The Java™ Tutorials

Trail: Collections

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases.

Lesson: Aggregate Operations

Note: To better understand the concepts in this section, review the sections Lambda Expressions and Method References.

For what do you use collections? You don't simply store objects in a collection and leave them there. In most cases, you use collections to retrieve items stored in them.

Consider again the scenario described in the section Lambda Expressions. Suppose that you are creating a social networking application. You want to create a feature that enables an administrator to perform any kind of action, such as sending a message, on members of the social networking application that satisfy certain criteria.

As before, suppose that members of this social networking application are represented by the following `Person` class:

```
public class Person {

    public enum Sex {
        MALE, FEMALE
    }

    String name;
    LocalDate birthday;
    Sex gender;
    String emailAddress;

    // ...

    public int getAge() {
        // ...
    }

    public String getName() {
        // ...
    }
}
```

The following example prints the name of all members contained in the collection `roster` with a for-each loop:

```
for (Person p : roster) {
    System.out.println(p.getName());
}
```

The following example prints all members contained in the collection `roster` but with the aggregate operation `forEach`:

```
roster
    .stream()
    .forEach(e -> System.out.println(e.getName());
```

Although, in this example, the version that uses aggregate operations is longer than the one that uses a for-each loop, you will see that versions that use bulk-data operations will be more concise for more complex tasks.

The following topics are covered:

- Pipelines and Streams
- Differences Between Aggregate Operations and Iterators

Find the code excerpts described in this section in the example `BulkDataOperationsExamples`.

Pipelines and Streams

A *pipeline* is a sequence of aggregate operations. The following example prints the male members contained in the collection `roster` with a pipeline that consists of the aggregate operations `filter` and `forEach`:

```
roster
    .stream()
    .filter(e -> e.getGender() == Person.Sex.MALE)
    .forEach(e -> System.out.println(e.getName()));
```

Compare this example to the following that prints the male members contained in the collection `roster` with a for-each loop:

```
for (Person p : roster) {
    if (p.getGender() == Person.Sex.MALE) {
        System.out.println(p.getName());
    }
}
```

A pipeline contains the following components:

- A source: This could be a collection, an array, a generator function, or an I/O channel. In this example, the source is the collection `roster`.
- Zero or more *intermediate operations*. An intermediate operation, such as `filter`, produces a new stream.

  A *stream* is a sequence of elements. Unlike a collection, it is not a data structure that stores elements. Instead, a stream carries values from a source through a pipeline. This example creates a stream from the collection `roster` by invoking the method `stream`.

  The `filter` operation returns a new stream that contains elements that match its predicate (this operation's parameter). In this example, the predicate is the lambda expression `e -> e.getGender() == Person.Sex.MALE`. It returns the boolean value `true` if the `gender` field of object `e` has the value `Person.Sex.MALE`. Consequently, the `filter` operation in this example returns a stream that contains all male members in the collection `roster`.
- A *terminal operation*. A terminal operation, such as `forEach`, produces a non-stream result, such as a primitive value (like a double value), a collection, or in the case of `forEach`, no value at all. In this example, the parameter of the `forEach` operation is the lambda expression `e -> System.out.println(e.getName())`, which invokes the method `getName` on the object `e`. (The Java runtime and compiler infer that the type of the object `e` is `Person`.)

The following example calculates the average age of all male members contained in the collection `roster` with a pipeline that consists of the aggregate operations `filter`, `mapToInt`, and `average`:

```
double average = roster
    .stream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .mapToInt(Person::getAge)
    .average()
    .getAsDouble();
```

The `mapToInt` operation returns a new stream of type `IntStream` (which is a stream that contains only integer values). The operation applies the function specified in its parameter to each element in a particular stream. In this example, the function is `Person::getAge`, which is a method reference that returns the age of the member. (Alternatively, you could use the lambda expression `e -> e.getAge()`.) Consequently, the `mapToInt` operation in this example returns a stream that contains the ages of all male members in the collection `roster`.

The `average` operation calculates the average value of the elements contained in a stream of type `IntStream`. It returns an object of type `OptionalDouble`. If the stream contains no elements, then the `average` operation returns an empty instance of `OptionalDouble`, and invoking the method `getAsDouble` throws a `NoSuchElementException`. The JDK contains many terminal operations such as `average` that return one value by combining the contents of a stream. These operations are called *reduction operations*; see the section Reduction for more information.

Differences Between Aggregate Operations and Iterators

Aggregate operations, like `forEach`, appear to be like iterators. However, they have several fundamental differences:

- **They use internal iteration**: Aggregate operations do not contain a method like `next` to instruct them to process the next element of the collection. With *internal delegation*, your application determines *what* collection it iterates, but the JDK determines *how* to iterate the collection. With *external iteration*, your application determines both what collection it iterates and how it iterates it. However, external iteration can only iterate over the elements of a collection sequentially. Internal iteration does not have this limitation. It can more easily take advantage of parallel computing, which involves dividing a problem into subproblems, solving those problems simultaneously, and then combining the results of the solutions to the subproblems. See the section Parallelism for more information.
- **They process elements from a stream**: Aggregate operations process elements from a stream, not directly from a collection. Consequently, they are also called *stream operations*.
- **They support behavior as parameters**: You can specify lambda expressions as parameters for most aggregate operations. This enables you to customize the behavior of a particular aggregate operation.

Previous page: Previous Lesson
Next page: Reduction