CS 213 – Software Methodology

Spring 2019

Lecture 26: May 2

Collecting Data With Streams

Movies: Ratings < 3

Want to get list of movies with ratings < 3

Collecting data from a stream
(terminal operation, does not return a stream):

```java
List<String> badMovies =
    movies.stream()
          .filter(m -> m.getRating() < 3)
          .map(Movie::getName)
          .collect(toList());    ← collect the stream
                                   into a list
System.out.println(badMovies);
```

```
[Fifty Shades of Grey, Transcendence, Conan The Barbarian, The Last Airbender]
```

The `Collectors` class

`toList` is actually a static method in the
`java.util.stream.Collectors` class

```
import static java.util.stream.Collectors.toList;

List<String> badMovies =
    movies.stream()
          .filter(m -> m.getRating() < 3)
          .map(Movie::getName)
          .limit(2)
          .collect(toList());
```

`toList` returns a `Collector` instance that can gather items in a stream into a `List` instance

This `Collector` is passed as a parameter to the `collect` method of a `Stream` instance

The `Collectors` class

The `Collectors` class provides static methods that return pre-defined `Collector` instances for the following:

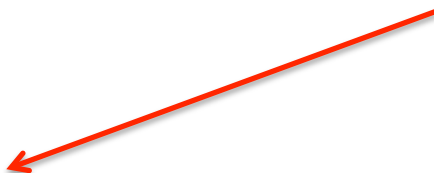- Reducing and Summarizing
- Grouping
- Partitioning

For all examples that follow assume we have coded the statement below to import the required `Collectors` static methods as needed:

```
import static java.util.stream.Collectors.*;
```

Collector for Grouping

Example: Grouping movies by genre:

Grouping by genre gives a Map from genre to all movies (List) in that genre

```
Map<Movie.Genre, List<Movie>> moviesByGenre =
movies.stream()
    .collect(groupingBy(Movie::getCategory));

System.out.println(moviesByGenre);
```

```
{
 ADVENTURE=[Conan The Barbarian, The Last Airbender, Harry Potter and the Deathly Hallows: Part 1],
 ACTION=[Max Max: Fury Road, American Sniper],
MYSTERY=[Sicario, The Gift], THRILLER=[Transcendence],
DRAMA=[Straight Outta Compton, Fifty Shades of Grey]
}
```

Collector for Partitioning

Partitioning is a special kind of grouping based on a predicate, so that there are two groups (partitions), one for when the predicate is false, and the other for when it is true

Example: Partition movies into before 2014, and the rest

```java
Map<Boolean, List<Movie>> partitionedMovies =
    movies.stream()
    .collect(partitioningBy(m -> m.getYear() < 2014));

System.out.println(partitionedMovies);


{
    false=[Max Max: Fury Road, ...],
    true=[Conan The Barbarian, ...]
}
```

Collector for Reduction

Example: Count – find number of movies in list

```
long numMovies = movies.stream().collect(counting());
```

which is equivalent to:

```
long numMovies = movies.stream().count();
```

Example: Max – find any highest rated movie

```
Optional<Movie> maxRatedMovie =
movies.stream()
    .collect(maxBy(Comparator.comparingInt(Movie::getRating)));
maxRatedMovie.ifPresent(System.out::println);
```

which is equivalent to:

```
movies.stream().max(Comparator.comparingInt(Movie::getRating))
```

Collector for Summarizing

Example: Average – find average rating of movies in list

```
double avgRating =
movies.stream()
   .collect(averagingInt(Movie::getRating));
System.out.println(avgRating);
```

Returns 0 if no elements are present

which can be achieved without collect:

```
movies.stream()
      .mapToInt(Movie::getRating)
      .average()
      .ifPresent(System.out::println);
```

Can also do summingInt and summing/averaging for double and long types

Collector for Summary Stats

Example: Get summary rating stats for movies in list

In `java.util`

```
IntSummaryStatistics ratingStats =
movies.stream()
    .collect(summarizingInt(Movie::getRating));

System.out.println(ratingStats);
```

```
IntSummaryStatistics{count=10, sum=31, min=1, average=3.100000, max=5}
```

Can also do `DoubleSummaryStatistics` and `LongSummaryStatistics`

Collector for Grouping

Example: Group movies as good (rating > 3), ok (== 3), or bad (< 3)

```
   Use:  public static enum MovieQuality { GOOD, OK, BAD};

Map<MovieQuality, List<Movie>> moviesByQuality =
    movies.stream()
     .collect(groupingBy(
        movie -> {
            if (movie.getRating() > 3) return MovieQuality.GOOD;
            else if (movie.getRating() < 3) return MovieQuality.BAD;
            else return MovieQuality.OK;
        }));
System.out.println(moviesByQuality);



{
 BAD=[Fifty Shades of Grey, Transcendence, Conan The Barbarian, The Last Airbender],
 OK=[The Gift],
 GOOD=[Max Max: Fury Road, Straight Outta Compton, American Sniper, Harry …, Sicario]
}
```

Two-level Grouping

Example: Group movies by genre, then year

```java
Map<Movie.Genre, Map<Integer, List<Movie>>>
moviesByGenreYear =
  movies.stream()
    .collect(groupingBy(Movie::getCategory,
               groupingBy(Movie::getYear)));
System.out.println(moviesByGenreYear);
```

```
{
 ADVENTURE={
   2010=[The Last Airbender, Harry Potter and the Deathly Hallows: Part 1],
   2011=[Conan The Barbarian]},
 ACTION={
   2014=[American Sniper],
   2015=[Max Max: Fury Road]},
 MYSTERY={
   2000=[The Gift], 2015=[Sicario]},
 THRILLER={
   2014=[Transcendence]},
  DRAMA={
   2015=[Straight Outta Compton, Fifty Shades of Grey]}
}
```

Two-level Collection

The second argument for groupingBy does not have to be
a groupingBy, it could be other Collector functions.

Example: Count movies by genre

```
Map<Movie.Genre, Long>
numMoviesByGenre =
  movies.stream()
    .collect(groupingBy(Movie::getCategory,
              counting()));
System.out.println(numMoviesByGenre);


{ADVENTURE=3, ACTION=2, MYSTERY=2, THRILLER=1, DRAMA=2}
```

The single-argument groupingBy is actually short for a 2-arg
version with the second argument being toList(), which is
why the resulting Map has a List for its value.

Two-level Collection

Example: Get (a) top-rated movie by genre

```
Map<Movie.Genre, Optional<Movie>>
topMoviesByGenre =
  movies.stream()
    .collect(groupingBy(Movie::getCategory,
             maxBy(comparingInt(Movie::getRating))));
System.out.println(topMoviesByGenre);
```

```
{ADVENTURE=Optional[Harry Potter and the Deathly Hallows: Part 1],
 ACTION=Optional[Max Max: Fury Road], MYSTERY=Optional[Sicario],
 THRILLER=Optional[Transcendence], DRAMA=Optional[Straight Outta Compton]
}
```

`Optional` is of no relevance in this example, because if there is no movie in a genre, there will not be a key for it in the map.

Collecting and Transforming

Since `Optional` is of no use in the previous example, we want to replace the `Optional` value in the mapping with the movie that it holds

To make this happen, we use a different `Collector`, generated by the method `collectingAndThen`

```
Map<Movie.Genre, Movie>
topMovieByGenre = movies.stream()
    .collect(
        groupingBy(Movie::getCategory,
          collectingAndThen(maxBy(comparingInt(Movie::getRating)),
                            Optional::get)));
System.out.println(topMovieByGenre);


  {
    ADVENTURE=Harry Potter and the Deathly Hallows: Part 1,
    ACTION=Max Max: Fury Road, MYSTERY=Sicario,
    THRILLER=Transcendence, DRAMA=Straight Outta Compton
  }
```

Two-level Collection

Suppose we want to list, for each genre, all the years for which movies are available in that genre: we want a mapping from genre to years

```
System.out.println("\nYears of movies by genre");

    Map<Movie.Genre, Map<Integer, List<Movie>>> yearMoviesByGenre =
    movies.stream()
            .collect(groupingBy(Movie::getCategory,
                        groupingBy(Movie::getYear)));
System.out.println(yearMoviesByGenre);


        {
          ADVENTURE={2010=[The Last Airbender, Harry Potter ...],
                     2011=[Conan The Barbarian]},
          ACTION=...
        }
```

Not quite what we want: applying groupingBy on year at the second level gives another mapping, not a list of years.

Two-level Collection

To just get a list of years at the second level, we can use the static method `Collectors.mapping` that returns a `Collector` that can map unique values of a particular attribute (year in this case) to a set

```
System.out.println("\nYears of movies by genre");
   Map<Movie.Genre, Set<Integer>>> yearMoviesByGenre =
      movies.stream()
            .collect(groupingBy(Movie::getCategory,
                        mapping(Movie::getYear, toSet())));
System.out.println(yearMoviesByGenre);


{
  ADVENTURE=[2010, 2011], ACTION=[2014, 2015], MYSTERY=[2000, 2015],
  THRILLER=[2014],  DRAMA=[2015]
}
```

Three-level Collection

Example: Get movies by genre, then year, then rating

```
Map<Movie.Genre, Map<Integer, Map<Integer, List<Movie>>>>
movies3way =
    movies.stream()
        .collect(groupingBy(Movie::getCategory,
                    groupingBy(Movie::getYear,
                        groupingBy(Movie::getRating))));
```

```
{
 ADVENTURE= {
   2010={2=[The Last Airbender], 4=[Harry Potter and the Deathly Hallows: Part 1]},
   2011={2=[Conan The Barbarian]}},
 ACTION={
   2014={4=[American Sniper]}, 2015={5=[Max Max: Fury Road]}},
 MYSTERY={
   2000={3=[The Gift]}, 2015={4=[Sicario]}},
 THRILLER={2014={1=[Transcendence]}},
  DRAMA={2015={1=[Fifty Shades of Grey], 5=[Straight Outta Compton]}}}
}
```

Collecting on Primitive Stream

If you set up a primitive stream (such as `IntStream)` you will need to convert it into a stream of Integer objects before you can `collect`

```
IntStream.rangeClosed(2,n)

        .boxed()  ⟵———— returns Stream<Integer>

        .collect(partitioningBy(i -> i % 2 == 0))
```

Useful `Collectors` static factory methods

Factory method	Return Type	Used to
toList	List<T>	Gather into a list
toSet	Set<T>	Gather into a set
counting	Long	Count items
summingInt/averagingInt	Integer/Double	Sum/average items
summarizingInt	IntSummaryStatistics	Max, min, total, average
maxBy/minBy	Optional<T>	Max/Min with Comparator
reducing	Type of reduction	Reduce to single value
collectingAndThen	Type of transformation	Collect+transform
groupingBy	Map<K,List<T>>	Group by K
partitioningBy	Map<Boolean,List<T>	Group by false/true

For both `groupingBy` and `partitioningBy,` the single argument version produces a `List`, and the two-argument version takes a `Collector` as the second argument, which will change the value type of the returned `Map`