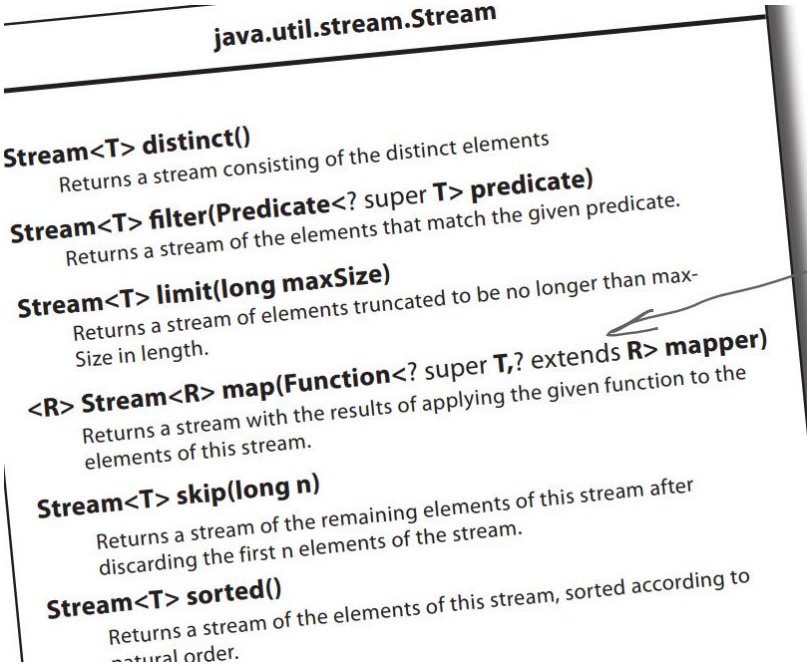
CHAPTER 12

Lambdas and Streams: What, Not How (EXERCISE)`



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.

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);
```

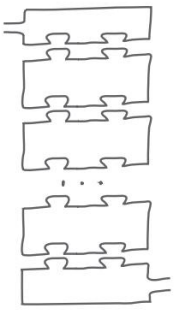
```
File Edit Window Help FinallyAResult
%java LimitWithStream

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.

Building blocks can be stacked and combined

Every intermediate operation acts on a Stream and returns a Stream. That means you can stack together as many of these operations as you want, before calling a terminal operation to output the results.



The source, the intermediate operation(s), and the terminal operation all combine to form a Stream Pipeline. This pipeline represents a query on the original collection.

This is where the Streams API becomes really useful. In the earlier example, we needed three building blocks (stream, limit, collect) to create a shorter version of the original List, which may seem like a lot of work for a simple operation.

But to do something more complicated, we can stack together multiple operations in a single **stream pipeline**.

EXERCISE

1.MIXED MESSAGES

1. for (int i = 1; i < nums.size(); i++)
   output += nums.get(i) + " ";

Ans - 2 3 4 5

2. for (int i = 0; i <= nums.size(); i++)
   output += nums.get(i) + " ";

Ans –

[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]

3. for (int i = 0; i <= nums.length; i++)
   output += nums.get(i) + " ";

Ans - Compiler error

4. for (Integer num : nums)
   output += nums;

Ans- Exception thrown

2.WHO DOES WHAT

filter ->Only allows elements that match the given criteria to remain in the stream
skip -> This is the number of elements at the start of the Stream that will not be processed
limit -> Sets the maximum number of elements that can be output from this Stream
distinct ->Use this to make sure duplicates are removed
sorted -> States the result of the stream should be ordered in some way
map -> Changes the current element in the stream into something else
dropWhile -> Will only process elements while the given criteria is true
takeWhile -> While a given criteria is true, will not process elements

3.CODE MAGNETS

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

public class CoffeeOrder {
   public static void main(String[] args) {
      List<String> coffees = List.of(
         "Cappuccino", "Americano", "Espresso",
         "Cortado", "Mocha", "Cappuccino",
         "Flat White", "Latte"
      );

      List<String> coffeesEndingInO = coffees.stream()
                                             .filter(s -> s.endsWith("o"))
                                             .sorted()
                                             .distinct()
                                             .collect(Collectors.toList());

      System.out.println(coffeesEndingInO);
   }
}
```

4.BE THE COMPILER

```
Runnable r = () -> System.out.println("Hi!");

Consumer<String> c = s -> System.out.println(s);

Function<String, Integer> f = s -> s.length();

Supplier<String> s = () -> "Some string";
```

5.SHARPEN YOUR PENCIL

BiPredicate,Function and Action Listener has only one abstract classes.

6.FIVE MINUTES MYSTERY

Alex should have sorted the coffee first and then mapped with the name.This is because by sorting first,the weakest coffee would have been kept first and such that during mapping the initial coffee would have been displayed.But Alex did a vice versa and so she received Americano as first

7.POOL PUZZLE

```
public class StreamPuzzle {
   public static void main(String[] args) {
      SongSearch songSearch = new SongSearch();
      songSearch.printTopFiveSongs();
      songSearch.search("The Beatles");
      songSearch.search("The Beach Boys");
   }
}

class SongSearch {
   private final List<Song> songs = new JukeboxData.Songs().getSongs();

   void printTopFiveSongs() {
      List<String> topFive = songs.stream()
         .sorted(Comparator.comparingInt(Song::getTimesPlayed))
         .map(song -> song.getTitle())
         .limit(5)
         .collect(Collectors.toList());
      System.out.println(topFive);
   }

   void search(String artist) {
      Optional<Song> result = songs.stream()
         .filter(song -> song.getArtist().equals(artist))
         .findFirst();
      if (result.isPresent()) {
         System.out.println(result.get().getTitle());
      } else {
         System.out.println("No songs found by: " + artist);
      }
   }
}
```