13.3 Datenströme (java.util.stream)

- Seit Java 8 gibt es das neue Paket java.util.stream im JDK
  - Das Paket enthält Funktionalität zur Verarbeitung eines Stroms von Werten (Objekten).
  - Damit wird es einfacher Lambdas für die Programmierung zu nutzen.

Operatoren auf Datenströmen

- Unterteilung in drei Klassen
  - Erzeugende Datenstromoperationen (Generatoren)
    - Dadurch können Container-Objekte (und andere), wie z. B. Listen, als Datenstrom zur Verfügung gestellt werden.
  - Transformationsoperationen (Transformatoren)
    - Operationen, die einen Datenstrom in einen anderen überführen.
    - Die Ausführung eines Transformators wird solange nicht begonnen bis dies von einer nachfolgenden Terminaloperation nicht explizit eingefordert wird.
  - Terminaloperatoren
    - Konsumieren des Datenstroms und Erzeugung eines Ergebnisses.

Generatoren

- Collections

```
List<Integer> li = new LinkedList<>();
li.add(1);
li.add(2);
li.add(3);
Stream<Integer> s = li.stream();
```

- Streams

```
Stream<Integer> s = Stream.of(1, 2, 3);
```

Generatoren

- Supplier

```
class NaturalNumbers implements Supplier<Integer> {
    Integer next = 1;
    @Override
    public Integer get() {
        return next++;
    }
}
```

```
Stream<Integer> s = Stream.generate(new NaturalNumbers());
```

Warum hier keine Verwendung eines Lambda-Ausdrucks?

Wir müssen uns die letzte Zahl merken und Lambdas haben keinen Zustand!

Generatoren

- Supplier, Beispiel mit Methodenreferenz

```
Random rand =  new Random(System.currentTimeMillis());
Stream<Integer> s2 = Stream.generate(rand::nextInt);
s2.limit(10).forEach(System.out::println);
```

Transformatoren (1)

- Filter
  - Eliminiere alle Elemente, die das gegebene Prädikat **nicht** erfüllen

```
Stream<Integer> s = Stream.of(1,2,3,4)
    .filter( x -> x%2 == 0 );
```

- Map
  - Überführe Datenstrom von Typ I nach Datenstrom vom Typ O

```
Stream<String> s = Stream.of(3,2,4,1)
        .map( x -> Integer.toString(x) );
```

Zurück zum Motivierenden Beispiel

- Jetzt können wir den Musiker-Filter implementieren
  - Keine Redundanz
  - Keine unnötigen Klassen
  - Filter Separat

```
List<Musician> mlist = new LinkedList<>();
...
mlist.stream().
    filter(m -> m.getName().equals("Wilson")).
    forEach(m -> System.out.println(m));
```

Transformatoren (2)

- FlatMap
  - Bilde jedes Objekt im Eingabestrom auf einen Datenstrom ab. Liefere neuen Datenstrom, der alle Elemente der erzeugten Ströme enthält

```
Stream<List<Integer>> ls = Stream.of(
    Arrays.asList( 1, 3, 5, 7 ),
    Arrays.asList( 2, 4, 6, 7 ) );
Stream<Integer> s = ls.flatMap( l -> l.stream() );
```

- Sorted / Sorted(<Comparator<T>)
  - Liefert alle Elemente des Stroms sortiert nach ihrer natürlichen/der gegebenen Ordnung

```
Stream<Integer> = Stream.of(3,2,4,1)
    .sorted();
```

Terminaloperatoren (1)

- forEach
  - Führe die gegebene Aktion für jedes Element des Streams aus

```
Stream.of(1,2,3,4)
      .forEach(System.out::println);
```

- allMatch
  - Überprüfe ob alle Elemente des Datenstroms das gegebene Prädikat erfüllen

```
boolean all = Stream.of(1,2,3,4)
                    .allMatch( x -> x < 10);
```

Terminaloperatoren (2)

- anyMatch
  - Überprüfe ob mindestens ein Element des Datenstroms das gegebene Prädikat erfüllt.

```
boolean any = Stream.of(1,2,3,4)
    .anyMatch( x -> x % 2 == 0 );
```

- reduce
  - Reduziert die Elemente des Stroms auf genau ein Ausgabeelement.

```
Integer sum = Stream.of(1,2,3,4)
    .reduce( 0,
             (acc, value) -> acc+value
    );
```

Startwert acc

Das Ergebnis wird als acc an den nächsten Aufruf des Lamdas übergeben.

Terminaloperatoren (3)

- collect
  - Reduziert die Elemente des Stroms auf einen veränderbaren (mutable) Akkumulator

```
List<Integer> ls = Stream.of(
            Arrays.asList( 1, 3, 5, 7 ),
            Arrays.asList( 2, 4, 6, 7 ) )
    .flatMap( l -> l.stream() )
    .collect(Collectors.toList());
```

Parametertyp: Collector
Keine funktionale Schnittstelle!

Bedeutung der Collector-Schnittstelle ist kompliziert und führt für diese Vorlesung zu weit.
Über die Methoden der Klasse Collectors lassen sich aber vorgefertigte Collector-Objekte erzeugen.

Beispiel Pipeline

```
Stream.of(5,4,7,2,10,8)
    .filter( x -> x % 2 == 0 )
    .map( x -> x*x )
    .sorted()
    .forEach( System.out::println );

// Ergebnis: ?
```

Beispiel Pipeline

```
Stream.of(5,4,7,2,10,8)
    .filter( x -> x % 2 == 0 )
    .map( x -> x*x )
    .sorted()
    .forEach( System.out::println );

// Ausgabe:
// 4
// 16
// 64
// 100
```

Lazy Evaluation (1)

- Auswertung wird ausschließlich durch Terminaloperatoren angestoßen:

```
Stream.of(5,4,7,2,10,8)
    .filter( x -> {
                    System.out.println("Filter: " + x);
                    return x % 2 == 0;
            });
```

  - Liefert keine Ausgabe, da kein Terminaloperator benutzt wird.

Lazy Evaluation (2)

- Auswertung wird ausschließlich durch Terminaloperatoren angestoßen:

```
Stream.of(5,4,7,2,10,8)
    .filter( x -> { System.out.println("Filter: " + x);
                    return x % 2 == 0; })
    .forEach(
            x -> System.out.println("ForEach: " + x)
    );

// Ausgabe:
// Filter: 5
// Filter: 4
// ForEach: 4
// Filter: 7
// Filter: 2
// ForEach: 2
...
```

Lazy Evaluation (3)

- Operationen werden vertikal ausgeführt
  - Ein Element durchläuft alle Operationen der Pipeline, bevor das nächste ausgewertet wird
  - Es werden nur so viele Elemente betrachtet, wie zur Berechnung des Ergebnisses nötig sind
    - Reduziert ggf. die Anzahl der benötigten Operationen

```
Stream.of(5,4,7,2,10,8)
    .map( x -> { System.out.println("Map: " + x);
                 return x*x; })
    .anyMatch( x -> { System.out.println("AnyMatch: " + x);
                      return x%2 == 0; });
    );
// Ausgabe:
// Map: 5
// AnyMatch: 25
// Map: 4
// AnyMatch: 16 -- DONE
```

Ausnahme: Stateful Transformations

- Manche Transformatoren benötigen einen Status
  - Sortierung ist erst möglich, wenn alle Elemente bekannt sind.

```
Stream.of(5,4,7,2,10,8)
    .sorted( (x,y) -> {
        System.out.println("Sort: " + x + ";" + y);
        return x.compareTo(y);} )
    .filter( x -> {
        System.out.println("Filter: " + x);
        return x % 2 == 0; } )
    .forEach(
        x -> System.out.println("ForEach: " + x)
    );
```

Ausgabe:
Sort: 4;5
Sort: 7;4
Sort: 7;5
Sort: 2;5
Sort: 2;4
Sort: 10;5
Sort: 10;7
Sort: 8;5
Sort: 8;10
Sort: 8;7
Filter: 2
ForEach: 2
Filter: 4
ForEach: 4
Filter: 5
Filter: 7
Filter: 8
ForEach: 8
Filter: 10
ForEach: 10

Order Matters

- Reduziere Anzahl der Operationen durch Umsortieren

```
Stream.of(5,4,7,2,10,8)
      .filter( x -> {
          System.out.println("Filter: " + x);
          return x % 2 == 0; } )
      .sorted( (x,y) -> {
          System.out.println("Sort: " + x + ";" + y);
          return x.compareTo(y);} )
      .forEach(
          x -> System.out.println("ForEach: " + x)
      );
```

Ausgabe:
Filter: 5
Filter: 4
Filter: 7
Filter: 2
Filter: 10
Filter: 8
Sort: 2;4
Sort: 10;2
Sort: 10;4
Sort: 8;4
Sort: 8;10
ForEach: 2
ForEach: 4
ForEach: 8
ForEach: 10

Eigentliche Ausgabe gleich.

Zusammenfassung

- Innere Klassen
  - Insbesondere lokale (anonyme) Klassen haben zur Erleichterung der Programmierung beigetragen.
  - Klassen werden dort definiert, wo sie verwendet werden.
- Durch Lambdas werden in Java funktionale Programmierkonzepte zur Verfügung gestellt.
  - Vereinfachung der Programmierung
    - weniger Code, um das Gleiche ausdrücken.
  - Verbesserung der Codequalität
    - Typisierung wird zur Zeit der Übersetzung sichergestellt.
- Datenströme
  - Operatoren zur Verarbeitung von großen Daten
  - Anwendung im Bereich Data Sciene in Verarbeitungssystemen wie Apache Hadoop und Apache Spark