7 Streams

Streams sind neben den Lambda-Ausdrücken die wichtigste Neuerung in Java zur funktionalen Programmierung. Streams unterstützen das Verarbeiten einer Folge von Elementen, sowohl sequentiell als auch parallel. In diesem Kapitel werden wir uns auf die sequentielle Verarbeitung konzentrieren, um uns im nächsten Kapitel dann intensiv mit der parallelen Verarbeitung auseinanderzusetzen.

7.1 Grundlagen von Streams

Streams sind in Java Objekte, die einen Zugriff auf eine Folge von Elementen bereitstellen. Sie haben damit eine gewisse Verwandtschaft zu den Collections. Streams verhalten sich aber in wichtigen Aspekten ganz unterschiedlich zu Collections. Die besonderen Eigenschaften von Streams lassen sich gut erkennen, wenn man Streams und Collections gegenüberstellt. Tabelle 7–1 vergleicht dazu Collections und Streams:

Collection	Stream
Speichern Elemente	Liefern Elemente
Anfügen, Löschen und Zugriff	Nur Zugriff
Mehrfache Iteration	Nur einmalige Iteration
Externe Iteration	Interne Iteration
Verarbeitung strikt im eifrigen Modus	Verarbeitung nicht-strikt nach Bedarf

Tab. 7–1 *Unterschied Collections und Streams*

- Im Gegensatz zu Collections speichern Streams keine Elemente, sondern stellen nur einen Zugriff auf Elemente bereit. Die Quelle der Elemente wird vom Stream abstrahiert. Ist die Quelle der Elemente eine Collection, wird auf die Elemente der Collection zugegriffen. Die Quelle kann aber zum Beispiel auch ein externes Medium oder ein Mechanismus zum Erzeugen der Elemente sein.
- Mit Streams kann man nur auf die Elemente zugreifen. Ein Anfügen und Löschen und ganz allgemein ein Verändern der Menge der Elemente ist bei Streams nicht möglich.

- Mit Streams kann man nur einmal über die Elemente iterieren. Will man die Elemente nochmals verarbeiten, muss ein neues Stream-Objekt erzeugt werden.
- Streams arbeiten mit einer internen Iteration. Das bedeutet, dass die Iteration über die Elemente nicht im Anwenderprogramm implementiert wird, sondern intern im Stream erfolgt.
- Streams arbeiten nach dem Prinzip der Bedarfsauswertung, bei dem Elemente erst dann geliefert werden, wenn diese angefordert werden.

Die beiden letzten Eigenschaften, interne Iteration und Bedarfsauswertung, sind für Streams besonders wichtig. Wir werden daher diese Eigenschaften noch genauer analysieren.

7.1.1 Ein erstes Beispiel

Schauen wir uns aber zuerst ein erstes Beispiel an. In folgender Anweisung wird mit Stream-Operationen eine Liste von Artikeln verarbeitet. Es werden zuerst jene Artikel herausgefiltert, die einen Preis kleiner als 1000 haben, diese nach dem Preis sortiert, dann auf ihren Namen abgebildet und schließlich daraus wieder eine Liste erzeugt. Die Ergebnisliste enthält somit die Namen der Artikel mit Preisen unter 1000:

```
List<Article> articles =
    List.of(new Article("A", 1500), new Article("B", 700), ...);
List<String> cheapArticleNames =
    articles.stream()
      .filter(a -> a.price < 1000.0)
      .sorted(Comparator.comparingDouble((Article a) -> a.price))
      .map(a -> a.name)
      .collect(Collectors.toList());
```

Wir sehen, dass Streams dem Gestaltungsprinzip von Monaden folgen, wie wir sie in Abschnitt 6.3 besprochen haben. `Stream` ist dabei die monade Behälterklasse. Zuerst wird mit der Operation `stream` ein `Stream`-Objekt erzeugt. Dieses wird jeweils mit den höheren Funktionen `map` und `filter` in weitere Streams abgebildet, bis schließlich mit einer Reduktionsoperation, hier als `collect` ausgeführt, ein Ergebnis erzeugt wird.

7.1.2 Externe vs. interne Iteration

Betrachten wir nun den Unterschied zwischen einer externen Iteration bei einer imperativen Lösung und einer internen Iteration bei Streams. Wir verwenden zur Veranschaulichung das obige Beispiel, aber diesmal ohne die Sortieroperation:

```
List<String> cheapArticleNames =
    articles.stream()
      .filter(a -> a.price < 1000.0)
      .map(a -> a.name)
      .collect(Collectors.toList());
```

Eine imperative Lösung der gleichen Aufgabe würde so aussehen:

```
List<String> cheapArticleNames = new ArrayList<String>();
for (Article a : articles) {
  if (a.price < 1000.0) {
    cheapArticleNames.add(a.name);
  }
}
```

Bei der imperativen Variante ist die Iteration über die Elemente im Anwenderprogramm in der Form der for-Schleife implementiert. Die Elemente werden im Schleifenrumpf verarbeitet. Bei der funktionalen Variante ist die Schleife im Stream verborgen. Die Verarbeitungsschritte werden außerhalb der Schleife definiert.

Bei der funktionalen Variante werden die einzelnen Verarbeitungsschritte in deklarativer Weise in der Form von Aggregatsfunktionen angegeben und beziehen sich somit auf den Stream als Ganzes. Es sind damit wesentliche Vorteile verbunden:

- Das Programm enthält keine Seiteneffekte.
- Soll ein neuer Schritt eingefügt werden, kann man das einfach durch Anfügen eines weiteren Schrittes erreichen (siehe unser erstes Beispiel aus Abschnitt 7.1).
- Die Iteration ist intern und kann damit für den Stream spezifisch optimiert werden. Auch kann man, wenn grundsätzlich die Anwendung für eine parallele Verarbeitung geeignet ist, einfach von einer sequentiellen Verarbeitung zu einer parallelen Verarbeitung übergehen. Wir werden das in Kapitel 8 sehen.

Im Gegensatz dazu hat die externe Iteration den Vorteil, dass man im Anwenderprogramm die volle Kontrolle über die Iteration hat. Diese muss man naturgemäß bei der internen Iteration aufgeben.

7.1.3 Bedarfsauswertung

Eine ganz wesentliche Eigenschaft von Streams ist, dass die Zugriffe auf die Elemente nach Bedarf erfolgen. Eine Auswertung nach Bedarf haben wir bereits in Abschnitt 4.6 besprochen. Streams werden nach dem gleichen Prinzip verarbeitet.

Wir wollen im Folgenden die Verarbeitung bei Streams mit einer eifrigen Auswertung bei Listen vergleichen. In Abschnitt 4.1 haben wir höhere Funktionen für die funktionale Liste `FList` eingeführt. Mit `FList` kann man das Filtern der Artikel und die Abbildung auf den Namen somit ganz ähnlich implementieren:

```
FList<Article> articles =
    FList.of(new Article("A", 1500), new Article("B", 700), ...);
FList<String> cheapArticleNames =
    articles
        .filter(a -> a.price < 1000.0)
        .map(a -> a.name);
```

Abbildung 7–1 veranschaulicht die Verarbeitung für eine Beispielliste mit mehreren Artikeln. Mit der `filter`-Operation wird zuerst eine Liste mit den Artikeln mit einem Preis kleiner 1000 und mit `map` eine Liste mit den Namen dieser Artikel erzeugt.

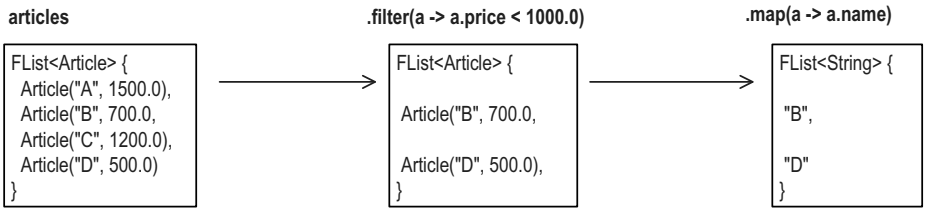

Abb. 7–1 *Strikte Verarbeitung der Liste von Artikeln bei FList*

Die Verarbeitung mit Streams funktioniert aber wie in Abbildung 7–2 gezeigt. Die Streams sind vorerst passiv und warten untätig, bis jemand auf die Elemente zugreift. Die Zugriffe gehen somit von der `collect`-Operation aus, die die Ergebnisliste aufbauen will und sich dafür die Elemente holt. Damit werden die Streams aktiv, liefern nacheinander die benötigten Elemente und wenden auf diese die Verarbeitungsschritte an. Dabei werden keine Zwischenlisten erzeugt. Bei sehr vielen Elementen kann das ein entscheidender Vorteil sein.

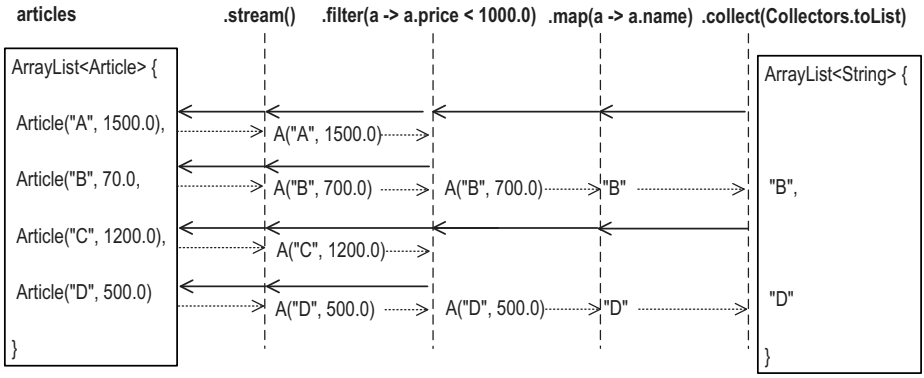

Abb. 7–2 *Faule Verarbeitung der Liste von Artikeln mit Streams*

Der wirkliche Vorteil wird aber erst deutlich, wenn wir unser Beispiel etwas ändern und nicht die Liste der günstigen Artikel bilden, sondern nur den ersten Artikel mit einem Preis kleiner 1000 benötigen. Das können wir für `FList` durch einen Zugriff auf das erste Element erreichen (dass die Liste leer sein könnte, wollen wir hier einmal ignorieren):

```
String cheapArticleName =
    articles.filter(a -> a.price < 1000.0)
            .map(a -> a.name)
            .head();
```

Es ergibt sich ein Verhalten wie in Abbildung 7–3. Es wird zuerst mit der `filter`-Operation eine Liste mit den Artikeln mit Preis kleiner 1000 gebildet, dann mit `map` eine Liste der Namen, und von dieser Liste wird das erste Element zurückgegeben. Vor allem bei langen Listen ist dies viel Aufwand, um ein einziges Element zu ermitteln.

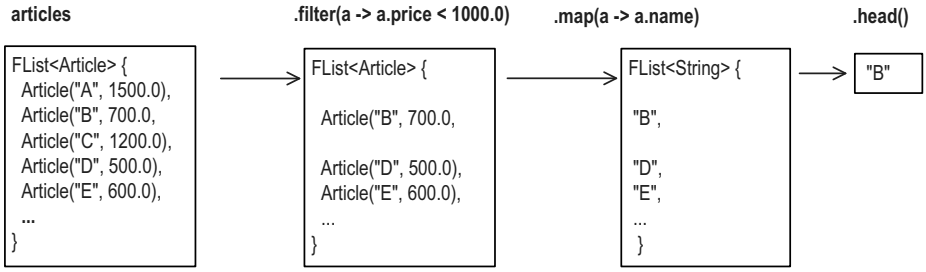

Abb. 7–3 *Zugriff auf das erste günstige Element bei* `FList`

Bei Streams gibt es eine Operation `findFirst`, die das erste Element eines Streams in einem Optional liefert. Damit ergibt sich für den Zugriff auf den ersten Artikel folgende Stream-Anweisung:

```
Optional<String> cheapArticleName =
    articles.stream()
            .filter(a -> a.price < 1000.0)
            .map(a -> a.name)
            .findFirst();
```

Abbildung 7–4 zeigt den wesentlichen Unterschied, der sich durch die Bedarfsauswertung ergibt. Die Operation wird hier von `findFirst` angestoßen, die genau ein Element benötigt. Der erste Zugriff wird durch `filter` noch weggefiltert, das zweite Element wird aber durch `filter` durchgereicht. Das von `findFirst` benötigte Element wird geliefert und die Berechnung terminiert. Statt alle Elemente zu filtern und abzubilden, ist insgesamt nur auf zwei Elemente zugegriffen worden.

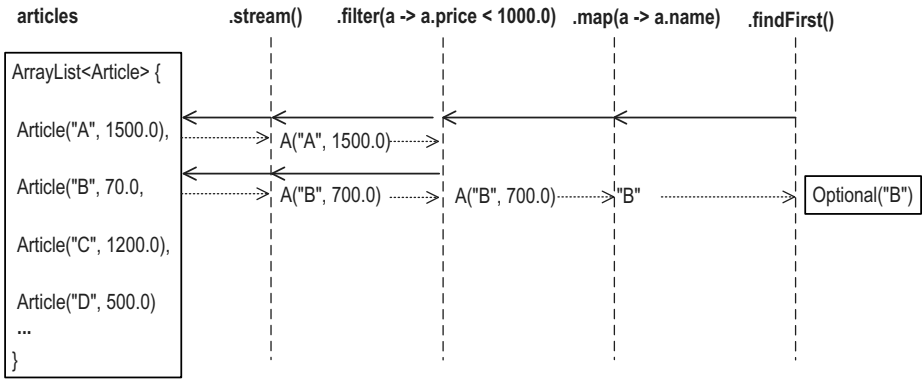

Abb. 7–4 *Faule Verarbeitung bei Streams bei Zugriff auf das erste teure Element*