6 Übungen zu den API-Neuerungen in JDK 11 bis 17

Die folgenden Aufgaben sollen helfen, die Neuerungen in den APIs aus den JDKs 11 bis 17 anhand von Übungen in ihrer Handhabung zu vertiefen.

6.1 Aufgaben

✍ Aufgabe 1 – HTTP/2-API ✍

Gegeben sei eine HTTP-Kommunikation, die mit den Möglichkeiten von JDK 8 auf die Webseite von Oracle zugreift und diese textuell aufbereitet. Wandeln Sie den Sourcecode so um, dass das HTTP/2-API aus JDK 11 zum Einsatz kommt.

```
private static void readOraclePageJdk8() throws MalformedURLException,
                                                IOException
{
    var url = new URL("https://www.oracle.com/index.html");

    final String contents = readContent(url.openStream());
    System.out.println(contents);
}

public static String readContent(final InputStream is) throws IOException
{
    try (final InputStreamReader isr = new InputStreamReader(is);
         final BufferedReader br = new BufferedReader(isr))
    {
        final StringBuilder content = new StringBuilder();

        String line;
        while ((line = br.readLine()) != null)
        {
            content.append(line + "\n");
        }

        return content.toString();
    }
}
```

Bonus Experimentieren Sie mit einem asynchronen Aufruf.

Tipp Nutzen Sie die Klassen `HttpRequest` und `HttpResponse` und erstellen Sie eine Methode `printResponseInfo(HttpResponse)`, die den Body sowie den HTTP-Statuscode ausliest und auf die Konsole schreibt.

```
public static void main(final String[] args) throws Exception
{
    var uri = new URI("https://www.oracle.com/index.html");

    var request = // ...
    var asString = // ...
    var httpClient = HttpClient.newHttpClient();
    var response = // ...

    printResponseInfo(response);
}
```

✍ Aufgabe 2 – Strings ✍

Aufgabe 2a Realisieren Sie eine Ausgabe, die sieben Zahlen untereinander ausgibt, jeweils so oft wiederholt wie der Zahlenwert der Ziffer, also wie folgt:

```
22
4444
7777777
333
999999999
1
55555
```

Nutzen Sie folgenden Stream als Eingabe:

```
Stream.of(2, 4, 7, 3, 9, 1, 5)
```

Aufgabe 2b Modifizieren Sie die Ausgabe so, dass die Zahlen rechtsbündig mit maximal 10 Zeichen ausgegeben werden:

```
'        22'
'      4444'
'   7777777'
'       333'
'         1'
' 999999999'
'     55555'
```

Implementieren und nutzen Sie dazu folgende Hilfsmethode:

```
private static String formatRightAligned(final int num,
                                         final int desiredLength)
{
    // TODO
}
```

Bonus Als Bonus ergänzen Sie noch eine aufsteigende Sortierung und wenden wiederum die formatierte Ausgabe an:

```
'         1'
'        22'
'       333'
'      4444'
'     55555'
'   7777777'
' 999999999'
```

Aufgabe 2c Ändern Sie das Ganze so ab, dass nun statt Leerzeichen führende Nullen ausgegeben werden:

```
'0000004444'
'0007777777'
'0999999999'
```

✍ Aufgabe 3 – Strings (`indent()`) ✍

Die Verarbeitung von Strings wurde in Java 12 um zwei Methoden erweitert. Lernen Sie die Methode `indent()` genauer kennen.

Aufgabe 3a Rücken Sie die folgende Eingabe um sieben Zeichen ein, geben Sie diese aus und entfernen Sie wieder drei Zeichen von der Einrückung – beobachten Sie insbesondere auch die Länge des eingerückten Strings.

```
String originalString = "first_line\nsecond_line\nlast_line";
System.out.println(originalString.length());
```

Aufgabe 3b Was passiert, wenn man einen linksbündigen Text mit negativen Werten für den Indent versieht, wenn also konkret für die nachfolgende Eingabe ein Indent von -10 zum Einsatz kommt?

```
String multipleIndentedString = "LINE 1\n    LINE 2" +
                                "\n            LINE 3\n     LINE 4";
```

✍ Aufgabe 4 – Strings (`transform()`) ✍

Lernen Sie hier die Methode `transform()` genauer kennen. Gegeben sei dazu folgende kommaseparierte Eingabe:

```
var csvText = "HELLO,WORKSHOP,PARTICIPANTS,!,LET'S,HAVE,FUN";
```

Aufgabe 4a Wandeln Sie diesen Text vollständig in Kleinbuchstaben um und ersetzen Sie die Kommas durch Leerzeichen. Natürlich könnte man das durch einen Aufruf von `csvText.toLowerCase().replace(",", " ")` umsetzen, jedoch soll hier die Methode `transform()` zum Einsatz kommen.

Aufgabe 4b Ersetzen Sie den Text »HELLO« mit dem Schweizer Gruß »GRÜEZI« und spalten Sie das Ganze dann in Einzelbestandteile auf, sodass folgendes Array als Ergebnis entsteht:

```
[GRÜEZI, WORKSHOP, PARTICIPANTS, !, LET'S, HAVE, FUN]
```

✍ Aufgabe 5 – Strings und Dateien ✍

Vor Java 11 war es etwas mühsam, Texte direkt in eine Datei zu schreiben bzw. daraus zu lesen. Dazu gibt es in JDK 11 die Methoden `writeString()` und `readString()` aus der Klasse `Files`. Schreiben Sie mit deren Hilfe folgende Zeilen in eine Datei:

```
1: One
2: Two
3: Three
```

Lesen Sie diese Zeilen wieder ein und bereiten Sie daraus eine `List<String>` auf.

✍ Aufgabe 6 – Die Klasse `CompactNumberFormat` ✍

Schreiben Sie ein Programm, um mithilfe der Klasse `CompactNumberFormat` die Kurzversionen für die Zahlen 1.000, 1.000.000 und 1.000.000.000 abhängig von Locale und Style auszugeben und zu parsen. Verwenden Sie die Locale `GERMANY` für `SHORT` und `ITALY` für `LONG`.

Die nachfolgenden Werte können Sie zum Parsing verwenden:

```
// Parsing, Achtung hier muss man ein non-breakable space (\u00a0) statt des
// normalen (\u0020) einsetzen
List.of("13 KILO", "1 Mio.", "1 Mrd.")
List.of("mille", "1 milione")
```

✍ Aufgabe 7 – Teeing-Kollektor Basics ✍

Nutzen Sie den Teeing-Kollektor, um in einem Durchlauf sowohl das Minimum als auch das Maximum bezüglich `String::compareTo` zu finden. Beginnen Sie mit folgenden Zeilen:

```
Stream<String> values = Stream.of("CCC", "BB", "A", "DDDD");
List<Optional<String>> optMinMax = values.collect(teeing(...
```

Tipp Um eine lesbare Hintereinanderschaltung der Kollektoren zu ermöglichen, importieren wir diese statisch:

```
import static java.util.stream.Collectors.*;
```

✍ Aufgabe 8 – Teeing-Kollektor Advanced ✍

Nutzen Sie einen Teeing-Kollektor, um aus einem Stream in einem Durchlauf sowohl die Namen aller europäischen Städte als auch die Anzahl der Städte in Asien zu ermitteln:

```
Stream<City> cities = Stream.of(new City("Zurich", "Europe"),
                                new City("Bremen", "Europe"),
                                new City("Kiel", "Europe"),
                                new City("San Francisco", "America"),
                                new City("Lima", "America"),
                                new City("Aachen", "Europe"),
                                new City("Hong Kong", "Asia"),
                                new City("Tokyo", "Asia"));

Predicate<City> isInEurope = city -> city.locatedIn("Europe");
Predicate<City> isInAsia = city -> city.locatedIn("Asia");

var result = cities.collect(teeing(...
```

Als Ergebnis wird folgende Ausgabe erwartet – natürlich dürfen Sie ruhig kreativ in der Aufbereitung werden, aber die gezeigten Ergebnisse sollten geliefert werden:

```
cities in europe=[Zurich, Bremen, Kiel, Aachen]
#asian cities=2
```

Bonus Gegeben sei noch die folgende innere Klasse `City`, die in einen Record überführt werden soll:

```
static class City
{
    private final String name;
    private final String region;

    public City(String name, String region)
    {
        this.name = name;
        this.region = region;
    }

    public String getName()
    {
        return name;
    }

    public String getRegion()
    {
        return region;
    }

    public boolean locatedIn(String region)
    {
        return this.region.equalsIgnoreCase(region);
    }
}
```