Java - Streams

Dozent: Prof. Dr. Michael Eichberg
Kontakt: michael.eichberg@dhbw.de, Raum 149B

Version: 2.0 [Themed]

Folien: https://delors.github.io/prog-adv-java-streams/folien.de.rst.html
https://delors.github.io/prog-adv-java-streams/folien.de.rst.html.pdf

Kontrollfragen: https://delors.github.io/prog-adv-java-streams/kontrollfragen.de.rst.html

Fehler melden: https://github.com/Delors/delors.github.io/issues

Quellen: Die Folien wurden insbesondere basierend auf der offiziellen Java Dokumentation (JavaDoc 24 und JEPs), sowie basierend auf der Scala (2.13 bzw. 3) Documentation erstellt. Die Folien bzgl. der konkreten *Java Streams API (Deepdive)* basieren auf Folien von Th. Letschert

KI Verwendung: Bei der Erstellung der Folien wurden KI Assistenten unterstützend eingesetzt. Dies erfolgte insbesondere, um effizient Grafiken zu generieren, die sich leicht in die Folien integrieren lassen, oder um sich Übersichtstabellen generieren zu lassen. Weiterhin wurde KI zur allgemeinen Qualitätssicherung eingesetzt. Alle Inhalte, die in diesem Rahmen von der KI (zusäztlich) vorgeschlagen wurden, wurden überprüft. Es wurde **keine** KI verwendet für den Aufbau, die Struktur und insbesondere die Auswahl der grundlegenden Inhalte.

1. Motivation

(Aktuelle) Herausforderungen

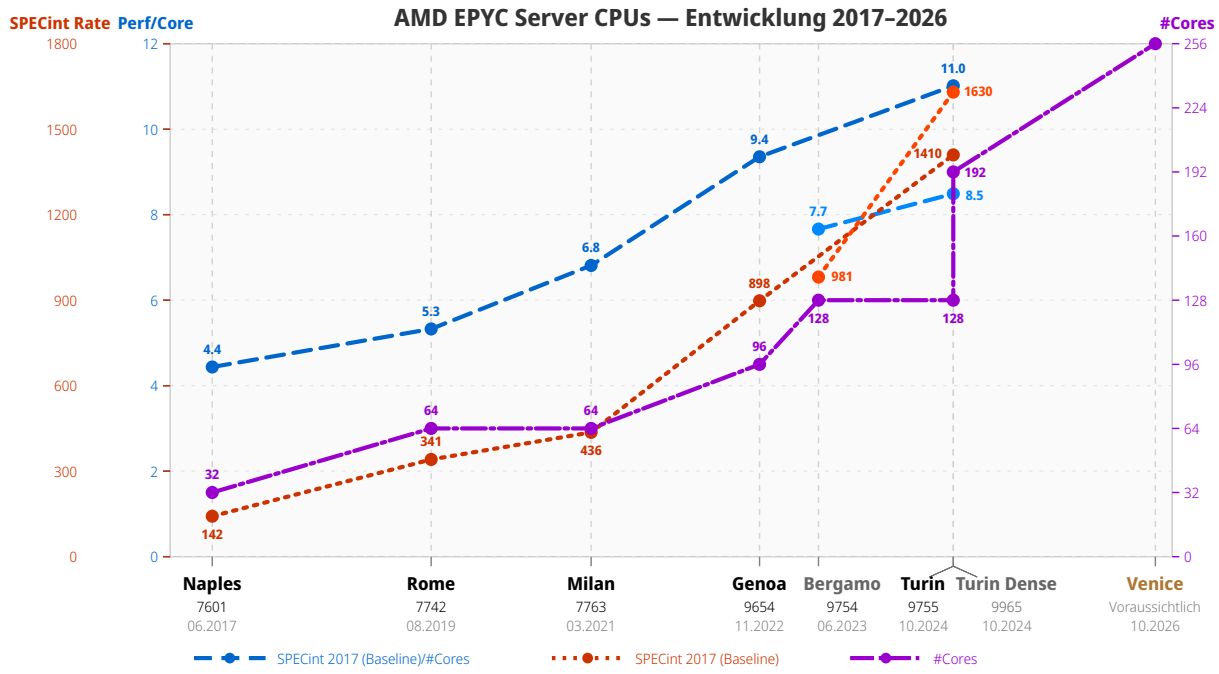
Bei der Datenverarbeitung von:

- großen Datenmengen,
  - die nicht (ohne weiteres) in den Speicher passen oder
  - bei denen es keinen Sinn macht diese vorab vollständig in den Speicher zu laden, da immer nur ein kleiner Abschnitt analysiert werden muss
- Daten, die kontinuierlich verarbeitet werden müssen

Beispiel: Konkrete Szenarien der Datenverarbeitung

- HTTP Request/Response Handling (Audio/Video Streaming)
- Ver-/Entschlüsselung von großen Datenmengen
- kontinuierliche/echtzeit Verarbeitung von allg. Daten (z. B. Finanzdaten/-transaktionen) oder (IoT) Sensoren
- Verarbeitung von Ereignissen (Event)
- Aufbereitung großer Wörterbücher (z.B. für Passwortwiederherstellung)
- Verarbeitung von AI Streaming Responses
- Verarbeitung große Logdateien

- Parallele Verarbeitung und effiziente Nutzung von modernen CPU-Architekturen



Insbesondere die korrekte und effiziente Nutzung mehrere Threads ist häufig schwierig. Die Verwendung von (soweit möglich) „transparent“ parallelisierten Streams ist dabei sehr hilfreich. (Z. B. mit Java Parallel Streams oder `.par` in Scala ist eine weitgehend transparente Parallelisierung möglich.)

- Entwicklung von Software nahe am Domänenmodell, um die korrekte Implementierung zu unterstützen.

Bemerkung

Das Ziel sollte (immer) eine möglichst geringe Repräsentationslücke (*Low representational gap*) zwischen Code und Domänenmodell sein.

Beispiel: Finden der besten Studierenden

Studierende Auswählen - Beispiel

Eingabe: (Flache) Liste von allen Studierenden

Ausgabe: (Flache) Liste von förderungswürdigen Studierenden

Logik: *„Nimm alle Studierenden, filtere diejenigen heraus, die bereits ein Stipendium haben, gruppiere die verbleibenden nach Studiengang, wähle pro Studiengang den mit der besten Durchschnittsnote, und erstelle daraus eine nach Note sortierte Empfehlungsliste."*

Setup

```java
record Student(boolean hatStipendium, String studiengang, double schnitt){};

List<Student> alleStudierenden =
    List.of(new Student(false, "Informatik", 1.3),  ...)
```

Klassische Implementierung der Geschäftslogik

```java
// Schritt 1: Studierende ohne Stipendium sammeln
List<Student> ohneStipendium = new ArrayList<>();
for (Student s : alleStudierenden) {
    if (!s.hatStipendium()) {
        ohneStipendium.add(s);
    }
}
// Schritt 2: Nach Studiengang gruppieren
Map<String, List<Student>> nachStudiengang = new HashMap<>();
for (Student s : ohneStipendium) {
    nachStudiengang
        .computeIfAbsent(s.studiengang(), k -> new ArrayList<>())
        .add(s);
}
// Schritt 3: Pro Gruppe den Besten finden
List<Student> empfehlungen = new ArrayList<>();
for (var eintrag : nachStudiengang.entrySet()) {
    Student bester = null;
    for (Student s : eintrag.getValue()) {
        if (bester == null || s.schnitt() < bester.schnitt()) {
            bester = s;
        }
    }
    if (bester != null) {
        empfehlungen.add(bester);
    }
}
```

2. Zentrale Konzepte von (Java) Streams

> ⚠ Achtung!
>
> Im Folgenden betrachten wir die **Java Stream API** und nicht **Java I/O Streams**.

	Java I/O Streams	Java Stream API
Package	`java.io.*` / `java.nio.*`	`java.util.stream.*`
Fokus	Byte-/Zeichentransport	Datenverarbeitung
Paradigma	Imperativ	Funktional/Deklarativ

Imperativ Programmierung:

Das Vorgehen wird detailliert durch konkrete Anweisungen beschrieben, die genau vorgeben welche einzelnen Schritte von dem Computer ausgeführt werden sollen, um das Ziel zu erreichen.
Viele gängige *general-purpose* Programmiersprachen (C, C++, Rust, Java, Go, JavaScript, Python) sind im Kern imperative Programmiersprachen.

Deklarative Programmierung:

Das Ziel ist es auszudrücken *was* erreicht werden soll, ohne das *wie* genau anzugeben.
(Prominentes Beispiel für eine deklarative Programmiersprache: SQL als Datenbankabfragesprache.)

> ◆ Bemerkung
>
> - Auch für gängige Programmiersprachen, die im Kern imperativ sind, ist zu beobachten, dass mehr und mehr Ideen und Konzepte aus der deklarativen Programmierung Einzug in diese Sprachen - insbesondere auch über APIs - finden.
> - Es ist in der Zwischenzeit so, dass die Grenzen zwischen (klassischen) prozeduralen (d. h. primär imperativen) Programmiersprachen und funktionalen sowie deklarativen Programmiersprachen verschwimmen.
> - Scala - als ein Beispiel für eine moderne Programmiersprache - ist von Grund auf als objektorientiert-funktionale Sprache entworfen wurde.

Streams - Einführung am Beispiel

Pragmatische Sicht:

Streams[1] sind umgeformte Sammlungen, die durch die Umformung für funktional-orientierte Massen-Operationen geeignet sind.

Konzeptionelle Sicht:

Streams erlauben die „korrekte, effiziente, lesbare (domänennahe)“ Verarbeitung von Daten mit Hilfe von Konzepten und Ideen aus der funktionalen und deklarativen Programmierung.

Beispiel

Benötigte Imports (nicht in der JShell)

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
```

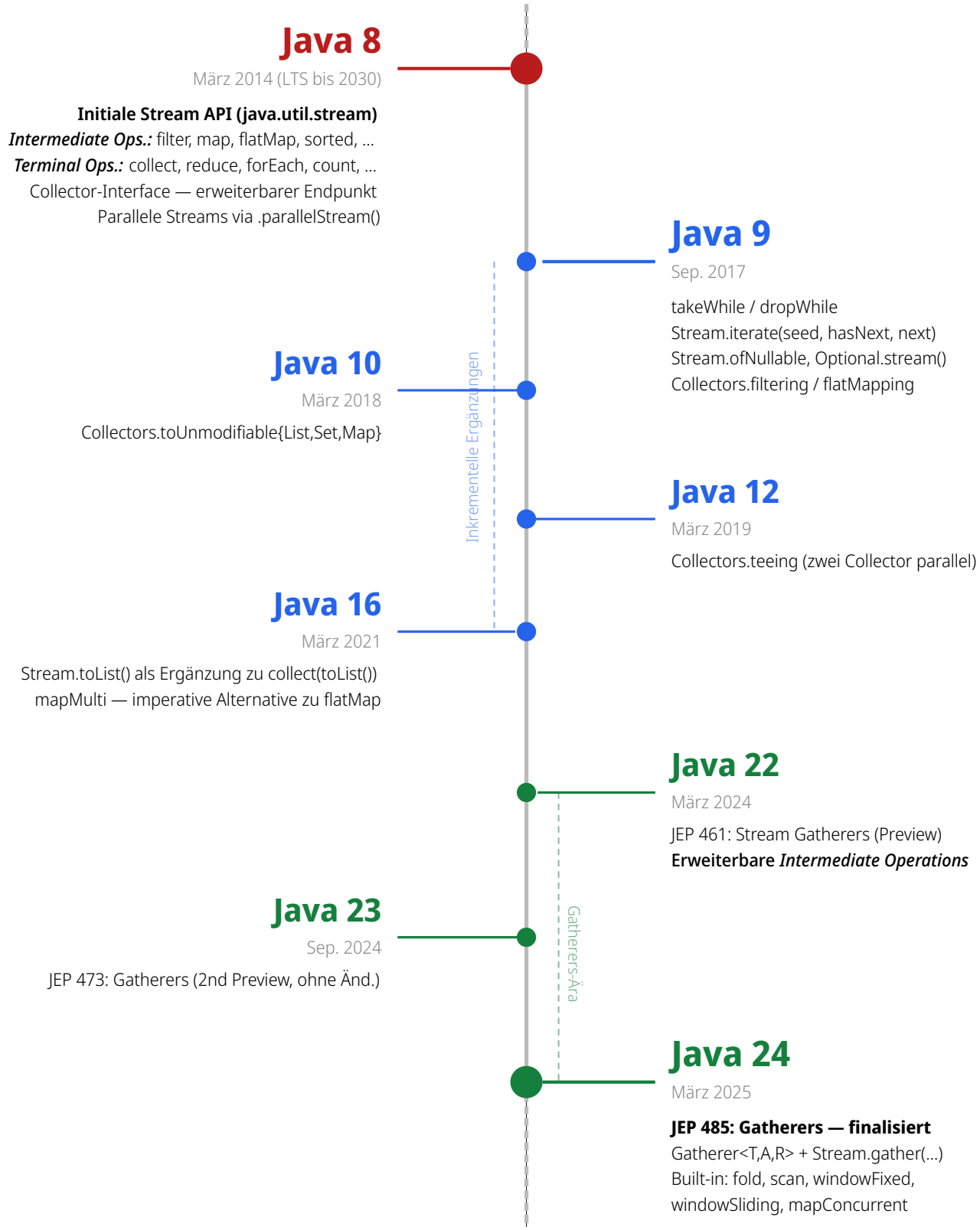
```
1 List<Integer> l  = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);
2 List<Integer> ll = l
3     .stream()                      // list → stream
4     .filter(x → x % 2 == 0)        // filter list with predicate
5     .map(x → 10 * x)               // map each element to a new one
6     .collect(Collectors.toList()); // back to a list (alt. toList())
7 ll.forEach(x → System.out.println(x));
```

Ausgabe: `20 40 60 80`

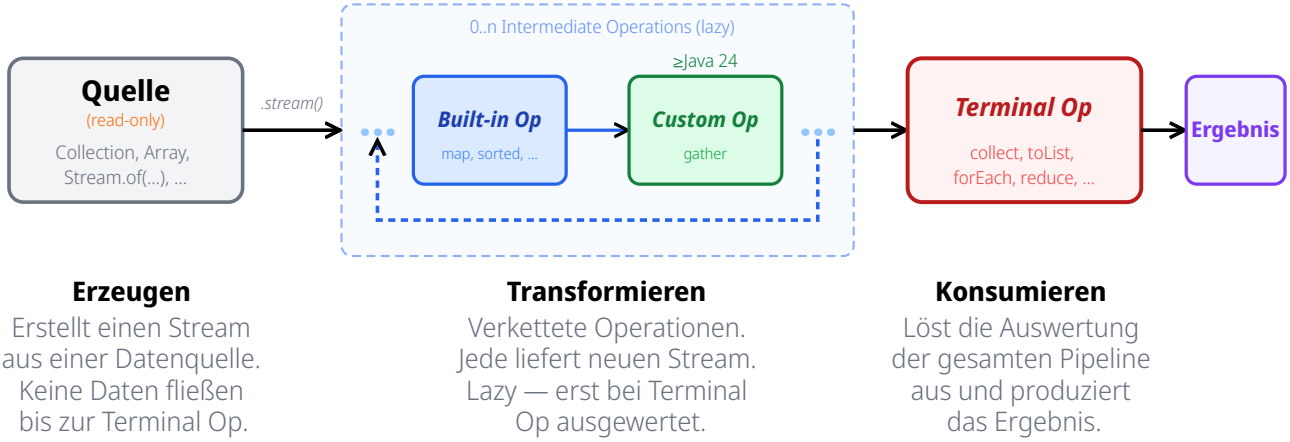
---

[1] Th. Letschert

Streams API - Evolution



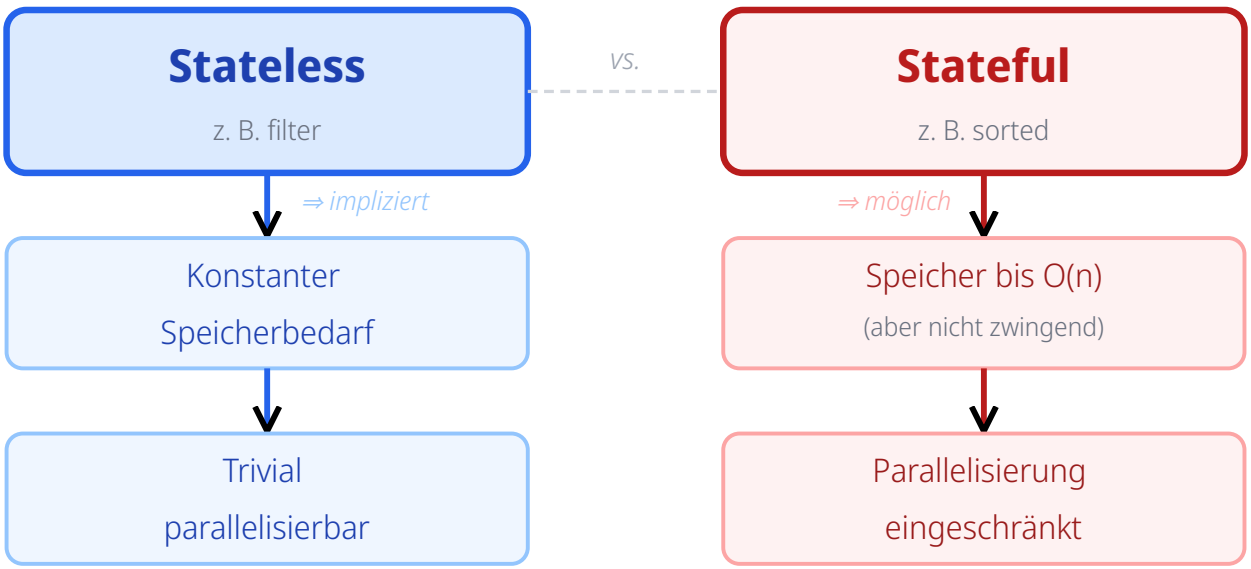
Streams - Grundlegendes Konzept

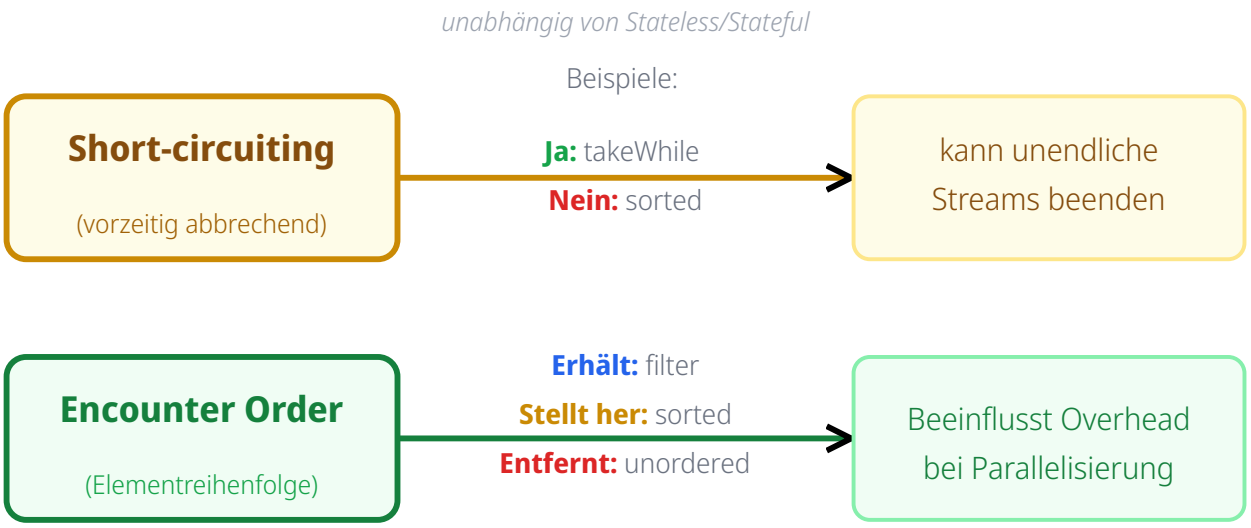


Es ist möglich als terminale Operation einen `Iterator` bzw. `Spliterator` zu erzeugen. In diesen beiden Fällen erfolgt die Evaluation der Pipeline nicht vollständig (d.h. nicht *eager* sondern *lazy*). Es wird somit nur so viel ausgewertet, wie für die Erzeugung des `Iterator`/`Spliterator` notwendig ist. Das hat zur Folge, dass die Elemente des Streams erst dann verarbeitet werden, wenn sie tatsächlich über den `Iterator`/`Spliterator` angefordert werden.

Die Verwendung dieser beiden Methoden führt zu einem Bruch des deklarativen Pipeline-Modells, der in den allermeisten Fällen vermieden werden kann. Häufig nur noch im Zusammenhang mit Legacy-APIs relevant, die mit `Iterator`/`Spliterator` arbeiten.

Eigenschaften von *Intermediate Operations*





Stateless Ops(🏴 *zustandslose Operationen*):
Transformieren die Elemente jeweils völlig unabhängig von allen anderen.

Stateful Ops(🏴 *zustandsbehaftete Operationen*):
Transformieren die Elemente abhängig von anderen.

Überblick über die wichtigsten Operationen und Ihre Eigenschaften

Operation	Stateless / Stateful	Speicherbedarf	Parallelisierbar	Short-circuiting
`filter`	Stateless	Konstant	Trivial	Nein
`map`	Stateless	Konstant	Trivial	Nein
`flatMap`	Stateless	Konstant	Trivial	Nein
`mapMulti`	Stateless	Konstant	Trivial	Nein
`peek`	Stateless	Konstant	Trivial	Nein
`sorted`	Stateful	O(n)	Eingeschränkt	Nein
`distinct`	Stateful	O(n)	Eingeschränkt	Nein
`limit`	Stateful	Konstant	Eingeschränkt	Ja
`skip`	Stateful	Konstant	Eingeschränkt	Nein
`takeWhile`	Stateful	Konstant	Eingeschränkt	Ja
`dropWhile`	Stateful	Konstant	Eingeschränkt	Nein
`gather`	Konfigurierbar	Konfigurierbar	Konfigurierbar	Konfigurierbar

Warnung

Seiteneffekte in Funktionen, die an Stream-Operationen übergeben werden, sind grundsätzlich zu vermeiden. Wird ein Stream parallelisiert und eine übergebene Funktion (z. B. an die `peek` Methode) hat dennoch Seiteneffekte, so muss diese Thread-sicher sein.

---

3. Java Streams API - Deep Dive

Streams mit primitiven Daten und Objekten

- `Stream<T>` ist der Typ der Streams mit Objekten vom Typ `T`
- Streams mit primitiven Daten:
  - `IntStream`
  - `LongStream`
  - `DoubleStream`

  Diese Streams mit primitiven Daten arbeiten in vielen Fällen effizienter jedoch sind manche Operationen nur auf `Object`-Streams erlaubt. „Primitive“ Streams können mit der Methode `boxed` in `Object`-Streams umgewandelt werden.

Beispiel

```
1 IntStream isPrim = IntStream.range(1, 10);
2 Stream<Integer> isObj = isPrim.boxed();
```

Erzeugung von Streams

Statische Methoden in `Arrays`

- Die Klasse `java.util.Arrays` hat mehrere überladene statische stream-Methoden, mit denen Arrays in Ströme umgewandelt werden können.
- Die Streams können Objekte oder primitive Daten enthalten.

Beispiel

```
// Stream of primitive data:
IntStream isP = Arrays.stream(new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 });
// Stream of objects:
Stream<Integer> isO = Arrays.stream(
    new Integer[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 }
);
```

Statische Methoden in `Stream`

- Das Interface `java.util.stream.Stream` enthält mehrere statische Methoden mit denen Streams erzeugt werden können.
- Für die Klassen der Streams mit primitiven Werten (z.B. `java.util.stream.IntStream`) gibt es äquivalente Methoden.
- Mit `of` werden die übergebenen Wert in einen Stream gepackt.
- Mit `iterate` und `generate` hat man eine einfache Möglichkeit unendliche Ströme zu erzeugen.

Beispiel

```
// Object-Stream 1, 2 ... 9, 0:
Stream<Integer> is1a = Stream.of(1,2,3,4,5,6,7,8,9,0);

// int-Stream 1, 2, ... 9, 0
IntStream is1b = IntStream.of(1,2,3,4,5,6,7,8,9,0);

// (infinite) Stream 1, 2, ...
Stream<Integer> is2 = Stream.iterate(1, ((x) -> x+1));

int[] z = new int[]{1};
Stream<Integer> is3 = Stream.generate(() -> z[0]++); // (infinite) Stream 1, 2, ...
```

Statische range-Methoden in `IntStream` und `LongStream`

Die Interfaces `java.util.stream.IntStream` und `java.util.stream.LongStream` enthalten jeweils zwei statische `range`-Methoden mit denen Streams erzeugt werden können.

Beispiel

```
IntStream isPrimA = IntStream.range(1, 10); // 1,2, .. 9
IntStream isPrimA = IntStream.rangeClosed(1, 10); // 1,2, .. 9, 10
```

Nicht-statische Methoden der Collection-API

Das Interface `java.util.Collection` enthält die Methode stream mit der die jeweilige Kollektion in einen Stream umgewandelt werden kann.

Beispiel

```
1 Stream<Integer> is = Arrays.asList(1,2,3,4,5,6,7,8,9,0).stream();
```

Verwendung von Streams

Zustandslose Verarbeitungsoperationen

- `map(Function<? super T,? extends R> mapper)`: Transformiert jedes Element in ein anderes.
- `filter(Predicate<? super T> predicate)`: Filtert Elemente heraus.
- `flatMap(Function<? super T,? extends Stream<? extends R>> mapper)`: Transformiert jedes Element in einen Stream und fügt die Streams zusammen.

Beispiel

```
1  import java.util.List;
2  import java.util.stream.Collectors;
3  import java.util.stream.IntStream;
4
5  List<Integer> is = IntStream.range(1, 10)
6      .filter(i -> i % 2 != 0)
7      .peek(i -> System.out.print(i+ " "))
8      .map(i -> 10 * i)
9      .boxed()
10     .collect(Collectors.toList());
11 System.out.println(is);
```

Ausgabe: `1 3 5 7 9 [10, 30, 50, 70, 90]`

Beispiel

```
1  import java.util.List;
2  import java.util.stream.Collectors;
3  import java.util.stream.IntStream;
4  import java.util.stream.Stream;
5
6  static Stream<Integer> range(int from, int to) {
7      return IntStream.range(from, to).boxed();
8  }
9
10 List<Integer> is = Stream.of(0, 1, 2)
11     .flatMap(i -> range(10 * i, 10 * i + 10))
12     .collect(Collectors.toList());
```

Ausgabe: `is ⟹ [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]`

Zustandsbehaftete Verarbeitungsoperationen

- `distinct()`: Entfernt Duplikate.
- `sorted()`: Sortiert die Elemente.
- `sorted(Comparator<? super T> comparator)`: Sortiert die Elemente mit einem gegebenen Comparator.
- `limit(long maxSize)`: Begrenzt die Anzahl der Elemente.
- `skip(long n)`: Überspringt die ersten n Elemente.

Beispiel

```
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

List<Integer> lst = Stream.of(9, 0, 3, 1, 7, 3, 4, 7, 2, 8, 5, 0, 6, 2)
    .distinct()
    .sorted((i, j) -> i - j)
    .skip(1)
    .limit(3)
    .collect(Collectors.toList());
```

Ausgabe: `is ⟹ [1, 2, 3]`

Verarbeitungsoperationen

Eine terminale Operation hat im Gegensatz zu den Verarbeitungsoperationen *keinen Stream* als Ergebnis.

Terminale Operationen ohne Ergebnis

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

  Wendet die übergebene Aktion auf alle Elemente des Streams an.

Terminale Operationen mit Ergebnis

- Operationen mit Array-Ergebnis: `Stream ⇒ Array`

  Operationen die den Stream in ein äquivalentes Array umwandeln.
- Operationen mit Kollektions-Ergebnis: `Stream ⇒ Kollektion`

  Operationen die den Stream in eine äquivalente Kollektion umwandeln.
- Operationen mit Einzel-Ergebnis: Aggregierende Operationen

  Operationen die den Stream zu einem einzigen Wert verarbeiten.

Beispiel

`forEach`

```
Stream.of(9, 0, 3, 1, 7, 3, 4, 7, 2, 8, 5, 0, 6, 2)
    .distinct()
    .sorted( (i,j) -> i-j )
    .limit(3)
    .forEach( System.out::println );
```

Beispiel

`toArray`

```
int[] a = IntStream.range(1, 3).toArray();

Object[] a = Stream.of("1", "2", "3").map( Integer::parseInt )
        .toArray();

Integer[] a = (Integer[]) Stream.of(1, 2, 3)
        .toArray();
```

```
8
9  String[] a = Stream.of(1, 2, 3).map( (i) -> i.toString() )
10         .toArray( String[]::new ); // using generator
```

Terminale Operationen mit Kollektions-Ergebnis

- Die Methode `collect` erzeugt eine Kollektion aus den Elementen des Streams.
- `IntStream` und andere Streams mit primitiven Daten haben keine entsprechende Operation.
- Das Argument von `collect` ist ein *java.util.stream.Collector*. Die Erzeugung einer Kollektion ist damit Sonderfall einer aggregierenden Operation.
- Für die Erzeugung einer Kollektion verwendet man typischerweise einen vordefinierten `Collector` aus der Klasse *java.util.stream.Collectors*.
- Einfache Kollektionserzeuger in Collectors sind:
  - `toList()`
  - `toSet()`
  - `toCollection(Supplier<C> collectionFactory)`

Beispiel

`collect`

```
1  List<Integer> l1 = Stream.of(1, 2, 3).collect( Collectors.toList() );
2
3  List<Integer> l2 = IntStream.range(1, 4).boxed()
4          .collect( Collectors.toList() );
5
6  Set<String> s1 = (Set<String>) Stream.of("1", "2", "3")
7          .collect( Collectors.toSet());
8
9  Set<String> s2 = (Set<String>) Stream.of("1", "2", "3")
10         .collect( Collectors.toCollection( HashSet::new) );
```

```
1  // Generating a map from a stream of strings
2
3  Map<String, Integer> m = Stream.of("1", "2", "3")
4          .collect(
5              Collectors.toMap(
6                  (s) -> s,
7                  Integer::parseInt
8              )
9          );
```

In `Collectors` finden sich **Kollektoren mit denen Maps erzeugt werden können**, die eine Gruppierung bzw. eine Partitionierung der Stream-Elemente darstellen:

- `static <T,K> Collector<T,?,Map<K,List<T>>> groupingBy(Function<? super T,? extends K> classifier)`

  Gruppiert die Elemente entsprechend einer Klassifizierungsfunktion.
- `static <T> Collector<T,?,Map<Boolean,List<T>>> partitioningBy(Predicate<?`

`super T> predicate)`

Partitioniert die Elemente entsprechend einem Prädikat.

Beispiel

```
collect(groupingBy)
```

Hilfreiche Methoden

```
import static java.util.stream.Collectors.groupingBy;
import static java.util.stream.Collectors.partitioningBy;
import static java.util.stream.Collectors.counting;
```

```
Map<Integer, List<Integer>> groupedByMod3 = Stream.of(1, 2, 3, 4, 5, 6 ,7 ,8, 9)
        .collect( groupingBy( (x) -> x%3 ) );
```

Ausgabe: `groupedByMod3 = {0=[3, 6, 9], 1=[1, 4, 7], 2=[2, 5,8]}`

```
Map<Integer, List<String>> groupedByLength = Stream.of(
            "one", "two", "three", "four", "five", "six", "seven", "eight")
        .collect( groupingBy( (s) -> s.length() ) );
```

Ausgabe: `groupedByLength ⟹ {3=[one, two, six], 4=[four, five], 5=[three, seven, eight]}`

Das Interface `Stream` bzw. die Interfaces für Ströme primitiver Daten (`IntStream`, etc.) bieten einige **einfache aggregierende Funktionen für Standardoperationen** auf allen Elementen des Stroms.

Beispiel

```
long count = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9).count();

long sum = IntStream.of(1, 2, 3, 4, 5, 6, 7, 8, 9).sum();

OptionalDouble av = IntStream.of(1, 2, 3, 4, 5, 6, 7, 8, 9).average();
```

Das Interface Stream bieten einige einfache **aggregierende Funktionen für den Test aller Elemente des Stroms** mit einem übergebenen Prädikat.

Beispiel

```
boolean anyEven = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9)
        .anyMatch( (x) -> x%2 == 0 );

boolean allEven = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9)
        .allMatch( (x) -> x%2 == 0 );

boolean noneEven = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9)
        .noneMatch( (x) -> x%2 == 0 );
```

Das Interface `Stream` bietet die Funktionen `findFirst` und `findAny` für die „Suche“ nach dem ersten bzw. irgendeinem Element in einem Stream.

Achtung!

Diese Methoden haben kein Prädikat als Parameter. Es empfiehlt sich darum den

`Stream` vorher mit dem entsprechenden Prädikat zu filtern.

Beispiel

```
Optional<Integer> firstEven = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9)
        .filter( (x) -> x%2 == 0 )
        .findFirst();
```

Ausgabe: `firstEven ⟹ Optional[2]`

Das Interface Stream bietet die Funktion

```
Optional<T> reduce(BinaryOperator<T> accumulator)
```

mit der eine Funktion auf jedes Element und das bisherige Zwischenergebnis angewendet werden kann.

Falls der erste Wert nicht der Startwert sein soll, verwendet man:

```
Optional<T> reduce(T identity, BinaryOperator<T> accumulator)
```

Beispiel

reduce

```
Optional<Integer> sumOfAll = Stream.of(1, 2, 3, 4, 5).reduce( (a, x) -> a+x );
```

Ausgabe: `sumOfAll ⟹ Optional[15]`

```
Optional<Integer> subOfAll = Stream.of(1, 2, 3, 4, 5).reduce( (a, x) -> a-x );
```

Ausgabe: `subOfAll ⟹ Optional[-13]`

```
int sumOfAllPlus100 = Stream.of(1, 2, 3, 4, 5)
        .reduce(100, (a, x) -> a+x );
```

Ausgabe: `sumOfAllPlus100 ⟹ 115`

Es gibt einen Kollektor mit dem String-Elemente zu einem String konkateniert werden können:

```
static Collector<CharSequence,?,String> joining(CharSequence delimiter)
```

Beispiel

reduce

```
String concat = Stream.of("one", "two", "three")
        .collect( joining("+") );
```

Ausgabe: `concat = one+two+three`

Streams - fortgeschrittene Konzepte

Ausgewählte Eigenschaften des *Basisinterface* aller Streams

Parallele und sequentielle Streams.

```
1  package java.util.stream;
2
3  public interface BaseStream<T, S extends BaseStream<T,S>> {
4      /** Closes the stream, releasing any resources associated with it. */
5      void close();
6
7      /** Returns an equivalent stream that is parallel. */
8      S parallel();
9      /** Returns an equivalent stream that is sequential. */
10     S sequential();
11 }
```

```
13 public interface Stream<T> extends BaseStream<T, Stream<T>> {
14     // ...
15 }
```

Die Interfacedefintion (`BaseStream<T, S extends BaseStream<T,S>>`) ist eine Anwendung des CRTP; d. h. des *Curiously Recurring Template Patterns*. Bei diesem Idiom haben wir eine Klasse `X`, die von einer generischen Klasse oder einem generischen Interface `S` abgeleitet wird, wobei die ableitende Klasse `X` sich selber als Typparameter verwendet. Dies erlaubt die Definition einer Fluent-API, bei der Methoden, die in der Basisklasse definiert sind, den abgeleiteten Typ zurückgeben.

Erzeugen von eigenen Streams mittels StreamSupport

Die Implementierung des Interfaces `Stream<T>` ist ggf. sehr aufwändig. Alternativ kann die Klasse `StreamSupport` verwendet werden, um auf einem `Spliterator` basierende Streams zu erzeugen.

```
1  package java.util.stream;
2
3  public final class StreamSupport {
4
5      /** Creates a new sequential or parallel Stream from a Spliterator. */
6      static <T> Stream<T> stream(Spliterator<T> spliterator, boolean parallel);
7
8      // ...
9  }
```

Übung - Streams Grundlagen

3.1. Streams und Lambda Ausdrücke

```java
void main() {
    List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "Diana", "Eve");
    List<Integer> grades = Arrays.asList(85, 42, 91, 67, 55);

    // TODO 1: Using a lambda, filter the grades list to only keep grades ⩾ 60
    // and print the passing grades.
    // Hint: Use stream(), filter(), and forEach().


    // TODO 2: Create a Predicate<Integer> lambda called `isExcellent`
    // that returns true if a grade is 90 or above. Then print all excellent grades.


    // TODO 3: Create a Function<Integer, String> lambda called `toLetterGrade`
    // that converts a numeric grade to a letter:
    //   90+  → "A"; 75+  → "B" ;  60+  → "C"; below 60 → "F"
    // Then print each grade alongside its letter grade.


    // TODO 4: Using a lambda, calculate and print the average of all grades.
    // Hint: Use stream() and mapToInt().
}
```

Übung

3.2. Java Streams

> **Bemerkung**
>
> Verwenden Sie ausschließlich Streams und Lambda-Ausdrücke.

1. Schreiben Sie eine Methode `int sumOfSquares(int[] a)` die die Elemente des Arrays quadriert und dann die Summe berechnet.
2. Schreiben Sie eine Methode `int sumOfSquaresEven(int[] a)` die die Elemente des Arrays quadriert, und dann die Summe berechnet für alle Elemente die gerade sind.
3. Schreiben Sie eine Methode, die eine Liste von Strings (`List<String>`) in eine flache Liste von Zeichen (`List<Integer>`) umwandelt.
4. Schreiben Sie eine Methode, die die Zahlen von `1` bis `Integer.MAX_VALUE` addiert. Nutzen Sie `IntStream.range()` um die Zahlen zu iterieren. Messen Sie die Ausführungsdauer für die *sequentielle* und *parallele* Ausführung (siehe Anhang für eine entsprechende Methode zur Zeitmessung.)

Um die Ausführungsdauer Ihrer Methode zu messen, können Sie folgenden Methode verwenden:

```
1 void time(Runnable r) {
2     final var startTime = System.nanoTime();
3     r.run();
4     final var endTime = System.nanoTime();
5     System.out.println("elapsed time: "+(endTime - startTime));
6 }
```

Ein Aufruf der Methode `time` könnte dann so aussehen:

```
1 time(() -> System.out.println(sumOfSquares(new int[]{1,2,3,4,5,6,7,8,9,0})));
```

Bewertung der Fähigkeiten der Standardoperationen von Java Streams

Herausforderung *Low-representational Gap* gelöst?

Lösung mit standard *Stream* Primitiven (≤ Java 24)

Benötigte Imports

```
1 import static java.util.Comparator.comparingDouble;
2 import static java.util.function.Predicate.not;
3 import static java.util.stream.Collectors.groupingBy;
4 import static java.util.stream.Collectors.minBy;
```

Geschäftslogik

```
1 List<Student> empfehlungenS = alleStudierenden.stream()
2     .filter(not(Student::hatStipendium))
3     .collect(groupingBy(Student::studiengang,
4                         minBy(comparingDouble(Student::schnitt))))
5     .values().stream().flatMap(Optional::stream)
6     .sorted(comparingDouble(Student::schnitt))
7     .toList();
```

Bewertung

Wir sind näher an der Geschäftslogik, aber technische Artefakte scheinen noch durch!

- Um die gewünschte Gruppierung zu erhalten, werden die besten Studierenden in einer Zwischendatenstruktur (*Map*) aufgesammelt (`collect(...)`). Diese muss - um eine Stream-orientierte Weiterverarbeitung zu ermöglichen - wieder in einen Stream verwandelt werden, der über den *Values* der *Map* operiert (`values().stream()`).
- Da wir in eine *Map* aufgesammelt hatten, haben wir einen *Stream of Optionals*; `minBy(...)` liefert ein `Optional`. Somit müssen wir die `Optional`s im `Stream<Optional<Student>>` über `flatMap(Optional::stream)` entpacken.

Benutzerdefinierte Zwischenoperationent aka *Stream Gatherers*

Im Folgenden verwenden wir die Begriffe *Custom Intermediate Operations* und *Stream Gatherers* faktisch synonym.

TODODODODODO

Bewertung der Fähigkeiten von Java Streams

Herausforderungen gelöst?

Custom Intermediate Operation/Custom Gatherer

```java
static <T, K> Gatherer<T, ?, T> reducePerGroup(
            Function<T, K> grouping, BinaryOperator<T> reducer) {
    return Gatherer.ofSequential(
        HashMap<K, T>::new,
        (map, element, downstream) -> {
            map.merge(grouping.apply(element), element, reducer);
            return true;
        },
        (map, downstream) -> map.values().forEach(downstream::push)
    ); }
```

```java
List<Student> empfehlungenG = alleStudierenden.stream()
    .filter(not(Student::hatStipendium))
    .gather(reducePerGroup(
                Student::studiengang,
                BinaryOperator.minBy(comparingDouble(Student::schnitt))))
    .sorted(comparingDouble(Student::schnitt))
    .toList();
```

---

benötigte Imports

```java
import static java.util.Comparator.comparingDouble;
import static java.util.function.Predicate.not;
import static java.util.stream.Collectors.groupingBy;
import static java.util.stream.Collectors.minBy;
```

4. Parallelisierung

Effizienz von Parallelisierung

> Als allgemeine Faustregel gilt, dass Geschwindigkeitssteigerungen in der Regel dann spürbar sind, wenn die Sammlung groß ist, typischerweise mehrere tausend Elemente umfasst.[2]
>
> —Aleksandar Prokopec, Heather Miller - Parallel Collections

---

[2] Übersetzung aus dem Englischen mit Hilfe von DeepL.

5. Java Streams - abschließende Betrachtung

(Java) Streams unterstützen die effiziente, korrekte Verarbeitung von (Massen-)Daten durch die Anwendung von funktionalen und deklarativen Konzepten mit dem Ziel einer möglichst geringen Repräsentationslücke ( *Low representational gap*).

6. Vergleich von Java's Stream-API mit Scala's Stream-API

Java Streams vs. Scala Collections — Konzeptioneller Vergleich

	Java Streams	Scala Collections
Pipeline-Modell	Deklarativ, verkettet	Deklarativ, verkettet
Funktionen als Parameter	Lambdas, Method References	Funktionsliterale, Platzhalter _
Quelle unveränderlich	Ja	Ja
Streams = Collections?	Nein — separates Konzept, explizites `.stream()` / `.toList()`	Ja — Operationen direkt auf Collections
Auswertung	Lazy (erst bei Terminal Op)	Eager (sofort); `LazyList` als Opt-in
Erweiterbarkeit	Collectors seit Java 8, Gatherers seit Java 24	'*„Nicht nötig“* — umfangreiche Standardbibliothek'
Parallelisierung	Tief integriert (`.parallelStream()`)	Separate Bibliothek (`.par`) seit Scala 2.13

Scala verfolgt einen anderen Designansatz: Collections *sind* die Pipeline. Es gibt keine Trennung zwischen Datenstruktur und Transformations-API. Das macht den Einstieg einfacher und den Code kürzer, bedeutet aber auch, dass Laziness explizit gewählt werden muss.

Javas Trennung in Collection und Stream ist verboseer, ermöglicht dafür aber Lazy Evaluation als Standard und eine tief integrierte Parallelisierung.

Java Streams vs. Scala Collections — Codebeispiel

Java (mit Gatherer)

```java
alleStudierenden.stream()
  .filter(not(Student::hatStipendium))
  .gather(reducePerGroup(
      Student::studiengang,
      BinaryOperator.minBy(
        comparingDouble(
          Student::schnitt))))
  .sorted(comparingDouble(
      Student::schnitt))
  .toList();
```

Scala

```scala
alleStudierenden
  .filterNot(_.hatStipendium)
  .groupBy(_.studiengang)
  .values
  .map(_.minBy(_.schnitt))
  .toList
  .sortBy(_.schnitt)
```

Bewertung

Syntaktisches Rauschen: Java erfordert `.stream()` / `.toList()` als Rahmen, Typnamen in Method References (`Student::schnitt`) und Wrapper wie `comparingDouble`. Scala's _-Platzhalter und die direkte Arbeit auf Collections eliminieren diesen Overhead.

Gruppierung: In Scala ist `groupBy` eine normale Collection-Methode — kein Pipeline-Bruch, kein Collector. Java brauchte für denselben Effekt zunächst den `collect`/`Collector`-Umweg und seit Java 24 die `Gatherer`-API.

Vergleich und Reduktion: `_.minBy(_.schnitt)` in Scala vs. `BinaryOperator.minBy(comparingDouble(Student::schnitt))` in Java — die explizite Typmaschinerie des Java-Typsystems ist hier deutlich sichtbar.

Lazy vs. Eager: Scalas Code wird sofort ausgewertet — `groupBy` erzeugt *sofort* eine `Map`. Javas Pipeline hingegen ist vollständig lazy; erst `.toList()` löst die Berechnung aus. Für große Datenmengen kann das ein relevanter Unterschied sein.