `Collector`, das sind Aufzählungen wie `CONCURRENT`, `IDENTITY_FINISH` und `UNORDERED`.

Dazu kommt eine Utility-Klasse `Collector`**s** mit knapp 40 statischen Methoden, die alle `Collector`-Exemplare liefern, etwa `Collectors.toSet()`, `toList()`, `toXXXMap(…)`, aber auch `joining(…)` zum Zusammenhängen von Zeichenketten und Methoden, die keine Sammlung liefern, sondern Werte wie `averagingLong(…)` und Methoden zum einfachen Bilden von Gruppen, was wir uns gleich anschauen werden.

[»]

Beispiel

Füge Ganzzahlen in einem Strom zu einer Zeichenkette zusammen:

```
String s = Stream.of( 192, 0, 0, 1 )
                 .map( Integer::toUnsignedString )
                 .collect( Collectors.joining(".") );
System.out.println( s );  // 192.0.0.1
```

Baue ein `Set<String>` mit zwei Startwerten auf:

```
Set<String> set = Stream.of( "a", "b" ).collect( Collectors.toSet() );
```

Baue eine `Map<Integer, String>` auf, die aus zwei Schlüssel-Wert-Paaren besteht:

```
Map<Integer,String> map =
  Stream.of( "1=one", "2=two" )
        .collect( Collectors.toMap( k -> Integer.parseInt(k.split("=")[0]),
                                    v -> v.split("=")[1] ) );
```

```
final class java.util.stream.Collectors
```

- `static <T> Collector<T,?,List<T>> toList()`
- `static <T> Collector<T,?,Set<T>> toSet()`

Die weiteren Methoden sollten Leser in der Javadoc studieren.

[+]

Hinweis

Ein `Collector` ist das einzig vernünftige Mittel, um die Stream-Elemente in einen Container zu transferieren. Von einem Seiteneffekt aus irgendeiner intermediären oder terminalen Operation sollten Entwickler absehen, da erstens der Vorteil der funktionalen Programmierung dahin ist (Operationen haben dort keine Seiteneffekte) und zweitens würde bei paralleler Verarbeitung bei nichtsynchronisierten Datenstrukturen Chaos ausbrechen. Folgendes ist also *keine* Alternative zum charmanten `Collectors.joining(".")`:

```
StringBuilder result = new StringBuilder();
Stream.of( 192, 0, 0, 1 )
      .map( Integer::toUnsignedString )
      .forEach( s -> result.append( result.length() == 0 ? "" : "." ).append( s ) );
System.out.print( result );
```

Wenn ein Lambda-Ausdruck Schreibzugriffe auf äußere Variablen macht, sollten die Alarmglocken schrillen. Das Gleiche gilt im Übrigen auch für Reduktionen. Zwar kann im Prinzip in einem `forEach(…)` alles zum Beispiel in einen threadsicheren Atomic-Datentyp geschrieben werden, aber wenn es auch funktional geht, dann richtig mit `reduce(…)`.

Ergebnisse in einen Container schreiben, Teil 3: Gruppierungen

Oftmals lassen sich Objekte dadurch in Gruppen einteilen, dass referenzierte Unterobjekte eine Kategorie bestimmen. Da das furchtbar abstrakt klingt, ein paar Beispiele:

- Konten haben einen Kontentyp, und gesucht ist eine Aufstellung aller Konten nach Kontotyp. Mit dem Kontotyp sind also N Konten verbunden, die alle den gleichen Kontotyp haben.
- Menschen haben einen Beruf, und gesucht ist eine Liste aller Menschen nach Berufen.
- Threads können in unterschiedlichen Zuständen (wartend, schlafend, ...) sein; gesucht ist eine Assoziation zwischen Zustand und Threads in diesem Zustand.

Um dies mit der Stream-API zu lösen, lässt sich `collect(…)` mit einem besonderen `Collector` nutzen, den die `groupingByXXX(…)`-Methoden von `Collectors` liefern. Wir wollen uns nur mit der einfachsten Variante beschäftigen.

Beispiel

Gib alle laufenden Threads aus:

```
Map<Thread.State, List<Thread>> map =
  Thread.getAllStackTraces().keySet().stream()
        .collect( Collectors.groupingBy( Thread::getState ) );
System.out.println( map.get( Thread.State.RUNNABLE ) );
```

Zunächst liefert `Thread.getAllStackTraces().keySet()` ein `Set<Threads>` mit allen aktiven Threads. Aus der Menge leiten wir einen Strom ab und nutzen einen Kollektor, der nach Thread-Status gruppiert. Das Ergebnis ist eine Assoziation zwischen `Thread.State` und einer `List<Threads>`.

Wenn es Methoden in der Java-API gibt, die Entwicklern Generics nur so um die Ohren hauen, dann sind es die gruppierenden `Collector`-Exemplare – in der Übersicht:

```
final class java.util.stream.Collectors
```

- `static <T,K> Collector<T,?,Map<K,List<T>>> groupingBy(Function<? super T,? extends K> classifier)`
- `static <T,K,A,D> Collector<T,?,Map<K,D>> groupingBy(Function<? super T,? extends K> classifier, Collector<? super T,A,D> downstream)`
- `static <T,K,D,A,M extends Map<K,D>> Collector<T,?,M> groupingBy(Function<? super T,? extends K> classifier, Supplier<M> mapFactory, Collector<? super T,A,D> downstream)`
- `static <T,K> Collector<T,?,ConcurrentMap<K,List<T>>> groupingByConcurrent(Function<? super T,? extends K> classifier)`
- `static <T,K,A,D> Collector<T,?,ConcurrentMap<K,D>> groupingByConcurrent(Function<? super T,? extends K> classifier, Collector<? super T,A,D> downstream)`

- `static <T,K,A,D,M extends ConcurrentMap<K,D>> Collector<T,?,M> groupingByConcurrent(Function<? super T,? extends K> classifier, Supplier<M> mapFactory, Collector<? super T,A,D> downstream)`

Die Funktion, die den Schlüssel für den Assoziativspeicher extrahiert, ist in der API der `classifier`. Er muss immer angegeben werden. Wir haben uns die einfachste Methode – die erste – an einem Beispiel angeschaut; sie liefert immer eine `Map` von Listen. Die anderen Methoden können auch eine `Map` mit anderen Datenstrukturen liefern (bzw. statt Listen einfache akkumulierte Werte), dafür ist der `downstream` gedacht. Auch können die konkreten `Map`-Implementierungen bestimmt werden (etwa `TreeMap`), das gibt die `mapFactory` an – sonst wählt `groupingBy(Function)` für uns eine `Map`-Klasse aus.

[»]

Beispiel

Sammle die Namen aller Threads nach Status ein:

```
Map<Thread.State, List<String>> map =
  Thread.getAllStackTraces().keySet().stream()
        .collect( Collectors.groupingBy(
             Thread::getState,
             Collectors.mapping( Thread::getName, Collectors.toList() ) ) );
System.out.println( map.get( Thread.State.RUNNABLE ) );
```

Baue einen Assoziativspeicher auf, der den Thread-Status nicht mit den Threads selbst verbindet, sondern einfach nur mit der Anzahl an Threads:

```
Map<Thread.State, Long> map =
  Thread.getAllStackTraces().keySet().stream()
         .collect( Collectors.groupingBy( Thread::getState,
                                          Collectors.counting() ) );
System.out.println( map.get( Thread.State.RUNNABLE ) );    // 3
```

Assoziiere jeden Thread-Status mit einem Durchschnittswert der Prioritäten derjenigen Threads mit demselben Status:

```
Map<Thread.State, Double> map =
  Thread.getAllStackTraces().keySet().stream()
      .collect( Collectors.groupingBy( Thread::getState,
                                       Collectors.averagingInt(Thread::getPriority) )
);
System.out.println( map.get( Thread.State.RUNNABLE ) );  // 6.33..
System.out.println( map.get( Thread.State.WAITING ) );   // 9.0
```

Stream-Elemente in Array oder Iterator übertragen

Kümmern wir uns abschließend um die letzten drei aggregierenden Methoden. Zwei `toArray(…)`-Methoden von `Stream` übertragen die Daten des Stroms in ein Array. `iterator()` auf dem

`Stream` wiederum liefert einen `Iterator` mit all den Daten (die Methode stammt aus dem Obertyp `BaseStream`).

```
interface java.util.stream.Stream<T>
extends BaseStream<T,Stream<T>>
```

- `Object[] toArray()`
- `<A> A[] toArray(IntFunction<A[]> generator)`
- `Iterator<T> iterator()`

Die zwei `toArray(…)`-Methoden sind nötig, da im ersten Fall unklar ist, was für ein Typ das Feld hat – es kommt standardmäßig nur `Object[]` heraus. Der `IntFunction` wird die Größe des Streams übergeben, und Ergebnis ist in der Regel ein Feld dieser Größe. Konstruktor-Referenzen kommen hier sehr schön zum Zuge. Primitive Felder können nie die Rückgabe bilden, hierfür müssen wir spezielle primitive Stream-Klassen nutzen.

Beispiel

Setze alle Elemente eines Stroms in ein String-Feld:

```
String[] strings = Stream.of( "der", "die", "das" ).toArray( String[]::new );
```

Es bleibt zu bemerken, dass `Stream` selbst nicht `Iterable` implementiert, sondern `iterator()` eine terminierende Operation ist, nachdem der Stream »leergesaugt« ist. Bei Klassen, die `Iterable` implementieren, muss ein Aufruf von `iterator()` beliebig oft möglich sein. Bei Streams ist das nicht gegeben, da die Streams selbst nicht für die Daten stehen wie eine `Collection`, die daher `Iterable` ist.

`Iterator` ist ein Datentyp, der häufig in der Rückgabe verwendet wird, seltener als Parametertyp. Mit `stream.iteraor()` ist es aber möglich, die Daten vom Stream genau an solchen Stellen zu übergeben. Einen `Stream` in einen `Iterator` zu konvertieren, um diesen dann mit `hasNext()`/`next()` abzulaufen, ist wenig sinnvoll, hierfür bietet sich genauso gut `forEarch(…)` auf dem `Stream` an.

Ist auf der anderen Seite ein `Iterator` gegeben, lässt sich dieser nicht direkt in einen Stream bringen. Iteratoren können wie `Streams` unendlich sein, und es gibt keinen Weg, die Daten eines Iterators als Quelle zu benutzen. Natürlich ist es möglich, den `Iterator` abzulaufen und daraus einen neuen Stream aufzubauen, dann muss der Iterator aber endlich sein.

Beispiel

Die Methode `ImageIO.getImageReadersBySuffix(String)` liefert einen `Iterator` von `ImageReader`-Objekten – sie sollen über einen Strom zugänglich sein:

```
Builder<ImageReader> builder = Stream.builder();
for ( Iterator<ImageReader> iter = ImageIO.getImageReadersBySuffix( "jpg" );
```

```
      iter.hasNext(); )
  builder.add( iter.next() );
Stream<ImageReader> stream = builder.build();
System.out.println( stream.count() );    // 1
```

Ist eine alte `Enumeration` gegeben, hilft `Collections.list(enumeration).stream()`, denn `list(…)` liefert eine `ArrayList` mit allen Einträgen; `list(Iterator)` gibt es da hingegen nicht.

4.11.3 Intermediäre Operationen

Alle terminalen Operationen führen zu keinem neuen veränderten Stream, sondern beenden den aktuellen Stream. Anders sieht das bei intermediären Operationen aus, sie verändern den Stream, indem sie etwa Elemente herausnehmen, abbilden auf andere Werte und Typen oder sortieren. Jede der intermediären Methoden liefert ein neues Stream-Objekt – das haben wir in den ersten Schreibweisen durch die Konkatenation etwas verschleiert, aber vollständig sieht es so aus:

Vollständige Schreibweise

```
Stream<Object> s1 =
 Stream.of(" ",'3',null,"2",1,"");
Stream<Object> s2 =
 s1.filter( Objects::nonNull );
Stream<String> s3 =
 s2.map( Objects::toString );
Stream<String> s4 = s3.map( String::trim );
Stream<String> s5 = s4.filter( s ->
 !s.isEmpty() );
Stream<Integer> s6 =
 s5.map( Integer::parseInt );
Stream<Integer> s7 = s6.sorted();
s7.forEach( System.out::println );
```

Kaskadierte Schreibweise

```
Stream.of(" ",'3',null,"2",1,"")
      .filter( Objects::nonNull )
      .map( Objects::toString )
      .map( String::trim )
      .filter( s -> ! s.isEmpty() )
      .map( Integer::parseInt )
      .sorted()
      .forEach( System.out::println );
```

Tabelle 4.12 Ausführliche und kompakte Schreibweise im Vergleich

[+]

Hinweis

Keine Strommethode darf die Stromquelle modifizieren, da das Ergebnis sonst unbestimmt ist. Veränderungen treten nur entlang der Kette von einem Strom zum nächsten auf. Wir haben das schon im Abschnitt »Ergebnisse in einen Container schreiben, Teil 2: Collector und Collectors« diskutiert.

Element-Vorschau

Eine einfache intermediäre Operation ist `peek(…)`; sie darf sich das aktuelle Element während des Durchlaufs anschauen.

```
interface java.util.stream.Stream<T>
extends BaseStream<T,Stream<T>>
```

- `Stream<T> peek(Consumer<? super T> action)`

Beispiel

Schaue in den Strom vor und nach einer Sortierung:

```
System.out.println( Stream.of( 9, 4, 3 )
                        .peek( System.out::println ) // 9 4 3
                        .sorted()
                        .peek( System.out::println ) // 3 4 9
                        .collect( Collectors.toList() ) );
```

Filtern von Elementen

Eine der wichtigsten Stream-Methoden ist `filter(…)`: Sie belässt alle Elemente im Stream, die einem Kriterium genügen, und löscht alle anderen, die nicht diesem Kriterium genügen.

```
interface java.util.stream.Stream<T>
extends BaseStream<T,Stream<T>>
```

- `Stream<T> filter(Predicate<? super T> predicate)`

Beispiel

Gib alle Wörter aus, bei denen zwei beliebige Vokale hintereinander vorkommen:

```
Stream.of( "Moor", "Aha", "Meister" )
      .filter( Pattern.compile( "[aeiou]{2}" ).asPredicate() )
      .forEach( System.out::println );      // Moor Meister
```

Statusbehaftete intermediäre Operationen

Die allermeisten intermediären Operationen können Elemente direkt während des Durchlaufs bewerten und verändern, sodass keine speicherintensive Zwischenspeicherung nötig ist. Einige intermediäre Operationen haben jedoch einen Status, dazu zählen `limit(long)` – begrenzt den Strom auf eine gewisse Anzahl maximaler Elemente –, `skip(long)` – überspringt eine Anzahl Elemente –, `distinct()` – löscht alle doppelten Elemente – und `sorted(…)` – sortiert den Strom.

```
interface java.util.stream.Stream<T>
extends BaseStream<T,Stream<T>>
```

- `Stream<T> limit(long maxSize)`
- `Stream<T> skip(long n)`
- `Stream<T> distinct()`
- `Stream<T> sorted()`
- `Stream<T> sorted(Comparator<? super T> comparator)`

Beispiel

Zerlege eine Zeichenkette in Teilzeichenketten, bilde diese auf das ersten Zeichen ab, und lösche aus dem Stream doppelte Elemente:

```
List<String> list = Pattern.compile( " " ).splitAsStream( "Pu Po Aha La" )
    .map( s -> s.substring(0, 1) )
    .peek( System.out::println ) // P P A L
    .distinct()                  // \/\/\/
    .peek( System.out::println ) //  P A L
    .collect( Collectors.toList() );
System.out.print( list );                       // [P, A, L]
```

`peek(…)` macht gut deutlich, wie die Elemente vor und nach dem Anwenden von `distinct()` aussehen.

Hinweis

Eine Methode wie `skip(…)` sieht auf den ersten Blick unschuldig aus, kann aber ganz schön auf den Speicherbedarf und die Performance gehen, wenn parallele Ströme mit Ordnung (etwa durch Sortierung) ins Spiel kommen. Bei einem `stream.parallel().sorted().skip(10000)` müssen trotzdem alle Elemente erst sortiert werden, damit die ersten 10.000 übersprungen werden können. Sequenzielle Ströme bzw. Ströme ohne Ordnung (die liefert die `Stream`-Methode `unordered()`) sind ungleich schneller, aber natürlich nicht in jedem Fall möglich.

Abbildungen

Die Abbildung von Elementen im Strom auf neue Elemente ist eines der mächtigsten Werkzeuge der Stream-API. Zu unterscheiden sind drei verschiedene Typen von Abbildungsmethoden:

- Die einfachste Variante ist `map(Function)`. Die Funktion bekommt das Stromelement als Argument und liefert ein Ergebnis, das dann in den Strom kommt. Hierbei kann sich der Typ ändern, wenn die Funktion nicht den gleichen Typ gibt, wie sie nimmt. Die Funktion wird nach und nach auf jedem Element angewendet, die Reihenfolge ergibt sich aus der Ordnung des Stroms.

- Die drei Methoden `mapTo[Int|Long|Double]([int|long|double]Function)` arbeiten wie `map(Function)`, nur liefern die Funktionen einen primitiven Wert, und das Ergebnis ist ein primitiver Stream – diese Typen schauen wir uns später an.
- `flatMapXXX(XXXFunction)`-Methoden ersetzen ebenfalls Elemente im Stream, mit dem Unterschied, dass die Funktionen alle selbst `Stream`-Objekte liefern, deren Elemente dann in den Ergebnisstrom gesetzt werden. Der Begriff »flat« (engl. für »flach«) kommt daher, dass die »inneren« Streams nicht selbst alle als Elemente in den Ursprungs-Stream kommen (sozusagen ein `Stream<Stream>`), sondern flachgeklopft werden. Anwendungsbeispiel sind in der Regel Szenarien wie »N Spieler assoziieren M Gegenstände, gesucht ist ein Strom aller Gegenstände aller Spieler«. Die Funktion kann `null` liefern, wenn wird nichts in den Stream gelegt haben.

```
interface java.util.stream.Stream<T>
extends BaseStream<T,Stream<T>>
```

- `<R> Stream<R> map(Function<? super T,? extends R> mapper)`
- `IntStream mapToInt(ToIntFunction<? super T> mapper)`
- `LongStream mapToLong(ToLongFunction<? super T> mapper)`
- `DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper)`
- `<R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper)`
- `IntStream flatMapToInt(Function<? super T,? extends IntStream> mapper)`
- `LongStream flatMapToLong(Function<? super T,? extends LongStream> mapper)`
- `DoubleStream flatMapToDouble(Function<? super T,? extends DoubleStream> mapper)`

Beispiel

Die Methode `Locale.getAvailableLocales()` liefert ein Feld von `Locale`-Objekten, die ein System unterstützt. Wir interessieren uns für alle Ländercodes:

```
Stream.of( Locale.getAvailableLocales() )
      .map( Locale::getCountry )
      .distinct()
      .forEach( System.out::println );
```

Über ein `Locale`-Objekt erfragt `DateFormatSymbols.getInstance(locale).getWeekdays()` die Namen der Wochentage. Wir interessieren uns als Ergebnis für alle Wochentage aller installierten Gebiete:

```
Stream.of( Locale.getAvailableLocales() )
      .flatMap( l -> Stream.of( DateFormatSymbols.getInstance( l ).getWeekdays() ) )
      .filter( s -> ! s.isEmpty() )
      .distinct()
      .forEach( System.out::println );
```

Tipp

Die an `flatMap(…)` übergebene Funktion muss als Ergebnis einen Stream liefern oder `null`, wenn nichts passieren soll. Es ist nicht verkehrt, immer einen Stream zu liefern und statt `null` der Funktion einen leeren Stream mit `Stream.empty()` zurückgeben zu lassen. Anwendungen für diese leeren Ströme kommen aus folgendem Szenario: In der API gibt es Methoden, die statt Feldern/Sammlungen nur `null` geben. Nehmen wir an, `result` ist so eine Rückgabe, die `null` oder ein Feld sein kann, dann bekommen wir einen `Stream` wie folgt:

```
Stream<…> flatStream = Optional.ofNullable( result )
                              .map( Stream::of ).orElse( Stream.empty() );
```

Die Fallunterscheidung, ob die Sammlung `null` ist, ist hier funktional mit `Optional` gelöst, alternativ `result != null ? result : Stream.empty()`. Das statische `Stream.of(…)`, was wir hier über eine Methodenreferenz nutzen, funktioniert für ein Feld, für einen Stream aus einer Sammlung wäre `result.stream()` nötig.

Fassen wir das in einem Beispiel zusammen. Die Methode `getEnumConstants()` auf ein `Class`-Objekt liefert ein Feld von Konstanten, wenn das `Class`-Objekt eine Aufzählung repräsentiert – andernfalls ist das Feld nicht etwa leer, sondern `null` (so lässt sich unterscheiden, ob das `Class`-Objekt entweder keine Elemente hat oder überhaupt kein Aufzählungstyp ist). Wir wollen von drei `Class`-Objekten alle Konstanten einsammeln und ausgeben:

```
Stream.of( Object.class, Thread.State.class, DayOfWeek.class )
      .flatMap( clazz -> Optional.ofNullable( clazz.getEnumConstants() )
                                 .map( Stream::of ).orElse( Stream.empty() ) )
      .forEach( System.out::println );
```

4.11.4 Streams mit primitiven Werten

Datenstrukturen der Collection-API referenzieren immer nur Objekte. Wenn primitive Werte wie ein `int` oder `boolean` in zum Beispiel einer `ArrayList` oder einem `HashSet` gespeichert werden sollen, so werden die Werte immer nur in einem Wrapper abgelegt, der alle primitiven Werte verpackt. Nun gibt es aber neben Datenstrukturen der Collection-API auch echte Arrays, und es macht einen Unterschied, ob ein Typ `int[]` oder `Integer[]` ist. Aber die Stream-API, so wie wir sie kennengelernt haben, arbeitet zunächst einmal auf Objekten, und das heißt bei Zahlen, auf Wrapper-Objekten. Aber da weiterhin Felder mit primitiven Werten vorkommen und es auch Erzeuger von einfachen Werten gibt (wie Ströme von Zufallszahlen), ist das Umpacken in Wrapper-Objekte nicht die beste Lösung. Daher bietet die Java-API einige besondere Stream-Klassen extra für primitive Datenströme, damit die bekannten Stream-Operationen wie Filtern und Transformieren auch bei Arrays mit Primitiven möglich sind.

Primitive Streams aus Arrays

Die Klasse `java.util.Arrays` deklariert insgesamt folgende statische `stream(…)`-Methoden, die für Felder einen Stream generieren, wobei wir die ersten beiden Methoden schon kennen:

Rückgabe	Arrays-Methode
Stream<T>	stream(T[] array)
	stream(T[] array, int startInclusive, int endExclusive)
IntStream	stream(int[] array)
	stream(int[] array, int startInclusive, int endExclusive)
LongStream	stream(long[] array)
	stream(long[] array, int startInclusive, int endExclusive)
DoubleStream	stream(double[] array)
	stream(double[] array, int startInclusive, int endExclusive)

Tabelle 4.13 Statische stream(...)-Methoden in Arrays

Drei Dinge fallen auf:

- Von jeder Methode gibt es zwei Varianten: eine, die das ganze Feld betrachtet, und eine für einen Ausschnitt.
- Nicht für jeden primitiven Typ gibt es einen eigenen Stream-Typ, sondern nur für `int`, `long` und `double`, aber es fehlen `boolean`, `char`, `byte` und `short`. Die Konsequenz daraus ist, dass zum Beispiel bei primitiven Feldern vom Typ `byte` die komfortable Stream-API nicht verwendet werden kann.
- Für die primitiven Typen `int`, `long` und `double` gibt es jeweils drei eigene Stream-Klassen: `IntStream`, `LongStream` und `DoubleStream`. Wie wir gleich sehen werden, duplizieren sie im Wesentlichen die API von `Stream`, sind aber typisiert auf jeweils den primitiven Datentyp, den sie repräsentieren.

Da es nur `IntStream`, `LongStream` und `DoubleString` gibt, fehlen primitive Streams für `boolean`, `byte`, `short`, `char` und `float`. Damit lässt sich leben und ja immer noch Wrapper verwenden.

Tipp: IntStream vom String

Jede `CharSequence`, also `String`, `StringBuilder` usw., deklariert die Default-Methoden `chars()` und `codePoints()`, die einen `IntStream` liefern. Wichtig ist zu bedenken, dass der `IntStream` Ganzzahlen vom Typ `int` liefert, gewünscht ist aber in der Regel ein `char`, was eine Typanpassung nötig macht:

```
"Gentrifizierer".chars().forEach( System.out::print );                // 71101110…
"Gentrifizierer".chars().forEach( c -> System.out.print( (char) c ) ); // Gen…
```

Fabrikmethoden der XXXStream-Klassen

Die drei Klassen für primitive Streams bieten wie `Stream` direkte Möglichkeiten zum Aufbau. Dazu deklarieren die drei Klassen statische Methoden:

- `of(…)`: Die überlade Methode nimmt Werte direkt an.
- `builder()`: Bezieht ein `Builder`-Objekt, dem Primitive hinzugefügt werden können.
- `generate(supplier)`: Erzeugt einen endlosen Strom aus einem Datengeber.
- `iterate(seed, f)`: Erzeugt einen endlosen Strom aus einem Startwert und einer Funktionsanwendung.

Primitive Streams aus anderen Datenquellen

Während zum Beispiel `lines()` vom `BufferedReader` auf den normalen `Stream`-Typ zurückgreift und einen Strom vom Strings liefert, nutzen anderen Klasse wie `BitSet`, `Random` oder `ThreadLocalRandom` direkt die primitiven Stream-Typen:

Klasse	Rückgabe	Methode
`BitSet`	`IntStream`	`stream()`
`Random`	`IntStream`	`ints()`, `ints(long)`, `ints(int, int)`, `ints(long, int, int)`
	`LongStream`	`longs()`, `longs(long)`, `longs(long, long)`, `longs(long, long, long)`
	`DoubleStream`	`doubles()`, `doubles(long)`, `doubles(long, double, double)`, `doubles(double, double)`
`ThreadLocalRandom`	`IntStream`	`ints()`, `ints(long)`, `ints(int, int)`, `ints(long, int, int)`
	`LongStream`	`longs()`, `longs(long)`, `longs(long, long, long)`, `longs(long, long)`
	`DoubleStream`	`doubles()`, `doubles(long)`, `doubles(long, double, double)`, `doubles(double, double)`

Tabelle 4.14 Klassen außerhalb der Stream-API, die primitive Streams liefern

Elemente vom Stream in primitiven Stream konvertieren

Mit den Methoden `mapToXXX(ToXXXFunction)` und `flatMapToXXX(Function)` lässt sich ein `Stream` in einen primitiven Stream konvertieren. Übergeben wird eine Funktion, die einen Wert extrahiert – das kann eine einfache Typkonvertierung sein oder eine sonstige Berechnung.

Beispiel

Welche Gesamtlänge haben gegebene Strings?

```
System.out.println( Stream.of( "Sharp", "dressed", "man" )   // Stream<String>
                           .mapToInt( String::length )       // IntStream
                           .sum() );                         // 15
```

`sum()` ist eine Spezialmethode vom `IntStream` und summiert über alle Elemente.

Rückgabetyp	Stream-Methode
IntStream	mapToInt(ToIntFunction<? super T> mapper)
	flatMapToInt(Function<? super T,? extends IntStream> mapper)
LongStream	mapToLong(ToLongFunction<? super T> mapper)
	flatMapToLong(Function<? super T,? extends LongStream> mapper)
DoubleStream	mapToDouble(ToDoubleFunction<? super T> mapper)
	flatMapToDouble(Function<? super T,? extends DoubleStream> mapper)

Tabelle 4.15 Stream-Methoden, die primitive Streams geben

Beispiel

Zeichne auf der Konsole eine Sinus-Kurve:

```
DoubleStream.iterate( 3*Math.PI/2, d -> d + 0.1 )   // DoubleStream
            .limit( 100 )                           // DoubleStream
            .map( Math::sin )                       // DoubleStream
            .map( d -> d * 40 + 40 )                // DoubleStream
            .mapToInt( d -> (int) d )               // IntStream
            .mapToObj( i -> String.format( "%" + (i + 1) + "s", "*") ) // Stream
            .forEach( System.out::println );
```

Die Konstruktion mit `format(…)` erzeugt einen Leer-String einer gewissen Länge, um das Sternchen einzurücken. So liefert etwa `String.format("%2s", "*")` den String `" *"`; `2` gibt die Gesamtlänge an, also gibt es ein Leerzeichen im Ziel-String.

Konvertieren von primitivem Stream in Objekt-Stream

Produziert ein primitiver Stream Werte, können diese auf zwei Arten in einen regulären Stream gebracht werden. Die erste Möglichkeit ist mit boxed() zu arbeiten, was die primitiven Werte in Wrapper-Objekte bringt. Im nächsten Schritt kann dann etwa map(...) die Wrapper-Objekte in etwas anderes konvertieren.

Beispiel

Erzeuge einen String mit zwei Zufallszahlen, die durch einen Doppelpunkt getrennt sind:

```
String s = new Random().doubles().limit( 2 ).boxed()
                       .map( Objects::toString ).collect( Collectors.joining( ":" ) );
System.out.println(s);    // z.B. 0.749886795314456:0.9826385271336343
```

Geht es um eine Folge von `boxed()` und `map(…)`, gibt es eine Abkürzung in Form von `mapToObj(…)`-Methoden:

Primitive Stream-Klasse	Methode
`IntStream`	`<U> Stream<U> mapToObj(IntFunction<? extends U> mapper)`
`LongStream`	`<U> Stream<U> mapToObj(LongFunction<? extends U> mapper)`
`DoubleStream`	`<U> Stream<U> mapToObj(DoubleFunction<? extends U> mapper)`

Tabelle 4.16 Umwandeln eines primitiven Streams in einen regulären Stream

Beispiel

Alternative zum Erzeugen eines Strings mit zwei Zufallszahlen:

```
String s = new Random().doubles().limit( 2 )
                       .mapToObj( Objects::toString )
                       .collect( Collectors.joining( ":" ) );
```

API-Unterschiede von Stream zu [Int|Long|Double]Stream

Die Schnittstelle `Stream` ist mit einem generischen Typen `T` deklariert, und dieses `T` bestimmt den Typ vieler Übergaben und Rückgaben der API, so etwa beim Filtern: `Stream<T> filter(Predicate<? super T> predicate)` – ein Stream hat den Elementtyp `T`, und nach dem Filtern kommen Elemente vom Typ `T` wieder »heraus«. Bei den Streams für Primitive ist das genauso, doch bei einem `IntStream` ist die Filteroperation als `IntStream filter(IntPredicate predicate)` deklariert, damit der primitive Stream auch nach der Verarbeitung ein primitiver Stream bleibt. Für Prädikate, Konsumenten, Funktionen, Operatoren, optionale Rückgaben, Spliteratoren, Stream-Builder gibt es ebenfalls individuelle Typen (wie `IntPredicate`/`LongPredicate`/`DoublePredicate`) statt der generischen Typen `Predicate`, `Consumer`, `Optional` usw.

Methoden von Stream	Methoden von DoubleStream
boolean allMatch(**Predicate**<? super T> predicate)	boolean allMatch(**DoublePredicate** predicate)
boolean anyMatch(**Predicate**<? super T> predicate)	boolean anyMatch(**DoublePredicate** predicate)
	OptionalDouble average()
	Stream<Double> boxed()
static <T> **StreamBuilder**<T> builder()	static **DoubleStream.Builder** builder()
void close()	void close()
<R> R collect(Collector<? super T,R> collector)	
<R> R collect(Supplier<R> resultFactory, **BiConsumer**<R,? super T> accumulator, BiConsumer<R,R> combiner)	<R> R collect(Supplier<R> resultFactory, **ObjDoubleConsumer**<R> accumulator, BiConsumer<R,R> combiner)
static <T> **Stream**<T> concat(**Stream**<? extends T> a, **Stream**<? extends T> b)	static **DoubleStream** concat(**DoubleStream** a, DoubleStream b)
long count()	long count()
Stream<T> distinct()	**DoubleStream** distinct()
static <T> **Stream**<T> empty()	static **DoubleStream** empty()
Stream<T> filter(**Predicate**<? super T> predicate)	**DoubleStream** filter(**DoublePredicate** predicate)
Optional<T> findAny()	**OptionalDouble** findAny()
Optional<T> findFirst()	**OptionalDouble** findFirst()
<R> **Stream**<R> flatMap(**Function**<? super T,? extends Stream<? extends R>> mapper)	**DoubleStream** flatMap(**DoubleFunction**<? extends DoubleStream> mapper)
DoubleStream flatMapToDouble(Function<? super T,? extends DoubleStream> mapper)	
IntStream flatMapToInt(Function<? super T,? extends IntStream> mapper)	
LongStream flatMapToLong(Function<? super T,? extends LongStream> mapper)	

Tabelle 4.17 Vergleich von Stream und DoubleStream inklusive geerbter Methoden von BaseStream

Methoden von Stream	Methoden von DoubleStream
void forEach(**Consumer**<? super T> action)	void forEach(**DoubleConsumer** action)
void forEachOrdered(**Consumer**<? super T> action)	void forEachOrdered(**DoubleConsumer** action)
static <T> **Stream**<T> generate(Supplier<T> s)	static **DoubleStream** generate(DoubleSupplier s)
static <T> **Stream**<T> iterate(T seed, **UnaryOperator**<T> f)	static **DoubleStream** iterate(double seed, **DoubleUnaryOperator** f)
Iterator<T> iterator()	**PrimitiveIterator.OfDouble** iterator()
Stream<T> limit(long maxSize)	**DoubleStream** limit(long maxSize)
<R> **Stream**<R> map(**Function**<? super T,? extends R> mapper)	
DoubleStream mapToDouble(**ToDoubleFunction**<? super T> mapper)	**DoubleStream** map(**DoubleUnaryOperator** mapper)
IntStream mapToInt(**ToIntFunction**<? super T> mapper)	IntStream mapToInt(**DoubleToIntFunction** mapper)
LongStream mapToLong(**ToLongFunction**<? super T> mapper)	LongStream mapToLong(**DoubleToLongFunction** mapper)
	<U> Stream<U> mapToObj(DoubleFunction<? extends U> mapper)
Optional<T> max(Comparator<? super T> comparator)	**OptionalDouble** max()
Optional<T> min(Comparator<? super T> comparator)	**OptionalDouble** min()
boolean noneMatch(**Predicate**<? super T> predicate)	boolean noneMatch(**DoublePredicate** predicate)
Stream<T> onClose(Runnable closeHandler)	**DoubleStream** onClose(Runnable closeHandler)
static <T> **Stream**<T> of(**T**... values)	static **DoubleStream** of(**double**... values)
static <T> **Stream**<T> of(**T** t)	static **DoubleStream** of(**double** t)
S parallel()	**DoubleStream** parallel()

Tabelle 4.17 Vergleich von Stream und DoubleStream inklusive geerbter Methoden von BaseStream

Methoden von Stream	Methoden von DoubleStream
Stream<T> peek(**Consumer**<? super T> consumer)	**DoubleStream** peek(**DoubleConsumer** consumer)
Optional<T> reduce(**BinaryOperator**<T> accumulator)	**OptionalDouble** reduce(**DoubleBinaryOperator** op)
T reduce(**T** identity, **BinaryOperator**<T> accumulator)	**double** reduce(**double** identity, **DoubleBinaryOperator** op)
<U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)	
S sequential()	**DoubleStream** sequential()
Stream skip(long n)	**DoubleStream** skip(long n)
Stream<T> sorted()	**DoubleStream** sorted()
Stream<T> sorted(Comparator<? super T> comparator)	
Spliterator<T> spliterator()	**Spliterator.OfDouble** spliterator()
	double sum()
	DoubleSummaryStatistics summaryStatistics()
Object[] toArray()	**double**[] toArray()
<A> A[] toArray(IntFunction<A[]> generator)	
S unordered()	**DoubleStream** unordered()

Tabelle 4.17 Vergleich von Stream und DoubleStream inklusive geerbter Methoden von BaseStream

Neben den bekannten Methoden haben die primitiven Stream-Typen einige neue Spezialmethoden, unter ihnen:

- `average()` und `sum()`, die den arithmetischen Mittelwert und die Summe bilden, sowie die Methode `summaryStatistics()`, die Anzahl, Minimum, Maximum und Durchschnitt auf einmal ermittelt und in einem Objekt zugänglich macht
- `boxed()`, das einen Stream von Wrapper-Objekten generiert
- `range(…)`/`rangeClosed(…)`, die einen Stream für Von-bis-Bereiche generieren – die Methoden haben nur `IntStream` und `LongStream`, nicht aber `DoubleStream`

- `IntStream` hat `asLongStream()` sowie `asDoubleStream()` und `LongStream` ein `asDoubleStream()`. In `DoubleStream` gibt es keine `asXXXStream()`-Methoden.

Beispiel

Die Implementierung von `count()` ist:

Listing 4.38 java.util.stream.DoublePipeline.java, Ausschnitt

```
abstract class DoublePipeline<E_IN>
        extends AbstractPipeline<E_IN, Double, DoubleStream>
        implements DoubleStream {
  …
  public final long count() {
    return mapToObj( e -> null ).mapToInt( e -> 1 ).sum();
  }
  …
}
```

Bereiche

Mit der Stream-API lassen sich externe Iterationen durch interne ersetzen und auf diese Weise auch Dinge parallelisieren. Auch für klassische Zählschleifen, die von einem Anfangswert bis zu einem Endwert zählen, bieten `IntStream` und `LongStream` eine Lösung über `range(…)`/`rangeClosed(…)`-Methoden, denn sie liefern einen Stream vom Primitiven:

Schnittstelle	Statische Methode
`IntStream`	`range(int startInclusive, int endExclusive)` `rangeClosed(int startInclusive, int endInclusive)`
`LongStream`	`range(long startInclusive, long endExclusive)` `rangeClosed(long startInclusive, long endInclusive)`

Tabelle 4.18 Statische rangeXXX(...)-Methoden in den zwei primitiven Stream-Schnittstellen

Beispiel

Laufe alle Kommandozeilenargumente ab, und gib sie mit Position aus:

```
IntStream.range( 0, args.length ).forEach(
  i -> System.out.printf( "Argument an Postion %d ist %s%n", i, args[ i ] )
);
```

Würden wir mit dem normalen Stream über das Feld laufen, hätten wir keinen Index zur Verfügung.

Hinweis

Die Schrittweite ist immer 1, und `DoubleStream` deklariert überhaupt kein `range(…)`/`rangeClosed(…)`. Das ist an sich kein Problem, denn für die beiden Hilfsmethoden gibt es Alternativen über einen Generator und ein Limit. Beispiel: Zähle von 1.0 bis inklusive 10.0 in 0.5er Schritten:

```
DoubleStream.iterate( 1, v -> v + .5 ).limit( 19 ).forEach( System.out::println );
```

Natürlich ist das nicht besonders lesbar, da aus 19 nicht direkt der Endwert abzulesen ist.

Abbildungen

Mit einem `IntStream` kann mehr gemacht werden, als einfach nur ein Feld abzulaufen. Insbesondere `map(…)` und `flatMap(…)` geben kreative (nicht unbedingt lesbare und bevorzugte) Lösungen, einen Index auf ein Objekt zu übertragen.

Beispiel

Wir laufen ein Feld in Zweierschritten ab, erfragen mit dem Index das Feldelement, übertragen dieses auf ein `Pair` (Klasse aus JavaFX) und setzen diese am Schluss in ein `Properties`-Objekt:

```
String[] keyVal = { "name", "chris", "age", "40" };
Properties props =
  IntStream.iterate( 0, i -> i + 2 ).limit( keyVal.length / 2 )
           .mapToObj( i -> new Pair<String, String>(keyVal[i], keyVal[i+1]) )
           .collect( Properties::new,
                     (map, pair) -> map.setProperty( pair.getKey(), pair.getValue() ),
                     (a, b) -> {} /* Ignore in sequential stream */ );
System.out.println( props );  // {age=40, name=chris}
```

XXXSummaryStatistics

Jede der drei primitiven Stream-Schnittstellen deklariert eine Methode `summaryStatistics()`, die ein Objekt liefert, das fünf Arten von Zusammenfassungen kapselt: die Summe, das Minimum, das Maximum, den Durchschnitt und die Anzahl an Elementen eines Stroms von Primitiven.

Schnittstelle	summaryStatistics()-Methode und Rückgabe
`IntStream`	`IntSummaryStatistics summaryStatistics()`
`LongStream`	`LongSummaryStatistics summaryStatistics()`
`DoubleStream`	`DoubleSummaryStatistics summaryStatistics()`

Tabelle 4.19 summaryStatistics()-Methoden der drei Stream-Schnittstellen

Beispiel

Bestimme von 10.000 Zufallszahlen einige Statistiken:

```
DoubleSummaryStatistics summaryStatistics =
  ThreadLocalRandom.current().doubles().limit( 10000 ).summaryStatistics();
System.out.println( summaryStatistics );
```

Die Ausgabe sieht aus wie diese:

```
DoubleSummaryStatistics{count=10000, sum=5013,689626, min=0,000397, average=
0,501369, max=0,999774}
```

Alle drei Statistikklassen realisieren eine praktische `toString()`-Methode und liefern bis auf die fünf Zustände keine weiteren Informationen:

IntSummaryStatistics	LongSummaryStatistics	DoubleSummaryStatistics
`long getCount()`		
`int getMax()`	`long getMax()`	`double getMax()`
`int getMin()`	`long getMin()`	`double getMin()`
`long getSum()`		`double getSum()`
`double getAverage()`		

Tabelle 4.20 Statistikmethoden der drei XXXSummaryStatistics-Klassen

Die Klassen berücksichtigen also die Wertebereiche, wobei anzumerken ist, dass die Summation ohne Überlaufkontrolle realisiert wird.

Die Objekte sind nicht nur einfache Behälter, sondern auch Konsumenten, was sich in folgender Typbeziehung widerspiegelt:

- `class DoubleSummaryStatistics implements DoubleConsumer`
- `class IntSummaryStatistics implements IntConsumer`
- `class LongSummaryStatistics implements LongConsumer, IntConsumer`

`LongSummaryStatistics` nimmt eine Doppelrolle ein, indem es `LongConsumer` und `IntConsumer` gleichzeitig implementiert. Als `XXXSummaryStatistics`-Klassen haben sie also eine `accept(XXX)`-Methode sowie eine Methode `combine(XXXSummaryStatistics)`, die nützlich ist, wenn ein `XXXSummaryStatistics` in `collect(…)` eingesetzt wird, etwa so: `IntSummaryStatistics stats = intStream.collect(IntSummaryStatistics::new, IntSummaryStatistics::accept, IntSummaryStatistics::combine)`. Praktisch sind auch spezielle `Collectors`-Methoden `summarizingXXX(…)`, die uns über eine Funktion (hier ist sehr gut eine Methodenreferenz eingesetzt) ein `XXXSummaryStatistics` liefern.

Beispiel

Beziehe einen Stream aller aktiven Threads, und erfrage Statistikinformationen über die Prioritäten der Threads:

```
IntSummaryStatistics priorities = Thread.getAllStackTraces().keySet().stream()
    .collect( Collectors.summarizingInt( Thread::getPriority ) );
System.out.println( priorities ); // IntSummaryStatistics{count=5, sum=37, min=5,
    average=7,400000, max=10}
```

4.11.5 Stream-Beziehungen, AutoCloseable

`Stream` ist eine Schnittstelle, die mit einer Typvariablen deklariert ist, denn jeder `Stream` ist typisiert. Mit `Stream` nimmt die Typbeziehung jedoch nicht den Anfang, denn `Stream` erweitert eine andere Schnittstelle `BaseStream`:

- `interface BaseStream<T,S extends BaseStream<T,S>> extends AutoCloseable`

Die Typvariable `T` steht für den Typ der Elemente und `S` für den Stream-Typ, der `BaseStream` implementiert. `Stream` ist eine der Erweiterungen von `BaseStream`, aber mit den primitiven Stream gibt es insgesamt vier:

- `interface Stream<T> extends BaseStream<T,Stream<T>>`
- `interface IntStream extends BaseStream<Integer,IntStream>`
- `interface LongStream extends BaseStream<Long,LongStream>`
- `interface DoubleStream extends BaseStream<Double,DoubleStream>`

Als Entwickler kommen wir mit dem Typ `BaseStream` nicht direkt in Kontakt, und eigene Implementierungen sind unwahrscheinlich. In `BaseStream` sind jedoch einige Methoden deklariert, auf die wir in diesem Kapitel schon hingewiesen haben:

```
interface java.util.stream.BaseStream<T,S extends BaseStream<T,S>>
extends AutoCloseable
```

- `Iterator<T> iterator()`
- `Spliterator<T> spliterator()`
- `boolean isParallel()`
- `S sequential()`
- `S parallel()`
- `S unordered()`
- `S onClose(Runnable closeHandler)`
- `void close()`

Schließen von Streams

Da `BaseStream` die Schnittstelle `AutoCloseable` implementiert, hat jeder `Stream` ein `close()` und kann in einem `try` mit Ressourcen eingesetzt werden. Üblicherweise ist das nicht nötig, da ein Stream oftmals durch Datenstrukturen wie Arrays iteriert, doch gibt es im Bereich der Ein-/Ausgabe auch Stream-Lieferanten, bei denen das Schließen eine Rolle spielt, nämlich genau dann, wenn der Stream eine Ressource öffnet, die dann wieder geschlossen werden muss. War die Ressource schon geöffnet, und holt der Stream nur Daten ab, ist das etwas anderes. In Java 8 existiert in `BufferedReader` die praktische Methode `Stream<String> lines()`, hier muss der Stream nicht geschlossen werden, da `lines()` ja den `BufferedReader` (die eigentliche Ressource) nicht aufmacht, und folglich ist ein `close()`/`try` mit Ressourcen auf `lines()` nicht nötig.

Anders sieht es bei der `Files`-Methode `Stream<String> lines(Path path)` aus. Die Methode öffnet die Datei zum Auslesen, und dann muss die offene Datei nach dem Ablaufen der Zeilen wieder von uns geschlossen werden – das kann `lines(…)` nicht selbst machen. Dass die Methode statisch ist, gibt schon einen ersten Hinweis darauf, dass der offene Zustand hier nicht in der Klasse `Files` selbst liegt, sondern bei uns bzw. in der `lines(…)`, und es ist unsere Aufgabe `close()` bzw. `try` mit Ressourcen zu verwenden.

Hinweis

Das Schließen eines Streams wird durchgeleitet zu einem Schließen der darunterliegenden Ressource. Dafür ruft der Entwickler direkt `close()` auf dem Stream auf oder nutzt `try` mit Ressourcen. Doch auch implizit schließt die Stream-Implementierung Ressourcen, und zwar dann, wenn mit `flatMap(Function<…> mapper)` die Elemente eines anderen Streams in den Ergebnis-Stream zusammengelegt werden. In diesem Fall wird der Stream, der die `mapper`-Funktion liefert, von `flatMap(…)` selbst geschlossen. Das ist sinnvoll, denn der Entwickler bekommt den Stream der `mapper`-Funktion ja gar nicht zu Gesicht.

Interessant ist nun die Frage, wie `lines()` das `close()` auf dem Stream zum Schließen der Ressource verwendet. Hier kommt die `onClose(Runnable closeHandler)`-Methode jedes `Stream` ins Spiel. An den `Stream` wird ein Callback-Handler gesetzt, der immer dann aufgerufen wird, wenn auf dem Stream `close()` aufgerufen wird. Der `closeHandler` wird in `lines()` dann die Ressource schließen. Vereinfacht (ohne die umfangreiche Ausnahmebehandlung!) sieht das so aus:

```
public static Stream<String> lines(Path path, Charset cs) throws IOException {
  BufferedReader br = Files.newBufferedReader(path, cs);
  return br.lines()                      // Liefert Stream<Stream>
           .onClose( () -> br.close() ); // Liefert Stream<Stream> mit Close-Handler
}
```

Bisher gibt es in der Java-API nicht viele Implementierungen von `onClose()`. Was Ressourcen angeht, ist `Files` der einzige Nutzer, und neben `lines(…)` sind es noch die Methoden `walk(…)` und `list(…)`.

Hinweis

Von außen ist einem Stream nicht anzusehen, ob er Ressourcen hält, die geschlossen werden müssen. Entwickler sollten daher immer die API-Dokumentation konsultieren, was für einen Stream sie in der Hand halten. In der Entwicklung der Stream-API haben die Oracle-Entwickler immer wieder herumexperimentiert, und zwischendurch waren spezielle schließende Streams (Untertyp von `Stream`, `CloseableStream`), eine Schnittstelle `MayHoldCloseableResource` und eine Annotation `HoldsResource` im Gespräch, doch das ist alles Vergangenheit.

4.11.6 Stream-Builder

Streams werden in der Regel aus Arrays oder Datenstrukturen bezogen. Für eine beliebige Anzahl von Elementen bietet sich auch ein Stream-Builder an, damit nicht zuerst Elemente in einen Container gesetzt und von diesem Container dann ein Stream besorgt werden muss.

Hinweis

Sammle alle Eingaben von einem Dialog ein, sortiere sie, und gib sie dann aus:

```
StreamBuilder<String> streamBuilder = Stream.builder();
for ( String input; (input = JOptionPane.showInputDialog( "Eingabe" )) != null; )
  streamBuilder.add( input );
streamBuilder.build().sorted().forEach( System.out::println );
```

Es gibt insgesamt vier Typen zum Aufbauen von Strömen, alle realisiert als innere statische Schnittstellen:

- `java.util.stream.Stream.Builder`
- `java.util.stream.IntStream.Builder`
- `java.util.stream.LongStream.Builder`
- `java.util.stream.DoubleStream.Builder`

Die API der Schnittstellen ist – abgesehen von der Tatsache, dass `Stream` generisch ist und die primitiven Stream-Klassen nicht – ähnlich:

Stream.Builder<T> extends Consumer<T>	**IntStream.Builder extends IntConsumer**	**LongStream.Builder extends LongConsumer**	**DoubleStream.Builder extends DoubleConsumer**
`void accept(T t)`	`void accept(int t)`	`void accept(long t)`	`void accept(double t)`
`default Stream.Builder<T> add(T t)`	`default IntStream.Builder add(int t)`	`default LongStream.Builder add(long t)`	`default DoubleStream.Builder add(double t)`
`Stream<T> build()`	`IntStream build()`	`LongStream build()`	`DoubleStream build()`

Tabelle 4.21 Stream-Builder im Vergleich

Die `add(…)`-Methoden basieren auf `accept(…)` und geben die `this`-Referenz zurück, was das Kaskadieren der Aufrufe ermöglicht.

4.11.7 Spliterator

Neben dem `Iterator` zum Ablaufen von Datensammlungen deklariert Java 8 einen neuen Typ `Spliterator`. Der hat zwei Aufgaben, und zwar zum Ablaufen und Partitionieren, also zum Zerlegen, von Sammlungen. Das ist zentral bei paralleler Verarbeitung, denn kann eine Sammlung zum Beispiel von zwei Threads abgelaufen werden, so entscheidet der Spliterator, wo genau eine Sammlung in zwei (oder mehr) Teile zerlegt wird und wo welches Element verarbeitet wird.

Ein `Spliterator` hat eine Reihe von Eigenschaften, die Charakteristiken genannt werden. Für diese gibt es `static int`-Konstanten (dieses Mal keine Aufzählungen) in `Spliterator` wie `CONCURRENT`, `DISTINCT`, `IMMUTABLE`, `NONNULL`, `ORDERED`, `SIZED`, `SORTED`, `SUBSIZED`.

Beispiel

```
Spliterator<?> split = Stream.of( 1, 2 ).parallel().sorted().spliterator();
System.out.println( split.characteristics() );                            // 16468
System.out.println( split.hasCharacteristics( Spliterator.CONCURRENT ) ); // false
System.out.println( split.hasCharacteristics( Spliterator.ORDERED ) );    // true
System.out.println( split.hasCharacteristics( Spliterator.SIZED | Spliterator.SORTED )
 ); // true
```

Auch für primitive Typen deklariert die Java-Bibliothek neue Typen – sie sind innere Schnittstellen von `Spliterator`. Zusammenfassend: `Spliterator<T>`, `Spliterator.OfDouble`, `Spliterator.OfInt`, `Spliterator.OfLong`, `Spliterator.OfPrimitive<T,T_CONS,T_SPLITR extends Spliterator.OfPrimitive<T,T_CONS,T_SPLITR>>`.

Da ein `Spliterator` unabhängig von der Stream-API ist, sitzt er direkt in `java.util` und nicht in `java.util.stream`. Aber wirklich interessant ist er nur im Zusammenhang mit der Stream-API. Und viele Klassen aus der Java-API haben eine neue Methode `spliterator()` bekommen, denn das gibt jede `java.util.Collection` seit Java 8 vor. Zudem deklariert auch `java.util.Arrays` einige neue `spliterator(…)`-Methoden.

Zu `Spliterator` gibt es ebenso eine neue Utility-Klasse `java.util.Spliterators` mit fast 30 statischen Methoden, wobei die meisten einen `Spliterator.OfXXX` liefern und sinnvoll sind für eigene Implementierungen, die einen `Spliterator` liefern müssen; zudem sind hier Konvertierungsmethoden vom `Spliterator` zum `Iterator` zu finden. Auch befinden sich ein paar abstrakte innere Klassen für eigene Spliteratoren in `Spliterators`.

4.11.8 Klasse StreamSupport

Als letzte verbleibende Klasse der Stream-API bleibt `StreamSupport`. Sie ist eher nicht für Endanwender gedacht, sondern für Bibliotheksentwickler und besteht ausschließlich aus statischen