Java 8 Sequential SearchStreamGang

Example (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

• Know how to apply sequential streams to the SearchStreamGang program

• Recognize how a Spliterator is used in SearchWithSequentialStreams

```java
SearchResults searchForPhrase(String phrase, CharSequence input, String title, boolean parallel) {
    return new SearchResults(..., phrase, ..., StreamSupport.
        stream(new PhraseMatchSpliterator(input, phrase),
            parallel)
        .collect(toList()));
}
```
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
Using Java Spliterator in SearchStreamGang
Using Java Spliterator in SearchStreamGang

- SearchStreamGang uses PhraseMatchSpliterator that works for both sequential & parallel streams

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

Using Java Spliterator in SearchStreamGang

- SearchStreamGang uses PhraseMatchSpliterator that works for both sequential & parallel streams
- We focus on the sequential portions now

```java
stream().
  map(phrase -> searchForPhrase(...)).
  filter(not(SearchResults::isEmpty)).
  collect(toList());
```
Using Java Spliterator in SearchStreamGang

- SearchStreamGang uses PhraseMatchSpliterator that works for both sequential & parallel streams
- We focus on the sequential portions now
- We’ll cover the parallel portions later

See “Java 8 Parallel SearchStreamGang Example (Part 2)”
Using Java Spliterator in SearchStreamGang

- `searchForPhrase()` uses `PhraseMatchSpliterator` to find all phrases in input & return `SearchResults`

```java
SearchResults searchForPhrase(String phrase, CharSequence input, String title, boolean parallel) {
    return new SearchResults(..., phrase, ..., StreamSupport.stream(new PhraseMatchSpliterator(mInput, word), parallel)
        .collect(toList()));
}
```

• searchForPhrase() uses PhraseMatchSpliterator to find all phrases in input & return SearchResults

SearchResults searchForPhrase
(String phrase, CharSequence input,
String title, boolean parallel) {
  return new SearchResults
    (... , phrase, ..., StreamSupport
      .stream(new PhraseMatchSpliterator
        (input, phrase),
        parallel)
      .collect(toList()));
}

StreamSupport.stream() creates a sequential or parallel stream via PhraseMatchSpliterator

See [docs.oracle.com/javase/8/docs/api/java/util/stream/StreamSupport.html#stream](docs.oracle.com/javase/8/docs/api/java/util/stream/StreamSupport.html#stream)
Using Java Splitterator in SearchStreamGang

- `searchForPhrase()` uses `PhraseMatchSpliterator` to find all phrases in input & return `SearchResults`

```java
SearchResults searchForPhrase(String phrase, CharSequence input, String title, boolean parallel) {
  return new SearchResults(..., phrase, ..., StreamSupport.stream(new PhraseMatchSpliterator(input, phrase), parallel).
          .collect(toList()));
}
```

For `SearchWithSequentialStreams` "parallel" is false, so we’ll use a sequential splitterator

See [docs.oracle.com/javase/8/docs/api/java/util/stream/StreamSupport.html#stream](docs.oracle.com/javase/8/docs/api/java/util/stream/StreamSupport.html#stream)
Here's the input/output of PhraseMatchSpliterator for SearchWithSequentialStreams

List `<String>`

Stream `<String>`

Stream `<SearchResults>`

Stream `<SearchResults>`

List `<SearchResults>`

stream()

map(phrase -> searchForPhrase(…))

filter(not(SearchResults::isEmpty))

collect(toList())

Using Java Spliterator in SearchStreamGang
Using Java Spliterator in SearchStreamGang

Here’s the input/output of PhraseMatchSpliterator for SearchWithSequentialStreams

“...
My liege, and madam, to expostulate
What majesty should be, what duty is,
Why day is day, night is night, and time is time.
Were nothing but to waste night, day, and time.
Therefore, since brevity is the soul of wit,
And tediousness the limbs and outward flourishes,
I will be brief. ...”

"Brevity is the soul of wit” matches at index [54739]
Using Java Spliterator in SearchStreamGang

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults Result objects that match the # of times a phrase appears in an input string.

```java
class PhraseMatchSpliterator implements Spliterator<Result> {
    private CharSequence mInput;
    private final String mPhrase;
    private final Pattern mPattern;
    private Matcher mPhraseMatcher;
    private final int mMinSplitSize;
    private int mOffset = 0;
    ...
}
```

See SearchStreamGang/src/main/java/livelessons/utils/PhraseMatchSpliterator.java
Using Java Spliterator in SearchStreamGang

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults Result objects that match the # of times a phrase appears in an input string

```java
class PhraseMatchSpliterator implements Spliterator<Result> {
    private CharSequence mInput;
    private final String mPhrase;
    private final Pattern mPattern;
    private Matcher mPhraseMatcher;
    private final int mMinSplitSize;
    private int mOffset = 0;
    ...
}
```

Splitter is an interface that defines eight methods, including tryAdvance() & trySplit()
Using Java Spliterator in SearchStreamGang

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults Result objects that match the # of times a phrase appears in an input string.

```java
class PhraseMatchSpliterator implements Spliterator<Result> {
    private CharSequence mInput;
    private final String mPhrase;
    private final Pattern mPattern;
    private Matcher mPhraseMatcher;
    private final int mMinSplitSize;
    private int mOffset = 0;
    ...
```

These fields implement PhraseMatchSpliterator for both sequential & parallel use-cases.

Some fields are updated in the trySplit() method, which is why they aren’t final.
PhraseMatchSpliterator uses Java regex to create a stream of SearchResults Result objects that match the # of times a phrase appears in an input string.

```java
class PhraseMatchSpliterator implements Spliterator<Result> {

    PhraseMatchSpliterator(CharSequence input, String phrase) {
        String regexPhrase = "\\b" + phrase.trim().replaceAll("\\s+", "\\\\b\\\\s+\\\\b") + "\\b";
        mPattern = Pattern.compile(regexPhrase,
                                   Pattern.CASE_INSENSITIVE | Pattern.DOTALL);
        mPhraseMatcher = mPattern.matcher(input);
        mInput = input; mPhrase = phrase;
        mMinSplitSize = input.length() / 2;
    }
    ...
}
```

See [docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html](docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html)
Using Java Spliterator in SearchStreamGang

- **PhraseMatchSpliterator** uses Java regex to create a stream of SearchResults Result objects that match the # of times a phrase appears in an input string.

class PhraseMatchSpliterator implements Spliterator<SearchResults> {
    ...
    PhraseMatchSpliterator(CharSequence input, String phrase) {
        String regexPhrase = "\\b" + phrase.trim().replaceAll("\\s+", "\\\\b\\\\s+\\\\b")
            + "\\b"; ...  

        mPattern = Pattern.compile(regexPhrase, Pattern.CASE_INSENSITIVE | Pattern.DOTALL);
        mPhraseMatcher = mPattern.matcher(input);
        mInput = input; mPhrase = phrase;
        mMinSplitSize = input.length() / 2;
    } ...

    A matcher is created to search the input for the regex pattern

See docs.oracle.com/javase/8/docs/api/java/util/regex/Matcher.html
Using Java Spliterator in SearchStreamGang

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults Result objects that match the # of times a phrase appears in an input string.

```java
class PhraseMatchSpliterator implements Spliterator<Result> {
    ...

    PhraseMatchSpliterator(CharSequence input, String phrase) {
        String regexPhrase = "\\b" + phrase.trim().replaceAll("\\s+", "\\b\\s+\\b") + "\\b";
        mPattern = Pattern.compile(regexPhrase,
                                      Pattern.CASE_INSENSITIVE | Pattern.DOTALL);
        mPhraseMatcher = mPattern.matcher(input);
        mInput = input; mPhrase = phrase;
        mMinSplitSize = input.length() / 2;
    }
    ...
```
Using Java Spliterator in SearchStreamGang

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults Result objects that match the # of times a phrase appears in an input string

```java
class PhraseMatchSpliterator implements Spliterator<Result> {
    ...
    boolean tryAdvance(Consumer<? super Result> action) {
        if (!mPhraseMatcher.find())
            return false;
        else {
            action.accept(new Result(
                mOffset + mPhraseMatcher.start()));
            return true;
        }
    }
    ...
}
```

This method plays the role of hasNext() & next() in Java’s Iterator interface

See docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html#tryAdvance
Using Java Spliterator in SearchStreamGang

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults Result objects that match the # of times a phrase appears in an input string.

```java
class PhraseMatchSpliterator implements Spliterator<Result> {

    ... boolean tryAdvance(Consumer<? super Result> action) {
        if (!mPhraseMatcher.find())
            return false;
        else {
            action.accept(new Result(mOffset + mPhraseMatcher.start()));
            return true;
        }
    }

    ...}
```

Passes the result (if any) back "by reference"

See [docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html)
PhraseMatchSpliterator uses Java regex to create a stream of SearchResults Result objects that match the # of times a phrase appears in an input string.

```java
class PhraseMatchSpliterator implements Spliterator<Result> {
    ...
    boolean tryAdvance(Consumer<? super Result> action) {
        if (!mPhraseMatcher.find())
            return false;
        else {
            action.accept(new Result(mOffset + mPhraseMatcher.start()));
            return true;
        }
    }
    ...
    }
```

It first checks if there are any remaining phrases in the input that match the regex.

See docs.oracle.com/javase/8/docs/api/java/util/regex/Matcher.html#find
Using Java Spliterator in SearchStreamGang

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults Result objects that match the # of times a phrase appears in an input string

```java
class PhraseMatchSpliterator implements Spliterator<Result> {
    ...
    boolean tryAdvance(Consumer<? super Result> action) {
        if (!mPhraseMatcher.find())
            return false;
        else {
            action.accept(new Result(mOffset + mPhraseMatcher.start()));
            return true;
        }
    }
    ...
}
```

If there is a match, then accept() keeps track of which index in the input string the match occurred

See docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html#accept
Using Java Spliterator in SearchStreamGang

- PhraseMatchSpliterator uses Java regex to create a stream of SearchResults Result objects that match the # of times a phrase appears in an input string.

```java
class PhraseMatchSpliterator implements Spliterator<SearchResults.Result> {
    ...
    public Spliterator<SearchResults.Result> trySplit() {
        ...
    }
    ...
}
```

We’ll analyze the `trySplit()` method when we discuss `SearchWithParallelStreams` (it’s not used for the sequential version)

See [docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html#trySplit](https://docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html#trySplit)
Pros of the SearchWith SequentialStreams Class
Pros of the SearchWithSequentialStreams Class

• There are several benefits with this sequential streams implementation

```java
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

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

    List<SearchResults> results = mPhrasesToFind
        .stream()  // Streams use “internal” iterators versus “external” iterators used by collections
        .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

```java
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

```java
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.
Pros of the SearchWithSequentialStreams Class

- There are several benefits with this sequential streams implementation

```java
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::isNotEmpty))
        .collect(toList());
    return results;
}
```

Focus on “what” operations to perform, rather than on “how” they’re implemented.
There are several benefits with this sequential streams implementation:

```java
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

- These lambda functions have no side-effects.
There are several benefits with this sequential streams implementation:

```java
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:
- 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

Tests conducted on a quad-core Lenovo P50 with 32 Gbytes of RAM
Cons of the SearchWithSequentialStreams Class

- This class only used a few Java 8 aggregate operations

```java
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 8 aggregate operations

```java
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;
```

However, these aggregate operations are also useful for parallel streams
### Cons of the `SearchWithSequentialStreams` Class

- *Many* other aggregate operations are part of the Java 8 stream API

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>allMatch(Predicate&lt;? 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;? 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 <code>&lt;T&gt; Stream.Builder&lt;T&gt;</code></td>
<td>builder()</td>
</tr>
<tr>
<td></td>
<td>Returns a builder for a <code>Stream</code>.</td>
</tr>
<tr>
<td><code>&lt;R,A&gt;</code> R</td>
<td>collect(Collectors&lt;? super T,A,R&gt; collector)</td>
</tr>
<tr>
<td></td>
<td>Performs a <em>mutable reduction</em> operation on the elements of this stream using a <code>Collector</code>.</td>
</tr>
<tr>
<td><code>&lt;R&gt;</code> R</td>
<td>collect(Supplier&lt;? extends R&gt; supplier, BiFunction&lt;? super T,R,R&gt; accumulator, BiConsumer&lt;? super R,R&gt; combiner)</td>
</tr>
<tr>
<td></td>
<td>Performs a <em>mutable reduction</em> operation on the elements of this stream.</td>
</tr>
<tr>
<td>static <code>&lt;T&gt; Stream&lt;T&gt;</code></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><code>Stream&lt;T&gt;</code></td>
<td>distinct()</td>
</tr>
<tr>
<td></td>
<td>Returns a stream consisting of the distinct elements (according to <code>Object.equals(Object)</code>) of this stream.</td>
</tr>
<tr>
<td>static <code>&lt;T&gt; Stream&lt;T&gt;</code></td>
<td>empty()</td>
</tr>
<tr>
<td></td>
<td>Returns an empty sequential <code>Stream</code>.</td>
</tr>
<tr>
<td><code>Stream&lt;T&gt;</code></td>
<td>filter(Predicate&lt;? 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 <code>Optional</code> describing some element of the stream, or an empty <code>Optional</code> if the stream is empty.</td>
</tr>
<tr>
<td>Optional&lt;T&gt;</td>
<td>findFirst()</td>
</tr>
<tr>
<td></td>
<td>Returns an <code>Optional</code> describing the first element of this stream, or an empty <code>Optional</code> if the stream is empty.</td>
</tr>
<tr>
<td><code>&lt;R&gt;</code> Stream&lt;R&gt;</td>
<td>flatMap(Function&lt;? super T,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 mapping function to each element.</td>
</tr>
</tbody>
</table>

See [docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html](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 8 stream API
- We’ll cover more of them later

See “Java 8 Parallel ImageStreamGang Example”
End of Java 8 Sequential SearchStreamGang Example (Part 3)