WORKED EXAMPLE 19.2 A Movie Database

In this Worked Example, we analyze a large database of movies and use streams to obtain interesting statistics from the data. To follow along, download the companion code for Worked Example 19.2.

Problem Statement The file `movies.txt` in the book's companion code has information about 23,000 movies, taken from the database of facts at `http://freebase.com`. Each movie has a year, title, and lists of directors, producers, and actors. What interesting facts can you find?

© Ivan Cholakov/iStockphoto.

Step 1 Get the data.

The `movies.txt` file has five lines for each movie which look like this:

```
Name: Five Easy Pieces
Year: 1970
Directed by: Bob Rafelson
Produced by: Bob Rafelson, Richard Wechsler, Harold Schneider
Actors: Jack Nicholson, Karen Black, Billy Green Bush, more...
```

First, let's come up with a class that describes a movie:

```
public class Movie
{
   private String title;
   private int year;
   private List<String> directors;
   private List<String> producers;
   private List<String> actors;

   public Movie(String title, int year, List<String> directors,
      List<String> producers, List<String> actors) { . . . }
   public String getTitle() { return title; }
   // Accessors for the other fields ...
}
```

Next, we need to read in the movies. Because we need to consume five input lines per movie, there is nothing to be gained by reading the input as a stream of lines. Instead, we just put the movies into an `ArrayList`:

```
public static List<Movie> readMovies(String filename) throws IOException
{
   List<Movie> movies = new ArrayList<>();
   try (Scanner in = new Scanner(new File(filename)))
   {
      while (in.hasNextLine())
      {
         String nameLine = in.nextLine();
         String yearLine = in.nextLine();
         String directorsLine = in.nextLine();
         String producersLine = in.nextLine();
         String actorsLine = in.nextLine();
         movies.add(new Movie(getString(nameLine),
            Integer.parseInt(getString(yearLine)),
            getList(directorsLine),
```

```
                getList(producersLine),
                getList(actorsLine)));
         }
      }
      return movies;
   }
```

Here, `getString` is a helper method that strips off the field header, and `getList` is a helper that breaks up a comma-separated list:

```
private static String getString(String line)
{
   int colon = line.indexOf(":");
   return line.substring(colon + 1).trim();
}
private static List<String> getList(String line)
{
   return Stream.of(getString(line).split(", "))
      .collect(Collectors.toList());
}
```

Step 2 Make a stream.

Because we have a method for reading a collection of `Movie` objects, simply call

```
List<Movie> movieList = readMovies("movies.txt");
Stream<Movie> movies = movieList.stream();
```

Step 3 Transform the stream.

Now we are ready to work with the data. The problem statement was rather vague. What interesting facts might be hidden in the data? Let's start with something simple: Are there any movie titles that start with the letter X?

This is a simple application of `map` and `filter`: Map each movie to its title, and filter the ones that start with an X.

```
List<String> result1 = movieList.stream()
   .map(m -> m.getTitle())
   .filter(t -> t.startsWith("X"))
   .collect(Collectors.toList());
```

Indeed, they are a few: *XX/XY*, *Xiu Xiu: The Sent Down Girl*, *X-15*, *X Marks the Spot*, *X-Men: First Class*, and so on. In the next step, you will see how many movies start with a given letter.

Is it common for a director to also be an actor? To answer this question, we want to check for each movie whether the list of directors and the list of actors have an element in common. You can map a movie to the intersection of the two lists (using a helper method), and then count the ones with nonempty intersections:

```
long count = movieList.stream()
   .map(m -> intersect(m.getDirectors(), m.getActors()))
   .filter(l -> l.size() > 0)
   .count();
```

The Java library doesn't have a method for computing the intersection of two collections, but it is easy to provide one:

```
public static Set<String> intersect(Collection<String> a, Collection<String> b)
{
   Set<String> intersection = new HashSet<>(a);
   intersection.retainAll(b);
   return intersection;
}
```

However, it is simpler to filter on the criterion without actually computing the intersection. For a given movie `m`, we want to know if any of the directors is also an actor. Perhaps surprisingly, this can be expressed more concisely with a stream than with methods from the `Collection` interface:

```
public static boolean commonActorAndDirector(Movie m)
{
   return m.getDirectors().stream().anyMatch(d -> m.getActors().contains(d));
}
```

Then simply compute

```
long count = movieList.stream()
   .filter(m -> commonActorAndDirector(m))
   .count();
```

You could dispense with the `commonActorAndDirector` helper method, but then it would be quite hard to follow what is going on.

Which movie has the most actors? It is easy to get the maximum number. First map each movie to the size of the actor list, then get the maximum:

```
int result2 = movieList.stream()
   .mapToInt(m -> m.getActors().size())
   .max()
   .orElse(0);
```

The call to `orElse` is necessary because `max` returns an `OptionalInt`.

As it turns out, the maximum is 100. But that only tells us that there is a movie with a hundred actors, not which one it is. To get that answer, we need to refine our strategy. Instead of mapping movies to numbers, we need to compute the "largest" movie, where one movie is larger than another if it has more actors:

```
movieList.stream()
   .max((a, b) -> a.getActors().size() - b.getActors().size())
   .ifPresent(m -> System.out.println("Movie with most actors: " + m));
```

Note the call to the `ifPresent` method. The `max` method returns an `Optional<Movie>`. In general, it is not a good idea to call `get` because that would cause an exception if there was no result. In our case, we know there has to be one, so we could call `get` anyway. But it is just as easy to use the safe call to `ifPresent`, passing a function to print the movie.

Also, if you find the comparator cumbersome, you may want to read Special Topic 19.4. There is a more elegant way to describe it, as

```
Comparator.comparing(m -> m.getActors().size())
```

The point of this example is that it isn't always necessary to process a stream. Sometimes you can obtain the result directly.

Step 4 Collect the results.

In the preceding examples, it was easy to collect the results. Let's look at a couple of examples that are more challenging. First, we want to know how many movies start with a given letter. This is a typical use of `groupingBy` with a secondary collector:

```
Map<String, Long> firstLetters = movieList.stream()
   .collect(Collectors.groupingBy(
      m -> m.getTitle().substring(0, 1),
      Collectors.counting()));
```

Almost 6,000 movies start with the letter T. There is a simple reason that you can verify by running the query

```
movieList.stream()
   .filter(m -> m.getTitle().startsWith("The "))
   .count();
```

Almost 5,000 titles start with the word "The".

Interestingly, the letters A, B, and C have high frequencies, probably because there is some incentive in picking a title that shows up at the top of alphabetical listings.

Who is the most prolific director? This is not so easy to answer because a movie can have more than one director. Fortunately, this only happens with about five percent of movies, so let's just pick the first one. There are, however, a number of movies with no directors in the data set. We filter those out first. Then we can group by the first director:

```
Map<String, List<Movie>> moviesByDirector = movieList.stream()
   .filter(m -> m.getDirectors().size() > 0)
   .collect(Collectors.groupingBy(
      m -> m.getDirectors().get(0)));
```

This map associates all directors with a list of the movies that they directed. Unfortunately, that's a large map with about 10,000 entries. How can we find out which director had the most movies? It's the map entry with the longest list.

There is no need to use streams. The `Collections.max` method yields the largest value of a collection:

```
String mostProlificDirector = Collections.max(
   moviesByDirector.entrySet(),
   Comparator.comparing(e -> e.getValue().size())).getKey();
```

It turns out that this director is D. W. Griffith, a pioneer of silent films who directed over 150 movies. Which movies? To extract the titles, it is easy to use `map` with a stream:

```
List<String> titles = moviesByDirector.get(mostProlificDirector)
   .stream()
   .map(m -> m.getTitle())
   .collect(Collectors.toList());
```

This example has shown you how you can discover facts in a data set. Once you start finding interesting results, you can issue additional queries to dig deeper. Streams are a good tool for exploring data because they let you focus on the "what, not how", allowing you to generate and refine queries quickly.