Java Parallel Stream Internals: Partitioning

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

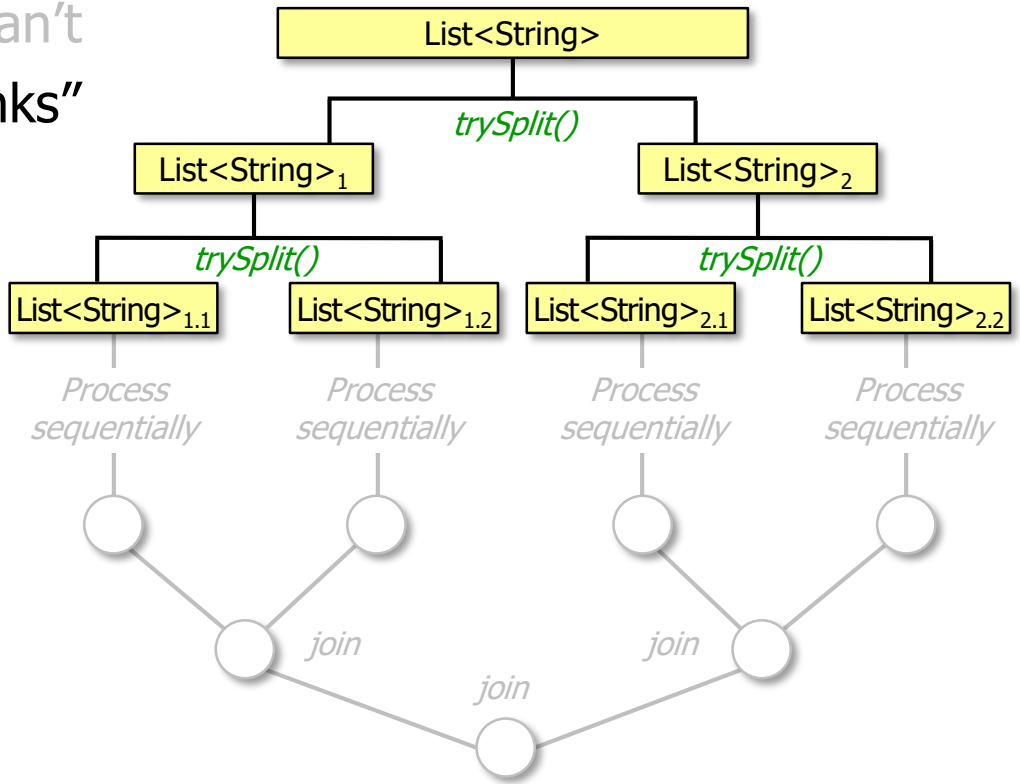
Professor of Computer Science

Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Part of the Lesson

- Understand parallel stream internals, e.g.
  - Know what can change & what can't
  - Partition a data source into "chunks"



See www.ibm.com/developerworks/library/j-java-streams-3-brian-goetz

Partitioning a Parallel Stream

Partitioning a Parallel Stream

- A "splittable iterator" (spliterator) partitions a Java parallel stream into chunks

Interface Spliterator<T>

Type Parameters:

`T - the type of elements returned by this Spliterator`

All Known Subinterfaces:

`Spliterator.OfDouble, Spliterator.OfInt, Spliterator.OfLong, Spliterator.OfPrimitive<T,T_CONS,T_SPLITR>`

All Known Implementing Classes:

`Spliterators.AbstractDoubleSpliterator, Spliterators.AbstractIntSpliterator, Spliterators.AbstractLongSpliterator, Spliterators.AbstractSpliterator`

`public interface Spliterator<T>`

An object for traversing and partitioning elements of a source. The source of elements covered by a Spliterator could be, for example, an array, a `Collection`, an IO channel, or a generator function.

A Spliterator may traverse elements individually (`tryAdvance()`) or sequentially in bulk (`forEachRemaining()`).

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

Partitioning a Parallel Stream

- We've shown how a spliterator can *traverse* elements in a source

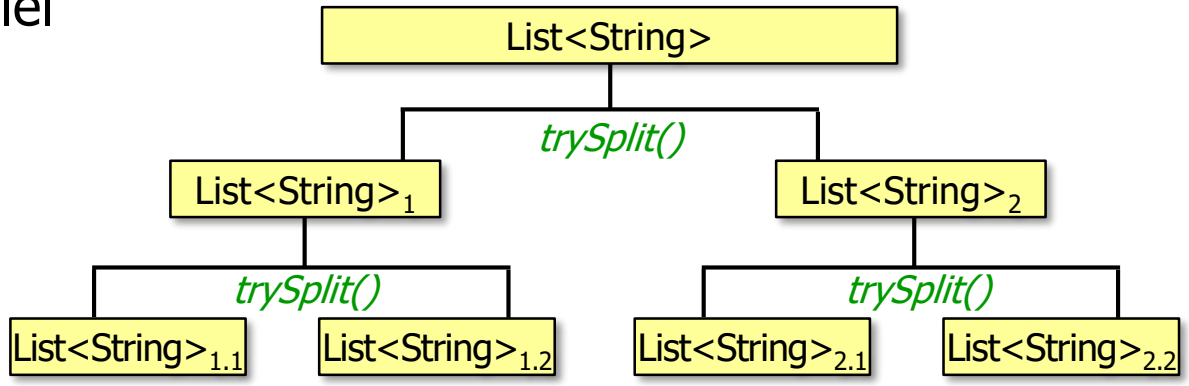
```
List<String> quote = Arrays.asList
    ("This ", "above ", "all- ",
     "to ", "thine ", "own ",
     "self ", "be ", "true", ",\n",
     ...);

for(Spliterator<String> s =
       quote.spliterator();
    s.tryAdvance(System.out::print)
       != false;
    )
  continue;
```

See earlier lesson on "*Java Streams: Overview of Spliterators*"

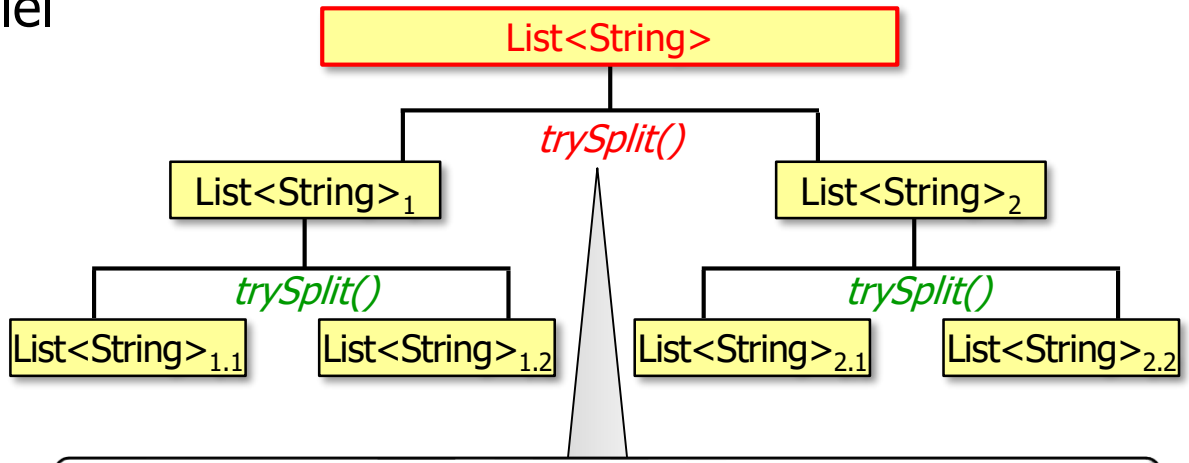
Partitioning a Parallel Stream

- We now outline how a parallel spliterator can *partition* all elements in a source



Partitioning a Parallel Stream

- We now outline how a parallel spliterator can *partition* all elements in a source

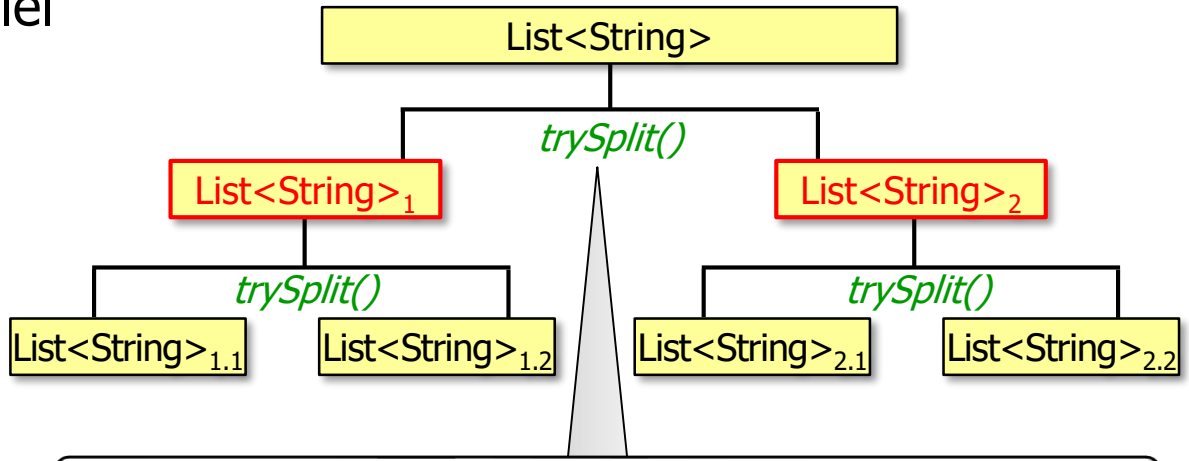



```
Spliterator<T> trySplit() {
  if (input is <= minimum size)
    return null
  else {
    split input in 2 (even-sized) chunks
    return a spliterator for "left chunk"
  }
```

The streams framework calls a spliterator's trySplit() method, not a user's app

Partitioning a Parallel Stream

- We now outline how a parallel spliterator can *partition* all elements in a source

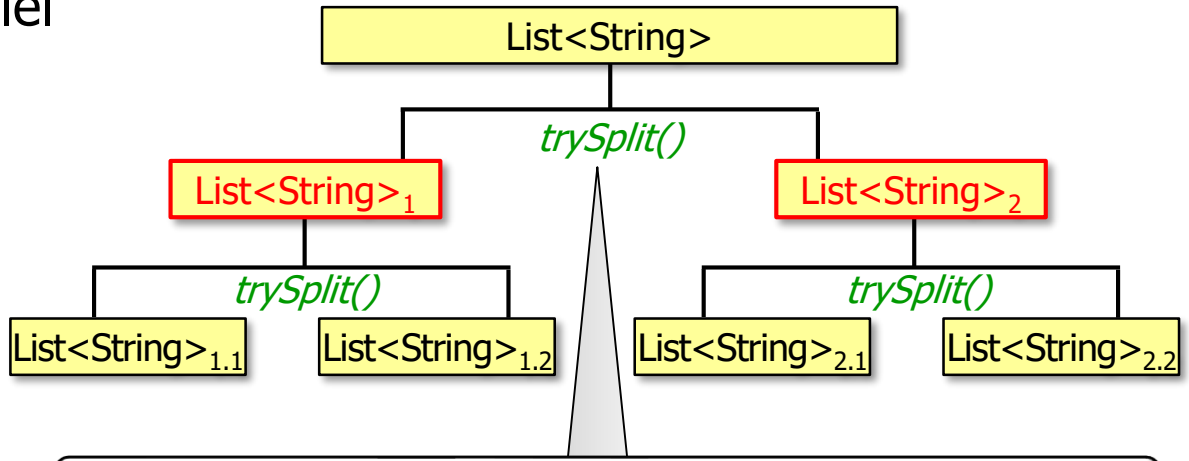



```
Spliterator<T> trySplit() {
  if (input is <= minimum size)
    return null
  else {
    split input in 2 (even-sized) chunks
    return a spliterator for "left chunk"
  }
```

trySplit() attempts to split the input evenly (if it's not <= the minimum size)

Partitioning a Parallel Stream

- We now outline how a parallel spliterator can *partition* all elements in a source

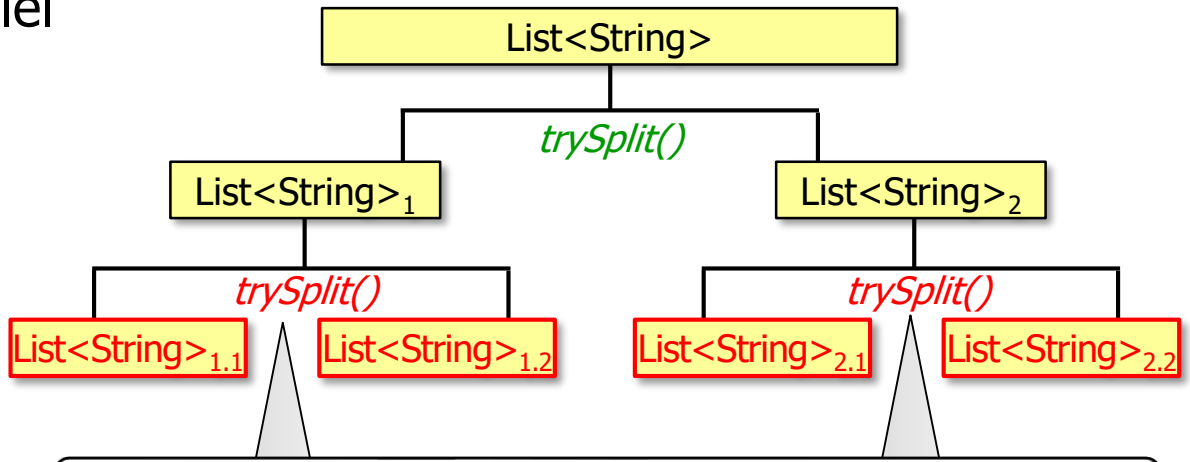



```
Spliterator<T> trySplit() {
  if (input is <= minimum size)
    return null
  else {
    split input in 2 (even-sized) chunks
    return a spliterator for "left chunk"
  }
```

A spliterator usually needs no synchronization nor does it need a "join" phase!

Partitioning a Parallel Stream

- We now outline how a parallel spliterator can *partition* all elements in a source

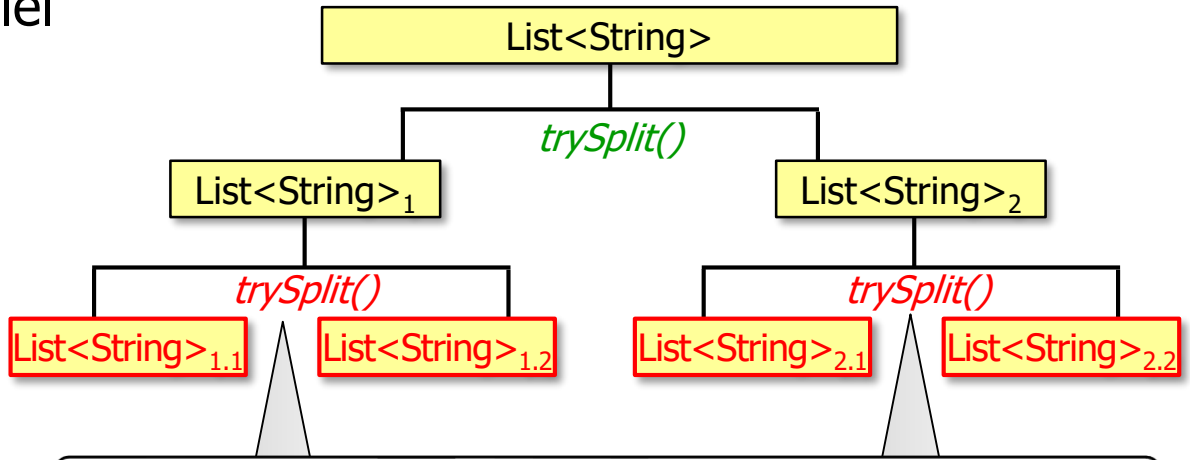



```
Spliterator<T> trySplit() {
  if (input is <= minimum size)
    return null
  else {
    split input in 2 (even-sized) chunks
    return a spliterator for "left chunk"
  }
```

trySplit() is called recursively until all chunks are <= to the minimize size

Partitioning a Parallel Stream

- We now outline how a parallel spliterator can *partition* all elements in a source

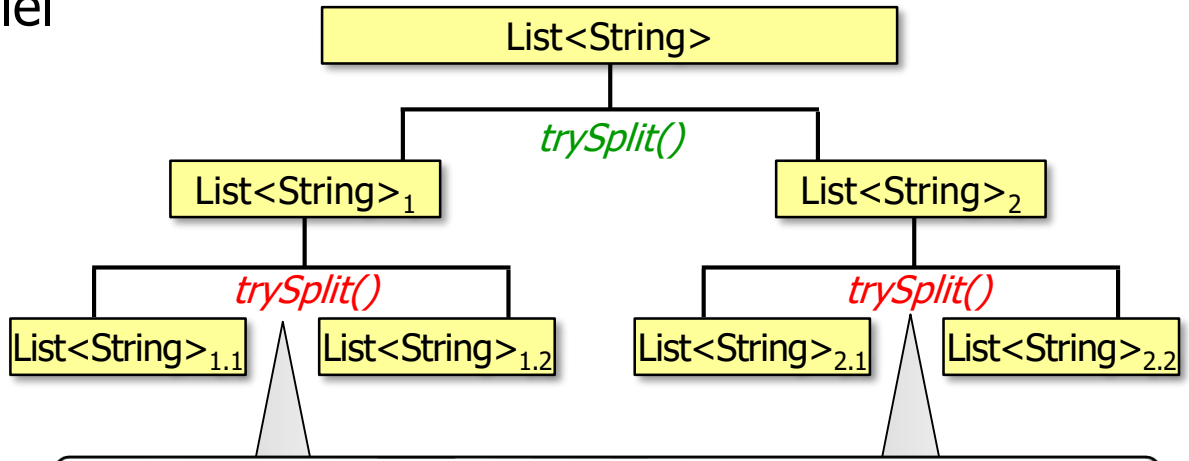


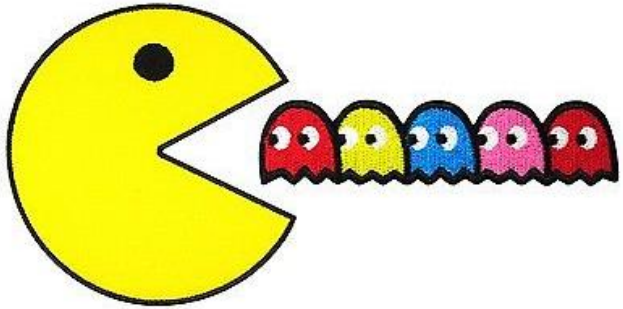

```
Spliterator<T> trySplit() {
  if (input is <= minimum size)
    return null
  else {
    split input in 2 (even-sized) chunks
    return a spliterator for "left chunk"
  }
```

trySplit() is finished when a chunk is <= to the minimize size

Partitioning a Parallel Stream

- We now outline how a parallel spliterator can *partition* all elements in a source





```
Spliterator<T> trySplit() {
  if (input is <= minimum size)
    return null
  else {
    split input in 2 (even-sized) chunks
    return a spliterator for "left chunk"
  }
```

When null is returned the streams framework processes this chunk sequentially

Partitioning a Parallel Stream

- Some Java collections split evenly & efficiently, e.g., ArrayList

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

boolean tryAdvance(Consumer<? super E> action) {
  ...
  if (index < getFence()) {
    action.accept((E) list.elementData[i]); ...
    return true;
  } return false;
}
```

See openjdk/8u40-b25/java/util/ArrayList.java

Partitioning a Parallel Stream

- Some Java collections split evenly & efficiently, e.g., ArrayList

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

Split the array evenly each time until there's nothing left to split

```
boolean tryAdvance(Consumer<? super E> action) {
  ...
  if (index < getFence()) {
    action.accept((E) list.elementData[i]); ...
    return true;
  } return false;
}
```

Partitioning a Parallel Stream

- Some Java collections split evenly & efficiently, e.g., ArrayList

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

Try to consume a single element on each call

```
boolean tryAdvance(Consumer<? super E> action) {
  ...
  if (index < getFence()) {
    action.accept((E) list.elementData[i]); ...
    return true;
  } return false;
}
```

Partitioning a Parallel Stream

- Other Java collections do *not* split evenly & efficiently, e.g., LinkedList

```
Spliterator<E> trySplit() { ...
  int n = batch + BATCH_UNIT, j = 0; Object[] a = new Object[n];
  do { a[j++] = p.item; }
  while ((p = p.next) != null && j < n); ...
  return Spliterators.spliterator(a, 0, j, Spliterator.ORDERED);
}

boolean tryAdvance(Consumer<? super E> action) { ...
  Node<E> p;
  if (getEst() > 0 && (p = current) != null) {
    --est; E e = p.item; current = p.next;
    action.accept(e); return true;
  } return false;
}
```

See openjdk/8u40-b25/java/util/LinkedList.java

Partitioning a Parallel Stream

- Other Java collections do *not* split evenly & efficiently, e.g., LinkedList

```
Spliterator<E> trySplit() { ...
  int n = batch + BATCH_UNIT, j = 0; Object[] a = new Object[n];
  do { a[j++] = p.item; }
  while ((p = p.next) != null && j < n); ...
  return Spliterators.spliterator(a, 0, j, Spliterator.ORDERED);
}
```

Split the list into "batches", rather than evenly in half

```
boolean tryAdvance(Consumer<? super E> action) { ...
  Node<E> p;
  if (getEst() > 0 && (p = current) != null) {
    --est; E e = p.item; current = p.next;
    action.accept(e); return true;
  } return false;
}
```

Partitioning a Parallel Stream

- Other Java collections do *not* split evenly & efficiently, e.g., LinkedList

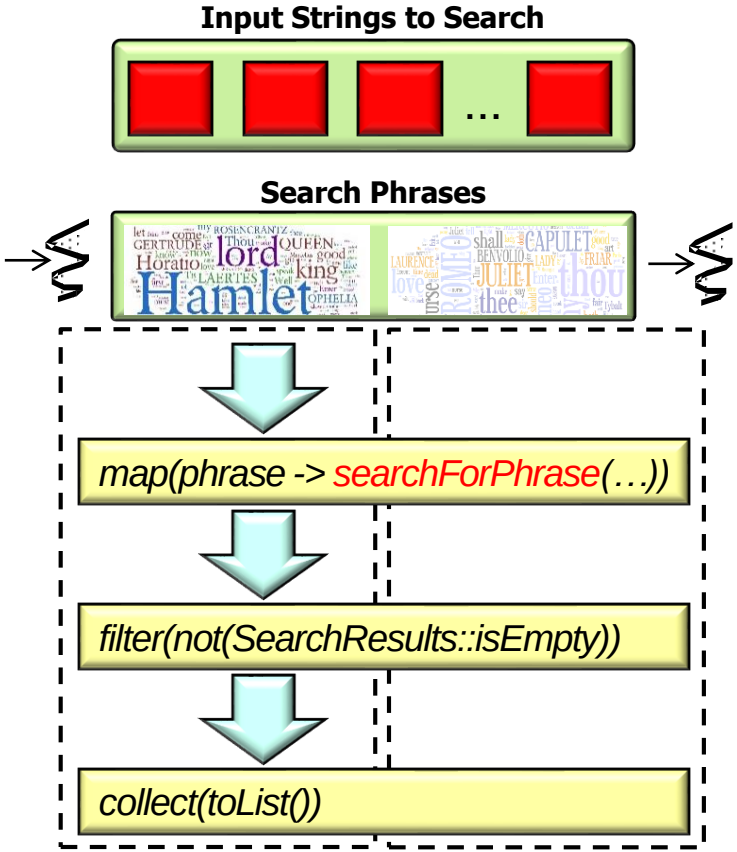
```
Spliterator<E> trySplit() { ...
  int n = batch + BATCH_UNIT, j = 0; Object[] a = new Object[n];
  do { a[j++] = p.item; }
  while ((p = p.next) != null && j < n); ...
  return Spliterators.spliterator(a, 0, j, Spliterator.ORDERED);
}
```

Try to consume a single element on each call

```
boolean tryAdvance(Consumer<? super E> action) { ...
  Node<E> p;
  if (getEst() > 0 && (p = current) != null) {
    --est; E e = p.item; current = p.next;
    action.accept(e); return true;
  } return false;
}
```

Partitioning a Parallel Stream

- We'll cover the implementation details of parallel spliterators in upcoming lessons



See *"Java SearchWithParallelSpliterator Example: trySplit()"*

End of Java Parallel Stream Internals: Partitioning