Java Parallel Streams: Evaluating the Pros & Cons

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

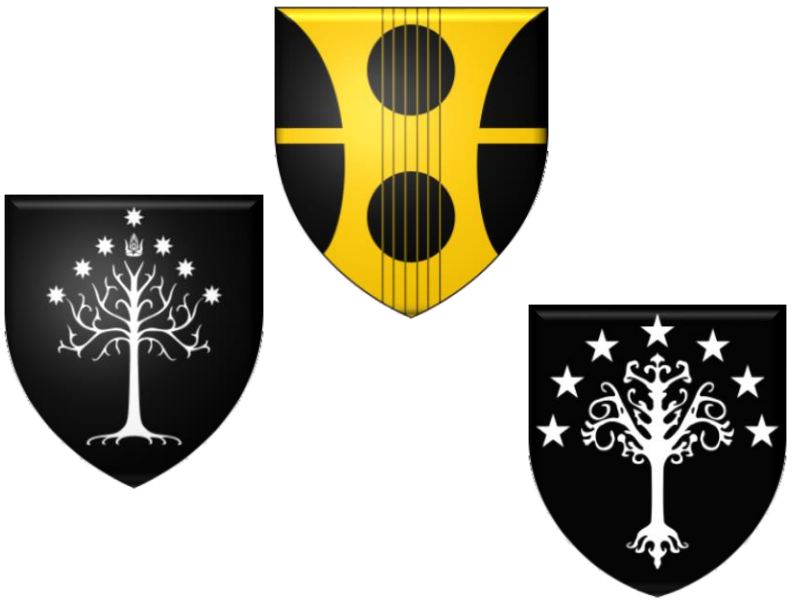
Learning Objectives in this Lesson

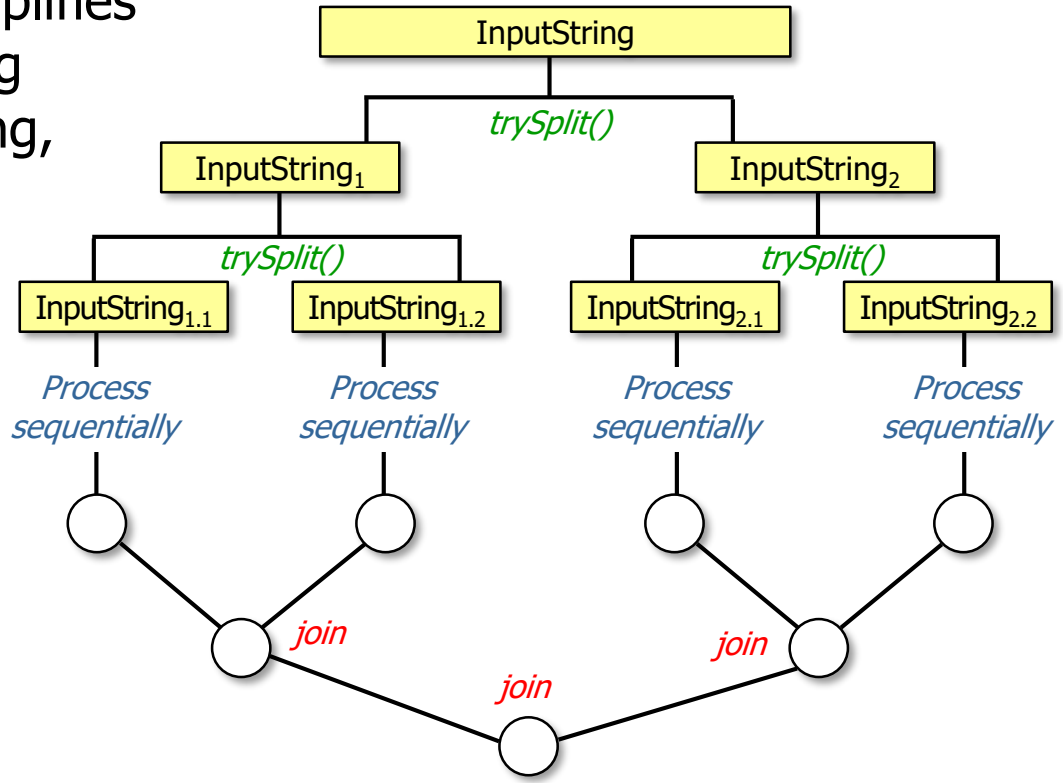
- Evaluate the pros & cons of Java parallel streams

Pros of Java Parallel Streams

Pros of Java Parallel Streams

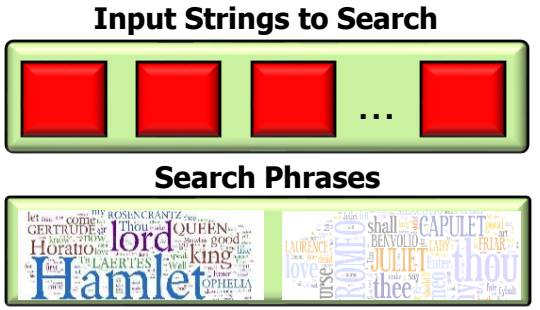
- The Java streams framework simplifies parallel programming by shielding developers from details of splitting, applying, & combining results

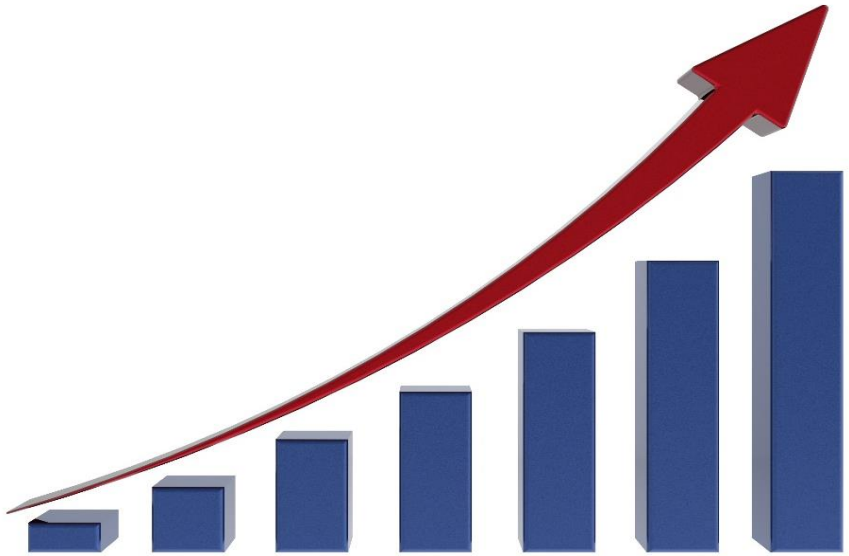


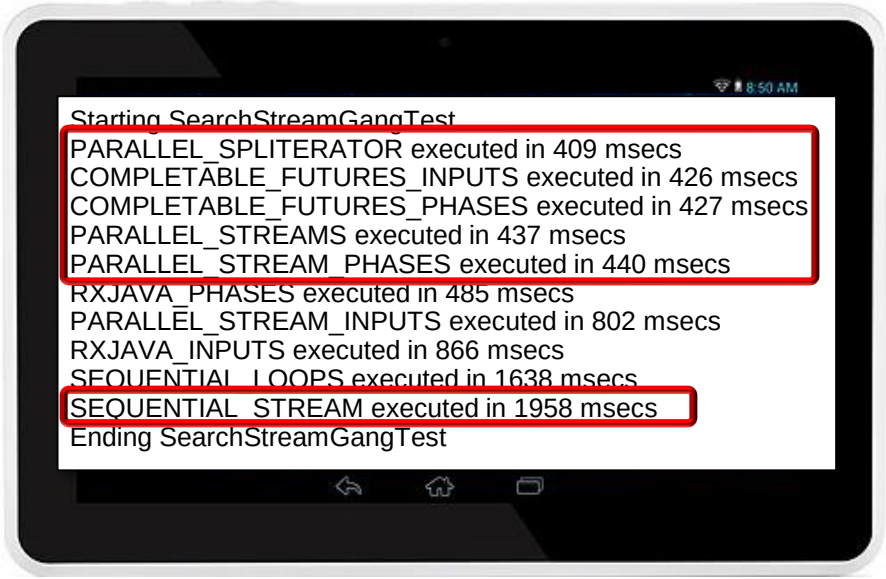


Pros of Java Parallel Streams

- Parallel stream implementations are often (much) faster & more scalable than sequential (stream & loops) implementations

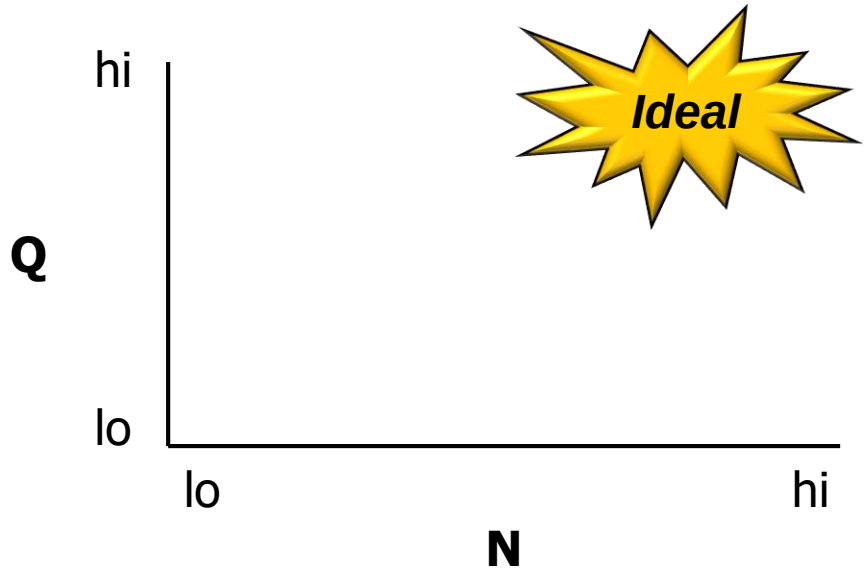






Pros of Java Parallel Streams

- The performance speedup is a largely a function of the partitioning strategy for the input (N), the amount of work performed (Q), & the # of cores



The NQ model

- *N is the # of data elements to process per thread*
- *Q quantifies how CPU-intensive the processing is*

Pros of Java Parallel Streams

- Apps often don't need explicit synchronization or threading

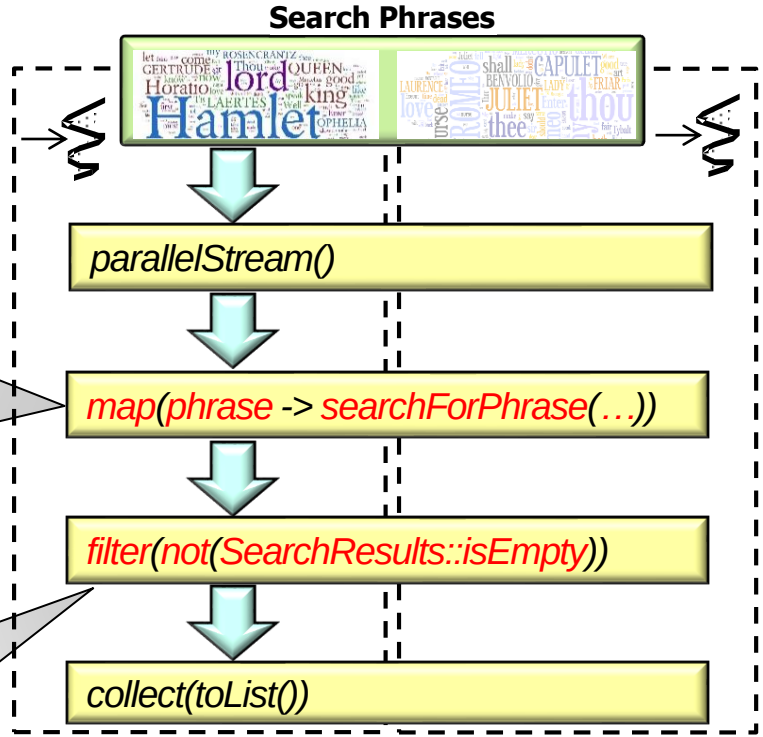
Alleviates many accidental & inherent complexities of concurrency/parallelism

Pros of Java Parallel Streams

- Apps often don't need explicit synchronization or threading
  - Stateless behaviors alleviate the need to access shared mutable state

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

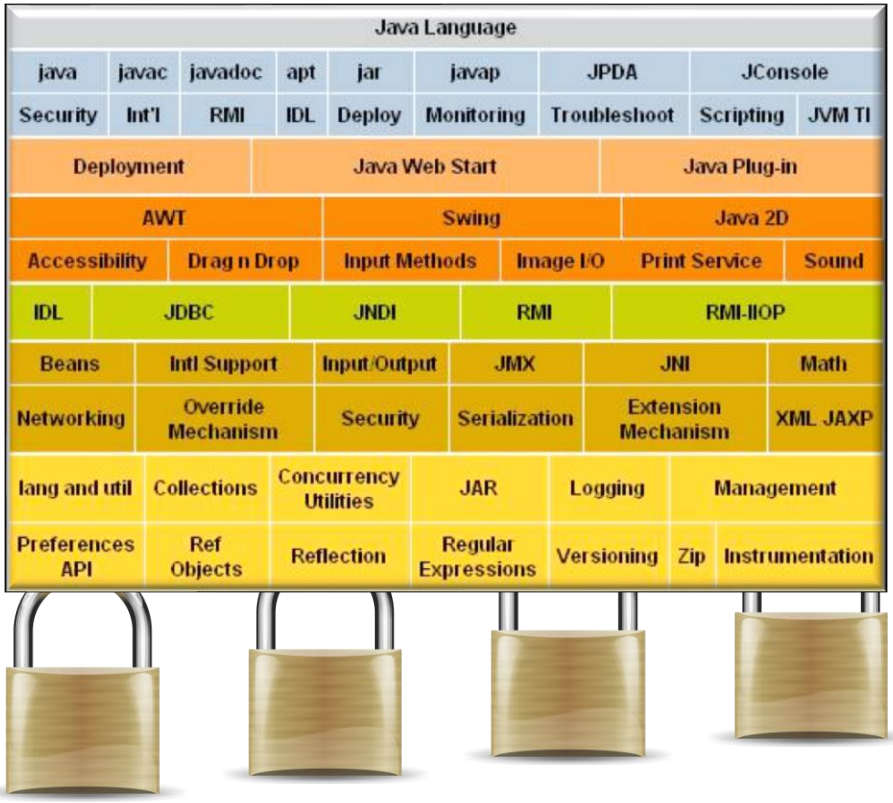
```
return mList.size() == 0;
```



See en.wikipedia.org/wiki/Pure_function

Pros of Java Parallel Streams

- Apps often don't need explicit synchronization or threading
  - Stateless behaviors alleviate the need to access shared mutable state
  - Java class library handles locking needed to protect shared mutable state



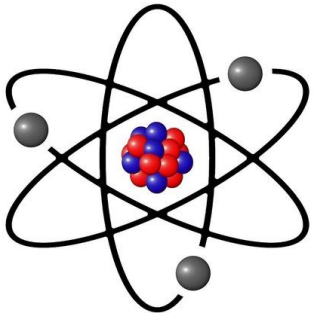
See docs.oracle.com/javase/tutorial/essential/concurrency/collections.html

Pros of Java Parallel Streams

- Streams ensures that the structure of sequential & parallel code is the same

```
List<List<SearchResults>>
            processStream() {
  return getInput()
    .stream()
    .map(this::processInput)
    .collect(toList());
}
```

```
List<List<SearchResults>>
            processStream() {
  return getInput()
    .parallelStream()
    .map(this::processInput)
    .collect(toList());
}
```



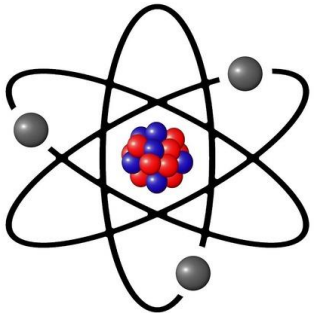
Converting sequential to parallel streams only require minuscule changes!

Pros of Java Parallel Streams

- Streams ensures that the structure of sequential & parallel code is the same

```
List<SearchResults> results =
  mPhrasesToFind
    .parallelStream()
    .map(phase ->
         searchForPhrase(...,
                         false))
    .filter(not(SearchResults
                ::isEmpty))
    .collect(toList());
```

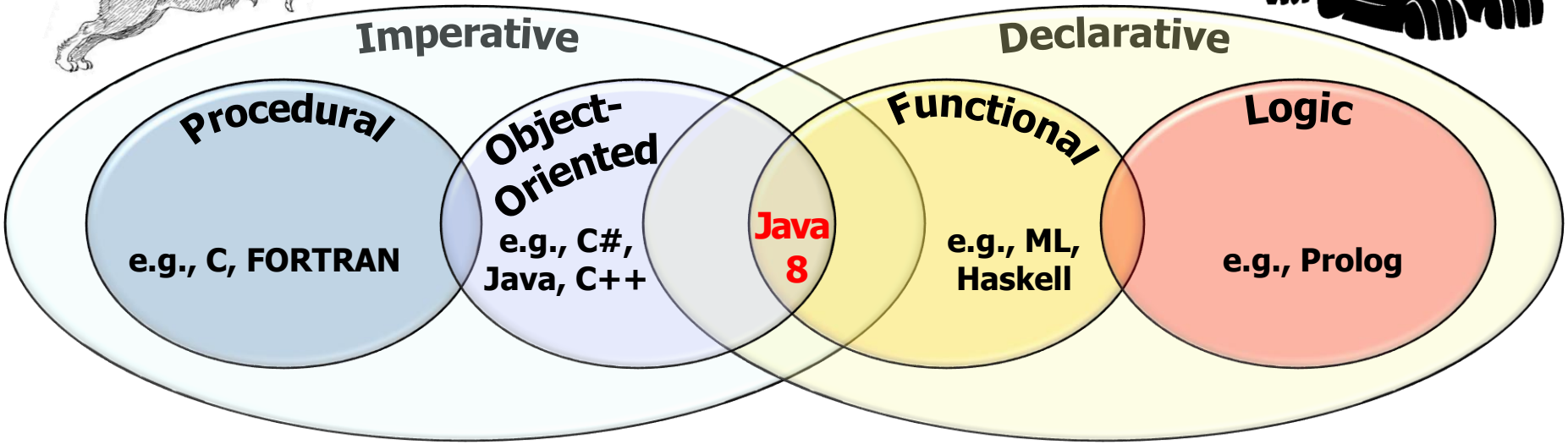
```
List<SearchResults> results =
  mPhrasesToFind
    .parallelStream()
    .map(phase ->
         searchForPhrase(...,
                         true))
    .filter(not(SearchResults
                ::isEmpty))
    .collect(toList());
```



Converting sequential to parallel streams only require minuscule changes!

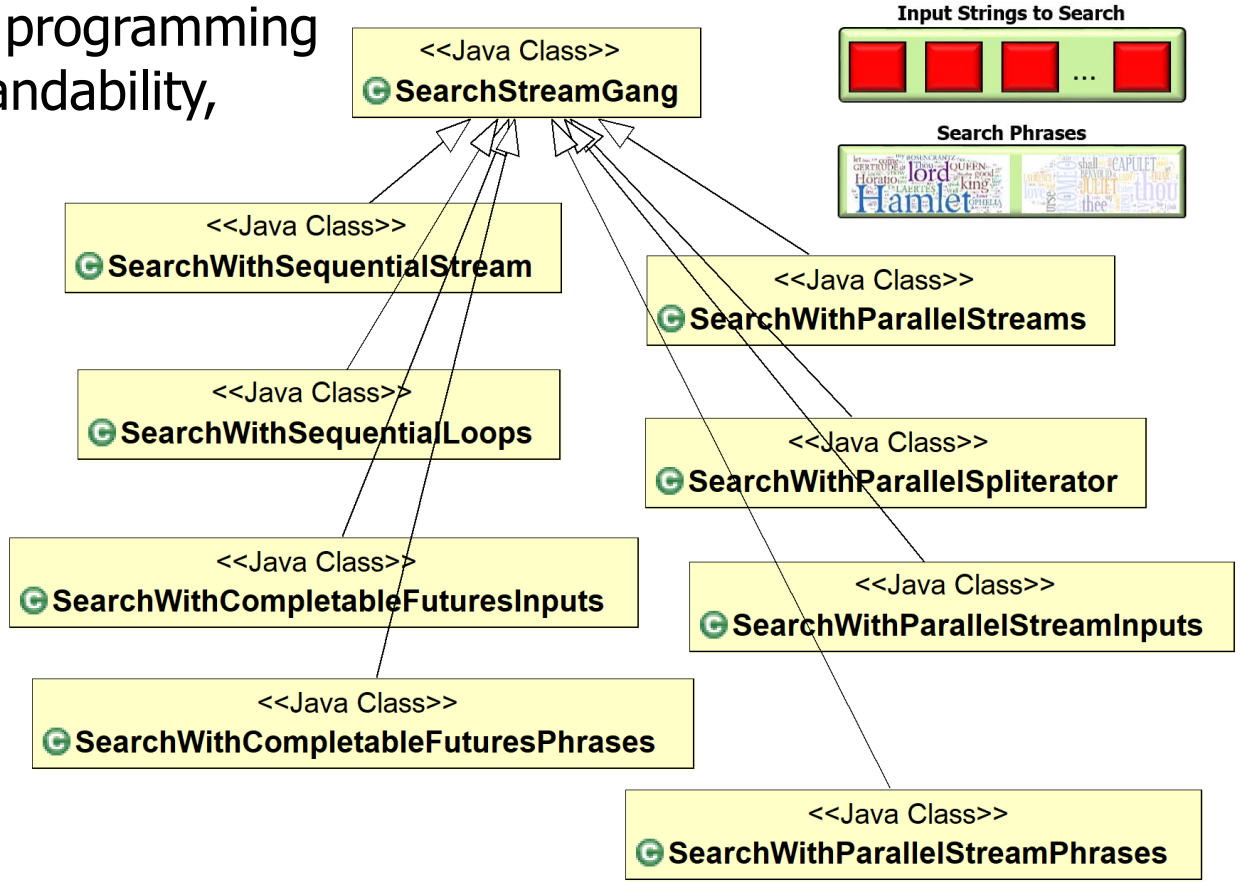
Pros of Java Parallel Streams

- Examples show synergies between functional & object-oriented programming



Pros of Java Parallel Streams

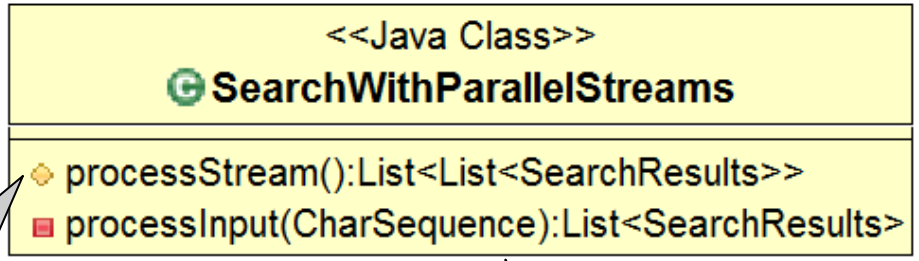
- Object-oriented design & programming features simplify understandability, reusability, & extensibility



Object-oriented techniques emphasize systematic reuse of *structure*

Pros of Java Parallel Streams

- Implementing object-oriented hook methods with functional programming features helps to close gap between domain intent & computations



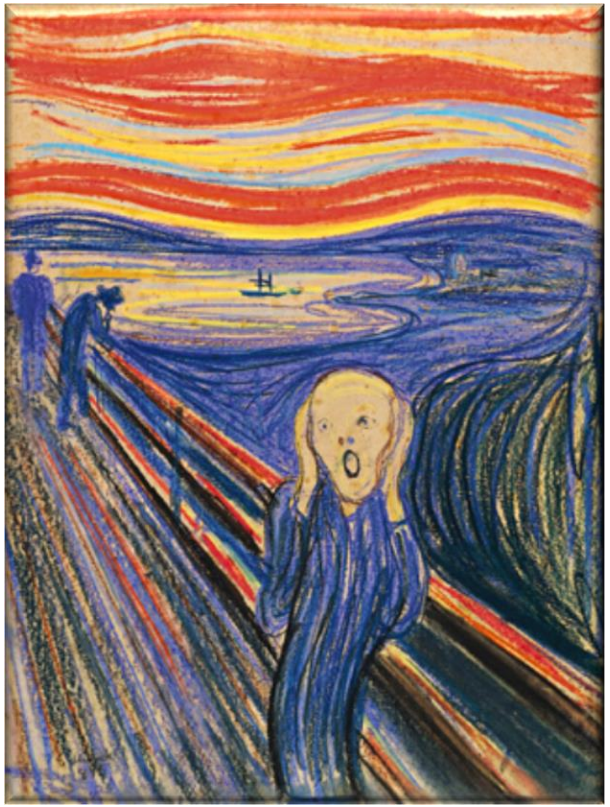

```
getInput()
   .parallelStream()
   .map(this::processInput)
   .collect(toList());
```

```
return mPhrasesToFind
   .parallelStream()
   .map(phrase -> searchForPhrase(phrase, input, title, false))
   .filter(not(SearchResults::isEmpty)
   .collect(toList());
```

Cons of Java Parallel Streams

Cons of Java Parallel Streams

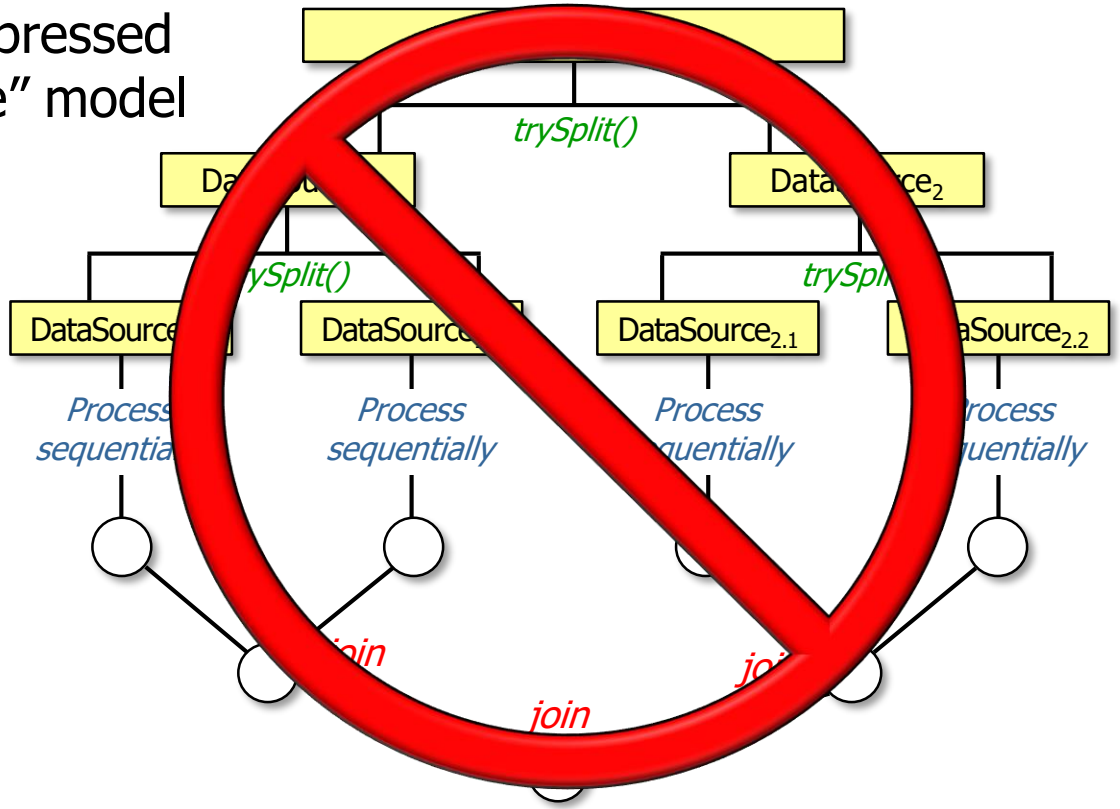
- There are some limitations with Java parallel streams



The Java parallel streams framework is not all unicorns & rainbows!!

Cons of Java Parallel Streams

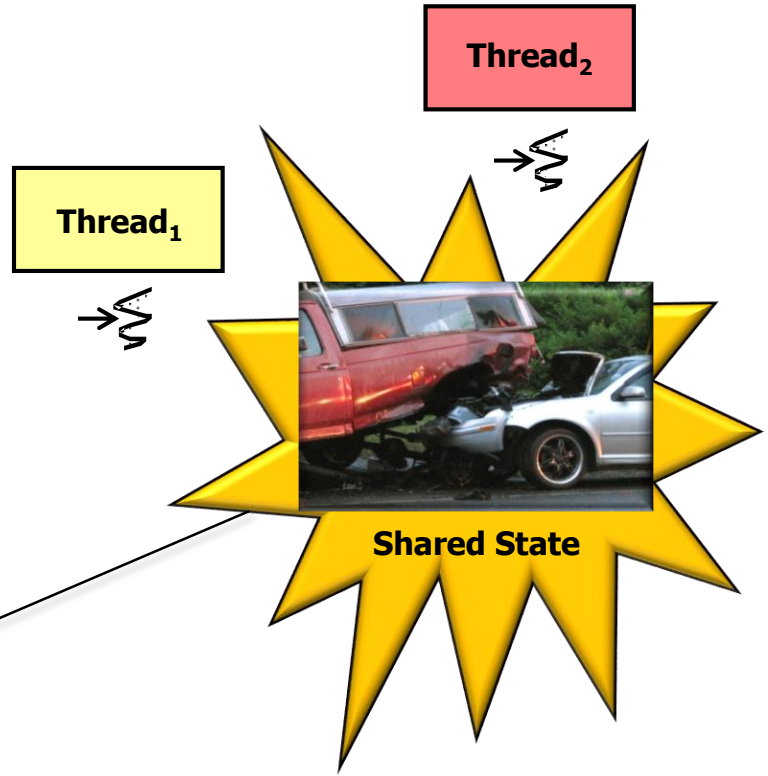
- There are some limitations with Java parallel streams, e.g.
  - Some problems can't be expressed via the "split-apply-combine" model



See dzone.com/articles/whats-wrong-java-8-part-iii

Cons of Java Parallel Streams

- There are some limitations with Java parallel streams, e.g.
  - Some problems can't be expressed via the "split-apply-combine" model
  - If behaviors aren't stateless & thread-safe race conditions may occur



Race conditions occur when a program depends on the sequence or timing of threads for it to operate properly

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

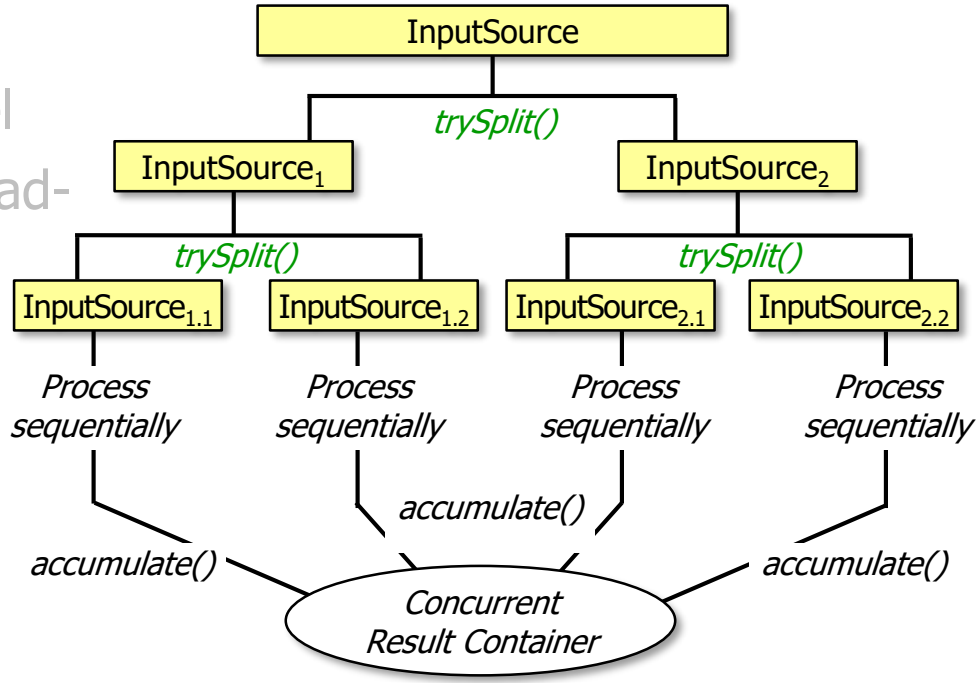
Cons of Java Parallel Streams

- There are some limitations with Java parallel streams, e.g.
  - Some problems can't be expressed via the "split-apply-combine" model
  - If behaviors aren't stateless & thread-safe race conditions may occur
  - Parallel spliterators may be tricky…

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

Cons of Java Parallel Streams

- There are some limitations with Java parallel streams, e.g.
  - Some problems can't be expressed via the "split-apply-combine" model
  - If behaviors aren't stateless & thread-safe race conditions may occur
  - Parallel spliterators may be tricky…
    - Concurrent collectors are easier



See lesson on "*Java Parallel Stream Internals: Non-Concurrent & Concurrent Collectors*"

Cons of Java Parallel Streams

- There are some limitations with Java parallel streams, e.g.
  - Some problems can't be expressed via the "split-apply-combine" model
  - If behaviors aren't stateless & thread-safe race conditions may occur
  - Parallel spliterators may be tricky…
  - All parallel streams share a common fork-join pool

See dzone.com/articles/think-twice-using-java-8

Cons of Java Parallel Streams

- There are some limitations with Java parallel streams, e.g.
  - Some problems can't be expressed via the "split-apply-combine" model
  - If behaviors aren't stateless & thread-safe race conditions may occur
  - Parallel spliterators may be tricky…
  - All parallel streams share a common fork-join pool
    - Java completable futures don't have this limitation

See dzone.com/articles/think-twice-using-java-8

Cons of Java Parallel Streams

- There are some limitations with Java parallel streams, e.g.
  - Some problems can't be expressed via the "split-apply-combine" model
  - If behaviors aren't stateless & thread-safe race conditions may occur
  - Parallel spliterators may be tricky…
  - All parallel streams share a common fork-join pool
    - Java completable futures don't have this limitation
    - It's important to know how to apply ManagedBlockers

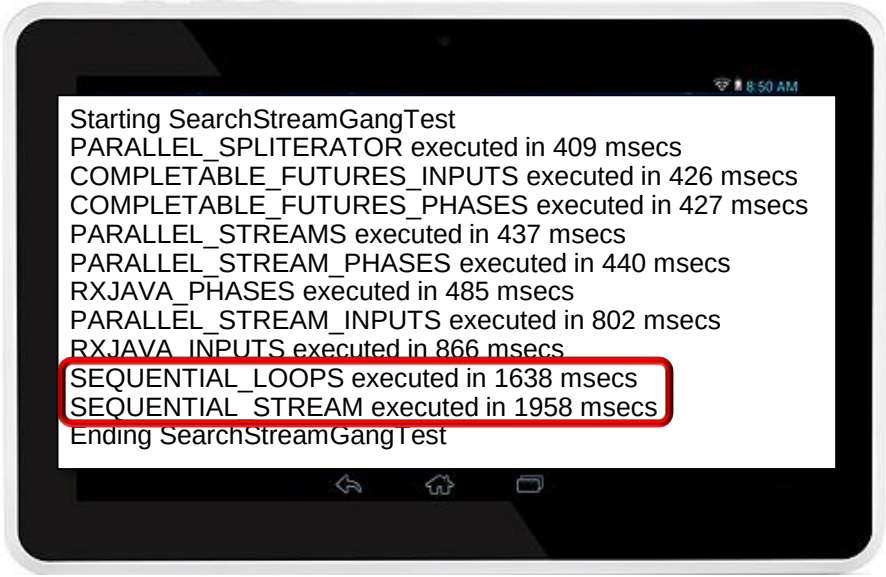
See "*The Java Fork-Join Pool: Applying the ManagedBlocker Interface*"

Cons of Java Parallel Streams

- There are some limitations with Java parallel streams, e.g.
  - Some problems can't be expressed via the "split-apply-combine" model
  - If behaviors aren't stateless & thread-safe race conditions may occur
  - Parallel spliterators may be tricky…
  - All parallel streams share a common fork-join pool
  - Streams incur some overhead

Cons of Java Parallel Streams

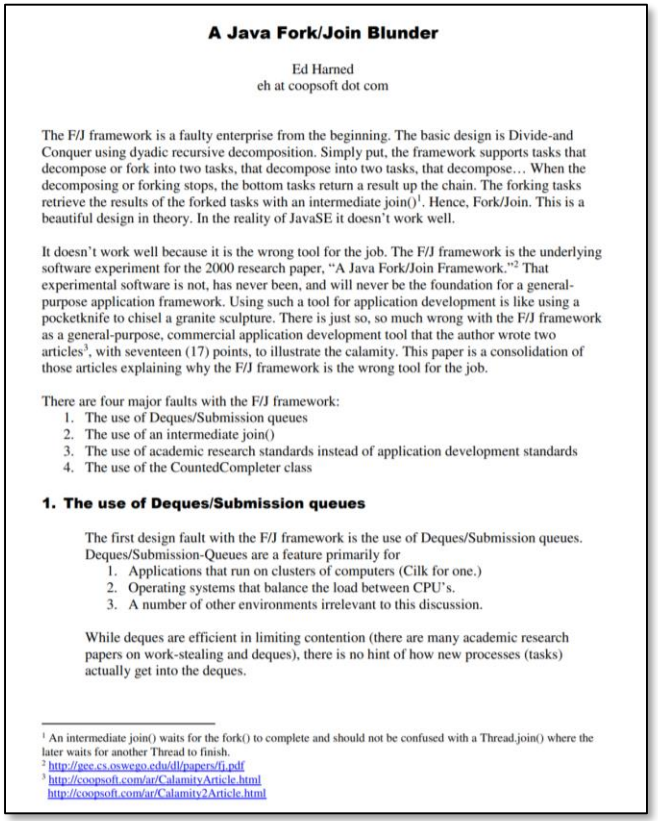
- There are some limitations with Java parallel streams, e.g.
  - Some problems can't be expressed via the "split-apply-combine" model
  - If behaviors aren't stateless & thread-safe race conditions may occur
  - Parallel spliterators may be tricky…
  - All parallel streams share a common fork-join pool
  - Streams incur some overhead, e.g.
    - Splitting & combining overhead



See blog.jooq.org/2015/12/08/3-reasons-why-you-shouldnt-replace-your-for-loops-by-stream-foreach

Cons of Java Parallel Streams

- There are some limitations with Java parallel streams, e.g.
  - Some problems can't be expressed via the "split-apply-combine" model
  - If behaviors aren't stateless & thread-safe race conditions may occur
  - Parallel spliterators may be tricky…
  - All parallel streams share a common fork-join pool
  - Streams incur some overhead, e.g.
    - Splitting & combining overhead
    - Fork-join framework



A Java Fork/Join Blunder

Ed Harned
eh at coopsoft dot com

The F/J framework is a faulty enterprise from the beginning. The basic design is Divide-and Conquer using dyadic recursive decomposition. Simply put, the framework supports tasks that decompose or fork into two tasks, that decompose into two tasks, that decompose… When the decomposing or forking stops, the bottom tasks return a result up the chain. The forking tasks retrieve the results of the forked tasks with an intermediate join()[1]. Hence, Fork/Join. This is a beautiful design in theory. In the reality of JavaSE it doesn't work well.

It doesn't work well because it is the wrong tool for the job. The F/J framework is the underlying software experiment for the 2000 research paper, "A Java Fork/Join Framework."[2] That experimental software is not, has never been, and will never be the foundation for a general-purpose application framework. Using such a tool for application development is like using a pocketknife to chisel a granite sculpture. There is just so, so much wrong with the F/J framework as a general-purpose, commercial application development tool that the author wrote two articles[3], with seventeen (17) points, to illustrate the calamity. This paper is a consolidation of those articles explaining why the F/J framework is the wrong tool for the job.

There are four major faults with the F/J framework:
1. The use of Deques/Submission queues
2. The use of an intermediate join()
3. The use of academic research standards instead of application development standards
4. The use of the CountedCompleter class

1. The use of Deques/Submission queues

The first design fault with the F/J framework is the use of Deques/Submission queues. Deques/Submission-Queues are a feature primarily for
1. Applications that run on clusters of computers (Cilk for one.)
2. Operating systems that balance the load between CPU's.
3. A number of other environments irrelevant to this discussion.

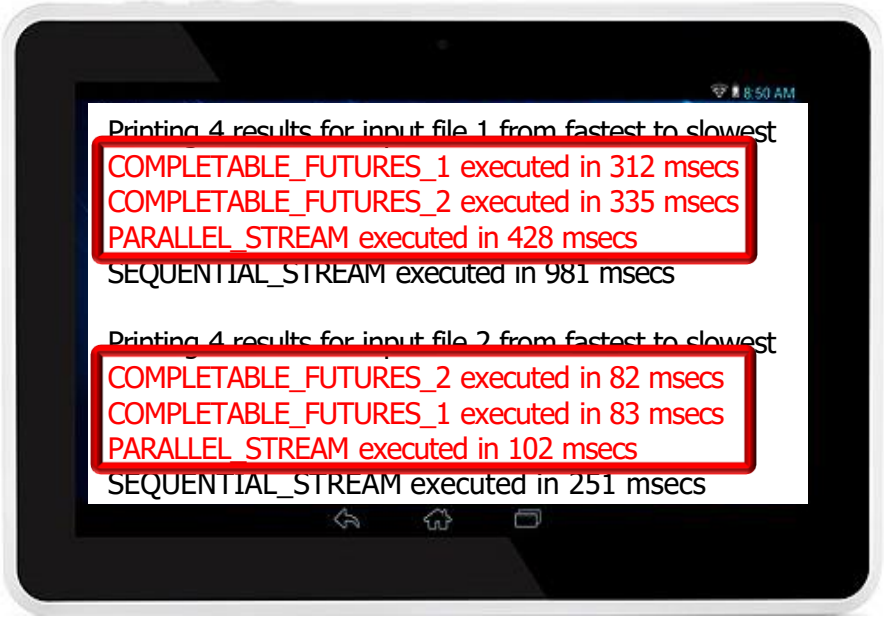
While deques are efficient in limiting contention (there are many academic research papers on work-stealing and deques), there is no hint of how new processes (tasks) actually get into the deques.

[1] An intermediate join() waits for the fork() to complete and should not be confused with a Thread.join() where the later waits for another Thread to finish.
[2] http://gee.cs.oswego.edu/dl/papers/fj.pdf
[3] http://coopsoft.com/ar/CalamityArticle.html
http://coopsoft.com/ar/Calamity2Article.html

See coopsoft.com/dl/Blunder.pdf

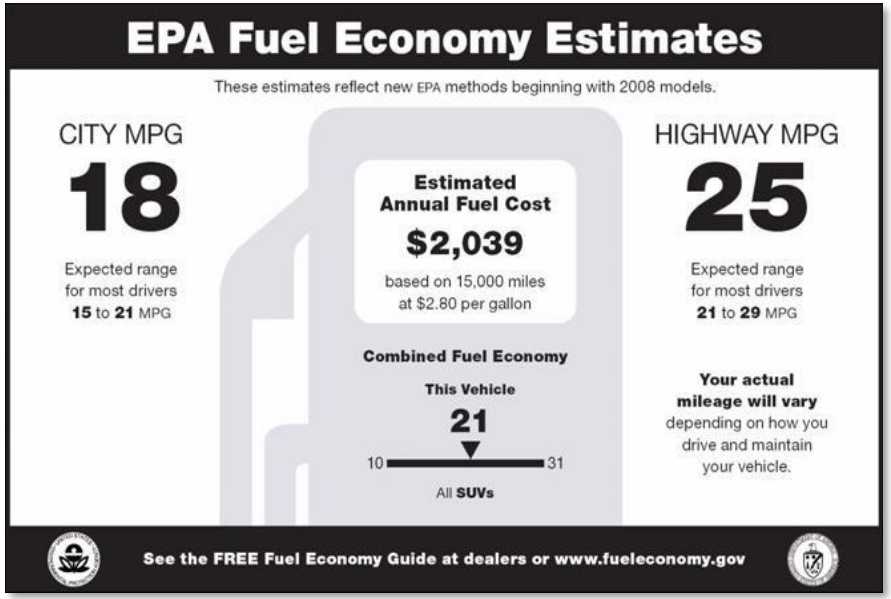
Cons of Java Parallel Streams

- There are some limitations with Java parallel streams, e.g.
  - Some problems can't be expressed via the "split-apply-combine" model
  - If behaviors aren't stateless & thread-safe race conditions may occur
  - Parallel spliterators may be tricky…
  - All parallel streams share a common fork-join pool
  - Streams incur some overhead, e.g.
    - Splitting & combining overhead
    - Fork-join framework
    - Java completable futures may be more efficient & scalable



Cons of Java Parallel Streams

- There are some limitations with Java parallel streams, e.g.
  - Some problems can't be expressed via the "split-apply-combine" model
  - If behaviors aren't stateless & thread-safe race conditions may occur
  - Parallel spliterators may be tricky…
  - All parallel streams share a common fork-join pool
  - Streams incur some overhead, e.g.
    - Splitting & combining overhead
    - Fork-join framework
    - Java completable futures may be more efficient & scalable



Naturally, your mileage may vary..

Cons of Java Parallel Streams

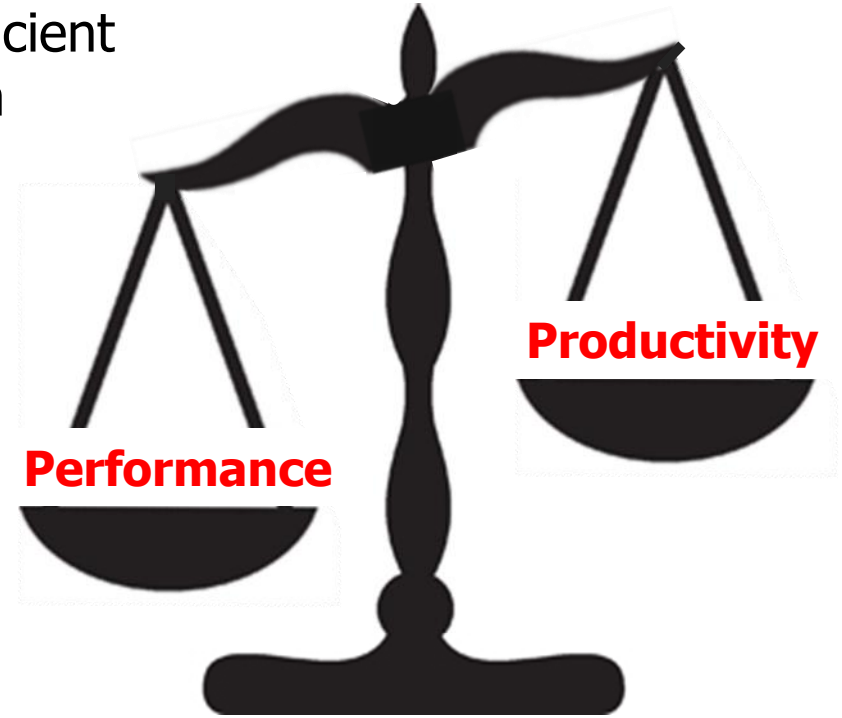
- There are some limitations with Java parallel streams, e.g.
  - Some problems can't be expressed via the "split-apply-combine" model
  - If behaviors aren't stateless & thread-safe race conditions may occur
  - Parallel spliterators may be tricky…
  - All parallel streams share a common fork-join pool
  - Streams incur some overhead
  - There's no substitute for benchmarking!

See java-performance.info/jmh

Wrapping Up Java Parallel Streams

Wrapping Up Java Parallel Streams

- In general, there's a tradeoff between computing performance & programmer productivity when choosing amongst these frameworks
  - i.e., completable futures are more efficient & scalable, but are harder to program



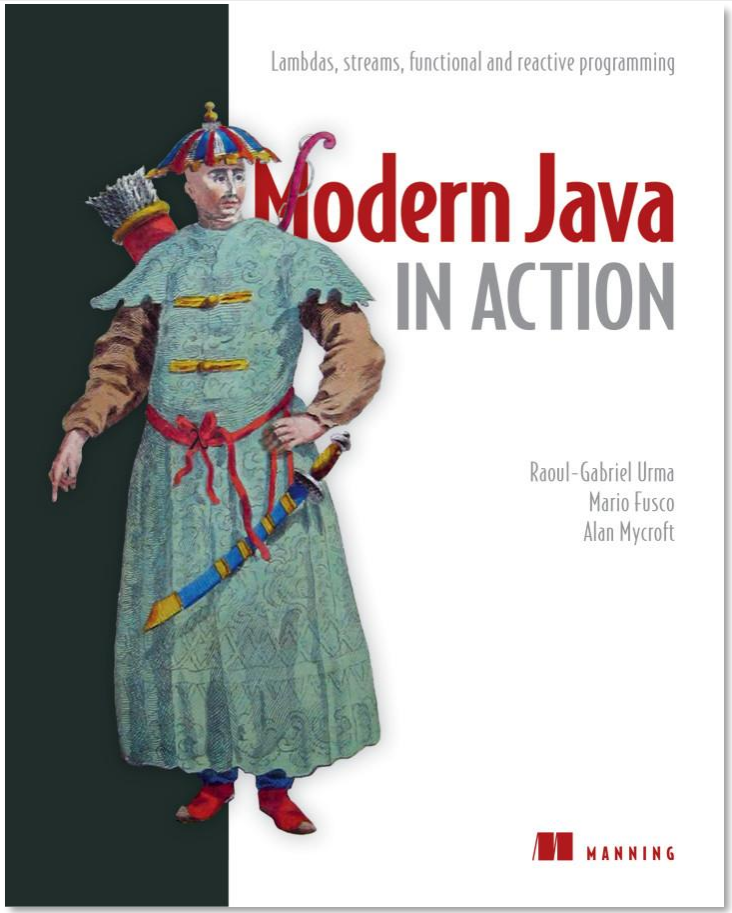
Wrapping Up Java Parallel Streams

- In general, however, the pros of Java parallel streams far outweigh the cons for many use cases!!

See www.ibm.com/developerworks/library/j-jvmc2

Wrapping Up Java Parallel Streams

- Good coverage of parallel streams appears in the book "Modern Java in Action"



See www.manning.com/books/modern-java-in-action

End of Java Parallel Streams: Evaluating the Pros & Cons