Technische Hochschule Rosenheim
Technical University of Applied Sciences

Object-oriented programming
Chapter 09 – Data processing 2

Prof. Dr Kai Höfig

Basic operations

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Data processing is essentially based on four basic operations:

Sorting: sorting the data into the desired order
- List of clubs in ascending order by club name?
- First by league, then by club name?

Filtering: removing certain data, or only retaining certain data
- *Given*: a list of all clubs
- *Wanted*: a list of all 2nd league clubs

Mapping: Converting data from one format to another
- *Given*: a list of games
- *Wanted*: a list of match pairings (date, home, away)

Summarising: summarising/combining data
- *Given*: a list of games
- *Wanted*: how many goals were scored overall?

Generalisation of the operations

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Let's start with the simplest version of the iteration. Assuming we want to output each club to `System.out`, we have previously programmed *iteration* and *processing* together, e.g. with the `for-each` loop.

```
Bundesliga b = Bundesliga.loadFromResource();
for (Club c : b.clubs.values())  // iteration
   System.out.println(c);            // processing of `c`
```

- We can see that with the above operations, the iteration, i.e. *accessing* the elements, and the actual logic, i.e. *processing* of the elements, are combined with each other. But this also means that the code for iteration must be written again and again, although it is always the same. In order to make the actual processing of the data clearer, we now *separate* the accessing and processing.

Generalisation of processing

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- The processing (here: `System.out.println`) is thus a method that takes exactly one argument (here of the type `Club`) and has no return type. This is called a Consumer and is available as an interface in Java.

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

- If we now want to separate the processing, we could initially implement this as follows:

```
// processing
Consumer<Club> cons = new Consumer<Club>() {
  @Override
  public void accept(Club c) {
        System.out.println(c);
    }
};

// iteration
for (Club c : b.clubs.values())
    cons.accept(c);
```

Generalisation of the iteration

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- As the next step, we can now export the iteration logic:

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

- We put the building blocks together:

```
// before: iterate and process in one go
for (Club c : b.clubs.values())
    System.out.println(c);

// after: only processing logic, no iteration code
forEach(b.clubs.values(), new Consumer<Club>() {
  public void accept(Club c) {
        System.out.println(c);
    }
});
```

- *Strictly speaking, the second version is now two lines longer, but we will now write much shorter anonymous inner classes*

Lambda expressions in Java

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Since Java 8 there are *lambda expressions*, a short notation for the instantiation of so-called functional interfaces, i.e. interfaces that prescribe exactly one method (and are annotated with `@FunctionalInterface`).

- This drastically shortens the instantiation of anonymous inner classes with only one method:

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

// but with a lambda expression, it can also be:
Comparator<String> sc2 = (o1,o2)->(o1.compareTo(o2)*-1);
```

Iteration and operation with a lambda expression

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- This

```
forEach(b.clubs.values(), new Consumer<Club>() {
    public void accept(Club c) {
       System.out.println(c);
     }
});
```

- becomes

```
forEach(b.clubs.values(),c -> System.out.println(c));
```

Filtering

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- When filtering, it is noticeable that the iteration occurs first, and thereafter the element is added to a new list, subject to a specific condition.

```
List<Club> secondLeague = new LinkedList<>();
for (Club c : b.clubs.values()) {
   // check condition...
   if (c.getLeague() == 2) {
        secondLeague.add(c);  // add to list
   }
}
```

Abstraction of filtering

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Similar to the `Comparator<T>`, which abstracts the comparison when sorting, the condition can be abstracted here with a `Predicate<T>` :

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

- ..and filtering defined abstractly

```
static <T> List<T> filter(List<T> list, Predicate<T> pred) {
    List<T> filtered = new LinkedList<>();

  for (T c : list) {
    if (pred.test(c))
            filtered.add(c);
    }

  return filtered;
}
```

Filtering with and without a lambda expression

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```
// now we can also abstract here
List<Club> secondL = filter(b.clubs.values(), new Predicate<Club>() {
   public boolean test(Club c) {
      return c.getLeague() == 2;
    }
});
```

```
// and it gets better with lambda
List<Club> sL = filter(b.clubs.values(),c->c.getLeague()==2);
```

Mapping

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- When mapping, we recognise that although both input and output are lists, the data types are usually different!

```
// new list with target data type
List<Triple<String, String, String>> pairings = new LinkedList<>();
for (Game g : b.games) {

   // use club table to reference ID to club

   Club home = b.clubs.get(g.getHome());
    Club away = b.clubs.get(g.getAway());


   // create new Triple from date and club name

   pairings.add(Triple.of(g.getDate(), home.getName(), away.getName()));
}
```

Generalisation of the mapping

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Here, a `List<Game>` is mapped to a `List<Triple<String, String, String>>`. As with filtering, the entire list is run through, and then a new entry is created in the target list for each original entry. This operation can be generalised as a method that takes an argument of type `T` and returns an object of type `R` (`return`). Similar to `Comparator` and `Predicate`, this method is passed in the context of an interface:

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

```
static <T, R> List<R> map(Collection<T> list, Function<T, R> func) {
    List<R> mapped = new LinkedList<>();

  for (T c : list)
    mapped.add(func.apply(c));

  return mapped;
}
```

Abstract mapping, with and without lambda

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```
// abstract
List<Triple<String, String, String>> pairings2 = map(b.games,
        new Function<Game, Triple<String, String, String>>() {
   public Triple<String, String, String> apply(Game game) {
        Club home = b.clubs.get(game.getHome());
        Club away = b.clubs.get(game.getAway());
      return Triple.of(game.getDate(), home.getName(), away.getName());
    }
});

// with lambda
List<Triple<String, String, String>> pairings3 = map(b.games,g->{
    Club home = b.clubs.get(g.getHome());
    Club away = b.clubs.get(g.getAway());
   return Triple.of(g.getDate(), home.getName(), away.getName());
});
```

Summarising

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- In the initial example, we mapped a list of games to an integer value (the total number of goals).

```
int goals = 0;
for (Game g : b.games) {
    goals = goals + g.getGoalsHome() + g.getGoalsAway();
}


System.out.println("There was a total of " + goals +
" goals scored in " + b.games.size() + " games");
// "There was a  total of 1741 goals scored in 714 games"
```

Generalisation of summarising

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Now we must specify the summarising method a little more precisely, so that the actual summarising uses the data elements, but a different data type can be returned:

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

static <T> T summarise(Collection<T> list, T identity, BinaryOperator<T> op) {
    T a = identity;
    for (T t : list)
         a = op.apply(a, t);
    return a;
}
```

Summarising abstractly and with lambda expressions

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```
// and also the summarising
Integer goals = Abstraction.summarise(b.games, 0, new BiFunction<Integer, Game, Integer>() {

    @Override

    public Integer apply(Integer integer, Game game) {

        return integer + game.getGoalsAway() + game.getGoalsHome();
    }
});


// again with lambda
Integer goals2 = Abstraction.summarise(b.games, 0, (i,g)->i + g.getGoalsAway() + g.getGoalsHome());
```

Another example of summarising: minimum and maximum

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```
// summarising highest number of home goals
Integer mostGoals = Abstraction.summarise(b.games, b.games.get(0).getGoalsHome(),
new BiFunction<Integer, Game, Integer>() {

    @Override

    public Integer apply(Integer max, Game g) {

        return max<g.getGoalsHome()?g.getGoalsAway():max;
     }
});


// again with lambda
Integer mostGoals2 = Abstraction.summarise(b.games, b.games.get(0).getGoalsHome(),
(max,g)->max<g.getGoalsHome()?g.getGoalsAway():max);
```

Data streams in Java: motivation

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- How do you feel about that?

```
// now we can easily put analyses together
// we now output the games in the second league where the most goals were scored
forEach(
    map(
        filter(
            filter(
                b.games,c->c.getGoalsHome()+c.getGoalsAway()
                                        >=
                    Abstraction.summarise(
                        b.games,
                        0,
                                                (i,g)->i<g.getGoalsHome()+g.getGoalsAway()?g.getGoalsHome()+g.getGoalsAway():i)
                        c->b.clubs.get(c.getHome()).getLeague()==2),
                g->{
        Club home = b.clubs.get(g.getHome());
        Club away = b.clubs.get(g.getAway());
    return Triple.of(g.getGoalsHome()+g.getGoalsAway(), home.getName(), away.getName());
}), c-> System.out.println(c));

// sorting interrupts our nice chain, because
// .sort returns void :/
//.sort((t1,t2)->t1.getMiddle().compareTo(t2.getMiddle()))
```

- *Answer: It's terrible, thanks.*

java.util.Stream

Technische Hochschule Rosenheim
Technical University of Applied Sciences

We have also seen from the above examples that data processing is often implemented as a data flow with modifiers or operations.

- Starting from a list, various intermediate results are generated, and finally the actual result is returned in the form of a new list or a summarised value.
- Such data streams can be implemented in Java since version 8 using the stream concept. At the heart of this methodology is the `java.util.Stream` class, which models a data flow that can now be processed using the following methods, among others:

- **`Stream<T>`** `sorted()` or `Stream<T> sorted(Comparator<T> comparator)` for sorting;
- `Stream<T> filter(Predicate<T> pred)` to create a stream that contains only elements that have been tested positive with pred;
- `Stream<R> map(Function<T, R> mapper)` to convert a stream of `T` to a stream of `R`; and
- `T reduce(T identity, BinaryOperator<T> op)` or `U reduce(U identity, BiFunction<U, T, U> acc, BinaryOperator<U> comb)` to summarise a stream into a single value.

Example

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```
Stream<Integer> ints = Stream.of(1, 2, 3, 4, 5);
// checksum of all elements less than 5
System.out.println(ints.filter(i->i<=4).reduce(0,(i,c)->i+c).toString());
```

- Of course, streams can also be generated from `collections` !

```
b.games.stream()
   .filter(g->b.clubs.get(g.getAway()).getLeague()==2)
   .filter(g->g.getGoalsAway()+g.getGoalsHome()>=
     b.games.stream().reduce(
         0,
                  (t,sp) -> t<sp.getGoalsAway()+sp.getGoalsHome()?sp.getGoalsHome()+sp.getGoalsAway():t,
                  (i1,i2)->0)//)
 )
   .map(g->{
       Club home = b.clubs.get(g.getHome());
       Club away = b.clubs.get(g.getAway());
    return Triple.of(g.getDate(), home.getName(), away.getName());
   })
   .sorted((t1,t2)->t1.getMiddle().compareTo(t2.getMiddle()))
   .forEach(g-> System.out.println(g));
```