Yesterday we talked about the Collections and Arrays utility classes that provide many useful methods for manipulating collections and arrays. Remember to use these classes for common tasks such as sorting and searching elements.

Today, we continue our journey by exploring the Stream API which is added to JDK since Java 8.

You know, the Stream API brings totally new ways for working with collections. Once you are familiar with it, you will love using it, as it makes you write code more natural, more succinct, more readable and most importantly, your productivity will go up incredibly. Trust me!

Let’s start by looking at some code examples to understand how the Stream API radically changes the way we work with collections. Don’t worry if you don’t fully understand the code right away.

Keep reading…

Suppose that we have the Student class as shown below:

class Student implements Comparable< Student > {

      private String name;

      private int score;

      public Student(String name, int score) {

[this.name](http://this.name/) = name;

            this.score = score;

      }

      // getters and setters....

      public String toString() {

            return [this.name](http://this.name/) + " - " + this.score;

      }

      public int compareTo(Student another) {

            return another.getScore() - this.score;

      }

}

and a list of students:

List< Student > listStudents = new ArrayList<>();

listStudents.add(new Student("Alice", 82));

listStudents.add(new Student("Bob", 90));

listStudents.add(new Student("Carol", 67));

listStudents.add(new Student("David", 80));

listStudents.add(new Student("Eric", 55));

listStudents.add(new Student("Frank", 49));

listStudents.add(new Student("Gary", 88));

listStudents.add(new Student("Henry", 98));

listStudents.add(new Student("Ivan", 66));

listStudents.add(new Student("John", 52));

We are required to do some calculations on this list.

First, find the students whose scores are greater than or equal to 70.

A non-stream solution would look like this:

// find students whose score >= 70

List< Student > listBadStudents = new ArrayList<>();

for (Student student : listStudents) {

      if (student.getScore() >= 70) {

            listBadStudents.add(student);

      }

}

for (Student student : listBadStudents) {

      System.out.println(student);

}

With the Stream API, we can replace the above code with the following:

// find students whose score >= 70

List< Student > listGoodStudents = listStudents.stream()

                              .filter(s -> s.getScore() >= 70)

                              .collect(Collectors.toList());

listGoodStudents.stream().forEach(System.out::println);

Don’t worry if you don’t understand the code. Just see the differences between non-stream code and stream-based code.

Second, calculate average score of all students. A trivial solution would look like this:

// calculate average score of all students

double sum = 0.0;

for (Student student : listStudents) {

      sum += student.getScore();

}

double average = sum / listStudents.size();

System.out.println("Average score: " + average);

And here’s a stream-based version:

// calculate average score of all students

double average = listStudents.stream()

                  .mapToInt(s -> s.getScore())

                  .average().getAsDouble();

System.out.println("Average score: " + average);

That’s it!

So what the differences do you see between the non-stream code and the stream-based code?

They look totally different, right? Do you notice that the stream-based version looks more natural, something like a query, right? But that’s not all.

Continue reading and you’ll see how streams are really powerful and flexible.

**\* What is a Stream?**

A stream represents a sequence of elements supporting sequential and parallel aggregate operations. Since Java 8, we can generate a stream from a collection, an array or an I/O channel.

Every collection class now has the **stream()** method that returns a stream of elements in the collections:

Stream stream = listStudents.stream();

Obtaining a stream from an array:

int[] arrayIntegers = {1, 8, 2, 3, 98, 11, 35, 91};

IntStream streamIntegers = Arrays.stream(arrayIntegers);

Obtaining a stream from a file:

BufferedReader bufferReader = new BufferedReader(new FileReader("students.txt"));

Stream streamLines = bufferReader.lines();

Operations can be performed on a stream are categorized into intermediate operations and terminal operations. We’ll see details of these operations shortly. Consider the following code:

List< Student > top3Students = listStudents.stream()

                        .filter(s -> s.getScore() >= 70)

                        .sorted()

                        .limit(3)

                        .collect(Collectors.toList());

System.out.println("Top 3 Students by Score:");

top3Students.forEach(s -> System.out.println(s));

This code can be read as: select top 3 students whose scores >= 70, and sort them by score in descending order (the natural ordering of the Student class). Here we can see the following intermediate operations: filter, sorted and limit; and the terminal operation is collect.

As you can see, the operations on a stream can be chained together (intermediate operations) and end with a terminal operation. Such a chain of stream operations is called stream pipeline.

**\* Stream Pipeline:**

We can say that a stream is a pipeline of aggregate operations that can be evaluated. A pipeline consists of the following parts:

- a source: can be a collection, an array or an I/O channel.

- zero or more intermediate operations which produce new streams, such as filter, map, sorted, etc.

- a terminal operation that produces a non-stream result such as a primitive value, a collection, or void (such as the forEach operation).

**\* Intermediate Operations:**

An intermediate operation processes over a stream and return a new stream as a response. Then we can execute another intermediate operation on the new stream, and so on, and finally execute the terminal operation.

One interesting point about intermediate operations is that they are lazily executed. That means they are not run until a terminal operation is executed.

The Stream API provides the following common intermediate operations:

* map()
* filter()
* sorted()
* limit()
* distinct()

For a full list of intermediate operations, consult the [Stream Javadoc](http://app.getresponse.com/click.html?x=a62b&lc=fvuXG&mc=Iw&s=I3kzkY&u=ml0B&y=d&).

**\* Terminal Operations:**

A stream pipeline always ends with a terminal operation, which returns a concrete type or produces a side effect. For instances, the collect operation produces a collection; the forEach operation does not return a concrete type, but allows us to add side effect such as print out each element.

Unlike lazily-executed terminate operations, a terminal operation is always eagerly executed. The common terminal operations provided by the Stream API include:

* collect()
* reduce()
* forEach()

See the [Stream Javadoc](http://app.getresponse.com/click.html?x=a62b&lc=fvuXG&mc=Iw&s=I3kzkY&u=ml0B&y=d&) for a complete list of terminal operations supported.

**\* Parallel Streams:**

The powerful feature of streams is that stream pipelines may execute either sequentially or in parallel. All collections support the parallelStream() method that returns a possibly parallel stream:

Stream parallelStream = listStudents.parallelStream();

When a stream executes in parallel, the Java runtime divides the stream into multiple sub streams. The aggregate operations iterate over and process these sub streams in parallel and then combine the results.

The advantage of parallel streams is performance increase on large amount of input elements, as the operations are executed currently by multiple threads on a multi-core CPU.

For example, the following code may execute stream operations in parallel:

listStudents.parallelStream()

            .filter(s -> s.getScore() >= 70)

            .sorted()

            .limit(3)

            .forEach(System.out::println);

The Collection’s stream() method returns a sequential stream. We can convert a sequential stream to a parallel stream by calling the **parallel()** method on the current stream. The following example shows a stream executes the sorted operation sequentially, and then execute the filter operation in parallel:

listStudents.stream()

            .sorted()

            .parallel()

            .filter(s -> s.getScore() >= 70)

            .forEach(System.out::println);

**\* Streams and Lambda Expressions:**

As you can see in the above examples, Lambda expressions can be used as arguments in aggregate functions. This allows us to write code more flexibility and more compact. Remember that the parameter in the Lambda expression is implicitly the object being processed in the stream.

Consider the following example:

listStudents.stream()

      .sorted()

      .filter(s -> s.getScore() >= 70)

      .forEach(System.out::println);

Here, we use a Lambda expression in the filter operation and a static method reference in the forEach operation. The s parameter is of type Student because the stream is a sequence of Student objects.

**NOTE:**Some operations can transform a stream of type A to a stream of type B, such as the map operation in the following example:

listStudents.stream()

      .filter(s -> s.getScore() >= 70)

      .map(s -> s.getName())

      .sorted()

      .forEach(name -> System.out.println(name));

In Lambda expressions used with the filter and map operations, the s parameter is of type Student. However the map operation produces a stream of Strings, so the name parameter in the Lambda expression in the forEach operation is of type String. So pay attention to this kind of transformation when using Lambda expressions.

**\* Streams vs. Collections:**

A collection is a data structure that holds elements. Each element is computed before it actually becomes a part of that collection.

On the other hand, a stream is not a data structure. A stream is a pipeline of operations that compute the elements on-demand. Though we can create a stream from a collection and apply a number of operations, the original collection doesn’t change. Hence streams cannot mutate data.

And a key characteristic of streams is that they can transform data, as operations on a stream can produce another data structure, like the map and collect operation as shown in the above examples.