Lernskript: Die Java Stream API

Die in Java 8 eingeführte Stream API hat die Art und Weise, wie man mit Datensammlungen arbeitet, revolutioniert. Sie ermöglicht einen deklarativen, funktionalen Stil zur Datenverarbeitung.

---

1. Das Konzept: Ein Fließband für Daten

Ein **Stream** ist keine Datenstruktur, die Elemente speichert, sondern ein Datenfluss aus einer Quelle. Man kann ihn sich wie ein **Fließband** vorstellen:

1. **Quelle (Source):** Die Daten werden auf das Fließband gelegt (z.B. aus einer `List`).
2. **Zwischenoperationen (Intermediate Operations):** Das Fließband durchläuft verschiedene Bearbeitungsstationen, die die Daten filtern, umwandeln oder sortieren.
3. **Endoperation (Terminal Operation):** Am Ende des Fließbands wird ein Ergebnis erzeugt oder eine Aktion ausgeführt.

Streams verändern dabei nie die ursprüngliche Datenquelle.

---

2. Die drei Teile einer Stream-Pipeline

Quelle (Source)

Man erzeugt einen Stream typischerweise aus einer Collection:

```
List<String> meineListe = List.of("A", "B", "C");
Stream<String> meinStream = meineListe.stream();



Zwischenoperationen (Intermediate Operations)

Diese Operationen geben immer einen neuen Stream zurück und sind "lazy" (faul),
d.h., sie werden erst bei Aufruf einer Endoperation ausgeführt. Man kann sie
verketten.

    filter(Predicate<T> predicate): Lässt nur Elemente durch, die eine Bedingung
erfüllen.

    map(Function<T, R> mapper): Wandelt jedes Element von Typ T in Typ R um.

    sorted(): Sortiert die Elemente.

    distinct(): Entfernt Duplikate.

Endoperation (Terminal Operation)

Diese Operation konsumiert den Stream und erzeugt ein Endergebnis. Sie startet die
```

gesamte Pipeline.

    collect(Collector): Sammelt die Elemente in einer neuen Collection (z.B. Collectors.toList()).

    forEach(Consumer<T> action): Führt für jedes Element eine Aktion aus.

    count(): Zählt die Anzahl der Elemente.

    anyMatch(Predicate<T> predicate): Prüft, ob mindestens ein Element eine Bedingung erfüllt.

3. Klassisch vs. Modern: Ein Beispiel

Aufgabe: Finde alle einzigartigen Namen, die länger als 4 Buchstaben sind, konvertiere sie zu Großbuchstaben und speichere sie sortiert in einer neuen Liste.

Modern (mit der Stream API):```java
List<String> namen = List.of("Anna", "Peter", "Chris", "Anna", "Sebastian");
List<String> ergebnis = namen.stream() // 1. Quelle
.filter(name -> name.length() > 4) // 2. Zwischenop
.distinct() // 2. Zwischenop
.map(String::toUpperCase) // 2. Zwischenop (mit Methodenreferenz)
.sorted() // 2. Zwischenop
.collect(Collectors.toList()); // 3. Endop
Generated code

Der moderne Weg ist deklarativ ("was" passieren soll) und oft deutlich lesbarer und kürzer.

---

4. Vertiefung (JavaMasta's Profi-Tipps)

Der `Collector` ist dein bester Freund
Die `Collectors`-Klasse ist eine Schatztruhe. Man kann damit nicht nur in eine `List` (`Collectors.toList()`) oder `Set` (`Collectors.toSet()`) sammeln, sondern auch Daten gruppieren (`Collectors.groupingBy()`) oder Strings verbinden (`Collectors.joining(", ")`).

`flatMap` - Der "Stream im Stream"-Entpacker
Wenn man eine verschachtelte Struktur hat (z.B. `List<List<String>>`) und eine einzige, flache Liste aller Elemente daraus machen will, ist `flatMap` die richtige Wahl. Es "bügelt" die inneren Streams zu einem einzigen großen Stream glatt.

Performance-Warnung: `parallelStream()`
Sei vorsichtig mit `.parallelStream()`. Es versucht, die Arbeit auf mehrere CPU-Kerne zu verteilen, was aber bei einfachen Operationen durch den Verwaltungsaufwand oft langsamer ist. Es birgt zudem Risiken bezüglich der Thread-Sicherheit. Regel: Immer `.stream()` benutzen, es sei denn, man hat nach Messungen einen triftigen Grund für Parallelität.