Fachhochschule
Münster University of
Applied Sciences

Die Java Stream API

Funktionale Programmierung mit der *Stream API* des *JDK 1.8*

Prof. Dr. Nikolaus Wulff

Funktionale Programmierung

- Neben der Collection API mit **default** Methoden ist als weitere Neuerung eine Stream API ins JDK v1.8 integriert worden.
- Jede Java Collection liefert per **default** **stream()** Methode einen ***java.util.stream.Stream*** zurück.
- Mittels eines **Stream** ist Funktionale Programmierung in Java gerade für große Datenmengen möglich.
- Während Collections reine Datenkontainer sind, verwenden Streams Algorithmen, die auf den Daten z.B. einer Collection effizient operieren.

Stream Programmiermodel

- Die funktionale Stream Programmierung erfolgt im Prinzip in drei Schritten:

1. Der Erzeugung des Stream.
2. Einer Abfolge von Operationen für die Elemente.
3. Einer Abschließenden Terminierung mit Ergebnis.

Quelle => Stream => OP_1 => OP_2 => … => OP_n => Ergebnis

- Die Abfolge der Operationen OP_k auf den Daten lässt sich per Stream API sehr leicht parallel ausführen.

Schleifenabarbeitung

- Eine Liste von Wörtern soll in Abhängigkeit von der Wortlänge auf der Konsole ausgegeben werden.
- Der Standardweg über einen Iterator sieht so aus:

```
List<String> list = ...
Iterator<String> iterator = list.iterator();
while (iterator.hasNext()) {
    String s = iterator.next();
    if (s.length()>wordLength)
        System.out.printf("%s %s \n", Thread.currentThread(), s);
}
```

- Das Codefragment gibt alle Zeichenketten größer als die Variable wordLength auf der Konsole aus und zeigt auch an in welchem Thread dies geschieht.
- Eine Parallelisierung ist nur schwer möglich...

Moderne for-each-Schleife

- Seit dem JDK v1.5 wird meistens die while gegen die modernere for-each Schleife ersetzt:

```
List<String> list = ...
for (String s : list) {
    if (s.length()>wordLength)
        System.out.printf("%s %s \n", Thread.currentThread(), s);
}
```

- Der Code sieht kompakter aus, der Compiler realisiert intern die while Schleife des vorherigen Beispiels.
- Die aufwändige Angabe der printf Anweisung lässt sich seit dem JDKv1.8 als Lambda Ausdruck kapseln.

```
Consumer<Object> printer = s ->
          System.out.printf("%s %s \n",
                  Thread.currentThread(), s.toString());
```

forEach Methode

- Seit dem JDKv1.8 besitzen alle Collections eine default **forEach** Methode zur Implementierung einer Schleife, die einen Consumer erwartet.

```
List<String> list = ...
list.forEach((s) -> {
            if (s.length()>wordLength)
                        printer.accept(s);
        });
```

Zwei Consumer per λ Ausdrücke

- Eine Parallelisierung der Aufgabe kann einfach durch einer andere Implemtierung der **forEach** Methode innerhalb einer spezialisierten Liste realisiert werden … oder aber durch die Stream API.

Stream per VarArgs

- Streams lassen sich direkt per Varargs erzeugen:

```
Stream<String> stream = Stream.of("Die", "neue", "Stream", "API");
```

- Hierzu bietet die Schnittstelle Stream die statische Methode of an mit der Signatur:

```
static <T> Stream<T>    of(T... values)
```

- Diese generische statische Schnittstellen Methode erlaubt eine einfache Konstruktion eines Stream.

Streams aus Feldern

- Eine weitere Möglichkeit der Erzeugung eines Stream ist per Hilfsmethode aus der Klasses Arrays:

```
String[] array = { "Die", "neue", "Stream", "API" };
Stream<String> stream = Arrays.stream(array);
```

  - Für primitive Daten wie **`double`** oder **`int`** gibt es spezialisierte Stream Klassen, z.B. DoubleStream...

- Es ist somit möglich ein Stream direkt aus einem Feld oder per VarArgs zu erzeugen und per Collection, wie das nächste Beispiel zeigt.

Von Collection zu Stream

- Jede Collection lässt sich in einen Stream überführen auf dem dann die Aufgaben abgearbeitet werden:

```
List<String> list = ...
Stream<String> stream = list.stream();
stream.forEach((s) -> {
    if (s.length()>wordLength)
            printer.accept(s);
});
```

- Auch Streams stellen eine forEach Schleife zur Verfügung, ähnlich den Collections.
- Auf dem ersten Blick ist nicht viel gewonnen, doch Streams lassen sich verketten, erlauben effizientes Filtern, weitere Algorithmen und Parallelisierung.

Stream mit Filter

- Jeder Stream hat eine filter Methode, die mittels eines Prädikats – meist ein Lambda Ausdruck – parameterisiert wird.

```
List<String> list = …
Stream<String> stream = list.stream();
stream.filter(s -> s.length()>wordLength)
        .forEach(s -> {
            printer.accept(s);
        });
```

- filter ist der forEach Methode vorgeschaltet.
- Nur die Elemente, die der Filterbedingung genügen werden überhaupt in der Schleife ausgegeben.

Stream mit Map

- Jeder Stream kennt eine funktionale Abbildung aller Elemente per map Funktion:

```
List<String> list = …
Stream<String> stream = list.stream();
stream.filter(s -> s.length()>wordLength)
        .map(s -> s.toUpperCase())
        .forEach(s -> {
            printer.accept(s);
        });
```

- map und filter lassen sich hintereinander verketten. Sie liefern wieder einen Stream für weitere Verarbeitung zurück. So lässt sich effizient eine Pipes & Filter Architektur realisieren.

Streams parallelisieren

- Stream lassen sich sehr elegant und für den Programmier fast transparent parallelisieren ohne das filter oder die forEach Schleife zu verändern:

```
List<String> list = …
Stream<String> stream = list.parallelStream();

stream.filter(s -> s.length()>wordLength)
        .forEach(s -> {
            printer.accept(s);
        });
```

- Der Aufruf von parallelStream anstatt stream sorgt dafür das alle Stream-Operationen multithreaded ausgeführt werden.

Stream nachträglich parallelisieren

- Auf der vorhergehenden Folie wurde der Stream parallel erzeugt, aber auch ein schon existierender Stream lässt sich nachträglich Parallelisieren per parallel Methode:

```
List<String> list = …
Stream<String> stream = list.stream().parallel();
        stream.filter(s -> s.length()>wordLength)
                .forEach(s -> {
                    printer.accept(s);
                });
```

- Diese Methode liefert eine Stream Instanz zurück, die den ursprünglichen Stream parallel ausführt.
- Somit lässt sich ein sequentieller Algorithmus einfach und transparent parallelisieren.

Stream Operationen

Ein Stream stellt einen Satz von Operationen zur Verfügung, die sich mittels @FunctionalInterface parametrisieren lassen.

Es gibt prinzipiell zwei Typen von Operationen:

Intermediäre Stream-Operationen

- Diese modifizieren die Daten und liefern wieder einen Stream zurück, sie lassen sich daher verketten. Die Stream Verarbeitung wird weiter fortgesetzt.

Terminale Stream-Operationen

- Ihr Aufruf stellt das Ende einer Stream Verarbeitungs-kette da und liefert das Endresultat.

Intermediäre Operationen

- *filter*(*Predicate*) Daten nach Prädicat gefiltert
- *map*(*Function*) Datentransformation per Abbildung
- *distinct*() Stream um Dubletten bereinigt
- *limit*(*long n*) Stream wird auf Länge n begrenzt
- *peek*(*Consumer*) Daten per Consumer verwenden
- *skip*(*long n*) die nächsten n Daten überspringen
- *sorted*() liefert Stream mit sortierten Daten

Alle Intermediären Operationen liefern als Ergebnis wieder einen Stream zurück und lassen sich verketten. Sie arbeiten *lazy*, die Verarbeitung wird erst durch eine terminale Operation angestossen.

Terminale Operationen

- *collect*(*Collector*) Zusammenfassen von Daten
- *count*() Anzahl der Daten im Stream
- *forEach*(*Consumer*) Schleife über alle Daten
- *max*/*min*(*Comperator*) Maximum/Minimum
- *reduce*(*BinaryOperator*) Reduktion der Daten

Terminale Operationen sind das letzte Glied in der Verarbeitungskette eines Stream. Sie liefern keinen weiteren Stream zurück, lassen sich nicht verketten und der Stream ist „verbraucht“, es lassen sich keine weiteren Operationen mehr ausführen. Sie stoßen die Verarbeitung der intermediären Operationen an.

Filter-Map-Reduce

- Big-Data erfordert häufig Algorithmen, die aus gefilterten Daten ein Ergebnis berechnen.
- Der Google filter-map-reduce Ansatz hilft:

```
Stream<String> stream = list.stream(); //.parallel(); // optional
Optional<String> result =
        stream.filter(s -> s.length() > wordLength)
                .map(s -> s.toUpperCase())
                .reduce((s1, s2) -> s1 + s2);
String msg = result.get();
```

- Die *reduce* Operation konkateniert die Zeichenketten bis alle gefilterten Zeichen zusammengesetzt sind.

Zusammenfassung

- Streams operieren effizient auf großen Datenmengen einer zugrunde liegenden Container Klasse.
- Sie erlauben es beliebige *filter* oder *map* Algorithmen zu verschachteln und sowohl seriell oder parallel auszuführen.
- Algorithmen können seit JDK v1.8 intuitiv als Lambda Ausdrücke bereitgestellt oder aber auch als „große selbstgeschriebene Funktionen“ programmiert werden.
- Mittels der Stream API bewegt sich Java in Richtung Funktionale Programmierung.