Streams

Programmiermethodik 2

Wiederholung

- Lambdas
- Default-Methoden
- Iteration
- Collections-Erweiterungen

Ausblick

Agenda

- Streams von Elementen
- Serialisierung

Streams

Listen vs. Streams

- bisher kennengelernt (u.a.): Listen, um mehrere Elemente "hintereinander" / "in einer Kette" abzulegen
- manche Anwendungsfälle
    - sequenzielle Verarbeitung einer solchen "Kette"
    - z.B. nicht alle Elemente im Hauptspeicher
    - Beispiel: Streaming eines Videos aus dem Internet

Streams in Java

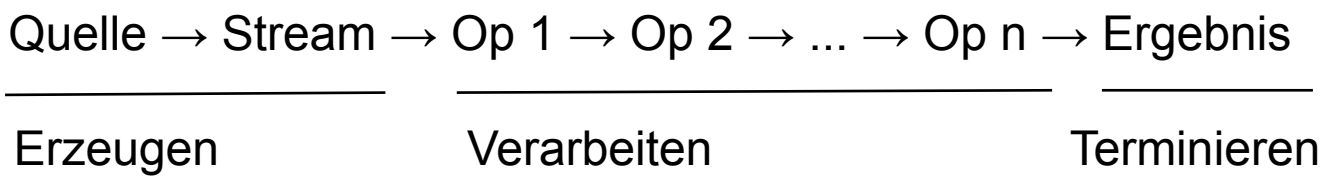
- Streams von Informationen/Daten
- Streams von Elementen (Java 8)

Streams von Elementen (Java 8)

Zum Nachlesen:
Michael Inden: Java 8 - Die Neuerungen: Lambdas, Streams, Date And Time API und JavaFX 8 im Überblick, dpunkt Verlag

Java 8 Streams

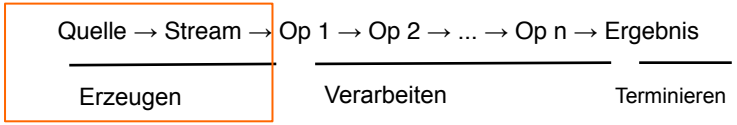
- java.util.stream.Stream
- Muster bei der Verarbeitung mit Stream:
  - Erzeugen
  - Verarbeiten (beliebig viele Operationen)
  - Terminieren



- Beispiel:

```
List<Person> erwachsene = personen.stream(). // Erzeugen
filter(Person::istErwachsen). // Verarbeiten
collect(Collectors.toList()); // Terminieren
```

Streams erzeugen



- Collections: Methode `stream()`
  - liefert einen Stream

```
String[] namenArray = { "Karl", "Ralph", "Andi",
     "Andy", "Mike" };
List<String> namen = Arrays.asList(namenArray);
Stream<String> namenStream = namen.stream();
```

- Utility-Klasse Arrays kann auch Streams erzeugen

```
Stream<String> namenStream = Arrays.stream(namenArray);
```

- Ausblick: Parallelisierung
  - Collections-Container können auch parallelen Stream erzeugen

```
Stream<String> namenStream = namen.parallelStream();
```

  - ansonsten: Stream intern parallel machen

```
Stream<String> stream = stream.parallel();
```

Stream aus Werten erzeugen

- Stream kann auch aus einer Liste von Werten generiert werden
  - statische Methode `of()` der Klasse Stream
- Beispiele

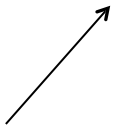
```
Stream<String> namen = Stream.of("Jan", "Hein", "Klaas");
Stream<Integer> zahlen = Stream.of(1, 4, 7, 7, 9, 7, 2);
```

- Stream aller Zahlen in Bereich

```
IntStream zahlen = IntStream.range(0, 100);
```

- Stream der Indizes der Buchstaben in einem String

```
IntStream zeichen = "Dies ist ein Text".chars();
```



Es gibt eigene Stream-Klassen für primitven Datentypen (int, long, double, alle anderen Typen sind dazu kompatibel).

Unendliche Streams

- auch unendliche Streams (Folgen) sind vorhanden
- Beispiel: Folge der ganzen Zahlen
- Definition:
    - Startelement
    - Berechnungsvorschrift für Nachfolge-Element
- Beispiel:

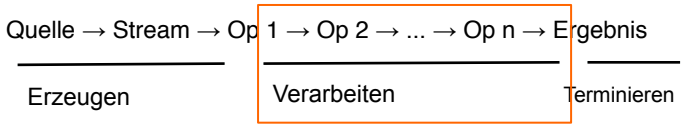
```
IntStream zahlen = IntStream.iterate(0, x -> x + 1);
```

- aber: tatsächliche Verwendung muss begrenzt werden
    - Methode: limit()
- Beispiel:

```
zahlen.limit(10) // liefert Stream der ersten 10 Elemente
```

Streams Verarbeiten

- Unterscheidung
  - zustandslose Varianten
  - zustandsbehaftete Varianten
- Beispiele
  - Filterung = zustandslos, Entscheidung für jedes Element einzeln
  - Sortierung = zustandsbehaftet, vergleich zwischen Elementen



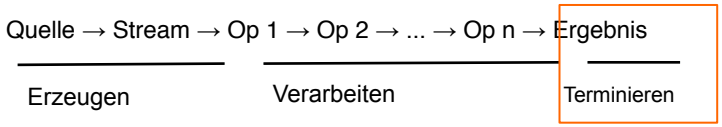
Zustandslose Verarbeitung

map()	Transformiert Elemente mithilfe einer Function<T,R> vom Typ T auf solche mit dem Typ R . Im Speziellen können die Typen auch gleich sein.
flatMap()	Bildet verschachtelte Streams als einen flachen Stream ab.
peek()	Führt eine Aktion für jedes Element des Streams aus. Dies kann für Debuggingzwecke sehr nützlich sein.

Zustandsbehaftete Verarbeitung

sorted()	Sortiert die Elemente eines Streams gemäß einem Sortierkriterium basierend auf einem Comparator<T>.
limit()	Begrenzt die maximale Anzahl der Elemente eines Streams auf einen bestimmten Wert. Dies ist eine Short-circuiting Operation.
skip()	Überspringt die ersten n Elemente eines Streams.

Terminierung



forEach()	Führt eine Aktion für jedes Element des Streams aus.
toArray()	Überträgt die Elemente aus dem Stream in ein Array.
collect()	Überträgt die Elemente aus dem Stream in eine Collection.
reduce()	Verbindet die Elemente eines Streams. Ein Beispiel ist die kommaseparierte Konkatenation von Strings. Alternativ kann man aber auch Summationen, Multiplikationen usw. ausführen, um einen Ergebniswert zu berechnen.

Terminierung

min()/max()	Ermittelt das Minimum/Maximum der Elemente eines Streams gemäß einem Sortierkriterium basierend auf einem Comparator<T>.
count()	Zählt die Anzahl an Elementen in einem Stream.
anyMatch()/ allMatch()/ noneMatch()	Prüft, ob es mindestens ein Element (alle Elemente/kein Element) gibt, das die Bedingung eines Predicate<T> erfüllt.
findFirst()/ findAny()	Liefert das erste (eine beliebiges) Element des Streams, falls es ein solches gibt.

Beispiel: Filter

```
public class Person {
  private int alter;
  private String name;
  private LocalDate geburtstag;
  public Person(String name, int alter) {
    this.name = name;
    this.alter = alter;
  }
  public Person(String name, LocalDate geburtstag) {
    this.name = name;
    this.geburtstag = geburtstag;
  }
  @Override
  public String toString() {
    return String.format("%s %s (%d)", name, geburtstag,
alter);
  }
  public boolean istErwachsen() {
    return alter >= 18;
  }
  public String getName() {
    return name;
  }
  private int getAlter() {
    return alter;
  }
  public LocalDate getGeburtstag() {
    return geburtstag;
  }
```

```
List<Person> personen = new ArrayList<>();
personen.add(new Person("Micha", 43));
personen.add(new Person("Barbara", 40));
personen.add(new Person("Yannis", 5));
Predicate<Person> istErwachsen = person ->
    person.istErwachsen();
personen.stream().filter(istErwachsen)
    .forEach(System.out::println);
// Alternative mit Methodenreferenz
personen.stream().filter(
    Person::istErwachsen).forEach(
    System.out::println);
```

- liefert

Micha (43)
Barbara (40)

Beispiel: Mapping

```
// Mapping von Personen-Stream auf Namen-Stream
personen.stream().map(person -> person.getName()).forEach(
    name -> System.out.print(name + " "));

// Mapping von Personen-Stream auf Alter-Stream
personen.stream().map(person -> person.getAlter())
    .forEach(alter -> System.out.print(alter + " "));
```

- liefert

Micha Barbara Yannis
und

43 40 5

Übung: Map

- Schreiben Sie einen Lambda-Ausdruck, der zum funktionalen Interface `Function<T,R>` kompatibel ist (und daher zusammen mit `map()` verwendet werden kann
- Der Lambda-Ausdruck wandelt eine Zeichenkette in ihre Länge um

Beispiel: Flat-Map

- Lesen von Zeilen aus einer Text-Datei:

```
List<String> zeilen = Arrays.asList(
      "In einem Loch im Boden, da lebte ein Hobbit."
      "Nicht in einem feuchten, schmutzigen Loch,"
      "wo es nach Moder riecht"
      "und Wurmzipfel von den Wänden herabhängen." );
Stream<String> zeilenStream = zeilen.stream();
```

- jeden String am Leerzeichen aufspalten:

```
Stream<Stream<String>> text = zeilenStream.map(zeile ->
Stream.of(zeile.split(" "));
```

Beispiel: Flat-Map

- liefert (Stream von Stream von Strings)

```
<<In>, <einem>, <Loch>, <im>, <Boden, >, <da>, <lebte>, <ein>,
<Hobbit.>, <<Nicht>, <in>, <einem>, <feuchten, >, <schmutzigen>,
<Loch,>, <wo ...
```

- besser: Verflachen = innere Streams auflösen (liefert Stream der Wörter)

```
Stream<Stream<String>> text = zeilenStream.flatMap(
      zeile -> Stream.of(zeile.split(" "));
```

Beispiel: "Reingucken"

- Stream kann nur einmal verarbeitet werden
  - Was mache ich, wenn ich einen Zwischenstand ansehen möchte?
  - Operation: peek()
    - Consumer<T> als Parameter
    - liefert neuen Stream zurück → kann weiterverarbeitet werden
- Beispiel

```
List<Person> personen = ...
Stream<Person> personenStream = personen.stream();
Stream<String> alleMikes = personenStream.
      filter(Person::istErwachsen).
      peek(System.out::println).
      map(Person::getName).
      filter(name -> name.startsWith("Mi"));
```

Beispiel: distinct() und sorted()

- Beispiel

```
Stream<Integer> zahlen = Stream.of(
      7, 1, 4, 3, 7, 2, 6, 5, 7, 9, 8);
```

- Sortieren

```
zahlen.sorted().forEach(zahl ->
      System.out.println(String.format("%d ", zahl)));
```

- Duplikate entfernen

```
zahlen.distinct().forEach(zahl ->
      System.out.println(String.format("%d ", zahl)));
```

Stream → Collection

- Rückverwandlung eines Streams in eine Collection
- Beispiel

```
List<Integer> alter = alterStream.collect(
     Collectors.toList());
List<String> name = namenStream.collect(
     Collectors.toCollection(ArrayList::new));
```

Elemente zusammenfassen: Reduce

- Zusammenfassen aller Elemente eines Streams
- jeweils paarweise
    - alle bisherigen + das nächste
- Terminiere reduce(T, BinaryOperator<T>)
    - erster Parameter: Identität
    - damit wird das erste Element vereint
- Beispiel: Summe eines int-Streams

```
IntStream zahlen = IntStream.range(4, 23);
int summe = zahlen.reduce(
      0, (zahl1, zahl2) -> zahl1 + zahl2));
```

Übung: Im-Juli-Geboren

- Gegeben ist folgender Datensatz:

Person hat Objektvariable geburtstag vom Typ LocalDate (mit Methode getMonth())

```
List<Person> personen =
  Arrays.asList(new Person("Stefan", LocalDate.of(1971, MAY, 12)),
    new Person("Micha", LocalDate.of(1971, FEBRUARY, 7)), new Person(
      "Andi Bubolz", LocalDate.of(1968, JULY, 17)), new Person(
      "Andi Steffen", LocalDate.of(1970, JULY, 17)), new Person(
      "Merten", LocalDate.of(1975, JUNE, 16)));
```

- Aufgaben
  - Filterung auf alle im Juli Geborenen
  - Extraktion des Namens
  - Ausgabe als kommagetrennte Liste (ein String)

Serialisierung

Zum Nachlesen:
Christian Ullenboom: Java ist auch eine Insel, Kapitel 17.10
http://openbook.rheinwerk-verlag.de/javainsel9/javainsel_17_010.htm

Serialisierung

- Problem
  - Speicherung von komplexen Objekten in einem Datenstrom
  - Datei, Netz, …
- bisher
  - Die interne Struktur jedes Objekts müsste komplett "nachprogrammiert" werden
  - Objektvariablen mit aktuellen Werten inkl. aller referenzierten Objekte
- Lösung
  - Serialisierung
  - Objekte werden "automatisch" in ein Byte-Format konvertiert und in einen Datenstrom geschrieben
- Deserialisierung
  - Java-Objekte werden aus einem Byte-Datenstrom gelesen und komplett wiederhergestellt

Serialisierung

- Voraussetzung:
  - zu serialisierendes Objekt muss Interface Serializable implementieren

- Vorgehensweise bei der Programmierung:
  - Erzeugen eines ObjectOutputStream mit Übergabe eines existierenden OutputStream an den Konstruktor
    - wohin soll das Objekt geschrieben werden?
  - Methode writeObject(Object obj) zum Schreiben eines Objekts aufrufen
    - ggf. mehrfach für mehrere Objekte

Serialisierung

- Folgende Daten werden in den Output-Stream geschrieben
  - Klasse des als Argument übergebenen Objekts
  - Signatur der Klasse
  - alle nicht-statischen, nicht-transienten Objektvariablen des Objekts
    - inkl. aller geerbten Objektvariablen
  - Objektvariablen können selbst wieder Objekte enthalten!

Serialisierung

- Beispiel: Serialisieren eines Objektes

```
try (ObjectOutputStream dateiStream = new ObjectOutputStream(
    new FileOutputStream(dateiname))) {
  dateiStream.writeObject(object);
} catch (IOException e) {
  System.out.println("Failed to serialize object");
}
```

Weitere Möglichkeiten

- Schreiben einzelner primitiver Datentypen in den selben Stream

```
void writeBoolean (boolean value)
void writeBytes (String string)
void writeDouble (Double double)
```

- Ausschließen einzelnen Objektaviablen
  - werden nicht serialisiert
  - Markierung um Schlüsselwort transient

```
private transient String password;
```

Deserialisierung

- Voraussetzungen:
  - class – Datei des Objekts muss auffindbar sein
  - sonst sind die Methoden des Objekts nicht bekannt
  - Klassenstruktur vorliegenden Objekts (Byteformat) und aktuelle Klassendefinition müssen zueinander kompatibel sein

Serialisierung

- Beispiel: Deserialisieren eines Byte-Streams

```
Object objekt = null;
try (ObjectInputStream dateiStream = new ObjectInputStream(
    new FileInputStream(filename))) {
  objekt = dateiStream.readObject();
} catch (ClassNotFoundException | IOException e) {
  System.out.println("Failed to deserialize file.");
}
return objekt;
```

Vorgehensweise der JVM für jedes Objekt

- neues Objekt erzeugen
- Objektvariablen mit Default-Werten initialisieren
- aber keinen Konstruktor aufrufen!
- serialisierte Daten lesen
- entsprechenden Objektvariablen zuweisen

Versionsprüfung

- mögliches Problem
  - Klassendefinition eines serialisierten Objekts hat sich verändert hat
  - Deserialisierung schlägt fehl
  - Objekt kann nicht mehr gelesen werden!

- Java-Lösung
  - writeObject erzeugt aus den Klasseninformationen eine eindeutige Versionsnummer serialVersionUID
  - speichert diese mit ab
  - falls in der Klassendefinition eigene Konstante existiert, wird dieser Wert verwendet
  - static final long serialVersionUID = ..
  - Versionsnummern stimmen bei Deserialisierung nicht überein?
  - JVM verweigert Deserialisierung

Zusammenfassung

- Streams
- Streams von Elementen
- Serialisierung

Quellen

- Dieser Satz Folien basiert teilweise Teilen auf folgender Literatur: Michael Inden: Java 8 – Die Neuerungen, dpunkt Verlag, 2014
- Dieser Satz Folien basiert teilweise auf Vorlesungsfolien von Prof. Martin Hübner, Hochschule für Angewandte Wissenschaften Hamburg