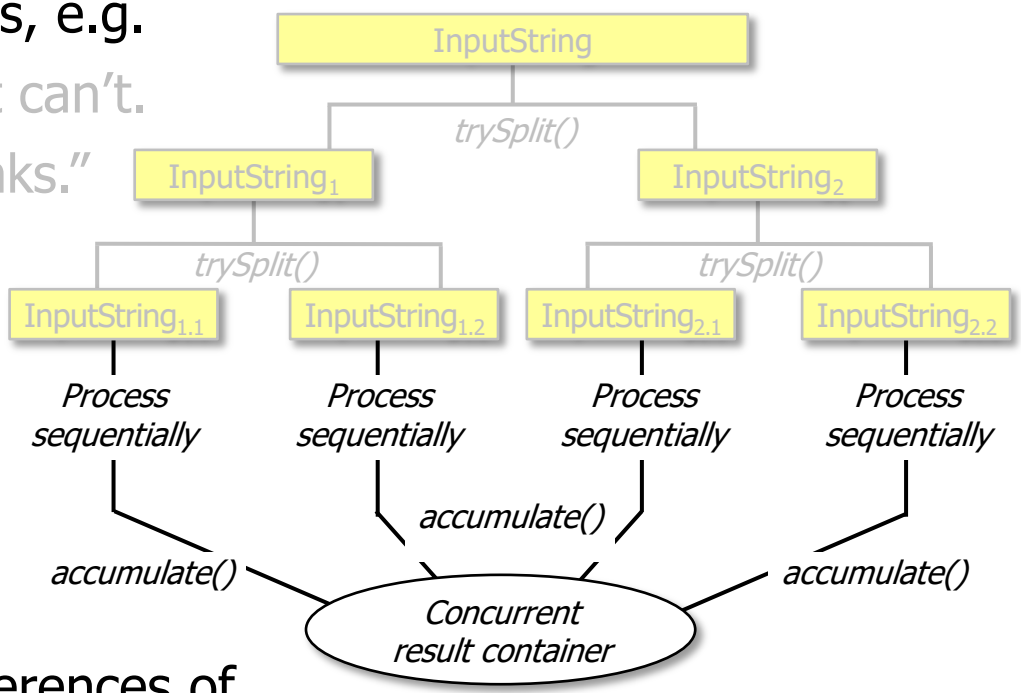
Java Parallel Stream Internals

Non-Concurrent and Concurrent Collectors (Part I)

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

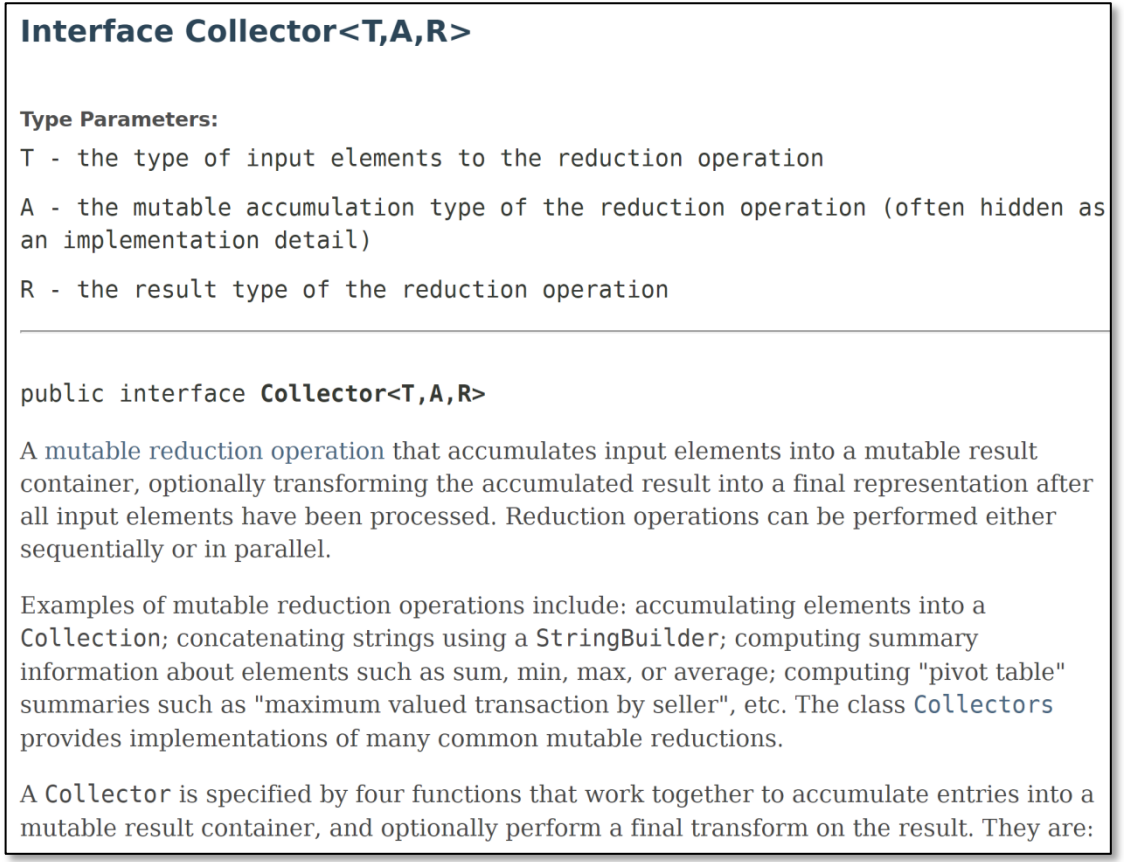
- Understand parallel stream internals, e.g.
  - Know what can change and what can't.
  - Partition a data source into "chunks."
  - Process chunks in parallel via the common ForkJoinPool.
  - Configure the Java parallel stream common ForkJoinPool.
  - Perform a reduction to combine partial results into a single result.
- Recognize key behaviors and differences of non-concurrent and concurrent collectors.



Overview of Concurrent and Non-Concurrent Collectors

Overview of Concurrent and Non-Concurrent Collectors

- Collector defines an interface whose implementations can accumulate input elements in a mutable result container.



Interface Collector<T,A,R>

Type Parameters:

`T` - the type of input elements to the reduction operation

`A` - the mutable accumulation type of the reduction operation (often hidden as an implementation detail)

`R` - the result type of the reduction operation

`public interface` **`Collector<T,A,R>`**

A mutable reduction operation that accumulates input elements into a mutable result container, optionally transforming the accumulated result into a final representation after all input elements have been processed. Reduction operations can be performed either sequentially or in parallel.

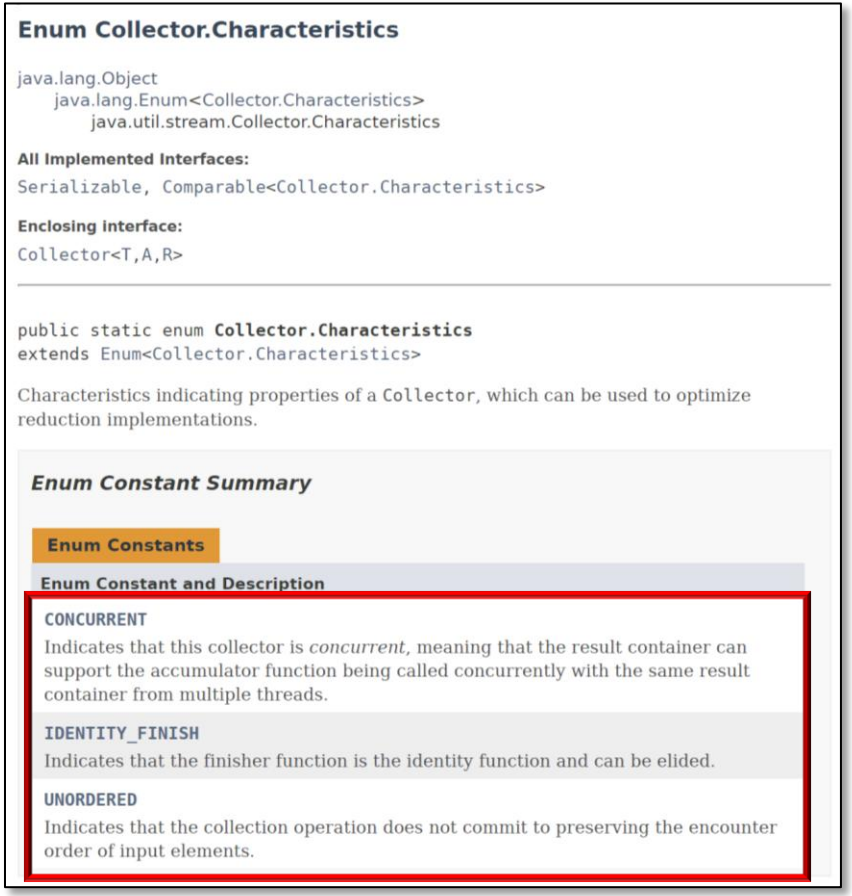
Examples of mutable reduction operations include: accumulating elements into a `Collection`; concatenating strings using a `StringBuilder`; computing summary information about elements such as sum, min, max, or average; computing "pivot table" summaries such as "maximum valued transaction by seller", etc. The class `Collectors` provides implementations of many common mutable reductions.

A `Collector` is specified by four functions that work together to accumulate entries into a mutable result container, and optionally perform a final transform on the result. They are:

See docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.html

Overview of Concurrent and Non-Concurrent Collectors

- Collector implementations can either be concurrent or non-concurrent based on their characteristics.



Enum Collector.Characteristics

java.lang.Object
  java.lang.Enum<Collector.Characteristics>
    java.util.stream.Collector.Characteristics

All Implemented Interfaces:

`Serializable, Comparable<Collector.Characteristics>`

Enclosing interface:

`Collector<T,A,R>`

```
public static enum Collector.Characteristics
extends Enum<Collector.Characteristics>
```

Characteristics indicating properties of a `Collector`, which can be used to optimize reduction implementations.

Enum Constant Summary

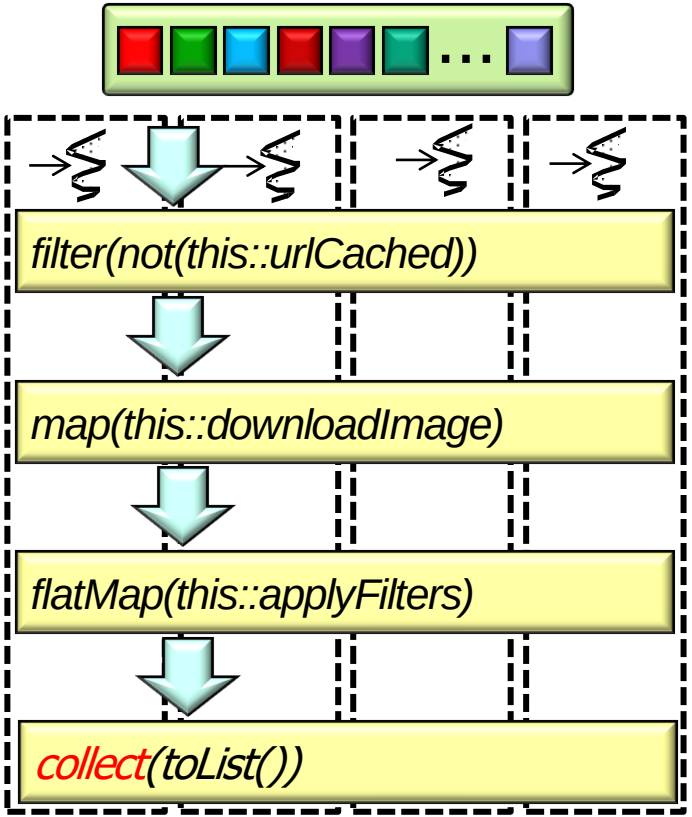
Enum Constants

Enum Constant and Description
`CONCURRENT` Indicates that this collector is *concurrent*, meaning that the result container can support the accumulator function being called concurrently with the same result container from multiple threads.
`IDENTITY_FINISH` Indicates that the finisher function is the identity function and can be elided.
`UNORDERED` Indicates that the collection operation does not commit to preserving the encounter order of input elements.

See docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.Characteristics.html

Overview of Concurrent and Non-Concurrent Collectors

- Collector implementations can either be concurrent or non-concurrent based on their characteristics.
  - This distinction is only relevant for *parallel* streams.



See *"Java Streams: Introducing Non-Concurrent Collectors."*

Overview of Concurrent and Non-Concurrent Collectors

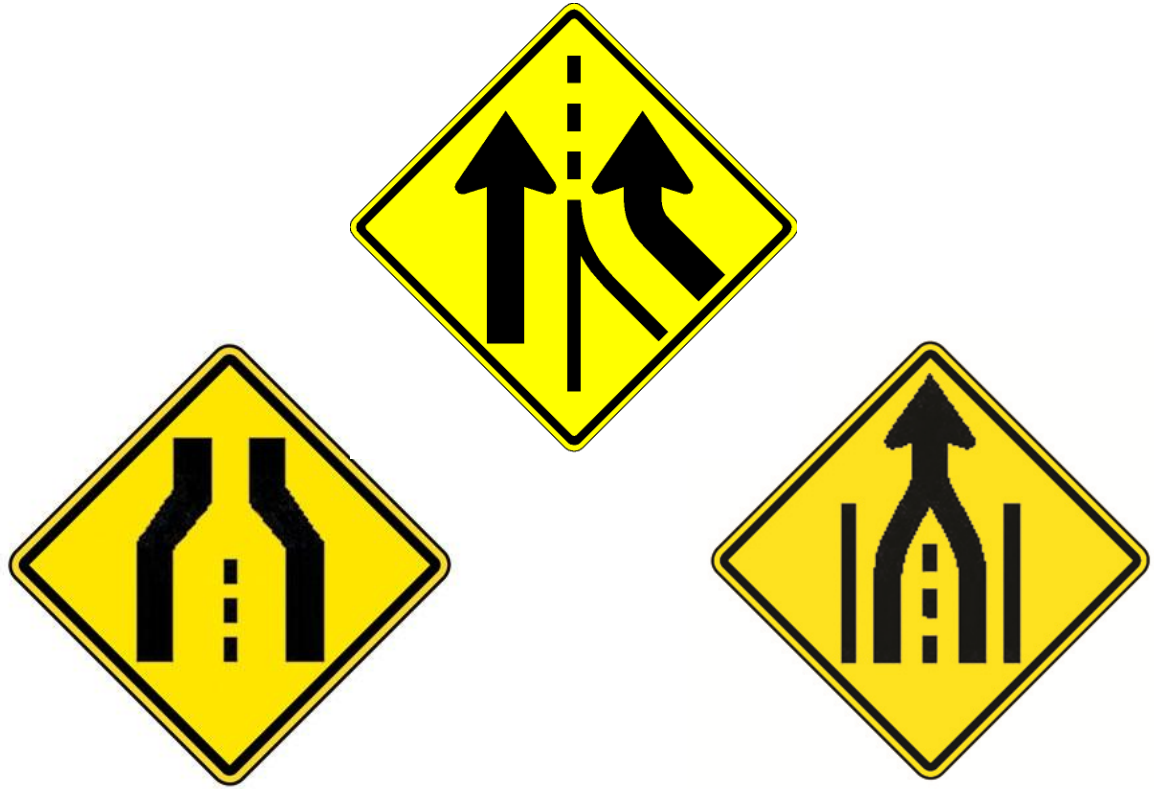
- Collector implementations can either be concurrent or non-concurrent based on their characteristics.
  - This distinction is only relevant for *parallel* streams.
  - A non-concurrent collector can be used for either a sequential stream or a parallel stream!

We just focus on parallel streams in this lesson.

Structure and Functionality of Non-Concurrent Collectors

Structure and Functionality of Non-Concurrent Collectors

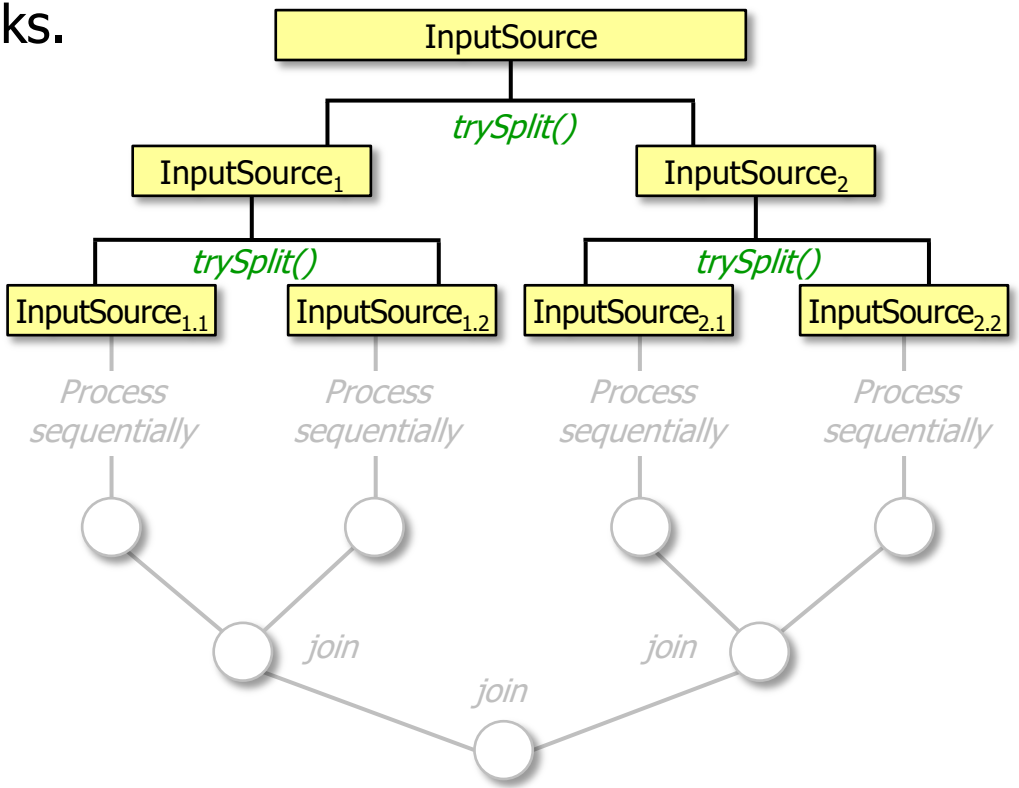
- A non-concurrent collector operates by merging sub-results.



See stackoverflow.com/questions/22350288/parallel-streams-collectors-and-thread-safety

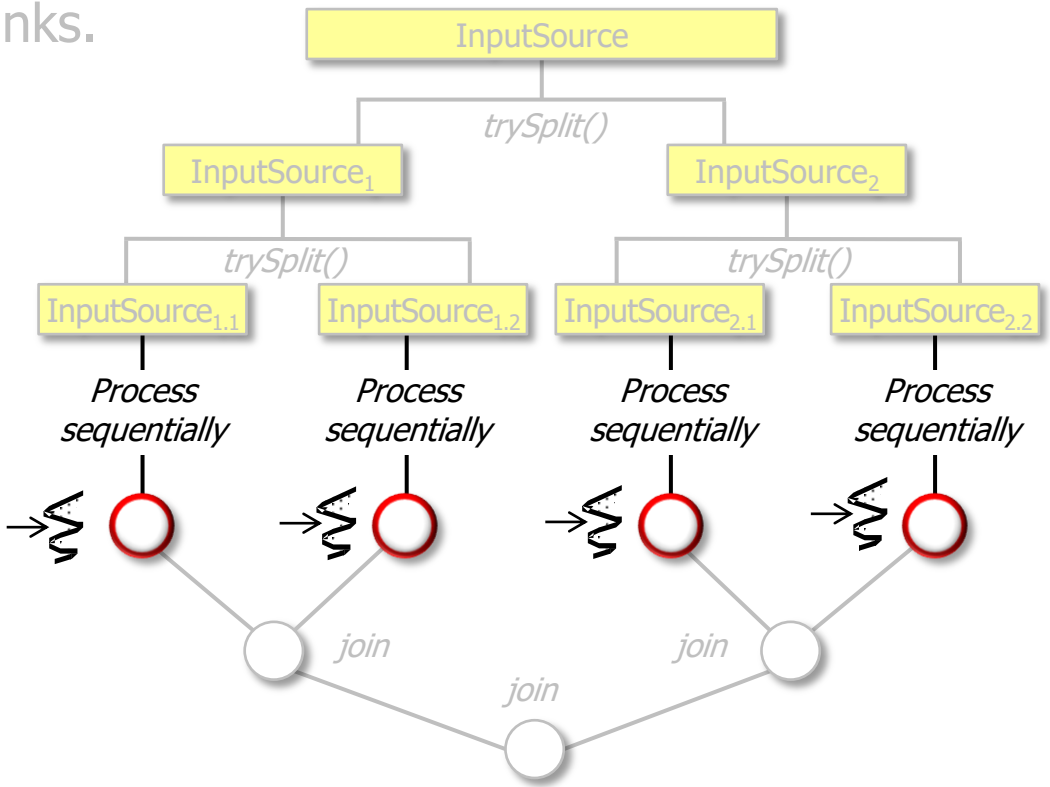
Structure and Functionality of Non-Concurrent Collectors

- A non-concurrent collector operates by merging sub-results.
  - The input is partitioned into chunks.



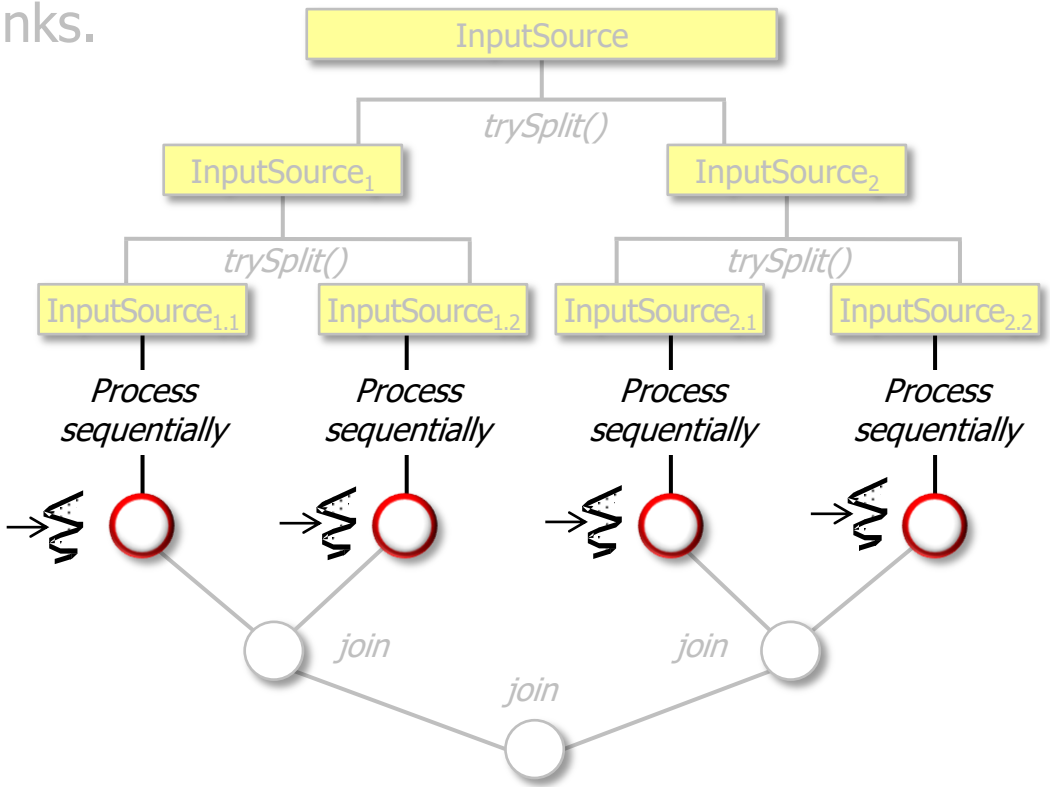
Structure and Functionality of Non-Concurrent Collectors

- A non-concurrent collector operates by merging sub-results.
  - The input is partitioned into chunks.
  - Each chunk runs in parallel in the common ForkJoinPool.



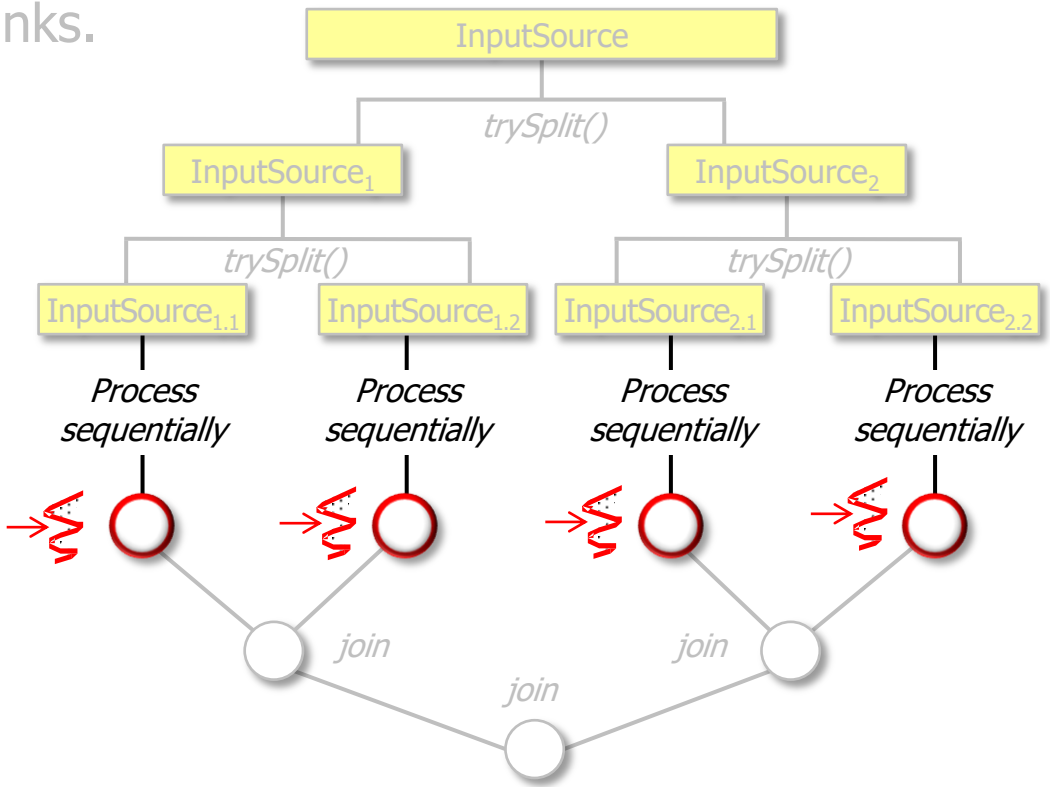
Structure and Functionality of Non-Concurrent Collectors

- A non-concurrent collector operates by merging sub-results.
  - The input is partitioned into chunks.
  - Each chunk runs in parallel in the common ForkJoinPool.
  - Chunk sub-results are collected into an intermediate mutable result container.
    - e.g., list, set, map, etc.



Structure and Functionality of Non-Concurrent Collectors

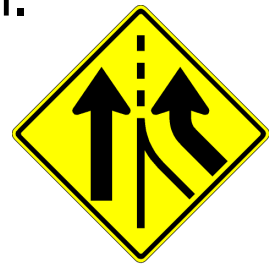
- A non-concurrent collector operates by merging sub-results.
  - The input is partitioned into chunks.
  - Each chunk runs in parallel in the common ForkJoinPool.
  - Chunk sub-results are collected into an intermediate mutable result container.
    - e.g., list, set, map, etc.

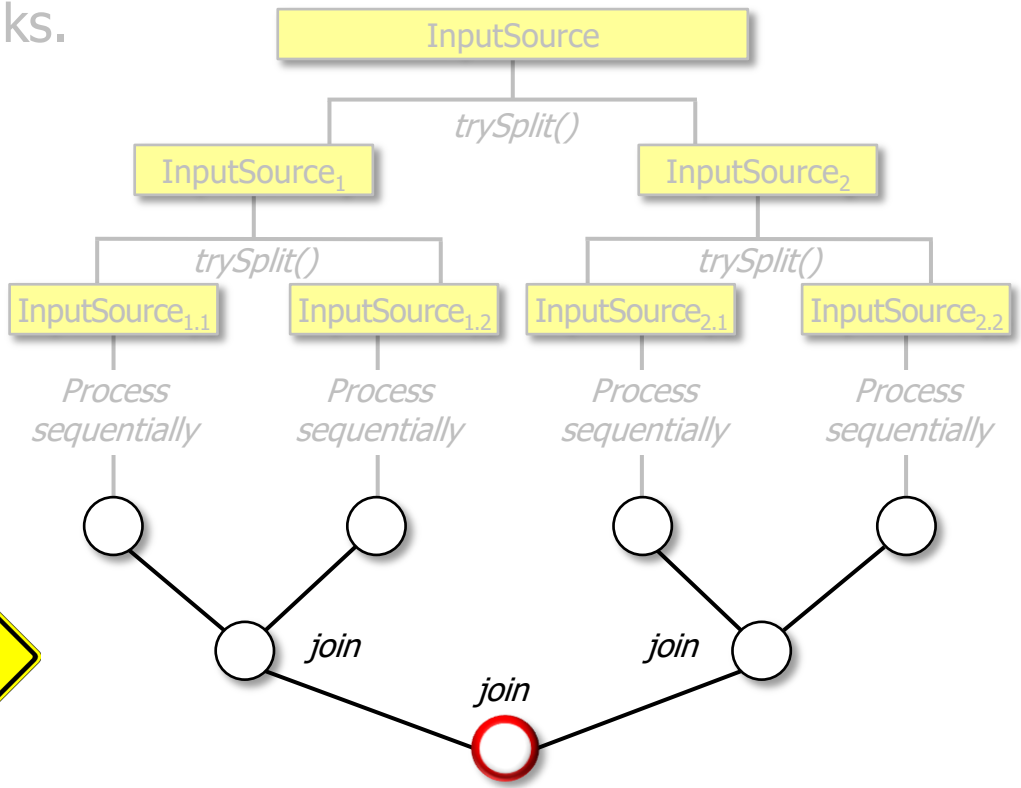


Different threads operate on different instances of intermediate result containers.

Structure and Functionality of Non-Concurrent Collectors

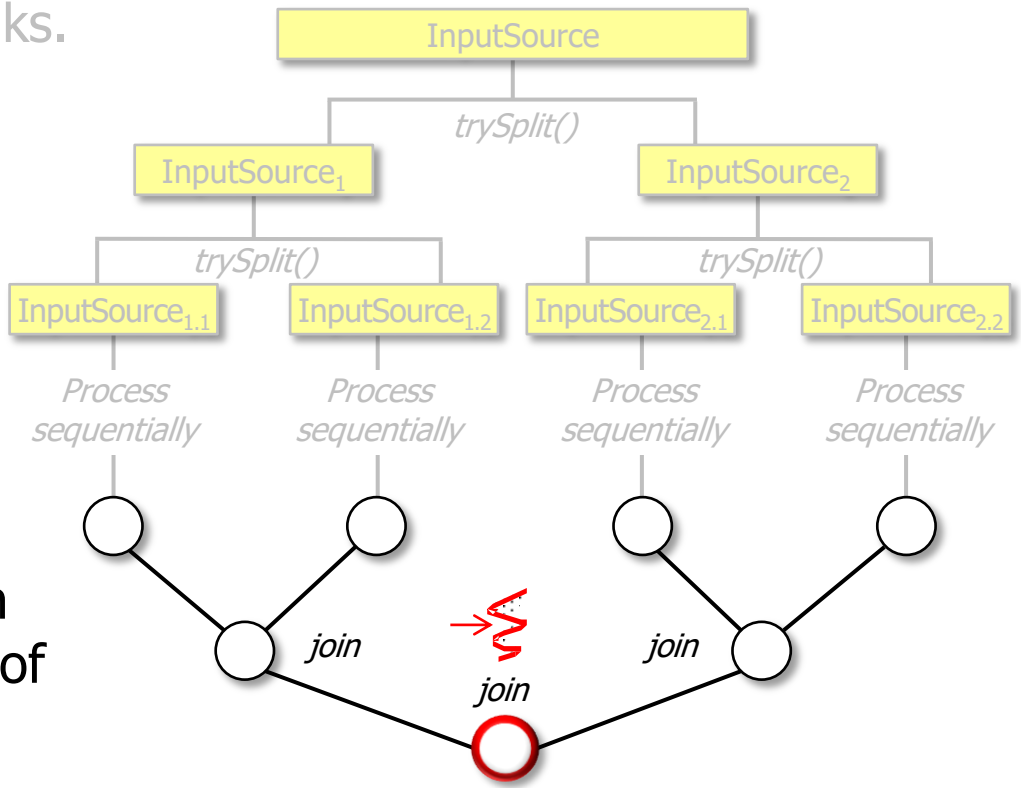
- A non-concurrent collector operates by merging sub-results.
  - The input is partitioned into chunks.
  - Each chunk runs in parallel in the common ForkJoinPool.
  - Chunk sub-results are collected into an intermediate mutable result container.
  - Sub-results are merged into one mutable result container.





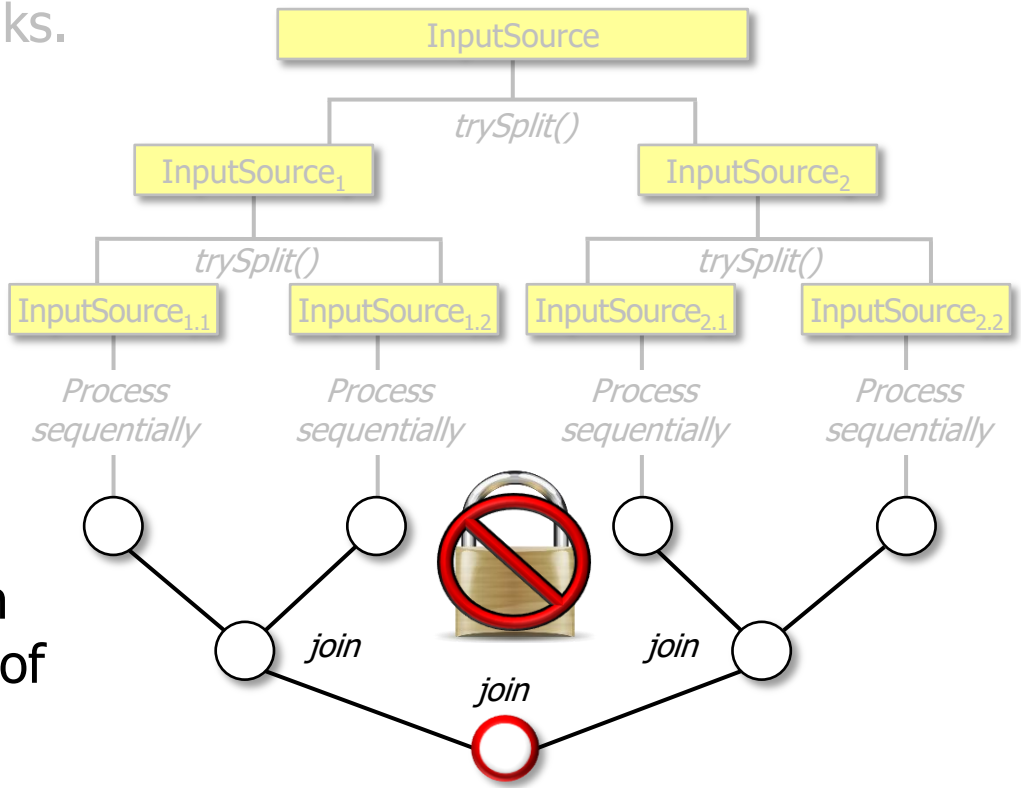
Structure and Functionality of Non-Concurrent Collectors

- A non-concurrent collector operates by merging sub-results.
  - The input is partitioned into chunks.
  - Each chunk runs in parallel in the common ForkJoinPool.
  - Chunk sub-results are collected into an intermediate mutable result container.
  - Sub-results are merged into one mutable result container.
    - Only one thread in the ForkJoin Pool is used to merge any pair of intermediate sub-results.



Structure and Functionality of Non-Concurrent Collectors

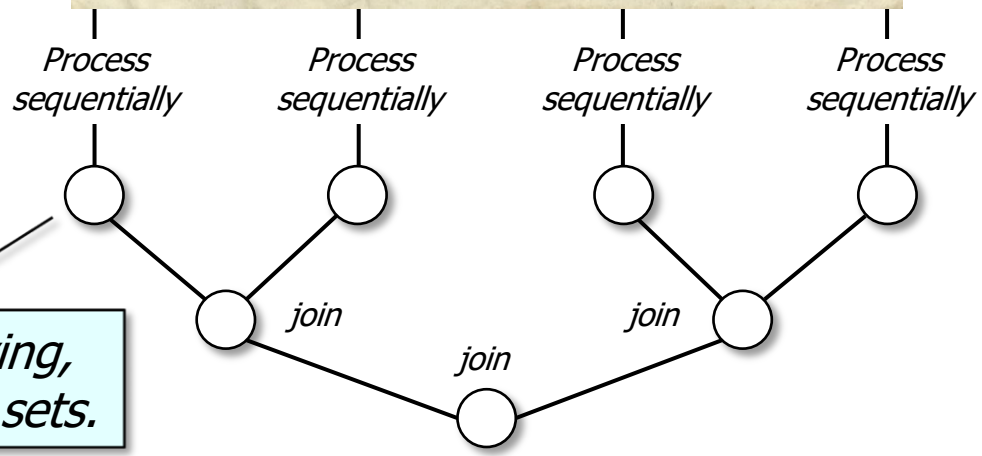
- A non-concurrent collector operates by merging sub-results.
  - The input is partitioned into chunks.
  - Each chunk runs in parallel in the common ForkJoinPool.
  - Chunk sub-results are collected into an intermediate mutable result container.
  - Sub-results are merged into one mutable result container.
    - Only one thread in the ForkJoin Pool is used to merge any pair of intermediate sub-results.



Thus there's no need for any synchronizers in a non-concurrent collector.

Structure and Functionality of Non-Concurrent Collectors

- A non-concurrent collector operates by merging sub-results.
  - The input is partitioned into chunks.
  - Each chunk runs in parallel in the common ForkJoinPool.
  - Chunk sub-results are collected into an intermediate mutable result container.
  - Sub-results are merged into one mutable result container.



This process is safe and order-preserving, but costly for containers like maps and sets.

Structure and Functionality of Concurrent Collectors

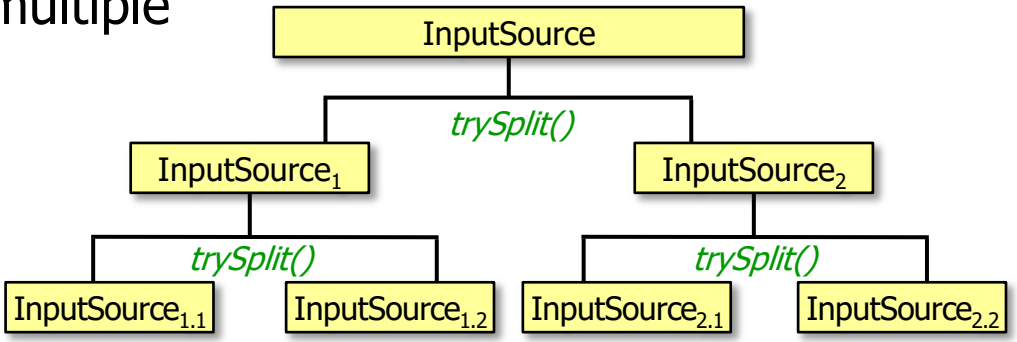
Structure and Functionality of Concurrent Collectors

- A concurrent collector creates one concurrent mutable result container and accumulates elements into it from multiple threads in a parallel stream.

See stackoverflow.com/questions/22350288/parallel-streams-collectors-and-thread-safety

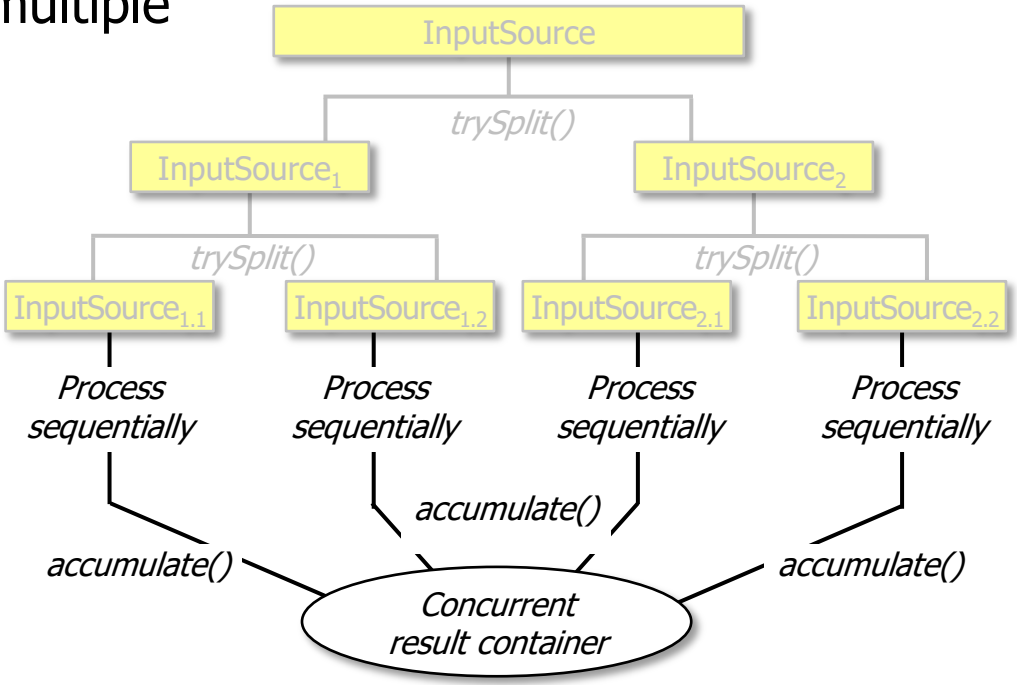
Structure and Functionality of Concurrent Collectors

- A concurrent collector creates one concurrent mutable result container and accumulates elements into it from multiple threads in a parallel stream.
  - As usual, the input is partitioned into chunks.



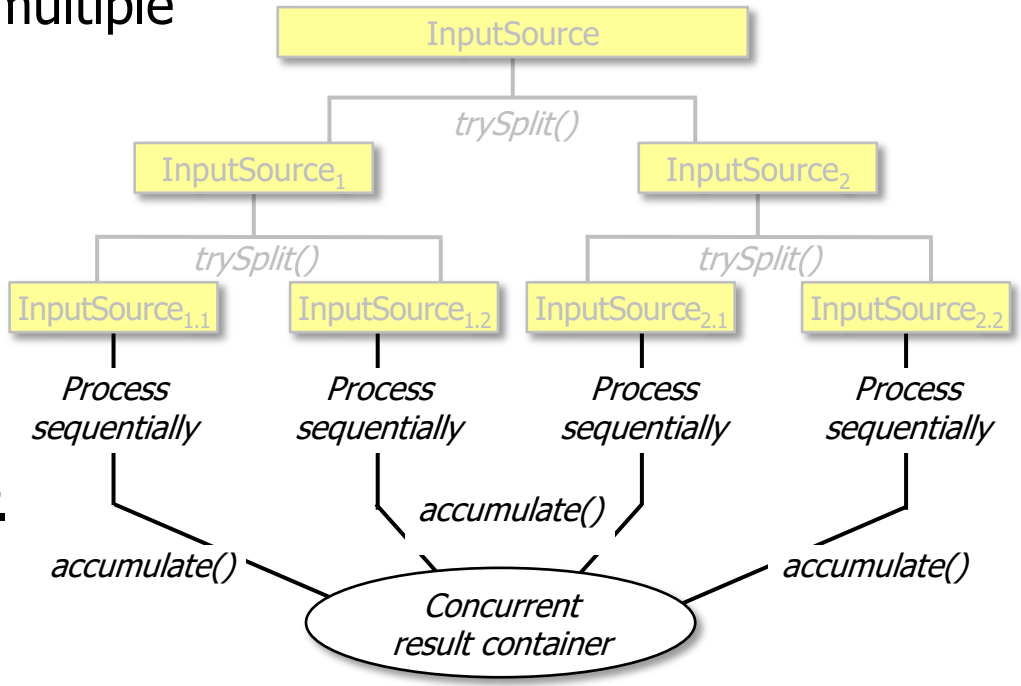
Structure and Functionality of Concurrent Collectors

- A concurrent collector creates one concurrent mutable result container and accumulates elements into it from multiple threads in a parallel stream.
  - As usual, the input is partitioned into chunks.
  - Each chunk runs in parallel in the common ForkJoinPool.



Structure and Functionality of Concurrent Collectors

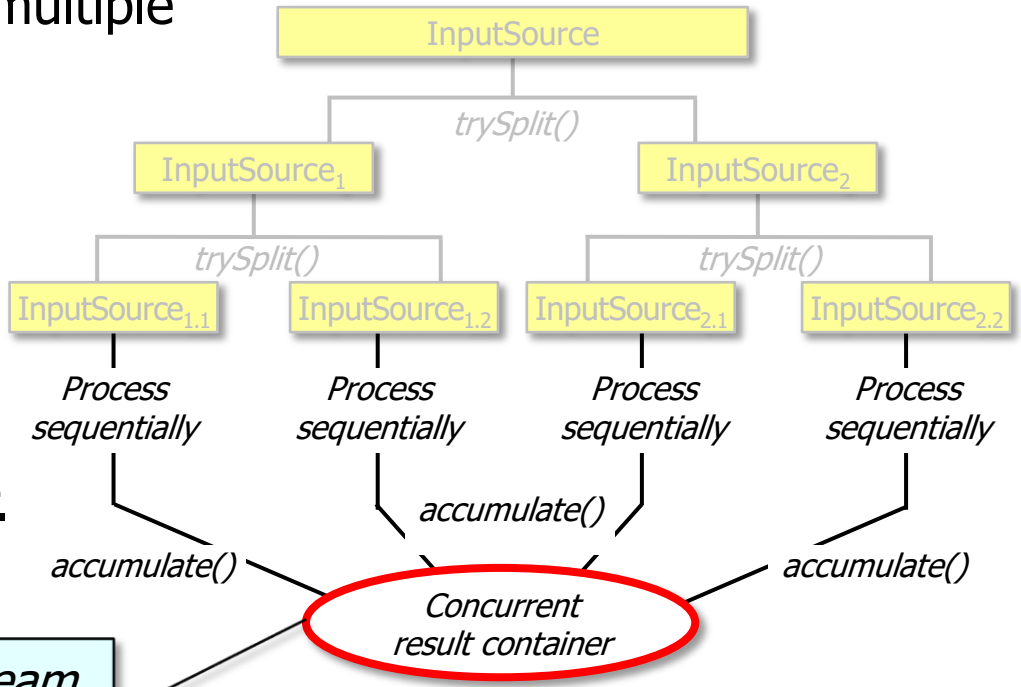
- A concurrent collector creates one concurrent mutable result container and accumulates elements into it from multiple threads in a parallel stream.
  - As usual, the input is partitioned into chunks.
  - Each chunk runs in parallel in the common ForkJoinPool.
  - Chunk sub-results are collected into one mutable result container.
    - e.g., a concurrent collection



See docs.oracle.com/javase/tutorial/essential/concurrency/collections.html

Structure and Functionality of Concurrent Collectors

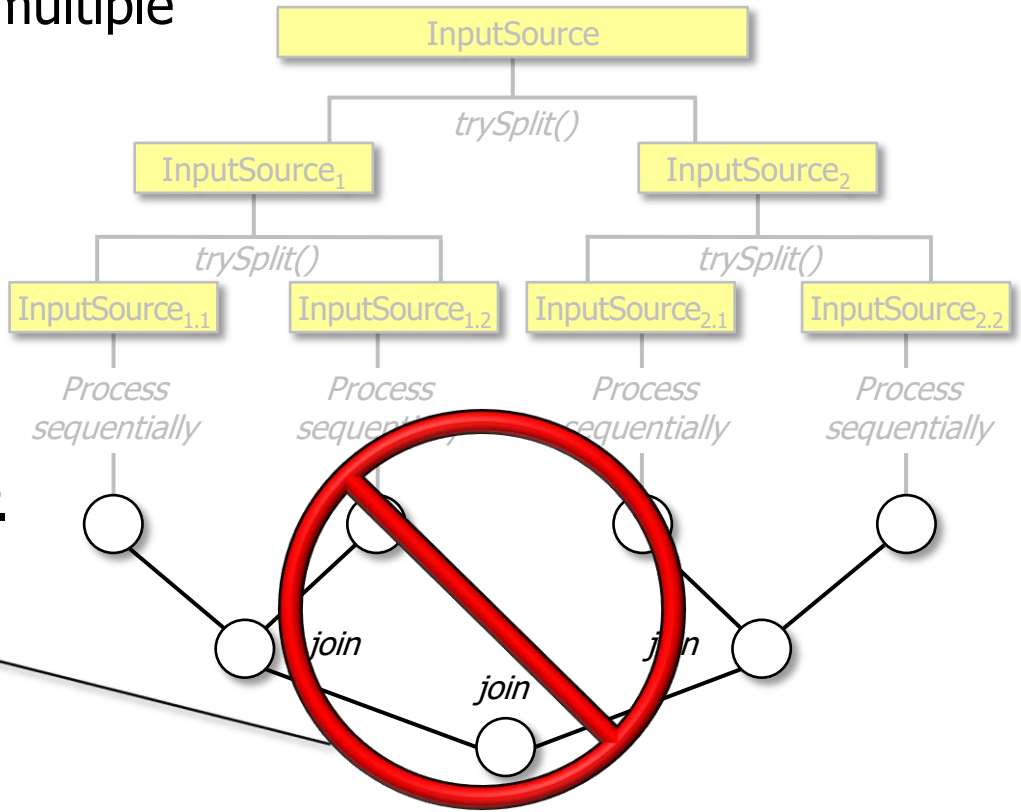
- A concurrent collector creates one concurrent mutable result container and accumulates elements into it from multiple threads in a parallel stream.
  - As usual, the input is partitioned into chunks.
  - Each chunk runs in parallel in the common ForkJoinPool.
  - Chunk sub-results are collected into one mutable result container.
    - e.g., a concurrent collection



Different threads in a parallel stream share one concurrent result container.

Structure and Functionality of Concurrent Collectors

- A concurrent collector creates one concurrent mutable result container and accumulates elements into it from multiple threads in a parallel stream.
  - As usual, the input is partitioned into chunks.
  - Each chunk runs in parallel in the common ForkJoinPool.
  - Chunk sub-results are collected into one mutable result container.

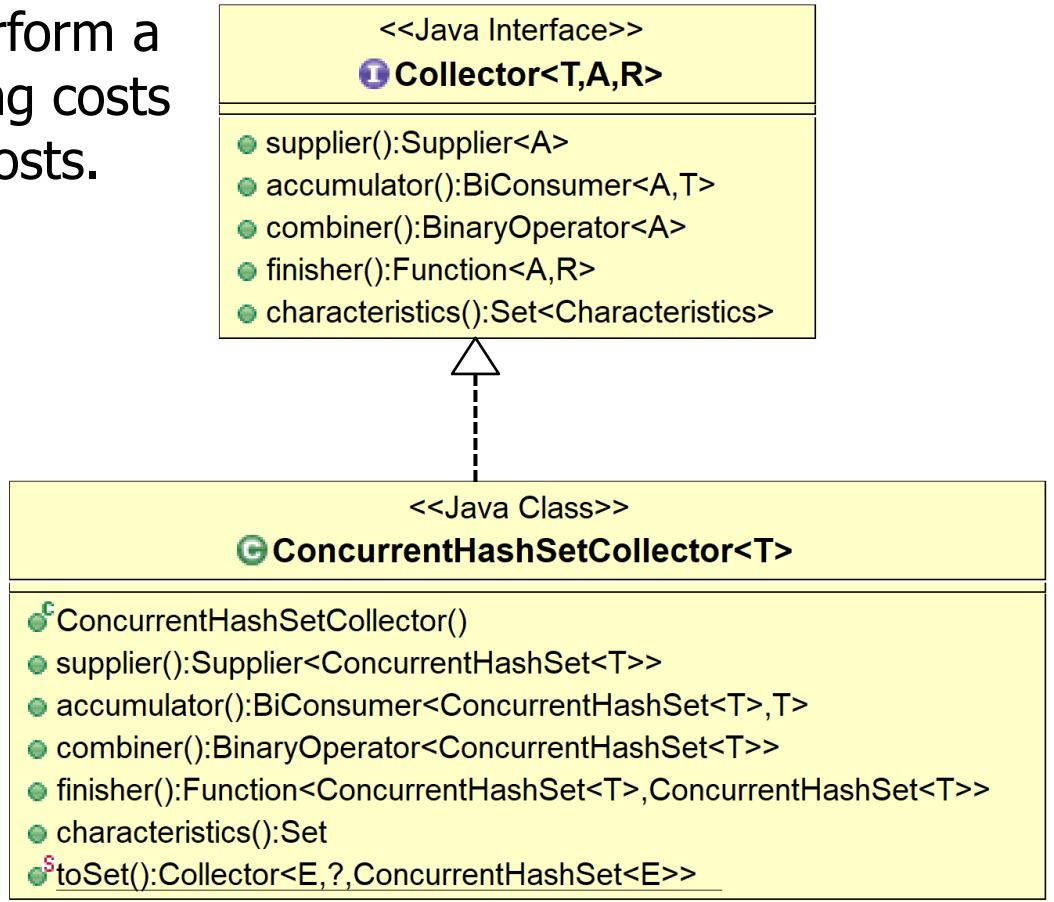


Thus there's no need to merge any intermediate sub-results!

Of course, encounter order is not preserved and synchronization is required.

Structure and Functionality of Concurrent Collectors

- A concurrent collector *may* outperform a non-concurrent collector *if* merging costs are higher than synchronization costs.

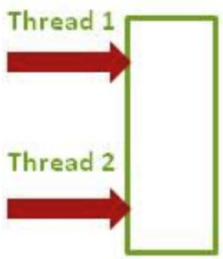


See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex14

Structure and Functionality of Concurrent Collectors

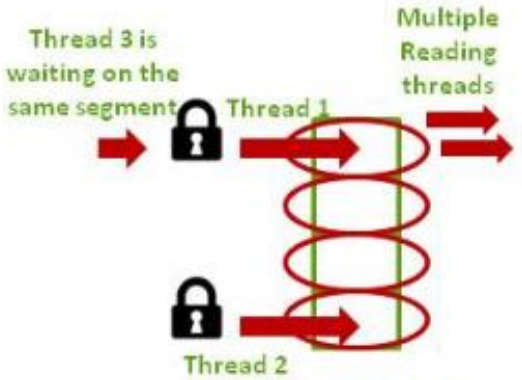
- A concurrent collector *may* outperform a non-concurrent collector *if* merging costs are higher than synchronization costs.
  - Highly optimized result containers like ConcurrentHashMap may be more efficient than merging HashMaps.

HashMap



Not Thread-Safe.
Can have one null key and multiple null values

ConcurrentHashMap

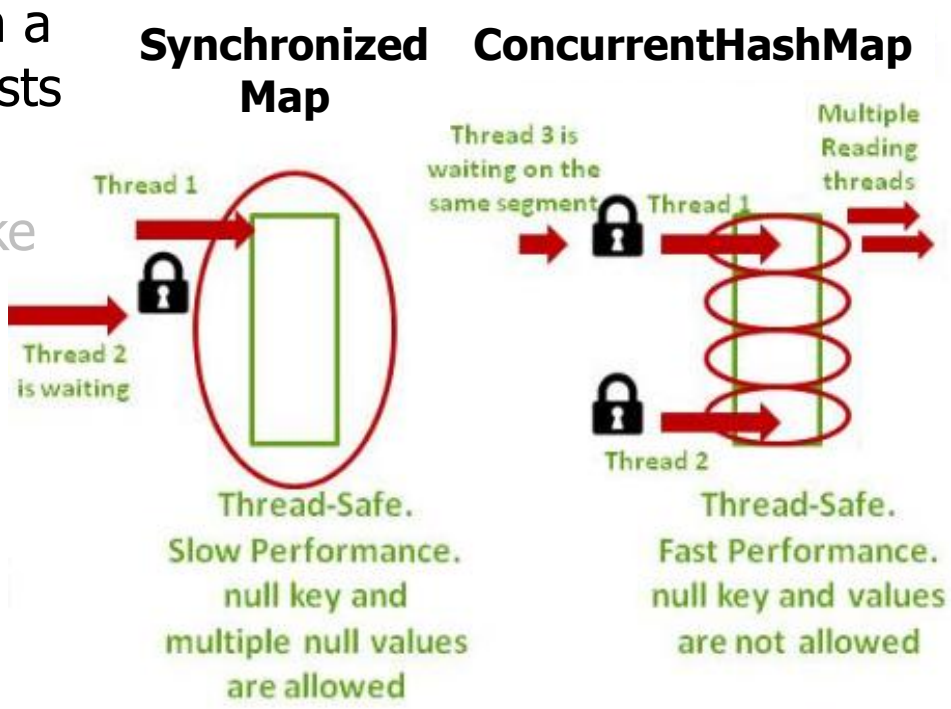


Thread-Safe.
Fast Performance.
null key and values are not allowed

See codepumpkin.com/hashtable-vs-synchronizedmap-vs-concurrenthashmap

Structure and Functionality of Concurrent Collectors

- A concurrent collector *may* outperform a non-concurrent collector *if* merging costs are higher than synchronization costs.
  - Highly optimized result containers like ConcurrentHashMap may be more efficient than merging HashMaps.
  - ConcurrentHashMap is also more efficient than a SynchronizedMap.



See www.quora.com/What-is-the-difference-between-synchronize-and-concurrent-collection-in-Java

Java Parallel Stream Internals:
Non-Concurrent and Concurrent Collectors (Part I)

The End

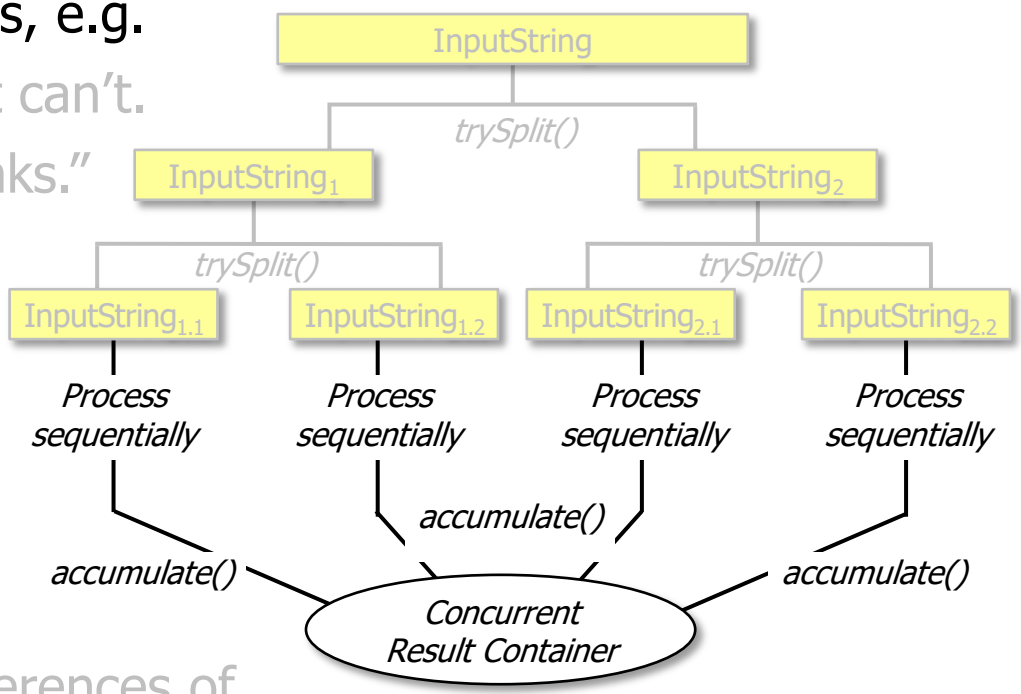
Java Parallel Stream Internals

Non-Concurrent and Concurrent Collectors (Part II)

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Understand parallel stream internals, e.g.
  - Know what can change and what can't.
  - Partition a data source into "chunks."
  - Process chunks in parallel via the common ForkJoinPool.
  - Configure the Java parallel stream common ForkJoinPool.
  - Perform a reduction to combine partial results into a single result.
  - Recognize key behaviors and differences of non-concurrent and concurrent collectors.
- Learn how to implement non-concurrent and concurrent collectors.



Implementing Non-Concurrent and Concurrent Collectors

Implementing Non-Concurrent and Concurrent Collectors

- The Collector interface defines three generic types.

<<Java Interface>>

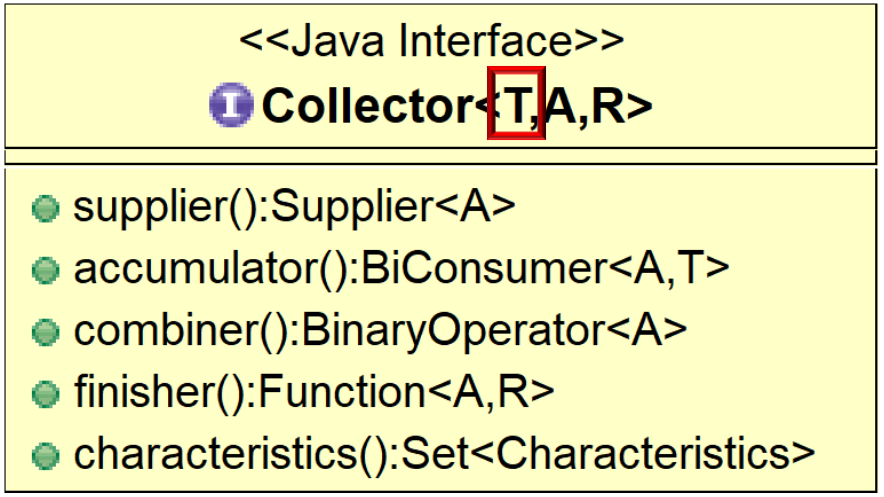
Collector<T,A,R>

- supplier():Supplier<A>
- accumulator():BiConsumer<A,T>
- combiner():BinaryOperator<A>
- finisher():Function<A,R>
- characteristics():Set<Characteristics>

See www.baeldung.com/java-8-collectors

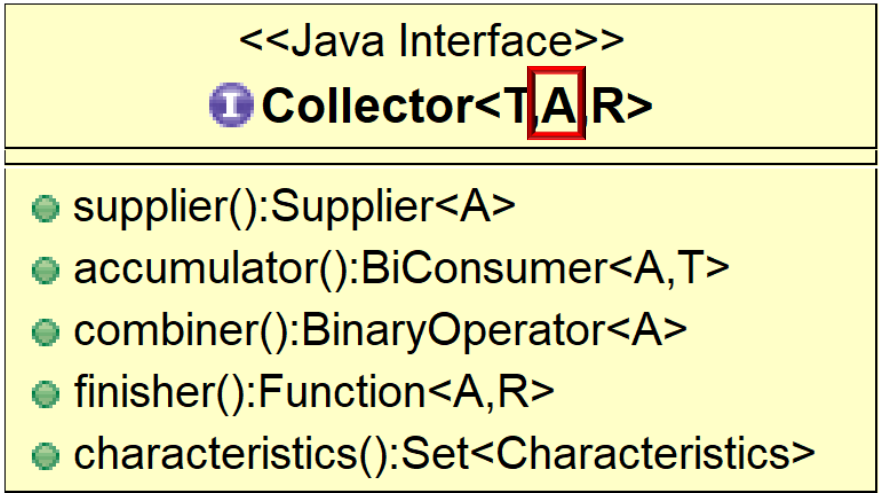
Implementing Non-Concurrent and Concurrent Collectors

- The Collector interface defines three generic types.
  - **T**: the type of objects available in the stream
    - e.g., Integer, String, SearchResults, etc.



Implementing Non-Concurrent and Concurrent Collectors

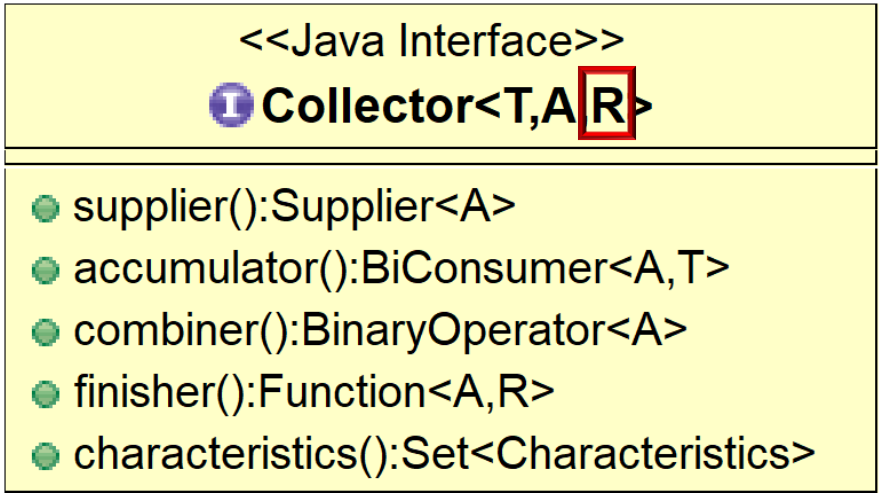
- The Collector interface defines three generic types
  - T
  - **A**: the type of a mutable accumulator object for collection
    - e.g., ConcurrentHashSet, List of T, Future of T, etc.
      - Lists can be implemented by ArrayList, LinkedList, etc.



See Java8/ex14/src/main/java/utils/ConcurrentHashSet.java

Implementing Non-Concurrent and Concurrent Collectors

- The Collector interface defines three generic types.
  - T
  - A
  - **R**: the type of a final result
    - e.g., ConcurrentHashSet, List of T, Future to List of T, etc.



See www.baeldung.com/java-8-collectors

Implementing Non-Concurrent and Concurrent Collectors

- Five methods are defined in the Collector interface.

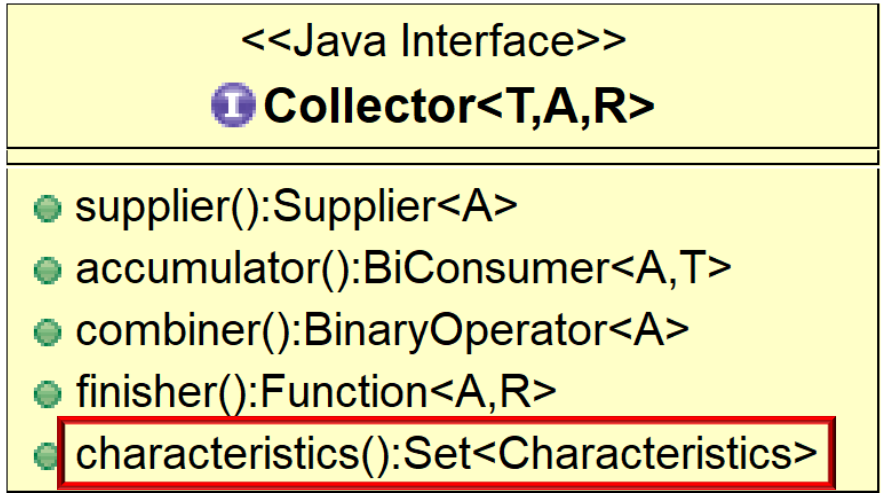
<<Java Interface>>

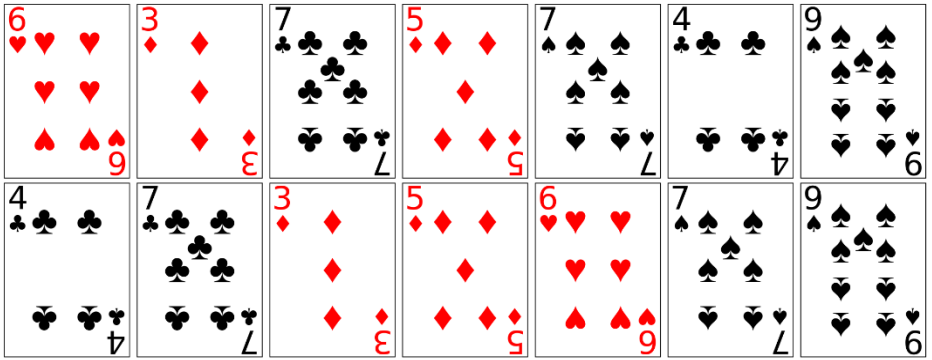
Collector<T,A,R>

- supplier():Supplier<A>
- accumulator():BiConsumer<A,T>
- combiner():BinaryOperator<A>
- finisher():Function<A,R>
- characteristics():Set<Characteristics>

Implementing Non-Concurrent and Concurrent Collectors

- Five methods are defined in the Collector interface.
  - **characteristics()** provides a stream with additional information used for internal optimizations, e.g.
    - UNORDERED
      - The collector need not preserve the encounter order.

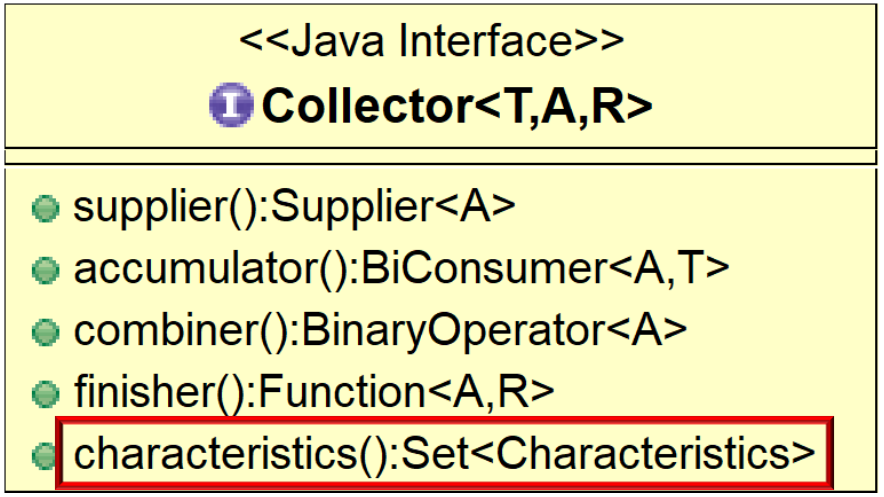




A concurrent collector *should* be unordered, but a non-concurrent collector *can* be ordered.

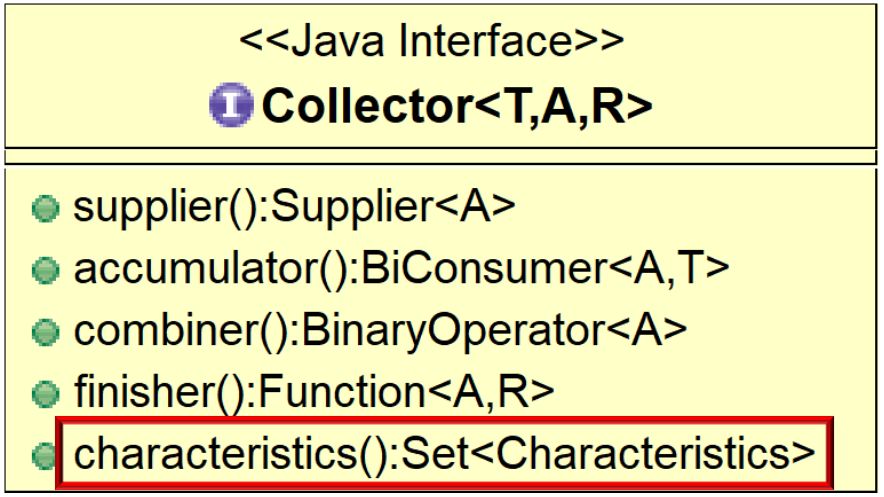
Implementing Non-Concurrent and Concurrent Collectors

- Five methods are defined in the Collector interface.
  - **characteristics()** provides a stream with additional information used for internal optimizations, e.g.
    - UNORDERED
    - IDENTITY_FINISH
      - The finisher() is the identity function so it can be a no-op.
        - e.g., finisher() just returns null.



Implementing Non-Concurrent and Concurrent Collectors

- Five methods are defined in the Collector interface.
  - **characteristics()** provides a stream with additional information used for internal optimizations, e.g.
    - UNORDERED
    - IDENTITY_FINISH
    - CONCURRENT
      - Accumulator() is called concurrently on result container.

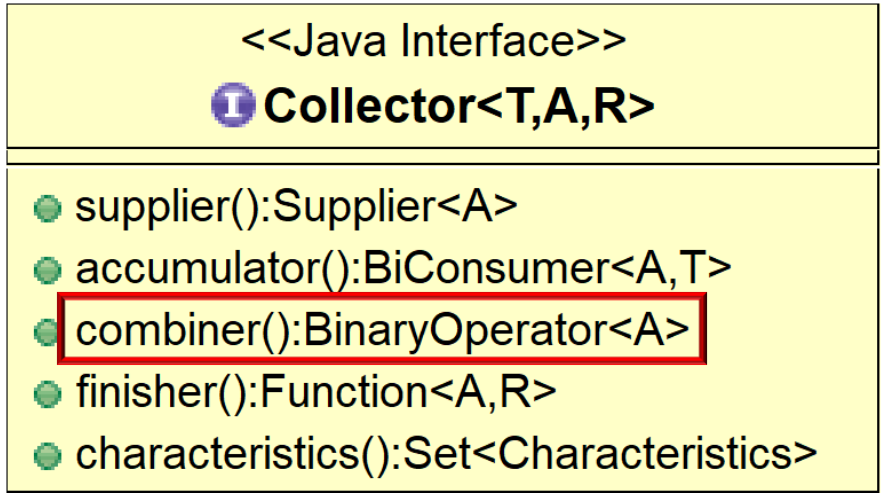


The mutable result container must be synchronized!

A concurrent collector *should* be concurrent, but a non-concurrent collector should *not* be!

Implementing Non-Concurrent and Concurrent Collectors

- Five methods are defined in the Collector interface.
  - **characteristics()** provides a stream with additional information used for internal optimizations, e.g.
    - UNORDERED
    - IDENTITY_FINISH
    - CONCURRENT
      - Accumulator() is called concurrently on result container.
      - The combiner() method is a no-op.



Implementing Non-Concurrent and Concurrent Collectors

- Five methods are defined in the Collector interface.
  - **characteristics()** provides a stream with additional information used for internal optimizations, e.g.
    - UNORDERED
    - IDENTITY_FINISH
    - CONCURRENT
      - Accumulator() is called concurrently on result container.
      - The combiner() method is a no-op.
      - A non-concurrent collector can be used with either sequential or parallel streams.

<<Java Interface>>

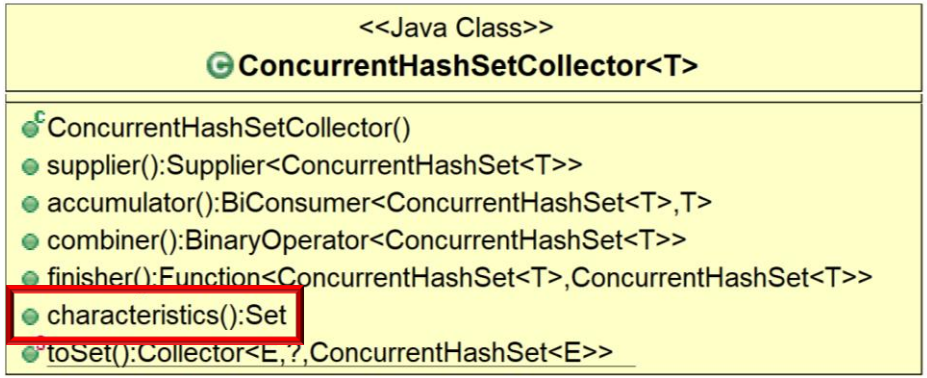
Collector<T,A,R>

- supplier():Supplier<A>
- accumulator():BiConsumer<A,T>
- combiner():BinaryOperator<A>
- finisher():Function<A,R>
- characteristics():Set<Characteristics>

Internally, the streams framework decides how to ensure correct behavior.

Implementing Non-Concurrent and Concurrent Collectors

- Five methods are defined in the Collector interface.
  - **characteristics()** provides a stream with additional information used for internal optimizations, e.g.



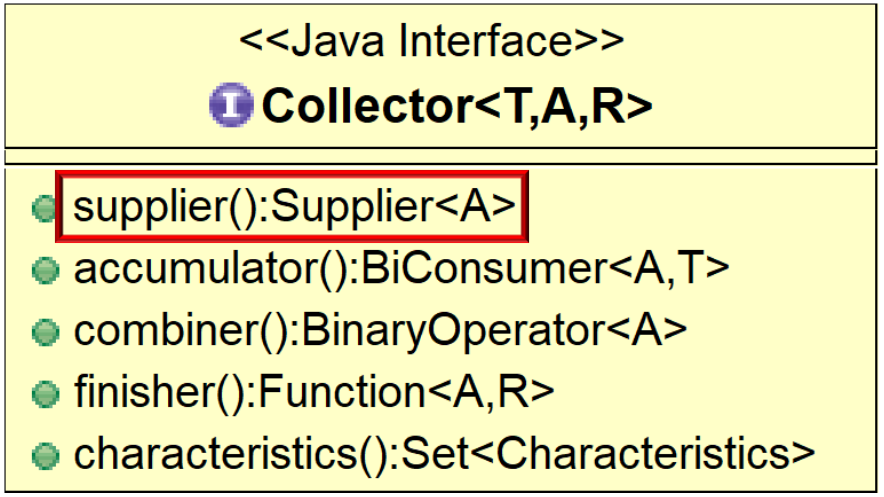
Any/all characteristics can be set using EnumSet.of().

```
Set characteristics() {
  return Collections.unmodifiableSet
    (EnumSet.of(Collector.Characteristics.CONCURRENT,
                Collector.Characteristics.UNORDERED,
                Collector.Characteristics.IDENTITY_FINISH));
}
```

See docs.oracle.com/javase/8/docs/api/java/util/EnumSet.html

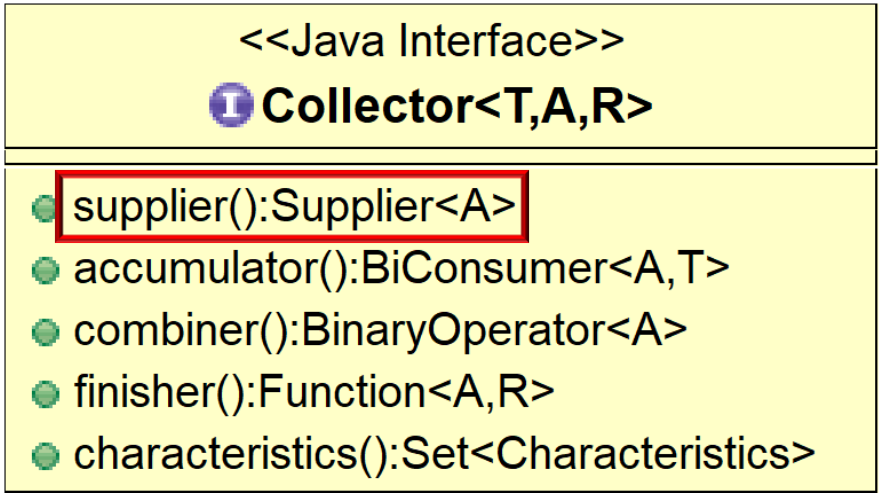
Implementing Non-Concurrent and Concurrent Collectors

- Five methods are defined in the Collector interface.
  - characteristics()
  - **supplier()** returns a supplier that acts as a factory to generate an empty result container.



Implementing Non-Concurrent and Concurrent Collectors

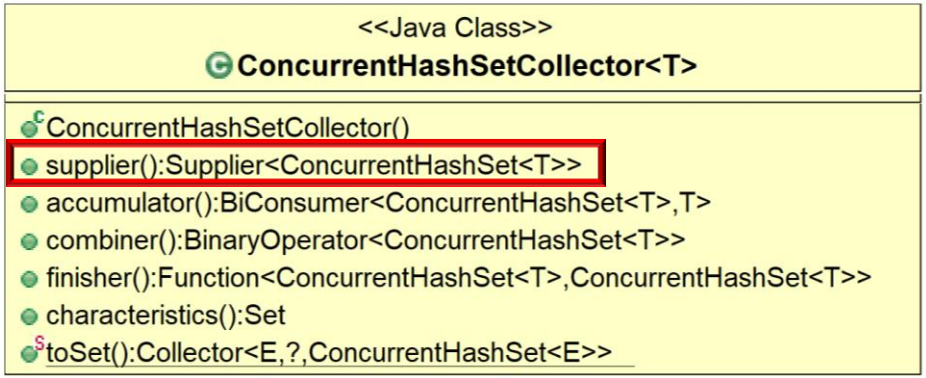
- Five methods are defined in the Collector interface.
  - characteristics()
  - **supplier()** returns a supplier that acts as a factory to generate an empty result container, e.g.
    - `return ArrayList::new`



A non-concurrent collector provides a result container for each thread in a parallel stream.

Implementing Non-Concurrent and Concurrent Collectors

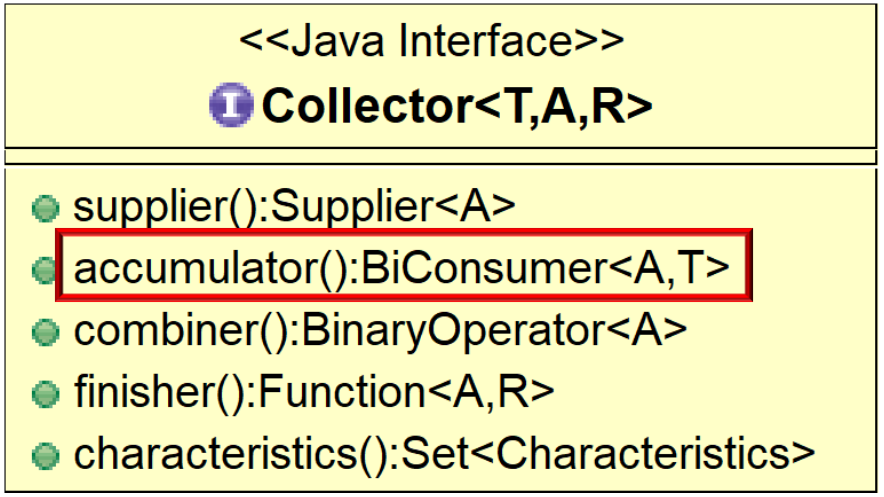
- Five methods are defined in the Collector interface.
  - characteristics()
  - **supplier()** returns a supplier that acts as a factory to generate an empty result container, e.g.
    - `return ArrayList::new`
    - **`return ConcurrentHashSet::new`**



A concurrent collector has one result container shared by all threads in a parallel stream.

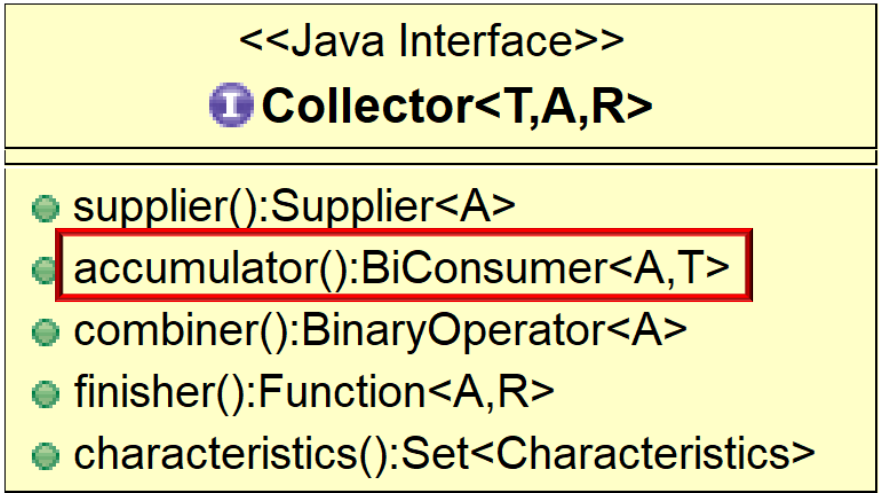
Implementing Non-Concurrent and Concurrent Collectors

- Five methods are defined in the Collector interface.
  - characteristics()
  - supplier()
  - **accumulator()** returns a BiConsumer that adds a new element to an existing result container.



Implementing Non-Concurrent and Concurrent Collectors

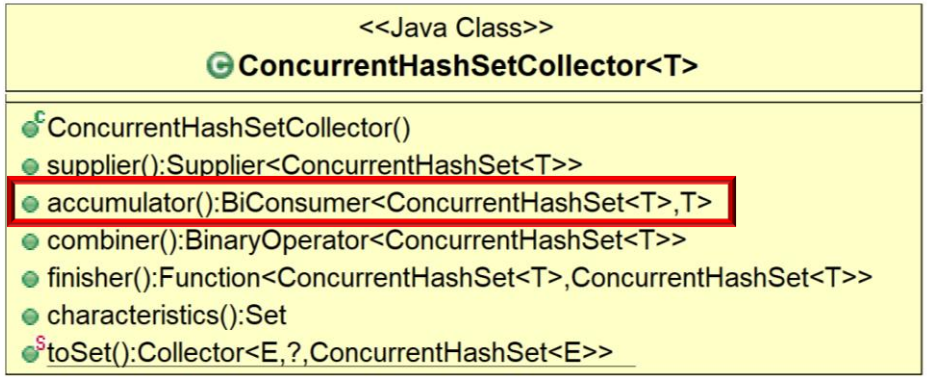
- Five methods are defined in the Collector interface.
  - characteristics()
  - supplier()
  - **accumulator()** returns a BiConsumer that adds a new element to an existing result container, e.g.
    - `return List::add`



A non-concurrent collector needs no synchronization.

Implementing Non-Concurrent and Concurrent Collectors

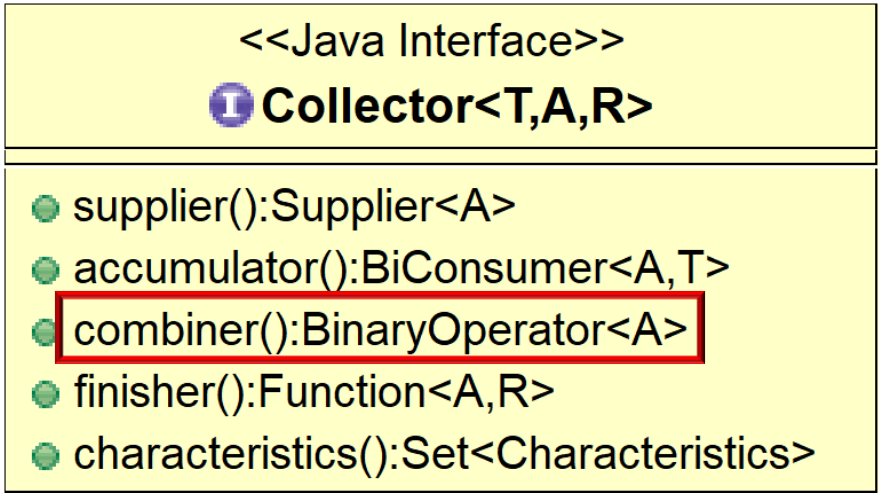
- Five methods are defined in the Collector interface.
  - **characteristics()**
  - **supplier()**
  - **accumulator()** returns a BiConsumer that adds a new element to an existing result container, e.g.
    - `return List::add`
    - **`return ConcurrentHashSet::add`**



A concurrent collector must be synchronized.

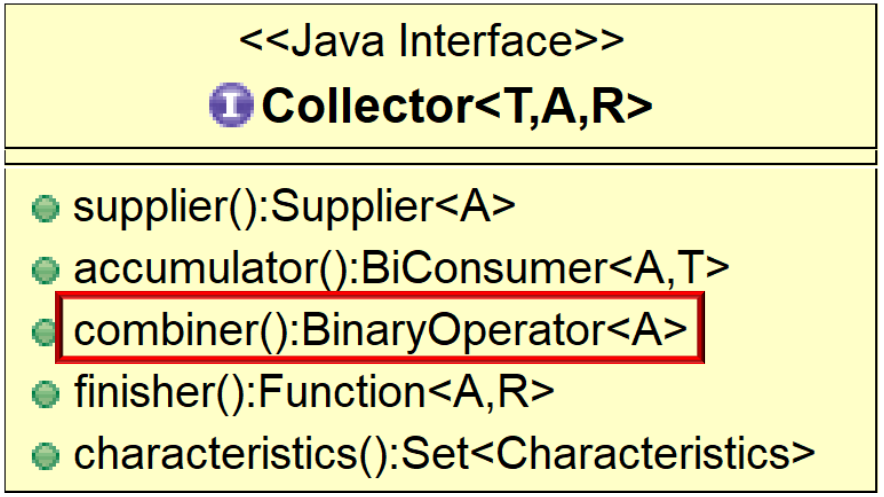
Implementing Non-Concurrent and Concurrent Collectors

- Five methods are defined in the Collector interface.
  - **characteristics()**
  - **supplier()**
  - **accumulator()**
  - **combiner()** returns a binary operator that merges two result containers together.



Implementing Non-Concurrent and Concurrent Collectors

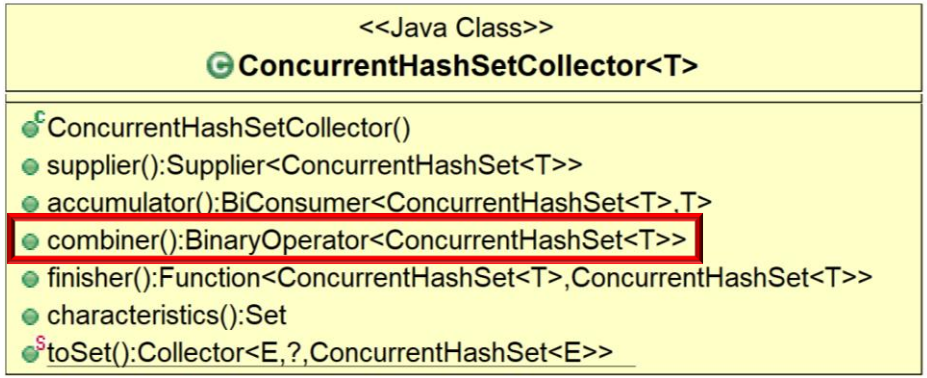
- Five methods are defined in the Collector interface.
  - characteristics()
  - supplier()
  - accumulator()
  - **combiner()** returns a binary operator that merges two result containers together, e.g.
    - `return (one, another) -> { one.addAll(another); return one; }`



A combiner() is only used for a non-concurrent collector.

Implementing Non-Concurrent and Concurrent Collectors

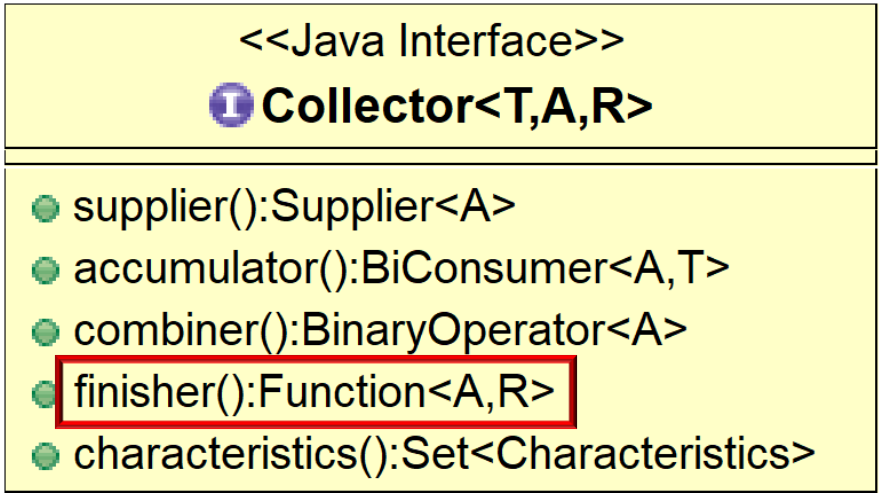
- Five methods are defined in the Collector interface.
  - characteristics()
  - supplier()
  - accumulator()
  - **combiner()** returns a binary operator that merges two result containers together, e.g.
    - `return (one, another) -> { one.addAll(another); return one; }`
    - `return null`



The combiner() method is not called when CONCURRENT is set.

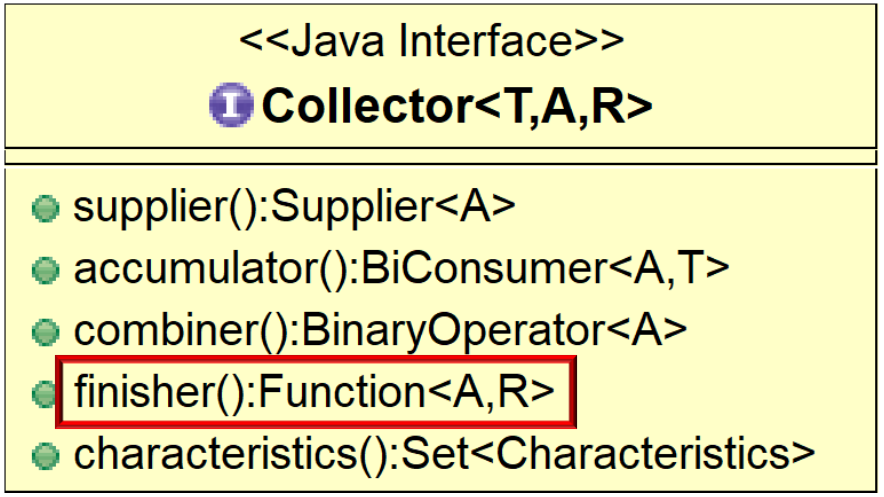
Implementing Non-Concurrent and Concurrent Collectors

- Five methods are defined in the Collector interface.
  - characteristics()
  - supplier()
  - accumulator()
  - combiner()
  - **finisher()** returns a function that converts the result container to final result type.



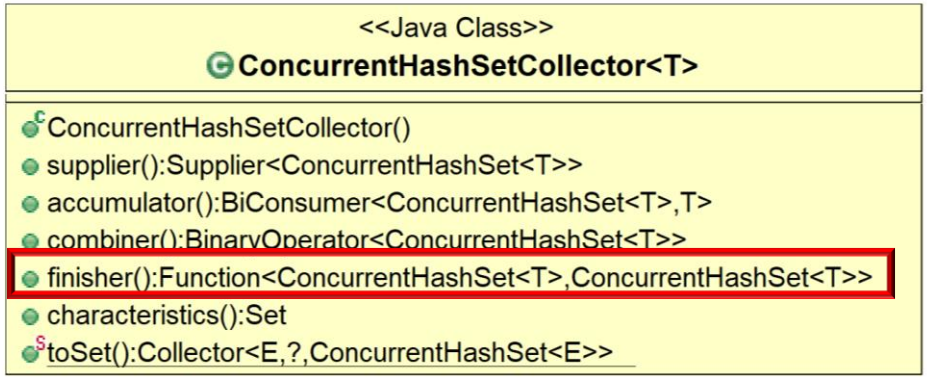
Implementing Non-Concurrent and Concurrent Collectors

- Five methods are defined in the Collector interface.
  - characteristics()
  - supplier()
  - accumulator()
  - combiner()
  - **finisher()** returns a function that converts the result container to final result type, e.g.
    - `Function.identity()`



Implementing Non-Concurrent and Concurrent Collectors

- Five methods are defined in the Collector interface.
  - characteristics()
  - supplier()
  - accumulator()
  - combiner()
  - **finisher()** returns a function that converts the result container to final result type, e.g.
    - `Function.identity()`
    - **`return null`**



Should be a no-op if IDENTITY_FINISH characteristic is set.

Implementing Non-Concurrent and Concurrent Collectors

- Five methods are defined in the Collector interface.
  - characteristics()
  - supplier()
  - accumulator()
  - combiner()
  - **finisher()** returns a function that converts the result container to final result type, e.g.
    - `Function.identity()`
    - `return null`

```
Stream
   .generate(() ->
      makeBigFraction
         (new Random(), false))
   .limit(sMAX_FRACTIONS)

   .map(reduceAndMultiplyFraction)
   .collect(FuturesCollector
            .toFuture())

   .thenAccept
       (this::sortAndPrintList);
```

Finisher() can also be much more interesting!

See Java8/ex19/src/main/java/utils/FuturesCollector.java

Java Parallel Stream Internals:
Non-Concurrent and Concurrent Collectors (Part II)

The End

Java Parallel Stream Internals

Demo’ing Collector Performance

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Understand parallel stream internals, e.g.
  - Know what can change and what can't
  - Partition a data source into "chunks"
  - Process chunks in parallel via the common fork-join pool
  - Configure the Java parallel stream common fork-join pool
  - Perform a reduction to combine partial results into a single result
  - Recognize key behaviors and differences between non-concurrent and concurrent collectors
  - Learn how to implement non-concurrent and concurrent collectors
  - Be aware of performance variance in concurrent and non-concurrent collectors

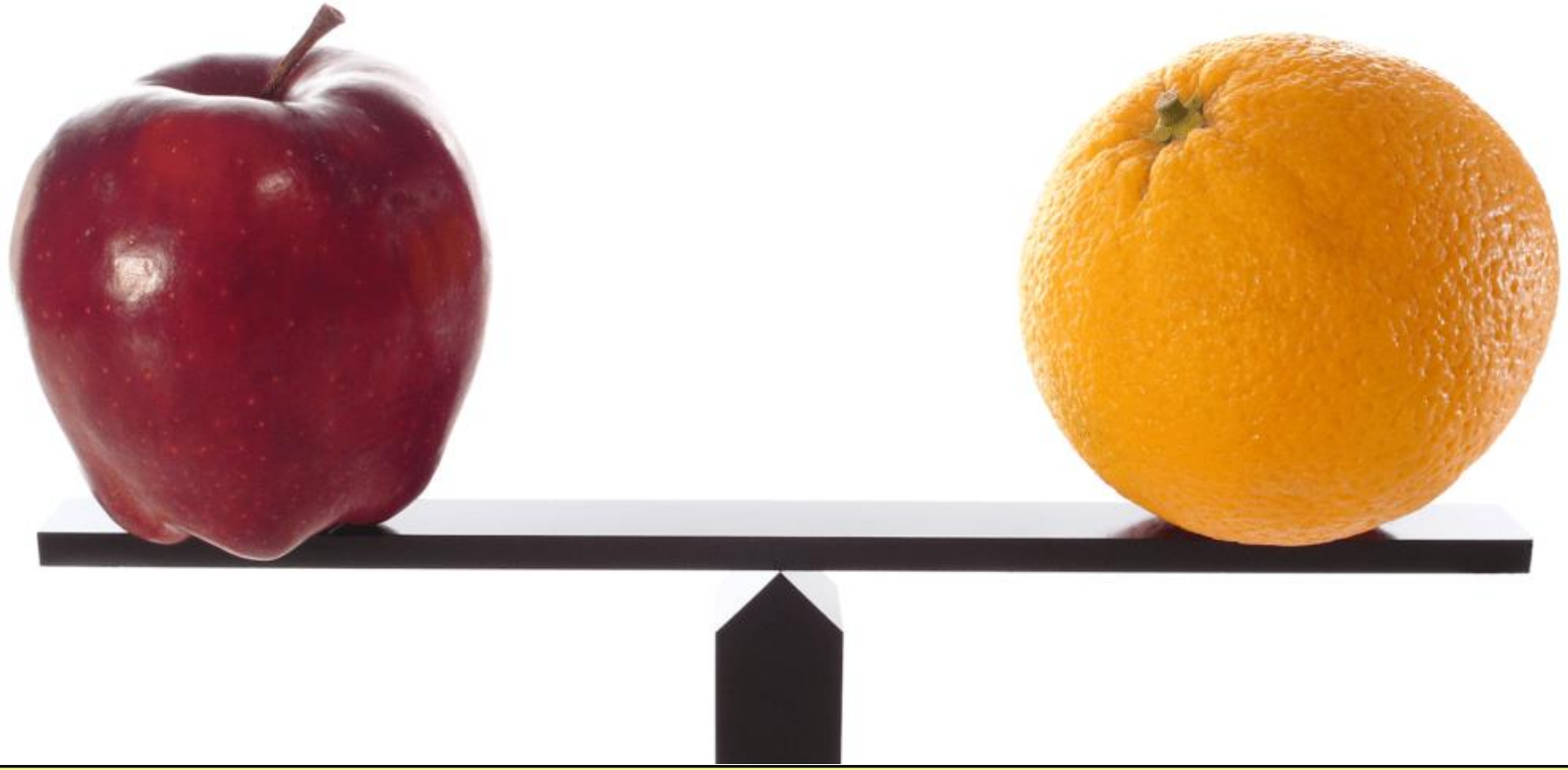
```
Starting collector tests for 1000 words..printing results
    21 msecs: sequential timeStreamCollectToSet()
    30 msecs: parallel timeStreamCollectToSet()
    39 msecs: sequential timeStreamCollectToConcurrentSet()
    59 msecs: parallel timeStreamCollectToConcurrentSet()
...
Starting collector tests for 100000 words..printing results
   219 msecs: parallel timeStreamCollectToConcurrentSet()
   364 msecs: parallel timeStreamCollectToSet()
   657 msecs: sequential timeStreamCollectToSet()
   804 msecs: sequential timeStreamCollectToConcurrentSet()
Starting collector tests for 883311 words..printing results
  1782 msecs: parallel timeStreamCollectToConcurrentSet()
  3010 msecs: parallel timeStreamCollectToSet()
  6169 msecs: sequential timeStreamCollectToSet()
  7652 msecs: sequential timeStreamCollectToConcurrentSet()
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex14

Demonstrating Collector Performance

Demonstrating Collector Performance

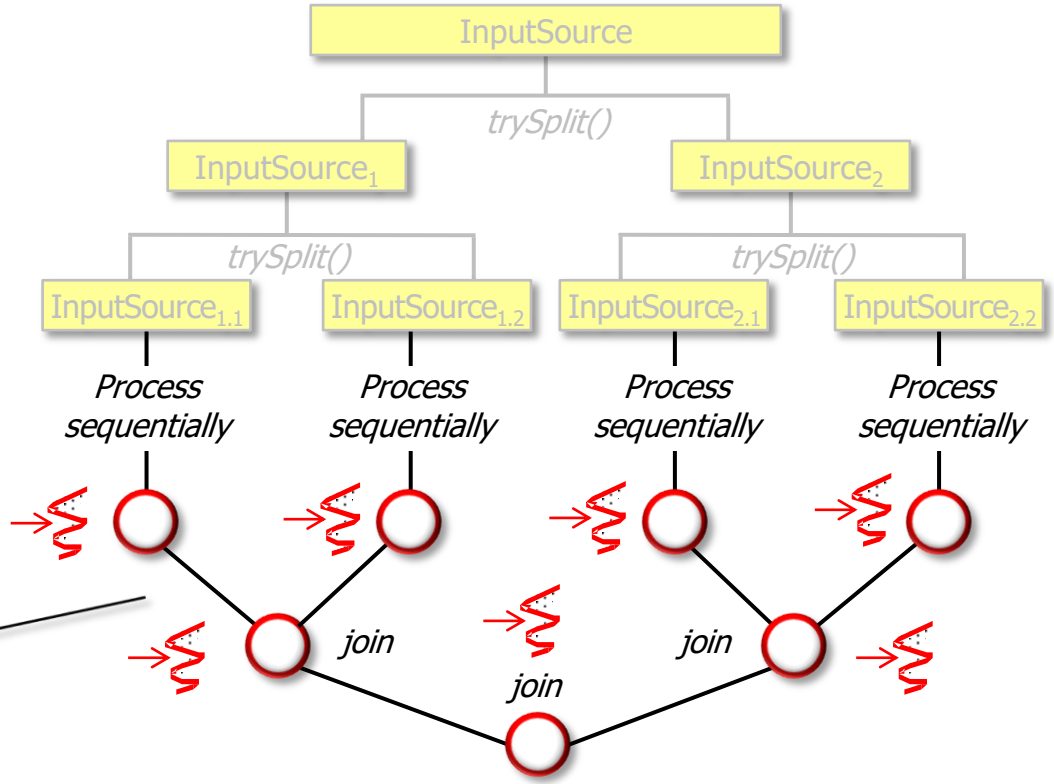
- Concurrent and non-concurrent collectors perform differently when used in parallel and sequential streams on different input sizes.



See prior lessons on "*Java Parallel Streams Internals: Non-Concurrent and Concurrent Collectors*"

Demonstrating Collector Performance

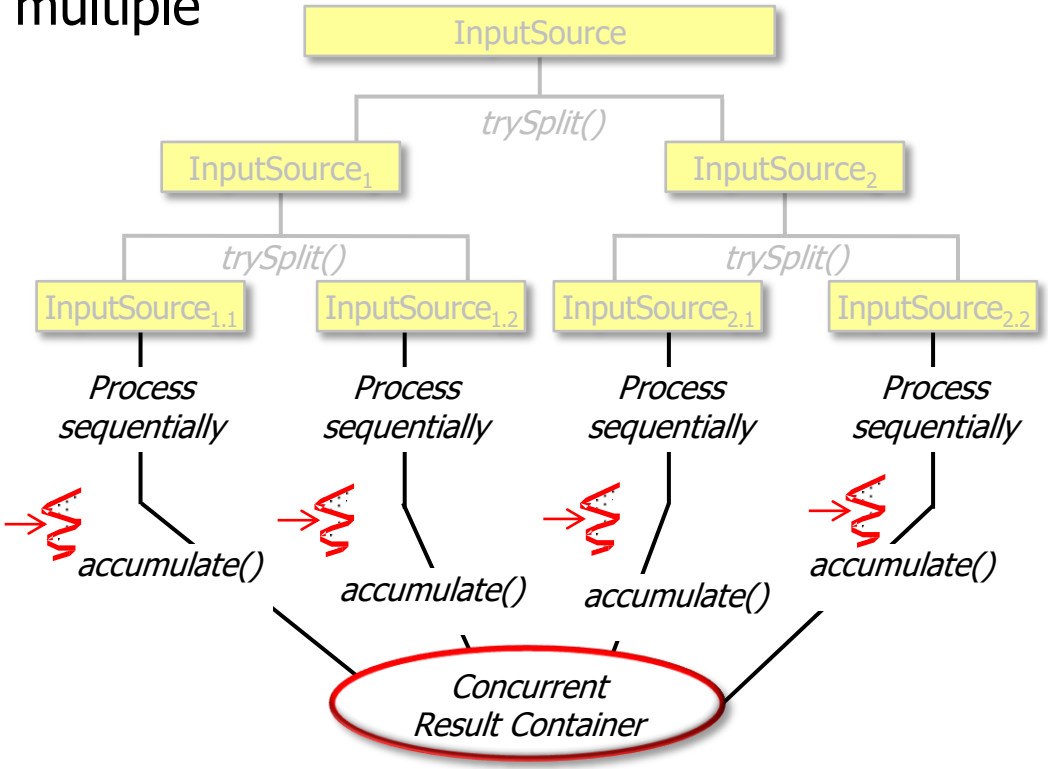
- A non-concurrent collector operates by merging sub-results



Different threads operate on different instances of the intermediate result containers

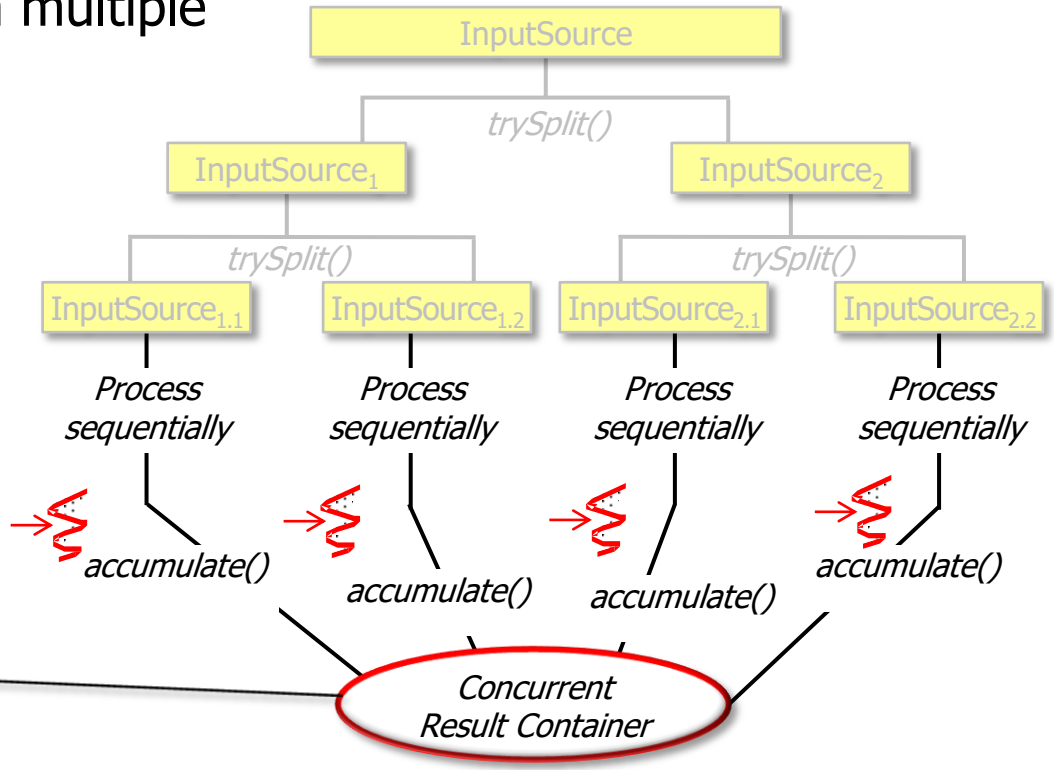
Demonstrating Collector Performance

- A concurrent collector creates one concurrent mutable result container & accumulates elements into it from multiple threads in a parallel stream



Demonstrating Collector Performance

- A concurrent collector creates one concurrent mutable result container & accumulates elements into it from multiple threads in a parallel stream



Thus there's no need to merge any intermediate sub-results!

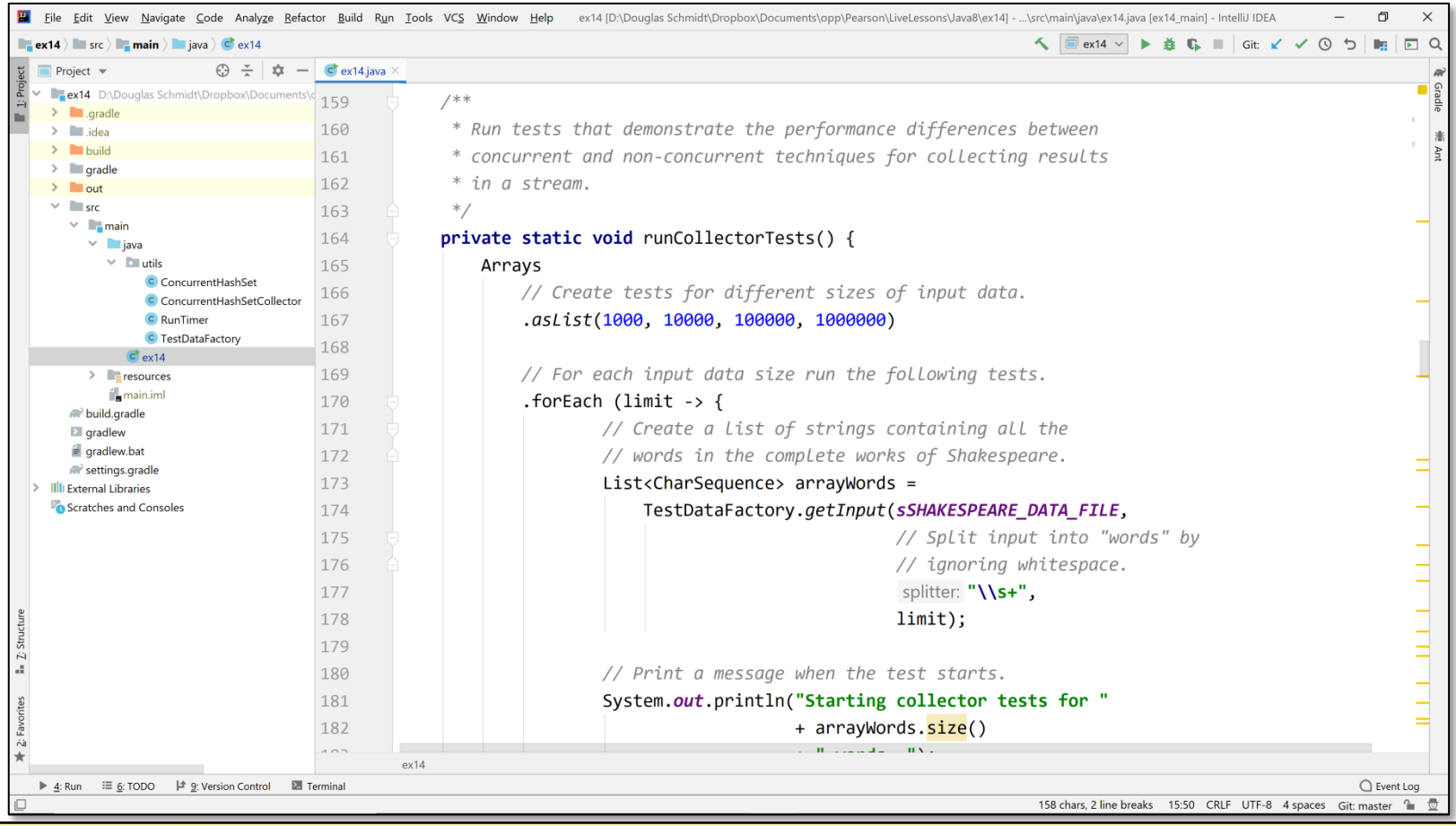
Demonstrating Collector Performance

- Results show collector differences become more significant as input grows

```
Starting collector tests for 1000 words..printing results
    21 msecs: sequential timeStreamCollectToSet()
    30 msecs: parallel timeStreamCollectToSet()
    39 msecs: sequential timeStreamCollectToConcurrentSet()
    59 msecs: parallel timeStreamCollectToConcurrentSet()
...
Starting collector tests for 100000 words....printing results
   219 msecs: parallel timeStreamCollectToConcurrentSet()
   364 msecs: parallel timeStreamCollectToSet()
   657 msecs: sequential timeStreamCollectToSet()
   804 msecs: sequential timeStreamCollectToConcurrentSet()
Starting collector tests for 883311 words....printing results
  1782 msecs: parallel timeStreamCollectToConcurrentSet()
  3010 msecs: parallel timeStreamCollectToSet()
  6169 msecs: sequential timeStreamCollectToSet()
  7652 msecs: sequential timeStreamCollectToConcurrentSet()
```

See upcoming lessons on "*When [Not] to Use Parallel Streams*"

Demonstrating Collector Performance



See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex14

Demonstrating Collector Performance

[Source code analysis video goes here!]

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex14

Java Parallel Stream Internals:
Demo’ing Collector Performance

The End

Java SearchWithParallel Spliterator Example

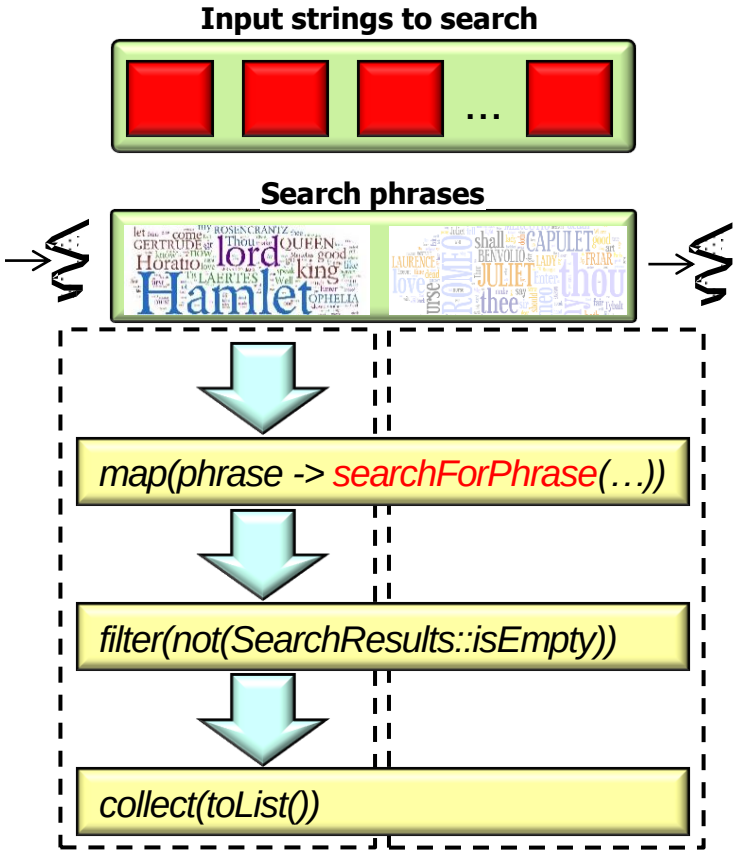
Introduction

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

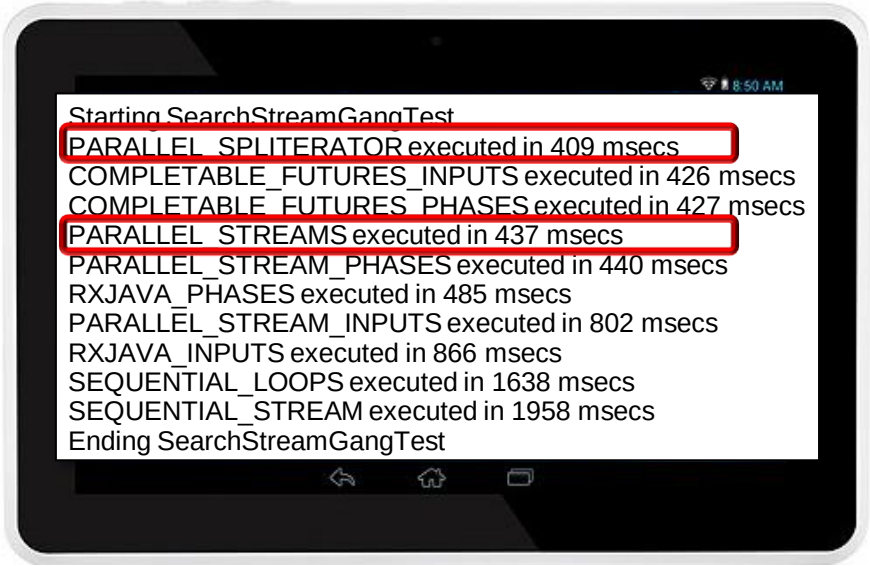
- Be aware of how a parallel Spliterator can improve parallel stream performance.

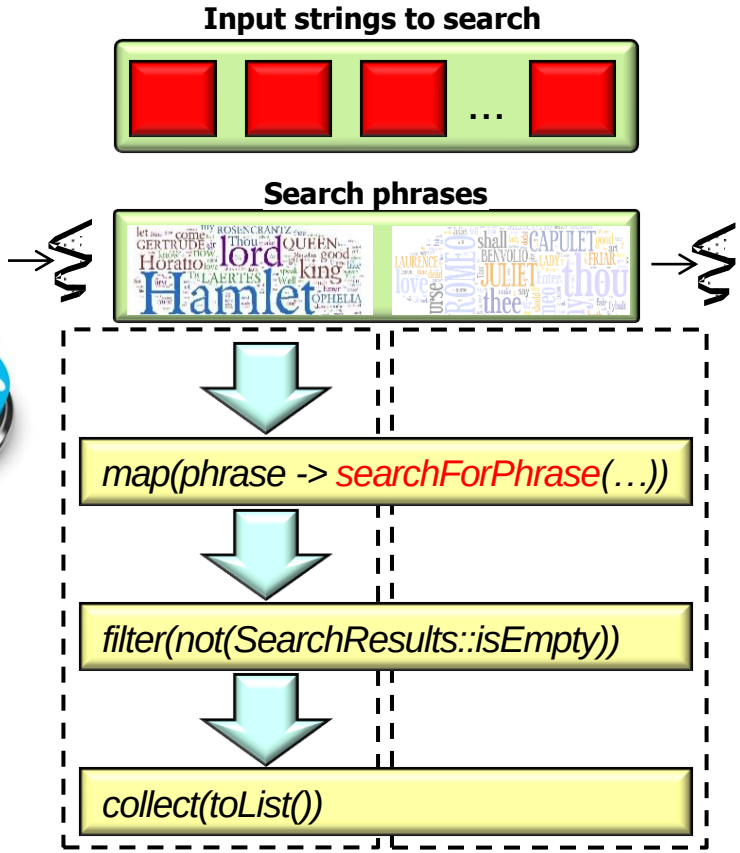
```
SearchResults searchForPhrase
   (..., boolean parallel) {
   return new SearchResults
     (..., StreamSupport.stream
       (new PhraseMatchSpliterator(...),
        parallel)
       .collect(toList()));
}
```



Learning Objectives in This Part of the Lesson

- Be aware of how a parallel Spliterator can improve parallel stream performance.
  - This solution fixes a "con" (limited performance) covered earlier.



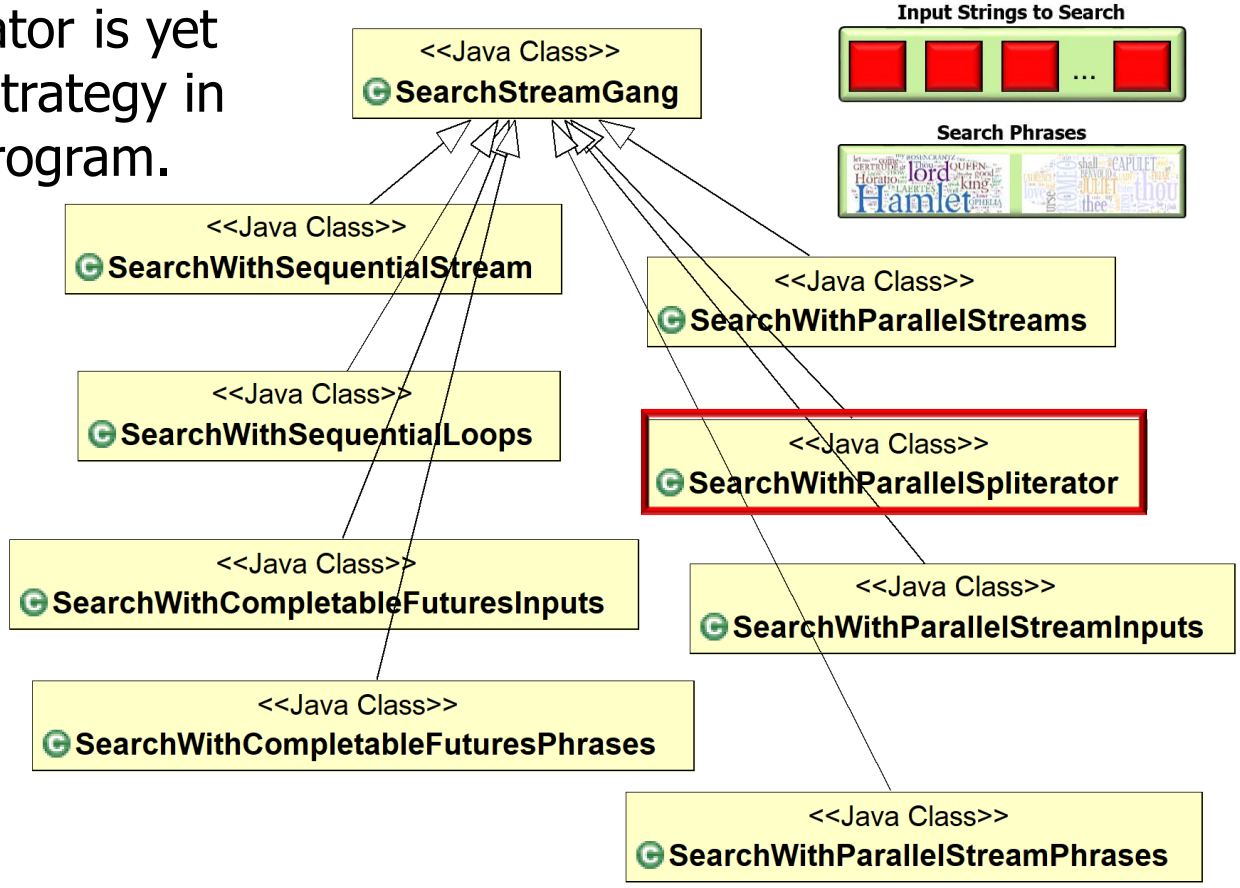


See *"Java SearchWithParallelStreams Example."*

Overview of SearchWithParallelSpliterator

Overview of SearchWithParallelSpliterator

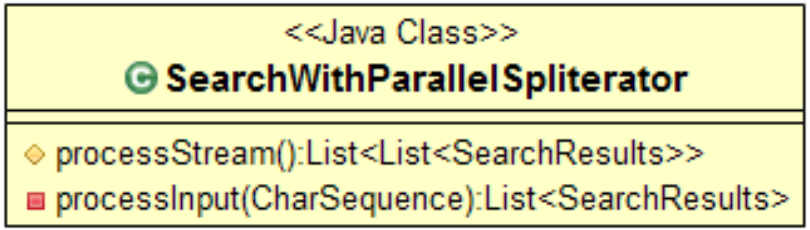
- SearchWithParallelSpliterator is yet another implementation strategy in the SearchStreamGang program.



See SearchStreamGang/src/main/java/livelessons/streamgangs/SearchWithParallelSpliterator.java

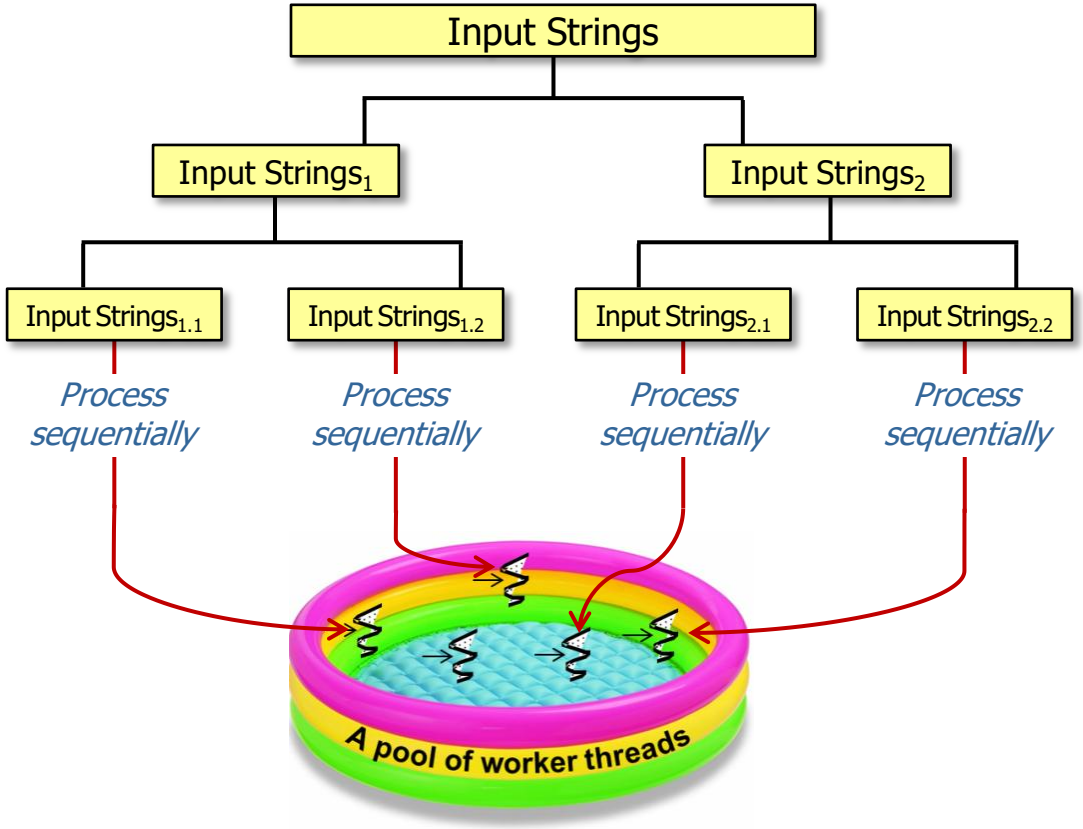
Overview of SearchWithParallelSpliterator

- SearchWithParallelSpliterator uses parallel streams in three ways.



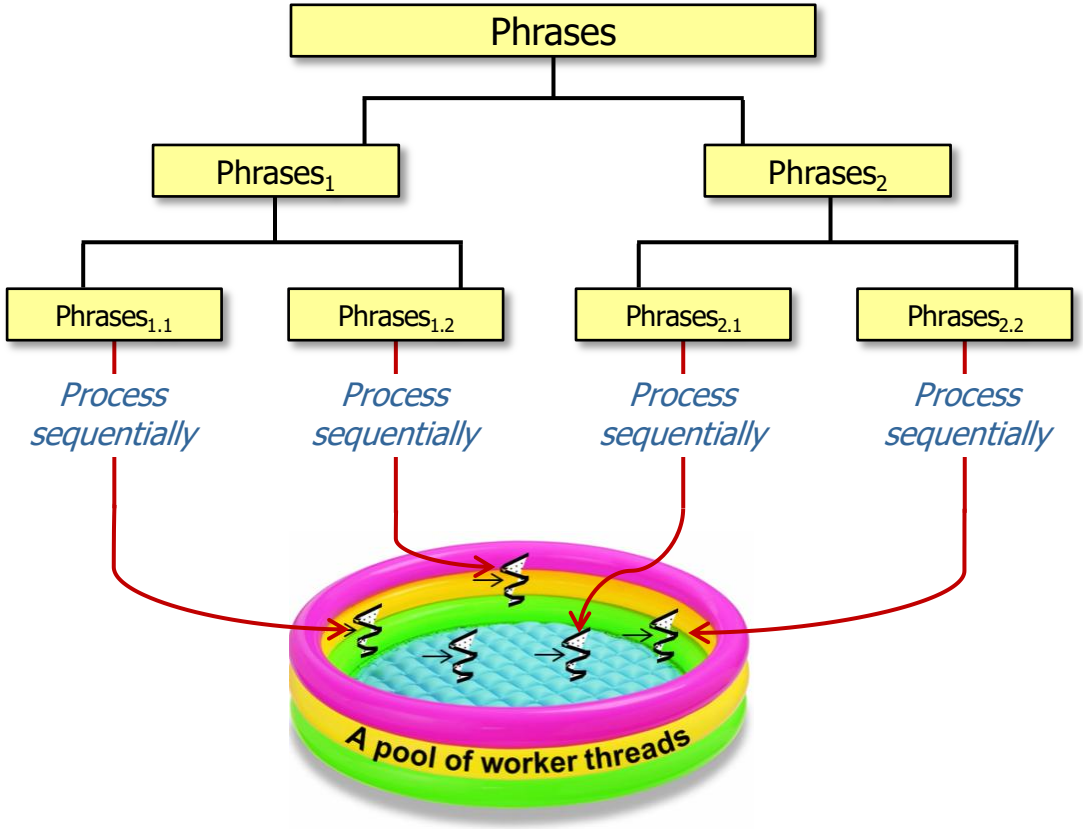
Overview of SearchWithParallelSpliterator

- SearchWithParallelSpliterator uses parallel streams in three ways.
  - Search chunks of input in parallel



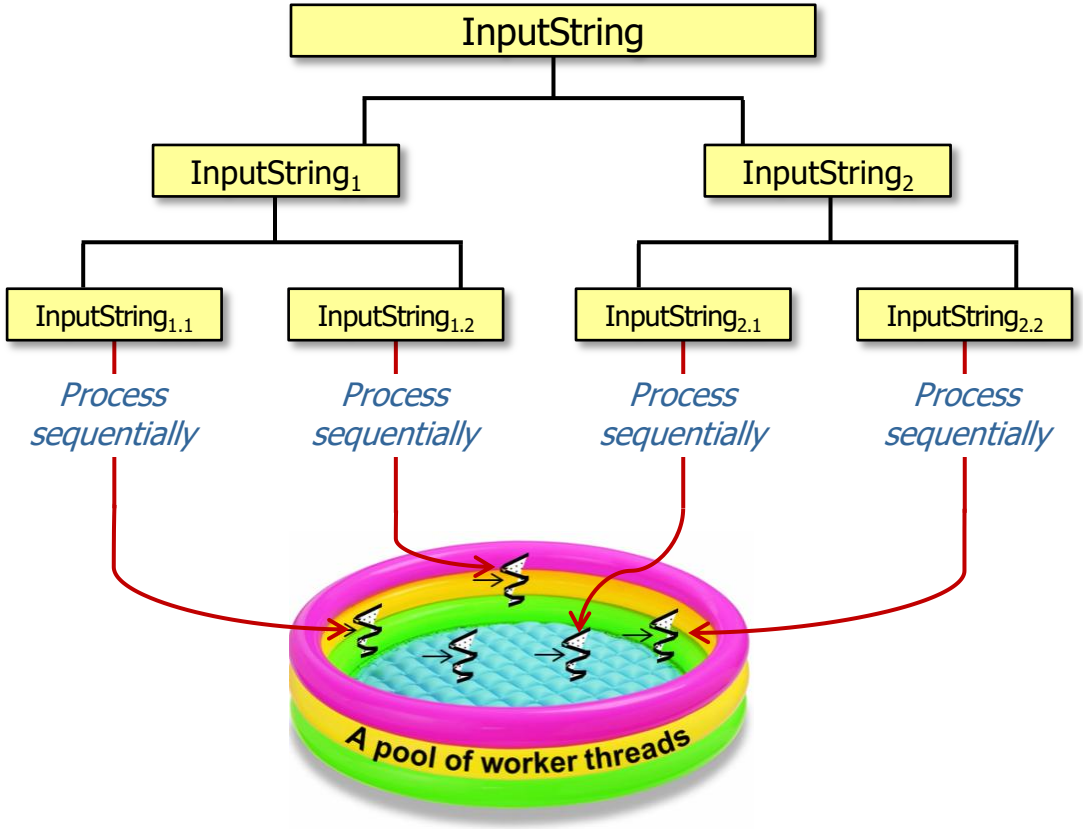
Overview of SearchWithParallelSpliterator

- SearchWithParallelSpliterator uses parallel streams in three ways.
  - Search chunks of input in parallel
  - Search chunks of phrases in parallel



Overview of SearchWithParallelSpliterator

- SearchWithParallelSpliterator uses parallel streams in three ways.
  - Search chunks of input in parallel
  - Search chunks of phrases in parallel
  - Search chunks of *each* input string in parallel



Overview of SearchWithParallelSpliterator

- SearchWithParallelSpliterator uses parallel streams in three ways.
  - Search chunks of input in parallel
  - Search chunks of phrases in parallel
  - Search chunks of *each* input string in parallel

SearchWithParallelSpliterator is thus the most aggressive parallelism strategy!

Overview of SearchWithParallelSpliterator

- The relative contribution of each parallel streams model is shown here:

Time for 38 strings = 462 ms (parallelSpliterator|parallelPhrases|parallelInput)

Time for 38 strings = 470 ms (sequentialSpliterator|parallelPhrases|parallelInput)

Time for 38 strings = 477 ms (sequentialSpliterator|parallelPhrases|sequentialInput)

Time for 38 strings = 490 ms (parallelSpliterator|parallelPhrases|sequentialInput)

Time for 38 strings = 498 ms (parallelSpliterator|sequentialPhrases|parallelInput)

Time for 38 strings = 510 ms (sequentialSpliterator|sequentialPhrases|parallelInput)

Time for 38 strings = 1326 ms (parallelSpliterator|sequentialPhrases|sequentialInput)

Time for 38 strings = 2463 ms (sequentialSpliterator|sequentialPhrases|sequentialInput)

Tests conducted on a 2.6 GHz six-core Lenovo P52 with 64 GB of RAM

Overview of SearchWithParallelSpliterator

- Longer input strings leverage the parallel spliterator even better:

Time for 2 strings = 452 ms (parallelSpliterator|parallelPhrases|parallelInput)

Time for 2 strings = 462 ms (sequentialSpliterator|parallelPhrases|parallelInput)

Time for 2 strings = 466 ms (sequentialSpliterator|parallelPhrases|sequentialInput)

Time for 2 strings = 478 ms (parallelSpliterator|parallelPhrases|sequentialInput)

Time for 2 strings = 788 ms (parallelSpliterator|sequentialPhrases|parallelInput)

Time for 2 strings = 1,298 ms (sequentialSpliterator|sequentialPhrases|parallelInput)

Time for 2 strings = 1,488 ms (parallelSpliterator|sequentialPhrases|sequentialInput)

Time for 2 strings = 2,467 ms (sequentialSpliterator|sequentialPhrases|sequentialInput)

Longer strings may provide better opportunity to leverage benefits of parallelism.

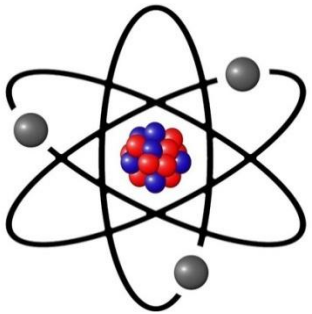
Overview of SearchWithParallelSpliterator

- SearchWithParallelSpliterator processInput() has just one minuscule change.

```
List<SearchResults> processInput(CharSequence inputSeq) {
  String title = getTitle(inputString);
  CharSequence input = inputSeq.subSequence(...);

  List<SearchResults> results = mPhrasesToFind
    .parallelStream()
    .map(phase ->
         searchForPhrase(phase, input, title, true))
    .filter(not(SearchResults::isEmpty))

    .collect(toList());
  return results;
}
```



The value of "true" triggers the use of a parallel search for a phrase in an input string.

Overview of SearchWithParallelSpliterator

- SearchForPhrase() uses a parallel Spliterator to break the input into "chunks" that are processed in parallel.

```
SearchResults searchForPhrase(String phrase, CharSequence input,
                              String title, boolean parallel) {
  return new SearchResults
    (..., ..., phrase, title, StreamSupport
     .stream(new PhraseMatchSpliterator(input, phrase),
             parallel)
     .collect(toList()));
}
```

See livelessons/streamgangs/SearchStreamGang.java

Overview of SearchWithParallelSpliterator

- SearchForPhrase() uses a parallel Spliterator to break the input into "chunks" that are processed in parallel.

```
SearchResults searchForPhrase(String phrase, CharSequence input,
                              String title, boolean parallel) {
  return new SearchResults
    (..., ..., phrase, title, StreamSupport
     .stream(new PhraseMatchSpliterator(input, phrase),
             parallel)
     .collect(toList()));
  }
```

StreamSupport.stream() creates a sequential or parallel stream via PhraseMatchSpliterator.

See docs.oracle.com/javase/8/docs/api/java/util/stream/StreamSupport.html#stream

Overview of SearchWithParallelSpliterator

- SearchForPhrase() uses a parallel Spliterator to break the input into "chunks" that are processed in parallel.

```
SearchResults searchForPhrase(String phrase, CharSequence input,
                              String title, boolean parallel) {
  return new SearchResults
    (..., ..., phrase, title, StreamSupport
     .stream(new PhraseMatchSpliterator(input, phrase),
             parallel)
     .collect(toList()));
  }
```

The value of "parallel" is true when searchForPhrase() is called in the SearchWithParallelSpliterator program.

See docs.oracle.com/javase/8/docs/api/java/util/stream/StreamSupport.html#stream

Overview of SearchWithParallelSpliterator

- SearchForPhrase() uses a parallel Spliterator to break the input into "chunks" that are processed in parallel.

```
SearchResults searchForPhrase(String phrase, CharSequence input,
                              String title, boolean parallel) {
  return new SearchResults
    (..., ..., phrase, title, StreamSupport
     .stream(new PhraseMatchSpliterator(input, phrase),
             parallel)
     .collect(toList()));
  }
```

We now focus in depth on the PhraseMatchSpliterator methods

See the rest of the lessons on "*Java SearchWithParallelSpliterator Example.*"

The End

Java SearchWithParallelSpliterator Example

Phrase Match Spliterator and Fields

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Be aware of how a parallel Spliterator can improve parallel stream performance.
- Know the intent of—and fields in—the PhraseMatchSpliterator.

```
class PhraseMatchSpliterator implements Spliterator<Result> {
  private CharSequence mInput;

  private final String mPhrase;

  private final Pattern mPattern;

  private Matcher mPhraseMatcher;

  private final int mMinSplitSize;

  private int mOffset = 0; ...
```

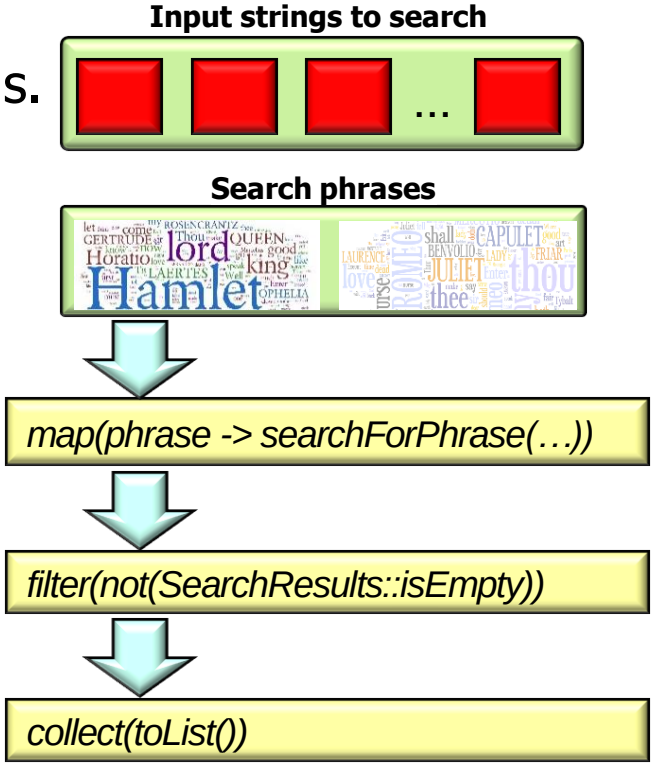
These fields are identical with the SearchWithSequentialStreams class.

See "*Java Sequential SearchStreamGang Example: Applying Spliterator.*"

Overview of PhraseMatchSpliterator

Overview of PhraseMatchSpliterator

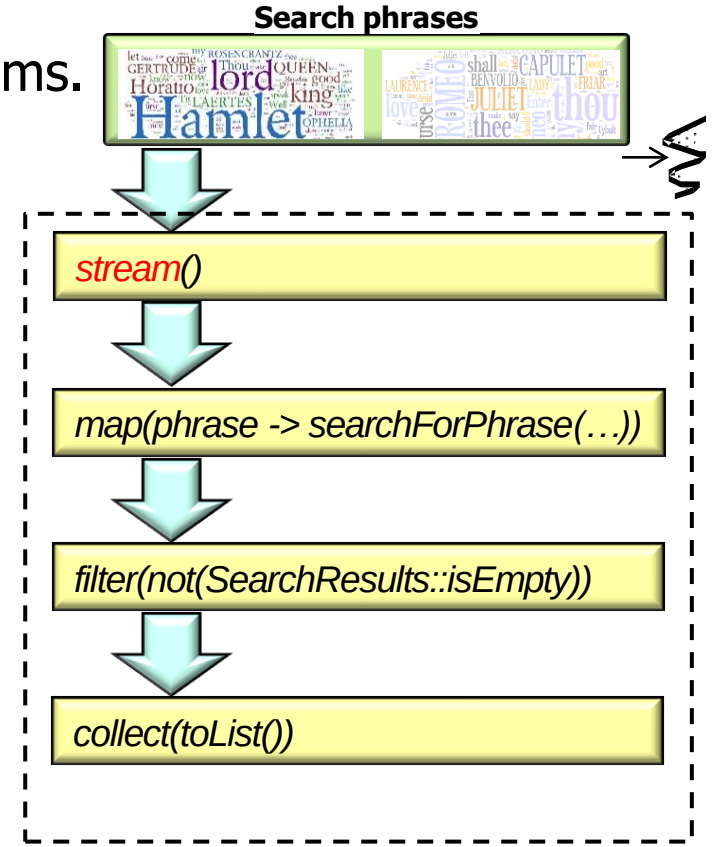
- SearchStreamGang uses PhraseMatchSpliterator that works for both sequential and parallel streams.



See github.com/douglascraigschmidt/LiveLessons/tree/master/SearchStreamGang

Overview of PhraseMatchSpliterator

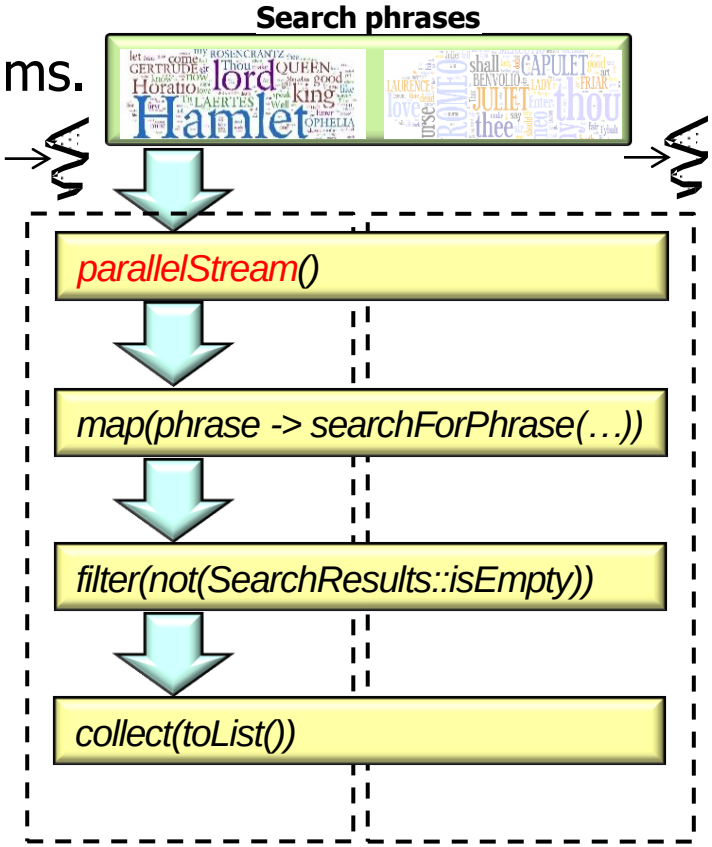
- SearchStreamGang uses PhraseMatchSpliterator that works for both sequential and parallel streams.
  - We focused on the sequential portions earlier
    - And will review them again now briefly



See *"Java Sequential SearchStreamGang Example: Applying Spliterator."*

Overview of PhraseMatchSpliterator

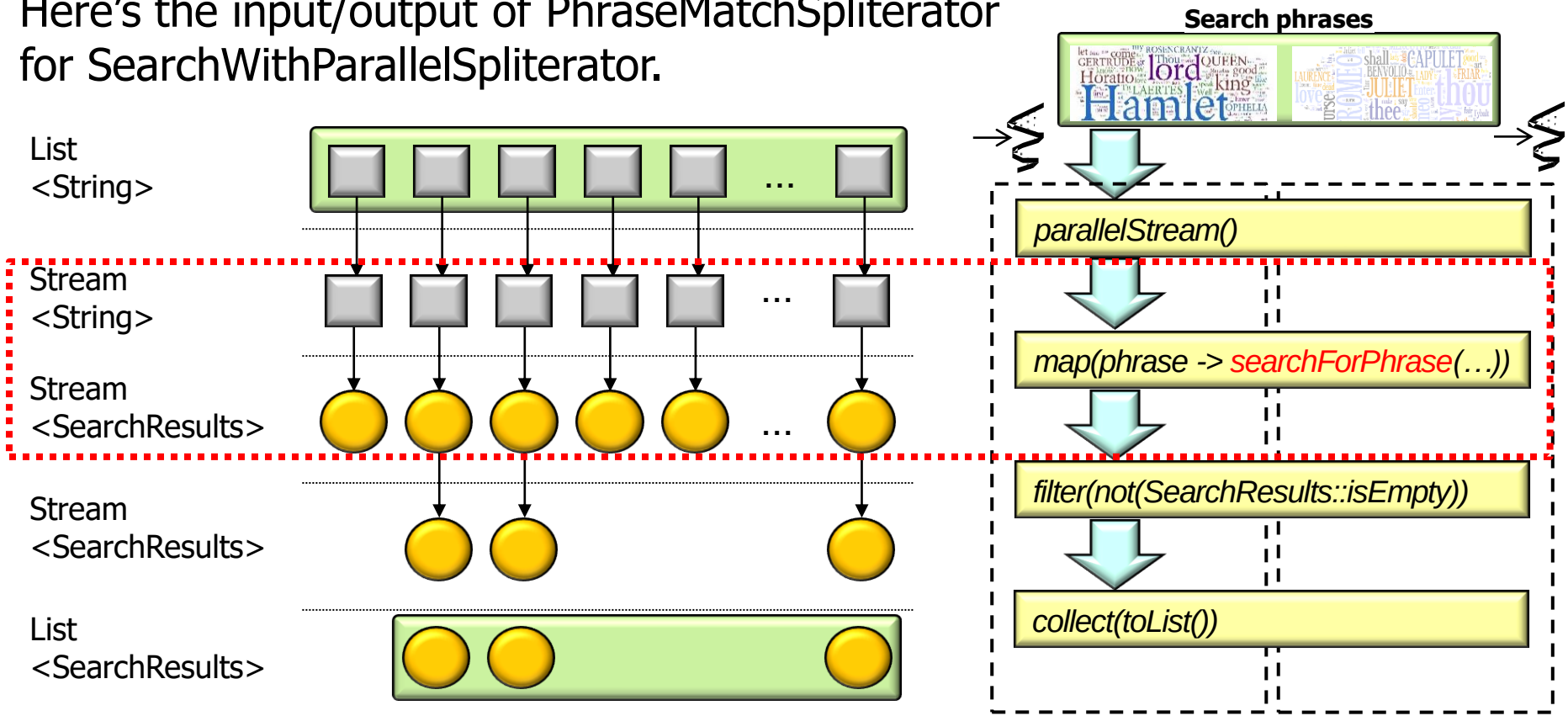
- SearchStreamGang uses PhraseMatchSpliterator that works for both sequential and parallel streams.
  - We focused on the sequential portions earlier.
  - We'll cover the parallel portions next.



The goal is to further optimize the performance of the parallel streams solution.

Overview of PhraseMatchSpliterator

- Here's the input/output of PhraseMatchSpliterator for SearchWithParallelSpliterator.

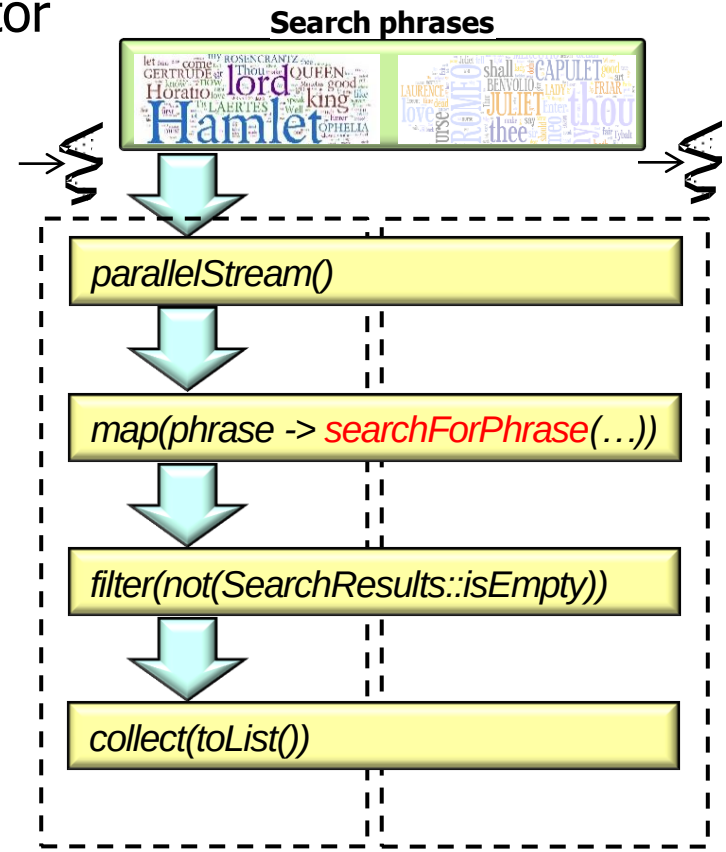


Overview of PhraseMatchSpliterator

- Here's the input/output of PhraseMatchSpliterator for SearchWithParallelSpliterator.

"Brevity is the soul of wit" at [54739]

". . .
My liege, and madam, to expostulate
What majesty should be, what duty is,
Why day is day, night is night, and time is time.
Were nothing but to waste night, day, and time.
Therefore, since brevity is the soul of wit,
And tediousness the limbs and outward flourishes,
I will be brief. Your noble son is mad.
Mad call I it; for, to define true madness,
What is't but to be nothing else but mad?
But let that go . . ."



This spliterator splits the input into multiple chunks and searches them in parallel.

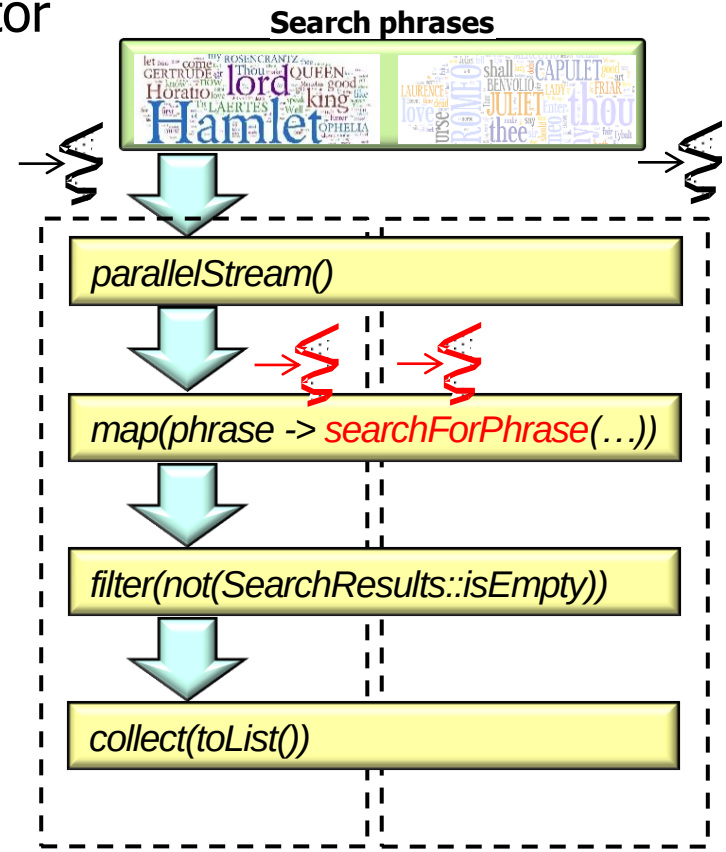
Overview of PhraseMatchSpliterator

- Here's the input/output of PhraseMatchSpliterator for SearchWithParallelSpliterator.

"*"Brevity is the soul of wit" at [54739]*

". . .
My liege, and madam, to expostulate
What majesty should be, what duty is,
Why day is day, night is night, and time is time.
Were nothing but to waste night, day, and time.
Therefore, since brevity is the soul of wit,"

"And tediousness the limbs and outward flourishes,
I will be brief. Your noble son is mad.
Mad call I it; for, to define true madness,
What is't but to be nothing else but mad?
But let that go . . ."



When the split occurs efficiently/evenly, the speedups can be substantial!

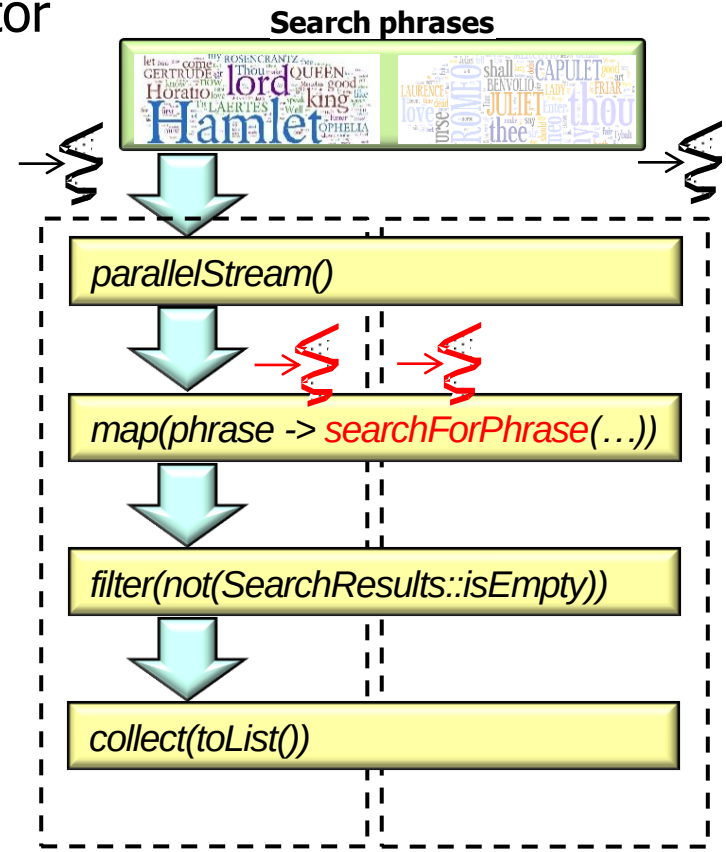
Overview of PhraseMatchSpliterator

- Here's the input/output of PhraseMatchSpliterator for SearchWithParallelSpliterator.

"Brevity is the soul of wit" not found!

". . .
My liege, and madam, to expostulate
What majesty should be, what duty is,
Why day is day, night is night, and time is time.
Were nothing but to waste night, day, and time.
Therefore, since brevity is the soul of"

"Wit, And tediousness the limbs and outward flourishes, I will be brief. Your noble son is mad.
Mad call I it; for, to define true madness,
What is't but to be nothing else but mad?
But let that go . . ."



However, the spliterator must be careful not to split input *across* phrases . . .

Analysis of PhraseMatchSpliterator Fields

Analysis of PhraseMatchSpliterator Fields

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults, result objects that match the # of times a phrase appears in an input string.

```
class PhraseMatchSpliterator implements Spliterator<Result> {
  private CharSequence mInput;

  private final String mPhrase;

  private final Pattern mPattern;

  private final Matcher mPhraseMatcher;

  private final int mMinSplitSize;

  private int mOffset = 0;
  ...
```

Spliterator is an interface that defines eight methods, including tryAdvance() and trySplit().

See SearchStreamGang/src/main/java/livelessons/utils/PhraseMatchSpliterator.java

Analysis of PhraseMatchSpliterator Fields

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults, result objects that match the # of times a phrase appears in an input string.

```
class PhraseMatchSpliterator implements Spliterator<Result> {
  private CharSequence mInput;

  private final String mPhrase;

  private final Pattern mPattern;

  private final Matcher mPhraseMatcher;

  private final int mMinSplitSize;

  private int mOffset = 0;
  ...
```

These fields implement PhraseMatchSpliterator for both sequential and parallel use cases.

Some fields are updated in the trySplit() method, which is why they aren't final.

Analysis of PhraseMatchSpliterator Fields

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults, result objects that match the # of times a phrase appears in an input string.

```
class PhraseMatchSpliterator implements Spliterator<Result> {
  private CharSequence mInput;

  private final String mPhrase;

  private final Pattern mPattern;

  private Matcher mPhraseMatcher;

  private final int mMinSplitSize;

  private int mOffset = 0;
  ...
```

Contains a single work of Shakespeare

Analysis of PhraseMatchSpliterator Fields

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults, result objects that match the # of times a phrase appears in an input string.

```
class PhraseMatchSpliterator implements Spliterator<Result> {
  private CharSequence mInput;

  private final String mPhrase;

  private final Pattern mPattern;

  private Matcher mPhraseMatcher;

  private final int mMinSplitSize;

  private int mOffset = 0;
  ...
```

Contains the phrase to search for in the work

Analysis of PhraseMatchSpliterator Fields

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults, result objects that match the # of times a phrase appears in an input string.

```
class PhraseMatchSpliterator implements Spliterator<Result> {
  private CharSequence mInput;

  private final String mPhrase;

  private final Pattern mPattern;

  private Matcher mPhraseMatcher;

  private final int mMinSplitSize;

  private int mOffset = 0;
  ...
```

Contains the regular expression representation of the phrase

See docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html

Analysis of PhraseMatchSpliterator Fields

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults, result objects that match the # of times a phrase appears in an input string.

```
class PhraseMatchSpliterator implements Spliterator<Result> {
  private CharSequence mInput;

  private final String mPhrase;

  private final Pattern mPattern;

  private Matcher mPhraseMatcher;

  private final int mMinSplitSize;

  private int mOffset = 0;
  ...
```

Contains a matcher that searches for the phrase in the input

See docs.oracle.com/javase/8/docs/api/java/util/regex/Matcher.html

Analysis of PhraseMatchSpliterator Fields

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults, result objects that match the # of times a phrase appears in an input string.

```
class PhraseMatchSpliterator implements Spliterator<Result> {
  private CharSequence mInput;

  private final String mPhrase;

  private final Pattern mPattern;

  private Matcher mPhraseMatcher;

  private final int mMinSplitSize;

  private int mOffset = 0;
  ...
```

Dictates the minimum size to perform a split

Analysis of PhraseMatchSpliterator Fields

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults, result objects that match the # of times a phrase appears in an input string.

```
class PhraseMatchSpliterator implements Spliterator<Result> {
  private CharSequence mInput;

  private final String mPhrase;

  private final Pattern mPattern;

  private Matcher mPhraseMatcher;

  private final int mMinSplitSize;

  private int mOffset = 0;
  ...
```

The offset needed to return the appropriate index into the original input string

This value is reset by each Spliterator to account for different chunks.

Java SearchWithParallelSpliterator Example:
PhraseMatchSpliterator and Fields

The End

Java SearchWithParallelSpliterator Example

PhraseMatchSpliterator Constructor and tryAdvance()

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Be aware of how a parallel Spliterator can improve parallel stream performance.
- Know the intent of—and fields in—the PhraseMatchSpliterator.
- Recognize the PhraseMatchSpliterator constructor and tryAdvance() method implementation.

```
class PhraseMatchSpliterator
      implements Spliterator<Result> {
  ...
  PhraseMatchSpliterator(CharSequence input,
                         String phrase) { ... }

  boolean tryAdvance(Consumer<? super Result> action) { ... }
  ...
```

These methods are identical w/the SearchWithSequentialStreams class.

See "*Java Sequential SearchStreamGang Example: Applying Spliterator.*"

Analysis of PhraseMatchSpliterator Constructor

Analysis of PhraseMatchSpliterator Constructor

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults, result objects that match the # of times a phrase appears in an input string.

```
class PhraseMatchSpliterator implements Spliterator<Result> {
  ...
  PhraseMatchSpliterator(CharSequence input, String phrase) {
    String regexPhrase = "\\b" + phrase.trim().replaceAll
                                      ("\\s+", "\\\\b\\\\s+\\\\b")
                               + "\\b"; ...

    mPattern = Pattern.compile(regexPhrase,
                  Pattern.CASE_INSENSITIVE | Pattern.DOTALL);
    mPhraseMatcher = mPattern.matcher(input);
    mInput = input; mPhrase = phrase;
    mMinSplitSize = input.length() / 2;
  } ...
```

See SearchStreamGang/src/main/java/livelessons/utils/PhraseMatchSpliterator.java

Analysis of PhraseMatchSpliterator Constructor

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults, result objects that match the # of times a phrase appears in an input string.

```
class PhraseMatchSpliterator implements Spliterator<Result> {
  ...
  PhraseMatchSpliterator(CharSequence input, String phrase) {
    String regexPhrase = "\\b" + phrase.trim().replaceAll
                                   ("\\s+", "\\\\b\\\\s+\\\\b")
                       + "\\b"; ...
    mPattern = Pattern.compile(regexPhrase,
                  Pattern.CASE_INSENSITIVE | Pattern.DOTALL);
    mPhraseMatcher = mPattern.matcher(input);
    mInput = input; mPhrase = phrase;
    mMinSplitSize = input.length() / 2;
  } ...
```

One work of Shakespeare and a phrase to search for in this work

Analysis of PhraseMatchSpliterator Constructor

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults, result objects that match the # of times a phrase appears in an input string.

```
class PhraseMatchSpliterator implements Spliterator<Result> {
  ...
  PhraseMatchSpliterator(CharSequence input, String phrase) {
    String regexPhrase = "\\b" + phrase.trim().replaceAll
                                   ("\\s+", "\\\\b\\\\s+\\\\b")
                       + "\\b"; ...

    mPattern = Pattern.compile(regexPhrase,
                               Pattern.CASE_INSENSITIVE | Pattern.DOTALL);
    mPhraseMatcher = mPattern.matcher(input);
    mInput = input; mPhrase = phrase;
    mMinSplitSize = input.length() / 2;
  } ...
```

Create a regex that matches phrases.

See docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html

Analysis of PhraseMatchSpliterator Constructor

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults, result objects that match the # of times a phrase appears in an input string.

```
class PhraseMatchSpliterator implements Spliterator<Result> {
  ...
  PhraseMatchSpliterator(CharSequence input, String phrase) {
    String regexPhrase = "\\b" + phrase.trim().replaceAll
                                   ("\\s+", "\\\\b\\\\s+\\\\b")
                       + "\\b"; ...
    mPattern = Pattern.compile(regexPhrase,
                               Pattern.CASE_INSENSITIVE | Pattern.DOTALL);
    mPhraseMatcher = mPattern.matcher(input);
    mInput = input; mPhrase = phrase;
    mMinSplitSize = input.length() / 2;
  } ...
```

A regex is compiled into a pattern that matches a phrase across lines.

See docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html

Analysis of PhraseMatchSpliterator Constructor

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults, result objects that match the # of times a phrase appears in an input string.

```
class PhraseMatchSpliterator implements Spliterator<Result> {
  ...
  PhraseMatchSpliterator(CharSequence input, String phrase) {
    String regexPhrase = "\\b" + phrase.trim().replaceAll
                                    ("\\s+", "\\\\b\\\\s+\\\\b")
                               + "\\b"; ...

    mPattern = Pattern.compile(regexPhrase,
                  Pattern.CASE_INSENSITIVE | Pattern.DOTALL);
    mPhraseMatcher = mPattern.matcher(input);
    mInput = input; mPhrase = phrase;
    mMinSplitSize = input.length() / 2;
  } ...
```

A matcher is created to search the input for the regex pattern.

See docs.oracle.com/javase/8/docs/api/java/util/regex/Matcher.html

Analysis of PhraseMatchSpliterator Constructor

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults, result objects that match the # of times a phrase appears in an input string.

```
class PhraseMatchSpliterator implements Spliterator<Result> {
  ...
  PhraseMatchSpliterator(CharSequence input, String phrase) {
    String regexPhrase = "\\b" + phrase.trim().replaceAll
                                    ("\\s+", "\\\\b\\\\s+\\\\b")
                               + "\\b"; ...

    mPattern = Pattern.compile(regexPhrase,
                  Pattern.CASE_INSENSITIVE | Pattern.DOTALL);
    mPhraseMatcher = mPattern.matcher(input);
    mInput = input; mPhrase = phrase;
    mMinSplitSize = input.length() / 2;
  } ...
```

Set key fields with parameters.

Analysis of PhraseMatchSpliterator Constructor

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults, result objects that match the # of times a phrase appears in an input string.

```
class PhraseMatchSpliterator implements Spliterator<Result> {
  ...
  PhraseMatchSpliterator(CharSequence input, String phrase) {
    String regexPhrase = "\\b" + phrase.trim().replaceAll
                                    ("\\s+", "\\\\b\\\\s+\\\\b")
                               + "\\b"; ...

    mPattern = Pattern.compile(regexPhrase,
                  Pattern.CASE_INSENSITIVE | Pattern.DOTALL);
    mPhraseMatcher = mPattern.matcher(input);
    mInput = input; mPhrase = phrase;
    mMinSplitSize = input.length() / 2;
  } ...
```

Define the minimum split size.

This field is used by the trySplit() method for a parallel Spliterator.

Analysis of PhraseMatchSpliterator tryAdvance() Method

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults, result objects that match the # of times a phrase appears in an input string.

```
class PhraseMatchSpliterator implements Spliterator<Result> {
  ...
  boolean tryAdvance(Consumer<? super Result> action) {
    if (!mPhraseMatcher.find())
        return false;

    else {
        action.accept(new Result
                              (mOffset + mPhraseMatcher.start()));
        return true;
    }
  }
  ...
```

Called by Java streams framework, which attempts to advance the Spliterator by one matching phrase.

See docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html#tryAdvance

Analysis of PhraseMatchSpliterator tryAdvance() Method

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults, result objects that match the # of times a phrase appears in an input string.

```
class PhraseMatchSpliterator implements Spliterator<Result> {
  ...
  boolean tryAdvance(Consumer<? super Result> action) {
    if (!mPhraseMatcher.find())
        return false;

    else {
        action.accept(new Result
                              (mOffset + mPhraseMatcher.start()));
        return true;
    }
  }
  ...
```

Passes the result (if any) back "by reference" to the streams framework.

See docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html

Analysis of PhraseMatchSpliterator tryAdvance() Method

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults, result objects that match the # of times a phrase appears in an input string.

```
class PhraseMatchSpliterator implements Spliterator<Result> {
  ...
  boolean tryAdvance(Consumer<? super Result> action) {
    if (!mPhraseMatcher.find())
        return false;

    else {
        action.accept(new Result
                              (mOffset + mPhraseMatcher.start()));
        return true;
    }
  }
  ...
```

Check if any remaining phrases in the input match the regex.

See docs.oracle.com/javase/8/docs/api/java/util/regex/Matcher.html#find

Analysis of PhraseMatchSpliterator tryAdvance() Method

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults, result objects that match the # of times a phrase appears in an input string.

```
class PhraseMatchSpliterator implements Spliterator<Result> {
  ...
  boolean tryAdvance(Consumer<? super Result> action) {
    if (!mPhraseMatcher.find())
        return false;

    else {
        action.accept(new Result
                              (mOffset + mPhraseMatcher.start()));
        return true;
    }
  }
  ...
```

Inform the streams framework to cease calling tryAdvance() if there's no match.

Analysis of PhraseMatchSpliterator tryAdvance() Method

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults, result objects that match the # of times a phrase appears in an input string.

```
class PhraseMatchSpliterator implements Spliterator<Result> {
  ...
  boolean tryAdvance(Consumer<? super Result> action) {
    if (!mPhraseMatcher.find())
        return false;

    else {
        action.accept(new Result
                              (mOffset + mPhraseMatcher.start()));
        return true;
    }
  }
  ...
```

If there is a match, then accept() keeps track of which index in the input string the match occurred.

See docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html#accept

Analysis of PhraseMatchSpliterator tryAdvance() Method

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults, result objects that match the # of times a phrase appears in an input string.

```
class PhraseMatchSpliterator implements Spliterator<Result> {
  ...
  boolean tryAdvance(Consumer<? super Result> action) {
    if (!mPhraseMatcher.find())
        return false;

    else {
        action.accept(new Result
                              (mOffset + mPhraseMatcher.start()));
        return true;
    }
  }
  ...
```

Inform the streams framework to continue calling tryAdvance().

Java SearchWithParallelSpliterator Example:
PhraseMatchSpliterator Constructor and tryAdvance()

The End

Java SearchWithParallelSpliterator Example

trySplit()

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Be aware of how a parallel Spliterator can improve parallel stream performance.
- Know the intent of—and fields in—the PhraseMatchSpliterator.
- Recognize the PhraseMatchSpliterator constructor and tryAdvance() method implementation.
- Understand the PhraseMatchSpliterator trySplit() method implementation.

```
class PhraseMatchSpliterator implements Spliterator<Result> {
  Spliterator<Result> trySplit() {
    if (mInput.length() <= mMinSplitSize) return null;
    int startPos, splitPos = mInput.length() / 2;
    if ((startPos = computeStartPos(splitPos)) < 0) return null;
    if ((splitPos = tryToUpdateSplitPos(startPos, splitPos)) < 0)
      return null;
    return splitInput(splitPos);
    ...
```

This method is used with the SearchWithParallelSpliterators class.

Analysis of the PhraseMatchSpliterator trySplit() Method

Analysis of the PhraseMatchSpliterator trySplit() Method

- The streams framework uses trySplit() to partition a work of Shakepeare into chunks that can be searched in parallel.

```
class PhraseMatchSpliterator implements Spliterator<Result> {
  ...
  Spliterator<Result> trySplit() { ... }

  int computeStartPos(int splitPos) { ... }

  int tryToUpdateSplitPos(int startPos,
                          int splitPos) { ... }

  PhraseMatchSpliterator splitInput(int splitPos) { ... }

  ...
```

These methods are used for parallel streams.

Analysis of the PhraseMatchSpliterator trySplit() Method

- The streams framework uses trySplit() to partition a work of Shakepeare into chunks that can be searched in parallel.

```
class PhraseMatchSpliterator implements Spliterator<Result> {
  ...
  Spliterator<Result> trySplit() { ... }

  int computeStartPos(int splitPos) { ... }

  int tryToUpdateSplitPos(int startPos,
                          int splitPos) { ... }

  PhraseMatchSpliterator splitInput(int splitPos) { ... }

  ...
```

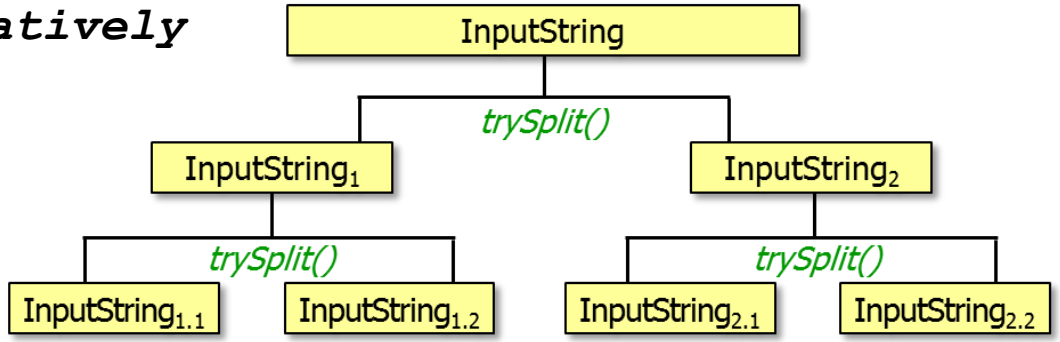
These methods are used for parallel streams.

There is *no* synchronization in any of these methods!

Analysis of the PhraseMatchSpliterator trySplit() Method

- The streams framework uses trySplit() to partition a work of Shakepeare into chunks that can be searched in parallel.

```
class PhraseMatchSpliterator implements Spliterator<Result> {
  ...
  Spliterator<Result> trySplit() {
    if (input is below minimum size) return null
    else {
      split input in 2 relatively
        even-sized chunks
      return a spliterator
        for "left chunk"
    }
  }
  ...
```

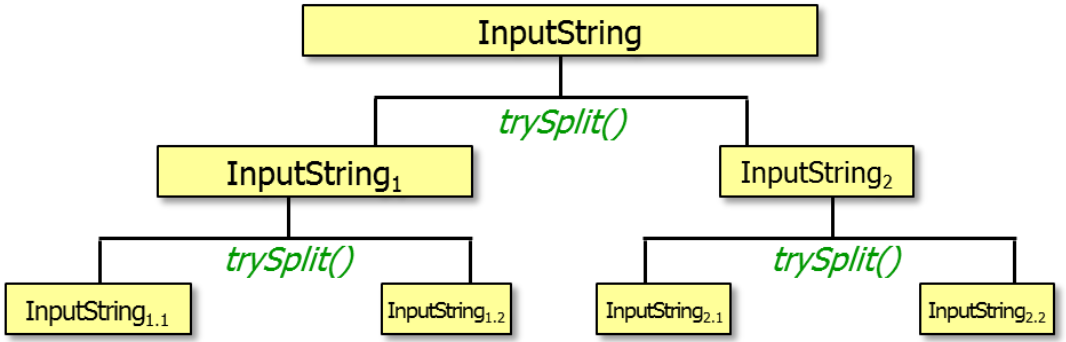


trySplit() attempts to split the input "evenly" so phrases can be matched in parallel.

Analysis of the PhraseMatchSpliterator trySplit() Method

- The streams framework uses trySplit() to partition a work of Shakepeare into chunks that can be searched in parallel.

```
class PhraseMatchSpliterator implements Spliterator<Result> {
  ...
  Spliterator<Result> trySplit() {
```



Splits needn't be perfectly equal in order for the spliterator to run efficiently.

Analysis of the PhraseMatchSpliterator trySplit() Method

- The streams framework uses trySplit() to partition a work of Shakepeare into chunks that can be searched in parallel.

```
class PhraseMatchSpliterator implements Spliterator<Result> {
  ...
  Spliterator<Result> trySplit() {
    if (mInput.length() <= mMinSplitSize) return null;
    int startPos,
        splitPos = mInput.length() / 2;

    if ((startPos = computeStartPos(splitPos)) < 0) return null;

    if ((splitPos = tryToUpdateSplitPos(startPos, splitPos)) < 0)
      return null;

    return splitInput(splitPos); ...
```

This code is heavily commented, so please check it out.

Analysis of the PhraseMatchSpliterator trySplit() Method

- The streams framework uses trySplit() to partition a work of Shakepeare into chunks that can be searched in parallel.

```
class PhraseMatchSpliterator implements Spliterator<Result> {
  ...
  Spliterator<Result> trySplit() {
    if (mInput.length() <= mMinSplitSize) return null;
    int startPos,
        splitPos = mInput.length() / 2;

    if ((startPos = computeStartPos(splitPos)) < 0) return null;

    if ((splitPos = tryToUpdateSplitPos(startPos, splitPos)) < 0)
      return null;

    return splitInput(splitPos); ...
```

Bail out if input is too small to split further.

Analysis of the PhraseMatchSpliterator trySplit() Method

"... Therefore, since brevity is the soul of" "wit, And tediousness the limbs and outward ..."


```
class PhraseMatchSpliterator implements Spliterator<Result> {
  ...
  Spliterator<Result> trySplit() {
    if (mInput.length() <= mMinSplitSize) return null;
    int startPos,
        splitPos = mInput.length() / 2;

    if ((startPos = computeStartPos(splitPos)) < 0) return null;

    if ((splitPos = tryToUpdateSplitPos(startPos, splitPos)) < 0)
      return null;

    return splitInput(splitPos); ...
```

Initial guess at the split position

Analysis of the PhraseMatchSpliterator trySplit() Method

"... Therefore, since brevity is the soul of" "wit, And tediousness the limbs and outward ..."


```
class PhraseMatchSpliterator implements Spliterator<Result> {
  ...
  Spliterator<Result> trySplit() {
    if (mInput.length() <= mMinSplitSize) return null;
    int startPos,
        splitPos = mInput.length() / 2;

    if ((startPos = computeStartPos(splitPos)) < 0) return null;

    if ((splitPos = tryToUpdateSplitPos(startPos, splitPos)) < 0)
      return null;

    return splitInput(splitPos); ...
```

Initial guess at where to start the search

Analysis of the PhraseMatchSpliterator trySplit() Method

"... Therefore, since brevity is the soul of" "wit, And tediousness the limbs and outward ..."


```
class PhraseMatchSpliterator implements Spliterator<Result> {
  ...
  int computeStartPos(int splitPos) {

    int phraseLength = mPhrase.length();

    int startPos = splitPos - phraseLength;

    if (startPos < 0 || phraseLength > splitPos)
      return -1;
    else
      return startPos;
}
```

Identify the position to start determining if a phrase spans the split position.

Analysis of the PhraseMatchSpliterator trySplit() Method

"... Therefore, since brevity is the soul of" "wit, And tediousness the limbs and outward ..."


```
class PhraseMatchSpliterator implements Spliterator<Result> {
  ...
  int computeStartPos(int splitPos) {

    int phraseLength = mPhrase.length();

    int startPos = splitPos - phraseLength;

    if (startPos < 0 || phraseLength > splitPos)
      return -1;
    else
      return startPos;
}
```

Store the length of the phrase.

Analysis of the PhraseMatchSpliterator trySplit() Method

"... Therefore, since brevity is the soul of" "wit, And tediousness the limbs and outward ..."


```
class PhraseMatchSpliterator implements Spliterator<Result> {
  ...
  int computeStartPos(int splitPos) {

    int phraseLength = mPhrase.length();

    int startPos = splitPos - phraseLength;

    if (startPos < 0 || phraseLength > splitPos)
      return -1;
    else
      return startPos;
}
```

Compute the initial startPos by subtracting the phrase length from the splitPos.

Analysis of the PhraseMatchSpliterator trySplit() Method

"... Therefore, since brevity is the soul of" "wit, And tediousness the limbs and outward ..."


```
class PhraseMatchSpliterator implements Spliterator<Result> {
  ...
  int computeStartPos(int splitPos) {

    int phraseLength = mPhrase.length();

    int startPos = splitPos - phraseLength;

    if (startPos < 0 || phraseLength > splitPos)
      return -1;
    else
      return startPos;
}
```

Fail if phrase is too long for this input segment.

Analysis of the PhraseMatchSpliterator trySplit() Method

"... Therefore, since brevity is the soul of" "wit, And tediousness the limbs and outward ..."


```
class PhraseMatchSpliterator implements Spliterator<Result> {
  ...
  int computeStartPos(int splitPos) {

    int phraseLength = mPhrase.length();

    int startPos = splitPos - phraseLength;

    if (startPos < 0 || phraseLength > splitPos)
      return -1;
    else
      return startPos;
}
```

Return to the computed start position.

Analysis of the PhraseMatchSpliterator trySplit() Method

"... Therefore, since brevity is the soul of" "wit, And tediousness the limbs and outward ..."


```
class PhraseMatchSpliterator implements Spliterator<Result> {
  ...
  Spliterator<Result> trySplit() {
    if (mInput.length() <= mMinSplitSize) return null;
    int startPos,
        splitPos = mInput.length() / 2;

    if ((startPos = computeStartPos(splitPos)) < 0) return null;

    if ((splitPos = tryToUpdateSplitPos(startPos, splitPos)) < 0)
      return null;

    return splitInput(splitPos); ...
```

Update splitPos if phrase spans the initial splitPos.

Analysis of the PhraseMatchSpliterator trySplit() Method

"... Therefore, since brevity is the soul of" "wit, And tediousness the limbs and outward ..."


```
class PhraseMatchSpliterator implements Spliterator<Result> {
  ...
  int tryToUpdateSplitPos(int startPos, int splitPos) {
    int endPos =
      splitPos + mPattern.toString().length();
    if (endPos >= mInput.length()) return -1;
    CharSequence substr =
      mInput.subSequence(startPos, endPos);
    Matcher pm = mPattern.matcher(substr);
    if (pm.find()) splitPos = startPos
      + pm.start() + pm.group().length();
    return splitPos;
  }
```

Don't split a string across a phrase.

Analysis of the PhraseMatchSpliterator trySplit() Method

"... Therefore, since brevity is the soul of" "wit, And tediousness the limbs and outward ..."


```
class PhraseMatchSpliterator implements Spliterator<Result> {
  ...
  int tryToUpdateSplitPos(int startPos, int splitPos) {
    int endPos =
      splitPos + mPattern.toString().length();
    if (endPos >= mInput.length()) return -1;
    CharSequence substr =
      mInput.subSequence(startPos, endPos);
    Matcher pm = mPattern.matcher(substr);
    if (pm.find()) splitPos = startPos
      + pm.start() + pm.group().length();
    return splitPos;
  }
```

Set endPos to the very end of the input that could match the pattern.

Analysis of the PhraseMatchSpliterator trySplit() Method

"... Therefore, since brevity is the soul of" "wit, And tediousness the limbs and outward ..."


```
class PhraseMatchSpliterator implements Spliterator<Result> {
  ...
  int tryToUpdateSplitPos(int startPos, int splitPos) {
    int endPos =
      splitPos + mPattern.toString().length();
    if (endPos >= mInput.length()) return -1;
    CharSequence substr =
      mInput.subSequence(startPos, endPos);
    Matcher pm = mPattern.matcher(substr);
    if (pm.find()) splitPos = startPos
      + pm.start() + pm.group().length();
    return splitPos;
  }
```

Ensure phrase isn't longer than the input string!

Analysis of the PhraseMatchSpliterator trySplit() Method

"... Therefore, since brevity is the soul of" "wit, And tediousness the limbs and outward ..."


```
class PhraseMatchSpliterator implements Spliterator<Result> {
  ...
  int tryToUpdateSplitPos(int startPos, int splitPos) {
    int endPos =
      splitPos + mPattern.toString().length();
    if (endPos >= mInput.length()) return -1;
    CharSequence substr =
      mInput.subSequence(startPos, endPos);
    Matcher pm = mPattern.matcher(substr);
    if (pm.find()) splitPos = startPos
      + pm.start() + pm.group().length();
    return splitPos;
  }
```

"brevity is the soul of wit"

Check to see if the phrase matches within the substring that span the initial splitPos.

Analysis of the PhraseMatchSpliterator trySplit() Method

"... Therefore, since brevity is the soul of" "wit, And tediousness the limbs and outward ..."


```
class PhraseMatchSpliterator implements Spliterator<Result> {
  ...
  int tryToUpdateSplitPos(int startPos, int splitPos) {
    int endPos =
      splitPos + mPattern.toString().length();
    if (endPos >= mInput.length()) return -1;
    CharSequence substr =
      mInput.subSequence(startPos, endPos);
    Matcher pm = mPattern.matcher(substr);
    if (pm.find()) splitPos = startPos
      + pm.start() + pm.group().length();
    return splitPos;
  }
```

If there's a match, update the splitPos to handle phrase spanning newlines.

Analysis of the PhraseMatchSpliterator trySplit() Method

"... Therefore, since brevity is the soul of" "wit, And tediousness the limbs and outward ..."


```
class PhraseMatchSpliterator implements Spliterator<Result> {
  ...
  int tryToUpdateSplitPos(int startPos, int splitPos) {
    int endPos =
      splitPos + mPattern.toString().length();
    if (endPos >= mInput.length()) return -1;
    CharSequence substr =
      mInput.subSequence(startPos, endPos);
    Matcher pm = mPattern.matcher(substr);
    if (pm.find()) splitPos = startPos
      + pm.start() + pm.group().length();
    return splitPos;
  }
```

Return the final splitPos.

Analysis of the PhraseMatchSpliterator trySplit() Method


```
class PhraseMatchSpliterator implements Spliterator<Result> {
  ...
  Spliterator<Result> trySplit() {
    if (mInput.length() <= mMinSplitSize) return null;
    int startPos,
        splitPos = mInput.length() / 2;

    if ((startPos = computeStartPos(splitPos)) < 0) return null;

    if ((splitPos = tryToUpdateSplitPos(startPos, splitPos)) < 0)
      return null;

    return splitInput(splitPos); ...
```

Create and return a new Spliterator.

Analysis of the PhraseMatchSpliterator trySplit() Method


```
class PhraseMatchSpliterator implements Spliterator<Result> {
  ...
  Spliterator<Result> splitInput(int splitPos) {
    CharSequence lhs =
      mInput.subSequence(0, splitPos);
    mInput = mInput.subSequence(splitPos,
                                mInput.length());
    mPhraseMatcher = mPattern.matcher(mInput);
    mOffset = splitPos;
    ...


    return new PhraseMatchSpliterator(lhs, ...); ...
```

Create and return a new Spliterator.

Analysis of the PhraseMatchSpliterator trySplit() Method


```
class PhraseMatchSpliterator implements Spliterator<Result> {
  ...
  Spliterator<Result> splitInput(int splitPos) {
    CharSequence lhs =
      mInput.subSequence(0, splitPos);
    mInput = mInput.subSequence(splitPos,
                                mInput.length());
    mPhraseMatcher = mPattern.matcher(mInput);
    mOffset = splitPos;
    ...
```

Create a sub-sequence for the left-hand Spliterator.

```
    return new PhraseMatchSpliterator(lhs, ...); ...
```

Analysis of the PhraseMatchSpliterator trySplit() Method


```
class PhraseMatchSpliterator implements Spliterator<Result> {
  ...
  Spliterator<Result> splitInput(int splitPos) {
    CharSequence lhs =
      mInput.subSequence(0, splitPos);
    mInput = mInput.subSequence(splitPos,
                                mInput.length());
    mPhraseMatcher = mPattern.matcher(mInput);
    mOffset = splitPos;
    ...

    return new PhraseMatchSpliterator(lhs, ...); ...
```

Update "this" to reflect changes to "right hand" portion of input.

Analysis of the PhraseMatchSpliterator trySplit() Method

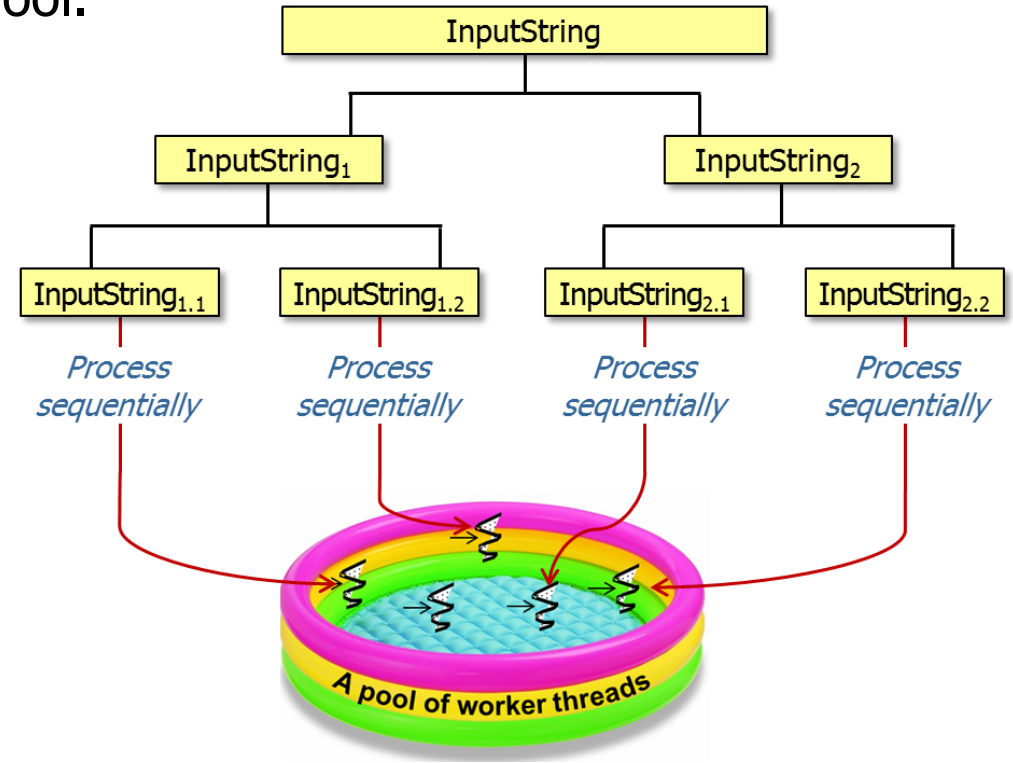

```
class PhraseMatchSpliterator implements Spliterator<Result> {
  ...
  Spliterator<Result> splitInput(int splitPos) {
    CharSequence lhs =
      mInput.subSequence(0, splitPos);
    mInput = mInput.subSequence(splitPos,
                                mInput.length());
    mPhraseMatcher = mPattern.matcher(mInput);
    mOffset = splitPos;
    ...

    return new PhraseMatchSpliterator(lhs, ...); ...
```

This Spliterator handles "left hand" portion of input, while "this" object handles "right hand" portion.

Analysis of the PhraseMatchSpliterator trySplit() Method

- Java streams framework processes all Spliterator chunks for each input string in parallel in the common ForkJoinPool.



This parallelism is in addition to parallelism of input string and phrase chunks!

The End

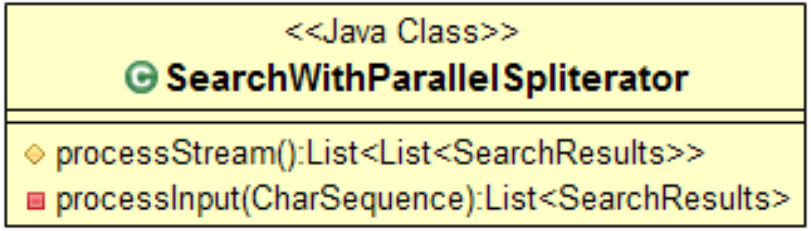
Java SearchWithParallelSpliterator Example

Evaluating Pros and Cons

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

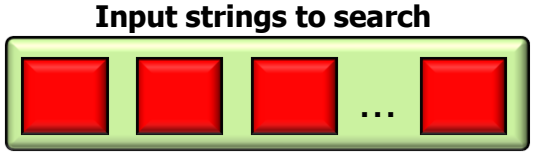
- Be aware of how a parallel Spliterator can improve parallel stream performance.
- Know the intent of—and fields in—the PhraseMatchSpliterator.
- Recognize the PhraseMatchSpliterator constructor and tryAdvance() method implementation.
- Understand the PhraseMatchSpliterator trySplit() method implementation.
- Understand the pros and cons of the SearchWithParallelSpliterator class.

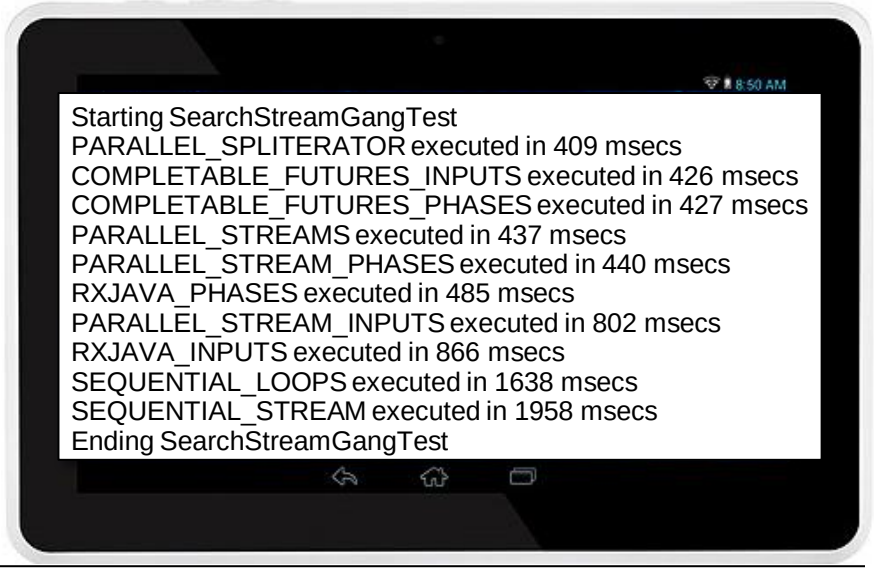


Pros of the SearchWithParallelSpliterator Class

Pros of the SearchWithParallelSpliterator Class

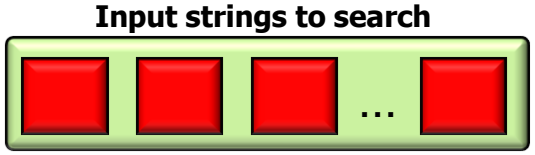
- This example shows how a parallel Spliterator can help transparently improve program performance.

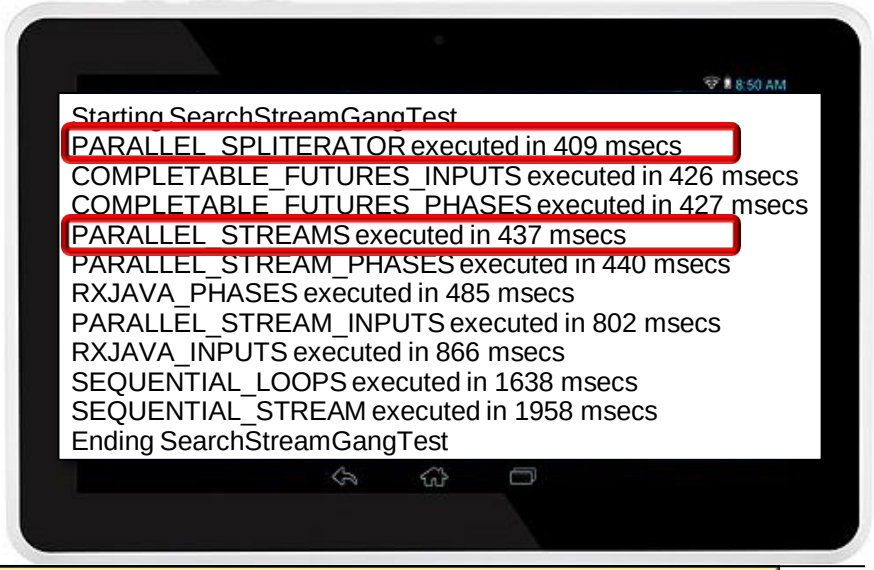




Pros of the SearchWithParallelSpliterator Class

- This example shows how a parallel Spliterator can help transparently improve program performance.

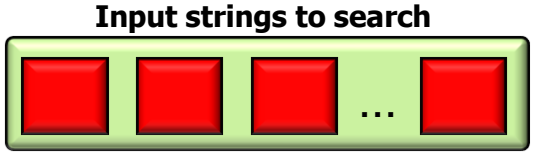


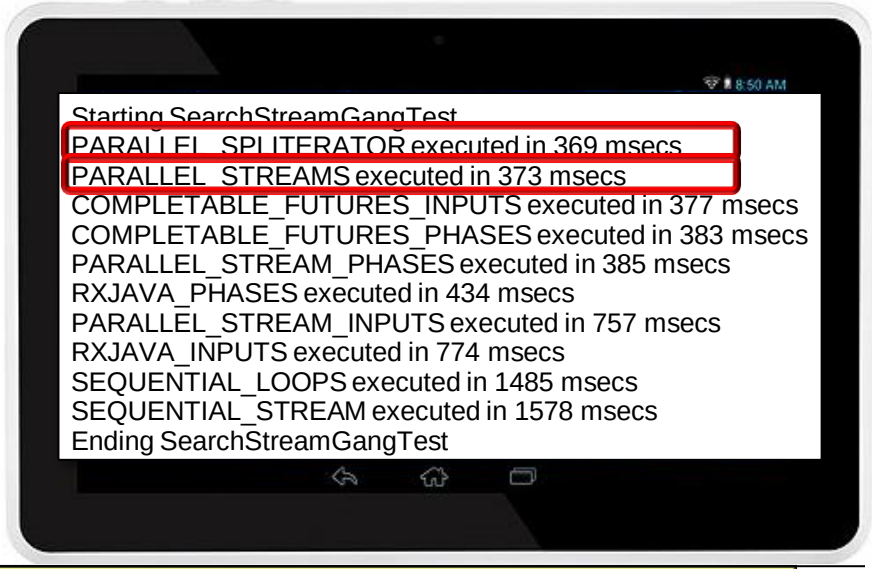


Tests conducted on a 2.7 GHz quad-core Lenovo P50 with 32 GB of RAM

Pros of the SearchWithParallelSpliterator Class

- This example shows how a parallel Spliterator can help transparently improve program performance.

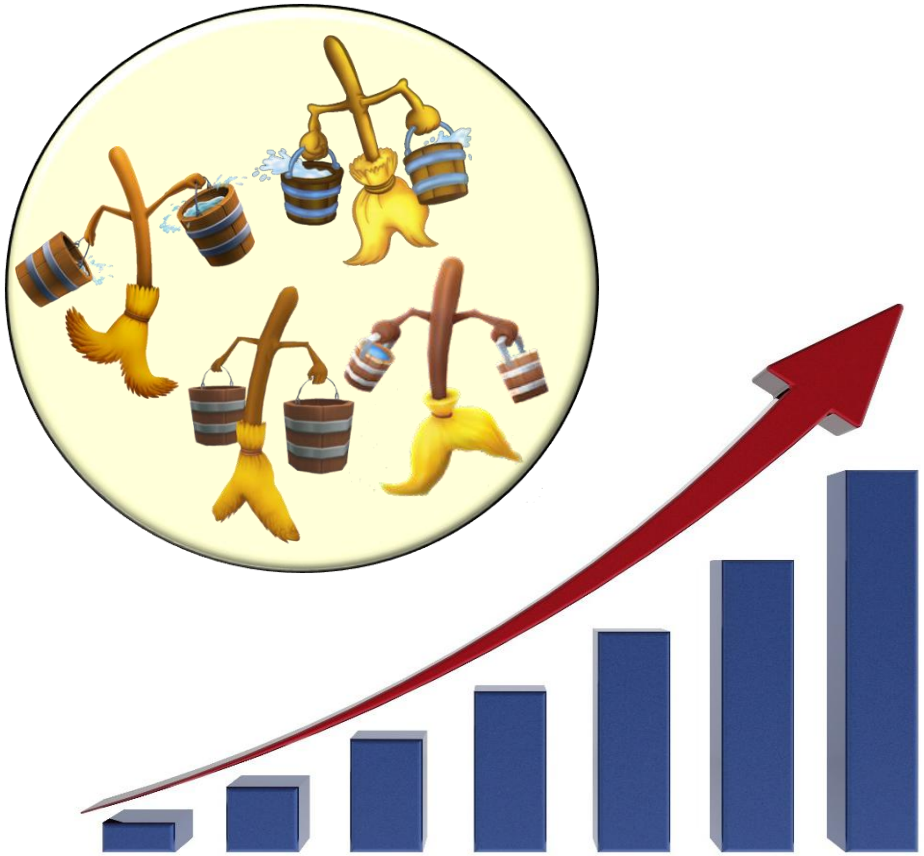




Tests conducted on a 2.9 GHz quad-core MacBook Pro with 16 GB of RAM

Pros of the SearchWithParallelSpliterator Class

- This example shows how a parallel Spliterator can help transparently improve program performance.
  - These speedups occur since the granularity of parallelism is finer and thus better able to leverage available cores.

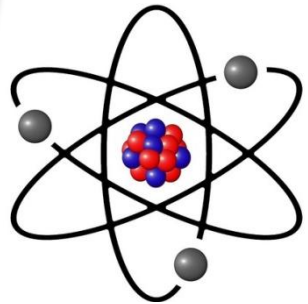


See docs.oracle.com/javase/tutorial/collections/streams/parallelism.html

Pros of the SearchWithParallelSpliterator Class

- This example also shows that the difference between using sequential vs. parallel Spliterator can be minuscule!

```
SearchResults searchForPhrase(String phrase, CharSequence input,
                              String title, boolean parallel) {
  return new SearchResults
    (..., ..., phrase, title, StreamSupport
     .stream(new PhraseMatchSpliterator(input,
                                        phrase),
             parallel)
     .collect(toList()));
}
```



Switching this boolean from "false" to "true" controls whether the Spliterator runs sequentially or in parallel.

Pros of the SearchWithParallelSpliterator Class

- This example also shows that the difference between using sequential vs. parallel Spliterator can be minuscule!

```
SearchResults searchForPhrase(String phrase, CharSequence input,
                              String title, boolean parallel) {
  return new SearchResults
    (..., ..., phrase, title, StreamSupport
     .stream(new PhraseMatchSpliterator(input,
                                        phrase),
             parallel)
     .collect(toList()));
}
```

Of course, it took nontrivial time/effort to create PhraseMatchSpliterator.

Cons of the SearchWithParallelSpliterator Class

Cons of the SearchWithParallelSpliterator Class

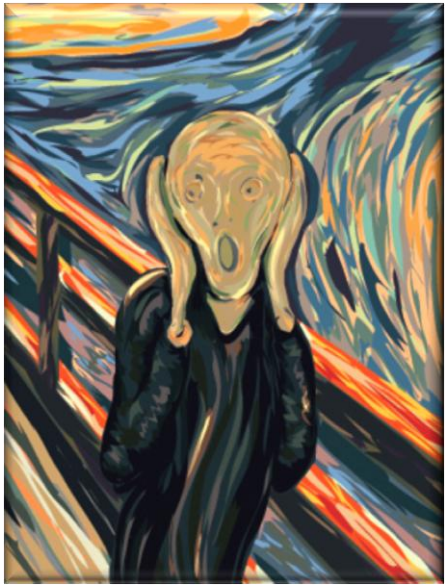
- The parallel-related portions of PhraseMatchSpliterator are *much* more complicated to program than the sequential-related portions . . .

```
class PhraseMatchSpliterator
      implements Spliterator<Result> {
  ...
  Spliterator<Result> trySplit() { ... }

  int computeStartPos(int splitPos) { ... }

  int tryToUpdateSplitPos(int startPos,
                          int splitPos)
    { ... }

  PhraseMatchSpliterator splitInput(int splitPos) { ... }
  ...
```



Cons of the SearchWithParallelSpliterator Class

- The parallel-related portions of PhraseMatchSpliterator are *much* more complicated to program than the sequential-related portions . . .

```
class PhraseMatchSpliterator
      implements Spliterator<Result> {
  ...
  Spliterator<Result> trySplit() { ... }

  int computeStartPos(int splitPos) { ... }

  int tryToUpdateSplitPos(int startPos,
                          int splitPos)
    { ... }

  PhraseMatchSpliterator splitInput(int splitPos) { ... }
  ...
```

Must split carefully..

JUnit tests are extremely useful.

Cons of the SearchWithParallelSpliterator Class

- The parallel-related portions of PhraseMatchSpliterator are *much* more complicated to program than the sequential-related portions . . .

```
class PhraseMatchSpliterator
      implements Spliterator<Result> {
  ...
  Spliterator<Result> trySplit() { ... }

  int computeStartPos(int splitPos) { ...

  int tryToUpdateSplitPos(int startPos,
                          int splitPos)
    { ... }

  PhraseMatchSpliterator splitInput(int splitPos) { ... }
  ...
```

Writing the parallel Spliterator took longer than writing the rest of the program!

Java SearchWithParallelSpliterator Example:
Evaluating Pros and Cons

The End