Streams

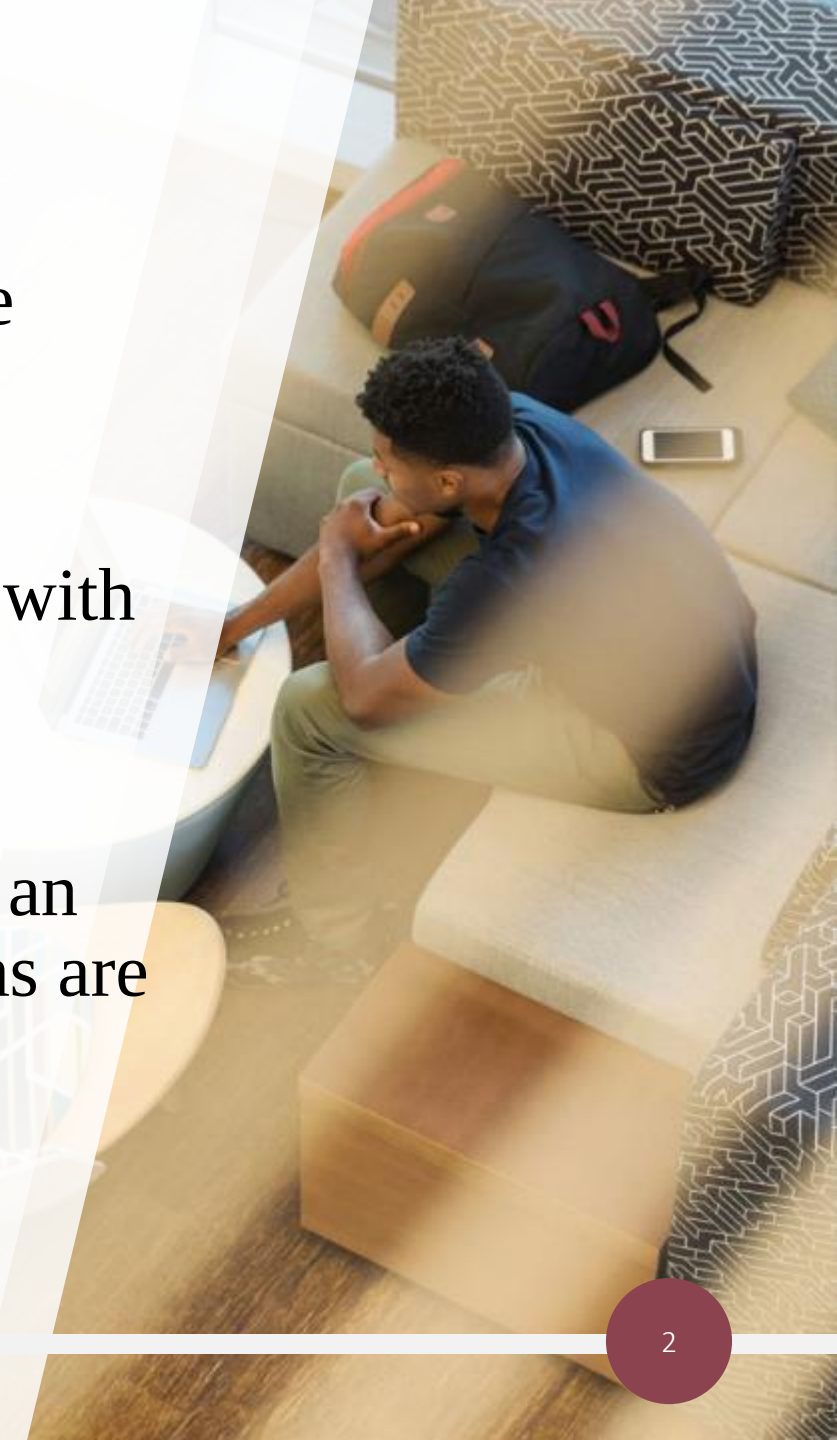
Introduction and Stream Pipelines

What is a Stream?

- Like lambdas and functional interfaces, streams were another new addition in Java 8.

- A *stream* is a sequence of data that can be processed with operations.

- Streams are **not** another way of organising data, like an array or a *Collection*. Streams do not hold data; streams are all about processing data efficiently.



What is a Stream?

- While streams make code more concise, their big advantage is that streams, by using a pipeline, can, in certain situations, greatly improve the efficiency of data processing.

- The real power of streams comes from the multiple intermediate operations you can perform on the stream.

The Pipeline

- A *stream pipeline* consists of the operations that run on a stream to produce a result.

- There are 3 parts to a stream pipeline:
  a) Source – where the stream comes from e.g. array, collection or file.
  b) Intermediate operations – transforms the stream into another one. There can as few or as many as required.
  c) Terminal operation – required to start the whole process and produces the result. Streams can only be used once i.e. streams are no longer usable after a terminal operation completes (re-generate the stream if necessary).

The Pipeline

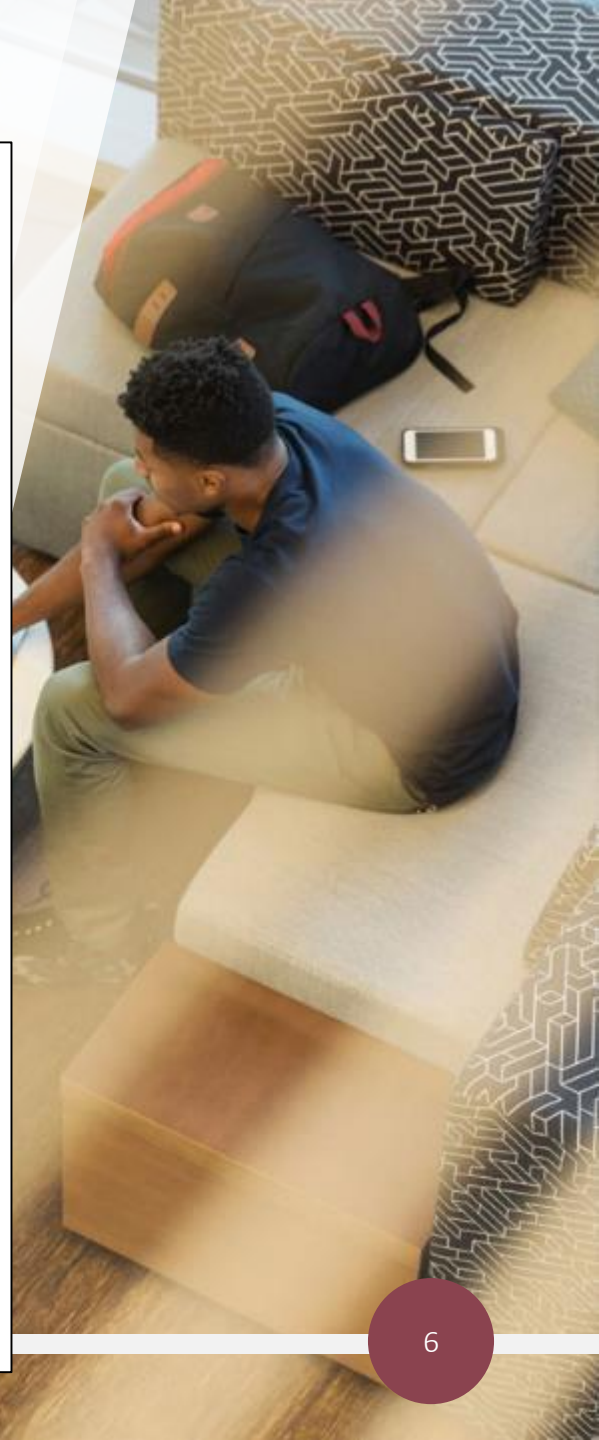
- *filter()* is an intermediate operation and as such can filter the stream and pass on the filtered stream to the next operation (another intermediate operation or a terminal operation).

- *count()* and *forEach()* are both terminal operations that end the stream.

- The pipeline operations are the way in which we specify how and in what order we want the data in the source manipulated. Remember, streams don't hold any data.

The Pipeline

```
/* Output:
    98.4
    100.2
    100.2
    87.9
    102.8
    102.8
    Number of temps > 100 is: 2
*/
List<Double> temps = Arrays.asList(98.4, 100.2, 87.9, 102.8);
System.out.println("Number of temps > 100 is: "+
        temps
            .stream() // create the stream
            .peek(System.out::println) // show the value
            .filter(temp -> temp > 100)// filter it
            .peek(System.out::println) // show the value
            .count());// 2
```



Streams

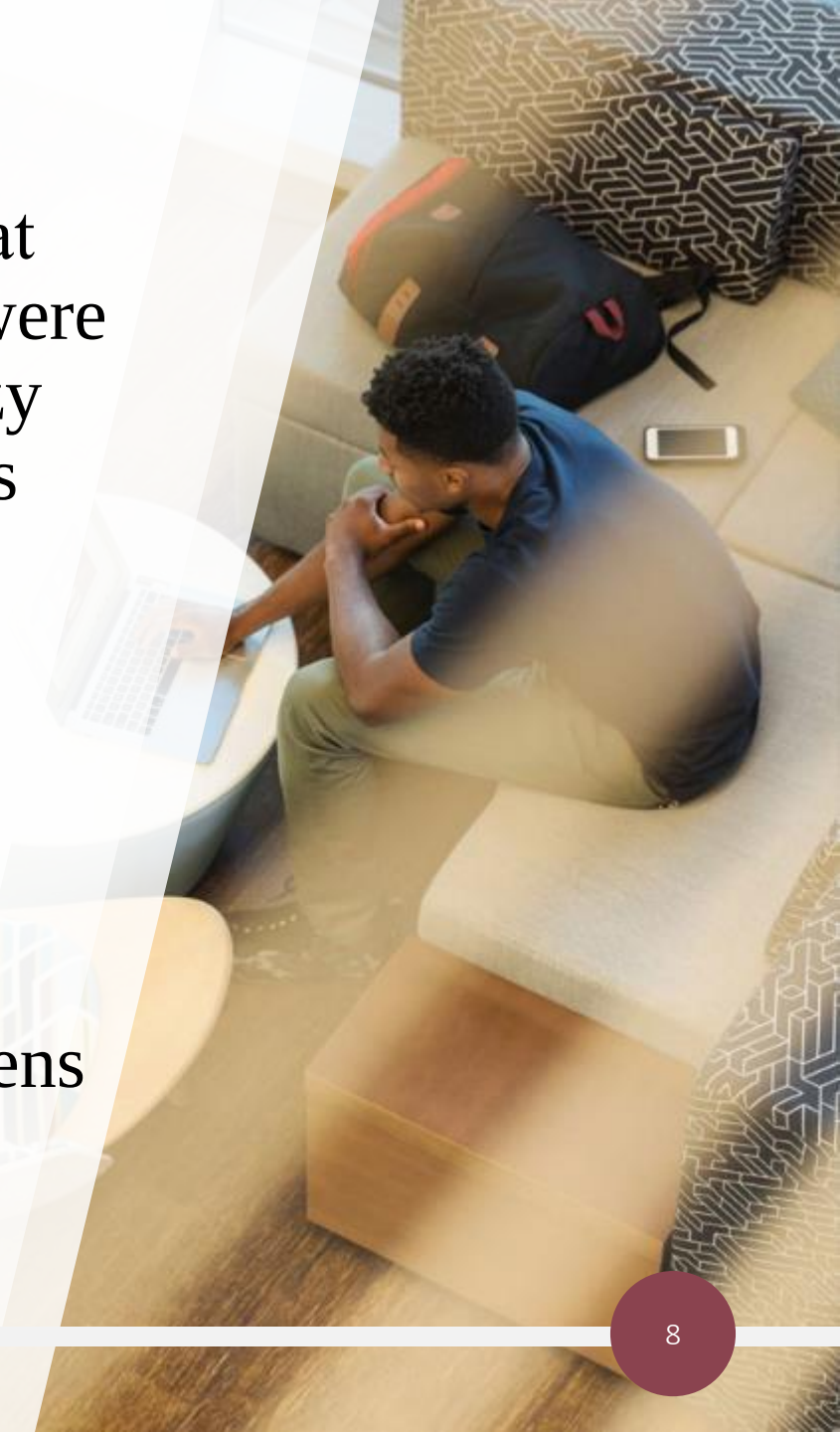
Streams are Lazy

Streams are Lazy

- The principle of “lazy” evaluation is that you get what you need only when you need it. For example, if you were displaying 10,000 records to a user, the principle of lazy evaluation would be to retrieve 50 and while the user is viewing these, retrieve another 50 in the background.

- “Eager” evaluation would be to retrieve all 10,000 records in one go.

- With regard to streams, this means that nothing happens until the terminal operation occurs.

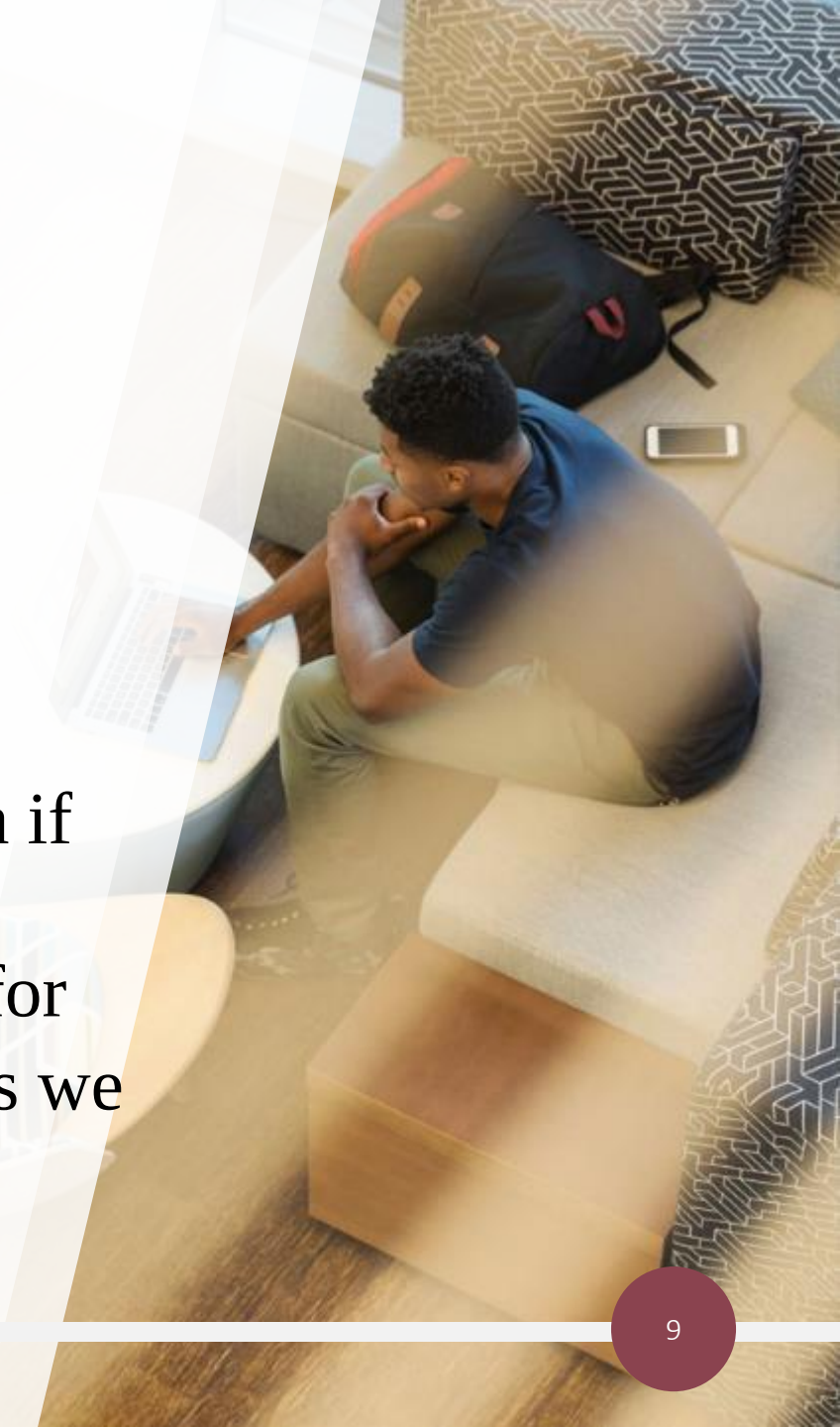


Streams are Lazy

- The pipeline specifies what operations we want performed (on the source) and in which order.

- This enables the JDK to reduce operations whenever possible.

- For example, why run an operation on a piece of data if the operation is not required:
  - ➢we have found the data element we were looking for
  - ➢we may have a limit set on the number of elements we want to operate on



```
/*  Each element moves along the chain vertically:
        filter: Alex
        forEach: Alex
        filter: David
        forEach: David
        filter: April
        forEach: April
        filter: Edward
        forEach: Edward     */
Stream.of("Alex", "David", "April", "Edward")
        .filter(s -> {
            System.out.println("filter: "+s);
            return true;
        })
        .forEach(s -> System.out.println("forEach: "+s));
```

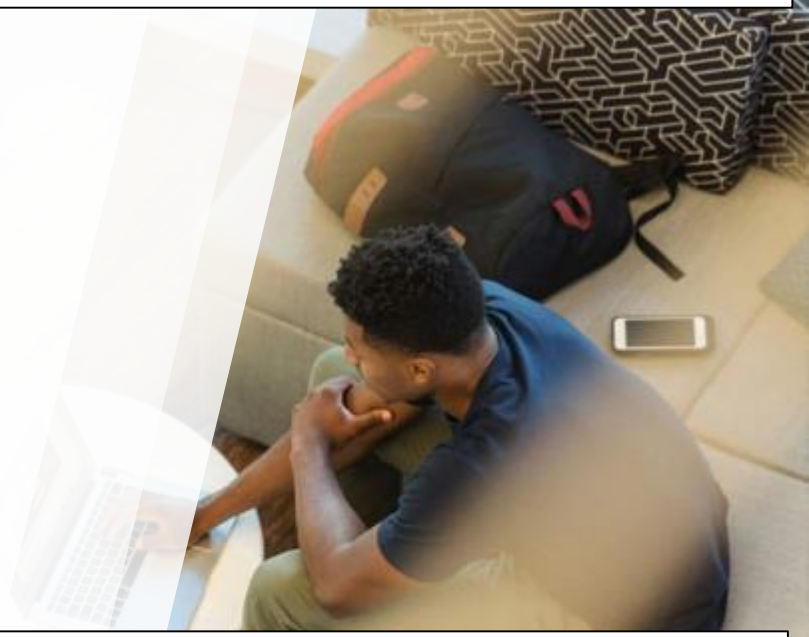
Streams are Lazy

```
/* This can help in reducing the actual number of operations - instead of
    mapping "Alex", "David", "April" and "Edward" and then anyMatch() on
   "Alex" (5 operations in total), we process the elements vertically resulting in
   only 2 operations. While this is a small example, it shows the benefits to be
   had if we had millions of data elements to be processed.
        map: Alex
        anyMatch: ALEX     */
Stream.of("Alex", "David", "April", "Edward")
        .map(s -> {
            System.out.println("map: "+s);
            return s.toUpperCase();
        })
        .anyMatch(s -> { // ends when first true is returned (Alex)
            System.out.println("anyMatch: "+s);
            return s.startsWith("A");
        });
```

Streams are Lazy

Streams are Lazy

```
List<String> names =
        Arrays.asList("April", "Ben", "Charlie",
                      "David", "Benildus", "Christian");
names.stream()
    .peek(System.out::println)
    .filter(s -> {
        System.out.println("filter1 : "+s);
        return s.startsWith("B") || s.startsWith("C"); } )
    .filter(s -> {
        System.out.println("filter2 : "+s);
        return s.length() > 3; } )
    .limit(1) // intermediate operation  Stream<T> limit(long)
    .forEach(System.out::println);  // terminal operation
```



```
April                 - peek
filter1 : April       - filter1 removes April
Ben                   - peek
filter1 : Ben         - filter1 passes Ben on
filter2 : Ben         - filter2 removes Ben
Charlie               - peek
filter1 : Charlie     - filter1 passes Charlie on
filter2 : Charlie     - filter2 passes Charlie on
Charlie               - forEach()

Note: limit(1) means David, Benildus or Christian are not
      processed at all i.e. none of them appear in the output
      via "peek()"
```