Overview of Java 8 Parallel Streams (Part 1)

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

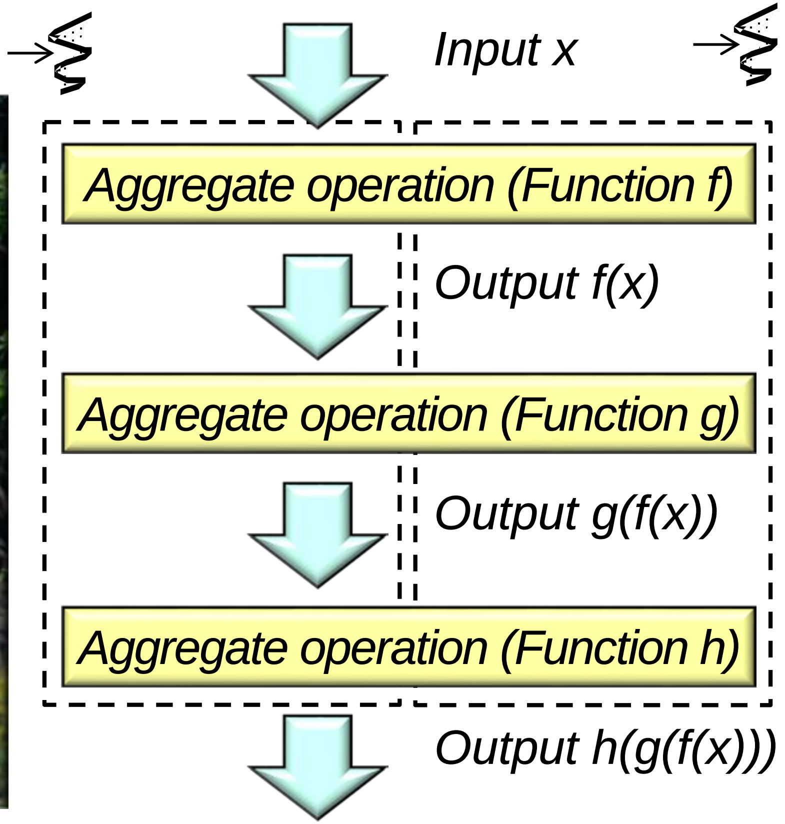
Professor of Computer Science

Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

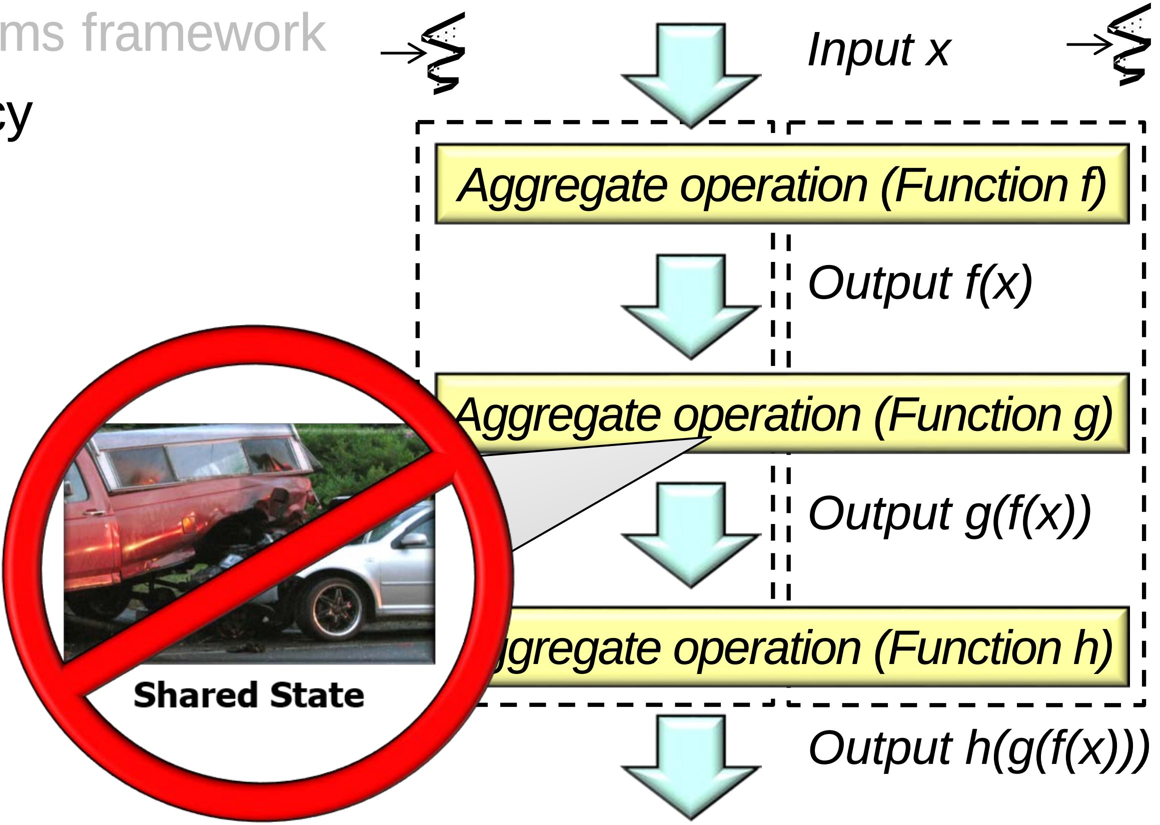
Learning Objectives in this Part of the Lesson

- Recognize how Java 8 applies aggregate operations & functional programming features in the parallel streams framework


Input x
Aggregate operation (Function f)
Output f(x)
Aggregate operation (Function g)
Output g(f(x))
Aggregate operation (Function h)
Output h(g(f(x)))

Learning Objectives in this Part of the Lesson

- Recognize how Java 8 applies aggregate operations & functional programming features in the parallel streams framework
- Be able to avoid concurrency hazards in parallel streams

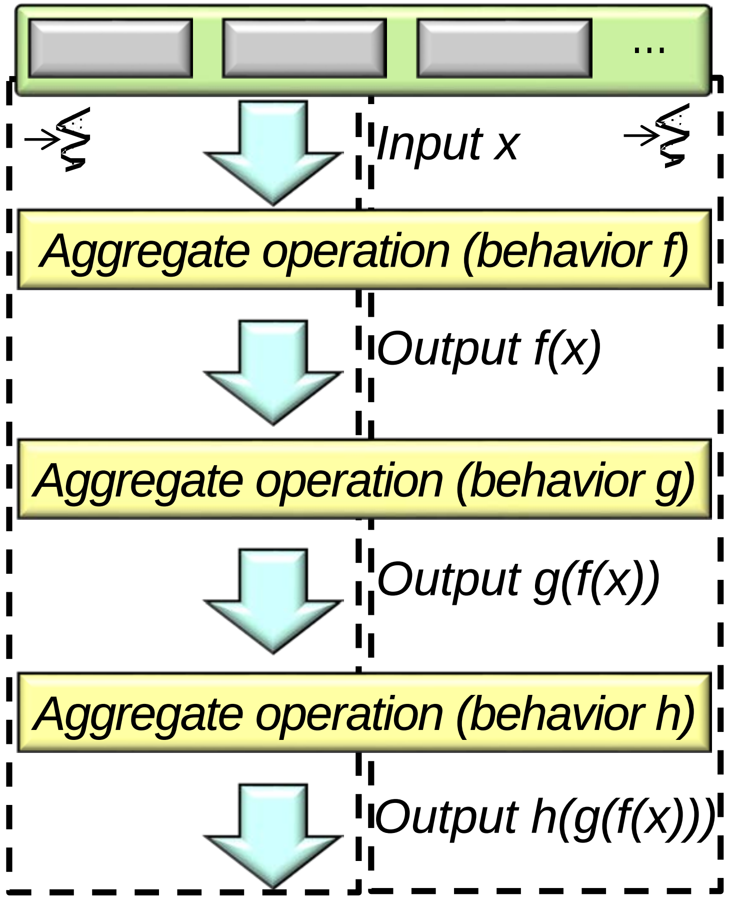


Input x
Aggregate operation (Function f)
Output f(x)
Aggregate operation (Function g)
Output g(f(x))
ggregate operation (Function h)
Output h(g(f(x)))
Shared State

Overview of Java 8 Parallel Streams

Overview of Java 8 Parallel Streams

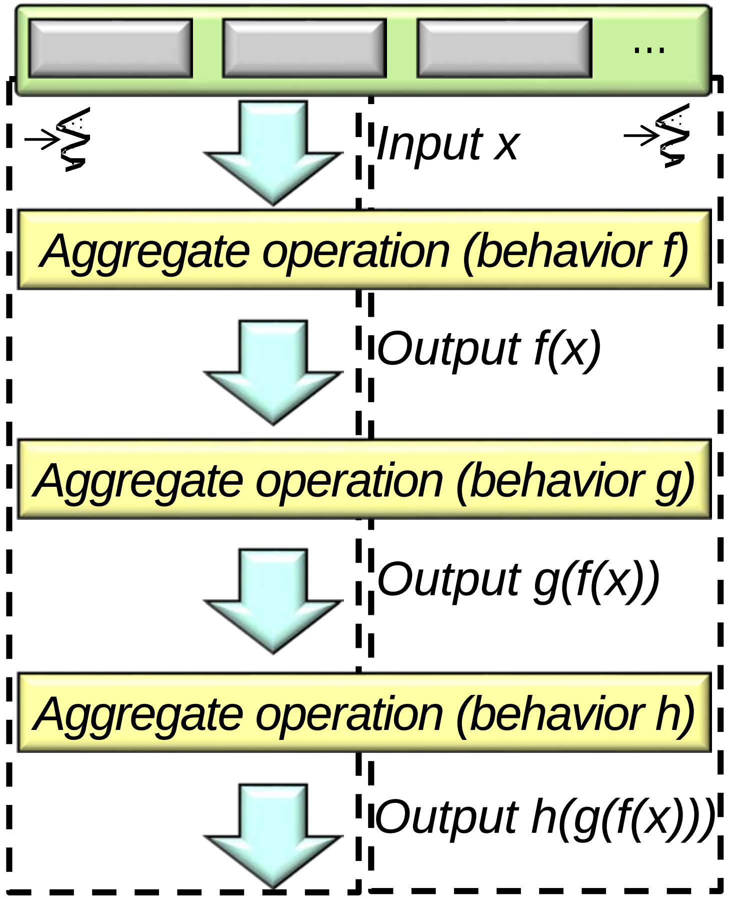
- A Java 8 parallel stream splits its elements into multiple chunks & uses a thread pool to process these chunks independently


...
Input x
Aggregate operation (behavior f)
Output f(x)
Aggregate operation (behavior g)
Output g(f(x))
Aggregate operation (behavior h)
Output h(g(f(x)))

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

Overview of Java 8 Parallel Streams

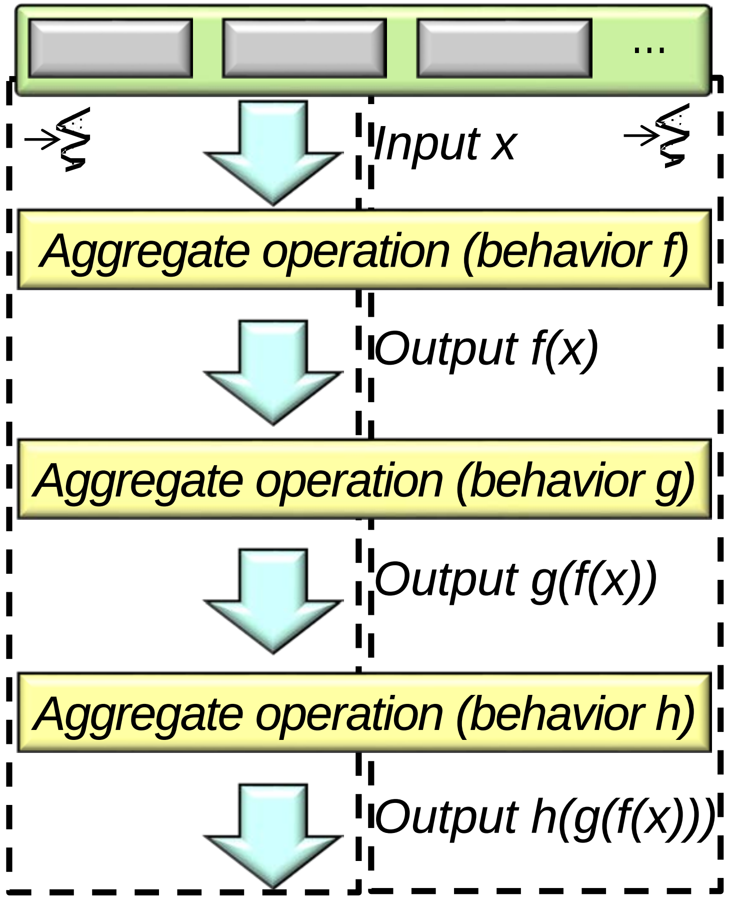
- A Java 8 parallel stream splits its elements into multiple chunks & uses a thread pool to process these chunks independently
  - This splitting & thread pool are often invisible to programmers



...
Input x
Aggregate operation (behavior f)
Output f(x)
Aggregate operation (behavior g)
Output g(f(x))
Aggregate operation (behavior h)
Output h(g(f(x)))

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

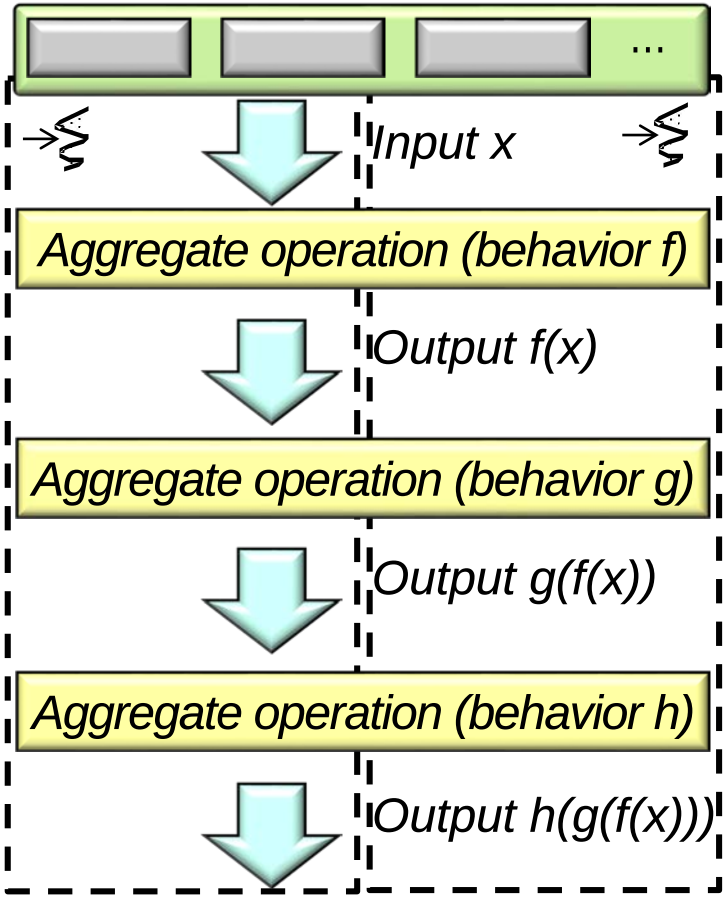


...
Input x
Aggregate operation (behavior f)
Output f(x)
Aggregate operation (behavior g)
Output g(f(x))
Aggregate operation (behavior h)
Output h(g(f(x)))

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

Overview of Java 8 Parallel Streams

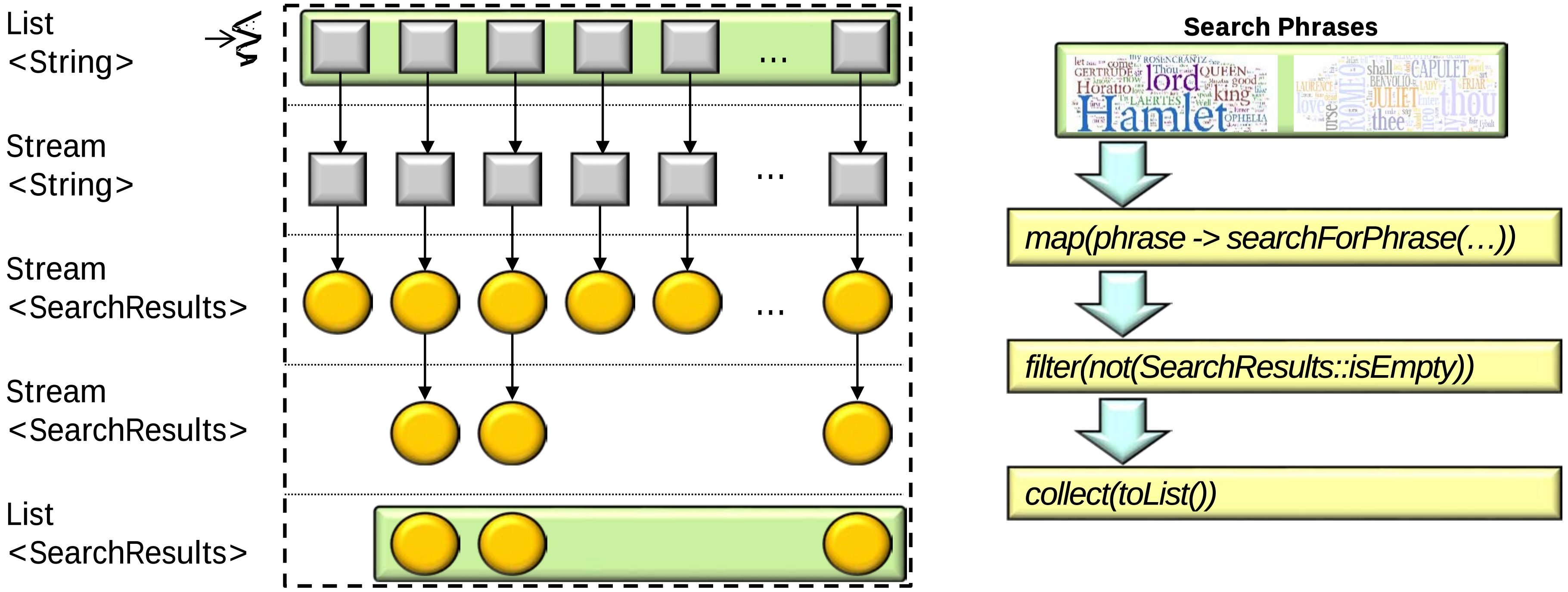
- A Java 8 parallel stream splits its elements into multiple chunks & uses a thread pool to process these chunks independently
  - This splitting & thread pool are often invisible to programmers
  - The *order* in which chunks are processed is likely non-deterministic
  - The *results* of the processing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


...
Input x
Aggregate operation (behavior f)
Output f(x)
Aggregate operation (behavior g)
Output g(f(x))
Aggregate operation (behavior h)
Output h(g(f(x)))

Programmers have more control over how the results are presented

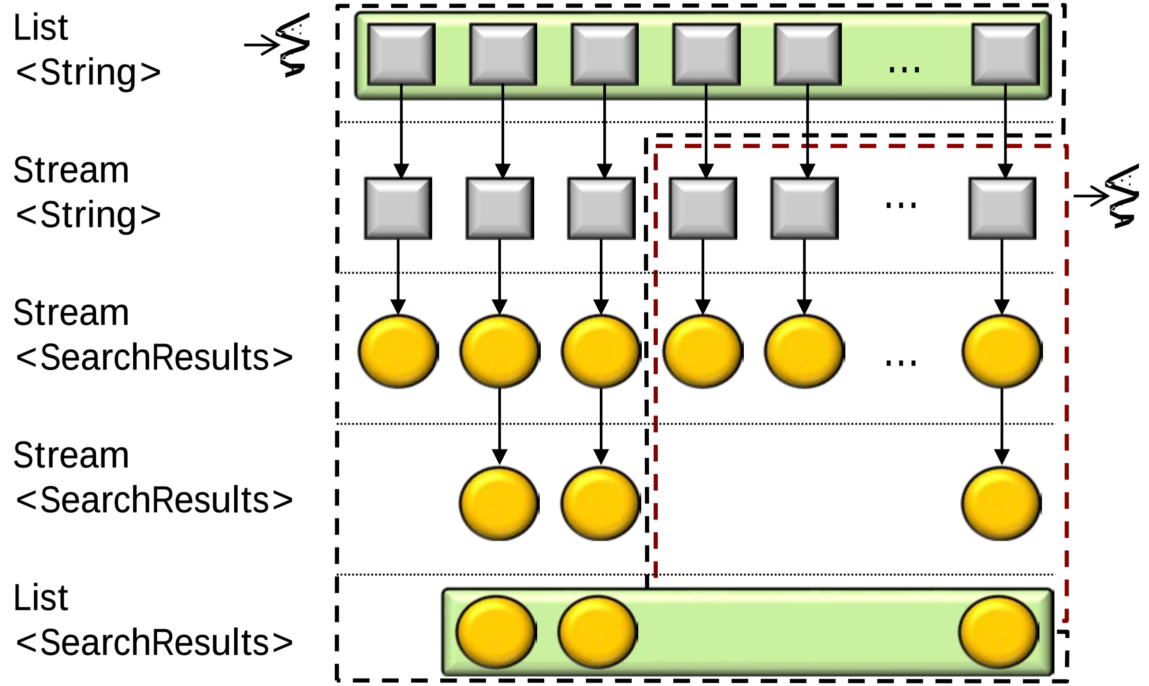
Overview of Java 8 Parallel Streams

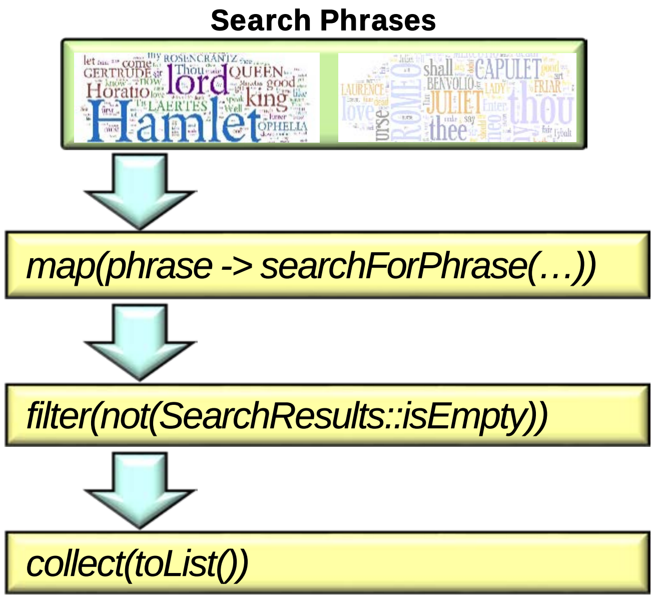
- When a stream executes sequentially all of its aggregate operations run in a single thread



Overview of Java 8 Parallel Streams

- When a stream executes in parallel, the Java runtime partitions it into multiple substream “chunks” that run in a common fork-join pool

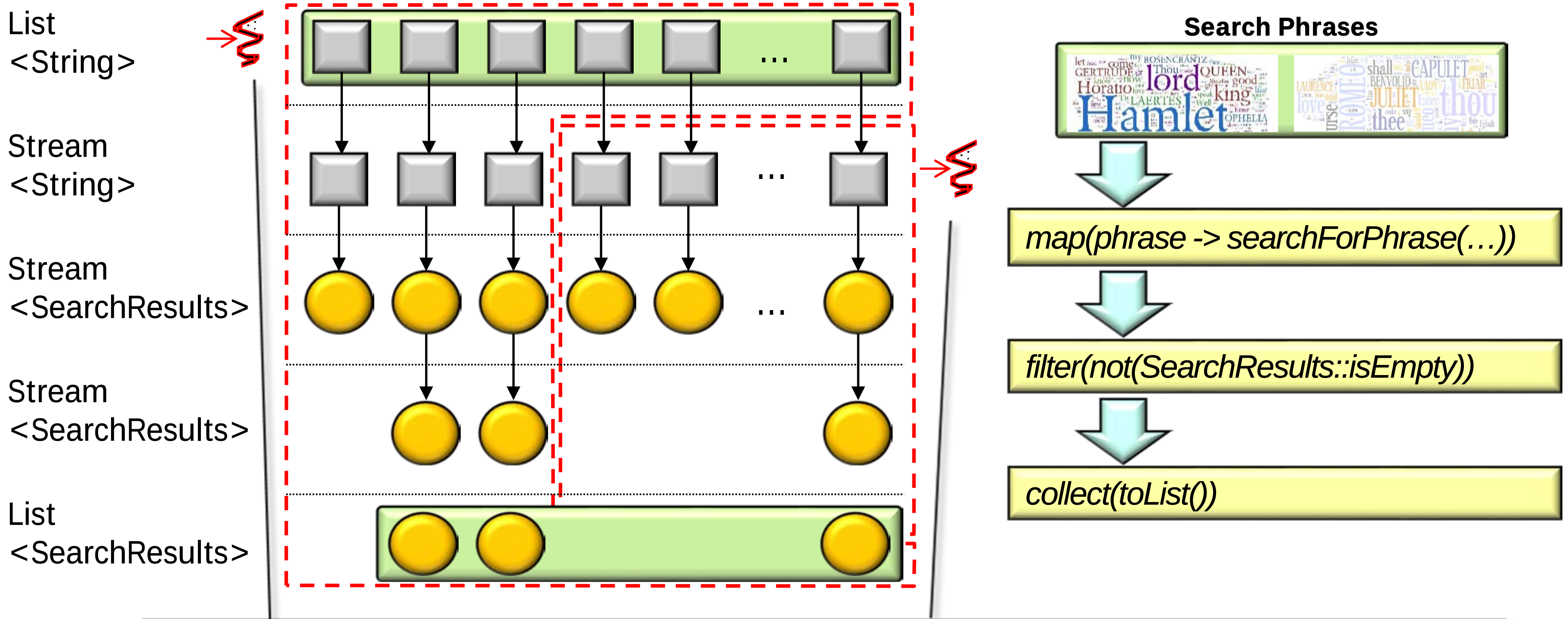



Search Phrases
map(phrase -> searchForPhrase(…))
filter(not(SearchResults::isEmpty))
collect(toList())

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

Overview of Java 8 Parallel Streams

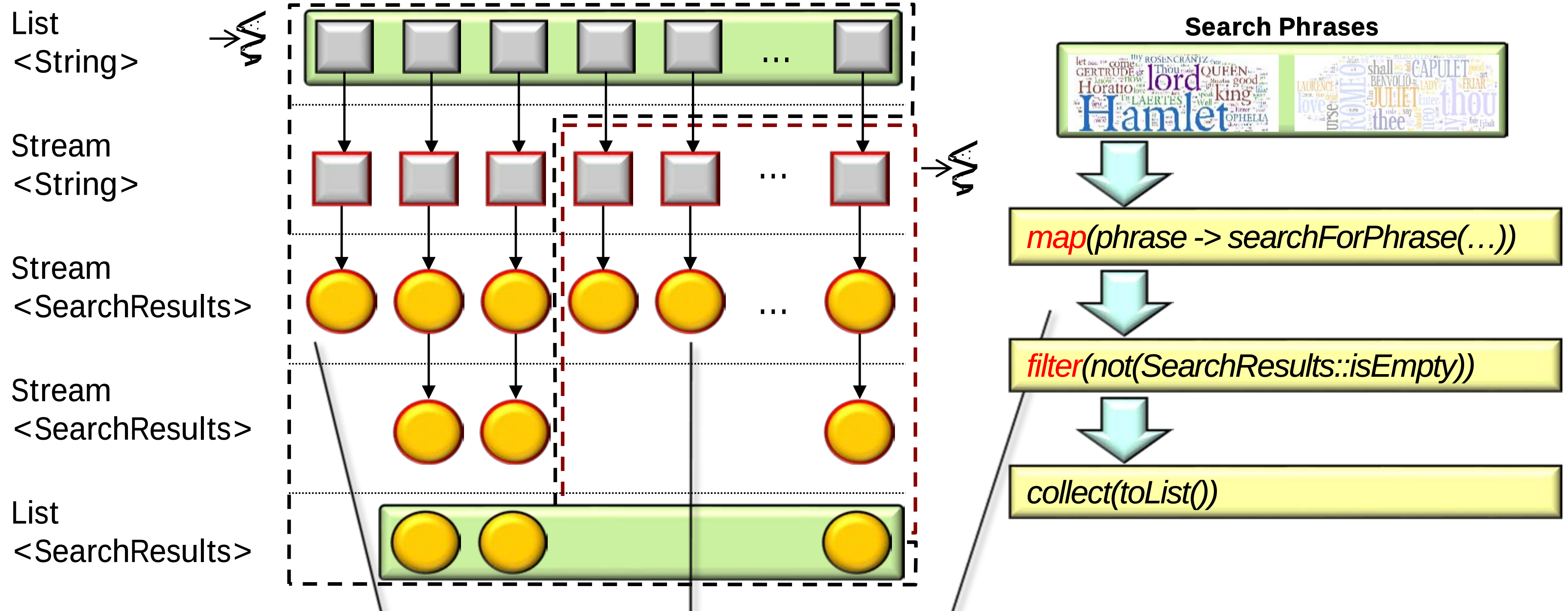
- When a stream executes in parallel, the Java runtime partitions it into multiple substream "chunks" that run in a common fork-join pool



Threads in the pool process different chunks in a non-deterministic order

Overview of Java 8 Parallel Streams

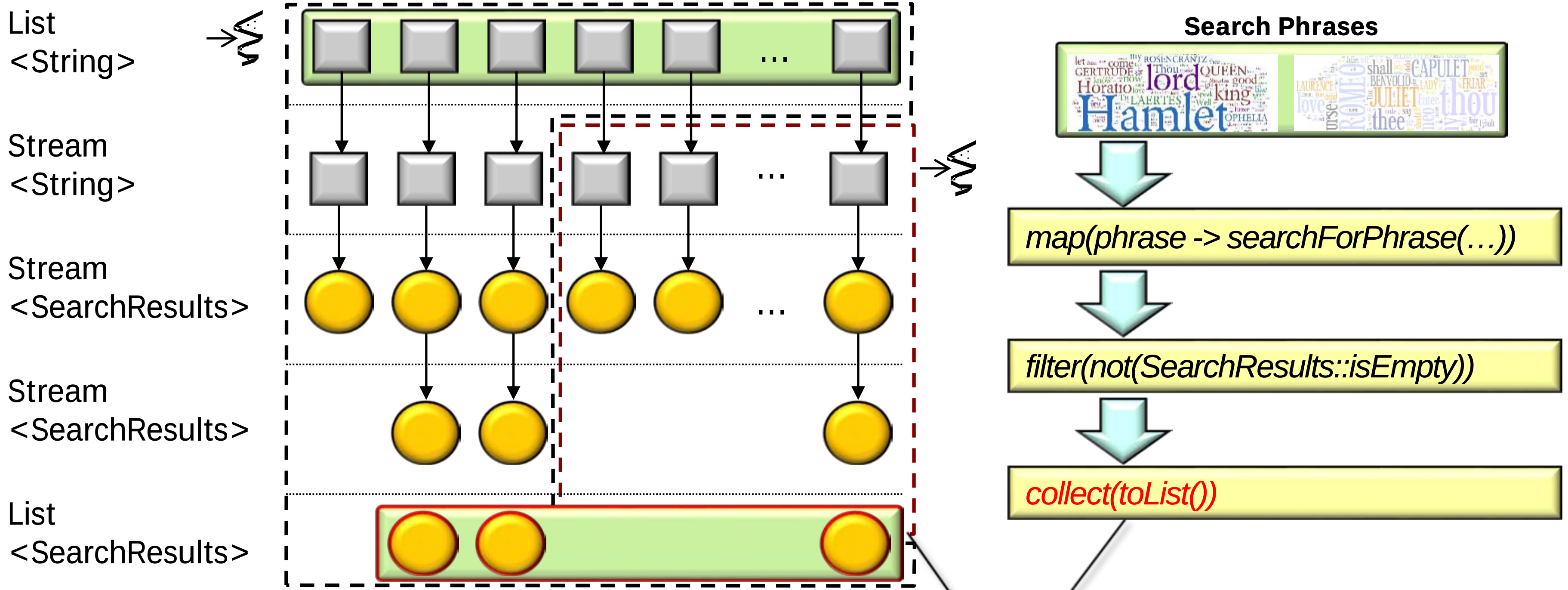
- When a stream executes in parallel, the Java runtime partitions it into multiple substream "chunks" that run in a common fork-join pool



Intermediate operations iterate over & process these chunks in parallel

Overview of Java 8 Parallel Streams

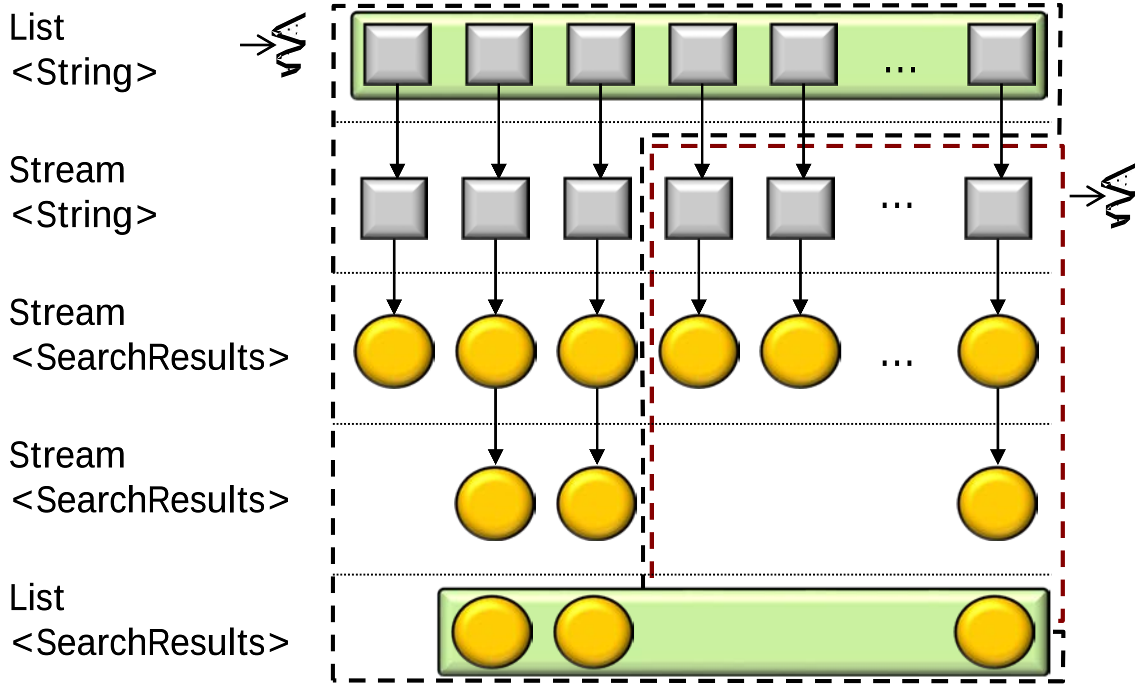
- When a stream executes in parallel, the Java runtime partitions it into multiple substream "chunks" that run in a common fork-join pool

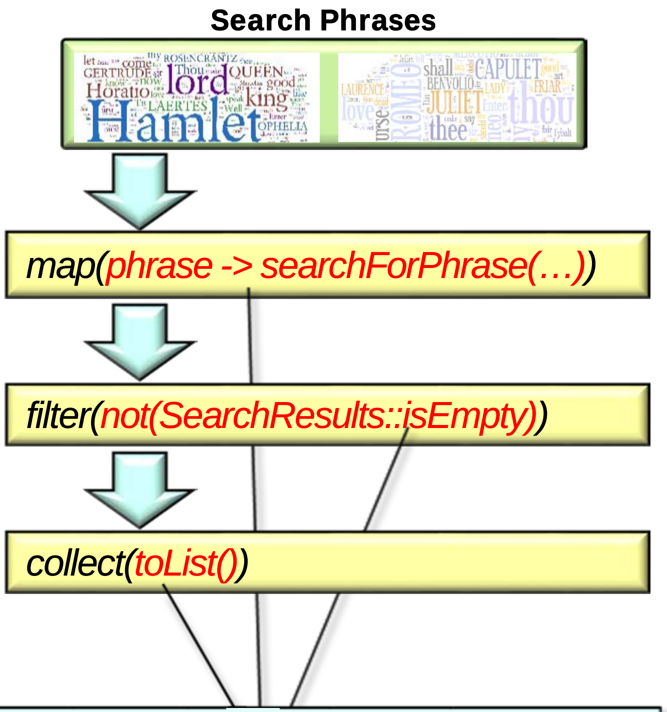


A terminal operation then combines the chunks into a single result

Overview of Java 8 Parallel Streams

- When a stream executes in parallel, the Java runtime partitions it into multiple substream "chunks" that run in a common fork-join pool

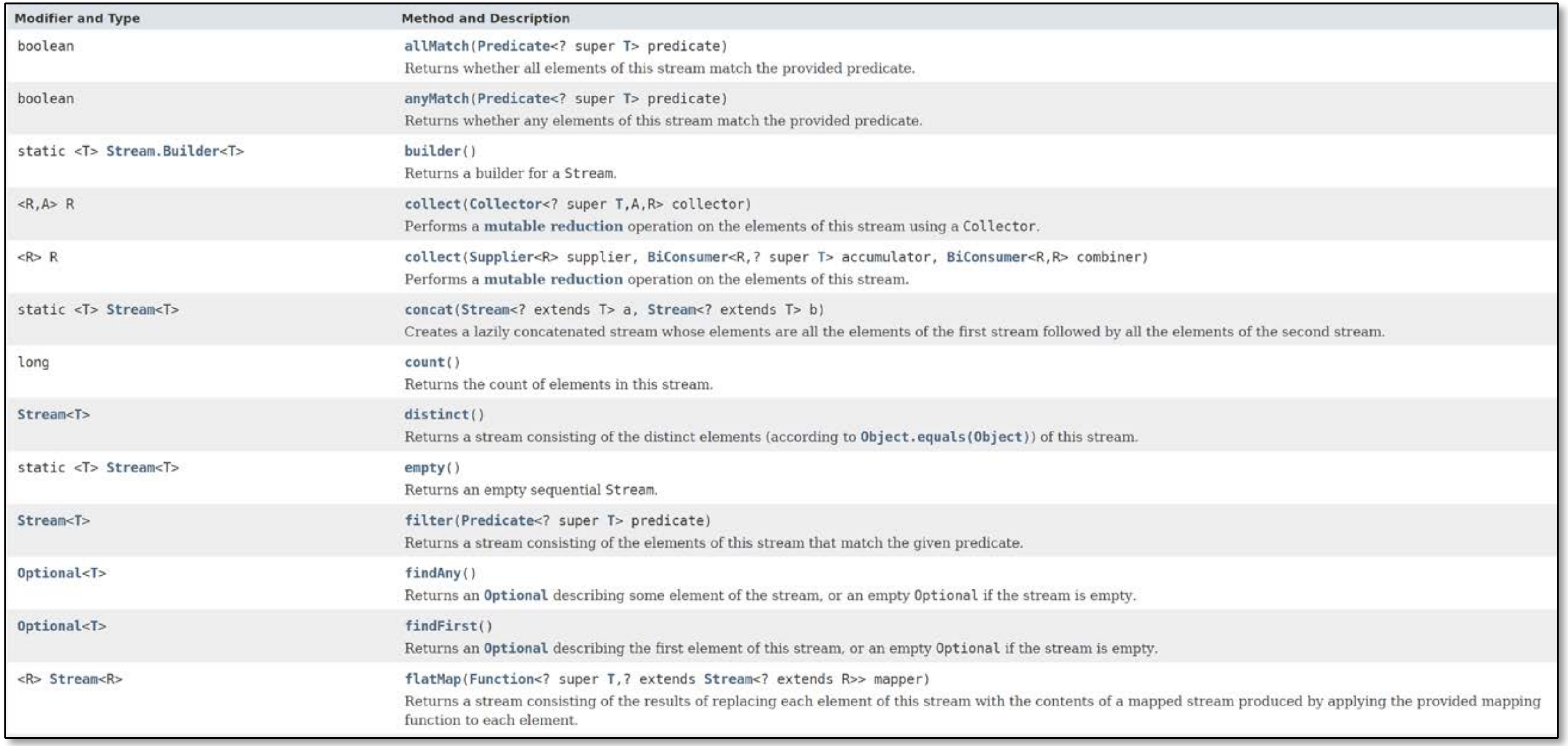



Search Phrases
map(phrase -> searchForPhrase(…))
filter(not(SearchResults::isEmpty))
collect(toList())

(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



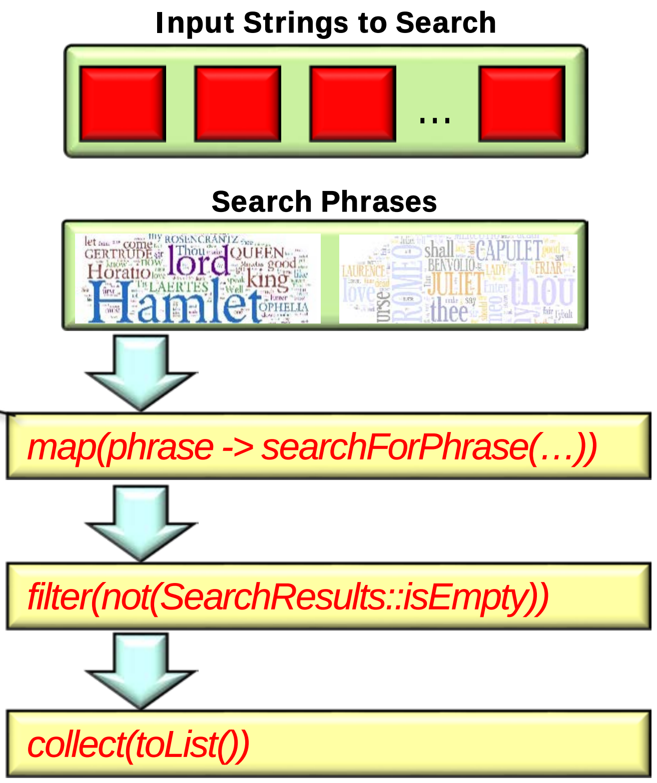
Modifier and Type	Method and Description
`boolean`	`allMatch(Predicate<? super T> predicate)` Returns whether all elements of this stream match the provided predicate.
`boolean`	`anyMatch(Predicate<? super T> predicate)` Returns whether any elements of this stream match the provided predicate.
`static <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

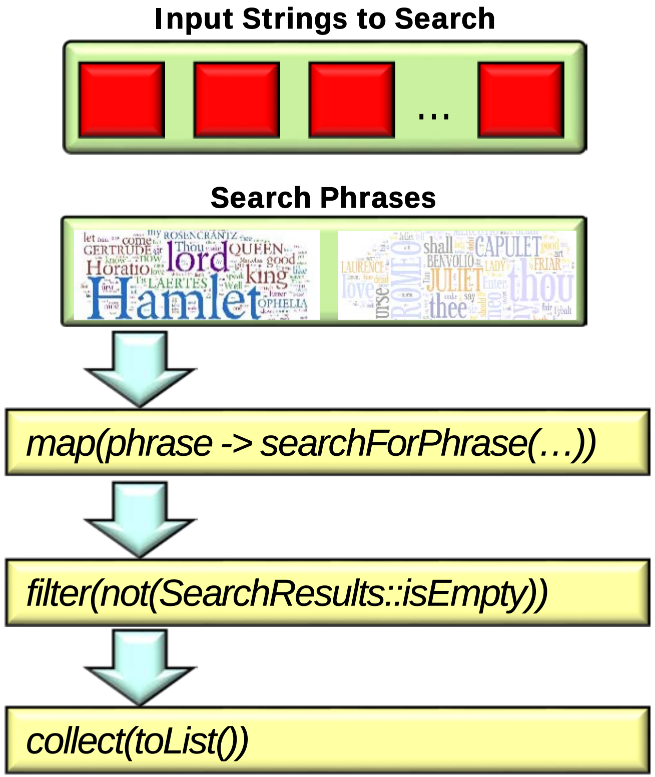


Input Strings to Search
...
Search Phrases
map(phrase -> searchForPhrase(…))
filter(not(SearchResults::isEmpty))
collect(toList())

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

Overview of Java 8 Parallel Streams

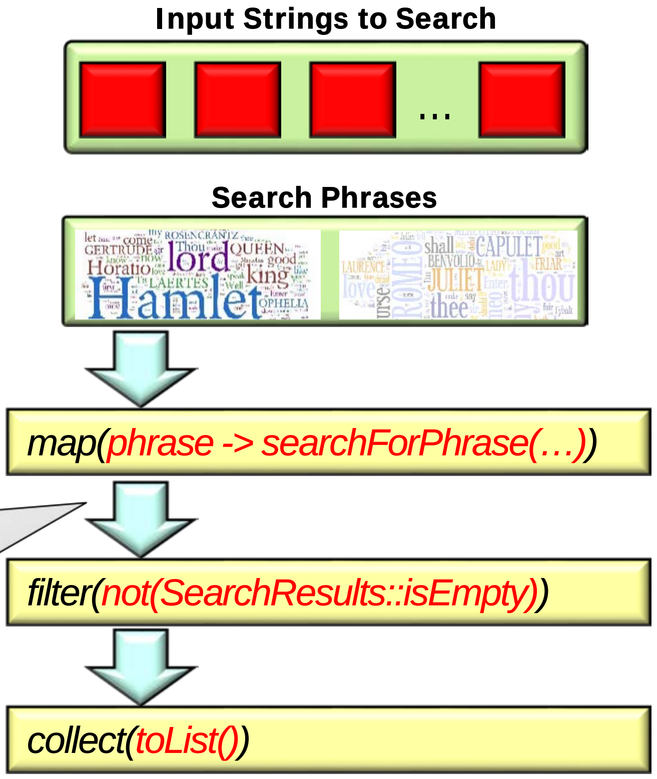
- The same aggregate operations can be used for sequential & parallel streams
  - Java 8 streams can thus treat parallelism as an optimization & leverage all available cores!



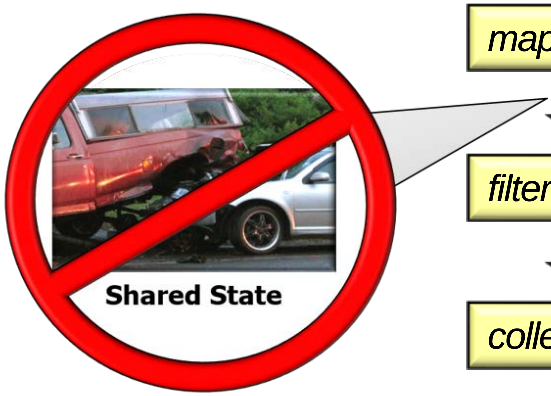
See qconlondon.com/london-2017/system/files/presentation-slides/concurrenttoparallel.pdf

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..



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



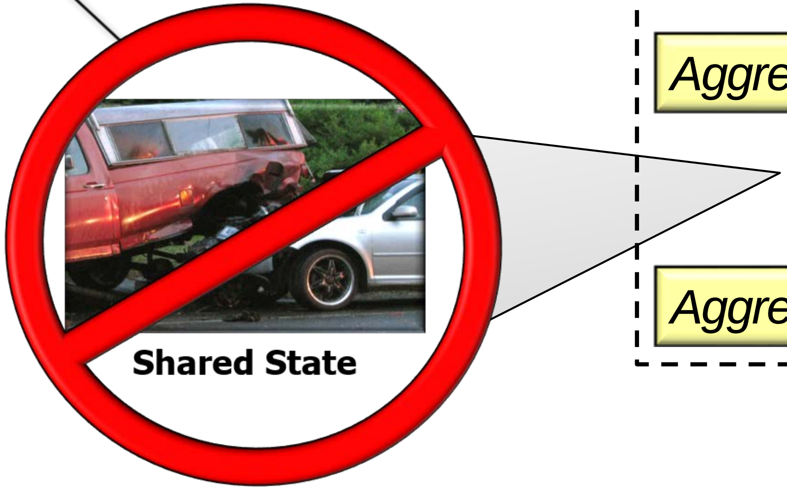
See henrikeichenhardt.blogspot.com/2013/06/why-shared-mutable-state-is-root-of-all.html

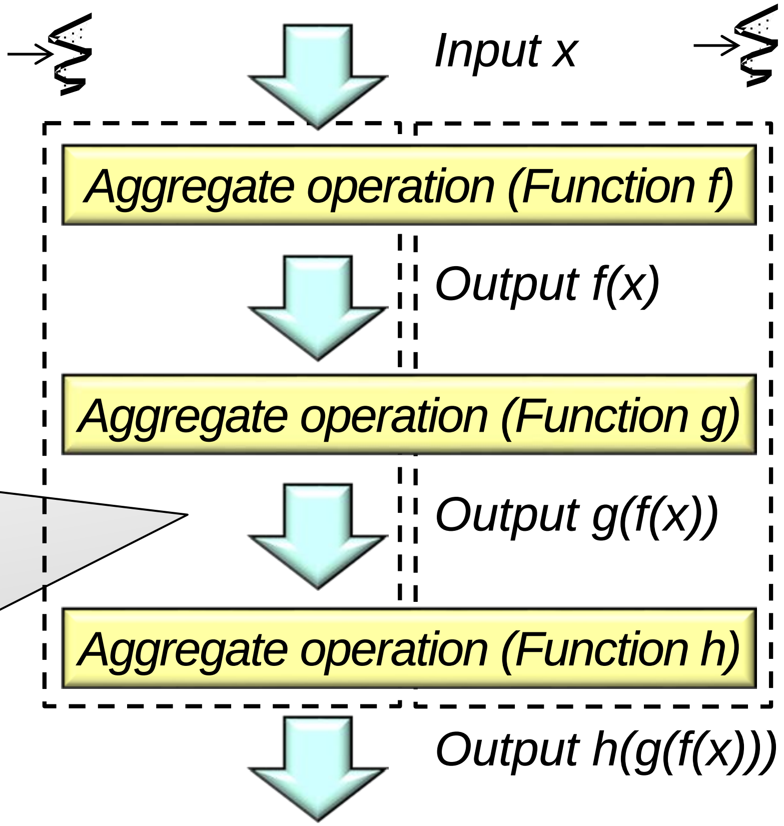
Avoiding Concurrency Hazards in Java 8 Parallel Streams

Avoiding Concurrency Hazards in Java 8 Parallel Streams

- Java 8 parallel streams assume behaviors incur no race conditions

Race conditions arise when an app depends on the sequence or timing of threads for it to operate properly

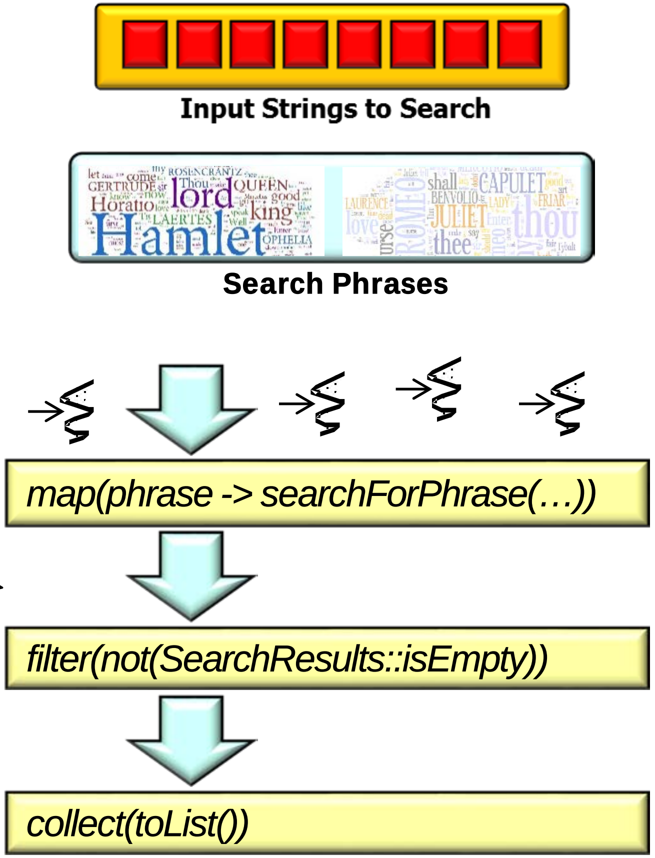



Input x
Aggregate operation (Function f)
Output f(x)
Aggregate operation (Function g)
Output g(f(x))
Aggregate operation (Function h)
Output h(g(f(x)))

See en.wikipedia.org/wiki/Race_condition#Software

Avoiding Concurrency Hazards in Java 8 Parallel Streams

- Parallel streams should therefore avoid operations with side-effects


Input Strings to Search
Search Phrases
map(phrase -> searchForPhrase(…))
filter(not(SearchResults::isEmpty))
collect(toList())

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

Avoiding Concurrency Hazards in Java 8 Parallel Streams

- Parallel streams should therefore avoid operations with side-effects, e.g.
  - *Stateful lambda expressions*
    - Where results depends on state that may change in concurrent execution of a pipeline

```
class BuggyFactorial {
  static class Total {
    long mTotal = 1;
    void multiply(long n)
    { mTotal *= n; }
  }

  static long factorial(long n){
    Total t = new Total();
    LongStream
      .rangeClosed(1, n)
      .parallel()
      .forEach(t::multiply);

    return t.mTotal;
  } ...
```

See docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html#Statelessness

Avoiding Concurrency Hazards in Java 8 Parallel Streams

- Parallel streams should therefore avoid operations with side-effects, e.g.
  - *Stateful lambda expressions*
    - Where results depends on state that may change in concurrent execution of a pipeline

```
class BuggyFactorial {
  static class Total {
    long mTotal = 1;
    void multiply(long n)
    { mTotal *= n; }
  }

  static long factorial(long n){
    Total t = new Total();
    LongStream
      .rangeClosed(1, n)
      .parallel()
      .forEach(t::multiply);

    return t.mTotal;
  } ...
```

Race conditions can arise due to the unsynchronized access to mTotal field

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex16

Avoiding Concurrency Hazards in Java 8 Parallel Streams

- Parallel streams should therefore avoid operations with side-effects, e.g.
  - *Stateful lambda expressions*
  - *Interference w/the data source*
    - Occurs when the source of a stream is modified while a pipeline processes the stream

```
List<Integer> list = IntStream
  .range(0, 10)
  .boxed()
  .collect(toList());

list
  .parallelStream()
  .peek(list::remove)
  .forEach(System.out::println);
```

See docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html#NonInterference

Avoiding Concurrency Hazards in Java 8 Parallel Streams

- Parallel streams should therefore avoid operations with side-effects, e.g.
  - *Stateful lambda expressions*
  - *Interference w/the data source*
    - Occurs when the source of a stream is modified while a pipeline processes the stream

```
List<Integer> list = IntStream
  .range(0, 10)
  .boxed()
  .collect(toList());

list
  .parallelStream()
  .peek(list::remove)
  .forEach(System.out::println);
```

Aggregate operations enable parallelism with non-thread-safe collections provided the collection is not modified while it's being operated on..

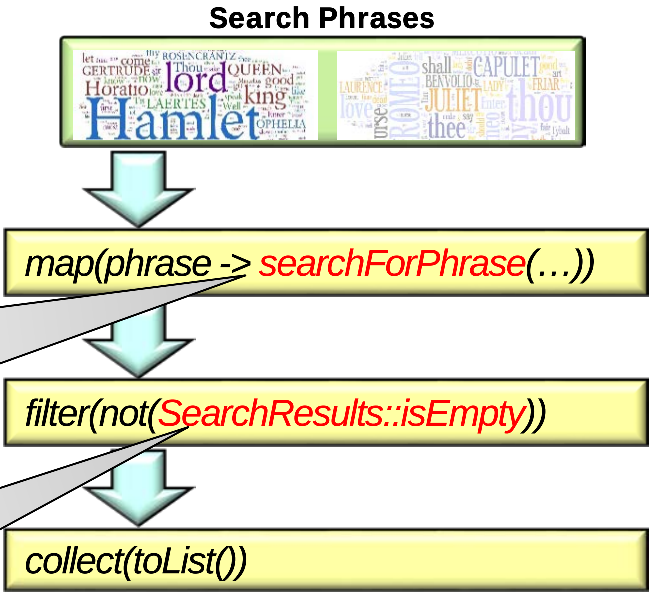
See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex11

Avoiding Concurrency Hazards in Java 8 Parallel Streams

- Java 8 lambda expressions & method references containing no shared state are useful for parallel streams since they needn't be explicitly synchronized

```
return new SearchResults
  (Thread.currentThread().getId(),
   currentCycle(), phrase, title,
   StreamSupport
     .stream(new PhraseMatchSpliterator
                  (input, phrase),
                parallel)
     .collect(toList()));
```

```
return mList.size() == 0;
```


Search Phrases
map(phrase -> searchForPhrase(…))
filter(not(SearchResults::isEmpty))
collect(toList())

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

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