Sven Eric Panitz

Lehrbriefe
Programmierung in Java

Iteraterieren mit Streams

Parallel Iterieren mit `Stream` und `Spliterator`

Sven Eric Panitz

28. April 2023

Inhaltsverzeichnis

1 Iterieren mit `Stream` und `Spliterator`

Mit Java 8 wurde eine neue Art von Iteratoren eingeführt, die als Grundlage der der Ströme (*Streams*) dienen. Die Grundfunktionalität ist wieder dieselbe, wie schon für Iteratoren: es werden nach und nach Elemente geliefert. Die zusätzliche Funktionalität ist, dass es möglich sein soll, die Iteration wenn möglich zu verteilen auf mehrere nebenläufige Iterationen.

Dieses neue API findet sich im Paket `java.util.streams`[1]. Hier übernimmt die Rolle der Schnittstelle `Iterator` die neue Schnittstelle `Spliterator` und die Rolle der Schnittstelle `Iterable` die neue Schnittstelle `Stream`.

Anders als bei der Schnittstelle `java.util.Iterator` gibt es in `Spliterable` nicht zwei Methoden, die jeweils prüfen, ob es ein neues Element gibt (`hasNext`), bzw. dieses Element liefern

[1] Achtung, es hat nichts mit den *Streams* aus dem Paket `java.io` zu tun.

(next), sondern nur eine Methode für beide Aufgaben. Diese Methode hat wie hasNext eine Wahrheitswert als Rückgabe, der angibt, ob es noch ein weiteres Element gegeben hat. Gleichzeitig führt diese Methode eine Aktion auf, falls es ein weiteres Element gegeben hat.

Die Ströme in Java 8 benutzen exzessiv Methoden höherer Ordnung, d.h. Methoden, die Parameter einer funktionalen Schnittstelle haben. Damit können diesen Methoden per Lambda-Ausdrücke die Argumente übergeben werden.

1.1 Die Schnittstelle `java.util.Spliterator`

Die zentrale neue Schnittstelle, die die Rolle von `Iterator` übernimmt, heißt `Spliterator`. In diesen Abschnitt soll diese Schnittstelle mit dem bekannten Beispiel eines Integerbereichs zum Iterieren veranschaulicht werden. Wir beginnen wie auch bei den `Iterator`-Beispielen mit einer Klasse `Range`, die einen Startwert, einen Endpunkt und einen Iterationsschritt enthält und damit wieder die klassischen drei Elemente einer for-Schleife hat. Statt der Schnittstelle `Iterator` soll jetzt die Schnittstelle `Spliterator` implementiert werden:

Range.java

```
package name.panitz.util.streams;

import java.util.Spliterator;
import java.util.function.Consumer;

public class Range implements Spliterator<Integer> {
  int i, to, step;

  public Range(int from, int to, int step) {
    this.i = from;
    this.to = to;
    this.step = step;
  }
```

1.1.1 `tryAdvance`

In der Schnittstelle `Spliterator` existiert nur eine Methode, die alle Teile einer klassischen Schleife übernimmt, die Method `tryAdvance` mit folgender Signatur:

boolean tryAdvance(Consumer<? super Integer> action)

Die Methode hat einen Rückgabewert, der wie `hasNext` anzeigt, ob es noch ein weiteres Element für die Iteration gegeben hat. Wenn die Methode `true` als Ergebnis liefert, dann sind drei Dinge erfolgt:

- Das nächste Element der Iteration wurde erfragt.
- Mit diesem Element wurde die als Parameter übergebene Aktion ausgeführt. Dieses entspricht in einer klassischen Schleife dem Schleifenrumpf.

- Der Iterator wurde intern weiter geschaltet, um beim nächsten Aufruf ein Element weiter zu liefern.

Der Parameter dieser Methode entspricht also der Aktion, die in einer Schleife im Rumpf für das aktuelle Iterationsobjekt durchgeführt wird. Betrachten wir die ursprüngliche Schleife eines Iterators der Form:

IteratorSchleife

```
for (Iterator<A> it = someIterator; it.hasNext();){
  A x = it.next();
  doSomethingSmartWithElement(x);
}
```

Diese würde mit einem Spliterator wie folgt funktionieren:

SpliteratorSchleife

```
while (someSplitertor.tryAdvance(x->doSomethingSmartWithElement(x))){}
```

Kurioser Weise kommt diese Schleife sogar mit einem leeren Schleifenrumpf aus. Der Aufruf von `tryAdvance` führt ja sowohl eine Aktion für das nächste Element aus, zeigt aber mit dem Ergebnis, ob überhaupt noch Elemente in der Iteration existieren.

Mit diesem Hintergrundwissen ergibt sich für den Spliterator eines Zahlenbereichs, folgende Implementierung:

Range.java

```
@Override
public boolean tryAdvance(Consumer<? super Integer> action) {
  if (i<=to){
    action.accept(i);
    i += step;
    return true;
  }
  return false;
}
```

In diesem Fall sind also alle Komponenten einer Schleife in eine Methode gepackt. Die if-Bedingung prüft, ob es noch einen Schleifendurchgang gibt. Ansonsten wird direkt `false` zurückgegeben. Dann wird der Schleifenrumpf durch die Methode `accept` des übergebenen `Consumer`-Objekts auf die aktuelle Schleifenvariable `i` aufgerufen. Schließlich wird diese um den vorgegebenen Schritt weiter geschaltet. Da es noch ein weiteres Element gab, das verarbeitet wurde, gibt die Methode aus der if-Bedingung schließlich `true` zurück.

Damit können wir die Objekte der Klasse `Range` zum Iterieren verwenden:

Shell

```
jshell> import name.panitz.util.streams.Range

jshell> var r=new Range(1,100,5)
r ==> name.panitz.util.streams.Range@46f7f36a

jshell> while(r.tryAdvance(x->System.out.print(x*x+" ")));
1 36 121 256 441 676 961 1296 1681 2116 2601 3136 3721 4356 5041 5776 6561 7396 8281 9216
jshell>
```

1.1.2 trySplit

Die Schnittstelle `Spliterator` hat aber noch drei weitere abstrakte Methoden. Diese beschäftigen sich alle damit, ob der Spliterator zur nebenläufigen Abarbeitung geteilt werden kann. Die wichtigste Methode ist dabei die Methode `trySplit` mit folgender Signatur:

public Spliterator<T> trySplit()

Gibt die Methode `null` zurück, so zeigt das an, dass der Spliterator nicht aufgeteilt werden kann und sequentiell abzuarbeiten ist. Wird hingegen ein neues Spliteratorobjekt zurückgegeben, so gibt es dann zwei Spliteratoren: den ursprünglichen und den durch `trySplit` erzeugten. Beide zusammen sollen über alle Elemente des ursprünglichen Spliterators iterieren. Es ändert sich also dann auch der Iterationsbereich des Originalspliterators.

Am deutlichsten wird dieses sicher durch unser Standardbeispiel. Zunächst starten wir mit einer if-Bedigung, die prüft, ob sich ein Aufsplitten des Spliterators auf zwei Objekte noch lohnt. Wenn keine 4 Elemente mehr zu iterieren sind, geben wir einfach `null` zurück.

Range.java

```
@Override
public Spliterator<Integer> trySplit() {
  if (to-i<4*step){
    return null;
  }
```

Spannender ist es natürlich, tatsächlich zu splitten und einen neuen Spliterator zu erzeugen. Hierzu berechnen wir die Mitte des Iterationsbereichs. Setzen den Startwert `i` des aktuellen Spliterators auf diesen Mittelwert und geben einen neuen Spliterator zurück, der vom Startwert `i` bis zu diesen Mittelwert iteriert.

Range.java

```
  int steps = (to-i)/step;
  int middle = (i+steps/2*step);
  int iOld = i;
  i=middle;
  return new Range(iOld, middle-1, step);
}
```

Die Arbeit wurde also auf zwei Iteratorobjekte geteilt. Ein neues Objekt, das zur Iteration vom Anfang bis zur Mitte dient und das ursprüngliche, das die Iteration von der Mitte bis zum Ende darstellt.

Nicht immer lässt sich die Arbeit so gut aufteilen. Hierzu dient ein schönes Bild: Wenn ein Graben von 100m Länge, ein Meter Breite und ein Meter Tiefe auszuheben ist, lässt sich diese Arbeit wunderbar schnell mit 100 Menschen bewerkstelligen. Ist hingegen eine Loch von einen Quadratmeter und 100m Tiefe auszuheben, so ist zwar das gleiche Volumen auszuheben, aber 100 Menschen können dieses schwer effektiv gemeinsam bewerkstelligen. Wahrscheinlich würden wenige schuften, und der Rest die ganze Aktion verwalten. Ein wenig wie in unserer Hochschule.

1.1.3 `estimateSize`

Eine weitere abstrakte Methode der Schnittstelle `Spliterator` soll dazu dienen, abzuschätzen, über wie viele Elemente iteriert wird. Diese Information ist hilfreich zur Planung, ob es sich lohnt die Iteration zu parallelisieren. Es gibt Spliteratoren, die nicht genau wissen, wie viel Elemente sie haben, oder es kann sein, dass die Information zu berechnen zu aufwändig ist. Dann soll die Methode die größtmögliche Zahl zurückgeben. Ebenso, wenn der Spliterator über unendlich viele Elemente läuft. Dann ist der Wert `Long.MAX_VALUE` zurückzugeben.

In unserem Beispiel können wir die Größe leicht genau berechnen:

Range.java

```
@Override
public long estimateSize() {
  return (to-i)/step;
}
```

1.1.4 `characteristics`

Die letzte abstrakte Methode heißt `characteristics`. In einer Zahl codiert werden hier einige Charakteristiken genannt. Die Menge der Charakteristiken beinhaltet. `ORDERED`, `DISTINCT`, `SORTED`, `SIZED`, `NONNULL`, `IMMUTABLE`, `CONCURRENT`, `SUBSIZED`.

Viele dieser Charakteristiken erklären sich vom Namen her. Für die genaue Bedeutung verweisen wir hier auf die Java API Dokumentation.

Die Werte der einzelnen Charakteristiken können bitweise verodert werden. In unserem Beispiel können wir vier Charakteristiken setzen.

Range.java

```
@Override
public int characteristics() {
  return ORDERED | SIZED | IMMUTABLE | SUBSIZED;
}
```

```
}
```

Wir haben einen ersten Spliterator implementiert. Wir können uns davon überzeugen, dass die Arbeit mit einem Aufruf von `trySplit` tatsächlich auf zwei Objekte verteilt wird.

Shell

```
jshell> var r1=new Range(1,100,5)
r1 ==> name.panitz.util.streams.Range@1a86f2f1

jshell> var r2=r1.trySplit()
r2 ==> name.panitz.util.streams.Range@506c589e

jshell> while(r1.tryAdvance(x->System.out.print(x+" ")));
46 51 56 61 66 71 76 81 86 91 96
jshell> while(r2.tryAdvance(x->System.out.print(x+" ")));
1 6 11 16 21 26 31 36 41
```

Spliteratoren haben noch weitere default-Methoden, die bei Bedarf überschrieben werden können. Auch hier verweisen wir nur auf die Dokumentation.

1.2 Ströme

Spliteratoren sind die Grundlage für die Ströme (streams) in Java 8. Ströme kapseln noch einmal Spliteratoren und bieten eine Vielzahl von Operationen, um mit ihnen zu arbeiten. Ströme bieten einen Iterationsbereich. Sie sind zum einmaligen Gebrauch gemacht. Ein Strom hat drei Phasen seiner Lebenszeit:

- Zunächst muss ein Strom erzeugt werden. Oft wird ein Strom aus einer Datenhaltungsklasse, meist Sammlungsklassen, heraus erzeugt. Es gibt aber auch andere Möglichkeiten Ströme zu erzeugen.
- Es können viele Transformationen auf einem Strom durchgeführt werden. Er kann gefiltert werden, die Elemente können auf andere Elemente abgebildet werden, aber auch komplexere Operationen wie eine Sortierung kann durchgeführt werden.
- Ein Strom muss erst zur Verarbeitung angestoßen werden. Hierzu dienen die terminierenden Methoden auf Strömen. Sie stoßen die eigentliche Iteration an, um ein Endergebnis zu erzielen.

Betrachten wir alle drei Phasen im Einzelnen:

1.2.1 Erzeugen von Strömen

Stromelemente Aufzählen Die vielleicht einfachste Form, einen Strom zu erzeugen, besteht darin, die Stromelemente einfach aufzuzählen. Hierzu gibt es in der Klasse `Stream` die statische Methode `of`. Diese hat eine variable Parameterzahl, es können also beliebig viele Parameter eines in diesem Fall generisch gehaltenen Typ übergeben werden.

Shell

```
jshell> import java.util.stream.Stream

jshell>  Stream.of("Hallo", "Freunde", "Hallo", "Ilja").
   ...>    forEach(x->System.out.println(x.toUpperCase()))
HALLO
FREUNDE
HALLO
ILJA
```

In diesem Beispiel rufen wir testweise bereits die terminierende Methode `forEach` des Stroms auf.

Ströme für Spliteratoren Eine Möglichkeit ist es, einen Strom für ein bereits existierendes Spliterator-Objekt zu erzeugen. Hierzu gibt es in der Hilfsklasse `StreamSupport` eine statische Methode `stream`, die einen Spliterator als Parameter erhält. So können wir für unsere Spliterator-Objekte der Klasse `Range` einen Strom erzeugen.

Shell

```
shell> import java.util.stream.*

jshell> import name.panitz.util.streams.Range

jshell> var stream = StreamSupport.stream(new Range(1,100,7),false)
stream ==> java.util.stream.ReferencePipeline$Head@31cefde0

jshell> stream.forEach(x->System.out.print(x+" "))
1 8 15 22 29 36 43 50 57 64 71 78 85 92 99
```

Wie man in diesem Beispiel sieht, hat die Methode `stream` einen zweiten Parameter. Dieser zeigt an, ob für den Strom eine nebenläufige Abarbeitung erlaubt ist. Starten Sie einmal die zweite Version dieses Beispiels, in dem diesem Parameter `true` übergeben wird. kommen die Ausgaben nicht mehr in aufsteigender sondern in einer zufälligen Reihenfolge.

Shell

```
jshell> var stream = StreamSupport.stream(new Range(1,100,7),true)
stream ==> java.util.stream.ReferencePipeline$Head@5197848c

jshell> stream.forEach(x->System.out.print(x+" "))
50 57 64 1 22 29 36 85 43 71 78 8 92 15 99
```

Ströme aus Generatorfunktionen Es gibt zwei Möglichkeiten, wie man unendliche Ströme mit Hilfe einer Generatormethode erzeugen kann:

- Eine Möglichkeit unendliche Ströme zu erzeugen, besteht darin, das erste Element des Stroms zu geben und eine Funktion, die berechnet, wie das nächste Element aus dem aktu-

ellen berechnet wird. Für einen Startwert s und eine Funktion f wird damit folgender Strom erzeugt:

$$s, f(s), f(f(s)), f(f(f(s))), f(f(f(f(s)))), \ldots$$

Die entsprechende Methode, auf diese Weise einen Strom zu erzeugen, befindet sich in der Klasse `Stream` und heißt: `iterate`.

So lässt sich zum Beispiel auf einfache Weise die Folge der natürlichen Zahlen als Strom erzeugen. Hierzu ist der Startwert die Zahl 1 und die Funktion addiert jeweils 1 auf den aktuellen Wert.

Shell

```
jshell> Stream.iterate(1l, x -> x+1l).forEach(x-> System.out.print(x+" "))
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...
```

Mit dieser Methode kann auch ein simpler Strom erzeugt werden, der immer wieder ein Element unendlich oft wiederholt. Hierzu ist die zu übergebene Funktion die Identität `(x) -> x`.

Shell

```
jshell> Stream.iterate("hallo", x -> x).forEach(x-> System.out.print(x+" "))
hallo hallo hallo hallo hallo hallo hallo hallo hallo hallo hallo hallo hallo...
```

- Eine zwei Möglichkeit, mit Hilfe eines Generators einen Strom zu erzeugen, ist die Methode `generate`

```
static <T> Stream<T> generate(Supplier<T> s)
```

Das Supplier-Objekt liefert über die Methode `get` auf wiederholten Aufruf ein Objekt. So kann auch diese Methode dazu verwendet werden, eine unendliche Folge von aufsteigenden Zahlen zu liefern.

Shell

```
jshell> var stream = Stream.generate(
   ...> new Supplier<Long>(){
   ...>     long l = 1L;
   ...>     public Long get(){return l++;}
   ...> })
stream ==> java.util.stream.ReferencePipeline$Head@421faab1

jshell> stream.forEach(x->System.out.print(x+" "))
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...
```

Ströme aus Standardsammlungsklassen Die in der Praxis am häufigsten vorkommende Methode, um einen Strom zu erzeugen, dürfte über eine Standard-Sammlungsklasse sein. Alle Sammlungsklassen haben die Methode `stream`, die für die Elemente der Sammlung einen Strom erzeugen.

Shell

```
jshell> List.of("Freunde","Roemer","Landsleute").stream().
   ...> forEach(x->System.out.println(x))
Freunde
Roemer
Landsleute
```

Die Sammlungsklassen haben eine zweite Methode, um einen Strom zu liefern. Diese gibt einen Strom, der nebenläufig abgearbeitet werden darf.

Shell

```
jshell> List.of("Freunde","Roemer","Landsleute").parallelStream().
   ...> forEach(x->System.out.println(x))
Roemer
Landsleute
Freunde
```

Spezielle Ströme für primitive Typen Da in Java generische Klassen nur Referenztypen als Elemente für Typvariablen erhalten können, müssen primitive Werte in die entsprechenden Wrapper-Klassen verpackt werden. So gibt es keinen Typ `Stream<int>` sondern nur einen Typ `Stream<Integer>`. Möchte man also mit Zahlen arbeiten, sind diese immer erst in einen Referenztypen zu verpacken und zum Rechnen mit den Zahlen wieder auszupacken. Das kann ineffizient sein. Wenn allein auf primitiven Werten mit einem Stream gearbeitet werden soll, bietet das Java-API drei spezialisierte Stream Schnittstellen an: `IntStream`, `LongStream` und `DoubleStream`. Diese haben fast dieselben Methoden wie die Schnittstelle `Stream` nur spezialisiert auf die entsprechenden primitiven Typen als Elementtyp.

1.2.2 Terminierende Methoden auf Strömen

count Die mit Sicherheit einfachste terminierende Methode für Ströme, iteriert den Strom einmal durch, und zählt die Elemente auf. Die entschrechende Methode heißt count:
`long count()`

Es ist zu beachten, dass diese Methode den Strom verbraucht. Einmal durchiteriert, um die Elemente zu zählen, bedeutet, dass der Strom verbraucht wurde. Deshalb werden diese Methoden als *terminierende* Methoden bezeichnet.

Auf einem Stream kann nur einmal eine terminierende Methode aufgerufen werden. Ein zweiter Aufruf führt zu einem Fehler.

Shell

```
jshell> var stream = List.of(1,2,3,4).parallelStream()
stream ==> java.util.stream.ReferencePipeline$Head@2e0fa5d3

jshell> stream.count()
$9 ==> 4

jshell> stream.count()
|  Exception java.lang.IllegalStateException: stream has already been operated upon or closed
|        at AbstractPipeline.evaluate (AbstractPipeline.java:229)
|        at ReferencePipeline.count (ReferencePipeline.java:605)
|        at (#10:1)
```

Man könnte also auch nicht ersteinmal mit `count` die Elemente eines Stroms durchzählen, um dann anschließend mit dem Strom zu iterieren und irgendetwas mit den Elementen zu machen.

`foreach` Eine übliche Form einen Strom zu verbrauchen, ist die Methode `forEach`, die wir in all den Beispielen bereits aufgerufen haben. Nach Aufruf der Methode `forEach` ist jedes Element des Stroms bereits iteriert worden. Die Methode `forEach` entspricht dem Aufruf einer Schleife die über die Elemente der Iteration läuft und einen Code-Block für jedes Element im Rumpf der Schleife aufruft. Die genaue Signatur lautet:

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

Der Parameter ist ein Objekt der funktionalen Schnittstelle `Consumer`, das die Elemente von Typ `T` konsumieren kann, sprich als Parameter der Methode `accept` erhalten kann. Das ist für alle Konsumer vom Elementtyp `T` oder einem Obertyp von `T` der Fall. Deshalb ist der Parameter als `Consumer<? super T>` gesetzt. Der Parametertyp der Methode `accept` kann ein beliebiger Obertyp von `T` sein. Daher `? super T`.

In den vorangegangenden Beispielaufrufen haben wir bereits die Methode `foreach` oft aufgerufen.

`allMatch` **und** `anyMatch` Diese beiden Funktionen entsprechen den Prädikatenlogischen Junktoren für-alle $\forall$ und existiert $\exists$. Die Elemente des Stroms werden mit einem Prädikat getestet und geschaut, dieses Prädikat für alle Elemente erfüllt ist bzw. für mindestens ein Element erfüllt ist.

Faltungen Ein Blick in die Schnittstelle `Stream` zeigt, dass sich hier mehrere überladene Versionen einer Methode `reduce` befinden, die die Faltung realisieren. Der Grund dafür, dass verschiedene überladene Versionen angeboten werden, liegt darin, dass auch die Methode `reduce` nach Möglichkeit die Iteration über die Elemente parallelisiert.

- Die einfachste Form der Faltung hat folgende Signatur:

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

Anders als in unserer Methode `fold` gibt es nicht zwei unterschiedliche Typen für die Elemente des Iterationsbereichs und das Ergebnis der Faltung, sondern beide sind von demselben Typ `T`. Somit ist die Funktion eine Funktion, die zwei Elemente des Typs `T` zu einem neuen Element des Typs `T` verknüpfen.
Der Parameter für den Startwert heißt ganz bewusst `identity`. Es soll ein neutrales Element bezüglich der Operation sein, mit der die Elemente verknüpft werden.

- Eine Version verzichtet auf einen Parameter für den Startwert

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

  Da es keinen Startwert gibt, mit dem das Ergebnis initial initialisiert ist, ist nicht klar, was das Ergebnis sein soll, wenn der Strom gar kein Element zum Iterieren enthält. Daher ist hier der Ergebnistyp nicht vom Typ `T` sondern vom Typ `Optional<T>`. Dieser Typ enthält entweder ein Element vom Typ `T` oder zeigt an, dass es kein Element gibt.

- Die komplette Version benötigt drei Parameter:

  <U> U reduce (U id, BiFunction<U,? super T,U> acc, BinaryOperator<U> comb)

  Diese Version entspricht am nächsten unserer Methode `fold`. Sie hat allerdings noch einen dritten Parameter. Dieser wird benötigt, damit die Abarbeitung der Methode auch parallelisiert werden kann.

Das Faltungen eine mächtige Funktionalität sind, haben wir schon vielleicht gesehen. So lassen sich mit einem geschickten Aufruf von `reduce` die Elemente eines Strom-Objektes in einer Liste sammeln:

Shell

```
jshell> List<Integer> xs = Stream.of(1,2,3,4).reduce(new ArrayList<>()
   ...> ,(rs,e)->{rs.add(e);return rs;}
   ...> ,(rs1,rs2)->{rs1.addAll(rs2);return rs1;})
xs ==> [1, 2, 3, 4]
```

`collect` Da es ein häufiger Anwendungsfall ist, dass schließlich die Elemente eines Stroms in einem Sammlungsobjekt, zumeist einer Liste, aufenommen werden soll, gibt es hierfür sie darauf spezialisierten Funktionen `collect`, obwohl wir ja gerade gesehen haben, wie diese Aufgabe mit `reduce` gelöst werden kann.

Die Methode benötigt ein Objekt des Typs `Collector`.

<R,A> R collect(Collector<? super T,A,R> collector)

Praktischer Weise gibt es in der Klasse `Collectors` bereit gestellte Objekte, um die Stromelemente in Listen oder in Mengen zu sammeln:

Shell

```
jshell> Stream.of(3251,21,525,22,22,545,452).collect(Collectors.toList())
$2 ==> [3251, 21, 525, 22, 22, 545, 452]

jshell> Stream.of(3251,21,525,22,22,22,545,452).collect(Collectors.toSet())
$3 ==> [545, 3251, 452, 21, 22, 525]
```

1.2.3 Transformierende Methoden auf Strömen

Betrachten wir jetzt Funktionen, die einen Strom in einen neuen Strom transformieren. Diese Funktionen führen noch nicht zur eigentlichen Iteration, sondern sind bei erst einer eventuellen Iteration zu berücksichtigen.

Die Funktionen sollten Ihnen alle recht bekannt vorkommen, denn wir haben Sie in unseren Listenklassen alle bereits für Listen umgesetzt,

`limit` und `skip` Einen Strom zu limitieren, bedeutet, dass man eine maximale Anzahl von Elementen bei der Iteration haben möchte, nämlich die ersten n-Elemente. Falls es gar nicht so viele Elemente gibt, endet die Iteration trotzdem schon früher.

Dieses ist die Funktion, die wir auf Listen unter den Namen `take` umgesetzt haben.

Hier ein kurzer Beispielaufruf der Methode:

Shell

```
jshell> Stream.iterate("hallo", x -> x).limit(10).
   ...> forEach(x-> System.out.print(x+" "))
hallo hallo hallo hallo hallo hallo hallo hallo hallo hallo

jshell> Stream.of(1).limit(10).forEach(x-> System.out.print(x+" "))
1
```

Von einem Strom die ersten n Elemente ignorieren und bei Iteration erst mit dem $n+1$-ten Element beginnen, hierzu dient die Funktion `skip`. Diese entspricht der Funktion `drop`, die wir bereits für die Listen umgesetzt haben.

Zur Illustration ein Beispielaufruf.

Shell

```
jshell> Stream.of(1,2,3,4,5,6,7,8,9,10).skip(4).
   ...> forEach(x-> System.out.print(x+" "))
5 6 7 8 9 10
```

`filter` Mit einem Prädikat über die Elemente einen Filter zu setzen ist eine häufig bnötigte Funktionalität. So wie wir diese in unseren Listen bereits umgesetzt hatten, existiert sie auch für Strom-Objekte.

Shell

```
jshell> Stream.iterate(1L, x -> x+1L).filter(x->x%3==0).limit(10).
   ...> forEach(x-> System.out.print(x+" "))
3 6 9 12 15 18 21 24 27 30
```

`map` Eine Funktion auf jedes Element des Stroms anzuwenden, dafür dient die Funktion `map`. Auch diese ist uns von den Listen bereits geläufig.

Shell

```
jshell> Stream.iterate(1L, x -> x+1L).map(x->x*x).limit(10).
   ...> forEach(x-> System.out.print(x+" "))
1 4 9 16 25 36 49 64 81 100
```

`dropWhile` **und** `takeWhile` Diese beiden Funktion nehmen Elemente oder lassen Elemente vom Anfang des Stroms fallen. Das Kriterum ist dabei ein Prädikat, Auch diese beiden Funktionen haben wir bereits in unserer Listenimplementierung selbst umgesetzt.

Ein Beispiel für das Fallenlassen:

Shell

```
jshell> Stream.iterate(1L, x -> x+1L).dropWhile(x->x<10).limit(10).
   ...> forEach(x-> System.out.print(x+" "))
10 11 12 13 14 15 16 17 18 19
```

Ein Beispielaufruf für das nehmen.

Shell

```
jshell> Stream.iterate(1L, x -> x+1L).takeWhile(x->x<10).
   ...> forEach(x-> System.out.print(x+" "))
1 2 3 4 5 6 7 8 9
```

`distinct` Strom-Objekte lassen sich durch Aufruf der Funktion `distinct` zu Strom-Objekten ohne doppelte transformieren.

Shell

```
jshell> Stream.of(1,1,2,4,55,3,21,1,2,2,2,2,2,2,2,2).distinct().
   ...> forEach(x-> System.out.print(x+" "))
1 2 4 55 3 21
```

`sorted` Auch Sortierfunktionen sind für Strom-Objekte vorhanden:

Shell

```
jshell> Stream.of(1,321,42,2,65,85,432,65).sorted().
   ...> forEach(x-> System.out.print(x+" "))
1 2 42 65 65 85 321 432
```

2 Aufgaben

In den Aufgaben soll das Arbeiten mit Strom-Objekten eingeübt, aber auch ein eigener kleiner Spliterator entwickelt werden.

Wir benötigen eine Reihe von Imports für die Lösung:

Strom.java

```
package name.panitz.util;
import java.util.function.Function;
import java.util.function.Predicate;
import java.util.function.BiFunction;
import java.util.function.Consumer;
import java.util.*;
import java.io.*;
import java.util.stream.Collectors;
import java.util.stream.*;
import java.math.BigInteger;
```

Wir schreiben eine Schnittstelle in der sich alle Lösungen als statische Eigenschaften befinden:

Strom.java

```
public interface Strom{
```

Aufgabe 1 In dieser Aufgabe sollen Sie den Aufruf der Methode reduce üben. Es sind eine Reihe von Aufgaben zu lösen, die Sie bisher iterativ oder rekursiv gelöst haben. Ergänzen Sie die folgenden Methoden mit den passenden Parametern für reduce.

a) Es soll der Stringparameter als Binärzahl gelesen werden.

Strom.java

```
static long readBinary(String x){
  return x.chars().reduce( /*TODO*/ );
}
```

Beispielaufruf:

Shell

```
jshell> readBinary("101010")
$2 ==> 42
```

b) Zur Berechnung der Quersumme einer Zahl.

Strom.java

```
static long quersumme(long x){
  return LongStream
    .iterate(x,(y)->y/10)
    .takeWhile(y->y>0)
    .reduce( /*TODO*/ );
}
```

Beispielaufruf:

Shell

```
jshell> quersumme(4242)
$3 ==> 12
```

c) Zur Berechnung der Fakultät.

Strom.java

```
static long factorial(int x){
  return LongStream
    .iterate(1L,(y)->y+1L)
    .limit(x)
    .reduce( /*TODO*/ );
}
```

Beispielaufruf:

Shell

```
jshell> factorial(10)
$4 ==> 3628800
```

d) Zur Erzeugung eines Strings, der die Zahl als Binärzahl darstellt.[2]

Strom.java

```
static String asBinary(long x){
  return (x==0) ? "0"
    :LongStream
    .iterate(x,(y)->y/2)
    .takeWhile(y->y>0)
    .mapToObj(y->y)
    .reduce( /*TODO*/ );
}
```

Beispielaufruf:

Shell

```
jshell> asBinary(42)
$5 ==> "101010"
```

e) Verallgemeinern Sie hier die Methode toBinary, so dass die Zahl x in einen String im Stellenwertsystem mit der entsprechenden Basis $2 \leq b \leq 16$ konvertiert wird.

Strom.java

```
static String convertToBase(int b,long x){
  return "" /*TODO*/ ;
}
static String digits="0123456789ABFDEFGHIJKLMNOPQRSTUVWXYZ";
static char toDigit(int x){
  return digits.charAt(x);
}
```

Beispielaufruf:

Shell

```
jshell> convertToBase(8,42)
$6 ==> "52"

jshell> convertToBase(16,42)
$7 ==> "2A"
```

f) Verallgemeinern Sie hier die Methode readBinary, so dass der String x als eine Zahl im Stellenwertsystem mit der entsprechenden Basis $2 \leq b \leq 16$ gelesen wird.

[2]Hinweis: In dem Lösungsfragment wird die seit Java 9 neue Stream-Methode takeWhile verwendet.

Strom.java

```java
static long readFromBase(int b,String x){
  return 0L /*TODO*/ ;
}
```

Beispielaufruf:

Shell

```
jshell> readFromBase(16,"2A")
$8 ==> 42

jshell> readFromBase(8,"52")
$9 ==> 42
```

Aufgabe 2 Gegeben sei die zusätzliche Record-Klasse, die ein Paar aus zwei Zahlen des Typs `long` darstellt:

Strom.java

```java
public static record TwoLong(long i1, long i2){}
```

Schreiben Sie in dieser Aufgabe Methoden, die Stream-Objekte für die Fibonaccizahlen erzeugen.

a) Es soll der unendliche Stream erzeugt werden, aus Elementen der Klasse TwoLong, so dass in den Paaren immer zwei Fibonaccizahlen stehen entsprechend der folgenden Auflistung:

$$(0,1),(1,1),(1,2),(2,3),(3,5),(5,8),(8,13),(13,21),\ldots$$

Strom.java

```java
static public Stream<TwoLong> fibPairs(){
  //TODO
  return null;
}
```

Verwenden Sie hierzu die Methode `Stream.iterate`.

Beispielaufruf:

Shell

```
jshell> fibPairs().limit(10).collect(Collectors.toList())
$11 ==> [TwoLong[i1=0, i2=1], TwoLong[i1=1, i2=1], TwoLong[i1=1, i2=2], TwoLong[i1=2, i2=3], TwoLong[i1=3
```

b) Erzeugen Sie nun daraus den unendliche Stream aller Fibonaccizahlen. Die erste Zahl sei dabei die 0.

Strom.java

```
static public Stream<Long> fibs(){
  //TODO
  return null;
}
```

c) Erzeugen Sie daraus einen Stream, der über die ersten 100 Fibonaccizahlen iteriert.

Strom.java

```
static public Stream<Long> fibs100(){
  //TODO
  return null;
}
```

Beispielaufruf:

Shell

```
jshell> fibs().limit(20).collect(Collectors.toList())
$14 ==> [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181]
```

d) Und verwenden Sie die bisherigen Funktionen, zur Realisierung er Funkion, die die i-te Fibonaccizahl berechnet. Für $i = 1$ sei das Ergebnis die Zahl 0.

Strom.java

```
static public long fib(int n){
  //TODO
  return 0L;
}
```

Beispielaufruf:

Shell

```
jshell> fib(20)
$15 ==> 4181
```

Aufgabe 3 Gegeben sei folgenden Klasse um zwei `BigInteger`-Objekte zu speichern.

Strom.java

```
static public record TwoBig(BigInteger i1, BigInteger i2){}
```

Schreiben Sie in dieser Aufgabe Methoden, die Stream-Objekte für die Fakultäten erzeugen.

a) Schreiben Sie eine Funktion, die den unendlichen Stream erzeugt aus Elementen der Klasse TwoBig, so dass in den Paaren immer n und fac(n) stehen entsprechend der folgenden Auflistung:

$$(1,1),(2,2),(3,6),(4,24),(5,120),\ldots$$

Strom.java

```
static public Stream<TwoBig> facPairs(){
  //TODO
  return null;
}
```

Beispielaufruf:

Shell

```
jshell> facPairs().limit(5).collect(Collectors.toList())
$16 ==> [TwoBig[i1=1, i2=1], TwoBig[i1=2, i2=2], TwoBig[i1=6, i2=3], TwoBig[i1=24, i2=4], TwoBig[i1=120,
```

b) Verwenden Sie die `facPairs`-Methode um einen Stream aller Fakultäten zu erzeugen. Die erste Fakultät sei dabei die 1.

Strom.java

```
static public Stream<BigInteger> facs(){
  //TODO
  return null;
}
```

Beispielaufruf:

Shell

```
jshell> facs().limit(20).collect(Collectors.toList())
$17 ==> [1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800, 39916800, 479001600, 6227020800, 871782912
```

c) Verwenden Sie jetzt die zuletzt geschriebene Funktion, um die Fakultät zu errechnen.

Strom.java

```
static public BigInteger fac(int n){
  //TODO
  return null;
}
```

Beispielaufruf:

Shell

```
jshell> fac(42)
$18 ==> 1405006117752879898543142606244511569936384000000000
```

Aufgabe 4 In dieser Aufgabe ist ein spezieller Spliterator zu entwickeln.

a) Schreiben Sie eine Klasse, die es erlaubt über die Zeichen eines String zu iterieren. Sehen Sie vor, dass ein `trySplit` die Arbeit fair auf zwei Spliterator-Objekte verteilt. Setzen Sie auch `estimateSize` auf einen möglichst genauen Wert.

Strom.java

```
static public class SpliterateString
                    implements Spliterator<Character> {
  int i = 0;
  int end;
  String s;

  public SpliterateString(String s) {
    this(0, s.length() - 1, s);
  }

  public SpliterateString(int i, int end, String s) {
    this.i = i;this.end = end;this.s = s;
  }
}
```

Strom.java

```
}
```

Stream
ParallelStream
Spliterator