Java 8 - Streams

# Inhaltsverzeichnis

[Inhaltsverzeichnis](#h.m734kxlrph7p)

[Abstract](#h.bxki54d8ldz7)

[Streams](#h.198aqsmpxjqv)

[Konzept](#h.y3jznz92cn4x)

[Umsetzung](#h.2i6kdlysfbbx)

[Beispiel 1 - Erzeugung und Benutzung eines Streams](#h.j9p6q2ixnwl1)

[Beispiel 2 - Erzeugung eines Streams mit einer Generator-Methode](#h.xb3q2enfgl6l)

[Beispiel 3 - Funktionalität von Short-Circuit-Methoden](#h.rkz66hnwyeux)

[Beispiel 4 - Funktionalität von parallelen Streams](#h.8x2564aamins)

[Methoden](#h.l10t88y0p2xl)

[Intermediate Methoden](#h.ke2foujauo45)

[map](#h.gzwvtl4z82se)

[map-Benchmark](#h.o48u7df6at7f)

[filter](#h.fb0t8hpzso4c)

[filter-Benchmark](#h.s5l74aa347r7)

[sorted](#h.2ouh4ik7tp5k)

[sorted-Benchmark](#h.1nrvwhdyqmat)

[Terminal Methoden](#h.rlk6y8xw43rp)

[forEach](#h.tqqswqnk856j)

[forEach-Benchmark](#h.hvse2zt22tps)

[reduce](#h.w2t48w2yba3h)

[reduce-Benchmark](#h.tldev48itp9f)

[Short-Circuit Methoden](#h.pkufk1ppubii)

[findFirst](#h.9i34l957khcr)

[findFirst-Benchmark](#h.lkgifmr013sq)

[limit](#h.ym9apanfsj07)

[limit-Benchmark](#h.d504vo2zf5va)

[Fazit](#h.ugjnmuk7f9ag)

[Quellen](#h.3y0usefvacmo)

[Anhang](#h.3rtehtwbf747)

[Benchmark Tool](#h.5aqadub9bnaw)

# Abstract

Zunächst werden die Grundideen und der Nutzen von Streams im Allgemeinen beleuchtet. Dabei werden Grundlagen und nützliche Eigenschaften mit Beispielen verdeutlicht. Im Hauptteil werden wichtige Methoden aus dem Interface Stream<T> in die Kategorien intermediate, terminal und short-circuit Methoden eingeteilt und mit, vor Java 8 nutzbaren, Funktionen in diversen Benchmarks vergleichen. Dabei stellt sich heraus, dass sequentielle Streams in den meisten Fällen keinen Performancegewinn bringen, dafür aber ein wichtiger Schritt in Richtung automatischer Parallelisierung sind.

# Streams

## Konzept

Streams sind Hüllen um Datenquellen wie Arrays oder Listen[[1]](#footnote-1). Diese erweitern die Datenquellen um Eigenschaften an denen es ihnen ansonsten fehlt. Folgend sind drei der nützlichsten neuen Eigenschaften aufgeführt[[2]](#footnote-2):

*Lazy evaluation*

Die Ausführung von Methoden wird so lange herausgezögert, bis eine Evaluation zwingend notwendig ist[[3]](#footnote-3).

Dieses Konzept findet zum Beispiel in funktionalen Programmiersprachen wie Haskell Verwendung.

*Automatische Parallelisierung*

Wenn ein Stream als parallel gekennzeichnet ist, werden Operationen auf ihm automatisch nebenläufig ausgeführt.

*Unendliche Streams*

Streams können aus einer Generatorfunktion erzeugt werden. Diese Funktion erzeugt während der Verarbeitung des Streams beliebig viele neue Einträge[[4]](#footnote-4).

## Umsetzung

Streams können aus vielen verschiedenen Datenquellen erzeugt werden. Beispiele sind beliebige Collections, native Arrays, Generatorfunktionen oder I/O-Kanäle[[5]](#footnote-5). Ein solches erzeugtes Stream-Objekt hält selber keine Daten, sondern ist nur eine Hülle um die jeweilige bestehende Datenquelle. Dabei können Streams die Datenquelle nicht verändern (nur Elemente filtern) und gewähren keinen indexierten Zugriff auf diese. Dadurch soll erreicht werden, dass Streams möglichst leichtgewichtig und performant sind[[6]](#footnote-6).

Diese Eigenschaften stehen sie im Kontrast zu Collections, die sich ihre eigenen Daten halten und verwalten[[7]](#footnote-7).

Auf einem Stream lässt sich eine beliebige Anzahl aneinander gereihter intermediate Methoden anwenden, die selber jeweils wieder einen Stream erzeugen. Diese intermediate Methoden nutzen in der Regel die, in Java 8 hinzugekommenden, Lambda-Ausdrücke. Gängige Beispiele hierfür sind map oder filter.

Ein Stream wird mit einer einzelnen terminal Methode abgeschlossen. Diese terminal Methode darf im Gegensatz zu den intermediate Methoden Seiteneffekte haben und einen Wert zurückgeben[[8]](#footnote-8). Durch die Benutzung von Streams mit Quelle, intermediate Methode und abschließender terminal Methode wird eine Stream-Pipeline erstellt, durch die die Elemente des Streams gereicht werden.[[9]](#footnote-9)

### Beispiel 1 - Erzeugung und Benutzung eines Streams[[10]](#footnote-10)

1. List<Integer> foo = new ArrayList<Integer>(Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10));
2. foo.stream().

filter(i -> i % 2 == 0).

forEach(System.out::println);

In diesem Beispiel wird in Zeile 1 zunächst eine Liste aus Integern mit dem Namen foo erzeugt. In Zeile 2 wird dann die Methode stream aus java.util.Collection benutzt um einen Stream aus Integern zu initialisieren.

Auf diesen Stream wird nun die Methode filter mit einem Lambda-Ausdruck aufgerufen, der für gerade Zahlen true und für ungerade Zahlen false ausgibt. Die Methode filter ist eine Intermediate-Methode, weil sie einen neuen Stream aus allen, nicht herausgefilterten, Elementen erzeugt.

Auf den gefilterten Stream wird dann die Methode forEach mit einem einfachen System.out::println aufgerufen, um die übrig gebliebenen Elemente anzuzeigen. Die Methode forEach ist eine Terminal-Methode, allerdings macht sie in diesem Fall keinen Gebrauch von Seiteneffekten.

Streams können aus Generatormethoden erzeugt werden. Hierzu lässt sich die statische default Methode generate aus dem Stream-Interface nutzen. Diese Generator Methode erzeugt einen unendlichen, sequentiellen, ungeordneten Stream der sich gut dazu eignet, eine große Menge beliebig erzeugbarer Elemente zu verarbeiten[[11]](#footnote-11).

### Beispiel 2 - Erzeugung eines Streams mit einer Generator-Methode[[12]](#footnote-12)

1. private static Integer generator() {
2. return i++;
3. }
4. Stream.generate(Generierung::generator)

In diesem Beispiel wird ein Stream mittels einer Generator-Methode erzeugt. Die Generator-Methode (Zeile 1-3) basiert auf einer Zählvariablen, die in jedem Schritt inkrementiert wird. In Zeile 4 wird die Methode generate aus Stream aufgerufen. Immer wenn der Stream nun ein Element benötigt ruft er die Generator-Methode auf.

Um Streams performanter zu machen werden viele Methoden (z.B. map oder filter) erst aufgerufen wenn von ihnen ein Element gefordert wird - sie setzen also lazy-evaluation um. Dazu kommen short-circuit Methoden, die oftmals nur wenige Elemente aus den vorherigen Streams benötigen um zu terminieren. Wenn nun eine short-circuit Methode nach einer Methode mit lazy-evaluation aufgerufen wird, so wird die Methode mit lazy-evaluation nur so viele Elemente zunächst verarbeiten und dann neu erzeugen, wie die short-circuit Methode benötigt.

### Beispiel 3 - Funktionalität von Short-Circuit-Methoden[[13]](#footnote-13)

Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10).

map(i -> {

System.out.println("Map has been called!");

return i;

}).

limit(3).

forEach(System.out::println);

Programm Output:

Map has been called!

1

Map has been called!

2

Map has been called!

3

In diesem Beispiel wird ein Stream aus den Zahlen 1 bis 10 erzeugt und die Methode map aufgerufen. Auf dem erzeugten Stream wird die Methode limit mit einer Beschränkung von 3 aufgerufen. Zuletzt werden die übrig gebliebenen per System.out::println ausgegeben.

Es entsteht der Eindruck, als würde map für alle 10 Elemente aufgerufen, von denen durch limit aber nur 3 benutzt werden. Wie allerdings am Output zu erkennen ist, wird map nur genau 3 mal aufgerufen.

Um Streams noch performanter zu machen, können diese auf sehr einfache Weise parallelisiert werden. Hierzu reicht es die intermediate-Methode parallel aufzurufen. Wenn diese Methode aufgerufen wurde laufen alle folgenden intermediate-Methoden und die abschließende terminal-Methode nebenläufig. Hierdurch sinkt die Laufzeit der Streams fast linear mit der Anzahl der verfügbaren Prozessoren[[14]](#footnote-14).

### Beispiel 4 - Funktionalität von parallelen Streams[[15]](#footnote-15)

1. Stream<Integer> serialStream = Stream.of(1, 2, 3);
2. Stream<Integer> parallelStream = Stream.of(1, 2, 3).parallel();
3. long serialStartTime = System.currentTimeMillis();
4. serialStream.forEach(ParallelStream::slowOperation);
5. long serialEndTime = System.currentTimeMillis();
6. long parallelStartTime = System.currentTimeMillis();
7. parallelStream.forEach(ParallelStream::slowOperation);
8. long parallelEndTime = System.currentTimeMillis();
9. System.out.println("Serial time: " + (serialEndTime - serialStartTime) + " milliseconds");
10. System.out.println("Parallel time: " + (parallelEndTime - parallelStartTime) + " milliseconds");

Output (getestet auf einem 4-Kern-Computer):

Serial time: 3033 milliseconds

Parallel time: 1058 milliseconds

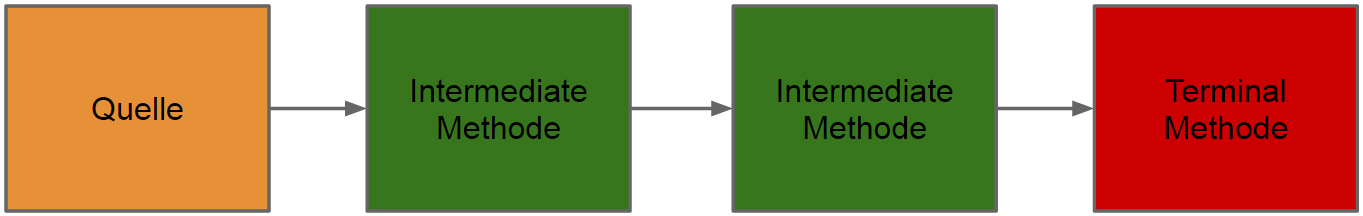
In Zeile 1 wird ein Serieller Stream mit den Zahlen 1-3 initialisiert. In Zeile 2 wird sein paralleles Gegenstück erzeugt. Der einzige Schritt, der notwendig ist um diesen Stream zu parallelisieren, ist es, die Methode parallel aufzurufen. In den Zeilen 4 und 7 wird sowohl auf den parallelen als auch auf den seriellen Stream die Methode slowOperation aufgerufen. Die einzige Aufgabe dieser Methode ist es, 1 Sekunde zu warten. In den Zeilen 3, 5, 6 und 8 werden Zeitstempel genommen, um die Laufzeit der jeweiligen Stream-Aufrufe zu messen.

Wie am Output zu sehen ist, braucht der parallele Stream nur ein drittel der Laufzeit des seriellen Streams, die Elemente werden also parallel verarbeitet.

# Methoden

Das Stream-Interface besitzt eine Vielzahl an Methoden, um mit diversen Streams zu arbeiten[[16]](#footnote-16). Diese Methoden lassen sich in zwei Kategorien einteilen: intermediate Methoden und terminal Methoden. Des Weiteren gibt es short-circuit Methoden welche sowohl intermediate als auch terminal Methoden sein können.

Zusammen lässt sich aus diesen Methoden eine Stream-Pipeline bauen, die mit einer Quelle beginnt, beliebig viele intermediate Methoden einthält und mit einer terminal Methode endet.



## Intermediate Methoden

Diese Methoden konsumieren die Elemente des aufrufenden Streams. Zusätzlich verarbeiten, verändern oder filtern sie diese und erzeugen aus den neuen Elementen einen neuen Stream. Diese Methoden können beliebig viel hintereinander gereiht werden.

Intermediate-Methoden können in zwei weitere Kategorien eingeteilt werden: stateless und stateful Methoden. Stateless Methoden wie filter und map setzen das Prinzip der lazy-evaluation um. Dieses Prinzip wird soweit umgesetzt, dass kein einziges Element aus der Quelle von der erzeugten “Stream-Pipeline” verarbeitet wird, bis die terminal Methode am Ende der Pipeline aufgerufen wird. Stateful Methoden wie distinct und sorted setzen das Prinzip der lazy-evaluation nur teilweise oder gar nicht um. Dies kann bei, beispielsweise sorted, so weit gehen, dass vom vorherigen Stream jedes Element verarbeitet sein muss, damit sorted anfangen kann Elemente an die nächste Methode weiter zu geben. Des Weiteren speichern stateful Methoden im Gegensatz zu stateless Methoden Informationen von vorher verarbeiteten Elementen zwischen, um bei folgenden Elementen darauf zurückgreifen zu können[[17]](#footnote-17).

Im Folgenden werden einige wichtige Intermediate-Methoden vorgestellt und Benchmarks[[18]](#footnote-18) für diese gezeigt. Anschließend werden diese mit Java7-Alternativen verglichen.

### 

### map

Die Methode map ist eine der grundlegenden Methoden für die Verarbeitung von Datensammlungen. Mit ihr wird auf alle Elemente einer Datensammlung eine Funktion ausgeführt. Der Rückgabewert dieser Methode ist die veränderte Datensammlung.

Bekannt ist diese Methode beispielsweise aus funktionalen Programmiersprachen wie Haskell[[19]](#footnote-19) oder dem map-reduce-Prinzip[[20]](#footnote-20).

Im Stream Interface Stream<T> ist die Methode mit folgender Signatur definiert:

<R> Stream<R> map(Function<? super T,? extends R> mapper)

Als Argument wird also eine Funktion übergeben, die die Elemente vom selben Typ <? super [T](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html)> aus denen der Stream besteht, konsumiert. Produzieren tut diese Methode Elemente vom Typ <? extends R>. Die Methode map produziert einen Stream<R>, auf dem wiederum eine beliebige intermediate oder terminal Methode aufgerufen werden kann.

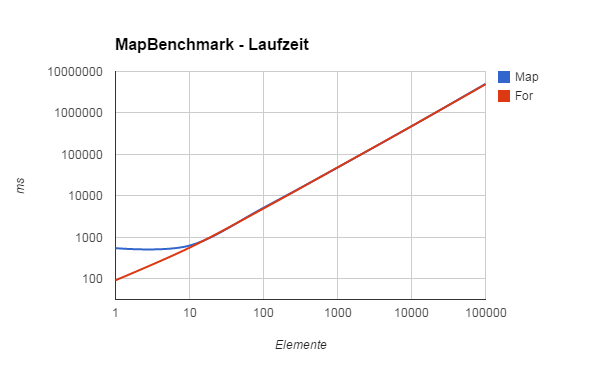
Die Methode map gehört zu den stateless Methoden.

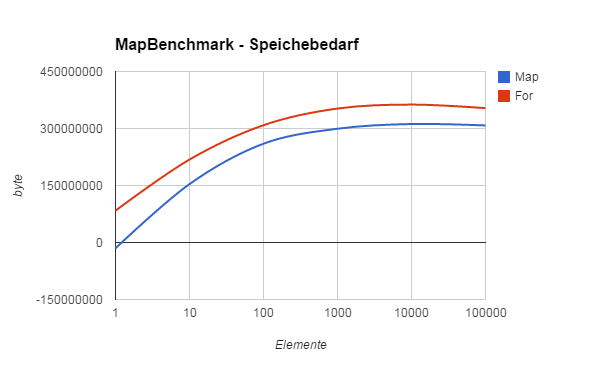
### map-Benchmark[[21]](#footnote-21)

1. list.stream().
2. map(s -> s.toUpperCase()).
3. forEach(s -> s.length());
4. for(String s : list) {
5. s.toUpperCase().length();
6. }

In Zeile 1-3 ist der Aufruf der map-Benchmark Methode zu erkennen. In den Zeilen 4-6 sind die Vergleichsmethode mit einer for-Schleife umgesetzt. Damit die Funktion von map für jedes Element aufgerufen wird ist es notwendig, eine terminal Methode auf den von map erzeugten Stream aufzurufen. Hierfür wurde forEach gewählt. Die durchgeführten Operationen beim map-Benchmark und for-Benchmark sind jedoch die selben.

**Ergebnisse[[22]](#footnote-22):**





Die Ergebnisse zeigen, dass der Stream, der für map erstellt werden muss, beim Erzeugen einen Overhead bei der Laufzeit hat. Diesen Overhead hat die Vorschleife nicht. Allerdings fällt der Overhead bei mehr verarbeiteten Elementen nicht mehr ins Gewicht und map und for haben in etwa die selbe Laufzeit.

Beim Speicherbedarf hat for dagegen einen fixen Speicheroverhead.

Insgesamt schneidet die map-Methode besser ab als die for-Schleife, da im Test ab 10 Elementen die Laufzeit ausgeglichen, der Speicherbedarf jedoch bei map geringer war.

### filter

Die Methode filter gehört zu den grundlegenden Methoden zum Verarbeiten von Datensammlungen. Sie lässt sich zum filtern von Elementen mit bestimmten Eigenschaften nutzen. Der Rückgabewert dieser Methode ist ein Stream aus Elementen, die der Bedingung genügen. Das Prinzip ist beispielsweise aus SQL mit der WHERE Klausel bekannt[[23]](#footnote-23).

Im Stream Interface Stream<T> ist die Methode mit folgender Signatur definiert:

Stream<T> filter(Predicate<? super [T](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html)> predicate)

Als Argument wird ein Prädikat mit dem Eingabetyp <? super [T](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html)>, bekannt aus Stream<T>, gefordert. Die Prädikatsfunktion muss einen boolean zurück geben. Die Methode filter produziert wiederum einen Stream<T>, also einen Stream vom selben Typ wie der Stream, von dem filter aufgerufen wurde.

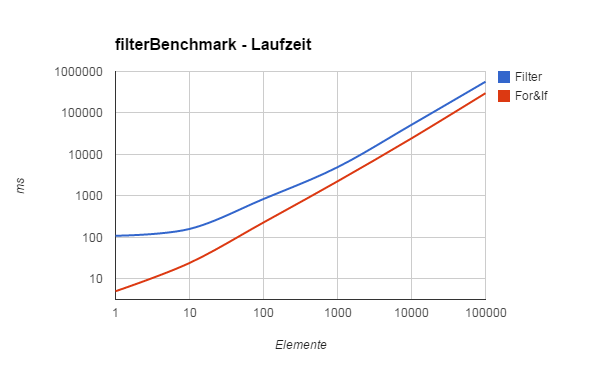
Die Methode filter gehört zu den stateless Methoden.

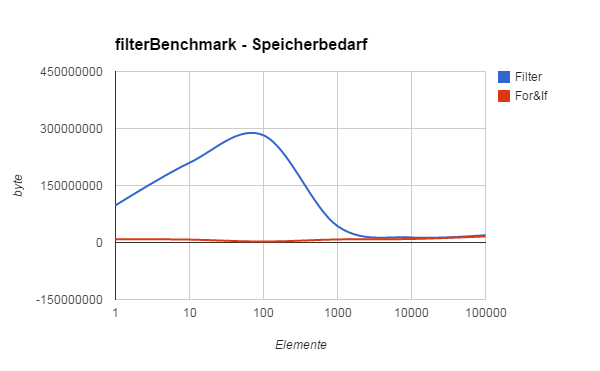
### filter-Benchmark[[24]](#footnote-24)

1. list.stream().
2. filter(s -> s.startsWith("a")).
3. forEach(s -> s.length());
4. for(String s : list) {
5. if(s.startsWith("a")) {
6. s.length();
7. }
8. }

Zeile 1-3 ist der Aufruf von filter, gefolgt von der terminal Methode forEach um zu gewährleisten, dass jedes Element aus dem Stream von filter bearbeitet wird. In filter wird geprüft, ob der erste Buchstabe des Eingabestrings ein ‘a’ ist, in forEach wird die Methode length auf die übrig gebliebenen Strings aufgerufen. In den Zeilen 4-8 werden die selben Aufrufe mit einer for-Schleife mit einer if-Abfrage umgesetzt.

**Ergebnisse[[25]](#footnote-25):**





Die Messergebnisse für Laufzeit zeigen bei wenigen Elementen zunächst wieder den bereits bekannten Laufzeit-Overhead für das Initialisieren des Streams. Mit der steigenden Anzahl verarbeiteter Elemente wird der Laufzeitunterschied zwischen filter und for&if kleiner, allerdings benötigt der Stream auch bei 100.000 Elementen noch knapp doppelt so viel Laufzeit wie for&if. Dieser große Unterschied ist möglicherweise darauf zurückzuführen, dass die if-Operation sehr atomar ist und daher vom JIT-Compiler leicht und frühzeitig in nativen Code umgewandelt wird.

Die Messergebnisse für Speicherbedarf zeigen zwischen 1 und 1000 Elementen einen sehr großen Unterschied, bei dem for&if wesentlich besser abschneidet als filter. Dies ist vermutlich darauf zurückzuführen, dass die Garbage-Collection bei filter erst sehr spät angefangen hat zu arbeiten. Zwischen 1.000 und 100.000 Elementen ist der Speicherverbrauch nicht mehr so unterschiedlich, dennoch schneiden for&if auch hier noch um ungefähr ein Drittel besser ab als filter.

Insgesamt schneiden for&if bedeutend besser ab als filter.

### sorted

Mit der Methode sorted lässt sich eine Datenquelle sortieren. Dies kann mit der unverarbeiteten Datenquelle, zwischen zwei Bearbeitungsschritten oder zuletzt vor der Ausgabe der verarbeiteten Datenquelle passieren. In der Regel kann eine Vergleichsfunktion angegeben werden, mit der die einzelnen Elemente untereinander verglichen werden. Generell ist es empfehlenswert, dass die Laufzeit der Vergleichsfunktion in O(1) liegt, da das Sortieren ansonsten sehr viel Laufzeit in Anspruch nehmen kann. Bekannt ist diese Funktion unter Anderem aus SQL mit der Klausel ORDER BY[[26]](#footnote-26).

Im Stream Interface Stream<T> ist die Methode mit den folgenden Signaturen definiert:

Stream<T> sorted()

Stream<T> sorted(Comparator<? super T> comparator)

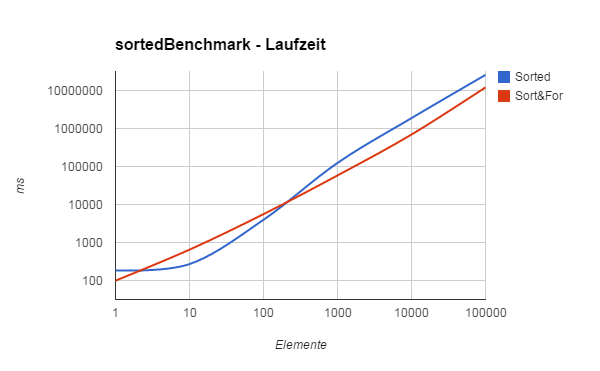
sorted ohne Argumente verlangt, dass T das Interface Comparable implementiert, ansonsten wird bei Aufruf der terminal Methode am Ende des Streams eine ClassCastException geworfen[[27]](#footnote-27). sorted mit einem Comparator, also einer Vergleichsfunktion als Argument fordert dies nicht, dafür muss die Vergleichsfunktion selber definiert oder zumindest übergeben werden. Die Vergleichsfunktion muss Elemente vom Typ <? super T> verarbeiten und einen Rückgabewert vom Typ int haben.

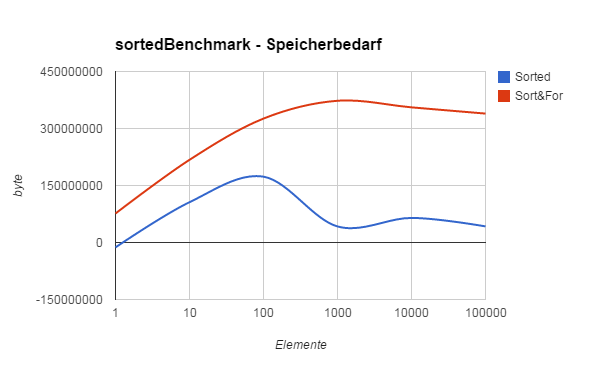
### sorted-Benchmark[[28]](#footnote-28)

1. list.stream().
2. sorted(String::compareTo).
3. forEach(s -> s.length());
4. Collections.sort(list, String::compareTo);
5. for(String s : list) {
6. s.toUpperCase().length();
7. }

Zeile 1-3 ist der Aufruf von sorted gefolgt von der terminal Methode forEach um zu gewährleisten, dass jedes Element aus dem Stream von sorted bearbeitet wird. Obwohl sorted stateful ist und schon beim Abruf von einem Element alle Elemente des vorhergehenden Streams bearbeitet, wurde hier forEach mit length als Operation als terminal Methode gewählt um eine bessere Vergleichbarkeit zwischen den anderen verschiedenen Benchmarks herzustellen. Die Zeilen 4-7 sind eine zu sorted möglichst ähnliche Abfolge von Operationen die unter Java 7 verfügbar waren. In Zeile 2 wird die Datenquelle mit Collections.sort komplett sortiert. In den Zeilen 3 und 4 wird mit einer for-Schleife über die sortierte Liste iteriert und die length Methode für jedes Element aufgerufen. Die Ergebnisse der beiden Benchmarks unterscheiden sich darin, dass Collection.sort ein Mutator ist, Stream.sorted dagegen nicht.

**Ergebnisse[[29]](#footnote-29):**





Die Messergebnisse für Laufzeit zeigen bei wenigen Elementen zunächst wieder den bereits bekannten Laufzeit-Overhead für das Initialisieren des Streams. Bei ca 100 und weniger Elementen hat sorted eine geringere Laufzeit als Sort&For. Allerdings ist der Laufzeitanstieg von Sort&For durchgängig linear, wogegen der von sorted nach gut 100 Elementen stark ansteigt und erst dann linear weiter steigt. Für sehr viele Elemente ist Sort&For besser als sorted, für wenige 100 Elemente ist hingegen sorted effizienter oder gleichwertig effizient.

Die Messergebnisse für Speicherbedarf zeigen, dass sorted mit Abstand weniger Speicher benötigt als Sort&For. Wenn zwischen 100 und 1000 verarbeiteten Elementen die Garbage-Collection angesprungen ist, wird sogar noch weniger Speicher benötigt. Der höhere und anfangs relativ lineare Anstieg des Speicherbedarfs von Sort&For ist vermutlich darauf zurück zu führen, dass eine vollständige sortierte Kopie der Liste erstellt wird, was sorted nicht in dem Umfang tun muss. Zudem sind in der Kopie der Liste sämtliche Meta-Informationen, die diese mit sich bringt, enthalten. Bei vielen Elementen springt allerdings auch hier die Garbage-Collection an und löscht die nicht mehr benötigte Ursprungsliste aus dem Arbeitsspeicher.

Zusammenfassend ist zwar der Speicherbedarf von sorted wesentlich geringer als der von Sort&For, dafür ist bei vielen Elementen die Laufzeit von Sort&For um ungefähr 40% kürzer als die von sorted. Da beim Sortieren aber die Laufzeit oftmals der entscheidene Faktor ist muss hier je nach Situation entschieden werden was die bessere Methode ist.

## 

## Terminal Methoden

Eine terminal Methode ist das Endstück in der Stream-Pipeline. Wenn eine terminal Methode aufgerufen wurde und terminiert ist gilt der Stream als verbraucht und kann nicht mehr benutzt werden. Wie auch intermediate Methoden unterstützen einige terminal Methoden lazy evaluation, wodurch unendliche Streams in vielen Fällen terminieren können. Terminal Methoden konsumieren in der Regel die Elemente des aufrufenden Streams. Nur die Ausnahmen iterator und spliterator tun dies nicht, sondern erlauben dem Nutzer das eigenständige Traversieren über die Elemente aus dem aufrufenden Stream. Terminal Methoden sind verantwortlich dafür, Anfragen an die Stream-Pipeline zu stellen, sodass von der Quelle aus Elemente bis zur terminal Methode durchgereicht werden.

Im Gegensatz zu intermediate Methoden können einige terminal Methoden wie forEach und peek Seiteneffekte haben. Diese sind allerdings wie alle Seiteneffekte mit Vorsicht zu benutzen und können besonders bei parallelen Streams ungewünschte Effekte haben, wenn die aufgerufenen Methoden oder Variablen der Elemente nicht threadsave sind.

Terminal Methoden haben keinen weiteren zu verarbeitenden Stream, sondern können einzelne Elemente oder Informationen über den Stream als Rückgabewert haben. Beispiele hierfür sind anyMatch, findFirst oder match[[30]](#footnote-30).

Im Folgenden werden einige wichtige terminal Methoden vorgestellt und Benchmarks[[31]](#footnote-31) für diese gezeigt. Anschließend werden sie mit Java7-Alternativen verglichen.

### forEach

Das Traversieren über alle Elemente einer Datenquelle um diese Elemente zu manipulieren gehört zu den grundlegenden Methoden der imperativen Programmierung. Dies wird mit der Methode forEach, oder auch nur for genannt, in den meisten imperativen und Objekt orientierten Sprachen umgesetzt. Auf diese Weise können Aktionen, die für jedes Element einer Datenquelle ausgeführt werden sollen, in wenigen Zeilen und unabhängig von der Menge der Elemente durchgeführt werden. Des Weiteren lassen sich mit dieser Hilfe beliebige Suchen leicht umsetzen. Bekannt ist das Prinzip beispielsweise aus C[[32]](#footnote-32) mit dem for-Statement oder früheren Java-Versionen.

Im Stream Interface Stream<T> ist die Methode mit der folgenden Signatur definiert:

void forEach(Consumer<? super T> action)

forEach hat als Argument eine Consumer-Funktion, welche Elemente vom Typ <? super T> verarbeitet. Als Rückgabewert hat forEach void, also nichts. Dafür kann die Consumer-Funktion, die als Argument an forEach übergeben wurde, Seiteneffekte haben.

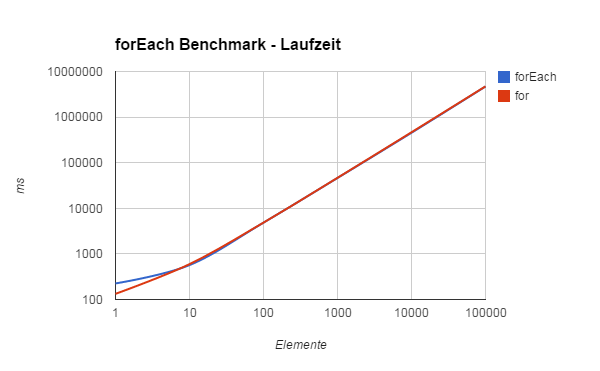
Bei der Verarbeitung von parallelen Streams ist forEach explizit nicht deterministisch. Dies würde eine Einhaltung der Reihenfolge der Elemente fordern, was den Vorteil der Parallelisierung zunichte machen würde. Das heißt, dass für jedes Element die Consumer-Funktion zu jedem möglichen Augenblick und in jedem möglichen Thread ausgeführt werden kann. Falls die Reihenfolge der Elemente entscheidend ist, kann auf die Funktion forEachOrdered zugegriffen werden, welche diese beibehält[[33]](#footnote-33).

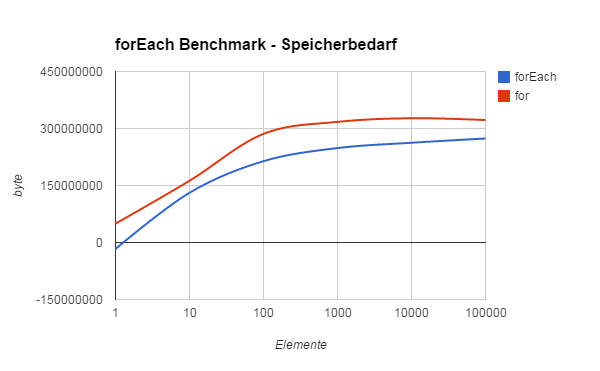
### forEach-Benchmark[[34]](#footnote-34)

1. list.stream().
2. forEach(s -> s.toUpperCase());
3. for(String s : list) {
4. s.toUpperCase();
5. }

Die Zeilen 1 und 2 sind der forEach-Benchmark. Dabei wird aus einer Liste fünfstelliger Strings ein Stream erzeugt, welcher mit der terminal Methode forEach verarbeitet wird. In forEach werden die Elemente mit toUpperCase bearbeitet. Die Zeilen 3-5 sind der Vergleichstest mit einer for-Schleife. Die Strings werden wie in forEach mit toUpperCase bearbeitet.

**Ergebnisse[[35]](#footnote-35):**





Die Messergebnisse für Laufzeit zeigen bei wenigen Elementen für forEach zunächst den bereits bekannten Laufzeit-Overhead für das Initialisieren des Streams. Bei einer Menge von ca 10 Elementen fällt dieser aber nicht mehr ins Gewicht und die Laufzeit steigt für beide Methoden linear an.

Beim Speicherbedarf hat for einen fixe Menge bytes mehr belegt als forEach. Da aber beide Methoden im Bereich zwischen 1.000 und 10.000 Elementen konstant werden, fällt dieser zusätzliche Speicherbedarf bei einer großen Menge Elemente nicht mehr ins Gewicht.

Insgesamt schneidet forEach aufgrund des geringeren Speicherbedarfs leicht besser ab als for. Allerdings sind die Ergebnisse nicht so verschieden, dass diese ein ausschlaggebendes Argument für eine der beiden Methoden darstellt.

### reduce

reduce wird dafür verwendet alle Elemente in einer Datenquelle zu einem einzelnen Element zu reduzieren. Dafür benötigt reduce eine Datenquelle und eine Akkumulationsfunktion, um die Elemente der Datenquelle miteinander zu akkumulieren. So können zum Beispiel viele verschiedene Zählungen zu einer Zählung reduziert werden.

Bekannt ist diese Methode beispielsweise aus funktionalen Programmiersprachen wie Haskell[[36]](#footnote-36) unter dem Namen fold oder aus dem map-reduce-Prinzip[[37]](#footnote-37).

Im Stream Interface Stream<T> ist die Methode mit den folgenden Signaturen definiert:

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

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

Die erste Signatur erwartet ein Identitätselement vom Typ T, welches als Basiselement für die Akkumulationsfunktion genutzt wird. Die Akkumulationsfunktion erwartet zwei Elemente vom Typ T. Als Rückgabewert hat diese Signatur ein Element vom Typ T.

Die zweite Signatur erwartet kein Identitätselement und hat ansonsten, wie auch die erste Signatur, eine Akkumulationsfunktion für Elemente vom Typ T als Argument. Als Rückgabewert hat die zweite Signatur im Gegensatz zur ersten ein Optional<T>. Dies ist darauf zurück zu führen, dass die erste Signatur durch ihr Identitätselement immer einen Rückgabewert hat, dies aber bei der zweiten Signatur nicht garantiert ist, weil der Stream auch leer sein kann.

Keine der Versionen von reduce garantiert eine Abarbeitungsreihenfolge, sodass die Akkumulationsfunktion das Assoziativgesetz einhalten muss[[38]](#footnote-38).

Im Stream<T>-Interface existiert eine speziellere Version von reduce namens collect. Diese ist dafür gedacht, mit Methoden aus dem Collectors-Interface aus Stream-Elementen neue Datenstrukturen wie Collections zu generieren.

Des Weiteren existiert die folgende Signatur, die ich hier nicht weiter betrachten werde:

<U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)

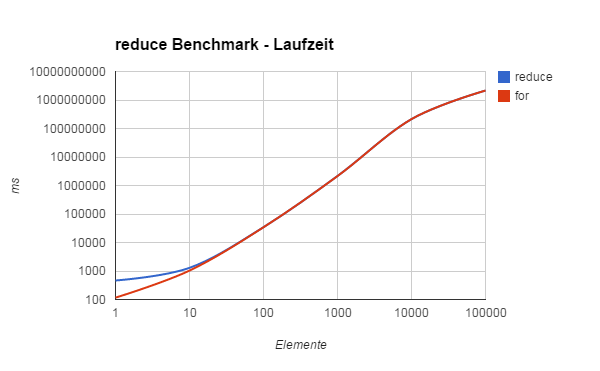
### reduce-Benchmark[[39]](#footnote-39)

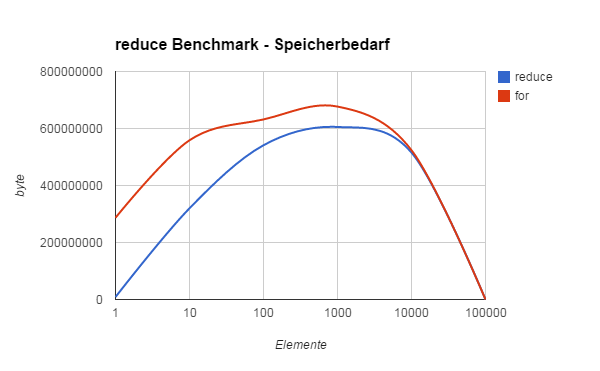
1. list.stream().
2. reduce("", (s1, s2) -> String.join(" ", s1, s2));
3. String akku = "";
4. for(String s : list) {
5. akku = String.join(" ", akku, s);
6. }

In den Zeilen 1 und 2 ist der reduce Benchmark. In Zeile 1 wird ein Stream aus einer Liste fünfstelliger Strings erzeugt. In der zweiten Zeile wird reduce mit einem leeren String als neutralem Element und String.join als Akkumulationsmethode aufgerufen.

In den Zeilen 3 bis 6 steht der Vergleichsbenchmark mit einem Akkumulator in Zeile 3 und einer for-Schleife in den Zeilen 4-6. Der Akkumulator wird mit dem neutralen Element initialisiert und dann in Zeile 5 mit String.join in jedem Schleifendurchlauf mit jedem neuen String-Element konkateniert.

**Ergebnisse[[40]](#footnote-40):**





Die Messergebnisse für Laufzeit zeigen bei wenigen Elementen für reduce zunächst den bereits bekannten Laufzeit-Overhead für das Initialisieren des Streams. Bei mehr als ca. 10 Elementen fällt dieser Overhead nicht mehr ins Gewicht und beide Methoden nehmen den selben Verlauf. Dabei ist zu beobachten, dass von 10.000 zu 100.000 Elementen die Laufzeit weniger stark ansteigt als bei weniger Elementen.

Bei der Speicherbedarfsmessung zeigt sich, dass die for-Schleife besonders bei wenigen verarbeiteten Elementen sehr viel mehr Speicher verbraucht als reduce. Bei steigender Elementzahl nähern sich die Messungen allerdings aneinander an.

Auffallend ist die Messungen für 100.000 Elemente bei beiden Methoden. In mehrfachen Durchläufen war das Ergebnis für den Speicherverbrauch jedes Mal, bei beiden Methoden, genau 2024. Dieses Ergebnis ist mit aller Wahrscheinlichkeit ein Messfehler und wird daher nicht weiter berücksichtigt.

Insgesamt hat reduce aufgrund der zwar selben Laufzeit wie for, aber des zum Teil wesentlich niedrigeren Speicherbedarfs, besser abgeschnitten als for. Eine Empfehlung kann hier nicht ausgesprochen werden, da die Messergebnisse möglicherweise nicht zuverlässig sind. Dies liegt vermutlich daran, dass der Speicherbedarf aufgrund der, bei vielen Elementen langen zu verarbeitenden Strings, zu groß für eine Messung wird.

## Short-Circuit Methoden

Short-circuit Methoden sorgen dafür, dass vorhergegangene intermediate Methoden mit lazy evaluation nur so viele Elemente verarbeiten, bis die short-circuit Methode terminieren kann[[41]](#footnote-41). Short-circuit Methoden zeichnen sich dadurch aus, dass sie oftmals nicht alle oder nur ein einzelnes Element aus der vorhergegangenen intermediate Methode benötigen.

Eine short-circuit Methode kann sowohl zu den terminal, als auch zu den intermediate Methoden gehören. Beispiele hierfür sind limit und skip aus den intermediate Methoden und findFirst und allMatch aus den terminal Methoden[[42]](#footnote-42).

Im Folgenden werden einige wichtige terminal Methoden vorgestellt und Benchmarks[[43]](#footnote-43) für diese gezeigt. Anschließend werden diese mit Java7-Alternativen verglichen.

### findFirst

findFirst ist dazu gedacht, das erste aufgefundene Element aus einer Datenquelle zurückzugeben. Vor eine solche Methode wird in der Regel eine Methode zum aussortieren, wie im Kapitel filter beschrieben, geschaltet. Wenn nur wenige Elemente aus der Datenquelle der Bedingung von filter entsprechen und dementsprechend möglicherweise viele Elemente verarbeitet werden, bis eines zu findFirst durchgereicht wird. Hier kann es sinnvoll sein, das vorgeschaltete filter zu parallelisieren.

Bekannt ist findFirst zum Beispiel aus SQL mit dem FIRST Statement[[44]](#footnote-44) oder Haskell mit der Funktion find[[45]](#footnote-45).

Im Stream Interface Stream<T> ist die Methode mit der folgenden Signatur definiert:

Optional<T> findFirst()

Die Methode erwartet kein Argument und hat als Rückgabewert einen Optional<T> für den Fall, dass die vorhergegangene intermediate Methode kein Element liefern kann. Wenn der Stream auf dem findFirst aufgerufen wurde keine Ordnung hat, kann ein beliebiges Element aus diesem Stream im zurückgegebenen Optional<T> enthalten sein[[46]](#footnote-46). Da findFirst, wie der Name sagt, nur ein Element benötigt um zu terminieren, werden hierdurch in vielen Fällen viele unnötige Rechenoperationen übersprungen.

### findFirst-Benchmark[[47]](#footnote-47)

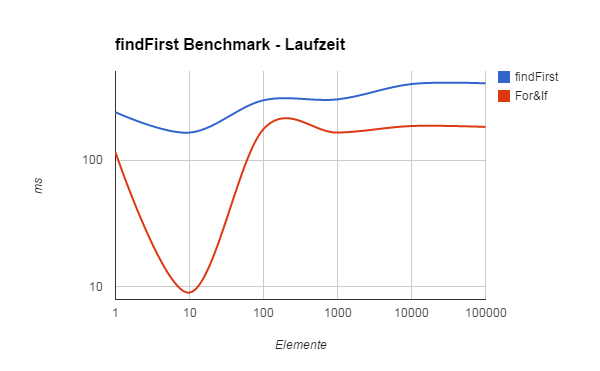
1. Optional<String> o = list.stream().
2. filter(s -> s.startsWith("sound")).
3. findFirst();
4. if(o.isPresent()) {
5. o.get().toUpperCase();
6. }
7. for(String s : list) {
8. if(s.startsWith("sound")) {
9. s.toUpperCase();
10. return;
11. }
12. }

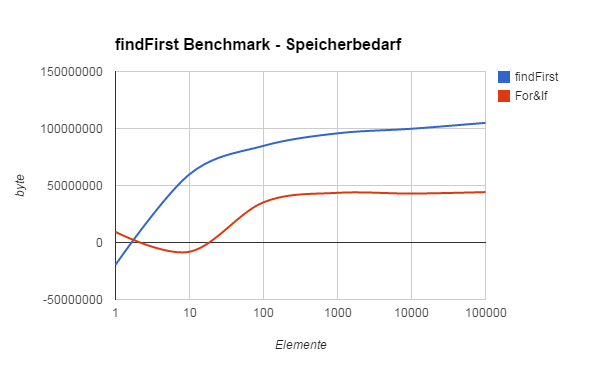
In den Zeilen 1-6 ist der Benchmark für findFirst. Dabei wird in den Zeilen 1-3 ein Optional o mittels filter und findFirst aus einer Liste mit einem Stream geholt. Das gesuchte Element befindet sich an Stelle 20 der Liste. In Zeile 4 wird geprüft, ob o ein Element enthält, also ob das betreffende Wort gefunden wurde. Wenn ja wird toUpperCase darauf angewandt.

In den Zeilen 7-12 ist der Vergleichsbenchmark mit einer for-Schleife und einem if zum filtern. Wenn ein Element gefunden wurde was dem if genügt dann wird, wie beim findFirst, toUpperCase darauf angewandt.

Obwohl toUpperCase nicht direkt mit dem Auffinden eines Elements in einem Stream oder einer Liste zu tun hat, ist es diese Methode hier im Benchmark zu verwenden. Wenn Anlass besteht ein Element aus einer Datenquelle zu suchen, dann wird dieses auch benutzt werden. Da dies im Fall des for&if einfacher geht als bei findFirst, weil kein Optional mehr geprüft werden muss, sollte die Benutzung des gefundenen Elementes Teil eines praxisrelevanten Tests sein.

**Ergebnisse[[48]](#footnote-48):**





Die Laufzeitergebnisse zeigen, dass for&if um eine zumeist fixe Zeit schneller laufen als findFirst. Diese fixe Zeit ist vermutlich dem geschuldet, dass für findFirst zunächst ein Stream initialisiert werden und nach dem Finden des Elements noch ein Optional bearbeitet werden muss. Nach 100 Elementen bleibt die Laufzeit bei beiden Methoden relativ konstant. Dies liegt daran, dass das zu findende Element an Stelle 20 der Datenquelle zu finden ist. Bei einer Liste von 10 Elementen zeigen beide Methoden eine sehr kurze Laufzeit, wobei die von for&if kürzer ausfällt. Dass diese bei for&if extrem gering ist ist vermutlich einem Messfehler zuzuschreiben.

Die Speicherbedarfsmessung zeigt, dass findFirst eine fixe Menge mehr Speicher benötigt als for&if. Diese ist vermutlich dem zu speichernden Optional zuzuschreiben. Ansonsten fällt auf, dass der Speicherbedarf für findFirst mit einer steigenden Anzahl Elemente weiter ansteigt, obwohl nur 20 Elemente der Datenquelle verarbeitet werden müssen. Dies ist bei for&if nicht der Fall.

Insgesamt schneiden for&if sowohl bei Speicherbedarf als auch Laufzeitmessung besser ab als findFirst. Dies liegt allerdings daran, dass nur ein Bruchteil der Elemente verarbeitet werden muss und so ein Großteil der Laufzeit und des Speicherbedarfs durch den Overhead von Streams oder Optionals belegt wird.

### limit

Diese Methode limitiert die Anzahl der weitergegebenen Elemente auf einen fixen Wert. Dies findet dann Einsatz, wenn zum Beispiel nur wenige Elemente für eine Berechnung gebraucht werden oder die benutzte Datenquelle zu groß ist um alle Elemente aus ihr zu verarbeiten.

Bekannt ist diese Methode besonders aus Datenbanken wie MongoDB mit der Aggregation limit[[49]](#footnote-49).

Im Stream Interface Stream<T> ist die Methode mit der folgenden Signatur definiert:

Stream<T> limit(long maxSize)

limit erwartet als Argument einen long maxSize, der die maximale Anzahl der weiter zu gebenden Elemente bestimmt. Da limit eine intermediate Methode ist gibt sie einen Stream<T> für das nächste Glied in der Stream-Pipeline zurück. In einem sequentiellen Stream garantiert limit, dass nicht nur irgendwelche Elemente weiter geben werden, sondern die ersten maxSize Elemente aus dem vorherigen Stream. Dies kann bei einem parallelen geordneten Stream zu großen Performanceeinbußen führen. Wenn die Ordnung der Elemente hier nicht mehr notwendig ist empfiehlt es sich, mit der Methode unordered aus BaseStream den Stream ungeordnet zu verwenden[[50]](#footnote-50).

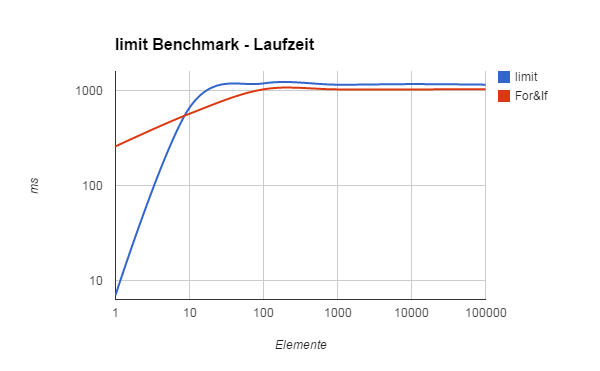
### limit-Benchmark[[51]](#footnote-51)

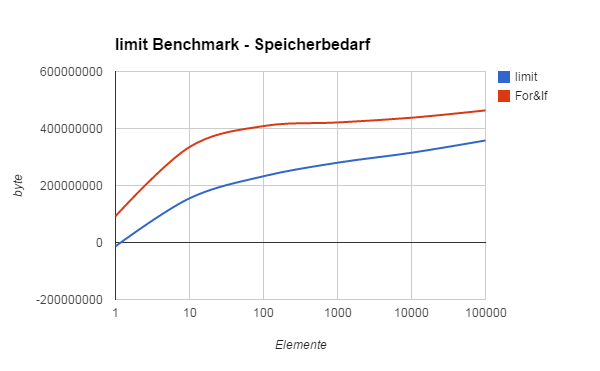
1. list.stream().
2. limit(20).
3. forEach(s -> s.toUpperCase());
4. long counter = 0;
5. for(String s : list) {
6. if(counter >= 20) {
7. return;
8. }
9. s.toUpperCase();
10. counter++;
11. }

In den Zeilen 1-3 befindet sich der Benchmark für limit. In der ersten Zeile wird ein Stream aus einer Liste fünfstelliger Strings erzeugt. In Zeile 2 wird die Anzahl der Elemente auf 20 beschränkt und in Zeile 3 werden diese Elemente mit toUpperCase verarbeitet.

In den Zeilen 4-11 ist der Benchmark für die Vergleichsmethode mit for&if. In Zeile 4 wird ein Zähler initialisiert und in Zeile 10 in jedem Durchlauf der Schleife in den Zeilen 5-11 inkrementiert. In Zeile 6 wird geprüft, ob schon 20 Elemente verarbeitet wurden, und im Fall von 20 abgeschlossenen Inkrementationen mit return terminiert.

**Ergebnisse[[52]](#footnote-52):**





Die Ergebnisse für Laufzeit zeigen, dass beide Methoden eine ähnliche Laufzeit haben. Aus dem Muster fällt das Messergebnis für 1 Element bei for&if, welches sehr hoch ausfällt. Dies ist vermutlich eine Messungenauigkeit. Ansonsten läuft for&if nur unwesentlich schneller als limit. Bei 100 Elementen steigt die Laufzeit nicht mehr weiter an. Dies ist auf die Begrenzung von maximal 20 verarbeiteten Elementen zurückzuführen.

Die Ergebnisse für den Speicherbedarf zeigen, dass for&if eine, in etwa konstante Menge mehr Speicherbedarf als limit, hat. Auffällig ist, dass der Speicherbedarf bei beiden Methoden auch nach 100 Elementen noch weiter ansteigt, obwohl die Begrenzung bei 20 Elementen liegt.

Insgesamt schneidet limit leicht besser ab als for&if. Dies liegt daran, dass der Speicherbedarf von limit niedriger ist als der von for&if, welcher sich aber bei einer größeren Anzahl von limit und for&if zugelassener Elemente relativieren sollte. In der Laufzeit unterscheiden sich limit und for&if nur bei sehr wenigen Elementen.

# Fazit

Mit Streams scheint versucht worden zu sein, positive Eigenschaften aus vielen verschiedenen Sprachen in Java zu vereinen. Zum einen wird die Syntax durch Lambda-Methoden und vielen vorgegebenen, oft benötigten, Methoden wie limit oder map übersichtlicher. Zum anderen werden Seiteneffekte bei intermediate Methoden komplett verboten, sodass eine Parallelisierung, ähnlich wie in funktionalen Sprachen, relativ einfach durchzuführen ist.

Lazy evaluation scheint eine sehr gute Methode zu sein, um Performanceeinbußen durch Programmierfehler zu vermeiden. Beispielsweise ist es mit limit nicht mehr möglich, innerhalb einer Stream-Pipeline unnötigerweise eine Datenquelle erst komplett zu verarbeiten und dann nur die ersten 20 Elemente daraus zu benutzen. Wenn allerdings mit Funktionen, die vor Java 8 verfügbar waren, sauber gearbeitet und keine Rechenzeit verschwendet wird, bringt dies keinen Vorteil. Im Gegenteil kostet es Laufzeit durch das Initialisieren des Streams.

Insgesamt wird empfohlen Streams zu benutzen, denn Streams werden vermutlich in den nächsten Java-Versionen weiter verbessert. Des Weiteren kann die Möglichkeit der automatischen Parallelisierung, die hier nicht genauer betrachtet wurde, die Laufzeit fast linear mit steigender Prozessorzahl verringern. Dies wäre Inhalt einer weiteren wissenschaftlichen Arbeit.

# Quellen

[MHS1] Hall, Marty: “Streams in Java8: Part1”. <http://www.java-programming.info/tutorial/pdf/java/java-8/Java-8-Streams-Part-1.pdf> (27.12.2014).

[MHS2] Hall, Marty: “Streams in Java8: Part2”. [http://www.java-programming.info/tutorial/pdf/java/java-8/Java-8-Streams-Part-2.pdf](http://www.java-programming.info/tutorial/pdf/java/java-8/Java-8-Streams-Part-1.pdf) (27.12.2014).

[CANH] Giersch, Steffen: Codeanhang an diese Hausarbeit. Am besten zu Öffnen als Eclipse-Projekt. Referenzen werden aus dem Ordner /src angegeben.

[DANH] Giersch, Steffen: Ordner in dem sich dieses Dokument befindet.

[DGMR] Dean, Jeffrey & Ghemawat, Sanjay: “MapReduce: Simplified Data Processing on Large Clusters”. OSDI, 2004.

[J8API] Oracle: “Java 8 API”. <http://docs.oracle.com/javase/8/docs/api/> (27.12.2014).

[HAPI] Mitchell, Neil: “Hoogle” <https://www.haskell.org/hoogle/> (02.01.2015).

[SAPI] Oracle & Apache Software Foundation “Apache Derby”, Kapitel “SQL language reference”. <http://docs.oracle.com/javadb/10.8.3.0/ref/> (08.01.2015).

[MAPI] MongoDB. Inc: “The MongoDB 2.6 Manual”, Kapitel “Reference”. <http://docs.mongodb.org/manual/reference/> (17.01.2015).

[ISO9899] ISO/IEC 9899: “Programming Languages - C” <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf> (14.01.2015)

[SGBW] Knuth, Donald E.: “SGB-Words”. <http://www-cs-faculty.stanford.edu/~uno/sgb.html> (02.01.2015).

# Anhang

## Benchmark Tool

Für das Benchmarking wurde das für die zweite Praktikumsaufgabe erweiterte Benchmarktool auf Basis von Kent Becks Micro-Framework verwendet. Zu finden ist dieses Tool in [CANH] im Package benchmarkTool.

Das generelle Vorgehen dieses Benchmark-Tools ist zunächst die Laufzeit und den Speicherbedarf für das Aufrufen einer leeren Methode zu messen. Auf diese Weise soll der Overhead der zu benchmarkenden Methode bestimmt werden. Daraufhin wird die zu benchmarkende Methode aufgerufen und auch hier Laufzeit und Speicherbedarf gemessen. Zwischen je zwei Methodenaufrufen wird die Java Garbage-Collection aufgerufen und eine Sekunde gewartet um für jede Messung einen möglichst gleichmäßigen Startzustand in der JVM zu schaffen. Um auf dem PC der den Benchmark durchführt bei jedem Benchmark möglichst vergleichbare Startbedingungen zu haben wurde dieser vor jedem Durchlauf neu gestartet.

Die Elemente der zu bearbeitenden Datenquellen sind Strings aus Donald E. Knuths SGB-Words[[53]](#footnote-53) (5757 verschiedene 5-Stellige Wörter) um gleichlange und gleichzeitig praxisrelevante Eingabelemente zu verarbeiten.

Das Ergebnis des Benchmarkings sind die vom Overhead bereinigte Laufzeit und der vom Overhead bereinigte Speicherbedarf für verschieden große bearbeitete Datenquellen.

1. vgl. [MHS1] Folie 7 [↑](#footnote-ref-1)
2. vgl. [MHS1] Folie 6 [↑](#footnote-ref-2)
3. vgl. [MHS1] Folie 43 [↑](#footnote-ref-3)
4. vgl. [MHS1] Folie 9 [↑](#footnote-ref-4)
5. vgl. [J8API] java.util.stream.Stream<T> [↑](#footnote-ref-5)
6. vgl. [MHS1] Folie 6-8 [↑](#footnote-ref-6)
7. vgl. [MHS1] Folie 11 [↑](#footnote-ref-7)
8. vgl. [MHS1] Folie 44 [↑](#footnote-ref-8)
9. vgl. [J8API] java.util.stream.Stream<T> [↑](#footnote-ref-9)
10. vgl. [CANH] vgl. beispiele.Generierung [↑](#footnote-ref-10)
11. vgl. [J8API] java.util.stream.Stream<T> [↑](#footnote-ref-11)
12. vgl. [CANH] beispiele.Generierung [↑](#footnote-ref-12)
13. [CANH] beispiele.ShortCircuitMethoden [↑](#footnote-ref-13)
14. vgl. [MHS2] Folie 28 [↑](#footnote-ref-14)
15. [CANH] beispiele.ParallelStream [↑](#footnote-ref-15)
16. Hier werden nur einige Methoden vorgestellt. Für alle Methoden siehe [J8API] java.util.stream.Stream<T> [↑](#footnote-ref-16)
17. vgl. [J8API] java.util.stream [↑](#footnote-ref-17)
18. Alle Benchmarks wurden mit dem Programm [Benchmark Tool](#h.5aqadub9bnaw) durchgeführt. (s. Anhang) [↑](#footnote-ref-18)
19. [HAPI] map [↑](#footnote-ref-19)
20. [DGMR] [↑](#footnote-ref-20)
21. [CANH] benchmarks.intermediateMethoden.BenchmarkMap [↑](#footnote-ref-21)
22. [DANH] Originaloutput als csv-Datei namens “mapBenchmarkOutput.csv” [↑](#footnote-ref-22)
23. [SAPI] SQL clauses -> WHERE clause [↑](#footnote-ref-23)
24. [CANH] benchmarks.intermediateMethoden.BenchmarkFilter [↑](#footnote-ref-24)
25. [DANH] Originaloutput als csv-Datei namens “filterBenchmarkOutput.csv” [↑](#footnote-ref-25)
26. [SAPI] vgl. SQL clauses -> ORDER BY clause [↑](#footnote-ref-26)
27. [J8API] vgl. java.util.stream.Stream<T> [↑](#footnote-ref-27)
28. [CANH] benchmarks.intermediateMethoden.BenchmarkSorted [↑](#footnote-ref-28)
29. [DANH] Originaloutput als csv-Datei namens “sortedBenchmarkOutput.csv” [↑](#footnote-ref-29)
30. vgl. [J8API] java.util.stream [↑](#footnote-ref-30)
31. Alle Benchmarks wurden mit dem Programm [Benchmark Tool](#h.5aqadub9bnaw) durchgeführt. (s. Anhang) [↑](#footnote-ref-31)
32. vgl. [ISO9899] 6.8.5.3 “The for statement” [↑](#footnote-ref-32)
33. vgl. [JAPI] java.util.stream.Stream<T> [↑](#footnote-ref-33)
34. vgl. [CANH] benchmarks.terminalMethoden.BenchmarkForEach [↑](#footnote-ref-34)
35. [DANH] Originaloutput als csv-Datei namens “forEachBenchmarkOutput.csv” [↑](#footnote-ref-35)
36. [HAPI] fold [↑](#footnote-ref-36)
37. [DGMR] [↑](#footnote-ref-37)
38. [J8API] vgl. java.util.stream.Stream<T> [↑](#footnote-ref-38)
39. vgl. [CANH] benchmarks.terminalMethoden.BenchmarkReduce [↑](#footnote-ref-39)
40. [DANH] Originaloutput als csv-Datei namens “reduceBenchmarkOutput.csv” [↑](#footnote-ref-40)
41. vgl. [MHS1] Folie 44 [↑](#footnote-ref-41)
42. vgl. [MHS1] Folie 44 [↑](#footnote-ref-42)
43. Alle Benchmarks wurden mit dem Programm [Benchmark Tool](#h.5aqadub9bnaw) durchgeführt. (s. Anhang) [↑](#footnote-ref-43)
44. vgl. [SAPI] SQL clauses -> The result offset and fetch first clauses [↑](#footnote-ref-44)
45. vgl. [HAPI] find [↑](#footnote-ref-45)
46. vgl. [J8API] java.util.stream.Stream<T> [↑](#footnote-ref-46)
47. vgl. [CANH] benchmarks.shortCircuitMethoden.BenchmarkFindFirst [↑](#footnote-ref-47)
48. [DANH] Originaloutput als csv-Datei namens “findFirstBenchmarkOutput.csv” [↑](#footnote-ref-48)
49. vgl. [MAPI] operator -> aggregation -> limit [↑](#footnote-ref-49)
50. vgl. [J8API] java.util.stream.Stream<T> [↑](#footnote-ref-50)
51. vgl. [CANH] benchmarks.shortCircuitMethoden.BenchmarkLimit [↑](#footnote-ref-51)
52. [DANH] Originaloutput als csv-Datei namens “limitBenchmarkOutput.csv” [↑](#footnote-ref-52)
53. [SGBW] [↑](#footnote-ref-53)