dadurch vermeiden, dass man keine Überladungen schreibt, die sich nur durch verschiedene funktionale Schnittstellen an der gleichen Argumentposition unterscheiden. Dies ist ein Sonderfall des Hinweises aus Thema 52, »Verwenden Sie Überladung mit Bedacht«.

Fazit: Nun, da Java um das Konzept der Lambda-Ausdrücke erweitert wurde, sollten Sie beim Entwurf Ihrer APIs stets auch den Einsatz von Lambdas im Hinterkopf haben. Akzeptieren Sie funktionale Schnittstellentypen als Eingabe und liefern Sie sie als Ergebnis zurück. Verwenden Sie nach Möglichkeit die im Standard vordefinierten Schnittstellen aus `java.util.function.function.Function`, aber seien Sie offen für die relativ seltenen Fälle, in denen es besser ist, eine eigene funktionale Schnittstelle zu schreiben.

7.4 Thema 45: Setzen Sie Streams mit Bedacht ein

Die Streams-API wurde zusammen mit Java 8 eingeführt, um die Durchführung sowohl sequenzieller als auch paralleler Massenoperationen zu erleichtern. Die API stellt zwei zentrale Abstraktionen zur Verfügung: den *Stream*, der eine endliche oder unendliche Folge von Datenelementen darstellt, und die *Stream-Pipeline*, die eine mehrstufige Berechnung auf diesen Elementen repräsentiert. Die Elemente in einem Stream können beliebigen Ursprungs sein. Häufige Quellen sind Sammlungen, Arrays, Dateien, Ergebnisse von Mustervergleichen mit regulären Ausdrücken, Pseudozufallszahlengeneratoren oder andere Streams. Die Datenelemente in einem Stream können Objektreferenzen oder elementare Werte sein. Drei elementare Typen werden unterstützt: `int`, `long` und `double`.

Eine Stream-Pipeline besteht aus einem Quell-Stream, gefolgt von keiner oder mehreren *Zwischenoperationen* und einer abschließenden *Endoperation*. Jede Zwischenoperation transformiert den Stream in irgendeiner Weise, beispielsweise indem sie jedes Element auf eine Funktion dieses Elements abbildet oder alle Elemente herausfiltert, die irgendeine Bedingung nicht erfüllen. Zwischenoperationen transformieren einen Stream in einen anderen Stream, dessen Elementtyp sowohl gleich als auch verschieden vom Eingangs-Stream sein kann. Die Endoperation führt auf dem Stream, der sich aus der letzten Zwischenoperation ergibt, eine abschließende Berechnung aus, zum Beispiel das Speichern der Elemente in einer Sammlung, die Rückgabe eines bestimmten Elements oder die Ausgabe aller Elemente im Stream.

Stream-Pipelines werden nach dem Faulheitsprinzip ausgewertet (lazy evaluation): Die Auswertung beginnt erst mit dem Aufruf der Endoperation, und Datenelemente, die nicht benötigt werden, um die Endoperation abzuschließen, werden überhaupt nicht berechnet. Diese faule oder späte Auswertung macht es möglich, mit unendlichen Streams zu arbeiten. Beachten Sie, dass eine Stream-Pipeline ohne Endoperation eine stillschweigende Keine-Operationen-Pipeline ist. Also vergessen Sie nicht, eine Endoperation vorzusehen.

Streams-APIs sind *sprechende* APIs: Sie sind so konzipiert, dass alle Aufrufe, die zu einer Pipeline gehören, in einem einzigen Ausdruck verkettet werden. Tatsächlich können sogar mehrere Pipelines zu einem einzigen Ausdruck verkettet werden.

Standardmäßig werden Stream-Pipelines sequenziell ausgeführt. Um eine Pipeline parallel auszuführen, bedarf es allerdings nicht mehr, als die `parallel`-Methode für irgendeinen Stream in der Pipeline aufzurufen – aber es ist selten angebracht, dies zu tun (Thema 48).

Die Streams-API ist so flexibel, dass praktisch jede Berechnung mithilfe von Streams durchgeführt werden kann. Doch nur weil Sie es können, heißt das nicht, dass Sie es sollten. Bei sachgemäßer Verwendung können Streams die Programme kürzer und übersichtlicher machen. Bei unsachgemäßer Verwendung können sie das Lesen und Warten von Programmen erschweren. Es gibt keine festen, einfachen Regeln für die Verwendung von Streams, aber es gibt Heuristiken.

Sehen Sie sich das folgende Programm an, das die Wörter aus einer Wörterbuchdatei liest und alle Anagrammgruppen ausgibt, deren Größe einem vom Benutzer festgelegten Minimum entspricht. Zur Erinnerung, zwei Wörter bilden ein Anagramm, wenn sie aus den gleichen Buchstaben in einer anderen Reihenfolge bestehen. Das Programm liest alle Wörter aus einer vom Benutzer vorgegebenen Wörterbuchdatei und fügt sie in eine Map ein. Der Schlüssel für den Map-Eintrag ist das Wort mit seinen Buchstaben in alphabetischer Reihenfolge. Für das Wort »AMPEL« wäre der Schlüssel also »AELMP«, und der Schlüssel für »LAMPE« wäre ebenfalls »AELMP«: Die beiden Wörter sind Anagramme, und alle Anagramme haben die gleiche alphabetische Form (auch als *Alphagramm* bezeichnet). Der Wert des Map-Eintrags ist eine Liste, die alle Wörter mit der gleichen alphabetischen Form enthält. Nach Abarbeitung des Wörterbuchs repräsentiert jede Liste eine vollständige Anagrammgruppe. Das Programm braucht dann nur noch die `values()`-Auflistung der Map zu durchlaufen und die Listen auszugeben, deren Größe den Schwellenwert erreicht:

```
// Gibt alle großen Anagramm-Gruppen in einem Wörterbuch aus
public class Anagrams {
    public static void main(String[] args) throws IOException {
        File dictionary = new File(args[0]);
        int minGroupSize = Integer.parseInt(args[1]);

        Map<String, Set<String>> groups = new HashMap<>();
        try (Scanner s = new Scanner(dictionary)) {
            while (s.hasNext()) {
                String word = s.next();
                groups.computeIfAbsent(alphabetize(word),
                    (unused) -> new TreeSet<>()).add(word);
            }
        }

        for (Set<String> group : groups.values())
```

```
            if (group.size() >= minGroupSize)
                System.out.println(group.size() + ": " + group);
        }

        private static String alphabetize(String s) {
            char[] a = s.toCharArray();
            Arrays.sort(a);
            return new String(a);
        }
    }
```

Ein Schritt in diesem Programm ist es wert, dass man ihn näher erläutert. Das Einfügen der einzelnen Wörter in die Map, im Listing fett hervorgehoben, erfolgt mithilfe der Methode `computeIfAbsent`, die in Java 8 hinzugekommen ist. Diese Methode sucht in der Map einen Schlüssel: Wenn der Schlüssel vorhanden ist, gibt die Methode einfach den dazugehörigen Wert zurück. Wenn nicht, berechnet die Methode einen Wert, indem sie das angegebene Funktionsobjekt auf den Schlüssel anwendet, assoziiert diesen Wert mit dem Schlüssel und liefert den berechneten Wert zurück. Die Methode `computeIfAbsent` vereinfacht die Implementierung von Maps, die jedem Schlüssel mehrere Werte zuordnen.

Betrachten Sie nun das folgende Programm, das das gleiche Problem mithilfe von Streams löst. Beachten Sie, dass praktisch das gesamte Programm, mit Ausnahme des Codes, der die Wörterbuchdatei öffnet, aus einem einzigen Ausdruck besteht. Und der einzige Grund, warum das Wörterbuch in einem separaten Ausdruck geöffnet wird, ist die Verwendung der `try`-with-resources-Anweisung, die sicherstellt, dass die Wörterbuchdatei geschlossen wird:

```
// Überbeanspruchung von Streams – nicht nachahmen!
public class Anagrams {
  public static void main(String[] args) throws IOException {
    Path dictionary = Paths.get(args[0]);
    int minGroupSize = Integer.parseInt(args[1]);

      try (Stream<String> words = Files.lines(dictionary)) {
        words.collect(
          groupingBy(word -> word.chars().sorted()
                      .collect(StringBuilder::new,
                        (sb, c) -> sb.append((char) c),
                        StringBuilder::append).toString()))
          .values().stream()
            .filter(group -> group.size() >= minGroupSize)
            .map(group -> group.size() + ": " + group)
            .forEach(System.out::println);
        }
    }
}
```

Wenn es Ihnen schwerfällt, diesen Code zu lesen und zu verstehen, machen Sie sich keine Sorgen, Sie sind nicht allein. Der Code ist zweifelsohne kürzer, aber es

ist auch schlechter lesbar, besonders für Programmierer, die keine Experten im Umgang mit Streams sind. **Die Überbeanspruchung von Streams erschwert das Lesen und Warten von Programmen.**

Glücklicherweise gibt es einen guten Mittelweg. Das folgende Programm löst das gleiche Problem, indem es Streams verwendet, ohne sie zu sehr zu beanspruchen. Das Ergebnis ist ein Programm, das kürzer und klarer als das Original ist:

```
// Eine angemessene Verwendung von Streams fördert Klarheit und Prägnanz
public class Anagrams {
   public static void main(String[] args) throws IOException {
      Path dictionary = Paths.get(args[0]);
      int minGroupSize = Integer.parseInt(args[1]);

      try (Stream<String> words = Files.lines(dictionary)) {
         words.collect(groupingBy(word -> alphabetize(word)))
           .values().stream()
           .filter(group -> group.size() >= minGroupSize)
           .forEach(g -> System.out.println(g.size() + ": " + g));
      }
   }

   // alphabetize-Methode wie im Original
}
```

Selbst für Programmierer, die zuvor nur selten mit Streams zu tun hatten, ist dieses Programm nicht schwer zu verstehen. Es öffnet die Wörterbuchdatei in einem `try`-with-resources-Block und erhält so einen Stream, der aus den Zeilen der Datei besteht. Als Name für die `stream`-Variable wurde `words` gewählt, um anzudeuten, dass die Elemente im Stream einzelne Wörter sind. Die Pipeline auf diesem Stream hat keine Zwischenoperationen. Ihre Endoperation sammelt alle Wörter in einer Map, die die Wörter nach ihrer alphabetischen Form gruppiert (Thema 46). Dies ist exakt die gleiche Map, wie sie auch in den beiden Vorgängerversionen des Programms erstellt wurde. Dann wird ein neuer Stream vom Typ `Stream<List<String>>` für die `values()`-Auflistung der Map geöffnet. Die Elemente in diesem Stream sind natürlich die Anagrammgruppen. Der Stream wird so gefiltert, dass alle Gruppen, deren Größe kleiner als `minGroupSize` ist, ignoriert werden. Zum Schluss werden die übrig gebliebenen Gruppen von der Endoperation `forEach` ausgegeben.

Beachten Sie, dass die Namen der Lambda-Parameter sorgfältig ausgewählt wurden. Der Parameter `g` sollte eigentlich `group` heißen, aber die Codezeile wäre dann zu breit für das Buchlayout geworden. **Wo es keine expliziten Typangaben gibt, ist eine sorgfältige Benennung der Lambda-Parameter für die Lesbarkeit von Stream-Pipelines unerlässlich.**

Beachten Sie auch, dass die Alphabetisierung der Wörter in einer separaten Methode `alphabetize` erfolgt. Auch dies trägt zur besseren Lesbarkeit bei, da die Operation so einen Namen erhält und Implementierungsdetails aus dem Haupt-

programm herausgehalten werden. **Die Verbesserung der Lesbarkeit mithilfe von Hilfsmethoden ist für Stream-Pipelines noch wichtiger als für iterativen Code,** da Pipelines keine expliziten Typinformationen oder benannte temporäre Variablen enthalten.

Die `alphabetize`-Methode hätte ebenfalls auf Streams umgestellt werden können, aber eine Stream-basierte `alphabetize`-Methode wäre weniger klar, schwieriger zu schreiben und vermutlich langsamer gewesen. Der Grund dafür ist, dass es in Java keine Unterstützung für Streams des elementaren Datentyps `char` gibt. Das soll nicht heißen, dass Java `char`-Streams hätte unterstützen sollen, das wäre einfach nicht sinnvoll gewesen. Um Ihnen die Gefahren der Verarbeitung von `char`-Werten mit Streams vor Augen zu führen, sehen Sie sich den folgenden Code an:

```
"Hello world!".chars().forEach(System.out::print);
```

Sie würden wahrscheinlich erwarten, dass die Ausgabe dieses Codes `Hello world!` lautet, aber wenn Sie den Code ausführen, werden Sie feststellen, dass er `721011081081113211911111410810033` ausgibt. Dies liegt daran, dass die Elemente des Streams, die von `"Hello world!".chars()` zurückgeliefert werden, keine `char`-, sondern `int`-Werte sind, sodass die `int`-Überladung von `print` aufgerufen wird. Zugegeben, es ist natürlich verwirrend, dass eine Methode namens `chars` einen Stream von `int`-Werten zurückgibt, und Sie *könnten* das Programm korrigieren, indem Sie eine Typumwandlung einfügen, die den Aufruf der korrekten Überladung erzwingt:

```
"Hello world!".chars().forEach(x -> System.out.print((char) x));
```

Doch im Idealfall sollten Sie **auf die Verwendung von Streams zur Verarbeitung von char-Werten verzichten.**

Wenn Sie anfangen, mit Streams zu arbeiten, könnten Sie den Drang verspüren, alle Ihre Schleifen auf Streams umzustellen. Machen Sie das nicht. Selbst wenn es möglich wäre, würde es wahrscheinlich die Lesbarkeit und Wartbarkeit Ihres Codes beeinträchtigen. Als Grundregel sollten Sie sich merken, dass selbst mäßig komplexe Aufgaben am besten mit einer Kombination aus Streams und Iteration gelöst werden, so wie von dem obigen Programm `Anagrams` demonstriert. **Stellen Sie also bestehenden Code nur dort auf Streams um und verwenden Sie Streams nur dort in neuem Code, wo es sinnvoll ist.**

Wie in den Programmen dieses Themas gezeigt, drücken Stream-Pipelines wiederholte Berechnungen mithilfe von Funktionsobjekten (typischerweise Lambdas oder Methodenreferenzen) aus, während iterativer Code wiederholte Berechnungen mithilfe von Codeblöcken ausdrückt. Es gibt einige Dinge, die Sie mit Codeblöcken, aber nicht mit Funktionsobjekten machen können:

- Von einem Codeblock aus können Sie jede lokale Variable im Gültigkeitsbereich lesen oder ändern. Von einem Lambda aus können Sie nur Final- oder Quasi-Final-Variablen lesen [JLS 4.12.4]. Und Sie können keine lokalen Variablen ändern.

- Von einem Codeblock aus können Sie mit `return` aus der umschließenden Methode zurückkehren, eine umschließende Schleife mit `break` abbrechen oder mit `continue` fortsetzen oder eine geprüfte Ausnahme werfen, die von der Methode zum Werfen deklariert ist. Von einem Lambda-Ausdruck aus können Sie nichts dergleichen tun.

Wenn eine Berechnung am besten mithilfe dieser Techniken ausgedrückt wird, eignet sie sich wahrscheinlich nicht für die Umstellung auf Streams. Umgekehrt gibt es Dinge, die durch Streams sehr vereinfacht werden:

- Gleichmäßiges Umwandeln von Element-Sequenzen
- Filtern von Element-Sequenzen
- Element-Sequenzen in einer einzigen Operation kombinieren (zum Beispiel um sie aufzuaddieren, zu verketten oder ihr Minimum zu berechnen).
- Element-Sequenzen in einer Collection sammeln, möglicherweise gruppiert nach einem gemeinsamen Attribut.
- Element-Sequenzen nach einem Element durchsuchen, das einem bestimmten Kriterium entspricht.

Wenn eine Berechnung am besten mithilfe dieser Techniken ausgedrückt wird, ist sie ein guter Kandidat für Streams.

Eine Sache, die sich mit Streams nur schwer bewerkstelligen lässt, ist der simultane Zugriff auf korrespondierende Elemente aus mehreren Stufen einer Pipeline. Sobald Sie nämlich einen Wert auf einen anderen Wert abbilden, geht der ursprüngliche Wert verloren. Ein möglicher Workaround wäre, die Werte auf *Wertepaare* abzubilden, die den ursprünglichen und den neuen Wert enthalten. Aber das ist keine befriedigende Lösung. Besonders dann nicht, wenn die Wertepaar-Objekte für mehrere Stufen einer Pipeline benötigt werden. Der resultierende Code ist chaotisch und langatmig, was einem der Hauptziele für den Einsatz von Streams widerspricht. Ein besserer Workaround, falls umsetzbar, ist, die Zuordnung bei Bedarf zu invertieren.

Um dies zu demonstrieren, lassen Sie uns ein Programm schreiben, das die ersten zwanzig *Mersenne-Primzahlen* ausgibt. Zur Erinnerung: *Mersenne-Zahlen* sind Zahlen der Form $2^p - 1$. Wenn p eine Primzahl ist, ist die entsprechende Mersenne-Zahl möglicherweise ebenfalls eine Primzahl. Wenn ja, dann ist es eine Mersenne-Primzahl. Als ersten Stream in unserer Pipeline hätten wir gerne die Folge aller Primzahlen. Die folgende Methode liefert uns diesen unendlichen Stream. Der Code geht davon aus, dass ein statischer Import verwendet wurde, um unkompliziert auf die statischen Member von `BigInteger` zugreifen zu können:

```
static Stream<BigInteger> primes() {
    return Stream.iterate(TWO, BigInteger::nextProbablePrime);
}
```

Der Name der Methode (`primes`) ist ein Nomen im Plural, das die Elemente des Streams beschreibt. Diese Namenskonvention wird für alle Methoden, die

Streams zurückgeben, dringend empfohlen, da sie die Lesbarkeit von Stream-Pipelines verbessert. Die Methode verwendet die statische Factory-Methode `Stream.iterate`, die zwei Parameter entgegennimmt: das erste Element im Stream und eine Funktion, um das nächste Element im Stream aus dem vorherigen zu generieren. Das folgende Programm verwendet die Methode `primes()`, um die ersten zwanzig Mersenne-Primzahlen auszugeben:

```
public static void main(String[] args) {
    primes().map(p -> TWO.pow(p.intValueExact()).subtract(ONE))
        .filter(mersenne -> mersenne.isProbablePrime(50))
        .limit(20)
        .forEach(System.out::println);
}
```

Dieses Programm ist die direkte Umsetzung der obigen Prosabeschreibung: Es beginnt mit den Primzahlen, berechnet die entsprechenden Mersenne-Zahlen, filtert alle Nicht-Primzahlen heraus – die magische Zahl 50 steuert den probabilistischen Primzahlentest –, begrenzt den resultierenden Stream auf zwanzig Elemente und gibt das Ergebnis aus.

Als Nächstes möchten wir in der Ausgabe den einzelnen Mersenne-Primzahlen ihren Exponenten (p) voranstellen. Die Exponenten sind jedoch nur im Anfangsstream vorhanden, weswegen die Endoperation, die die Ergebnisse ausgibt, nicht mehr darauf zugreifen kann. Zum Glück ist es nicht schwer, die Exponenten durch Umkehr der Zuordnung der ersten Zwischenoperation aus den Mersenne-Zahlen zurückzurechnen. Der Exponent ist einfach die Anzahl der Bits in der Binärdarstellung, sodass die folgende Endoperation das gewünschte Ergebnis liefert:

```
.forEach(mp -> System.out.println(mp.bitLength() + ": " + mp));
```

Ob eine Aufgabe besser mit Streams oder durch Iteration zu lösen ist, lässt sich in vielen Fällen gar nicht so einfach entscheiden. Denken Sie zum Beispiel an die Initialisierung eines neuen Kartenspiels. Ausgangspunkt sei eine Klasse `Card`, eine unveränderliche Werteklasse, die einen Wert `Rank` und eine Farbe `Suit`, beides Aufzählungstypen, kapselt. Diese Aufgabe ist repräsentativ für alle Aufgaben, bei denen sämtliche Elementpaare berechnet werden müssen, die aus zwei Mengen gebildet werden können. Mathematiker nennen dies das *kartesische Produkt* der beiden Mengen. Die folgende iterative Implementierung mit einer geschachtelten for-each-Schleife dürfte Ihnen wohl vertraut sein:

```
// Iterative Berechnung des kartesischen Produkts
private static List<Card> newDeck() {
    List<Card> result = new ArrayList<>();
    for (Suit suit : Suit.values())
        for (Rank rank : Rank.values())
            result.add(new Card(suit, rank));
    return result;
}
```

Und hier ist eine Stream-basierte Implementierung, die die Zwischenoperation `flatMap` verwendet. Diese Operation bildet jedes Element in einem Stream auf einen Stream ab und verbindet dann alle diese neuen Streams zu einem einzigen Stream – oder »ebnet sie ein«, englisch *flatten*. Beachten Sie, dass diese Implementierung einen verschachtelten Lambda-Ausdruck enthält (siehe Fettdruck):

```
// Stream-basierte Berechnung des kartesischen Produkts
private static List<Card> newDeck() {
    return Stream.of(Suit.values())
        .flatMap(suit ->
            Stream.of(Rank.values())
                .map(rank -> new Card(suit, rank)))
        .collect(toList());
}
```

Welche der beiden Versionen von `newDeck` ist nun besser? Tatsächlich sind hier letzten Endes persönliche Vorlieben und die Umgebung entscheidend, in der Sie programmieren. Die erste Version ist einfacher und scheint näherliegend. Die meisten Java-Programmierer werden in der Lage sein, diesen Code zu verstehen und zu warten. Einige Programmierer werden sich dagegen mit der zweiten Stream-basierten Version wohler fühlen. Sie ist etwas prägnanter und ebenfalls nicht allzu schwer zu verstehen, wenn man sich mit Streams und funktionaler Programmierung einigermaßen auskennt. Wenn Sie sich nicht sicher sind, welche Version Sie bevorzugen sollen, ist die iterative Version wahrscheinlich die sicherste Wahl. Wenn Sie die Stream-Version bevorzugen und davon ausgehen können, dass andere Programmierer, die mit dem Code arbeiten werden, Ihre Präferenz teilen, dann sollten Sie die Stream-Version verwenden.

Zusammenfassend lässt sich sagen, dass einige Aufgaben am besten mit Streams, andere am besten durch Iteration und wieder andere am besten durch Kombination der beiden Ansätze gelöst werden. Es gibt keine festen, einfachen Regeln, welcher Ansatz für eine Aufgabe der richtige ist, lediglich einige nützliche Heuristiken. In vielen Fällen ist es klar, welcher Ansatz zu verwenden ist, in manchen Fällen nicht. **Wenn Sie sich nicht sicher sind, ob eine Aufgabe besser mit Streams oder durch Iteration zu lösen ist, versuchen Sie beides und sehen Sie, was besser funktioniert.**