Introducing the Streams API

The Streams API is a set of operations we can perform on a collection, so when we read these operations in our code, we can understand what we're trying to do with the collection data. If you were successful in the "Who Does What?" exercise on the previous page (the complete answers are at the end of this chapter), you should have seen that the names of the operations describe what they do.

java.util.stream.Stream

Stream<T> distinct()
Returns a stream consisting of the distinct elements

Stream<T> filter(Predicate<? super T> predicate)
Returns a stream of the elements that match the given predicate.

Stream<T> limit(long maxSize)
Returns a stream of elements truncated to be no longer than max-Size in length.

<R> Stream<R> map(Function<? super T,? extends R> mapper)
Returns a stream with the results of applying the given function to the elements of this stream.

Stream<T> skip(long n)
Returns a stream of the remaining elements of this stream after discarding the first n elements of the stream.

Stream<T> sorted()
Returns a stream of the elements of this stream, sorted according to natural order.

// more

(These are just a few of the methods in Stream... there are many more.)

These generics do look a little intimidating, but don't panic! We'll use the map method later, and you'll see it's not as complicated as it seems.

Streams, and lambda expressions, were introduced in Java 8.

You don't need to worry too much about the generic types on the Stream methods; you'll see that using Streams "just works" the way you'd expect.

In case you are interested:

- **<T>** is usually the Type of the object in the stream.
- **<R>** is usually the type of the Result of the method.

Getting a result from a Stream

Yes, we've thrown a **lot** of new words at you: *streams*; *intermediate operations*; *terminal operations*... And we still haven't told you what streams can do!

To start to get a feel for what we can do with streams, we going to show code for a simple use of the Streams API. After that, we'll step back and learn more about what we're seeing here.

```
List<String> strings = List.of("I", "am", "a", "list", "of", "Strings");

Stream<String> stream = strings.stream();
Stream<String> limit = stream.limit(4);
long result = limit.count();
System.out.println("result = " + result);
```

Call the count terminal operator, and store the output in a variable called result

```
File Edit Window Help WellDuh
%java LimitWithStream

result = 4
```

This works, but it's not very useful. One of the most common things to do with Streams is put the results into another type of collection. The API documentation for this method might seem intimidating with all the generic types, but the simplest case is straightforward:

The stream contained Strings, so the output object will also contain Strings.

Terminal operation that will collect the output into some sort of Object.

This method returns a Collector that will output the results of the stream into a List.

```
List<String> result = limit.collect(Collectors.toList());
```

The toList Collector will output the results as a List.

A helpful class that contains methods to return common Collector implementations.

```
System.out.println("result = " + result);
```

```
File Edit Window Help FinallyAResult
%java LimitWithStream

result = [I, am, a, list]
```

We'll see collect() and the Collectors in more detail later.

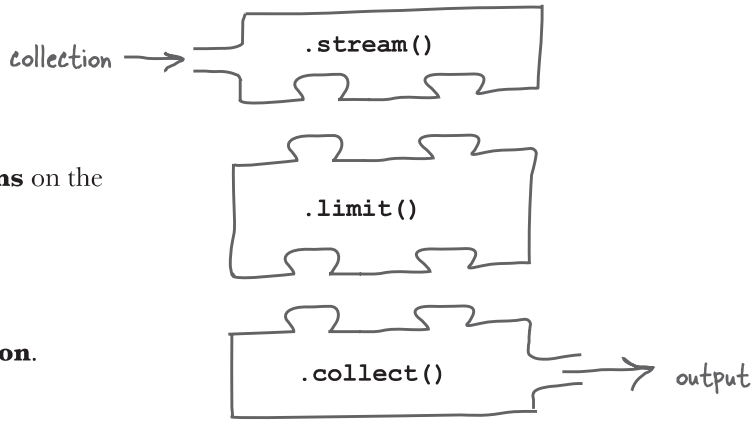
For now, `collect(Collectors.toList)` is a magic incantation to get the output of the stream pipeline in a List.

Finally, we have a result that looks like something we would have expected: we had a List of Strings, and we asked to **limit** that list to the first four items and then **collect** those four items into a new List.

Stream operations are building blocks

We wrote a lot of code just to output the first four elements in the list. We also introduced a lot of new terminology: streams, intermediate operations, and terminal operations. Let's put all this together: you create a **stream pipeline** from three different types of building blocks.

1. Get the Stream from a **source** collection.
2. Call zero or more **intermediate operations** on the Stream.
3. Output the results with a **terminal operation**.



You need at least the ***first*** and ***last*** pieces of the puzzle to use the Streams API. However, you don't need to assign each step to its own variable (which we were doing on the last page). In fact, the operations are designed to be **chained**, so you can call one stage straight after the previous one, without putting each stage in its own variable.

On the last page, all the building blocks for the stream were highlighted (stream, limit, count, collect). We can take these building blocks and rewrite the limit-and-collect operation in this way:

```
List<String> strings = List.of("I", "am", "a", "list", "of", "Strings");

List<String> result = strings.stream()
                             .limit(4)
                             .collect(Collectors.toList());

System.out.println("result = " + result);
```

Get the stream for the collection

Set a limit to return a maximum of 4 results from the stream

Formatted to align each operation directly underneath the one above, to clearly show each stage.

Returns the results of the operation as a List