PM2 Java: Collections → Verwenden von Lambdas und Streams

Quelltext: v7-Collection and Streams

Einstieg für alle Beispiele:
coreservlets.missing.EmployeeMain.java

Fahrplan

- Einführung
- Streams erzeugen / Umwandeln in Array und List
- Wichtige Stream-Methoden
- Lazy-Evaluation
- Stream Operationen mit Ordnung auf Objekten
- Quantifizierende Stream Operationen
- Spezialisierte Ströme für Zahlen
- Streams reduzieren
- Sammeln und Gruppieren
- Transformieren

EINFÜHRUNG

Überblick über Streams

- Streams sind Wrapper um Datenquellen. Sie speichern **keine** Daten. Sie **transportieren** Daten einer Quelle über eine **Verarbeitungs-Pipeline**.

- Alle Operationen
  - haben **Lambda-Ausdrücke** oder **Methodenreferenzen** als Argumente,
  - erlauben keinen indizierten Zugriff, **aber**
  - können in Listen und Arrays umgewandelt werden.

- Streams verfügen über mehr „komfortable“ Methoden als Listen:
  - ***forEach, filter, map, reduce, sort, min, max, distinct, limit***, etc.

- Streams haben Eigenschaften, die Listen fehlen.
  - Lazy-Evaluation
  - parallele Verarbeitung
  - potentiell unendlich

- Mit Java 8 wird das generische Interface ***Stream<T>*** sowie eine Reihe von Spezialisierungen: ***IntStream***, ***DoubleStream*** etc. eingeführt.

KLASSEN DER NACHFOLGENDEN BEISPIELE

Die Klasse *EmployeeUtils*

```
public class EmployeeUtils {

   public static Employee findById(Integer id) {
      return EmployeeSamples.getSampleEmployees().stream()
            .filter(e -> e.getEmployeeId() == id)
            .findFirst().orElse(null);
   }
}
```

Die Klasse *EmployeeSamples*

```
package coreservlets.streams;

import java.util.*;

public class EmployeeSamples {
  private static final List<Employee> SAMPLE_EMPLOYEES =
    Arrays.asList(
      new Employee("Harry", "Hacker", 1, 234567),
      new Employee("Polly", "Programmer", 2, 333333),
      new Employee("Cody", "Coder", 8, 199999),
      new Employee("Devon", "Developer", 11, 175000),
      new Employee("Desiree", "Designer", 14, 212000),
      new Employee("Archie", "Architect", 16, 144444),
      new Employee("Tammy", "Tester", 19, 166777),
      new Employee("Sammy", "Sales", 21, 45000),
      new Employee("Larry", "Lawyer", 22, 33000),
      new Employee("Amy", "Accountant", 25, 85000) );

  public static List<Employee> getSampleEmployees() {
    return(SAMPLE_EMPLOYEES);
  }

  private EmployeeSamples() {} // Uninstantiatable class
}
```

Die Klasse *Person*

```
public class Person {
  protected String firstName, lastName;

  public Person(String firstName,String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }
  public Person() {
    this.firstName = PersonUtils.randomFirstName();
    this.lastName = PersonUtils.randomLastName();
  }
  public String getFirstName() { return (firstName);}
  public void setFirstName(String firstName) { this.firstName = firstName;}
  public String getLastName() {return (lastName);}
  public void setLastName(String lastName) {this.lastName = lastName;}

  public String getFullName() {return(firstName + " " + lastName);}

  public int firstNameComparer(Person other) {
      return(firstName.compareTo(other.getFirstName()));
  }

  @Override
  public String toString() {
    return(String.format("%s (%s)", getFullName(), getClass().getSimpleName()));
  }
}
```

Die Klasse *Employee*

```
public class Employee extends Person {
  private int employeeId, salary;

  public Employee(String firstName, String lastName,int employeeId, int
salary) {
    super(firstName, lastName);
    this.employeeId = employeeId;
    this.salary = salary;
  }
  public int getEmployeeId() {return (employeeId);}

  public void setEmployeeId(int employeeId) {this.employeeId = employeeId;}

  public int getSalary() { return(salary); }

  public void setSalary(int salary) { this.salary = salary;}

  @Override
  public String toString() {
    return(String.format("%s %s [Employee#%s $%,d]",
           firstName, lastName, employeeId, salary));
  }
}
```

Überblick über Streams

- Streams sind Wrapper um Datenquellen wie Listen und Arrays etc.
- Streams haben eine Reihe an mächtigen Funktionen um die Quellen sequentiell oder parallel auszuwerten.

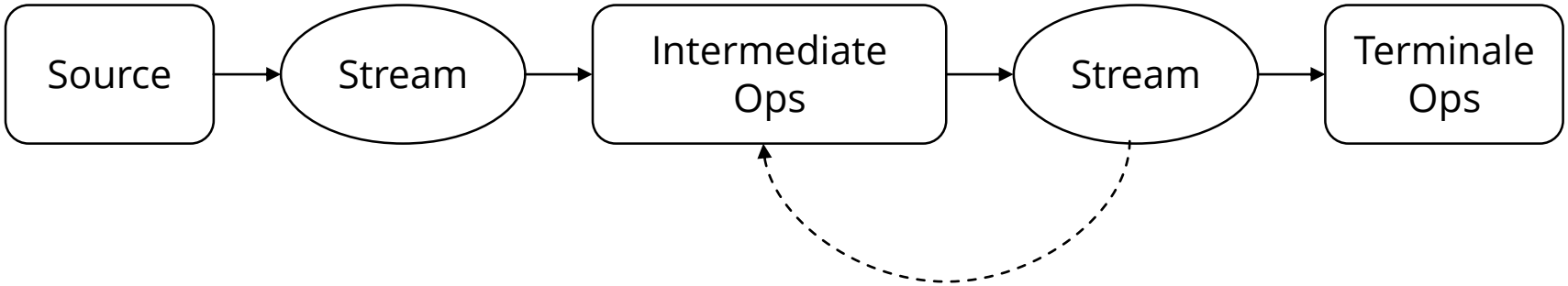
```
public class EmployeeMain {

    public static void main(String[] args) {
        Integer[] idArray = { 1, 2, 10, 13 };
        List<Employee> empls = Stream.of(idArray)
                .map(EmployeeUtils::findById)
                .filter(e -> e != null)
                .filter(e -> e.getSalary() > 200000)
                .collect(Collectors.toList());
        System.out.println(empls);
    }
}
```

Methodenreferenz

Lambda-Ausdruck

```
[Harry Hacker [Employee#1 $234.567], Polly Programmer [Employee#2 $333.333]]
```

Verarbeitungs-Pipeline für Streams

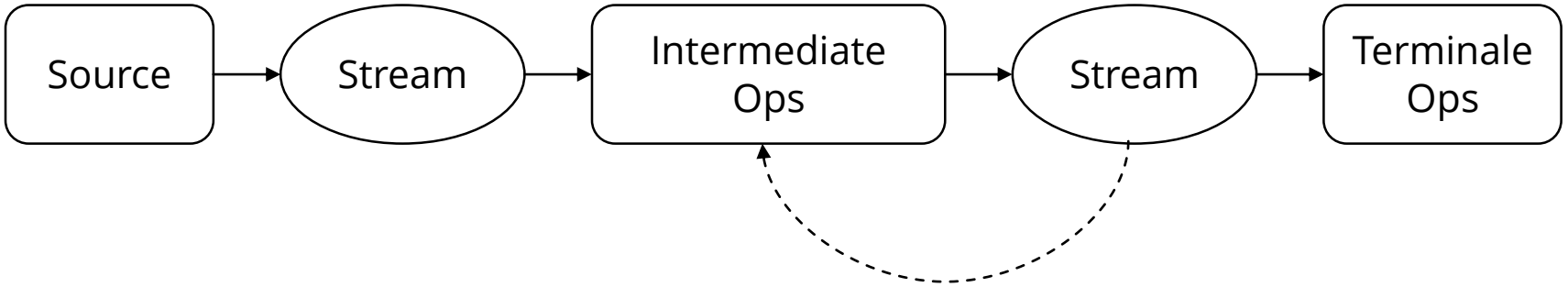


- ***Collection***
- ***Array***
- Generator Funktion
- I/O Channel

- Prädikat
- Funktion

- ***forEach***
- Reduktions-Ops

Verarbeitungs-Pipeline am Beispiel




```
Stream.of(idArray)
                    .map(EmployeeUtils::findById)
                    .filter(e -> e != null)
                    .filter(e -> e.getSalary() > 200000)
                                        .collect(Collectors
                                                .toList());
```

STREAMS ERZEUGEN UMWANDELN IN ARRAY / LIST

Streams erzeugen

Methode ***sampleStreamCreate*** in ***EmployeeMain***

```
// aus Collections
List<String> words = Arrays.asList("one", "two", "three" );
Stream<String> stringStream = words.stream();
Collection<Employee> employees = EmployeeSamples.getSampleEmployees();
Stream<Employee> emplStream = employees.stream();

// aus Arrays
String[] wordAry = new String[] { "one", "two", "three" };
stringStream = Stream.of(wordAry);

// aus Aufzaehlungen einzelner Elemente
stringStream = Stream.of("first", "second", "third");

// aus Strings: chars erzeugt einen IntStream
"one".chars().forEach(e -> p((char) e));
Stream.of("o n e".split(" ")).forEach(e -> p(e));

// aus Verzeichnissen
Files.list(Paths.get(".")).forEach(e -> p(e));

// aus Streams: map, filter, sort, distinct, limit, skip
```

Streams in Collection und Array wandeln

- Methode ***sampleStreamConvert*** in ***EmployeeMain***
- Collections: ***collect(Collectors.toList() / toSet())***
- Arrays: ***toArray(<Type>::new)***

```
Collection<Employee> employees = EmployeeSamples.getSampleEmployees();
List<Employee> empls =
        employees.stream()
        .filter(e -> e.getSalary() > 200000)
        .collect(Collectors.toList());
p(empls);

String[] words = new String[] { "one", "two", "three" };
String[] filtered =
        Stream.of(words)
        .filter(e -> e.startsWith("t"))
        .toArray(String[]::new);
p(Arrays.deepToString(filtered));
```

WICHTIGE STREAM-METHODEN

Die wichtigsten Stream Methoden

- ***forEach(Consumer)***: Lambda-Ausdruck oder Methodenreferenz (***void*** Methode)
- ***map(Function)***: Lambda-Ausdruck, dessen Körper eine Funktion / ein Ausdruck ist. Ergebnistyp ***Stream***.
- ***filter(Predicate)***: Lambda-Ausdruck, dessen Körper ein logischer Ausdruck ist, Ergebnistyp ***Stream***.
- ***findFirst()***: erstes Element des Streams. Ergebnistyp ***Optional***.
- ***toArray(<Type>[]:new)***: Wandelt den Inhalt eines Streams in ein Arrays um.
- ***collect(Collectors.toList()) / collect(Collectors.toSet()) / etc...*** Wandelt den Inhalt eines Streams in eine Collection um.

nur 1 Pipeline pro Stream aber mehrere Streams pro Datenquelle

- Methode ***moreThanOnePipeline*** in ***EmployeeMain***
- mehrere Pipelines für einen Stream sind **nicht** möglich → ***IllegalStateException***

```
List<Employee> employees = EmployeeSamples.getSampleEmployees();
// Richtige Verwendung: 1 Stream pro Pipeline
employees.stream().map(e -> e.getEmployeeId()).filter(id -> id > 2)
         .forEach(System.out::println);
employees.stream().map(e -> e.getSalary()).filter(sal -> sal > 200000)
         .forEach(System.out::println);

Stream<Employee> emplStream = EmployeeSamples.getSampleEmployees().stream();
// Fehlerhafte Verwendung: n Pipelines fuer 1 Stream
emplStream.map(e -> e.getEmployeeId()).filter(id -> id > 2)
         .forEach(System.out::println);
emplStream.map(e -> e.getSalary()).filter(sal -> sal > 200000)
         .forEach(System.out::println);
```

forEach vs. for

- Methode ***forEach*** versus ***for*** in ***EmployeeMain***

```
List<Employee> employees = EmployeeSamples.getSampleEmployees();
// Ausgabe mit for
for (Employee e : employees) {
   System.out.println(e);
}
// Ausgabe mit foreach
employees.stream().forEach(e -> System.out.println(e));
```

- Vorteile:
  - Lambdas ggf. kürzere Schreibweise (+)
  - Parallelisierung möglich (+++)

map – Transformieren der Elemente eines Streams

- Methode ***mapExample*** in ***EmployeeMain***
- z.B. Abbildung von Zahlen auf deren Quadrate
- z.B. Abbildung von id's auf Angestellte

```
Double[] nums = { 1.0, 2.0, 3.0, 4.0, 5.0 };
Double[] squares = Stream.of(nums).map(e -> e * e).toArray(Double[]::new);

Integer[] idArray = { 1, 2, 10, 13 };
List<Employee> empls = Stream.of(idArray).map(EmployeeUtils::findById)
          .collect(Collectors.toList());
```

Die Klasse *StreamUtils*
kommentierte Ausgabe des Inhaltes eines Streams

```
public class StreamUtils {

   public static <T> void printStreamAsList(
            Stream<T> s, String message) {
      System.out.printf("%s: %s.%n"
            ,message
            ,s.collect(Collectors.toList()));
   }
}
```

Wiederholte Anwendung von map

Methode ***mapNumberExample*** in ***EmployeeMain***

```
Double[] nums = { 1.0, 2.0, 3.0, 4.0, 5.0 };
printStreamAsList(Stream.of(nums), "Originale");
printStreamAsList(Stream.of(nums).map(e -> e * e), "Quadratzahlen");
printStreamAsList(Stream.of(nums).map(e -> e * e).map(Math::sqrt),
                                                  "Wurzel der Quadratzahlen");
```

```
Originale: [1.0, 2.0, 3.0, 4.0, 5.0].
Quadratzahlen: [1.0, 4.0, 9.0, 16.0, 25.0].
Wurzel der Quadratzahlen: [1.0, 2.0, 3.0, 4.0, 5.0].
```

map Beispiel mit Employees

Methode ***mapEmployeeExample*** in ***EmployeeMain***

```
Integer[] idArray = { 1, 2, 11, 14 };
printStreamAsList(Stream.of(idArray), "ID's");
printStreamAsList(Stream.of(idArray).map(EmployeeUtils::findById)
                            .map(e -> e.getFullName()), "Namen zu den ID's");
```

```
ID's: [1, 2, 11, 14].
Namen zu den ID's:
     [Harry Hacker, Polly Programmer, Devon Developer, Desiree Designer].
```

filter – Elemente weitergeben, die ein Prädikat erfüllen

Methoden ***firstFilterExample*** und ***employeeFilterExample*** in ***EmployeeMain***

```
Integer[] intAry = { 1, 2, 3, 4, 5, 6, 7, 8 };
Integer[] gerade = Stream.of(intAry)
                    .filter(i -> i % 2 == 0)
                    .toArray(Integer[]::new);
p(Arrays.deepToString(gerade));
Integer[] gerade2 = Stream.of(intAry)
                    .filter(i -> i % 2 == 0)
                    .filter(i -> i > 3)
                    .toArray(Integer[]::new);
p(Arrays.deepToString(gerade2));
```

```
Integer[] ids = { 14, 11, 10, 8, 7, 2, 1 };
printStreamAsList(
        Stream.of(ids)
                .map(EmployeeUtils::findById)
                .filter(e -> e!= null)
                .filter(e -> e.getSalary() > 200000),
                "Salary > 200000");
```

findFirst

- Methode ***findFirstExample*** in ***EmployeeMain***
- liefert das erste Element eines Streams als ***Optional***
- Streams, die das Ergebnis einer Filter-Operation sind, können auch leer sein → dann ist auch das ***Optional*** leer.

```
Integer[] ints = { 14, 11, 10, 8, 7, 2, 1 };
// wenn sicher gestellt ist, dass es ein Element gibt
Integer erstesGerades = Stream.of(ints).filter(i -> i % 2 == 0)
        .findFirst().get();
p(erstesGerades);
// wenn der Ergebnis Stream leer sein kann
List<Employee> empls = EmployeeSamples.getSampleEmployees();
Employee lucky = empls.stream().filter(e -> e.getSalary() > 300000)
        .findFirst().orElse(null);
p(lucky);
Employee rich = empls.stream().filter(e -> e.getSalary() > 400000)
        .findFirst().orElse(null);
p(rich);
```

Umgang mit *Optional*

- Methode ***optionalExample*** in ***EmployeeMain***
- ***Optional<T>*** enthält entweder ein Objekt vom Typ ***T***, oder ist leer.

```
/*  ein Optional erzeugen */
Optional<Integer> oInt = Optional.of(14);
Optional<Employee> oEmpl = Optional.empty();
/*
 *  Operationen fuer Optional
 *  get: liefert den Wert in Optional oder generiert einen
 *       Fehler (NoSuchElementException)
 */
Integer anInt = oInt.get();
Employee anEmpl = oEmpl.get(); // Fehler
/*
 * orElse: liefert den Wert in Optional
 *         Wenn Optional leer ist, dann das Argument von orElse
 */
anEmpl = oEmpl.orElse(new Employee("", "", 0, 0));
/*
 *  Prueft, ob ein Wert enthalten ist
 */
boolean hasInt = oInt.isPresent();
boolean hasEmpl = oEmpl.isPresent();
```

LAZY EVALUATION

Beispiel Setup: Die Klasse *EmployeeSamples*

```
package coreservlets.streams;

import java.util.*;

public class EmployeeSamples {
  private static final List<Employee> SAMPLE_EMPLOYEES =
    Arrays.asList(
      new Employee("Harry", "Hacker", 1, 234567),
      new Employee("Polly", "Programmer", 2, 333333),
      new Employee("Cody", "Coder", 8, 199999),
      new Employee("Devon", "Developer", 11, 175000),
      new Employee("Desiree", "Designer", 14, 212000),
      new Employee("Archie", "Architect", 16, 144444),
      new Employee("Tammy", "Tester", 19, 166777),
      new Employee("Sammy", "Sales", 21, 45000),
      new Employee("Larry", "Lawyer", 22, 33000),
      new Employee("Amy", "Accountant", 25, 85000) );

  public static List<Employee> getSampleEmployees() {
    return(SAMPLE_EMPLOYEES);
  }

  private EmployeeSamples() {} // Uninstantiatable class
}
```

Lazy Evaluation

Methode ***lazyEvaluationExample*** in ***EmployeeMain***

```
System.out.println("lazyEvaluationExample");
Integer[] ids = { 3, 4, 8, 15, 21, 14, 1, 22, 25 };
Stream.of(ids).map(id -> {
    p("Suche nach Angestellter mit id " + id + ".");
    return EmployeeUtils.findById(id);
}).filter(e -> {
    p("Pruefe auf null ");
    return e != null;
}).filter(e -> {
    p("Pruefe Gehalt von " + e);
    return e.getSalary() > 200000;
}).findFirst().orElse(null);
```

Frage: Was gibt das Programm aus?
Info: zu ids 3,4,15 gibt es keinen Employee

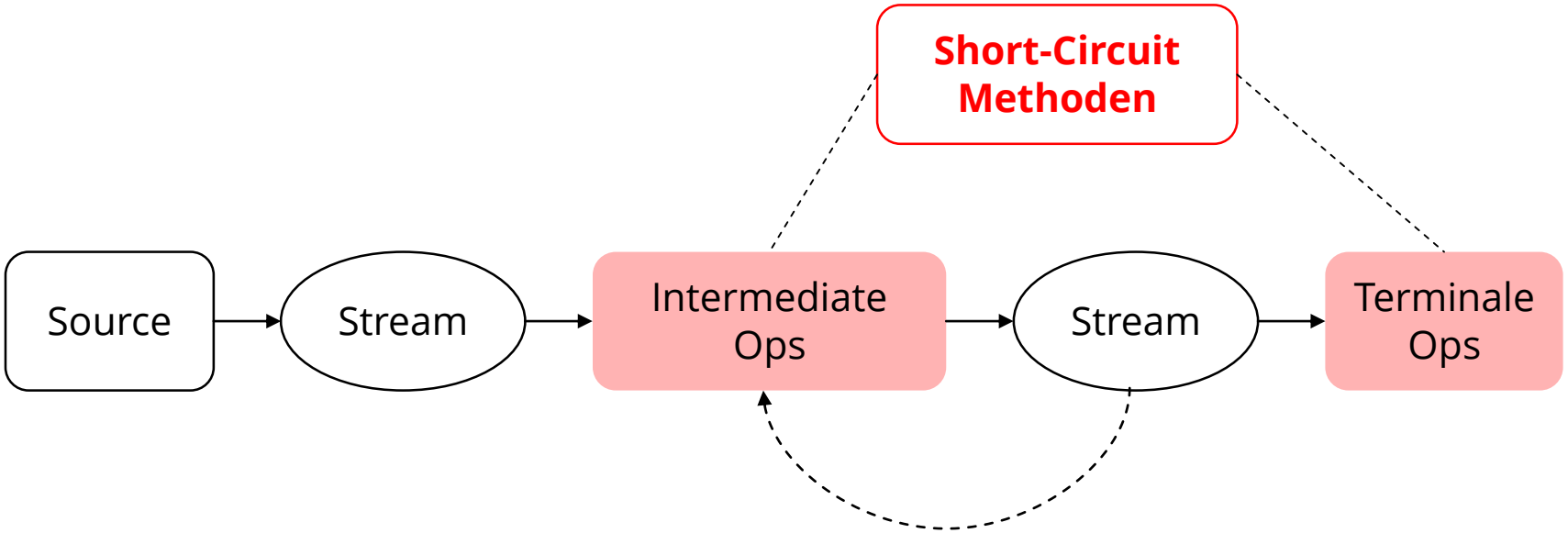
Lazy Evaluation

```
lazyEvaluationExample
Suche nach Angestellter mit id 3.
Pruefe auf null
Suche nach Angestellter mit id 4.
Pruefe auf null
Suche nach Angestellter mit id 8.
Pruefe auf null
Pruefe Gehalt von Cody Coder [Employee#8 $199.999]
Suche nach Angestellter mit id 15.
Pruefe auf null
Suche nach Angestellter mit id 21.
Pruefe auf null
Pruefe Gehalt von Sammy Sales [Employee#21 $45.000]
Suche nach Angestellter mit id 14.
Pruefe auf null
Pruefe Gehalt von Desiree Designer [Employee#14 $212.000]
```

Vielleicht nicht das was Sie intuitiv denken?!

Ursache: Lazy Evaluation von Streams

Lazy Evaluation – Methodenkategorien bei der Verarbeitung von Streams



- ***Collection***
- ***Array***
- Generator Function
- I/O Channel

- Prädikat
- Funktion

- ***forEach***
- Reduktions-Ops

Lazy Evaluation – Methodenkategorien bei der Verarbeitung von Streams

- **Intermediate Methoden**:
  - werden nur verarbeitet, wenn eine terminale Methode aufgerufen wird.
  - ein Element durchläuft immer die gesamte Pipeline.

- **Terminale Methoden**:
  - Nachdem die Methode die Elemente des Streams abgearbeitet hat, ist der Stream geschlossen.
  - Es sind keine weiteren Methoden mehr möglich.

- **Short-Circuit Methoden**:
  - Bei der Bearbeitung eines Stream werden nicht alle Elemente verarbeitet.
  - Können sowohl intermediate als auch terminale Methoden sein.
  - Intermediate: z.B. ***limit, skip***
  - Terminale: z.B. ***findAny, anyMatch***

Lazy Evaluation – Methoden nach Kategorie

```
map, filter, distinct,
sorted, peek, limit,
skip, parallel,
sequential, unordered
```

Intermediate Methoden

```
forEach, forEachOrdered,
toArray, reduce, collect, min,
max, count, anyMatch,
allMatch, noneMatch,
findFirst, findAny, iterator
```

Terminale Methoden

Short-Circuit Methoden

```
anyMatch, allMatch,
nonMatch, findFirst,
findAny, limit, skip
```

Operationen, die die Stream-Größe beschränken

- Methode ***limitSkipExample*** in ***EmployeeMain***
- ***limit(n)*** liefert einen Stream mit den ersten n Elementen
- ***skip(n)*** liefert einen Stream, in der die ersten n Elemente fehlen

```
System.out.println("limitSkipExample");
List<Employee> empls = EmployeeSamples.getSampleEmployees();
List<String> emplNames = empls
        .stream()
        .map(e -> e.getFirstName())
        .limit(8)
        .skip(3)
        .collect(Collectors.toList());
System.out.printf("Names of %d employees skipping first 3: %n%s%n",
        emplNames.size(), emplNames);
```

limit und skip sind short-circuit Methoden! Frage: Wie oft wird dann map ausgewertet?

```
limitSkipExample
Names of 5 employees skipping first 3:
[Devon, Desiree, Archie, Tammy, Sammy]
```

STREAM-OPS UND ORDNUNG VON OBJEKTEN

Operationen, die Ordnungen von Objekten nutzen

- ***sorted***:
  - sortiert die Elemente eines Streams nach der von der Klasse definierten Ordnungsrelation (Klasse implementiert ***Comparable***)
  - sortiert die Elemente eines Streams auf Basis eines Lambda-Ausdrucks
  - in Kombination mit ***limit*** werden immer alle Elemente sortiert und erst dann wird das Limit angewendet.

- ***min / max***:
  - bestimmt minimales / maximales Element einer Streams
  - muss immer mit einem Lambda-Ausdruck verwendet werden
  - liefert ein ***Optional*** als Ergebnis

- ***distinct***
  - entfernt Dubletten

Operationen, die Ordnungen von Objekten nutzen

- Methode ***firstComparisonExamples*** in ***EmployeeMain***
- ***sorted***:
  - ohne Lambda Ausdruck: Elemente des Streams müssen das Interface ***Comparable*** implementieren
  - mit Lambda: 2 Variablen für den Vergleich, der Vergleichsausdruck muss die Semantik von ***compareTo*** abbilden.
- ***min*** / ***max***: siehe ***sorted***.
- ***distinct***: ohne Argumente

Methode *firstComparisonExamples* in *EmployeeMain*

sorted mit lambda-Ausdruck und zwei Blockvariablen

```
private static void firstComparisonExamples() {

    List<Employee> empls = EmployeeSamples.getSampleEmployees();

    List<Employee> emplsSortedBySal = empls.stream().sorted((e1, e2) ->
                    e1.getSalary() - e2.getSalary())
          .collect(Collectors.toList());

    System.out.println(emplsSortedBySal);

    Employee richest = empls.stream().max((e1, e2) ->
                            e1.getSalary() - e2.getSalary()).get();

    Integer[] noDups = Stream.of(
          new Integer[] { 1, 2, 3, 4, 1, 5, 6, 3, 4, 1, 2 }).distinct()
          .toArray(Integer[]::new);
    System.out.println(Arrays.deepToString(noDups));

}
```

max mit lambda-Ausdruck und zwei Blockvariablen

entfernen von Dubletten mit distinct

Methode *sortedExamples* in *EmployeeMain*

```
private static void sortedExamples() {
   List<Employee> empls = EmployeeSamples.getSampleEmployees();
   List<Employee> emplsSorted1 = empls
         .stream()
         .sorted((e1, e2) ->
            e1.getLastName().compareTo(e2.getLastName()))
         .collect(Collectors.toList());
   System.out.printf("Angestellte nach Nachname sortiert %s.%n", emplsSorted1);

   /*
    * limit mit sort verhält sich nicht als short-circuit Operator
    */
   List<Employee> emps3 = empls
         .stream()
         .sorted((e1, e2) -> {p("Vergleiche Vorname");
               return e1.getFirstName().compareTo(e2.getFirstName());
   }).limit(2).collect(Collectors.toList());
   System.out.printf("Angestellte nach Vorname sortiert %s.%n", emps3);
}
```

Methode *minMaxExamples* in *EmployeeMain*

```
private static void minMaxExamples() {
   List<Employee> empls = EmployeeSamples.getSampleEmployees();

   Employee alphabetisch = empls
         .stream()
         .min((e1, e2) ->
            e1.getFullName().compareTo(e2.getFullName()))
         .orElse(null);
   System.out.printf("Erster Angestellte im Alpahbet:  %s.%n", alphabetisch);


   Employee kroesus = empls
         .stream()
         .max((e1, e2) ->
            e1.getSalary() - e2.getSalary()).orElse(null);
   System.out.printf("Krösus %s.%n", kroesus);

}
```

QUANTIFIZIERENDE STREAM-OPS

Quantifizierende Stream-Ops

- ***allMatch, anyMatch, noneMatch***
  - nehmen einen Lambda-Ausdruck, der ein Prädikat enthält,
  - werten das Prädikat aus und
  - beenden die Verarbeitung, sobald der Wahrheitswert der gesamten Verarbeitung bestimmt werden kann
  - siehe auch Ruby ***any?*** und ***all?***
  - ***anyMatch*** endet mit ***true***, wenn ein Element gefunden wurde, das das Prädikat erfüllt
  - ***allMatch*** endet ***false***, wenn ein Element gefunden wurde, das das nicht Prädikat erfüllt
  - ***noneMatch*** endet mit ***false***, wenn ein Element gefunden wurde, das das Prädikat erfüllt

- ***count***: zählt die Anzahl der Elemente in einem Stream

Quantifizierende Stream-Ops

Methode ***anyAllNoneCountExamples*** in ***EmployeeMain***

```
List<Employee> empls = EmployeeSamples.getSampleEmployees();
boolean istNiemandArm = empls.stream().noneMatch(e-> e.getSalary() < 100000);
boolean jemandReich = empls.stream().anyMatch(e -> e.getSalary() > 300000);
boolean alleReich = empls.stream().allMatch(e -> e.getSalary() > 300000);
long anzahlReiche = empls.stream().filter(e -> e.getSalary()> 300000).count();
System.out.printf("Ist niemand arm? -> %s.%n", istNiemandArm);
System.out.printf("Ist irgend jemand reich? -> %s.%n", jemandReich);
System.out.printf("Sind alle reich? -> %s.%n", alleReich);
System.out.printf("Anzahl der Reichen -> %d.%n",anzahlReiche);
System.out.println("Das Leben ist ungerecht!?");
```

```
Ist niemand arm? -> false.
Ist irgend jemand reich? -> true.
Sind alle reich? -> false.
Anzahl der Reichen -> 1.
Das Leben ist ungerecht!?
```

SPEZIALISIERTE STRÖME FÜR ZAHLEN

Streams für Zahlen

- Klassen ***IntStream***, ***LongStream***, ***DoubleStream***
- Gleiche Methoden wie ***Stream***, aber keine Subklassen von ***Stream***
- Spezialisierungen von Streams für den komfortablen Umgang mit Streams von Zahlen (***int, long, double***):
  - ***min, max, sum, average***, ohne Argument
  - Umwandlung: ***IntStream.toArray() -> int[]*** (analog für ***LongStream***, ***DoubleStream***)
- Erzeugen:
  - ***Stream.mapToInt***
  - ***IntStream.of***
  - ***IntStream.range, IntStream.rangeClosed***
  - ***new Random.ints()*** : erzeugt einen ***IntStream*** mit „unendlich vielen Elementen“

IntStream erzeugen

Methode ***intStreamCreateExample*** in ***EmployeeMain***

```
/*
 * Stream.mapToInt: Lambda Ausdruck muss auf int Werte abbilden
 */
List<Employee> empls = EmployeeSamples.getSampleEmployees();
IntStream salary = empls.stream().mapToInt(e -> e.getSalary());
System.out.printf("Salaries %s.%n", Arrays.toString(salary.toArray()));
/*
 *  IntStream.of
 */
IntStream numbers = IntStream.of(1,2,5,7);
IntStream numbers2 = IntStream.of(new int[]{798,0,3,4,5});
```

Salaries [234567, 333333, 199999, 175000, 212000, 144444, 166777, 45000, 33000, 85000].

IntStream erzeugen

Methode ***intStreamCreateExample*** in ***EmployeeMain***

```
/*
 * IntStream.range, rangeClosed
 */
IntStream range = IntStream.range(12, 15);
System.out.printf("IntStream range -> %s.%n", Arrays.toString(range.toArray()));
IntStream rangeClosed = IntStream.rangeClosed(12, 15);
System.out.printf("IntStream rangeClosed -> %s.%n",
                                        Arrays.toString(rangeClosed.toArray()));
/*
 * Infinite IntStream
 */
IntStream infinite = new Random().ints();
infinite.limit(100).forEach(e -> System.out.print(e + " "));
```

```
IntStream range -> [12, 13, 14].
IntStream rangeClosed -> [12, 13, 14, 15].
-1616936592 1386327135 -540102373 -1416907927 1207862820 -1686226365 1644629069 -
762976723 -462333936 619701438 -137248208 -1462352637 -609618158 ...
```

REDUKTION VON STRÖMEN

Reduktion von Streams

- **Idee**: Reduzieren von n Elementen eines Streams auf 1 Element

- Triviale Beispiele:
  - ***findFirst().orElse(other)***
  - ***findAny().orElse(other)***

- für Streams:
  - ***min(comparator),max(comparator),***
  - ***reduce(akku,binaererOP)*** (Semantik wie ***inject*** in Ruby mit explizitem Startwert für ***akku***)
  - ***reduce(binaererOp)*** (Semantik wie ***inject*** in Ruby ohne explizitem Startwert für ***akku***)

- für IntStreams:
  - ***min(), max(), sum(),average()***

reduce in Java ist das inject in Ruby

- ***reduce(akku,binaererOP)***:
  - initialisiere ***akku*** mit einem Wert
  - wende den binären Operator des Lambda-Ausdrucks auf Akku und das erste Element des Streams an
  - weise das Ergebnis ***akku*** zu
  - wende den binären Operator des Lambda-Ausdrucks auf Akku und das nächste Element des Streams an
  - weise das Ergebnis ***akku*** zu
  - usw. bis alle Elemente des Streams abgearbeitet sind

- ***reduce(binaererOp)***: ohne expliziten ***akku***
  - ***akku*** erhält im ersten Durchlauf das erste Element des Streams

Einfache Beispiele für reduce

Methode ***firstReduceExamples*** in ***EmployeeMain***

```
private static void firstReduceExamples() {

    Double[] nums = { 1.5, 788.4, 0.45, 13.0, -0.089, -0.1
};
    Double maxWert1 = Stream.of(nums).
           reduce(Double.MIN_VALUE, (e1, e2) ->
                                 Double.max(e1, e2));
    Double maxWert2 = Stream.of(nums).
           reduce(Double.MIN_VALUE, Double::max);
    System.out.println(maxWert1);
    System.out.println(maxWert2);
    Double produkt = Stream.of(nums).
                     reduce(1.0, (d1, d2) -> d1 * d2);
    System.out.println(produkt);

    String[] woerter = { "im", "Jahr", "der", "Schlange" };
    String konkat = Stream.of(woerter).
           reduce((s1, s2) -> s1 + " " + s2).orElse("");
    String konkat2 = Stream.of(woerter).
           reduce(String::concat).orElse("");
    System.out.println(konkat);
    System.out.println(konkat2);
}
```

Lambda mit binärem Operator ODER Methodenreferenz

reduce ohne akku hat für den leeren Stream kein Ergebnis daher hier ein Optionale

Wer hat gut verhandelt?

Methode ***reduceToMaxSalaryExample*** in ***EmployeeMain***

```
List<Employee> empls = EmployeeSamples.getSampleEmployees();
Employee dieGlueckliche =
        empls.stream()
             .reduce((e1, e2) ->
                      e1.getSalary() >= e2.getSalary() ? e1 : e2)
             .orElse(null);
System.out.printf("Das Glueckskind ist: %s.%n",dieGlueckliche);
```

binärer Operator

Das Glueckskind ist: Polly Programmer [Employee#2 $333.333].

SAMMELN UND GRUPPIEREN

Sammeln und Gruppieren von Stream-Elementen

- Mit den statischen Methoden der ***Collectors*** Klasse lassen sich Stream-Inhalte in andere Datentypen konvertieren.

- Methoden: **List**: ***anyStream.collect(toList())***

dargestellte Verwendung durch statischen Import der Methoden von Collectors

  - **String**: ***stringStream.collect(joining(delim)).toString()***
  - **Set**: ***anyStream.collect(toSet())***
  - andere **Collection** Klasse: ***anyStream.collect(CollectionType::new)***
  - **Map**: ***anyStream.collect(partitionBy(...)), anyStream.collect(groupBy(...))***

Sammeln von Zeichenketten mit Trennzeichen

Methode ***collectStringsWithDelims*** in ***EmployeeMain***

```
List<Employee> empls = EmployeeSamples.getSampleEmployees();
String nachNamen =
          empls.stream()
              .map(Employee::getLastName)
              .collect(joining(", "))
              .toString();
System.out.printf("Nachnamen %s.%n", nachNamen);
```

Nachnamen Hacker, Programmer, Coder, Developer, Designer, Architect, Tester, Sales, Lawyer, Accountant.

TRANSFORMIEREN

Stream-Inhalte in Maps transformieren (1)

Methode ***streamToMapExamples*** in ***EmployeeMain***

```
Map<Boolean, List<Employee>> tabelleDerReichen = EmployeeSamples
        .getSampleEmployees()
        .stream()
        .collect(partitioningBy(e -> e.getSalary() > 90000));
System.out.printf("Angestellte mit Gehalt ueber 90000: %s.%n",
        tabelleDerReichen.get(true));
System.out.printf("Angestellte mit Gehalt unter 90000: %s.%n",
        tabelleDerReichen.get(false));
```

2 Partitionen true und false durch Auswerten eines Prädikats

Angestellte mit Gehalt ueber 90000: [Harry Hacker [Employee#1 $234.567], Polly Programmer [Employee#2 $333.333], Cody Coder [Employee#8 $199.999], Devon Developer [Employee#11 $175.000], Desiree Designer [Employee#14 $212.000], Archie Architect [Employee#16 $144.444], Tammy Tester [Employee#19 $166.777]].
Angestellte mit Gehalt unter 90000: [Sammy Sales [Employee#21 $45.000], Larry Lawyer [Employee#22 $33.000], Amy Accountant [Employee#25 $85.000]].

Stream-Inhalte in Maps transformieren (2)

Methode ***streamToMapExamples*** in ***EmployeeMain***

```
Map<String, List<Emp>> abteilungen =
        EmpSamples.getSampleEmps()
                  .stream()
                  .collect(groupingBy(Emp::getOffice));
System.out.printf("Emps in Mountain View: %s.%n",
                  abteilungen.get("Mountain View"));
System.out.printf("Emps in NY: %s.%n",
                  abteilungen.get("New York"));
System.out.printf("Emps in Zurich: %s.%n",
                  abteilungen.get("Zurich"));
```

n Gruppen gegeben durch die Abteilungsnamen (Schlüssel der Map) Alle Objekte mit gleicher Abteilung stehen in der selben Liste (Wert der Map)

```
Emps in Mountain View: [Larry Page [Mountain View], Sergey Brin [Mountain
View]].
Emps in NY: [Lindsay Hall [New York], Hesky Fisher [New York]].
Emps in Zurich: [Reto Strobl [Zurich], Fork Guy [Zurich]].
```

Zusammenfassung

- Streams sind **Wrapper** um Datenquellen.
- Streams transportieren Daten einer Quelle über eine **Verarbeitungs-Pipeline**.
- Ein Element der Datenquelle durchläuft immer die gesamte Pipeline, bevor das nächste Element verarbeitet wird.
- Argumente der Stream Methoden sind Lambda-Ausdrücke oder Methodenreferenzen.
- **Intermediate** Methoden verarbeiten Elemente und geben diese in einen nachfolgenden Stream weiter. Intermediate Methoden beginnen erst zu arbeiten, wenn ein Element durch eine terminale Methode angefordert wird.
- **Terminale** Methoden konsumieren die Elemente eines Streams. Danach sind keine weiteren Stream-Methoden mehr möglich.
- **Lazy Evaluation**: Short-Circuit Methoden begrenzen die Elemente, die von vorhergehenden intermediate Methoden bearbeitet werden. Vorhergehende intermediate Methoden arbeiten solange, bis / nachdem die short-circuit Methode ausgewertet werden kann.

Quellenangaben

- **Verwendete Vorlagen**: Ausgelassen wurden alle Beispiele und Folien, die ein tieferes Verständnis von Lambda Ausdrücken in Java 8 voraussetzen.
  - Marty Hall, Streams in Java 8: Part 1, http://www.java-programming.info/tutorial/pdf/java/18-Java-8-Streams-Part-1.pdf
  - Marty Hall, Streams in Java 8: Part 2, http://www.java-programming.info/tutorial/pdf/java/18-Java-8-Streams-Part-2.pdf
  - Marty Hall, Streams in Java 8: Part 3, http://www.java-programming.info/tutorial/pdf/java/18-Java-8-Streams-Part-3.pdf , hier nur die Kapitel über das Sammeln und Gruppieren von Streams.

- Quelle für den im Projekt ***v7-Collection and Streams*** verwendeten Quelltext: http://www.java-programming.info/tutorial/java-code/streams-1.zip. Bereinigt um alle Beispiele, die nicht auf den vorhergehenden Folien erklärt werden.

Referenzen

- Oracle, The Java ™ Tutorials, Trail: Collections, Aggregate Operations, https://docs.oracle.com/javase/tutorial/collections/streams/

- Oracle, Java Language and Virtual Machine Specifications, https://docs.oracle.com/javase/specs/