Aufbereitung und Verknüpfung nicht textueller Werte Gerade haben wir gesehen, wie einfach die kommaseparierte Aufbereitung von textuellen Werten mithilfe von Lambdas und den Neuerungen im Stream-API implementiert werden kann.

Etwas mehr Aufwand ist erforderlich, wenn man Zahlen oder Objekte auf diese Weise miteinander verknüpfen möchte und beispielsweise die Altersangaben von Personen kommasepariert aufbereitet. Dazu müssen wir die Objekte, hier vom Typ `Integer`, in Strings wandeln, bevor wir sie mit `joining()` verknüpfen können. Die Umwandlung kann explizit durch einen Aufruf von `toString()` erfolgen oder aber implizit durch die Notation `"" + value`. Im folgenden Listing sind beide Varianten gezeigt:

```
// Explizite Umwandlung / Mapping mit toString
final String joined1 = persons.stream().mapToInt(Person::getAge)
                                       .mapToObj(Integer::toString)
                                       .collect(joining(", "));

// Implizite Umwandlung / Mapping durch "" + value
final String joined2 = persons.stream().map( person -> "" + person.getAge() )
                                       .collect(joining(", "));
```

Hinweis: Die Methode `String.join()`

Wenn lediglich textuelle Werte miteinander verknüpft werden sollen, und dabei nur eine Verknüpfungszeichenfolge, aber keine Start- und Endkennungen benötigt werden, dann kann man die in der Klasse `String` mit JDK 8 eingeführte Methode `join(CharSequence delimiter, CharSequence... elements)` nutzen:[a]

```
final String stringConcat = String.join(", ", names);
```

Etwas unglücklich empfinde ich die Signatur, in der der Delimiter vor den zu verknüpfenden Elementen angegeben werden muss. Das liegt aber einfach daran, dass in Java Varargs nur für den letzten Parameter in einer Signatur auftreten dürfen.

[a] Wir haben zuvor mit `joinStrings()` eine ähnliche Methode entworfen.

3.5 Datenaufbereitung mit Kollektoren

In den vorangegangenen Abschnitten haben wir uns intensiver mit der Verarbeitung von Daten mithilfe von Streams beschäftigt und dabei auch schon einige vordefinierte Kollektoren genutzt, etwa `toList()`, `joining()`, `groupingBy()`. Kollektoren sind ein so mächtiges Instrument, um Daten aus Streams aufzubereiten, dass ich hier nochmals etwas genauer darauf eingehen möchte. Neben Transformationen in Instanzen einer Containerklasse wie `List<E>` werden Kollektoren auch dazu genutzt, die Elemente eines Streams in ein kumuliertes Resultat zu überführen, etwa um die Summe, das Maximum, Minimum oder den Durchschnitt zu berechnen.

Beginnen wir mit einem kurzen Blick auf das Interface `Collector<T,A,R>` und die Methode `collect()`. Zum leichten Einstieg in die Thematik schauen wir uns da-

nach Varianten der bereits genutzten Kollektoren an und beginnen mit `joining()`, bevor wir dann weitere vordefinierte Kollektoren anhand von Beispielen kennenlernen werden bzw. das Wissen darüber vertiefen.

3.5.1 Das Interface `Collector` und die Methode `collect()`

Kollektoren sind vor allem in Kombination mit der Methode `collect()` besonders nützlich. In diesem Rahmen haben wir sie auch schon des Öfteren genutzt – wahrscheinlich ohne auf die ganzen generischen Parameter zu schauen und uns darüber Gedanken zu machen, wie das genau funktioniert:

```
<R, A> R collect(Collector<? super T, A, R> collector);
```

Ein solcher `Collector<T,A,R>` besteht aus folgenden vier Komponenten:

- einem `Supplier<A>`, um eine initiale Datenstruktur bereitzustellen,
- einem `BiConsumer<A,T>` für Berechnungen,
- einem `BinaryOperator<A>`, um Zwischenergebnisse zusammenzuführen, und
- einer `Function<A,R>` zum Berechnen des Ergebnisses.

Daher ist das Interface `java.util.stream.Collector<T,A,R>` im JDK wie folgt definiert:

```
public interface Collector<T, A, R>
{
   /**
    * A function that creates and returns a new mutable result container.
    */
   Supplier<A> supplier();

   /**
    * A function that folds a value into a mutable result container.
    */
   BiConsumer<A, T> accumulator();

   /**
    * A function that accepts two partial results and merges them.
    */
   BinaryOperator<A> combiner();

   /**
    * Perform the final transformation from the intermediate accumulation type
    * {@code A} to the final result type {@code R}.
    */
   Function<A, R> finisher();

   // ...
}
```

Wollte man dieses Interface selbst implementieren, wäre das ziemlich kompliziert und potenziell fehlerträchtig. Glücklicherweise nimmt uns die Utility-Klasse `Collectors` durch die dort definierten Kollektoren einige Arbeit ab.

Der Vollständigkeit halber sei erwähnt, dass es eine überladene Variante der Methode `collect()` gibt, der man einzelne Bestandteile eines Kollektors übergeben kann:

```
<R> R collect(Supplier<R> supplier,
              BiConsumer<R, ? super T> accumulator,
              BiConsumer<R, R> combiner);
```

Die drei Parameter sind analog zu denen im Interface `Collector<T,A,R>` und dienen zum Bereitstellen einer initialen Datenstruktur, dem Zusammenfassen von Elementen und dem Zusammenführen der zuvor über Akkumulatoren ermittelten Teilergebnisse.

```
final List<String> names = personsStream.map(Person::getName)
                                        .collect(ArrayList::new,
                                                 ArrayList::add,
                                                 ArrayList::addAll);
```

Nach einem Blick auf `joining()` werden wir feststellen, dass man die oben genutzten drei Bestandteile besser verständlicher und kürzer mithilfe des Kollektors `toList()` aus der Klasse `Collectors` wie folgt schreibt:

```
final List<String> strings = personsStream.map(Person::getName)
                                          .collect(Collectors.toList());
```

3.5.2 Vertiefung bereits vorgestellter Kollektoren

Wir schauen zur Erinnerung im Anschluss zunächst nochmals die Kollektoren `joining()` sowie `toCollection()` bzw. `toList()` an.

Varianten von `joining()`

Die Methode `joining()` ist überladen: Deren parameterlose Variante konkateniert die Elemente eines Streams direkt hintereinander. Die Variante mit drei Parametern erlaubt neben der Festlegung der Trennzeichenfolge auch eine Angabe von Start- und Abschlusszeichenfolge. Ein plakatives Beispiel zeigt das folgende Listing:

```
public static void main(final String[] args)
{
    final String[] words = { "This", "Is", "The", "Joining", "Collector" };

    // 3 Varianten der Kombination
    System.out.println(Arrays.stream(words).collect(joining()));
    System.out.println(Arrays.stream(words).collect(joining(" | ")));
    System.out.println(Arrays.stream(words).collect(joining(", ", "[", "]")));

    final Stream<String> names = Stream.of("Andy", "Dirk", "Merten");
    final String prefix = "Many thanks to ";
    final String postfix = " for your comments!";
    System.out.println(names.collect(joining(", ", prefix, postfix)));
}
```

Listing 3.24 *Ausführbar als* **'COLLECTORSJOININGEXAMPLE'**

Startet man das Programm COLLECTORSJOININGEXAMPLE, so erkennt man sehr gut die Arbeitsweise der Varianten:

```
ThisIsTheJoiningCollector
This | Is | The | Joining | Collector
[This, Is, The, Joining, Collector]
Many thanks to Andy, Dirk, Merten for your comments!
```

Aufbereitung als Collection

Häufiger als in einer simplen Konkatenation besteht die Anforderung darin, die in einem Stream existierenden Daten in eine Instanz einer Containerklasse zu überführen, etwa in eine Ergebnisliste oder in eine Menge wie in diesem Beispiel:

```
final List<Person> persons = personStream.collect(Collectors.toList());
final Set<Integer> uniqueNames = namesStream.collect(Collectors.toSet());
```

Ähnliches haben wir zuvor schon einige Male genutzt. Schauen wir hier etwas genauer auf die Angabe einer gewünschten Ergebnisdatenstruktur.

Manchmal möchte man den Typ der resultierenden Containerklasse bestimmen können. Das kann man durch Angabe eines `Supplier<T>` erreichen:

```
Collectors.toCollection(Supplier<T>)
```

Das Functional Interface `Supplier<T>` wurde kurz in Abschnitt 2.2.1 vorgestellt. Dort wurde darauf hingewiesen, dass man es für Factory-Funktionalität nutzen kann. Im folgenden Beispiel wird bewusst der `Collector<T,A,R>` als eigene Variable definiert, um zu verdeutlichen, dass die Methoden aus der Utility-Klasse `Collectors` Instanzen vom Typ `Collector<T,A,R>` liefern. Dadurch lassen sich die Details der Verarbeitung einfacher zeigen. In der Praxis wäre ein Aufruf von `toCollection(TreeSet::new)` zu bevorzugen – für dieses Beispiel hätte ich aber weder die beteiligten Interfaces noch die dahinterstehende Komplexität aufzeigen können:

```
public static void main(final String[] args)
{
    final Supplier<TreeSet<String>> collectionFactory = TreeSet::new;
    final Collector<String, ?, TreeSet<String>> sortingCollector =
                    Collectors.toCollection(collectionFactory);

    final Stream<String> letters = Stream.of("a", "b", "X", "B", "A");
    final TreeSet<String> sortedLetters = letters.collect(sortingCollector);
    System.out.println(sortedLetters);
}
```

Listing 3.25 *Ausführbar als* **'COLLECTORSTOCOLLECTIONEXAMPLE'**

Das Programm COLLECTORSTOCOLLECTIONEXAMPLE wandelt einen Stream in ein `TreeSet<E>`, wodurch die Eingabedaten automatisch sortiert werden:

```
[A, B, X, a, b]
```

3.5.3 Aufbereitung statistischer Daten

Mitunter sollen Berechnungen auf den durch einen Stream bereitgestellten Daten ausgeführt werden, z. B. eine Summierung, Durchschnittsbildung usw., Dazu haben wir bereits verschiedene Terminal Operations, etwa `count()`, `sum()`, `min()`, `max()` und `average()`, in Abschnitt 3.3.5 genutzt. Interessanterweise gibt es ähnliche Methoden nochmals im Bereich der Kollektoren:

- `counting()` – Zählt die Elemente im Stream.
- `summingInt()/-Long()/-Double()` – Berechnet die Summe der Elemente des Streams.
- `averagingInt()/-Long()/-Double()` – Berechnet den Durchschnitt der Elemente des Streams.
- `maxBy(Comparator<T>)` bzw. `minBy(Comparator<T>)` – Berechnet das Maximum bzw. Minimum der Elemente des Streams bezüglich des übergebenen `Comparator<T>`s.
- `summarizingInt()/-Long()/-Double()` – Sind mehrere dieser statistischen Werte zu ermitteln, so lassen sich mit diesen `summarizingXYZ()`-Methoden, die Berechnungen in einem Rutsch ausführen.

Die Arbeitsweise der obigen Kollektoren verdeutlicht das unten gezeigte Programm, das im Prinzip immer wieder Aktionen ähnlich zu folgender ausführt:

```
final List<Integer> ints = Arrays.asList(1,2,3,4,5,6,7,8,9,10);
System.out.println("averagingInt(): " + ints.stream().
                                          collect(averagingInt(x -> x)));
```

Um das Ganze etwas kürzer und vom Design her eleganter zu gestalten, definiere ich diese Kollektoren in Form einer Map. Die anschließende Verarbeitung wird dann durch eine Iteration über diese und den Aufruf von `collect()` folgendermaßen realisiert:

```
public static void main(final String args[])
{
    final Map<String, Collector<Integer, ?, ?>> collectorsMap = new HashMap<>();
    collectorsMap.put("counting():       ", counting());
    collectorsMap.put("summingInt():     ", summingInt(x -> x));
    collectorsMap.put("averagingInt():   ", averagingInt(x -> x));
    collectorsMap.put("maxBy():          ", maxBy(Integer::compare));
    collectorsMap.put("minBy():          ", minBy(Integer::compare));
    collectorsMap.put("summarizingInt(): ", summarizingInt(x -> x));

    for (final Map.Entry<String, Collector<Integer, ?, ?>> mapping :
                                                collectorsMap.entrySet())
    {
        final Collector<Integer, ?, ?> collector = mapping.getValue();

        final List<Integer> ints = Arrays.asList(1,2,3,4,5,6,7,8,9,10);
        System.out.println(mapping.getKey() + ints.stream().collect(collector));
    }
}
```

Listing 3.26 *Ausführbar als* **'COLLECTORSCALCULATIONSEXAMPLE'**

Das Programm COLLECTORSCALCULATIONSEXAMPLE erzeugt folgende Ausgabe:

```
averagingInt():   5.5
counting():       10
maxBy():          Optional[10]
minBy():          Optional[1]
summarizingInt(): IntSummaryStatistics{count=10, sum=55, min=1,
                                       average=5,500000, max=10}
summingInt():     55
```

Wieso existieren diese Kollektoren?

Die vorgestellten Kollektoren scheinen eine Dopplung zu den eingangs erwähnten Terminal Operations zu sein. So recht einleuchtend ist deren Existenz daher nicht.

Ein erstes Verständnis erhält man, wenn man sich verdeutlicht, dass diese Aktionen nicht nur auf Streams mit Zahlen, sondern auch auf Streams mit anderen Typen, etwa Personen, ausgeführt werden können, z. B. um die jüngste oder älteste Person aus einem Stream zu ermitteln. Aber auch da gäbe es andere Varianten. Die zuvor vorgestellten Kollektoren beginnen an Bedeutung zu gewinnen, wenn man sie mit anderen Kollektoren, insbesondere der Gruppierung und Partitionierung, in Kombination nutzt, wie wir es später kennenlernen werden.

3.5.4 Aufbereitung als Map – einfache Gruppierungen

Für einige Anwendungsfälle sollen die Daten etwas spezieller, z. B. in Form einer Map, aufbereitet werden. Das ist der einfachste Fall der Gruppierung von Daten. Beispielsweise könnte eine Liste von Personen nach deren Wohnort gruppiert werden.

Gruppierung mit JDK 7 realisieren

Bevor wir gleich anschauen, wie leicht eine Gruppierung mit JDK 8 zu realisieren ist, wollen wir zunächst rekapitulieren, welche Schritte dazu mit JDK 7 notwendig sind.

Bleiben wir bei unserer Liste von Personen. Wenn wir diese nach deren Wohnort gruppieren wollten, könnten wir in etwa folgende Methode schreiben:

```
public static Map<String, List<Person>> groupByCity(final List<Person> persons)
{
    final Map<String, List<Person>> result = new HashMap<>();

    for (final Person person : persons)
    {
        final String key = person.getCity();
        if (!result.containsKey(key))
        {
            final List<Person> personsInCity = new ArrayList<>();
            result.put(key, personsInCity);
        }
        result.get(key).add(person);
    }
    return result;
}
```