WIF-Objekorientiertes Programmieren (WIF-OOP)

10 - Datenverarbeitung

Fakultät Informatik, TH Rosenheim

Arbeiten mit Datenstrukturen

Wir haben in den vergangenen Wochen die wichtigsten Datenstrukturen und Sortieralgorithmen kennengelernt. Die Beispiele dazu waren aber meistens knapp und eher abstrakt – in den wenigsten Fällen aber wirklich anschaulich.

Heute wollen wir üben, das Gelernte praxisnah anzuwenden, in dem wir exemplarisch einige typische Problemstellungen in der Datenverarbeitung durchgehen. Als Datenbasis nehmen wir die Bundesligaergebnisse der Saison 2017/2018 (Quelle: fussballdaten.de, Spiele, Vereine). Es liegen Daten aus der 1. (18 Vereine), 2. (18 Vereine) und 3. Bundesliga (20 Vereine) vor, aus der 1. und 2. Liga bis einschließlich des 32. Spieltags und aus der 3. Liga bis einschließlich des 36. Spieltags.

Spiel_ID	Spieltag	Datum	Uhrzeit	Heim	Gast	Tore_Heim	Tore_Gast
1	1	2017-08-18	20:30:00	1	5	3	1
2	1	2017-08-19	15:30:00	7	12	1	0

V_ID	Name	Liga
1	FC Bayern München	1
2	FC Schalke 04	1

Man sieht, dass die Vereine Heim und Gast in der Spieltabelle als *foreign key* auf die Vereinstabelle aufzulösen sind. Für unsere heutigen Beispiele können wir die Fabrikmethode Bundesliga.loadFromResource() verwenden, um die Daten aus den beiliegenden CSV Dateien einzulesen.

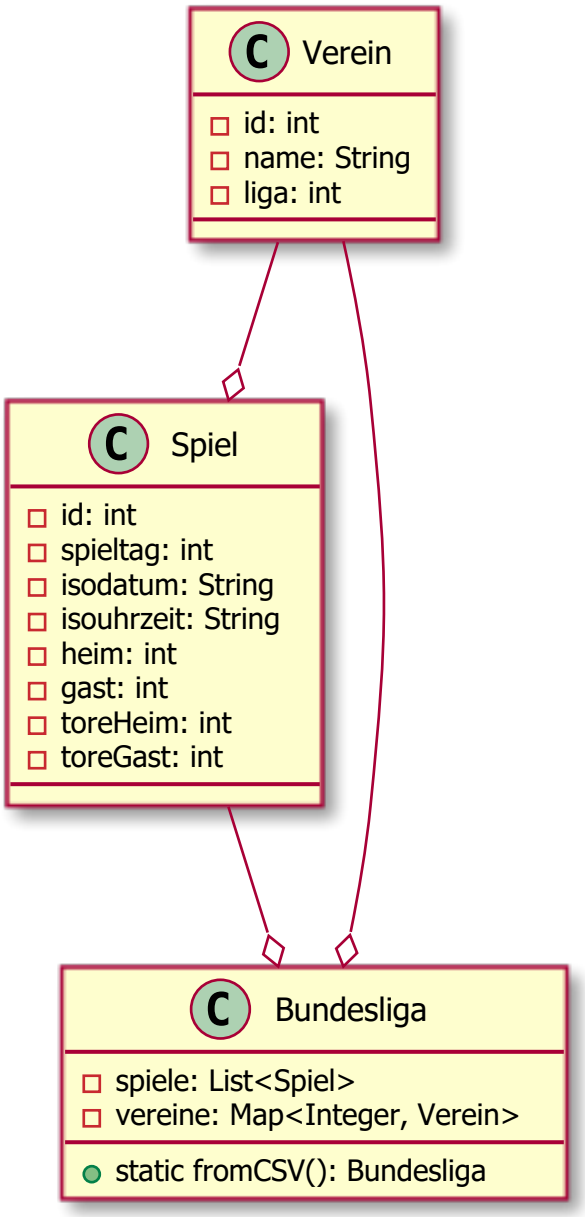

Figure 1: Datenmodell

Grundoperationen

Datenverarbeitung beruht im Wesentlichen auf vier Grundoperationen:

- **Sortieren**, also die Daten in eine gewünschte Reihenfolge bringen;
- **Filtern**, also das Entfernen gewisser Daten, bzw. das Behalten von nur gewissen Daten;
- **Abbilden**, also Daten von einem Format auf ein anderes umzurechnen;
- **Reduzieren**, also Daten zusammenzufassen.

Sortieren

Oft ist die Reihenfolge, in der man Daten zur Verfügung gestellt bekommt, nicht die Reihenfolge, in welcher man diese darstellen oder weiterverarbeiten möchte.

In unserem Beispiel sind die Vereine nach aktuellem Tabellenplatz und nach Liga sortiert. Möchte man aber die Liste aufsteigend nach Vereinsnamen sortiert haben, so muss man diese Liste entsprechend *sortieren* (siehe Kapitel 9: Sortieren).

```
1  Bundesliga b = Bundesliga.loadFromResource();
2
3  // neue Liste erstellen, alle Vereine hinzufügen
4  List<Verein> nachName = new LinkedList<>(b.vereine.values());
5
6  // nach Vereinsnamen sortieren
7  nachName.sort(new Comparator<Verein>() {
8      @Override
9      public int compare(Verein o1, Verein o2) {
10         return o1.getName().compareTo(o2.getName());
11     }
12 });
```

Möchte man erst nach Liga und dann nach Namen sortieren, so muss man einen Comperator schreiben, der dieses Verhalten erfüllt:

```
1  Bundesliga b = Bundesliga.loadFromResource();
2
3  List<Verein> nachLigaName = new LinkedList<>(b.vereine.values());
4  nachLigaName.sort(new Comparator<Verein>() {
5      @Override
6      public int compare(Verein o1, Verein o2) {
7          // erst wenn Ligen gleich sind, dann nach Name!
8          if (o1.getLiga() == o2.getLiga())
9              return o1.getName().compareTo(o2.getName());
10         else
11             return Integer.compare(o1.getLiga(), o2.getLiga());
12     }
```

```
13 });
```

Filtern

Je nach Anwendung kann es aber gut sein, dass wir garnicht an allen Vereinen interessiert sind, sondern nur an den Vereinen der 2. Liga.

```
1  Bundesliga b = Bundesliga.loadFromResource();
2
3  List<Verein> zweiteLiga = new LinkedList<>();
4  for (Verein v : b.vereine.values()) {
5      if (v.getLiga() == 2)
6          zweiteLiga.add(v);
7  }
8
9  zweiteLiga.sort(new Comparator<Verein>() {
10     @Override
11     public int compare(Verein o1, Verein o2) {
12         return o1.getName().compareTo(o2.getName());
13     }
14 });
```

Man sieht hier auch: Es ist sinnvoll *vor* dem Sortieren zu filtern, da man sonst Daten sortiert, an denen man nicht interessiert ist.

Abbilden

Wir werfen nun einen Blick auf die Spieltabelle.

Spiel_ID	Spieltag	Datum	Uhrzeit	Heim	Gast	Tore_Heim	Tore_Gast
1	1	2017-08-18	20:30:00	1	5	3	1
2	1	2017-08-19	15:30:00	7	12	1	0

Jeder Eintrag (Klasse Spiel) hat also die vollständige Spielinformation. Wenn wir nun aber nur an den *Spielpaarungen* interessiert sind, also an welchem Datum welche Mannschaften gegeneinander spielen, so müssen wir ein Spiel auf ein Tripel abbilden, welches aus Datum sowie den Namen von jeweils Heim- und Gastverein besteht. Wir wollen also eine List<Spiel> in eine List<Triple<String, String, String>> abbilden.

```
1  Bundesliga b = Bundesliga.loadFromResource();
2
3  // neue Liste mit Zieldatentyp
4  List<Triple<String, String, String>> paarungen = new LinkedList<>();
5  for (Spiel s : b.spiele) {
```

```
6       // Verwende Vereinstabelle um ID in Verein aufzulösen
7       Verein heim = b.vereine.get(s.getHeim());
8       Verein gast = b.vereine.get(s.getGast());
9
10      // Erstelle neues Triple aus Datum sowie Vereinsnamen
11      paarungen.add(Triple.of(s.getDatum(), heim.getName(), gast.getName()));
12  }
```

Die Klassen org.apache.commons.lang3.tuple.Pair und org.apache.commons.lang3.tuple.Triple sind generische Klassen für Paare und Tripel, welche mit den Fabrikmethoden Pair.of(...) sowie Triple.of(...) einfach instanziiert werden können.

Reduzieren

Abbilden bedeutet also immer das Umwandeln einer Liste in eine andere Liste. *Reduzieren* bedeutet, eine Liste von Werten auf einen einzigen Wert zu reduzieren. Ein einfaches Beispiel wäre das Aufsummieren von Werten in einem Array:

```
1  int[] a = {1, 2, 3};
2  int summe = 0;
3  for (int v : a)
4      summe += a;
```

In unserem Bundesligabeispiel wäre eine ähnliche Fragestellung: Wie viele Tore wurden insgesamt geschossen?

```
1  Bundesliga b = Bundesliga.loadFromResource();
2
3  int tore = 0;
4  for (Spiel s : b.spiele) {
5      tore = tore + s.getToreGast() + s.getToreHeim();
6  }
7
8  System.out.println("Es fielen insgesamt " + tore + " Tore in " + b.spiele.size() + "
       Spielen");
9  // "Es fielen insgesamt 1741 Tore in 714 Spielen"
```

Beispiele

Wir wollen nun versuchen, für die folgenden Fragestellungen die entsprechende Datenverarbeitung zu programmieren. Jeder dieser Datenverarbeitungsflüsse (engl. *pipelines*) soll dabei je nach Fragestellung Sortieren, Filtern, Abbilden und/oder Reduzieren.

1. Torstatistiken.
    1. Wie viele Tore fallen durchschnittlich in jedem Spiel?
    2. Wie viele Tore fallen durchschnittlich in einem Spiel der 1. Liga?
    3. Wie viele Tore fallen durchschnittlich an einem Spieltag der 2. Liga?
    4. Stimmt es, dass in den Nachmittagsspielen (15:30:00) im Schnitt mehr Tore fallen, wie in den Abendspielen?
    5. Stimmt es, dass Vereine der 3. Liga zuhause im Schnitt mehr Tore schießen als auswärts?
2. Vereine.
    1. Wie viele Tore hat der FC Bayern München (Verein 1) erzielt?
    2. Wie viele Tore hat der FC Schalke 04 (Verein 2) kassiert?
    3. Wie viele Punkte hat der 1. FC Nürnberg (Verein 20)? Ein Sieg zählt 3 Punkte, ein Unentschieden 1, eine Niederlage 0 Punkte.
    4. Was ist das Torverhältnis des VfL Bochum (Verein 26), also die Rate von erzielten zu kassierten Toren?
    5. Welche drei Vereine haben die meisten Tore zuhause geschossen, und wie viele?
    6. Welcher Verein hat die wenigsten Tore auswärts geschossen, und wie viele?
3. 1. Liga.
    2. Wie ist der aktuelle Tabellenstand? Die Tabelle wird als Vereinsname, gefolgt von Punkten und Torverhältnis definiert.
    3. Wie ist der Tabellenstand nach dem 10. Spieltag?
    4. Wie ist der Tabellenplatzverlauf des Hamburger SV (Verein 18) über alle 32 Spieltage?
    5. Wer hat die Rote Laterne in jeweils der 1., 2. und 3. Liga?
    6. Wie hat sich die Tabellenführung bis zum 32. Spieltag entwickelt? Die Liste soll nicht einen Verein pro Spieltag haben, sondern immer nur wenn ein neuer Verein die Führung übernimmt (d.h. jeder Eintrag ist von dem vorherigen verschieden).

Für die systematische Bearbeitung dieser Fragestellungen sollten Sie für jede dieser Fragestellungen zuerst die folgenden Fragen beantworten:

1. Was sind die **Eingabedaten**, und in welcher Datenstruktur liegen diese vor?
2. Was sind die **Ausgabedaten**, und in welcher Datenstruktur bzw. Form sollen diese vorliegen?
3. Welche Operationen (sortieren, filtern, abbilden, reduzieren) sind nötig, und in welcher Reihenfolge? Was sind die Zwischenprodukte?

Verwenden Sie die Klasse `Bundesliga` sowie die Fabrikmethode `Bundesliga.loadFromResource()`; die Attribute `List<Spiel> spiele` und `Map<Integer, Verein> vereine` sind öffentlich sichtbar (wie bereits den obigen Beispielen zu entnehmen).

Die Musterlösungen zu obigen Aufgaben finden Sie in der Klasse ch10.AnalysenTest sowie unten an dieses Kapitel angehängt.

Verallgemeinerung der Operationen

> Hinweis: Die folgenden Verallgemeinerungen sind zwar im Prinzip nicht klausurrelevant, machen die Datenverarbeitung aber klarer und strukturierter. Sie stellen auch eine gute Überleitung zu den ebenso nicht klausurrelevanten Streams (java.util.Stream) dar, welche die Datenverarbeitung aber ebenso deutlich übersichtlicher und damit einfacher machen.

Man sieht, dass bei den oben beschriebenen Operationen dir Iteration, also der *Besuch* der Elemente, und die eigentliche Logik, also die *Verarbeitung* der Elemente, verquickt sind. Das heisst aber auch, der Code zur Iteration immer wieder geschrieben werden, obwohl er immer gleich ist. Um die eigentliche Verarbeitung der Daten übersichtlicher zu gestalten, *trennt* man nun Besuch und Verarbeitung.

Iteration als Terminale Operation

Beginnen wir mit der einfachsten Version der Iteration. Angenommen man möchte jeden Verein auf System.out ausgeben, so haben wir bisher *Iteration* und *Verabeitung* gemeinsam programmiert, z.B. mit der for−each Schleife:

```
1 for (Verein v : b.vereine.values()) // Iteration
2     System.out.println(v); // Verarbeitung von `v`
```

Die Verabeitung (hier: System.out.println) ist also eine Methode, welche genau ein Argument (hier vom Typ Verein) entgegen nimmt, und keinen Rückgabetyp hat. So etwas bezeichnet man als *Consumer* (Verbraucher), und ist in Java als Schnittstelle verfügbar:

```
1 interface Consumer<T> {
2     void accept(T t);
3 }
```

Mochte man nun die Verabeitung herausnehmen, so könnte man das zunächst wie folgt realisieren:

```
1 // Verarbeitung
2 Consumer<Verein> cons = new Consumer<Verein>() {
3     @Override
4     public void accept(Verein v) {
5         System.out.println(v);
6     }
```

```
};

// Iteration
for (Verein v : b.vereine.values())
    cons.accept(v);
```

Weiterhin kann man nun die Iterationslogik auslagern:

```
static <T> void fuerJedes(Collection<T> coll, Consumer<T> cons) {
    for (T t : coll)
        cons.accept(t);
}
```

Wir setzen die Bausteine zusammen:

```
// vorher: iterieren und verarbeiten in einem
for (Verein v : b.vereine.values())
    System.out.println(v);

// nacher: nur Verarbeitungslogik, kein Iterationscode
fuerJedes(b.vereine.values(), new Consumer<Verein>() {
        public void accept(Verein v) {
            System.out.println(v);
        }
    });
```

> Genau genommen ist die zweite Version nun zwei Zeilen länger, wir sehen aber im Abschnitt Lambdaausdrücke wie man anonyme innere Klassen kürzer schreiben kann.

Sortieren

Die allgemeine Form des Sortierens mit Hilfe von Comparable<T> bzw. Comparator<T> wurde bereits ausführlich im Kapitel 9: Sortieren besprochen.

Filtern

Beim Filtern fällt auf, dass zunächst iteriert wird, und dann nach einem bestimmten Bedingung in eine neue Liste eingefügt wird.

```
List<Verein> zweiteLiga = new LinkedList<>();
for (Verein v : b.vereine.values()) {
    // Bedingung prüfen...
    if (v.getLiga() == 2) {
        zweiteLiga.add(v); // hinzufügen
    }
```

```
}
```

Ähnlich zum Comparator<T>, welcher den Vergleich beim Sortieren abstrahiert, kann man hier die Bedingung mit einem Predicate<T> abstrahieren:

```
interface Predicate<T> {
    boolean test(T t);
}
```

```
static <T> List<T> filtern(Collection<T> liste, Predicate<T> pred) {
    // neue Liste erstellen
    List<T> gefiltert = new LinkedList<>();

    // iterieren...
    for (T v : liste) {
        // Bedingung prüfen...
        if (pred.test(v))
            gefiltert.add(v);
    }

    return gefiltert;
}
```

Entsprechend kürzer wird die Anwendung:

```
List<Verein> zweiteLiga = filtern(b.vereine.values(), new Predicate<Verein>() {
        public boolean test(Verein v) {
            return v.getLiga() == 2;
        }
    });
```

Abbilden

Beim Abbilden erkennen wir, dass zwar sowohl Ein- als auch Ausgabe Listen sind, allerdings sind die Datentypen i.d.R. verschieden!

```
// neue Liste mit Zieldatentyp
List<Triple<String, String, String>> paarungen = new LinkedList<>();
for (Spiel s : b.spiele) {
    // Verwende Vereinstabelle um ID in Verein aufzulösen
    Verein heim = b.vereine.get(s.getHeim());
    Verein gast = b.vereine.get(s.getGast());

    // Erstelle neues Triple aus Datum sowie Vereinsnamen
    paarungen.add(Triple.of(s.getDatum(), heim.getName(), gast.getName()));
```

```
10 }
```

Hier wird eine List<Spiel> auf eine List<Triple<String, String, String>> abgebildet. Es wird wie beim Filtern die gesamte Liste durchlaufen, um dann für jeden Eintrag einen neuen Eintrag in der Zielliste zu erstellen. Diese Operation kann man als eine Methode generalisieren, welche ein Argument eines Typs T entgegennimmt und ein Objekt vom Typ R (*return*) zurückgibt. Analog zum Comparator und Predicate wird diese Methode im Rahmen eines Interfaces übergeben:

```
1 interface Function<T, R> {
2     R apply(T t);
3 }
```

```
1 static <T, R> List<R> abbilden(List<T> liste, Function<T, R> func) {
2     List<R> abgebildet = new LinkedList<>();
3
4     for (T v : liste)
5         abgebildet.add(func.apply(v));
6
7     return abgebildet;
8 }
```

Auch hier die Anwendung:

```
1 List<Triple<String, String, String>> paarungen = abbilden(b.spiele, new
      Function<Spiel, Triple<String, String, String>>() {
2         public Triple<String, String, String> apply(Spiel spiel) {
3             Verein heim = b.vereine.get(spiel.getHeim());
4             Verein gast = b.vereine.get(spiel.getGast());
5             return Triple.of(spiel.getDatum(), heim.getName(), gast.getName());
6         }
7     });
```

Reduzieren

Für die Abstraktion der Reduktion bleiben wir zunächst am einfachen Beispiel:

```
1 List<Integer> a = Arrays.asList(1, 2, 3);
2 Integer summe = 0;
3 for (Integer v : a)
4     summe = summe + a;
```

Es wird also ein Array *reduziert*, indem ein aktueller Zwischenwert mit dem

nächsten Element aus dem Array addiert wird. Die Addition ist dabei ein *binärer Operator*, da er zwei Operatoren entgegennimmt (summe und a) um das Ergebnis zu berechnen. Abstrahiert man diese Operation, so kann man die Reduktion allgemein fassen:

```
interface BinaryOperator<T> {
    T apply(T a, T b);
}
```

```
static <T> T reduzieren(List<T> liste, T identity, BinaryOperator<T> op) {
    T a = identity;
    for (T t : liste)
        a = op.apply(a, t);
    return a;
}
```

```
List<Integer> a = Arrays.asList(1, 2, 3);

// Reduzieren, mit "Null"-Element 0 und Addition als Reduzierer
Integer summe = reduzieren(a, 0, new BinaryOperator<Integer>() {
    public Integer apply(Integer a, Integer b) {
        return a + b;
    }
});
```

Im Eingangsbeispiel hatten wir eine Liste von Spielen auf einen Integerwert (die Gesamtsumme der Tore abgebildet). Dazu müssen wir die reduzieren Methode etwas genauer spezifizieren, so dass beim eigentlichen reduzieren die Datenelemente verwendet, aber ein anderer Datentyp zurückgegeben werden kann:

```
interface BiFunction<A, B, C> {
    C apply(A a, B b);
}
```

```
static <T, U> U reduzieren(List<T> liste, U identity, BiFunction<U, T, U> op) {
    U a = identity;

    for (T t : liste)
        a = op.apply(a, t);

    return a;

}
```

Und die Anwendung:

```
1 // Reduzieren mit 0 als "Null"-Element und
2 // einem Reduzierer, welcher den aktuellen Torstand mit den Heim- und Gasttoren
      addiert.
3 Integer tore = Abstraktion.reduzieren(b.spiele, 0, new BiFunction<Integer, Spiel,
      Integer>() {
4         @Override
5         public Integer apply(Integer integer, Spiel spiel) {
6             return integer + spiel.getToreGast() + spiel.getToreHeim();
7         }
8     });
```

Besondere Reduktionen

Es gibt nun einige Algorithmen, die im Kern Reduktionen sind, Ihnen aber vielleicht noch nicht als solche aufgefallen sind.

Minimum bzw. Maximum

Das Minimum (Maximum) in einer Liste ist ein einziger Wert, man *reduziert* also eine Liste auf ihr Minimum. Als neutrales Element wird hier z.B. das erste Element verwendet, der binäre Operator zur Reduktion ist eine Minimum- bzw. Maximumauswahl.

```
1 List<Integer> li = Arrays.asList(3, 5, 2, 1, 6, 7);
2
3 int min = reduzieren(li, li.get(0), new BinaryOperator<Integer> () {
4     public Integer apply(Integer a, Integer b) {
5         return Integer.min(a, b);
6     }
7 });
8 // "1"
```

Duplikatentfernung

Möchte man (konsekutive) Duplikate entfernen, so ist das eine Reduktion von einer Liste auf eine *kürzere* Liste. Das neutrale Element ist hier die leere Liste, der binäre Operator fügt das nächste Element nur dann in die Liste ein, sofern das Element nicht bereits enthalten ist.

```
1 List<Integer> li = Arrays.asList(4, 2, 4, 5, 5, 1, 2, 3);
2
3 List<Integer> duplikatfrei = reduzieren(li, new LinkedList<Integer>(),
4     new BiFunction<LinkedList<Integer>, Integer, LinkedList<Integer>> op) {
5         public LinkedList<Integer> apply(LinkedList<Integer> l, Integer i) {
```

```
6                   if (!l.contains(i))
7                       l.add(i);
8               }
9           });
10  // "[4, 2, 5, 1, 3]"
11
12
13  // nur keine konsekutiven Duplikate
14  List<Integer> konseqdupfrei = reduzieren(li, new LinkedList<Integer>(),
15          new BiFunction<LinkedList<Integer>, Integer, LinkedList<Integer>> op) {
16              public LinkedList<Integer> apply(LinkedList<Integer> l, Integer i) {
17                  // wenn die Liste bisher nicht leer ist,
18                  // oder das vorherige Element anders war
19                  if (l.size() == 0 || !l.getLast().equals(i))
20                      l.add(i);
21              }
22          });
23  // "[4, 2, 4, 5, 1, 2, 3]" (nur "5" nach "5" entfernt)
```

Lambdaausdrücke

Nicht klausurrelevant.

Die obigen Beispiele sind auf Grund der vielen anonymen inneren Klassen recht unübersichtlich, und enthalten viel "sinnbefreiten" Code. Es fällt insbesondere auf, dass alle der obigen Interfaces nur genau eine Methode vorschreiben:

```
1  interface Consumer<T> {
2      void accept(T t);
3  }
```

```
1  interface Predicate<T> {
2      boolean test(T t);
3  }
```

```
1  interface Function<T, R> {
2      R apply(T t);
3  }
```

```
1  interface BinaryOperator<T> {
2      T apply(T a, T b);
3  }
```

```
interface BiFunction<A, B, C> {
    C apply(A a, B b);
}
```

Diese wurden bisher immer als anonyme innere Klassen instanziiert, z.B.

```
// Testet ob eine Zahl gerade ist, also ob die Zahl
// ohne Rest durch 2 teilbar ist
Predicate<Integer> istGerade = new Predicate<Integer> () {
    @Override
    public boolean test(Integer i) {
        return i % 2 == 0;
    }
};
```

Es wurden also (je nach Schreibstil) 6 Zeilen geschrieben, von denen genau eine wichtig war (return i % 2 == 0).

Hierzu gibt es seit Java 8 Lambdaausdrücke, eine Kurzschreibweise zur Instanziierung von sog. *funktionalen Schnittstellen*, also Interfaces welche genau eine Methode vorschreiben (und mit @FunctionalInterface annotiert sind).

Ein Lambdaausdruck ist definiert als (Argumente)−> { Anweisungen }, also am obigen Beispiel:

```
Predicate<Integer> istGerade = (i) −> { return i % 2 == 0; };
```

Weitere Vereinfachungen sind:

- Gibt es genau ein Argument, so können die Klammern weggelassen werden.
- Gibt es genau nur eine Anweisung (die return Anweisung), so können return, {} sowie ; weggelassen werden.

```
Predicate<Integer> istGerade = i −> i % 2 == 0;
```

Die Kombination dieser Kurzschreibweise mit den oben erarbeiteten Verallgemeinerungen macht die Datenverarbeitung besonders lesbar bzw. verständlich:

```
List<Integer> li = Arrays.asList(4, 2, 1, 5, 3, 6, 8);

li = filtern(li, i −> i % 2 == 0); // nur gerade Zahlen behalten
li = abbilden(li, i −> i ∗ i); // Zahlen quadrieren
int s = reduzieren(li, 0, (a, b) −> a + b); // ...und aufsummieren
```

Datenströme in Java

Nicht klausurrelevant.

Die oben erarbeiteten Verallgemeinerungen sind eine Hinführung auf die *funktionale* Programmierung, in der Funktionen (Methoden) wie normale Objekte verwendet werden können. Da Java rein nach Sprachspezifikation keine solche *funktionale* Sprache ist, behilft man sich hier mit sog. *funktionalen Schnittstellen*, also Interfaces welche genau eine (nicht-default) Methode haben, und deren Zweck einzig darin liegt, eine Funktion als Objekt zu übergeben. Wir hatten dazu Comparator<T>, Predicate<T>, Function<T, R> sowie BinaryOperator<T> kennen gelernt.

Wir haben an den obigen Beispielen auch gesehen, dass die Datenverarbeitung oft als Datenfluss mit Modifikatoren bzw. Operationen realisiert wird: Ausgehend von einer Liste werden verschiedene Zwischenergebnisse erstellt und abschließend das eigentliche Ergebnis in Form einer neuen Liste oder eines reduzierten Werts zurück gegeben. Solche Datenströme können in Java seit der Version 8 mit dem Streamkonzept implementiert werden. Herzstück dieser Methodik ist die Klasse java.util.Stream, welche einen Datenfluss modelliert, welcher nun unter anderem mit den folgenden Methoden bearbeitet werden kann:

- Stream<T> sorted() bzw. Stream<T> sorted(Comparator<T> comparator) zum Sortieren;
- Stream<T> filter(Predicate<T> pred) um einen Stream zu erstellen, welcher nur noch Elemente enthält, welche mit pred positiv getestetet wurden;
- Stream<R> map(Function<T, R> mapper) um einen Stream von T in einen Stream von R umzuwandeln; und
- T reduce(T identity, BinaryOperator<T> op) bzw. U reduce(U identity, BiFunction<U, T, U> acc, BinaryOperator<U> comb) um einen Stream auf einen einzelnen Wert zu reduzieren.

Solche Streams können z.B. von Arrays oder Listen erstellt werden:

```
1 Stream<Integer> ints = Stream.of(1, 2, 3, 4, 5);
2 Stream<Float> floats = Stream.of(new Float[] { 1.0, 2.0, 3.0});
3
4 List<String> liste = Arrays.asList("Hans", "Dampf");
5 Stream<String> strings = liste.stream();
```

Um mit Streams nun den aktuellen Punktestand sowie die Tordifferenz für Stuttgart (Verein 8) auszurechnen, können wir den originalen Datenstrom *Spiele* (bl.spiele.stream())…

1. …auf Spiele mit Stuttgart filtern;

2. …dann auf ein Paar von Punkten und Tordifferenz abbilden (hierzu beachten, ob Stuttgart Gast oder Gastgeber ist);
3. …den Stream von solchen Paaren nun reduzieren, indem jeweils die linken und rechten Teile der Paare aufaddiert werden.

```
1  Bundesliga bl = Bundesliga.loadFromResource();
2
3  final int team = 8; // VfB Stuttgart
4  Pair<Integer, Integer> scores = bl.spiele.stream()
5          // nur Spiele mit Stuttgart
6          .filter(s -> s.getHeim() == team || s.getGast() == team)
7          // bilde ab Spiel --> Paar(Punktgewinn, Tordifferenz) pro Spiel
8          .map(new Function<Spiel, Pair<Integer, Integer>> () {
9              public Pair<Integer, Integer> apply(Spiel s) {
10                 boolean heimspiel = (s.getHeim() == team);
11
12                 // Punkte: 0-1-3 fuer Niederlage, Unentschieden, Sieg
13                 int punkte = 0;
14                 if (s.getToreHeim() == s.getToreGast())
15                     punkte = 1;
16                 else if ((heimspiel && s.getToreHeim() > s.getToreGast()) ||
17                         !heimspiel && s.getToreGast() > s.getToreHeim())
18                     punkte = 3;
19
20                 // Punktedifferenz -- Achtung: ist das Team Host/Gast?
21                 int diff = heimspiel
22                         ? s.getToreHeim() - s.getToreGast()
23                         : s.getToreGast() - s.getToreHeim();
24
25                 // System.out.println(s.toString(bl.vereine) + " " + punkte + " "
                       + diff);
26
27                 return Pair.of(punkte, diff);
28             }
29         })
30         // Jetzt Punkte (left) und Differenzen (right) aufaddieren, fuer
               Gesamtpunkte und -differenz.
31         .reduce(Pair.of(0, 0),
32                 (a, b) -> Pair.of(a.getLeft()+b.getLeft(),
                       a.getRight()+b.getRight()),
33                 (a, b) -> Pair.of(a.getLeft()+b.getLeft(),
                       a.getRight()+b.getRight()));
```

Erweiterte Operationen

Für Streams gibt es noch weitere nützliche Funktionen:

- limit(int n): terminiert den Stream nach n Elementen;

- skip(int n): entfernt die ersten n Elemente des Streams;
- findFirst() bzw. findLast(): Gibt ein Optional<T> auf das erste bzw. letzte Element zurück, welches mit .get() ausgepackt werden kann.
- collect(...): Sammelt Elemente des Streams ein, z.B. in eine Liste (Collectors.toList()) oder Map (Collectors.toMap(...)); mit Collectors.groupingBy(...) können Elemente nach einem Schlüssel in Listen einsortiert werden; siehe ausführliche Dokumentation.

...sowie viele mehr, zu finden in der Dokumentation.

Zusammenfassung

- Datenverarbeitung besteht meist aus einer Kombination von Sortieren, Filtern, Abbilden und Reduzieren
- *Sortieren* ändert die Reihenfolge der Objekte, und wird oft mit Comparator<T>-Objekten realisiert.
- *Filtern* entfernt Objekte, welche nicht positiv mit einem Predicate<T> getestet werden.
- *Abbilden* heißt Objekte eines Typs (z.B. Spiel) in Objekte eines anderen Typs (z.B. Pair<Integer, Integer> für Punkte und Tordifferenz) umzuwandeln.
- *Reduzieren* heißt Objekte eines Typs (z.B. Pair<Integer, Integer>) auf ein Objekt zu akkumulieren, wozu eine entsprechende Operation definiert werden muss.
- Die vier Operationen können explizit durch Listengenerierung und Iteration realisiert werden; abstrahiert man die Operationen, so kann man sich auf die Implementierung der eigentlichen Logik konzentrieren.
- Lambdaausdrücke ((Argumente)−> { Anweisungen }) sind eine kompakte Alternative zu anonymen inneren Klassen für funktionale Interfaces (Schnittstellen, die genau eine Methode vorschreiben).
- Java Streams sind eine Variante, bei der nur die eigentliche Logik zu implementieren ist; die Iterationslogik ist versteckt.

Lösungen zu den Beispielen

1. Torstatistiken.
    1. 2.645
    2. 2.746
    3. 24.968
    4. Nein (2.76 nachmittags, 2.60 abends)
    5. Nein (0.80 daheim, 1.03 auswärts)
2. Vereine.
    1. 88

2. 36
3. 57
4. $35 - 36 = -1$

3. 1. Liga.
2. Lösung
3. Lösung
4. 8, 3, 7, 8, 12, 15, 16, 15, 16, 16, 15, 15, 15, 15, 15, 16, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 18, 18, 17, 17, 17, 17
5. Köln, Kaiserslautern, Erfurt