# Lektion 24

Methodenreferenzen

Reduction

Collectors toMap, groupingBy

flatMap

# Funktionale Programmierung II

Methodenreferenzen

Reduction

Processing Order (Streaming Pipeline, Lazyness, terminal und intermediate operations)

Immutability und Parallelisierung

Exception Testing mit assertThrows (JUnit)

Checked Exception Handling

# Methodenreferenzen

Wenn wir die Darstellung noch ein wenig kürzen wollen, können wir überall wo ein Functional Interface erwartet wird, anstelle eines Lambda-Ausdrucks eine **Methodenreferenz** nutzen.

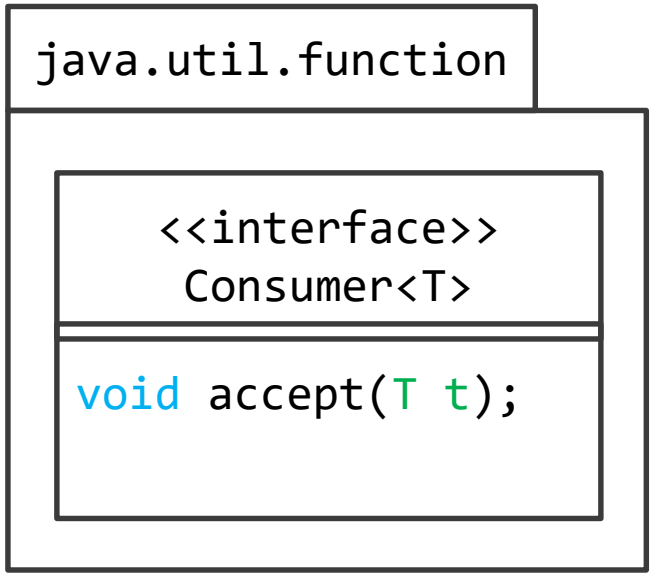
Eine Methodenreferenz wird durch den :: Operator dargestellt, bspw.:

```
System.out::println
```

Hier wird die Methode println von dem Objekt *out* der Klasse *System* referenziert.

```java
List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");

bBrothers.stream()
    .map(name -> name.toUpperCase())
    .forEach(System.out::println);
```

Der Compiler überprüft, ob Parameterliste und Rückgabetyp der referenzierten Methode zu dem Functional Interface passen.




```java
void println(boolean)
void println(char)
void println(int)
void println(long)
void println(float)
void println(double)
void println(char[])
void println(String)
void println(Object)
```

```java
List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");

bBrothers.stream()
    .map(name -> name.toUpperCase())
    .forEach(System.out::println);
```

Beim Ausführen der forEach-Methode, wird jedes Element des Streams an die Methode println übergeben.

Es gibt verschiedene Arten von Methodenreferenzen:

Static method references
Bound method references
Unbound method references
Class Constructor references
Array Constructor references

*(siehe auch:* Joshua Bloch: Effective Java, 3rd Edition, S.198, Addison-Wesley Professional, 2017)

Es gibt verschiedene Arten von Methodenreferenzen:

Static method references
Bound method references
Unbound method references
Class Constructor references
Array Constructor references

```
List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");

bBrothers.stream()
    .map(name -> name.toUpperCase())
    .forEach(System.out::println);
```

```
bBrothers.stream()
    .map(name -> name.toUpperCase())
    .forEach(name -> System.out.println(name));
```

Die Referenz ist an eine **Methode** (println) eines konkreten Objekts (out) **gebunden**. Der String wird (von forEach) als Argument an die println-Methode übergeben.

Es gibt verschiedene Arten von Methodenreferenzen:

Static method references
Bound method references
Unbound method references
Class Constructor references
Array Constructor references

```
List<String> lines = Files.readAllLines(new File("numbers.txt").toPath());
lines.stream()
    .map(Integer::valueOf)
    .forEach(i -> System.out.println(i));
```

```
lines.stream()
    .map(i -> Integer.valueOf(i))
    .forEach(i -> System.out.println(i));
```

Die Referenz ist an eine **statische Methode** gebunden. Wie bei bound method references, wird (von map) das Objekt (der String) als Argument an die statische Methode (valueOf von Integer) weitergereicht.

# Es gibt verschiedene Arten von Methodenreferenzen:

Static method references
Bound method references
Unbound method references
Class Constructor references
Array Constructor references

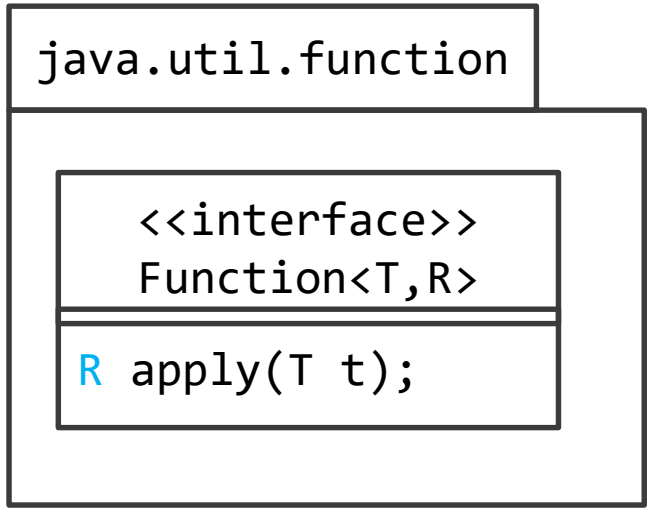
```
List<String> bBrothers
  = Arrays.asList("Burt", "Bronski", "Peter");

bBrothers.stream()
    .map(String::toUpperCase)
    .forEach(System.out::println);
```

```
bBrothers.stream()
    .map(name -> name.toUpperCase())
    .forEach(name -> System.out.println(name));
```

Die Referenz ist nicht an ein Objekt gebunden. Der Aufruf erfolgt auf dem (von map) übergebenen Objekt.

Es gibt verschiedene Arten von Methodenreferenzen:

Static method references
Bound method references
Unbound method references
Class Constructor references
Array Constructor references




```
String toUpperCase()
```

Der Compiler überprüft, ob
der Rückgabetyp zum FunctionalInterface passt.

Es gibt verschiedene Arten von Methodenreferenzen:

Static method references
Bound method references
Unbound method references
Class Constructor references
Array Constructor references

*(siehe auch:* Joshua Bloch: Effective Java, 3rd Edition, S.198, Addison-Wesley Professional, 2017)

Constructor References dienen der Erstellung von Objekten.

# In unserem collect-Beispiel…

```
List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");

List<String> list = bBrothers.parallelStream()
  .filter(name -> name.startsWith("B"))
  .collect(() -> new ArrayList<>(), (l, name) -> l.add(name), (l1, l2) -> l1.addAll(l2));
System.out.println(list);
```

# …können wir eine Class Constructor reference einsetzen:

```
List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");

List<String> collect = bBrothers.parallelStream()
  .filter(name -> name.startsWith("B"))
  .collect(ArrayList::new, (l, name) -> l.add(name), (l1, l2) -> l1.addAll(l2));
System.out.println(collect);
```

## Im übrigen können wir die anderen Lambda-Ausdrücke auch noch durch Methodenreferenzen ersetzen:

```
List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");

List<String> collect = bBrothers.parallelStream()
  .filter(name -> name.startsWith("B"))
  .collect(ArrayList::new, (l, name) -> l.add(name), (l1, l2) -> l1.addAll(l2));
System.out.println(collect);
```

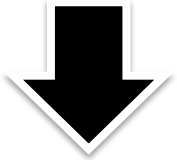
## Den Lambda-Ausdruck für den Accumulator können wir durch eine Unbound Method Reference ersetzen:

```
List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");

List<String> collect = bBrothers.parallelStream()
  .filter(name -> name.startsWith("B"))
  .collect(ArrayList::new, ArrayList::add, (l1, l2) -> l1.addAll(l2));
System.out.println(collect);
```

Der Aufruf von add erfolgt auf der von collect übergebenen ArrayList. collect übergibt weiterhin das Element, das der Liste hinzugefügt werden soll.

# Der Lambda-Ausdruck für den Combiner kann ebenfalls durch eine Unbound Method Reference ersetzt werden:

```
List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");

List<String> collect = bBrothers.parallelStream()
  .filter(name -> name.startsWith("B"))
  .collect(ArrayList::new, ArrayList::add, (l1, l2) -> l1.addAll(l2));
System.out.println(collect);
```



```
List<String> bBrothers = Arrays.asList("Burt", "Bronski", "Peter");

List<String> collect = bBrothers.parallelStream()
  .filter(name -> name.startsWith("B"))
  .collect(ArrayList::new, ArrayList::add, ArrayList::addAll);
System.out.println(collect);
```

Der Aufruf von addAll erfolgt auf der von collect übergebenen ArrayList. Collect übergibt eine weitere ArrayList als Argument für addAll.

Es gibt verschiedene Arten von Methodenreferenzen:

Static method references
Bound method references
Unbound method references
Class Constructor references
Array Constructor references

*(siehe auch:* Joshua Bloch: Effective Java, 3rd Edition, S.198, Addison-Wesley Professional, 2017)

Array Constructor References funktionieren weitest gehend analog zu den Class Constructor References.

# Eindimensionale Arrays mit Lambda

```
Function<Integer, String[]> createStringArray = size -> new String[size];
String[] s = createStringArray.apply(5);
Arrays.stream(s).forEach(System.out::println);
```

```
null
null
null
null
null
```

# Eindimensionale Arrays mit Methodenreferenz

```
Function<Integer, String[]> createStringArray = String[]::new;
String[] s = createStringArray.apply(5);
```

# Zweidimensionale Arrays mit Lambda; zweite Dimension offen

```
Function<Integer, String[][]> create2DStringArray = size -> new String[size][];
String[][] s = create2DStringArray.apply(3);
```

# Zweidimensionale Arrays mit Methodenreferenz; zweite Dimension offen

```
Function<Integer, String[][]> create2DStringArray = String[][]::new;
String[][] s = create2DStringArray.apply(3);
```

# Zweidimensionale Arrays mit Lambda; zweite Dimension spezifiziert

```
BiFunction<Integer, Integer, String[][]> create2DStringArray
  = (rows, cols) -> new String[rows][cols];
String[][] s = create2DStringArray.apply(2,3);
```

# Zweidimensionale Arrays mit Methodenreferenz; zweite Dimension spzifiziert

# Reduction

Reduction allgemein

Map to Int-, Long-, und DoubleStream

Spezialisierte Reduce-Methoden

Nach dem Rausschmiss ist ein neuer B-Bruder der Organisation beigetreten.

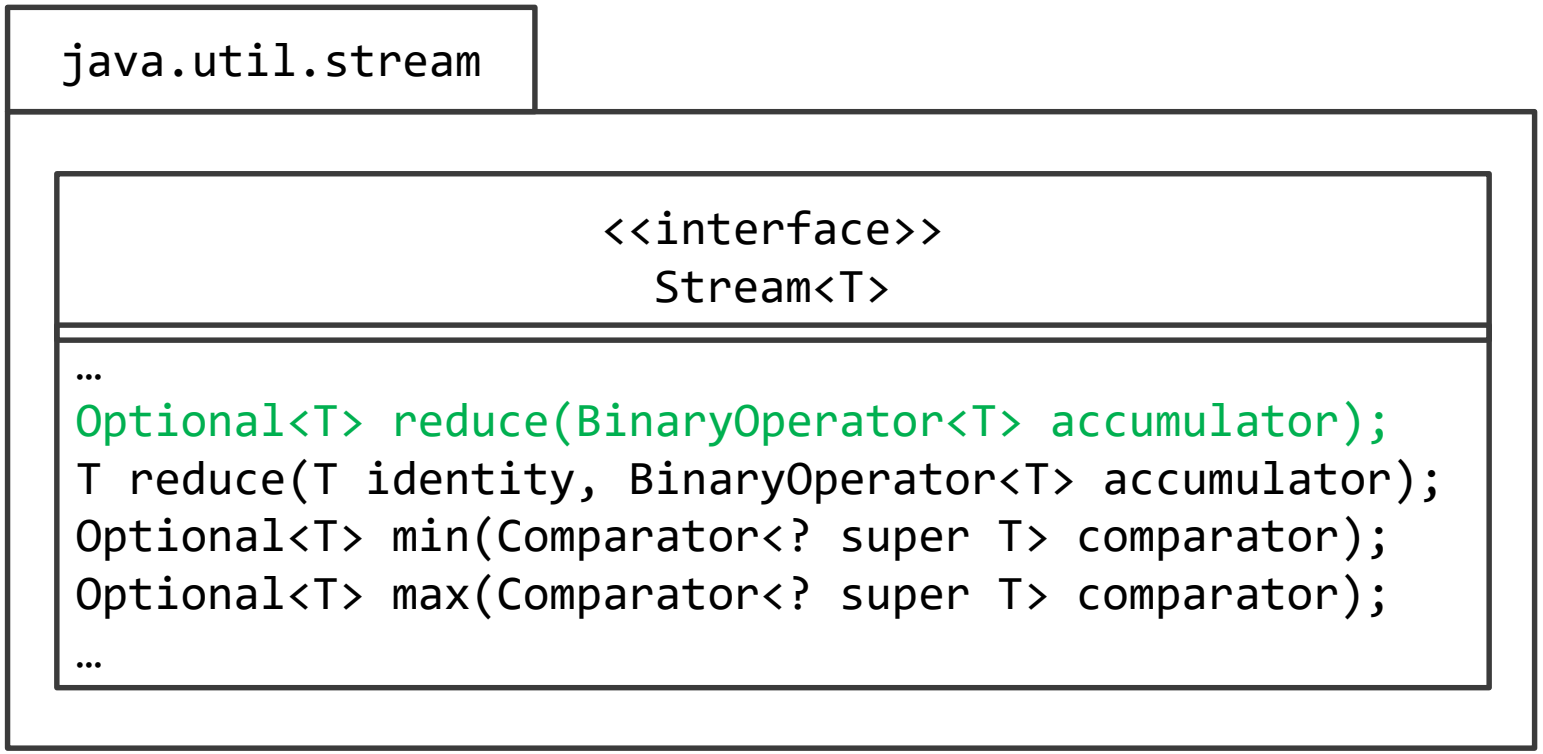
Burt Bronski Bruno


Die B-Brüder wollen herausfinden, wie stark ihre Organisation ist. Dazu werden die Rekorde der verschiedenen B-Brüder zusammengestellt.

# Zunächst müssen wir die B-Brüder geeignet modellieren:

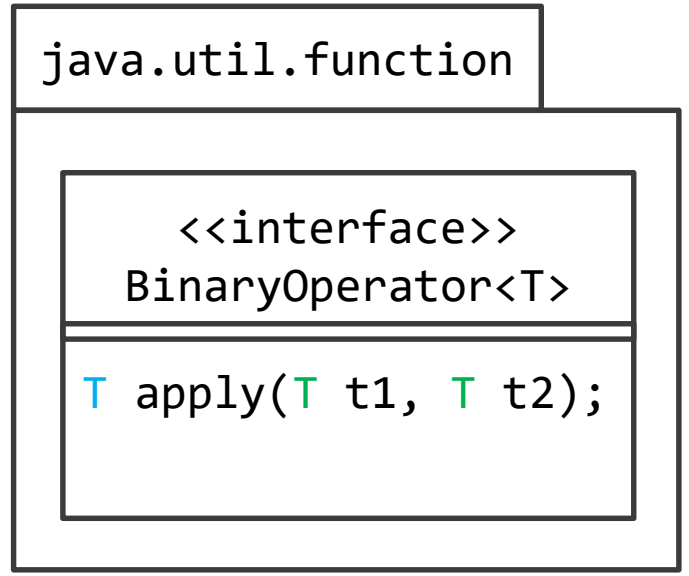
```
public class BBruder
{
  String name;
  int bankDrueckenGewicht;
  int knieBeugenGewicht;

  public BBruder(String name, int bankDrueckenGewicht, int knieBeugenGewicht)
  {
    this.name = name;
    this.bankDrueckenGewicht = bankDrueckenGewicht;
    this.knieBeugenGewicht = knieBeugenGewicht;
  }

  @Override
  public String toString()
  {
    return String.format("Name: %s, Bankdrücken: %dkg, Kniebeugen %dkg",
      name, bankDrueckenGewicht, knieBeugenGewicht);
  }
}
```

Wir wollen in jeder Disziplin ermitteln, was die beste Leistung ist.



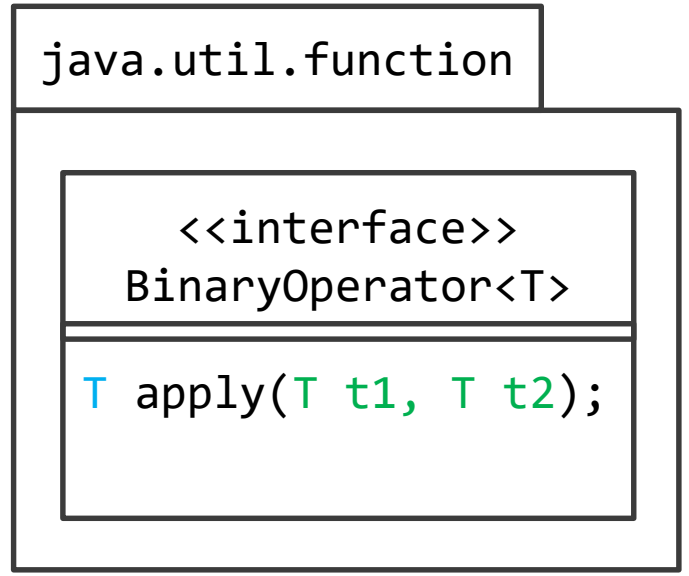

Durch die reduce-Methode werden solange immer zwei Objekte des Streams zu einem Objekt zusammengeführt, bis noch ein Objekt übrig ist.

```
List<BBruder> bBrothers = List.of(
  new BBruder("Burt", 238, 311),
  new BBruder("Bronski", 200, 274),
  new BBruder("Bruno", 236, 328));

//reduce with BinaryOperator
Optional<BBruder> best = bBrothers.stream()
 .reduce(new BinaryOperator<BBruder>()
  {
    @Override
    public BBruder apply(BBruder b1, BBruder b2)
    {
      return new BBruder(
        "Bester",
        b1.bankDrueckenGewicht > b2.bankDrueckenGewicht ? b1.bankDrueckenGewicht : b2.bankDrueckenGewicht,
        b1.knieBeugenGewicht > b2.knieBeugenGewicht ? b1.knieBeugenGewicht : b2.knieBeugenGewicht);
    }
  });
best.ifPresent(b -> System.out.println(b));
```



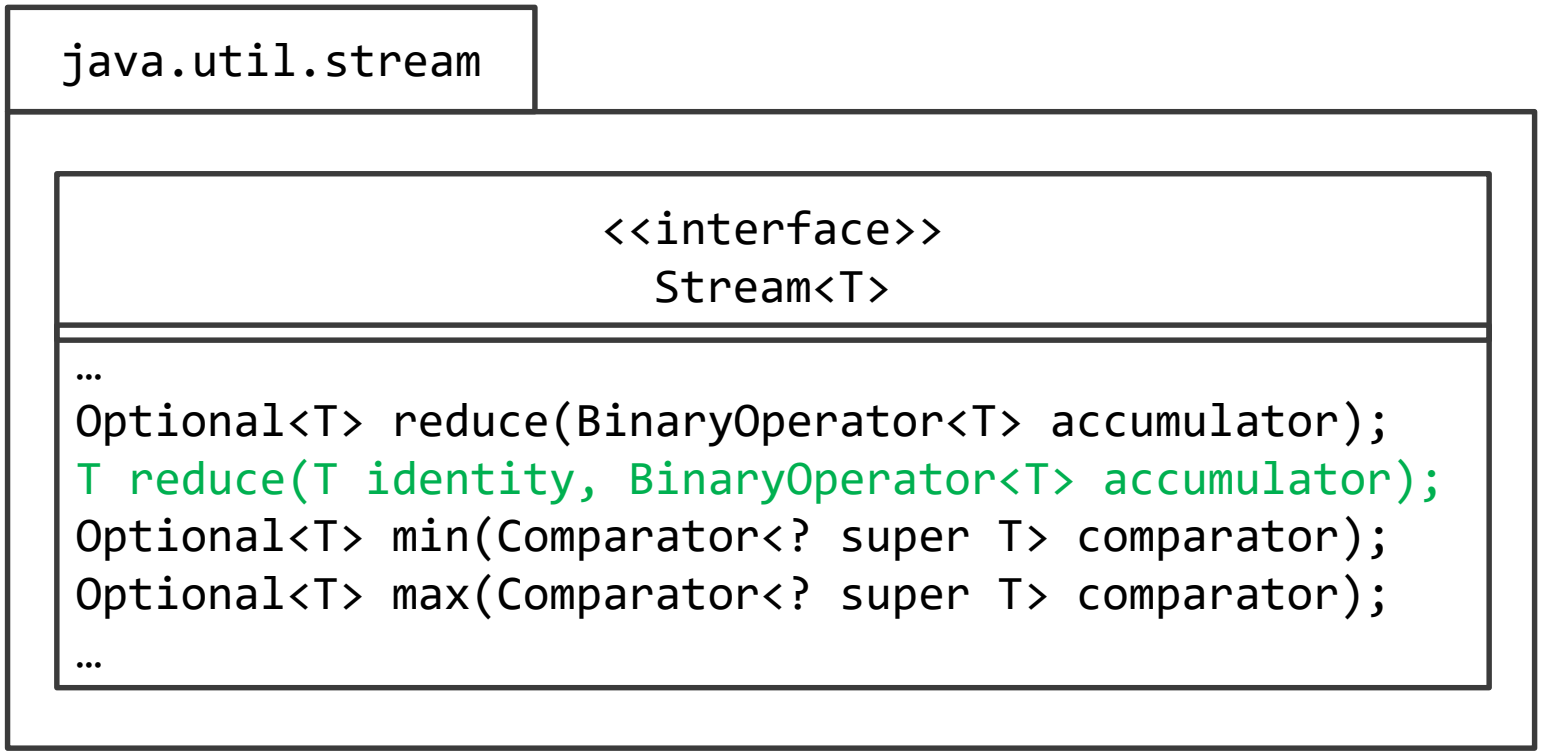

Die reduce-Methode gibt ein Optional mit dem (zusammengeführten) Objekt zurück. Das Optional ist empty, wenn der Stream leer war.

# Kürzer mit einem Lambda-Ausdruck:




```
List<BBruder> bBrothers = List.of(
  new BBruder("Burt", 238, 311),
  new BBruder("Bronski", 200, 274),
  new BBruder("Bruno", 236, 328));

//reduce with BinaryOperator
bBrothers.stream()
  .reduce((b1,b2) -> new BBruder(
    "Bester",
    b1.bankDrueckenGewicht > b2.bankDrueckenGewicht ? b1.bankDrueckenGewicht : b2.bankDrueckenGewicht,
    b1.knieBeugenGewicht > b2.knieBeugenGewicht ? b1.knieBeugenGewicht : b2.knieBeugenGewicht))
  .ifPresent(b -> System.out.println(b));
```

# Es gibt noch alternative reduce-Methoden:



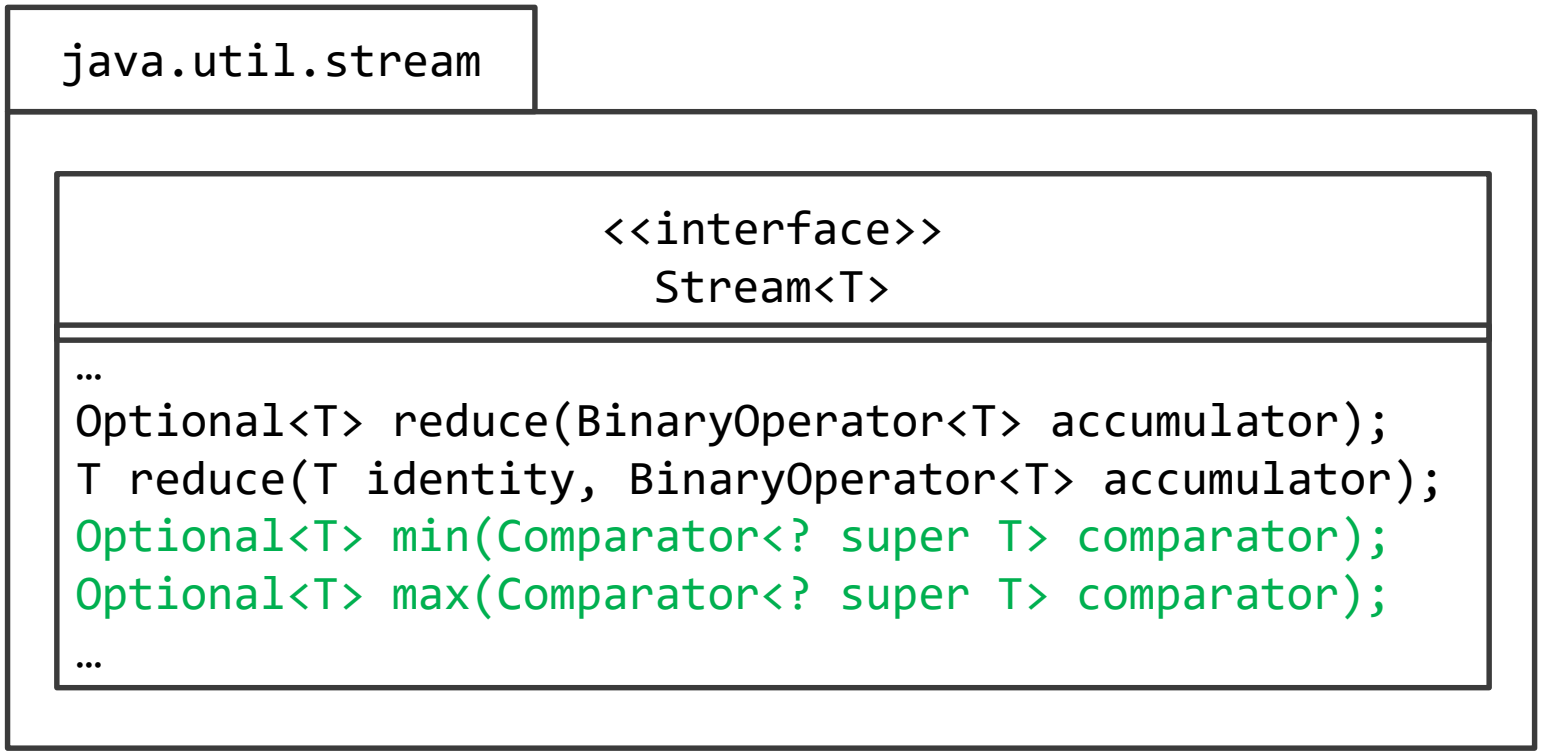

Sinngemäß wird das identity-Objekt **vor** dem reduce-Vorgang im Stream ergänzt. Hierdurch wird sicher gestellt, dass selbst wenn der Stream leer war, immer ein Objekt zurückgegeben werden kann.

```
List<BBruder> bBrothers = List.of(
    new BBruder("Burt", 238, 311),
    new BBruder("Bronski", 200, 274),
    new BBruder("Bruno", 236, 328));

BBruder best = bBrothers.stream()
  .reduce(new BBruder("Bester", 0, 0),
    (b1, b2) -> new BBruder(
      "Bester",
      b1.bankDrueckenGewicht > b2.bankDrueckenGewicht ? b1.bankDrueckenGewicht : b2.bankDrueckenGewicht,
      b1.knieBeugenGewicht > b2.knieBeugenGewicht ? b1.knieBeugenGewicht : b2.knieBeugenGewicht));
System.out.println(best);
```

Sinngemäß wird dem Stream zuerst ein neuer BBruder („Bester“) hinzugefügt.

Im weiteren Durchlauf werden solange immer zwei B-Brüder miteinander verglichen bis nur noch einer übrig ist.

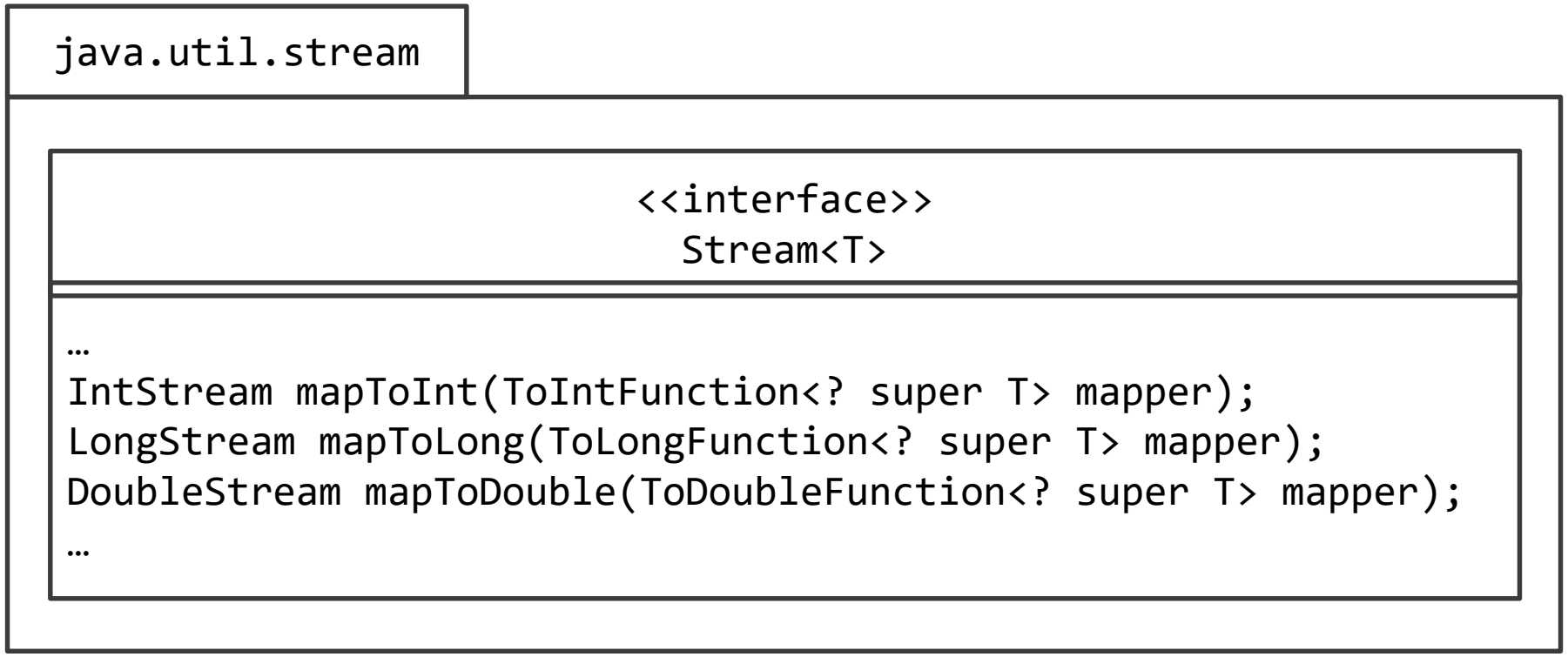
Weiterhin gibt es spezialisierte reduce-Methoden. Diese Methoden ähneln den Aggregatoren aus dem Datenbankbereich.

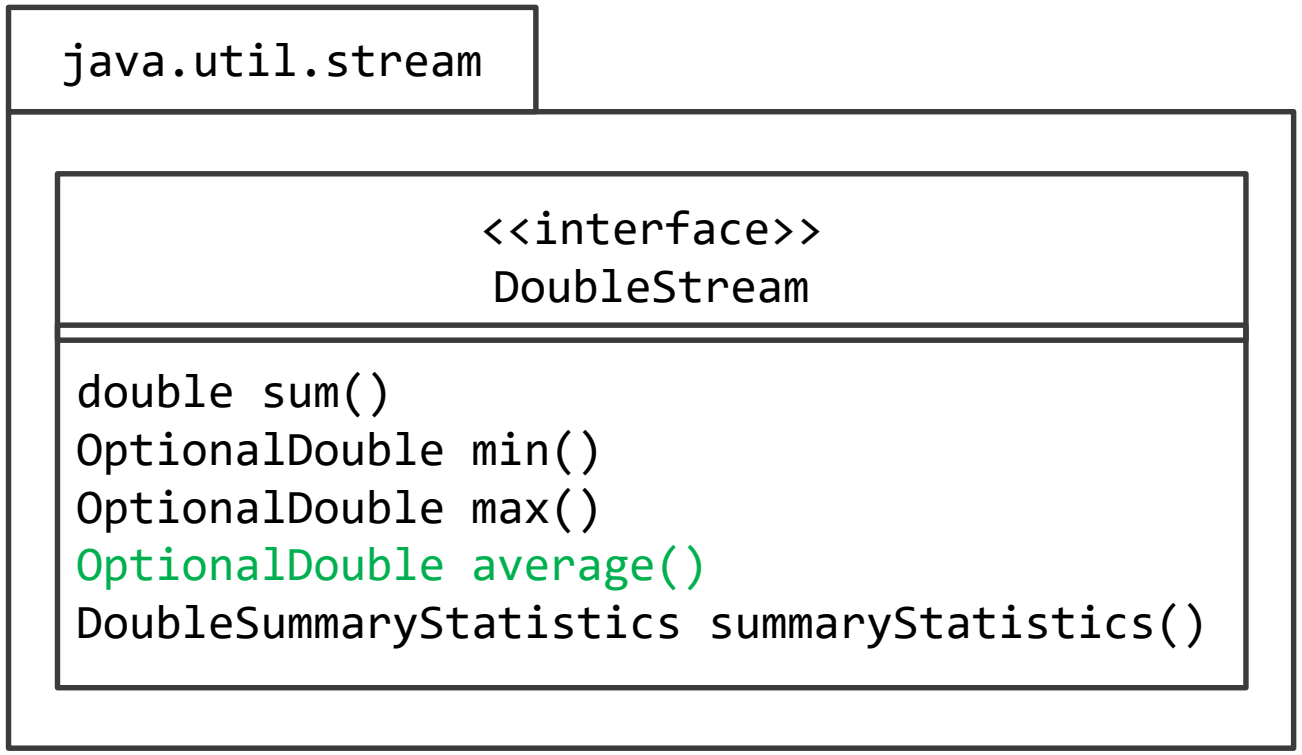



min und max geben anhand einer Comparators ein Optional mit dem kleinsten bzw. größten Element des Streams. Falls der Stream leer ist, wird ein leeres Optional zurückgegeben.

Für Streams von Zahlen (int, long, double) gibt es noch weitere spezialisierte Reduction-Methoden.

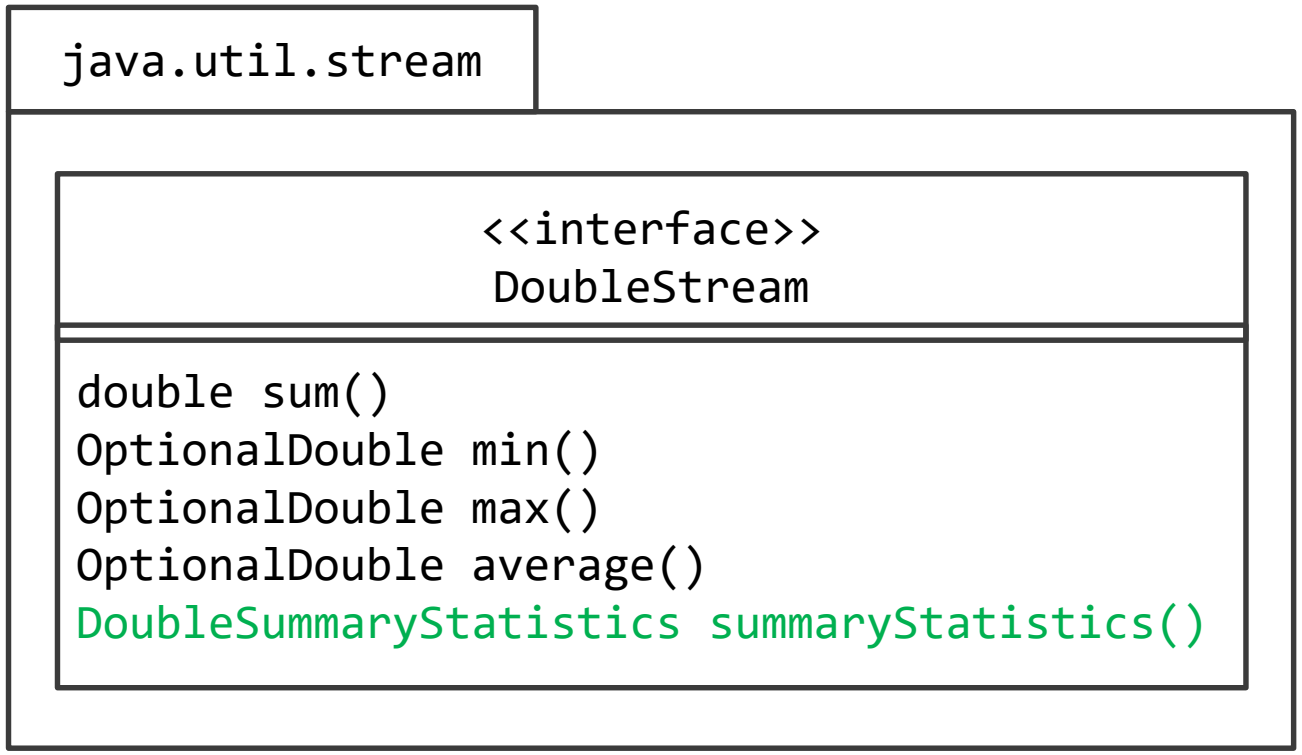
Da die (allgemeine) Stream-Klasse nicht über diese Methoden verfügt, gibt es mapping-Methoden auf spezielle Stream-Klassen.






# Spezialisierte Reduce-Methoden von Int-, Long-, und DoubleStream

```
List<BBruder> bBrothers = List.of(
    new BBruder("Burt", 238, 311),
    new BBruder("Bronski", 200, 274),
    new BBruder("Bruno", 236, 328));

bBrothers.stream()
    .mapToDouble(p -> p.bankDrueckenGewicht)
    .average()
    .ifPresent(System.out::println);
```




# Spezialisierte Reduce-Methoden von Int-, Long-, und DoubleStream

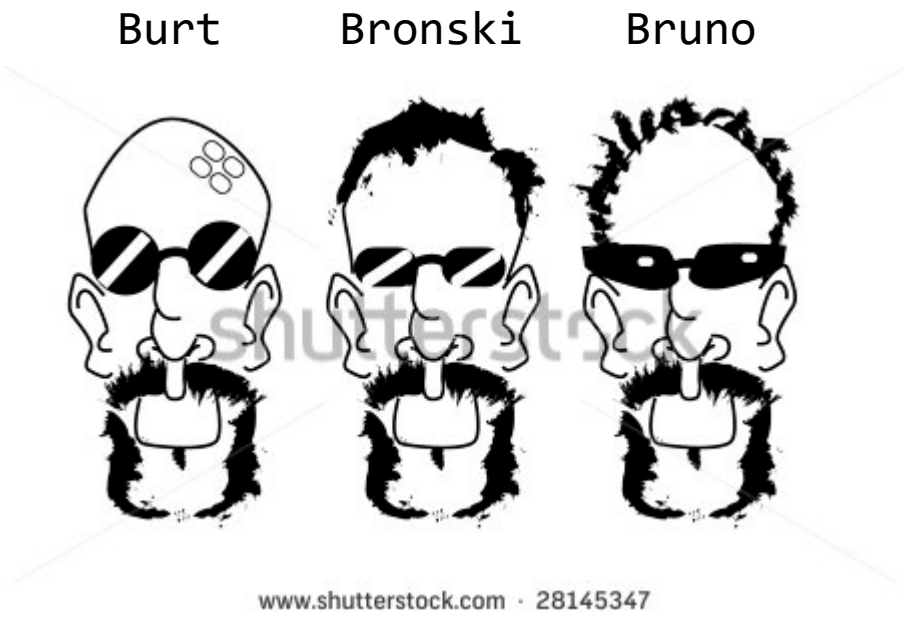
```
List<BBruder> bBrothers = List.of(
    new BBruder("Burt", 238, 311),
    new BBruder("Bronski", 200, 274),
    new BBruder("Bruno", 236, 328));

DoubleSummaryStatistics summaryStatistics = bBrothers.stream()
        .mapToDouble(p -> p.bankDrueckenGewicht)
        .summaryStatistics();
        System.out.println(summaryStatistics.toString());
```

Ausgabe:
DoubleSummaryStatistics{count=3, sum=674,000000, min=200,000000, average=224,666667, max=238,000000}

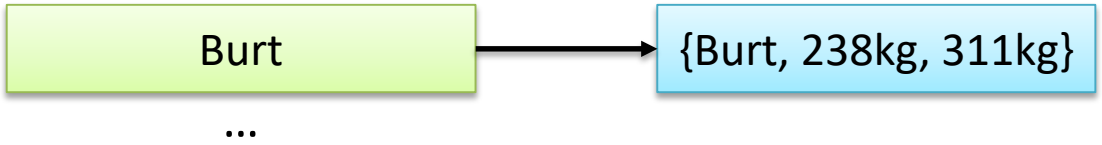
# Collectors II

toMap

Durch das kürzliche Hinzukommen eines Mitglieds hat die Organisation beschlossen, eine Kartei für die verschiedenen Mitglieder anzulegen.




Auf die Kartei soll anhand des Namens zugegriffen werden können.



Wir bilden die Kartei mit einer Map ab, bei der der Name auf das B-Bruder-Objekt verweist.

Zum Erzeugen der Map wird der Collector toMap eingesetzt:

java.util.stream

**Collectors**

```
<T, K, U> Collector<T, ?, Map<K,U>> toMap(Function<? super T, ? extends K> keyMapper,
                                          Function<? super T, ? extends U> valueMapper)
…
```

Der KeyMapper legt fest, wie ein Objekt des Streams auf einen Key für die Map abgebildet wird.

Der ValueMapper legt fest, wie ein Objekt des Streams auf einen Value für die Map abgebildet wird.

```
List<BBruder> bBrothers = List.of(
    new BBruder("Burt", 238, 311),
    new BBruder("Bronski", 200, 274),
    new BBruder("Bruno", 236, 328));

Map<String, BBruder> map = bBrothers.stream()
  .collect(Collectors.toMap(b -> b.name, b -> b));
System.out.println(map);
```

toMap erstellt für jedes B-Bruder des Streams einen Map-Eintrag mit
dem Namen des B-Bruders als Key und
dem B-Bruder-Objekt selbst als Value.

# Mapping II

flatMap

distinct

sorted

Die B-Brüder bilden sich durch die Lektüre von Kampfsportbüchern fort.

Burt Bronski Bruno


Die Organisation möchte einen Überblick über alle Bücher ihrer Mitglieder bekommen.

# Wir ergänzen die B-Bruder-Klasse um eine Bücherliste.

```
public class BBruder
{
  String name;
  int bankDrueckenGewicht;
  int knieBeugenGewicht;
  List<String> kampfsportBuecher = new ArrayList<>();

  public BBruder add(String... buecher)
  {
    kampfsportBuecher.addAll(List.of(buecher));
    return this;
  }
  …
}
```

# Erste Lösung:

```
List<BBruder> bBrothers = List.of(
  new BBruder("Burt", 238, 311).add("The Dow of Master Ken: Vol. 1", "The TB12 Method"),
  new BBruder("Bronski", 200, 274)
    .add("Das Buch der fünf Ringe", "Shaolin - Du musst nicht kämpfen, um zu siegen!"),
  new BBruder("Bruno", 236, 328).add("Tao des Jeet Kune Do", "Das Buch der fünf Ringe"));

List<String> buecher = new ArrayList<>();
bBrothers.stream()
  .map(b -> b.kampfsportBuecher)
  .forEach(k -> buecher.addAll(k));
buecher.forEach(System.out::println);
```

Wir mappen jeden B-Bruder auf seine Bücher und fügen diese dann einer Collection hinzu und geben diese aus.

forEach kann man auch direkt auf einer Collection ausführen.

# Wir können auch innerhalb der Fluent-API der Stream-Klassen bleiben, indem wir flatMap verwenden.

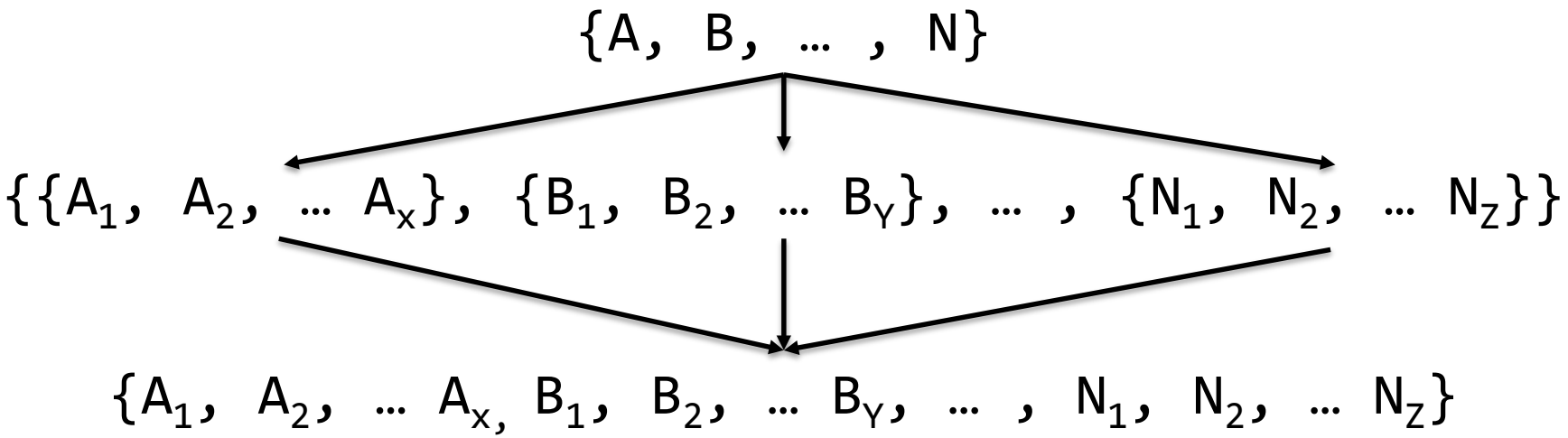
java.util.stream

<<interface>>
Stream<T>

```
…
<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper);
Stream<T> sorted();
Stream<T> sorted(Comparator<? super T> comparator);
Stream<T> distinct();
```

flatMap mapped (vereinfacht gesagt)
ein **Element vom Typ T** aus dem Stream auf einen **Stream mit Typ R**
und legt die verschiedenen Streams zu einem einzelnen Stream zusammen.

Auf mathematische Mengen bezogen macht die flatMap-Methode die folgenden Zuordnungen:




Jedes Element einer Menge wird auf eine Menge abgebildet. In einem zweiten Schritt werden alle Mengen zu einer Menge vereinigt.

# Lösungsansatz mit flatMap und anonymer Klasse

```
List<BBruder> bBrothers = List.of(
  new BBruder("Burt", 238, 311).add("The Dow of Master Ken: Vol. 1", "The TB12 Method"),
  new BBruder("Bronski", 200, 274)
    .add("Das Buch der fünf Ringe", "Shaolin - Du musst nicht kämpfen, um zu siegen!"),
  new BBruder("Bruno", 236, 328).add("Tao des Jeet Kune Do", "Das Buch der fünf Ringe"));

bBrothers.stream()
  .flatMap(new Function<BBruder, Stream<String>>() {
     @Override
     public Stream<String> apply(BBruder b)
     {
       return b.kampfsportBuecher.stream();
     }
   })
   .forEach(System.out::println);
```

Für jeden B-Bruder wird eine Function erstellt, die den B-Bruder auf einen Stream<String> von Büchern mapped (und zu einem Stream zusammenlegt).

# Lösungsansatz mit flatMap und einem Lambda-Ausdruck

```
List<BBruder> bBrothers = List.of(
  new BBruder("Burt", 238, 311).add("The Dow of Master Ken: Vol. 1", "The TB12 Method"),
  new BBruder("Bronski", 200, 274)
    .add("Das Buch der fünf Ringe", "Shaolin - Du musst nicht kämpfen, um zu siegen!"),
  new BBruder("Bruno", 236, 328).add("Tao des Jeet Kune Do", "Das Buch der fünf Ringe"));

bBrothers.stream()
  .flatMap(b -> b.kampfsportBuecher.stream())
  .forEach(System.out::println);
```

Ausgabe:

The Dow of Master Ken: Vol. 1
The TB12 Method
Das Buch der fünf Ringe
Shaolin - Du musst nicht kämpfen, um zu siegen!
Tao des Jeet Kune Do
Das Buch der fünf Ringe

Bei der Ausgabe taucht ein Buch doppelt auf. Das lässt sich durch die Methode distinct beheben.

```
List<BBruder> bBrothers = List.of(
  new BBruder("Burt", 238, 311).add("The Dow of Master Ken: Vol. 1", "The TB12 Method"),
  new BBruder("Bronski", 200, 274)
    .add("Das Buch der fünf Ringe", "Shaolin - Du musst nicht kämpfen, um zu siegen!"),
  new BBruder("Bruno", 236, 328).add("Tao des Jeet Kune Do", "Das Buch der fünf Ringe"));

bBrothers.stream()
  .flatMap(b -> b.kampfsportBuecher.stream())
  .distinct()
  .forEach(System.out::println);
```

Ausgabe:

The Dow of Master Ken: Vol. 1
The TB12 Method
Das Buch der fünf Ringe
Shaolin - Du musst nicht kämpfen, um zu siegen!
Tao des Jeet Kune Do

distinct gibt einen neuen Stream ohne Duplikate zurück.

# Wir können die Elemente des Streams zusätzlich sortieren.

```
List<BBruder> bBrothers = List.of(
  new BBruder("Burt", 238, 311).add("The Dow of Master Ken: Vol. 1", "The TB12 Method"),
  new BBruder("Bronski", 200, 274)
    .add("Das Buch der fünf Ringe", "Shaolin - Du musst nicht kämpfen, um zu siegen!"),
  new BBruder("Bruno", 236, 328).add("Tao des Jeet Kune Do", "Das Buch der fünf Ringe"));

bBrothers.stream()
  .flatMap(b -> b.kampfsportBuecher.stream())
  .distinct()
  .sorted()
  .forEach(System.out::println);
```

Ausgabe:

Das Buch der fünf Ringe
Shaolin - Du musst nicht kämpfen, um zu siegen!
Tao des Jeet Kune Do
The Dow of Master Ken: Vol. 1
The TB12 Method

Die sorted-Methode sortiert nach der natürlichen Ordnung.

java.util.stream

<<interface>>
Stream<T>

```
…
<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper);
Stream<T> sorted();
Stream<T> sorted(Comparator<? super T> comparator);
Stream<T> distinct();
```

sorted kann als Parameter auch einen Comparator nutzen.

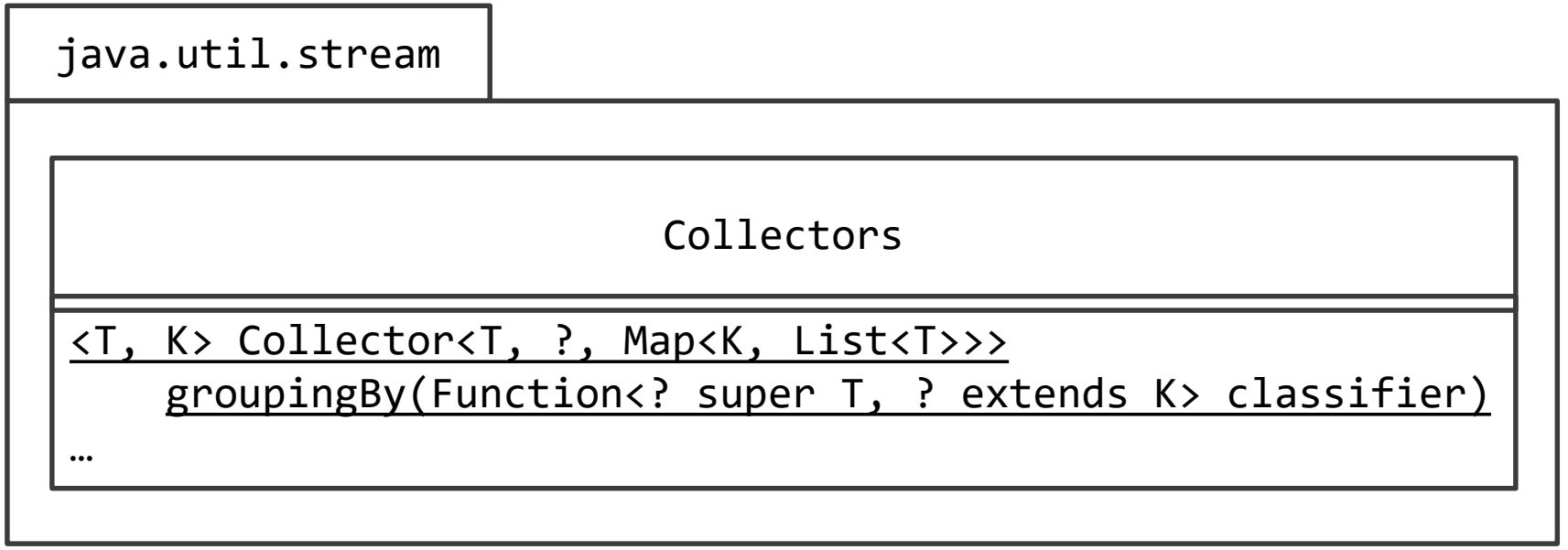
# Collectors III

groupingBy

Wir wollen die B-Brüder in belesene und unbelesene B-Brüder unterteilen.

Burt Bronski Bruno


Belesene B-Brüder haben mind. zwei Bücher gelesen unbelesene weniger als 2 Bücher.




groupingBy liefert einen Collector zurück, der einen Stream anhand einer Function in verschiedene Teillisten unterteilt und diese über eine Map verfügbar macht.

```
List<BBruder> bBrothers = List.of(
    new BBruder("Burt", 238, 311).add("The Dow of Master Ken: Vol. 1", "The TB12 Method"),
    new BBruder("Bronski", 200, 274).add("Das Buch der fünf Ringe",
      "Shaolin - Du musst nicht kämpfen, um zu siegen!"),
    new BBruder("Bruno", 236, 328).add("Das Buch der fünf Ringe"));

Map<Boolean, List<BBruder>> map = bBrothers.stream()
    .collect(Collectors.groupingBy(b -> b.kampfsportBuecher.size() >= 2));
map.forEach((k,v) -> System.out.println(k ? "Mehr als zwei Bücher gelesen:  " + v
                                          : "Höchstens zwei Bücher gelesen: " + v));
```

Ausgabe:

Höchstens zwei Bücher gelesen: [Name: Bruno, Bankdrücken: 236kg, Kniebeugen 328kg]

Mehr als zwei Bücher gelesen:  [Name: Burt, Bankdrücken: 238kg, Kniebeugen 311kg, Name: Bronski, Bankdrücken: 200k

# Wir können die B-Brüder auch nach der Anzahl gelesener Bücher unterteilen.

Burt Bronski Bruno

```
List<BBruder> bBrothers = List.of(
    new BBruder("Burt", 238, 311).add("The Dow of Master Ken: Vol. 1", "The TB12 Method"),
    new BBruder("Bronski", 200, 274).add("Das Buch der fünf Ringe",
      "Shaolin - Du musst nicht kämpfen, um zu siegen!"),
    new BBruder("Bruno", 236, 328).add("Das Buch der fünf Ringe"));

Map<Integer, List<BBruder>> map = bBrothers.stream()
    .collect(Collectors.groupingBy(b -> b.kampfsportBuecher.size()));
map.forEach((k,v) -> System.out.println(k + (k == 1 ? " Buch gelesen: " : " Bücher gelesen: ") + v));
```

Ausgabe:

1 Buch gelesen: [Name: Bruno, Bankdrücken: 236kg, Kniebeugen 328kg]

2 Bücher gelesen: [Name: Burt, Bankdrücken: 238kg, Kniebeugen 311kg, Name: Bronski, Bankdrücken: 200kg, Kniebeugen 274kg]