Overview of Java 8 Parallel Streams (Part 1)

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

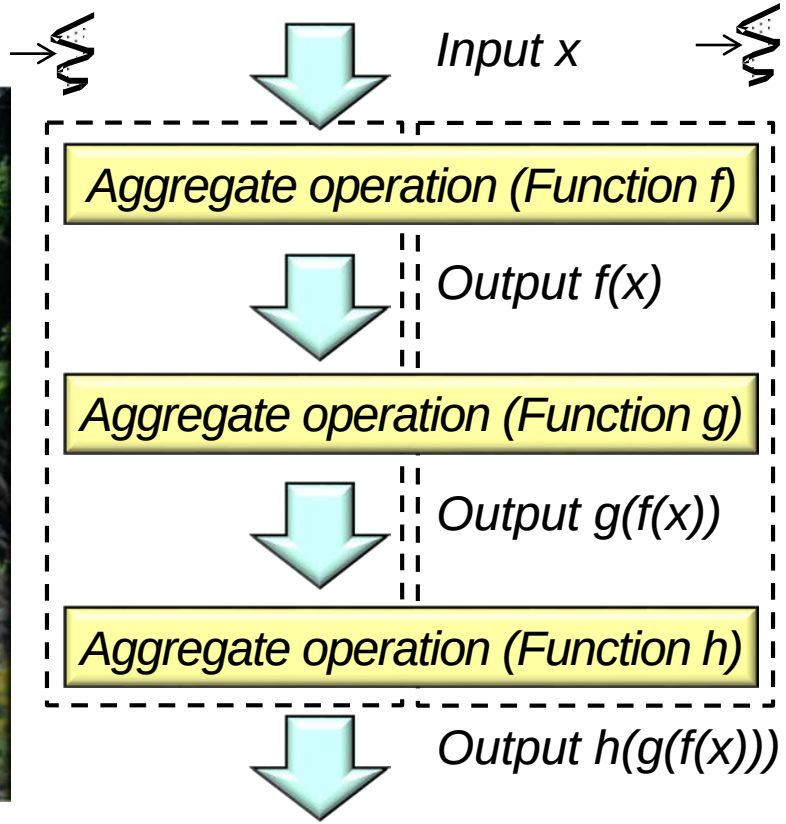
Professor of Computer Science

Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

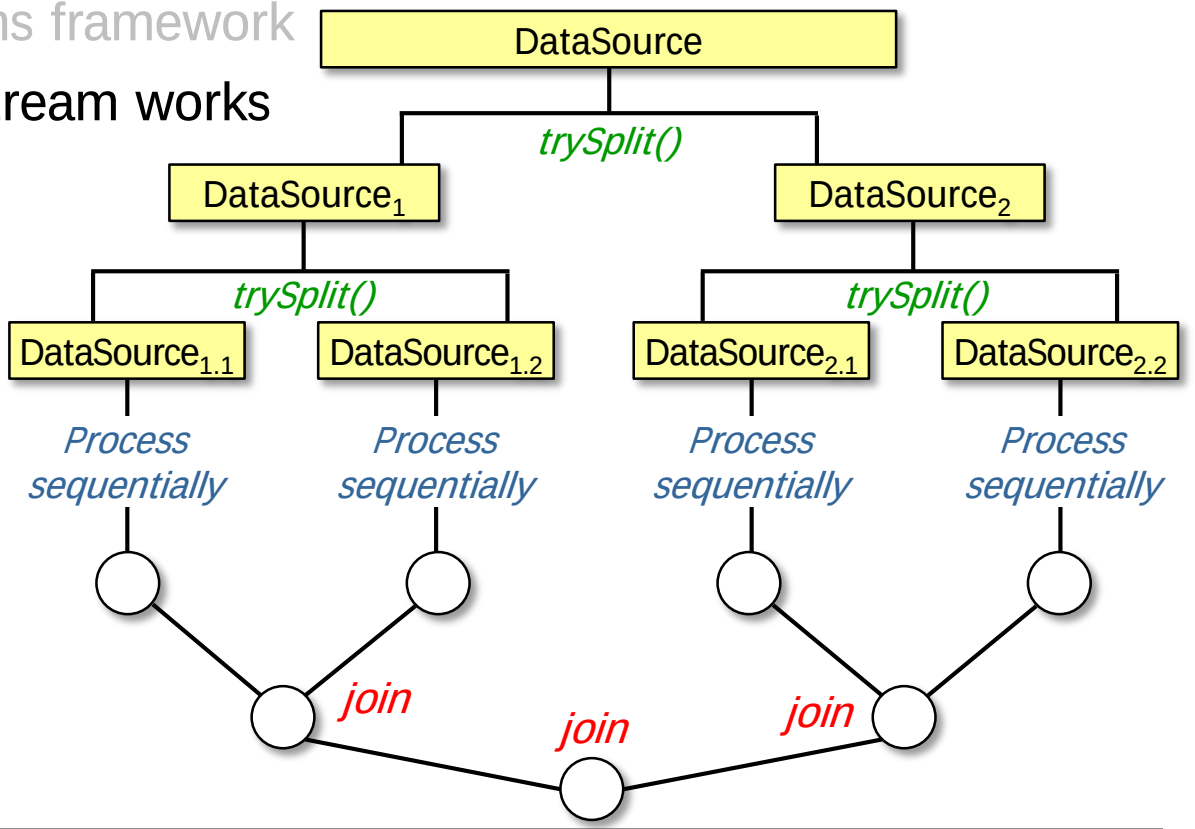
Learning Objectives in this Part of the Lesson

- Know how aggregate operations & functional programming features are applied in the parallel streams framework



Learning Objectives in this Part of the Lesson

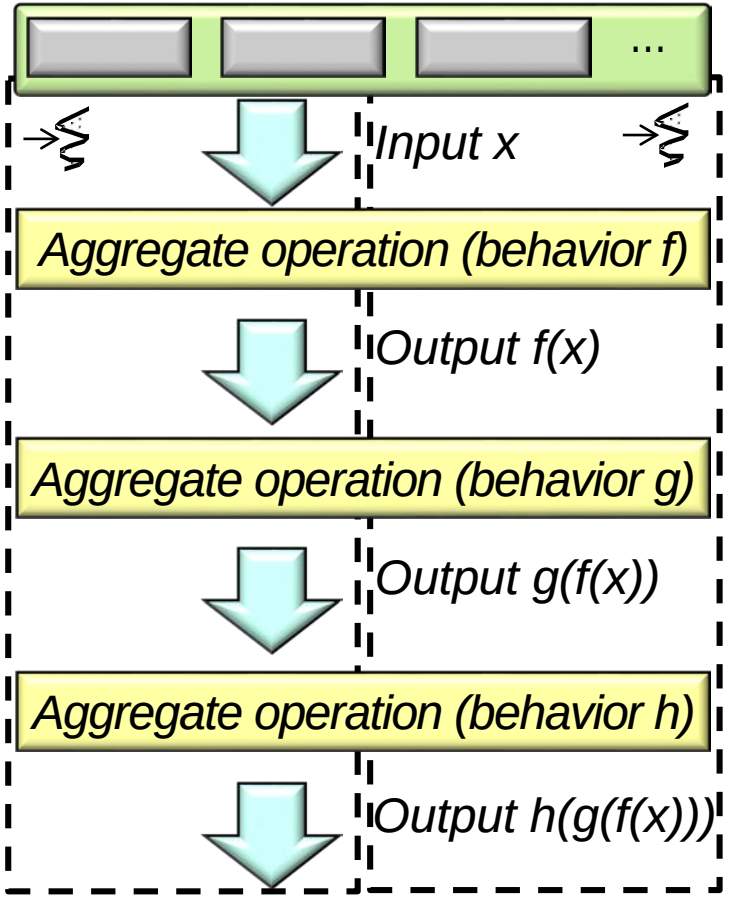
- Know how aggregate operations & functional programming features are applied in the parallel streams framework
- Understand how a parallel stream works



Overview of Java 8 Parallel Streams

Overview of Java 8 Parallel Streams

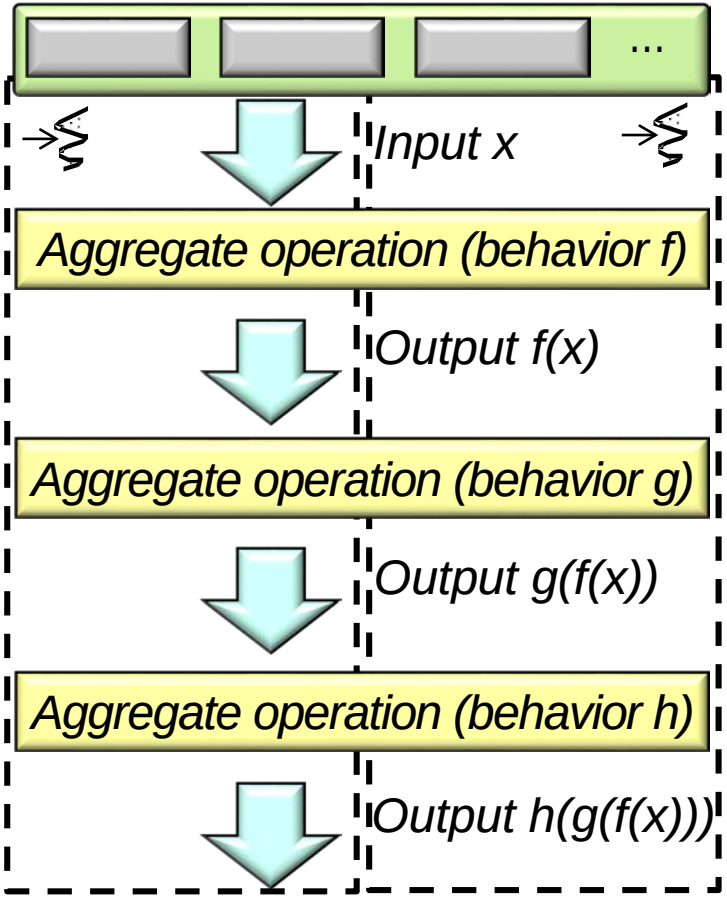
- A Java 8 parallel stream splits its elements into multiple chunks & uses a thread pool to process these chunks independently



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

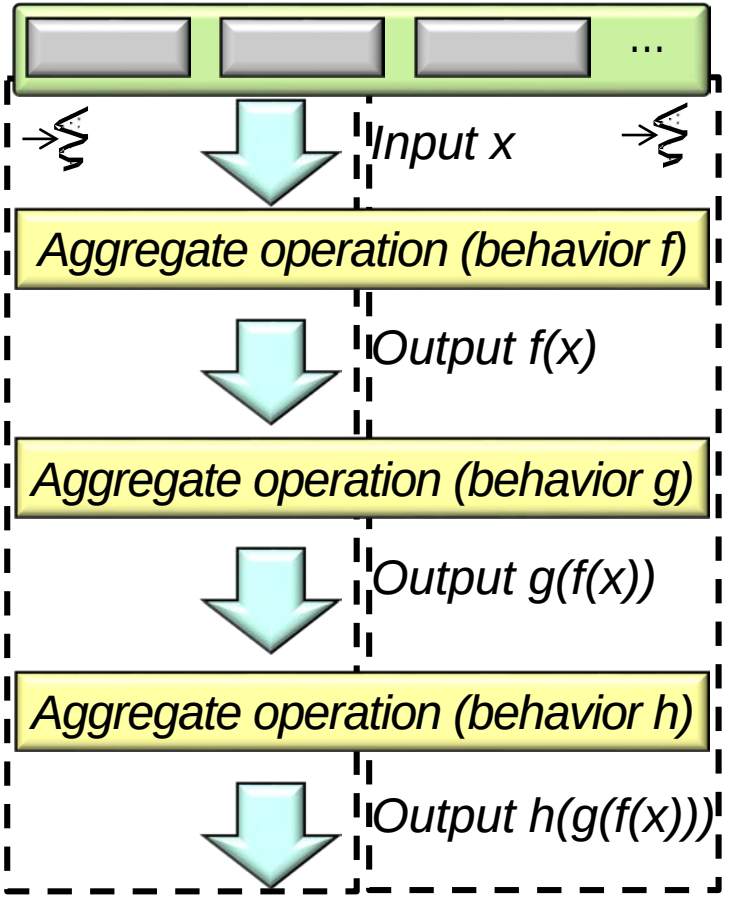
Overview of Java 8 Parallel Streams

- A Java 8 parallel stream splits its elements into multiple chunks & uses a thread pool to process these chunks independently
  - This splitting & thread pool are often invisible to programmers



Overview of Java 8 Parallel Streams

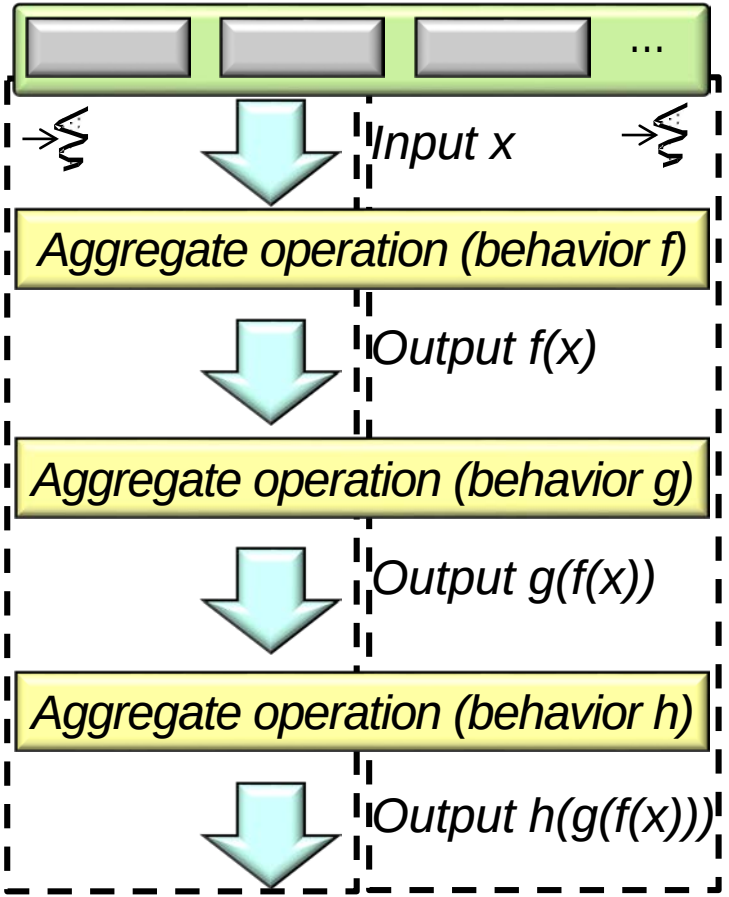
- A Java 8 parallel stream splits its elements into multiple chunks & uses a thread pool to process these chunks independently
  - This splitting & thread pool are often invisible to programmers
  - The *order* in which chunks are processed is likely non-deterministic



i.e., programmers often have little/no control over how chunks are processed

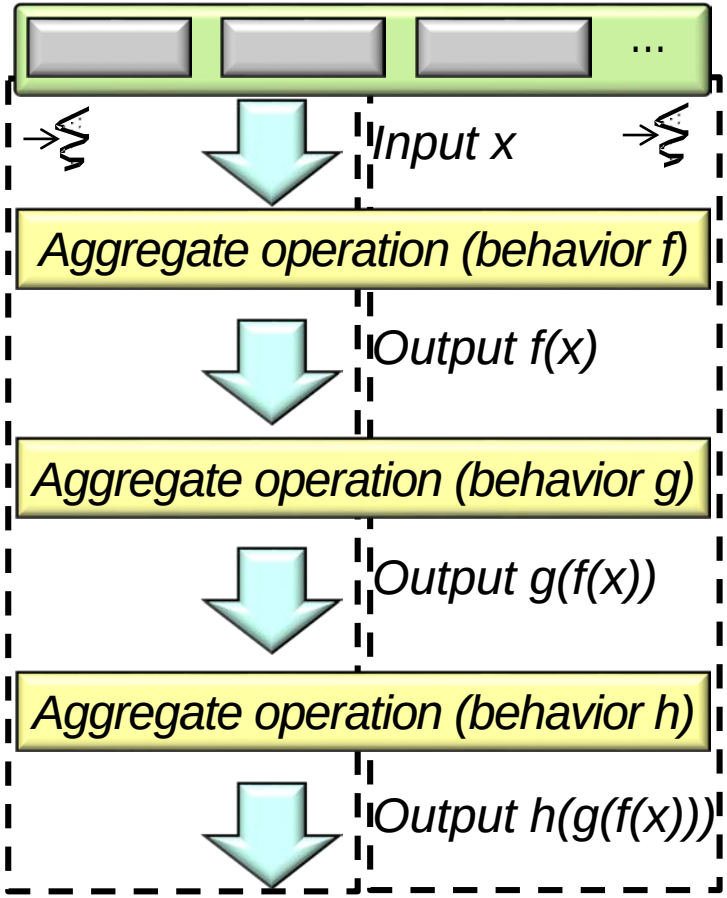
Overview of Java 8 Parallel Streams

- A Java 8 parallel stream splits its elements into multiple chunks & uses a thread pool to process these chunks independently
  - This splitting & thread pool are often invisible to programmers
  - The *order* in which chunks are processed is likely non-deterministic
    - This non-determinism is usually a good thing!



Overview of Java 8 Parallel Streams

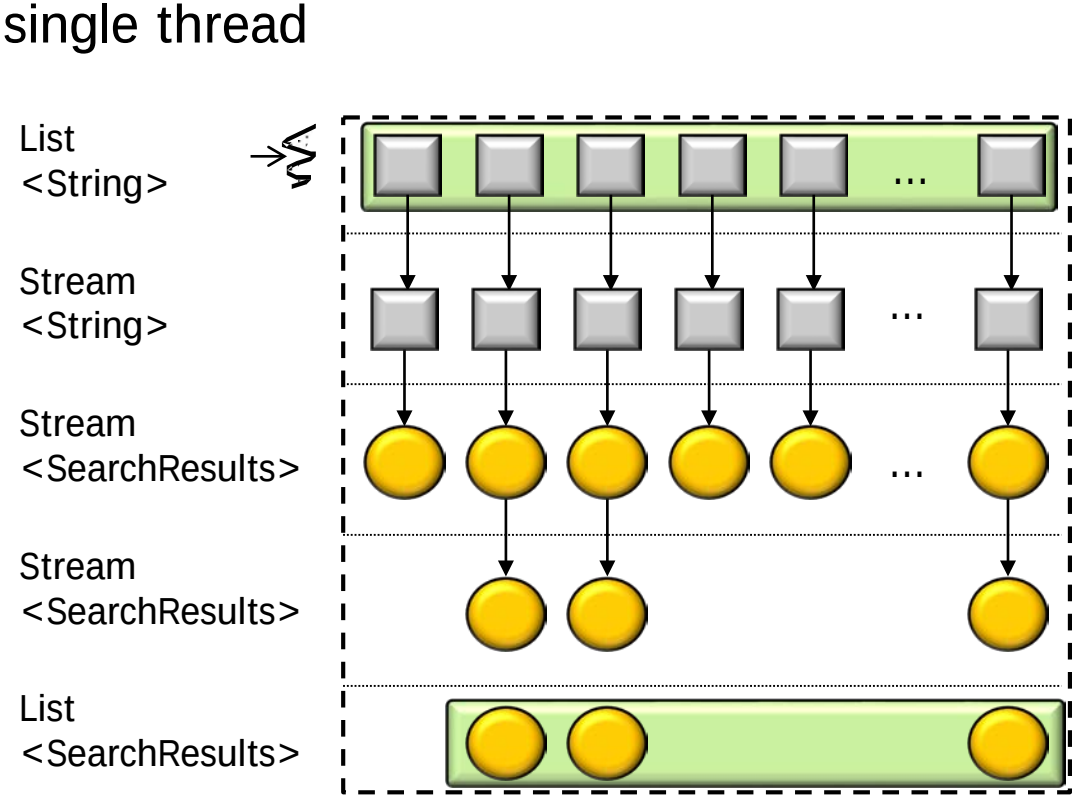
- A Java 8 parallel stream splits its elements into multiple chunks & uses a thread pool to process these chunks independently
  - This splitting & thread pool are often invisible to programmers
  - The *order* in which chunks are processed is likely non-deterministic
  - The *results* of the processing are likely deterministic

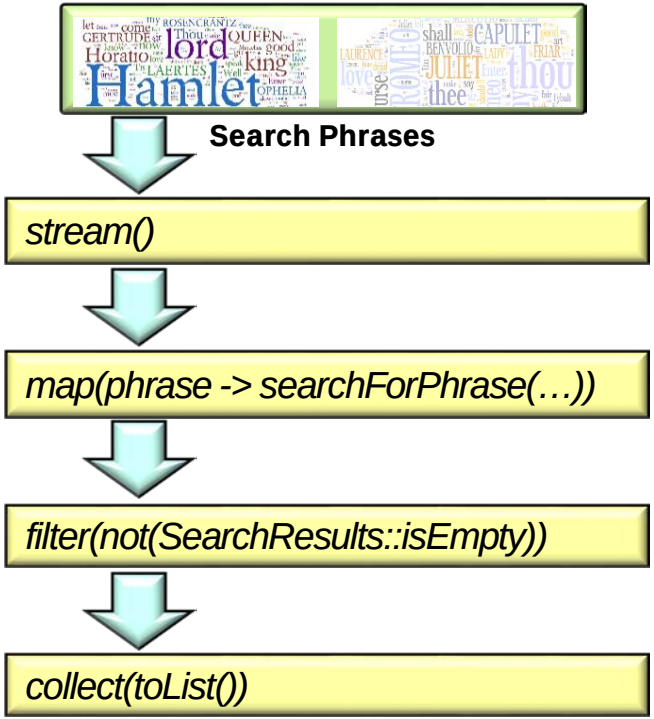


Programmers have more control over how the results are presented

Overview of Java 8 Parallel Streams

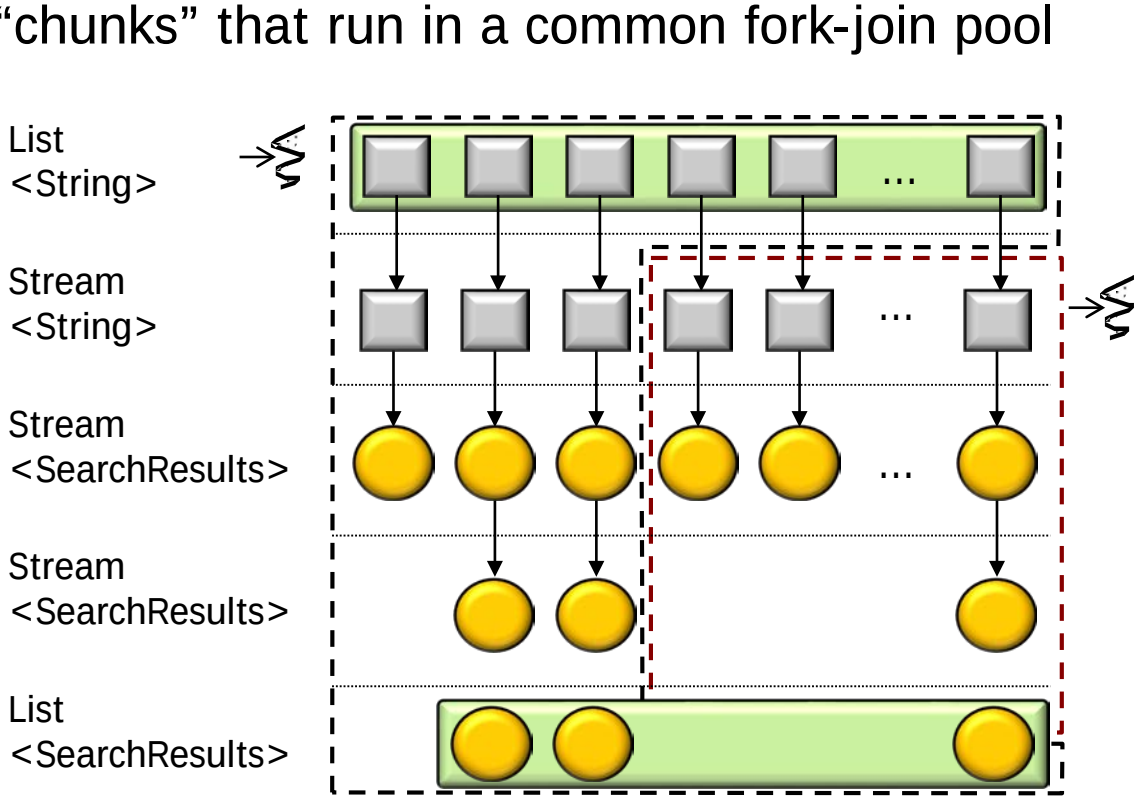
- When a stream executes sequentially all of its aggregate operations run in a single thread

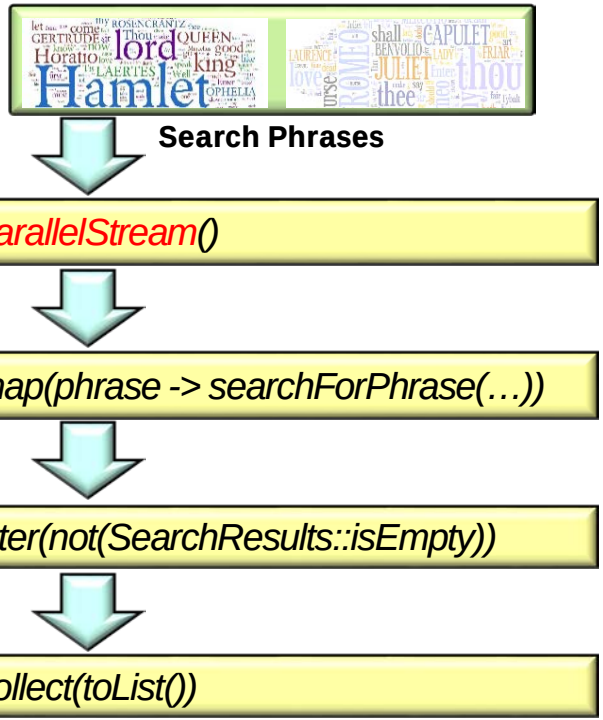




Overview of Java 8 Parallel Streams

- When a stream executes in parallel, it is partitioned into multiple substream "chunks" that run in a common fork-join pool

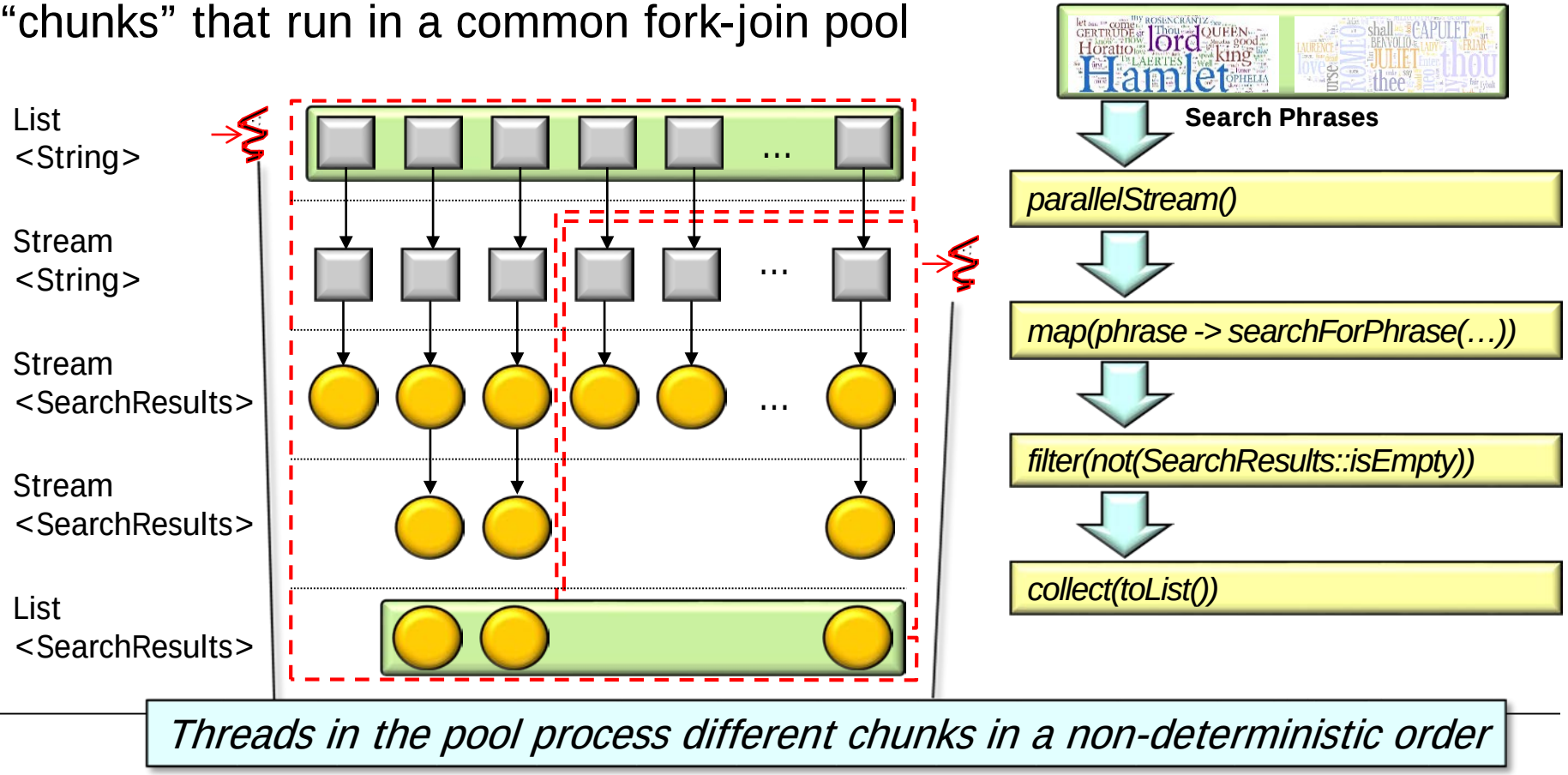




See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html

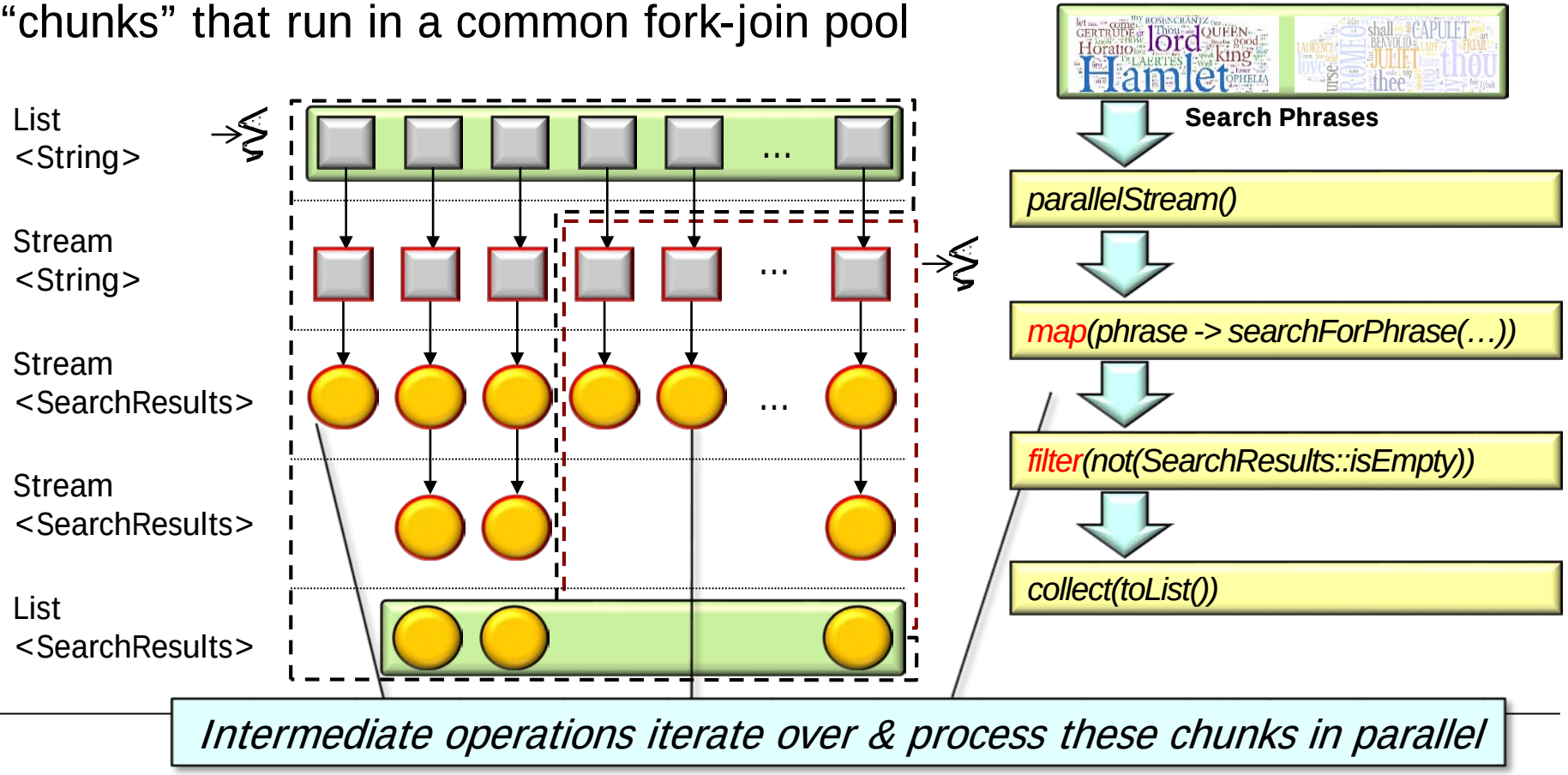
Overview of Java 8 Parallel Streams

- When a stream executes in parallel, it is partitioned into multiple substream "chunks" that run in a common fork-join pool



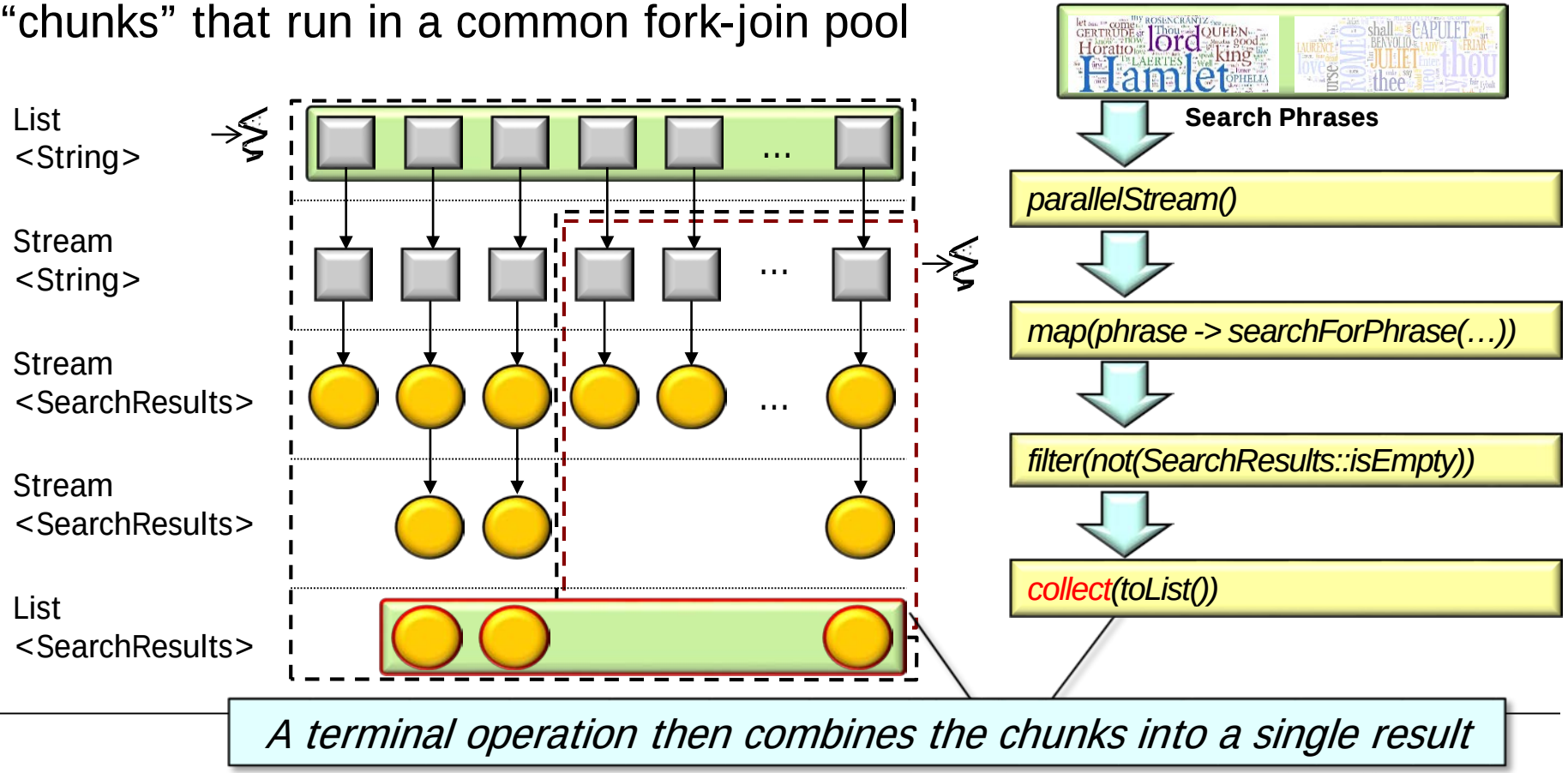
Overview of Java 8 Parallel Streams

- When a stream executes in parallel, it is partitioned into multiple substream "chunks" that run in a common fork-join pool



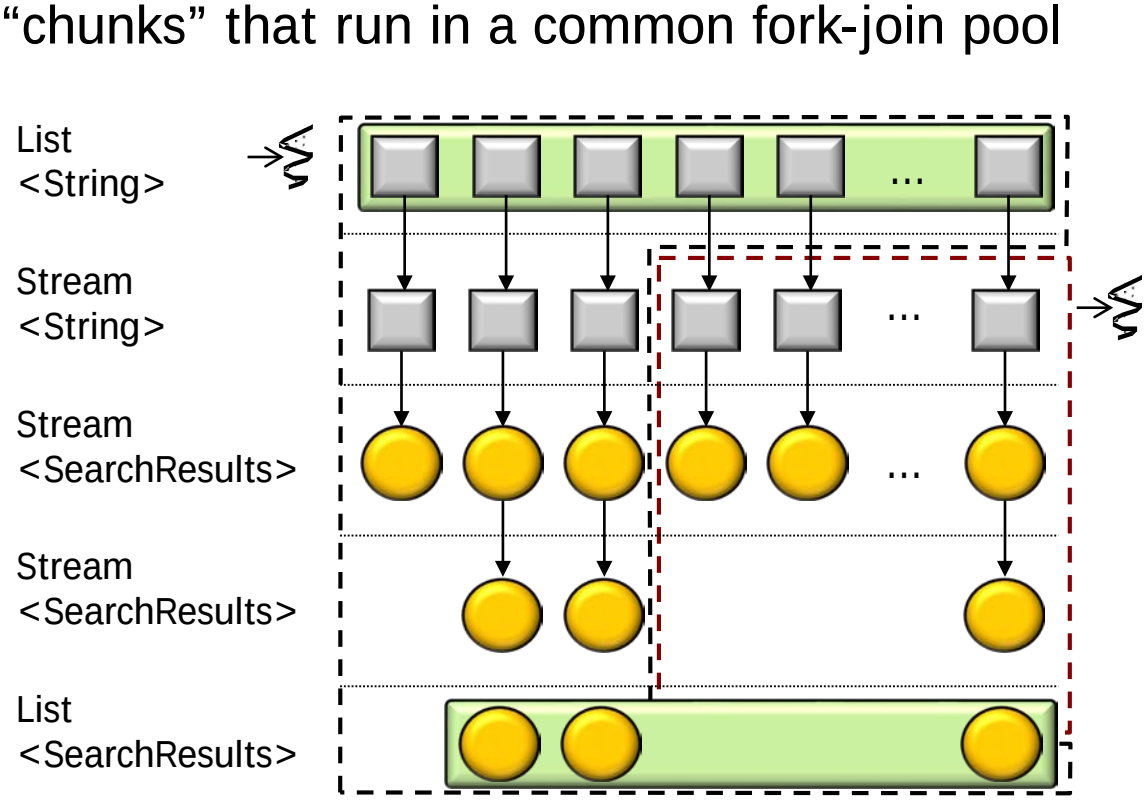
Overview of Java 8 Parallel Streams

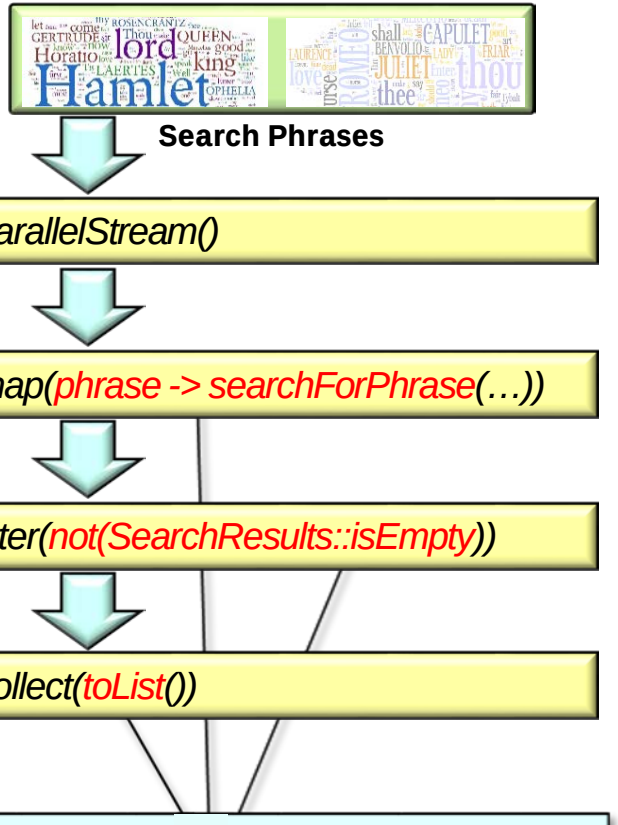
- When a stream executes in parallel, it is partitioned into multiple substream "chunks" that run in a common fork-join pool



Overview of Java 8 Parallel Streams

- When a stream executes in parallel, it is partitioned into multiple substream "chunks" that run in a common fork-join pool





(Stateless) Java 8 lambda expressions & method references are used to pass behaviors

Overview of Java 8 Parallel Streams

- The same aggregate operations can be used for sequential & parallel streams

Modifier and Type	Method and Description
boolean	**allMatch**(Predicate<? super T> predicate) Returns whether all elements of this stream match the provided predicate.
boolean	**anyMatch**(Predicate<? super T> predicate) Returns whether any elements of this stream match the provided predicate.
static <T> Stream.Builder<T>	**builder**() Returns a builder for a Stream.
<R,A> R	**collect**(Collector<? super T,A,R> collector) Performs a **mutable reduction** operation on the elements of this stream using a Collector.
<R> R	**collect**(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner) Performs a **mutable reduction** operation on the elements of this stream.
static <T> Stream<T>	**concat**(Stream<? extends T> a, Stream<? extends T> b) Creates a lazily concatenated stream whose elements are all the elements of the first stream followed by all the elements of the second stream.
long	**count**() Returns the count of elements in this stream.
Stream<T>	**distinct**() Returns a stream consisting of the distinct elements (according to **Object.equals(Object)**) of this stream.
static <T> Stream<T>	**empty**() Returns an empty sequential Stream.
Stream<T>	**filter**(Predicate<? super T> predicate) Returns a stream consisting of the elements of this stream that match the given predicate.
Optional<T>	**findAny**() Returns an **Optional** describing some element of the stream, or an empty Optional if the stream is empty.
Optional<T>	**findFirst**() Returns an **Optional** describing the first element of this stream, or an empty Optional if the stream is empty.
<R> Stream<R>	**flatMap**(Function<? super T,? extends Stream<? extends R>> mapper) Returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.

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

Overview of Java 8 Parallel Streams

- The same aggregate operations can be used for sequential & parallel streams

e.g., SearchStreamGang uses the same aggregate operations for both SearchWithSequentialStreams & SearchWithParallelStreams implementations

Search Phrases

stream() vs. parallelStream()

map(phrase -> searchForPhrase(…))

filter(not(SearchResults::isEmpty))

collect(toList())

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

Overview of Java 8 Parallel Streams

- The same aggregate operations can be used for sequential & parallel streams
  - Java 8 streams can thus treat parallelism as an optimization & leverage all available cores!

Search Phrases

parallelStream()

map(phrase -> searchForPhrase(…))

filter(not(SearchResults::isEmpty))

collect(toList())

See qconlondon.com/london-2017/system/files/presentation-slides/concurrenttoparallel.pdf

Overview of Java 8 Parallel Streams

- The same aggregate operations can be used for sequential & parallel streams
  - Java 8 streams can thus treat parallelism as an optimization & leverage all available cores!
  - Naturally, behaviors run by these aggregate operations must be designed carefully to avoid accessing unsynchronized shared state..

Search Phrases

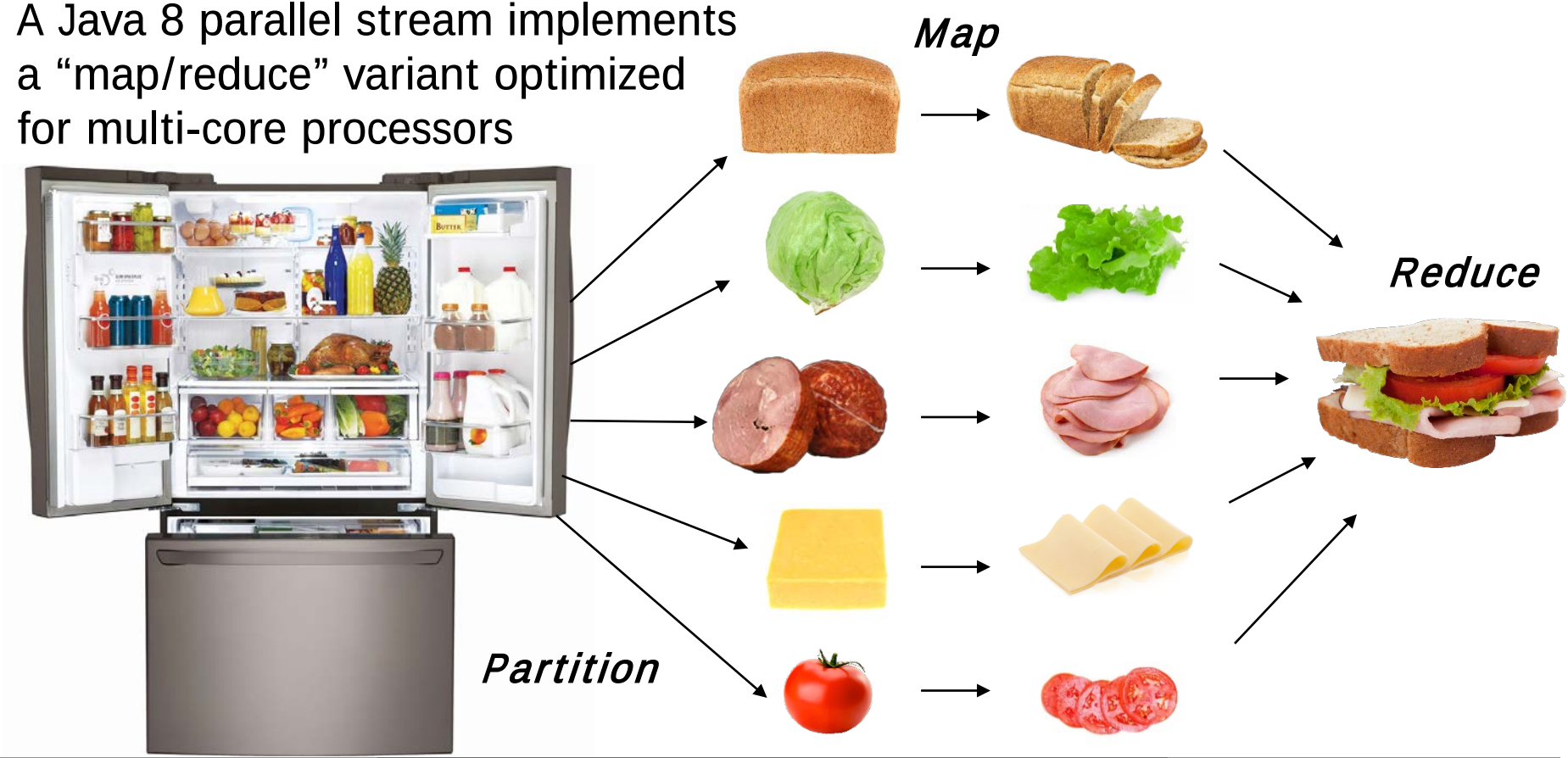
```
parallelStream()
map(phrase -> searchForPhrase(…))
filter(not(SearchResults::isEmpty))
collect(toList())
```

See henrikeichenhardt.blogspot.com/2013/06/why-shared-mutable-state-is-root-of-all.html

Overview of How a Parallel Stream Works

Overview of How a Parallel Stream Works

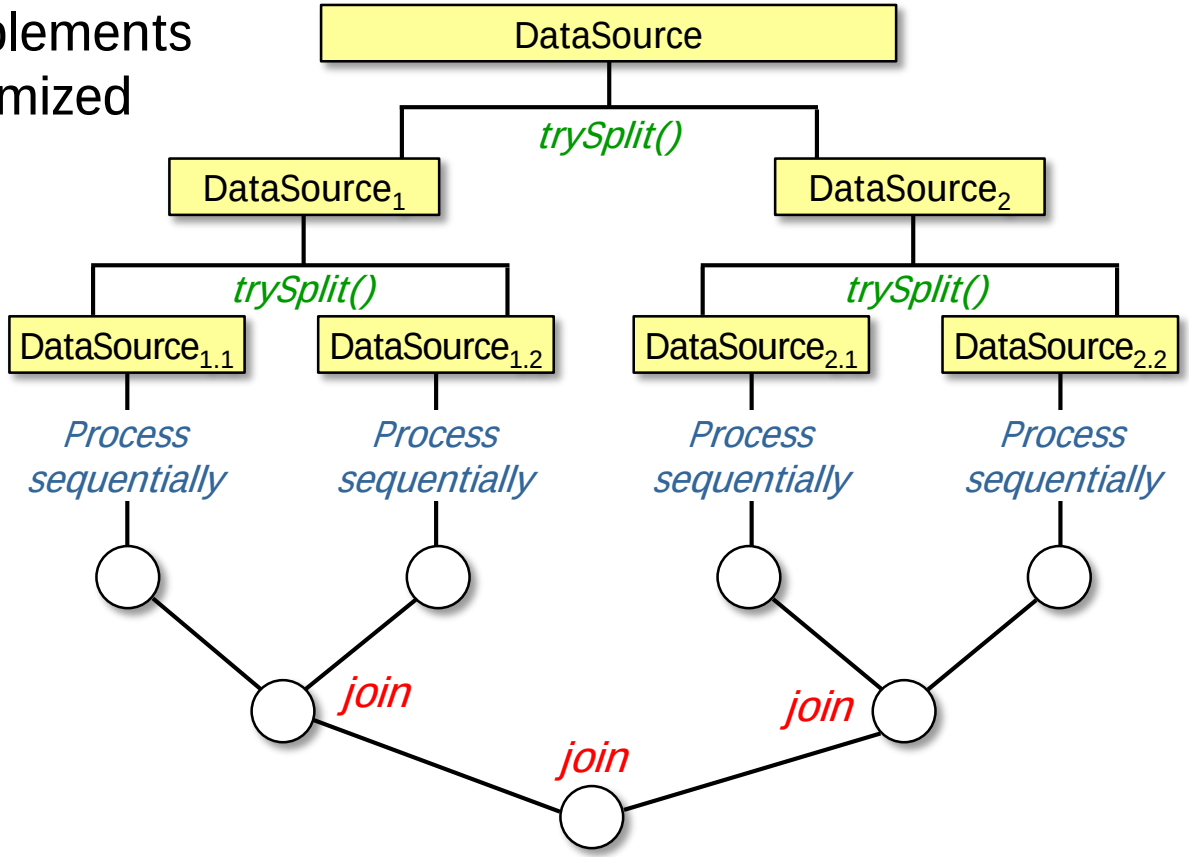
- A Java 8 parallel stream implements a “map/reduce” variant optimized for multi-core processors



See en.wikipedia.org/wiki/MapReduce

Overview of How a Parallel Stream Works

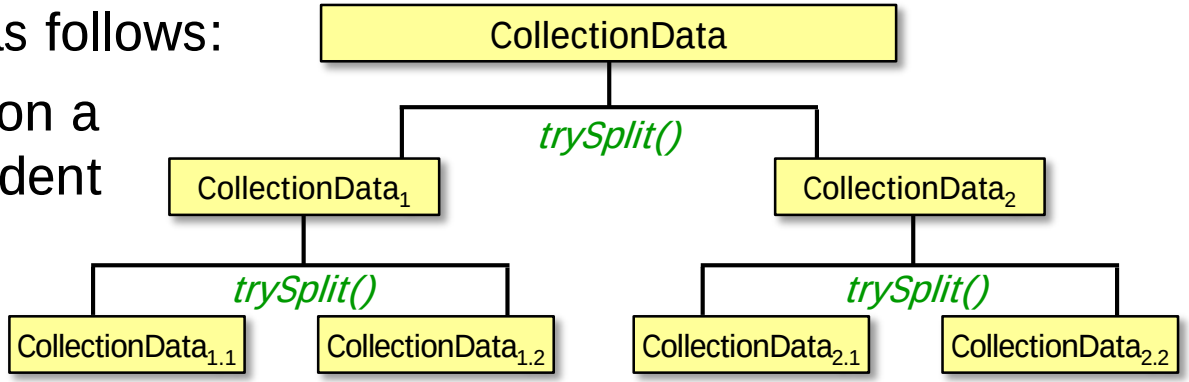
- A Java 8 parallel stream implements a "map/reduce" variant optimized for multi-core processors
  - It's actually more like the "split-apply-combine" data analysis strategy



See www.jstatsoft.org/article/view/v040i01

Overview of How a Parallel Stream Works

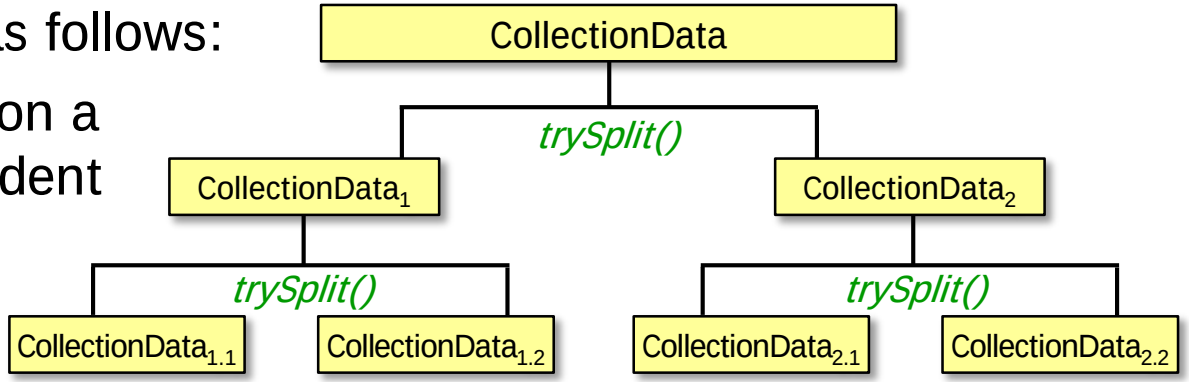
- Split-apply-combine works as follows:
  1. Split – Recursively partition a data source into independent "chunks"



See en.wikipedia.org/wiki/Divide_and_conquer_algorithm

Overview of How a Parallel Stream Works

- Split-apply-combine works as follows:
  1. Split – Recursively partition a data source into independent "chunks"
     - Spliterators are defined to partition collections in Java 8

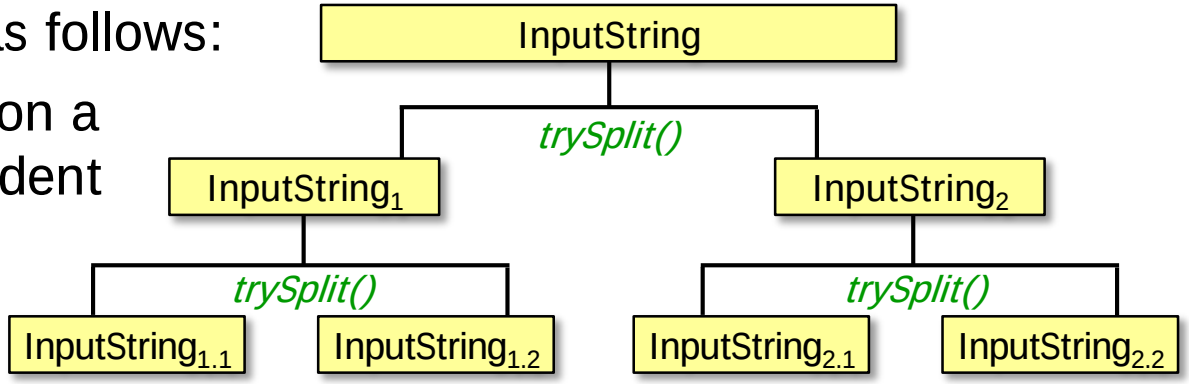


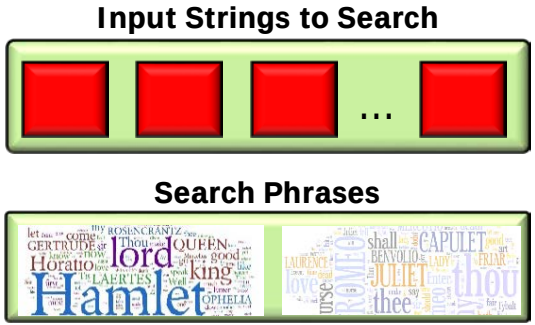

```
public interface Spliterator<T> {

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

   Spliterator<T> trySplit() ;

   long estimateSize();

   int characteristics();
}
```

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

Overview of How a Parallel Stream Works

- Split-apply-combine works as follows:
  1. Split – Recursively partition a data source into independent "chunks"
     - Spliterators are defined to partition collections in Java 8
     - You can also define custom spliterators

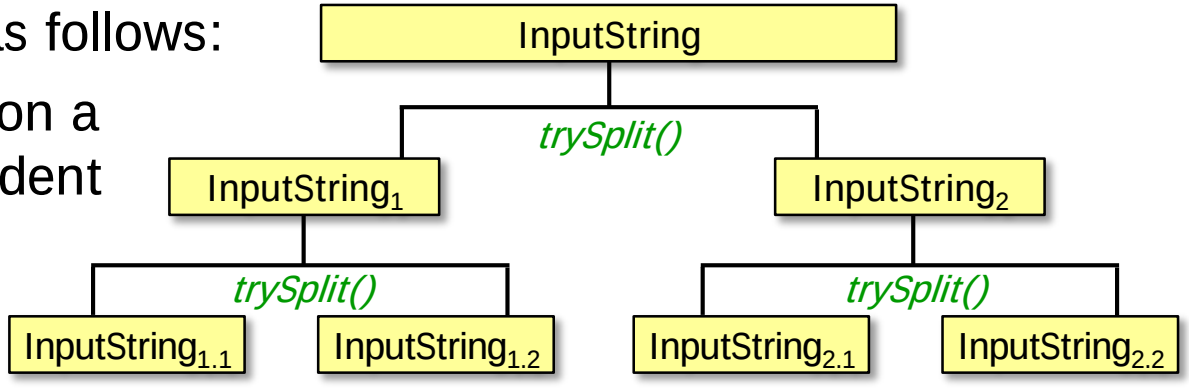




See github.com/douglascraigschmidt/LiveLessons/tree/master/SearchStreamSpliterator

Overview of How a Parallel Stream Works

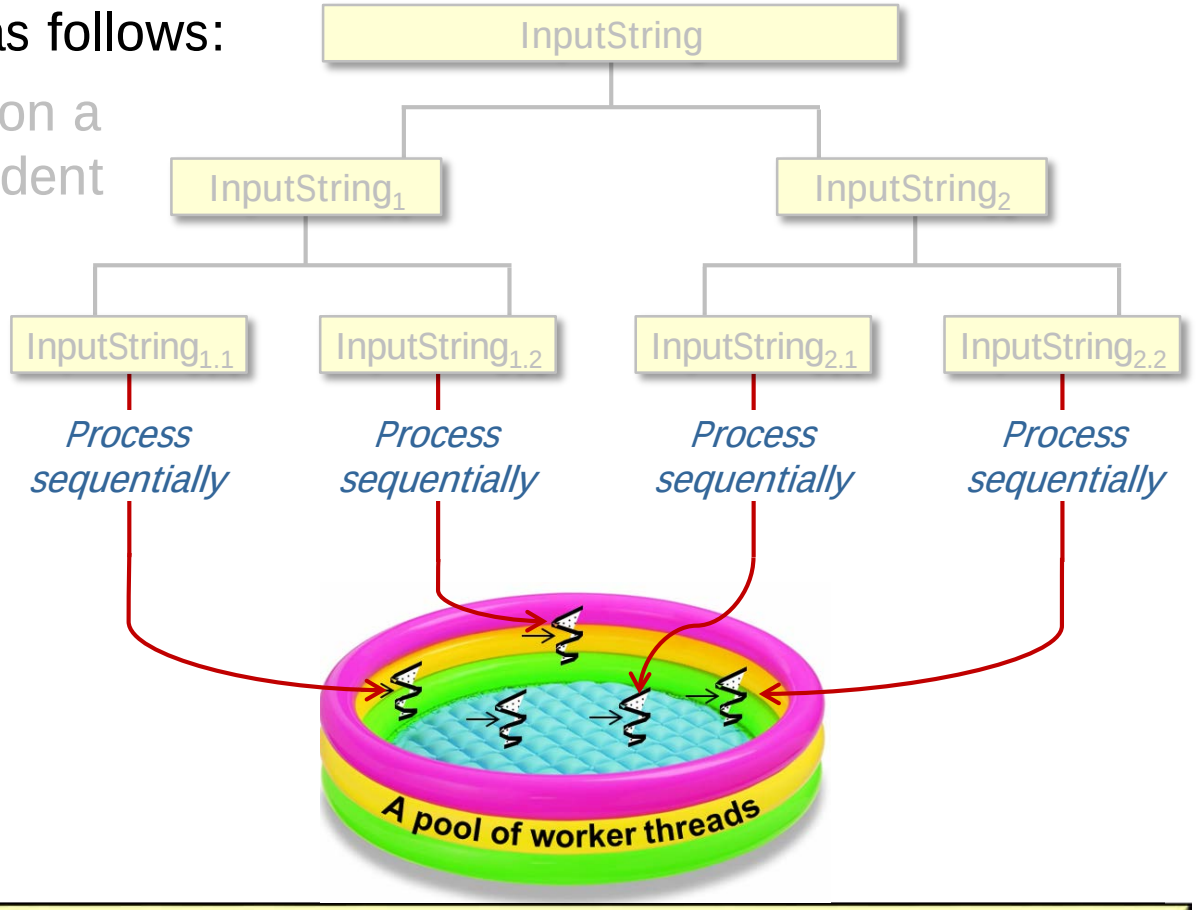
- Split-apply-combine works as follows:
  1. Split – Recursively partition a data source into independent "chunks"
     - Spliterators are defined to partition collections in Java 8
     - You can also define custom spliterators
     - Parallel streams perform better on data sources that can be split efficiently & evenly



See www.airpair.com/java/posts/parallel-processing-of-io-based-data-with-java-streams

Overview of How a Parallel Stream Works

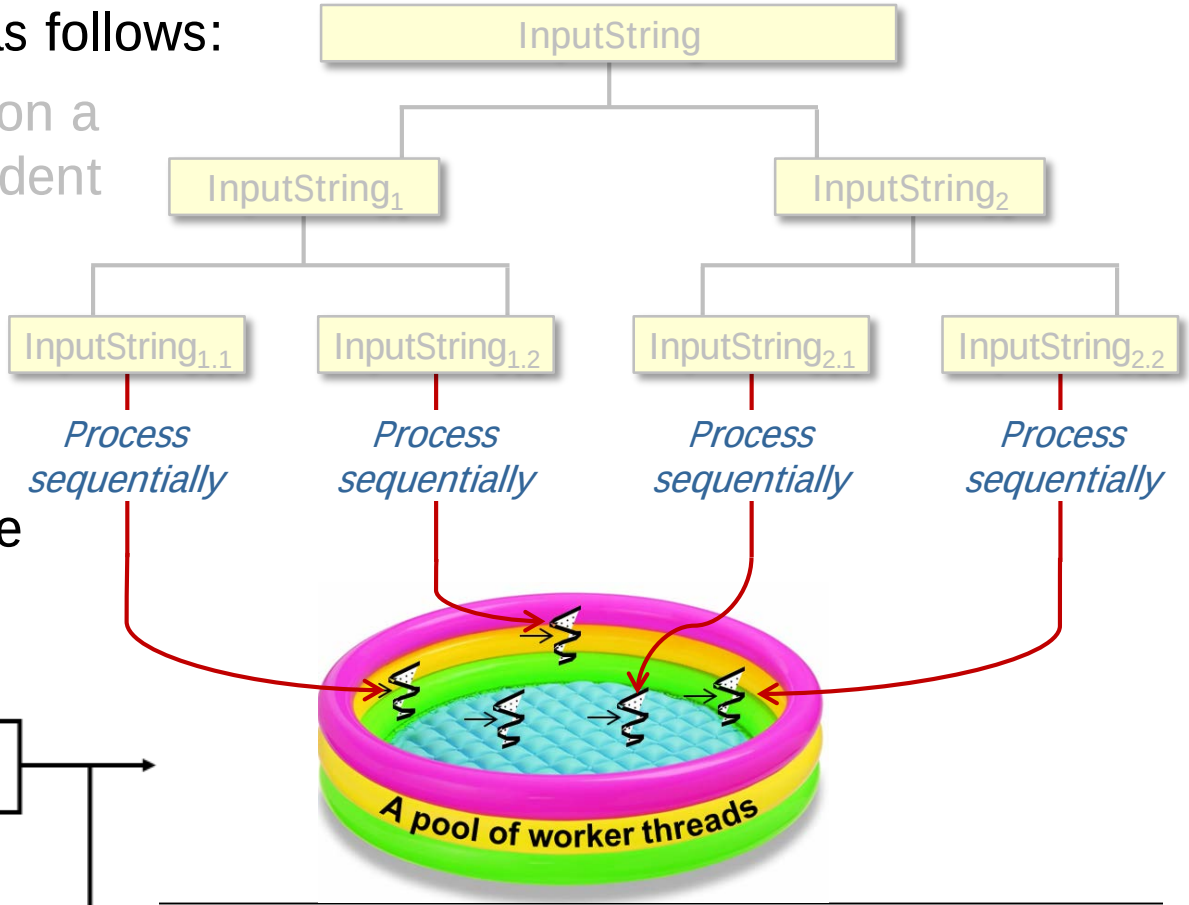
- Split-apply-combine works as follows:
  1. Split – Recursively partition a data source into independent "chunks"
  2. Apply – Process chunks independently in a pool of threads

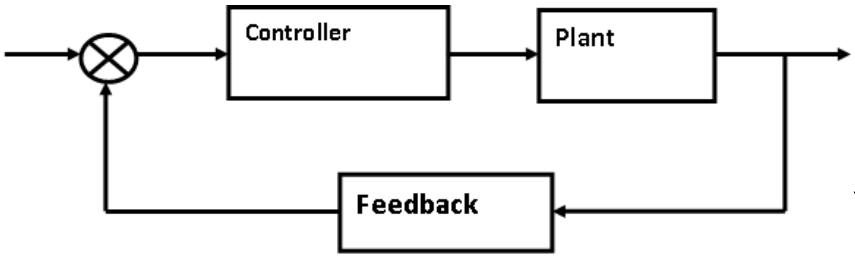


Splitting & applying run simultaneously (after certain limit met), not sequentially

Overview of How a Parallel Stream Works

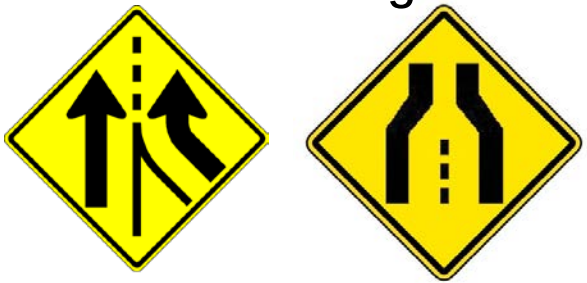
- Split-apply-combine works as follows:
  1. Split – Recursively partition a data source into independent "chunks"
  2. Apply – Process chunks independently in a pool of threads
     - Programmers have some control over how many threads are in the pool

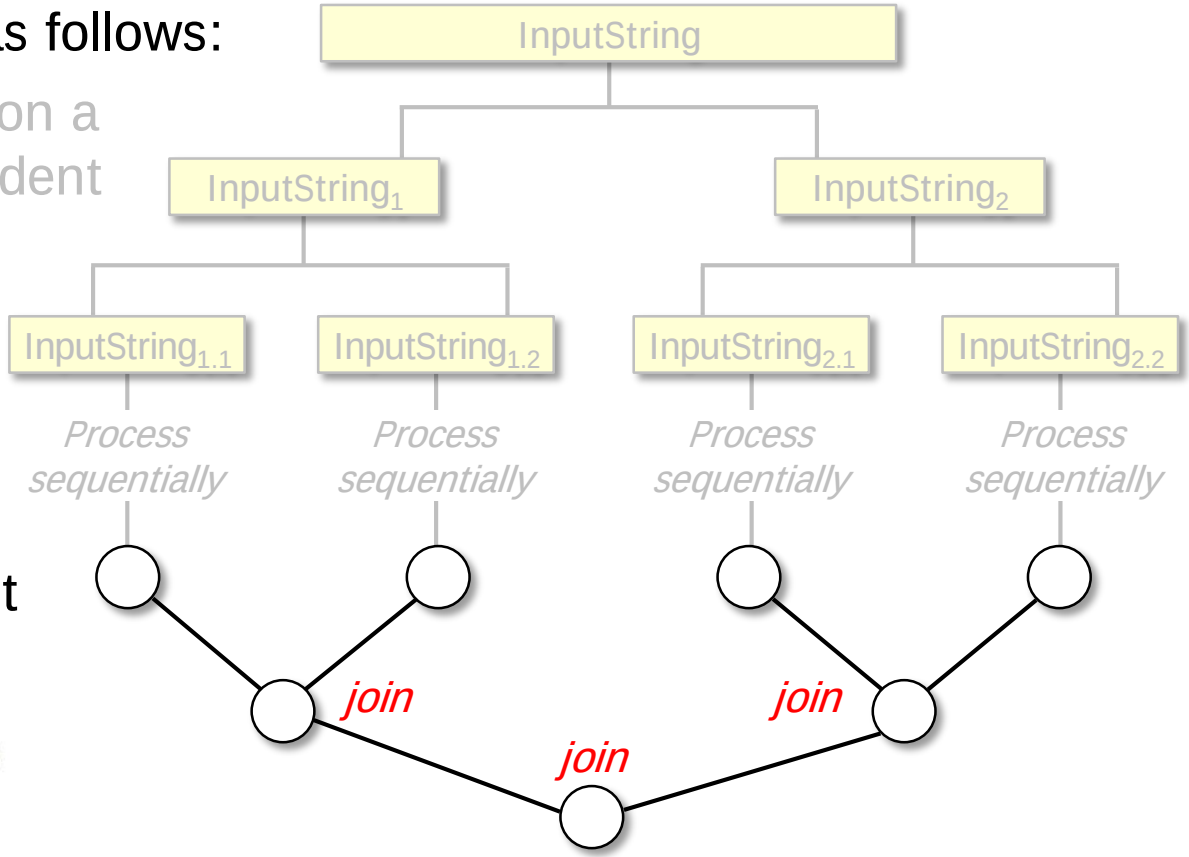




Overview of How a Parallel Stream Works

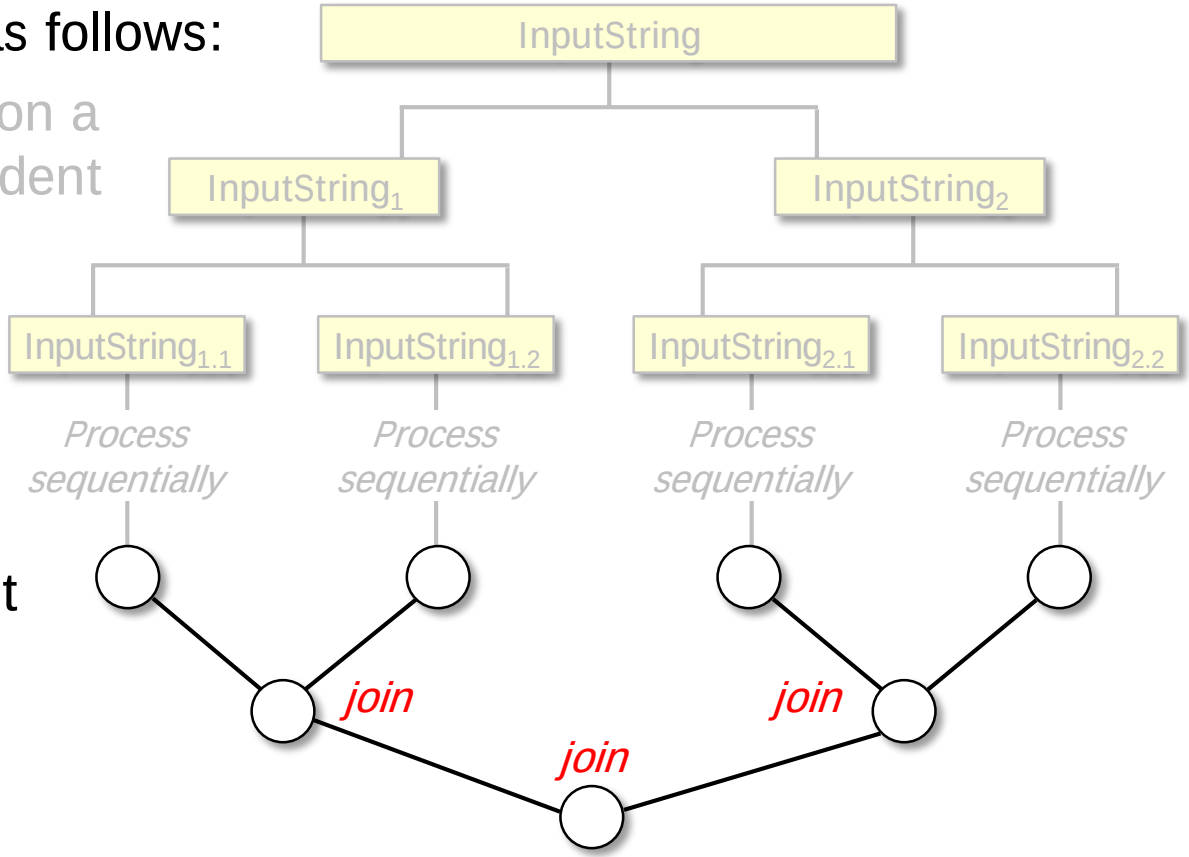
- Split-apply-combine works as follows:
  1. Split – Recursively partition a data source into independent "chunks"
  2. Apply – Process chunks independently in a pool of threads
  3. Combine – Join partial results into a single result





Overview of How a Parallel Stream Works

- Split-apply-combine works as follows:
  1. Split – Recursively partition a data source into independent "chunks"
  2. Apply – Process chunks independently in a pool of threads
  3. Combine – Join partial results into a single result
     - Performed by terminal operations like collect() & reduce()



See www.codejava.net/java-core/collections/java-8-stream-terminal-operations-examples

End of Overview of Java 8 Parallel Streams (Part 1)