Technische Hochschule Rosenheim
Technical University of Applied Sciences

Objektorientierte Programmierung
Kapitel 09 – Datenverarbeitung 2

Prof. Dr. Kai Höfig

Verallgemeinerung der Operationen

Technische Hochschule Rosenheim
Technical University of Applied Sciences

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

```
Bundesliga b = Bundesliga.loadFromResource();
for (Verein v : b.vereine.values())  // Iteration
    System.out.println(v);           // Verarbeitung von `v`
```

- Man sieht, dass bei den oben beschriebenen Operationen der 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 muss immer wieder geschrieben werden, obwohl er immer gleich ist. Um die eigentliche Verarbeitung der Daten übersichtlicher zu gestalten, *trennt* man nun Besuch und Verarbeitung.

Verallgemeinerung der Verarbeitung

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Die Verarbeitung (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:

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

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

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

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

Verallgemeinerung der Iteration

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- 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 werden aber nun anonyme innere Klassen viel kürzer schreiben*

Lambda Ausdrücke in Java

Technische Hochschule Rosenheim
Technical University of Applied Sciences

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

- Dadurch wird die Instanziierung von anonymen inneren Klassen mit nur einer Methode drastisch verkürzt:

```
Comparator<String> sc = new Comparator<String>() {
    @Override
    public int compare(String o1, String o2) {
        return o1.compareTo(o2)*-1;
    }
};

// Aber mit einem Lambda Audruck geht es auch so:
Comparator<String> sc2 = (o1,o2)->(o1.compareTo(o2)*-1);
```

Iteration und Operation mit Lambda Ausdruck

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Aus

```
fuerJedes(b.vereine.values(), new Consumer<Verein>() {
    public void accept(Verein v) {
        System.out.println(v);
    }
});
```

- Wird

```
fuerJedes(b.vereine.values(),v -> System.out.println(v));
```

Filtern

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- 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
    }
}
```

Abstraktion des Filterns

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- 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);
}
```

- ..und das Filtern abstrakt definieren

```
static <T> List<T> filtern(List<T> liste, Predicate<T> pred) {
    List<T> gefiltert = new LinkedList<>();

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

    return gefiltert;
}
```

Filter mit und ohne Lambda Ausdruck

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```
// nun kann man auch hier abstrahieren
List<Verein> zweiteL = filtern(b.vereine.values(), new Predicate<Verein>() {
    public boolean test(Verein v) {
        return v.getLiga() == 2;
    }
});
```

```
// und schöner wirds mit Lambda
List<Verein> zL = filtern(b.vereine.values(),v->v.getLiga()==2);
```

Abbilden

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- 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()));
}
```

Verallgemeinerung der Abbildung

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- 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:

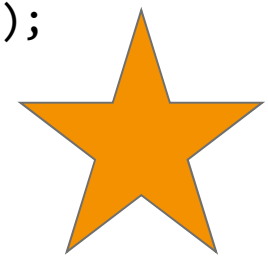
```
interface Function<T, R> {
    R apply(T t);
}
```

```
static <T, R> List<R> abbilden(Collection<T> liste, Function<T, R> func) {
    List<R> abgebildet = new LinkedList<>();

    for (T v : liste)
        abgebildet.add(func.apply(v));

    return abgebildet;
}
```

Abbildung abstrakt, mit und ohne Lambda

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```
// Abstrakt
List<Triple<String, String, String>> paarungen2 = abbilden(b.spiele,
        new Function<Spiel, Triple<String, String, String>>() {
    public Triple<String, String, String> apply(Spiel spiel) {
        Verein heim = b.vereine.get(spiel.getHeim());
        Verein gast = b.vereine.get(spiel.getGast());
        return Triple.of(spiel.getDatum(), heim.getName(), gast.getName());
    }
});

// mit Lambda
List<Triple<String, String, String>> paarungen3 = abbilden(b.spiele,s->{
    Verein heim = b.vereine.get(s.getHeim());
    Verein gast = b.vereine.get(s.getGast());
    return Triple.of(s.getDatum(), heim.getName(), gast.getName());
});
```



Reduktion

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Im Eingangsbeispiel hatten wir eine Liste von Spielen auf einen Integerwert (die Gesamtsumme der Tore abgebildet).

```
int tore = 0;
for (Spiel s : b.spiele) {
    tore = tore + s.getToreGast() + s.getToreHeim();
}

System.out.println("Es fielen insgesamt " + tore +
" Tore in " + b.spiele.size() + " Spielen");
// "Es fielen insgesamt 1741 Tore in 714 Spielen"
```

Verallgemeinerung der Reduktion

Technische Hochschule Rosenheim
Technical University of Applied Sciences

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

Reduktion abstrakt und mit Lambda Ausdrücken

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```
// und noch die Reduktion
Integer tore = Abstraktion.reduzieren(b.spiele, 0, new BiFunction<Integer, Spiel, Integer>() {
    @Override
    public Integer apply(Integer integer, Spiel spiel) {
        return integer + spiel.getToreGast() + spiel.getToreHeim();
    }
});

// wieder mit Lambda
Integer tore2 = Abstraktion.reduzieren(b.spiele, 0, (i,s)->i + s.getToreGast() + s.getToreHeim());
```

Weiteres Beispiel für eine Reduktion: Minimum und Maximum

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```
// Reduktion auf höchste Anzahl Heim-Tore
Integer meisteTore = Abstraktion.reduzieren(b.spiele, b.spiele.get(0).getToreHeim(),
new BiFunction<Integer, Spiel, Integer>() {

    @Override
    public Integer apply(Integer max, Spiel s) {
        return max<s.getToreHeim()?s.getToreGast():max;
    }
});

// wieder mit Lambda
Integer meisteTore2 = Abstraktion.reduzieren(b.spiele, b.spiele.get(0).getToreHeim(),
(max,s)->max<s.getToreHeim()?s.getToreGast():max);
```

Datenströme in Java: Motivation

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Wie finden Sie das?

```
// Jetzt können wir ja bequem Analysen zusammenstöpseln
// wir geben jetzt die Spiele der zweiten Liga aus wo die meisten Tore gefallen sind
fuerJedes(
        abbilden(
                filtern(
                        filtern(
                                b.spiele,v->v.getToreGast()+v.getToreHeim()
                                        >=
                                        Abstraktion.reduzieren(
                                                b.spiele,
                                                0,
                                                (i,s)->i<s.getToreHeim()+s.getToreGast()?s.getToreHeim()+s.getToreGast():i)),
                        v->b.vereine.get(v.getHeim()).getLiga()==2),
                 s->{
        Verein heim = b.vereine.get(s.getHeim());
        Verein gast = b.vereine.get(s.getGast());
        return Triple.of(s.getToreHeim()+s.getToreGast(), heim.getName(), gast.getName());
}), v-> System.out.println(v));

// Sortieren unterbricht unsere schöne Kette, denn
// .sort returns void :/
//.sort((t1,t2)->t1.getMiddle().compareTo(t2.getMiddle()))
```

- *Antwort: Danke, ätzend.*

java.util.Stream

Technische Hochschule Rosenheim
Technical University of Applied Sciences

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.

Beispiel

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```
Stream<Integer> ints = Stream.of(1, 2, 3, 4, 5);
// Quersumme aller Elemente kleiner als 5
System.out.println(ints.filter(i->i<=4).reduce(0,(i,v)->i+v).toString());
```

- Streams lassen sich natürlich auch aus `collections` erzeugen!

```
b.spiele.stream()
   .filter(s->b.vereine.get(s.getGast()).getLiga()==2)
   .filter(s->s.getToreGast()+s.getToreHeim()>=
          b.spiele.stream().reduce(
                  0,
                  (t,sp) -> t<sp.getToreGast()+sp.getToreHeim()?sp.getToreHeim()+sp.getToreGast():t,
                  (i1,i2)->0)//)
   )
   .map(s->{
       Verein heim = b.vereine.get(s.getHeim());
       Verein gast = b.vereine.get(s.getGast());
       return Triple.of(s.getDatum(), heim.getName(), gast.getName());
   })
   .sorted((t1,t2)->t1.getMiddle().compareTo(t2.getMiddle()))
   .forEach(s-> System.out.println(s));
```