TECHNISCHE HOCHSCHULE LÜBECK

Programmieren II

Informatik (B. Sc.)
2. Semester

Unit 06

Thinking in Filter → Map → Reduce

Prof. Dr. Nane Kratzke

Units

Rekursion

01 Einführung in die rekursive Programmierung
02 Sequenzbasierte Rekursionen
03 Rekursive Datenstrukturen

Funktionale Programmierung

04 Einführung in die funktionale Programmierung
05 Funktionale Programmierung mit Streams
06 Thinking in Filter → Map → Reduce

Generische Datentypen <T>

07 Einführung in Gen. Datentypen (Type Erasure)
08 Bounded Types

Graphical User Interfaces

09 Einführung in Swing (Typvertreter)
10 Model - View - Controller (MVC)
11 MVC an einer Beispielanwendung

Thinking in filter, map, reduce

Ein Stream ist ...

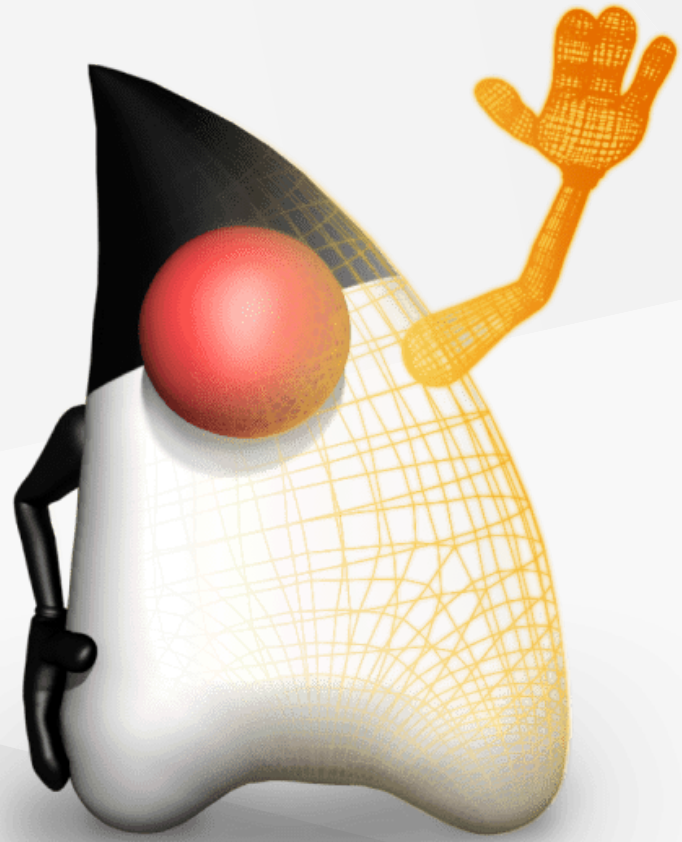
- ... eine Sequenz von Operationen, die in einer Pipeline zur Verarbeitung von Daten effizient angewendet werden können.
- *Aber wie ordnet man diese Operationen zielgerichtet an?*

Übliche Phasen von Streams

- **Vorverarbeitung**: Filtern relevanter Daten (`filter()`)
- **Verarbeitung** relevanter Daten (`map()`)
- **Nachbearbeitung**: Aufbereitung des Ergebnisses (`reduce()`)

Übungsaufgaben

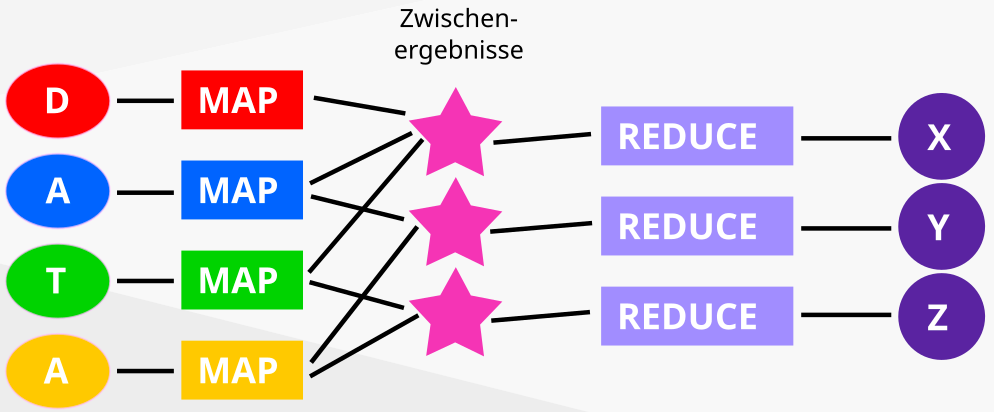
- Primzahlen bestimmen, gruppieren und zählen
- Textdateien analysieren (Faust vs Mephisto)

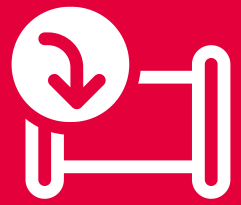


Ursprung von Map-Reduce

Das Map-Reduce-Konzept ist ein bekanntes Paradigma für die Verarbeitung von großen Datenmengen in skalierbaren und verteilten Systemen (Google Patent, 2004). Es wurde ursprünglich in der Programmiersprache Lisp entwickelt und ist besonders aus dem Bereich der Datenverarbeitung in großen Clustern populär geworden, wie beispielsweise bei Apache Spark und Hadoop.

- Map-Phase: In der ersten Phase (Map) wird die Eingabedatenmenge in eine Menge von Schlüssel-Wert-Paaren transformiert. Jeder Mapper (ein Rechenprozess) bearbeitet einen Teil der Dateneingabe.
- Shuffle-Phase: Nach der Map-Phase werden die von den Mappern erzeugten Schlüssel-Wert-Paare nach Schlüssel sortiert und an den entsprechenden Reducer gesendet.
- Reduce-Phase: In der Reduce-Phase übernehmen die Reducer die vom Shuffle-Prozess erhaltenen Schlüssel-Wert-Paare und aggregieren die Daten, um die endgültigen Ergebnisse zu produzieren.





Vorverarbeitung (Filter-Phase)

Eine Sequenz von Operationen zur Selektion relevanter Daten

Intermediäre Methoden zur Vorverarbeitung in einem Stream

Einige der häufigsten **intermediären Methoden**, die überwiegend für die Vorverarbeitung von Streams verwendet werden:

- `filter(Predicate)` Filtert die Elemente des Streams basierend auf einem gegebenen Prädikat. Nur die Elemente, für die das Prädikat $true$ ergibt, werden im neuen Stream beibehalten.
- `limit(long)` Begrenzt die Anzahl der Elemente im Stream auf eine maximale Größe.
- `skip(long)` Überspringt die ersten n Elemente und gibt den Rest des Streams zurück.
- `takeWhile(Predicate)` Gibt Elemente zurück, solange das Prädikat $true$ ergibt, und stoppt, wenn es $false$ ergibt.
- `dropWhile(Predicate)` Überspringt Elemente des Streams, solange das Prädikat $true$ ergibt, und gibt den Rest zurück, sobald ein $false$-Ergebnis auftritt.
- `distinct()` Entfernt doppelte Werte aus einem Stream.

Anmerkung: *Diese Methoden können natürlich überall in einem Stream angewendet werden, aufgrund des filternden Charakters werden sie aber häufig in der "**vorderen**" Phase der Verarbeitung genutzt.*

Beispiel: `filter()`

Hier ist ein einfaches Beispiel, das zeigt, wie man die `filter`-Methode in einem Stream verwendet. In diesem Beispiel erstellen wir eine Liste von Ganzzahlen, filtern die geraden Zahlen heraus, multiplizieren sie mit 2 und geben die Ergebnisse auf der Konsole aus:

```
List<Integer> zahlen = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10); // Datenquelle des Streams

zahlen.stream()                          // Erzeugt einen Stream aus der Liste
  .filter(n -> n % 2 == 0)               // Intermediäre Operation: filtert gerade Zahlen
  .forEach(System.out::println);         // Terminale Operation: gibt das Ergebnis aus
```

Ausgabe:

```
2
4
6
8
10
```

Beispiel: `dropWhile()` und `takeWhile()`

- Mit `takeWhile()` nehmen wir alle Zahlen, die kleiner als 5 sind.
- Mit `dropWhile()` lassen wir alle Zahlen fallen, die kleiner als 5 sind und nehmen dann die verbleibenden.

`takeWhile()`

```
List<Integer> zahlen = List.of(
  1, 2, 3, 4, 5, 6, 7, 8, 9
);

zahlen.stream()
  .takeWhile(n -> n < 5) // <= takeWhile
  .forEach(System.out::println);
```

```
1
2
3
4
```

`dropWhile()`

```
List<Integer> zahlen = List.of(
  1, 2, 3, 4, 5, 6, 7, 8, 9
);

zahlen.stream()
  .dropWhile(n -> n < 5) // <= dropWhile
  .forEach(System.out::println);
```

```
5
6
7
8
9
```

Beispiel: `limit()` und `skip()`

- Mit `limit()` können wir einen Stream auf n Elemente beschränken. Ist n länger als der Stream, werden alle Elemente des Streams (aber eben weniger als n Elemente) zurück gegeben.
- Mit `skip()` können wir die ersten n Elemente eines Streams überspringen. Achtung: Unendliche Streams bleiben damit unendlich.

```java
import java.util.stream.Stream;

Stream.iterate(1, n -> n + 1)         // Erzeugt einen unendlichen Stream von Zahlen
      .skip(5)                        // Überspringe die ersten 5 Zahlen
      .limit(5)                       // Begrenze auf den verbleibenden Stream auf 5 Zahlen
      .forEach(System.out::println); // Ergebnisse ausgeben
```

Ausgabe:

```
6
7
8
9
10
```

Beispiel: `distinct()`

Die Methode `distinct()` in Java Streams entfernt doppelte Elemente aus einem Stream und gibt einen neuen Stream zurück, der nur einzigartige Elemente enthält.

```
List<String> namen = List.of("Anna", "Berta", "Anna", "Clara", "Berta", "David");

List<String> einzigartigeNamen = namen.stream()
    .distinct() // <= Entfernt doppelte Namen
    .collect(Collectors.toList());

System.out.println(einzigartigeNamen);
```

Ausgabe:

```
[Anna, Berta, Clara, David]
```

Verarbeitung (Map-Phase)

Anwenden einer Sequenz von Operationen auf relevanten Daten

Intermediäre Methoden zur Verarbeitung in einem Stream

Einige der häufigsten intermediären Methoden, die überwiegend im Hauptverarbeitungsteil von Streams verwendet werden:

- `map(Function)`: Wendet eine Funktion auf jedes Element des Streams an. Der Rückgabewert dieser Funktion wird in den neuen Stream eingefügt.
- `flatMap(Function)`: Ähnlich wie `map`, aber anstelle eines einzelnen Werttyps gibt die Funktion einen Stream zurück. Die Ergebnisse von allen Streams werden in einem einzelnen neuen Stream zusammengeführt.
- `peek(Consumer)`: Ermöglicht das Einfügen von Debugging- oder Logging-Ausgaben, indem eine Aktion auf jedes Element des Streams ausgeführt wird, ohne die Elemente zu ändern.
- `sorted()`, `sorted(Comperator)`: Sortiert die Elemente des Streams in natürlicher Reihenfolge oder auf einem angegebenen Comparator. Achtung: Stream sollte endlich sein!

Anmerkung: *Diese Methoden können natürlich überall in einem Stream angewendet werden, aufgrund ihres verarbeitenden Charakters werden diese aber häufig in der "**mittleren**" Phase der Verarbeitung genutzt.*

Beispiel: `map()`

Die `map`-Funktion wendet eine Transformation auf jedes Element in einem Stream an. Es wird oft verwendet, um die Daten in einem Stream in ein anderes Format zu bringen, Berechnungen vorzunehmen oder um bestimmte Werte aus den Objekten zu extrahieren.

Die `map`-Methode nimmt eine Funktion als Argument, die angibt, wie jedes Element des ursprünglichen Streams verarbeitet werden soll. Der Rückgabewert dieser Funktion wird dann in einen neuen Stream überführt.

```
List<Integer> zahlen = List.of(1, 2, 3, 4, 5); // Datenquelle

zahlen.stream()
    .map(n -> n * 2)                  // <= Anwendung von Map (verdoppelt jedes Element des Streams)
    .forEach(System.out::println); // Gibt die Ergebnisse aus
```

```
2
4
6
8
10
```

Beispiel: `flatMap()`

Die `flatMap()`-Methode ist eine Operation, die verwendet wird, um einen Stream von Streams in einen einzelnen Stream zu transformieren. Sie ist besonders nützlich, wenn geschachtelte Datenstrukturen verarbeitet werden oder wenn mehrere Werte pro Eingabewert erzeugt werden.

`flatMap()` wendet eine `Function` auf jedes Element eines Streams an, wobei die `Function` einen Stream zurück gibt. Alle Streams, die aus dieser Funktion erzeugt werden, werden dann in einen einzigen Stream "flach" zusammengeführt.

Folgendes Beispiel zeigt, wie eine Liste von Texten (`texts`) verarbeitet werden kann, so dass jeder Text in seine Einzelwörter zerlegt wird und aus dem Streams von Einzelworten ein einziger "flacher" Stream von Worten wird.

```
List<String> texts = List.of("Hallo Welt", "Java ist toll", "Streams machen Spaß");

String words = texts.stream()
  .flatMap(t -> Stream.of(t.split(" +"))) // Zerlege Text in Wörter (Return ist ein STREAM!)
  .collect(Collectors.joining(", "));
System.out.println(words);
```

```
Hallo, Welt, Java, ist, toll, Streams, machen, Spaß
```

Beispiel: `peek()`

Die `peek()`-Methode ermöglicht es, eine Aktion für jedes Element eines Streams auszuführen, ohne dabei die Elemente des Streams selbst zu verändern; sie wird häufig für Debugging- oder Logging-Zwecke verwendet.

Um die Wirkungsweise der `flatMap()` Methode beispielsweise besser zu verstehen, könnten wir vor und hinter der `flatMap()` Methode jeweils ein `peek()` mit einer Debugging Print-Anweisung einschieben, um Einblick in die jeweilige Verarbeitungsstufe des Streams zu erhalten.

```
List<String> texts = List.of(
  "Hallo Welt",
  "Java ist toll",
  "Streams machen Spaß"
);

String words = texts.stream()
  .map(t -> List.of(t.split(" +")))
  .peek(e -> System.out.println("Debug 1: " + e))
  .flatMap(List::stream)
  .peek(e -> System.out.println("Debug 2: " + e))
  .collect(Collectors.joining(", "));
System.out.println(words);
```

```
Debug 1: [Hallo, Welt]
Debug 2: Hallo
Debug 2: Welt
Debug 1: [Java, ist, toll]
Debug 2: Java
Debug 2: ist
Debug 2: toll
Debug 1: [Streams, machen, Spaß]
Debug 2: Streams
Debug 2: machen
Debug 2: Spaß
Hallo, Welt, Java, ist, toll, Streams, machen,
Spaß
```

Beispiel: `sorted()`

Nehmen wir an, wir wollten nicht nur alle Worte in mehreren Texten bestimmen, sondern diese auch sortiert ausgeben, so können wir dies mit `sorted()` wie folgt machen:

Natürliche Ordnung:

```
List<String> texts = List.of(
  "Hallo Welt",
  "Java ist toll",
  "Streams machen Spaß"
);

String words = texts.stream()
  .flatMap(t -> Stream.of(t.split(" +")))
  .sorted()                          // <= Sortierung
  .collect(Collectors.joining(", "));
System.out.println(words);
```

Ausgabe alphabetisch sortiert:

```
Hallo, Java, Spaß, Streams, Welt, ist, machen, toll
```

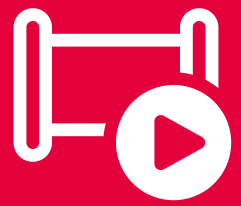
Definierbare Ordnung (`Comparator`):

```
List<String> texts = List.of(
  "Hallo Welt",
  "Java ist toll",
  "Streams machen Spaß"
);

String words = texts.stream()
  .flatMap(t -> Stream.of(t.split(" +")))
  .sorted((w1, w2) -> w1.length() - w2.length())
  .collect(Collectors.joining(", "));
System.out.println(words);
```

Ausgabe nach Länge sortiert:

```
ist, Welt, Java, toll, Spaß, Hallo, machen, Streams
```



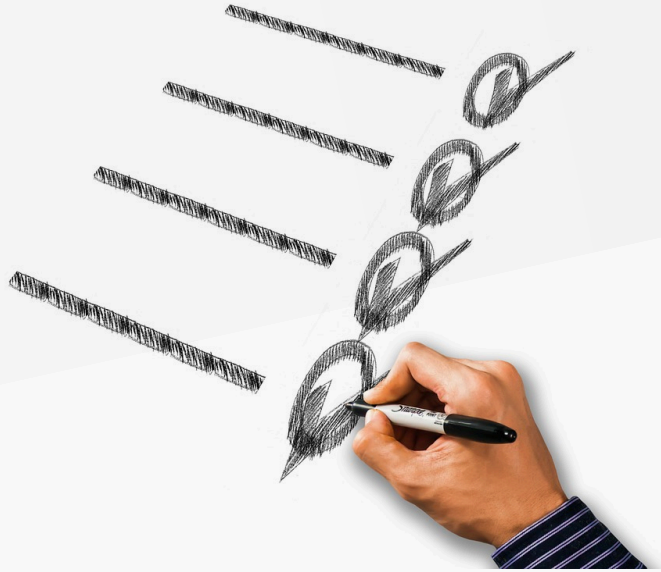
Terminierung des Streams (Reduce-Phase)

Aufbereiten des Ergebnisses

Terminierende Methoden in einem Stream

Einige der häufigsten **terminierenden Methoden**, die zum Bearbeitungsende eines Streams führen:

- `reduce()`: Reduziert die Elemente im Stream mit einer bestimmten Funktion auf einen einzigen Wert.
- `count()`: Gibt die Anzahl der Elemente im Stream zurück. Entspricht `size()` bei Collections, bzw. `length()` bei Strings. Achtung: Gibt allerdings einen `long`-Wert zurück.
- `anyMatch()`, `allMatch()`, `noneMatch()`: Bekannt aus der Prädikatenlogik (∃,∀, ∄). Überprüft, ob mindestens ein/alle/kein Element im Stream ein bestimmtes Kriterium erfüllt.
- `min()` und `max()`: Gibt das minimale/maximale Element des Streams zurück, basierend auf einem Comparator.



Anmerkung: *Diese Methoden können nur als letzte Operation in einem Stream angewendet werden, da sie die Stream-Verarbeitung beenden. Ferner gehören zu dieser Liste auch die Methoden* `collect()` *und* `forEach()`*, die wir allerdings schon genutzt bzw. besprochen haben.*

Beispiel: `reduce()`

Die `reduce()`-Methode wird verwendet, um eine Zusammenführung (Reduzierung) der Elemente eines Streams zu einem einzigen Wert durchzuführen. Diese Methode ist sehr nützlich, um aggregierte Informationen zu erstellen, wie z. B. die Summe oder das Produkt von Zahlen.

Beispiele:

```
List<Integer> values = List.of(1, 2, 3, 4, 5);
int result = values.stream().reduce(0, (a, b) -> a + b);
System.out.println(result); // => 15
```

Veranschaulichung:

```
0  [1, 2, 3, 4, 5] mit (a, b) -> a + b
1  [2, 3, 4, 5]
3  [3, 4, 5]
6  [4, 5]
10 [5]
15 []
```

Die Methode erwartet

- einen Startwert mit dem die Reduktion beginnt,
- und einen `BiOperator` (Akkumulator), der zwei Parameter nimmt: den aktuellen aggregierten Wert b und das nächste Element aus dem Stream a.

Der Abschlusswert des Akkumulators wird dann verwendet, um das nächste Element im Stream zu verarbeiten, und dieser Vorgang wiederholt sich für alle Elemente im Stream.

Beispiel: `reduce()`

Die `reduce()`-Methode kann extrem vielseitig eingesetzt werden.

Beispiele:

Zeichenkettenkonkatenation:

```
List<String> values = List.of("Ein", "doofes", "Beispiel");
String result = values.stream().reduce("", (a, b) -> a + "+" + b);
System.out.println(result);
```

```
Ein+doofes+Beispiel
```

Längstes Wort:

```
String result = values.stream().reduce("", (a, b) -> a.length() > b.length() ? a : b);
System.out.println(result);
```

```
Beispiel
```

Beispiel: `reduce()`

Letztlich sind alle Collectors nichts weiter als häufig benutzte Spezialfälle von `reduce()`.
Hier ein Beispiel für `toList()`.

Nicht generische Implementierung von `Collectors.toList()`:

```
String text = "Dies ist mal wieder no so ein doofes Beispiel";
List<String> list = Stream.of(text.split(" "))
  .map(List::of)
  .reduce(new ArrayList<>(), (a, b) -> {
    a.addAll(b);
    return a;
  });
System.out.println(list);
```

Ausgabe:

```
[Dies, ist, mal, wieder, nur, so, ein, doofes, Beispiel]
```

Anmerkung: *Zum Thema Generics kommen wir noch in den folgenden Units.*

Beispiel: ∀ = `allMatch()`, ∃ = `anyMatch()` und ∄ = `noneMatch()`

Die Methoden `allMatch()`, `anyMatch()` und `noneMatch()` entsprechen den Quantoren (∀ und ∃) der Prädikatenlogik. Sie prüfen ob alle/mindestens ein/kein Element im Stream ein bestimmtes Kriterium erfüllt.

Anwendungsbeispiel für `noneMatch()`: Bestimmen, ob eine Zahl eine Primzahl ist.

```
IntPredicate isPrim = n -> n > 1 && IntStream.range(2, n).noneMatch(i -> n % i == 0);
```

Ferner gegeben, die Liste aller Primzahlen:

```
Supplier<IntStream> prims = () -> IntStream.iterate(2, n -> n + 1).filter(isPrim);
```

Dann stimmen diese Aussagen: *Sie können gerne unlimitiert prüfen, aber dann warten Sie ewig auf die Antwort ;-)*

```
prims.get().limit(1000).noneMatch(i -> i % 2 == 0) == false
// Die Aussage, dass keine Primzahl gerade ist, ist falsch (die 2 ist gerade und eine Primzahl)
prims.get().limit(1000).anyMatch(i -> i % 2 == 0) == true
// Es gibt tatsächlich eine Primzahl, die gerade ist (nämlich die 2)
prims.get().limit(1000).allMatch(i -> i % 2 == 1) == false
// Die Aussage, das alle Primzahlen ungerade sind, ist falsch (wieder ist die 2 das Problem)
```

Beispiel: `count()`

Zum Abschluss wollen wir wissen, wie viele Primzahlen es gibt (und nutzen dafür `count()`).

```
IntPredicate isPrim = n -> n > 1 && IntStream.rangeClosed(2, (int)Math.sqrt(n))
  .noneMatch(i -> n % i == 0);

long prims = IntStream.iterate(0, n -> n + 1).filter(isPrim).count();
```

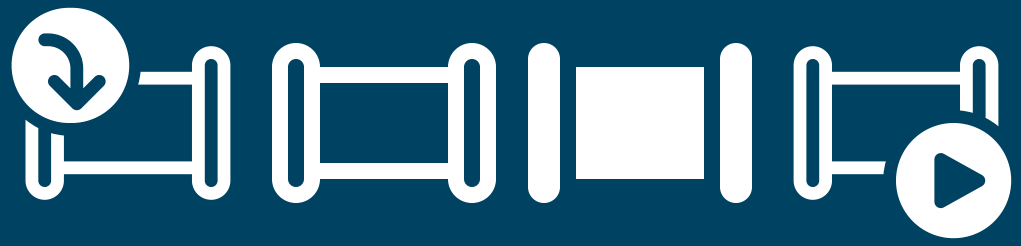
Auf diese Weise würde man keine Antwort bekommen, da man einen unendlich langen Stream zählen würde (und das dauert bekanntlich ewig). Wichtig: `count()` also niemals auf unlimitierte Streams anwenden.

Wenn wir unser Problem etwas einschränken, können wir es auch beantworten:

Wieviele Primzahlen gibt es bis 1000?

```
long prims = IntStream.rangeClosed(0, 1000).filter(isPrim).count();
System.out.println(prims); // => 168
```

Achtung: *Anders als* `size()` *und* `length()` *gibt* `count()` *bei Streams ein* `long`*-Wert zurück und keinen* `int`*!*



Übungsaufgaben

Gruppieren/Zählen von Primzahlen, Ausgabe von Primzahlen, Faust vs Mephisto

1. Aufgabe:

Bestimmen Sie nun die Primzahlen-Verteilung in 1000er Schritten. Wieviele Primzahlen gibt es bis 999, wieviele von 1000 bis 1999, von 2000 bis 2999, usw.

Vorgabe:

```
Map<Integer, Long> primDistrib = ... ;
System.out.println(primDistrib);
```

Ausgabe: *(Reihenfolge kann variieren)*

```
{8000=110, 6000=117, 4000=119, 2000=127, 0=168, 9000=112, 7000=107, 5000=114, 3000=120, 1000=135}
```

Lösung:

>>> YOUR-TURN <<<

2. Aufgabe:

Die Lösung ihrer ersten Aufgabe, sah vielleicht so oder ähnlich aus.

```
IntPredicate isPrim = n -> n > 1 && IntStream.rangeClosed(2, (int)Math.sqrt(n)).noneMatch(i -> n % i == 0);
Map<Integer, Long> primsDistrib = IntStream.rangeClosed(0, 10000)
  .filter(isPrim)
  .boxed().collect(Collectors.groupingBy(v -> v / 1000 * 1000, Collectors.counting()));
System.out.println(primsDistrib);
```

```
{8000=110, 6000=117, 4000=119, 2000=127, 0=168, 9000=112, 7000=107, 5000=114, 3000=120, 1000=135}
```

Die Ausgabe war aber eher so "naja". Erzeugen Sie nun auf Basis des Resultats der vorhergehenden Aufgabe folgende Ausgabe (natürlich nur mit Streams und Lambda-Funktionen).

```
0 - 999: 168
1000 - 1999: 135
2000 - 2999: 127
3000 - ...
```

Frag KIRA: Wie kann ich aus einer Map einen Stream machen?

3. Aufgabe: Faust vs Mephisto

Welcher der beiden Rollen in Goethes Faust (Faust oder Mephisto) hat mehr Sprechanteile? Das hat Ihnen vermutlich niemand im Deutschunterricht erzählt. Höchste Zeit, es rauszufinden! Unter

https://raw.githubusercontent.com/martinth/mobverdb/refs/heads/master/faust.txt

finden Sie eine Text-Datei von Goethes Faust. Nutzen Sie nun ausschließlich Streams und Lambdas, um diese Datei zu laden, vorzuverarbeiten, zu analysieren und das Ergebnis der Häufigkeiten **aller** Rollen in folgender Form nach Häufigkeiten **absteigend sortiert** auszugeben.

```
FAUST: ???
MEPHISTOPHELES: ???
MARGARETE: ???
WAGNER: ???
...
```

Frag KIRA: Wie kann ich in Java eine Text-Datei aus dem Internet laden und als Stream von Zeichenketten zeilenweise verarbeiten?

Frag KIRA: Wie lautet ein Regex für "EIN NAME (möglicher Zusatz):"?

Zusammenfassung

- Map-Reduce
  Das Map-Reduce-Konzept ist ein bekanntes Paradigma für die Verarbeitung von großen Datenmengen und wird in Java Streams genannt.
- Intermediäre Filter-Operationen auf Streams
  Diese Methoden bieten die Möglichkeit, Elemente zu filtern, die Anzahl der Elemente zu beschränken oder bestimmte Bedingungen zu überprüfen, Duplikate zu entfernen, um die Daten für die Hauptverarbeitung aufzubereiten.
- Intermediäre Map-Operationen auf Streams
  Diese Methoden ermöglichen die Anwendung von Funktionen auf jedes Element (map), die Zusammenführung von Streams (flatMap), die Ausführung von Aktionen auf Elemente ohne deren Veränderung (peek) und die Sortierung der Elemente eines Streams.
- Terminierende Reduce-Operationen auf Streams
  Diese Methoden führen die Verarbeitung des Streams zu Ende und verbrauchen dabei dessen Elemente; dazu gehören die Reduktion auf einen Wert, die Ermittlung der Anzahl der Elemente, sowie anyMatch(), allMatch() und noneMatch() zur generierung boolescher Aussagen auf Elementen des Streams.

Kontakt

Prof. Dr. Nane Kratzke

Technische Hochschule Lübeck

Mail: nane.kratzke(at)th-luebeck.de

Code Strong!

Alle Icons stammen von iconify.design • Bilder von Pixabay

Lösungsvorschlag zur 1. Aufgabe

```java
import java.util.*;
import java.util.function.*;
import java.util.stream.*;

public class Main {

  public static void main(String[] args) {
    IntPredicate isPrim = n -> n > 1 && IntStream.range(2, n).noneMatch(i -> n % i == 0);

    Map<Integer, Long> primsDistrib = IntStream.rangeClosed(0, 10000)
      .filter(isPrim)
      .boxed()
      .collect(Collectors.groupingBy(v -> v / 1000 * 1000, Collectors.counting()));

    System.out.println(primsDistrib);
  }
}
```

Lösungsvorschlag zur 2. Aufgabe

Eigentlich nur eine zweite Streamverarbeitung hinter die Lösung von Aufgabe 1 geschaltet.

```
import java.util.*; import java.util.function.*; import java.util.stream.*;

public class Main {

  public static void main(String[] args) {
    IntPredicate isPrim = n -> n > 1 && IntStream.range(2, n).noneMatch(i -> n % i == 0);

    IntStream.rangeClosed(0, 10000)
      .filter(isPrim)
      .boxed()
      .collect(Collectors.groupingBy(v -> v / 1000 * 1000, Collectors.counting()))
      .entrySet() // <= ab hier Fortsetzung von Aufgabe 1
      .stream()
      .sorted((a, b) -> a.getKey() - b.getKey())
      .map(e -> e.getKey() + " - " + (e.getKey() + 999) + ": " + e.getValue())
      .forEach(System.out::println);
  }
}
```

Lösungsvorschlag zur 3. Aufgabe (Faust vs Mephisto)

Da Hausaufgabe, gibt es den Lösungsvorschlag nur, wenn mir mindestens einer in der kommenden Woche (KW 21) seine Lösung vorstellt.