Java Sequential SearchStreamGang Example: Evaluating Pros & Cons

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

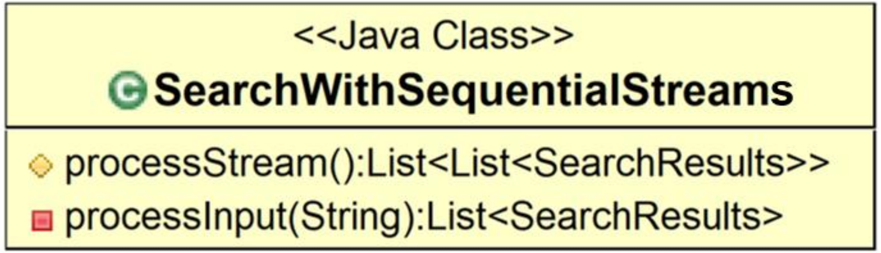
Professor of Computer Science

Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Part of the Lesson

- Know how to apply sequential streams to the SearchStreamGang program
- Recognize how a Spliterator is used in SearchWithSequentialStreams
- Understand the pros & cons of the SearchWithSequentialStreams class



<<Java Class>>

SearchWithSequentialStreams

◇ processStream():List<List<SearchResults>>

■ processInput(String):List<SearchResults>

See SearchStreamGang/src/main/java/livelessons/streamgangs/SearchWithSequentialStreams.java

Pros of the SearchWith SequentialStreams Class

Pros of the SearchWithSequentialStreams Class

- There are several benefits with this sequential streams implementation

```
List<SearchResults> processInput(CharSequence inputSeq) {
  String title = getTitle(inputString);
  CharSequence input = inputSeq.subSequence(...);

  List<SearchResults> results = mPhrasesToFind
    .stream()
    .map(phrase -> searchForPhrase
            (phrase, input, title, false))

    .filter(not(SearchResults::isEmpty))

    .collect(toList());
  return results;
}
```

Pros of the SearchWithSequentialStreams Class

- There are several benefits with this sequential streams implementation

```
List<SearchResults> processInput(CharSequence inputSeq) {
  String title = getTitle(inputString);
  CharSequence input = inputSeq.subSequence(...);

  List<SearchResults> results = mPhrasesToFind
    .stream()
    .map(phrase -> searchForPhrase
                (phrase, input, title, false))

    .filter(not(SearchResults::isEmpty))

    .collect(toList());
  return results;
}
```

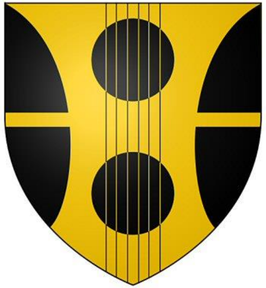
Streams use "internal" iterators versus "external" iterators used by collections

See www.javabrahman.com/java-8/java-8-internal-iterators-vs-external-iterators

Pros of the SearchWithSequentialStreams Class

- There are several benefits with this sequential streams implementation

```
List<SearchResults> processInput(CharSequence inputSeq) {
  String title = getTitle(inputString);
  CharSequence input = inputSeq.subSequence(...);

  List<SearchResults> results = mPhrasesToFind
    .stream()
    .map(phrase -> searchForPhrase
               (phrase, input, title, false))

    .filter(not(SearchResults::isEmpty))

    .collect(toList());
  return results;
}
```



Internal iterators shield programs from streams processing implementation details

Pros of the SearchWithSequentialStreams Class

- There are several benefits with this sequential streams implementation

```
List<SearchResults> processInput(CharSequence inputSeq) {
  String title = getTitle(inputString);
  CharSequence input = inputSeq.subSequence(...);

  List<SearchResults> results = mPhrasesToFind
    .stream()
    .map(phrase -> searchForPhrase
                 (phrase, input, title, false))

    .filter(not(SearchResults::isEmpty))

    .collect(toList());
  return results;
}
```

This pipeline is declarative since it's a series of transformations performed by aggregate operations

There are no explicit control constructs or memory allocations in this pipeline!

Pros of the SearchWithSequentialStreams Class

- There are several benefits with this sequential streams implementation

```
List<SearchResults> processInput(CharSequence inputSeq) {
  String title = getTitle(inputString);
  CharSequence input = inputSeq.subSequence(...);

  List<SearchResults> results = mPhrasesToFind
    .stream()
    .map(phrase -> searchForPhrase
               (phrase, input, title, false))

    .filter(not(SearchResults::isEmpty))

    .collect(toList());
  return results;
}
```

Focus on “what” operations to perform, rather than on “how” they’re implemented

Pros of the SearchWithSequentialStreams Class

- There are several benefits with this sequential streams implementation

```
List<SearchResults> processInput(CharSequence inputSeq) {
  String title = getTitle(inputString);
  CharSequence input = inputSeq.subSequence(...);

  List<SearchResults> results = mPhrasesToFind
    .stream()
    .map(phrase -> searchForPhrase
           (phrase, input, title, false))

    .filter(not(SearchResults::isEmpty))

    .collect(toList());
  return results;
}
```

These behaviors have no side-effects

Pros of the SearchWithSequentialStreams Class

- There are several benefits with this sequential streams implementation

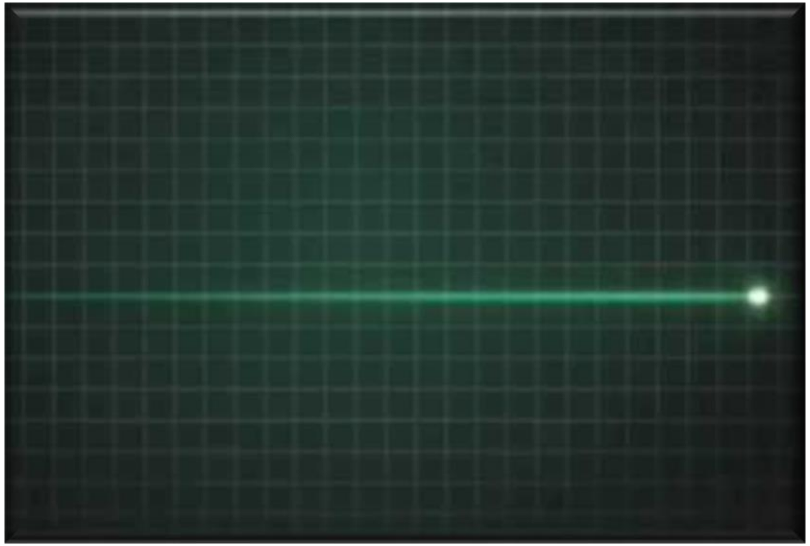
```
List<SearchResults> processInput(CharSequence inputSeq) {
  String title = getTitle(inputString);
  CharSequence input = inputSeq.subSequence(...);

  List<SearchResults> results = mPhrasesToFind
    .stream()
    .map(phrase -> searchForPhrase
           (phrase, input, title, false))

    .filter(not(SearchResults::isEmpty))

    .collect(toList());
  return results;
}
```

No side-effects makes it easier to reason about behavior & enables optimization

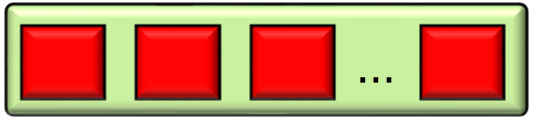
Cons of the SearchWith SequentialStreams Class

Cons of the SearchWithSequentialStreams Class

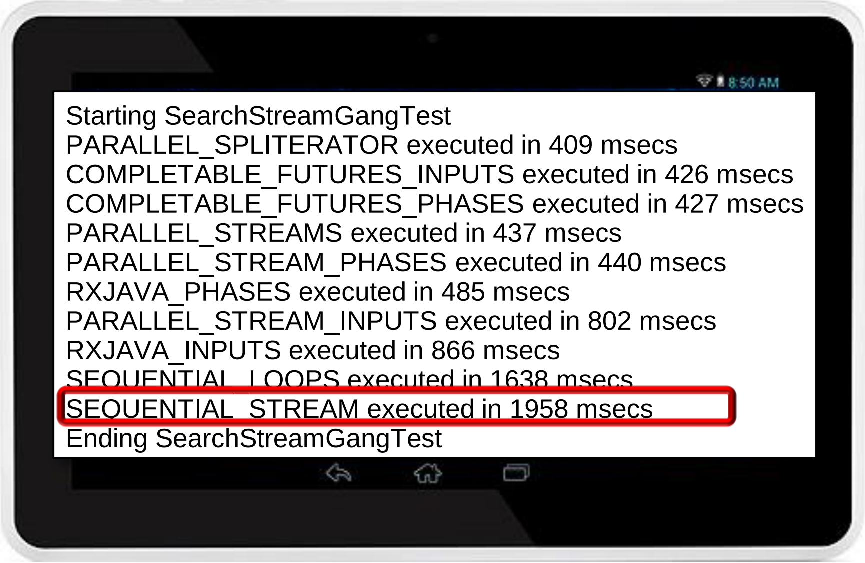
- The sequential implementation can't take advantage of multi-core processors



Input Strings to Search



Search Phrases



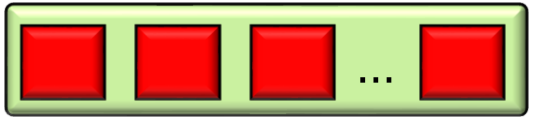
```
Starting SearchStreamGangTest
PARALLEL_SPLITERATOR executed in 409 msecs
COMPLETABLE_FUTURES_INPUTS executed in 426 msecs
COMPLETABLE_FUTURES_PHASES executed in 427 msecs
PARALLEL_STREAMS executed in 437 msecs
PARALLEL_STREAM_PHASES executed in 440 msecs
RXJAVA_PHASES executed in 485 msecs
PARALLEL_STREAM_INPUTS executed in 802 msecs
RXJAVA_INPUTS executed in 866 msecs
SEQUENTIAL_LOOPS executed in 1638 msecs
SEQUENTIAL_STREAM executed in 1958 msecs
Ending SearchStreamGangTest
```

Tests conducted on a quad-core Lenovo P50 with 32 Gbytes of RAM

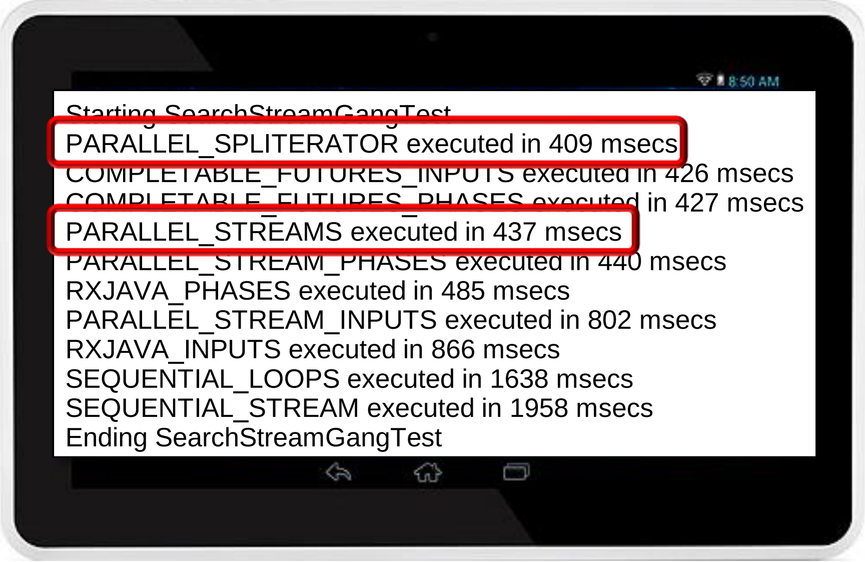
Cons of the SearchWithSequentialStreams Class

- The sequential implementation can't take advantage of multi-core processors
  - Parallel streams can often provide a significant performance boost!

Input Strings to Search



Search Phrases



```
Starting SearchStreamGangTest
PARALLEL_SPLITERATOR executed in 409 msecs
COMPLETABLE_FUTURES_INPUTS executed in 426 msecs
COMPLETABLE_FUTURES_PHASES executed in 427 msecs
PARALLEL_STREAMS executed in 437 msecs
PARALLEL_STREAM_PHASES executed in 440 msecs
RXJAVA_PHASES executed in 485 msecs
PARALLEL_STREAM_INPUTS executed in 802 msecs
RXJAVA_INPUTS executed in 866 msecs
SEQUENTIAL_LOOPS executed in 1638 msecs
SEQUENTIAL_STREAM executed in 1958 msecs
Ending SearchStreamGangTest
```

See upcoming lessons on "*Java Parallel Streams*"

Cons of the SearchWithSequentialStreams Class

- This class only used a few Java aggregate operations

```
List<SearchResults> processInput(CharSequence inputSeq) {
  String title = getTitle(inputString);
  CharSequence input = inputSeq.subSequence(...);

  List<SearchResults> results = mPhrasesToFind
    .stream()
    .map(phrase
         -> searchForPhrase(phrase, input, title))

    .filter(not(SearchResults::isEmpty))

    .collect(toList());
  return results; ...
```

Cons of the SearchWithSequentialStreams Class

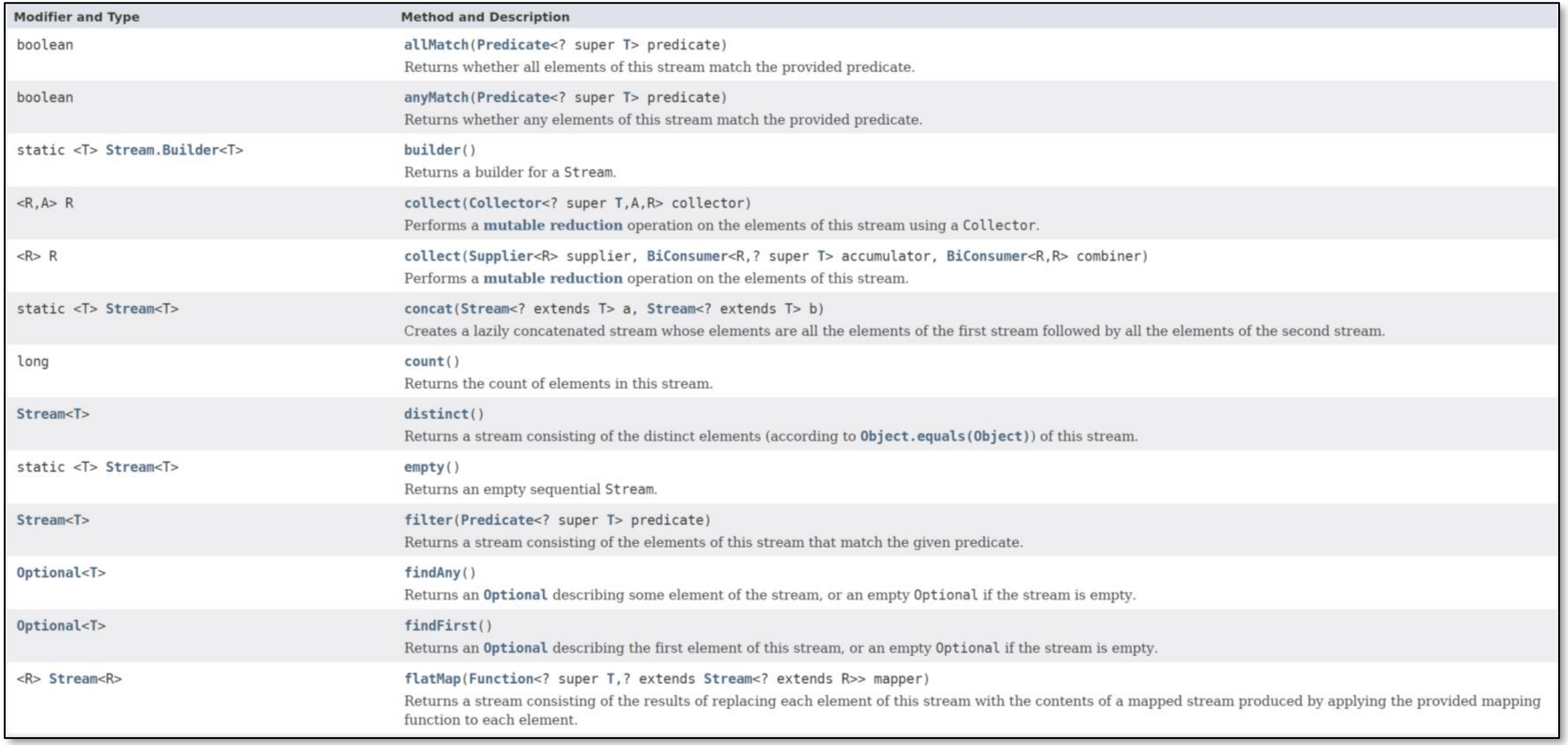
- This class only used a few Java aggregate operations

```
List<SearchResults> processInput(CharSequence inputSeq) {
  String title = getTitle(inputString);
  CharSequence input = inputSeq.subSequence(...);

  List<SearchResults> results
    .stream()
    .map(phrase
             -> searchForPhrase(p

    .filter(not(SearchResults

    .collect(toList());
  return results; ...
```

However, these aggregate operations are also useful for parallel streams

Cons of the SearchWithSequentialStreams Class

- *Many* other aggregate operations are part of the Java stream API

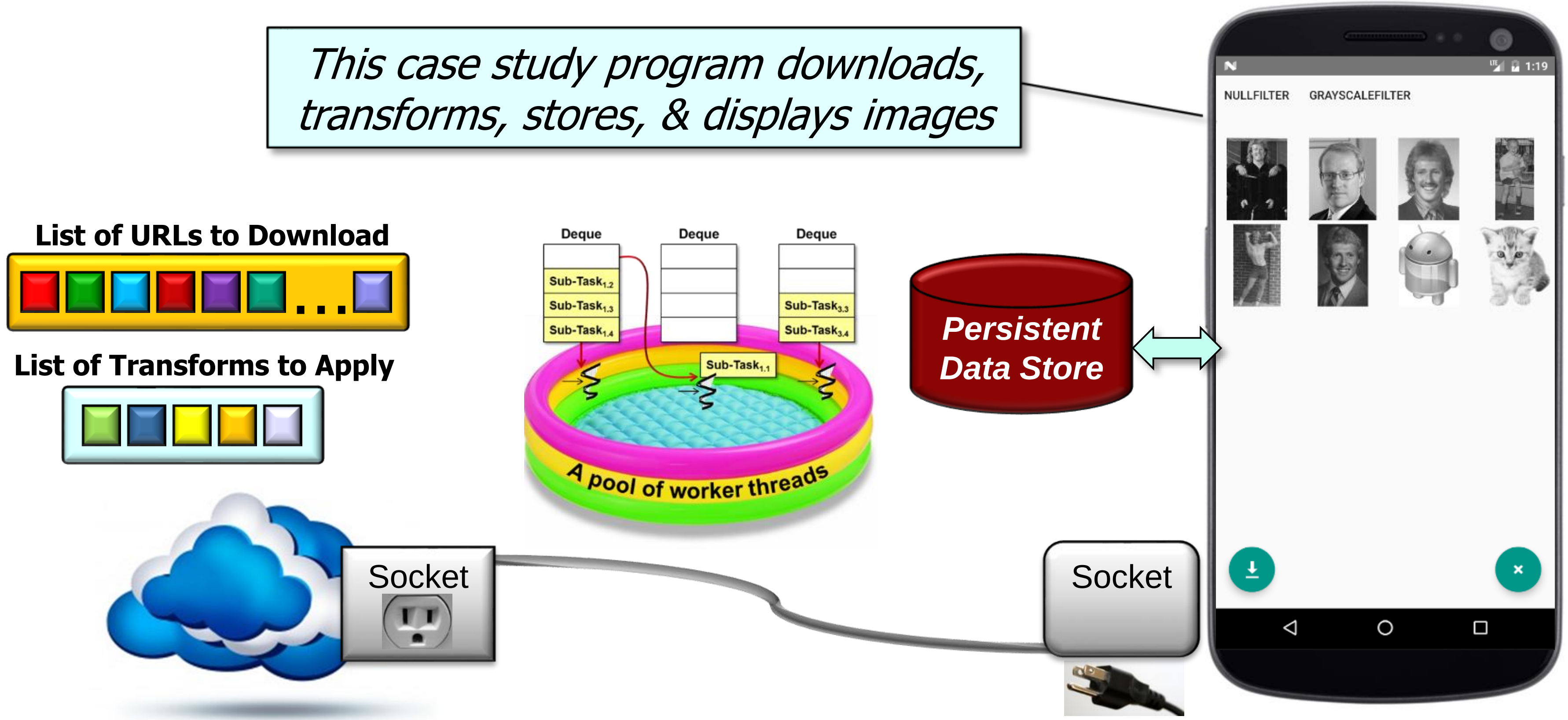


Modifier and Type	Method and Description
boolean	allMatch(Predicate<? super T> predicate) Returns whether all elements of this stream match the provided predicate.
boolean	anyMatch(Predicate<? super T> predicate) Returns whether any elements of this stream match the provided predicate.
static <T> Stream.Builder<T>	builder() Returns a builder for a Stream.
<R,A> R	collect(Collector<? super T,A,R> collector) Performs a **mutable reduction** operation on the elements of this stream using a Collector.
<R> R	collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner) Performs a **mutable reduction** operation on the elements of this stream.
static <T> Stream<T>	concat(Stream<? extends T> a, Stream<? extends T> b) Creates a lazily concatenated stream whose elements are all the elements of the first stream followed by all the elements of the second stream.
long	count() Returns the count of elements in this stream.
Stream<T>	distinct() Returns a stream consisting of the distinct elements (according to Object.equals(Object)) of this stream.
static <T> Stream<T>	empty() Returns an empty sequential Stream.
Stream<T>	filter(Predicate<? super T> predicate) Returns a stream consisting of the elements of this stream that match the given predicate.
Optional<T>	findAny() Returns an Optional describing some element of the stream, or an empty Optional if the stream is empty.
Optional<T>	findFirst() Returns an Optional describing the first element of this stream, or an empty Optional if the stream is empty.
<R> Stream<R>	flatMap(Function<? super T,? extends Stream<? extends R>> mapper) 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 mapping function to each element.

See docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

Cons of the SearchWithSequentialStreams Class

- *Many* other aggregate operations are part of the Java stream API, e.g.



This case study program downloads, transforms, stores, & displays images

See *"Java Parallel ImageStreamGang Example"*

End of Java Sequential SearchStreamGang Example: Evaluating Pros & Cons