Evaluating the Java Sequential SearchStreamGang Case Study

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 livelessons/streamgangs/SearchWithSequentialStreams.java
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()
        .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;
}
```
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 code is declarative since it is a pipeline 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

```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;
}
```

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 behaviors have no side-effects.
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::isEmtpy))
        .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

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!

See upcoming lessons on “Java Parallel Streams”
Cons of the SearchWithSequentialStreams Class

- This class only used a few Java 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 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 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 &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(Collector&lt;? 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,? super T&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;? 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 mapping function to each element.</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)
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 ImageStreamStreamGang Example”
End of Evaluating the Java Sequential SearchStreamGang Case Study