# Objektorientierte Programmierung

## 12. Streams

Prof. Dr. Marcel Luis

Westf. Hochschule

SS 2024

# Check-in

- Die Definition und Anwendung funktionaler Schnittstellen ist bekannt.

# Was ist ein Stream?

- Ein *Stream* ist eine Folge von Elementen, auf die sequentielle und parallele Operationen angewendet werden können.

# Was ist ein Stream?

- Ein *Stream* ist eine Folge von Elementen, auf die sequentielle und parallele Operationen angewendet werden können.
- Gab's da nicht schon mal so etwas Ähnliches?

# Was ist ein Stream?

- Ein *Stream* ist eine Folge von Elementen, auf die sequentielle und parallele Operationen angewendet werden können.
- Gab's da nicht schon mal so etwas Ähnliches?
- Ja, die Schnittstelle `NumberSequence` aus dem Praktikum.

# Schnittstelle NumberSequence

```
public interface NumberSequence {

  /**
   * Returns whether there is a next element in this sequence.
   *
   * @return true if there is a next element
   */
  boolean hasNext();

  /**
   * Returns the next element in this sequence if there is any.
   * Otherwise, a {@code NoSuchElementException} is thrown.

   * @return the next element of this sequence
   * @throws NoSuchElementException if this sequence has no more
   * elements
   */
  int next() throws NoSuchElementException;
}
```

# Schnittstelle `NumberSequence`

## Endliche und unendliche Zahlenfolgen

Objekte der Schnittstelle `NumberSequence` können *endlich* oder *unendlich* sein.

- *Unendliche* Zahlenfolgen werden generiert (z. B. Klasse `FibonacciSequence`).
- *Endliche* Zahlenfolgen können für explizit hinterlegte Daten erzeugt (z. B. Klasse `FiniteFolge`) oder generiert werden.

# Unendliche Zahlenfolge der natürlichen Zahlen

```
public class NaturalNumbers implements NumberSequence {

  private int number;

  public NaturalNumbers() {
    number = 1;
  }

  @Override
  public boolean hasNext() {
    return true;
  }

  @Override
  public int next() {
    number++;
    return number - 1;
  }
}
```

# Endliche Zahlenfolge natürlicher Zahlen

```
public class NaturalNumbers implements NumberSequence {

  private int number;
  private final int limit;

  public NaturalNumbers(int limit) {
    number = 1;
    this.limit = limit;
  }

  @Override
  public boolean hasNext() {
    return number <= limit;
  }

  @Override
  public int next() {
    if (!hasNext()) {
      throw new NoSuchElementException("...");
    }
    number++;
    return number - 1;
  }
}
```

# Was kann man mit Zahlenfolgen machen?

Man kann über sie iterieren, z. B. um das erste Element zu finden, das eine Primzahl ist.

## Beispiel (Anwendung von `NumberSequence`)

```
NumberSequence seq = ...;
...
int number = 0;
boolean primeFound = false;
while (seq.hasNext() && !primeFound) {
  number = seq.next();
  primeFound = isPrime(number);
}
// falls primeFound true ist, enthält number
// die erste Primzahl der Folge
```

# Was ist nun das Besondere an Streams?

## Vereinfacht ausgedrückt

Streams perfektionieren die Idee von `NumberSequence`.

# Was ist nun das Besondere an Streams?

## Vereinfacht ausgedrückt

Streams perfektionieren die Idee von `NumberSequence`.

## ... etwas genauer

- Elemente von Streams können ohne Iteration verarbeitet werden.
- Unmittelbare Unterstützung vieler Operationen auf Streams: Filtern, Transformieren, Aggregieren, ...
- Streams lassen sich parallel verarbeiten.
- Streams erhöhen das Abstraktionsniveau bei der Programmierung (Deklaratives Programmieren). Es steht mehr das *Was* als das *Wie* im Vordergrund.

# Einführendes Beispiel (1)

## Beispiel (Erzeugung von Stream, Operationen auf Stream)

```
Stream.of("08/15", "4711", "501", "S04", "1250", "333", "475")
    .filter(s -> s.matches("[0-9]+"))
    .map(s -> Integer.parseInt(s))
    .filter(n -> n >= 400)
    .sorted()
    .forEach(System.out::println);
```

# Einführendes Beispiel (2)

## Beispiel (Erzeugung von Stream, Operationen auf Stream)

```
IntStream.rangeClosed(111, 999)
    .filter(n -> n % (n / 100 + n / 10 % 10 + n % 10) == 0)
    .forEach(System.out::println);
```

# Externe Iteration

Externe Iteration

- wird von Entwickler/-in gesteuert (programmiert),
- ist *sequentiell*.

## *Externe* Iteration über Menge von Werten

Schleife mit Initialisierung, Bedingung, Aktualisierung; Beispiele:

```
for (int i = 1; i < n; i++) { ... }
for (long n = 1; n < limit; n = 2 * n + 1) { ... }
for (int n = start; n > 0; n = n / 10) { ... }
```

## *Externe* Iteration über Elemente von Feld oder Collection

- Schleife über Indizes oder Iterator
- erweiterte `for`-Schleife

# Interne Iteration

## *Interne* Iteration über Stream

- Es gibt keine (von außen) sichtbare Iteration.
- Der Stream steuert die Iteration.
- Er holt sich die Daten, die er benötigt.
- Interne Iteration kann den Ablauf von sich aus *parallelisieren*.

# Woher kommen die Daten?

## Eine Sequenz, keine Verwaltung der Daten

Ein Stream

- ist keine Datenstruktur, die Daten intern verwaltet.
- greift entweder auf bestehende Daten (eines Felds, einer Collection, eines Eingabestroms, einer Zeichenkette, ...) zu oder generiert sie ad hoc.

# Welche Stream-Klassen gibt es?

## Stream-Klassen

| Klasse | Typ der Elemente |
|---|---|
| `Stream<T>` | `T` |
| `DoubleStream` | `double` |
| `IntStream` | `int` |
| `LongStream` | `long` |

`IntStream` *unterscheidet* sich von `Stream<Integer>`!

# Stream erzeugen

## Stream für explizit vorhandene Daten erzeugen

- in Schnittstelle `Stream<T>` statische Methode

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

- in Schnittstelle `Collection<E>` Instanzmethode

  ```
  Stream<E> stream()
  ```

- in Klasse `Arrays` statische Methode

  ```
  Stream<T> stream(T[] array)
  ```

- in Klasse `BufferedReader` Instanzmethode

  ```
  Stream<String> lines()
  ```

# Stream erzeugen

## Unendlichen Stream generieren

- in Schnittstelle `Stream` statische Methode

  ```
  Stream<T> iterate(T seed, UnaryOperator<T> f)
  ```

  Analog für `IntStream`/`IntUnaryOperator`, `LongStream`/`LongUnaryOperator` ...
- in Schnittstelle `Stream` statische Methode

  ```
  Stream<T> generate(Supplier<T>)
  ```

  Analog für `IntStream`/`IntSupplier`, `LongStream`/`LongSupplier`, ...

# Stream erzeugen

## Endlichen Stream generieren

- in Schnittstelle `IntStream` statische Methode

  ```
  IntStream range(int startInclusive, int endExclusive)
  ```

  Analog für `LongStream`, jedoch *nicht* für `DoubleStream`.
- in Schnittstelle `IntStream` statische Methode

  ```
  IntStream rangeClosed(int startInclusive, int endInclusive)
  ```

  Analog für `LongStream`.

# Operationen auf Streams

Intermediäre Operationen: liefern wieder einen Stream

- (Anzahl-)Begrenzung (`limit`)
- Filterung
- Mapping (Transformation, Abbildung)

Terminale Operationen: liefern keinen Stream, sondern Wert, Objekt, oder haben nur einen Seiteneffekt (z. B. `forEach`)

- Quantoren
- Einzelne Elemente aus Stream
- Stream-Elemente (gruppiert) als Collection
- Minimum, Maximum
- Summe und Durchschnitt (nicht bei `Stream<T>`)

# Operationen auf Streams

Operationen auf Streams können zu einer Pipeline zusammengefügt werden.

- Am Anfang steht Erzeugung von Stream.
- Am Ende steht terminale Operation.
- Dazwischen stehen intermediäre Operationen.

Aber: eine Pipeline muss nicht in *einer* Anweisung stehen.

# Einzelne Elemente aus einem Stream holen

## Erstes Element

- `findFirst()` liefert das *erste* Element eines Streams, oder ein leeres Optional, wenn der Stream leer ist .
- Ergebnistyp ist `Optional<T>` (oder `OptionalInt`, ...). Ein Optional *kapselt* einen Wert bzw. Objekt, oder repräsentiert die *Abwesenheit* eines Werts bzw. Objekts.

## Beliebiges Element

- `findAny()` liefert ein *beliebiges* Element.
- Ergebnistyp ist `Optional<T>` (oder `OptionalInt`, ...).

Wichtig: Trotz des Namens sind die `find`-Methoden *keine* Suchmethoden, die einen Wert anhand *eines Kriteriums suchen*.

# All- und Existenzquantoren

## `...match`-Methoden

- `boolean allMatch(...)` gibt an, ob eine Bedingung auf alle Elemente eines Streams zutrifft.
- `boolean anyMatch(...)` gibt an, ob es ein Element gibt, auf das die Bedingung zutrifft.
- `boolean noneMatch(...)` gibt an, ob eine Bedingung auf keines der Elemente eines Streams zutrifft.

# All- und Existenzquantoren

## `...match`-Methoden

- `boolean allMatch(...)` gibt an, ob eine Bedingung auf alle Elemente eines Streams zutrifft.
- `boolean anyMatch(...)` gibt an, ob es ein Element gibt, auf das die Bedingung zutrifft.
- `boolean noneMatch(...)` gibt an, ob eine Bedingung auf keines der Elemente eines Streams zutrifft.

## Wie gibt man die Bedingung an?

Die Bedingung wird durch ein Objekt der Schnittstelle `Predicate<T>` (oder `IntPredicate`, ...) repräsentiert (Entwurfsmuster Strategie).

# Filter

## Eine Auswahl von Elementen eines Streams

- `filter` liefert einen Stream aller Elemente eines Streams, für die eine Bedingung zutrifft.
- Die Bedingung wird durch den Parameter von `filter` vom Typ `Predicate` angegeben.

| Klasse | Methode |
|---|---|
| `IntStream` | `IntStream filter(IntPredicate predicate)` |
| `IntPredicate` | `boolean test(int value)` |
| `Stream<T>` | `Stream<T> filter(Predicate<? super T> predic` |
| `Predicate<T>` | `boolean test(T t)` |

# Abbildungen

`map` (und andere Methoden, die mit `map` beginnen) wenden auf die Elemente eines Streams eine Abbildung an und liefern die resultierenden Elemente wieder als Stream.

# Abbildungen

## Beispiele für `map`-Funktionen

| Klasse | Methode |
|---|---|
| `Stream<T>` | `<R> Stream<R> map(Function<T,R> mapper)` |
| `IntStream` | `IntStream map(IntUnaryOperator mapper)` |
| `Stream<T>` | `IntStream mapToInt(ToIntFunction<? super T> mapper)` |

## Beispiele für `Functions`

| | |
|---|---|
| `Function<T,R>` | `R apply(T t)` |
| `IntUnaryOperator` | `int applyAsInt(int operand)` |
| `ToIntFunction<T>` | `int applyAsInt(T value)` |

# Elemente eines Streams in Container sammeln

## in Collection sammeln

- Durch die terminale Operation `forEach` können die Elemente eines Streams in einer zuvor erzeugten Collection gesammelt werden.
- Deklarativer geht es mit speziell dafür vorgesehenen Methoden der Stream-Klassen:
  - in `Stream<T>`: `collect` mit Übergabe eines `Collectors`; dadurch können die Stream-Elemente in einer Liste oder Menge gesammelt werden.
  - in `Stream<T>`, `IntStream`, etc.: `toList`; es gibt *kein* `toSet`!

# Elemente aggregieren

## Aggregation

- Aggregation bedeutet, Elemente zu einem Wert zusammenzufassen.
- `min`, `max` und count sind Beispiele für Aggregation.
- Es sind darüber hinaus beliebige Aggregationen möglich. Die Methode dafür ist `reduce` der Klasse `Stream`.

# Lazy Evaluation

- Streams sind *keine* Datenstruktur.
- Die Elemente eines Streams werden erst bei Bedarf erzeugt. Bsp.: durch `filter` wird kein Stream *komplett gefiltert*, sondern ein gefilterter Zugriff auf die Elemente des zugrunde liegende Streams ermöglicht.

# Parallele vs. sequentielle Streams

## Wer entscheidet über Parallelität?

- Viele nicht-terminale Operationen auf Streams sind parallelisierbar, z. B. `filter` und `map`.
- Manche terminale Operationen sind parallisierbar, z. B. `max`.
- Durch Anwendung der Methode `parallelStream` (anstelle von `stream`) wird ein Stream erzeugt, der seine Methoden parallelisiert.
- Durch Anwendung der Methode `parallel` der Schnittstelle `BaseStream` wird zu einem Stream ein paralleler Stream geliefert.
- Parallelisierung kann den Zeitbedarf für Operationen verringern.