Overview of Java 8 Parallel Streams (Part 2)

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

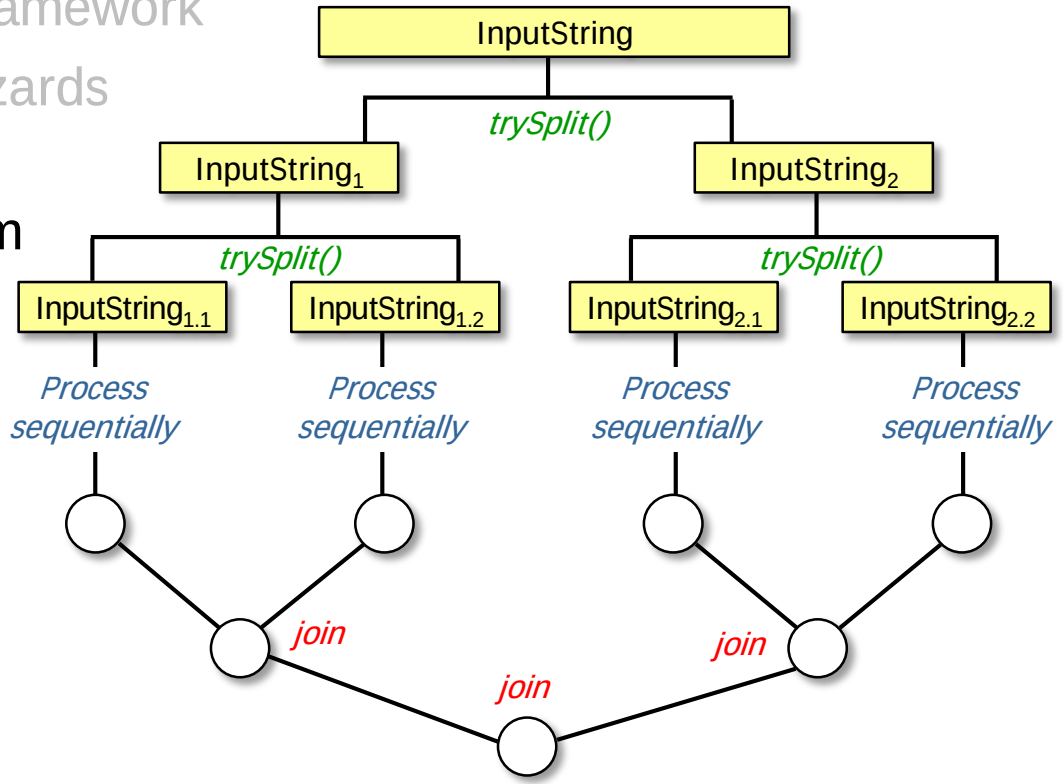
Professor of Computer Science

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
- Be able to avoid concurrency hazards in parallel streams
- Understand how a parallel stream splits its elements recursively, processes them independently, & combines the results

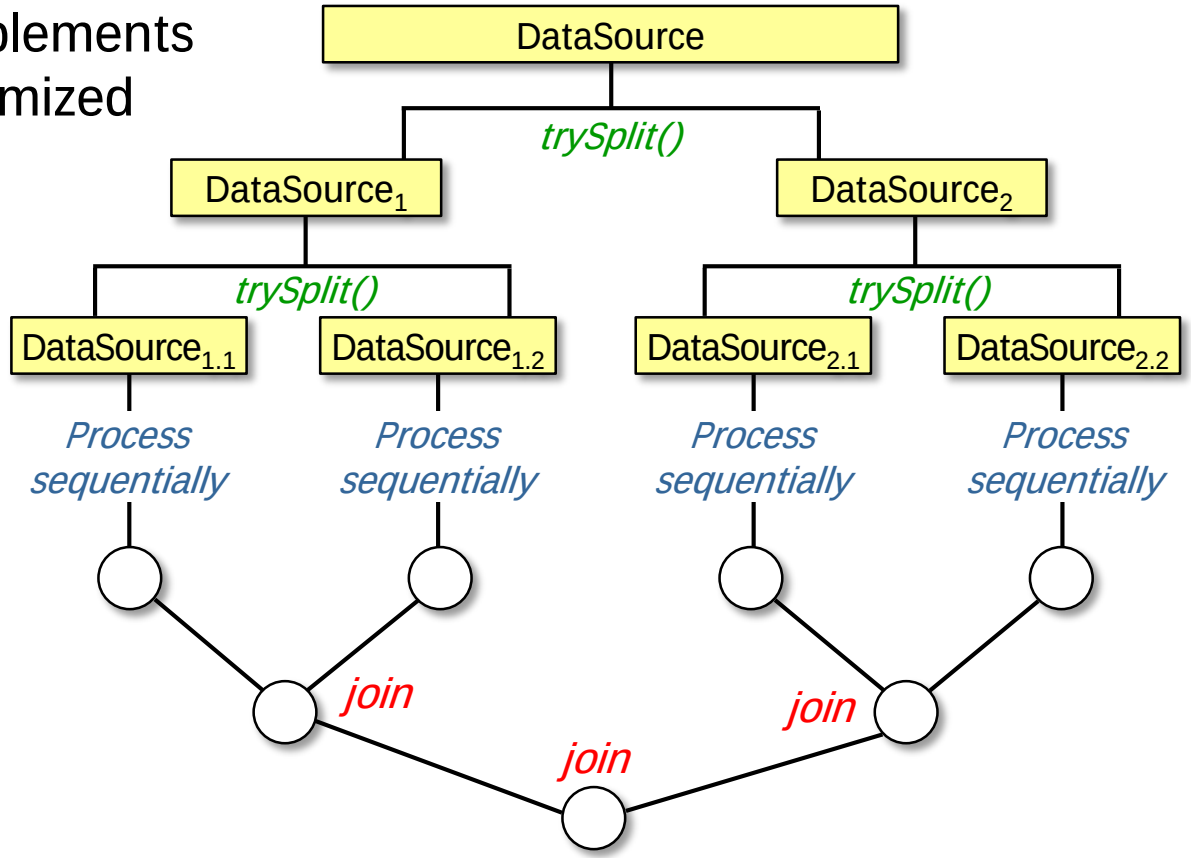


This knowledge of internals will make you a better Java 8 streams programmer!

Inner Workings of a Parallel Stream

Inner Workings of a Parallel Stream

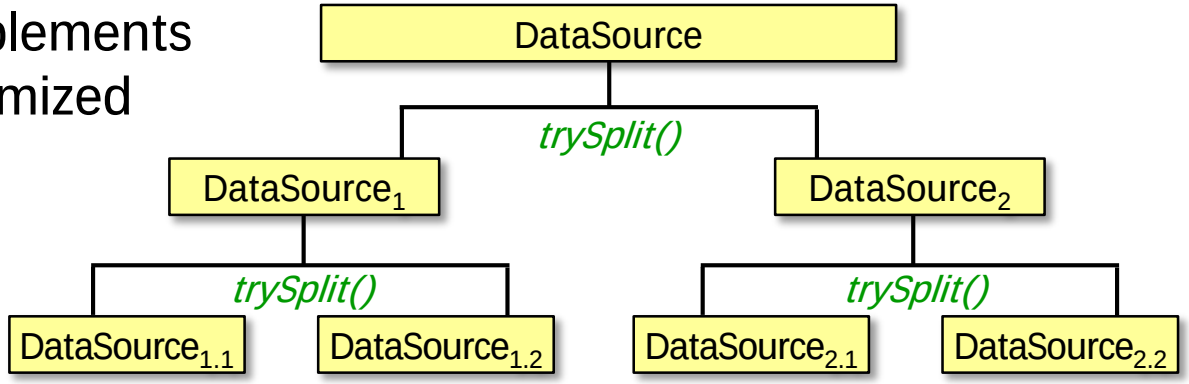
- A Java 8 parallel stream implements a “map/reduce” variant optimized for multi-core processors



See en.wikipedia.org/wiki/MapReduce

Inner Workings of a Parallel Stream

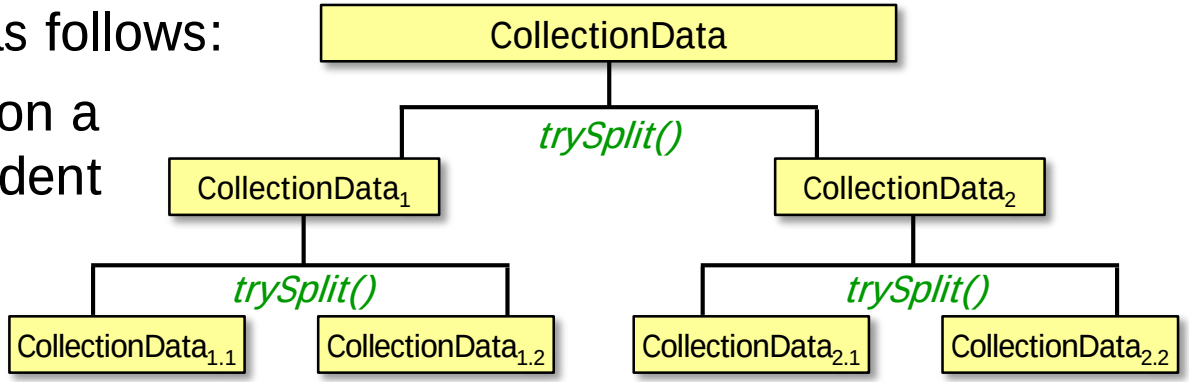
- A Java 8 parallel stream implements a “map/reduce” variant optimized for multi-core processors
  - It’s actually more like the “split-apply-combine” data analysis strategy



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

Inner Workings of a Parallel Stream

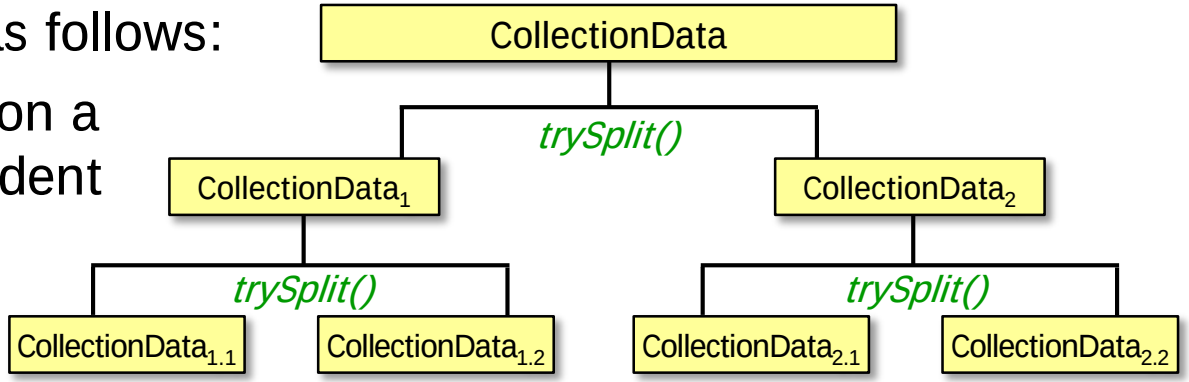
- Split-apply-combine works as follows:
  1. Split – Recursively partition a data source into independent "chunks"



See en.wikipedia.org/wiki/Divide_and_conquer_algorithm

Inner Workings of a Parallel Stream

- 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

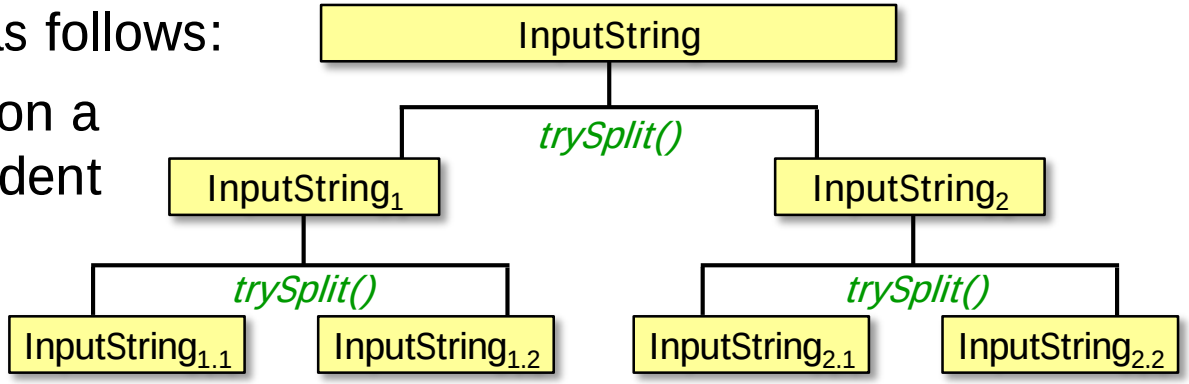


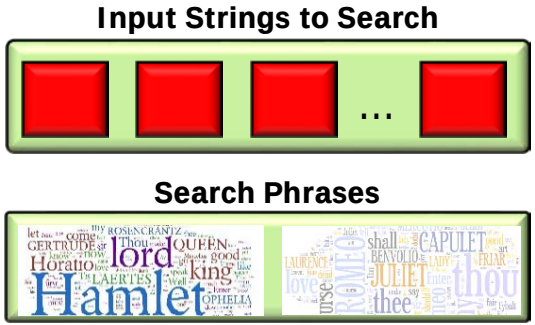

```
public interface Spliterator<T> {

   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

Inner Workings of a Parallel Stream

- 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

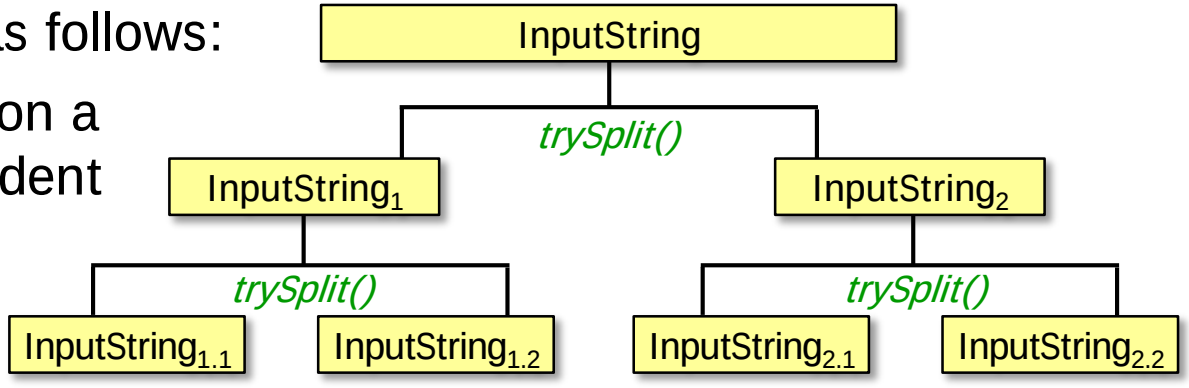




See github.com/douglascraigschmidt/LiveLessons/tree/master/SearchStreamSpliterator

Inner Workings of a Parallel Stream

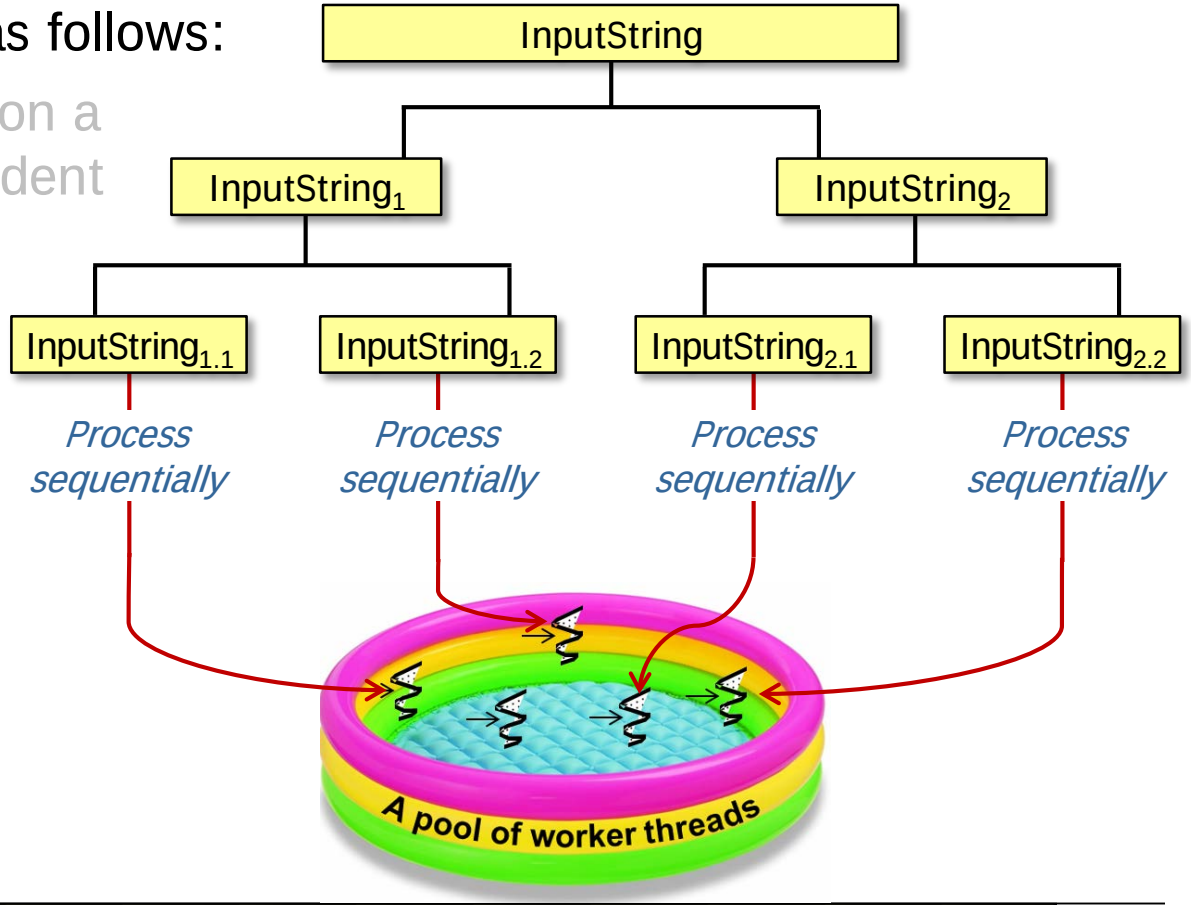
- Split-apply-combine works as follows:
  1. Split – Recursively partition a data source into independent "chunks"
     - Spliterators are defined to partition collections in Java 8
     - You can also define custom spliterators
     - Parallel streams perform better on data sources that can be split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



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

Inner Workings of a Parallel Stream

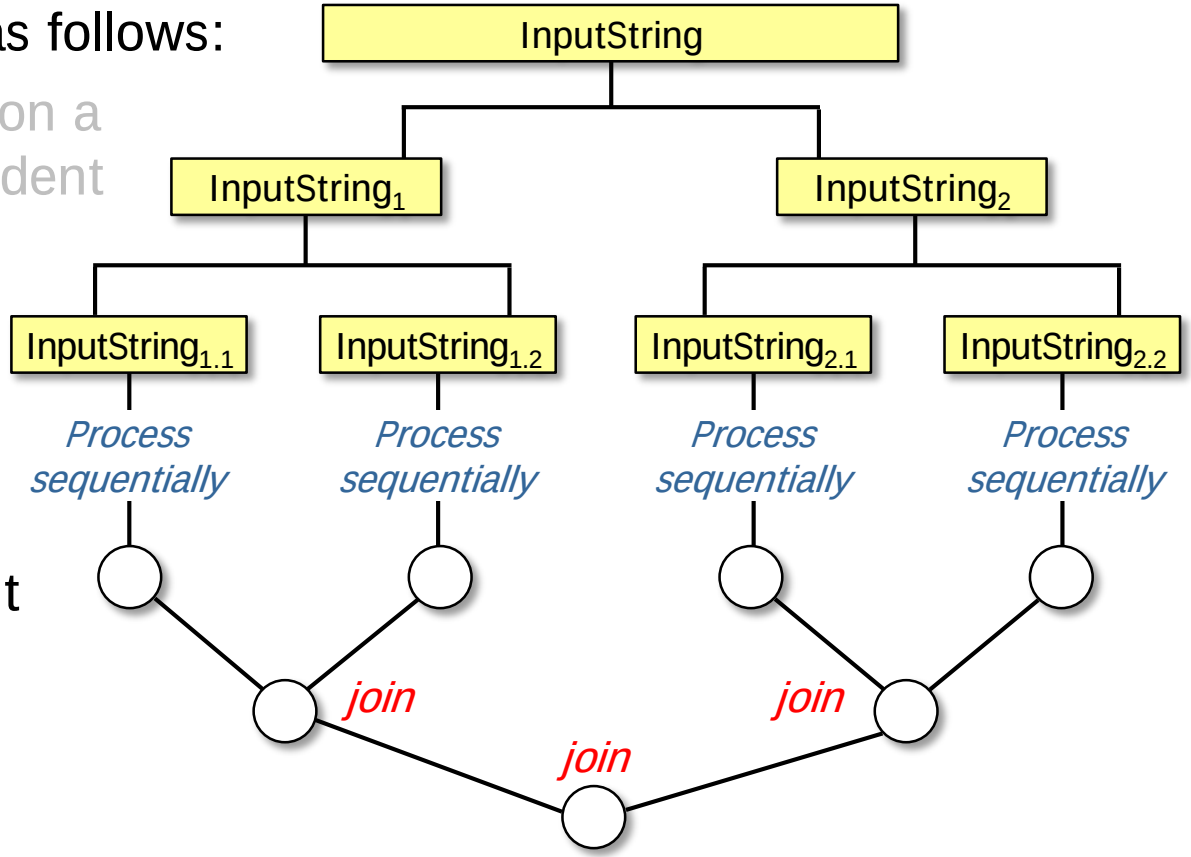
- Split-apply-combine works as follows:
  1. Split – Recursively partition a data source into independent "chunks"
  2. Apply – Process chunks independently in a pool of threads



In practice, splitting & applying *run* simultaneously, not sequentially

Inner Workings of a Parallel Stream

- 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

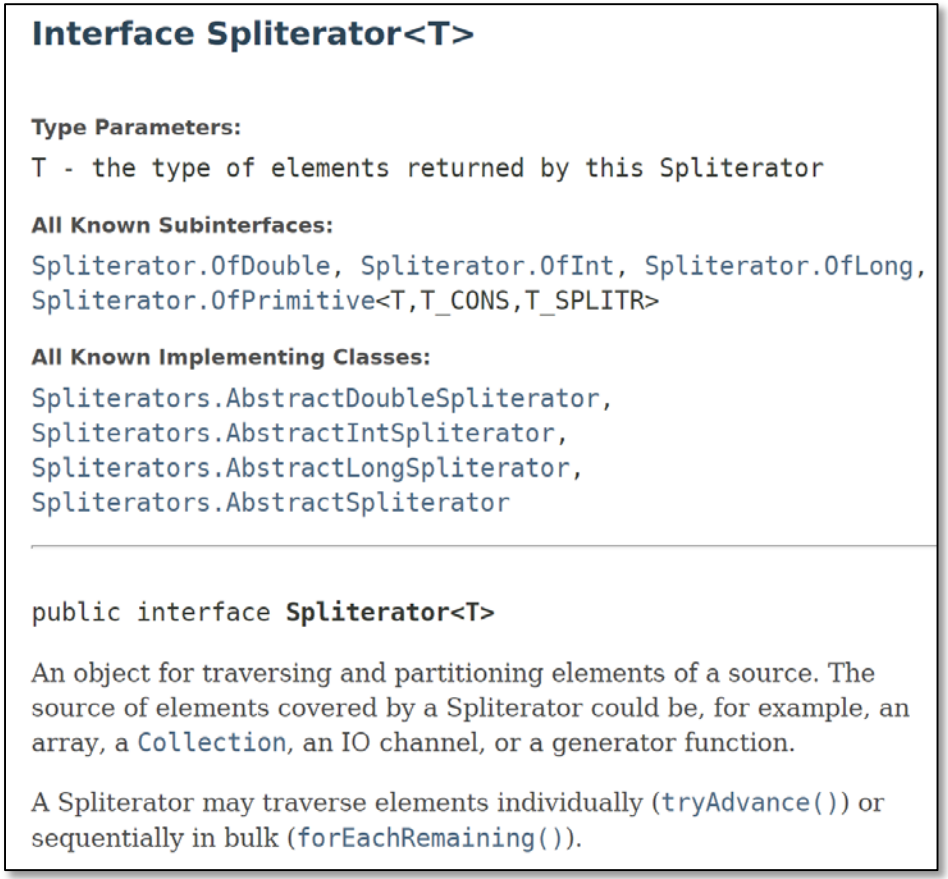


Combining is performed by terminal operations, such as collect() & reduce()

Partitioning a Parallel Stream

Partitioning a Parallel Stream

- A “splittable iterator” (spliterator) partitions a Java 8 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

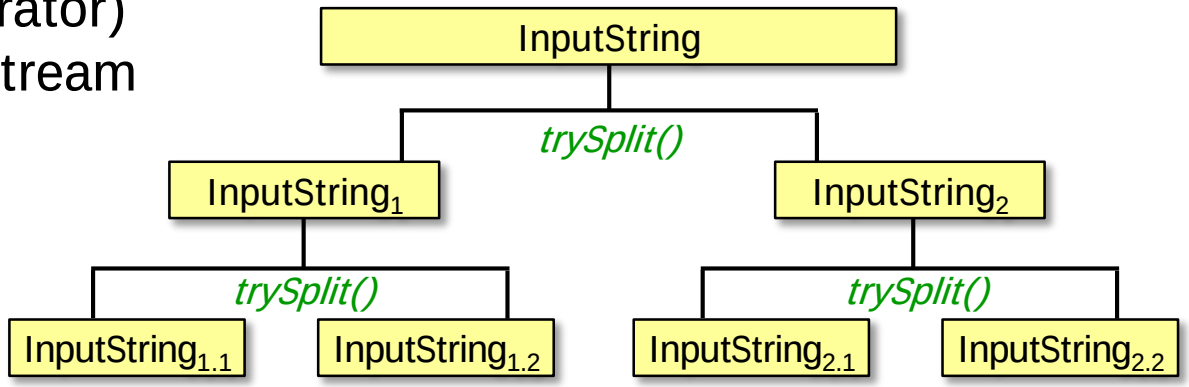
- A “splittable iterator” (spliterator) partitions a Java 8 parallel stream into chunks
  - We showed earlier how a spliterator can *traverse* elements in a source

```
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 *“Overview of Java Streams (Part 3)”*

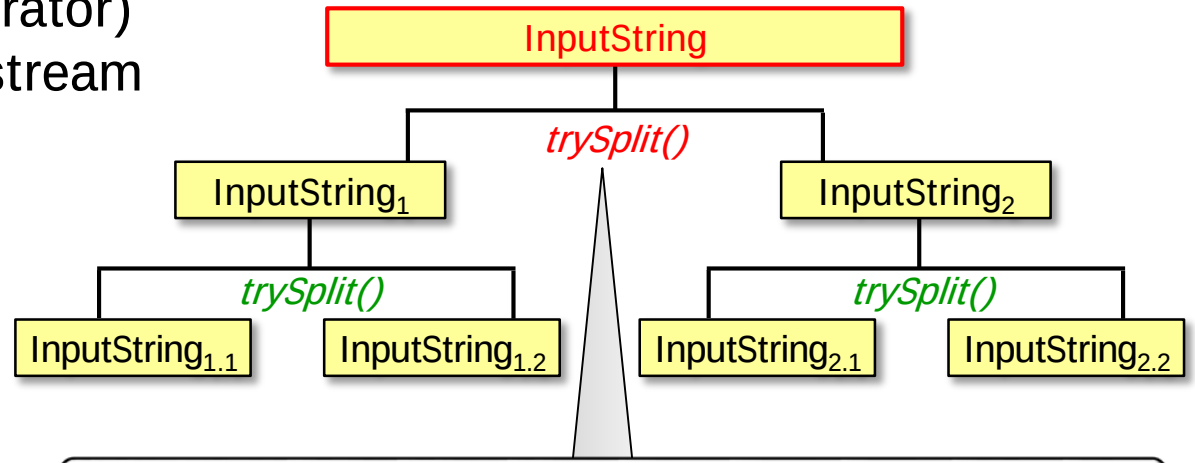
Partitioning a Parallel Stream

- A “splittable iterator” (spliterator) partitions a Java 8 parallel stream into chunks
  - We showed earlier how a spliterator can *traverse* elements in a source
  - We now outline how a parallel spliterator can *partition* all elements in a source



Partitioning a Parallel Stream

- A “splittable iterator” (spliterator) partitions a Java 8 parallel stream into chunks
  - We showed earlier how a spliterator can *traverse* elements in a source
  - We now outline how a parallel spliterator can *partition* all elements in a source

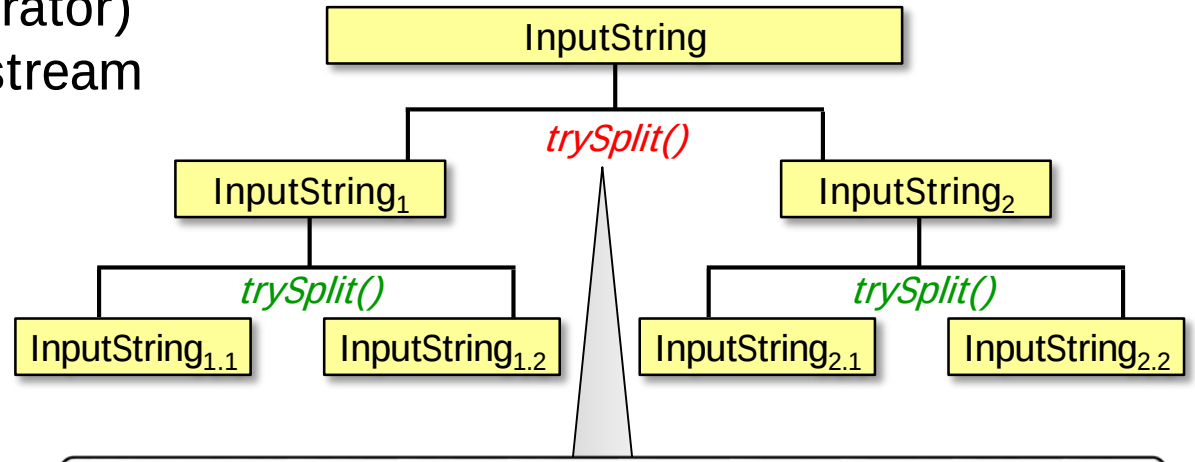



```
Spliterator<T> trySplit() {
  if (input is below minimum size)
    return null
  else {
    split input in 2 (even-sized) chunks
    return a spliterator for “left chunk”
  }
```

This partitioning is done via a spliterator’s trySplit() method

Partitioning a Parallel Stream

- A “splittable iterator” (spliterator) partitions a Java 8 parallel stream into chunks
  - We showed earlier how a spliterator can *traverse* elements in a source
  - We now outline how a parallel spliterator can *partition* all elements in a source

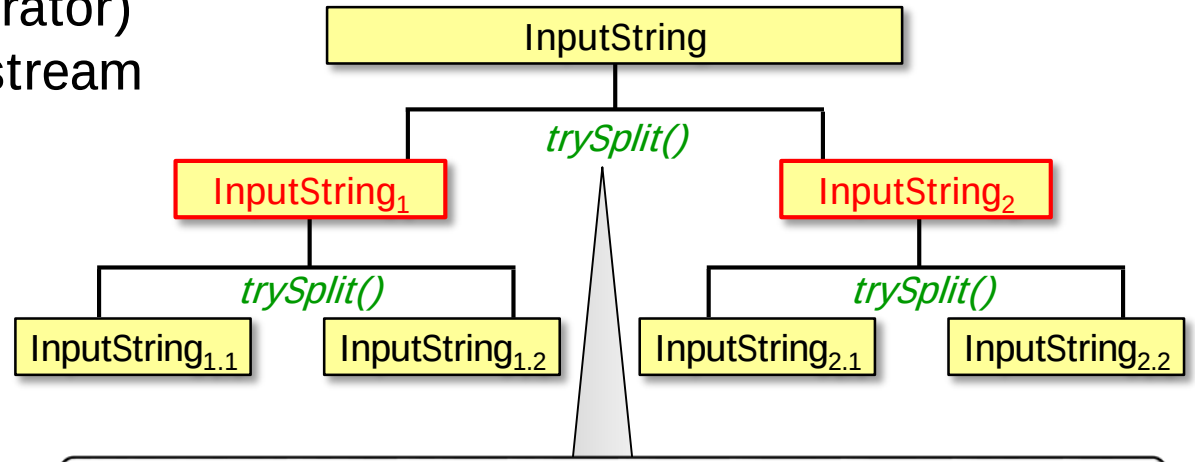



```
Spliterator<T> trySplit() {
  if (input is below minimum size)
    return null
  else {
    split input in 2 (even-sized) chunks
    return a spliterator for “left chunk”
  }
```

When null is returned the streams framework processes this chunk sequentially

Partitioning a Parallel Stream

- A “splittable iterator” (spliterator) partitions a Java 8 parallel stream into chunks
  - We showed earlier how a spliterator can *traverse* elements in a source
  - We now outline how a parallel spliterator can *partition* all elements in a source

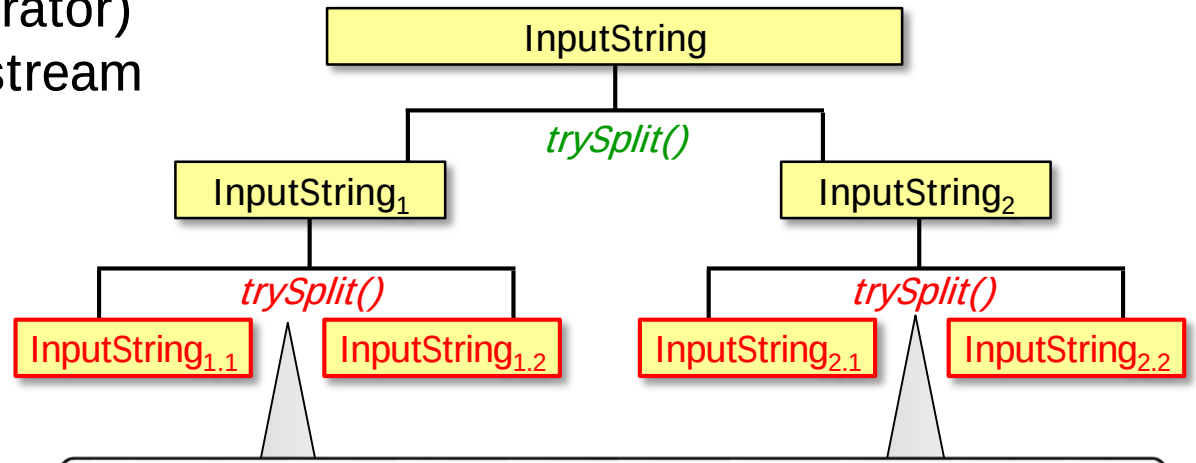



```
Spliterator<T> trySplit() {
  if (input is below minimum size)
    return null
  else {
    split input in 2 (even-sized) chunks
    return a spliterator for “left chunk”
  }
```

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

Partitioning a Parallel Stream

- A “splittable iterator” (spliterator) partitions a Java 8 parallel stream into chunks
  - We showed earlier how a spliterator can *traverse* elements in a source
  - We now outline how a parallel spliterator can *partition* all elements in a source

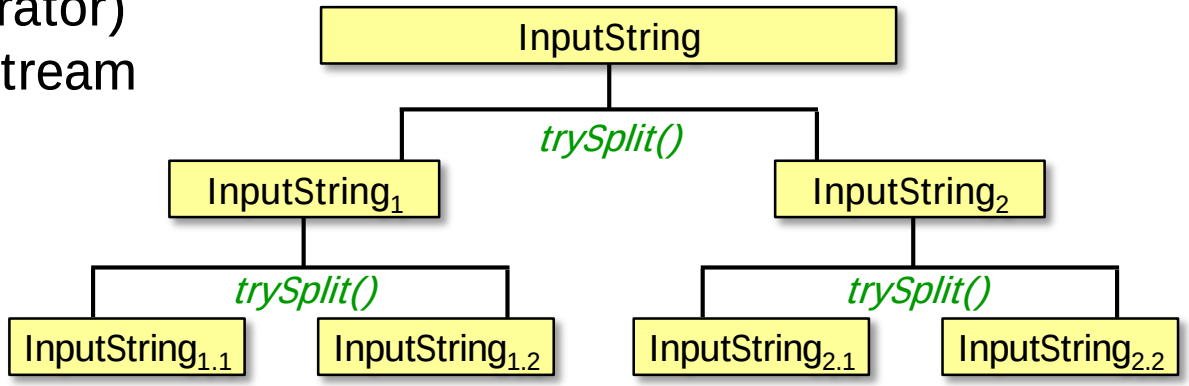


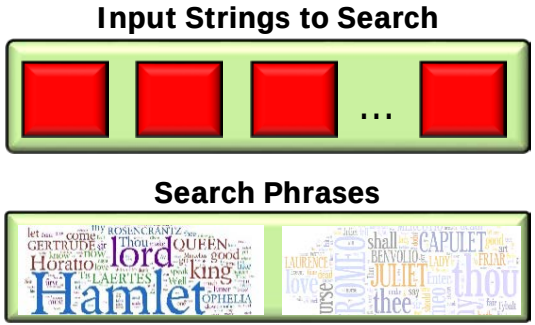

```
Spliterator<T> trySplit() {
  if (input is below minimum size)
    return null
  else {
    split input in 2 (even-sized) chunks
    return a spliterator for “left chunk”
  }
```

trySplit() is called recursively until all chunks are below the minimize size

Partitioning a Parallel Stream

- A "splittable iterator" (spliterator) partitions a Java 8 parallel stream into chunks
  - We showed earlier how a spliterator can *traverse* elements in a source
  - We now outline how a parallel spliterator can *partition* all elements in a source
  - More parallel spliterator implementation details are covered shortly



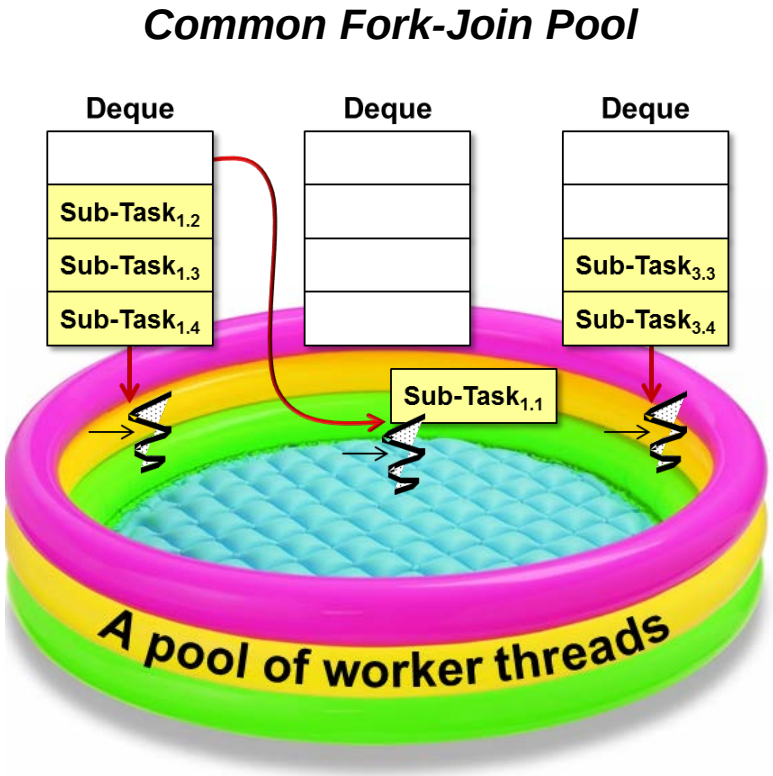


See *"Java 8 Parallel SearchStreamGang Example (Part 2)"*

Processing Chunks in a Parallel Stream

Processing Chunks in a Parallel Stream

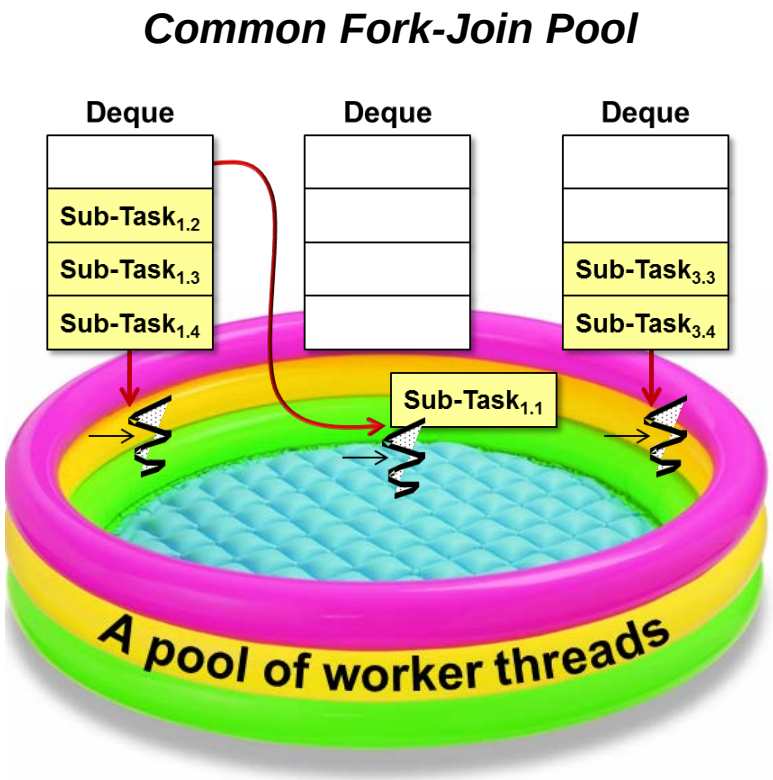
- The chunks created by a spliterator are processed in a common fork-join pool



See gee.cs.oswego.edu/dl/papers/fj.pdf

Processing Chunks in a Parallel Stream

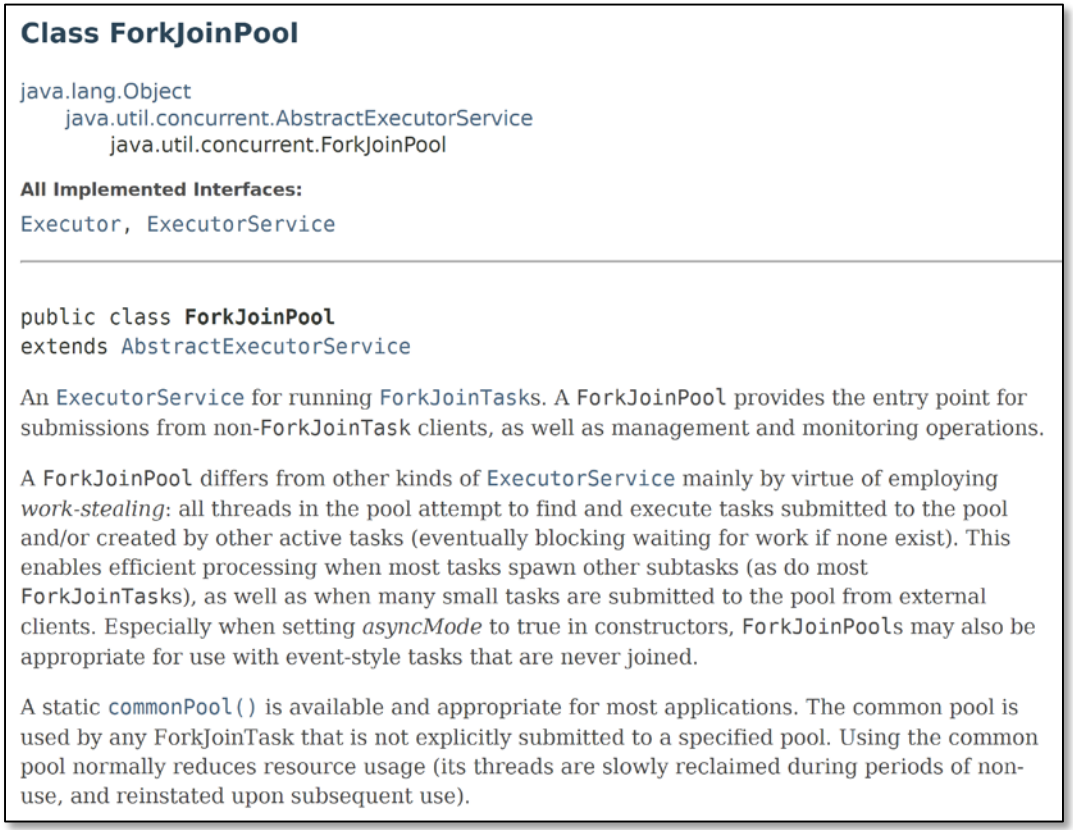
- The chunks created by a spliterator are processed in a common fork-join pool
  - All parallel streams in a process share this common fork-join pool



See gee.cs.oswego.edu/dl/papers/fj.pdf

Processing Chunks in a Parallel Stream

- ForkJoinPool is an Executor Service implementation that runs ForkJoinTasks



Class ForkJoinPool

java.lang.Object
    java.util.concurrent.AbstractExecutorService
        java.util.concurrent.ForkJoinPool

All Implemented Interfaces:

Executor, ExecutorService

```
public class ForkJoinPool
extends AbstractExecutorService
```

An ExecutorService for running ForkJoinTasks. A ForkJoinPool provides the entry point for submissions from non-ForkJoinTask clients, as well as management and monitoring operations.

A ForkJoinPool differs from other kinds of ExecutorService mainly by virtue of employing *work-stealing*: all threads in the pool attempt to find and execute tasks submitted to the pool and/or created by other active tasks (eventually blocking waiting for work if none exist). This enables efficient processing when most tasks spawn other subtasks (as do most ForkJoinTasks), as well as when many small tasks are submitted to the pool from external clients. Especially when setting *asyncMode* to true in constructors, ForkJoinPools may also be appropriate for use with event-style tasks that are never joined.

A static commonPool() is available and appropriate for most applications. The common pool is used by any ForkJoinTask that is not explicitly submitted to a specified pool. Using the common pool normally reduces resource usage (its threads are slowly reclaimed during periods of non-use, and reinstated upon subsequent use).

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

Processing Chunks in a Parallel Stream

- ForkJoinPool is an Executor Service implementation that runs ForkJoinTasks
  - It provides the entry point for submissions from non-ForkJoinTask clients

void	execute(ForkJoinTask<T>) – Arrange async execution
T	invoke(ForkJoinTask<T>) – Performs the given task, returning its result upon completion
ForkJoinTask<T>	submit(ForkJoinTask) – Submits a ForkJoinTask for execution, returns a future

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

Processing Chunks in a Parallel Stream

- ForkJoinPool is an Executor Service implementation that runs ForkJoinTasks
  - It provides the entry point for submissions from non-ForkJoinTask clients
  - It also provides management & monitoring operations

int	getParallelism() – Returns the targeted parallelism level of this pool
int	getPoolSize() – Returns the number of worker threads that have started but not yet terminated
int	getQueuedSubmissionCount() – Returns an estimate of the number of tasks submitted to this pool that have not yet begun executing
long	getStealCount() – Returns an estimate of the total number of tasks stolen from one thread's work queue by another

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

Processing Chunks in a Parallel Stream

- A ForkJoinTask is a chunk of data along with functionality on that data

Class ForkJoinTask<V>

java.lang.Object
    java.util.concurrent.ForkJoinTask<V>

All Implemented Interfaces:

`Serializable, Future<V>`

Direct Known Subclasses:

`CountedCompleter, RecursiveAction, RecursiveTask`

```
public abstract class ForkJoinTask<V>
extends Object
implements Future<V>, Serializable
```

Abstract base class for tasks that run within a `ForkJoinPool`. A `ForkJoinTask` is a thread-like entity that is much lighter weight than a normal thread. Huge numbers of tasks and subtasks may be hosted by a small number of actual threads in a ForkJoinPool, at the price of some usage limitations.

A "main" `ForkJoinTask` begins execution when it is explicitly submitted to a `ForkJoinPool`, or, if not already engaged in a ForkJoin computation, commenced in the `ForkJoinPool.commonPool()` via `fork()`, `invoke()`, or related methods. Once started, it will usually in turn start other subtasks. As indicated by the name of this class, many programs using `ForkJoinTask` employ only methods `fork()` and `join()`, or derivatives such as `invokeAll`. However, this class also provides a number of other methods that can come into play in advanced usages, as well as extension mechanics that allow support of new forms of fork/join processing.

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

Processing Chunks in a Parallel Stream

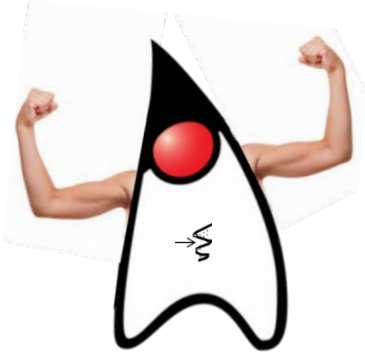
- A ForkJoinTask is a chunk of data along with functionality on that data
  - It defines two primary methods

ForkJoinTask <T>	fork() – Arranges to asynchronously execute this task in the appropriate pool
V	join() – Returns the result of the computation when it is done

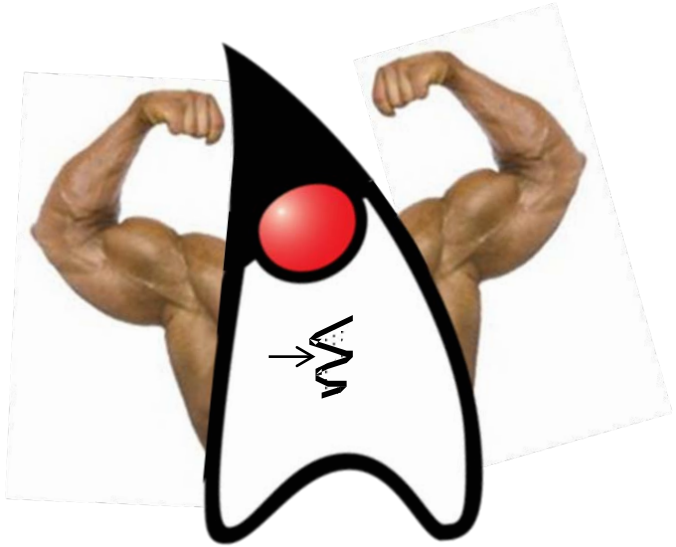
See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinTask.html

Processing Chunks in a Parallel Stream

- A ForkJoinTask is a chunk of data along with functionality on that data
  - It defines two primary methods
  - A ForkJoinTask is lighter weight than a Java thread



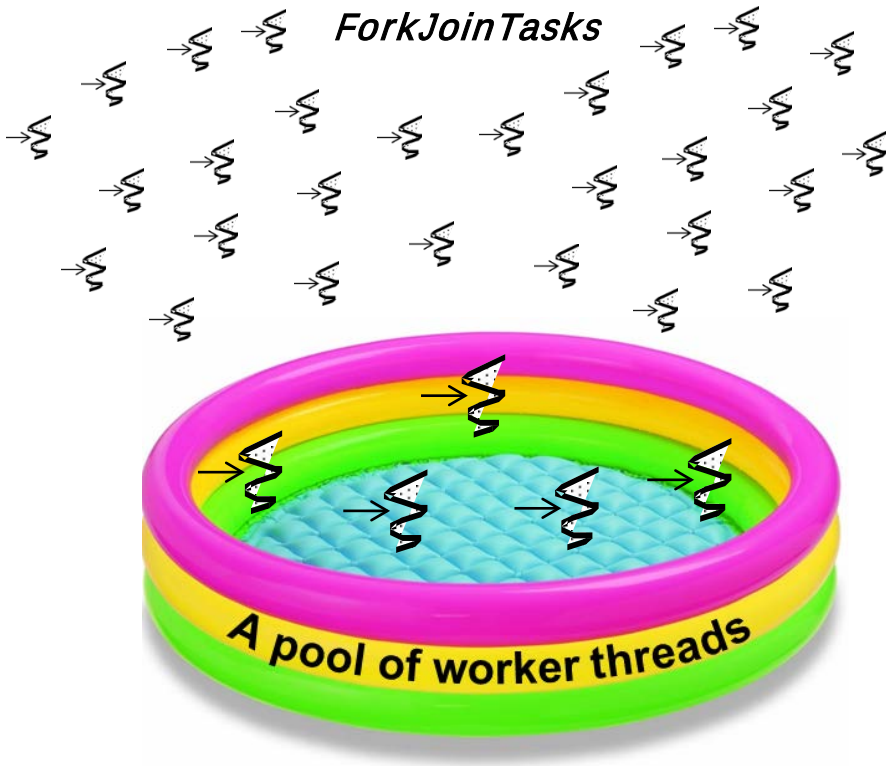
ForkJoinTask



Thread

Processing Chunks in a Parallel Stream

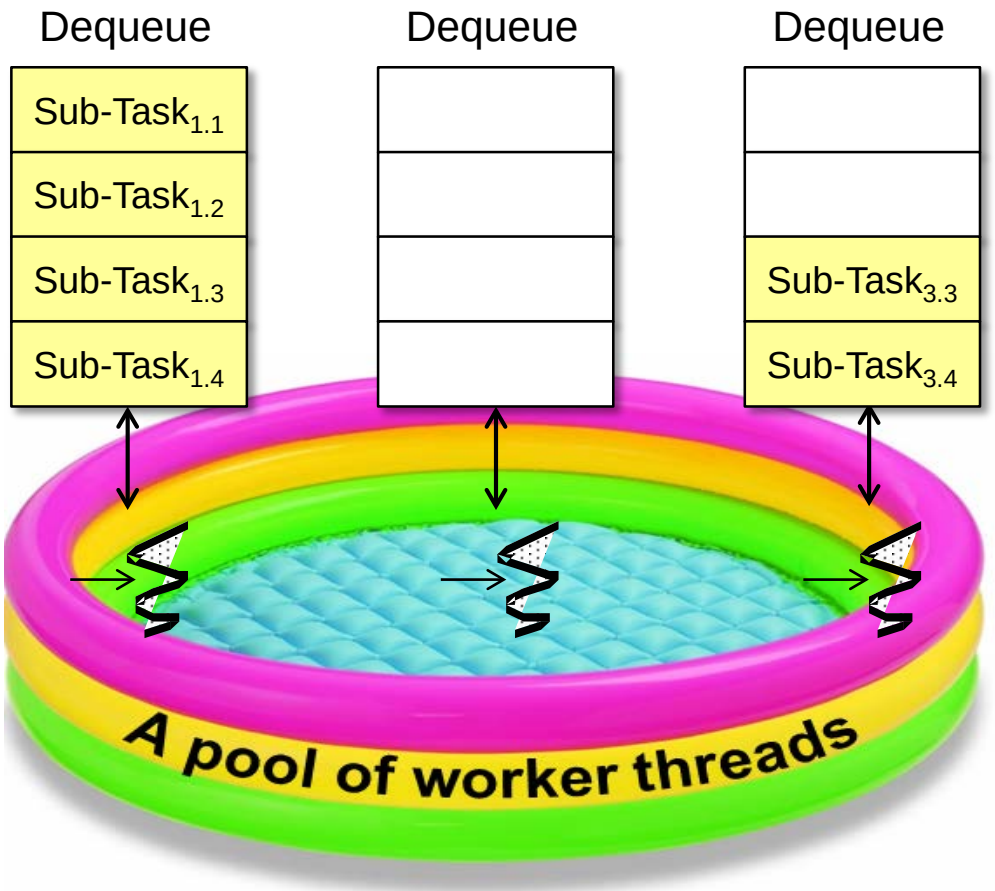
- A ForkJoinTask is a chunk of data along with functionality on that data
  - It defines two primary methods
  - A ForkJoinTask is lighter weight than a Java thread
  - A large # of ForkJoinTasks can run in a small # of Java threads in a ForkJoinPool



See www.infoq.com/interviews/doug-lea-fork-join

Processing Chunks in a Parallel Stream

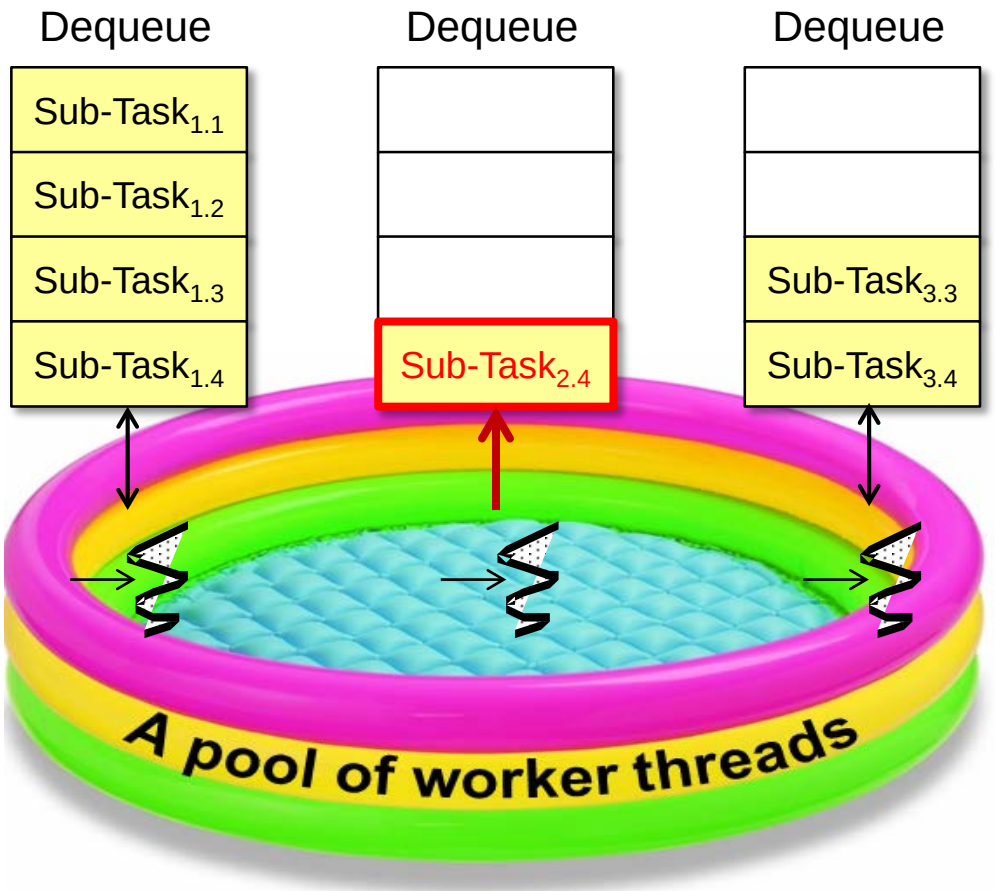
- A circular dequeue is associated with each ForkJoinPool thread



See en.wikipedia.org/wiki/Double-ended_queue

Processing Chunks in a Parallel Stream

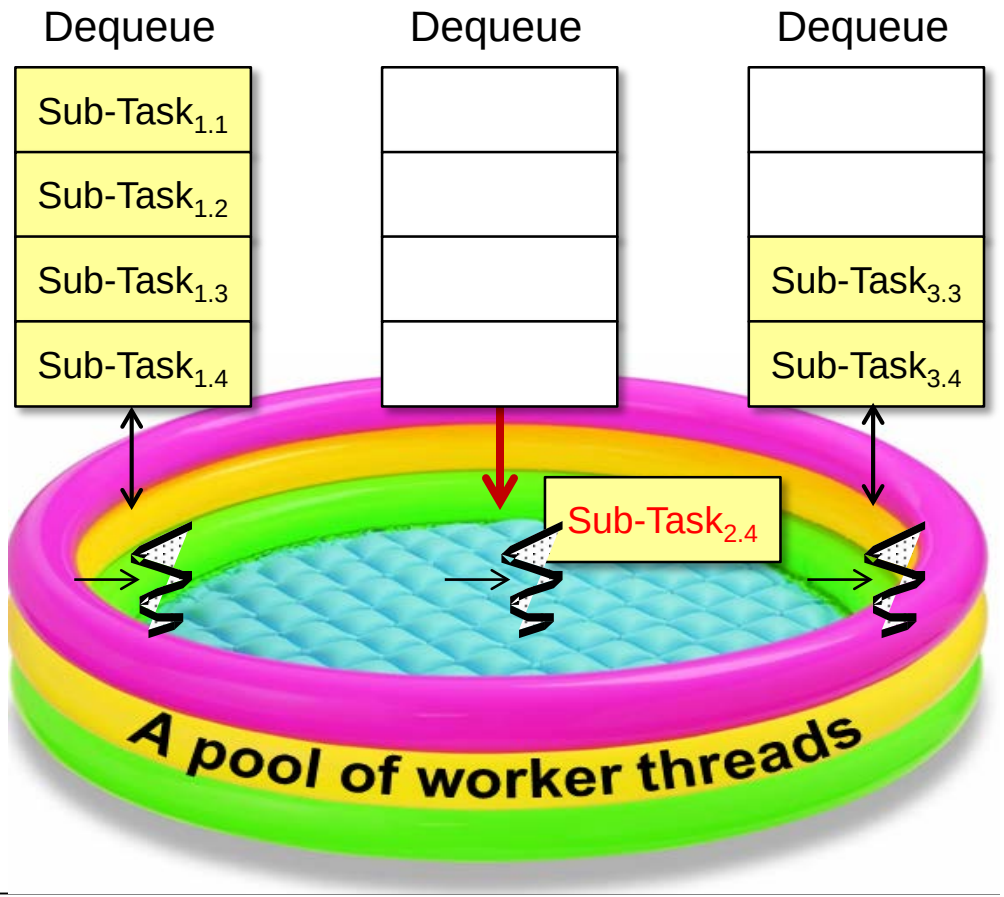
- A circular dequeue is associated with each ForkJoinPool thread
  - fork() pushes a new task to the head of its dequeue



See gee.cs.oswego.edu/dl/papers/fj.pdf

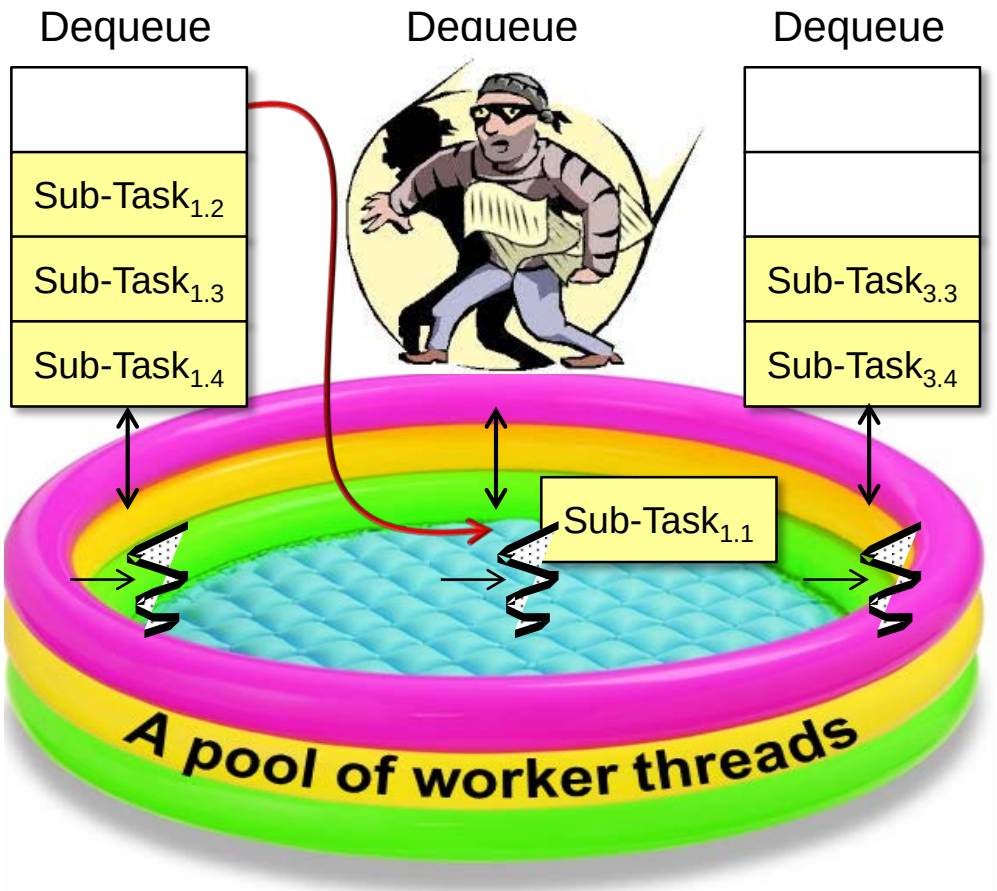
Processing Chunks in a Parallel Stream

- A circular dequeue is associated with each ForkJoinPool thread
  - fork() pushes a new task to the head of its dequeue
  - Likewise, a thread pops the next task its processes from the head of its dequeue



Processing Chunks in a Parallel Stream

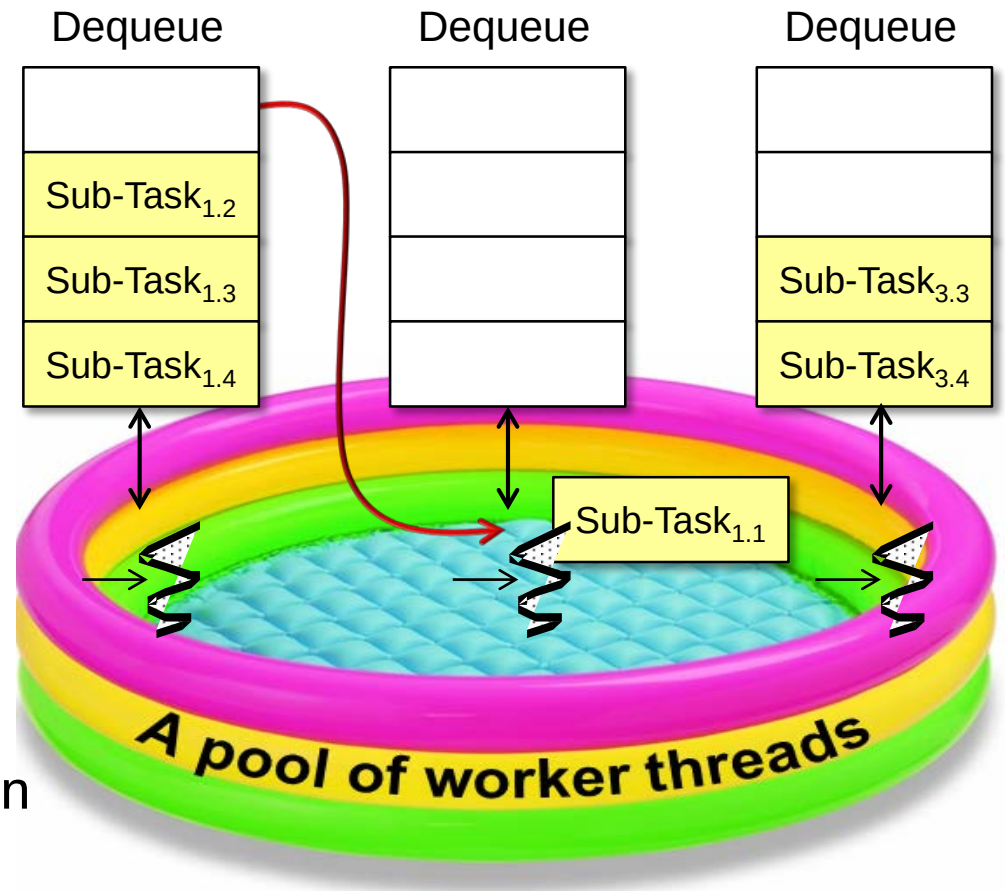
- A circular dequeue is associated with each ForkJoinPool thread
  - fork() pushes a new task to the head of its dequeue
  - Likewise, a thread pops the next task its processes from the head of its dequeue
  - An idle thread “steals” work from the tail of a busy thread to maximize core utilitization



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

Processing Chunks in a Parallel Stream

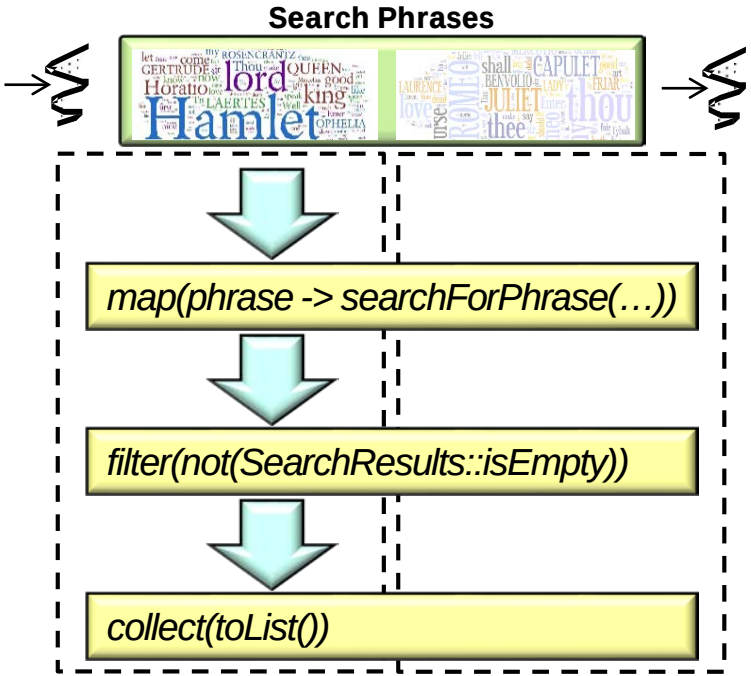
- A circular dequeue is associated with each ForkJoinPool thread
  - fork() pushes a new task to the head of its dequeue
  - Likewise, a thread pops the next task its processes from the head of its dequeue
  - An idle thread "steals" work from the tail of a busy thread to maximize core utilitization
    - The circular dequeue used for work-stealing lowers contention

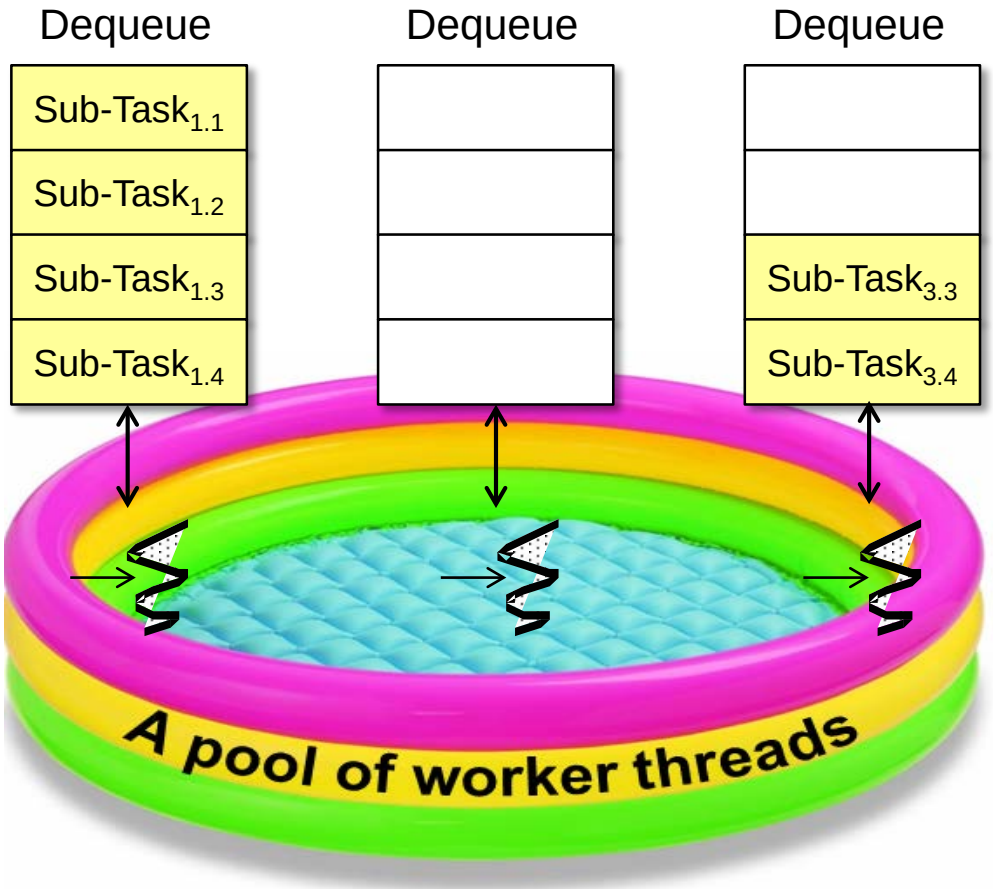


See www.dre.vanderbilt.edu/~schmidt/PDF/work-stealing-dequeue.pdf

Processing Chunks in a Parallel Stream

- Parallel streams is a “user friendly” ForkJoinPool façade





See espressoprogrammer.com/fork-join-vs-parallel-stream-java-8

Processing Chunks in a Parallel Stream

- Parallel streams is a “user friendly” ForkJoinPool façade
  - You can program directly to the ForkJoinPool API, though it can be somewhat tedious!

```
List<List<SearchResults>>
  listOfListOfSearchResults =
    ForkJoinPool
      .commonPool()
      .invoke(new
         SearchWithForkJoinTask
           (inputList,
            mPhrasesToFind,
            ...));
```

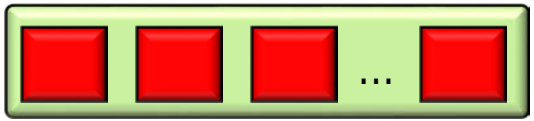
Processing Chunks in a Parallel Stream

- Parallel streams is a “user friendly” ForkJoinPool façade
  - You can program directly to the ForkJoinPool API, though it can be somewhat tedious!

Use the common fork-join pool to search input strings for phrases that match

```
List<List<SearchResults>>
  listOfListOfSearchResults =
    ForkJoinPool
      .commonPool()
      .invoke(new
        SearchWithForkJoinTask
          (inputList,
           mPhrasesToFind,
           ...));
```

Input Strings to Search



Search Phrases

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

Processing Chunks in a Parallel Stream

- Parallel streams is a “user friendly” ForkJoinPool façade
  - You can program directly to the ForkJoinPool API, though it can be somewhat tedious!
    - Mostly used for parallel algorithms that don’t match Java 8’s parallel streams programming model

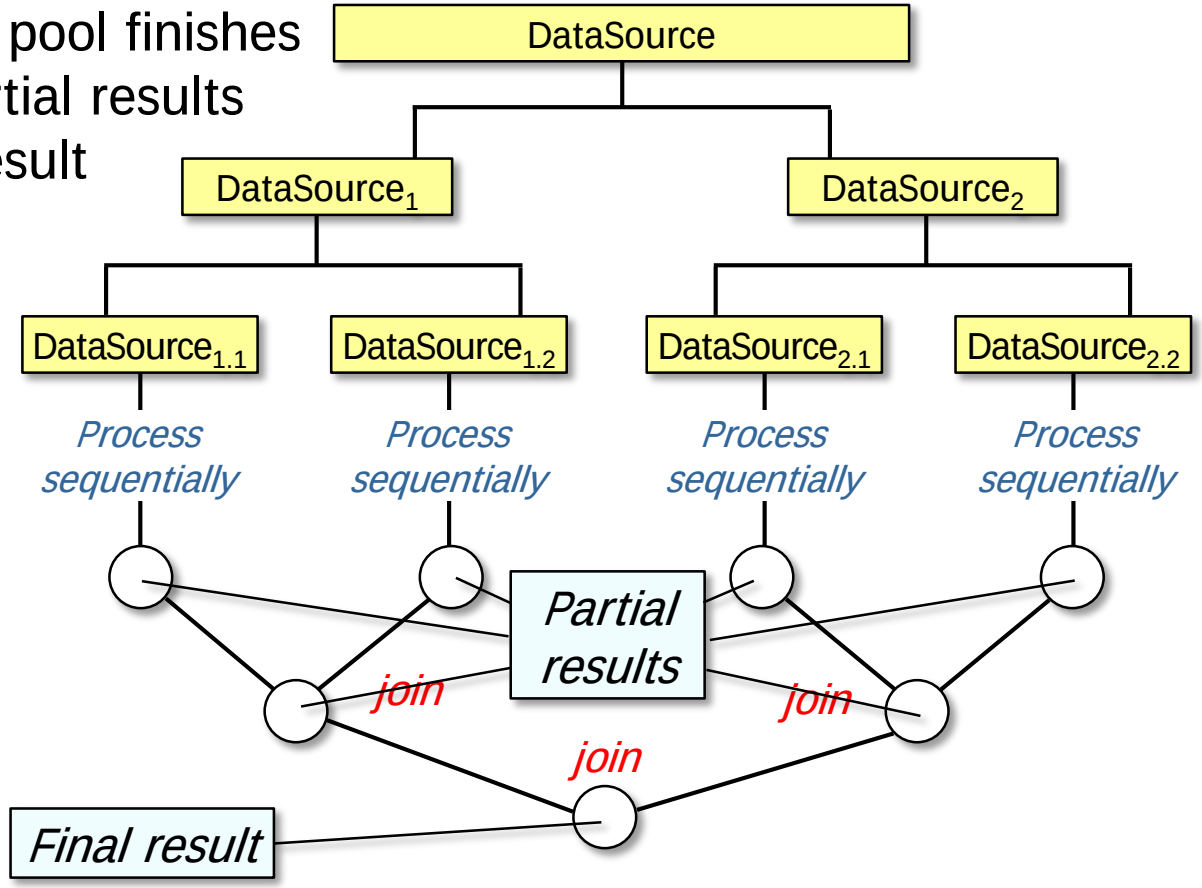
```
if (n % 2 == 1) {
  // f(2n-1) = f(n-1)^2 + f(n)^2
  int left = (n + 1) / 2;
  int right = (n + 1) / 2 - 1;
  FibonacciTask f0 =
    new FibonacciTask(left);
  FibonacciTask f1 =
    new FibonacciTask(right);
  f1.fork();
  BigInteger bi0 = f0.invoke();
  BigInteger bi1 = f1.join();
  if (isCancelled()) return null;
  result =
    square(bi1).add(square(bi0));
 } else {
   // f(2n) = (2*f(n-1)+f(n))*f(n)
```

See www.javaspecialists.eu/archive/Issue201.html

Combining Results in a Parallel Stream

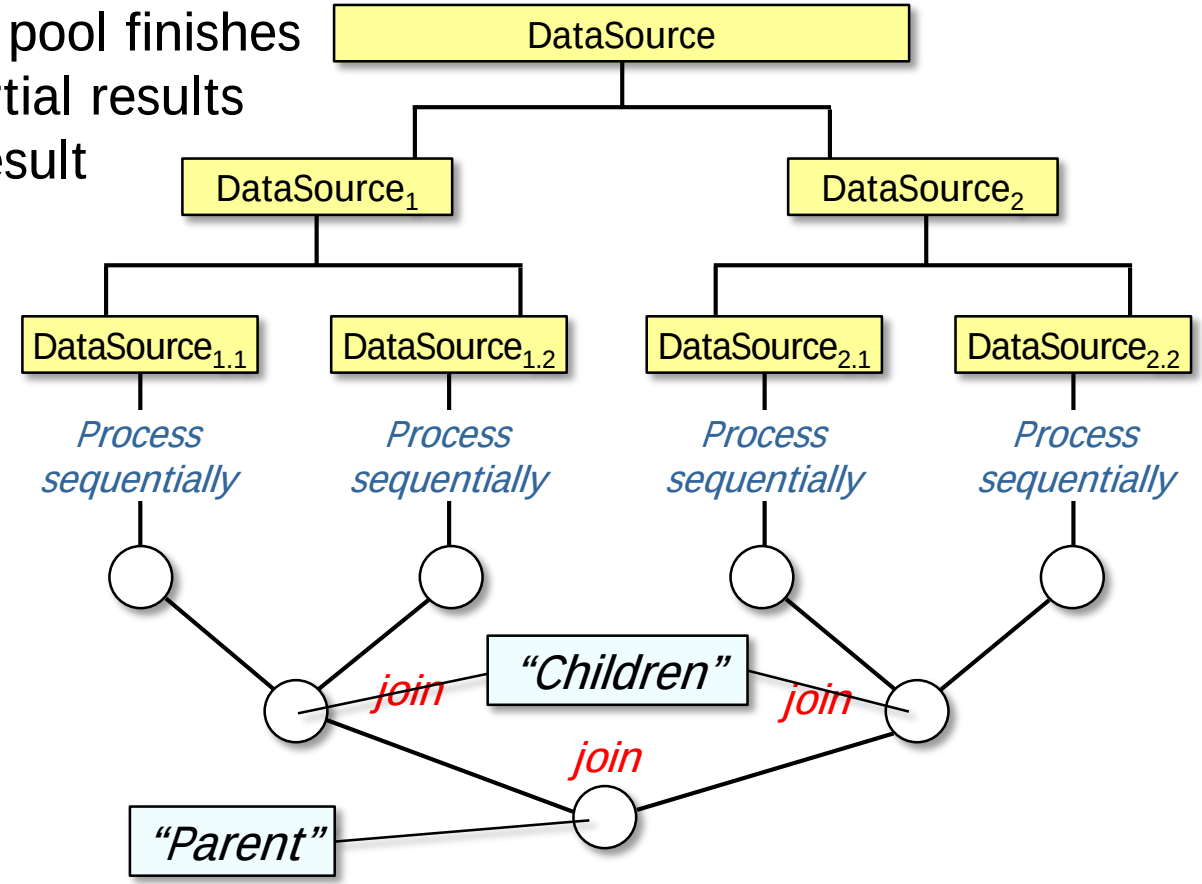
Combining Results in a Parallel Stream

- After the common fork-join pool finishes processing chunks their partial results are combined into a final result



Combining Results in a Parallel Stream

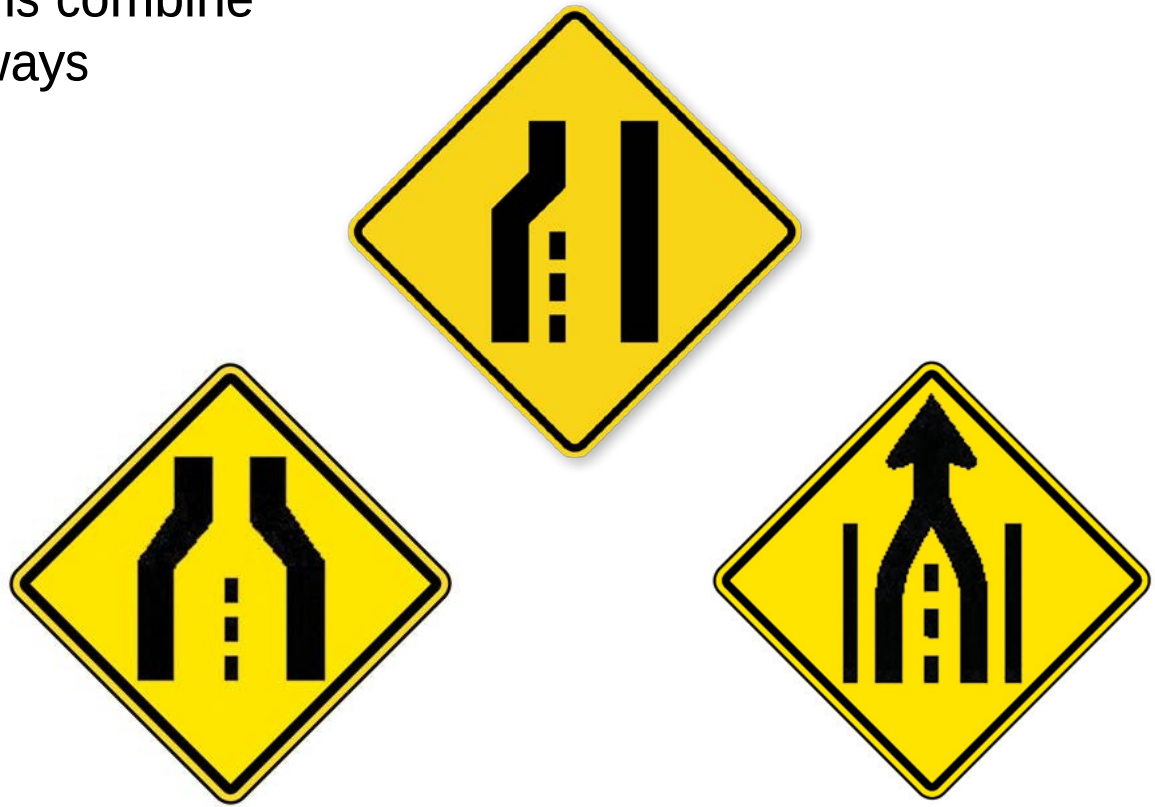
- After the common fork-join pool finishes processing chunks their partial results are combined into a final result
  - join() occurs in a single thread at each level
    - i.e., the “parent”



As a result, there’s typically no need for synchronizers during the joining

Combining Results in a Parallel Stream

- Different terminal operations combine partial results in different ways



Understanding these differences is particularly important for parallel streams

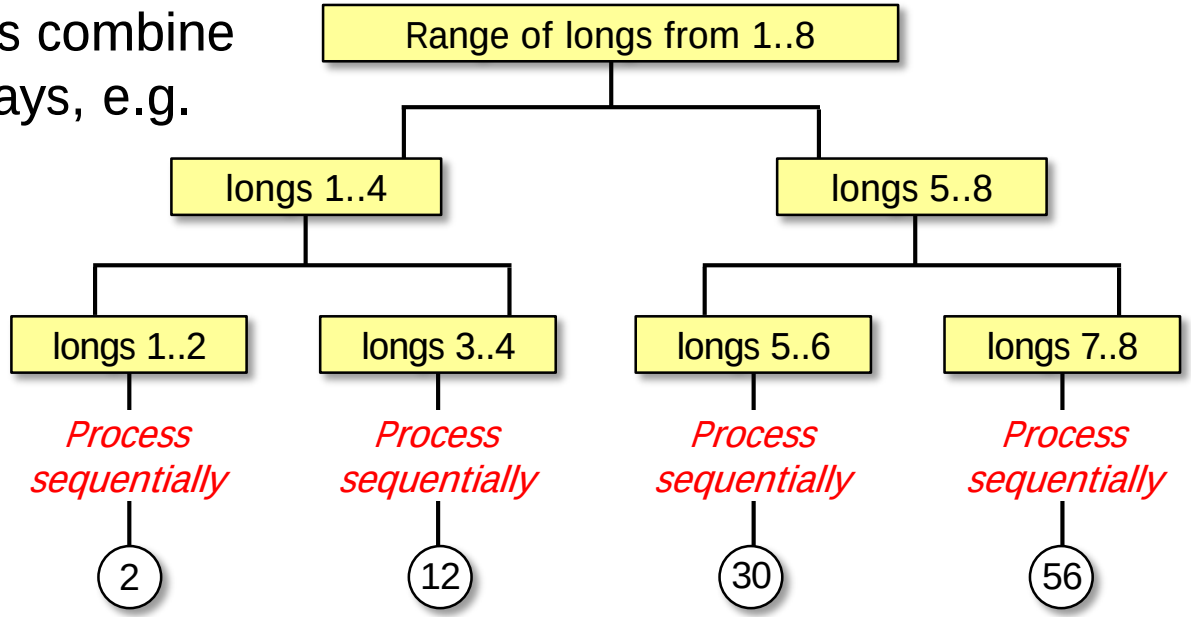
Combining Results in a Parallel Stream

- Different terminal operations combine partial results in different ways, e.g.
  - reduce() creates a new immutable value

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

Combining Results in a Parallel Stream

- Different terminal operations combine partial results in different ways, e.g.
  - reduce() creates a new immutable value

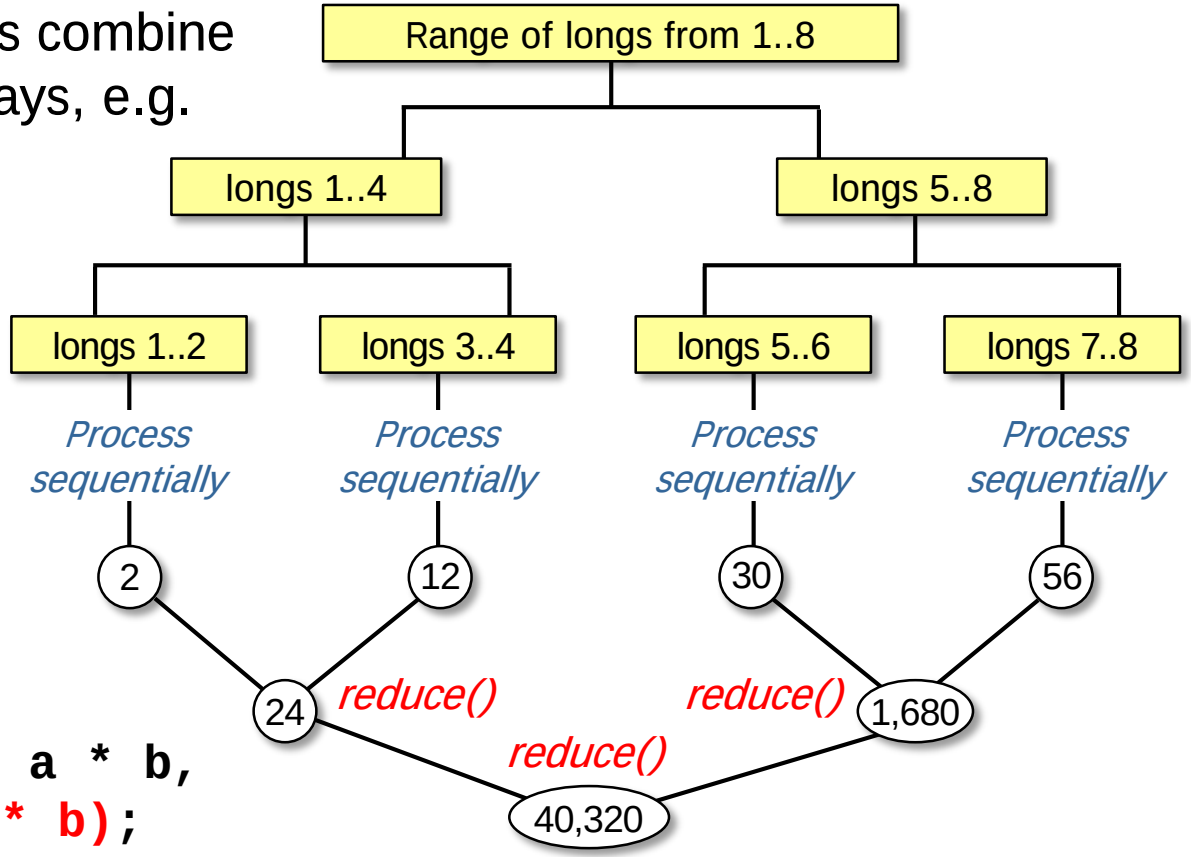



```
long factorial(long n) {
  return LongStream
    .rangeClosed(1, n)
    .parallel()
    .reduce(1, (a, b) -> a * b,
            (a, b) -> a * b);
}
```

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

Combining Results in a Parallel Stream

- Different terminal operations combine partial results in different ways, e.g.
  - reduce() creates a new immutable value




```
long factorial(long n) {
  return LongStream
    .rangeClosed(1, n)
    .parallel()
    .reduce(1, (a, b) -> a * b,
            (a, b) -> a * b);
}
```

reduce() combines two immutable values (e.g., long or Long) & produces a new one

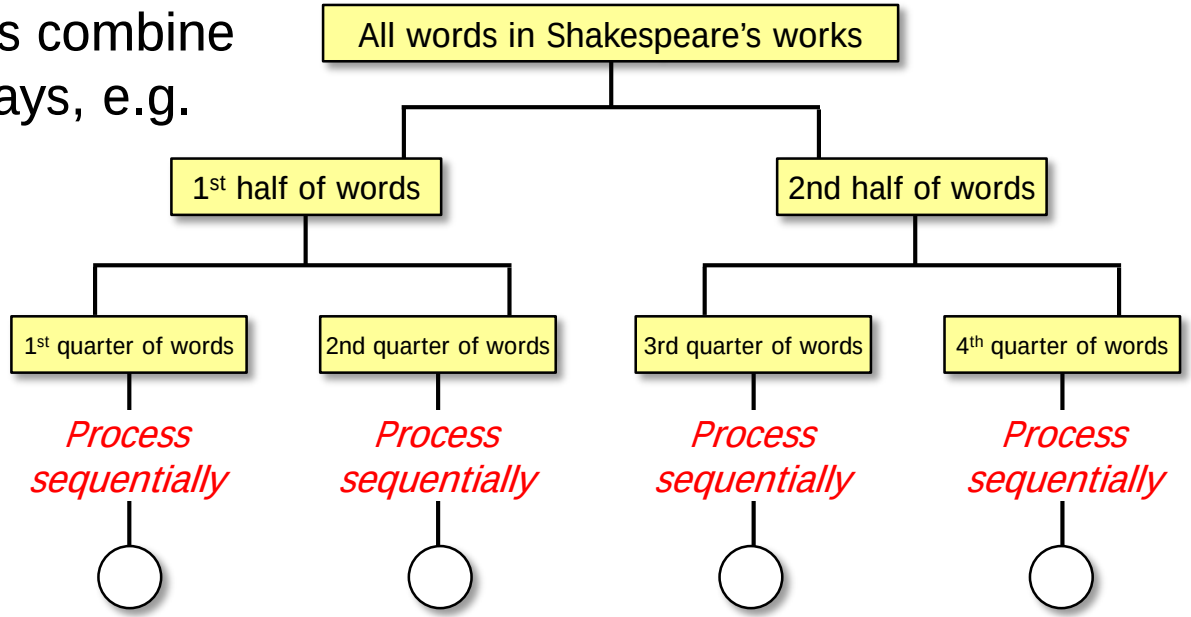
Combining Results in a Parallel Stream

- Different terminal operations combine partial results in different ways, e.g.
  - reduce() creates a new immutable value
  - collect() mutates an existing value

See greenteapress.com/thinkapjava/html/thinkjava011.html

Combining Results in a Parallel Stream

- Different terminal operations combine partial results in different ways, e.g.
  - reduce() creates a new immutable value
  - collect() mutates an existing value

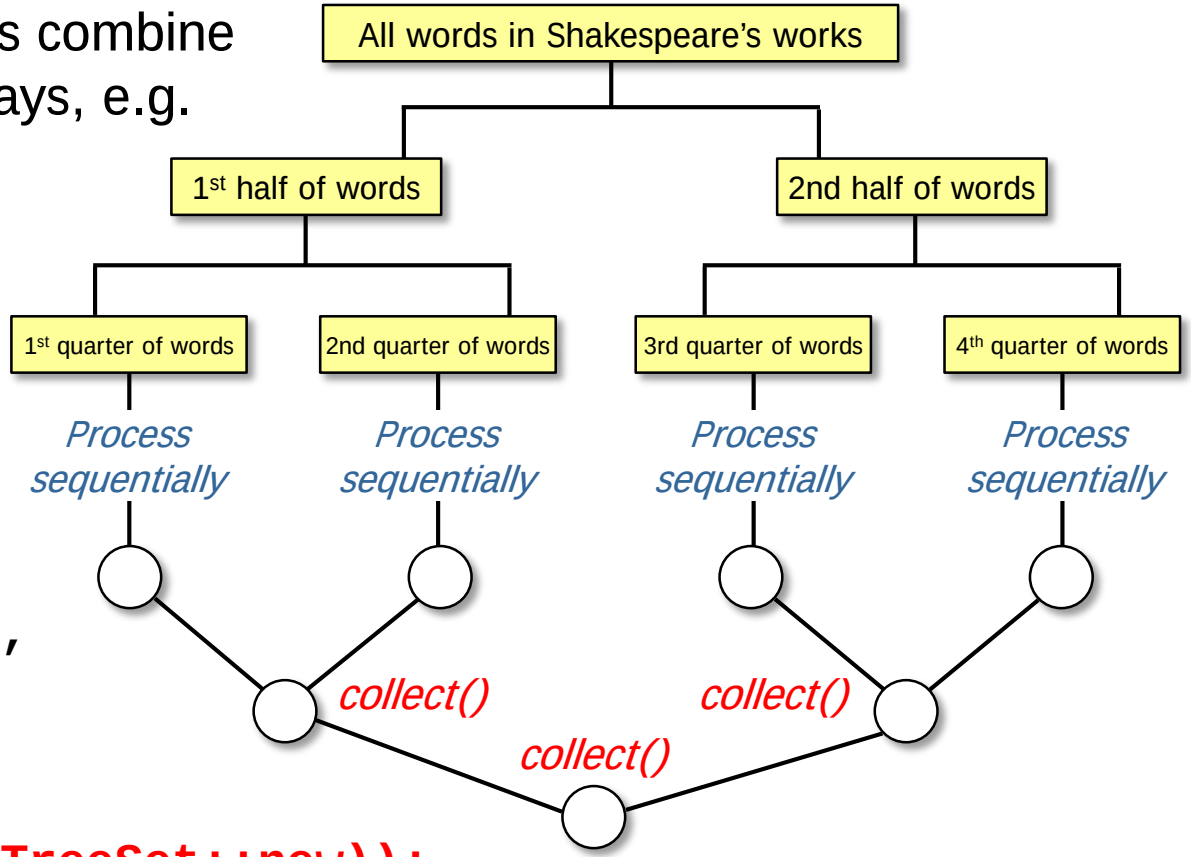



```
List<CharSequence>
      uniqueWords =
  getInput(sSHAKESPEARE),
           "\\s+")
  .parallelStream()
  ...
  .collect(toCollection(TreeSet::new));
```

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

Combining Results in a Parallel Stream

- Different terminal operations combine partial results in different ways, e.g.
  - reduce() creates a new immutable value
  - collect() mutates an existing value

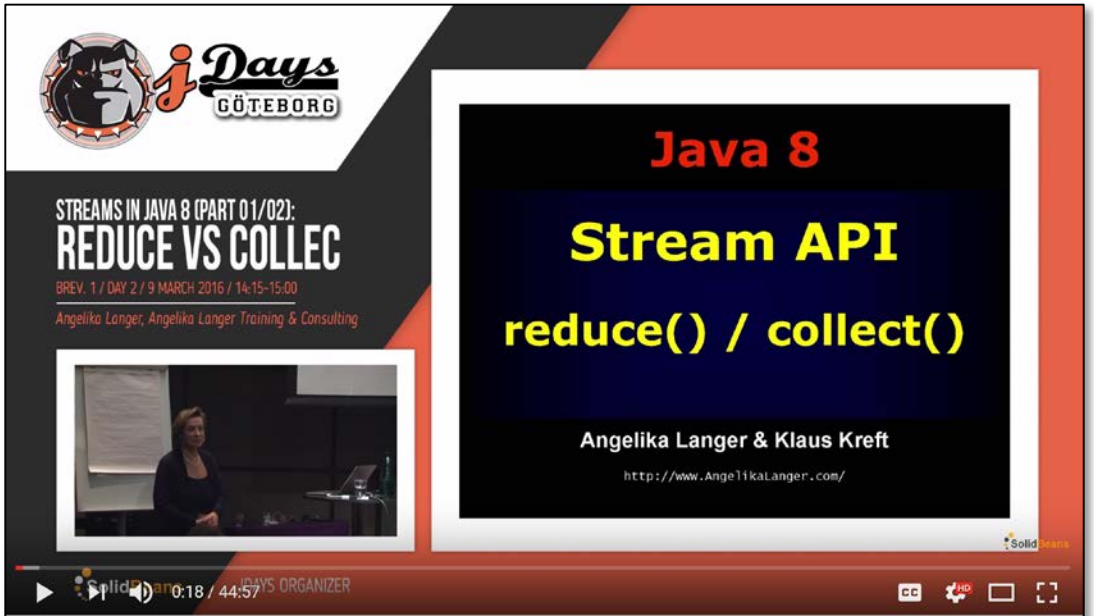



```
List<CharSequence>
      uniqueWords =
  getInput(sSHAKESPEARE),
           "\\s+")
  .parallelStream()
  ...
  .collect(toCollection(TreeSet::new));
```

collect() mutates a container to accumulate the result it's producing

Combining Results in a Parallel Stream

- More discussion about reduce() vs. collect() appears online



See www.youtube.com/watch?v=oWlWEKNM5Aw

Combining Results in a Parallel Stream

- More discussion about reduce() vs. collect() appears online, e.g.
  - Always test w/a parallel stream to detect mistakes wrt mutable vs. immutable reductions

```
void buggyStreamReduce
        (boolean parallel) {
  ...
  Stream<String> wordStream =
    allWords.stream();

  if (parallel)
    wordStream.parallel();

  String words = wordStream
    .reduce(new StringBuilder(),
            StringBuilder::append,
            StringBuilder::append)
    .toString();
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex17

Combining Results in a Parallel Stream

- More discussion about reduce() vs. collect() appears online, e.g.
  - Always test w/a parallel stream to detect mistakes wrt mutable vs. immutable reductions

```
void buggyStreamReduce
        (boolean parallel) {
  ...
  Stream<String> wordStream =
    allWords.stream();

  if (parallel)
    wordStream.parallel();

  String words = wordStream
    .reduce(new StringBuilder(),
            StringBuilder::append,
            StringBuilder::append)
    .toString();
```

This code fails when parallel() is used since reduce() expects to do an "immutable" reduction

Combining Results in a Parallel Stream

- More discussion about reduce() vs. collect() appears online, e.g.
  - Always test w/a parallel stream to detect mistakes wrt mutable vs. immutable reductions

```
void buggyStreamReduce
        (boolean parallel) {
  ...
  Stream<String> wordStream =
    allWords.stream();

  if (parallel)
    wordStream.parallel();

  String words = wordStream
    .reduce(new StringBuilder(),
            StringBuilder::append,
            StringBuilder::append)
    .toString();
```

There are race conditions since there's just a single shared StringBuilder, which is not thread-safe..

Combining Results in a Parallel Stream

- More discussion about reduce() vs. collect() appears online, e.g.
  - Always test w/a parallel stream to detect mistakes wrt mutable vs. immutable reductions

A stream can be dynamically switched to "parallel" mode!

```
void buggyStreamReduce
        (boolean parallel) {
  ...
  Stream<String> wordStream =
    allWords.stream();

  if (parallel)
    wordStream.parallel();

  String words = wordStream
    .reduce(new StringBuilder(),
            StringBuilder::append,
            StringBuilder::append)
    .toString();
```

Combining Results in a Parallel Stream

- More discussion about reduce() vs. collect() appears online, e.g.
  - Always test w/a parallel stream to detect mistakes wrt mutable vs. immutable reductions
  - Beware of issues related to association & identity

```
void testDifferenceReduce(...) {
  long difference = LongStream
    .rangeClosed(1, 100)
    .parallel()
    .reduce(0L,
            (x, y) -> x - y);
}

 void testBuggySequentialSum() {
   long sum = LongStream
     .rangeClosed(1, 100)
     .reduce(1L,
             Math::addExact);
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex17

Combining Results in a Parallel Stream

- More discussion about reduce() vs. collect() appears online, e.g.
  - Always test w/a parallel stream to detect mistakes wrt mutable vs. immutable reductions
- Beware of issues related to association & identity

This code fails for a parallel stream since subtraction is not associative

```
void testDifferenceReduce(...) {
  long difference = LongStream
    .rangeClosed(1, 100)
    .parallel()
    .reduce(0L,
            (x, y) -> x - y);
}

void testBuggySequentialSum() {
  long sum = LongStream
    .rangeClosed(1, 100)
    .reduce(1L,
            Math::addExact);
```

Combining Results in a Parallel Stream

- More discussion about reduce() vs. collect() appears online, e.g.
  - Always test w/a parallel stream to detect mistakes wrt mutable vs. immutable reductions
  - Beware of issues related to association & identity

```
void testDifferenceReduce(...) {
  long difference = LongStream
    .rangeClosed(1, 100)
    .parallel()
    .reduce(0L,
            (x, y) -> x - y);
}
```

```
void testSum(long identity, ...) {
  long sum = LongStream
    .rangeClosed(1, 100)
    .reduce(identity,
     // Could use (x, y) -> x + y
            Math::addExact);
```

This code fails if identity is not 0L

The “identity” of an OP is defined as “identity OP value == value”

Combining Results in a Parallel Stream

- More discussion about reduce() vs. collect() appears online, e.g.
  - Always test w/a parallel stream to detect mistakes wrt mutable vs. immutable reductions
  - Beware of issues related to association & identity
- Be aware of custom collectors


```
mList.stream().collect(Collector.of
                       (() -> new StringJoiner("|"),
                       (j, result) -> j.add(result.toString()),
                        StringJoiner::merge,
                        StringJoiner::toString));
```

SearchResults's custom collector formats itself

See www.youtube.com/watch?v=H7VbRz9aj7c

End of Overview of Java 8 Parallel Streams (Part 2)