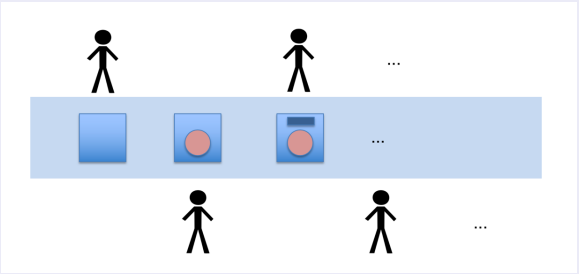
Streams

Motivation)

Fließband (pipeline

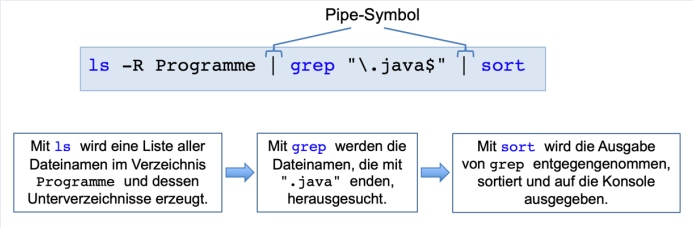


- Produktion einer Waschmaschine an einem Fließband
- Personen arbeiten parallel mit jeweils anderer Tätigkeit

Motivation

Unix Pipes

- Der Pipe-Mechanismus wurde Anfang der 70er-Jahre in Unix eingeführt
- Er gestattet den Austausch von Daten zwischen zwei Programmen
- Damit läßt sich eine Kette von Programmen zusammenbauen:
  jedes Programm nimmt Daten entgegen, verarbeitet sie und reich seine Ausgaben an das nächste Programm weiter(Pipeline-Verarbeitung)
- Die Programme laufen dabei (soweit möglich) parallel



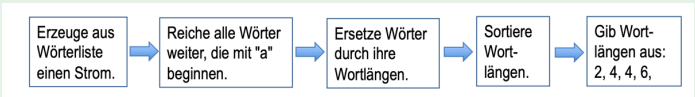
Ziel in Java

- Zu den größten Vorzügen von Streams zählt die Möglichkeit, die Code-Ausführung intern zu optimieren, beispielsweise durch die parallele Ausführung von Teilaufgaben
- Streams könnten also zu effizienterem Code führen, ohne dass sich die Programmierer um die Entwicklung nebenläufiger Algorithmen kümmern müssen
- Stream-Schnittstelle zur Verarbeitung von Collection-Elements durch Lambda-Ausdrücke

Streams seit Java 8 . . .

- Ströme sind eine (eventuell unendlich lange) Folge von Datenströmen
- Die Datenobjekte eines Stroms werden von Methoden verarbeitet und können dann zur nächsten Methode weitergereicht werden (Pipeline-Verarbeitung)
- Das Stromkonzept von Java hat damit große Ähnlichkeiten zu den Unix-Pipes

```
List<String> wordList = Arrays.asList("achten", "auch", "zum", "an", "bei", "aber", "vor")
wordList.stream()
  .filter(s -> s.startsWith("a"))
  .mapToInt(s -> s.length())
  .sorted()
  .forEach(n -> System.out.println(n + ","));
System.out.println("");
```



Aufbau eines Streams . . .

- Mit einer `initialen` Operation wird ein Strom erzeugt.
- Mit (einer oder mehreren) `intermediären` Operationen werden Ströme transformiert. Rückgabewert ist wieder ein Strom.
- Mit einer `terminale` Operation wird der Strom abgeschlossen. Terminale Operationen liefern ein Resultat (aber keinen Strom) zurück oder haben keinen Rückgabewert und eventuell einen Seiteneffekt.
- Intermediäre und terminale Operationen sind im Paket `java.util.stream` festgelegt

Verzögerte Generierung der Streams

- Ströme werden nie komplett im Voraus generiert. Ströme können prinzipiell unendlich lang werden
- Es werden nur solange Daten für den Strom generiert, wie die terminale Operation noch Daten benötigt. Der Strom wir verzögert generiert (lazy evaluation)

Verzögerte Generierung der Streams

```
new Random().ints()
  .map(n -> Math.abs(n)%1000)
  .peek(System.out::println)
  .anyMatch(n -> 10 <= n && n < 20)
```

- Die initiale Operation `ints` der Klasse `Random` erzeugt prinzipiell unendlichen Strom von Zufallszahlen
- Die intermediäre `map`-Operation transformiert die Zufallszahlen in das Intervall [0,1000)
- Die intermediäre `peek`-Operation gibt jede Zahl aus und reicht sie weiter
- Die terminale Operation `anyMatch` bricht mit Rückgabe von true ab, sobald eine Zahl im Intervall [10, 20) liegt

Ströme können aus zahlreichen „Datenbehältern der Java API“ erzeugt werden

Beispiele

- `Collection.stream()`: Instanz-Methode zum Erzeugen eines sequentiellen Stroms
- `Collection.parallelStream()`: Instanz-Methode zum Erzeugen eines parallelen Stroms
- `Arrays.stream(T[] a)`: statische Methode zum Erzeugen eines Stroms aus dem Feld `a`.
- `BufferedReader.lines()`: Instanz-Methode liefert einen Strom bestehend aus Zeilen

Ströme können aus zahlreichen „Datenbehältern der Java API “erzeugt werden

```
List<String> wordList =Arrays.asList("achten", "auch", "zum", "an", "bei", "aber", "vor");
Stream<String> s1 = wordList.stream(); // Strom mit den Strings aus wordList

int[] a = new int[]{1,2,3,4,5};
IntStream s0 = Arrays.stream(a);

BufferedReader in = new BufferedReader(new FileReader("test.txt"));
Stream<String> s2 = in.lines(); // Strom mit Zeilen der Datei test.txt
```

Initiale Stream-Operationen aus Paket Stream

Beispiel für statische Fabrik-Methoden

- `empty()`: Leerer Strom
- `of(...)`: Strom mit vorgegebenen Elementen
- `generate(s)`: Generiere Strom durch wiederhotels Aufrufen von `s`: `s(), s(), s(), ...`
- `iterate(a,f)`: Generiere Strom durch Iteration: `a, f(a), f(f(a)), ...`
- `range(a,b)`: Generiere Integer-Strom von `a` einschließlich bis `b` ausschließlich

Initiale Stream-Operationen aus Paket Stream

```
IntStream s3 = IntStream.of(1, 2, 3, 4); // Strom mit den Zahlen 1, 2, 3, 4

IntStream s4 = IntStream.iterate(1, x -> 2*x); // Unendlicher Strom mit allen 2er Potenzen

// Unendlicher Strom mit sin(x), wobei x eine Zufallszahl aus [0, 1) ist
DoubleStream s5 = DoubleStream.generate( )( -> Math.random()) );

IntStream s6 = IntStream.range(1, 10); // Strom mit int-Zahlen von 1 bis 9 (einschl.)
```

Initiale Stream-Operationen aus Klasse Random

- `doubles()`: Strom mit unendlich vielen zufälligen double-Zahlen aus [0,1)
- `ints()`: Strom mit unendlich vielen zufälligen int-Zahlen

```
IntStream s1 = new Random().ints();
DoubleStream s2 = new Random().doubles();
```

Intermediäre Stream-Operationen

- Intermediäre Operationen transformieren Ströme
- Rückgabewert ist wieder ein Strom
- Damit ist die typische Verkettung von mehreren Operationen möglich

```
strom.op1(...).op2(...)...opN();
```

Intermediäre Stream-Operationen

- `filter(pred)`: lasse nur Elemente x im Strom, für die dsaPrädikat pred(x) zutrifft
- `map(f)`: ersetze jedes Element x im Strom durch f(x)
- `flatMap(f)`: ersetze jedes Element x im Strom durch einen von f(x) erzeugten Strom
- `peek(action)`: führe für jede Methode die rückgabelose Funktion action druch.
- `sorted()`: sortiere die Elemente im Strom
- `distinct()`: entferne Duplikate aus dem Strom
- `skip(n)`: entferne die ersten n Elemente aus dem Strom
- `limit(n)`: begrenze den Strom auf maximal n Elemente

Beispiel mit map und flatMap

```
/* Datei test.txt:
Dies ist eine
kleine
Test Datei
*/
BufferedReader in = new BufferedReader( new FileReader("test.txt"));
in.lines()
  .peek(System.out::println)
  .flatMap(line -> Arrays.stream(line.split("+")))
  .map( s -> s.toUpperCase())
  .forEach(System.out::println);
/* Ausgabe:
DIES
IST
EINE
KLEINE
TEST
DATEI
*/
```

Beispiel: Stabiles Sortieren nach zwei Schlüsseln

Personen sind nach dem Geburtsjahr und innerhalb einer Jahrgangsstufe alphabetisch sortiert

```
public class Test {
  public static void main(String[] args) {
    List<Person> persList = new LinkedList<>();
    persList.add(new Person("Klaus", 1961));
    persList.add(new Person("Peter", 1959));
    persList.add(new Person("Maria", 1959));
    persList.add(new Person("Petra", 1961));
    persList.add(new Person("Albert", 1959));
    persList.add(new Person("Anton", 1961));
    persList.add(new Person("Iris", 1959));
    persList.stream()
      .sorted(Comparator.comparing(Person::getName))
      .sorted(Comparator.comparing(Person::getGeb))
      .forEach(System.out::println);
  }
}
```

Terminale Operationen

- Mit einer terminalen Operation wird der Strom abgeschlossen
- Terminale Operationen liefern ein Resultat zurück (keinen Strom) oder haben keinen Rückgabewert und eventuell einen Seiteneffekt

```
value = strom.terminalOperation(...);

strom.terminalOperation(...);
```

Logische Operationen

- `anyMatch(pred)` : liefert true, falls pred(x) für ein Element x des Stroms zutrifft
- `allMatch(pred)` : liefert true, falls pred(x) für alle Elemente x des Stroms zutrifft
- `noneMatch(pred)` : liefert true, falls pred(x) für kein Element x des Stroms zutrifft

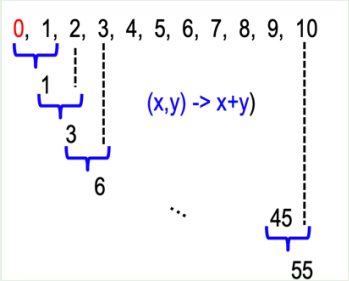
```
// gibt solange zufaellige Zahlen x aus, bis ein x ∈ [10,20). Rueckgabe true
new Random().ints()
  .map(n -> Math.abs(n)%1000)
  .peek(System.out::println)
  .anyMatch(n -> 10 <= n && n < 20)
// gibt solange zufaellige Zahlen x aus, bis ein x ∉ [10,20). Rueckgabe true
new Random().ints()
  .map(n -> Math.abs(n)%1000)
  .peek(System.out::println)
  .allMatch(n -> 10 <= n && n < 10000)
```

Reduktions-Operationen

- `reduce(e, op)`: reduziert einen Strom $x_0, x_1, x_2, \ldots$ zu dem Wert $(\ldots(((e\ op\ x0)\ op\ x_1)\ op\ x_2)\ op \ldots$. Dabei ist `op` ein 2-stelliger assoziativer Operator und `e` das neutrale Element bzgl. `op`.
- `count()`: Anzahl der Elemente im Strom
- `min(cmp)` bzw. `max(cmp)`: Liefert größtes bzw. kleinstes Element des Stroms bezüglich der Comparator-Funktion `cmp`.

Reduktions-Operationen

```
int sum = InStream.range(1,11).reduce(0, (x,y) -> x+y);
System.out.println(sum);
```



Beispiel: harmonisches Mittel mit reduce-Operation

$\bar{x}_{harm} = \frac{n}{\frac{1}{x_0}+\frac{1}{x_1}+...+\frac{1}{x_{n-1}}}$

Anwendung: auf einer Teilstrecke von jeweils 1km werden folgende Geschindigkeiten gefahren: 50, 100, 80, 120, 90 km/h. Dann ist die Durchschnittsgeschwindigkeit der Gesamtstrecke gerade das harmonische Mittel der Einzelgeschwindigkeiten: $v_{harm} = 80,71 km/h$.

```
double[] v_array = {50,100,80,120,90};
double v_harm = Arrays.stream(v_array).reduce{0,(s,v) -> s + 1/v);
v_harm = v_array.length / v_harm;
System.out.println(v_harm);
```

Statistik-Operationen für Basisdatentypen

- `count(), sum(), min(), max(), sverage()`: Liefert Anzahl, Summe, Minimum, Maxium bzw. Durchschnittswert der Elemente eines Stroms zurück
- `summaryStatistics()`: Liefert einen Wert vom Typ IntSummaryStatistics(bzw. DoubleIntSummaryStatistics, ...) zurückm der Anzahl, Summe, Minimum, Maximum und Durchschnittswert umfasst.

Beispiel: Zeilenstatistik für eine Datei

test.txt

Wir werden fünfzig! Die Fachhochschule Dortmund feiert ihr 50-jähriges Bestehen! Dieses besondere Jubiläum wollen wir mit Ihnen zusammen ausgiebig würdigen. Und weil wir sehr viel vorhaben, füllt unser umfangreiches Programm sogar ein ganzes Jahr –von August 2021 bis Juli 2022.

```
BufferedReader in = new BufferedReader(new FileReader("test.txt"));
DoubleSummaryStatistics stat = in.lines().peek(System.out::println).mapToDouble(s -> s.
    length()).summaryStatistics();
System.out.println(stat);
```

Ausgabe

DoubleSummaryStatisticscount=5, sum=276,000000, min=19,000000, average=55,200000, max=90,000000

Terminale Operationen collect und forEach

Collect

`collect(collector)`: Kann benutzt werden, um Elemente des Stroms in einem Container aufzusammeln. Beispielsweise werden mit folgender Anweisung alle Elemente eines String-Strom in einer Liste abgespeichert:

```
List<String> asList = stringStream.collect(Collectors.toList());
```

forEach

`forEach(action)`: führe für jedes Element des Strom die Consumer-Funktion `action` (einstellige Funktion ohne Rückgabewert) durch

Beispiel zu collect-Operation

```
List<Person> persList = new LinkedList<>();
persList.add(new Person("Klaus", 1961));
persList.add(new Person("Anton", 1959));
persList.add(new Person("Maria", 1959));
persList.add(new Person("Petra", 1961));
persList.add(new Person("Anton", 1973));
persList.add(new Person("Peter", 1970));
persList.add(new Person("Anton", 1961));
persList.add(new Person("Klaus", 1959));

// Sortiere die Namen alphabetisch und entferne die Duplikate
List<String> nameList = persList.stream().map(Person::getName).sorted(Comparator.
    naturalOrder()).distinct().collect(Collectors.toList());

System.out.println(nameList);
```

Parallele Streams

- Ströme können parallelisiert werden
- Mit einem Mehrkernprozessor kann damit die Performance verbessert werden

```
// sequentiell
int max = 10_000_000;
long numberOfPrimes = IntStream.range(1, max).filter(isPrime).count();

// parallel
int max = 10_000_000;
long numberOfPrimes = IntStream.range(1, max).parallel().filter(isPrime).count();
```

Zum experimentieren

```
public class Parallel {
  public static void main(String[] args) {
    int max = 100000;
    PrimeCounter pc = new PrimeCounter();
    long time = -System.currentTimeMillis();
    System.out.println(pc.countPrimes(max));
    System.out.println(time + System.currentTimeMillis() + "ms");
  }
  public static class PrimeCounter {

    public long countPrimes(int max) {
      return IntStream.range(1, max).parallel().filter(this::isPrime).count();
    }
    private boolean isPrime(int number) {
      return IntStream.range(2, number).allMatch(x -> (number % x) != 0);
    }
  }
}
```

Nicht-deterministische Reihenfolge bei parallelen Streams

title

- Bei der parallelen Bearbeitung eines Stroms ist die Reihenfolge, in der auf die Elemente des Stromszugegriffen wird, nicht vorhersehbar
- Elemente von 1 bis 20 werden in einer nicht vorhersehbaren Reihenfolge ausgegeben, z.B.: 13, 15, 14, 12, 11, 3, 5, 4, 16, 8, 6, 1, 10, 20, 18, 9 ,2, 19, 17, 7

```
IntStream.range(1, 21).parallel().forEach(System.out::print);
```

Vorsicht bei zustandsbehafteten Funktionen

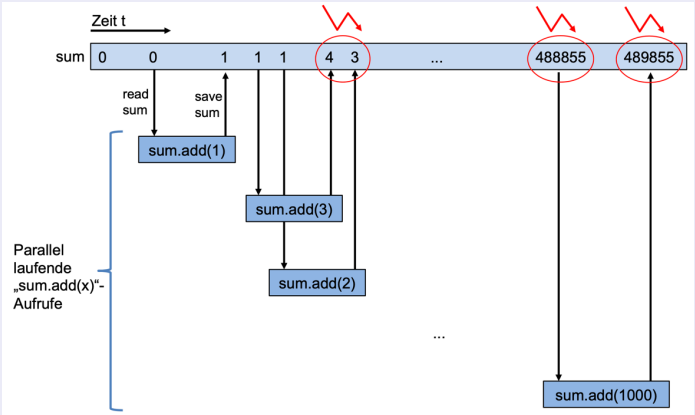
- Vorsicht bei Funktionen, die auf nicht-lokale Datenstrukturen zugreifen (zustandsbehaftete Funktionen)
- Das Ergebnis der Stromverarbeitung kann vom zeitlichen Ablauf der zustandsbehafteten Funktionsaufrufe abhängen (race condition)

```
public class MutableInt {
  int n = 0;
  public int get() {
    return n;
  }
  public void add(int x) {
    n += x;
}}
  public class Parallel2 {
    public static void main(String[] args) {
      MutableInt sm = new MutableInt();
      IntStream.range(1, 1001).parallel().forEach(x ->
            sm.add(x));
      System.out.println(sm.get());
}}
```

Es wird eine zustandsbehaftete Funktion aufgerufen, die auf die mutable nicht-lokale Variable `sum` zugreift. (Variable `sum` ist außerhalb des Lambda-Ausdrucks definiert)

Race Conditions

Race Condition: Die Berechnung von $sum = 1 + 2 + \ldots + 1000 = 500500$ hängt vom zeitlichen Ablauf der `sum.add(x)`-Aufrufe ab



Vermeidung von Race Conditions mit synchronisierten Datentypen

- `AtomicInteger` kapselt einen Integer-Wert und führt Änderungen des Integer-Werts atomar durch
- keine Race Conditions
- Programm wird durch Synchronisierung langsamer

```
public static void main(String[] args) {
  AtomicInteger sm = new AtomicInteger(0);
  IntStream.range(1, 1001).parallel().forEach(x -> sm.addAndGet(x));
  System.out.println(sm.get());
}
```

Bessere Lösung: auf zustandsbehaftete Funktionen verzichten!

```
//  Es wird die zustandslose Funktion sum() aufgerufen
int sum =IntStream.range(1, 1001).parallel().sum();
System.out.println(sum);
```