Die Schnittstelle `Stream`

Sven Eric Panitz

Hochschule Rhein-Main
Standort Wiesbaden

java.util.stream.Stream

- Seit Java 8 gibt es die Schnittstelle `Stream`.
- Sie stellt eine Erweiterung des Konzepts der Schnittstelle `Iterable` dar.
- Sie ermöglicht eine sequenzielle und nebenläufige Iteration.
- Sie realisiert Konzepte der funktionalen Programmierung.

Lebensphasen eines `Stream` Objekts

- Ein Stream Objekt durchläuft drei Phasen:
  - Einmalige Erzeugung des Objektes.
  - Aufruf modifizierende Methoden.
  - Einmaliger Aufruf einer terminierenden Methode, die durch den Stream iteriert.
- Erst die terminierende Methode führt dazu, dass tatsächlich die Iteration durchlaufen wird.

Beispiele: Erzeugung von Streams

- Aufzählung der Objekte:

```
var xs = Stream.of("das","Pferd","frisst","keinen","Gurkensalat")
```

- Aus einem Collection Objekt erzeugen:

```
var is = Set.of(1,2,3,4).stream()
```

- Über statische Methode, die ein Supplier-Objekt verwendet:

```
var hs = Stream.generate(() -> "hallo")
```

- Über iterative Anwendung einer Funktion:

```
var odds = Stream.iterate(1,(x)→x+2)
```

- Durch Verwendung eines Spliteratorobjektes (siehe hierzu separaten Film)

```
Stream<T> StreamSupport.<T>stream(Spliterator<T> spliterator, boolean parallel)
```

Eigenschaften von Stream-Objekten

- Die so erzeugten Stream-Objekte sind vorerst nur eine Beschreibung, wie eine Iteration aussehen wird.
- Die einzelnen Elemente über die iteriert wird müssen noch nicht einmal erzeugt worden sein.
- Ein Stream-Objekt kann potentiell sogar über unendlich viele Elemente iterieren.
- Stream-Objekte können auch bereits bei der Erzeugung als potentiell parallel zu iterieren definiert werden:

```
var is = Set.of(1,2,3,4).parallelStream()
```

Terminierende Methoden

- Stream-Objekte dienen zur einmaligen Iteration der Elemente.
- Hierzu gibt es terminierende Methoden auf Stream-Objekten.
- Terminierende Methoden starten die Iteration.
- Sie führen dazu, dass der Stream verbraucht wird.

Übersicht: Terminierende Methoden 1

- `long count()`

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

- `<R,A> R collect(Collector<? super T,A,R> collector)`
- `<R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)`

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

Übersicht: Terminierende Methoden 2

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

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

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

Terminierende Methoden: count()

- count startet die Iteration und zählt die Durchläufe:

```
jshell> is.count()
$12 ==> 4
```

- count verbraucht den Stream. Ein zweiter Aufruf führt zu einer Ausnahme:

```
jshell> is.count()
|  Exception java.lang.IllegalStateException: stream has already been operated upon or closed
```

Terminierende Methoden: count()

- Für unendliche Stream-Objekte terminiert count() nicht:

```
jshell> var nats = Stream.iterate(2,x->x+1)
nats ==>
java.util.stream.ReferencePipeline$Head@5fdef03a


jshell> nats.count()
$15 ==>
```

Terminierende Methode: forEach

- Die Methode forEach dient dem kompletten Durchlaufen der Iteration.
- Sie entspricht den Aufruf einer Schleife:

```
jshell>  var is =  Set.of(1,2,3,4).stream()


jshell> is.forEach( x-> System.out.println(x*x))
16
9
4
1
```

Terminierende Methode: collect

- Zum Aufsammeln aller Elemente der Iteration in zum Beispiel einer Liste, dient die Methode `collect`:

```
jshell> var ws = Stream.of("A","B","C")

ws ==>
java.util.stream.ReferencePipeline$Head@3f102e87


jshell> var vs = ws.collect(Collectors.toList())

vs ==> [A, B, C]
```

- Es gibt vordefinierte Collector-Objekte für Listen und Mengen.

Faltungen

- Die Faltungsmethode reduce haben wir bereits ausführlich in einem anderen Folienfilm behandelt:

```
jshell> var is =  Set.of(1,2,3,4).stream()

is ==>
java.util.stream.ReferencePipeline$Head@27abe2cd


jshell> is.reduce(0,(x,y)->x+y)

$23 ==> 10
```

logische Prädikate

- Die drei logischen Prädikatsmethoden anyMatch, allMatch, noneMatch, testen die Elemente der Streams.
- Sie entsprechen den Quantoren der Logik: ∃x, ∀x und ¬∃x

```
jshell> var is =  Set.of(1,2,3,4).stream()
jshell> is.anyMatch(x -> x==3)
$25 ==> true
jshell> var is =  Set.of(1,2,3,4).stream()
shell> is.allMatch(x -> x<10)
$35 ==> true
jshell> var is =  Set.of(1,2,3,4).stream()
jshell> is.noneMatch(x -> x>10)
$37 ==> true
```

Selektion einzelner Elemente

- findAny und findFirst dienen zur einmaligen Selektion eines Elements:

```
jshell> var is =  Set.of(1,2,3,4).stream()
jshell> is.findFirst()
$39 ==> Optional[4]
jshell> var is =  Set.of(1,2,3,4).stream()
jshell> is.findAny()
$41 ==> Optional[4]
jshell> var is =  Set.of().stream()
jshell> is.findAny()
$43 ==> Optional.empty
```

ACHTUNG

- Für ein Stream-Objekt kann genau ein einziges mal eine einzige terminierende Methode aufgerufen werden.
- Anschließend ist das Stream-Objekt verbraucht.

Modifizierende Methoden

- Zwischen Erzeugung und Verbrauch können Stream-Objekte modifiziert werden.
- Es kann gefiltert werden,

  die einzelnen Elemente verändert,

  die Anzahl der Elemente verringert werden,

  der Stream parallelisiert werden

  und vieles mehr.

Modifizierende Methoden: Übersicht

- `Stream<T> distinct()`
- `Stream<T> filter(Predicate<? super T> predicate)`
- `Stream<T> limit(long maxSize)`
- `Stream<T> skip(long n)`
- `<R> Stream<R> map(Function<? super T,? extends R> mapper)`
- `Stream<T> peek(Consumer<? super T> action)`
- `Stream<T> sorted()`
- `Stream<T> sorted(Comparator<? super T> comparator)`
- `<R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper)`
- `Stream<T> parallel()`

distinct

- Zum löschen von Doppelten Elementen aus einem Stream dient die Methode distinct:
- ```
  jshell> var is =  Stream.of(1,4,4,4,3,2,2,2,2,2,3,4)
  is ==>
  java.util.stream.ReferencePipeline$Head@7cdbc5d3


  jshell> is.distinct().collect(Collectors.toList())
  $46 ==> [1, 4, 3, 2]
  ```
  ```

filter

- filter wendet ein Prädikat auf die Elemente an und erzeugt den Stream mit den Elemente, für die das Prädikat gilt:
- ```
  jshell> var is =  Stream.of(1,2,3,4,5,6,7,8,9,10)
  is ==> java.util.stream.ReferencePipeline$Head@34340fab


  jshell> is.filter(x->x
  %2==0).collect(Collectors.toList())
  $48 ==> [2, 4, 6, 8, 10]
  ```
  ```

limit und skip

- `limit nimmt die ersten n Elemente, skip` lässt die ersten n Elemente aus.
- `Zusammen können sie einen beliebigen Teil-Stream erzeugen.`
-
```
jshell> var is =  Stream.of(1,2,3,4,5,6,7,8,9,10)
is ==> java.util.stream.ReferencePipeline$Head@3ab39c39


jshell> is.skip(3).limit(5).collect(Collectors.toList())
$50 ==> [4, 5, 6, 7, 8]
```

map

- Mit `map` kann auf jedes Element eine Funktion angewendet werden
- Beispiel. Aus einem Stream von Strings einen Stream der Längen dieser Strings erzeugen:

```
jshell> var is =
Stream.of("das","pferd","ist","keinen","gurkensalat")

is ==> java.util.stream.ReferencePipeline$Head@28864e92


jshell> is.map(x->x.length()).collect(Collectors.toList())

$52 ==> [3, 5, 3, 6, 11]
```

peek (im Vergleich zu map und forEach)

- Die Methode peek ist ein Zwischending zwischen map und forEach
- ```
  jshell> var is =  Stream.of("das","pferd","ist","keinen","gurkensalat")
  jshell> var is2 = is.peek(x -> System.out.println(x.toUpperCase()))
  jshell> is2.map(x -> x.length()).forEach(x->System.out.println(x))
  DAS
  3
  PFERD
  5
  IST
  3
  KEINEN
  6
  GURKENSALAT
  11
  ```
  ```

parallel

- Durch den Aufruf der Methode `parallel()` kann jeder Stream parallel abgearbeitet werden.

```
jshell> var xs = Stream.iterate(1,x->x+1).limit(10)


xs ==> java.util.stream.SliceOps$1@7f690630


jshell> xs.parallel()
.peek(x->System.out.print(x+" ")).collect(Collectors.toList())
3 5 2 1 4 9 6 8 7 10 $59 ==> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Soweit zu Stream

- Es gibt spezialisierte Stream-Klassen für primitive Typen
- Wie funktionieren Streams intern?
- Wie verläuft die Parallelisierung?
- Kann ich eigene Stream-Klassen schreiben?

Das alles in der nächsten Vorlesung.