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

- Recognize how Java 8 leverages its functional programming features in the parallel streams concurrency framework.
Overview of Java 8
Parallel Streams
Overview of Java 8 Parallel Streams

- Aggregate operations work for both 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&gt; super T&gt; predicate)</td>
</tr>
<tr>
<td></td>
<td>Returns whether all elements of this stream match the provided predicate.</td>
</tr>
<tr>
<td>boolean</td>
<td>anyMatch(Predicate&lt;T&gt; super T&gt; predicate)</td>
</tr>
<tr>
<td></td>
<td>Returns whether any elements of this stream match the provided predicate.</td>
</tr>
<tr>
<td>static &lt;T&gt; Stream.Builder&lt;T&gt;</td>
<td>builder()</td>
</tr>
<tr>
<td></td>
<td>Returns a builder for a Stream.</td>
</tr>
<tr>
<td>&lt;R,A&gt; R</td>
<td>collect(Collectors super T,A,R&gt; collector)</td>
</tr>
<tr>
<td></td>
<td>Performs a mutable reduction operation on the elements of this stream using a Collector.</td>
</tr>
<tr>
<td>&lt;R&gt; R</td>
<td>collect(Supplier&lt;R&gt; supplier, BiConsumer&lt;R,R&gt; accumulator, BiConsumer&lt;R,R&gt; combiner)</td>
</tr>
<tr>
<td></td>
<td>Performs a mutable reduction operation on the elements of this stream.</td>
</tr>
<tr>
<td>static &lt;T&gt; Stream&lt;T&gt;</td>
<td>concat(Stream&lt;? extends T&gt; a, Stream&lt;? extends T&gt; b)</td>
</tr>
<tr>
<td></td>
<td>Creates a lazily concatenated stream whose elements are all the elements of the first stream followed by all the elements of the second stream.</td>
</tr>
<tr>
<td>long</td>
<td>count()</td>
</tr>
<tr>
<td></td>
<td>Returns the count of elements in this stream.</td>
</tr>
<tr>
<td>Stream&lt;T&gt;</td>
<td>distinct()</td>
</tr>
<tr>
<td></td>
<td>Returns a stream consisting of the distinct elements (according to Object.equals(Object)) of this stream.</td>
</tr>
<tr>
<td>static &lt;T&gt; Stream&lt;T&gt;</td>
<td>empty()</td>
</tr>
<tr>
<td></td>
<td>Returns an empty sequential Stream.</td>
</tr>
<tr>
<td>Stream&lt;T&gt;</td>
<td>filter(Predicate&lt;T&gt; super T&gt; predicate)</td>
</tr>
<tr>
<td></td>
<td>Returns a stream consisting of the elements of this stream that match the given predicate.</td>
</tr>
<tr>
<td>Optional&lt;T&gt;</td>
<td>findAny()</td>
</tr>
<tr>
<td></td>
<td>Returns an Optional describing some element of the stream, or an empty Optional if the stream is empty.</td>
</tr>
<tr>
<td>Optional&lt;T&gt;</td>
<td>findFirst()</td>
</tr>
<tr>
<td></td>
<td>Returns an Optional describing the first element of this stream, or an empty Optional if the stream is empty.</td>
</tr>
<tr>
<td>&lt;R&gt; Stream&lt;R&gt;</td>
<td>flatMap(Function&lt;? super T, ? extends Stream&lt;? extends R&gt;&gt; mapper)</td>
</tr>
<tr>
<td></td>
<td>Returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapper.</td>
</tr>
</tbody>
</table>

See [docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html)
Overview of Java 8 Parallel Streams

- Aggregate operations work for both sequential & parallel streams
- Assuming functions don’t have side-effects that yield race conditions

See en.wikipedia.org/wiki/Race_condition#Software
Overview of Java 8 Parallel Streams

• Aggregate operations work for both sequential & parallel streams
  • Assuming functions don’t have side-effects that yield race conditions
• The difference between a sequential & parallel stream is often minuscule!

See docs.oracle.com/javase/tutorial/collections/streams/parallelism.html
Overview of Java 8 Parallel Streams

- Aggregate operations work for both sequential & parallel streams
- Assuming functions don’t have side-effects that yield race conditions
- The difference between a sequential & parallel stream is often minuscule!

```java
List<List<SearchResults>> processStream() {
    return getInput().stream()
        .map(this::processInput)
        .collect(toList());
}
```
Overview of Java 8 Parallel Streams

- Aggregate operations work for both sequential & parallel streams
- Assuming functions don’t have side-effects that yield race conditions
- The difference between a sequential & parallel stream is often minuscule!

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

VS

List<List<SearchResults>> processStream() {
    return getInput()
        .parallelStream()
        .map(this::processInput)
        .collect(toList());
}
```
Overview of Java 8 Parallel Streams

• Aggregate operations work for both sequential & parallel streams
• Assuming functions don’t have side-effects that yield race conditions
• The difference between a sequential & parallel stream is often minuscule!

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

VS

List<List<SearchResults>>
    processStream() {
        return getInput()
            .parallelStream()
            .map(this::processInput)
            .collect(toList());
    }
```
Overview of Java 8 Parallel Streams

- Aggregate operations work for both sequential & parallel streams
- Assuming functions don’t have side-effects that yield race conditions
- The difference between a sequential & parallel stream is often minuscule!

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

VS

List<List<SearchResults>> processStream() {
    return getInput()
        .parallelStream()
        .map(this::processInput)
        .collect(toList());
}
```
Overview of Java 8 Parallel Streams

• Internally, Java 8’s streams framework uses a ForkJoinPool executor

```java
public class ForkJoinPool
extends AbstractExecutorService

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

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

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

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html
Overview of Java 8 Parallel Streams

- Internally, Java 8’s streams framework uses a ForkJoinPool executor
- It supports algorithms that can be recursively split up

See en.wikipedia.org/wiki/Embarrassingly_parallel
Overview of Java 8 Parallel Streams

- Internally, Java 8’s streams framework uses a ForkJoinPool executor
- It supports algorithms that can be recursively split up

See en.wikipedia.org/wiki/MapReduce
Overview of Java 8 Parallel Streams

- Internally, Java 8’s streams framework uses a ForkJoinPool executor
- It supports algorithms that can be recursively split up
- Splitting is performed by a Spliterator

See docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html
Overview of Java 8 Parallel Streams

- Internally, Java 8’s streams framework uses a ForkJoinPool executor
- It supports algorithms that can be recursively split up
- Splitting is performed by a Spliterator
- Divides work up into smaller tasks that the ForkJoinPool executes

See docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html
Overview of Java 8 Parallel Streams

- Internally, Java 8’s streams framework uses a ForkJoinPool executor
- It supports algorithms that can be recursively split up
- Splitting is performed by a Spliterator
- Divides work up into smaller tasks that the ForkJoinPool executes

See docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html
Overview of Java 8 Parallel Streams

• Internally, Java 8’s streams framework uses a ForkJoinPool executor
  • It supports algorithms that can be recursively split up
  • Splitting is performed by a Spliterator
  • Idle threads in a pool “steal” work from busy threads

See docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html
Overview of Java 8 Parallel Streams

• Internally, Java 8’s streams framework uses a ForkJoinPool executor
  • It supports algorithms that can be recursively split up
  • Splitting is performed by a Spliterator
  • Idle threads in a pool “steal” work from busy threads

See docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html
Overview of Java 8 Parallel Streams

- Internally, Java 8’s streams framework uses a ForkJoinPool executor
  - It supports algorithms that can be recursively split up
  - Splitting is performed by a Spliterator
  - Idle threads in a pool “steal” work from busy threads

See docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html
Overview of Java 8 Parallel Streams

- Internally, Java 8’s streams framework uses a ForkJoinPool executor
- It supports algorithms that can be recursively split up
- Splitting is performed by a Spliterator
- Idle threads in a pool “steal” work from busy threads
  - Implemented via “circular work-stealing dequeues”

See www.dre.vanderbilt.edu/~schmidt/PDF/work-stealing-dequeue.pdf
Overview of Java 8 Parallel Streams

- Internally, Java 8’s streams framework uses a ForkJoinPool executor
  - It supports algorithms that can be recursively split up
  - Splitting is performed by a Spliterator
  - Idle threads in a pool “steal” work from busy threads
    - Implemented via “circular work-stealing deques”

See [www.infoq.com/interviews/doug-lea-fork-join](http://www.infoq.com/interviews/doug-lea-fork-join)
Overview of Java 8 Parallel Streams

- Internally, Java 8’s streams framework uses a ForkJoinPool executor
  - It supports algorithms that can be recursively split up
  - Splitting is performed by a Spliterator
  - Idle threads in a pool “steal” work from busy threads
  - ForkJoinPool size defaults to the number of cores available to the JVM

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html#ForkJoinPool--
Overview of Java 8 Parallel Streams

• Internally, Java 8’s streams framework uses a ForkJoinPool executor
  • It supports algorithms that can be recursively split up
  • Splitting is performed by a Spliterator
  • Idle threads in a pool “steal” work from busy threads
  • ForkJoinPool size defaults to the number of cores available to the JVM
  • The ForkJoinPool size can be controlled programmatically

System.setProperty("java.util.concurrent.ForkJoinPool.common.parallelism", numberOfThreads);
Overview of Java 8 Parallel Streams

• Internally, Java 8’s streams framework uses a ForkJoinPool executor
  • It supports algorithms that can be recursively split up
  • Splitting is performed by a Spliterator
  • Idle threads in a pool “steal” work from busy threads
  • ForkJoinPool size defaults to the number of cores available to the JVM
  • The ForkJoinPool size can be controlled programmatically

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

Modifying this property affects all the parallel streams in a process!

See [www.tobyhobson.co.uk/java-8-parallel-streams-fork-join-pool](http://www.tobyhobson.co.uk/java-8-parallel-streams-fork-join-pool)
End of Java 8 Parallel Streams (Part 1)
Java 8 Parallel Streams (Part 2)

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

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

• Recognize how Java 8 leverages its functional programming features in the parallel streams concurrency framework

• Know how these features are applied in a program that enhances the example in earlier lessons
Learning Objectives in this Part of the Lesson

• Recognize how Java 8 leverages its functional programming features in the parallel streams concurrency framework

• Know how these features are applied in a program that enhances the example in earlier lessons
  • SearchStreamGang enhances SearchTaskGang

See github.com/douglascraigschmidt/LiveLessons/tree/master/SearchStreamGang
Applying Parallel Streams to SearchStreamGang
Applying Parallel Streams to SearchStreamGang

- This SearchStreamGang variant uses two parallel streams

Printing results for input file 1 from fastest to slowest
PARALLEL_STREAM_INPUTS executed in 943 msecs
COMPLETABLE_FUTURES_INPUTS executed in 1035 msecs
hardCodedParallelStreamsSolution executed in 1239 msecs
PARALLEL_STREAMS executed in 2020 msecs
PARALLEL_STREAM_WORDS executed in 3173 msecs
SEQUENTIAL_STREAM executed in 3658 msecs
COMPLETABLE_FUTURES_WORDS executed in 3990 msecs

Printing results for input file 2 from fastest to slowest
COMPLETABLE_FUTURES_INPUTS executed in 518 msecs
PARALLEL_STREAM_INPUTS executed in 575 msecs
PARALLEL_STREAMS executed in 650 msecs
COMPLETABLE FUTURES_WORDS executed in 670 msecs

See SearchStreamGang/src/main/java/livelessons/streamgangs/SearchWithParallelStreams.java
Applying Parallel Streams to SearchStreamGang

- This SearchStreamGang variant uses two parallel streams
  - `processStream()`
Applying Parallel Streams to SearchStreamGang

- This SearchStreamGang variant uses two parallel streams
- `processStream()`
  - Uses a parallel stream to concurrently search a list of input strings

```
map(word -> searchForWord(...))
filter(result -> result.size() > 0)
collect(toList())
```
Applying Parallel Streams to SearchStreamGang

• This SearchStreamGang variant uses two parallel streams
  • `processStream()`
  • `processInput()`
Applying Parallel Streams to SearchStreamGang

- This SearchStreamGang variant uses two parallel streams
  - `processStream()`
  - `processInput()`

- `processInput()` uses a parallel stream to concurrently search each input string for all occurrences of words to find

```
map(word -> searchForWord(...))
filter(result -> result.size() > 0)
collect(toList())
```
Applying Parallel Streams to SearchStreamGang

- This `processStream()` implementation has one minuscule change

```java
protected List<List<SearchResults>> processStream() {
    return getInput()
        .parallelStream()
        .map(this::processInput)
        .collect(toList());
}
```
Applying Parallel Streams to SearchStreamGang

• This processStream() implementation has one minuscule change

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

*Creates a parallel stream that concurrently searches a list of input strings*
Applying Parallel Streams to SearchStreamGang

- This processStream() implementation has one minuscule change

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

See [docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html](http://docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html)
Likewise, this `processInput()` implementation also has one minuscule change.

```java
private List<SearchResults> processInput(String inputString) {
    String title = getTitle(inputString);
    String input = inputString.substring(title.length);

    List<SearchResults> results = mWordsToFind
        .parallelStream()
        .map(word -> searchForWord(word, input, title))
        .filter(result -> result.size() > 0)
        .collect(toList());

    return results;
}
```
Likewise, this processInput() implementation also has one minuscule change

```java
private List<SearchResults> processInput(String inputString) {
    String title = getTitle(inputString);
    String input = inputString.substring(title.length());

    List<SearchResults> results = mWordsToFind
        .parallelStream()
        .map(word -> searchForWord(word, input, title))
        .filter(result -> result.size() > 0)
        .collect(toList());

    return results;
}
```

Applying Parallel Streams to SearchStreamGang

Creates a parallel stream that concurrently searches each input string for all occurrences of words to find
Apply Parallel Streams to SearchStreamGang

Likewise, this processInput() implementation also has one minuscule change:

```java
private List<SearchResults> processInput(String inputString) {
    String title = getTitle(inputString);
    String input = inputString.substring(title.length());

    List<SearchResults> results = mWordsToFind
        .parallelStream()
        .map(word -> searchForWord(word, input, title))
        .filter(result -> result.size() > 0)
        .collect(toList());
    return results;
}
```

Creates a parallel stream that concurrently searches each input string for all occurrences of words to find.

See [docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html](http://docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html)
• Just because minuscule changes are needed doesn’t mean that parallel streams are always the right approach!

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

VS

List<List<SearchResults>>
processStream() {
    return getInput().parallelStream()
        .map(this::processInput)
        .collect(toList());
}
```
Just because minuscule changes are needed doesn’t mean that parallel streams are always the right approach!

Applying Parallel Streams to `SearchStreamGang`

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

**VS**

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

There’s no substitute for systematic benchmarking
End of Java 8 Parallel Streams (Part 2)
Java 8 Parallel Streams (Part 3)

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

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

• Recognize how Java 8 leverages its functional programming features in the parallel streams concurrency framework

• Know how these features are applied in a program that enhances the example in earlier lessons
  • SearchStreamGang enhances SearchTaskGang
  • ImageStreamGang enhances ImageTaskGang

Applying Parallel Streams to ImageStreamGang (Part 1)
Applying Parallel Streams to ImageStreamGang

- ImageStreamGang shows a more powerful use of parallel streams

Applying Parallel Streams to ImageStreamGang

- ImageStreamGang shows a more powerful use of parallel streams, e.g.,
  - Ignore cached images
  - Download non-cached images
  - Apply list of filters to each image
  - Store filtered images in the file system
  - Display images to the user

Combines Java 8 object-oriented & functional programming features
Applying Parallel Streams to ImageStreamGang

- ImageStreamGang shows a more powerful use of parallel streams, e.g.,
  - Ignore cached images
  - Download non-cached images
  - Apply list of filters to each image
  - Store filtered images in the file system
  - Display images to the user (after triggering stream processing)

Declarative stream pipeline closely aligns with the app description

```
List of URLs to Download

filter(not(this::urlCached))

map(this::downloadImage)

flatMap(this::applyFilters)

collect(toList())
```
Applying Parallel Streams to ImageStreamGang

- ImageStreamGang shows a more powerful use of parallel streams, e.g.,
  - Ignores cached images
  - Downloads non-cached images
  - Applies a list of filters
  - Stores images in file system
  - Displays images to the user

Closes gap between design intent & computations that implement the intent
Applying Parallel Streams to ImageStreamGang

- ImageStreamGang shows a more powerful use of parallel streams, e.g.,
  - Ignore cached images

```
List of URLs to Download

filter(not(this::urlCached))

map(this::downloadImage)

flatMap(this::applyFilters)

collect(toList())
```
Applying Parallel Streams to ImageStreamGang

- ImageStreamGang shows a more powerful use of parallel streams, e.g.,
  - Ignore cached images
  - Download non-cached images

```
List of URLs to Download

filter(not(this::urlCached))

map(this::downloadImage)

flatMap(this::applyFilters)

collect(toList())
```
Applying Parallel Streams to ImageStreamGang

- ImageStreamGang shows a more powerful use of parallel streams, e.g.,
  - Ignore cached images
  - Download non-cached images
  - Apply list of filters to each image
Applying Parallel Streams to ImageStreamGang

- ImageStreamGang shows a more powerful use of parallel streams, e.g.,
  - Ignore cached images
  - Download non-cached images
  - Apply list of filters to each image
  - Store filtered images in the file system

Diagram:
- List of URLs to Download
- Filter: `not(this::urlCached)`
- Map: `this::downloadImage`
- FlatMap: `this::applyFilters`
- Collect: `toList()`
Applying Parallel Streams to ImageStreamGang

- ImageStreamGang shows a more powerful use of parallel streams, e.g.,

  - Ignore cached images
  - Download non-cached images
  - Apply list of filters to each image
  - Store filtered images in the file system
  - Display images to the user (after triggering stream processing)
Applying Parallel Streams to ImageStreamGang

- ImageStreamGang shows a more powerful use of parallel streams, e.g.,
  - Ignore cached images
  - Download non-cached images
  - Apply list of filters to each image
  - Store filtered images in the file system
  - Display images to the user (after triggering stream processing)

The Java 8 streams framework orchestrates all these steps concurrently.
Applying Parallel Streams to ImageStreamGang

- ImageStreamGang shows a more powerful use of parallel streams, e.g.,
  - Ignore cached images
  - Download non-cached images
  - Apply list of filters to each image
  - Store filtered images in the file system
  - Display images to the user (after triggering stream processing)

The Java 8 streams framework orchestrates all these steps concurrently.
Applying Parallel Streams to ImageStreamGang (Part 2)
void processStream() {
    List<Image> filteredImages =
    getInput().parallelStream()
        .filter(not(this::urlCached))
        .map(this::downloadImage)
        .flatMap(this::applyFilters)
        .collect(toList());

    Log.d(TAG,
            "Image(s) filtered = 
            + filteredImages.size());
}
Applying Parallel Streams to ImageStreamGang

- We focus on processStream() in ImageStreamParallel.java

```java
void processStream() {
    List<Image> filteredImages =
        getInput()
            .parallelStream()
            .filter(not(this::urlCached))
            .map(this::downloadImage)
            .flatMap(this::applyFilters)
            .collect(toList());

    Log.d(TAG,
           "Image(s) filtered = "
           + filteredImages.size());
}
```
Applying Parallel Streams to ImageStreamGang

We focus on `processStream()` in `ImageStreamParallel.java`

```java
void processStream() {
    List<Image> filteredImages =
        getInput()
            .parallelStream()
            .filter(not(this::urlCached))
            .map(this::downloadImage)
            .flatMap(this::applyFilters)
            .collect(toList());

    Log.d(TAG,
          "Image(s) filtered = ",
          + filteredImages.size());
}
```

Convert a collection into a parallel stream
Applying Parallel Streams to ImageStreamGang

We focus on processStream() in ImageStreamParallel.java

```java
void processStream() {
    List<Image> filteredImages =
        getInput()
            .parallelStream()
            .filter(not(this::urlCached))
            .map(this::downloadImage)
            .flatMap(this::applyFilters)
            .collect(toList());

    Log.d(TAG,
         "Image(s) filtered = 
         + filteredImages.size()";
}
```

Returns an output stream consisting of the URLs in the input stream that are not already cached.

See docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#filter-java.util.function.Predicate-
void processStream() {
    List<Image> filteredImages =
    getInput()
        .parallelStream()
        .filter(not(this::urlCached))
        .map(this::downloadImage)
        .flatMap(this::applyFilters)
        .collect(toList());

    Log.d(TAG,
        "Image(s) filtered = "
        + filteredImages.size());
}

Returns an output stream consisting of the URLs in the input stream that are not already cached

See docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#filter-java.util.function.Predicate-
Applying Parallel Streams to ImageStreamGang

- We focus on `processStream()` in `ImageStreamParallel.java`

```java
void processStream() {
    List<Image> filteredImages =
        getInput()
            .parallelStream()
            .filter(not(this::urlCached))
            .map(this::downloadImage)
            .flatMap(this::applyFilters)
            .collect(toList());

    Log.d(TAG,
        "Image(s) filtered = "
        + filteredImages.size());
}
```

*Returns an output stream consisting of the URLs in the input stream that are not already cached*

Number of output stream elements may be less than number of input stream elements
Applying Parallel Streams to ImageStreamGang

- We focus on processStream() in ImageStreamParallel.java

```java
void processStream() {
    List<Image> filteredImages =
        getInput()
            .parallelStream()
            .filter(not(this::urlCached))
            .map(this::downloadImage)
            .flatMap(this::applyFilters)
            .collect(toList());

    Log.d(TAG,
        "Image(s) filtered = "
        + filteredImages.size());
}
```

Returns an output stream consisting of the images that were downloaded from the URLs in the input stream

Applying Parallel Streams to ImageStreamGang

- We focus on `processStream()` in `ImageStreamParallel.java`

```java
void processStream() {
    List<Image> filteredImages =
        getInput()
            .parallelStream()
            .filter(not(this::urlCached))
            .map(this::downloadImage)
            .flatMap(this::applyFilters)
            .collect(toList());

    Log.d(TAG,
        "Image(s) filtered = "
        + filteredImages.size());
}
```

Returns an output stream consisting of the images that were downloaded from the URLs in the input stream.
Applying Parallel Streams to ImageStreamGang

- We focus on processStream() in ImageStreamParallel.java

```java
void processStream() {
    List<Image> filteredImages =
        getInput()
            .parallelStream()
            .filter(not(this::urlCached))
            .map(this::downloadImage)
            .flatMap(this::applyFilters)
            .collect(toList());

    Log.d(TAG,
          "Image(s) filtered = "
          + filteredImages.size());
}
```

Returns an output stream consisting of the images that were downloaded from the URLs in the input stream.

Number of output stream elements matches number of input stream elements.
void processStream() {
    List<Image> filteredImages =
        getInput()
            .parallelStream()
            .filter(not(this::urlCached))
            .map(this::downloadImage)
            .flatMap(this::applyFilters)
            .collect(toList());
    Log.d(TAG,
        "Image(s) filtered = "
        + filteredImages.size());
}
void processStream() {
    List<Image> filteredImages =
        getInput()
            .parallelStream()
            .filter(not(this::urlCached))
            .map(this::downloadImage)
            .flatMap(this::applyFilters)
            .collect(toList());

    Log.d(TAG,
          "Image(s) filtered = "
            + filteredImages.size());
}
Applying Parallel Streams to ImageStreamGang

- We focus on `processStream()` in `ImageStreamParallel.java`

```java
void processStream() {
    List<Image> filteredImages =
        getInput()
            .parallelStream()
            .filter(not(this::urlCached))
            .map(this::downloadImage)
            .flatMap(this::applyFilters)
            .collect(toList());

    Log.d(TAG,
            "Image(s) filtered = "
            + filteredImages.size());
}
```

Returns an output stream containing the results of applying a list of filters to each image in the input stream & storing the results in the file system.

Number of output stream elements may differ from number of input stream elements.
Applying Parallel Streams to ImageStreamGang

- We focus on `processStream()` in `ImageStreamParallel.java`

```java
void processStream() {
    List<Image> filteredImages =
        getInput()
            .parallelStream()
            .filter(not(this::urlCached))
            .map(this::downloadImage)
            .flatMap(this::applyFilters)
            .collect(toList());

    Log.d(TAG,
          "Image(s) filtered = "
          + filteredImages.size());
}
```

Terminal operations yield values or side-effects, as well as triggering stream processing.

See docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#collect-java.util.stream.Collector-
Applying Parallel Streams to ImageStreamGang

- We focus on `processStream()` in `ImageStreamParallel.java`

```java
void processStream() {
    List<Image> filteredImages =
        getInput()
            .parallelStream()
            .filter(not(this::urlCached))
            .map(this::downloadImage)
            .flatMap(this::applyFilters)
            .collect(toList());

    Log.d(TAG,
        "Image(s) filtered = "
        + filteredImages.size());
}
```

`collect()` is a “reduction” operation that combines elements into one result.
Applying Parallel Streams to ImageStreamGang

- We focus on `processStream()` in `ImageStreamParallel.java`

```java
void processStream() {
    List<Image> filteredImages =
        getInput()
            .parallelStream()
            .filter(not(this::urlCached))
            .map(this::downloadImage)
            .flatMap(this::applyFilters)
            .collect(toList());

    Log.d(TAG,
        "Image(s) filtered = ",
        + filteredImages.size());
}
```

*Writes the number of images downloaded, filtered, & stored to logcat*
End of Java 8 Parallel Streams (Part 3)
Java 8 Parallel Streams (Part 4)

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

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

- Recognize how Java 8 leverages its functional programming features to create new concurrency frameworks
- Know how these features are applied in several programs that enhance examples in other lessons
- Evaluate the pros & cons of Java 8 parallel streams
Evaluating Java 8 Parallel Streams
Evaluating Java 8 Parallel Streams

- Parallel stream implementations are faster than sequential stream implementations

The performance of `processStream()` is largely a function of the number of cores.
Evaluating Java 8 Parallel Streams

No explicit synchronization is required in this implementation.
Java libraries handle any locking needed to read/write to files & connections.
Evaluating Java 8 Parallel Streams

• Converting `processStream()` to use a sequential stream requires a minuscule change!

```java
void processStream() {
    List<Image> filteredImages =
        getInput()
            .stream()
            .filter(not(this::urlCached))
            .map(this::downloadImage)
            .flatMap(this::applyFilters)
            .collect(toList());

    Log.d(TAG,
        "Image(s) filtered = "+ filteredImages.size());
}
```

Evaluating Java 8 Parallel Streams

• Converting `processStream()` to use a sequential stream requires a minuscule change!

```java
void processStream()
{
    List<Image> filteredImages =
        getInput()
            .stream()
            .filter(not(this::urlCached))
            .map(this::downloadImage)
            .flatMap(this::applyFilters)
            .collect(toList());

    Log.d(TAG,
        "Image(s) filtered = "
        + filteredImages.size());
}
```

Evaluating Java 8 Parallel Streams

- Converting `processStream()` to use a sequential stream requires a minuscule change!

```java
void processStream() {
    List<Image> filteredImages =
        getInput()
            .parallelStream()
            .filter(not(this::urlCached))
            .map(this::downloadImage)
            .flatMap(this::applyFilters)
            .collect(toList());

    Log.d(TAG, "Image(s) filtered = "+ filteredImages.size());
}
```

• The Java 8 streams framework shields programmers from the details of concurrently splitting, processing, & joining the results

```java
void processStream() {
    List<Image> filteredImages =
        getInput()
            .parallelStream()
            .filter(not(this::urlCached))
            .map(this::downloadImage)
            .flatMap(this::applyFilters)
            .collect(toList());

    Log.d(TAG,
        "Image(s) filtered = "
            + filteredImages.size());
}
```
Evaluating Java 8 Parallel Streams

- Examples show synergies between functional & object-oriented programming
Evaluating Java 8 Parallel Streams

StreamGang’s object-oriented design is easy to understand, reuse, & extend.
Evaluating Java 8 Parallel Streams

• processStream() hook methods close the gap between domain intent & computations performed

```java
void processStream() {
    List<Image> filteredImages =
        getInput()
            .stream()
            .filter(not(this::urlCached))
            .map(this::downloadImage)
            .flatMap(this::applyFilters)
            .collect(toList());

    Log.d(TAG,
        "Image(s) filtered = "
        + filteredImages.size());
}
```
Evaluating Java 8 Parallel Streams

- `processStream()` hook methods close the gap between domain intent & computations performed

```java
void processStream() {
    List<Image> filteredImages =
        getInput().stream()
            .filter(not(this::urlCached))
            .map(this::downloadImage)
            .flatMap(this::applyFilters)
            .collect(toList());

    Log.d(TAG,
        "Image(s) filtered = ",
        + filteredImages.size());
}
```
Evaluating Java 8 Parallel Streams

- There are some limitations with Java 8 parallel streams
Evaluating Java 8 Parallel Streams

- There are some limitations with Java 8 parallel streams, e.g.
  - Not all problems can be decomposed into the map/reduce model

See [dzone.com/articles/whats-wrong-java-8-part-iii](dzone.com/articles/whats-wrong-java-8-part-iii)
Evaluating Java 8 Parallel Streams

• There are some limitations with Java 8 parallel streams, e.g.
  • Not all problems can be decomposed into the map/reduce model
  • Race conditions may occur if functions called by aggregate operations aren’t thread-safe

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
Evaluating Java 8 Parallel Streams

- There are some limitations with Java 8 parallel streams, e.g.
  - Not all problems can be decomposed into the map/reduce model
  - Race conditions may occur if functions called by aggregate operations aren’t thread-safe
  - All parallel streams share the same ForkJoinPool

See dzone.com/articles/think-twice-using-java-8
Evaluating Java 8 Parallel Streams

- There are some limitations with Java 8 parallel streams, e.g.
  - Not all problems can be decomposed into the map/reduce model
  - Race conditions may occur if functions called by aggregate operations aren’t thread-safe
  - All parallel streams share the same ForkJoinPool
  - The parallel streams framework incurs overhead due to its use of the ForkJoinPool

See coopsoft.com/dl/Blunder.pdf
Evaluating Java 8 Parallel Streams

- In general, however, the pros of Java 8 parallel streams outweigh the cons in many common use cases!!

Evaluating Java 8 Parallel Streams

- Additional material on Java 8 parallel streams appears in the book “Java 8 in Action”

See www.manning.com/books/java-8-in-action
End of Java 8 Parallel Streams (Part 4)