Java Stream API

Dozent: Prof. Dr. Michael Eichberg
Kontakt: michael.eichberg@dhbw.de, Raum 149B
Version: 1.0

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

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

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

Quellen:

1. Dokumentation des JDK und JEP 485
2. Dokumentation der Scala (2.13 bzw. 3) Standardbibliothek;

   Aniefiok Akpan; Streams in Scala, July 2023
3. Das Rust Buch
4. Th. Letschert bzgl. der konkreten *Java Streams API (Deepdive)*

Externe Links: Die folgenden Youtube Videos besprechen Stream Gatherers sehr tiefgehend.

- Better Java Streams with Gatherers - JEP Cafe #23
- Deep Dive into Gatherers - JEP Cafe #24

KI Verwendung: Bei der Erstellung der Folien wurden KI Assistenten (insbesondere Claude Opus 4.6/4.7) unterstützend eingesetzt. Dies erfolgte insbesondere, um effizient die Grafiken (d. h. die SVG Dateien) zu generieren, oder um sich Übersichtstabellen generieren zu lassen. Weiterhin wurde KI zur allgemeinen Qualitätssicherung eingesetzt. Inhalte, die ggf. von der KI vorgeschlagen wurden, wurden im Falle der Übernahme explizit validiert. *KI wurde nicht verwendet* für den Aufbau, die Struktur und die Auswahl der grundlegenden Inhalte.

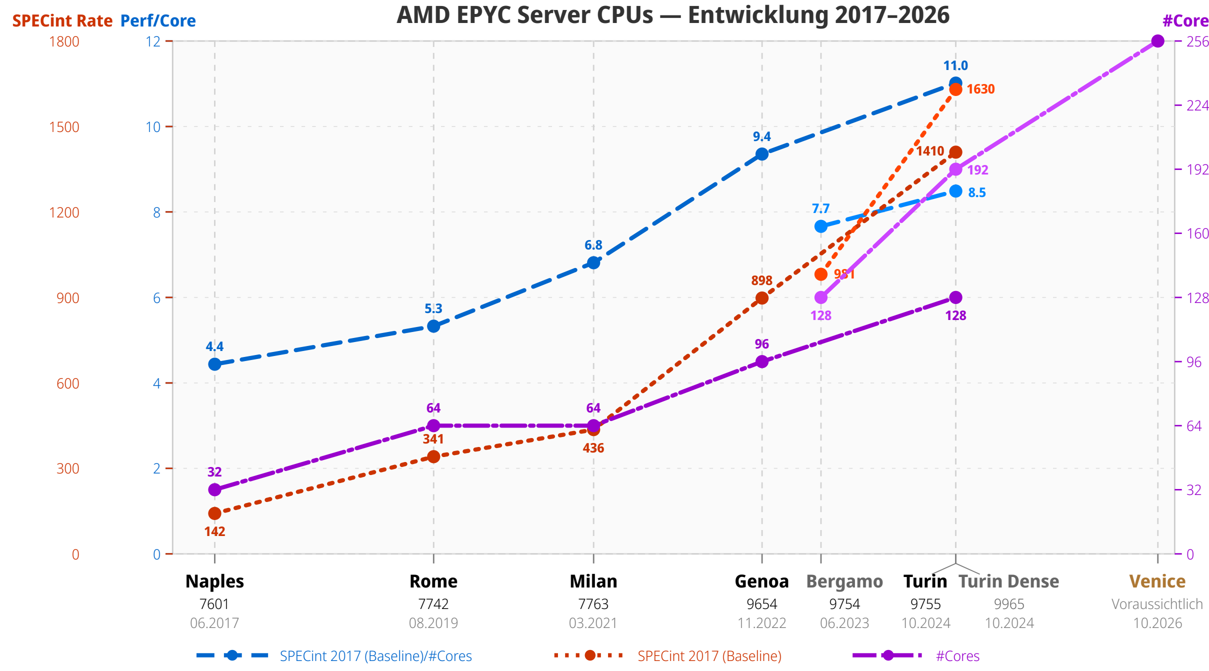
1. Motivation

(Aktuelle) Herausforderungen bei der Datenverarbeitung

- Große 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 verarbeitet 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 Verarbeitung von allg. Daten (z. B. Finanzdaten/-transaktionen) oder von (IoT) Sensoren
> - Verarbeitung von Ereignissen (Event)
> - Aufbereitung großer Wörterbücher (z. B. für Passwortwiederherstellung)
> - Verarbeitung von AI Streaming Responses
> - Verarbeitung großer Logdateien

- Parallele Verarbeitung und effiziente Nutzung von modernen CPU-Architekturen



Wenn man sich die Performance anschaut, dann hat sich die Leistung pro Core (für die Hauptlinie der EPYC CPUs) von 2017 bis 2024 um etwa den Faktor 2.5 verbessert, während die Rechenleistung pro CPU insgesamt um den Faktor 10 gestiegen ist. Der Gesamtanstieg ist somit wesentlich auf die Steigerung der Anzahl der Cores (Faktor 4) zurückzuführen.

Insbesondere die korrekte und effiziente Nutzung mehrere Threads (d. h. von nebenläufiger Programmierung) ist im Allgemeinen schwierig. Die Verwendung von (soweit möglich) „transparenter“ Parallelisierung 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 eine korrekte, leicht wartbare und erweiterbare 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) sortierte Liste von förderungswürdigen Studierenden nach Förderwürdigkeit

Logik:

1. Filtere aus allen Studierenden diejenigen heraus, die bereits ein Stipendium haben,
2. gruppiere die verbleibenden nach Studiengang,
3. wähle pro Studiengang den mit der besten Durchschnittsnote, und
4. erstelle daraus eine nach Note sortierte Empfehlungsliste.

Setup

```java
1  record Student(boolean hatStipendium, String studiengang, double schnitt){};
2  List<Student> alleStudierenden = List.of(new Student(false, "Inf.", 1.3), …);
```

Klassische Implementierung der Geschäftslogik

```java
3  // Schritt 1: Studierende ohne Stipendium sammeln
4  List<Student> ohneStipendium = new ArrayList<>();
5  for (Student s : alleStudierenden) {
6      if (!s.hatStipendium()) { ohneStipendium.add(s); }
7  }
8  // Schritt 2: Nach Studiengang gruppieren
9  Map<String, List<Student>> nachStudiengang = new HashMap<>();
10 for (Student s : ohneStipendium) {
11     nachStudiengang
12         .computeIfAbsent(s.studiengang(), k -> new ArrayList<>())
13         .add(s);
14 }
15 // Schritt 3: Pro Gruppe den Besten finden
16 List<Student> empfehlungen = new ArrayList<>();
17 for (var eintrag : nachStudiengang.entrySet()) {
18     Student bester = null;
19     for (Student s : eintrag.getValue()) {
20         if (bester == null || s.schnitt() < bester.schnitt()) { bester = s; }
21     }
22     if (bester != null) {
23         empfehlungen.add(bester);
24     }
25 }
```

```
26 // Schritt 4: Empfehlungen nach Note sortieren
27 empfehlungen.sort(Comparator.comparingDouble(Student::schnitt));
```

2. Zentrale Konzepte von (Java) Streams

Achtung!

Im Folgenden betrachten wir die Java Stream API und nicht Java I/O Streams.

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

Im Folgenden betrachten wir auch nicht Reactive Streams, die seit Java 9 über die `java.util.concurrent.Flow` API unterstützt werden.

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 bzw. erlauben einen imperativen Programmierstil.

Deklarative Programmierung:

Das Ziel ist es auszudrücken *was* erreicht werden soll, ohne das *wie* genau anzugeben.

(Ein sehr prominentes Beispiel für eine deklarative Programmiersprache ist die Datenbankabfragesprache SQL.)

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 worden.

Streams - Einführung am Beispiel

Programmiertechnische Sicht:

Streams erlauben auf Sammlungen (🇺🇸 *Collections*) die Ausführung von funktional-orientierten Massen-Operationen.[1]

Konzeptionelle Sicht:

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

Beispiel

Benötigte Imports (nicht in der JShell)

```
1 import java.util.Arrays;
2 import java.util.List;
```

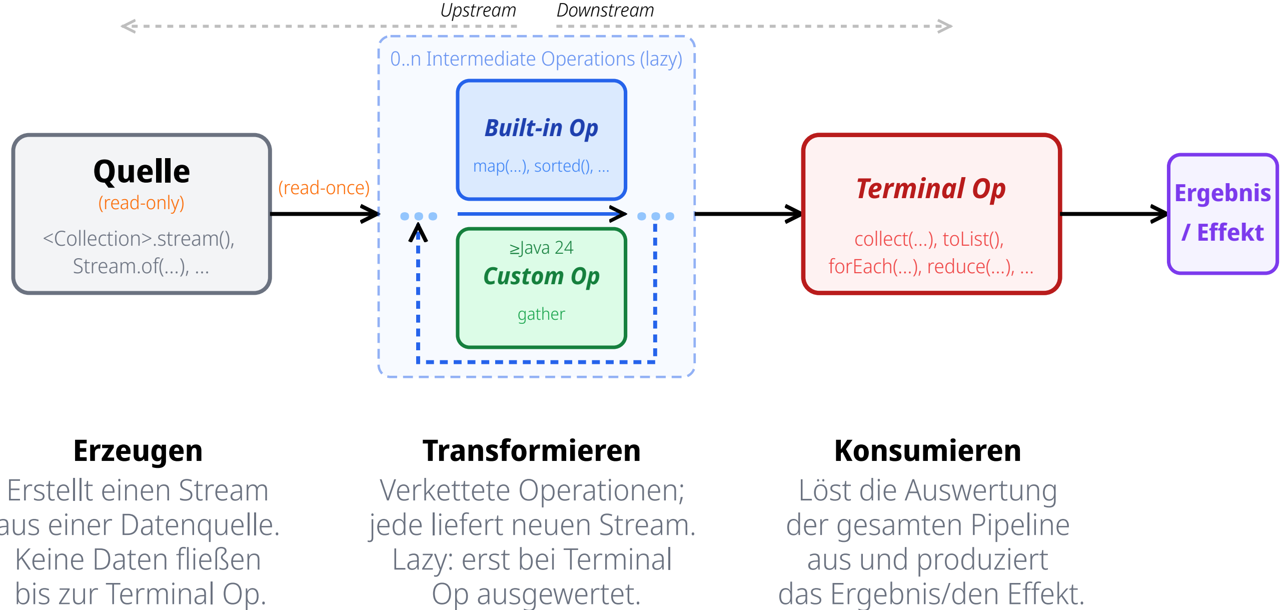
```
1 List<Integer> li = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);
2 List<Integer> lr = li
3     .stream()                 // Erzeugt einen Stream basierend auf der Liste
4     .filter(x -> x % 2 == 0)    // Filterung mit Hilfe eines Prädikats
5     .map(x -> 10 * x)         // Bilde jedes Element auf das Zehnfache ab
6     .toList();                // Aufsammeln der Ergebnisse
7 lr.forEach(x -> IO.print(x + " "));
```

Ausgabe: `20 40 60 80`

---

[1] Th. Letschert

Streams - Grundlegendes Konzept

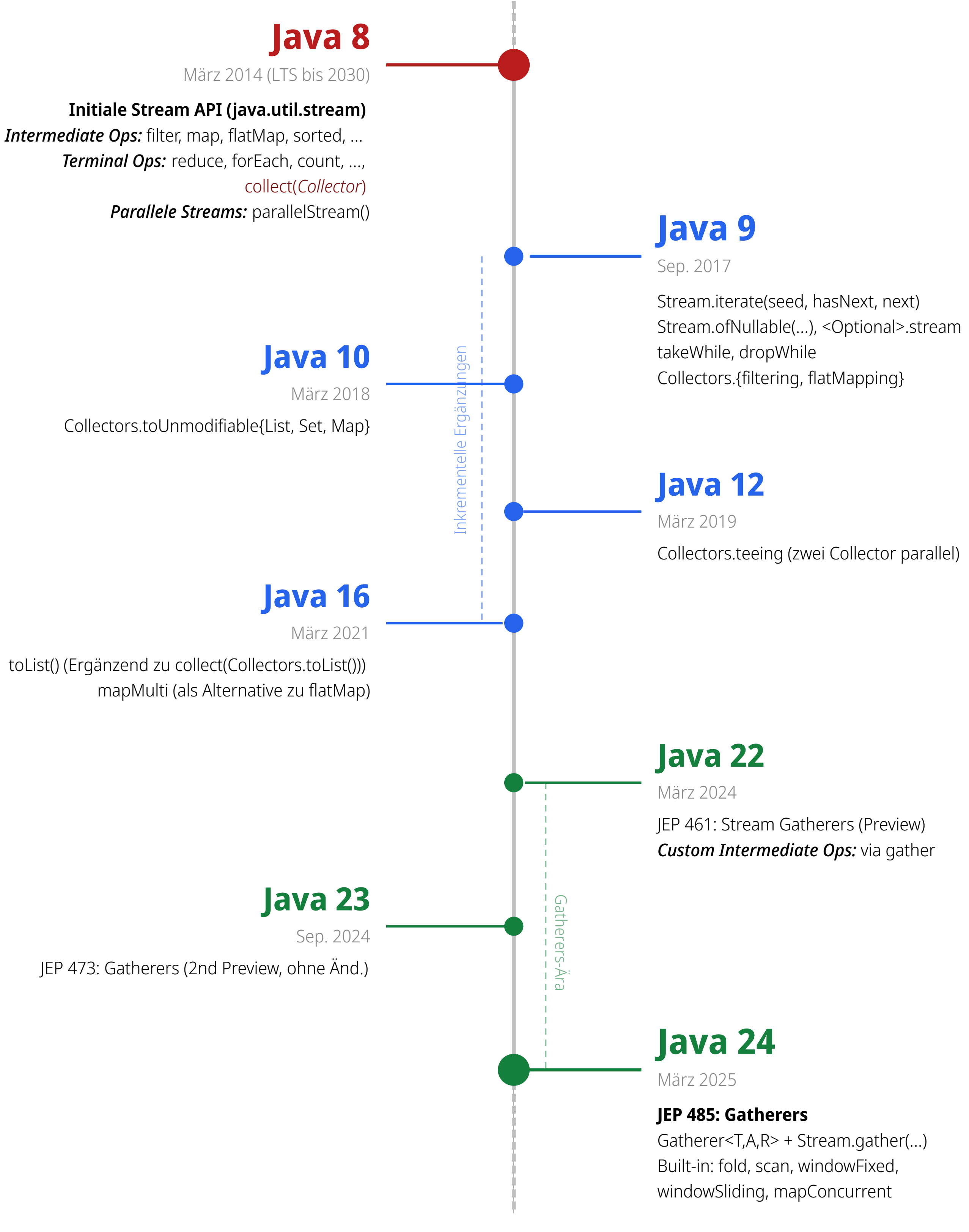


Wir sagen auch, dass die Daten *Downstream gepusht* werden. Dies bedeutet nichts anderes, als dass die Daten *stromabwärts* weitergereicht werden.

Upstream (*stromaufwärts*) bezeichnet die entgegengesetzte Richtung: eine Operation X liegt *upstream* von Y, wenn X zeitlich vor Y ausgeführt wird und somit näher an der Quelle liegt.

Es ist möglich als terminale Operation einen `Iterator` bzw. `Spliterator` zu erzeugen. In diesen beiden Fällen erfolgt die Evaluation der Pipeline nicht unmittelbar durch die terminale Operation (d. h. nicht *eager* sondern *lazy*). 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 (<BaseStream>.iterator()/<BaseStream>.spliterator()) führt zu einem Bruch des deklarativen Pipeline-Modells, der in den allermeisten Fällen vermieden werden kann. Häufig sind die Methoden nur noch im Zusammenhang mit Legacy-APIs relevant, die mit `Iterator`/`Spliterator` arbeiten.

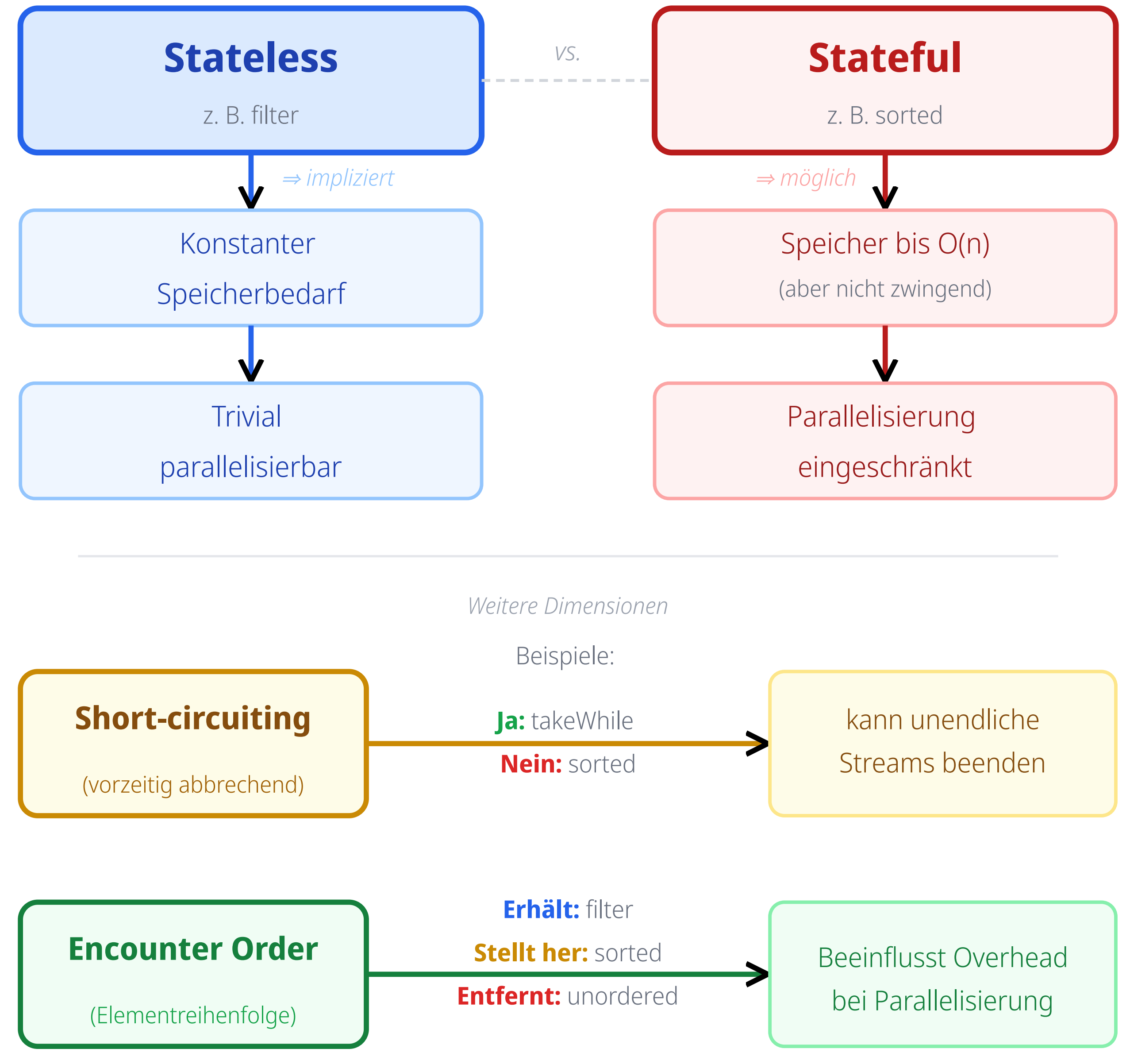
Java Stream API - Evolution



`<Stream>.flatMap(…)` bildet jedes Element auf einen Stream ab und ebnet (engl. flattens) diese zu einem einzigen Stream ein.

`<Stream>.mapMulti(…)` ist eine imperative Variante von `<Stream>.flatMap(…)` und ggf. auch schneller, wenn die Zielliste immer/meistens sehr klein ist [JavaDoc].

Eigenschaften der *Intermediate Operations* von Java 8 bis 26



Stateless Ops(🇩🇪 *zustandslose Operationen*):

Transformieren die Elemente jeweils völlig unabhängig von allen anderen.

Stateful Ops(🇩🇪 *zustandsbehaftete Operationen*):

Transformieren die Elemente abhängig von anderen.

Encounter-Order:

Die *Encounter Order* hängt typischerweise an der Quelle. Zum Beispiel ist ein Stream über die Elemente einer Liste *Ordered* während er für Sets *unordered* ist. Sollte ein Stream über die Elemente eines Sets allerdings sortiert werden (`sorted()`), dann ist dieser ab diesem Zeitpunkt ordered.

> ※ Hinweis
>
> Es kann interessant sein, einen Stream explizit als `unordered()` zu markieren,

da dann ggf. weitere Optimierungen bei der Auswertung möglich sind.

Stream-Operationen mit Seiteneffekten

> **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.

Überblick über wichtige 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`	Op. abhängig	Op. abhängig	Op. abhängig	Op. abhängig

Die Eigenschaften von `gather` hängen von der übergebenen Gatherer-Implementierung ab. Javas Gatherer-Implementierung für `windowFixed(k)` ist zum Beispiel eine stateful Operation mit O(k) Speicherbedarf, die nicht short-circuiting ist, aber dennoch gut parallelisierbar — insbesondere, wenn die Fenstergröße klein ist im Vergleich zur Anzahl an Elementen. Darüber hinaus ist es durchaus möglich Gatherer zu implementieren, die einen Speicherbedarf jenseits von O(n) haben. Zum Beispiel ist es möglich einen Gatherer zu implementieren, der alle Permutationen der Elemente berechnet. Dieser hätte einen Speicherbedarf von O(n!) und könnte somit nur auf sehr kleinen Streams sinnvoll eingesetzt werden.

3. Java Streams API - Deep Dive

Bewertung der Fähigkeiten der Standardoperationen von Java Streams

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 des entsprechenden Wrapper-Typs umgewandelt werden.

Beispiel

```
IntStream isPrim = IntStream.range(1, 10);
Stream<Integer> isObj = isPrim.boxed();     // Umwandlung in Boxed Stream
```

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, sequentielle 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 `range(…)` und `rangeClose(…)`-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

- `<R> map(Function<? super T,? extends R> mapper)`:

  Transformiert jedes Element in ein anderes.

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

  Filtert Elemente heraus.

- `<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper)`:

  Transformiert jedes Element in einen Stream und fügt die Streams zusammen.

- `<R> mapMulti(BiConsumer<? super T,? super Consumer<? super R>> mapper)`:

  Imperative Variante von `flatMap`.

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

  Führt eine Aktion für jedes Element aus, ohne den Stream zu verändern.

Beispiel

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

List<Integer> is = IntStream.range(1, 10)
    .filter(i -> i % 2 != 0)
    .peek(i -> System.out.print(i+ " "))
    .map(i -> 10 * i)
    .boxed()
    .toList();
System.out.println(is);
```

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

Beispiel

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

static Stream<Integer> range(int from, int to) {
    return IntStream.range(from, to).boxed();
}

List<Integer> is = Stream.of(0, 1, 2)
    .flatMap(i -> range(10 * i, 10 * i + 10))
    .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 gemäß ihrer natürlichen Ordnung (`T extends Comparable<T>`).
- `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)
    .toList();
```

Ausgabe: `is` ⟹ `[1, 2, 3]`

Terminale Operationen

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(e -> IO.print(e + " "));
```

Ausgabe: `0 1 2`

Beispiel: `toArray`

```java
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();

String[] a = Stream.of(1, 2, 3).map( (i) -> i.toString() )
                         .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`

```java
List<Integer> l1 = Stream.of(1, 2, 3).collect( Collectors.toList() );

List<Integer> l2 = IntStream.range(1, 4).boxed()
        .collect( Collectors.toList() );

Set<String> s1 = (Set<String>) Stream.of("1", "2", "3")
        .collect( Collectors.toSet() );

Set<String> s2 = (Set<String>) Stream.of("1", "2", "3")
        .collect( Collectors.toCollection( HashSet::new ) );
```

```java
// Generating a map from a stream of strings

Map<String, Integer> m = Stream.of("1", "2", "3")
        .collect(
            Collectors.toMap(
                Function.identity(), // (s) -> s
                Integer::parseInt
            )
        );
```

Resultat: `m` ⟹ `{1=1, 2=2, 3=3}`

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)`

Import hilfreicher Methoden

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

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

Resultat: `groupedByMod3` $\Longrightarrow$ `{0=[3, 6, 9], 1=[1, 4, 7], 2=[2, 5, 8]}`

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

Resultat: `groupedByLength` $\Longrightarrow$ `{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 **aggregierende Funktionen für Standardoperationen** auf allen Elementen des Stroms.

Beispiel

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

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

Beispiel

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

```java
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

```java
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:

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

Beispiel: `reduce`

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

Resultat: `sumOfAll` ⟹ `Optional[15]`

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

Resultat: `subOfAll` ⟹ `Optional[-13]`

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

Resultat: `sumOfAllPlus100` ⟹ `115`

In den Beispielen verwenden wir `a` als Variablenname für den ersten Parameter, da dies der Akkumulator ist, der das Zwischenergebnis enthält.

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

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

Beispiel: `collect(joining(…))`

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

Resultat: `concat = "one+two+three"`

Streams - fortgeschrittenere Konzepte

Ausgewählte Eigenschaften des *Basisinterface* aller Streams

Parallele und sequentielle Streams.

```java
package java.util.stream;

public interface BaseStream<T, S extends BaseStream<T,S>> {
    /** Closes the stream, releasing any resources associated with it. */
    void close();

    /** Returns an equivalent stream that is parallel. */
    S parallel();
    /** Returns an equivalent stream that is sequential. */
    S sequential();
}
```

```java
public interface Stream<T> extends BaseStream<T, Stream<T>> {
    // ...
}
```

Die Interfacedefintion (`BaseStream<T, S extends BaseStream<T,S>>`) ist eine Anwendung des *Curiously Recurring Template Pattern*s (*CRTP*). 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.

```java
package java.util.stream;

public final class StreamSupport {

    /** Creates a new sequential or parallel Stream from a Spliterator. */
    static <T> Stream<T> stream(Spliterator<T> spliterator, boolean parallel);

    // ...
}
```

Übung - Streams - erste Schritte

3.1. Streams und Lambda Ausdrücke

```java
List<Integer> punkte = Arrays.asList(85, 42, 91, 67, 55);
```

1. Filtere mit einem Lambda die Liste `punkte`, sodass nur Noten >= 60 übrig bleiben, und gib die bestandenen Noten aus.

   Hinweis: Verwende `stream()`, `filter()` und `forEach()`.

2. Erstelle ein `Predicate<Integer>`-Lambda namens `isExcellent`, das `true` zurückgibt bei 90 oder mehr Punkten. Gib dann alle ausgezeichneten Noten aus.
3. Erstelle ein `Function<Integer, String>`-Lambda namens `toLetterGrade`, das die Punkte in einen Buchstaben umwandelt: 90+ → "A"; 80-89 → "B"; 70-79 → "C"; 60-69 → "D"; unter 60 → "F"; gib dann die Punkte zusammen mit der Note aus.
4. Berechne den Durchschnitt aller Noten und gib diese aus.

   Hinweis: Verwende `stream()` und `mapToInt()` und schaue Dir die API von `IntStream` an.

Übung - Streams

3.2. Java Streams

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 Summe der quadrierten Elemente bildet, für die das Ergebnis der Quadrierung gerade ist.
3. Schreiben Sie eine Methode, die eine Liste von Strings (`List<String>`) in eine flache Liste von Zeichen-Codepoints (`List<Integer>`) umwandelt.

   > **Bemerkung**
   >
   > `String.chars()` liefert einen `IntStream` von Codepoints.

> **‼ Wichtig**
>
> Verwenden Sie ausschließlich Streams und Lambda-Ausdrücke.

4. Bewertung der Fähigkeiten der Standardoperationen von Java Streams

Herausforderung *Low-representational Gap* gelöst?

Lösung mit standard *Stream* Operationen (Java 8 - 26)

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 =
2     alleStudierenden.stream()
3         .filter(not(Student::hatStipendium))
4         .collect(groupingBy(Student::studiengang,
5                             minBy(comparingDouble(Student::schnitt))))
6         .values().stream().flatMap(Optional::stream)
7         .sorted(comparingDouble(Student::schnitt))
8         .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.

5. Benutzerdefinierte Intermediate Operations: Stream Gatherers

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

Welches Problem wollen wir lösen?

```java
List<Student> empfehlungenS = alleStudierenden.stream()
    .filter(not(Student::hatStipendium))
    .collect(groupingBy(Student::studiengang,
                        minBy(comparingDouble(Student::schnitt))))
    .values().stream().flatMap(Optional::stream)
    .sorted(comparingDouble(Student::schnitt))
    .toList();
```

Um (hier) den Bruch in der Pipeline (Stream → List → Stream) zu vermeiden, benötigen wir eine passende *Intermediate Operation*, die von der Standardbibliothek nicht zur Verfügung gestellt wird.

Stream Gatherers (JDK 24+)

Gatherers erlauben die Definition von benutzerdefinierten *Intermediate Operations*.

`Gatherer<T, A, R>` bestehen aus vier Komponenten:

`initializer`: Erzeugt den privaten, Gatherer-internen, ggf. veränderlichen Zustand (Typ `A`).

`integrator`: Verarbeitet jedes Eingabeelement (`T`) und kann beliebig viele Ausgabeelemente (`R`) *downstream* senden.

`combiner`: Vereinigt Zustände bei paralleler Ausführung *(optional)*.

`finisher`: Wird nach dem letzten Element aufgerufen; kann finale Elemente *downstream* senden *(optional)*.

- Damit lassen sich *zustandsbehaftete*, *short-circuiting* und *m:n*-Transformationen als wiederverwendbare, komponierbare, parallelisierbare Bausteine definieren.
- Eingebunden über: `stream.gather(myGatherer)`

Ausgewählte Standard Stream Gatherers

`Gatherers.scan(…)`

Der aktuelle Zustand wird basierend auf dem aktuellen Element aktualisiert. Der initiale Zustand kann explizit angegeben werden.

```
1 Stream<Integer> numbers = Stream.of(1, 2, 3, 4);
2 List<Integer> resultList =
3         numbers.gather(Gatherers.scan(() -> 0, Integer::sum)).toList();
4 // resultList => List.of(1, 3, 6, 10)
```

`Gatherers.windowFixed(…)`

Der Stream wird in Blöcke fester Größe aufgeteilt und diese werden dann *downstream* weitergereicht.

```
1 Map<String,String> nameToPoints =
2     Stream.of("A","90+","B","80+")
3         .gather(Gatherers.windowFixed(2))
4         .collect(Collectors.toMap((l) -> l.get(0),(l) -> l.get(1)));
5 // nameToPoints => {A=90+, B=80+}
```

Benutzerdefinierter Stream Gatherer: reducePerGroup(...)

```
static <T, K> Gatherer<T, ?, T> reducePerGroup(
            Function<T, K> grouping, BinaryOperator<T> reducer) {

    return Gatherer.ofSequential(
        /*initializer:*/ HashMap<K, T>::new,
        /*integrator: */ (map, element, downstream) -> {
            map.merge(grouping.apply(element), element, reducer);
            return true; // <= we will consume more elements
        },
        /*finisher:   */ (map, downstream) -> map.values().forEach(downstream::push)
    );
}
```

Aufgrund der Verwendung von `Gatherer.ofSequential(…)` müssen wir keinen `combiner` angeben; Parallelisierung wird aber auch nicht unterstützt.

Bewertung

Obwohl es sich um einen spezialisierten *Custom Gatherer* handelt, der die Gruppierung nur intern durchführt und diese nicht explizit downstream verfügbar macht, ist es dennoch sehr gut vorstellbar, dass dieser wiederverwendet werden kann.

6. Bewertung der Java Stream API

Herausforderungen gelöst?

Lösung mit `reducePerGroup` *Gatherer*

```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.function.BinaryOperator.minBy;
```

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

Bewertung

Die Umsetzung in Java ist noch einmal näher an der Geschäftslogik, aber der Code ist noch nicht parallelisiert und enthält noch immer einiges an syntaktischem Rauschen.

Übung

6.1. Fakultät für 1 bis 20

Berechnen Sie die Fakultät für n = 1 bis 20 und speichern Sie die Ergebnisse in einer Liste. Verwenden Sie Streams und einen passenden `Gatherer`. Bedenken sie, dass `20!` bereits sehr groß ist!

Bis zu welchem Wert können Sie bei der Verwendung eines passenden primitiven Datentyps ohne Präzisionsverlust die Fakultät von n (n!) berechnen.

> **Hinweis**
>
> Auf primitiven Streams ist `.gather(…)` nicht verfügbar.

Übung

6.2. Fakultät mit `BigInteger`

Berechnen Sie die Fakultät für n = 1 bis 100 und speichern Sie die Ergebnisse in einer Liste. Verwenden Sie Streams und einen passenden `Gatherer`. Verwenden Sie `BigInteger`, um Präzisionsverlust zu vermeiden.

7. Parallelisierung von Streams

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 von DeepL.

Korrektheit bei Parallelisierung

? Frage

Welches Ergebnis erwarten wir:

```
Arrays.stream(new int[]{1,2,3}).reduce(0,(x,y) -> x + y*y);
                                    // x ist das aktuelle Zwischenergebnis und
                                    // y ist der nächste Wert aus der Liste
```

✓ Antwort

14

? Frage

Welches Ergebnis erwarten wir:

```
Arrays.stream(new int[]{1,2,3}).parallel().reduce(0,(x,y) -> x + y*y);
```

✓ Antwort

7226?[3]

Ausgangscode

```
1 IntBinaryOperator f = (x,y) -> x + y*y;
2 Arrays.stream(new int[]{1,2,3}).parallel().reduce(0,f);
```

Erklärung

1. Die Parallelisierung[4] hat (hier) dazu geführt, dass die Liste in drei Teillisten aufgespalten wurde, um die Berechnung dafür zu parallelisieren. Danach wurde für jeden Wert der Teillisten zuerst eine Reduktion mit dem Basiswert `0` durchgeführt; d. h. es wurde erst: `f(0,1)=1`, `f(0,2)=4` und `f(0,3)=9` berechnet.
2. Danach wurden die Zwischenergebnisse verrechnet. D. h. es wurde (*in diesem Falle*) `f(f(0,2),f(0,3))=f(4,9)=85` berechnet und dann dieses Zwischenergebnis mit dem von `f(0,1)` verrechnet: `f(f(0,1),f(f(0,2),f(0,3)))=f(1,85)=7226`.

⚠ Warnung

Die übergebene Reduktionsfunktion `f` verletzt die von der Stream API gestellten Bedingungen (Assoziativität).

1. Lösung *mit Mutable Reduction* (`collect(…)`)

```
Arrays.stream(new int[]{1,2,3})
        .parallel()
        .collect(
                () -> new AtomicInteger(),
                (a,y) -> a.addAndGet(y*y),
                (a,v) -> a.addAndGet(v.get()));
```

2. Lösung effizienter mit `map(…)` und Reduktion über `sum()`

```
Arrays.stream(new int[]{1,2,3})
        .parallel()
```

```
    .map(x -> x * x)
    .sum();
```

---

[4] Im ersten Schritt wurde der Stream aufgeteilt, um die Berechnungen für jeden Teilstream parallel ausführen zu können.

[3] Das Ergebnis hängt von einigen Faktoren ab und kann variieren, ist aber wahrscheinlich nicht 14.

Übung - Streams - Parallelisierung

7.1. Java Streams

Bemerkung

Verwenden Sie ausschließlich Streams und Lambda-Ausdrücke.

Schreiben Sie eine Methode, die die Zahlen von `1` bis `Integer.MAX_VALUE` addiert. Nutzen Sie `IntStream.rangeClosed(…)`, 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:

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

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

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

8. Java Streams - abschließende Betrachtung

Grundlegender Aufbau und Bewertung

> Die drei erweiterbaren Bausteine der Java Stream API:
>
> `Stream pipeline = Spliterator + Gatherer? + Collector`

- (Java) Streams unterstützen die Verarbeitung von (Massen-)Daten durch die Anwendung von funktionalen und deklarativen Konzepten.
- Dies ermöglicht eine Umsetzung von fachlichen Konzepten in Java Code mit einer geringeren Repräsentationslücke (🇺🇸 *Low representational gap*).
- Eine effiziente - weitgehend - transparente Parallelisierung ist möglich.

9. Vergleich von Javas Stream-API mit vergleichbaren APIs anderer Sprachen

Java Streams vs. Scala Collections — Codebeispiel

Java (mit Gatherer)

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

Scala

```
alleStudierenden
  .filterNot(_.hatStipendium)
  .groupBy(_.studiengang)
  .values
  .map(_.minBy(

      _.schnitt)) //⇒Iterable[Student]
  .toList
  .sortBy(_.schnitt)
```

Bewertung

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

Gruppierung:
In Scala ist `groupBy` eine normale Collection-Methode. Allerdings erzeugt `groupBy` eine Map, die anschließend über `.values` wieder zu einem Iterable heruntergebrochen werden muss — ein milderer Bruch als Javas traditioneller `collect().values().stream()`-Ansatz über einen Collector, aber strukturell verwandt. Seit Java 24 kann die Gatherer-API verwendet werden, wodurch in Java dieser Bruch ganz vermieden werden kann.

Vergleich und Reduktion:
`_.minBy(_.schnitt)` in Scala vs. `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.

Vollständiger Scala Code

Der folgende Code kann direkt in der Scala REPL (`scala-cli`) ausgeführt werden.

```
// Ausführbar mit: scala Students.scala (Scala 3)
case class Student(
    hatStipendium: Boolean,
    studiengang: String,
    schnitt: Double
)

@main def stipendien(): Unit =
```

```
 9
10      val alleStudierenden = List(
11          Student(false, "Informatik",    1.3),
12          Student(false, "Informatik",    2.1),
13          Student(true,  "Informatik",    1.0),
14          Student(false, "Informatik",    2.7),
15          Student(false, "Informatik",    1.9),
16          Student(false, "BWL",           1.7),
17          Student(false, "BWL",           2.4),
18          Student(false, "BWL",           1.1),
19          Student(true,  "BWL",           1.2),
20          Student(false, "BWL",           3.0),
21          Student(false, "Maschinenbau",  1.5),
22          Student(false, "Maschinenbau",  2.3),
23          Student(false, "Maschinenbau",  1.8),
24          Student(true,  "Maschinenbau",  1.1),
25          Student(false, "Maschinenbau",  2.9),
26          Student(false, "Medizin",       1.0),
27          Student(false, "Medizin",       1.4),
28          Student(false, "Medizin",       2.2),
29          Student(true,  "Medizin",       1.3),
30          Student(false, "Medizin",       1.8),
31          Student(false, "Jura",          1.6),
32          Student(false, "Jura",          2.5),
33          Student(false, "Jura",          1.2),
34          Student(true,  "Jura",          1.0),
35          Student(false, "Jura",          3.1),
36          Student(false, "Physik",        1.1),
37          Student(false, "Physik",        2.0),
38          Student(false, "Physik",        1.7),
39          Student(true,  "Physik",        1.2),
40          Student(false, "Physik",        2.8)
41      )
42
43      val empfehlungen = alleStudierenden
44          .filterNot(_.hatStipendium)
45          .groupBy(_.studiengang)
46          .values
47          .map(_.minBy(_.schnitt))
48          .toList
49          .sortBy(_.schnitt)
50
51      println("Empfehlungsliste:")
52      empfehlungen.foreach(s =>
53          println(s"  ${s.studiengang}: ${s.schnitt}")
54      )
55
56      val lazyEmpfehlungen = alleStudierenden.to(LazyList)
57          .filterNot(_.hatStipendium)
58          .groupBy(_.studiengang)
59          .values
60          .map(_.minBy(_.schnitt))
61          .toList
62          .sortBy(_.schnitt)
```

```
63
64      println("Lazy Empfehlungsliste:")
65      lazyEmpfehlungen.foreach(s ⇒
66          println(s"  ${s.studiengang}: ${s.schnitt}")
67      )
```

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 unverändert	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	via Collectors und Gatherers	umfangreiche Standardbibliothek; ggf. *Extension Methods*
Parallelisierung	Tief integriert (`.parallelStream()`)	Separate Bibliothek (`.par`) seit Scala 2.13/3

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 (`<Collection>.to(LazyList)`).

Für Scala gibt es auch noch weitere 3rd Party Stream APIs wie Akka Streams oder fs2, die verschiedene Trade-offs zwischen Einfachheit, Leistungsfähigkeit und Funktionalität bieten. Hier haben wir jedoch „nur" solche APIs betrachtet, die direkt in der Standardbibliothek enthalten sind.

Javas Trennung in eine Collection API und eine Stream API führt zu syntaktisch komplexerem Code (🇺🇸 *verbose*), ermöglicht dafür aber *Lazy Evaluation* als Standard und eine tief integrierte Parallelisierung.

> **Bemerkung**
>
> Zwischen Scala 2.x und Scala 3 hat sich die Art wie man *Extension Methods* implementiert verändert. In Scala 2 erfolgte dies über Implizite Klassen (`implicit class`); in Scala 3 gibt es dafür den entsprechenden Sprachmechanismus (`extension`).

Java Streams vs. JavaScript Arrays — Codebeispiel

Java (mit Gatherer)

```
alleStudierenden.stream()
  .filter(not(Student::hatStipendium))
  .gather(reducePerGroup(
      Student::studiengang,
      minBy(
        comparingDouble(
          Student::schnitt))))

  .sorted(comparingDouble(
      Student::schnitt))
  .toList();
```

JS JavaScript

```
Object.values(
    Object.groupBy(
      alleStudierenden.
        filter(s => !s.hatStipendium),
      s => s.studiengang) ).
  map(gruppe =>
    gruppe.reduce((a, b) =>
      a.schnitt < b.schnitt ? a : b)).
  toSorted((a, b) =>
    a.schnitt - b.schnitt);
```

Bewertung

***Pipeline-Bruch* durch `Object.groupBy`:**

Die Gruppierung ist in JavaScript eine statische Funktion, keine Methode auf Arrays. Der Datenfluss muss *um* das Array herumgeschrieben werden statt linear verkettet zu bleiben. In Java vor Gatherers bestand dasselbe Problem; seit Java 24 ist die Pipeline durchgängig.

JavaScript hat kein `minBy`:

Die Vergleichslogik muss manuell im Reducer formuliert werden (`a.schnitt < b.schnitt ? a : b`). Das ist weniger deklarativ als Javas `minBy(comparingDouble(…))`.

Arrow Functions vs. Method References:

`s => !s.hatStipendium` vs. `not(Student::hatStipendium)` — JavaScript kürzer, aber ohne Typsicherheit.

Eager Evaluation:

Jede JavaScript-Operation erzeugt ein vollständiges Zwischenarray. `filter` kopiert alle passenden Elemente in ein neues Array, bevor `Object.groupBy` überhaupt beginnt. Javas Pipeline verarbeitet jedes Element einmal durch alle Stufen — ohne Zwischenspeicherung.

Vollständiger JavaScript Code

Der folgende Code kann direkt in einer aktuellen Version von Node.js ausgeführt werden.

```
/* Ausreichend für das Beispiel, aber nicht idomatischer Code:
class Student {
    constructor(hatStipendium, studiengang, schnitt) {
        this.hatStipendium = hatStipendium;
        this.studiengang = studiengang;
        this.schnitt = schnitt;
    }
    toString() {
```

```
 9          return `Student(${this.hatStipendium}, "${this.studiengang}", ${this.schnitt})`;
10      }
11  }
12  */
13
14  class Student {
15      #hatStipendium;
16      #studiengang;
17      #schnitt;
18
19      constructor(hatStipendium, studiengang, schnitt) {
20          this.#hatStipendium = hatStipendium;
21          this.#studiengang = studiengang;
22          this.#schnitt = schnitt;
23      }
24
25      get hatStipendium() { return this.#hatStipendium; }
26      get studiengang()   { return this.#studiengang; }
27      get schnitt()       { return this.#schnitt; }
28
29      toString() {
30          return `Student(${this.#hatStipendium}, "${this.#studiengang}", ${this.#schnitt})
31      }
32  }
33
34  const alleStudierenden = [
35      new Student(false, "Informatik", 1.3),
36      new Student(false, "Informatik", 2.1),
37      new Student(true, "Informatik", 1.0),
38      new Student(false, "Informatik", 2.7),
39      new Student(false, "Informatik", 1.9),
40      new Student(false, "BWL", 1.7),
41      new Student(false, "BWL", 2.4),
42      new Student(false, "BWL", 1.1),
43      new Student(true, "BWL", 1.2),
44      new Student(false, "BWL", 3.0),
45      new Student(false, "Maschinenbau", 1.5),
46      new Student(false, "Maschinenbau", 2.3),
47      new Student(false, "Maschinenbau", 1.8),
48      new Student(true, "Maschinenbau", 1.1),
49      new Student(false, "Maschinenbau", 2.9),
50      new Student(false, "Medizin", 1.0),
51      new Student(false, "Medizin", 1.4),
52      new Student(false, "Medizin", 2.2),
53      new Student(true, "Medizin", 1.3),
54      new Student(false, "Medizin", 1.8),
55      new Student(false, "Jura", 1.6),
56      new Student(false, "Jura", 2.5),
57      new Student(false, "Jura", 1.2),
58      new Student(true, "Jura", 1.0),
59      new Student(false, "Jura", 3.1),
60      new Student(false, "Physik", 1.1),
61      new Student(false, "Physik", 2.0),
```

```
62      new Student(false, "Physik", 1.7),
63      new Student(true, "Physik", 1.2),
64      new Student(false, "Physik", 2.8),
65  ];
66
67  const empfehlungen =
68      Object.values(
69          Object.groupBy(
70              alleStudierenden.filter((s) => !s.hatStipendium),
71              (s) => s.studiengang,
72          )
73      ).
74      map((gruppe) => gruppe.reduce((a, b) => (a.schnitt < b.schnitt ? a : b))).
75      toSorted((a, b) => a.schnitt - b.schnitt);
76
77  console.log("Empfehlungsliste:");
78  empfehlungen.forEach((s) => console.log(`  ${s.studiengang}: ${s.schnitt}`));
```

Java Streams vs. JavaScript Arrays — Konzeptioneller Vergleich

	Java Streams	JavaScript Arrays
Pipeline-Modell	deklarativ, verkettet	deklarativ, verkettet
Funktionen als Parameter	Lambdas, Method References	Arrow Functions
Quelle unverändert	Ja	Ja (Achtung: `sort()` mutiert!)
Streams = Collections?	Nein — separates Konzept	Ja — Operationen direkt auf `Array`
Auswertung	Lazy (erst bei Terminal Op)	Eager (Zwischenarrays bei jedem Schritt)
Erweiterbarkeit	via Collectors und Gatherers	„*Keine*"[5] — fester API-Satz
Parallelisierung	integriert (`.parallelStream()`)	Nein - Single-threaded
Unendliche Sequenzen	Ja (z. B. `Stream.iterate`)	Nein - Generatoren nicht in Array-API integriert
Typsicherheit	Compile-time	Keine — Fehler erst zur Laufzeit

JavaScripts Array-Methoden wie `filter`, `map` und `reduce` sind dem Stream-Modell oberflächlich sehr ähnlich. Der entscheidende Unterschied liegt in der Auswertungsstrategie: jede Array-Operation erzeugt sofort ein vollständiges Zwischenarray. Bei `array.filter(…).map(…)` werden während der Auswertung *zwei* neue Arrays erzeugt. Javas Lazy-Pipeline *fusioniert* die Schritte und erzeugt keine Zwischenergebnisse.

Ein weiterer fundamentaler Unterschied: `Array.prototype.sort()` mutiert das Array in-place — ein Bruch mit dem ansonsten funktionalen Charakter der Array-API. Erst seit ES2023 gibt es mit `toSorted()` eine nicht-mutierende Alternative.

---

[5] Eine Erweiterung über `Array.prototype` ist möglich aber nicht empfehlenswert.

Java Streams vs. Rust Iteratoren — Codebeispiel

Java (mit Gatherer)

```java
alleStudierenden.stream()
  .filter(not(Student::hatStipendium))
  .gather(reducePerGroup(
      Student::studiengang,


      minBy(
        comparingDouble(
          Student::schnitt))))
  .sorted(comparingDouble(
      Student::schnitt))
  .toList();
```

Rust (mit itertools)

```rust
alle_studierenden.iter()
  .filter(|s| !s.hat_stipendium)
  .into_group_map_by(
      |s| &s.studiengang)
  .into_values()
  .map(|g| g.into_iter()
    .min_by(|a, b|
      a.schnitt.total_cmp(&b.schnitt))
    .unwrap())
  .sorted_by(|a, b|
    a.schnitt.total_cmp(&b.schnitt))
  .collect::<Vec<_>>();
```

In Rust ist & der Referenzoperator. D. h. in Zeile 2 wird eine Referenz auf den String für den Studiengang übergeben. Die Zielmethode kann diese Referenz lesen, aber nicht verändern.

Bewertung

Zero-Costs:
Rusts Pipeline kompiliert zu praktisch dem selben Maschinencode wie eine handgeschriebene `for`-Schleife — keine Heap-Allokationen, keine virtuellen Dispatches. Javas Pipeline erzeugt Overhead, der vom JIT-Compiler erst zur Laufzeit *teilweise* optimiert wird.

Expliziter Float-Vergleich:
Rust erzwingt `total_cmp` statt eines einfachen `<`-Vergleichs, weil Gleitkommazahlen `NaN` enthalten können. Javas `comparingDouble(…)` behandelt `NaN` stillschweigend; ein potenziell versteckter Bug.

Ownership vs. Runtime-Schutz:
`into_iter()` in Rust *konsumiert* die Gruppe — ein erneuter Zugriff ist ein Compile-Fehler. Javas Stream wirft erst zur Laufzeit eine `IllegalStateException` bei Mehrfachnutzung.

Erweiterbarkeit:
Rust benötigt kein Gatherer-Konzept. Einen neuen Iterator zu bauen erfordert nur die Implementierung von `next()` — alle ~75 Adapter-Methoden (`filter`, `map`, `take_while`, …) stehen dann automatisch zur Verfügung. Das `itertools`-Crate erweitert dies nahtlos über Rusts Trait-System.

Parallelisierung:

In Java genügt `.parallelStream()`; in Rust ersetzt man `iter()` durch `par_iter()` (via `rayon`-Crate). Der entscheidende Unterschied: Rusts Ownership-System garantiert *zur Compile-Zeit*, dass keine Data Races entstehen. Javas Fork/Join-Pool kann das nicht.
Es gilt zu beachten, dass der Fehler bei der nicht-assoziativen reduce-Funktion (Korrektheit bei Parallelisierung) auch in Rust möglich wäre, da er auf einem logischen Fehler (Vertragsverletzung) beruht, nicht auf einer Race Condition."

Vollständiges Beispiel in Rust

```rust
use itertools::Itertools;

#[derive(Debug)]
struct Student {
    hat_stipendium: bool,
    studiengang: String,
    schnitt: f64,
}

impl Student {
    fn new(hat_stipendium: bool, studiengang: &str, schnitt: f64) -> Self {
        Student {
            hat_stipendium,
            studiengang: studiengang.to_string(),
            schnitt,
        }
    }
}

fn main() {
    let alle_studierenden = vec![
        Student::new(false, "Informatik", 1.3),
        Student::new(false, "Informatik", 2.1),
        Student::new(true, "Informatik", 1.0),
        Student::new(false, "Informatik", 2.7),
        Student::new(false, "Informatik", 1.9),
        Student::new(false, "BWL", 1.7),
        Student::new(false, "BWL", 2.4),
        Student::new(false, "BWL", 1.1),
        Student::new(true, "BWL", 1.2),
        Student::new(false, "BWL", 3.0),
        Student::new(false, "Maschinenbau", 1.5),
        Student::new(false, "Maschinenbau", 2.3),
        Student::new(false, "Maschinenbau", 1.8),
        Student::new(true, "Maschinenbau", 1.1),
        Student::new(false, "Maschinenbau", 2.9),
        Student::new(false, "Medizin", 1.0),
        Student::new(false, "Medizin", 1.4),
        Student::new(false, "Medizin", 2.2),
        Student::new(true, "Medizin", 1.3),
        Student::new(false, "Medizin", 1.8),
        Student::new(false, "Jura", 1.6),
```

```
        Student::new(false, "Jura", 2.5),
        Student::new(false, "Jura", 1.2),
        Student::new(true, "Jura", 1.0),
        Student::new(false, "Jura", 3.1),

        Student::new(false, "Physik", 1.1),
        Student::new(false, "Physik", 2.0),
        Student::new(false, "Physik", 1.7),
        Student::new(true, "Physik", 1.2),
        Student::new(false, "Physik", 2.8),
    ];

    let empfehlungen: Vec<&Student> = alle_studierenden
        .iter()
        .filter(|s| !s.hat_stipendium)
        .into_group_map_by(|s| &s.studiengang)
        .into_values()
        .map(|gruppe| {
            gruppe
                .into_iter()
                .min_by(|a, b| a.schnitt.total_cmp(&b.schnitt))
                .unwrap()
        })
        .sorted_by(|a, b| a.schnitt.total_cmp(&b.schnitt))
        .collect();

    println!("Empfehlungsliste:");
    for s in &empfehlungen {
        println!("  {}: {}", s.studiengang, s.schnitt);
    }
}
```

Zum Ausführen des Codes kann Play Rust-Lang.org verwendet werden.

Java Streams vs. Rust Iteratoren — Konzeptioneller Vergleich

	Java Streams	Rust Iteratoren
Pipeline-Modell	deklarativ, verkettet	deklarativ, verkettet
Funktionen als Parameter	Lambdas, Method References	Closures (`|x|` ...)
Quelle unverän-dert	Ja	Ja
Verhalten bei Stream Mehrfach-nutzung	`IllegalStateException` zur Laufzeit	Compile-Fehler bei Mehrfachnutzung (Ownership)
Streams = Collec-tions?	Nein — separates Konzept	Nein — `iter()` erzeugt Iterator über Collection
Auswertung	Lazy (erst bei Terminal Op)	Lazy (erst bei `collect`, `sum`, `for_each`, ...)
Performance-Overhead	JVM + GC + JIT (zur Laufzeit)	Null — Zero-Cost Abstractions (zur Compile-Zeit)
Erweiterbarkeit	Collectors und Gatherers	Nur `next()` implementieren — ~75 Methoden gratis
Parallelisierung	Tief integriert (`.parallelStream()`)	Über `rayon`-Crate (`.par_iter()`)
Fehlerbehand-lung	`Optional` (umgehbar via `get()`)	`Option` / `Result` (vom Compiler erzwungen)
Typsicherheit	Compile-time (Generics mit Type Erasure)	Compile-time (Generics, Monomorphisierung)

Rusts Iterator-Modell ist Javas Streams oberflächlich sehr ähnlich: beide sind lazy, beide unterscheiden zwischen Intermediate- und Terminal-Operationen, beide unterstützen `filter`, `map`, `flat_map`, `collect` etc. Der fundamentale Unterschied liegt tiefer:

Zero-Cost Abstractions:
Rusts Compiler (via LLVM) optimiert Iterator-Pipelines zu praktisch demselben Maschinencode wie handgeschriebene `for`-Schleifen — keine Zwischenstrukturen, keine Heap-Allokationen, keine virtuellen Methodenaufrufe. In Java erzeugt die Stream-Pipeline dagegen Overhead durch Objekt-Allokationen, virtuelle Dispatches und GC-Zyklen, der erst durch den JIT-Compiler *teilweise* kompensiert wird.

Ownership:
Javas Schutz gegen Mehrfachnutzung eines Streams greift erst zur Laufzeit (`IllegalStateException`). Rusts Ownership-System verhindert denselben Fehler bereits zur Compile-Zeit — der Code kompiliert schlicht nicht.

Parallelisierung:

Java bietet mit `.parallelStream()` / `.parallel()` eine in die Standardbibliothek integrierte Lösung. Rust lagert dies in das `rayon`-Crate aus — dort genügt es, `iter()` durch `par_iter()` zu ersetzen. Beide Ansätze sind ergonomisch ähnlich, aber Rusts Ownership-System garantiert zur Compile-Zeit, dass keine Data Races entstehen können. Javas Fork/Join-Pool bietet diese Garantie nicht.

10. Fragen?

Java Stream API und Stream(-like) APIs anderer Sprachen!