Streams, Collectors and Optionals for Data Processing in Java 8

Connecting Streams to Custom Sources: The Spliterator Pattern

José Paumard

@JosePaumard | blog.paumard.org

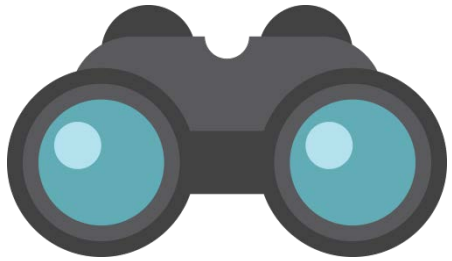
What Is This Course About?

- Advanced data processing topics
- To connect streams on custom sources
- Advanced stream patterns
- Parallel streams
- The Collector API
- To create custom collectors

What You Will Learn

- To connect streams to non-standard sources
- To use `flatMap`, streams of numbers
- To efficiently use parallel streams
- To master the concept of Optional
- To use and create Collectors

Course Overview



- The spliterator pattern
- Using `flatMap`, stream of numbers
- Parallel streams
- Using Optionals to process data
- Collectors
- Custom collectors

Targeted Audience

This is a Java course

Good knowledge of the language

Basic knowledge of the main APIs

Generics, collection API

Lambda, basic Stream API

Agenda of This Module

What is a Spliterator?

Building our own spliterator

Using it to regroup the lines of a text file

What Is a Spliterator?

The object on which a Stream is built

What Is a Spliterator?

- Let us check the Collection.stream method

```
default Stream<E> stream() {
   return StreamSupport.stream(spliterator(), false);
}
```

```
default Spliterator<E> spliterator() {
   return Spliterators.spliterator(this, 0);
}
```

- The returned stream is built on a *spliterator*
- A new interface in Java 8, that models the access to the data for a Stream

What Is a Spliterator?

- ArrayList and HashSet have different spliterators
- For ArrayList:

```
public Spliterator<E> spliterator() {
    return new ArrayListSpliterator<>(this, 0, -1, 0);
}
```

- For HashSet:

```
public Spliterator<E> spliterator() {
    return new HashMap.KeySpliterator<E,Object>(map, 0, -1, 0, 0);
}
```

What Is a Spliterator?

- A Stream is divided in two things:
- An object to access the data, this is the Spliterator
  - It is meant to be overriden to suit our needs
- An object to handle the processing of the data, this is the ReferencedPipeline
  - It is a very complex object, we do not need to override it

Can We Build Our Own Spliterator?

- This answer is yes!
- Let us examine this interface: four abstract methods

```
public interface Spliterator<T> Spliterator {

   boolean tryAdvance(Consumer<? super T> action) ;

   Spliterator<T> trySplit() ;

   long estimateSize();

   int characteristics();
}
```

Can We Build Our Own Spliterator?

- This answer is yes!
- Let us examine this interface: three main default methods

```
public interface Spliterator<T> Spliterator {

   default void forEachRemaining(Consumer<? super T> action) {
      do { } while (tryAdvance(action));
   }

   default long getExactSizeIfKnown() {
      return (characteristics() & SIZED) == 0 ? -1L : estimateSize();
   }
}
```

Can We Build Our Own Spliterator?

- This answer is yes!
- Let us examine this interface: three main default methods

```
public interface Spliterator<T> Spliterator {

   default boolean hasCharacteristics(int characteristics) {
      return (characteristics() & characteristics) == characteristics;
   }
}
```

Can We Build Our Own Spliterator?

- This answer is yes!
- Let us examine this interface: a set of constants

```
public interface Spliterator<T> Spliterator {

   public static final int ORDERED      = 0x00000010;
   public static final int DISTINCT     = 0x00000001;
   public static final int SORTED       = 0x00000004;
   public static final int SIZED        = 0x00000040;
   public static final int NONNULL      = 0x00000100;
   public static final int IMMUTABLE    = 0x00000400;
   public static final int CONCURRENT   = 0x00001000;
   public static final int SUBSIZED     = 0x00004000;
}
```

The ArrayList Example

- Let us see how the Spliterator is implemented in ArrayList

```
static final class ArrayListSpliterator<E> implements Spliterator<E> {

   private final ArrayList<E> list;
   private int index; // current index, modified on advance/split
   private int fence; // -1 until used; then one past last index
   private int expectedModCount; // initialized when fence set
}
```

The ArrayList Example

- Let us see how the Spliterator is implemented in ArrayList

```
public long estimateSize() {

   return (long) (getFence() - index);
}
```

The ArrayList Example

- Let us see how the Spliterator is implemented in ArrayList

```
public boolean tryAdvance(Consumer<? super E> action) {

    int hi = getFence(), i = index;
    if (i < hi) {
        index = i + 1;
        E e = (E)list.elementData[i];
        action.accept(e);
        return true;
    }
    return false;
}
```

The ArrayList Example

- Let us see how the Spliterator is implemented in ArrayList

```
public ArrayListSpliterator<E> trySplit() {

    int hi = getFence(), lo = index, mid = (lo + hi) >>> 1;
    return (lo >= mid) ? null : // divide range in half unless too small
    new ArrayListSpliterator<E>(list, lo, index = mid, expectedModCount);
}
```

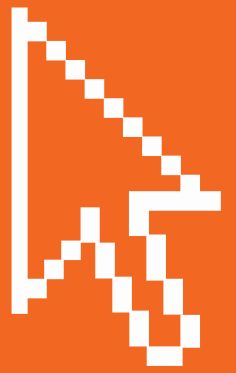
The ArrayList Example

- Implementing a Spliterator requires a good knowledge of the underlying source of data
- But it is not very complex!

Live Coding

Handling a file of people

Where each person is written on more than one line



Live Coding Summary

- We saw how to create a complex Spliterator from a given Spliterator
- We saw how to use it to process a text file by regrouping the lines in a clean way

Summary

- Advanced patterns to build custom streams on non-conventional sources of data
- Spliterator, how to create your own spliterators
- Use case: reading a text file by regrouping its lines