3.8 Übungen zu Collections und Bulk Operations

Lernziel: Die Stärke von Lambdas lässt sich unter anderem im Zusammenhang mit Bulk Operations für Collections erkennen. Nachfolgend vertiefen wir das Wissen zu Prädikaten, interner und externer Iteration sowie zu den neuen Hilfsmethoden in den Containerklassen bzw. deren Interfaces wie z. B. `List<E>`.

✍ Aufgabe 1 ✍

Wandeln Sie eine externe in eine interne Iteration um. Gegeben sei folgende Iteration mit einer `for`-Schleife in JDK 7:

```
final List<String> names = Arrays.asList("Tim", "Peter", "Mike");

for (final String name : names)
{
    System.out.println(name);
}
```

Tipp: Nutzen Sie die Defaultmethode `forEach()` aus dem Interface `Iterable<E>` zur Konsolenausgabe. Verwenden Sie die Methodenreferenz `System.out::println`.

✍ Aufgabe 2 ✍

Aufgabe 2a: Formulieren Sie die Bedingungen "Zahl ist gerade", "Zahl ist positiv", "Zahl ist null" und "Wert ist `null`" als `Predicate<Integer>` und, sofern möglich, auch als `IntPredicate`. Prüfen Sie diese Prädikate mit verschiedenen Eingaben.

```
final Predicate<Integer> isEven = // ... TODO ...
final Predicate<Integer> isNull = // ... TODO ...

final IntPredicate isPositive = // ... TODO ...
```

Aufgabe 2b: Formulieren Sie die Bedingung "Wort kürzer als 4 Buchstaben" und prüfen Sie das damit realisierte `Predicate<String> isShortWord`.

```
final Predicate<String> isShortWord = // ... TODO ...
```

Aufgabe 2c: Kombinieren Sie die Prädikate wie folgt und prüfen Sie wieder:

- Zahl ist positiv und Zahl ist gerade (Methode `and()`).
- Zahl ist positiv und ungerade (Aufruf von `negate()`).

Aufgabe 3

Gegeben sei folgende Klasse `Person`:

```
public class Person
{
    private final String name;
    private final int age;

    public Person(final String name, final int age)
    {
        this.name = name;
        this.age = age;
    }

    public int getAge()
    {
        return age;
    }

    public boolean isAdult()
    {
        return getAge() >= 18;
    }
    // ...
}
```

Formulieren Sie die Bedingung "18 Jahre oder älter" mit einem `Predicate<Person>`

1. durch Aufruf von `getAge()` als Lambda und
2. durch Aufruf von `isAdult()` mit Methodenreferenz.

Aufgabe 4

Sortieren Sie eine Liste von Namen basierend der natürlichen Ordnung in Form von `Comparable<String>` und geben Sie diese auf der Konsole aus. Die nachfolgend im Listing gezeigte klassische Variante soll mit JDK-8-Mitteln vereinfacht werden:

```
final List<String> names = Arrays.asList("Tim", "Peter", "Mike", "Andy");

final Comparator<String> naturalOrder = new Comparator<String>()
{
    @Override
    public int compare(final String str1, final String str2)
    {
        return str1.compareTo(str2);
    }
};

Collections.sort(names, naturalOrder);

for (final String name : names)
{
    System.out.println(name);
}
```

Tipp: Schreiben Sie einen passenden `Comparator<String>`. Nutzen Sie alternativ

- einen Lambda oder
- eine Methodenreferenz auf `String::compareTo`

als Eingabe für die Defaultmethode `sort()` aus dem Interface `List<E>`. Beachten Sie den Tipp aus Aufgabe 1, um auch die Ausgabe kürzer schreiben zu können.

Aufgabe 5

Aufgabe 5a: Löschen Sie aus einer Liste von Namen all diejenigen Einträge mit kurzem Namen. Wandeln Sie dazu die externe in eine interne Iteration um. Die JDK-7-Implementierung aus dem Listing soll mithilfe von JDK-8-Mitteln einfacher gestaltet werden:

```
private static List<String> removeIf_External_Iteration()
{
    final List<String> names = createNamesList();

    final Iterator<String> it = names.iterator();
    while (it.hasNext())
    {
        final String currentName = it.next();
        if (currentName.length() < 4)
        {
            it.remove();
        }
    }

    return names;
}

private static List<String> createNamesList()
{
    final List<String> names = new ArrayList<>();
    names.add("Michael");
    names.add("Tim");
    names.add("Flo");
    names.add("Clemens");
    return names;
}
```

Tipp: Nutzen Sie das in Aufgabe 2 erstellte `Predicate<String> isShortWord` und die Methode `removeIf(Predicate<T>)` aus dem Interface `Collection<E>`.

Aufgabe 5b: Überlegen Sie, wie man die gezeigte Methode für verschiedene Auswahlkriterien allgemeingültiger gestalten kann? Wie würde man es herkömmlich (ohne `Predicate<T>`) machen (müssen)?

✍ Aufgabe 6 ✍

Modifizieren Sie eine Liste mit Namen, indem Sie dort all diejenigen Einträge verändern, die mit dem Buchstaben "`M`" beginnen: Aus dem Eintrag "`Michael`" soll dann "`>>MICHAEL<<`" werden. Als Ausgangsbasis dient folgender Sourcecode:

```
private static List<String> replaceAll_External_Iteration()
{
    final List<String> names = createNamesList();

    final ListIterator<String> it = names.listIterator();
    while (it.hasNext())
    {
        final String currentName = it.next();
        if (currentName.startsWith("M"))
        {
            // set()-Methode aus ListIterator
            it.set(">>" + currentName.toUpperCase() + "<<");
        }
    }

    return names;
}

private static List<String> createNamesList()
{
    final List<String> names = new ArrayList<>();
    names.add("Michael");
    names.add("Tim");
    names.add("Flo");
    names.add("Merten");
    return names;
}
```

Tipp: Nutzen Sie eine Implementierung eines `UnaryOperator<T>` und die Methode `replaceAll(UnaryOperator<T>)` aus dem Interface `Collection<E>`.

3.9 Übungen zu Streams und Filter-Map-Reduce

Lernziel: In diesem Übungsteil wollen wir die Arbeit mit den in JDK 8 neu eingeführten Streams kennenlernen und dabei insbesondere auf das Filter-Map-Reduce-Framework eingehen.

✍ Aufgabe 1 ✍

Aufgabe 1a: Welche Arten von Operationen gibt es für Streams? Was sind typische Vertreter?

Aufgabe 1b: *Create:* Wie erzeugt man einen `Stream<String>`?

```
final String[] namesArray = { "Tim", "Tom", "Andy", "Mike", "Merten" };
final List<String> names = Arrays.asList(namesArray);

final Stream<String> streamFromArray = // ... TODO ...
final Stream<String> streamFromList = // ... TODO ...
final Stream<String> streamFromValues = // ... TODO ...
```

Tipp: Nutzen Sie `Arrays.stream()`, `List.stream()`, `Stream.of()`.

Aufgabe 1c: *Intermediate:* Filtern Sie eine Liste von Namen, z. B. alle beginnend mit "`T`" und führen Sie ein Mapping aus, z. B. auf `UPPERCASE`.

```
final Stream<String> filtered = // ... TODO ...
final Stream<String> mapped = // ... TODO ...
```

Tipp: Nutzen Sie `Stream.filter()` bzw. `Stream.map()`.

Aufgabe 1d: *Terminal:* Geben Sie den Inhalt des Streams auf der Konsole aus.

Tipp: Nutzen Sie `Stream.forEach()`.

✍ Aufgabe 2 ✍

Wandeln Sie den Inhalt eines Streams in ein Array bzw. in eine Liste um:

```
final Object[] contentsAsArray = streamFromArray. // ... TODO ...
final List<String> contentsAsList = streamFromValues. // ... TODO ...
```

Tipp: Verwenden Sie die Vorarbeiten aus Aufgabe 1 sowie die Methoden `toArray()` und `collect(Collectors.toList())`.

✍ Aufgabe 3 ✍

Wenn eine Ausgabe kommaseparierter Werte gewünscht ist, so kann man folgenden Lambda nutzen. Es wird jedoch abschließend ein Komma zu viel ausgegeben.

```
mapped.forEach(str -> System.out.print(str + ", "));
```

Das kann man aufwendig durch eine Spezialbehandlung vermeiden. Dabei stellt sich allerdings die Frage, wie man das letzte Zeichen bei einem parallelen Stream erkennt. Deshalb bietet es sich an, vordefinierte Methoden einzusetzen.

```
final String joined = streamFromArray. // ... TODO ...
System.out.println(joined);
```

Die erwartete Ausgabe ist: `Tim, Tom, Andy, Mike, Merten`

Tipp: Nutzen Sie `collect()` und `Collectors.joining(", ")` zur Ausgabe.

✍ Aufgabe 4 ✍

Aufgabe 4a: Gruppieren Sie den Inhalt nach dem Anfangsbuchstaben, sodass wir folgendes Ergebnis erhalten: `A=[Andy], T=[Tim, Tom], M=[Mike, Merten]`

```
final Stream<String> inputs = Stream.of("Tim", "Tom", "Andy", "Mike", "Merten");
final Map<Character, List<String>> grouped = inputs. // ... TODO ...
System.out.println(grouped);
```

Tipp: Nutzen Sie `collect()` und die `Collectors.groupingBy()`.

Aufgabe 4b: Teilen Sie den Inhalt gemäß einer vorgegebenen Länge, sodass wir folgende Ausgabe erhalten: `false=[Andy, Mike, Merten], true=[Tim, Tom]`

```
final Stream<String> inputs = Stream.of("Tim", "Tom", "Andy", "Mike", "Merten");
final Map<Boolean, List<String>> partioned = inputs. // ... TODO ...
System.out.println(partioned);
```

Tipp: Nutzen Sie `collect()` und die `Collectors.partioningBy()`.

Aufgabe 4c: Gruppieren Sie den Inhalt nach Anfangsbuchstaben und summieren Sie die Anzahl der Vorkommen, sodass wir diese Ausgabe erhalten: `A=1, T=2, M=2`

```
final Stream<String> inputs = Stream.of("Tim", "Tom", "Andy", "Mike", "Merten");
final Map<Character, Long> groupedAndCounted = inputs. // ... TODO ...
System.out.println(groupedAndCounted);
```

Tipp: Nutzen Sie `collect()` sowie die Methoden `Collectors.groupingBy()` und `Collectors.counting()`.

Aufgabe 5

Gegeben seien die Zahlen von 1 bis 10 als Eingabe. Berechnen Sie das Minimum, Maximum, die Summe und den Durchschnitt dieser Zahlenfolge.

```
final IntStream ints = IntStream.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
```

Tipp 1: Verwenden Sie die Stream-Methoden `min()`, `max()`, `sum()` usw.

Tipp 2: Man sieht recht schnell, dass man immer wieder neue Streams erzeugen muss, um die Informationen zu ermitteln. Im JDK findet sich mit der Klasse `IntSummaryStatistics` eine interessante Klasse. Versuchen Sie es mal mit folgender Programmzeile:

```
final IntSummaryStatistics stats = ints.summaryStatistics();
```

Aufgabe 6

Finden Sie eine grafische Notation für die Stream-Operationen `filter()`, `map()` und `partitioningBy()`. Eine Idee liefert Abbildung 3-4, die zwei Varianten von Gruppierungen (`groupingBy()`) zeigt, einmal nach Form und einmal nach Füllung.

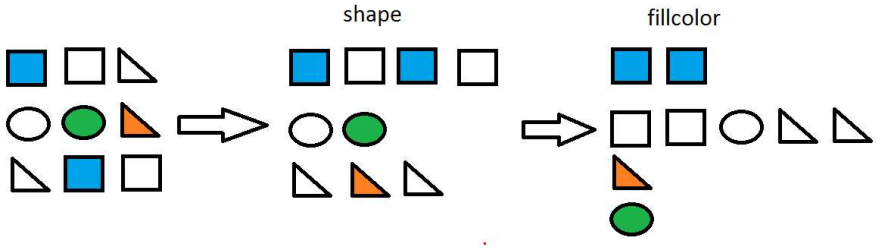

Abbildung 3-4 *Grafische Notation für die Methode* `groupingBy()`