Overview of Java 8 Parallel Streams
(Part 1)

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

Professor of Computer Science
Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
Learning Objectives in this Part of the Lesson

• Know how aggregate operations from Java 8 sequential streams are applied in the parallel streams framework.

```
<table>
<thead>
<tr>
<th>Input x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intermediate operation (behavior f)</td>
</tr>
<tr>
<td>Output f(x)</td>
</tr>
<tr>
<td>Intermediate operation (behavior g)</td>
</tr>
<tr>
<td>Output g(f(x))</td>
</tr>
<tr>
<td>Terminal operation (reducer)</td>
</tr>
</tbody>
</table>
```
Transitioning from Sequential Streams to Parallel Streams
Transitioning from Sequential Streams to Parallel Streams

• A Java 8 stream is a pipeline of aggregate operations that process a sequence of elements (aka, “values” or “data”)

Transitioning from Sequential Streams to Parallel Streams

- A Java 8 stream is a pipeline of aggregate operations that process a sequence of elements (aka, “values” or “data”)

An aggregate operation applies a “behavior” to every element in a stream
By default, a stream executes sequentially, so all its aggregate operations run behaviors in a single thread of control.
Transitioning from Sequential Streams to Parallel Streams

- When a stream executes in parallel, it is partitioned into multiple substream “chunks” that run in a common fork-join pool

List `<String>`
Stream `<String>`
Stream `<SearchResults>`
Stream `<SearchResults>`
List `<SearchResults>`

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html](docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html)
When a stream executes in parallel, it is partitioned into multiple substream “chunks” that run in a common fork-join pool.

```
Stream<SearchResults> Stream<String> Stream<SearchResults> List<SearchResults> ...
```

Threads in the fork-join pool (non-deterministically) process different chunks.

```
parallelStream()
map(phrase -> searchForPhrase(...))
filter(not(SearchResults::isEmpty))
collect(toList())
```

Transitioning from Sequential Streams to Parallel Streams
When a stream executes in parallel, it is partitioned into multiple substream "chunks" that run in a common fork-join pool.

- **List <String>**
- **Stream <String>**
- **Stream <SearchResults>**
- **Stream <SearchResults>**
- **List <SearchResults>**

Intermediate operations iterate over & process these chunks in parallel.
Transitioning from Sequential Streams to Parallel Streams

- When a stream executes in parallel, it is partitioned into multiple substream "chunks" that run in a common fork-join pool.

A terminal operation then combines the chunks into a single result.
Transitioning from Sequential Streams to Parallel Streams

- When a stream executes in parallel, it is partitioned into multiple substream “chunks” that run in a common fork-join pool

(Stateless) Java 8 lambda expressions & method references are used to pass behaviors
Transitioning from Sequential Streams to Parallel Streams

- When a stream executes in parallel, it is partitioned into multiple substream “chunks” that run in a common fork-join pool.

```
List<String> ->
Stream<String> ->
Stream<SearchResults> ->
Stream<SearchResults> ->
List<SearchResults>
```

```
parallelStream() vs. stream()
map(phrase -> searchForPhrase(…))
filter(not(SearchResults::isEmpty))
collect(toList())
```

Ideally, minuscule changes needed to transition from sequential to parallel stream.
Transitioning from Sequential Streams to Parallel Streams

- The same aggregate operations can be used for sequential & parallel streams

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>allMatch(Predicate&lt;T super T&gt; predicate)</td>
</tr>
<tr>
<td>boolean</td>
<td>anyMatch(Predicate&lt;T super T&gt; predicate)</td>
</tr>
<tr>
<td>static &lt;T&gt; Stream.Builder&lt;T&gt;</td>
<td>builder()</td>
</tr>
<tr>
<td>&lt;R,A&gt; R</td>
<td>collect(Collectors&lt;T super T,A,R&gt; collector)</td>
</tr>
<tr>
<td>&lt;R&gt; R</td>
<td>collect(Supplier&lt;R&gt; supplier, BiConsumer&lt;R,T&gt; accumulator, BiConsumer&lt;R,R&gt; combiner)</td>
</tr>
<tr>
<td>static &lt;T&gt; Stream&lt;T&gt;</td>
<td>concat(Stream&lt;T&gt; extends T -&gt; a, Stream&lt;T&gt; extends T -&gt; b)</td>
</tr>
<tr>
<td>long</td>
<td>count()</td>
</tr>
<tr>
<td>Stream&lt;T&gt;</td>
<td>distinct()</td>
</tr>
<tr>
<td>static &lt;T&gt; Stream&lt;T&gt;</td>
<td>empty()</td>
</tr>
<tr>
<td>Stream&lt;T&gt;</td>
<td>filter(Predicate&lt;T super T&gt; predicate)</td>
</tr>
<tr>
<td>Optional&lt;T&gt;</td>
<td>findAny()</td>
</tr>
<tr>
<td>Optional&lt;T&gt;</td>
<td>findFirst()</td>
</tr>
<tr>
<td>&lt;R&gt; Stream&lt;R&gt;</td>
<td>flatMap(Function&lt;T super T,R extends Stream&lt;R extends R&gt;&gt; mapper)</td>
</tr>
</tbody>
</table>

See docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html
Transitioning from Sequential Streams to Parallel Streams

- The same aggregate operations can be used for sequential & parallel streams

* e.g., SearchStreamGang uses the same aggregate operations for both SearchWithSequentialStreams & SearchWithParallelStreams implementations


```
SearchPhrases

map(phrase -> searchForPhrase(…))

filter(not(SearchResults::isEmpty))

collect(toList())
```
Transitioning from Sequential Streams to Parallel Streams

- The same aggregate operations can be used for sequential & parallel streams
- Java 8 streams can thus treat parallelism as an optimization & leverage all available cores!

```
parallelStream()
    .map(phrase -> searchForPhrase(…))
    .filter(not(SearchResults::isEmpty))
    .collect(toList())
```

The same aggregate operations can be used for sequential & parallel streams.
Java 8 streams can thus treat parallelism as an optimization & leverage all available cores!
Naturally, behaviors run by these aggregate operations must be designed carefully to avoid accessing unsynchronized shared state.

See [henrikeichenhardt.blogspot.com/2013/06/why-shared-mutable-state-is-root-of-all.html](http://henrikeichenhardt.blogspot.com/2013/06/why-shared-mutable-state-is-root-of-all.html)
End of Overview of Java 8 Parallel Streams (Part 1)