How Java Parallel Streams Work "Under the Hood"

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt
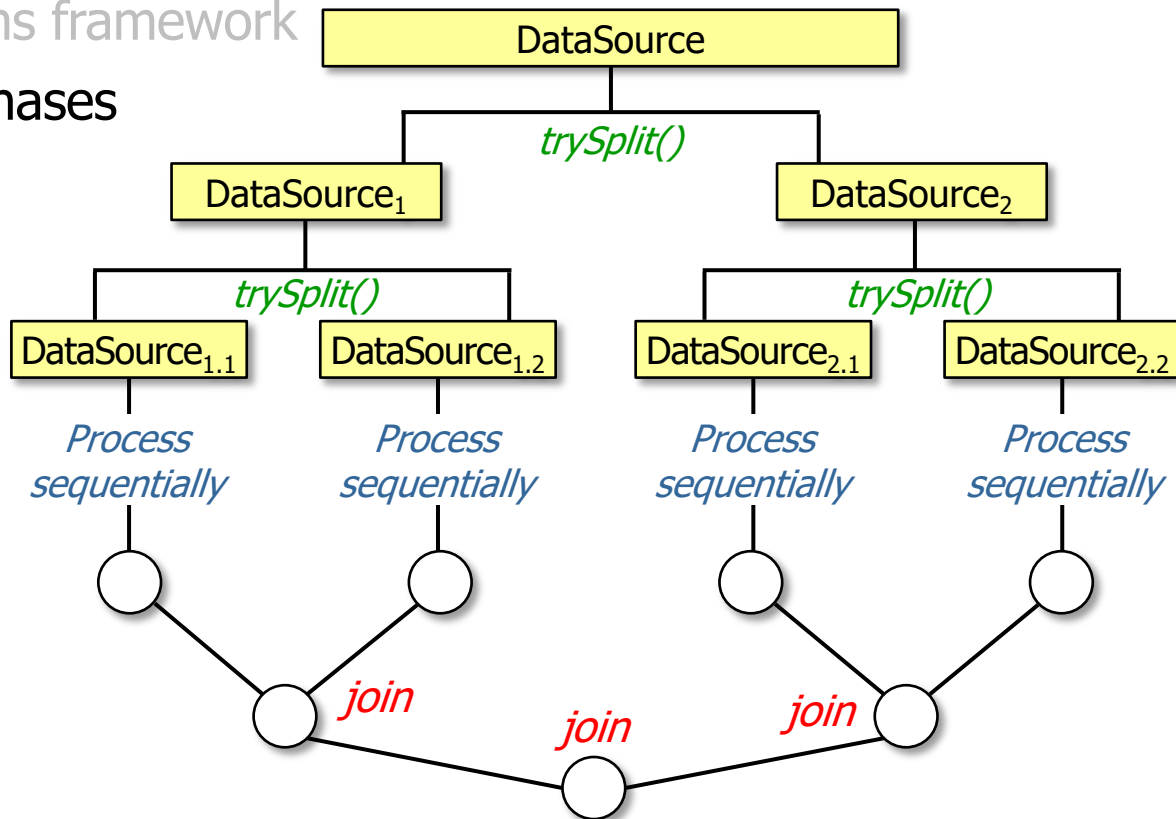
Professor of Computer Science

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

- Learn how parallel stream phases work "under the hood"


```
                        ┌──────────────┐
                        │  DataSource  │
                        └──────────────┘
                           trySplit()
          ┌──────────────┐        ┌──────────────┐
          │ DataSource₁  │        │ DataSource₂  │
          └──────────────┘        └──────────────┘
             trySplit()              trySplit()
  ┌──────────────┐ ┌──────────────┐ ┌──────────────┐ ┌──────────────┐
  │DataSource₁.₁ │ │DataSource₁.₂ │ │DataSource₂.₁ │ │DataSource₂.₂ │
  └──────────────┘ └──────────────┘ └──────────────┘ └──────────────┘
      Process          Process          Process          Process
    sequentially     sequentially     sequentially     sequentially
```

$DataSource_1$ $DataSource_2$ $DataSource_{1.1}$ $DataSource_{1.2}$ $DataSource_{2.1}$ $DataSource_{2.2}$

join   *join*   *join*

See developer.ibm.com/articles/j-java-streams-3-brian-goetz

Overview of How a Parallel Stream Works
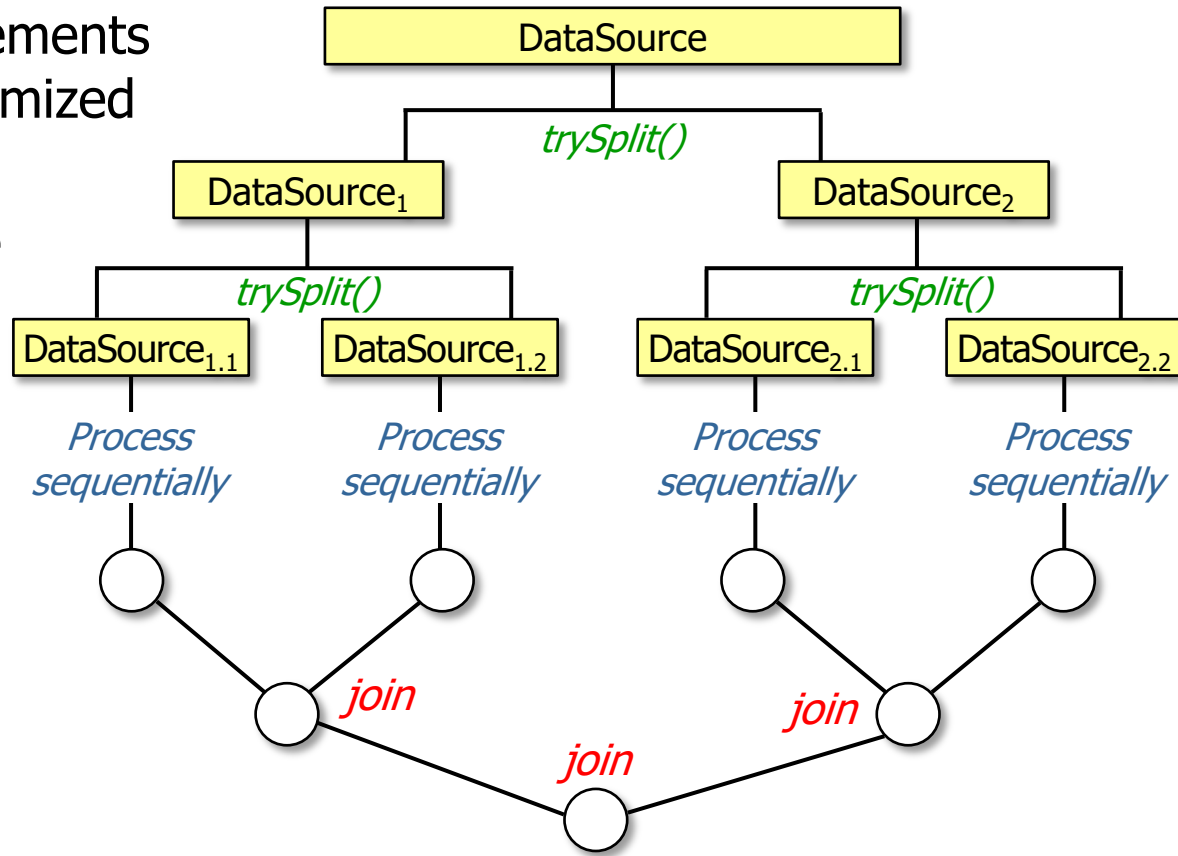
Overview of How a Parallel Stream Works

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

Map

Reduce

Partition
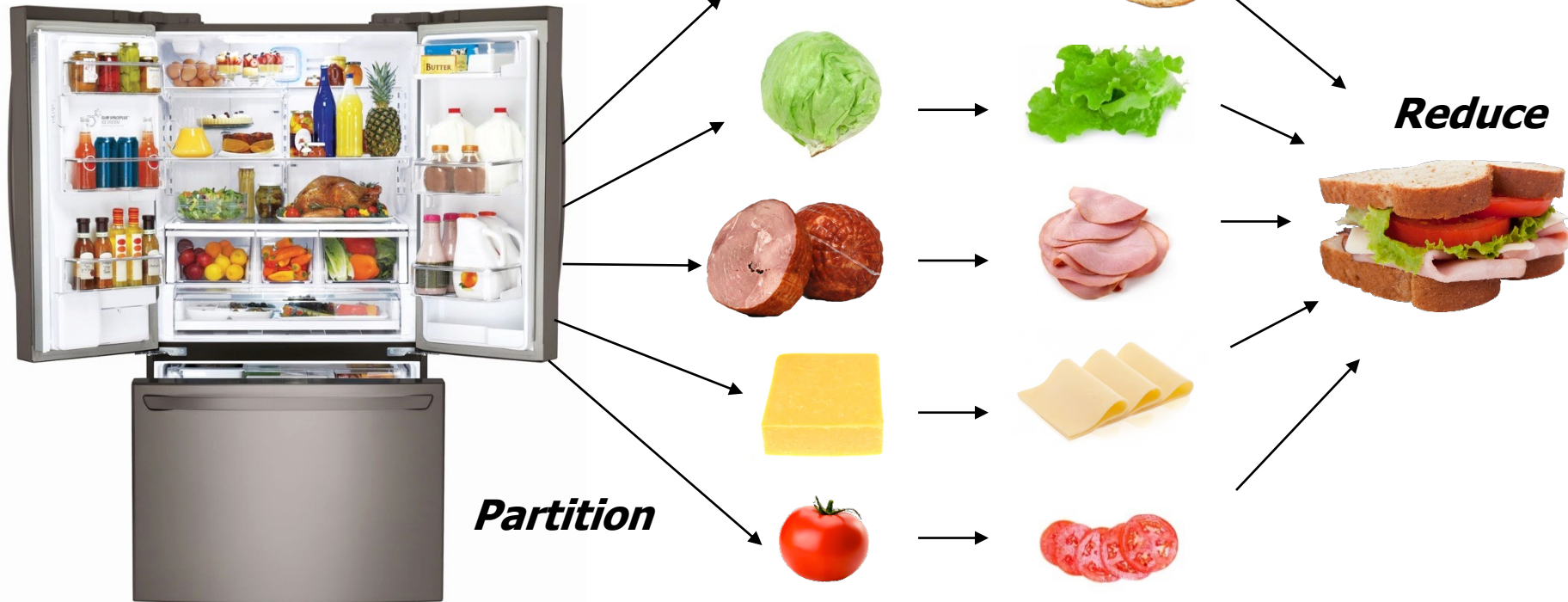
See en.wikipedia.org/wiki/MapReduce
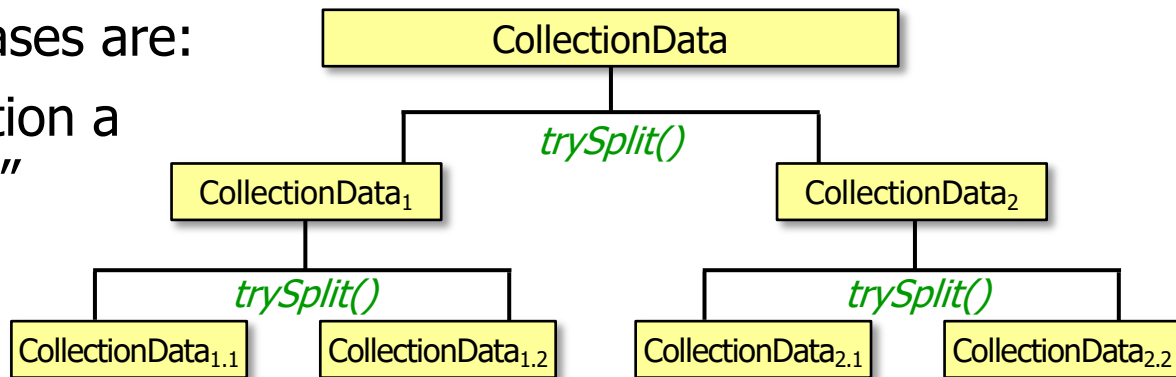
Overview of How a Parallel Stream Works

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

  - It's actually a three phase "split-apply-combine" data processing strategy

DataSource

$trySplit()$

$DataSource_1$ $DataSource_2$

$trySplit()$ $trySplit()$

$DataSource_{1.1}$ $DataSource_{1.2}$ $DataSource_{2.1}$ $DataSource_{2.2}$

Process sequentially   *Process sequentially*   *Process sequentially*   *Process sequentially*

join   *join*

join

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

Overview of How a Parallel Stream Works

- The split-apply-combine phases are:

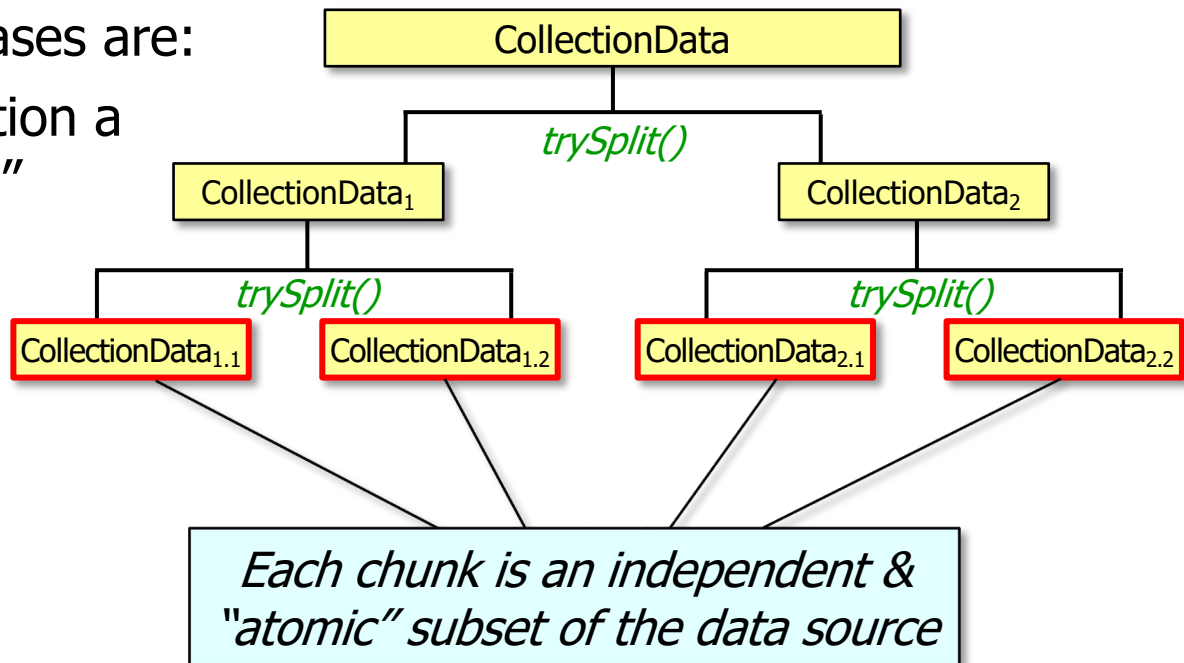 1. **Split** – Recursively partition a data source into "chunks"

See en.wikipedia.org/wiki/Divide_and_conquer_algorithm

Overview of How a Parallel Stream Works

- The split-apply-combine phases are:

  1. **Split** – Recursively partition a data source into "chunks"

```
                          CollectionData
                              │
                          trySplit()
                    ┌─────────┴─────────┐
            CollectionData₁        CollectionData₂
                 │                      │
             trySplit()             trySplit()
          ┌──────┴──────┐       ┌──────┴──────┐
  CollectionData₁.₁  CollectionData₁.₂  CollectionData₂.₁  CollectionData₂.₂
```

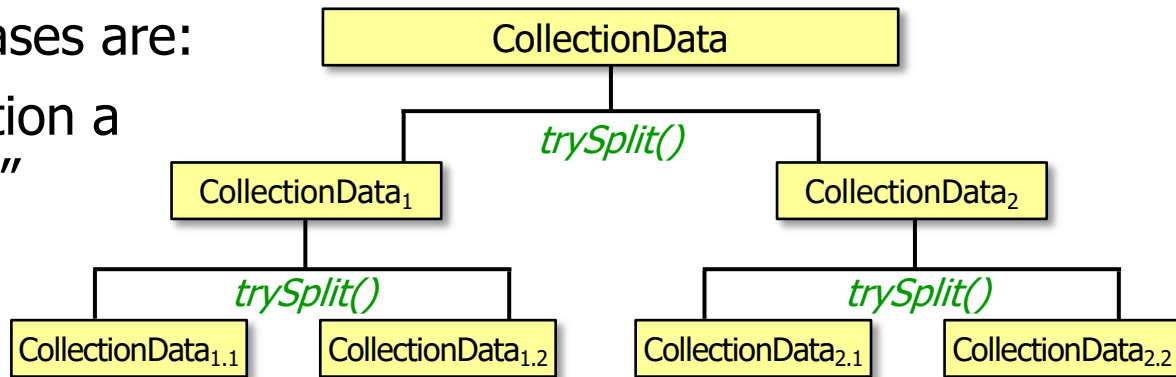*Each chunk is an independent & "atomic" subset of the data source*

See upcoming lesson on "*Java Parallel Stream Internals: Partitioning*"

Overview of How a Parallel Stream Works

- The split-apply-combine phases are:

1. **Split** – Recursively partition a data source into "chunks"
   - Spliterators partition collections in Java

```
public interface Spliterator<T> {
  boolean tryAdvance(Consumer<? Super T> action);

  Spliterator<T> trySplit();

  long estimateSize();

  int characteristics();
}
```
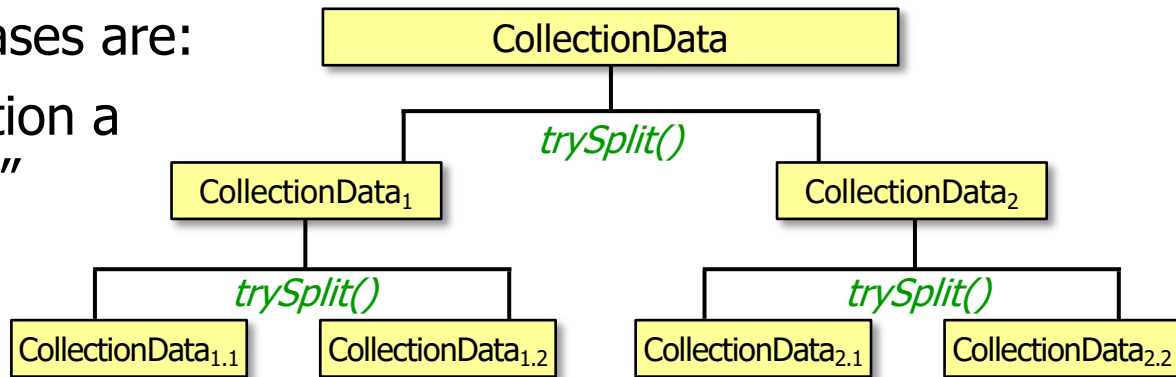
See docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html

Overview of How a Parallel Stream Works

- The split-apply-combine phases are:

1. **Split** – Recursively partition a data source into "chunks"

   - Spliterators partition collections in Java

```
CollectionData
        trySplit()
CollectionData₁              CollectionData₂
      trySplit()                   trySplit()
CollectionData₁.₁  CollectionData₁.₂   CollectionData₂.₁  CollectionData₂.₂
```



```java
public interface Spliterator<T> {
  boolean tryAdvance(Consumer<? Super T> action);

  Spliterator<T> trySplit();

  long estimateSize();

  int characteristics();
}
```

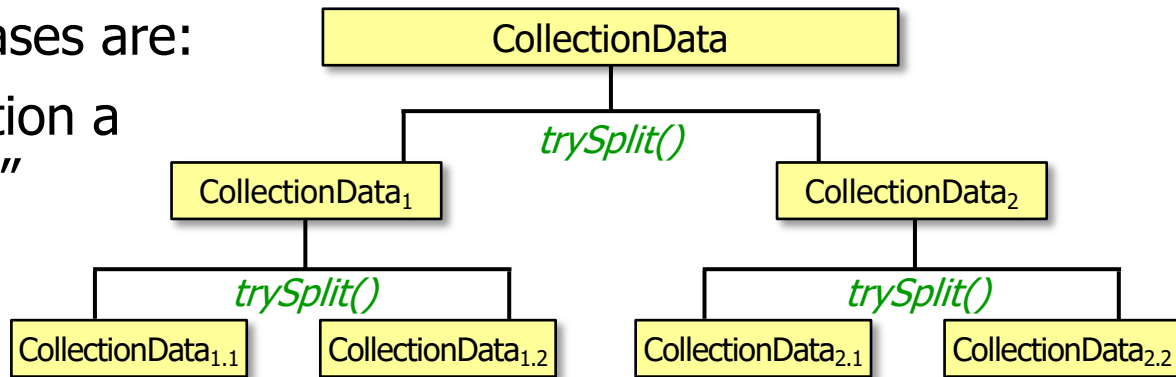*Used for sequential (& parallel) streams*

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

Overview of How a Parallel Stream Works

- The split-apply-combine phases are:

1. **Split** – Recursively partition a data source into "chunks"

   - Spliterators partition collections in Java

```
CollectionData
   trySplit()
CollectionData₁          CollectionData₂
   trySplit()               trySplit()
CollectionData₁.₁  CollectionData₁.₂   CollectionData₂.₁  CollectionData₂.₂
```

Used only for parallel streams

```java
public interface Spliterator<T> {
  boolean tryAdvance(Consumer<? Super T> action);

  Spliterator<T> trySplit();

  long estimateSize();

  int characteristics();
}
```
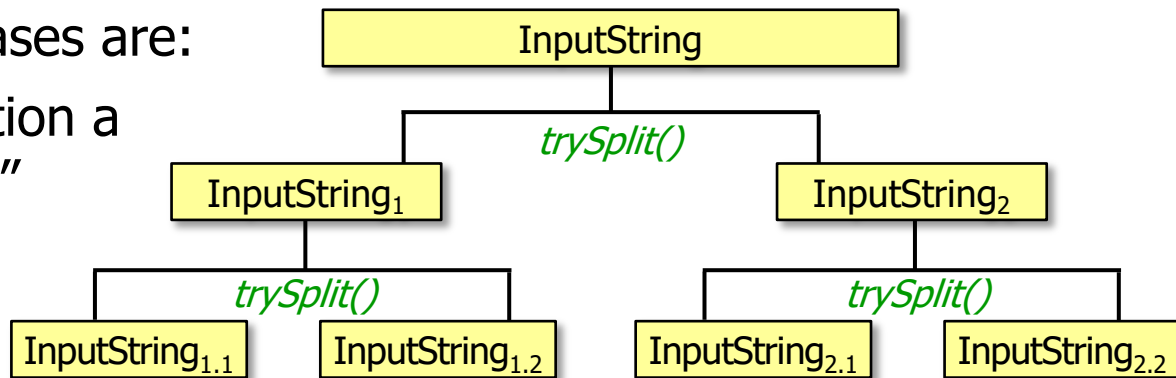
See docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html#trySplit

Overview of How a Parallel Stream Works

- The split-apply-combine phases are:

  1. Split – Recursively partition a data source into "chunks"

    - Spliterators partition collections in Java

    - Each Java collection has a spliterator


```java
interface Collection<E> {
  ...
  default Spliterator<E> spliterator() {
    return Spliterators.spliterator(this, 0);
  }

  default Stream<E> parallelStream() {
    return StreamSupport.stream(spliterator(), true);
  }
  ...
}
```
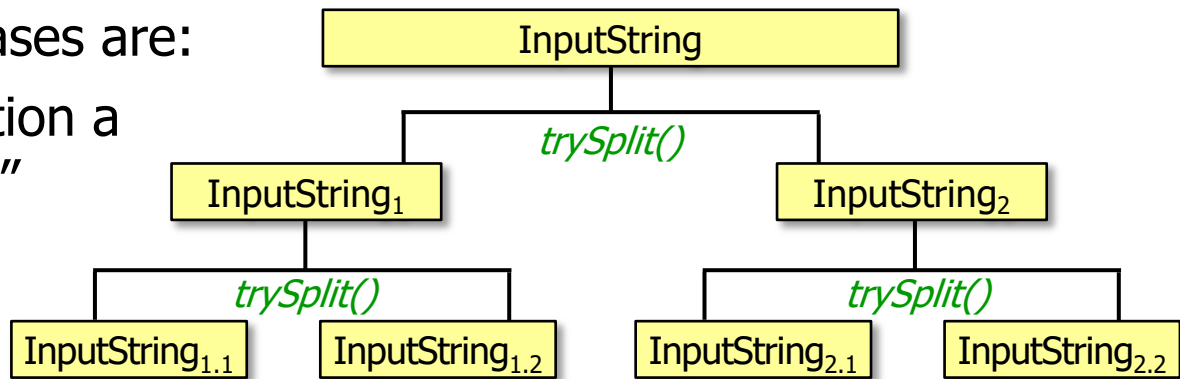
See docs.oracle.com/javase/8/docs/api/java/util/Collection.html

Overview of How a Parallel Stream Works

- The split-apply-combine phases are:

1. **Split** – Recursively partition a data source into "chunks"

  - Spliterators partition collections in Java

  - Each Java collection has a spliterator

  - Programmers can define custom spliterators


```
                    InputString
                   /            \
                trySplit()
        InputString₁              InputString₂
       /          \              /            \
    trySplit()                trySplit()
InputString₁.₁  InputString₁.₂  InputString₂.₁  InputString₂.₂
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/SearchStreamSpliterator
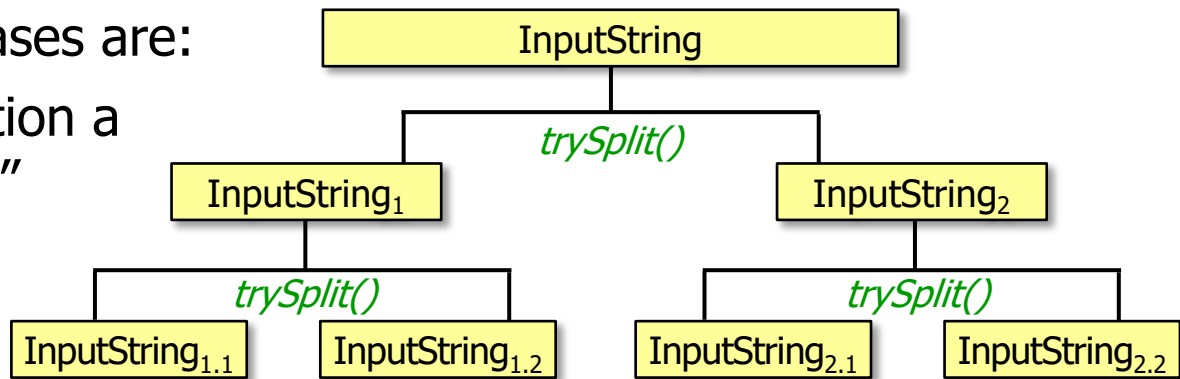
Overview of How a Parallel Stream Works

- The split-apply-combine phases are:

  1. **Split** – Recursively partition a data source into "chunks"

     - Spliterators partition collections in Java

     - Each Java collection has a spliterator

     - Programmers can define custom spliterators

     - Parallel streams perform better on data sources that can be split efficiently & evenly
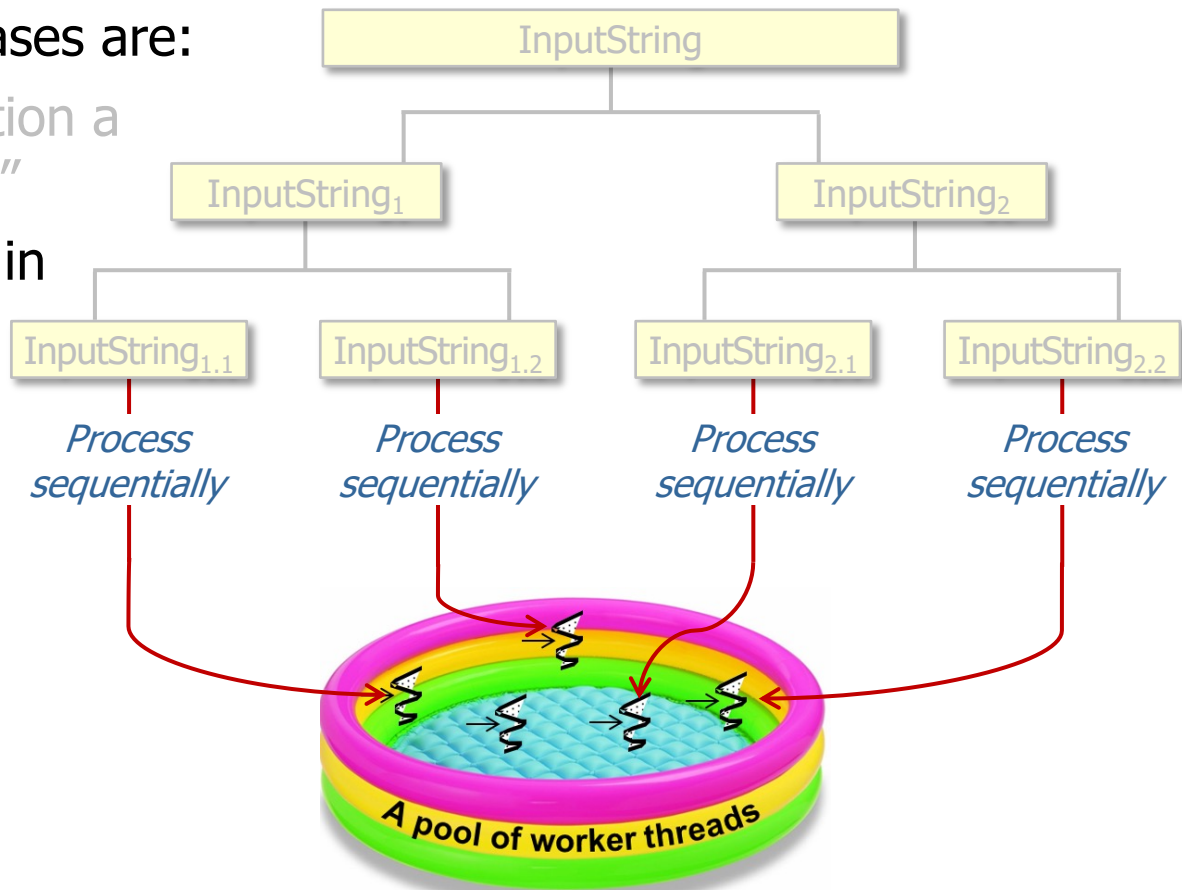
```
                    InputString
                  /            \
            trySplit()
       InputString_1          InputString_2
        /       \              /        \
   trySplit()            trySplit()
InputString_1.1  InputString_1.2  InputString_2.1  InputString_2.2
```

Overview of How a Parallel Stream Works

- The split-apply-combine phases are:

  1. **Split** – Recursively partition a data source into "chunks"

  2. **Apply** – Process chunks in a common thread pool

InputString

InputString$_1$

InputString$_2$

InputString$_{1.1}$

InputString$_{1.2}$

InputString$_{2.1}$

InputString$_{2.2}$

Process sequentially

Process sequentially

Process sequentially

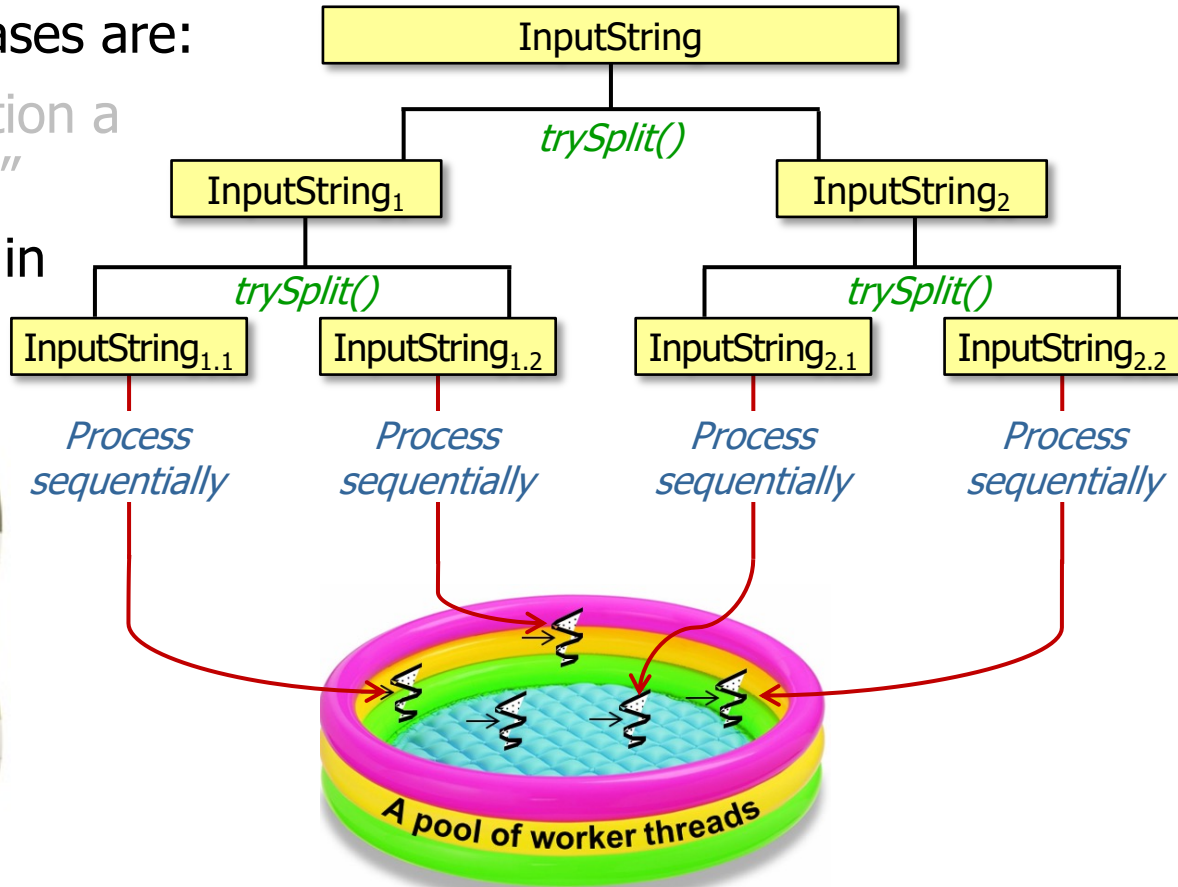*Process sequentially*

A pool of worker threads

See lesson on "*Java Parallel Stream Internals: Parallel Processing via the Common ForkJoinPool*"
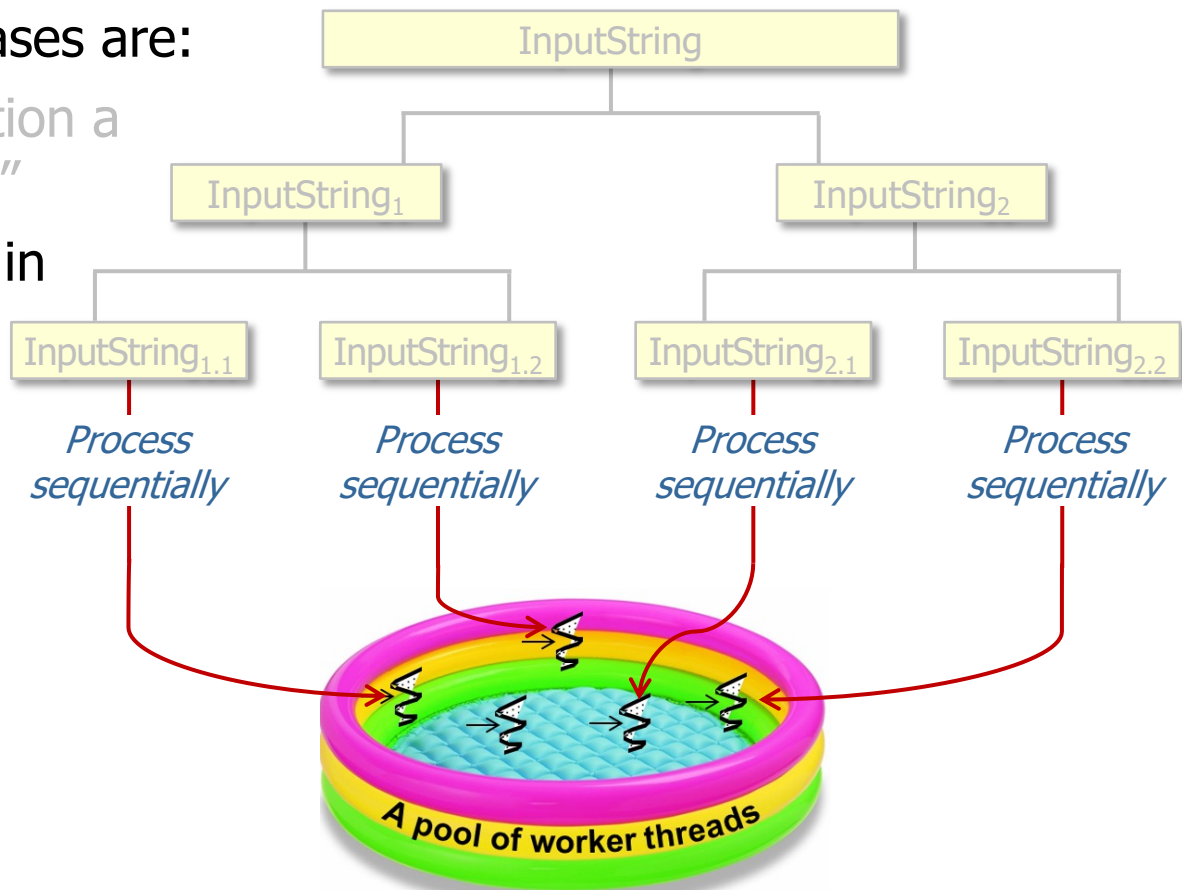
Overview of How a Parallel Stream Works

- The split-apply-combine phases are:

  1. **Split** – Recursively partition a data source into "chunks"

  2. **Apply** – Process chunks in a common thread pool

```
                    ┌─────────────────┐
                    │   InputString   │
                    └─────────────────┘
                         trySplit()
          ┌─────────────────┴─────────────────┐
┌──────────────────┐              ┌──────────────────┐
│  InputString_1   │              │  InputString_2   │
└──────────────────┘              └──────────────────┘
      trySplit()                        trySplit()
  ┌───────┴───────┐              ┌───────┴───────┐
┌──────────────┐ ┌──────────────┐ ┌──────────────┐ ┌──────────────┐
│InputString1.1│ │InputString1.2│ │InputString2.1│ │InputString2.2│
└──────────────┘ └──────────────┘ └──────────────┘ └──────────────┘
   Process          Process          Process          Process
 sequentially     sequentially     sequentially     sequentially
```

$InputString_1$, $InputString_2$, $InputString_{1.1}$, $InputString_{1.2}$, $InputString_{2.1}$, $InputString_{2.2}$

A pool of worker threads

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

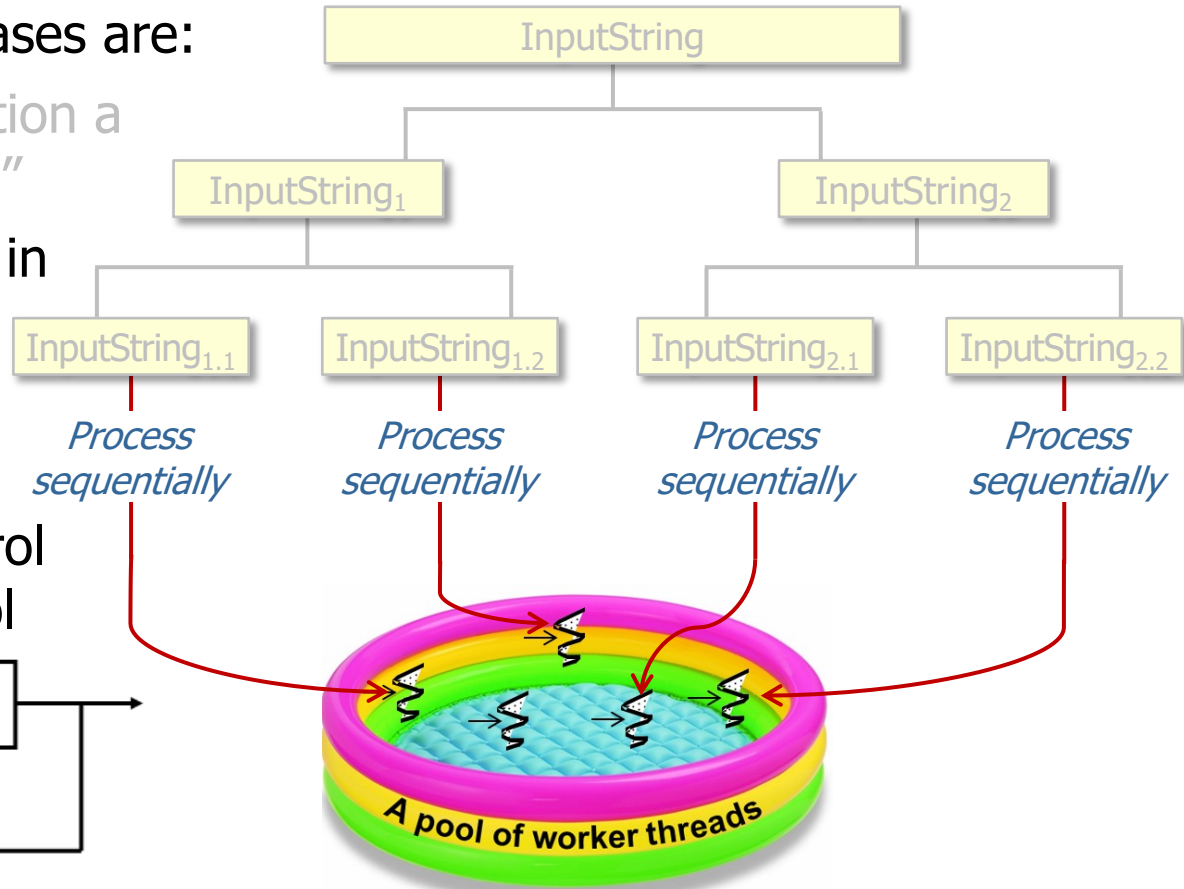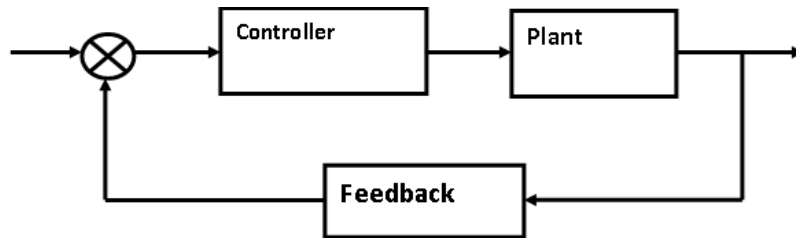# Overview of How a Parallel Stream Works

- The split-apply-combine phases are:

  1. **Split** – Recursively partition a data source into "chunks"

  2. **Apply** – Process chunks in a common thread pool

     - Utilization's maximized via "work-stealing"

InputString

InputString$_1$ InputString$_2$

InputString$_{1.1}$ InputString$_{1.2}$ InputString$_{2.1}$ InputString$_{2.2}$

Process sequentially   *Process sequentially*   *Process sequentially*   *Process sequentially*

A pool of worker threads

See lesson on "*Java Parallel Stream Internals: Mapping onto the Common ForkJoinPool*"
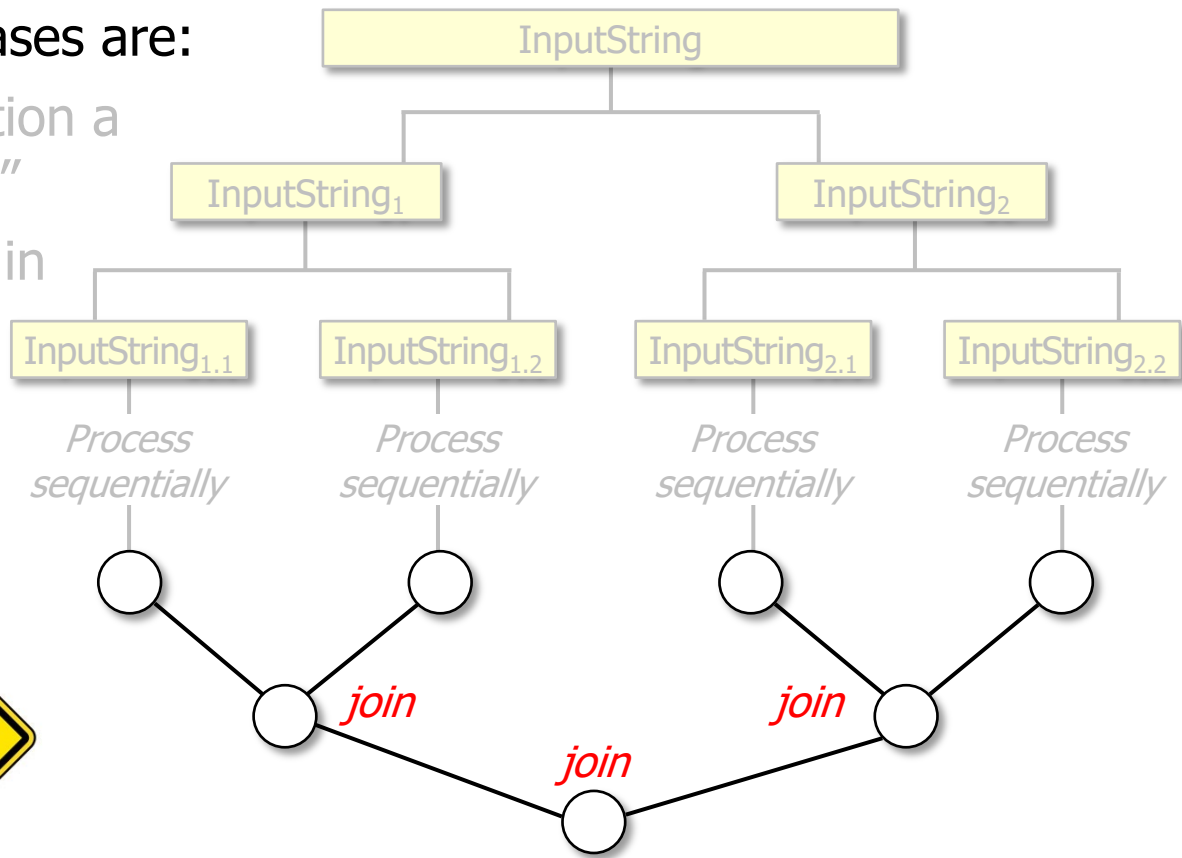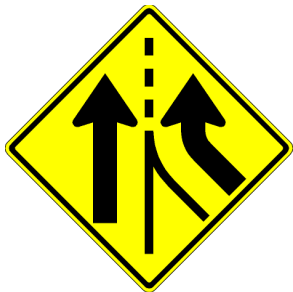
Overview of How a Parallel Stream Works

- The split-apply-combine phases are:

  1. **Split** – Recursively partition a data source into "chunks"

  2. **Apply** – Process chunks in a common thread pool

     - Utilization's maximized via "work-stealing"

     - Programmers can control # of threads in the pool

See lesson on "*Java Parallel Stream Internals: Configuring the Common Fork-Join Pool*"
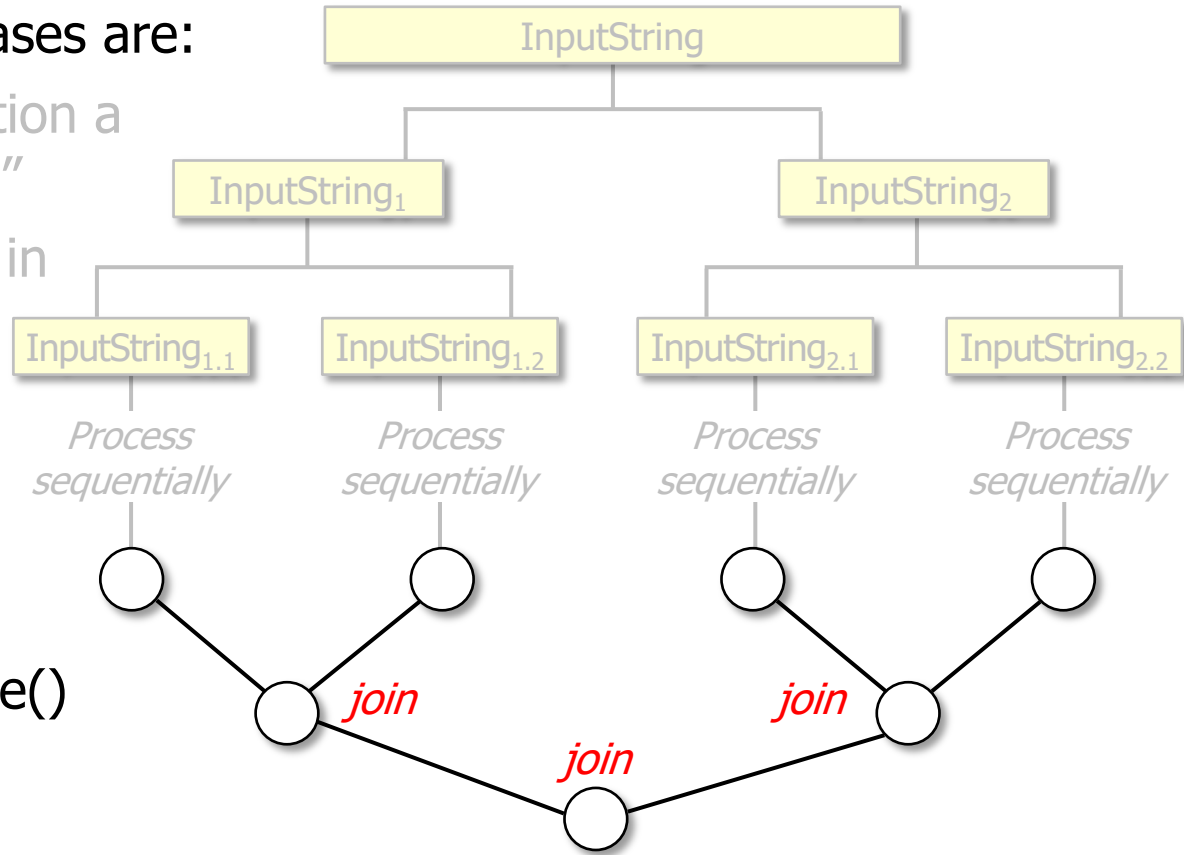
Overview of How a Parallel Stream Works

- The split-apply-combine phases are:

  1. **Split** – Recursively partition a data source into "chunks"

  2. **Apply** – Process chunks in a common thread pool

  3. **Combine** – Join partial results to a single result

InputString

InputString$_1$ InputString$_2$

InputString$_{1.1}$ InputString$_{1.2}$ InputString$_{2.1}$ InputString$_{2.2}$

Process sequentially   *Process sequentially*   *Process sequentially*   *Process sequentially*

join   *join*

join

See upcoming lessons on "*Java Parallel Stream Internals: Combining Results*"
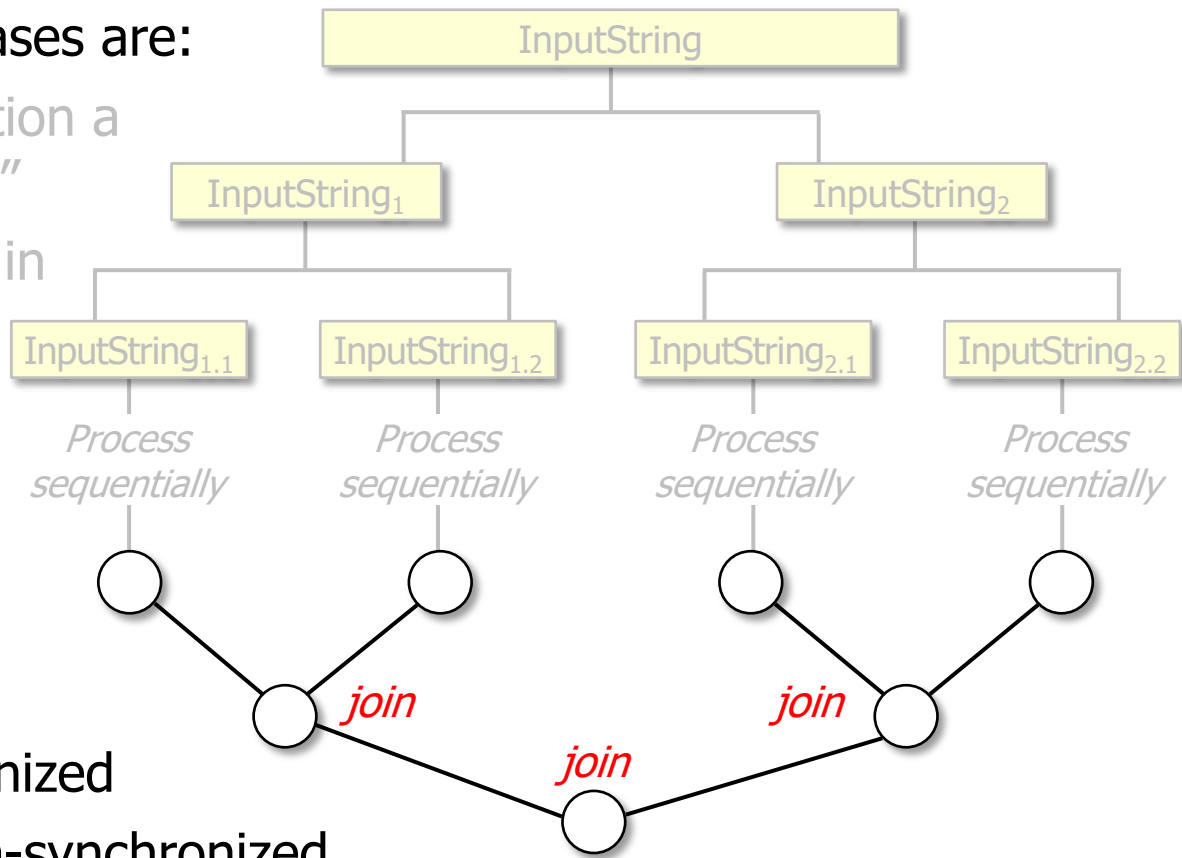
Overview of How a Parallel Stream Works

- The split-apply-combine phases are:

1. **Split** – Recursively partition a data source into "chunks"

2. **Apply** – Process chunks in a common thread pool

3. **Combine** – Join partial results to a single result

   - Performed by terminal operations
     - e.g., collect() & reduce()

InputString

InputString$_1$

InputString$_2$

InputString$_{1.1}$

InputString$_{1.2}$

InputString$_{2.1}$

InputString$_{2.2}$

Process sequentially

Process sequentially

Process sequentially

Process sequentially

join

join

join

join

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

Overview of How a Parallel Stream Works

- The split-apply-combine phases are:

  1. **Split** – Recursively partition a data source into "chunks"

  2. **Apply** – Process chunks in a common thread pool

  3. **Combine** – Join partial results to a single result

     - Performed by terminal operations

     - Collectors can either be

       - Concurrent – synchronized

       - Non-concurrent – non-synchronized

See lessons on "*Java Parallel Stream Internals: Non-Concurrent & Concurrent Collectors*"
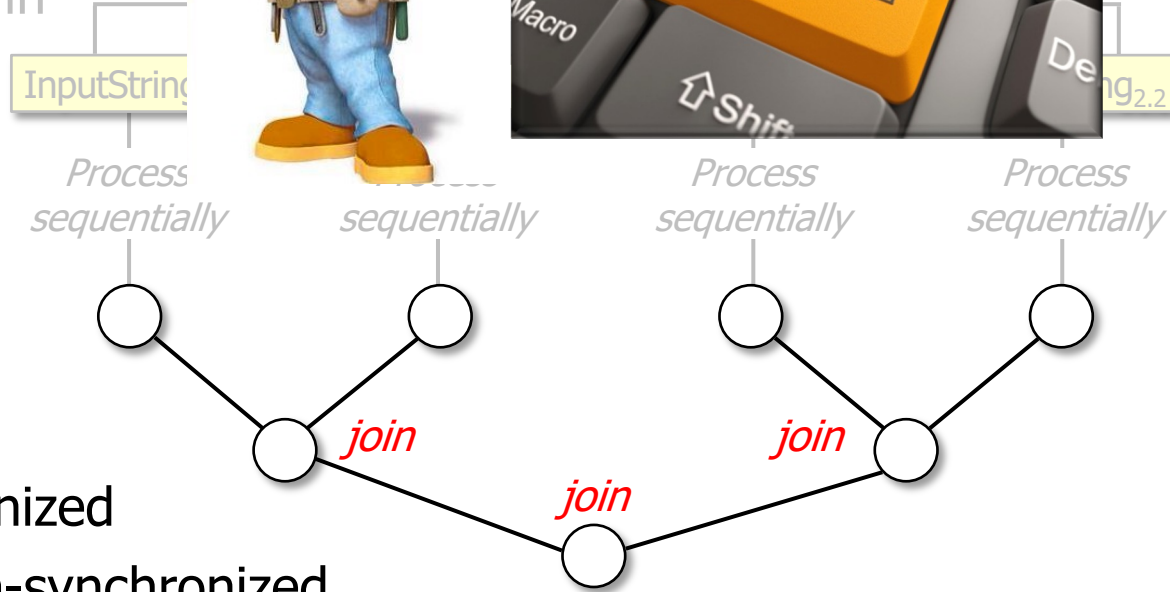
- The split-apply-combine phases are:

  1. **Split** – Recursively partition a data source into "chunks"

  2. **Apply** – Process chunks in a common thread pool

  3. **Combine** – Join partial results to a single result

     - Performed by terminal operations

  - Collectors can either be

    - Concurrent – synchronized

    - Non-concurrent – non-synchronized

InputString

Process sequentially *Process sequentially* *Process sequentially* *Process sequentially*

join *join*

join

Programmers can define custom collectors

End of How Java Parallel Streams Work "Under the Hood"