6.13 Übungen zu Diverses

Lernziel: Java 8 enthält eine Vielzahl an nützlichen Funktionalitäten unter anderem im Interface `Comparator<T>` und in der neuen Klasse `Optional<T>`. Zudem gibt es nun parallele Operationen auf Arrays. Diese Neuerungen sollen hier mithilfe von Übungsaufgaben erforscht werden.

✍ Aufgabe 1 ✍

Aufgabe 1a: Sortieren Sie eine Menge von Strings nach deren Länge und nutzen Sie dabei die mit JDK 8 neu eingeführten Erweiterungen im Interface `Comparator<T>`. Als Ausgangsbasis schauen wir uns eine Realisierung mit JDK 7 und inneren Klassen an:

```
public static void main(final String[] args)
{
    final List<String> names = createNamesList();
    final Comparator<String> byLength = new Comparator<String>()
    {
         @Override
         public int compare(final String str1, final String str2)
         {
            return Integer.compare(str1.length(), str2.length());
         }
    };

    Collections.sort(names, byLength);
    System.out.println(names);
}

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

Das obige Programm gibt Folgendes aus:

```
[Tim, Flo, Jörg, Andy, Michael, Clemens]
```

Diese Ausgabe soll mit JDK-8-Sprachmitteln implementiert werden.

Tipp: Nutzen Sie die Methode `Comparator<T>.comparing()`.

Aufgabe 1b: In der Ausgabe sind die Namen nach Länge sortiert, aber innerhalb der Länge nicht alphabetisch. Entdecken Sie weitere Möglichkeiten aus dem Interface `Comparator<T>`, um die Sortierung zu korrigieren. Das gewünschte Resultat ist dann:

```
[Flo, Tim, Andy, Jörg, Clemens, Michael]
```

Tipp: Die Methoden `thenComparing()` und `naturalOrder()` aus dem Interface `Comparator<T>` sind dabei hilfreich.

Aufgabe 1c: Modifizieren Sie danach die Sortierung so, dass längste Wörter zuerst erscheinen und zudem innerhalb gleicher Länge eine alphabetische Sortierung herrscht:

```
[Clemens, Michael, Andy, Jörg, Flo, Tim]
```

Tipp: Die Methoden `thenComparing()`, `naturalOrder()` und `reversed()` aus `Comparator<T>` sind dabei hilfreich.

✍ Aufgabe 2 ✍

Gestalten Sie das folgende Programmfragment so um, dass die Methode `findPersonByName(String)` nun ein `Optional<T>` zurückgibt. Auch Teile aus der Methode `processPerson(Person)` können modifiziert und herausgezogen werden.

```
public static void main(final String[] args)
{
    final Person person1 = findPersonByName("unknown");
    processPerson(person1);

    final Person person2 = findPersonByName("Micha");
    processPerson(person2);
}

private static void processPerson(final Person person)
{
    if (person != null)
    {
        System.out.println(person);
    }
    else
    {
        System.out.println("No person found!");
    }
}
```

✍ Aufgabe 3 ✍

Aufgabe 3a: Befüllen Sie ein `int[200]` fortlaufend mit den Werten `0 ...99`. Dadurch ergibt sich diese Wertebelegung:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 ... 97 98 99 0 1 2 3 4 ... 97 98 99
```

Eine mögliche Realisierung mithilfe von JDK 7 ist im folgenden Listing gezeigt und soll nun mit JDK-8-Mitteln vereinfacht werden. Zur Kontrolle und zur Prüfung der korrekten Erzeugung der Wertefolge und der Übergangsstellen werden exemplarisch einige Werte mithilfe der unten gezeigten Methode `printRange(int[], int, int)` ausgegeben.

```
public static void main(final String[] args)
{
    final int[] original = new int[200];

    specialFill(original);

    printRange(original, 0, 20);
    printRange(original, 100, 120);
    printRange(original, 180, 200);
}

private static void specialFill(final int[] values)
{
    for (int j = 0; j < values.length / 100; j++)
    {
        for (int i = 0; i < 100; i++)
        {
            values[j * 100 + i] = i;
        }
    }
}

private static void printRange(final int[] values,
                               final int startIdx, final int endIdx)
{
    for (int i = startIdx; i < endIdx; i++)
    {
        System.out.print(values[i] + " ");
    }
    System.out.println();
}
```

Tipp: Nutzen Sie `Arrays.parallelSet()` und einen `IntUnaryOperator`. Setzen Sie die Lambdas `i -> i` und `i -> i % 100` ein.

Aufgabe 3b: Modifizieren Sie den Inhalt des zuvor befüllten Arrays: Multiplizieren Sie alle geraden Zahlen mit -1 und alle ungeraden mit dem Wert 2. Mit JDK 7 lässt sich das wie im Listing gezeigt realisieren. Suchen Sie eine Alternative mit Java 8.

```
private static void modify(final int[] values)
{
    for (int i = 0; i < values.length; i++)
    {
        if (values[i] % 2 == 0)
        {
            values[i] *= -1;
        }
        else
        {
            values[i] *= 2;
        }
    }
}
```

Damit ergibt sich etwa folgende Belegung des `int`-Arrays (gekürzt):

```
0 2 -2 6 -4 10 -6 14 -8 18 -10 22 -12 26 -14 30 -16 34 -18 38 ...
```

✍ Aufgabe 4 ✍

Berechnen Sie die Summe für ein `int`-Array für 100.000 Elemente gefüllt als aufsteigende Folge 1, 2, 3, 4, 5, 6, ... Die im folgenden Listing vorgegebene Implementierung basiert auf JDK 7 und soll nun mit neuen Sprachfeatures aus JDK 8 vereinfacht werden.

```
public static void main(final String[] args)
{
    final int[] original = new int[100_000];
    for (int i=0; i < original.length ; i++)
    {
        original[i] = i + 1;
    }

    System.out.println("sum: " + sum(original));
}

private static long sum(final int[] values)
{
    int sum = 0;
    for (int i=0; i < values.length; i++)
    {
        sum = sum + values[i];
    }
    return sum;
}
```

Tipp: Füllen Sie das Array mit `parallelSetAll()` und einem `IntUnaryOperator`. Nutzen Sie für die Berechnung `parallelPrefix()` und einen weiteren `IntBinaryOperator`. Dieser kombiniert zwei `int`-Werte zu einem neuen. Versuchen Sie es mit dem Lambda `(val1, val2) -> val1 + val2` als `IntBinaryOperator`.

✍ Aufgabe 5 ✍

Aufgabe 5a: Sortieren Sie ein Array mit 1.000.000 Elementen durch Aufruf von `Arrays.parallelSort()`. Vergleichen Sie die Performance mit der von `Arrays.sort()`.

Aufgabe 5b: Führen Sie Vergleichsmessungen für eine sequentielle Summation über ein Array und die Summation mit `Arrays.parallelSort()` für 1.000.000 Elemente durch. Bedenken Sie, dass eine Parallelverarbeitung zwar die Verarbeitung beschleunigen kann, sich aber möglicherweise ein anderer Algorithmus viel entscheidender auf die Performance auswirkt. Im Beispiel könnte die Summation etwa durch die Gaußsche Summenformel ($\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$) in konstanter Zeit unabhängig von der Anzahl an Elementen berechnet werden. Welche Probleme kann die Summierung noch verursachen? Was ist mit Überläufen?!

✍ Aufgabe 6 ✍

Aufgabe 6a: Analysieren Sie den Inhalt einer ZIP-Datei.

Tipp: Nutzen Sie die Klasse `java.util.zip.ZipFile` und deren `stream()`-Methode.

Aufgabe 6b: Bereiten Sie eine nach Dateigröße sortierte Liste auf, wobei eine Filterbedingung mitgegeben werden soll, etwa alle im ZIP-Archiv enthaltenen PDFs.

Tipp: Nutzen Sie die Möglichkeiten des Stream-APIs und eines `Predicate<T>` sowie die Methode `Comparator.comparing()`.