Java Streams Intermediate Operation mapMulti()

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

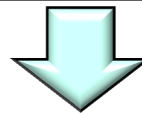
Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Part of the Lesson

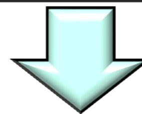
- Understand the structure & functionality of stream aggregate operations
  - Intermediate operations
    - map() & mapToInt()
    - filter() & flatMap()
    - mapMulti()

stream()



Input x

Stream mapMulti(BiConsumer<T, Consumer<R>> mapper)



Output f(x)

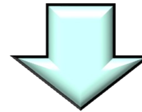
R collect(Collector<...> collector)

This stateless, run-to-completion operation can replace map()/filter() combinations

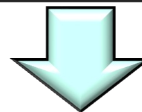
Learning Objectives in this Part of the Lesson

- Understand the structure & functionality of stream aggregate operations
  - Intermediate operations
    - map() & mapToInt()
    - filter() & flatMap()
    - mapMulti()
      - We also discuss how mapMulti() can work around limitations with flatMap() when used with parallel streams

parallelStream()



Stream mapMulti(BiConsumer<T, Consumer<R>> mapper)



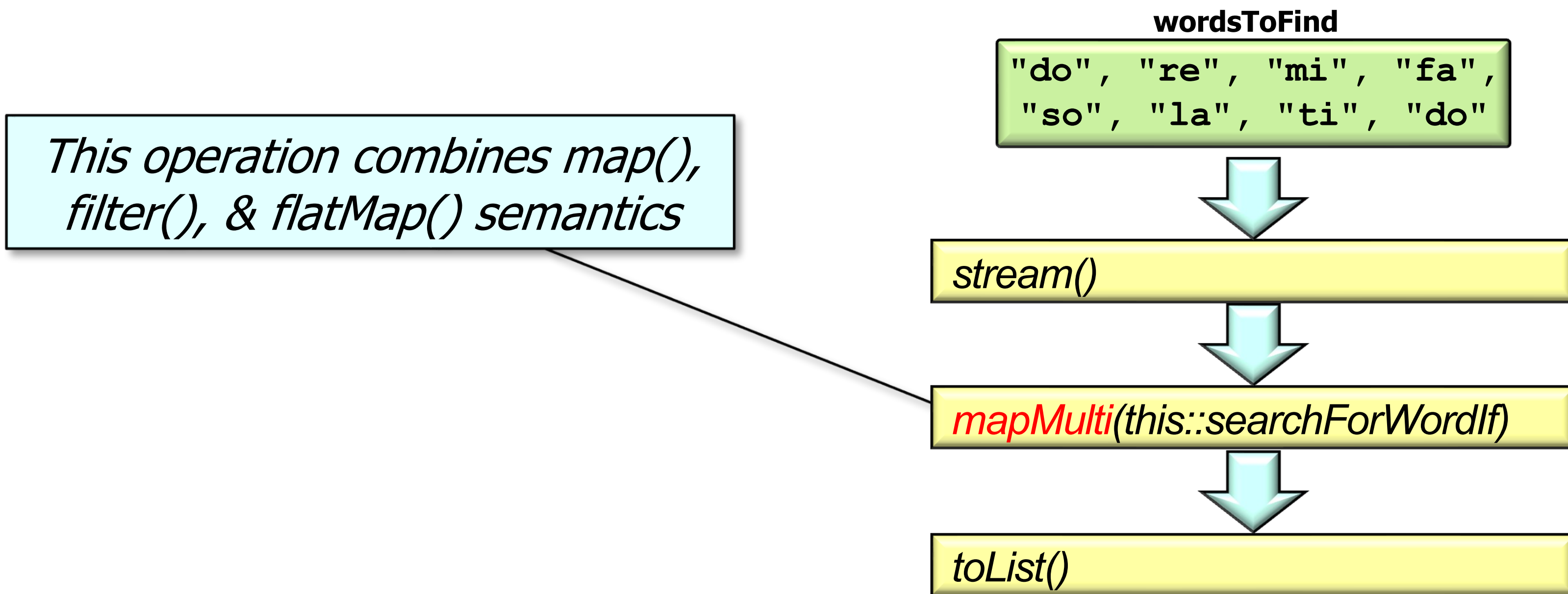
R collect(Collector<...> collector)

mapMulti() is available in the Streams API in Java 16+

Overview of the mapMulti() Intermediate Operation

Overview of the mapMulti() Intermediate Operation

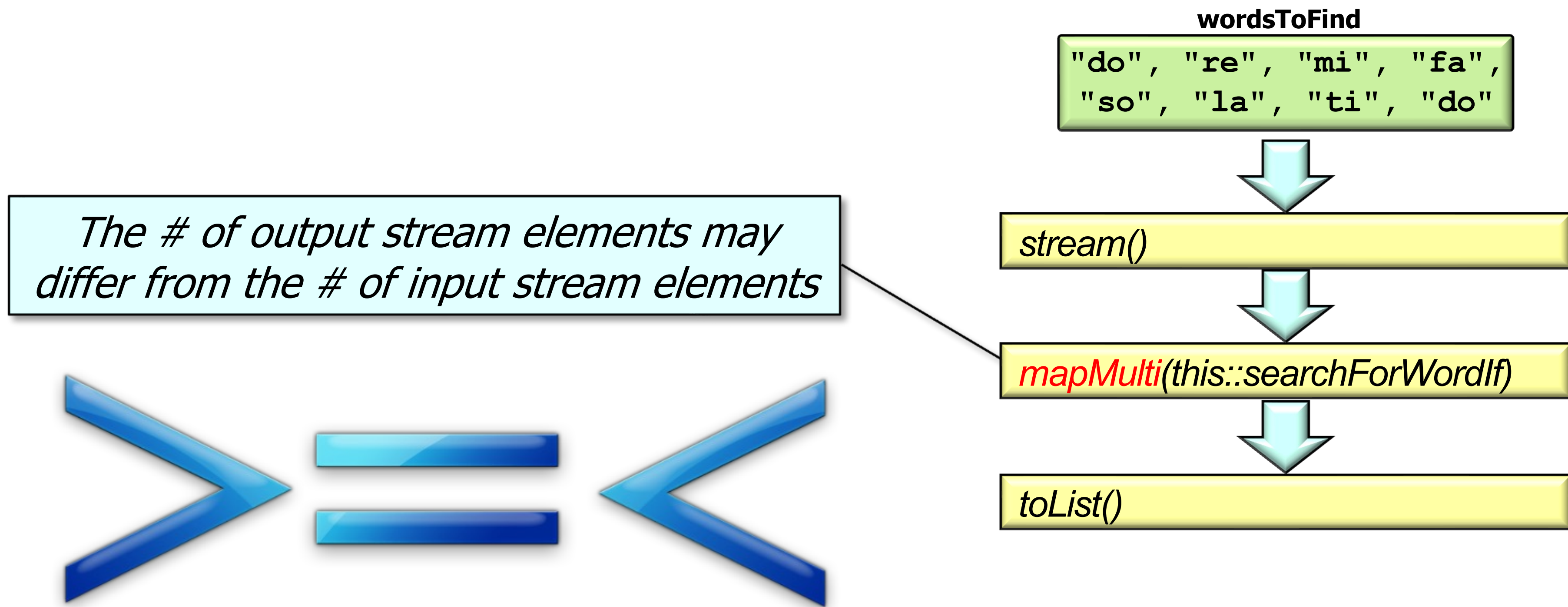
- Returns a stream consisting of the results of replacing each element of this stream with multiple elements, specifically zero or more elements



See docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/stream/Stream.html#mapMulti

Overview of the mapMulti() Intermediate Operation

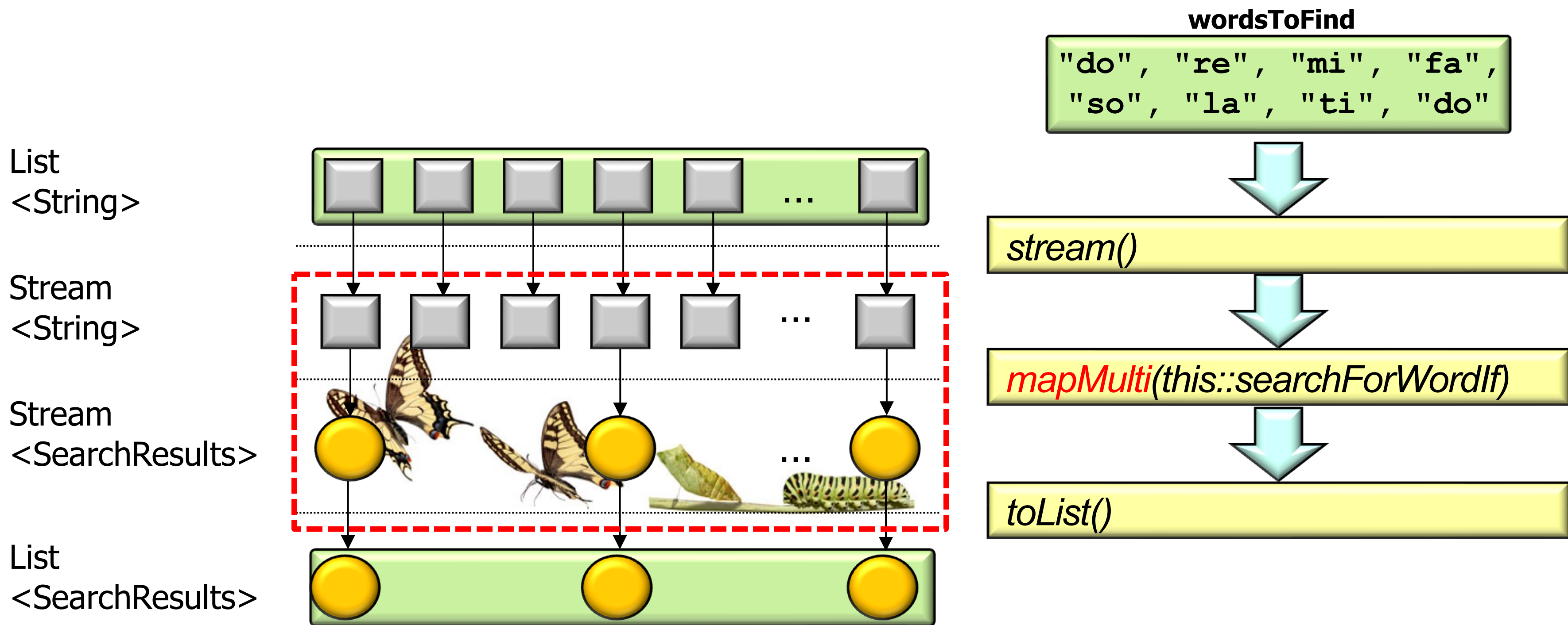
- Returns a stream consisting of the results of replacing each element of this stream with multiple elements, specifically zero or more elements



These semantics are similar to flatMap()

Overview of the mapMulti() Intermediate Operation

- Returns a stream consisting of the results of replacing each element of this stream with multiple elements, specifically zero or more elements

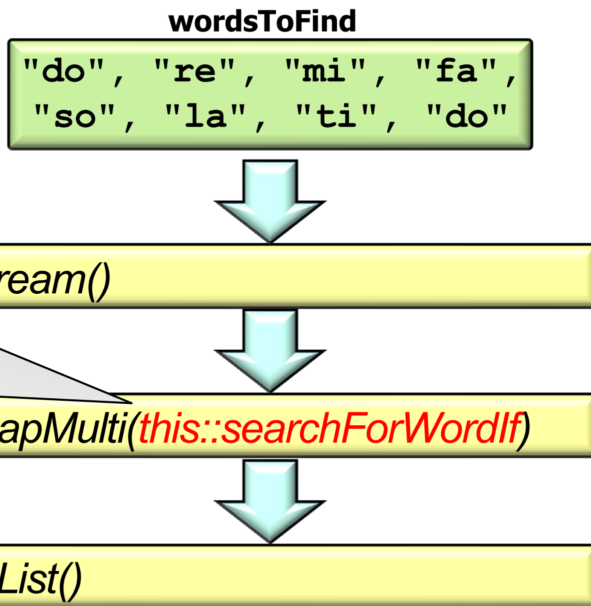


mapMulti() *may* filter & transform the type of elements it processes

Overview of the mapMulti() Intermediate Operation

- Returns a stream consisting of the results of replacing each element of this stream with multiple elements, specifically zero or more elements

```
void searchForWordIf
   (String word,
    Consumer<SearchResults>
      consumer) {
     var result =
       searchForWord(word);
     if (!result.isEmpty())
         consumer.accept(result);
}
```



Only update the consumer if there's a match

Eliminates the need for a separate filter() intermediate operation

mapMulti() Can Overcome Limitations with flatMap()

mapMulti() Can Overcome Limitations with flatMap()

- A limitation with the flatMap() implementation forces sequential processing

BEWARE!

```
<R> Stream<R> flatMap
  (Function<? super P_OUT,
   ? extends Stream<? extends R>>
   mapper) {
 ...
  public void accept(P_OUT u) {
    try(Stream<? extends R> result
        = mapper.apply(u)) {
    if (result != null) {
      if (...) {
        result
        .sequential()
        .forEach(downstream);
      ...
}
```

This code always runs sequentially for "inner streams" that use flatMap()

See stackoverflow.com/questions/45038120/parallel-flatmap-always-sequential/66386078

mapMulti() Can Overcome Limitations with flatMap()

- A limitation with the flatMap() implementation forces sequential processing

```
IntStream
    .rangeClosed(1, outerCount)
    .boxed()
    .parallel()

    .flatMap(innerCount -> IntStream
        .rangeClosed(1, innerCount)
        .boxed()
        .parallel())

    .anyMatch(...);
```

Due to a limitation with flatMap() this inner stream will always run sequentially, even though it is explicitly designated as .parallel()

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex35

mapMulti() Can Overcome Limitations with flatMap()

- A workaround is to replace flatMap() with mapMulti()

```
var result = IntStream
    .rangeClosed(1, outerCount)
    .boxed()
    .parallel()

    .mapMulti((innerCount,
               consumer) -> {
      int result = IntStream
        .rangeClosed(1, innerCount)
        .parallel()
        .mapMulti((i, c) -> ...)
        .sum();
      consumer.accept(result);})

    .allMatch(...);
```

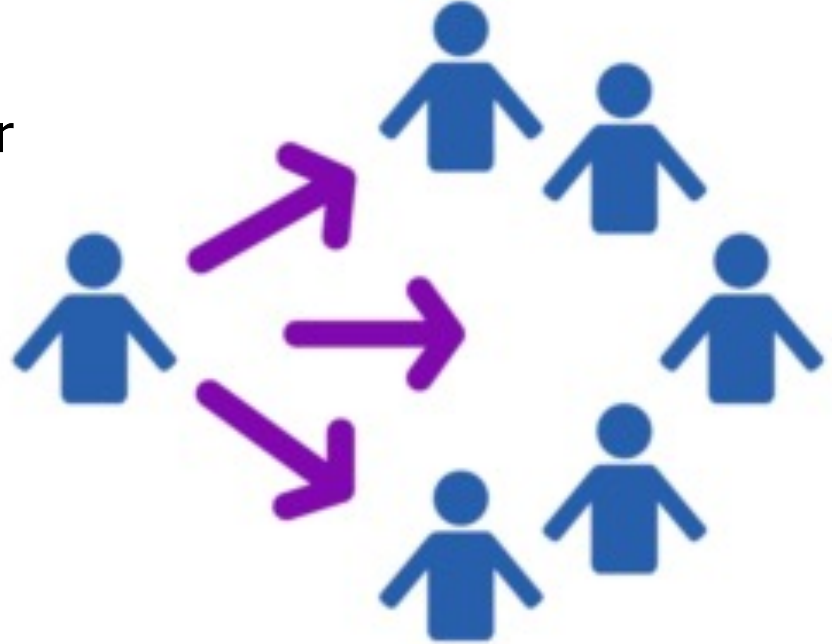
This inner stream now runs in parallel, as intended

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex35

When to Apply mapMulti()

When to Apply mapMulti()

- Variable number of output elements
  - If you have a single input element that can result in a variable number of output elements, mapMulti() is more suitable than map()



See www.baeldung.com/java-mapmulti

When to Apply mapMulti()

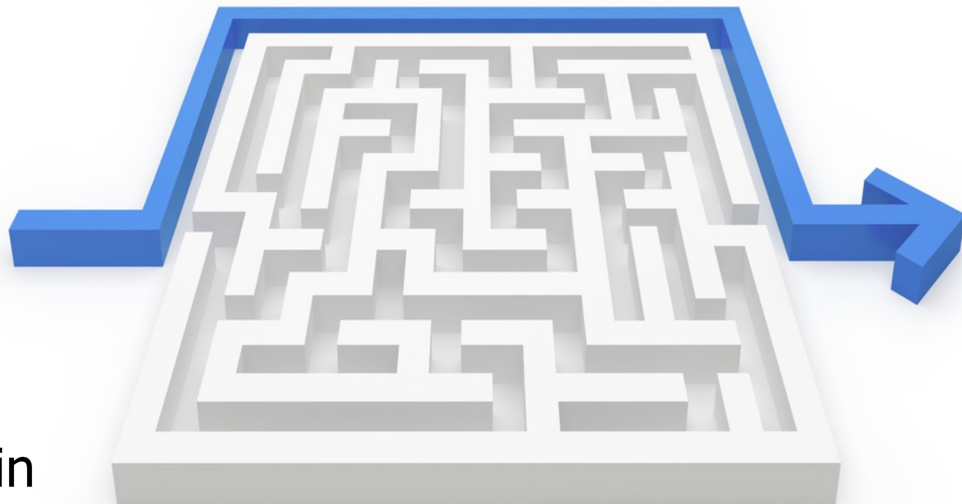
- Variable number of output elements
- Avoiding nulls & filters
  - If your transformation can result in null values that you would otherwise have to filter out, mapMulti() can handle this without needing a separate filter() call

When to Apply mapMulti()

- Variable number of output elements
- Avoiding nulls & filters
- Avoiding intermediate collections
  - With flatMap(), you often need to create intermediate collections when generating multiple output elements for a single input element
    - mapMulti() avoids this by emitting elements directly to the resulting stream

When to Apply mapMulti()

- Variable number of output elements
- Avoiding nulls & filters
- Avoiding intermediate collections
- Avoiding complex transformations
  - If your operation doesn't fit neatly into a map(), filter(), and/or flatMap() operations—or must chain multiple such operations together — mapMulti() is more flexible



End of Java Streams Intermediate Operation mapMulti()