Neues in Java 8

Collections-Verarbeitung

Collections-Verarbeitung

- Externe vs. Interne Iteration
- Unterstützung der Stream API
- Funktionale Verarbeitung von Collections
- Performance-Verbesserung für HashMaps mit Key-Kollision

Collections-Verarbeitung Externe vs. Interne Iteration

- Externe Iteration
  - Klassische Schleifenkonstrukte (Iterator, for, while, …)
  - Aufrufer kontrolliert die Iteration
- Interne Iteration
  - Erweiterung des Collections Frameworks in Java 8
  - Collection bzw. das Framework kontrolliert die Iteration
  - `.forEach()`-Methode übernimmt Callback-Methoden mit der Verarbeitungslogik

Collections-Verarbeitung Externe vs. Interne Iteration

- Externe Iteration
  - Explizite Kontrolle, aber …
  - Häufige Fehlerquelle (Index, Off-by-one, …)
  - Nur teilweise „fail fast“ bei Änderung der Collection während der Iteration
- Interne Iteration
  - „fail fast“-Verhalten, lässt keine Modifikation der Collection während Iteration zu.
  - Verarbeitungslogik kann als „code as data“ übergeben werden → Lambda / Methodenreferenz
  - Durch Framework parallelisierbar!

Collections-Verarbeitung Erweiterungen

- Interface `Predicate<T>`
- … z.B. für Methode `Collection.removeIf()`
- Interface `UnaryOperator<T>`
- … z.B. für Methode `List.replaceAll()`
- Verarbeitung nach dem Stream-Modell
  - Create-Operationen
  - Intermediate-Operationen
  - Terminal-Operationen

Stream API
Create-Operationen

- Erzeugung von Streams aus Arrays und Listen
- Erzeugung von Streams mit vorgegebenen Wertebereichen
  - Generischer Stream
  - Spezialisierte Streams, z.B. IntStream
- Erzeugung von endlosen Streams
  - Mit Iterator-Funktion für numerische Streams
  - Mit Generator-Funktion und Supplier für generische Streams
- Erzeugung von leeren Streams

Stream API Intermediate-Operationen

- Zustandslose Intermediate-Operationen:
  - `filter()`
  - `map()`
  - `flatMap()`
  - `peek`
- Zustandsbehaftete Intermediate-Operationen:
  - `distinct()`
  - `sorted()`
  - `limit()`
  - `skip()`

Stream API Terminal-Operationen

- Allgemeine Operationen:
  - `forEach()`
  - `toArray()`
  - `collect()`
  - `reduce()`
  - `min()`
  - `max()`
  - `count()`
- Short-Circuiting (bricht ggf. vor letztem Element ab):
  - `anyMatch()`
  - `allMatch()`
  - `findFirst()`
  - `findAny()`

Stream API
Kollektoren

- Interface für das „Einsammeln“ oder Aggregieren von (End-)Ergebnissen
- Konzept mit drei Bestandteile:
  - `Supplier<R>` stellt die Ergebnis-Struktur vom Typ R bereit
  - `BiConsumer<R, E>` „accumulator“ bildet Stream-Element vom Typ E in R ab
  - `BiConsumer<R, R>` „combiner“ führt Teil-Ergebnisse vom Typ R zu einem Gesamt-Ergebnis vom Typ R zusammen

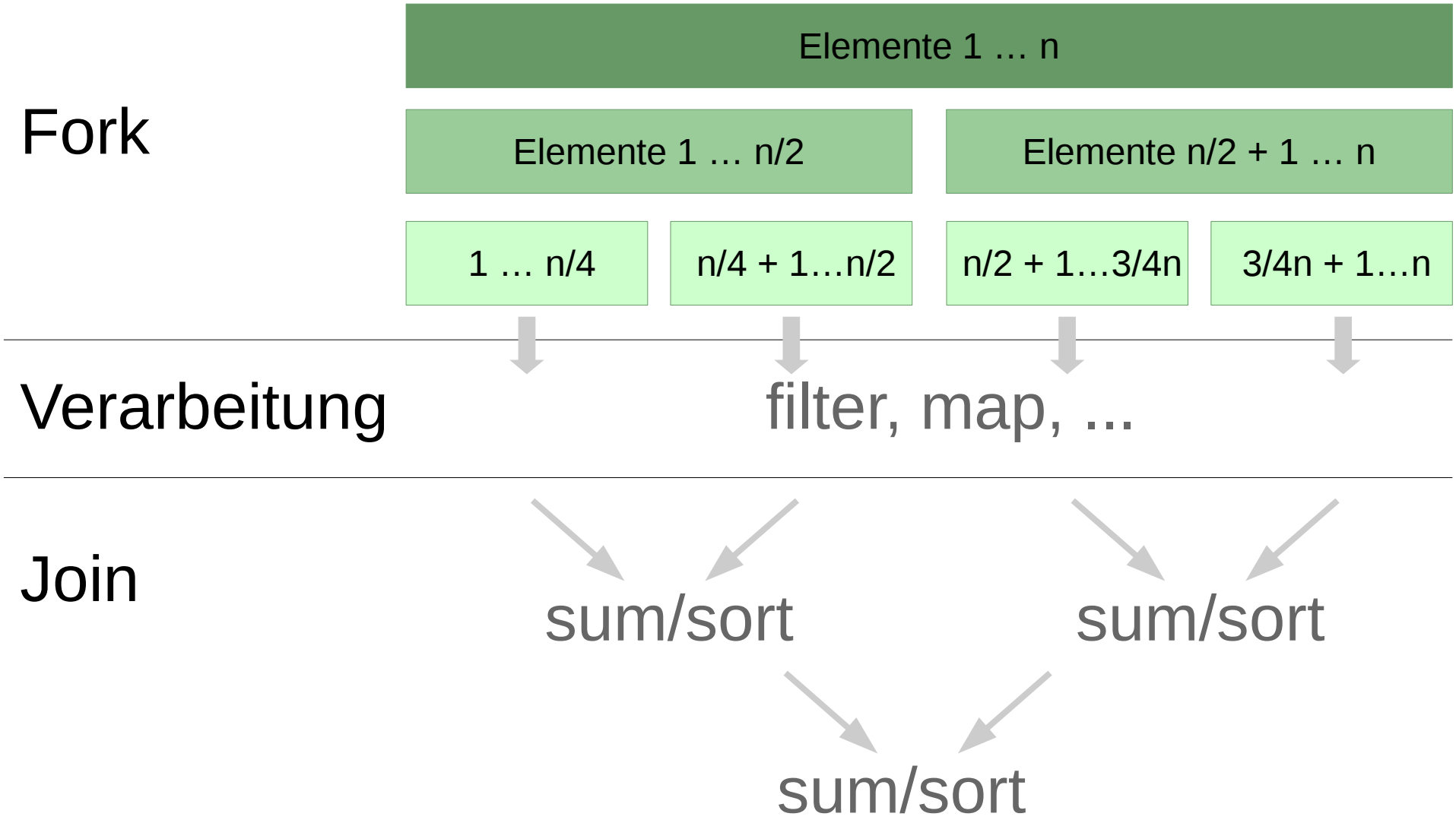
Stream API Kollektoren

- Standard-Kollektoren
  - `counting()`
  - `summing()`
  - `averaging()`
  - `maxBy()`
  - `minBy()`
  - `summarizingInt()`
  - `toList()`
  - `...`
- Gruppierende Kollektoren
  - `toMap()`
  - `groupingBy()`
  - `partitioningBy()`

Parallel-Verarbeitung Eckpunkte I

- Einfache Erzeugung von parallelen Streams
- Das Framework regelt die parallele Ausführung der Stream-Operationen mit internem Fork-Join-Pool
- ACHTUNG! Für eine Stream-Pipeline wird letztendlich immer ein einheitlicher Verarbeitungsmodus verwendet, obwohl in der Definition zwischen seriell und parallel gewechselt werden kann!
- ACHTUNG! Zustandsloses Arbeiten ist wichtig!
- forEach() arbeitet nicht wie gewohnt (s.u.)!

Parallel-Verarbeitung Eckpunkte II



Parallel-Verarbeitung Eckpunkte III

- Standardmäßig wird ein Thread pro CPU-Kern verwendet
- Keine Vorteile bei Single-Core-Systemen
- Parallele Collection-Verarbeitung ermöglicht extrem einfaches Multithreading mit minimalen Änderungen im Code
- … ist aber nicht so flexibel wie explizites Thread-Handling.

Parallel-Verarbeitung Praxis-Tipps

- Prüfen: „Ist meine Stream-Pipeline wirklich parallelisierbar?“
  - Sicherstellen das die parallele Verarbeitung dasselbe Ergebnis bringt wie serielle Verarbeitung!
- Prüfen: „Ist die parallele Ausführung wirklich schneller?“
  - … in realen Ausführungsumgebungen
  - … mit realen Datenmengen!
  - Achtung! Hier spielen viele Faktoren eine Rolle, z.B. CPU(-Architektur), JIT, …

Parallel-Verarbeitung Voraussetzungen

- Operationen müssen assoziativ sein
  - Addition und Multiplikation: für Integer und Long ja
  - Bei Double-Werten kann eine Änderungen in Gruppierung oder Reihenfolge der Operationen Abweichungen im Ergebnis erzeugen!
  - Subtraktion, Division und andere Operationen: Nein!
- Operationen dürfen keine Seiteneffekte auf globalen Daten erzeugen!
- Binäre Operatoren müssen zustandslos sein!

Parallel-Verarbeitung API

- Parallele Verarbeitung definieren:
  - `List.parallelStream()`
  - `Stream.parallel()`
- Collector-Klassen für parallele Ausführung:
  - `Collectors.groupingByConcurrent()`
  - `Collectors.toConcurrentMap()`

„Wo bringt Parallel-Verarbeitung identische Ergebnisse?“ I

- `sorted()`, `min()`, `max()` – Nein.
- `findFirst()` – Nein.
- `map()`, `filter()` – teilweise:
  - `map()` bringt dieselben Einzel-Ergebnisse,
  - … aber: die Ergebnisstreams müssen nicht identisch geordnet sein.
- `allMatch()`, `anyMatch()`, `noneMatch()`, `count()` – Ja.
- `sum()`, `average()` – Ja für `IntStream` und `LongStream`, vielleicht bei `DoubleStream`

„Wo bringt Parallel-Verarbeitung identische Ergebnisse?“ II

- `reduce()` – Ja, wenn:
  - … wenn keine Seiteneffekte auf globalen Daten entstehen, und
  - … wenn die Operation assoziativ ist,
    d.h. wenn Reihenfolge und Gruppierung in der Verarbeitung keine Rolle spielen.

  - Bei Double-Werten können *exakt* gleiche Ergebnisse nicht garantiert werden!

Neues in Java 8 Collections-Verarbeitung

Vielen Dank für Ihre Aufmerksamkeit!
Noch Fragen?

Quellen

- API: https://docs.oracle.com/javase/8/docs/api/index.html
- Oracle Java 8 Release Notes
  http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html
- Inden, M.: Java 8 – Die Neuerungen, dpunkt 2015, 2. Auflage.