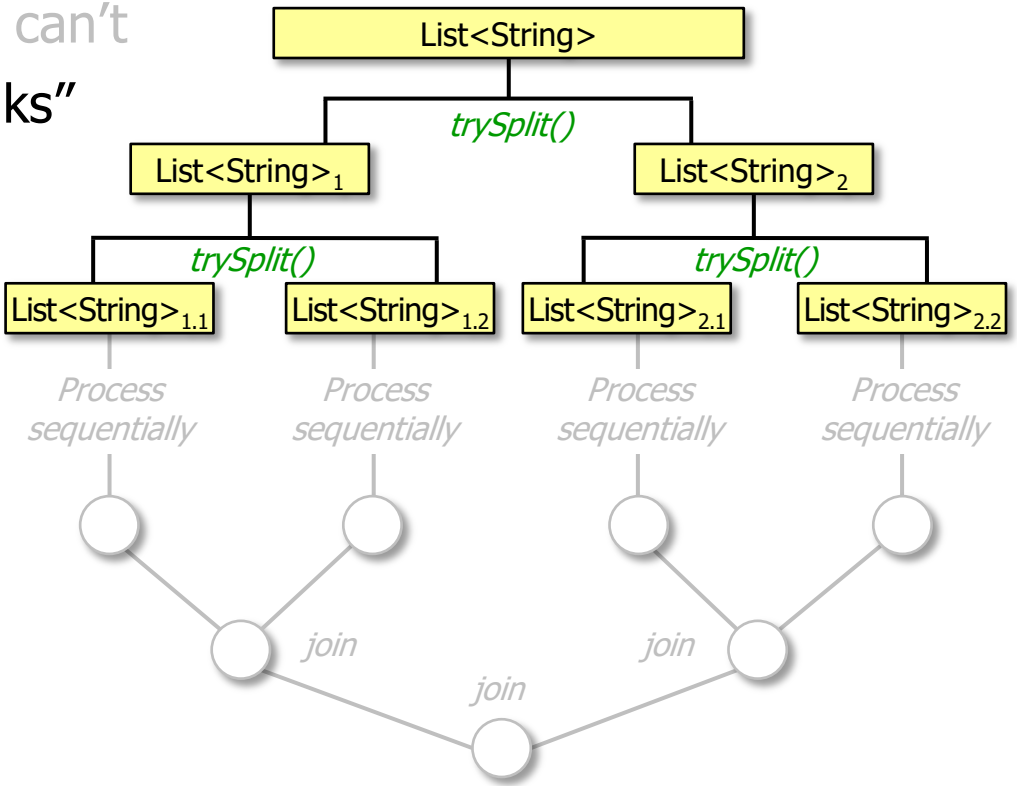
Java Parallel Stream Internals

Partitioning

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

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

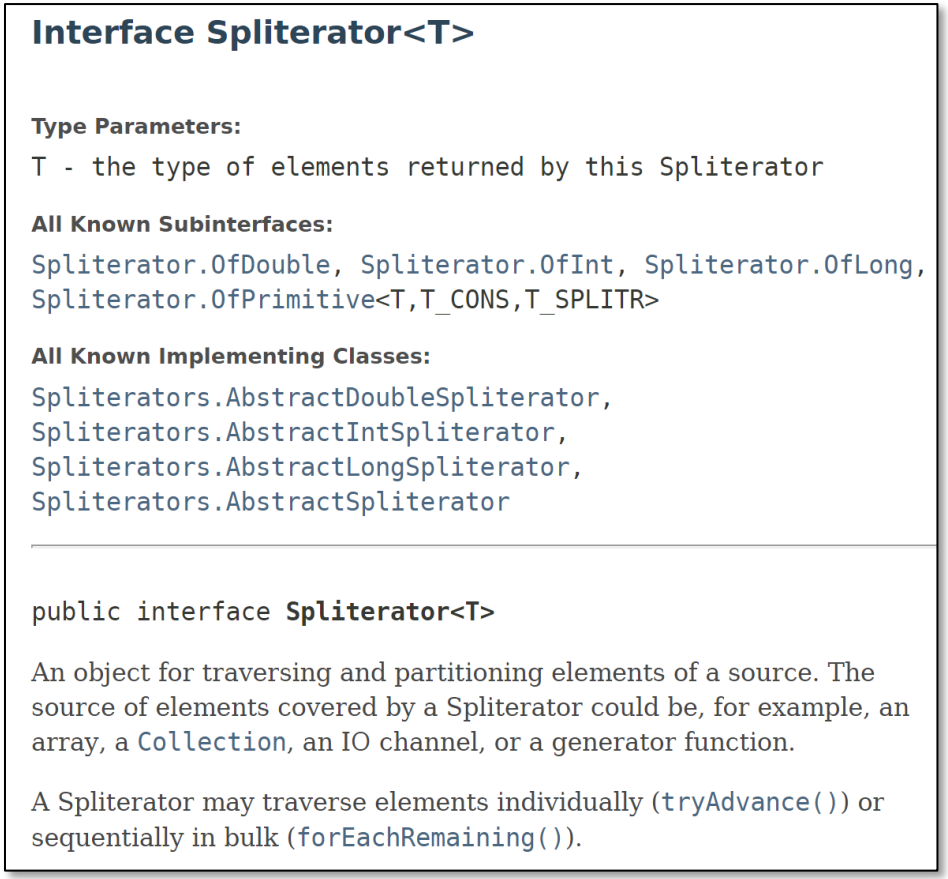


See www.ibm.com/developerworks/library/j-java-streams-3-brian-goetz

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

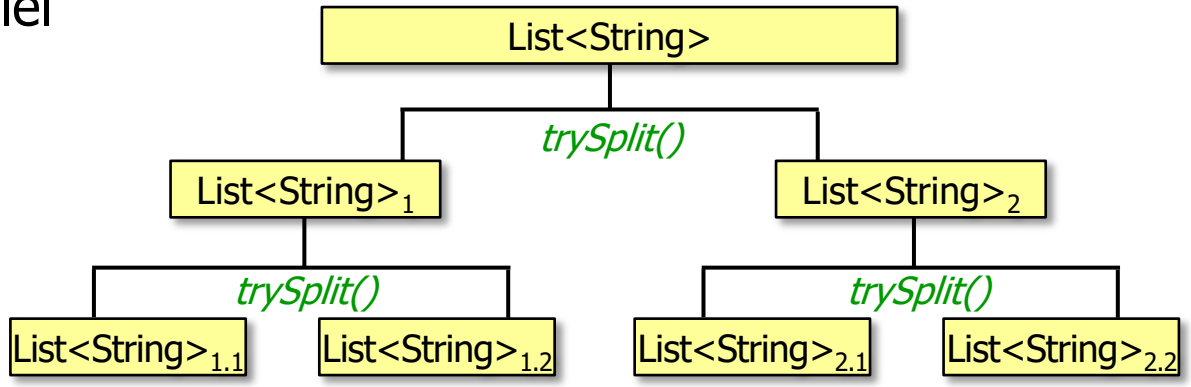
- We've shown how a Spliterator can *traverse* elements in a source.

```
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."*

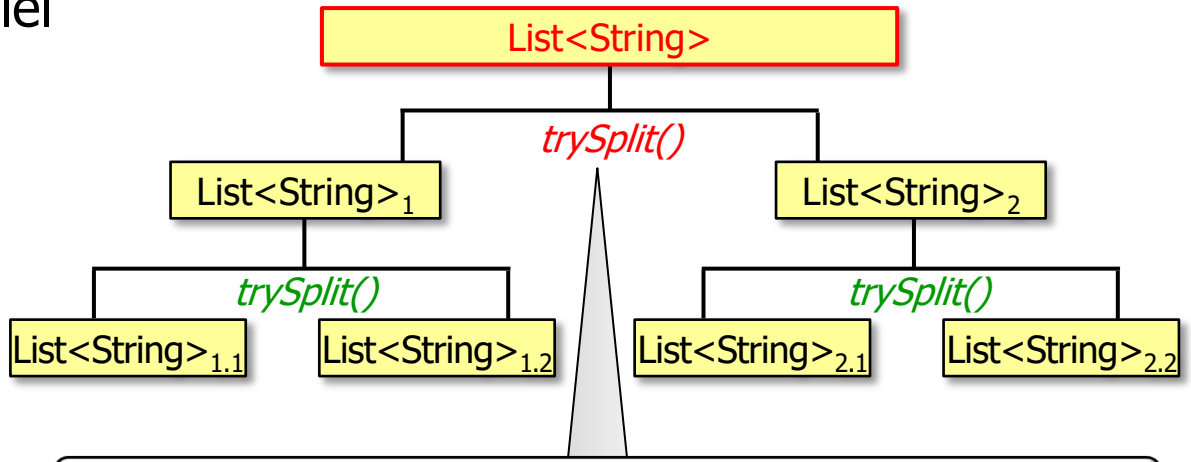
Partitioning a Parallel Stream

- 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.

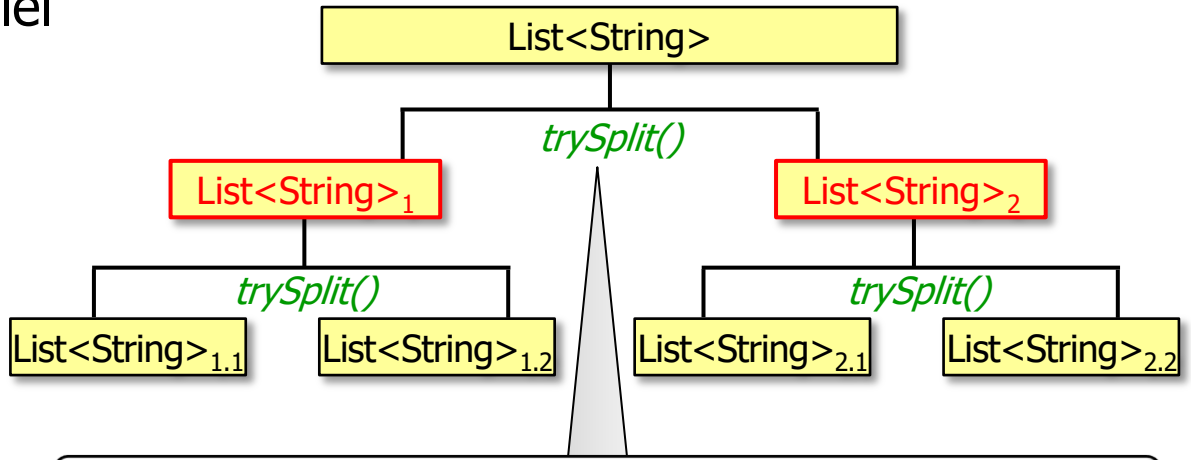



```
Spliterator<T> trySplit() {
  if (input is <= minimum size)
    return null
  else {
    split input in 2 (even-sized) chunks
    return a spliterator for "left chunk"
  }
```

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.

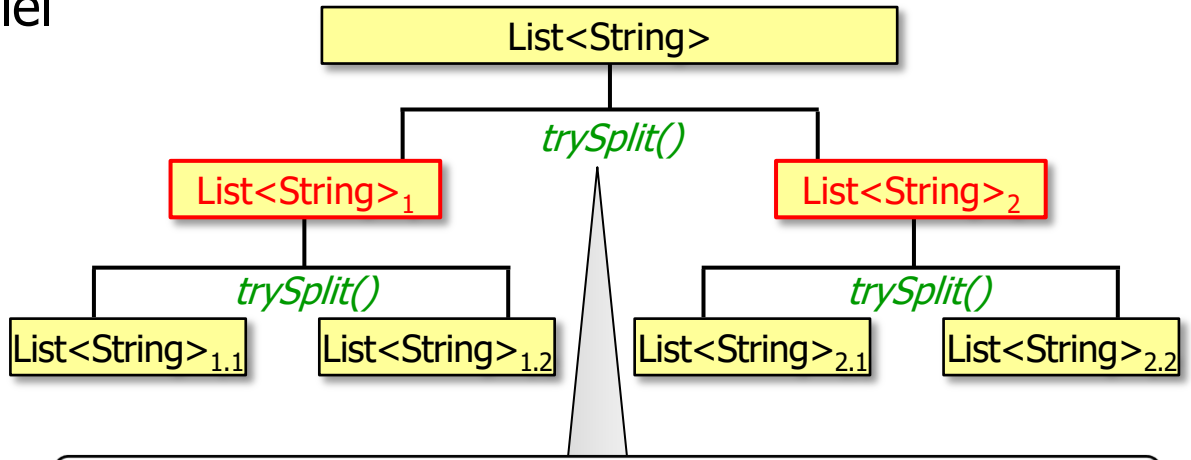



```
Spliterator<T> trySplit() {
  if (input is <= minimum size)
    return null
  else {
    split input in 2 (even-sized) chunks
    return a spliterator for "left chunk"
  }
```

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.

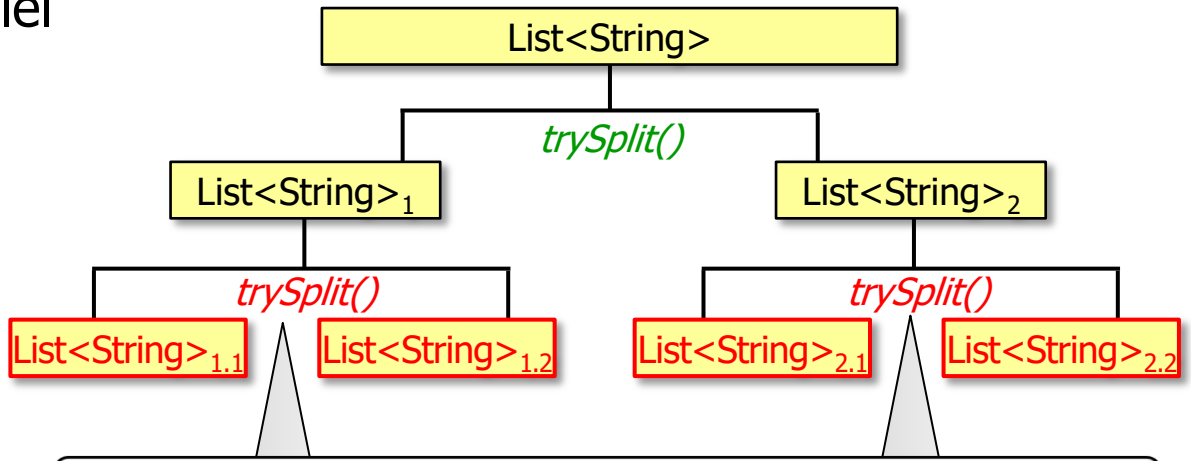



```
Spliterator<T> trySplit() {
  if (input is <= minimum size)
    return null
  else {
    split input in 2 (even-sized) chunks
    return a spliterator for "left chunk"
  }
```

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.

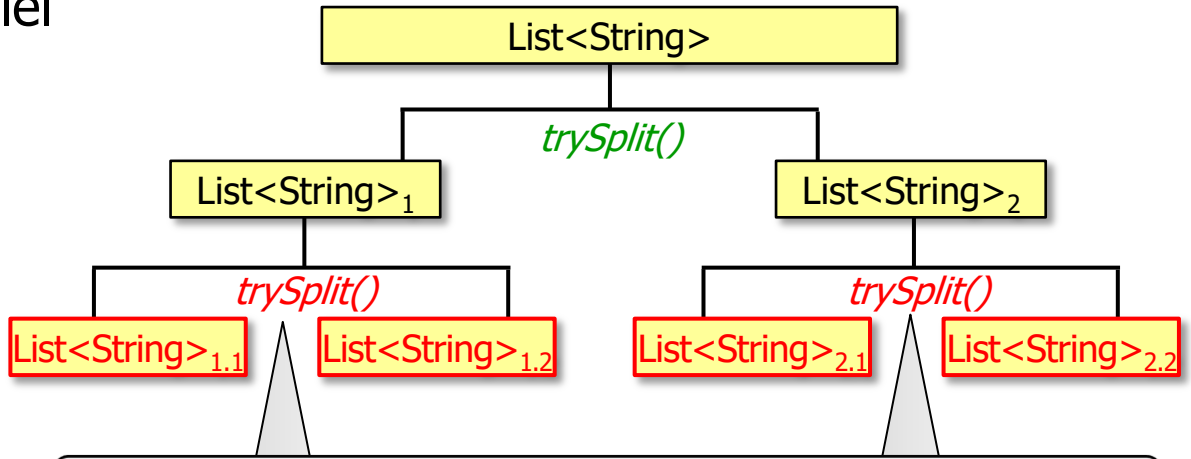



```
Spliterator<T> trySplit() {
  if (input is <= minimum size)
    return null
  else {
    split input in 2 (even-sized) chunks
    return a spliterator for "left chunk"
  }
```

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.

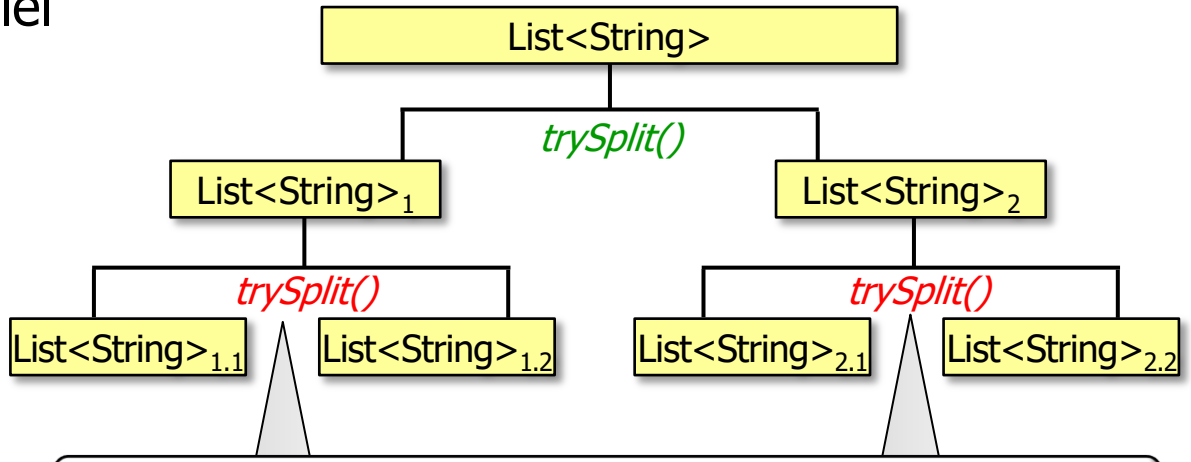


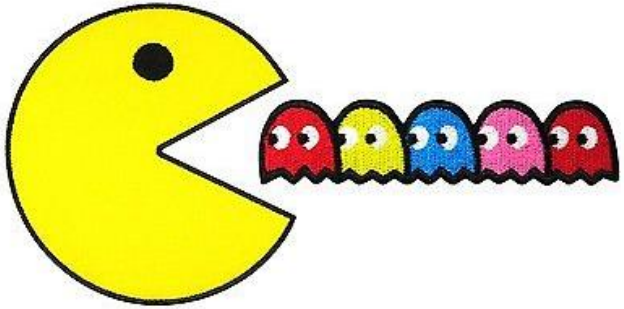

```
Spliterator<T> trySplit() {
  if (input is <= minimum size)
    return null
  else {
    split input in 2 (even-sized) chunks
    return a spliterator for "left chunk"
  }
```

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 and 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 and 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 and 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 and 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 and 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 and efficiently, e.g., LinkedList.

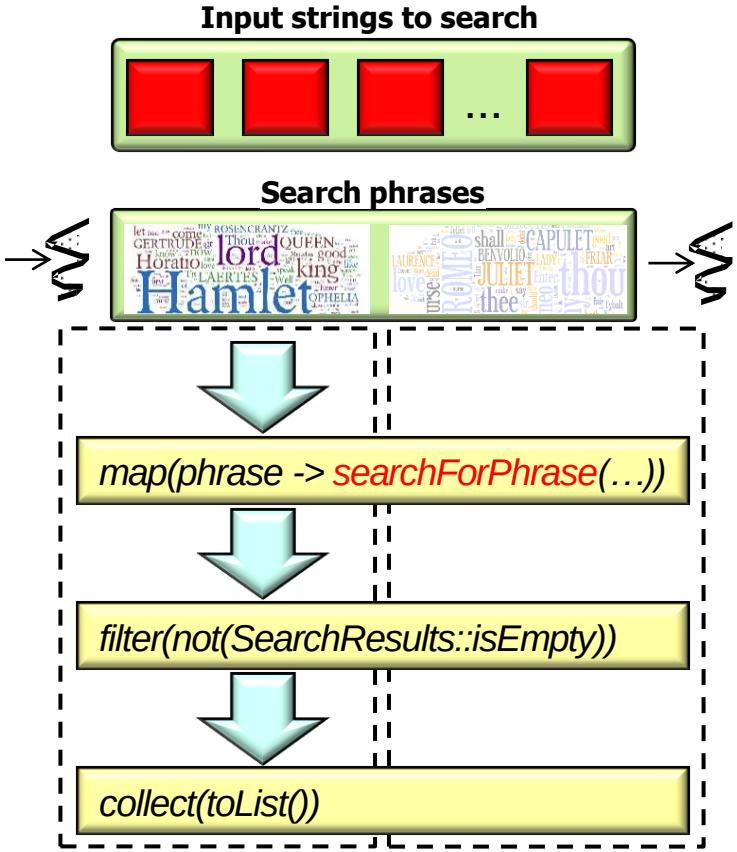
```
Spliterator<E> trySplit() { ...
  int n = batch + BATCH_UNIT, j = 0; Object[] a = new Object[n];
  do { a[j++] = p.item; }
  while ((p = p.next) != null && j < n); ...
  return Spliterators.spliterator(a, 0, j, Spliterator.ORDERED);
}
```

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()"*

The End

Java Parallel Stream Internals

Demo’ing Spliterator Performance

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Understand parallel stream internals, e.g.
  - Know what can change and what can't
  - Partition a data source into "chunks"
    - Know the impact of different Java collections on the performance of different spliterators

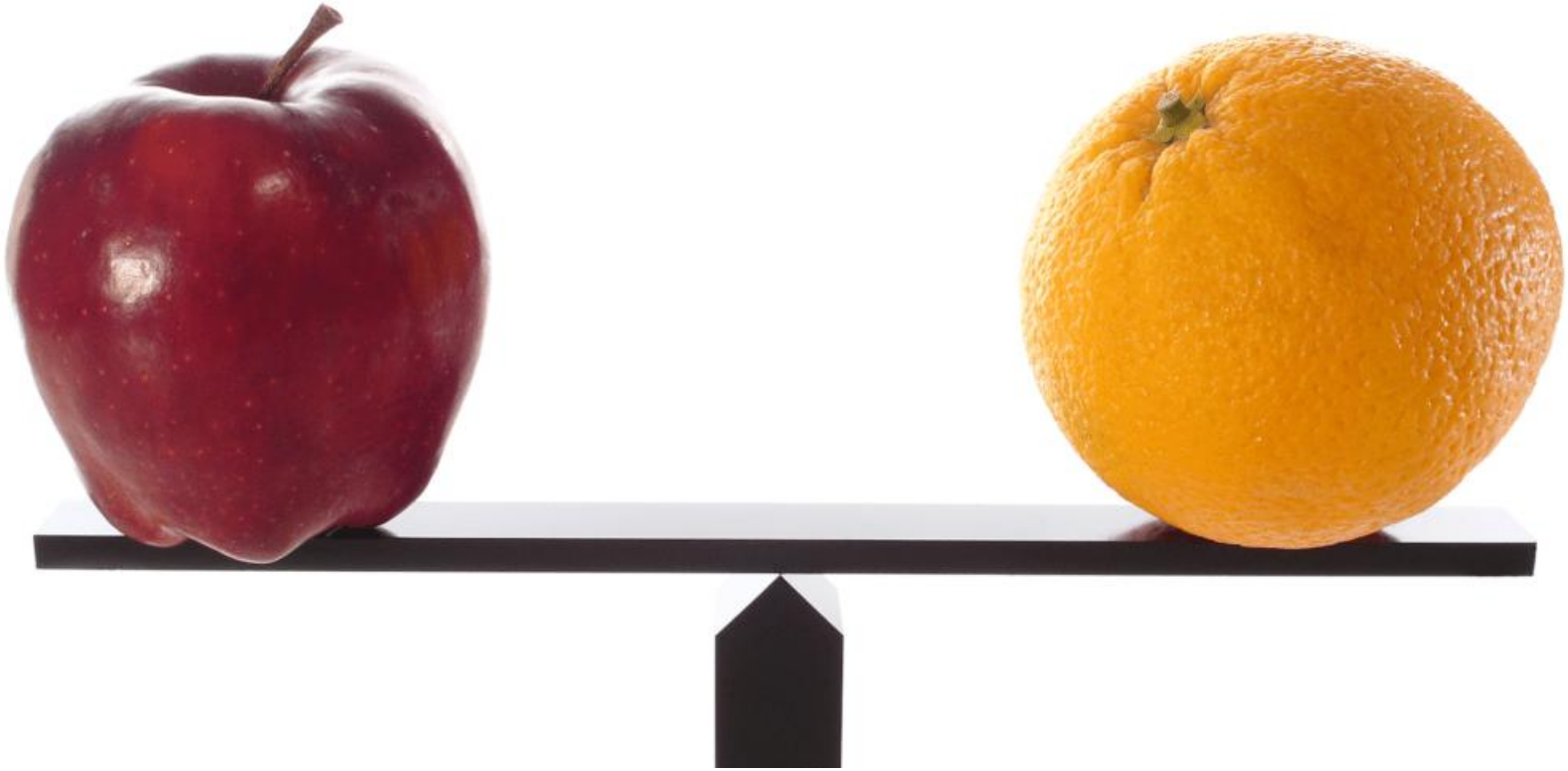
```
Starting spliterator tests for 1000 words..
..printing results
    17 msecs: ArrayList parallel
    19 msecs: LinkedList parallel

Starting spliterator tests for 10000 words..
..printing results
    88 msecs: LinkedList parallel
    90 msecs: ArrayList parallel

Starting spliterator tests for 100000 words..
..printing results
   599 msecs: ArrayList parallel
   701 msecs: LinkedList parallel

Starting spliterator tests for 883311 words..
..printing results
  5718 msecs: ArrayList parallel
 31226 msecs: LinkedList parallel
```

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

Demonstrating Spliterator Performance

Demonstrating Spliterator Performance

- Spliterators for ArrayList and LinkedList partition data quite differently.



See earlier lesson on "*Java Parallel Streams Internals: Partitioning*"

Demonstrating Spliterator Performance

- Spliterators for ArrayList and LinkedList partition data quite differently.

```
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, ...);
}
```

ArrayList's spliterator splits evenly & efficiently (e.g., doesn't copy data)

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

Demonstrating Spliterator Performance

- Spliterators for ArrayList and LinkedList partition data quite differently.

```
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);
}
```

LinkedList's spliterator does not split evenly & efficiently (e.g., it copies data)

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

Demonstrating Spliterator Performance

- This program shows the performance difference of parallel spliterators for ArrayList and LinkedList when processing the complete works of Shakespeare.

```
void timeParallelStreamUppercase(String testName,
                                 List<CharSequence> words) {
  ...
  List<String> list = new ArrayList<>();

  for (int i = 0; i < sMAX_ITERATIONS; i++)
    list
      .addAll(words
                .parallelStream()
                .map(charSeq ->
                     charSeq.toString().toUpperCase())
                .collect(toList())); ...
```

Demonstrating Spliterator Performance

- This program shows the performance difference of parallel spliterators for ArrayList and LinkedList when processing the complete works of Shakespeare.

```
void timeParallelStreamUppercase(String testName,
                                 List<CharSequence> words) {
  ...
  List<String> list = new ArrayList<>();

  for (int i = 0; i < sMAX_ITERATIONS; i++)
    list
      .addAll(words
                .parallelStream()
                .map(charSeq ->
                     charSeq.toString().toUpperCase())
                .collect(toList())); ...
```

The words param is passed an ArrayList & a LinkedList

Demonstrating Spliterator Performance

- This program shows the performance difference of parallel spliterators for ArrayList and LinkedList when processing the complete works of Shakespeare.

```
void timeParallelStreamUppercase(String testName,
                                 List<CharSequence> words) {
  ...
  List<String> list = new ArrayList<>();

  for (int i = 0; i < sMAX_ITERATIONS; i++)
    list
      .addAll(words
                .parallelStream()
                .map(charSeq ->
                     charSeq.toString().toUpperCase())
                .collect(toList())); ...
```

Split & uppercase the word list via a parallel spliterator

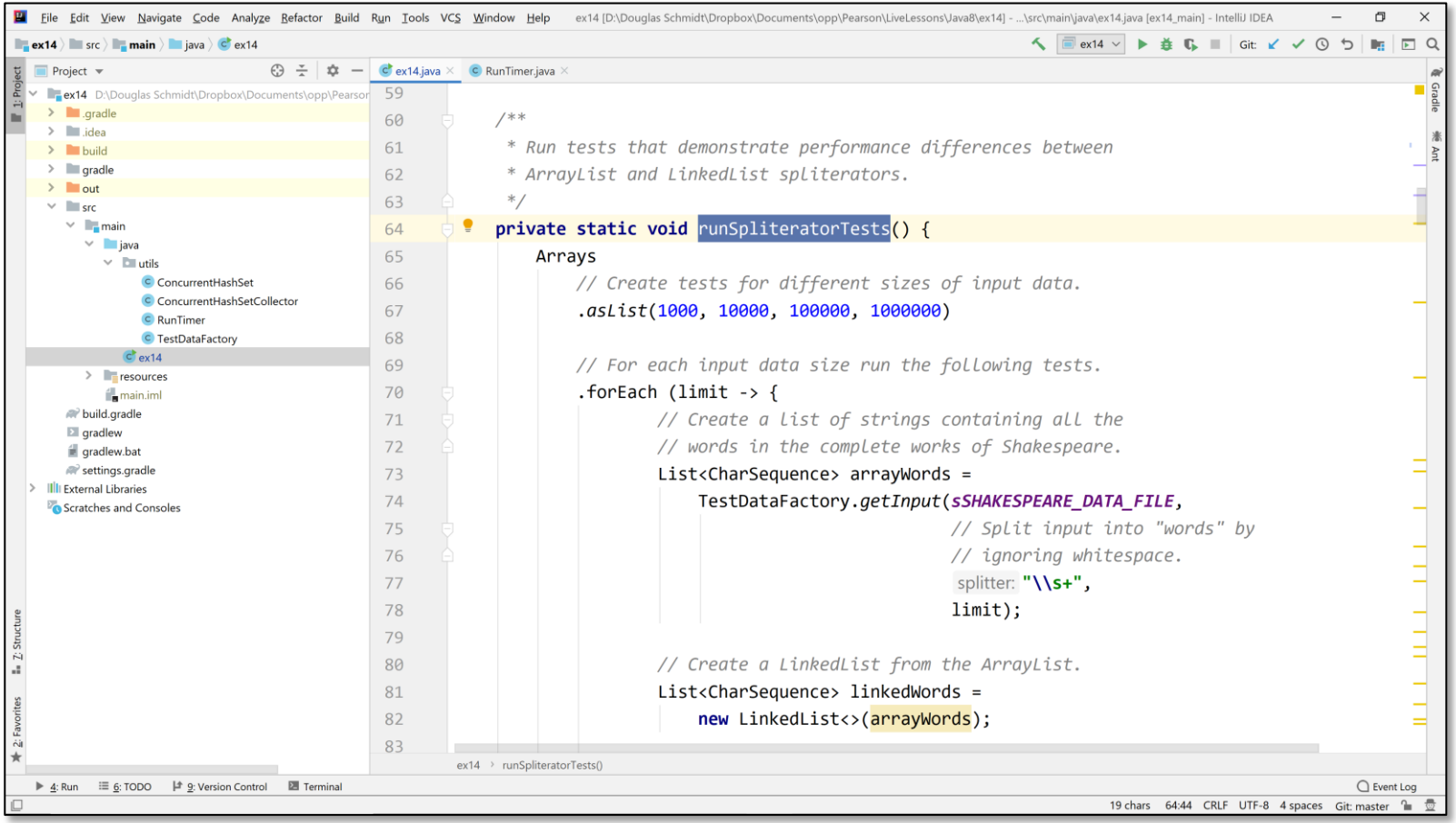
Demonstrating Spliterator Performance

- Results show spliterator differences become more significant as input grows.

```
Starting spliterator tests for 1000 words....printing results
    17 msecs: ArrayList parallel
    19 msecs: LinkedList parallel

Starting spliterator tests for 10000 words....printing results
    88 msecs: ArrayList parallel
    90 msecs: LinkedList parallel

Starting spliterator tests for 100000 words....printing results
   599 msecs: ArrayList parallel
   701 msecs: LinkedList parallel

Starting spliterator tests for 883311 words....printing results
  5718 msecs: ArrayList parallel
 31226 msecs: LinkedList parallel
```

See upcoming lessons on "*When [Not] to Use Parallel Streams*"

Demonstrating Spliterator Performance



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

Demonstrating Spliterator Performance

[Source code analysis video goes here!]

The End

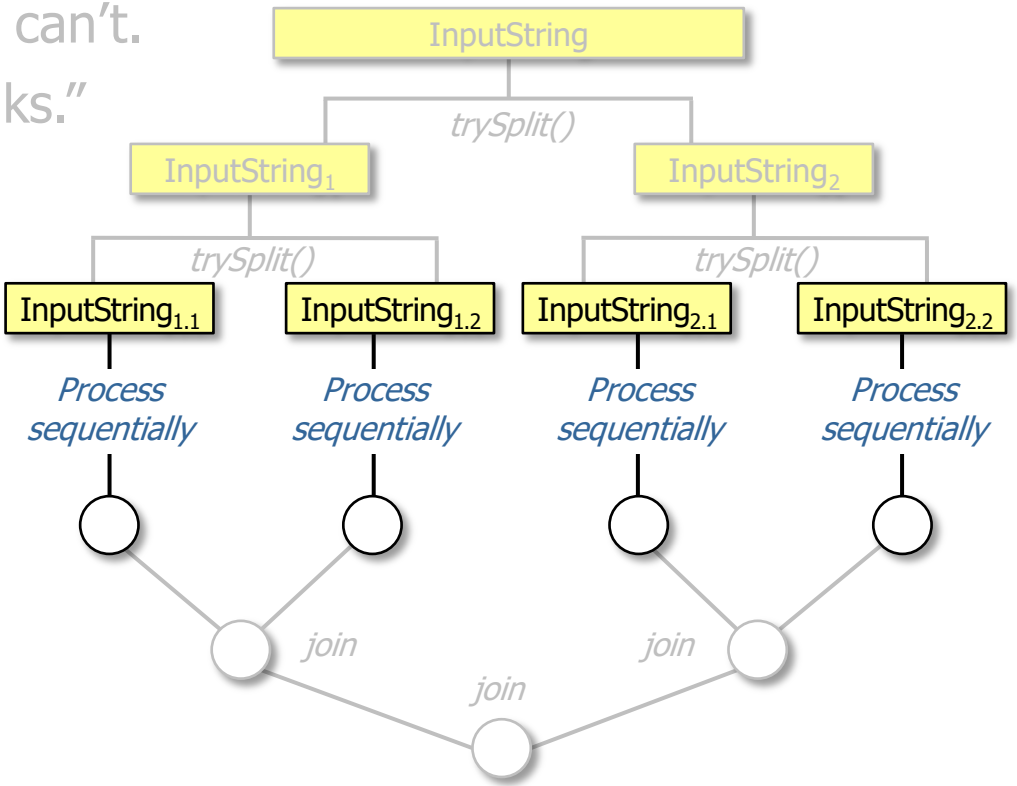
Java Parallel Stream Internals

Parallel Processing via the Common ForkJoinPool

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Understand parallel stream internals, e.g.
  - Know what can change and what can't.
  - Partition a data source into "chunks."
  - Process chunks in parallel via the common ForkJoinPool.

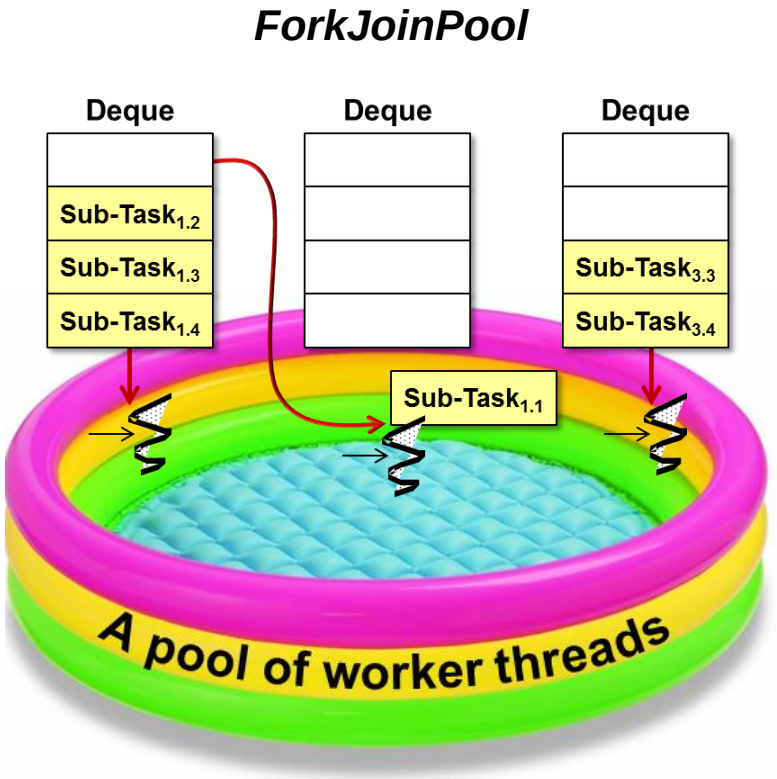


See www.ibm.com/developerworks/library/j-java-streams-3-brian-goetz

Processing Chunks in Parallel via the Common ForkJoinPool

Processing Chunks in Parallel via the Common ForkJoinPool

- Chunks created by a Spliterator are processed in the common ForkJoinPool.



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

Processing Chunks in Parallel via the Common ForkJoinPool

- A ForkJoinPool provides a high performance, fine-grained task execution framework for Java data parallelism.

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.

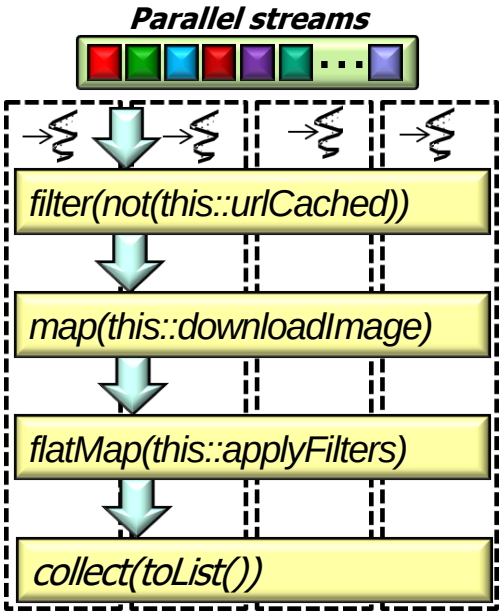
A static `commonPool()` is available and appropriate for most applications. The common pool is used by any ForkJoinTask that is not explicitly submitted to a specified pool. Using the common pool normally reduces resource usage (its threads are slowly reclaimed during periods of non-use, and reinstated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

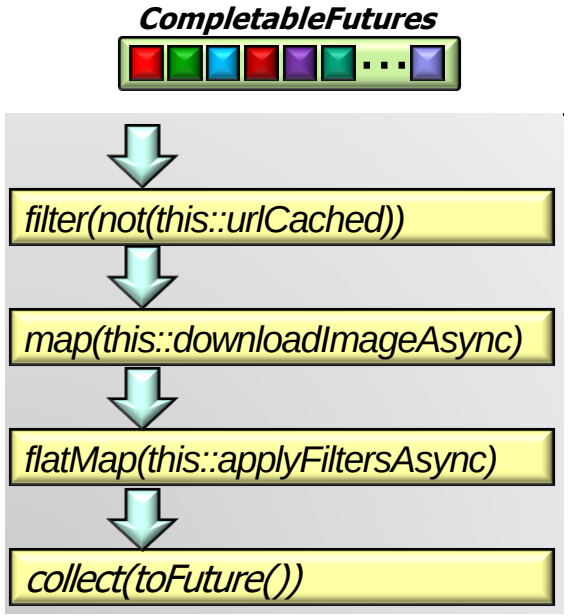
For applications that require separate or custom pools, a `ForkJoinPool` may be constructed with a given target parallelism level; by default, equal to the number of available processors. The pool attempts to maintain enough active (or available) threads by dynamically adding, suspending, or resuming internal worker threads, even if some tasks are stalled waiting to join others. However, no such adjustments are guaranteed in the face of blocked I/O or other unmanaged synchronization. The nested `ForkJoinPool.ManagedBlocker` interface enables extension of the kinds of synchronization accommodated.

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

Processing Chunks in Parallel via the Common ForkJoinPool

- A ForkJoinPool provides a high performance, fine-grained task execution framework for Java data parallelism.
  - It provides a parallel computing engine for many higher-level frameworks.

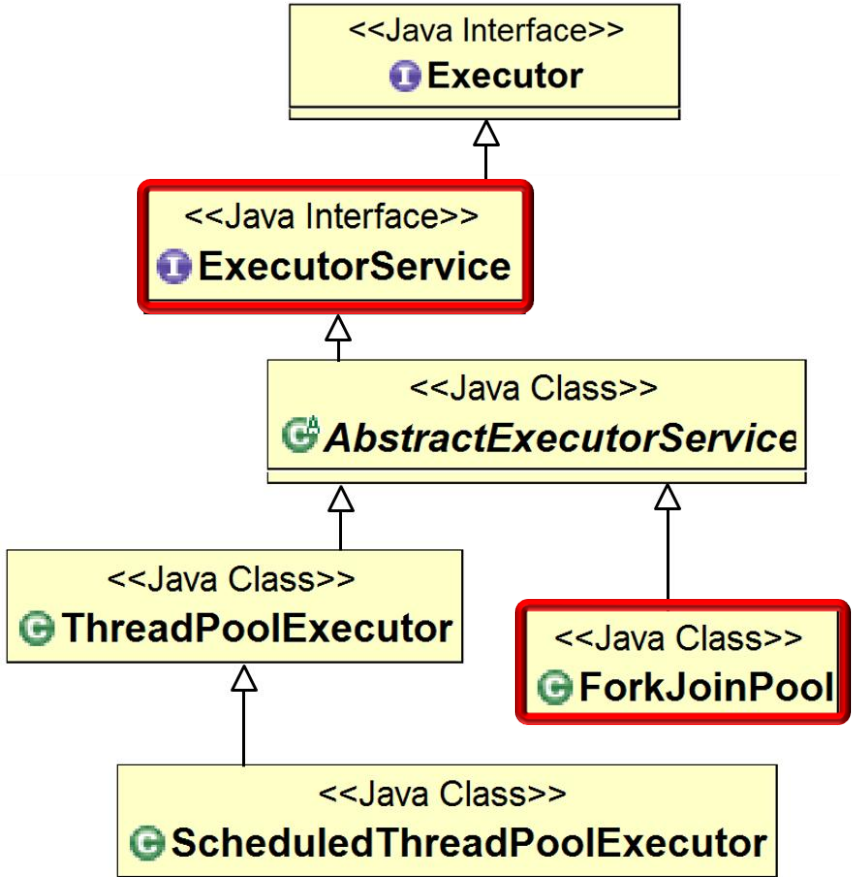




See www.infoq.com/interviews/doug-lea-fork-join

Processing Chunks in Parallel via the Common ForkJoinPool

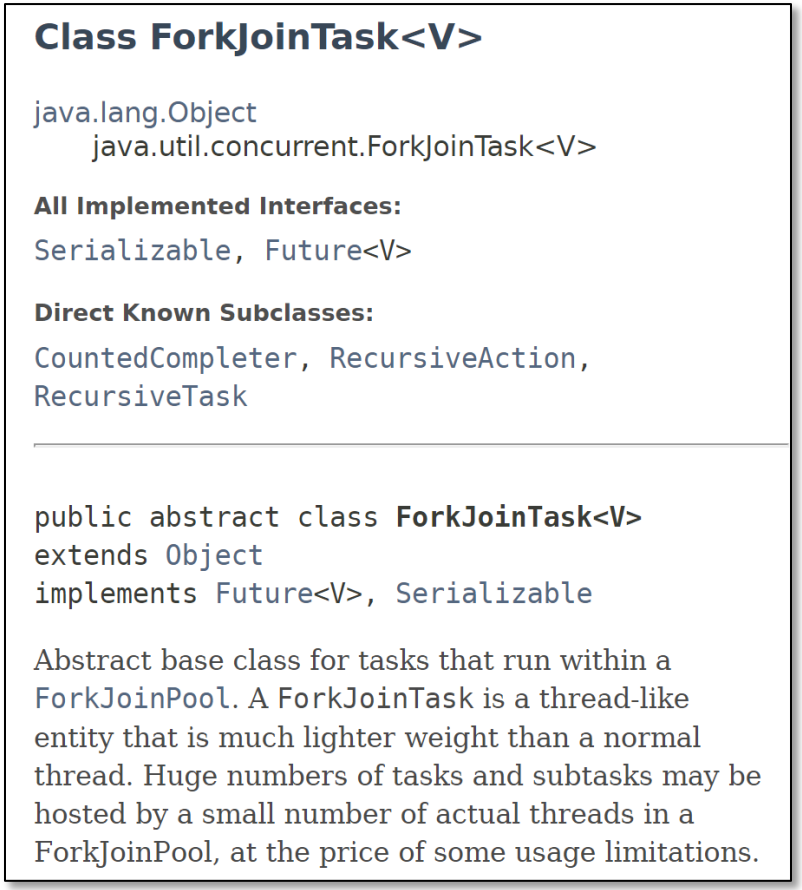
- ForkJoinPool implements the ExecutorService interface.



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

Processing Chunks in Parallel via the Common ForkJoinPool

- ForkJoinPool implements the ExecutorService interface.
  - A ForkJoinPool executes fork/join tasks.



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.

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

Processing Chunks in Parallel via the Common ForkJoinPool

- ForkJoinPool implements the ExecutorService interface.
  - A ForkJoinPool executes fork/join tasks.
  - A fork/join task associates a chunk of data along with a computation on that data to enable fine-grained parallelism.

Class ForkJoinTask<V>

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

All Implemented Interfaces:

`Serializable, Future<V>`

Direct Known Subclasses:

`CountedCompleter, RecursiveAction, RecursiveTask`

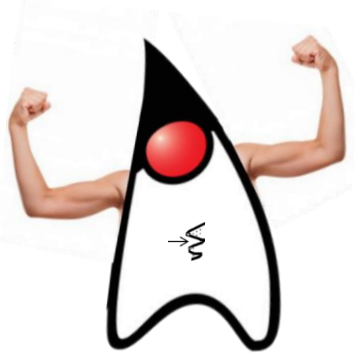
```
public abstract class ForkJoinTask<V>
extends Object
implements Future<V>, Serializable
```

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.

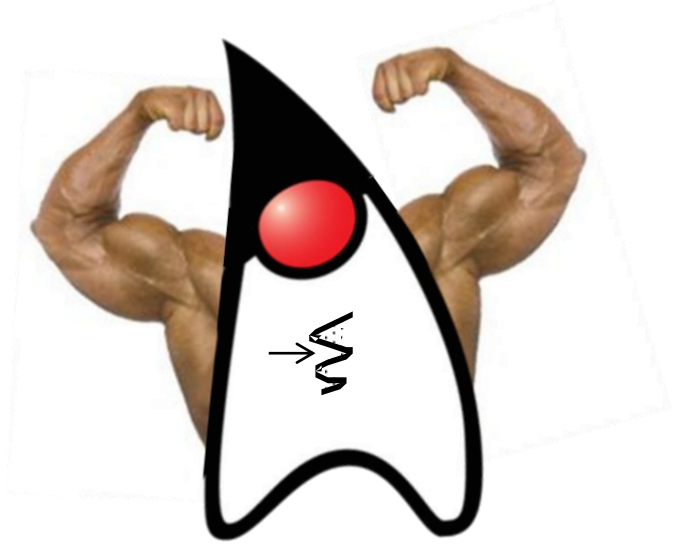
See www.dre.vanderbilt.edu/~schmidt/PDF/DataParallelismInJava.pdf

Processing Chunks in Parallel via the Common ForkJoinPool

- A fork/join task is similar to—but lighter weight—than a Java thread.



fork/join task

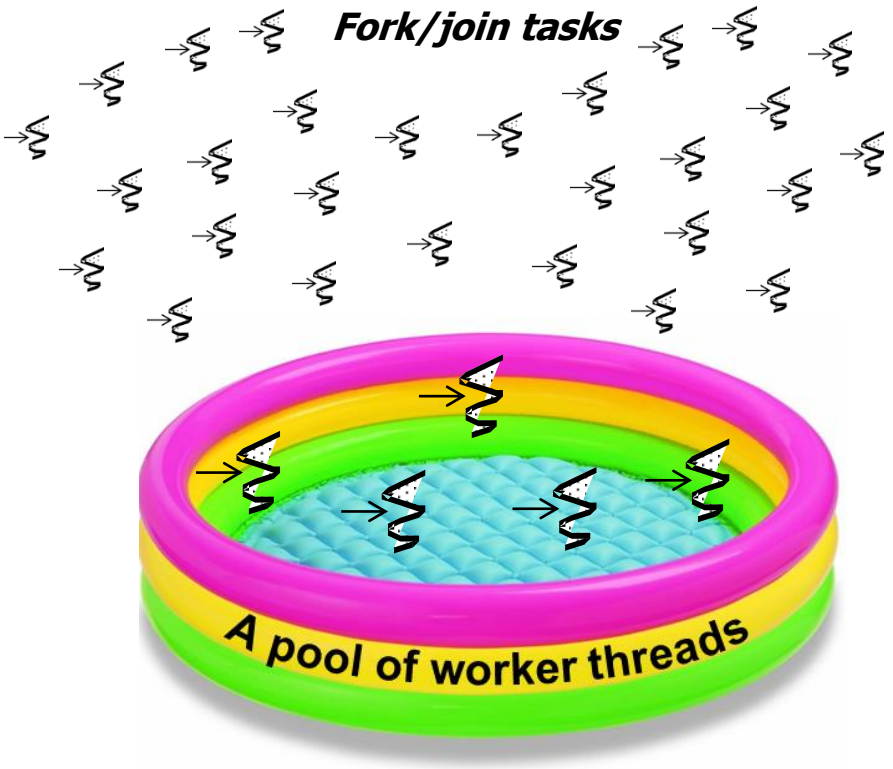


Thread

e.g., it omits its own run-time stack, registers, thread-local storage, etc.

Processing Chunks in Parallel via the Common ForkJoinPool

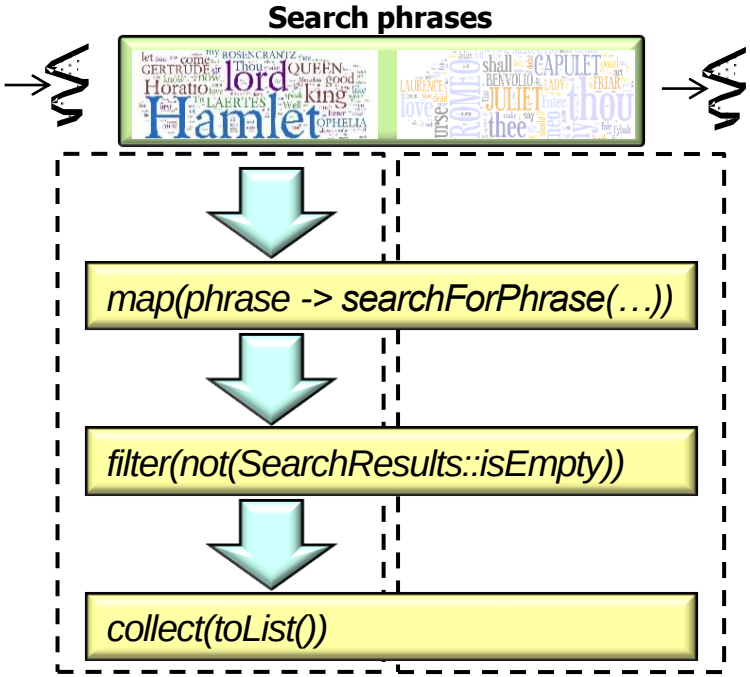
- A fork/join task is similar to—but lighter weight—than a Java thread.
  - A large # of fork/join tasks can thus run in a small # of Java worker threads in a ForkJoinPool.

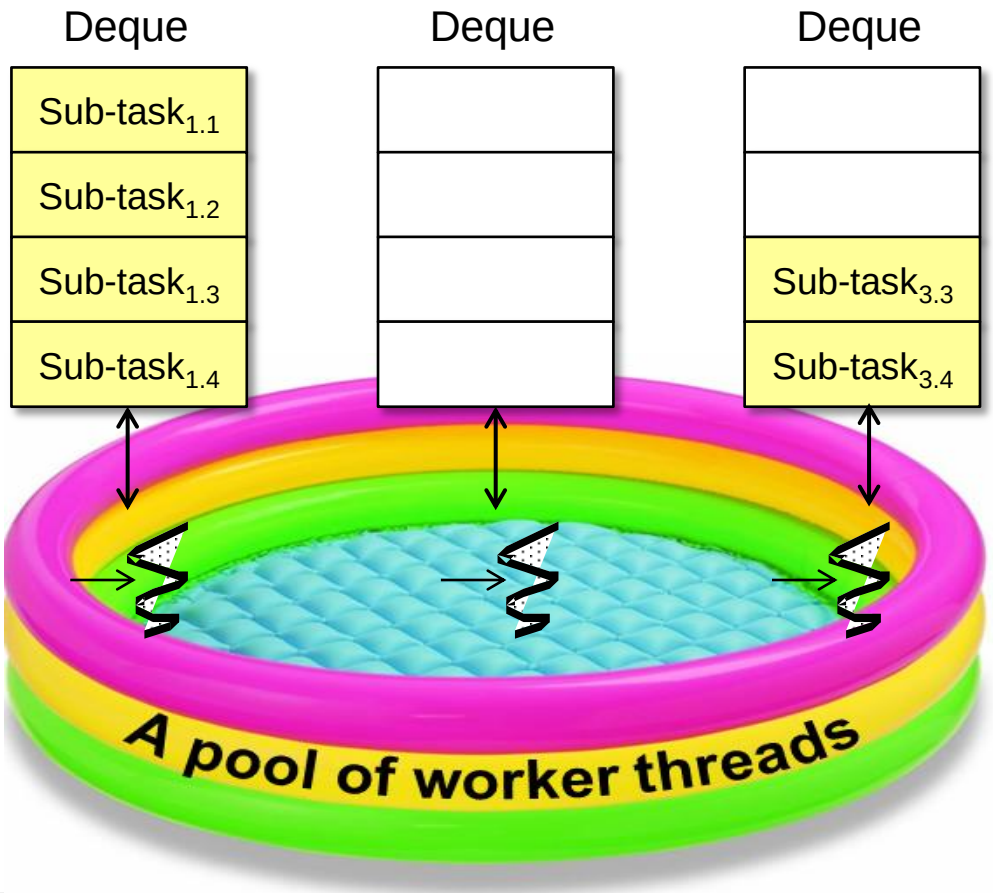


See www.infoq.com/interviews/doug-lea-fork-join

Processing Chunks in Parallel via the Common ForkJoinPool

- Parallel streams are a "user friendly" ForkJoinPool facade.





See en.wikipedia.org/wiki/Facade_pattern

Processing Chunks in Parallel via the Common ForkJoinPool

- You can program directly to the ForkJoinPool API, though it can be somewhat painful!

```
List<List<SearchResults>>
  listOfListOfSearchResults =
    ForkJoinPool.commonPool()
      .invoke(new
         SearchWithfork/join task
           (inputList,
            mPhrasesToFind, ...));
```

I gave you the chance of programming Java streams.

But you have elected the way of pain!

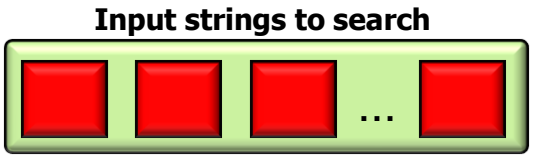
See espressoprogrammer.com/fork-join-vs-parallel-stream-java-8

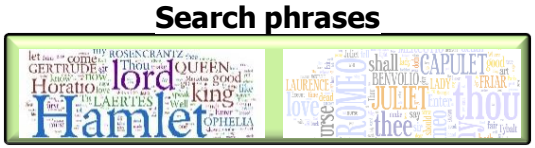
Processing Chunks in Parallel via the Common ForkJoinPool

- You can program directly to the ForkJoinPool API, though it can be somewhat painful!

```
List<List<SearchResults>>
  listOfListOfSearchResults =
    ForkJoinPool.commonPool()
      .invoke(new
        SearchWithfork/join task
          (inputList,
           mPhrasesToFind, ...));
```

Use the common ForkJoinPool to search input strings for phrases that match.





See livelessons/streamgangs/SearchWithForkJoin.java

Processing Chunks in Parallel via the Common ForkJoinPool

- ForkJoinPool is best used for programs that don't match the parallel streams model.

```
Long compute() {
  long count = 0L;
  List<RecursiveTask<Long>> forks =
    new LinkedList<>();
  for (Folder sub : mFolder.getSubs()){
    FolderSearchTask task = new
      FolderSearchTask(sub, mWord);
    forks.add(task); task.fork();
  }
  for (Doc doc : mFolder.getDocs()) {
    DocSearchTask task =
      new DocSearchTask(doc, mWord);
    forks.add(task); task.fork();
  }
  for (RecursiveTask<Long> task : forks)
    count += task.join();
  return count; ...
```

See en.wikipedia.org/wiki/Divide-and-conquer_algorithm

Processing Chunks in Parallel via the Common ForkJoinPool

- ForkJoinPool is best used for programs that don't match the parallel streams model.
  - e.g., this program counts the occurrence of a word in document folders.

```
Long compute() {
  long count = 0L;
  List<RecursiveTask<Long>> forks =
    new LinkedList<>();
  for (Folder sub : mFolder.getSubs()){
    FolderSearchTask task = new
      FolderSearchTask(sub, mWord);
    forks.add(task); task.fork();
  }
  for (Doc doc : mFolder.getDocs()) {
    DocSearchTask task =
      new DocSearchTask(doc, mWord);
    forks.add(task); task.fork();
  }
  for (RecursiveTask<Long> task : forks)
    count += task.join();
  return count; ...
```

See www.oracle.com/technetwork/articles/java/fork-join-422606.html

Processing Chunks in Parallel via the Common ForkJoinPool

- ForkJoinPool is best used for programs that don't match the parallel streams model.
  - e.g., this program counts the occurrence of a word in document folders.

Create a linked list of recursive task objects.

```
Long compute() {
  long count = 0L;
  List<RecursiveTask<Long>> forks =
    new LinkedList<>();
  for (Folder sub : mFolder.getSubs()){
    FolderSearchTask task = new
      FolderSearchTask(sub, mWord);
    forks.add(task); task.fork();
  }
  for (Doc doc : mFolder.getDocs()) {
    DocSearchTask task =
      new DocSearchTask(doc, mWord);
    forks.add(task); task.fork();
  }
  for (RecursiveTask<Long> task : forks)
    count += task.join();
  return count; ...
```

Processing Chunks in Parallel via the Common ForkJoinPool

- ForkJoinPool is best used for programs that don't match the parallel streams model.
  - e.g., this program counts the occurrence of a word in document folders.

Create and fork tasks to search folders recursively.

```
Long compute() {
  long count = 0L;
  List<RecursiveTask<Long>> forks =
    new LinkedList<>();
  for (Folder sub : mFolder.getSubs()){
    FolderSearchTask task = new
      FolderSearchTask(sub, mWord);
    forks.add(task); task.fork();
  }
  for (Doc doc : mFolder.getDocs()) {
    DocSearchTask task =
      new DocSearchTask(doc, mWord);
    forks.add(task); task.fork();
  }
  for (RecursiveTask<Long> task : forks)
    count += task.join();
  return count; ...
```

Processing Chunks in Parallel via the Common ForkJoinPool

- ForkJoinPool is best used for programs that don't match the parallel streams model.
  - e.g., this program counts the occurrence of a word in document folders.

```
Long compute() {
  long count = 0L;
  List<RecursiveTask<Long>> forks =
    new LinkedList<>();
  for (Folder sub : mFolder.getSubs()){
    FolderSearchTask task = new
      FolderSearchTask(sub, mWord);
    forks.add(task); task.fork();
  }
  for (Doc doc : mFolder.getDocs()) {
    DocSearchTask task =
      new DocSearchTask(doc, mWord);
    forks.add(task); task.fork();
  }
  for (RecursiveTask<Long> task : forks)
    count += task.join();
  return count; ...
```

Create and fork tasks to search documents.

Processing Chunks in Parallel via the Common ForkJoinPool

- ForkJoinPool is best used for programs that don't match the parallel streams model.
  - e.g., this program counts the occurrence of a word in document folders.

Join all the tasks together and count the # of search matches.

```
Long compute() {
  long count = 0L;
  List<RecursiveTask<Long>> forks =
    new LinkedList<>();
  for (Folder sub : mFolder.getSubs()){
    FolderSearchTask task = new
      FolderSearchTask(sub, mWord);
    forks.add(task); task.fork();
  }
  for (Doc doc : mFolder.getDocs()) {
    DocSearchTask task =
      new DocSearchTask(doc, mWord);
    forks.add(task); task.fork();
  }
  for (RecursiveTask<Long> task : forks)
    count += task.join();
  return count; ...
```

Processing Chunks in Parallel via the Common ForkJoinPool

- ForkJoinPool is best used for programs that don't match the parallel streams model.
  - e.g., this program counts the occurrence of a word in document folders.

```
Long compute() {
  long count = 0L;
  List<RecursiveTask<Long>> forks =
    new LinkedList<>();
  for (Folder sub : mFolder.getSubs()){
    FolderSearchTask task = new
      FolderSearchTask(sub, mWord);
    forks.add(task); task.fork();
  }
  for (Doc doc : mFolder.getDocs()) {
    DocSearchTask task =
      new DocSearchTask(doc, mWord);
    forks.add(task); task.fork();
  }
  for (RecursiveTask<Long> task : forks)
    count += task.join();
  return count; ...
```

Return the final count.

Processing Chunks in Parallel via the Common ForkJoinPool

- All parallel streams in a process share the common ForkJoinPool.

See dzone.com/articles/common-fork-join-pool-and-streams

Processing Chunks in Parallel via the Common ForkJoinPool

- All parallel streams in a process share the common ForkJoinPool.
  - This helps optimize resource utilization by knowing what cores are used globally in a process.

See dzone.com/articles/common-fork-join-pool-and-streams

Processing Chunks in Parallel via the Common ForkJoinPool

- All parallel streams in a process share the common ForkJoinPool.
  - This helps optimize resource utilization by knowing what cores are used globally in a process.
    - This "global" vs. "local" resource management trade-off is common in computing and other domains.

See blog.tsia.com/blog/local-or-global-resource-management-which-model-is-better

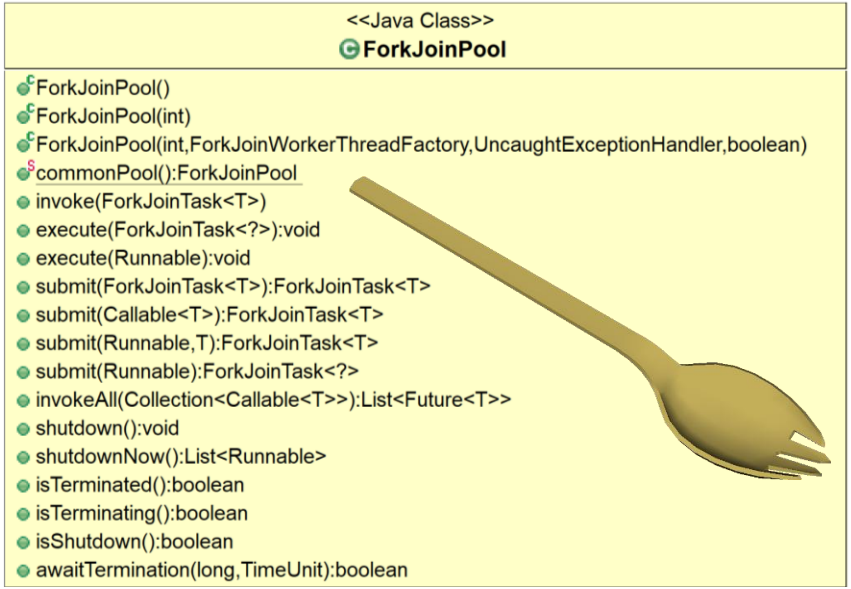
Processing Chunks in Parallel via the Common ForkJoinPool

- All parallel streams in a process share the common ForkJoinPool.
  - This helps optimize resource utilization by knowing what cores are used globally in a process.
  - There are few "knobs" to control this (or any) ForkJoinPool.

See www.infoq.com/presentations/tecniques-parallelism-java

Processing Chunks in Parallel via the Common ForkJoinPool

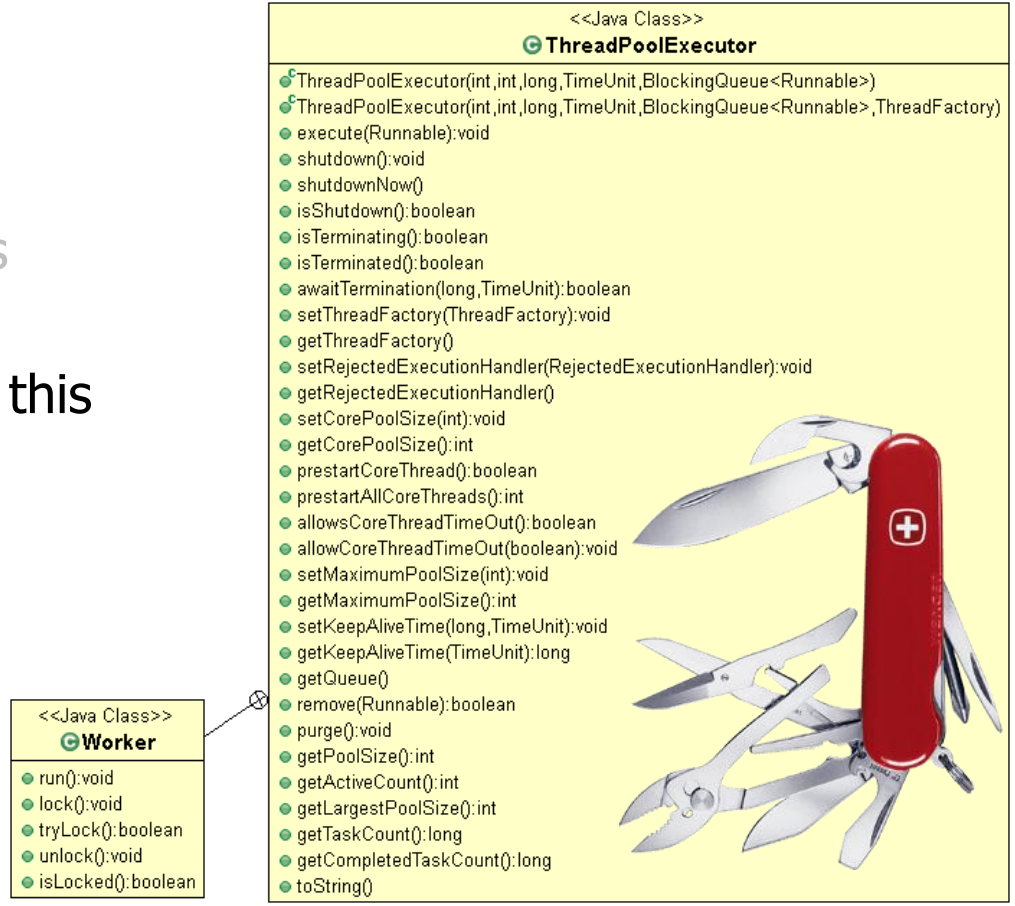
- All parallel streams in a process share the common ForkJoinPool.
  - This helps optimize resource utilization by knowing what cores are used globally in a process.
  - There are few "knobs" to control this (or any) ForkJoinPool.
    - This simplicity is intentional.



See www.youtube.com/watch?v=sq0MX3fHkro

Processing Chunks in Parallel via the Common ForkJoinPool

- All parallel streams in a process share the common ForkJoinPool.
  - This helps optimize resource utilization by knowing what cores are used globally in a process.
  - There are few "knobs" to control this (or any) ForkJoinPool.
    - This simplicity is intentional.
    - Contrast ForkJoinPool with ThreadPoolExecutor.



Processing Chunks in Parallel via the Common ForkJoinPool

- All parallel streams in a process share the common ForkJoinPool.
  - This helps optimize resource utilization by knowing what cores are used globally in a process.
  - There are few "knobs" to control this (or any) ForkJoinPool.
  - However, pool size *can* be configured.

```
System.setProperty
   ("java.util.concurrent"
    + ".ForkJoinPool.common"
    + ".parallelism",
    "8");
```

Set desired # of threads

Interface ForkJoinPool.ManagedBlocker

Enclosing class:
ForkJoinPool

public static interface **ForkJoinPool.ManagedBlocker**

Interface for extending managed parallelism for tasks running in ForkJoinPools.

See upcoming lesson on "*Java Parallel Stream Internals: Configuration.*"

Java Parallel Stream Internals:
Parallel Processing via the Common ForkJoinPool

The End

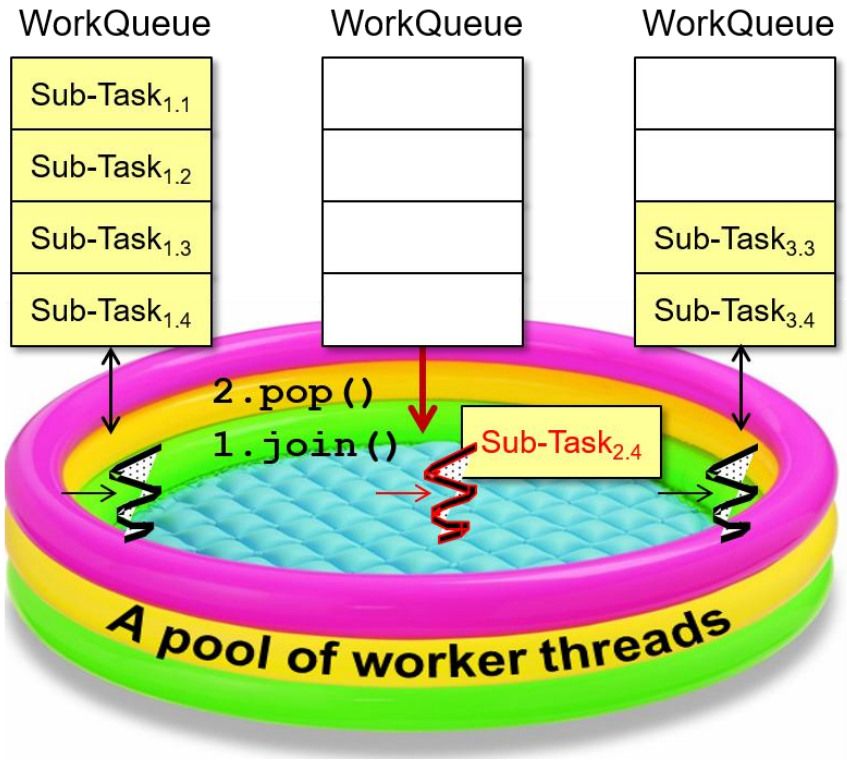
Java Parallel Stream Internals

Mapping Onto the Common ForkJoinPool

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Understand parallel stream internals, e.g.
  - Know what can change and what can't.
  - Partition a data source into "chunks."
  - Process chunks in parallel via the common ForkJoinPool.
    - Recognize how parallel streams are mapped onto the common ForkJoinPool framework.

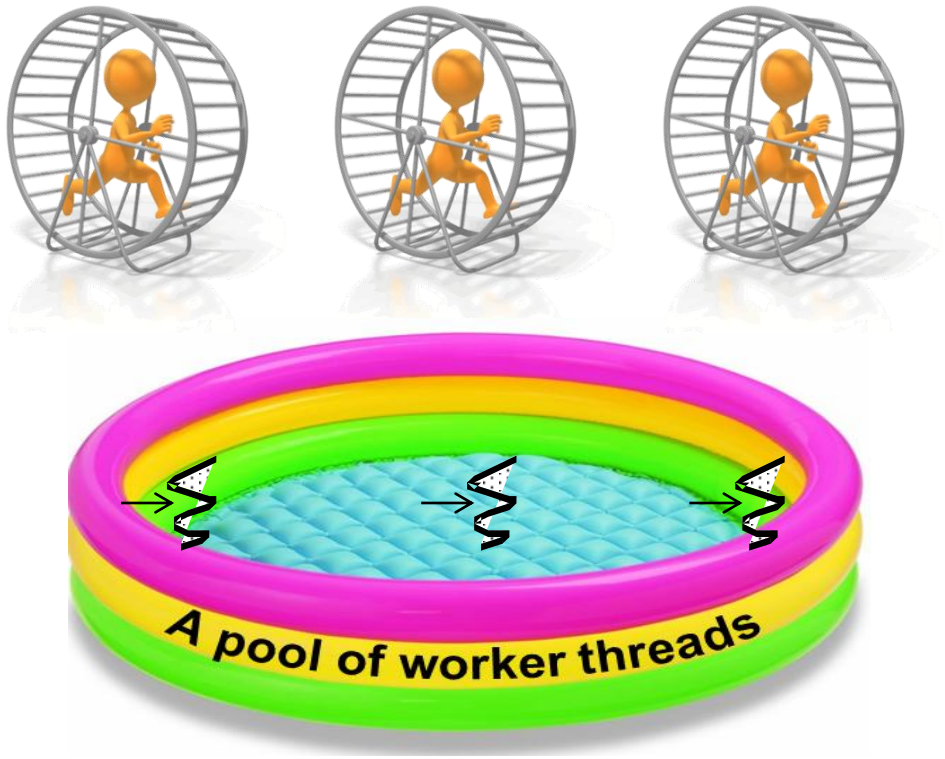


See gee.cs.oswego.edu/dl/papers/fj.pdf

Mapping Parallel Streams Onto the Java ForkJoinPool

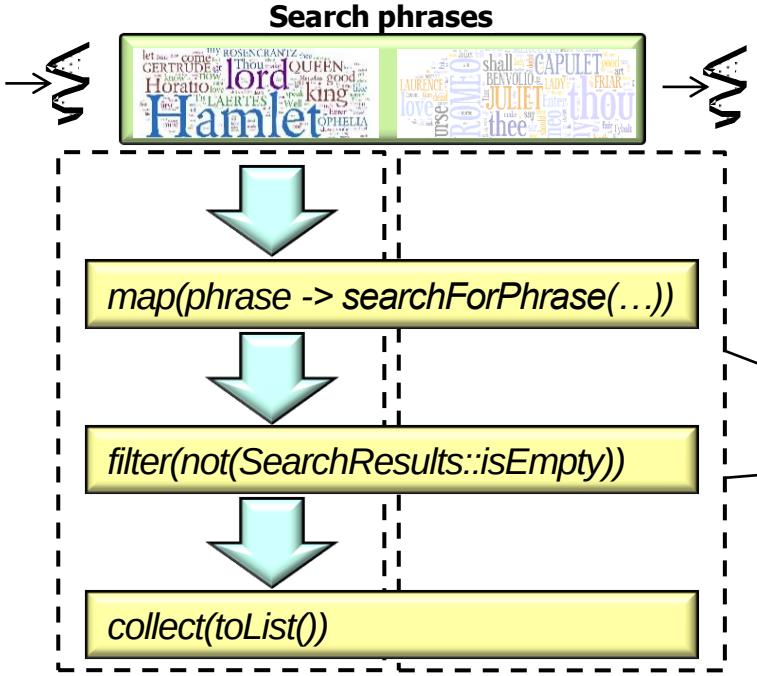
Mapping Parallel Streams Onto the Common ForkJoinPool

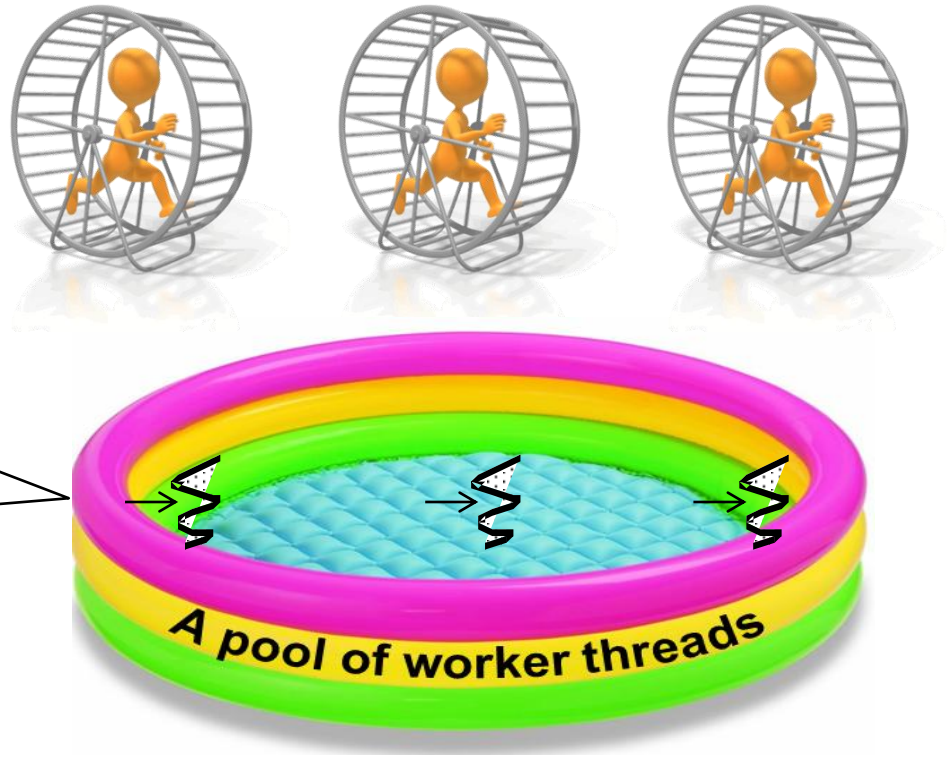
- Each worker thread in the common ForkJoinPool runs a loop scanning for tasks to run.



Mapping Parallel Streams Onto the Common ForkJoinPool

- Each worker thread in the common ForkJoinPool runs a loop scanning for tasks to run.

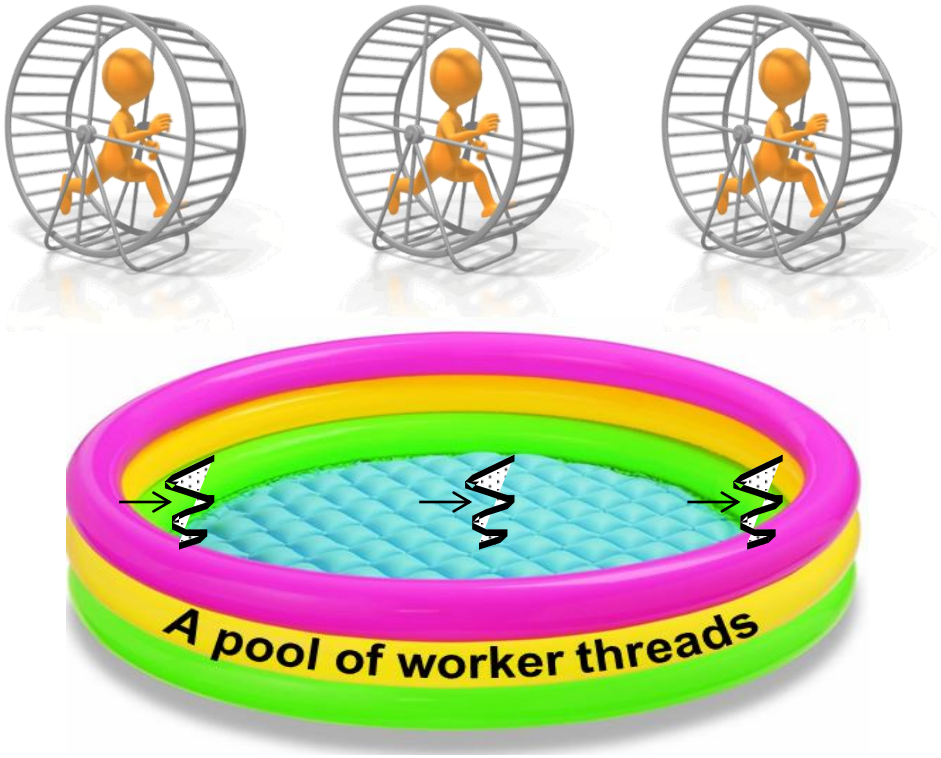




In this lesson, we just care about tasks associated with parallel streams.

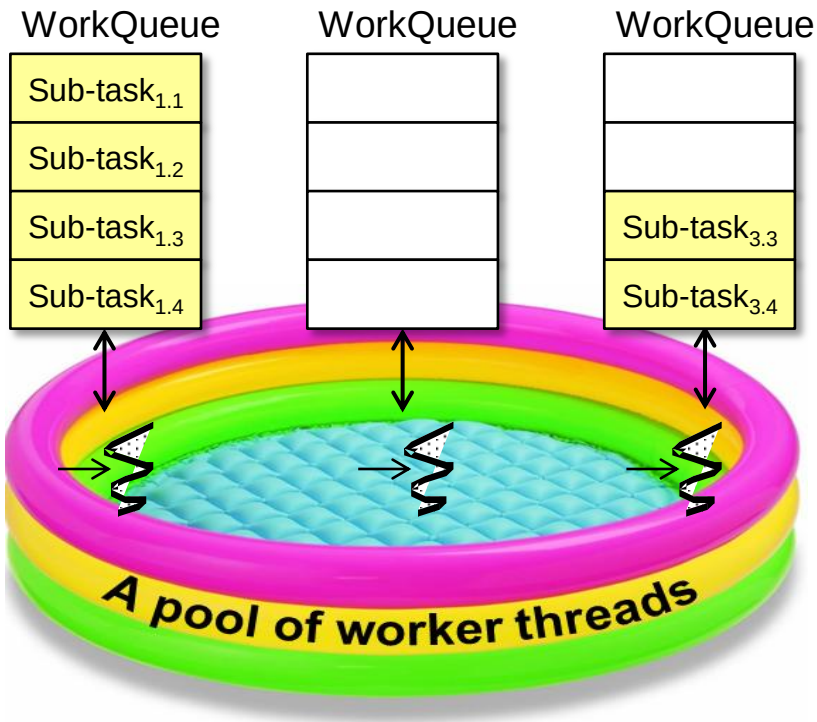
Mapping Parallel Streams Onto the Common ForkJoinPool

- Each worker thread in the common ForkJoinPool runs a loop scanning for tasks to run.
  - The goal is to keep worker threads and cores as busy as possible!



Mapping Parallel Streams Onto the Common ForkJoinPool

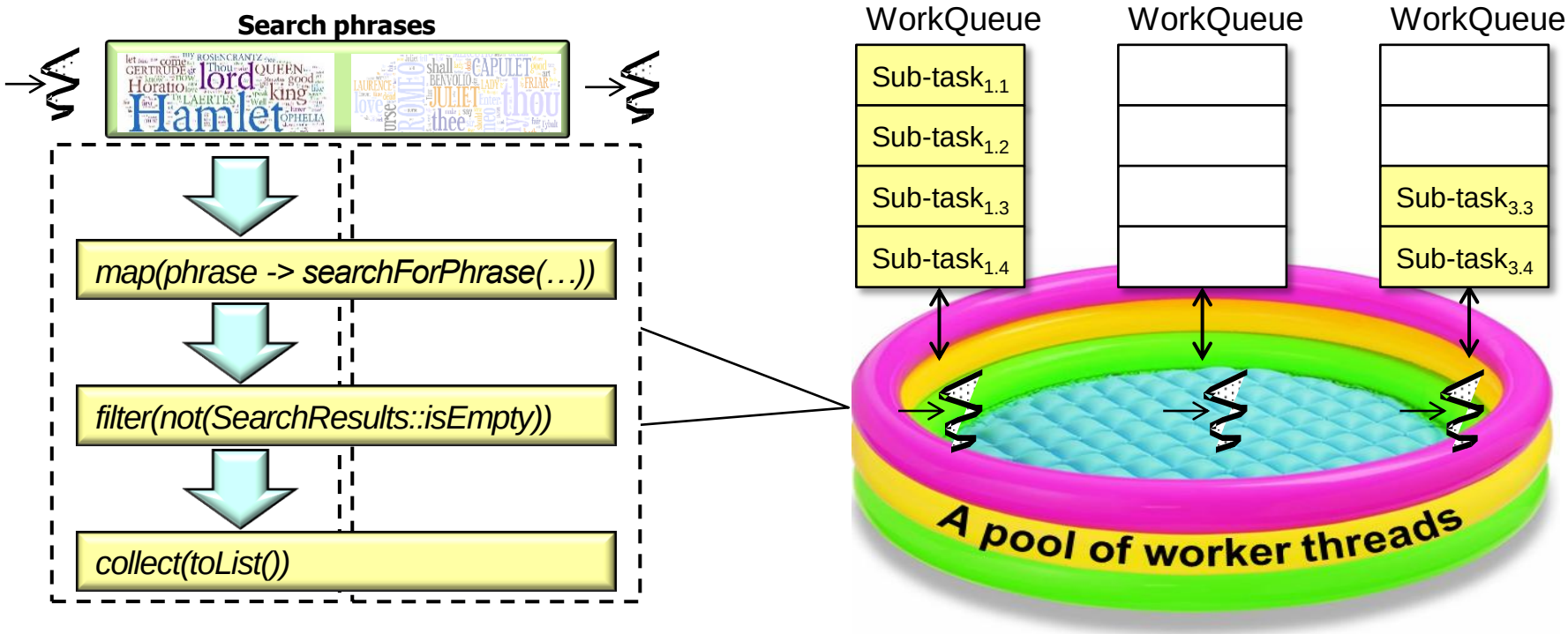
- Each worker thread in the common ForkJoinPool runs a loop scanning for tasks to run.
  - The goal is to keep worker threads and cores as busy as possible!
  - A worker thread has a "double-ended queue" (AKA "deque") that serves as its main source of tasks.



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

Mapping Parallel Streams Onto the Common ForkJoinPool

- The parallel streams framework automatically creates fork/join tasks that are run by worker threads in the common ForkJoinPool.



Mapping Parallel Streams Onto the Common ForkJoinPool

- The AbstractTask super class is used by most fork/join tasks to implement the parallel stream framework.

Manages splitting logic, tracking of child tasks, and intermediate results.

```
abstract class AbstractTask ... {
  public void compute() {
    Spliterator<P_IN> rs = spliterator, ls;
    boolean forkRight = false; ...
    while(... (ls = rs.trySplit()) != null){
      K taskToFork;
      if (forkRight)
      { forkRight = false; ... taskToFork = ...makeChild(rs); }
      else
      { forkRight = true; ... taskToFork = ...makeChild(ls); }
      taskToFork.fork();
    }
  } ...
```

See openjdk/8-b132/java/util/stream/AbstractTask.java

Mapping Parallel Streams Onto the Common ForkJoinPool

- The AbstractTask super class is used by most fork/join tasks to implement the parallel stream framework.

```
abstract class AbstractTask ... {
  public void compute() {
    Spliterator<P_IN> rs = spliterator, ls;
    boolean forkRight = false; ...
    while(... (ls = rs.trySplit()) != null){
      K taskToFork;
      if (forkRight)
      { forkRight = false; ... taskToFork = ...makeChild(rs); }
      else
      { forkRight = true; ... taskToFork = ...makeChild(ls); }
      taskToFork.fork();
    }
  } ...
```

This decides whether to split a task further and/or compute it directly.

Mapping Parallel Streams Onto the Common ForkJoinPool

- The AbstractTask super class is used by most fork/join tasks to implement the parallel stream framework.

```
abstract class AbstractTask ... { ...
  public void compute() {
    Spliterator<P_IN> rs = spliterator, ls;
    boolean forkRight = false; ...
    while(... (ls = rs.trySplit()) != null){
      K taskToFork;
      if (forkRight)
      { forkRight = false; ... taskToFork = ...makeChild(rs); }
      else
      { forkRight = true; ... taskToFork = ...makeChild(ls); }
      taskToFork.fork();
    }
} ...
```

Try to partition the input source until trySplit() returns null.

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

Mapping Parallel Streams Onto the Common ForkJoinPool

- The AbstractTask super class is used by most fork/join tasks to implement the parallel stream framework.

```
abstract class AbstractTask ... { ...
  public void compute() {
    Spliterator<P_IN> rs = spliterator, ls;
    boolean forkRight = false; ...
    while(... (ls = rs.trySplit()) != null){
      K taskToFork;
      if (forkRight)
      { forkRight = false; ... taskToFork = ...makeChild(rs); }
      else
      { forkRight = true; ... taskToFork = ...makeChild(ls); }
      taskToFork.fork();
    }
  } ...
```

Alternate which child is forked to avoid biased Spliterators.

Mapping Parallel Streams Onto the Common ForkJoinPool

- The AbstractTask super class is used by most fork/join tasks to implement the parallel stream framework.

```
abstract class AbstractTask ... { ...
  public void compute() {
    Spliterator<P_IN> rs = spliterator, ls;
    boolean forkRight = false; ...
    while(... (ls = rs.trySplit()) != null){
      K taskToFork;
      if (forkRight)
      { forkRight = false; ... taskToFork = ...makeChild(rs); }
      else
      { forkRight = true; ... taskToFork = ...makeChild(ls); }
      taskToFork.fork();
    }
} ...
```

Fork a new sub-task and continue processing the other in the loop.

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/fork/join task.html#fork

Mapping Parallel Streams Onto the Common ForkJoinPool

- The AbstractTask super class is used by most fork/join tasks to implement the parallel stream framework.

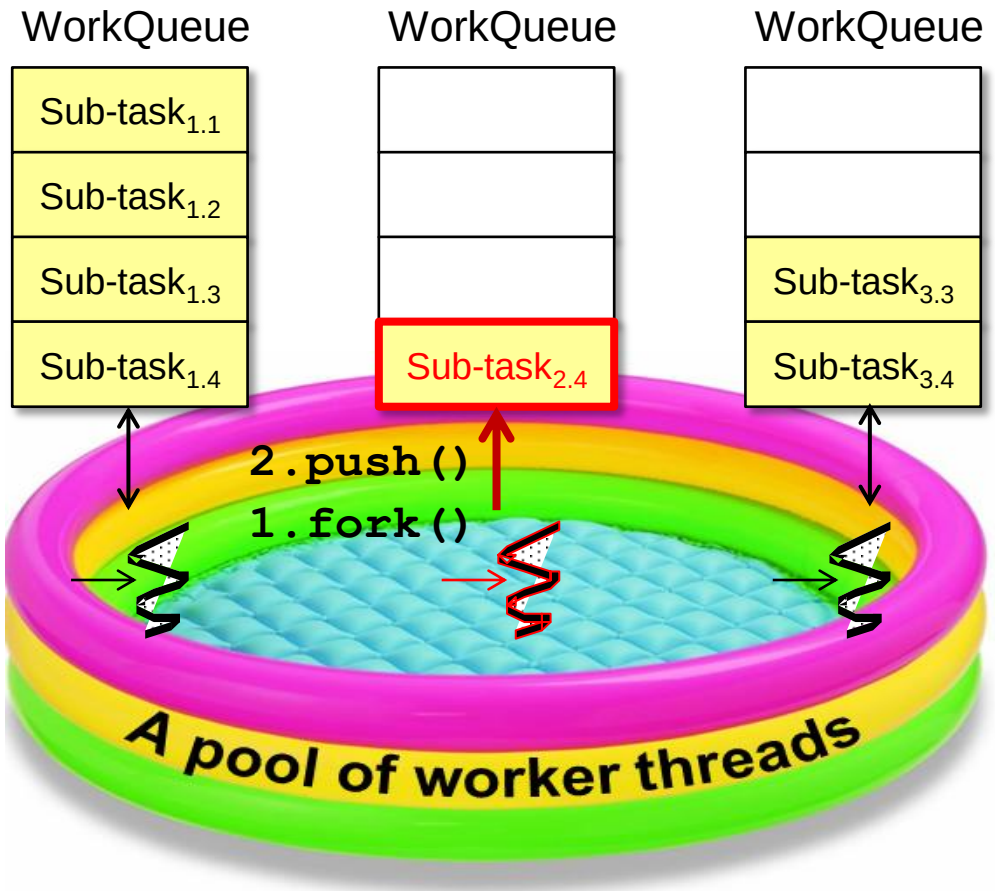
```
abstract class AbstractTask ... { ...
  public void compute() {
    Spliterator<P_IN> rs = spliterator, ls;
    boolean forkRight = false; ...
    while(... (ls = rs.trySplit()) != null){
      ...
    }
    task.setLocalResult(task.doLeaf());
} ...
```

After trySplit() returns null this method typically calls forEachRemaining() to process elements sequentially.

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

Mapping Parallel Streams Onto the Common ForkJoinPool

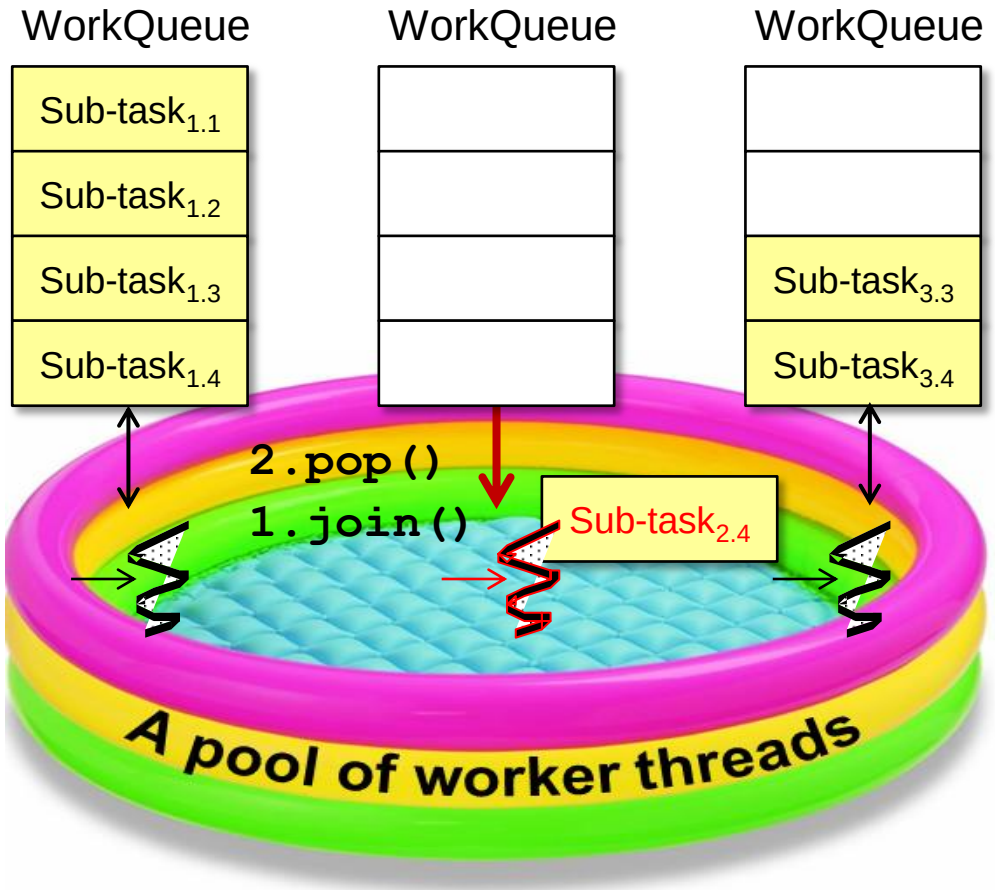
- After the AbstractTask.compute() method calls fork() on a task, this task is pushed onto the head of its worker thread's deque.



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

Mapping Parallel Streams Onto the Common ForkJoinPool

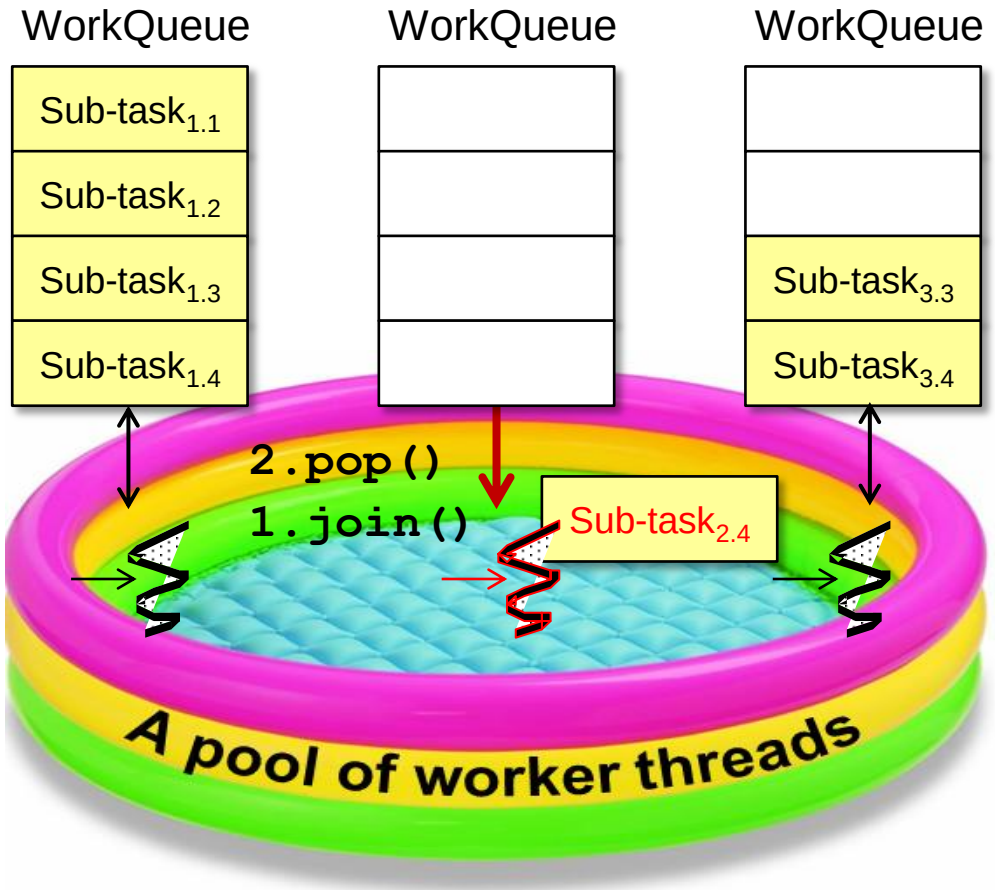
- After the AbstractTask.compute() method calls fork() on a task, this task is pushed onto the head of its worker thread's deque.
  - Each worker thread processes its deque in LIFO order.



See en.wikipedia.org/wiki/Stack_(abstract_data_type)

Mapping Parallel Streams Onto the Common ForkJoinPool

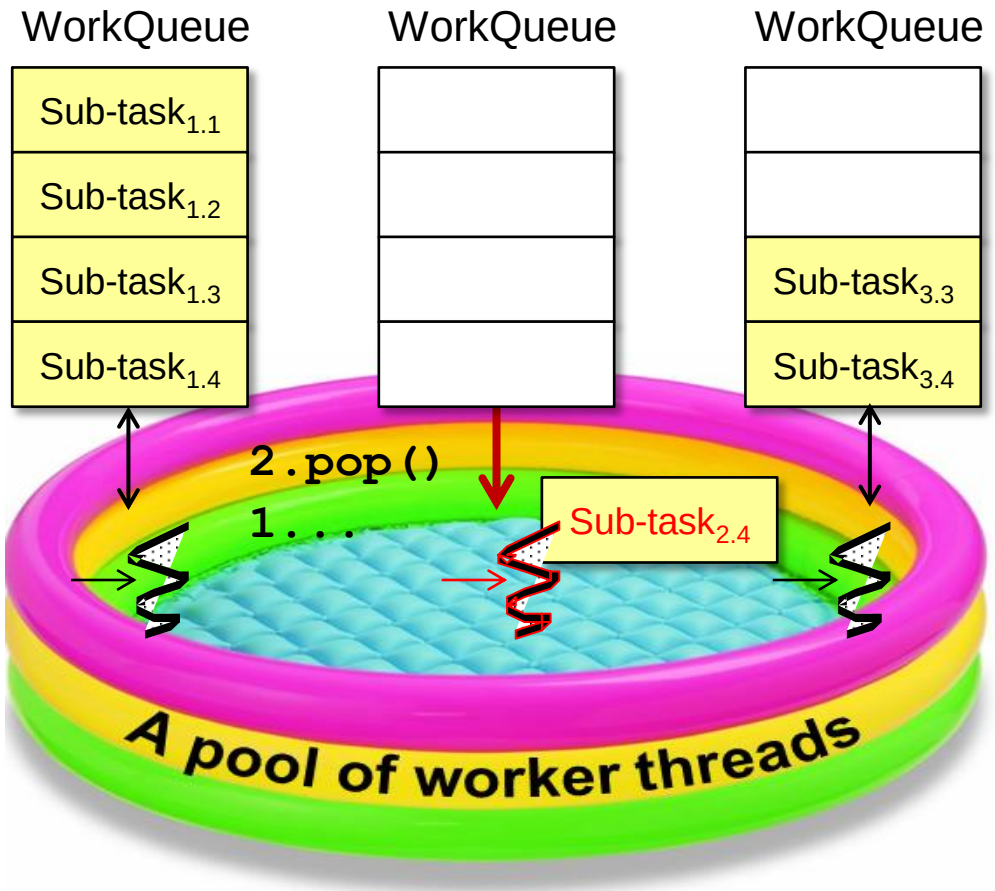
- After the AbstractTask.compute() method calls fork() on a task, this task is pushed onto the head of its worker thread's deque.
  - Each worker thread processes its deque in LIFO order.
    - A task popped from the head of a deque is run to completion.



See en.wikipedia.org/wiki/Run_to_completion_scheduling

Mapping Parallel Streams Onto the Common ForkJoinPool

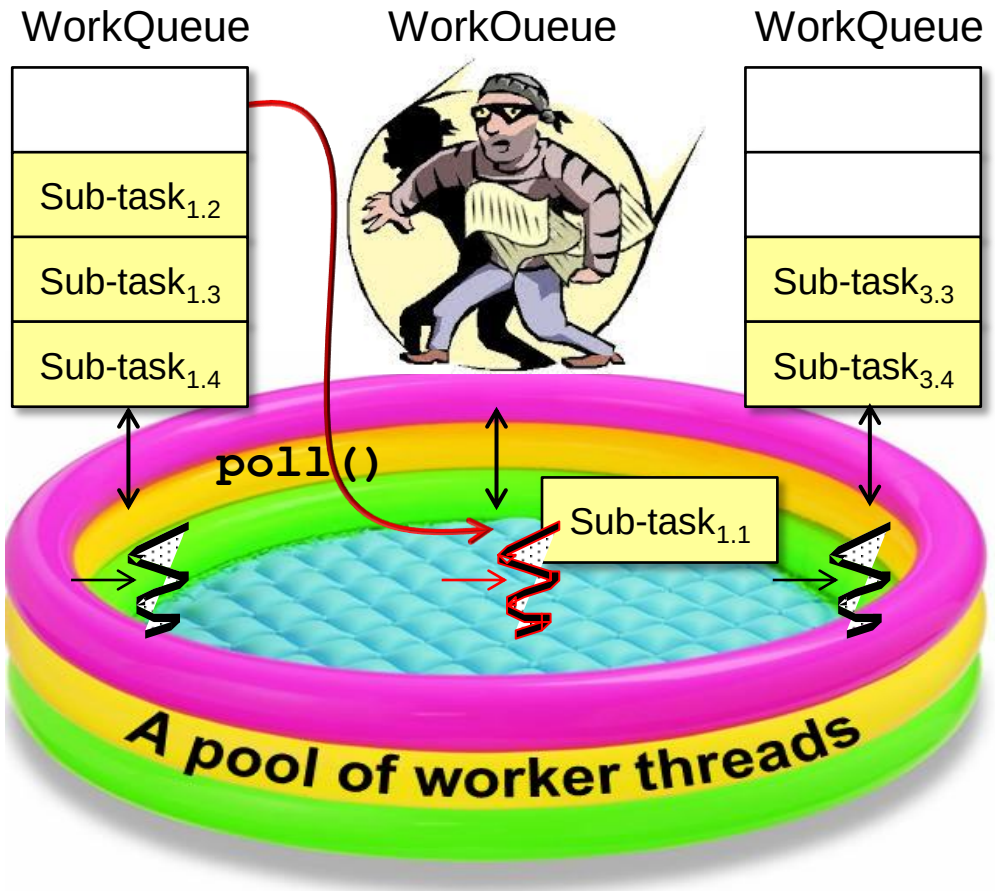
- After the AbstractTask.compute() method calls fork() on a task, this task is pushed onto the head of its worker thread's deque.
  - Each worker thread processes its deque in LIFO order.
  - LIFO order improves locality of reference and cache performance.



See en.wikipedia.org/wiki/Locality_of_reference

Mapping Parallel Streams Onto the Common ForkJoinPool

- To maximize core utilization, idle worker threads "steal" work from the tail of busy threads' deques.



See upcoming lessons on "*The Java Fork-Join Framework.*"

Java Parallel Stream Internals:
Mapping Onto the Common ForkJoinPool

The End

Java Parallel Stream Internals

Configuring the Common ForkJoinPool

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Understand parallel stream internals, e.g.
  - Know what can change and what can't.
  - Partition a data source into "chunks."
  - Process chunks in parallel via the common ForkJoinPool.
  - Configure the Java parallel stream common ForkJoinPool.

```
int desiredThreads = "8";
System.setProperty
  ("java.util.concurrent."
   + "ForkJoinPool.common."
   + "parallelism",
   desiredThreads);
```

Configuring the Parallel Stream Common ForkJoinPool

Configuring the Parallel Stream Common ForkJoinPool

- By default the common ForkJoinPool has one less thread than the # of cores.

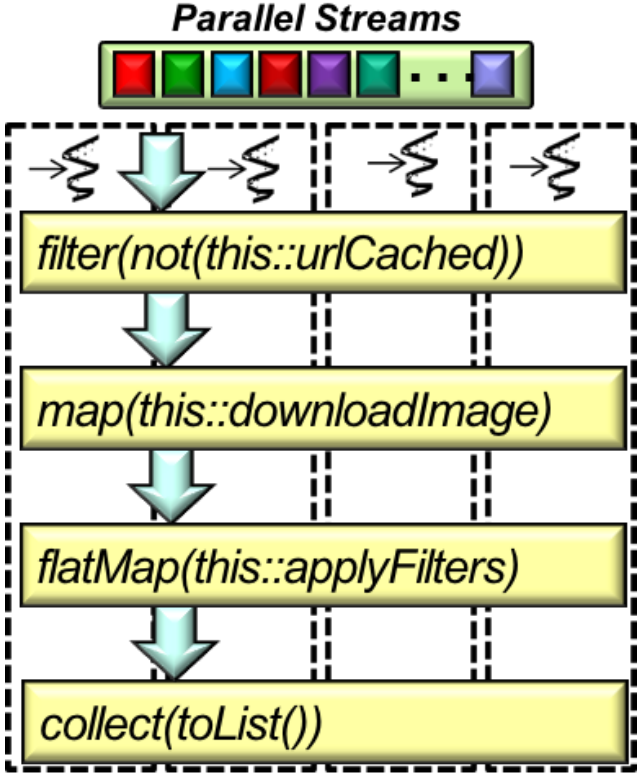
```
System.out.println
   ("The parallelism in the"
    + "common ForkJoinPoolis "
    + ForkJoinPool
        .getCommonPoolParallelism());
```

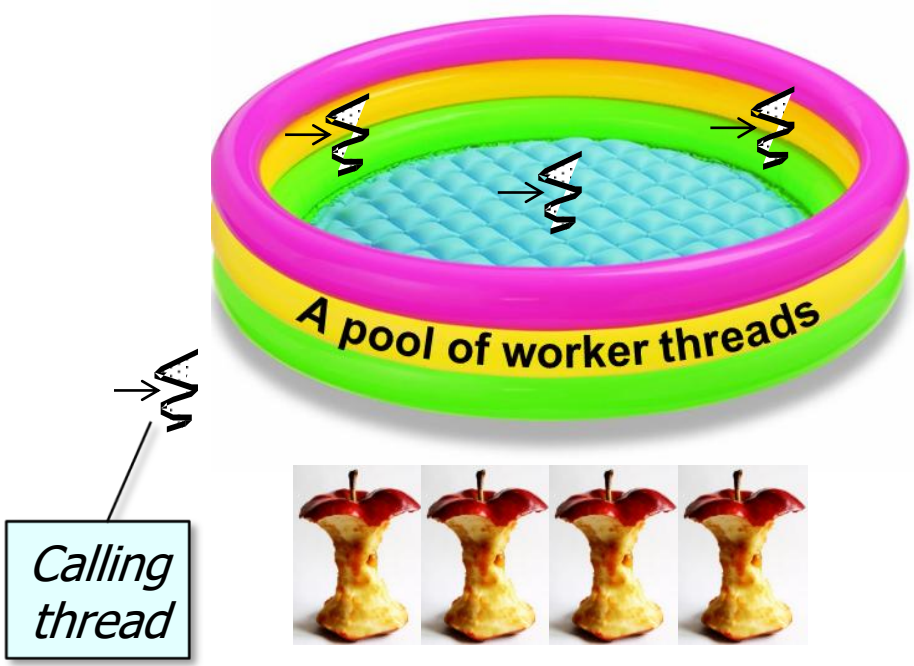
e.g., it returns three on a quad-core processor.

See github.com/douglascraigschmidt/LiveLessons/blob/master/SearchForkJoin

Configuring the Parallel Stream Common ForkJoinPool

- By default the common ForkJoinPool has one less thread than the # of cores.

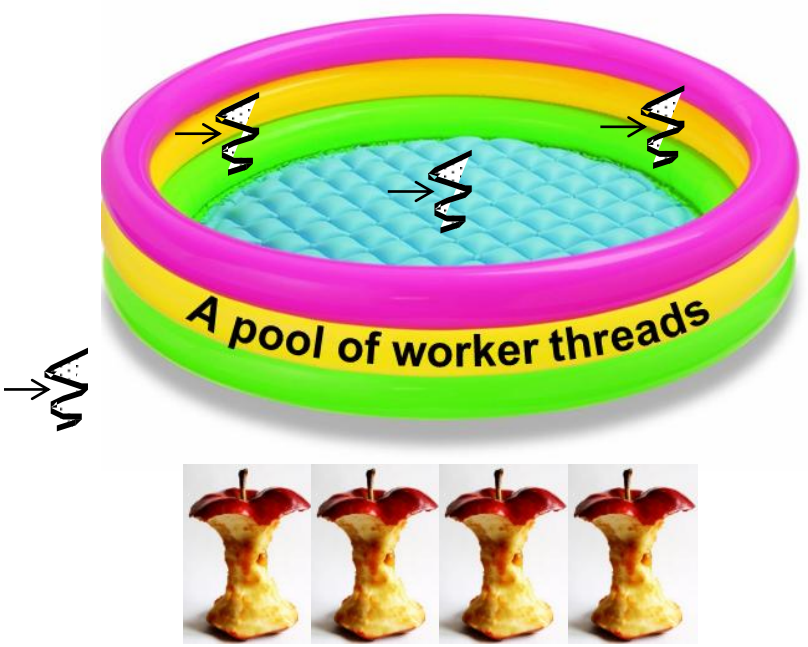




A parallel stream can use all cores since it uses the invoking thread, e.g., main thread.

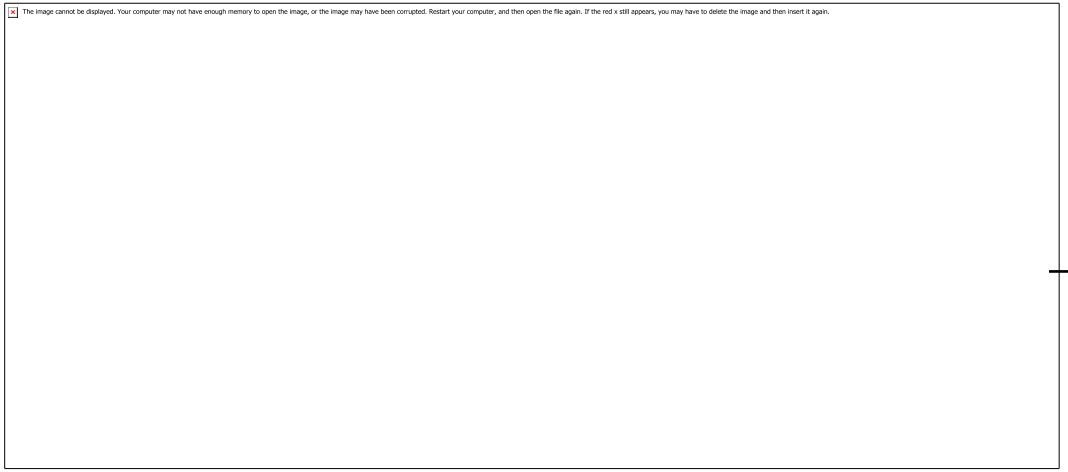
Configuring the Parallel Stream Common ForkJoinPool

- However, the default # of ForkJoinPool threads may be inadequate.



Configuring the Parallel Stream Common ForkJoinPool

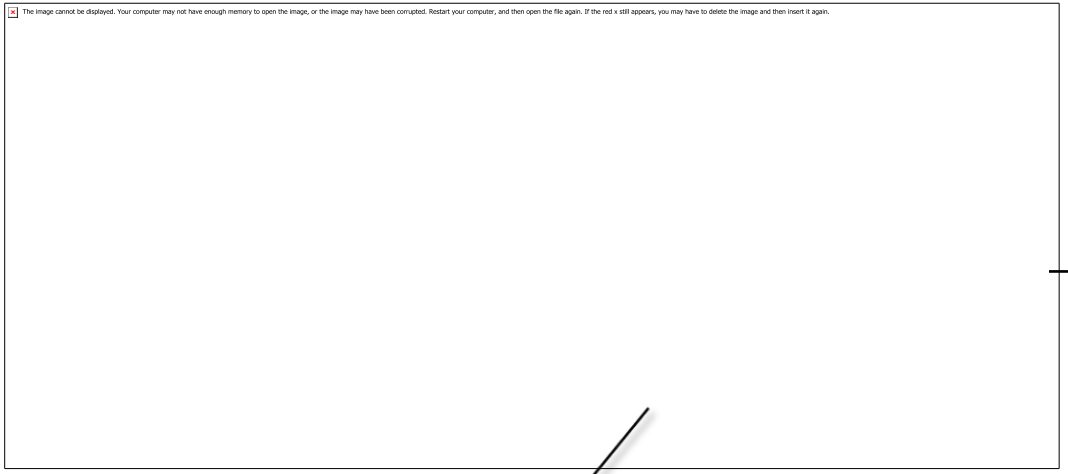
- However, the default # of ForkJoinPool threads may be inadequate, e.g.
  - Consider a parallel image downloading and processing app



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

Configuring the Parallel Stream Common ForkJoinPool

- However, the default # of ForkJoinPool threads may be inadequate, e.g.
  - Consider a parallel image downloading and processing app



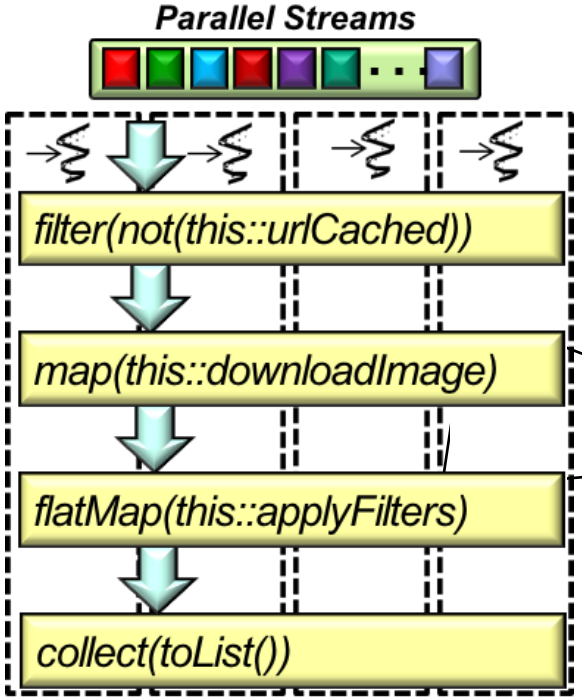
Problems may occur when trying to download more images than # of cores.

These problems may range from underutilization of processor cores to deadlock.

Configuring the Parallel Stream Common ForkJoinPool

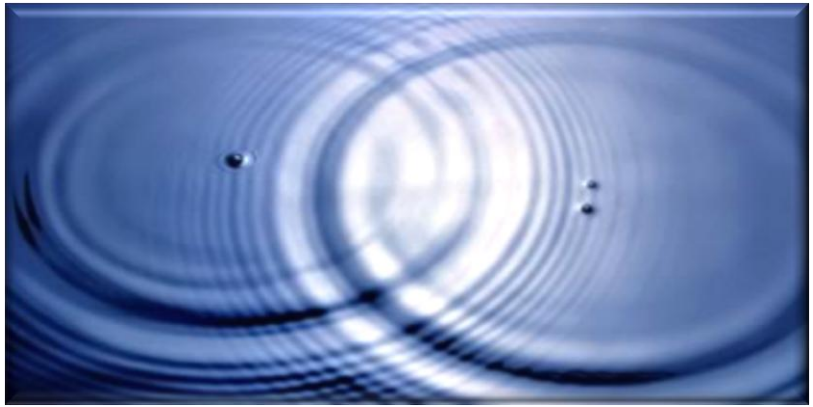
- The common ForkJoinPool size can be controlled programmatically.

```
int desiredThreads = "8";
System.setProperty
   ("java.util.concurrent."
    + "ForkJoinPool.common."
    + "parallelism",
    desiredThreads);
```



Configuring the Parallel Stream Common ForkJoinPool

- The common ForkJoinPool size can be controlled programmatically.
  - Setting this property affects all parallel streams in a process.



```
int desiredThreads = "8";
System.setProperty
   ("java.util.concurrent."
    + "ForkJoinPool.common."
    + "parallelism",
    desiredThreads);
```

It's hard to estimate the total # of threads to set in the common ForkJoinPool.

Configuring the Parallel Stream Common ForkJoinPool

- The common ForkJoinPool size can be controlled programmatically.
  - Setting this property affects all parallel streams in a process.
    - This property can be changed only before the common fork-join pool is initialized.
      - It's initialized "on-demand" the first time it's used.

```
int desiredThreads = "8";
System.setProperty
   ("java.util.concurrent."
    + "ForkJoinPool.common."
    + "parallelism",
    desiredThreads);
```

See en.wikipedia.org/wiki/Lazy_initialization

Configuring the Parallel Stream Common ForkJoinPool

- The common ForkJoinPool size can be controlled programmatically.
  - Setting this property affects all parallel streams in a process.
  - The ManagedBlocker interface can also be used to add worker threads to common ForkJoinPool temporarily.


```
SupplierManagedBlocker<T> mb =
  new SupplierManagedBlocker<>
    (supplier);
...
ForkJoinPool.managedBlock(mb);
...
return mb.getResult();
```

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

Configuring the Parallel Stream Common ForkJoinPool

- The common ForkJoinPool size can be controlled programmatically.
  - Setting this property affects all parallel streams in a process.
  - The ManagedBlocker interface can also be used to add worker threads to common ForkJoinPool temporarily.
    - This is useful for behaviors that block on I/O and/or synchronizers.

```
SupplierManagedBlocker<T> mb =
  new SupplierManagedBlocker<>
    (supplier);
...
ForkJoinPool.managedBlock(mb);
...
return mb.getResult();
```

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

Configuring the Parallel Stream Common ForkJoinPool

- The common ForkJoinPool size can be controlled programmatically.
  - Setting this property affects all parallel streams in a process.
  - The ManagedBlocker interface can also be used to add worker threads to common ForkJoinPool temporarily.
    - This is useful for behaviors that block on I/O and/or synchronizers.
    - This interface can only be used with the common ForkJoinPool.

```
SupplierManagedBlocker<T> mb =
  new SupplierManagedBlocker<>
    (supplier);
...
ForkJoinPool.managedBlock(mb);
...
return mb.getResult();
```

See lessons on "*The Java ForkJoinPool: the ManagedBlocker Interface.*"

Java Parallel Stream Internals:
Configuring the Common ForkJoinPool

The End

Java Parallel Stream Internals

Demo’ing How to Configure the Common Fork-Join Pool

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Understand parallel stream internals, e.g.
  - Know what can change and what can't.
  - Partition a data source into "chunks."
  - Process chunks in parallel via the common ForkJoinPool.
- Configure the Java parallel stream common ForkJoinPool.
  - Know the performance impact of configuring the common fork-join pool size.

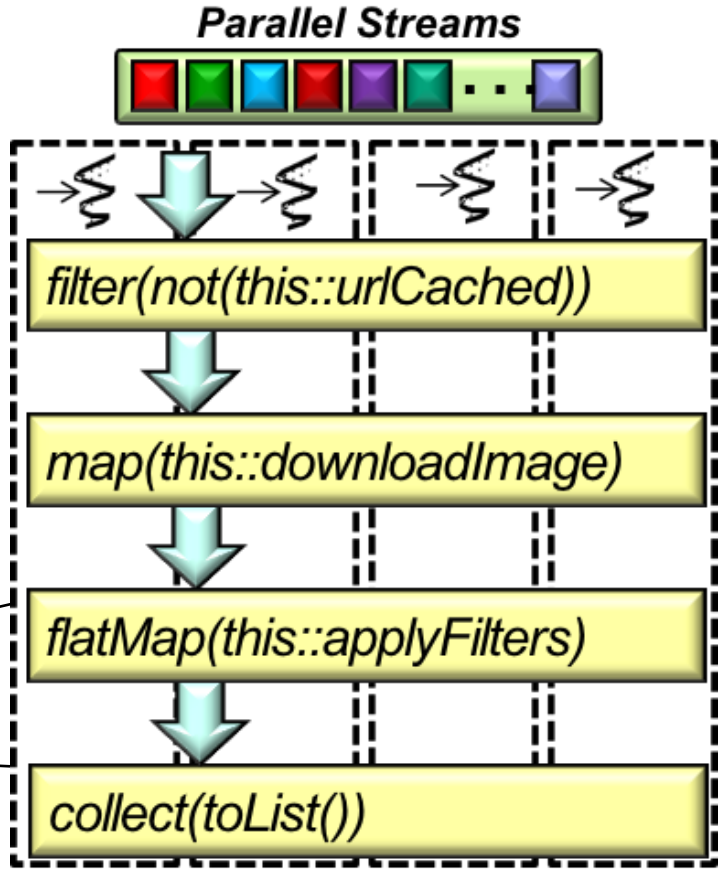
```
Entering the test program with 12 cores
ex20: testDefaultDownloadBehavior() downloaded
      and stored 42 images using 12 threads
      in the pool
ex20: testAdaptiveMBDownloadBehavior()
      downloaded and stored 42 images using
      43 threads in the pool
ex20: testAdaptiveBTDownloadBehavior()
      downloaded and stored 42 images using
      43 threads in the pool

Printing 3 results from fastest to slowest
testAdaptiveBTDownloadBehavior() executed in
                                  3598 msecs
testAdaptiveMBDownloadBehavior() executed in
                                  3910 msecs
testDefaultDownloadBehavior() executed in
                               4104 msecs

Leaving the test program
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex20

Demo’ing Impact of Configuring Common Fork-Join Pool

Demo'ing Impact of Configuring Common Fork-Join Pool

- The common fork-join pool size can be controlled programmatically.



See prior lesson on "*Java Parallel Stream Internals: Configuring the Common Fork-Join Pool*"

Demo'ing Impact of Configuring Common Fork-Join Pool

- The common fork-join pool size can be controlled programmatically.
  - This demo applies ManagedBlocker to add new worker threads to the common fork-join pool


```
File downloadAndStoreImageMB
   (URL url) {
   final Image[] image =
     new Image[1];
   ...
   ForkJoinPool
      .managedBlock(new ForkJoinPool
         .ManagedBlocker() {
          public boolean block() {
            image[0] =
              downloadImage(url);
         return true;
   } ... });

return image[0].store(); ...
```

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

Demo'ing Impact of Configuring Common Fork-Join Pool

- This program shows the performance difference of using ManagedBlocker versus not using ManagedBlocker for an I/O-intensive app.

```
void testDownloadBehavior(Function<URL, File>
                                    downloadAndStoreImage,
                          String testName) {
  ...
  List<File> imageFiles = Options.instance()
    .getUrlList()
    .parallelStream()

    .map(downloadAndStoreImage)

    .collect(Collectors.toList());
  printStats(testName, imageFiles.size()); ...
```

Demo'ing Impact of Configuring Common Fork-Join Pool

- This program shows the performance difference of using ManagedBlocker versus not using ManagedBlocker for an I/O-intensive app.

```
void testDownloadBehavior(Function<URL, File>
                                  downloadAndStoreImage,
                          String testName) {
  ...
  List<File> imageFiles = Options.instance()
    .getUrlList()
    .parallelStream()

    .map(downloadAndStoreImage)

    .collect(Collectors.toList());
  printStats(testName, imageFiles.size()); ...
```

This function param is used to pass different strategies for downloading & storing images from remote websites

See en.wikipedia.org/wiki/Strategy_pattern

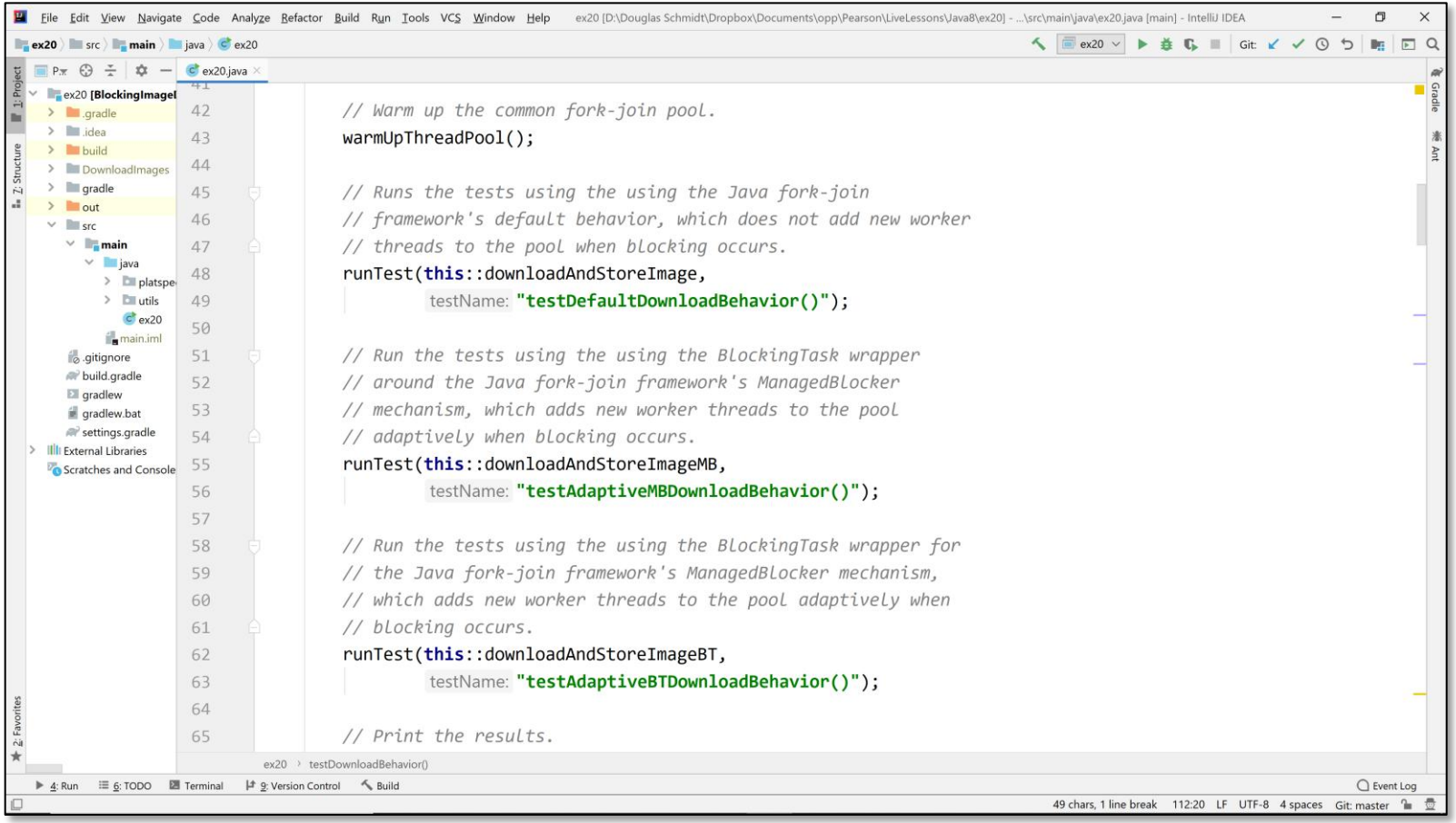
Demo'ing Impact of Configuring Common Fork-Join Pool

- Results show increasing worker threads in the pool improves performance.

```
Entering the test program with 12 cores
ex20: testDefaultDownloadBehavior() downloaded and stored 42 images
      using 12 threads in the pool
ex20: testAdaptiveMBDownloadBehavior() downloaded and stored 42 images
      using 43 threads in the pool
ex20: testAdaptiveBTDownloadBehavior() downloaded and stored 42 images
      using 43 threads in the pool

Printing 3 results from fastest to slowest
testAdaptiveBTDownloadBehavior() executed in 3598 msecs
testAdaptiveMBDownloadBehavior() executed in 3910 msecs
testDefaultDownloadBehavior() executed in 4104 msecs

Leaving the test program
```

See upcoming lessons on "*The Java Fork-Join Pool: the ManagedBlocker Interface*"

Demo'ing Impact of Configuring Common Fork-Join Pool



See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex20

Demo'ing Impact of Configuring Common Fork-Join Pool

[Source code analysis video goes here!]

Java Parallel Stream Internals:
Demo’ing How to Configure the Common Fork-Join Pool

The End

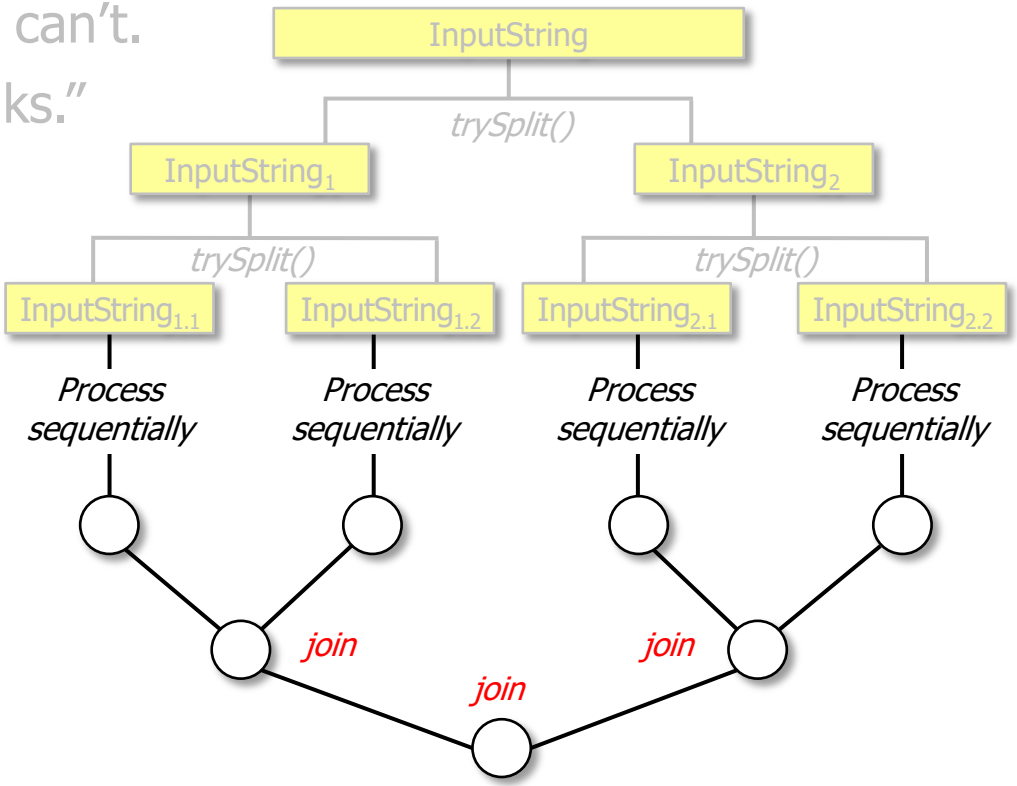
Java Parallel Stream Internals

Combining Results (Part I)

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Understand parallel stream internals, e.g.
  - Know what can change and what can't.
  - Partition a data source into "chunks."
  - Process chunks in parallel via the common ForkJoinPool.
  - Configure the Java parallel stream common ForkJoinPool.
  - Perform a reduction to combine partial results into a single result.

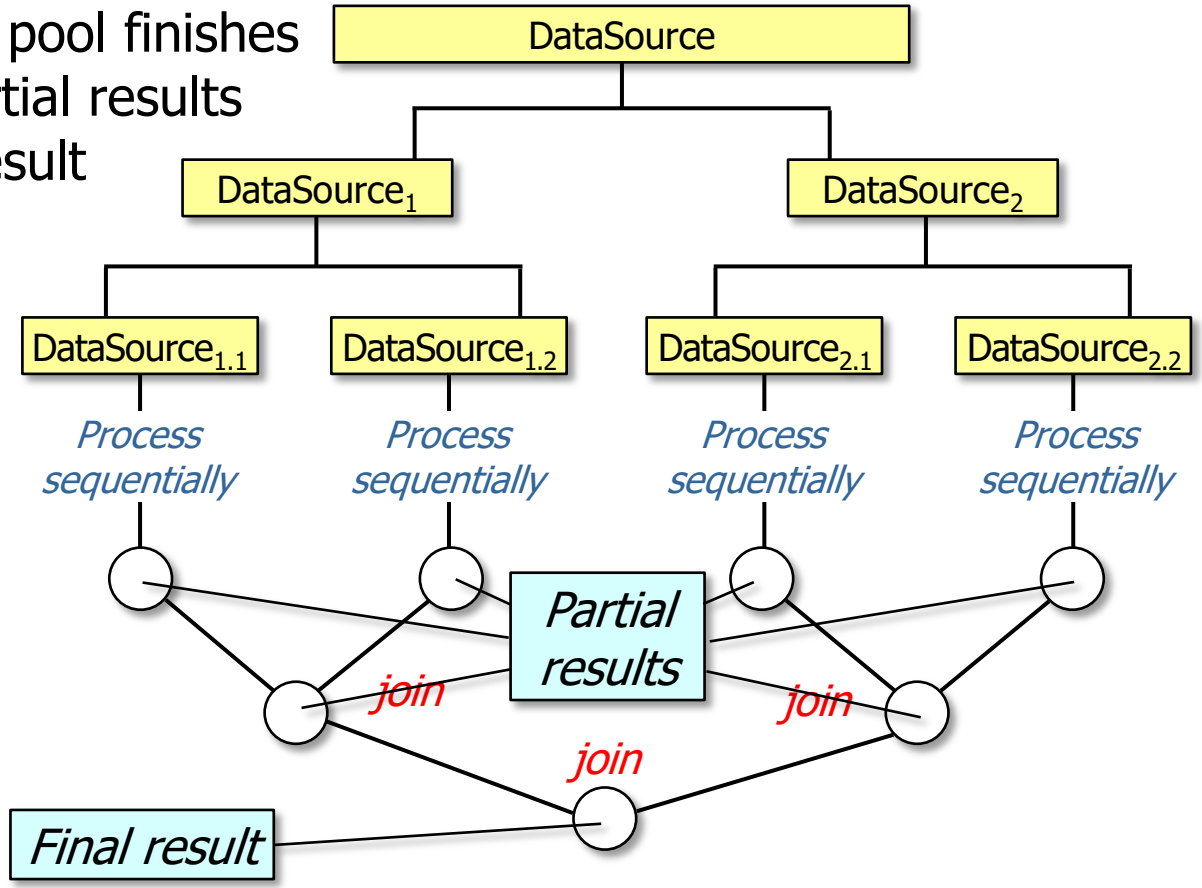


See www.ibm.com/developerworks/library/j-java-streams-3-brian-goetz

Combining Results in a Parallel Stream

Combining Results in a Parallel Stream

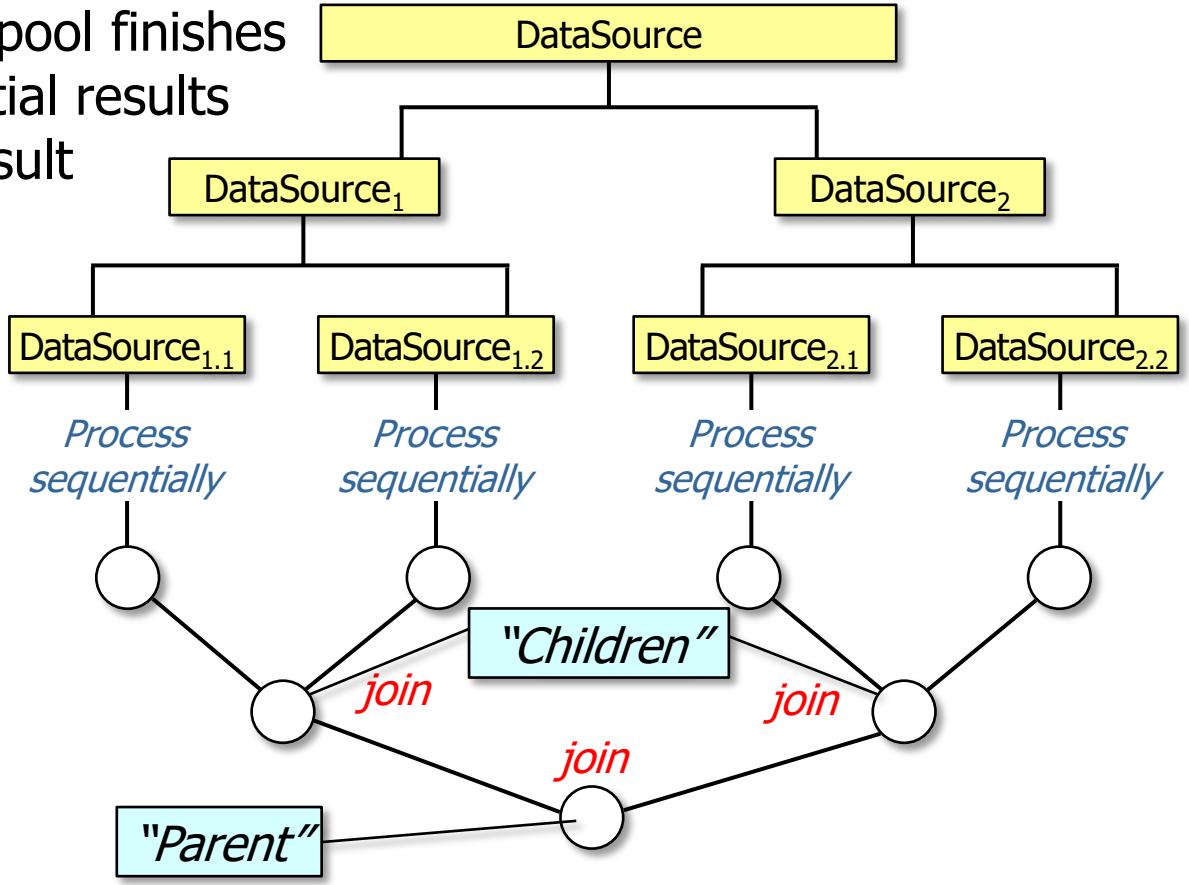
- After the common fork-join pool finishes processing chunks their partial results are combined into a final result



This discussion assumes a non-concurrent collector (other discussions follow)

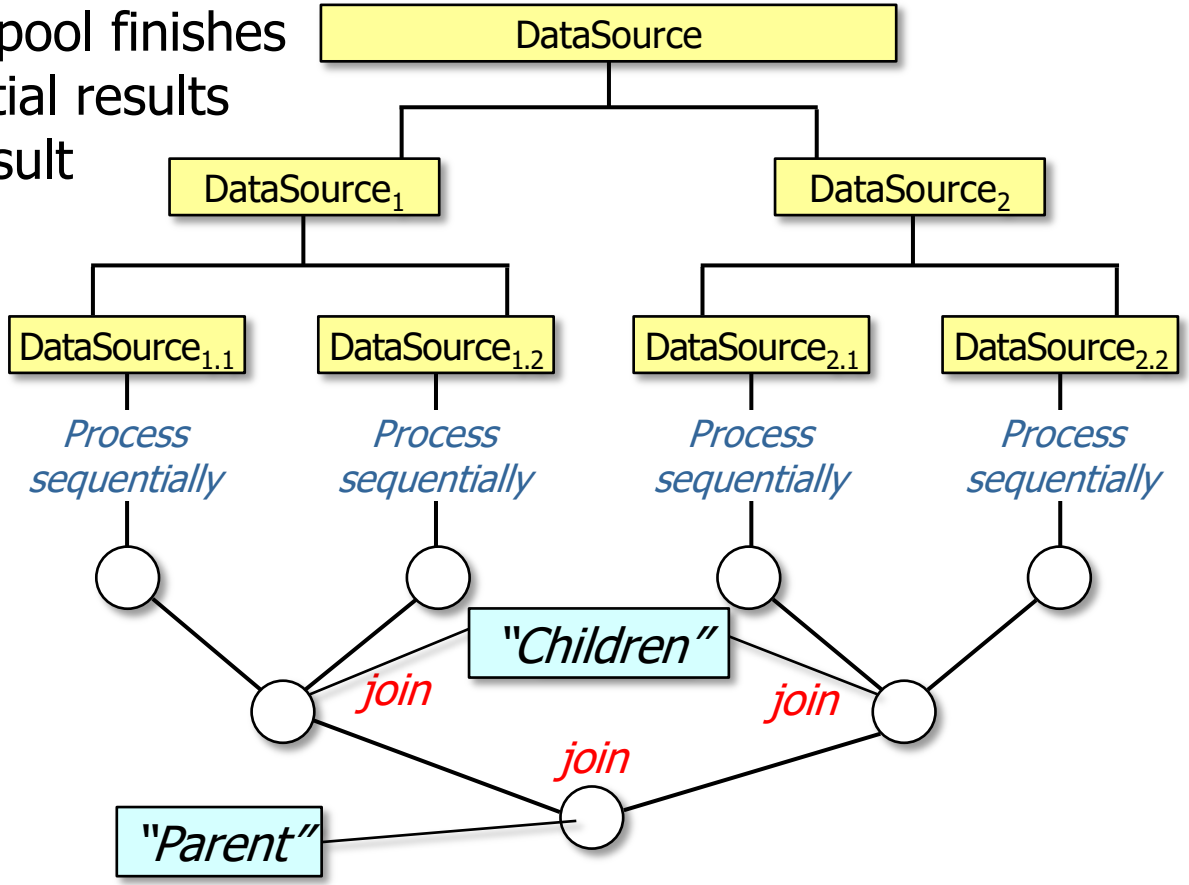
Combining Results in a Parallel Stream

- 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"



Combining Results in a Parallel Stream

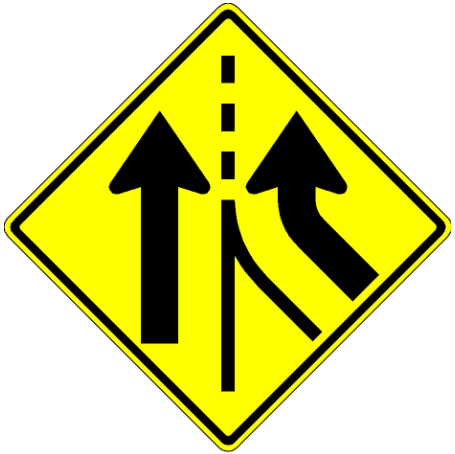
- After the common fork-join pool finishes processing chunks their partial results are combined into a final result
  - join() occurs in a single thread at each level
    - i.e., the "parent"



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

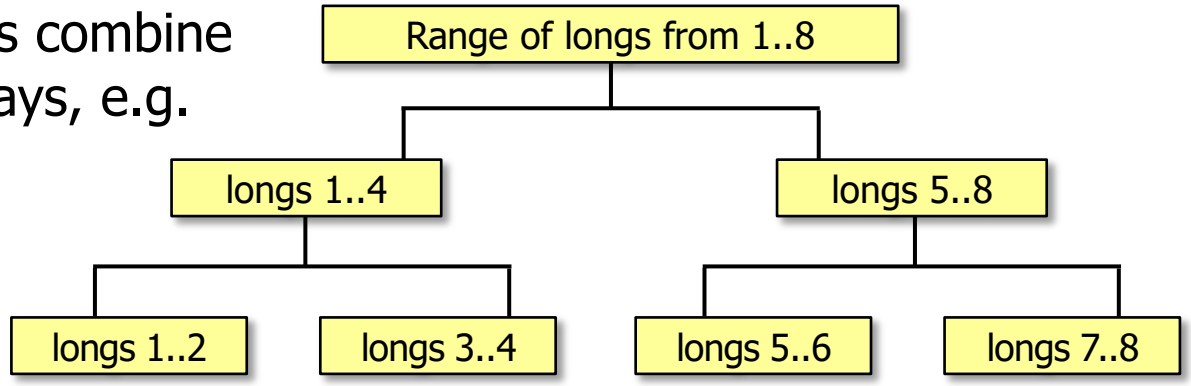
Combining Results in a Parallel Stream

- 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



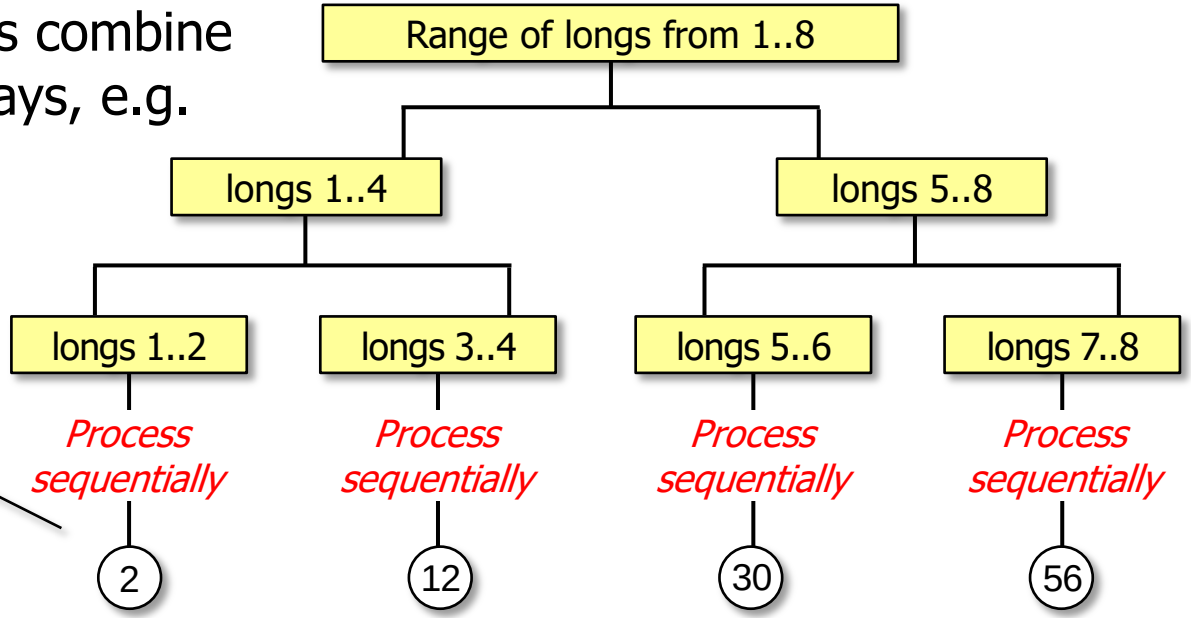

```
long factorial(long n) {
  return LongStream
    .rangeClosed(1, n)
    .parallel()
    .reduce(1, (a, b) -> a * b,
               (a, b) -> a * b);
}
```

Generate a range of longs from 1..8 in parallel

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



Multiply pair-wise values

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

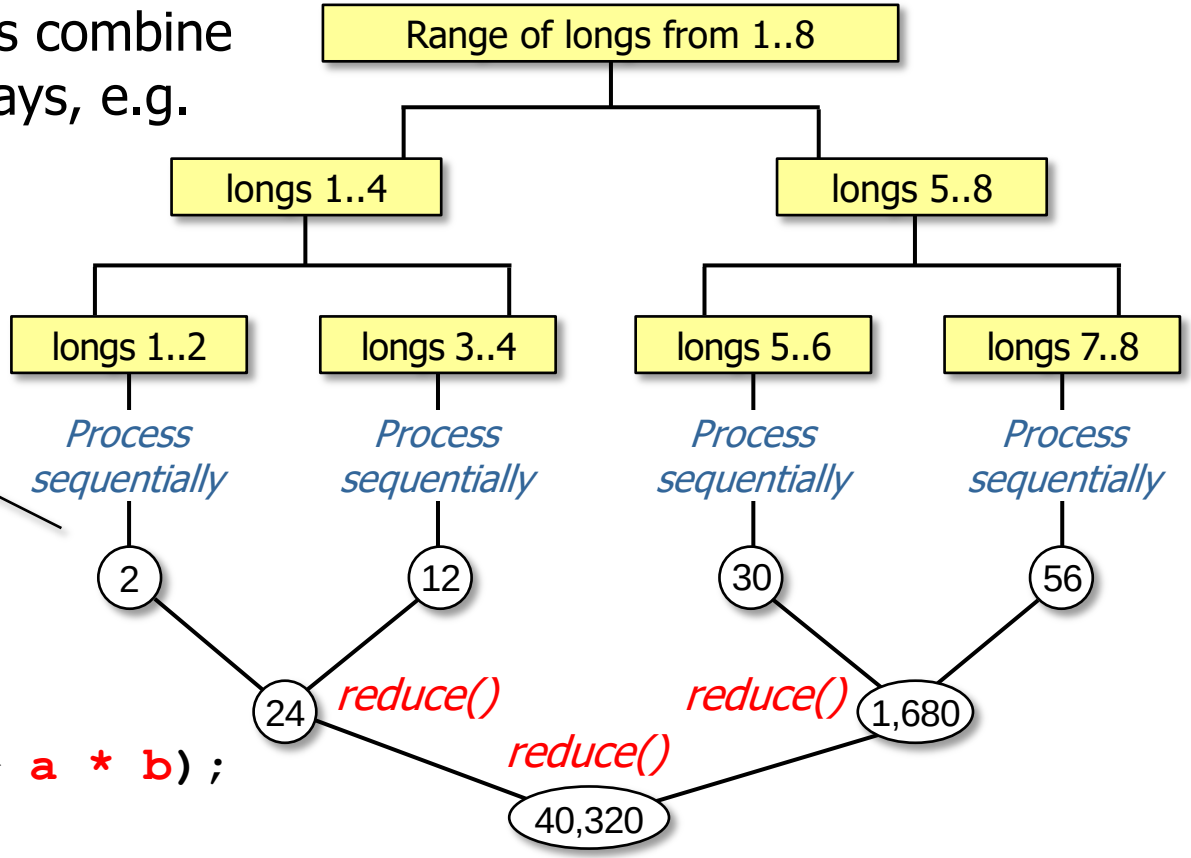
See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex16

Combining Results in a Parallel Stream

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

Multiply pair-wise values

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



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

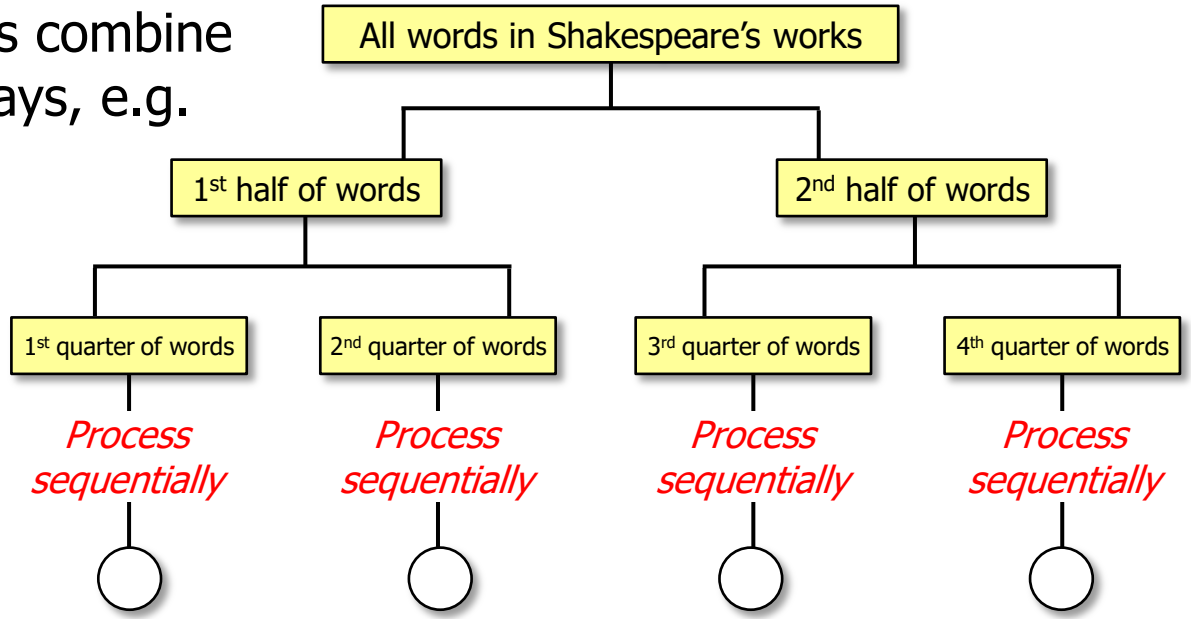
Combining Results in a Parallel Stream

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

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 results container

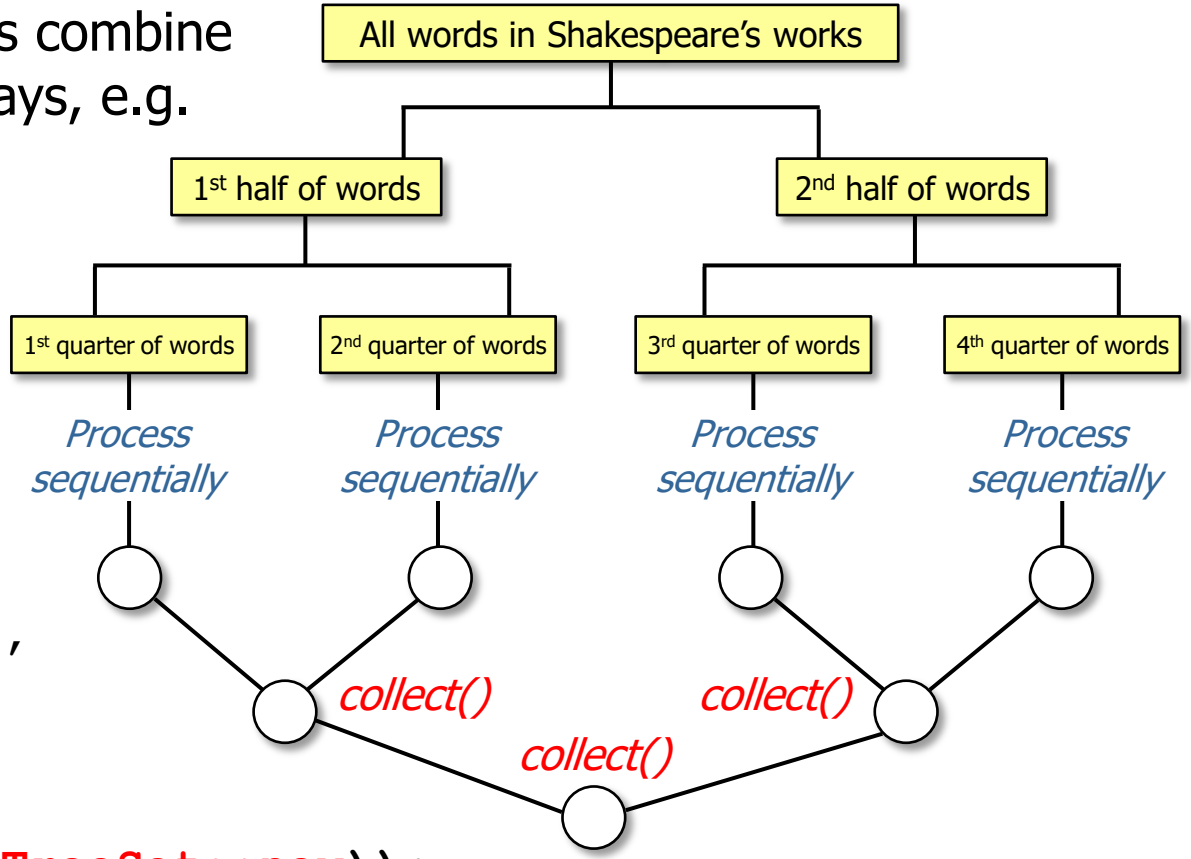



```
Set<CharSequence>
      uniqueWords =
  getInput(sSHAKESPEARE),
           "\\s+")
  .parallelStream()
  ...
  .collect(toCollection(TreeSet::new));
```

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 results container

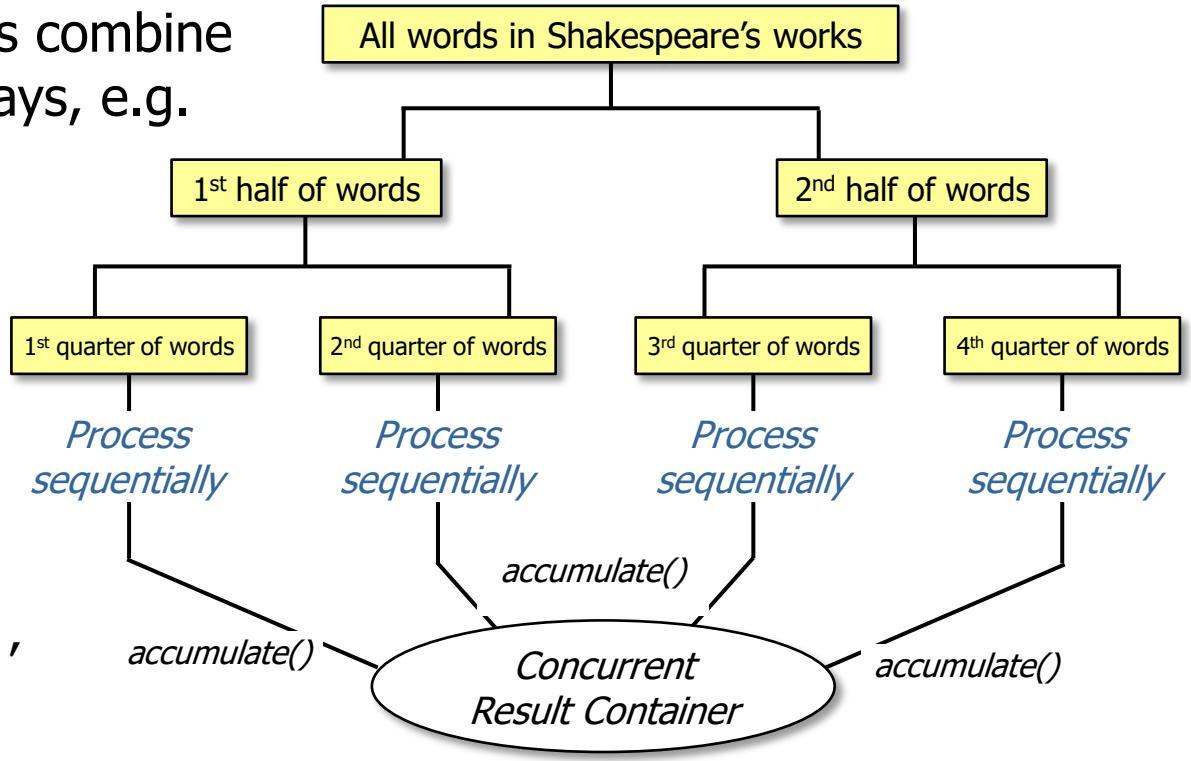



```
Set<CharSequence>
      uniqueWords =
  getInput(sSHAKESPEARE),
           "\\s+")
  .parallelStream()
  ...
  .collect(toCollection(TreeSet::new));
```

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

Combining Results in a Parallel Stream

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




```
Set<CharSequence>
      uniqueWords =
  getInput(sSHAKESPEARE),
           "\\s+")
  .parallelStream()
  ...
  .collect(ConcurrentHashSetCollector.toSet());
```

Concurrent collectors (covered later) are different than non-concurrent collectors

Java Parallel Stream Internals:
Combining Results (Part I)

The End

Java Parallel Stream Internals

Combining Results (Part II)

Douglas C. Schmidt

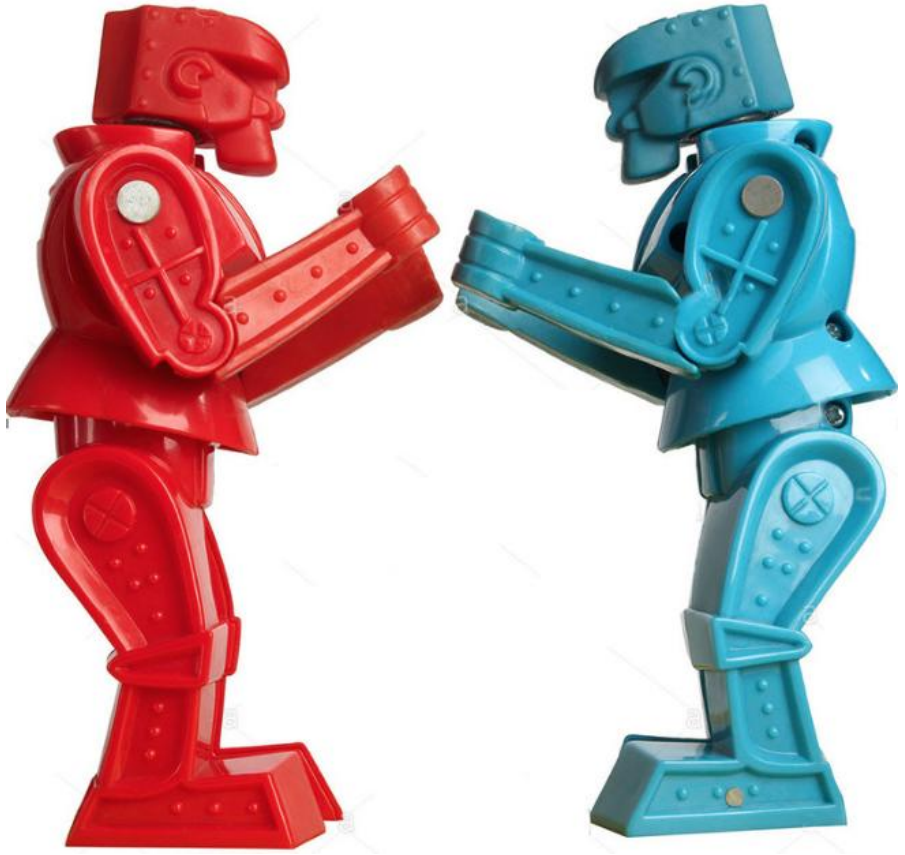
Learning Objectives in This Part of the Lesson

- Understand parallel stream internals, e.g.
  - Know what can change and what can't.
  - Partition a data source into "chunks."
  - Process chunks in parallel via the common ForkJoinPool.
  - Configure the Java parallel stream common ForkJoinPool.
  - Perform a reduction to combine partial results into a single result.
    - Be aware of common traps and pitfalls with parallel streams.

Differences for collect() and reduce() in a Parallel Stream

Differences for collect() and reduce() in a Parallel Stream

- It's important to understand the semantic differences between collect() and reduce()



Differences for collect() and reduce() in a Parallel Stream

- It's important to understand the semantic differences between collect() and reduce(), e.g.
  - Always test w/a parallel stream to detect mistakes wrt mutable vs. immutable reductions

```
void buggyStreamReduce3
        (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

Differences for collect() and reduce() in a Parallel Stream

- It's important to understand the semantic differences between collect() and reduce(), e.g.
  - Always test w/a parallel stream to detect mistakes wrt mutable vs. immutable reductions

Convert a list of words into a stream of works

```
void buggyStreamReduce3
        (boolean parallel) {
...
Stream<String> wordStream =
  allWords.stream();

if (parallel)
  wordStream.parallel();

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

Naturally, this call doesn't really do any work since streams are "lazy"

Differences for collect() and reduce() in a Parallel Stream

- It's important to understand the semantic differences between collect() and reduce(), 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 buggyStreamReduce3
        (boolean parallel) {
  ...
  Stream<String> wordStream =
    allWords.stream();

  if (parallel)
    wordStream.parallel();

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

See docs.oracle.com/javase/8/docs/api/java/util/stream/BaseStream.html#parallel

Differences for collect() and reduce() in a Parallel Stream

- It's important to understand the semantic differences between collect() and reduce(), e.g.
  - Always test w/a parallel stream to detect mistakes wrt mutable vs. immutable reductions

The "last" call to .parallel() or .sequential() in a stream "wins"

```
void buggyStreamReduce3
        (boolean parallel) {
...
Stream<String> wordStream =
  allWords.stream();

if (parallel)
  wordStream.parallel();

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

See mail.openjdk.java.net/pipermail/lambda-libs-spec-experts/2013-March/001504.html

Differences for collect() and reduce() in a Parallel Stream

- It's important to understand the semantic differences between collect() and reduce(), e.g.
  - Always test w/a parallel stream to detect mistakes wrt mutable vs. immutable reductions

This code works when parallel is false since the StringBuilder is only called in a single thread

```
void buggyStreamReduce3
        (boolean parallel) {
...
Stream<String> wordStream =
  allWords.stream();

if (parallel)
  wordStream.parallel();

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

See docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html

Differences for collect() and reduce() in a Parallel Stream

- It's important to understand the semantic differences between collect() and reduce(), e.g.
  - Always test w/a parallel stream to detect mistakes wrt mutable vs. immutable reductions

```
void buggyStreamReduce3
        (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 true since reduce() expects to do an "immutable" reduction

Differences for collect() and reduce() in a Parallel Stream

- It's important to understand the semantic differences between collect() and reduce(), e.g.
  - Always test w/a parallel stream to detect mistakes wrt mutable vs. immutable reductions

There's a race condition here since StringBuilder is not thread-safe..

```
void buggyStreamReduce3
        (boolean parallel) {
...
Stream<String> wordStream =
  allWords.stream();

if (parallel)
  wordStream.parallel();

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

See www.baeldung.com/java-string-builder-string-buffer

Differences for collect() and reduce() in a Parallel Stream

- It's important to understand the semantic differences between collect() and reduce(), e.g.
  - Always test w/a parallel stream to detect mistakes wrt mutable vs. immutable reductions
    - One solution use reduce() with string concatenation

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

if (parallel)
  wordStream.parallel();

String words = wordStream
  .reduce(new String(),
          (x, y) -> x + y);
```

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

Differences for collect() and reduce() in a Parallel Stream

- It's important to understand the semantic differences between collect() and reduce(), e.g.
  - Always test w/a parallel stream to detect mistakes wrt mutable vs. immutable reductions
    - One solution use reduce() with string concatenation

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

if (parallel)
  wordStream.parallel();

String words = wordStream
  .reduce(new String(),
          (x, y) -> x + y);
```

This simple fix is inefficient due to string concatenation overhead

See javarevisited.blogspot.com/2015/01/3-examples-to-concatenate-string-in-java.html

Differences for collect() and reduce() in a Parallel Stream

- It's important to understand the semantic differences between collect() and reduce(), e.g.
  - Always test w/a parallel stream to detect mistakes wrt mutable vs. immutable reductions
    - One solution use reduce() with string concatenation
    - Another solution uses collect() with the joining collector

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

if (parallel)
  wordStream.parallel();

String words = wordStream
  .collect(joining());
```

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

Differences for collect() and reduce() in a Parallel Stream

- It's important to understand the semantic differences between collect() and reduce(), e.g.
  - Always test w/a parallel stream to detect mistakes wrt mutable vs. immutable reductions
    - One solution use reduce() with string concatenation
    - Another solution uses collect() with the joining collector

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

if (parallel)
  wordStream.parallel();

String words = wordStream
  .collect(joining());
```

This is a much better solution!!

See www.mkyong.com/java8/java-8-stringjoiner-example

Differences for collect() and reduce() in a Parallel Stream

- Also beware of issues related to associativity and identity with reduce()

```
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);
```

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

Differences for collect() and reduce() in a Parallel Stream

- Also beware of issues related to associativity and identity with reduce()

```
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 for a parallel stream since subtraction is not associative

See www.ibm.com/developerworks/library/j-java-streams-2-brian-goetz

Differences for collect() and reduce() in a Parallel Stream

- Also beware of issues related to associativity and identity with reduce()

```
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" (and inverse)

Differences for collect() and reduce() in a Parallel Stream

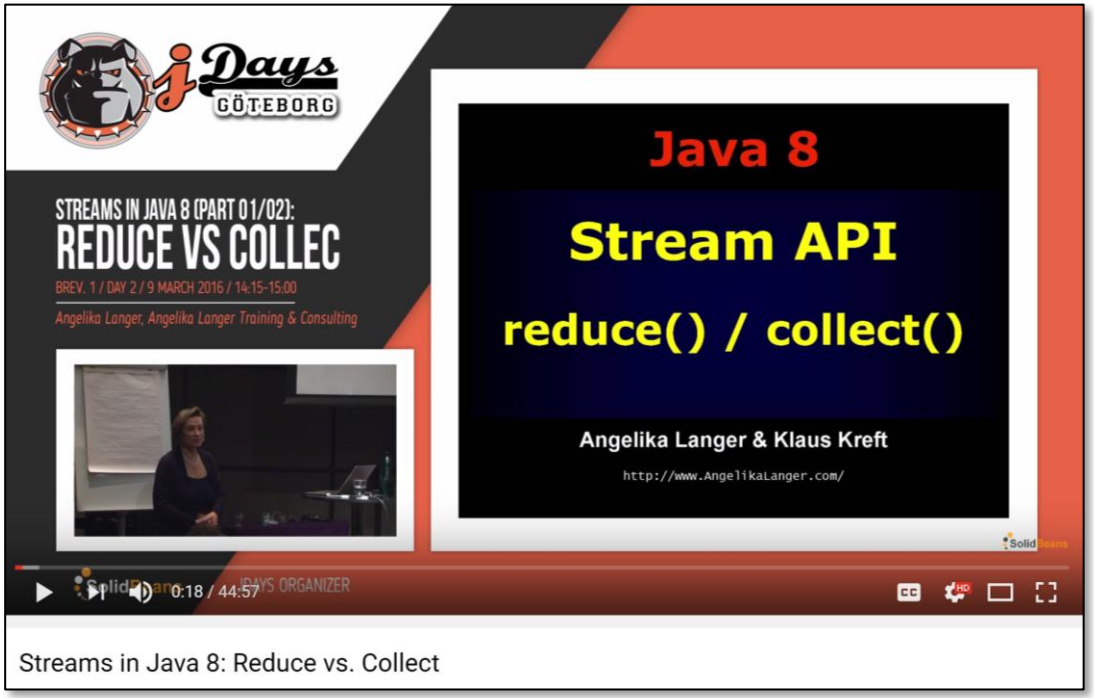
- Also beware of issues related to associativity and identity with reduce()

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

void testProd(long identity, ...) {
   long sum = LongStream
      .rangeClosed(1, 100)
      .reduce(identity,
              (x, y) -> x * y);
```

This code fails if identity is not 1L

Differences for collect() and reduce() in a Parallel Stream

- More good discussions about reduce() vs. collect() appear online



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

Java Parallel Stream Internals:
Combining Results (Part II)

The End