Bildung für nachhaltige Entwicklung – BNE

Name:	
Klasse:	
Datum:	

Schuljahr: 2024/25

Abbildung 1: generiert mit Bing

1 Einleitung

in einfacher Sprache – siehe Seite 4

Liebe Schülerinnen und Schüler,

in der heutigen Welt, in der Technologien einen immer größeren Teil unseres Lebens einnehmen, spielt die Art und Weise, wie wir Software entwickeln, eine entscheidende Rolle – nicht nur für die Funktionalität von Anwendungen, sondern auch für unsere Umwelt und die Zukunft unseres Planeten. Vielleicht habt ihr schon einmal von „gutem“ und „schlechtem“ Code gehört, aber was bedeutet das eigentlich? Und warum sollte es uns interessieren?

„Guter Code“ ist nicht nur der, der funktioniert, sondern auch der, der effizient, wartbar und umweltfreundlich ist. Es geht darum, Programme so zu schreiben, dass sie nicht nur heute, sondern auch in vielen Jahren noch zuverlässig laufen, einfach angepasst werden können und dabei so wenig Ressourcen wie möglich verbrauchen. Wenn ihr daran denkt, wie oft ihr ein Programm oder eine Website benutzt, dann merkt ihr schnell: Jeder Klick und jede Berechnung kostet Energie – und wenn wir nicht aufpassen, verbrauchen ineffiziente Programme viel mehr Energie, als nötig.

Ihr habt vielleicht schon gehört, dass große Rechenzentren enorme Mengen an Strom verbrauchen. Sie sind das Rückgrat des Internets und der digitalen Welt, aber ihr Energieverbrauch trägt auch erheblich zum Klimawandel bei. Und das bringt uns zum Kern dieses Themas: Nachhaltigkeit in der Softwareentwicklung. Guter Code kann dazu beitragen, den Energieverbrauch von Systemen zu senken und so einen positiven Einfluss auf den Klimaschutz zu haben.

Aber es geht nicht nur um die Umwelt. Wenn Code schlecht geschrieben ist, ist er oft schwer zu warten und anzupassen. Das führt dazu, dass Entwickler mehr Zeit und Ressourcen aufwenden müssen, um Fehler zu beheben oder Änderungen vorzunehmen. Das kostet nicht nur Geld, sondern führt auch dazu, dass Projekte länger dauern und mehr Stress verursachen.

Hier kommen auch die **Nachhaltigkeitsziele der Vereinten Nationen** ins Spiel. Besonders wichtig für uns sind zwei davon: **SDG 4: Hochwertige Bildung** und **SDG 13: Maßnahmen zum Klimaschutz**. Als zukünftige Anwendungsentwickler könnt ihr durch die Art und Weise, wie ihr Software schreibt, einen Beitrag zu beiden Zielen leisten. Ihr werdet lernen, wie man durch klugen, effizienten Code die Leistungsfähigkeit und den Energieverbrauch verbessert – und das ist etwas, das nicht nur für eure zukünftigen Arbeitgeber wichtig ist, sondern auch für unsere Umwelt.

In den kommenden Stunden werden wir uns genauer damit beschäftigen, was „guter“ und „schlechter“ Code bedeutet. Ihr werdet sehen, wie wichtig es ist, Performance und Wartbarkeit im Blick zu haben und warum es für euch als Entwickler so entscheidend ist, auf nachhaltige Entwicklung zu achten.

Lasst uns also gemeinsam herausfinden, wie ihr mit eurer Arbeit zu einer besseren, nachhaltigeren Welt beitragen könnt!

Einleitung in einfacher Sprache

in sehr einfacher Sprache – siehe Seite 5

Liebe Schülerinnen und Schüler,

heute wollen wir über etwas sprechen, das euch als angehende Anwendungsentwickler sehr wichtig sein wird: **guter** und **schlechter** Code. Aber was bedeutet das genau? Und warum ist es so wichtig?

Guter Code ist nicht nur Code, der funktioniert. Guter Code ist auch effizient, also schnell und sparsam mit den Ressourcen. Das bedeutet, dass er weniger Rechenleistung braucht, weniger Strom verbraucht und leicht zu verstehen und zu verbessern ist.

Schlechter Code hingegen kann langsam sein, mehr Energie verbrauchen und schwer zu ändern sein. Er kann dafür sorgen, dass Programme mehr Strom verbrauchen, als eigentlich nötig – und das ist schlecht für die Umwelt. Ihr wisst bestimmt, dass große Computerzentren viel Energie benötigen. Wenn Programme nicht gut geschrieben sind, verbrauchen diese Zentren noch mehr Energie und tragen so zum Klimawandel bei.

Wir sprechen hier über **Nachhaltigkeit**. Das bedeutet, dass wir darauf achten müssen, wie wir Dinge tun, damit sie lange halten und die Umwelt möglichst wenig belasten. Auch in der Programmierung können wir durch guten Code nachhaltiger arbeiten. Ihr könnt also mit eurer Arbeit etwas Gutes für die Umwelt tun!

Außerdem spart guter Code auch Zeit und Geld, weil er leichter zu verstehen und zu pflegen ist. Das bedeutet, dass zukünftige Programmierer und ihr selbst weniger Aufwand haben werdet, wenn ihr Programme verbessern oder Fehler beheben müsst.

Ihr habt vielleicht schon von den **Nachhaltigkeitszielen** der Vereinten Nationen gehört. Zwei davon sind für uns hier besonders wichtig:

- **SDG 4: Hochwertige Bildung**
  Ihr sollt lernen, wie man gute, nachhaltige Software entwickelt.
- **SDG 13: Maßnahmen zum Klimaschutz**
  Mit effizientem Code könnt ihr dazu beitragen, den Energieverbrauch zu senken und die Umwelt zu schützen.

In den nächsten Stunden werden wir uns genauer anschauen, wie ihr durch das Schreiben von gutem Code eine positive Wirkung erzielen könnt – für eure zukünftigen Jobs und für die Welt.

Einleitung in sehr einfacher Sprache

Liebe Schülerinnen und Schüler,

heute lernen wir etwas Wichtiges über das Programmieren: **guter** und **schlechter** Code.

- **Guter Code** ist:
  - schnell
  - braucht wenig Strom
  - einfach zu verstehen
  - einfach zu ändern
- **Schlechter Code** ist:
  - langsam
  - braucht viel Strom
  - schwer zu verstehen
  - schwer zu ändern

Warum ist das wichtig?

- Computer und Programme brauchen **Strom**.
- Wenn der Code schlecht ist, verbraucht das Programm mehr Strom.
- Das ist schlecht für die **Umwelt**.
- Gute Programme helfen, die Umwelt zu schützen.

Was bedeutet das für uns?

- Ihr könnt durch **guten Code** die Umwelt schützen.
- Guter Code spart **Zeit** und **Geld**.
- Guter Code hilft, dass andere den Code besser verstehen.

Wir schauen uns auch die **Nachhaltigkeitsziele** der Vereinten Nationen an. Zwei davon sind für uns wichtig:

1. **Gute Bildung**: Ihr lernt, wie man gute und nachhaltige Programme schreibt.
2. **Klimaschutz**: Mit gutem Code spart ihr Strom und helft der Umwelt.

In den nächsten Stunden lernt ihr, was „guter“ und „schlechter“ Code ist. Und wie ihr die Welt durch eure Arbeit besser machen könnt!

2 Voraussetzungen

- OpenJDK-21 LTS
- Maven 3.9.6 oder neuer
- Eclipse IDE for Enterprise Java and Web Developers - 2024-06 oder neuer
- Grundlagen Java: Datentypen, Schleifen, ...
- Grundlagen OOP, UML

3 Beispiele

Abbildung 2: generiert mit Bing

Nachfolgend werden viele Beispiele gezeigt, wie man Schleifen bzw. Lambda-Ausdrücke verwenden kann. Die Datenbasis ist dabei eine CSV-Datei (Name: `plz_orte.csv`, welche alle Ortsnamen in Deutschland enthält mit der zusätzlichen Information über PLZ, Landkreis, Bundesland und Einwohnerzahl.[1]

plz_orte.csv (Auszug)

```
"Ortsname";"PLZ";"Landkreis";"Bundesland";"Einwohnerzahl"
"Aach";"78267";"Landkreis Konstanz";"Baden-Württemberg";"2179"
"Aach";"54298";"Landkreis Trier-Saarburg";"Rheinland-Pfalz";"7787"
"Aachen";"52062";"Städteregion Aachen";"Nordrhein-Westfalen";"15989"
"Aachen";"52064";"Städteregion Aachen";"Nordrhein-Westfalen";"23014"
"Aachen";"52066";"Städteregion Aachen";"Nordrhein-Westfalen";"32021"
"Aachen";"52068";"Städteregion Aachen";"Nordrhein-Westfalen";"15319"
"Aachen";"52070";"Städteregion Aachen";"Nordrhein-Westfalen";"22608"
"Aachen";"52072";"Städteregion Aachen";"Nordrhein-Westfalen";"21262"
"Aachen";"52074";"Städteregion Aachen";"Nordrhein-Westfalen";"30180"
```

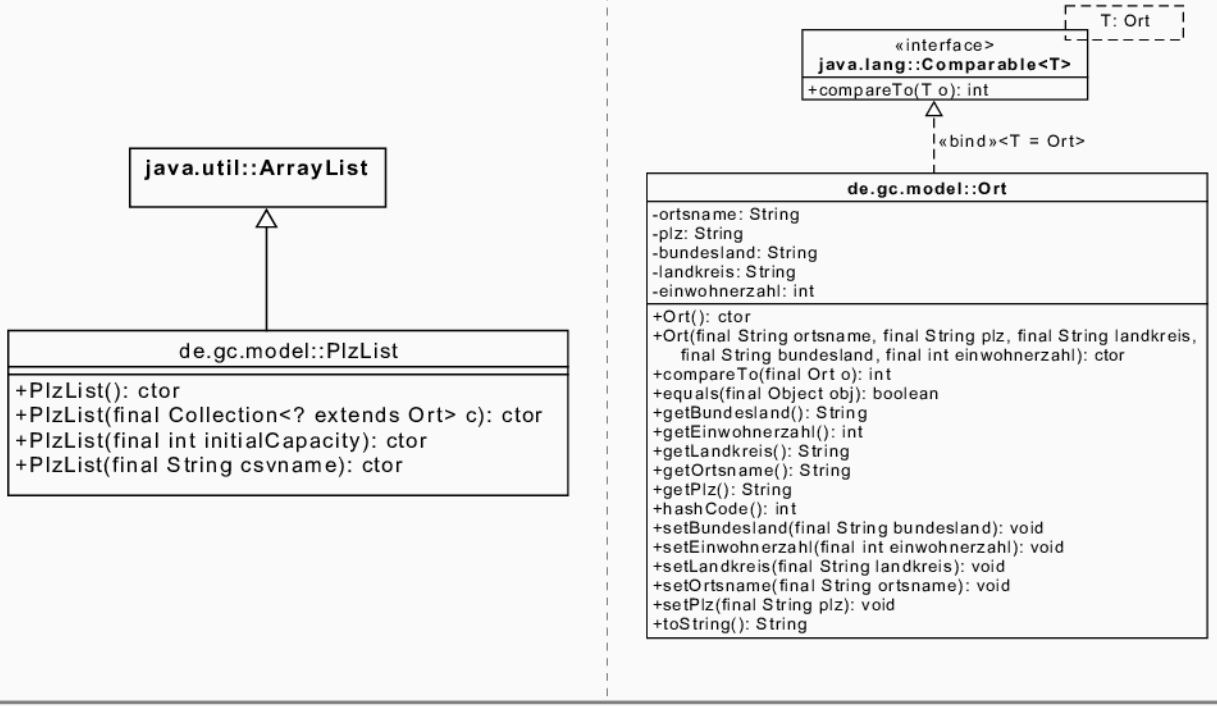
Die Daten werden mit den beiden Klassen `Ort` und `PlzList` verwaltet.

Notiz:

1 Die Quelle der Daten ist https://www.suche-postleitzahl.org/downloads. Dabei wurden die Dateien `plz_einwohner.csv` und `zuordnung_plz_ort.csv` verwendet, um mit der Klasse `Convert` die Datei `plz_orte.csv` zu erzeugen. Siehe hierzu Abschnitt 8.2 auf Seite 61.

Die Klasse `PlzList` erbt dabei von `ArrayList` und verhält sich somit wie eine normale Liste, nur dass diese bei der Initialisierung die Daten aus der Datei `plz_orte.csv` lädt. Die Klasse `Ort` bindet das Interface `Comparable` ein, so dass mit der Methoden `compareTo()` die Orte sortiert werden können.

Klasse PlzList und Ort



Notiz:

3.1 Einfache Beispiele

In nachfolgenden Beispielen wird zuerst das Ergebnis mit einer normalen Schleife erzeugt, danach mit einem entsprechenden Lambda-Stream. Zusätzlich wird die Laufzeit bzw. der Zeitunterschied zwischen den beiden Varianten ausgegeben. Die Zeitmessung hängt hier von vielen Faktoren ab, wie CPU-Geschwindigkeit, Anzahl an Kernen/Threads, Cache-Speicher, ..., daher kann das Ergebnis nur als Richtwert betrachtet werden und dient hier nur der Veranschaulichung.

3.1.1 Einfache Ausgabe

In diesem Beispiel wird jeweils die Liste (diese wird in Zeile 10 eingelesen) auf der Konsole ausgegeben. Vor jeder Ausführung wird die aktuelle Zeit (in `start` gespeichert, so dass danach die Dauer (in `time1`, ...) ermittelt werden kann.

Lamda_01_einf_01 (Auszug)

```
10      final PlzList plzListe = new PlzList("/plz_orte.csv");
11
12      // 1: einfache Ausgabe mit for-Schleife
13      long start = Util.getStartTime();
14      for (final Ort ort : plzListe) {
15         System.out.println(ort);
16      }
17      final long time1 = Util.getTime(start);
```

Hier wird die Liste in einer `foreach`-Schleife durchlaufen und der Inhalt auf der Konsole ausgegeben.

Notiz:

Lamda_01_einf_01 (Auszug)

```
19      // 2: einfache Ausgabe mit forEach kurz
20      start = Util.getStartTime();
21      plzListe.forEach(System.out::println);
22      final long time2 = Util.getTime(start);
```

Hier wird die Methode `forEach()` der Liste verwendet, um den Inhalt auszugeben.

1. **forEach**
   Die `forEach`-Methode ist eine Standardmethode, die in der `Collection`-Schnittstelle eingeführt wurde (seit Java 8). Sie ermöglicht das Durchlaufen (**Iteration**) über alle Elemente der Liste und die Ausführung einer bestimmten Operation auf jedem Element.

   Die `forEach`-Methode akzeptiert als Argument ein Objekt vom Typ `Consumer`. Ein Consumer ist eine Funktion, die ein Argument annimmt und eine Operation darauf ausführt, ohne einen Rückgabewert zu haben (ähnlich einer `void`-Methode).

2. **System.out::println**
   Dies ist eine sogenannte Methodenreferenz.
   `System.out` ist ein `PrintStream`-Objekt, und `println` ist eine Methode dieser Klasse, die dafür verantwortlich ist, eine Zeichenkette in die Konsole zu drucken.

   Methodenreferenz: Die `System.out::println`-Schreibweise ist eine kompakte und lesbare Form, die eine Methode referenziert, ohne sie direkt auszuführen. Dies ist besonders nützlich im Kontext von Lambdas und funktionalen Programmierungen. Statt die `println`-Methode direkt aufzurufen, übergibt man sie als Referenz an die `forEach`-Methode, die sie dann für jedes Element der Liste ausführt.

3. **Zusammenfassung der Funktionsweise**
   - Die `forEach`-Methode iteriert über jedes Element in der `plzListe`. Die Reihenfolge ist dabei aber nicht garantiert. Je nach „Datentopf“ kann dies unterschiedlich sein. Erwartet man eine konkrete Reihenfolge, so muss mit „Sortierung“ gearbeitet werden – später mehr dazu.
   - Für jedes Element wird die Methode `println` von `System.out` aufgerufen.
   - Das Ergebnis ist, dass jedes Element der Liste in der Konsole ausgegeben wird.

   Fazit:
   Der Befehl `plzListe.forEach(System.out::println)` ist eine elegante und effiziente Möglichkeit, jedes Element einer Liste in der Konsole auszugeben. Es kombiniert die Funktionalität von `forEach`, um über die Elemente zu iterieren, und die Methodenreferenz `System.out::println`, um die Ausgabe zu ermöglichen.

Notiz:

Lamda_01_einf_01 (Auszug)

```
24      // 3: einfache Ausgabe mit forEach lang
25      start = Util.getStartTime();
26      plzListe.forEach(s -> System.out.println(s));
27      final long time3 = Util.getTime(start);
28
29      System.out.println("######################");
30      System.out.println("### for-Schleife - forEach kurz");
31      Util.printDiff(time1, "erster", time2, "zweiter");
32      System.out.println("### for-Schleife - forEach lang");
33      Util.printDiff(time1, "erster", time3, "dritter");
```

Der Java-Befehl `plzListe.forEach(s -> System.out.println(s));` hat dieselbe Funktionalität wie `plzListe.forEach(System.out::println);`, ist jedoch auf eine andere Art und Weise formuliert. Hierbei wird ein Lambda-Ausdruck verwendet.

1. `forEach`: wie vorher auch
2. `s -> System.out.println(s)`
   Dieser Teil des Befehls ist ein Lambda-Ausdruck. Ein Lambda-Ausdruck ist eine Kurzform, um eine anonyme Funktion zu definieren, d. h. eine Funktion ohne Namen. Er ermöglicht es, kleine Codeblöcke als Argumente zu übergeben, ohne eine separate Methode dafür definieren zu müssen.

   Komponenten des Lambda-Ausdrucks:

   - **s**: Dies ist der Parameter des Lambda-Ausdrucks. In diesem Fall repräsentiert `s` jedes Element in der `plzListe`, das gerade durchlaufen wird.
   - **->** : Dieser Pfeiloperator trennt die Parameterliste (`s`) vom Funktionskörper (`System.out.println(s)`).
   - `System.out.println(s)`:
     Dies ist der Funktionskörper des Lambda-Ausdrucks. Hier wird der Parameter `s`, also das aktuelle Element der Liste, an die Methode `println` übergeben, um es in der Konsole auszugeben.
     Werden mehrere Befehle im Funktionskörper benötigt, so muss dieser in geschweifte Klammern { ... } gesetzt werden.
3. **Zusammenfassung der Funktionsweise**
   Wenn `plzListe.forEach(s -> System.out.println(s));` ausgeführt wird:
   - Die `forEach`-Methode iteriert über jedes Element in der `plzListe`.
   - Für jedes Element wird der Lambda-Ausdruck `s -> System.out.println(s)` ausgeführt.
   - Der Lambda-Ausdruck übergibt das aktuelle Element `s` an `System.out.println`, wodurch es in der Konsole ausgegeben wird.

   Vergleich mit `System.out::println`
   Obwohl `plzListe.forEach(s -> System.out.println(s));` und `plzListe.forEach(System.out::println);` funktional identisch sind, gibt es einige Unterschiede in der Art, wie sie geschrieben werden:

   - **Lambda-Ausdruck** `(s -> System.out.println(s))`:
     Mehr Flexibilität, da man den Funktionskörper anpassen kann, wenn nötig. Dies ist besonders nützlich, wenn man **mehr als nur eine** einfache Methode aufrufen oder zusätzliche Logik implementieren möchte.
   - **Methodenreferenz** `(System.out::println)`:
     Kürzer und lesbarer, wenn man **nur eine** existierende Methode ohne Änderungen verwenden möchte.

Fazit:
Der Befehl `plzListe.forEach(s -> System.out.println(s));` verwendet einen Lambda-Ausdruck, um jedes Element der Liste in der Konsole auszugeben. Er bietet mehr Flexibilität als eine Methodenreferenz, ist aber etwas länger. Beide Varianten sind jedoch in diesem Kontext funktional identisch.

Notiz:

Hier nun die Ausgabe mit dem Zeitvergleich:

Ausgabe

```
...
### for-Schleife - forEach kurz
Dauer: -126 ms erster war schneller
### for-Schleife - forEach lang
Dauer: -98 ms erster war schneller
```

Man sieht hier, dass bei einfachen Ausgaben, die `foreach`-Schleife schneller ist. Im Bezug auf BNE ist diese daher zu bevorzugen.

Aufgabe ▶ `f_bne_lambda_01`

Notiz:

3.1.2 Sortieren

Nun sollen die Daten sortiert ausgegeben werden.

Lamda_02_sort_01 (Auszug)

```
14      // 1: sortierte Ausgabe mit for-Schleife
15      long start = Util.getStartTime();
16      final PlzList sortListe = new PlzList(plzListe);
17      Collections.sort(sortListe);
18      for (final Ort ort : sortListe) {
19         System.out.println(ort);
20      }
21      final long time1 = Util.getTime(start);
22
23      // 2: sortierte Ausgabe mit stream und forEach
24      start = Util.getStartTime();
25      plzListe.stream().sorted().forEach(System.out::println);
26      final long time2 = Util.getTime(start);
```

Für die Schleifenlösung muss die Liste zuerst kopiert werden (Zeile 16), da die Originalliste nicht verändert werden soll. Danach kann diese über die statische Methode `sort()` aus der `Collections`-Klasse sortiert werden. Wird hier nichts weiter angegeben, so wird für die Sortierreihenfolge die Methode `compareTo()` aus der Klasse `Ort` verwendet. Diese sortiert die Orte zuerst nach dem Bundesland, dann nach dem Landkreis und zuletzt nach dem Ortsnamen.

Anschließend wird die Liste auf der Konsole ausgegeben.

Ort (Auszug)

```
27   /**
28    * sortiert nach bundesland, landkreis, ortsname
29    */
30   @Override
31   public int compareTo(final Ort o) {
32
33      int rt = 0;
34      rt = bundesland.compareTo(o.bundesland);
35      if (rt == 0) {
36         rt = landkreis.compareTo(o.landkreis);
37         if (rt == 0) {
38            rt = ortsname.compareTo(o.ortsname);
39         }
40      }
41      return rt;
42   }
```

Notiz:

Lamda_02_sort_01 (Auszug)

```
23      // 2: sortierte Ausgabe mit stream und forEach
24      start = Util.getStartTime();
25      plzListe.stream().sorted().forEach(System.out::println);
26      final long time2 = Util.getTime(start);
```

Mit Lambda geht dies wesentlich einfacher. Hier hängt man einfach die Methode `sorted()` ein und schon bekommt man eine sortierte Liste. Da Lambda-Streams die Originaldaten nicht anfassen, sondern nur darauf verweisen, muss die Liste auch nicht vorher kopiert werden.

Ausgabe

```
...
Dauer: -70 ms erster war schneller
```

Auch hier ist die Lösung mit der normalen Schleife schneller. Aber der große Vorteil hier ist die Einfachheit des Codes und das direkte Erkennen der „business logic“ / Geschäftslogik.

Aufgabe ▶ `f_bne_lambda_02`

Notiz:

3.1.3 Filtern

Im nächsten Schritt soll die Ausgabe gefiltert werden. Hier konkret, nur Ort die in „Bayern“ sind, sollen berücksichtigt werden.

Lamda_03_filter_01 (Auszug)

```
14      final String BNAME = "Bayern";
15
16      // 1: sortierte Ausgabe mit filter und for-Schleife
17      long start = Util.getStartTime();
18      final PlzList sortListe = new PlzList(plzListe);
19      Collections.sort(sortListe);
20      for (final Ort ort : sortListe) {
21         if (ort.getBundesland().equals(BNAME)) {
22            System.out.println(ort);
23         }
24      }
25      final long time1 = Util.getTime(start);
26
27      // 2: sortierte Ausgabe mit filter und stream, forEach
28      start = Util.getStartTime();
29      plzListe.stream().sorted().filter(o -> o.getBundesland().equals(
        BNAME)).forEach(System.out::println);
30      final long time2 = Util.getTime(start);
31
32      System.out.println("########################");
33      Util.printDiff(time1, "erster", time2, "zweiter");
```

In der normalen Schleife wird hierzu eine Verzweigung eingebaut, die nur Orte ausgibt, bei denen das Bundesland „Bayern“ ist.

Zum Vergleich, der Lambda-Stream hängt hier nur die Methode `filter()` „dazwischen“ und nimmt als Parameter einen Lambda-Ausdruck entgegen, der das Bundesland auf die Konstante `BNAME` prüft (hier ein Literal mit dem Wert „Bayern“).

Hier nun die Ausgabe mit dem Zeitvergleich:

Ausgabe

```
...
Dauer: -53 ms erster war schneller
```

Auch hier ist die Schleife etwas schneller, aber der Code ist zum Lambda-Code deutlich umfangreicher. Hier also die Entscheidung, ein etwas schnellerer Code oder ein einfacher Code, bei dem man sofort die damit verbundene Geschäftslogik erkennt.

Notiz:

Als Anwendungsentwickler sollten Sie mindestens folgende Kriterien beachten

- **Wie schnell ist der Code – Performance**?

Aspekte der Performance:

1. **Reaktionszeit** (Response Time):
   Die Zeit, die eine Anwendung benötigt, um auf eine Eingabe oder Anfrage zu reagieren. Dies ist besonders wichtig für Benutzererlebnisse in Echtzeitanwendungen.
2. **Durchsatz** (Throughput):
   Die Anzahl der Aufgaben oder Anfragen, die eine Anwendung in einem bestimmten Zeitraum verarbeiten kann. Dies ist relevant für Anwendungen, die große Mengen an Daten oder viele gleichzeitige Benutzeranfragen verarbeiten.
3. **Ressourcennutzung**:
   Wie effizient eine Software Systemressourcen wie CPU, Speicher (RAM), Festplattenspeicher und Netzwerkbandbreite nutzt. Eine gute Performance bedeutet oft, dass die Software ressourcenschonend arbeitet.
4. **Skalierbarkeit**:
   Die Fähigkeit einer Anwendung, ihre Leistung aufrechtzuerhalten oder zu verbessern, wenn die Arbeitslast steigt, z.B. durch Hinzufügen zusätzlicher Hardware-Ressourcen.
5. **Latenz**:
   Die Verzögerung zwischen dem Zeitpunkt, zu dem eine Aktion initiiert wird, und dem Zeitpunkt, zu dem ihre Wirkung spürbar wird.
6. **Bedeutung der Performance:**
   - **Benutzererfahrung**: Eine gute Performance ist entscheidend für eine positive Benutzererfahrung. Anwendungen, die schnell und reibungslos laufen, werden von den Nutzern als effizienter und zuverlässiger wahrgenommen.
   - **Effizienz**: Optimierte Performance bedeutet, dass eine Software ihre Aufgaben mit minimalem Ressourcenverbrauch erledigt, was Kosten und Energie spart.
   - **Wettbewerbsfähigkeit**: In vielen Branchen kann die Performance einer Software den Unterschied zwischen Erfolg und Misserfolg ausmachen, besonders in Märkten, in denen Geschwindigkeit und Effizienz entscheidend sind.
7. **Messung der Performance**:
   - **Profiling**: Werkzeuge zur Analyse der Performance, die Engpässe im Code identifizieren.
   - **Benchmarking**: Vergleich der Leistung einer Anwendung gegen bekannte Standards oder ähnliche Anwendungen.
   - **Lasttests**: Simulation von hohen Belastungen, um die Stabilität und Performance unter Stress zu überprüfen.

   Zusammengefasst beschreibt die Performance in der Softwareentwicklung, wie gut eine Software ihre Aufgaben unter bestimmten Bedingungen erfüllt und dabei gleichzeitig möglichst effizient mit Ressourcen umgeht.

Notiz:

- **Wie viel Speicher benötigt der Code – Ressourcen**?

In der Softwareentwicklung bezieht sich die Frage „Wie viel Speicher benötigt der Code?“ auf die Speichernutzung oder Speicherverbrauch einer Software. Diese Begriffe beschreiben, wie viel Arbeitsspeicher (RAM) und Festplattenspeicher eine Anwendung oder ein Stück Code zur Ausführung benötigt. Die Effizienz der Speichernutzung ist ein wichtiger Aspekt der Software-Performance und Ressourcennutzung.

Definition und Aspekte der Speichernutzung:

1. **Arbeitsspeicher (RAM) Nutzung**:
   - **Definition**: Dies bezieht sich auf die Menge an RAM, die eine Anwendung während ihrer Ausführung verwendet. Der RAM wird für die Zwischenspeicherung von Daten, Variablen, Objekten und temporären Informationen verwendet, die der Code benötigt, um seine Aufgaben auszuführen.
   - **Aspekte**: Dazu gehören der statische Speicher (fester Speicherbedarf für globale und statische Variablen), der Stack-Speicher (für lokale Variablen und Funktionsaufrufe) und der Heap-Speicher (für dynamisch allokierte Objekte und Datenstrukturen).
2. **Festplattenspeicher Nutzung**:
   - **Definition**: Dies beschreibt die Menge an Speicherplatz auf der Festplatte oder dem SSD, die der Code benötigt, einschließlich der Größe der ausführbaren Datei, der genutzten Bibliotheken und anderer Ressourcen wie Konfigurationsdateien, Datenbanken oder Log-Dateien.
   - **Aspekte**: Dazu gehören der Speicherbedarf für die Installation der Software, das Ablegen von Daten und temporärer Dateien, die während der Ausführung der Software erstellt werden.

Bedeutung der Speichernutzung:

- **Leistungsfähigkeit**: Ein Code, der effizient mit Speicherressourcen umgeht, kann auf einer Vielzahl von Hardwareplattformen besser und schneller laufen, insbesondere auf Geräten mit begrenztem Speicher.
- **Stabilität**: Eine Anwendung, die zu viel Speicher verbraucht (z.B. durch Speicherlecks oder ineffiziente Speicherverwaltung), kann instabil werden, abstürzen oder andere Anwendungen auf dem System beeinträchtigen.
- **Skalierbarkeit**: Anwendungen, die wenig Speicher benötigen, lassen sich oft besser skalieren, da sie weniger Hardware-Ressourcen pro Instanz verbrauchen, was insbesondere in verteilten Systemen und Cloud-Umgebungen wichtig ist.

Messung der Speichernutzung:

- **Profiler**: Werkzeuge, die die Speichernutzung von Anwendungen überwachen und analysieren können, um Engpässe und ineffiziente Speicherverwaltung zu identifizieren.
- **Heap Dumps**: Momentaufnahmen des Heaps, die während der Ausführung der Anwendung erstellt werden, um den aktuellen Zustand und die Verteilung des Speichers zu analysieren.
- **Statistiktools**: Tools zur Überwachung der Speichernutzung in Echtzeit, die Einblicke in die speicherintensivsten Teile der Anwendung geben.

Zusammengefasst:
Die Speichernutzung in der Softwareentwicklung bezieht sich auf die Menge an RAM und Festplattenspeicher, die eine Anwendung oder ein Stück Code benötigt, um ausgeführt zu werden. Eine effiziente Speichernutzung ist entscheidend für die Performance, Stabilität und Skalierbarkeit einer Software.

- **Ist der Code einfach zu warten und setzt die Geschäftslogik zielstrebig um - Maintainability**?

  In der Softwareentwicklung bezieht sich „maintainability“ auf die Eigenschaft eines Systems, einer Software oder eines Codes, die den Aufwand und die Leichtigkeit der Wartung, Aktualisierung und Fehlerbehebung betrifft. Es beschreibt, wie einfach oder schwierig es ist, Änderungen am System vorzunehmen, Fehler zu beheben oder es an neue Anforderungen anzupassen.

Für was entscheiden Sie sich?

Notiz:

3.1.4 Begrenzen

Im nächsten Schritt soll die Ausgabe gefiltert werden, wie zuvor. Zusätzlich soll die Ausgabe begrenzt werden, d. h. ein Limit wird gesetzt.

Lamda_04_filter_02 (Auszug)

```
15      final String BNAME = "Bayern";
16      final int limit = 2;
17
18      // 1: sortierte Ausgabe mit filter, limit und for-Schleife
19      long start = Util.getStartTime();
20      final PlzList sortListe = new PlzList(plzListe);
21      Collections.sort(sortListe);
22      int i = 0;
23      for (final Ort ort : sortListe) {
24         if (ort.getBundesland().equals(BNAME)) {
25            System.out.println(ort);
26            i++;
27            if (i >= limit) {
28               break;
29            }
30         }
31      }
32      System.out.println("########################");
33      final long time1 = Util.getTime(start);
34
35      // 2: sortierte Ausgabe mit filter, limit und stream, forEach
36      start = Util.getStartTime();
37      plzListe.stream().sorted()
38         .filter(o -> o.getBundesland().equals(BNAME))
39         .limit(limit).forEach(System.out::println);
40      final long time2 = Util.getTime(start);
41
42      System.out.println("########################");
43      Util.printDiff(time1, "erster", time2, "zweiter");
```

Es wird hier zusätzlich eine Konstante definiert, die das Limit festlegt, hier der Wert 2 (Zeile 15). Dabei wird in der `foreach`-Schleife zusätzlich ein Abbruch definiert, der bei erreichen des Limits die Schleife mit `break` abbricht.

In der Lambda-Variante wird hier einfach die Methode `limit` in den Stream eingebaut und die Ausgabe wird begrenzt.

Ausgabe

```
...
Dauer: -21 ms erster war schneller
```

Sie sehen, die Zeitdifferenz zwischen den beiden Lösungen wird immer geringer, der Codeaufwand aber immer deutlicher.

Notiz:

3.1.5 Neuen Datenbestand extrahieren

Im nächsten Schritt soll der gefilterte Datenbestand in einer neuen Liste gespeichert werden. Dazu wird in der Schleife eine zusätzliche neue Liste erzeugt, die die Orte in Bayern aufnimmt. Insgesamt 11 Zeilen Code. In der Lambda-Variante wird zusätzlich die Methode `collect()` eingebaut, die die Daten zu einem neuen Type wandelt. Wie gewandelt werden soll, legt hierbei der Parameter fest, hier `Collectors.toList()`, der eine neue Liste erzeugt. Ein Blick in die API zeigt, was noch so alles möglich ist.

Lamda_05_filter_03 (Auszug)

```
17      final String BNAME = "Bayern";
18
19      // 1: neue Liste: sortierte Ausgabe mit filter und for-Schleife
20      long start = Util.getStartTime();
21      final PlzList bayernListe = new PlzList();
22      final PlzList sortListe = new PlzList(plzListe);
23      Collections.sort(sortListe);
24      for (final Ort ort : sortListe) {
25         if (ort.getBundesland().equals(BNAME)) {
26            bayernListe.add(ort);
27         }
28      }
29      for (final Ort ort : bayernListe) {
30         System.out.println(ort);
31      }
32      final long time1 = Util.getTime(start);
33
34      // 2: neue Liste: sortierte Ausgabe mit filter und stream, forEach
35      start = Util.getStartTime();
36      final List<Ort> blist = plzListe.stream()
37            .sorted().filter(o -> o.getBundesland().equals(BNAME))
38            .collect(Collectors.toList());
39      blist.forEach(System.out::println);
40      final long time2 = Util.getTime(start);
41
42      System.out.println("########################");
43      Util.printDiff(time1, "erster", time2, "zweiter");
```

Ausgabe

```
...
Dauer: -47 ms erster war schneller
```

Welche Variante bevorzugen Sie?

Notiz:

3.1.6 Doppelte Einträge ignorieren

Im nächsten Schritt sollen doppelte Einträge ignoriert werden, d. h. hier sollen aus der kompletten Liste die Bundesländer ermittelt werden. Zur Info, Deutschland hat 16 Bundesländer, genauer gesagt Bundesländer und Stadtstaaten, wie z. B. die Hansestadt Hamburg.

Lamda_06_distinct_01 (Auszug)

```
15      // 1: distinct
16      long start = Util.getStartTime();
17      final ArrayList<String> bnamen = new ArrayList<>();
18      for (final Ort ort : plzListe) {
19         if (!bnamen.contains(ort.getBundesland())) {
20            bnamen.add(ort.getBundesland());
21         }
22      }
23      for (final String bn : bnamen) {
24         System.out.println(bn);
25      }
26      final long time1 = Util.getTime(start);
27      System.out.println("########################");
28
29      // 2: distinct mit stream
30      start = Util.getStartTime();
31      plzListe.stream().map(Ort::getBundesland)
32         .distinct().forEach(System.out::println);
33      final long time2 = Util.getTime(start);
34
35      System.out.println("########################");
36      Util.printDiff(time1, "erster", time2, "zweiter");
```

Im Lambda-Stream kann dies elegant über die Lambda-Methode `map` realisiert werden, gefolgt von der Methode `distinct()`.

Ausgabe

```
...
Baden-Württemberg
Rheinland-Pfalz
Nordrhein-Westfalen
Hessen
Schleswig-Holstein
Bayern
Thüringen
Niedersachsen
Mecklenburg-Vorpommern
Sachsen
Sachsen-Anhalt
Brandenburg
Saarland
Berlin
Bremen
Hamburg
########################
Dauer: 26 ms zweiter war schneller
```

Du siehst, wenn es komplexer wird, ist die Lambda-Variante schneller und eleganter. Eine Übersicht, wie Bundesländer, Landkreise, Gemeinden etc. in Deutschland strukturiert sind, zeigt Abbildung 3.

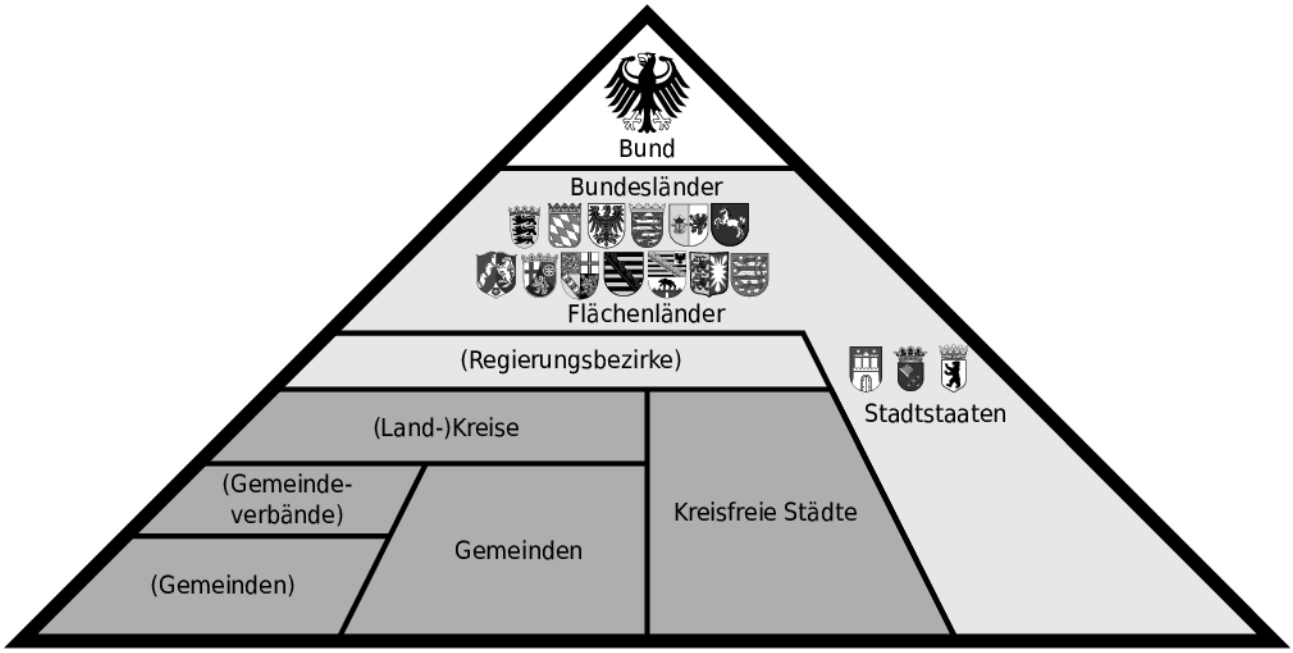


Abbildung 3: Quelle: Wikipedia

Notiz:

Aufgabe ▶ `f_bne_lambda_03`

3.1.7 Einträge gruppieren

Der nächste Schritt ist das Gruppieren von Einträgen. Da dies mit normalen Schleifen sehr aufwendig ist, nehmen wir ab jetzt nur noch die Lambda-Variante. Du kannst aber gerne selbst probieren, dies mit einer normalen Schleife umzusetzen.

Lamda_07_groupby_01 (Auszug)

```
16      final long start = Util.getStartTime();
17      final Map<String, List<Ort>> map = plzListe.stream().collect(
        Collectors.groupingBy(Ort::getBundesland));
```

Wie im Abschnitt 3.1.5 auf Seite 19 wird auch hier die Methode `collect()` verwendet. Mit der statischen Methode `groupingBy()` wird der Datenbestand gruppiert, hier nach dem Bundesland. Als Ergebnis bekommt man eine `Map`, die als „key“ den Namen des Bundeslandes enthält und als „value“ eine Liste aller Orte.

Lamda_07_groupby_01 (Auszug)

```
19      map.keySet().stream().forEach(key -> {
20         System.out.format("--- %s --------------------%n", key);
21         final List<Ort> bort = map.get(key);
22         bort.stream().sorted((o1, o2) -> Integer.compare(o1.
           getEinwohnerzahl(), o2.getEinwohnerzahl()))
23               .forEach(System.out::println);
24      });
25      System.out.println("########################");
26      Util.printTime(start);
```

Die Ausgabe wird hier aufgeteilt nach Bundesländern, wobei die Orte zusätzlich nach der Einwohnerzahl sortiert werden. Dazu wird bei `sorted()` als Parameter der Lambdaausdruck gesetzt – siehe `Comparable`.

Ausgabe

```
...
--- Bayern --------------------
...
Ort [ortsname=Theisseil, plz=92637, landkreis=Landkreis Neustadt an der
Waldnaab, bundesland=Bayern, einwohnerzahl=42982]
Ort [ortsname=Weiden in der Oberpfalz, plz=92637, landkreis=, bundesland=
Bayern, einwohnerzahl=42982]
Ort [ortsname=Dachau, plz=85221, landkreis=Landkreis Dachau, bundesland=
Bayern, einwohnerzahl=43425]
Ort [ortsname=Straubing, plz=94315, landkreis=, bundesland=Bayern,
einwohnerzahl=44503]
...
########################
Dauer: 248 ms
```

Ausführliche Beschreibung:

1. **Gruppieren der Orte nach Bundesland**
   - `plzListe.stream()`:
     Erstellt einen Stream von `Ort`-Objekten, die in `plzListe` enthalten sind.
   - `collect(Collectors.groupingBy(Ort::getBundesland))`:
     Diese Operation gruppiert die Orte nach ihrem Bundesland.
   - `Ort::getBundesland`:
     Eine Methode des `Ort`-Objekts, die das Bundesland des Ortes zurückgibt.
   - `Collectors.groupingBy`: Gruppiert die Elemente eines Streams in einer `Map`, wobei der Schlüssel das Bundesland ist und der Wert eine Liste von Orten in diesem Bundesland.
   - `Map<String, List<Ort>> map`:
     Eine `Map`, die Bundesländer (`String`) den jeweiligen Listen von `Ort`-Objekten zuordnet.
2. **Verarbeitung der gruppierten Orte nach Bundesland**
   - `map.keySet().stream().forEach(key -> ...)`:
     Iteriert über die Schlüssel (Bundesländer) der `Map`.
   - `System.out.format(-- %s --------------------%n", key)`:
     Gibt einen Header für jedes Bundesland aus, um die Ausgabe zu organisieren.
   - `%s`: Platzhalter für den String (Bundesland).
   - `final List<Ort> bort = map.get(key);`:
     Holt die Liste von Orten (`List<Ort>`) für das aktuelle Bundesland (`key`).
   - `bort.stream().sorted((o1, o2) -> Integer.compare(o1.getEinwohnerzahl(), o2.getEinwohnerzahl()))`:
     - Erstellt einen Stream der Orte (`bort`), sortiert diese nach der Einwohnerzahl.
     - `Integer.compare(o1.getEinwohnerzahl(), o2.getEinwohnerzahl())`:
       Vergleicht die Einwohnerzahlen der Orte, um sie aufsteigend zu sortieren.
     - `forEach(System.out::println)`:
       Gibt jeden Ort in der sortierten Liste auf der Konsole aus.

Zusammenfassung:

- **Gruppierung**: Die Orte werden nach ihrem Bundesland gruppiert.
- **Sortierung**: Innerhalb jedes Bundeslands werden die Orte nach ihrer Einwohnerzahl aufsteigend sortiert.
- **Ausgabe**: Die sortierten Listen der Orte werden für jedes Bundesland auf der Konsole ausgegeben, wobei jedes Bundesland eine eigene Überschrift hat.

Notiz:

3.1.8 Einträge gruppieren mit parallelStream

Im letzten Abschnitt 3.1.7 auf Seite 22 haben wir erst gruppiert und dann die Ausgabe sortiert. Nun wollen wir die Ausgabe zuerst sortieren und dann nach dem Bundesland wieder gruppieren. Somit ist dann bei der Ausgabe schon alles in der vorgegebenen Reihenfolge.

Damit der Lambdaausdruck hier nicht zu komplex wird, haben wir für das Sortieren vorher (Zeile 19...) eine Instanz von `Comparator<Ort>` erzeugt, die nach Bundesland und Einwohnerzahl sortiert.

Benötigt man diese Sortierung öfters, ist zu überlegen, ob man den `Comparator<Ort>` nicht als statisches Element in der Klasse `Ort` speichert.

Lamda_08_groupby_02 (Auszug)

```
18      final long start = Util.getStartTime();
19      final Comparator<Ort> s_BlE = (o1, o2) -> {
20         int rt = 0;
21         rt = o1.getBundesland().compareTo(o2.getBundesland());
22         if (rt == 0) {
23            rt = Integer.compare(o1.getEinwohnerzahl(), o2.getEinwohnerzahl());
24         }
25         return rt;
26      };
27
28      final Map<String, List<Ort>> map = plzListe.stream().sorted(s_BlE)
29            .collect(Collectors.groupingBy(Ort::getBundesland));
30
31      map.keySet().stream().forEach(key -> {
32         System.out.format("--- %s --------------------%n", key);
33         final List<Ort> bort = map.get(key);
34         bort.stream().forEach(System.out::println);
35      });
36      final long time1 = Util.getTime(start);
```

Ersetzt man jetzt die Methoden `stream()` durch `parallelStream()` und `groupingBy()` mit `groupingByConcurrent()`, so wird komplett mit Threads gearbeitet.

Lamda_08_groupby_02 (Auszug)

```
38      final Map<String, List<Ort>> map2 = plzListe.parallelStream()
39            .sorted(s_BlE)
40            .collect(Collectors.groupingByConcurrent(Ort::getBundesland));
41
42      map2.keySet().stream().forEach(key -> {
43         System.out.format("--- %s --------------------%n", key);
44         final List<Ort> bort = map.get(key);
45         bort.stream().forEach(System.out::println);
46      });
47      final long time2 = Util.getTime(start);
48
49      System.out.println("########################");
50      Util.printDiff(time1, "erster", time2, "zweiter");
51      System.out.println("Verwendete Kerne: " + Runtime.getRuntime().availableProcessors());
```

Ausgabe

```
...
Dauer: 152 ms zweiter war schneller
Verwendete Kerne : 4
```

Man sieht, der `parallelStream()` lohnt sich.

Erläuterung

In Java werden Lambda-Ausdrücke in Verbindung mit parallelen Streams (`parallelStream()`) genutzt, um die Verarbeitung von Daten über mehrere Threads hinweg zu optimieren. Bei der Implementierung von parallelen Streams nutzt Java das Fork/Join-Framework, welches die Aufgaben in kleinere Teilaufgaben (Tasks) zerlegt, die dann parallel auf verschiedenen Kernen ausgeführt werden können.

Hier ist ein Überblick darüber, wie Java bei der Ausführung des angegebenen Codes mit einem parallelen Stream verfährt:

1. **Initialisierung des `parallelStream`**
   - Der Aufruf von `parallelStream()` auf der Collection `plzListe` erstellt einen `Stream`, der in parallelen Modus gesetzt wird.
   - Dieser Stream wird durch das Fork/Join-Framework unterstützt, welches in Java seit Version 7 eingeführt wurde und im `java.util.concurrent`-Paket verfügbar ist.
2. **Fork/Join-Framework**
   - Das Fork/Join-Framework erstellt eine `ForkJoinPool`, die standardmäßig so viele Threads verwendet, wie es CPU-Kerne gibt.
   - Der Stream wird in verschiedene Teilaufgaben (Tasks) zerlegt, wobei jede Task ein Subset der Daten bearbeitet. Diese Subsets werden parallel auf verschiedenen Threads ausgeführt.
3. **Sortierung (`sorted(s_BlE)`)**
   - Die Methode `sorted(s_BlE)` sortiert die Elemente im Stream. Da es sich um einen parallelen Stream handelt, wird die Sortierung ebenfalls parallel durchgeführt.
   - Hierbei können die Daten in mehreren Subsets sortiert werden. Nach der Sortierung dieser Subsets müssen die Ergebnisse zu einem konsistenten Ganzen zusammengeführt werden. Dies geschieht durch die Merge-Phase des Sortieralgorithmus.
4. **Kollektieren (`collect(Collectors.groupingBy(Ort::getBundesland))`)**
   - Der `collect`-Befehl mit `Collectors.groupingBy()` gruppiert die sortierten Elemente nach dem `Bundesland`-Attribut.
   - Im parallelen Kontext sorgt `Collectors.groupingByConcurrent()` (eine parallele Variante) für thread-sicheres Gruppieren. Wenn `groupingBy` verwendet wird, handelt es sich normalerweise um eine sequentielle Gruppierung. Java sorgt dafür, dass die Ergebnisse korrekt zusammengeführt werden, auch wenn das Mapping parallel erfolgt.
5. **Zusammenführung der Ergebnisse**
   - Nachdem die Teilaufgaben abgeschlossen sind, müssen die Ergebnisse zusammengeführt werden. Dies geschieht wieder im Fork/Join-Framework, wo die Ergebnisse der parallelen Verarbeitungsschritte in einer finalen Map zusammengeführt werden.
6. **Optimierungen und Overhead**
   - Java optimiert die Parallelverarbeitung automatisch durch das Fork/Join-Framework. Allerdings kann der parallele Overhead den Nutzen übersteigen, wenn die Datenmenge klein ist oder die Operationen relativ schnell ausgeführt werden können.
   - Die Parallelisierung lohnt sich besonders bei großen Datenmengen oder rechenintensiven Operationen.

Fazit

In diesem Codebeispiel wird die Verarbeitung durch `parallelStream` parallelisiert, wobei das Fork/Join-Framework genutzt wird, um die Arbeit auf mehrere Threads zu verteilen. Sortierung und Gruppierung werden parallel durchgeführt und die Ergebnisse anschließend effizient zusammengeführt.

Tipp: Erstelle zuerst die Lambda-Kette mit `stream()` und messe die Zeit. Stelle dann auf `parallelStream` um, messe erneut die Zeit und entscheiden dann, welche Variante besser geeignet ist.

Aufgabe ▶ `f_bne_lambda_04`

Notiz:

3.1.9 Streams debuggen

Mit der Methode `peek` kann man jeden Wert im Stream ansehen, während der Stream läuft.

Lamda_09_peek_01 (Auszug)

```
11      final IntStream liste = IntStream.range(0, 5);
12
13      liste.peek(i -> System.out.println("Before sorting: " + i))
14         .boxed()
15         .sorted(Collections.reverseOrder())
16         .forEach(i -> System.out.println("After sorting: " + i));
```

Ausgabe

```
Before sorting: 0
Before sorting: 1
Before sorting: 2
Before sorting: 3
Before sorting: 4
After sorting: 4
After sorting: 3
After sorting: 2
After sorting: 1
After sorting: 0
```

Das gegebene Beispiel verwendet einen `IntStream`, um eine Reihe von Integer-Werten zu generieren, und nutzt die Methoden `peek()`, `boxed()`, `sorted()`, und `forEach()`, um die Werte zu inspizieren, zu verarbeiten und auszugeben.

1. **Erstellen des `IntStream`**:
   - `IntStream.range(0, 5)` erzeugt einen Stream von Ganzzahlen (`IntStream`), der die Werte von ‘0‘ bis ‘4‘ (einschließlich) enthält.
   - `IntStream` ist ein spezieller Stream für primitive `int`-Werte und ist Teil der Java Stream API, die eine sequenzielle oder parallele Verarbeitung von Daten ermöglicht.
2. **peek()-Methode**:
   - `peek()` ist ein intermediärer (zwischenzeitlicher) Stream-Operator, der jeden Wert im Stream inspiziert, während er durch den Stream fließt.
   - Der Lambda-Ausdruck `i -> System.out.println("Before sorting: "+ i)` wird auf jeden Wert des Streams angewendet und druckt den Wert zusammen mit dem Text „Before sorting:“ auf die Konsole.
   - Diese Operation verändert die Daten im Stream nicht, sondern erlaubt es, die Daten zu sehen, bevor sie weiterverarbeitet werden.
   - Wichtiger Punkt: Da `peek()` intermediär ist, wird es erst ausgeführt, wenn ein terminaler Operator (wie `forEach()`) aufgerufen wird.
3. **boxed()-Methode**:
   - `boxed()` konvertiert den `IntStream` (der primitive `int`-Werte enthält) in einen Stream von `Integer`-Objekten (`Stream<Integer>`).
   - Diese Umwandlung ist erforderlich, um Operationen wie `sorted(Collections.reverseOrder())` durchzuführen, die auf Objekten und nicht auf primitiven Werten arbeiten.

4. **sorted()-Methode**:
   - `sorted()` ist ebenfalls ein intermediärer Stream-Operator. In diesem Fall wird die Sortierung in umgekehrter Reihenfolge durchgeführt, da `Collections.reverseOrder()` als Comparator verwendet wird.
   - Der Comparator `Collections.reverseOrder()` sortiert die Werte in absteigender Reihenfolge.
   - Nach der Umwandlung durch `boxed()` wird der Stream nun nach dem Wert der `Integer`-Objekte sortiert, beginnend mit dem höchsten Wert.
5. **forEach()-Methode**:
   - `forEach()` ist ein terminaler Stream-Operator, der eine Aktion für jedes Element des Streams ausführt. In diesem Fall wird jeder Wert im Stream zusammen mit dem Text „After sorting:“ auf die Konsole gedruckt.
   - Da `forEach()` ein terminaler Operator ist, löst er die tatsächliche Verarbeitung des Streams aus. Alle vorherigen Operationen (`peek()`, `sorted()`) werden zu diesem Zeitpunkt ausgeführt.

Zusammenfassung:

- `peek()`
  Wird verwendet, um die ursprünglichen Werte des Streams vor der Sortierung anzuzeigen.
- `boxed()`
  Wandelt primitive `int`-Werte in `Integer`-Objekte um, damit diese sortiert werden können.
- `sorted(Collections.reverseOrder())`
  Sortiert die `Integer`-Werte in absteigender Reihenfolge.
- `forEach()`
  Gibt die sortierten Werte schließlich aus.

Notiz:

3.1.10 parallelStream beschleunigen

Ein `parallelStream` sorgt bei der Zusammenführung der Threads dafür, dass die Reihenfolge erhalten bleibt. Dies kostet aber Performance. Benötigt man die vorherige Reihenfolge nicht, so kann man diese auch „auflösen“, was die Performance erhöht.

Lamda_10_unordered_01 (Auszug)

```
10      final IntStream liste1 = IntStream.range(0, 20);
11      final IntStream liste2 = IntStream.range(0, 20);
12      liste1.forEach(i -> System.out.print(i + " "));
13      System.out.println("\n-----");
14      liste2.parallel()
15         .unordered()
16         .forEach(i -> System.out.print(i + " "));
```

Ausgabe

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
------
12 14 13 11 10 17 16 15 19 2 6 5 7 18 1 0 3 4 9 8
```

1. **Erstellen der Streams `liste1` und `liste2`:**
   - Beide Streams (`liste1` und `liste2`) erzeugen Ganzzahlen im Bereich von ‘0‘ bis ‘19‘ (exklusiv, ohne die 20) mit `IntStream.range(0, 20)`.
   - Diese Streams enthalten also die Zahlen ‘0‘ bis ‘19‘ in sequentieller Reihenfolge.
2. **Sequentielle Ausgabe mit `liste1`:**
   - `liste1` wird sequentiell verarbeitet, da kein Parallel-Stream verwendet wird. Die Methode `forEach()` durchläuft den Stream und gibt jeden Wert nacheinander auf der Konsole aus.
   - Da es sich um einen sequentiellen Stream handelt, wird die Ausgabe in der ursprünglichen Reihenfolge erfolgen.
3. **Parallele Verarbeitung mit `liste2` und `unordered()`:**
   - `liste2` wird als paralleler Stream verarbeitet, was bedeutet, dass die Elemente des Streams auf mehrere Threads verteilt werden.
   - Der Aufruf von `unordered()` signalisiert dem Stream, dass die ursprüngliche Reihenfolge der Elemente nicht beibehalten werden muss. Dies gibt dem Stream mehr Freiheit, die Elemente in beliebiger Reihenfolge zu verarbeiten und auszugeben, was die Parallelisierung möglicherweise effizienter macht.
   - Die Methode `forEach()` ist ein terminaler Operator, der die Elemente des Streams auf der Konsole ausgibt.
   - **Wichtig**: Da der Stream parallel und ungeordnet (`unordered()`) ist, gibt es keine Garantie, dass die Ausgabe in der Reihenfolge von ‘0‘ bis ‘19‘ erfolgt.

Notiz:

Zusammenfassung:

- `liste1` wird sequentiell verarbeitet, daher erfolgt die Ausgabe in der erwarteten Reihenfolge von ‘0‘ bis ‘19‘.
- `liste2` wird parallel und ohne Beachtung der Reihenfolge verarbeitet, da `unordered()` verwendet wird. Dadurch kann die Reihenfolge der Ausgabe zufällig oder unterschiedlich sein, je nachdem, wie die Elemente auf die verschiedenen Threads verteilt und bearbeitet werden.
- Der Einsatz von `unordered()` bei parallelen Streams kann die Performance verbessern, weil der Stream-Mechanismus nicht gezwungen ist, die ursprüngliche Reihenfolge der Elemente zu bewahren.

Aufgabe ▶ `f_bne_lambda_05`

Notiz:

3.2 Komplexere Beispiele

Jetzt werden die Beispiele und die Umsetzung deutlich komplexer.

3.2.1 Gruppieren und Zählen

Nun sollen die Ort (Stadtteile bei Stadtstaaten) der Bundesländer gezählt werden.

Lamda_11_groupby_03 (Auszug)

```
17      final Map<String, Long> anzOrtB = plzListe.parallelStream()
18            .unordered()
19            .collect(Collectors.groupingByConcurrent(
20                  Ort::getBundesland, Collectors.counting()));
21
22      anzOrtB.forEach((bundesland, anzahl)
23            -> System.out.format("%s : %d Orte%n", bundesland, anzahl));
```

Gruppieren und Zählen

```
Baden-Württemberg : 1303 Orte
Hessen : 559 Orte
Brandenburg : 471 Orte
Sachsen : 540 Orte
Berlin : 190 Orte
Nordrhein-Westfalen : 872 Orte
Bayern : 2272 Orte
Hamburg : 101 Orte
Mecklenburg-Vorpommern : 774 Orte
Niedersachsen : 1090 Orte
Bremen : 41 Orte
Thüringen : 665 Orte
Saarland : 69 Orte
Sachsen-Anhalt : 324 Orte
Rheinland-Pfalz : 2383 Orte
Schleswig-Holstein : 1200 Orte
######################
Dauer: 80 ms
```

1. **Paralleler Stream mit `unordered()`**
   - `plzListe` ist eine Liste von `Ort`-Objekten.
   - Mit `parallelStream()` wird ein paralleler Stream erstellt, der die Verarbeitung der Elemente auf mehrere Threads verteilt.
   - Die `unordered()`-Methode signalisiert, dass die ursprüngliche Reihenfolge der Elemente im Stream nicht beibehalten werden muss. Dadurch wird der Stream-Mechanismus flexibler und kann effizienter arbeiten, da die Verarbeitung der Elemente in beliebiger Reihenfolge erfolgen kann.
   - Diese Option kann die Performance bei paralleler Verarbeitung weiter verbessern, insbesondere, wenn die Reihenfolge der Elemente für das Endergebnis nicht relevant ist.
2. **Parallele Gruppierung und Zählung der Orte pro Bundesland**:
   - `Collectors.groupingByConcurrent()` ist eine Variante von `groupingBy()`, die speziell für parallele Streams optimiert ist. Sie ermöglicht eine thread-sichere Gruppierung, die in parallelen Umgebungen effizienter ist.
   - `Ort::getBundesland` ist eine Methodenreferenz, die das Bundesland jedes Ortes abruft.

- `Collectors.counting()` zählt die Anzahl der `Ort`-Objekte in jeder Gruppe (d. h. für jedes Bundesland).
- Das Ergebnis ist eine `ConcurrentMap<String, Long>`, bei der:
  - Der Schlüssel (`String`) das Bundesland ist.
  - Der Wert (`Long`) die Anzahl der Orte in diesem Bundesland darstellt.
- `groupingByConcurrent()` verwendet interne Mechanismen, die die Gruppierung parallel und thread-sicher durchführen. Dies kann die Performance in einer parallelen Umgebung erheblich verbessern, insbesondere bei großen Datenmengen.

3. **Effekte der Parallelisierung und 'ConcurrentMap'**:
   - Durch die Kombination von `unordered()` und `groupingByConcurrent()` wird die parallele Verarbeitung nicht nur effizienter, sondern auch die Ergebnis-Sammlung ermöglicht, dass mehrere Threads gleichzeitig auf die Map (`ConcurrentMap`) zugreifen und sie aktualisieren können, ohne dass es zu Inkonsistenzen kommt.
   - Dies ist besonders nützlich bei großen Datenmengen, bei denen die Verarbeitung und Sammlung der Ergebnisse in einem parallelen Umfeld schnell und sicher durchgeführt werden muss.
4. **Ausgabe der Ergebnisse**:
   - Die `ConcurrentMap` wird durch `forEach()` durchlaufen, um jedes Bundesland und die entsprechende Anzahl der Orte auszugeben.
   - Die Ausgabe ist sequentiell, aber die Berechnung der Werte und das Einfügen in die Map erfolgten parallel.

Zusammenfassung:

- `unordered()`
  Die Methode erlaubt es, dass die Verarbeitung der Elemente im Stream in beliebiger Reihenfolge erfolgt, was die Performance bei paralleler Verarbeitung verbessern kann.
- `groupingByConcurrent()`
  Diese Methode ist für parallele Streams optimiert und ermöglicht eine thread-sichere und effiziente Gruppierung der Elemente. Sie verwendet eine `ConcurrentMap`, die gleichzeitig von mehreren Threads beschrieben werden kann.
- **Performance**:
  Die Kombination aus parallelem Stream, `unordered()` und `groupingByConcurrent()` sorgt dafür, dass die Gruppierung und Zählung der Orte pro Bundesland in einem parallelen, hochperformanten Kontext durchgeführt wird, was insbesondere bei großen Datenmengen von Vorteil ist.
- **Ergebnis-Ausgabe**:
  Die Ausgabe erfolgt in einer sequentiellen Schleife über die `ConcurrentMap`, wobei jedes Bundesland und die Anzahl der Orte angezeigt werden.

Notiz:

3.2.2 Gruppieren, Zählen, Summe, ...

Jetzt soll für jedes Bundesland die Anzahl an Orten und die jeweilige Summe der Einwohner ermittelt werden.

Für die Datensammlung muss eine eigene Klasse erstellt werden, die die Werte speichert.

Lamda_12_groupby_04 (Auszug)

```
13   public static class BundeslandStats {
14      private final long anzahlOrte;
15      private final int summeEinwohner;
16
17      public BundeslandStats(final long anzahlOrte, final int
        summeEinwohner) {
18         this.anzahlOrte = anzahlOrte;
19         this.summeEinwohner = summeEinwohner;
20      }
21
22      public long getAnzahlOrte() {
23         return anzahlOrte;
24      }
25
26      public int getSummeEinwohner() {
27         return summeEinwohner;
28      }
29   }
```

Die Ermittlung der Daten beginnt ab Zeile 43.

Lamda_12_groupby_04 (Auszug)

```
36      // Ermitteln der Anzahl der Orte und der Summe der Einwohner pro Bundesland
37      final Map<String, BundeslandStats> anzahlUndEinwohnerProBundesland = plzListe
38        .parallelStream().unordered()
39        .collect(Collectors.groupingByConcurrent(
40          Ort::getBundesland,
41          Collectors.collectingAndThen(Collectors.toList(),
42          list -> {
43            final long anzahlOrte = list.size();
44            final int summeEinwohner = list.stream().mapToInt(Ort::getEinwohnerzahl).sum();
45            return new BundeslandStats(anzahlOrte, summeEinwohner);
46      })));
47
48      // Ausgabe der Ergebnisse
49      anzahlUndEinwohnerProBundesland.forEach((bundesland, stats)
50            -> System.out.format("%s : %d Orte, Summe Einwohner: %d%n",
51                  bundesland, stats.getAnzahlOrte(), stats.getSummeEinwohner()));
```

Gruppieren, Zählen, Summe, ...

```
Baden-Württemberg : 1303 Orte, Summe Einwohner: 11504820
Hessen : 559 Orte, Summe Einwohner: 6155950
Brandenburg : 471 Orte, Summe Einwohner: 5822556
Sachsen : 540 Orte, Summe Einwohner: 6434512
Berlin : 190 Orte, Summe Einwohner: 3291919
Nordrhein-Westfalen : 872 Orte, Summe Einwohner: 17727606
Bayern : 2272 Orte, Summe Einwohner: 14181067
Hamburg : 101 Orte, Summe Einwohner: 1726066
Mecklenburg-Vorpommern : 774 Orte, Summe Einwohner: 6523315
Niedersachsen : 1090 Orte, Summe Einwohner: 10663914
Bremen : 41 Orte, Summe Einwohner: 666600
Thüringen : 665 Orte, Summe Einwohner: 7620333
Saarland : 69 Orte, Summe Einwohner: 1000446
Sachsen-Anhalt : 324 Orte, Summe Einwohner: 3986342
Rheinland-Pfalz : 2383 Orte, Summe Einwohner: 17275690
Schleswig-Holstein : 1200 Orte, Summe Einwohner: 8864308
######################
Dauer: 87 ms
```

1. **Stream-Verarbeitung**:
   - `parallelStream()` wird verwendet, um den Stream parallel zu verarbeiten.
   - `unordered()` wird angewendet, um anzugeben, dass die Reihenfolge der Elemente nicht wichtig ist, was die Parallelverarbeitung optimieren kann.
2. **Gruppierung und Statistiken sammeln**:
   - `Collectors.groupingByConcurrent` wird verwendet, um die Orte nach Bundesland zu gruppieren, wobei die parallele Verarbeitung unterstützt wird.
   - `Collectors.collectingAndThen` wird verwendet, um nach der Gruppierung eine zusätzliche Verarbeitung durchzuführen:
     - `Collectors.toList()` sammelt die gruppierten `Ort`-Objekte in eine Liste.
     - Im nächsten Schritt (`collectingAndThen`) wird die Liste verwendet, um die Anzahl der Orte (`list.size()`) und die Summe der Einwohnerzahlen (`list.stream().mapToInt(Ort::getEinwohnerzahl).sum()`) zu berechnen.
     - Diese Statistiken werden in einer benutzerdefinierten `BundeslandStats`-Klasse gespeichert.
3. **Hilfsklasse `BundeslandStats`**:
   - Diese Klasse speichert die Anzahl der Orte und die Summe der Einwohner für jedes Bundesland.
   - Sie bietet Getter-Methoden, um diese Informationen abzurufen.
4. **Ausgabe der Ergebnisse**:
   - `forEach` wird verwendet, um die Ergebnisse für jedes Bundesland in der Konsole auszugeben.
   - Die `BundeslandStats`-Instanz liefert die Anzahl der Orte und die Summe der Einwohner, die dann formatiert ausgegeben werden.

3.2.3 Filtern, Gruppieren, Zählen, Summe, Durchschnitt, ...

Nun sollen für das Bundesland „Bayern“ die Orte nach Landkreisen gruppiert werden und für jeden Landkreis die Anzahl der Orte sowie die durchschnittliche Einwohnerzahl berechnet werden.

Für die Datensammlung muss eine eigene Klasse erstellt werden, die die Werte speichert.

Lamda_13_groupby_05 (Auszug)

```
13   public static class LandkreisStats {
14      private final long anzahlOrte;
15      private final double durchschnittEinwohner;
16      private final List<Ort> orte;
17
18      public LandkreisStats(final long anzahlOrte, final double durchschnittEinwohner, final
         List<Ort> orte) {
19         this.anzahlOrte = anzahlOrte;
20         this.durchschnittEinwohner = durchschnittEinwohner;
21         this.orte = orte;
22      }
23
24      public long getAnzahlOrte() {
25         return anzahlOrte;
26      }
27
28      public double getDurchschnittEinwohner() {
29         return durchschnittEinwohner;
30      }
31
32      public List<Ort> getOrte() {
33         return orte;
34      }
35   }
```

Die Ermittlung der Daten beginnt ab Zeile 43.

Lamda_13_groupby_05 (Auszug)

```
42      // Filtere nach Bundesland "Bayern", gruppiere nach Landkreis und berechne Statistiken
43      final Map<String, LandkreisStats> landkreiseStats = plzListe.parallelStream()
44          .unordered()
45          .filter(ort -> "Bayern".equals(ort.getBundesland())) // Filter für "Bayern"
46          .collect(Collectors.groupingByConcurrent(
47              Ort::getLandkreis,
48              Collectors.collectingAndThen(
49                  Collectors.toList(),
50                  list -> {
51                      final long anzahlOrte = list.size();
52                      final double durchschnittEinwohner = list.stream()
53                          .mapToInt(Ort::getEinwohnerzahl)
54                          .average()
55                          .orElse(0.0);
56                      return new LandkreisStats(anzahlOrte, durchschnittEinwohner, list);
57                  }
58              )
59          ));
60
61      // Ausgabe der Statistiken für jeden Landkreis
62      landkreiseStats.forEach((landkreis, stats) -> {
63          System.out.format("Landkreis: %s%n", landkreis);
64          System.out.format("Anzahl der Orte: %d%n", stats.getAnzahlOrte());
65          System.out.format("Durchschnittliche Einwohnerzahl: %.2f%n",
66                stats.getDurchschnittEinwohner());
67
68          // Ausgabe der Orte und ihrer Einwohnerzahlen
69          stats.getOrte().forEach(ort ->
70              System.out.format("Ort: %s, PLZ: %s, Einwohnerzahl: %d%n",
71                  ort.getOrtsname(), ort.getPlz(), ort.getEinwohnerzahl()));
72      });
```

Filtern, Gruppieren, Zählen, Summe, Durchschnitt, ...

```
...
Landkreis: Landkreis Unterallgäu
Anzahl der Orte: 53
Durchschnittliche Einwohnerzahl: 2637,81
Ort: Niederrieden, PLZ: 87767, Einwohnerzahl: 1362
Ort: Breitenbrunn, PLZ: 87739, Einwohnerzahl: 2352
Ort: Oberrieden, PLZ: 87769, Einwohnerzahl: 1247
Ort: Oberschönegg, PLZ: 87770, Einwohnerzahl: 941
...
########################
Dauer: 171 ms
```

1. **Filter für Bundesland „Bayern“**:
   - Wir filtern die Orte, sodass nur solche aus dem Bundesland „Bayern“ weiterverarbeitet werden.
2. **Gruppierung nach Landkreis**:
   - `Collectors.groupingByConcurrent` gruppiert die Orte nach Landkreis und verwendet `collectingAndThen`, um zusätzliche Statistiken zu berechnen:
     - `Collectors.toList()` sammelt die Orte in einer Liste.
     - `list.size()` ermittelt die Anzahl der Orte.
     - `list.stream().mapToInt(Ort::getEinwohnerzahl).average().orElse(0.0)` berechnet den Durchschnitt der Einwohnerzahlen. Wenn keine Einwohnerzahlen vorhanden sind, wird ‘0.0‘ als Standardwert verwendet.
3. **Hilfsklasse `LandkreisStats`**:
   - Diese Klasse speichert die Anzahl der Orte, den Durchschnitt der Einwohnerzahlen und die Liste der Orte für jeden Landkreis.
   - Sie bietet Getter-Methoden, um diese Informationen abzurufen.
4. **Ausgabe der Ergebnisse**:
   - Die Ergebnisse werden für jeden Landkreis formatiert ausgegeben. Dazu gehören die Anzahl der Orte, der Durchschnitt der Einwohnerzahlen und eine detaillierte Liste der Orte mit deren Einwohnerzahlen.

Notiz:

3.2.4 Filtern, Gruppieren und „having“, ...

Das nächste Beispiel zeigt, wie es sehr komplex werden kann. Zu überlegen ist, ob es hier nicht besser ist, die Aufgabe in kleine Schritte zu zerlegen.

Lamda_14_groupby_06 (Auszug)

```
21      // Erstelle die Map, die für jedes Bundesland die Summe der Einwohner und
22      // die Orte mit weniger als 1000 Einwohner enthält
23      final Map<String, Map<String, Object>> ergebnisMap = plzListe.parallelStream()
24          .unordered()
25          .collect(Collectors.groupingByConcurrent(
26              Ort::getBundesland,
27              Collectors.collectingAndThen(
28                  Collectors.toList(),
29                  liste -> {
30                      // Berechne die Summe der Einwohner
31                      final long summeEinwohner = liste.stream()
32                          .mapToLong(Ort::getEinwohnerzahl)
33                          .sum();
34
35                      // Filtere Orte mit weniger als 1000 Einwohnern und limitiere auf 10 ↩
                        Orte
36                      final List<Ort> gefilterteOrte = liste.stream()
37                          .filter(ort -> ort.getEinwohnerzahl() < 1000)
38                          .sorted(Comparator.comparingInt(Ort::getEinwohnerzahl))
39                          .limit(10)
40                          .collect(Collectors.toList());
41
42                      // Erstelle eine Map für die Ergebnisse
43                      final Map<String, Object> resultMap = new HashMap<>();
44                      resultMap.put("SummeEinwohner", summeEinwohner);
45                      resultMap.put("Orte", gefilterteOrte);
46                      return resultMap;
47                  }
48              )
49          ));
```

Lamda_14_groupby_06 (Auszug)

```
51      // Sortiere die Map nach Bundeslandnamen
52      final List<String> sortedBundesländer = new ArrayList<>(ergebnisMap.keySet());
53      sortedBundesländer.sort(String::compareTo);
54
55      // Ausgabe der Ergebnisse
56      sortedBundesländer.forEach(bundesland -> {
57          final Map<String, Object> resultMap = ergebnisMap.get(bundesland);
58          final long summeEinwohner = (Long) resultMap.get("SummeEinwohner");
59          final List<Ort> gefilterteOrte = (List<Ort>) resultMap.get("Orte");
60
61          // Ausgabe der Summe der Einwohner
62          System.out.format("Bundesland: %s, Summe Einwohner: %d%n", bundesland, ↩
            summeEinwohner);
63
64          // Ausgabe der Orte mit weniger als 1000 Einwohnern, max. 10 Orte
65          if (!gefilterteOrte.isEmpty()) {
66              System.out.println("Orte mit weniger als 1000 Einwohner:");
67              gefilterteOrte.forEach(ort ->
68                  System.out.format("Ort: %s, PLZ: %s, Einwohnerzahl: %d%n",
69                      ort.getOrtsname(), ort.getPlz(), ort.getEinwohnerzahl())
70              );
71          }
72          System.out.println();
73      });
```

Filtern, Gruppieren und „having“, ...

```
...
Bundesland: Thüringen, Summe Einwohner: 7620333
Orte mit weniger als 1000 Einwohner:
Ort: Schwarzburg, PLZ: 07427, Einwohnerzahl: 515
Ort: Weilar, PLZ: 36457, Einwohnerzahl: 886
Ort: Rippershausen, PLZ: 98639, Einwohnerzahl: 893
######################
Dauer: 113 ms
```

1. **Erstellen der Ergebnis-Map**
   - `parallelStream()`: Erzeugt einen Parallel-Stream der Orte, um die Verarbeitung zu beschleunigen.
   - `unordered()`: Gibt an, dass die Reihenfolge der Elemente nicht garantiert ist. Dies kann die Leistung in parallelen Streams verbessern.
   - `collect(Collectors.groupingByConcurrent(...))`:
     Gruppiert die Orte nach Bundesland und speichert die Ergebnisse in einer Map. `groupingByConcurrent` wird verwendet, um die Parallelität zu optimieren.
   - `Ort::getBundesland`: Gruppiert die Orte nach dem Bundesland.
   - `Collectors.collectingAndThen(...)`: Führt nach der Gruppierung eine weitere Verarbeitung auf den gesammelten Daten durch.
   - `Collectors.toList()`: Wandelt die gruppierten Orte in eine Liste um.
   - **Innerhalb der Lambda-Funktion**:
     - `long summeEinwohner = liste.stream()`
       `.mapToLong(Ort::getEinwohnerzahl).sum();`:
       Berechnet die Summe der Einwohnerzahlen für jedes Bundesland.
     - `List<Ort> gefilterteOrte = liste.stream()`
       `.filter(ort -> ort.getEinwohnerzahl() < 1000)`
       `.sorted(Comparator.comparingInt(Ort::getEinwohnerzahl))`
       `.limit(10).collect(Collectors.toList());`:
       Filtert Orte mit weniger als 1000 Einwohnern, sortiert sie nach Einwohnerzahl und begrenzt die Anzahl auf maximal 10.
   - `Map<String, Object> resultMap = new HashMap<>();`:
     Erstellt eine Map, die die Summe der Einwohner und die Liste der Orte speichert.
   - `resultMap.put(SSummeEinwohner", summeEinwohner);`:
     Fügt die Summe der Einwohner zur Map hinzu.
   - `resultMap.put(Örte", gefilterteOrte);`:
     Fügt die gefilterten Orte zur Map hinzu.
   - `return resultMap;`:
     Gibt die Map zurück, die dann in der Ergebnis-Map gespeichert wird.
2. **Sortieren der Bundesländer und Ausgabe**
   - `List<String> sortedBundesländer =`
     `new ArrayList<>(ergebnisMap.keySet());`:
     Holt die Schlüssel (Bundesländer) aus der Ergebnis-Map und erstellt eine Liste davon.
   - `sortedBundesländer.sort(String::compareTo);`:
     Sortiert die Liste der Bundesländer alphabetisch.

- `sortedBundesländer.forEach(bundesland -> ... );:`
  Iteriert über die sortierten Bundesländer und gibt für jedes die Ergebnisse aus.
- `Map<String, Object> resultMap = ergebnisMap.get(bundesland);:`
  Holt die Ergebnis-Map für das aktuelle Bundesland.
- `long summeEinwohner = (Long) resultMap.get(SSummeEinwohner");:`
  Holt die Summe der Einwohner aus der Map.
- `List<Ort> gefilterteOrte = (List<Ort>) resultMap.get(Örte");:`
  Holt die Liste der gefilterten Orte aus der Map.
- `System.out.format("Bundesland: %s, Summe Einwohner: %d%n" , bundesland, summeEinwohner);:`
  Gibt das Bundesland und die Summe der Einwohner aus.
- `gefilterteOrte.forEach(ort -> ... );:`
  Gibt die gefilterten Orte aus, falls vorhanden.

Zusammenfassung

Der Code verwendet parallele Streams und eine Kombination aus `groupingByConcurrent` und `collectingAndThen`, um die Orte nach Bundesland zu gruppieren, die Summe der Einwohner pro Bundesland zu berechnen und Orte mit weniger als 1000 Einwohnern zu filtern. Diese Ergebnisse werden in einer Map gespeichert, die dann sortiert und ausgegeben wird. Diese Methode optimiert die Verarbeitung großer Datenmengen und bietet eine klare, strukturierte Ausgabe der Ergebnisse.

Notiz:

Aufgabe ▶ `af_bne_lambda_flug`

4 Projekt

4.1 Beispielprojekt: Vergleich von Lambda-Ausdrücken und Schleifen in Java

Die Schüler sollen ein Programm entwickeln, welches die Noten einer Prüfung (Name, Vorname, Note) nach Namen sortiert ausgibt und zusätzlich die Verteilung etc. ausgeben.

Aufgabe

1. Die Noten etc. sind in einer CSV-Datei gespeichert (Auszug aus `noten.csv`).

```
Name,Vorname,Note
Müller,Anna,1
Schmidt,Peter,2
Schneider,Lisa,1
Fischer,Max,3
```

2. Die Schüler sollen sortiert nach ihrem Namen und Vornamen ausgegeben werden, gefolgt von der Note.
3. Nach der Liste soll in der Ausgabe eine Übersicht erstellt werden, wie viele Schüler eine 1 haben (mit Angabe des Prozentsatzes), eine 2 usw.
4. Zusätzlich soll in der Ausgabe eine Übersicht erfolgen, die die Noten wie folgt gruppiert:
   - gutes Ergebnis (Anzahl der Schüler mit 1 und 2)
   - ausreichendes Ergebnis (Anzahl der Schüler mit 3 und 4)
   - schlechtes Ergebnis (Anzahl der Schüler mit 5 und 6)

mögliche Ausgabe

```
Bauer, Max: 4
Becker, Paul: 3
Braun, Marie: 1
Fischer, Max: 3
Graf, Julia: 3
...

------------------------------
Note 1: 5 Schüler (20,00%)
Note 2: 6 Schüler (24,00%)
Note 3: 5 Schüler (20,00%)
Note 4: 4 Schüler (16,00%)
Note 5: 3 Schüler (12,00%)
Note 6: 2 Schüler (8,00%)

------------------------------
Gutes Ergebnis: 11 Schüler
Ausreichendes Ergebnis: 9 Schüler
Schlechtes Ergebnis: 5 Schüler
```

Es sind nun folgende Programmteile zu erstellen:

1. Eine Klasse, die Name, Vorname und Note „aufnimmt“.
2. Eine Klasse, welche die CSV-Datei einliest und als Liste zur Verfügung stellt.
3. Eine Klasse, die die Aufgabenstellung nur mit Schleifen umsetzt.
4. Eine Klasse, die die Aufgabenstellung mit Lambda-Streams umsetzt.

Beispiellösung

1. Eine Klasse, die Name, Vorname und Note „aufnimmt“.

```
1  package bsp.noten;
2
3  public class SchuelerNote implements Comparable<SchuelerNote> {
4
5     String name;
6     String vorname;
7     int note;
8
9     public SchuelerNote(final String name, final String vorname, final int
      note) {
10       this.name = name;
11       this.vorname = vorname;
12       this.note = note;
13    }
14
15    @Override
16    public int compareTo(final SchuelerNote o) {
17       int rt = this.name.compareTo(o.name);
18       if (rt == 0) {
19          rt = this.vorname.compareTo(o.vorname);
20       }
21       return rt;
22    }
23
24    public String getName() {
25       return name;
26    }
27
28    public int getNote() {
29       return note;
30    }
31
32    public String getVorname() {
33       return vorname;
34    }
35
36    public void setName(final String name) {
37       this.name = name;
38    }
39
40    public void setNote(final int note) {
41       this.note = note;
42    }
43
44    public void setVorname(final String vorname) {
45       this.vorname = vorname;
46    }
47
48    @Override
49    public String toString() {
50       return String.format("SchuelerNote [name=%s, vorname=%s, note=%s]",
      name, vorname, note);
51    }
52
53 }
```

2. Eine Klasse, welche die CSV-Datei einliest und als Liste zur Verfügung stellt.

```
1 package bsp.noten;
2
3 import java.io.FileReader;
4 import java.io.IOException;
5 import java.util.ArrayList;
6 import java.util.List;
7
8 import org.apache.commons.csv.CSVFormat;
9 import org.apache.commons.csv.CSVParser;
10 import org.apache.commons.csv.CSVRecord;
11
12 public class CsvReader {
13
14    public static List<SchuelerNote> readCsv(final String filePath) {
15       final List<SchuelerNote> notenListe = new ArrayList<>();
16
17       try (FileReader reader = new FileReader(filePath);
18             CSVParser csvParser = new CSVParser(reader, CSVFormat.DEFAULT.
              withHeader())) {
19
20          for (final CSVRecord record : csvParser) {
21             final String name = record.get("Name");
22             final String vorname = record.get("Vorname");
23             final int note = Integer.parseInt(record.get("Note"));
24
25             final SchuelerNote schuelerNote = new SchuelerNote(name,
               vorname, note);
26             notenListe.add(schuelerNote);
27          }
28
29       } catch (final IOException e) {
30          e.printStackTrace();
31       }
32
33       return notenListe;
34    }
35 }
```


3. Eine Klasse, die die Aufgabenstellung nur mit Schleifen umsetzt.

```
package bsp.noten;

import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class NotenAnalyseLoop {

   public static void main(final String[] args) {

      final List<SchuelerNote> notenListe = CsvReader.readCsv("src/main/resources/noten.csv");

      // Sortierung nach Namen
      Collections.sort(notenListe);

      // Ausgabe der sortierten Liste
      for (final SchuelerNote note : notenListe) {
         System.out.println(note.getName() + ", " + note.getVorname() + ": " + note.getNote());
      }
      System.out.println("\n------------------------------");

      // Berechnung der Notenverteilung
      final Map<Integer, Integer> notenVerteilung = new HashMap<>();
      for (final SchuelerNote note : notenListe) {
         notenVerteilung.put(note.getNote(), notenVerteilung.getOrDefault(note.getNote(), 0) + 1);
      }

      // Ausgabe der Notenverteilung
      final int gesamtAnzahl = notenListe.size();
      for (final Map.Entry<Integer, Integer> entry : notenVerteilung.entrySet()) {
         final int anzahl = entry.getValue();
         final double prozentsatz = anzahl / (double) gesamtAnzahl * 100;
         System.out.format("Note %d: %d Schüler (%.2f%%)%n", entry.getKey(), anzahl, prozentsatz);
      }
      System.out.println("\n------------------------------");

      // Gruppierung der Noten
      int gutesErgebnis = 0, ausreichendesErgebnis = 0, schlechtesErgebnis = 0;
      for (final SchuelerNote note : notenListe) {
         if (note.getNote() == 1 || note.getNote() == 2) {
            gutesErgebnis++;
         } else if (note.getNote() == 3 || note.getNote() == 4) {
            ausreichendesErgebnis++;
         } else if (note.getNote() == 5 || note.getNote() == 6) {
            schlechtesErgebnis++;
         }
      }

      // Ausgabe der gruppierten Ergebnisse
      System.out.println("Gutes Ergebnis: " + gutesErgebnis + " Schüler");
      System.out.println("Ausreichendes Ergebnis: " + ausreichendesErgebnis + " Schüler");
      System.out.println("Schlechtes Ergebnis: " + schlechtesErgebnis + " Schüler");
   }
}
```

4. Eine Klasse, die die Aufgabenstellung mit Lambda-Streams umsetzt.

```
package bsp.noten;

import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class NotenAnalyseLambda {

   public static void main(final String[] args) {
      final List<SchuelerNote> notenListe = CsvReader.readCsv("src/main/resources/noten.csv");

      // Sortierung und Ausgabe der Liste
      notenListe.stream().sorted().forEach(s -> System.out.println(s.getName() + ", " + s.getVorname() + ": "
            + s.getNote()));
      System.out.println("\n-----------------------------");

      // Berechnung der Notenverteilung
      final Map<Integer, Long> notenVerteilung = notenListe.stream().collect(Collectors.groupingBy(SchuelerNote::getNote, Collectors.counting()));

      // Ausgabe der Notenverteilung
      final int gesamtAnzahl = notenListe.size();
      notenVerteilung.forEach((note, anzahl) -> {
         final double prozentsatz = anzahl / (double) gesamtAnzahl * 100;
         System.out.format("Note %d: %d Schüler (%.2f%%)%n", note, anzahl, prozentsatz);
      });
      System.out.println("\n-----------------------------");

      // Gruppierung und Ausgabe
      final long gutesErgebnis = notenListe.stream().filter(s -> s.getNote() == 1 || s.getNote() == 2).count();
      final long ausreichendesErgebnis = notenListe.stream().filter(s -> s.getNote() == 3 || s.getNote() == 4).count();
      final long schlechtesErgebnis = notenListe.stream().filter(s -> s.getNote() == 5 || s.getNote() == 6).count();

      System.out.println("Gutes Ergebnis: " + gutesErgebnis + " Schüler");
      System.out.println("Ausreichendes Ergebnis: " + ausreichendesErgebnis + " Schüler");
      System.out.println("Schlechtes Ergebnis: " + schlechtesErgebnis + " Schüler");
   }
}
```

4.2 Schülerprojekt

Nun seid Ihr an der Reihe!

In Zweierteams soll nun Folgendes realisiert werden:

- Entwickelt eine eigene Idee, wie man Schleifen und Lambda-Streams vergleichen kann. Die Daten sollen auch hier aus einer CSV-Datei kommen. Das Beispiel muss aber etwas umfangreicher sein, als obiges Notenbeispiel.
  Wichtig: Stimmt euch in der Klasse ab, es darf kein Team die selbe Idee umsetzen!
- Setzt die Idee mit Java um.
- Definiert S.M.A.R.T-Ziele für Eurer Projekt[2]
- Erstellt einen kleinen Test (10 Fragen) für Eure Mitschüler.
- Erstellt dabei eine Präsentation (Umfang 2x3 Minuten):
  - **wichtige** Codebeispiele erklären
  - Was war im Bezug auf BNE besser und warum?

2 siehe z. B. https://de.wikipedia.org/wiki/SMART_(Projektmanagement)

5 Zusammenfassung

5.1 Funktionen und Methoden in Java Streams

Hier ist eine Liste von Methoden und Funktionen, die in Java Streams verwendet werden können:

1. **allMatch()**
   - **Beschreibung**: Prüft, ob alle Elemente im Stream einem bestimmten Prädikat entsprechen.
   - **Beispiel**:
   ```
   boolean allEven = stream.allMatch(n -> n % 2 == 0);
   ```
2. **anyMatch()**
   - **Beschreibung**: Prüft, ob mindestens ein Element im Stream einem bestimmten Prädikat entspricht.
   - **Beispiel**:
   ```
   boolean anyEven = stream.anyMatch(n -> n % 2 == 0);
   ```
3. **collect()**
   - **Beschreibung**: Reduziert den Stream auf eine Sammlung, wie eine `List`, `Set` oder `Map`.
   - **Beispiel**:
   ```
   List<String> result = stream.collect(Collectors.toList());
   ```
4. **count()**
   - **Beschreibung**: Gibt die Anzahl der Elemente im Stream zurück.
   - **Beispiel**:
   ```
   long count = stream.count();
   ```
5. **distinct()**
   - **Beschreibung**: Entfernt doppelte Elemente aus dem Stream.
   - **Beispiel**:
   ```
   Stream<String> distinctStream = stream.distinct();
   ```
6. **dropWhile()**
   - **Beschreibung**: Überspringt die längste Anfangssequenz von Elementen, die einem bestimmten Prädikat entsprechen.
   - **Beispiel**:
   ```
   Stream<String> result = stream.dropWhile(s -> s.length() < 3);
   ```
7. **filter()**
   - **Beschreibung**: Filtert Elemente basierend auf einem Prädikat.
   - **Beispiel**:
   ```
   Stream<String> result = stream.filter(s -> s.startsWith("a"));
   ```
8. **findAny()**
   - **Beschreibung**: Gibt ein beliebiges Element aus dem Stream zurück.
   - **Beispiel**:
   ```
   Optional<String> result = stream.findAny();
   ```

9. **findFirst()**
   - **Beschreibung**: Gibt das erste Element aus dem Stream zurück.
   - **Beispiel**:

```
Optional<String> result = stream.findFirst();
```

10. **flatMap()**
    - **Beschreibung**: Flacht die Ebenen eines Streams ab, indem es jedes Element eines Streams durch einen anderen Stream ersetzt.
    - **Beispiel**:

```
Stream<String> result = stream.flatMap(s -> Stream.of(s.split(" ")));
```

11. **forEach()**
    - **Beschreibung**: Führt eine Aktion für jedes Element im Stream aus.
    - **Beispiel**:

```
stream.forEach(System.out::println);
```

12. **forEachOrdered()**
    - **Beschreibung**: Wie `forEach()`, garantiert aber die Reihenfolge der Ausführung.
    - **Beispiel**:

```
stream.forEachOrdered(System.out::println);
```

13. **limit()**
    - **Beschreibung**: Begrenzt die Anzahl der Elemente im Stream.
    - **Beispiel**:

```
Stream<String> result = stream.limit(5);
```

14. **map()**
    - **Beschreibung**: Wandelt jedes Element im Stream mithilfe einer Funktion um.
    - **Beispiel**:

```
Stream<String> result = stream.map(String::toUpperCase);
```

15. **max()**
    - **Beschreibung**: Gibt das maximale Element im Stream zurück.
    - **Beispiel**:

```
Optional<String> max = stream.max(Comparator.naturalOrder());
```

16. **min()**
    - **Beschreibung**: Gibt das minimale Element im Stream zurück.
    - **Beispiel**:

```
Optional<String> min = stream.min(Comparator.naturalOrder());
```

17. **noneMatch()**
    - **Beschreibung**: Prüft, ob kein Element im Stream einem bestimmten Prädikat entspricht.
    - **Beispiel**:

```
boolean noneEven = stream.noneMatch(n -> n % 2 == 0);
```


18. **peek()**
    - **Beschreibung**: Führt eine Aktion für jedes Element im Stream aus und gibt den Stream unverändert weiter.
    - **Beispiel**:

```
Stream<String> result = stream.peek(System.out::println);
```

19. **reduce()**
    - **Beschreibung**: Reduziert den Stream auf einen einzigen Wert, basierend auf einer Binäroperation.
    - **Beispiel**:

```
Optional<String> result = stream.reduce((s1, s2) -> s1 + s2);
```

20. **skip()**
    - **Beschreibung**: Überspringt die ersten n Elemente im Stream.
    - **Beispiel**:

```
Stream<String> result = stream.skip(2);
```

21. **sorted()**
    - **Beschreibung**: Sortiert die Elemente im Stream.
    - **Beispiel**:

```
Stream<String> result = stream.sorted();
```

22. **sum()**
    - **Beschreibung**: Gibt die Summe der Elemente im Stream zurück (nur für primitive Streams).
    - **Beispiel**:

```
int sum = intStream.sum();
```

23. **takeWhile()**
    - **Beschreibung**: Gibt die längste Anfangssequenz von Elementen zurück, die einem bestimmten Prädikat entsprechen.
    - **Beispiel**:

```
Stream<String> result = stream.takeWhile(s -> s.length() < 3);
```

24. **toArray()**
    - **Beschreibung**: Wandelt den Stream in ein Array um.
    - **Beispiel**:

```
String[] array = stream.toArray(String[]::new);
```

25. **unordered()**
    - **Beschreibung**: Gibt an, dass der Stream keine feste Reihenfolge einhalten muss.
    - **Beispiel**:

```
Stream<String> result = stream.unordered();
```

5.2 Liste der verschiedenen Möglichkeiten, Streams in Java zu erzeugen

1. **Stream von einer Collection**
   - **Beschreibung**: Man kann einen Stream aus einer Java Collection (wie `List`, `Set`, etc.) erstellen.
   - **Beispiel**:

```
List<String> list = Arrays.asList("a", "b", "c");
Stream<String> stream = list.stream();
```

2. **Stream von einem Array**
   - **Beschreibung**: Man kann einen Stream aus einem Array erstellen.
   - **Beispiel**:

```
String[] array = {"a", "b", "c"};
Stream<String> stream = Arrays.stream(array);
```

3. **Stream mit `Stream.of()`**
   - **Beschreibung**: Erstellt einen Stream aus den angegebenen Elementen.
   - **Beispiel**:

```
Stream<String> stream = Stream.of("a", "b", "c");
```

4. **Leerer Stream mit `Stream.empty()`**
   - **Beschreibung**: Erstellt einen leeren Stream.
   - **Beispiel**:

```
Stream<String> stream = Stream.empty();
```

5. **Stream mit `Stream.generate()`**
   - **Beschreibung**: Erstellt einen endlosen Stream, bei dem jedes Element durch den angegebenen `Supplier` erzeugt wird.
   - **Beispiel**:

```
Stream<Double> stream = Stream.generate(Math::random).limit(10);
```

6. **Stream mit `Stream.iterate()`**
   - **Beschreibung**: Erstellt einen Stream, bei dem das erste Element der Startwert ist und jedes folgende Element durch Anwenden einer Funktion auf das vorherige Element erzeugt wird.
   - **Beispiel**:

```
Stream<Integer> stream = Stream.iterate(0, n -> n + 2).limit(10);
```

7. **Stream von einer Datei mit `Files.lines()`**
   - **Beschreibung**: Erstellt einen Stream von Strings, wobei jede Zeile einer Datei als ein Element im Stream dargestellt wird.
   - **Beispiel**:

```
Path path = Paths.get("file.txt");
Stream<String> stream = Files.lines(path);
```

8. **Stream von einem `Stream.Builder`**
   - **Beschreibung**: Mit einem `Stream.Builder` kann man Elemente nach und nach hinzufügen und dann einen Stream erstellen.
   - **Beispiel**:

```
Stream.Builder<String> builder = Stream.builder();
builder.add("a").add("b").add("c");
Stream<String> stream = builder.build();
```

9. **IntStream von einem Bereich mit `IntStream.range()`**
   - **Beschreibung**: Erstellt einen `IntStream`, der eine Sequenz von Werten innerhalb eines angegebenen Bereichs erzeugt.
   - **Beispiel**:

```
IntStream stream = IntStream.range(1, 10); // von 1 bis 9
```

10. **IntStream von einem Bereich mit `IntStream.rangeClosed()`**
    - **Beschreibung**: Wie `IntStream.range()`, aber der Endwert ist inbegriffen.
    - **Beispiel**:

```
IntStream stream = IntStream.rangeClosed(1, 10); // von 1 bis 10
```

11. **Stream von einem primitiven Array (z.B. `IntStream`)**
    - **Beschreibung**: Erstellt einen Stream aus einem Array von primitiven Werten (z. B. `int[]`).
    - **Beispiel**:

```
int[] numbers = {1, 2, 3, 4};
IntStream stream = Arrays.stream(numbers);
```

12. **Stream mit `Stream.concat()`**
    - **Beschreibung**: Kombiniert zwei Streams zu einem.
    - **Beispiel**:

```
Stream<String> stream1 = Stream.of("a", "b");
Stream<String> stream2 = Stream.of("c", "d");
Stream<String> stream = Stream.concat(stream1, stream2);
```

13. **Stream von einer Map mit `map.entrySet().stream()`**
    - **Beschreibung**: Erstellt einen Stream von den Einträgen (`Map.Entry`) einer `Map`.
    - **Beispiel**:

```
Map<String, Integer> map = new HashMap<>();
map.put("a", 1);
map.put("b", 2);
Stream<Map.Entry<String, Integer>> stream = map.entrySet().stream();
```

5.3 Zusammenfassung der wichtigsten Hintergrundinformationen

1. **Nachhaltige Entwicklung (BNE) und die Rolle von Softwareentwicklung**:
   - Bildung für nachhaltige Entwicklung (BNE) strebt an, Lernende zu befähigen, verantwortungsbewusste Entscheidungen für eine nachhaltige Zukunft zu treffen.
   - In der Softwareentwicklung bedeutet dies, dass Entwickler Code nicht nur funktional und performant schreiben sollten, sondern auch im Hinblick auf seine langfristige Auswirkung auf die Umwelt und Gesellschaft (u. a. SDGs).
   - Besonders relevant sind hier die Sustainable Development Goals (SDGs), insbesondere **SDG 4 (Hochwertige Bildung)** und **SDG 13 (Maßnahmen zum Klimaschutz)**.
2. **Guter Code vs. Schlechter Code**:
   - **Guter Code** zeichnet sich durch Effizienz, Wartbarkeit, Modularität, Lesbarkeit und geringeren Ressourcenverbrauch aus.
   - **Schlechter Code** verursacht oft mehr Energieverbrauch, führt zu schwer wartbaren Systemen, schlechter Skalierbarkeit und höherem Risiko für Systemausfälle.
   - **Energieeffizienz in der Softwareentwicklung**: Auch wenn Software immateriell wirkt, verbraucht sie physikalische Ressourcen (Hardware), die Energie benötigen. Ein ineffizientes Programm kann auf Dauer mehr Energie und Ressourcen verbrauchen, was wiederum den CO2-Fußabdruck erhöht.
3. **Performance und Leistung**:
   - Effiziente Algorithmen und Datenstrukturen können die Ausführungszeit von Programmen erheblich verkürzen und den Energieverbrauch senken.
   - Beispiele wie der Vergleich von Bubble Sort (ineffizient) und Quicksort (effizient) zeigen, wie sich unterschiedliche Algorithmen auf die Leistung auswirken.
4. **Stromverbrauch**:
   - Rechenzentren und Cloud-Services sind große Energieverbraucher. Software, die schlecht optimiert ist, erfordert mehr Rechenleistung und dadurch einen höheren Stromverbrauch.
   - Durch die Optimierung von Code können Entwickler den Stromverbrauch und somit die Umweltbelastung durch IT-Systeme reduzieren.
5. **Wartbarkeit und langfristige Nachhaltigkeit**:
   - **Wartbarkeit**: Guter Code ist leicht verständlich, modular und gut dokumentiert, was zukünftige Updates und Wartungen erleichtert.
   - **Nachhaltigkeit**: Software wird oft jahrelang verwendet. Wenn der Code von Anfang an nachhaltig entwickelt wurde, sinken langfristig die Wartungskosten und der Energieverbrauch.
6. **Bezug zu den SDGs (Sustainable Development Goals)**:
   - **SDG 4 (Hochwertige Bildung)**: Schüler sollen lernen, wie nachhaltige Softwareentwicklung zur Nachhaltigkeit beiträgt.
   - **SDG 13 (Maßnahmen zum Klimaschutz)**: Softwareentwickler können durch ihre Arbeit einen Beitrag zum Klimaschutz leisten, indem sie Code schreiben, der weniger Energie verbraucht.

5.4 Vergleich herkömmlicher Schleifen und Lambda-Streams in Java

In der Softwareentwicklung steht die Wahl zwischen verschiedenen Kontrollstrukturen häufig im Zusammenhang mit unterschiedlichen Anforderungen an den Code, wie Effizienz, Lesbarkeit und Wartbarkeit. In Java sind traditionelle Schleifen (wie `for` oder `while`) seit langem ein zentraler Bestandteil der Sprache. Mit der Einführung von **Lambda-Ausdrücken** und **Streams** in Java 8 wurde eine alternative Möglichkeit zur Datenverarbeitung und Manipulation eingeführt, die auf funktionalen Programmierkonzepten basiert. Beide Ansätze haben ihre Vor- und Nachteile, die im Kontext der Geschäftslogik, der Übersichtlichkeit und der Wartbarkeit genauer betrachtet werden müssen.

1. **Normale Schleifen**: Effizienz und Flexibilität
   Herkömmliche Schleifen, wie die `for`-Schleife oder `while`-Schleife, sind sehr flexibel und erlauben eine direkte und explizite Steuerung des Iterationsprozesses. Sie bieten insbesondere bei der Verarbeitung von Datenstrukturen wie **Listen** und **Maps** in Java eine detaillierte Kontrolle über den Ablauf und die Schritte der Iteration.

   Vorteile herkömmlicher Schleifen:

   - **Präzise Kontrolle über den Ablauf**: Herkömmliche Schleifen erlauben eine explizite Kontrolle über die Anzahl der Iterationen und die Art der Verarbeitung. Man kann auf jedes Element zugreifen, es modifizieren und je nach Geschäftslogik spezialisierte Abläufe implementieren. Beispielsweise kann man durch explizite `break`- oder `continue`-Anweisungen den Ablauf der Schleife genau steuern.
   - **Effizienz in einfachen Szenarien**: In einfachen Szenarien, insbesondere wenn der Geschäftslogik eine sequentielle Verarbeitung zugrunde liegt, können Schleifen effizient und leicht zu implementieren sein. Zudem erfolgt die Iteration oft in konstanter Zeit, was den Ressourcenverbrauch leicht einschätzbar macht.
   - **Leichte Debugging-Möglichkeiten**: Da Schleifen einen expliziten und sequentiellen Ablauf besitzen, ist das Debugging in der Regel einfacher. Der Entwickler kann leicht sehen, welcher Schritt wann ausgeführt wird, und hat mehr Kontrolle über jeden Teil der Schleife. Dies kann insbesondere in der Geschäftslogik hilfreich sein, wenn viele Bedingungen oder Operationen in einer Schleife integriert sind.

   Nachteile herkömmlicher Schleifen:

   - **Weniger deklarativer Ansatz**: Schleifen neigen dazu, den Code detailliert zu spezifizieren, was oft zu einer längeren und schwerer verständlichen Implementierung führt. Anstatt nur anzugeben, **was** gemacht werden soll, muss auch genau spezifiziert werden, **wie** es gemacht wird. Das führt zu einer stärker verfahrensorientierten Programmierung, die in komplexen Szenarien zu erhöhter Komplexität führt.
   - **Fehleranfälligkeit**: Durch die direkte Manipulation der Schleifenvariablen ist der Code anfälliger für Fehler, wie etwa Indexüberschreitungen oder unendliche Schleifen, die schwer zu erkennen sind.

2. **Lambda-Streams**: Abstraktion und Klarheit
   Lambda-Streams bieten eine deklarative und funktionale Möglichkeit zur Verarbeitung von Daten. Sie sind eine der wichtigsten Neuerungen seit Java 8 und ermöglichen es, auf eine elegantere Weise mit Daten zu arbeiten, insbesondere bei der Verarbeitung von Listen, Maps und anderen Collections.

   Vorteile von Lambda-Streams:

   - **Erhöhte Lesbarkeit und Übersichtlichkeit**: Lambda-Streams erlauben es, die Intention des Codes deutlicher zum Ausdruck zu bringen, indem sie klar trennen, **was** gemacht wird (z. B. Filtern, Transformieren) und `wie` es implementiert wird. Insbesondere für einfache Operationen wie Filtern, Mapping und Reduktion werden Lambda-Streams oft als deutlich lesbarer wahrgenommen, da sie Boilerplate-Code eliminieren und komplexe Schleifenstrukturen

abstrahieren. Statt mehrzeiliger Schleifen können Streams häufig in einer einzigen, gut lesbaren Zeile ausgedrückt werden.

Beispiel:

```
List<String> filtered = list.stream()
                          .filter(s -> s.startsWith("A"))
                          .collect(Collectors.toList());
```

Dieser Code ist deutlich kürzer und fokussiert auf das Ziel („Filtern nach einem Kriterium“), ohne den zugrunde liegenden Iterationsmechanismus detailliert darzustellen.

- **Vermeidung von Seiteneffekten**: Da Lambda-Streams auf einem funktionalen Ansatz basieren, sind sie per Konstruktion darauf ausgelegt, Seiteneffekte zu vermeiden. Dies führt zu einer sichereren und oft leichter wartbaren Implementierung, da Daten nicht versehentlich modifiziert werden können, wenn dies nicht beabsichtigt ist.
- **Parallele Verarbeitung**: Einer der größten Vorteile von Streams ist die Möglichkeit der parallelen Verarbeitung durch Aufrufe wie `parallelStream()`. Dies erlaubt es, große Datenmengen effizienter zu verarbeiten, ohne dass der Entwickler die parallele Ausführung manuell verwalten muss. In Geschäftslogiken, bei denen Performance und Skalierbarkeit entscheidend sind, kann dies von großem Nutzen sein.

Nachteile von Lambda-Streams:

- **Komplexität bei komplexen Geschäftslogiken**: Während Lambda-Streams für einfache Operationen sehr gut geeignet sind, können sie bei komplexen Geschäftslogiken schwer verständlich und wartbar werden. Dies liegt daran, dass der deklarative Ansatz zwar kompakt ist, aber manchmal schwer nachzuvollziehen ist, insbesondere wenn mehrere Filter-, Map- oder Reduce-Operationen hintereinander ausgeführt werden. Der „Fluss“ der Datenverarbeitung kann dadurch undurchsichtig werden, was es schwieriger macht, komplexe Logik oder Bedingungen zu implementieren.
- **Geringere Kontrolle über den Iterationsprozess**: Im Gegensatz zu herkömmlichen Schleifen bieten Lambda-Streams weniger Kontrolle über den exakten Ablauf der Iteration. Bestimmte Operationen wie das vorzeitige Beenden einer Schleife oder das Überspringen von Iterationen sind zwar möglich, aber oft weniger intuitiv als in einer herkömmlichen Schleife (z. B. durch `break` oder `continue`).
- **Debugging ist schwieriger**: Das Debugging von Lambda-Streams kann herausfordernder sein, da die einzelnen Schritte im Stream-Prozess nicht so leicht zu verfolgen sind wie in einer Schleife. In der Regel gibt es keine Schleifenvariablen, die explizit überwacht werden können, und das Verhalten von Methoden wie `filter` oder `map` hängt von den verwendeten Lambda-Ausdrücken ab, was den Debugging-Prozess erschwert.

5.5 Geschäftslogik, Übersichtlichkeit und Wartbarkeit

1. **Geschäftslogik**
   - **Schleifen**: Bei komplexen Geschäftslogiken, die viele Bedingungen, Verschachtelungen und Seiteneffekte erfordern, bieten herkömmliche Schleifen oft mehr Flexibilität und Präzision. Entwickler können den Fluss der Logik genau steuern und bestimmte Operationen direkt anpassen.
   - **Lambda-Streams**: Für einfachere Geschäftslogiken, bei denen Daten sequentiell verarbeitet, gefiltert oder transformiert werden, sind Streams oft besser geeignet. Sie erlauben es, die Absicht der Logik klarer darzustellen, ohne sich auf die Feinheiten des Iterationsprozesses zu konzentrieren.
2. **Übersichtlichkeit**
   - **Schleifen**: Herkömmliche Schleifen können schnell unübersichtlich werden, wenn viele Bedingungen und Operationen innerhalb der Schleife stattfinden. Dies führt oft zu längeren Methoden und schwieriger zu lesendem Code.
   - **Lambda-Streams**: Lambda-Streams fördern einen kompakteren und fokussierten Code. Allerdings kann diese Abstraktion bei zu vielen verschachtelten Operationen wieder zur Unübersichtlichkeit führen.
3. **Wartbarkeit**
   - **Schleifen**: Wartbarkeit ist in herkömmlichen Schleifen abhängig von der Komplexität. In einfachen Fällen sind sie leicht wartbar, aber in komplexeren Szenarien mit vielen Verschachtelungen und Bedingungen wird es schwieriger.
   - **Lambda-Streams**: Streams bieten eine klarere Trennung zwischen Datenoperationen und Geschäftslogik, was die Wartbarkeit oft verbessert. Jedoch kann es schwierig sein, komplexe Stream-Pipelines zu verstehen oder zu ändern.

Fazit

Herkömmliche Schleifen und Lambda-Streams haben jeweils ihre eigenen Anwendungsfälle und Vorteile. Während herkömmliche Schleifen mehr Kontrolle und Flexibilität bieten, sind Lambda-Streams häufig lesbarer, kompakter und besser für die parallele Verarbeitung geeignet. Die Wahl zwischen beiden Ansätzen hängt von der Komplexität der Geschäftslogik, der Notwendigkeit der Kontrolle über den Iterationsprozess und der langfristigen Wartbarkeit des Codes ab. Streams sind ideal für einfache, deklarative Datenverarbeitungsoperationen, während herkömmliche Schleifen ihre Stärke in der expliziten Steuerung und Anpassbarkeit der Iteration zeigen.

Abbildung 4: generiert und bearbeitet mit KI und Gimp

6 Wiederholung Grundlagen

6.1 Übersicht über Arrays in Java

1. Erstellung von Arrays

```
// Deklaration und Zuweisung mit direkter Initialisierung
int[] array1 = {1, 2, 3, 4, 5};

// Deklaration und Zuweisung später
int[] array2;
array2 = new int[5]; // Ein Array mit 5 Integer-Elementen

// Deklaration und Initialisierung mit Werten
int[] array3 = new int[]{10, 20, 30, 40, 50};
```

2. Zugriff auf Elemente

```
int value = array1[2]; // Zugriff auf das dritte Element (Index 2)
```

```
array1[2] = 100; // Ändert das dritte Element auf 100
```

3. Array-Länge

```
int length = array1.length; // Gibt die Länge des Arrays zurück
```

4. Durchlaufen von Arrays

```
for (int i = 0; i < array1.length; i++) {
    System.out.println(array1[i]);
}
```

```
for (int element : array1) {
    System.out.println(element);
}
```

5. Arrays sortieren

```
Arrays.sort(array1); // Sortiert das Array in aufsteigender Reihenfolge
```

6. Suchen in Arrays

```
int index = Arrays.binarySearch(array1, 20); // Gibt den Index des Elements ⤸
20 zurück
```

7. Arrays kopieren

```
int[] newArray = Arrays.copyOf(array1, 10); // Kopiert array1 in ein neues ⤸
Array mit 10 Elementen
```

```
int[] partArray = Arrays.copyOfRange(array1, 1, 4); // Kopiert Elemente von ⤸
Index 1 bis 3
```

8. Arrays vergleichen

```
boolean areEqual = Arrays.equals(array1, array2); // Vergleicht zwei Arrays auf Gleichheit
```

9. Arrays füllen

```
Arrays.fill(array1, 10); // Füllt das gesamte Array mit dem Wert 10
```

10. Mehrdimensionale Arrays

```
int[][] matrix = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
```

```
int value = matrix[1][2]; // Zugriff auf das Element in der zweiten Zeile, dritte Spalte
```

```
for (int i = 0; i < matrix.length; i++) {
    for (int j = 0; j < matrix[i].length; j++) {
        System.out.print(matrix[i][j] + " ");
    }
    System.out.println();
}
```

```
int[][] matrix = new int[3][3]; // Ein 3x3 Array
matrix[0][0] = 1;
matrix[0][1] = 2;
matrix[0][2] = 3;
matrix[1][0] = 4;
matrix[1][1] = 5;
matrix[1][2] = 6;
matrix[2][0] = 7;
matrix[2][1] = 8;
matrix[2][2] = 9;
```

```
for (int[] row : matrix) {
    for (int element : row) {
        System.out.print(element + " ");
    }
    System.out.println();
}
```

11. Arrays von Objekten

```
String[] names = {"Alice", "Bob", "Charlie"};
```

```
names[1] = "Robert"; // Ändert den zweiten Namen
```


6.2 Übersicht über Listen in Java

Java bietet verschiedene Implementierungen für Listen, darunter ‘ArrayList‘ und ‘LinkedList‘. Beide haben ihre eigenen Besonderheiten, Vor- und Nachteile. Diese Übersicht hilft dir, die Unterschiede und Einsatzzwecke zu verstehen.

ArrayList

Besonderheiten:

- Basierend auf einem dynamischen Array.
- Schnell beim Zugriff auf Elemente durch Index.
- Langsam bei Einfügungen und Löschungen (außer am Ende).

Vor- und Nachteile:

- **Vorteile:**
  - Schneller Zugriff auf Elemente.
  - Gute Leistung bei häufigem Lesezugriff.
- **Nachteile:**
  - Langsam bei Einfügungen und Löschungen.
  - Möglicher Speicherverbrauch durch Array-Größe.

Beispiel für `ArrayList`:

```
import java.util.ArrayList;

public class ArrayListExample {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();

        // Hinzufügen von Elementen
        list.add("Alice");
        list.add("Bob");

        // Zugriff auf ein Element
        String name = list.get(1); // Bob

        // Einfügen eines Elements
        list.add(1, "Charlie");

        // Entfernen eines Elements
        list.remove("Alice");

        // Durchlaufen der Liste
        for (String element : list) {
            System.out.println(element);
        }
    }
}
```

LinkedList

Besonderheiten:

- Basierend auf einer doppelt verketteten Liste.
- Schnell bei Einfügungen und Löschungen (besonders am Anfang oder Ende).
- Langsam beim Zugriff auf Elemente durch Index.

Vor- und Nachteile:

- **Vorteile:**
  - Schnelle Einfügungen und Löschungen.
  - Geringer Speicherverbrauch bei variabler Größe.
- **Nachteile:**
  - Langsamere Zugriffszeiten.
  - Höherer Speicherverbrauch durch zusätzliche Verweise.

Beispiel für `LinkedList`:

```
import java.util.LinkedList;

public class LinkedListExample {
    public static void main(String[] args) {
        LinkedList<String> list = new LinkedList<>();

        // Hinzufügen von Elementen
        list.add("Alice");
        list.add("Bob");

        // Zugriff auf ein Element
        String name = list.get(1); // Bob

        // Einfügen eines Elements am Anfang
        list.addFirst("Charlie");

        // Entfernen eines Elements
        list.remove("Alice");

        // Durchlaufen der Liste
        for (String element : list) {
            System.out.println(element);
        }
    }
}
```

7 Wissenstest

- `ff_bne_lambda_bz`
- `ff_bne_lambda_fkt_fragen`

8 Anhang

8.1 Klasse Util

Die Klasse `Util` enthält Methoden, um die Dauer von Codeabschnitten zu ermitteln.

```
1 package de.gc.util;
2
3 public class Util {
4
5    public static long getEndTime() {
6       return System.currentTimeMillis();
7    }
8
9    public static long getStartTime() {
10      return System.currentTimeMillis();
11   }
12
13   public static long getTime(final long startTime) {
14      return getEndTime() - startTime;
15   }
16
17   public static long getTime(final long startTime, final long endTime) {
18      return endTime - startTime;
19   }
20
21   public static void printDiff(final long time1, final String s1, final long time2, final String ↩
     s2) {
22      final long diff = time2 - time1;
23      System.out.format("Dauer: %d ms %s war schneller%n", diff, diff < 0 ? s1 : s2);
24   }
25
26   public static void printTime(final long startTime) {
27      printTime(startTime, getEndTime());
28   }
29
30   public static void printTime(final long startTime, final long endTime) {
31      System.out.format("Dauer: %d ms%n", endTime - startTime);
32   }
33 }
```

8.2 Klasse Convert

Die Klasse `Convert` liest die Dateien `plz_einwohner.csv` und `zuordnung_plz_ort.csv` ein, um die Datei `plz_orte.csv` zu erzeugen.

Dabei wird in der Methode `convertCsvToPlzList` zuerst die Datei `plz_einwohner.csv` eingelesen (Teil 1) und in einer Map die Einwohnerzahl zu einer PLZ gespeichert. Im Teil 2 wird dann die Datei `zuordnung_plz_ort.csv` eingelesen und für jeden Eintrag eine Instanz von `Ort` erzeugt, wobei die Einwohnerzahl aus der Map eingefügt wird.

In der `main`-Methode wird die eingelesene `plzList` in einer CSV-Datei gespeichert, die in den Beispielen verwendet wird.

```
1 package de.gc.util;
2
3 import java.io.BufferedReader;
4 import java.io.BufferedWriter;
5 import java.io.FileWriter;
6 import java.io.IOException;
7 import java.io.InputStreamReader;
8 import java.util.HashMap;
9
10 import org.apache.commons.csv.CSVFormat;
11 import org.apache.commons.csv.CSVPrinter;
12 import org.apache.commons.csv.CSVRecord;
13 import org.apache.commons.csv.QuoteMode;
14
15 import de.gc.model.Ort;
16 import de.gc.model.PlzList;
17
18 public class Convert {
19
20    private static final String PLZ_ORTE_CSV = "src/main/resources/plz_orte.csv";
21
22    public static void main(final String[] args) {
23       final Convert convert = new Convert();
24       try {
25          final PlzList plzList = convert.convertCsvToPlzList("/plz_einwohner.csv", "/zuordnung_plz_ort.csv");
26          plzList.forEach(System.out::println);
27          convert.writeToCsv(plzList, PLZ_ORTE_CSV);
```

```
28          System.out.format("Anzahl Orte: %d - in Datei %s gespeichert%n", plzList.size(), PLZ_ORTE_CSV);
29       } catch (final IOException e) {
30          e.printStackTrace();
31       }
32
33    }
34
35    public PlzList convertCsvToPlzList(final String plzEinwohnerCsv, final String zuordnungPlzOrtCsv)
36          throws IOException {
37
38       final PlzList plzList = new PlzList();
39
40       // Teil 1 --------------------
41       final BufferedReader plzEinReader = new BufferedReader(
42             new InputStreamReader(getClass().getResourceAsStream(plzEinwohnerCsv)));
43       final HashMap<String, Integer> plzEinMap = new HashMap<String, Integer>();
44
45       Iterable<CSVRecord> records = CSVFormat.EXCEL.builder().setDelimiter(',').setQuote('"').setIgnoreEmptyLines(true)
46             .setIgnoreSurroundingSpaces(true).setHeader().setSkipHeaderRecord(true).build().parse(plzEinReader);
47       for (final CSVRecord record : records) {
48          final String plz = record.get(0);
49          final String einwohner = record.get(2);
50          // System.out.println(plz + " " + einwohner);
51          int ieinwohner = 0;
52          try {
53             ieinwohner = Integer.parseInt(einwohner.trim());
54          } catch (final NumberFormatException e) {
55             System.out.println("Einwohnerzahl konnte nicht in Integer umgewandelt werden: " + einwohner);
56          }
57          plzEinMap.put(plz, ieinwohner);
58       }
59       plzEinReader.close();
60
61       // Teil 2 --------------------
62       final BufferedReader plzOrtReader = new BufferedReader(
63             new InputStreamReader(getClass().getResourceAsStream(zuordnungPlzOrtCsv)));
64       records = CSVFormat.EXCEL.builder().setDelimiter(',').setQuote('"').setIgnoreEmptyLines(true)
65             .setIgnoreSurroundingSpaces(true).setHeader().setSkipHeaderRecord(true).build().parse(plzOrtReader);
66       for (final CSVRecord record : records) {
67          final String ort = record.get(2).trim();
68          final String plz = record.get(3).trim();
69          final String landkreis = record.get(4).trim();
70          final String bundesland = record.get(5).trim();
71          int ieinwohner = 0;
72          if (plzEinMap.containsKey(plz)) {
73             ieinwohner = plzEinMap.get(plz);
74          }
75          final Ort o = new Ort(ort, plz, landkreis, bundesland, ieinwohner);
76          plzList.add(o);
77          // System.out.println(o);
78       }
79       plzOrtReader.close();
80
81       return plzList;
82    }
83
84    private void writeToCsv(final PlzList plzList, final String filename) {
85
86       try (BufferedWriter w = new BufferedWriter(new FileWriter(filename))) {
87
88          final CSVPrinter csvPrinter = new CSVPrinter(w,
89                CSVFormat.EXCEL.builder().setDelimiter(";").setQuote('"').setQuoteMode(QuoteMode.ALL)
90                      .setHeader(new String[] { "Ortsname", "PLZ", "Landkreis", "Bundesland", "Einwohnerzahl" })
91                      .build());
92
93          for (final Ort o : plzList) {
94             csvPrinter.printRecord(o.getOrtsname(), o.getPlz(), o.getLandkreis(), o.getBundesland(),
95                   o.getEinwohnerzahl());
96          }
97          csvPrinter.close();
98          w.close();
99
100      } catch (final IOException e) {
101         e.printStackTrace();
102      }
103   }
104}
```

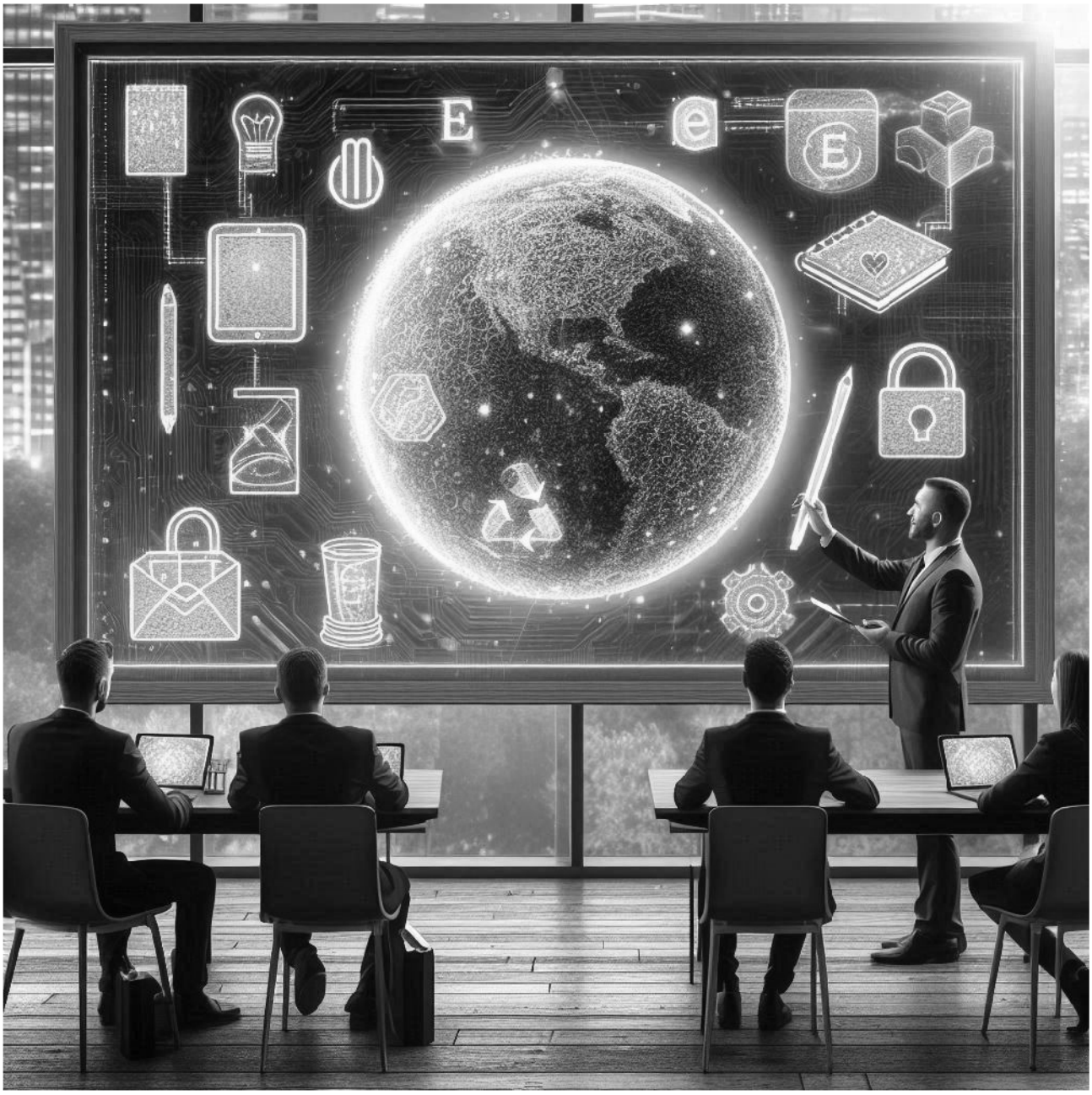



Abbildung 5: generiert mit Bing