

The Stream Pipeline

In the last video, I left off with our first example of using a stream in code.

This entire chain of operations is what's called a Stream Pipeline.

```
bingoPool.stream()  
    .limit(15)  
    .filter(s -> s.indexOf('G') == 0 || s.indexOf("O") == 0)  
    .map(s -> s.charAt(0) + "-" + s.substring(1))  
    .sorted()  
    .forEach(s -> System.out.print(s + " "));
```


The Pipeline starts with a Source

The source of the stream is where the data elements are coming from.

In our example, it's coming from a list, bingoPool.

All pipelines start with a stream, so in this example, we need to call the stream method on the bingoPool list to get a stream.

There are a lot of other kinds of sources, and ways to create new streams, including infinite streams.

```
bingoPool.stream()   
    .limit(15)  
    .filter(s -> s.indexOf('G') == 0 || s.indexOf("O") == 0)  
    .map(s -> s.charAt(0) + "-" + s.substring(1))  
    .sorted()  
    .forEach(s -> System.out.print(s + " "));
```

The Pipeline ends with a Terminal Operation

Stream Pipelines end in a **terminal operation**.

A terminal operation is **required**.

```
bingoPool.stream()  
    .limit(15)  
    .filter(s -> s.indexOf('G') == 0 || s.indexOf("O") == 0)  
    .map(s -> s.charAt(0) + "-" + s.substring(1))  
    .sorted()  
    .forEach(s -> System.out.print(s + " "));
```

TERMINAL OPERATION

The Intermediate Operations

Everything else, between the source and the terminal operation is an intermediate operation

An intermediate operation is **not required**.

You can have a pipeline that just has a source and terminal operation, and these are quite common.

Every intermediate operation processes elements on the stream, and returns a stream as a result.

INTERMEDIATE
OPERATIONS

```
bingoPool.stream()
    .limit(15)
    .filter(s -> s.indexOf('G') == 0 || s.indexOf("O") == 0)
    .map(s -> s.charAt(0) + "-" + s.substring(1))
    .sorted()
    .forEach(s -> System.out.print(s + " "));
```


Streams are Lazy

Without worrying about semantics, I want you to imagine the stream pipeline as a black box.

The source is your input, the result of your terminal operation is your output.

Everything in between, isn't going to happen until something tells that terminal operation to start.

What actually happens in that black box, may not happen exactly as you've described it, or in the order you've specified.

Execution of the intermediate operations is dependent, first on a terminal operation being specified, and second on an optimization process occurring.

Stream computations are optimized

What this means is that your stream pipeline is kind of a workflow suggestion.

Before the process begins, the stream implementation will perform an evaluation, to optimize the means to the end.

It will determine the best way to get the elements needed, and the most efficient way to process them, to give you the result you've asked for.

The result will be consistent each time, but the process to get there is not guaranteed to be.

Optimizations may change the order of the intermediate operations, it may combine operations, or even skip them altogether.

For this reason, **you should avoid side effects in your intermediate operations.**

You can't reuse a stream

Once you invoke a terminal operation on a stream, you can think of the pipeline as being opened, and the flow beginning.

The flow is allowed to continue until all processes have been performed and a result produced.

At that point, the valve is shut, and the pipeline closed.

You can't turn it back on, or reuse it for a new source.

If you want to do the same sort of thing with a different variable for one of the intermediate operations, you'd need to set up a new pipeline.