HANSER

Leseprobe

Fritz Jobst

Programmieren in Java

ISBN (Buch): 978-3-446-44134-7

ISBN (E-Book): 978-3-446-44150-7

Weitere Informationen oder Bestellungen unter

http://www.hanser-fachbuch.de/978-3-446-44134-7

sowie im Buchhandel.

Inhalt

Vorwort

Liebe Leserin, lieber Leser,

in der objektorientierten Sprache Java programmieren wir seit 1996 für diverse Systeme. Jetzt können wir mit Java 8 für manche Probleme einfachere und prägnantere Programme im funktionalen Stil formulieren.

Wir fokussieren auf zentrale Techniken der Programmierung und nutzen dabei den Komfort und die Leistungsfähigkeit der Entwicklungsumgebung Eclipse gleich beim Start ins Java-Land in *Kapitel 1*. In *Kapitel 2* besprechen wir die Grundlagen der Steuerung von Abläufen in Programmen, der sog. prozeduralen Programmierung. Erst wenn wir Verzweigungen, Schleifen und Aufrufe von Funktionen beherrschen, können wir uns in *Kapitel 3* den Herausforderungen der Objektorientierung stellen. Dort lernen wir auch die Lambda-Ausdrücke als neue Möglichkeiten in Java 8 zur Implementierung funktionaler Schnittstellen kennen.

Ein einzelnes Objekt kommt selten alleine. Zur Verwaltung von Objekten dienen die sog. Collections. Praktiker wissen, dass man mit den sequentiellen und assoziativen Sammlungen die meisten alltäglichen Probleme der Verwaltung von Daten lösen kann, auf die wir uns im Buch konzentrieren. Java unterstützt uns bei der Programmierarbeit auch mit anderen Klassen, von denen wir in *Kapitel 4* eine kleine, praxisorientierte Auswahl vorstellen. Mit Java 8 können wir mit Streams die Iteration über diese Datenbestände dem Laufzeitsystem überlassen, wir spezifizieren nur, was getan werden soll.

Kapitel 5 führt in die Grundlagen der Verarbeitung von Daten auf externen Speichermedien ein. Dazu nutzen wir die neuen Möglichkeiten von Java 8 zur vereinfachten Programmierung. Zum Verständnis der Programmierung von parallelen Streams und grafischen Benutzungsoberflächen machen wir uns in *Kapitel 6* mit den Chancen und Risiken der nebenläufigen Programmierung vertraut.

In *Kapitel 7* führen wir die Grundkonzepte der Programmierung grafischer Benutzungsoberflächen ein. Wir lernen die Steuerelemente sowie die Möglichkeiten des Aufbaus kennen. Die Anbindung von Programmcode an Benutzungsoberflächen gelingt uns mit den Lambda-Ausdrücken von Java 8 kürzer und prägnanter als bisher.

Java ist als Programmiersprache für Anwendungen im Internet konzipiert. Deswegen können wir in *Kapitel 8* mit relativ geringem Aufwand Programme für verteilte Anwendungen erstellen.

In *Kapitel 9* lernen wir den Zugriff auf Datenbanken. Damit können wir unsere Objekte in Datenbanken ablegen und daraus wiedergewinnen. Mit den Basistechniken zur Speicherung bzw. des Lesens von Daten im XML-Format in *Kapitel 10* runden wir unseren Streifzug durch das Java-Land ab.

Wäre es nicht einfach schön, das eigene Android-Smartphone programmieren zu können? Es gibt zwar viele Apps, denn Android gewinnt immer höhere Anteile am Markt für Smartphones. Dennoch fasziniert es, Anwendungen für das eigene Gerät selbst zu schreiben. Mit dem Buch „Programmieren in Java“ erhalten Sie eine solide Grundlage in der Programmiersprache Java. Damit könnten Sie Apps für Android selbst entwickeln. Aber aller Anfang ist schwer, so auch der Start in Android. Deswegen bietet dieses Buch Ergänzungen zum Download als Hilfe zum Einstieg in die Entwicklung für Android an.

Kapitel 1 der Ergänzungen zu diesem Buch begleitet Sie auf dem Weg der Installation des Entwicklungssystems für Android. Damit schaffen wir die erste App für Android.

Im Gegensatz zum PC benötigen wir am Smartphone bereits für einfachste Anwendungen Grundkenntnisse im Umgang mit grafischen Benutzungsoberflächen. Kapitel 7 dieser Ergänzungen führt Sie in die Grundproblematik ein. Dabei müssen wir auch einige Besonderheiten bei Android-Smartphones beachten.

Meine Empfehlung zum Einstieg in die Programmierung für Android: Arbeiten Sie die Kapitel 1, 2, 3 und 4 des Buchs „Programmieren in Java“ durch. Danach können Sie über Kapitel 1 der Ergänzungen zu diesem Buch in die Entwicklung für Android in Kapitel 7 der Ergänzungen einsteigen.

Wie bei Fremdsprachen gilt auch bei Programmiersprachen: Man muss die Sprache sprechen, d. h. selbst programmieren. Deswegen enthält jedes Kapitel kleinere oder größere Aufgaben. Das Spektrum der Aufgaben reicht von elementaren Übungen bis zu kleinen Projektarbeiten. Die Ergänzungen für Android enthalten die analogen Aufgabenstellungen für Android-Apps. Lösungsvorschläge zu allen Aufgaben finden Sie über das Internetportal des Carl Hanser Verlags http://downloads.hanser.de. Von dort können Sie auch alle Programme im Buch in vollständiger Form herunterladen.

Mein besonderer Dank gilt Frau Brigitte Bauer-Schiewek für die aufmerksame Begleitung bei der Konzeption und Durchführung sowie Frau Irene Weilhart für ihre Präzision und Sorgfalt bei der Herstellung.

Viel Erfolg mit diesem Buch!

Regensburg, im November 2014

Fritz Jobst

4.3.3 Fallbeispiele – Anwendungsfälle

In diesem Abschnitt stellen wir mögliche Anwendungsfälle für Streams vor. Besonders nützlich zum Aufsammeln von Ergebnissen sind die Methoden der Klasse `Collectors` aus dem API von Java 8. Sie bietet nützliche Operationen zur Reduktion, wie das Aufsummieren der Elemente, das Zusammenfassen aller Ergebnisse in einer geeigneten `Collection` oder das Gruppieren nach bestimmten Kriterien. Hiermit lösen wir Standardprobleme wie:

- Zeichenketten aneinanderhängen
- Sortieren
- Gruppieren von Objekten nach diversen Kriterien
- Verarbeiten von Textdateien
- Berechnungen durchführen

4.3.3.1 Zeichenketten aneinanderhängen

Mit Hilfe der `joining`-Methoden der Klasse `java.util.stream.Collectors` in Tabelle 4.29 können wir Zeichenketten bequem zusammenhängen. In der ersten Variante fügen wir zwischen zwei Texten ein Komma ein, in der zweiten Variante fügen wir zusätzlich vor und nach dem Ergebnis noch Zeichenketten ein. In Variante 3 benützen wir Methoden der `StringBuilder`-Klasse, um den Prozess des Aufsammelns der Zeichenketten zu kontrollieren. Dazu müssen wir drei Referenzen zu Methoden angeben:

1. Einrichten eines Suppliers. Er richtet ein Objekt zum Aufsammeln ein.
2. Methode zum Anhängen eines Elements an ein Objekt zum Aufsammeln
3. Methode zum Aneinanderhängen zweier Objekte zum Aufsammeln

```
List<String> strings = Arrays.asList("a", "b", "c", "d");
String c1 = strings
    .stream()
    .collect(Collectors.joining(","));
System.out.println (c1);    // Ergebnis = a,b,c,d

String c2 = strings
    .stream()
    .collect(Collectors.joining(", ", "[", "]"));
System.out.println (c2);    // Ergebnis = [a, b, c, d]

String c3 = strings
    .stream()
    .collect(
    StringBuilder::new,     // a) Einrichten
    StringBuilder::append,  // b) Anhängen eines Strings
    StringBuilder::append)  // c) Zusammenhängen zweier StringBuilder
    .toString();
System.out.println (c3);    // Ergebnis = abcd
```

4.3.3.2 Sortieren

Zum Sortieren von Elementen, die die `Comparable`-Schnittstelle implementieren, dient die `sorted ()`-Methode für Streams. Im folgenden Beispiel übergeben wir einen `Comparator` als Lambda-Ausdruck, um in absteigender Reihenfolge zu sortieren.

```
List<String> strings = Arrays.asList("a", "b", "c", "d");
strings.stream()
    .sorted((a,b) -> b.compareTo(a))
    .forEach(s->System.out.printf("%s,", s));
```

Sortierte Ausgabe aller Einträge in einem Verzeichnis

Wenn wir alle Einträge vom Typ `File` eines Verzeichnisses ausgeben wollen, lesen wir die Katalogeinträge zunächst mit Hilfsmitteln aus Kapitel 5 ein (siehe auch Abschnitt 5.2.4). Als Ergebnis erhalten wir alle Einträge in einer Liste. Jetzt können wir die Einträge nach verschiedenen Kriterien sortieren. Weil die `Comparator`-Schnittstelle eine funktionale Schnittstelle ist, können wir den Vergleich mit einem Lambda-Ausdruck formulieren. Die Liste der Dateien liefert einen Stream, für den wir mit der `sorted(…)`-Methode einen je nach Kriterium sortierten Stream erhalten. Die Elemente geben wir dann aus.

Listing 4.24 Beispiel: Dateien eines Verzeichnisses nach diversen Kriterien sortiert ausgeben

```
public class DirectoryLister {
  private static final Comparator<File> NAMEORDER
     = (f1, f2) -> f1.getName().compareTo(f2.getName());

  private static final Comparator<File> SIZEORDER
     = (f1, f2) -> Long.valueOf(f1.length()).compareTo(f2.length());

  public DirectoryLister(String dirName) {
    File dir = new File(dirName);
    String fileNames[] = dir.list();
    List<File> files = new ArrayList<> (fileNames.length);

    for (String fileName : fileNames) {
      File temp = new File(dir, fileName);
      files.add (temp);
    }

    // Ausgabe nach Groesse geordnet (groesste Datei zuletzt)
    files.stream().sorted(SIZEORDER).forEach(System.out::println);
    System.out.println();
    // Ausgabe nach Name geordnet
    files.stream().sorted(NAMEORDER).forEach(System.out::println);
    System.out.println ();
  }

  public static void main (String[] args) {
    new DirectoryLister (".\\src");
  }
}
```

4.3.3.3 Gruppieren von Objekten nach diversen Kriterien

Zum Gruppieren von Objekten benötigt man ein Kriterium. Als Ergebnis erhält man für jeden Wert des Kriteriums eine u. U. leere Sammlung von Objekten, die diesem Kriterium genügen.

Gruppieren von Zahlen

Im ersten Beispiel wollen wir eine Folge von Zahlen in ungerade und gerade Zahlen gruppieren. Zum Test dividieren wir eine Zahl durch 2. Wenn der Rest bei der Division gleich 0 ist, haben wir eine gerade Zahl, ansonsten eine ungerade Zahl. Ein Lambda-Ausdruck liefert das Ergebnis, welches wir an die Methode `Collectors.groupingBy` übergeben. Als Ergebnis erhalten wir eine Map `zuordnung` von `Boolean` zu Listen von Zahlen. Wir starten eine Iteration über die Schlüssel und geben den Wert jedes Schlüssels sowie die ihm zugeordneten Zahlen aus.

Listing 4.25 Gruppieren von `Essen`-Objekten nach Typ

```
Map<Boolean, List<Integer>> zuordnung =
  Stream.of(1,2,3,4,5)
    .collect(Collectors.groupingBy(i -> i %2 == 0));

zuordnung.keySet()  // Menge der Schlüssel auslesen
  .stream()         // Stream aus Schlüsseln bilden
  .forEach(k -> {   // Für jedes Paar (Schlüssel Liste…)
    System.out.printf ("\n%s:", k.toString()); // Schlüssel ausgeben
    li.get(k)       // Liste holen
      .stream()     // Stream der einzelnen Zahlen bearbeiten
      .forEach(i -> System.out.printf ("%d,", i));
    });
```

Ergebnis

```
false:1,3,5,
true:2,4,
```

Gruppieren von Objekten

Im zweiten Beispiel wollen wir Objekte mit einem `enum`-Attribut nach diesem Attribut gruppieren. Als Objekte nehmen wir die `Essen`-Objekte aus Kapitel 7.

```
public enum EssenTyp {
  OBST, GEMUESE, FASTFOOD, SLOWFOOD, ETC;
}

public class Essen {
  private String   name;
  private String   kommentar;
  private EssenTyp essenTyp; // Kriterium zum Gruppieren: Alle GEMUESE etc…
  //.. Dazu Konstruktoren, get- und set-Methoden, siehe Kapitel 7
  // z.B. public EssenTyp getEssenTyp () { return essenTyp; }
}
    Stream<Essen> essen = Arrays.stream(Essen.getAlleEssen());
```

```
    Map <EssenTyp, List<Essen>> mapEssen =
        essen.collect(Collectors.groupingBy(e -> e.getEssenTyp()));
    // Für alle Schlüssel tue
    //   Schlüssel ausgeben, alle Werte zum Schlüssel ausgeben
    mapEssen.keySet()
        .stream()
        .forEach(key -> {
           System.out.printf("\nTyp = %s\n", key);
           mapEssen.get(key)
             .stream()
             .forEach (e -> System.out.printf("  Essen = %s\n", e));
        });
```

Ergebnis (abgekürzt)

```
Typ = GEMUESE
  Essen = Broccoli
  Essen = Karotte
  Essen = Mais
  Essen = Rettich

Typ = FASTFOOD
  Essen = Hamburger
  Essen = Pommes usw…
```

4.3.3.4 Verarbeiten von Daten in Textdateien

Wir wollen alle Worte einer Textdatei bearbeiten, z. B. alle Zahlen aufsummieren. Eine Textdatei besteht aus einzelnen Zeilen. Diese Folge von Zeilen können wir mit `java.nio.file.Files.lines` als Stream lesen. Jede dieser Zeilen enthält ihrerseits eine Folge von Worten. Wir haben es hier mit einem Stream (aus Zeilen) zu tun, wobei jedes Element des Streams seinerseits wieder aus Streams (aus Worten) besteht. Solche „Streams aus Streams“ mit einer Art 1:n-Beziehung können wir mit der `flatMap`-Operation zu einem Stream (aus Worten) zusammenhängen und diesen Stream durchlaufen. In Listing 4.26 wandeln wir dies in einen `IntStream` um, den wir mit `sum()` aufaddieren können.

Listing 4.26 Addieren aller Zahlen einer Textdatei mit ganzen Zahlen

```
public class ZahlenAddieren {

  public static void main(String[] args) {
    String dateiName = "zahlen.txt";
    try {
      System.out.println (
          Files.lines(Paths.get(dateiName))           // Stream aus Zeilen
          .flatMap(line->Stream.of(line.split(" +")))  // Stream aus allen Texten
          .mapToInt(s -> Integer.parseInt(s))         // IntStream
          .sum ());                                   // Summe berechnen
    } catch (IOException e) {
      e.printStackTrace(System.err);
    }
  }

}
```

4.3.3.5 Berechnungen mit Zwischenergebnissen durchführen

Für die Berechnung von Summen aus **int** ganzen Zahlen bietet die Klasse `IntStream` die Methode `sum()`, für Durchschnitte die Methode `average()`. Will man Summen aus Integer-Objekten berechnen, steht keine der o. a. Methoden zur Verfügung. Für Summen kann man die `reduce(…)`-Anweisung benützen:

```
int summe = Stream.of (3, 5, usw… 3 ).reduce(0, (l, r) -> l + r);
```

Dies hilft bei der Berechnung des Durchschnitts nicht weiter, denn zur Berechnung des Durchschnitts benötigt man auch die Anzahl der Elemente. Diese könnte man separat mit der `count()`-Methode bestimmen. Wenn man die Folge der Elemente nicht zweimal durchlaufen will, reicht eine Schnittstelle nicht aus, wir benötigen eine Klasse, da die Attribute der Klasse Daten aufnehmen können. Dieses Verfahren setzen wir immer dann ein, wenn wir bei der Verarbeitung Daten benötigen, z. B. in Form übergebener Parameter.

Wir verwenden in Analogie zu einem Beispiel aus dem Java-API [API14] eine Klasse Summierer, die sowohl Summen bildet, als auch die Anzahl der Elemente mitzählt. Damit können wir in dieser Klasse eine Methode zur Berechnung des Durchschnitts angeben.

Wir berechnen den Durchschnitt in einer `Optional`-Klasse, da wir für den Fall einer leeren Sammlung von Elementen eine Division durch 0 vermeiden wollen. Siehe hierzu auch Aufgabe 4.3.

Listing 4.27 Berechnung von Durchschnittswerten

```
public class Zwischenergebnisse {
  public static class Summierer implements Consumer<Integer> {

    private int total = 0;
    private int count = 0;

    public Optional<Double> average() {
      return Optional.ofNullable(count > 0 ? ((double) total) / count : null);
    }

    @Override
    public void accept(Integer i) {
      total += i;
      count++;
    }

    public void combine(Summierer other) {
      total += other.total;
      count += other.count;
    }
  }
  public static void main(String[] args) {
    Optional<Double> d = Stream.of (3, 5, usw… 3 )
      .collect(Summierer::new,
               Summierer::accept,
               Summierer::combine)
      .average();
    System.out.println (d);
  }
}
```

Zusammenfassung

Java stellt im Package `java.util` Klassen zur Verfügung, die dem Programmierer die Entwicklung eigener Routinen für immer wieder zu lösende Probleme ersparen. Das `Collection`-Framework stellt ein leistungsfähiges System zur Aufbewahrung von Objekten zur Verfügung. Insbesondere kann man sequenziell anfallende Daten mit den Implementierungen der `List`-Schnittstelle verwalten. Wünscht man Sammlungen, die frei von Duplikaten sind, so kann man mit den Implementierungen der `Set`-Schnittstelle arbeiten. Für assoziative Zugriffe auf Daten bieten sich die Implementierungen der `Map`-Schnittstelle an.

Wir können alle Elemente von Sammlungen mit Hilfe der `Iterator`-Schnittstelle durchlaufen. Die erweiterte Form der `for`-Schleife vereinfacht dies in vielen Fällen. Ab Java 8 können wir mit Streams Grundaufgaben der Programmierung im funktionalen Stil effizient lösen: Sortieren, Gruppieren, elementare Operationen wie Saldieren usw.

4.4 Aufgaben

Aufgabe 4.1

Benutzen Sie die Klasse zur Verwaltung von Mitarbeitern aus Abschnitt 3.2.1. Verwalten Sie Personen mit einer generischen Liste. Implementieren Sie Methoden zum Hinzufügen von Personen und zum Ausgeben aller Personen.

Aufgabe 4.2

Benutzen Sie die Klasse zur Verwaltung von Personen aus Abschnitt 3.2.1. Verwalten Sie die Personen mit einer `HashMap`. Implementieren Sie Methoden zum Hinzufügen von Personen und zum Suchen der Personen durch Angabe des Namens.

Aufgabe 4.3: zu IntStream

In dieser Aufgabe gehen wir von ganzen Zahlen im **`int`**`[]`-Array aus. Bitte bearbeiten Sie die Aufgabe mit `IntStream`, nicht mit `Stream<Integer>` wie in der nächsten Aufgabe.

```
int[] zahlen = {3, 5, 1, 3, 7, 29, 33, 49, 5, 1, 1, 2, 3};
```

1. Berechnen Sie die Summe aller Zahlen.
2. Berechnen Sie den Durchschnitt aller Zahlen.
3. Bestimmen Sie das Maximum aller Zahlen.
4. Geben Sie ein Histogramm der Zahlen an. Gruppieren Sie dazu die Zahlen nach Werten und ermitteln Sie für jeden Wert die Anzahl der Zahlen mit diesem Wert. Siehe Abschnitt 4.3.3.3.
5. Geben Sie die Zahlen in der Form `[3, 5, 3, … 1, 1, 2, 3]` aus. Siehe Abschnitt 4.3.3.1.

Aufgabe 4.4 zu Stream<Integer>

In dieser Aufgabe gehen wir von Objekten in einem Array `Integer[]` aus und arbeiten mit `Stream<Integer>`.

```
Integer[] zahlen = { 3, 5, 1, 3, 7, 29, 33, 49, 5, 1, 1, 2, 3 };
```

1. Berechnen Sie die Summe aller Zahlen.
2. Berechnen Sie den Durchschnitt aller Zahlen.
3. Bestimmen Sie das Maximum aller Zahlen.
4. Geben Sie ein Histogramm der Zahlen an. Gruppieren Sie dazu die Zahlen nach Werten und ermitteln Sie für jeden Wert die Anzahl der Zahlen mit diesem Wert. Siehe Abschnitt 4.3.3.3.
5. Geben Sie die Zahlen in der Form `[3, 5, 3, … 1, 1, 2, 3]` aus. Siehe Abschnitt 4.3.3.1.

Aufgabe 4.5: Primzahlen mit Streams berechnen

Gegeben ist die folgende recht ineffiziente Primzahl-Testmethode für Zahlen >= 2. Diese Methode ist ein Prädikat für ganze Zahlen im Sinne von Java 8.

```
static boolean isPrime(int zahl) {
  int teiler = 2;
  while (zahl % teiler != 0)
    teiler++;
  if (zahl == teiler)
    return true;
  return false;
}
```

1. Geben Sie die ersten 100 Primzahlen aus
2. Geben Sie die ersten 1000 Primzahlen aus. Nützen Sie alle Kerne Ihres Rechners zur Berechnung!
3. Aufgabe (optional)
   Wenn Ihnen die obige Methode zur Ermittlung von Primzahlen zu primitiv ist, geben Sie eine effizientere Variante an.

Aufgabe 4.6

In Listing 4.25 aus Abschnitt 4.3.3.3 zeigen wir, wie man einzelne Essen nach dem Typ gruppiert. Ziel dieser Aufgabe ist es, die o. a. Objekte nach dem Anfangsbuchstaben ihres Namens zu gruppieren.

Aufgabe 4.7

Gegeben ist eine Datei mit Fußballern in der folgenden Form:

Listing 4.28 Die Nationalmannschaft im Herren-Fußball

```
Trikot;Vorname;Name;Geburtstag;Verein;Spiele;Tore;Rolle;
1;Manuel;Neuer;27.03.1986;FC Bayern München;52;0;TORWART;
15;Erik;Durm;12.05.1992;Borussia Dortmund;1;0;VERTEIDIGUNG;
7;Bastian;Schweinsteiger;01.08.1984;FC Bayern München;108;23;MITTELFELD;
11;Miroslav;Klose;09.06.1978;Lazio Rom;137;71;ANGRIFF;
```

Jede Zeile enthält die Daten für einen Fußballer. Die einzelnen Werte sind durch einen Strichpunkt getrennt. Für Fußballer gibt es die im o. a. Beispiel definierten Rollen und keine weiteren. Sie finden die komplette Datei mit allen Fußballern einer Mannschaft bei den Unterlagen zu diesem Buch.

1. Entwickeln Sie eine Klasse `Fussballer` mit allen acht in der ersten Zeile genannten Attributen. Definieren Sie für die Rolle eine geeignete **`enum`**-Klasse.
2. Geben Sie einen Konstruktor für diese Klasse aus einer Zeile an. (Hinweis: Die `split()`-Methode dient zum Aufteilen von Zeichenketten.)
3. Geben Sie eine Klasse `Mannschaft` an. Diese Klasse soll eine Liste aller Fußballer enthalten: `List<Fussballer> mannschaft`, die sog. Mannschaft.
4. Lesen Sie die Mannschaft mit Hilfe von b) und einer Methode der Klasse `Files` ein. (Hinweis: `Files.readAllLines (Paths.get("fussball.txt"), Charset.defaultCharset())` liefert eine Liste mit den Zeilen einer Textdatei).
5. Geben Sie die Namen der Spieler der Mannschaft aus.
6. Geben Sie die Namen der zehn Spieler mit den meisten Toren aus.
7. Geben Sie die Namen der Fußballspieler gruppiert nach Anfangsbuchstaben in der folgenden Form aus:

```
B: {Boateng}
D: {Durm,Draxler}
G: {Ginter,Großkreutz,Götze}
… usw.
W: {Weidenfeller}
Z: {Zieler}
```

8. Ermitteln Sie per Java-Programm, welcher Verein die meisten Spieler für die Nationalelf abgestellt hat.

Hinweise

Für die Teilaufgaben 7. und 8. empfiehlt sich eine Gruppierung nach Anfangsbuchstaben der Namen bzw. nach den Vereinen der Spieler.

Index

C

D

E

F

G

H

I

J

K

L

M

N

O

P

T

U

V

W

X

Y

Z