Java 8 Sequential SearchStreamGang
Example (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 to apply sequential streams to the SearchStreamGang program

See [github.com/douglascraigschmidt/LiveLessons/tree/master/SearchStreamGang](github.com/douglascraigschmidt/LiveLessons/tree/master/SearchStreamGang)
Learning Objectives in this Part of the Lesson

• Know how to apply sequential streams to the SearchStreamGang program
• Understand the SearchStreamGang processStream() & processInput() methods
Learning Objectives in this Part of the Lesson

• Know how to apply sequential streams to the SearchStreamGang program

• Understand the SearchStreamGang processStream() & processInput() methods

• This program is more interesting than the SimpleSearchStream program
Overview of SearchStreamGang
Overview of SearchStreamGang

SearchStreamGang is a Java 8 revision of SearchTaskGang.
Overview of SearchStreamGang

• SearchStreamGang is a Java 8 revision of SearchTaskGang
• SearchTaskGang showcases the Java executor framework for tasks that are “embarrassingly parallel”

![Diagram of executor framework with Search Words: "do", "re", "mi", "fa", "so", "la", "ti", "do" and Input Strings to Search]
Overview of SearchStreamGang

- SearchStreamGang is a more powerful revision of SimpleSearchStreamStream

Input Strings to Search

- Input Strings to Search

Search Phrases

- Search Phrases

Search Words

- Search Words

map(phrase -> searchForPhrase(…))

filter(not(SearchResults::isEmpty))

collect(toList())

map(word -> searchForWord(…))

filter(not(SearchResults::isEmpty))

collect(toList())

See [github.com/douglascraigschmidt/LiveLessons/tree/master/SimpleSearchStreamStream](github.com/douglascraigschmidt/LiveLessons/tree/master/SimpleSearchStreamStream)
Overview of SearchStreamGang

- SearchStreamGang is a more powerful revision of SimpleSearchStream, e.g.
  - It uses regular expressions to find phrases in works of Shakespeare

The Complete Works of William Shakespeare

Welcome to the Web’s first edition of the Complete Works of William Shakespeare. This site has offered Shakespeare’s plays and poetry to the Internet community since 1993.

For other Shakespeare resources, visit the Mr. William Shakespeare and the Internet Web site.

The original electronic source for this server was the Complete Moby(tm) Shakespeare. The HTML versions of the plays provided here are placed in the public domain.

Older news items

See shakespeare.mit.edu
Overview of SearchStreamGang

• SearchStreamGang is a more powerful revision of SimpleSearchStream, e.g.
• It uses regular expressions to find phrases in works of Shakespeare

“...
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”
Overview of SearchStreamGang

• SearchStreamGang is a more powerful revision of SimpleSearchStream, e.g.
  • It uses regular expressions to find phrases in works of Shakespeare

  “...
  What's in a name? That which we call a rose
  By any other name would smell as sweet.
  So Romeo would, were he not Romeo call'd,
  Retain that dear perfection which he owes
  Without that title. ...”

  “What’s in a name? That which we call a rose
  By any other name would smell as sweet.”

The phrases can also match across multiple lines
Overview of SearchStreamGang

- SearchStreamGang is a more powerful revision of SimpleSearchStream, e.g.
  - It uses regular expressions to find phrases in works of Shakespeare
  - It defines a framework for Java 8 concurrency & parallelism strategies
  
  e.g., parallel streams, parallel spliterator, & completable futures

- Example classes:
  - SearchWithSequentialStream
  - SearchWithSequentialLoops
  - SearchWithParallelStreams
  - SearchWithParallelSpliterator
  - SearchWithCompletetableFuturesInputs
  - SearchWithCompletetableFuturesPhrases
  - SearchWithParallelStreamInputs
  - SearchWithParallelStreamPhrases
Overview of SearchStreamGang

- SearchStreamGang is a more powerful revision of SimpleSearchStream, e.g.
  - It uses regular expressions to find phrases in works of Shakespeare
  - It defines a framework for Java 8 concurrency & parallelism strategies

This framework enables “apples-to-apples” performance comparisons
Overview of SearchStreamGang

- SearchStreamGang is a more powerful revision of SimpleSearchStream, e.g.
  - It uses regular expressions to find phrases in works of Shakespeare
  - It defines a framework for Java 8 concurrency & parallelism strategies

Starting SearchStreamGangTest
PARALLEL_SPLITTERATOR 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

We’ll cover Java 8 concurrency/parallel strategies after covering sequential streams
Applying Sequential Streams to SearchStreamGang
Applying Sequential Streams to SearchStreamGang

- We show aggregate operations in the SearchStreamGang’s processStream() & processInput() methods

Applying Sequential Streams to SearchStreamGang

- We show aggregate operations in the SearchStreamGang’s `processStream()` & `processInput()` methods

```java
getInput()
  .stream()
  .map(this::processInput)
  .collect(toList());

return mPhrasesToFind
  .stream()
  .map(phrase -> searchForPhrase(phrase, input, title, false))
  .filter(not(SearchResults::isEmpty))
  .collect(toList());
```

See [SearchStreamGang/src/main/java/livelessons/streamgangs/SearchWithSequentialStreams.java](SearchStreamGang/src/main/java/livelessons/streamgangs/SearchWithSequentialStreams.java)
Applying Sequential Streams to SearchStreamGang

- We show aggregate operations in the SearchStreamGang’s `processStream()` & `processInput()` methods

```java
return mPhrasesToFind
    .stream()
    .map(phrase -> searchForPhrase(phrase, input, title, false))
    .filter(not(SearchResults::isEmpty))
    .collect(toList());
```

i.e., the `map()`, `filter()`, & `collect()` aggregate operations
Applying Sequential Streams to SearchStreamGang

• We show aggregate operations in the SearchStreamGang’s processStream() & processInput() methods

• **processStream()**
  • Uses a sequential stream to search a list of input strings in one thread

Each input string corresponds to a different work of Shakespeare
Applying Sequential Streams to SearchStreamGang

- We show aggregate operations in the SearchStreamGang’s `processStream()` & `processInput()` methods
  - `processStream()`
  - `processInput()`
    - Uses a sequential stream to search an input string & locate all the occurrences of phases in one thread

```
stream()
map(phrase -> searchForPhrase(...))
filter(not(SearchResults::isEmpty))
collect(toList())
```
Visualizing processStream() & processInput()
Visualizing `processStream()` & `processInput()`

- `processStream()` searches a list of input strings

**Input Strings to Search**

**Input a list of input strings**

List `<String>`

`stream()`
Visualizing `processStream()` & `processInput()`

- `processStream()` searches a list of input strings

```
List <String>
```

Convert collection to a (sequential) stream
Visualizing `processStream()` & `processInput()`

- `processStream()` searches a list of input strings

```
List <String>… 
```

```
Stream <String> 
```

```
Output a stream of input strings
```

```
Input Strings to Search
```

```
stream()
```

---

24
Visualizing `processStream()` & `processInput()`

- `processStream()` searches a list of input strings

```
List <String>  

Stream <String>

input a stream of input strings
```

```java
List<String> stream();
Stream<String> map(this::processInput);
```
Visualizing `processStream()` & `processInput()`

- `processStream()` searches a list of input strings

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

Call `processInput()` to search for phrases in each input string
Visualizing `processStream()` & `processInput()`

- `processStream()` searches a list of input strings

Output a stream of lists of search results

List `<String>`

Stream `<String>`

Stream `<List <SearchResults>>`

map() also transforms its input stream type into a different output stream type

map(this::processInput)
processStream() searches a list of input strings

Visualizing processStream() & processInput()

Input a stream of lists of search results

List <String>

Stream <String>

Stream<List <SearchResults>>

Input Strings to Search

stream()  

map(this::processInput)  

collect(toList())
Visualizing `processStream()` & `processInput()`

- `processStream()` searches a list of input strings

```
List <String>

Stream <String>

Stream<List <SearchResults>>
```

```
\[
\text{Input Strings to Search}
\]
```


collect(toList())

```
stream()
```

```
map(this::processInput)
```

Trigger intermediate operation processing
Visualizing `processStream()` & `processInput()`

- `processStream()` searches a list of input strings

```
List<String> ...

Stream<String> ...

Stream<List<SearchResults>> ...

Stream<List<SearchResults>>
```

Input Strings to Search

Return a list of lists of search results based on “encounter order”
Visualizing `processStream()` & `processInput()`

- `processInput()` finds phrases in an input string

List `<String>`

**Input a list of phrases to find**

Search Phrases

`stream()`
Visualizing `processStream()` & `processInput()`

- `processInput()` finds phrases in an input string

```
List <String>
```

Convert collection to a (sequential) stream
Visualizing `processStream()` & `processInput()`

- `processInput()` finds phrases in an input string

Output a stream of phrases to find

```
List <String>
```

```
Stream <String>
```

```
Search Phrases
```

```
Hamlet
```

```
stream()
```

```
45,000+ phrases
```

```
Search Phrases
```

```
Hamlet
```

```
List <String>
```

```
Stream <String>
```

```
Output a stream of phrases to find
```

```
Search Phrases
```

```
Hamlet
```

```
stream()
```

```
45,000+ phrases
```

```
Search Phrases
```

```
Hamlet
```
Visualizing `processStream()` & `processInput()`

- `processInput()` finds phrases in an input string

**List**

`<String>`

**Stream**

`<String>`

---

**Input a stream of phrases to find**

`List <String>`

`Stream <String>`

---

**Search Phrases**

`map(phrase -> searchForPhrase(...))`
Visualizing processStream() & processInput()

• processInput() finds phrases in an input string

Search Phrases

List <String>

Stream <String>

map(phrase -> searchForPhrase(…))

Search for the phrase in each input string
Visualizing `processStream()` & `processInput()`

- `processInput()` finds phrases in an input string

```
List <String>

Stream <String>

Stream <SearchResults>
```

Output a stream of search results

```
searchForPhrase(...)
```
• processInput() finds phrases in an input string

Visualizing processStream() & processInput()

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

Input a stream of search results

Search Phrases

Hamlet

stream()

map(phrase -> searchForPhrase(…))

filter(not(SearchResults::isEmpty))
Visualizing `processStream()` & `processInput()`

- `processInput()` finds phrases in an input string

List `<String>`

Stream `<String>`

Stream `<SearchResults>`

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

Remove empty search results from the stream
processInput() finds phrases in an input string

Output a stream of non-empty search results

List <String>

Stream <String>

Stream <SearchResults>

Stream <SearchResults>

Search Phrases

stream()

map(phrase -> searchForPhrase(...))

filter(not(SearchResults::isEmpty))
- processInput() finds phrases in an input string

**Input a stream of non-empty search results**

```
List <String>
```

```
Stream <String>
```

```
Stream <SearchResults>
```

```
Stream <SearchResults>
```

```
map(phrase -> searchForPhrase(…))
```

```
filter(not(SearchResults::isEmpty))
```

```
collect(toList())
```
Visualizing `processStream()` & `processInput()`

- `processInput()` finds phrases in an input string

```
List <String>  // Initial list of phrases
-> stream()   // Trigger intermediate operation processing

Stream <String>  // List converted to stream
-> map(phrase -> searchForPhrase(...))

Stream <SearchResults>  // Stream of search results
-> filter(not(SearchResults::isEmpty))

Stream <SearchResults>  // Final stream of search results
-> collect(toList())
```

Search Phrases

---

Trigger intermediate operation processing
Visualizing `processStream()` & `processInput()`

- `processInput()` finds phrases in an input string

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

- Return a list of search results in encounter order

```
stream()
map(phrase -> searchForPhrase(...))
filter(not(SearchResults::isEmpty))
collect(toList())
```
Visualizing `processStream()` & `processInput()`

- We focus on sequential streams with one thread

![Diagram](image)
Visualizing `processStream()` & `processInput()`

- We focus on sequential streams with one thread
- We’ll cover parallel streams later

See "Overview of Java 8 Parallel Streams"
Visualizing `processStream()` & `processInput()`

- We focus on sequential streams with one thread
- We’ll cover parallel streams later

Minuscule changes are needed to transition from sequential to parallel streams!
Implementing processStream() as a Sequential Stream
Implementing `processStream()` as a Sequential Stream

- `processStream()` sequentially searches for phrases in lists of input “strings”

```
Stream<List<CharSequence>>

List <CharSequence>

List <List<SearchResults>>

Stream <List<SearchResults>>
```

```
Input Strings to Search

stream()

map(this::processInput)

collect(toList())
```
Implementing `processStream()` as a Sequential Stream

- `processStream()` sequentially searches for phrases in lists of input “strings”

```java
protected List<List<SearchResults>> processStream() {
    List<CharSequence> inputList = getInput();

    return inputList
        .stream()
        .map(this::processInput)
        .collect(toList());
}
```
Implementing processStream() as a Sequential Stream

- processStream() sequentially searches for phrases in lists of input “strings”

protected List<List<SearchResults>> processStream() {
    List<CharSequence> inputList = getInput();

    return inputList.stream()
        .map(this::processInput)
        .collect(toList());
}
Implementing `processStream()` as a Sequential Stream

- `processStream()` sequentially searches for phrases in lists of input “strings”

```java
protected List<List<SearchResults>> processStream() {
    List<CharSequence> inputList = getInput();

    return inputList
        .stream()
        .map(this::processInput)
        .collect(toList());
}
```

The `getInput()` method is defined in the StreamGang framework.
Implementing `processStream()` as a Sequential Stream

- `processStream()` sequentially searches for phrases in lists of input “strings”

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

`CharSequence` optimizes `subSequence()` to avoid memory copies (cf. `String substring()`)

See [www.javaspecialists.eu/archive/Issue230.html](http://www.javaspecialists.eu/archive/Issue230.html)
Implementing `processStream()` as a Sequential Stream

- `processStream()` sequentially searches for phrases in lists of input “strings”

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

Returns a list of lists of search results denoting how many times a search phrase appeared in each input string.
Implementing `processStream()` as a Sequential Stream

- `processStream()` sequentially searches for phrases in lists of input “strings”

```java
protected List<List<SearchResults>> processStream() {
    List<CharSequence> inputList = getInput();

    return inputList.stream()
        .map(this::processInput)
        .collect(toList());
}
```

We'll later show how `flatMap()` “flattens” `List<List<SearchResults>>` into a stream of `SearchResults`

See “Java 8 Sequential SearchStreamGang Example (Part 2)”
protected List<List<SearchResults>> processStream() {
    List<CharSequence> inputList = getInput();

    return inputList
            .stream()
            .map(this::processInput)
            .collect(toList());
}
Implementing processStream() as a Sequential Stream

- processStream() sequentially searches for phrases in lists of input “strings”

```java
protected List<List<SearchResults>> processStream() {
    List<CharSequence> inputList = getInput();

    return inputList
        .stream()
        .map(this::processInput)
        .collect(toList());
}
```

*processStream() is implemented via a sequential stream pipeline*
Implementing `processStream()` as a Sequential Stream

- `processStream()` sequentially searches for phrases in lists of input “strings”

```java
protected List<List<SearchResults>> processStream() {
    List<CharSequence> inputList = getInput();

    return inputList.stream()
        .map(this::processInput)
        .collect(toList());
}
```

This factory method converts the input list into a stream

The `stream()` factory method uses `StreamSupport.stream(spliterator(), false)`
protected List<List<SearchResults>> processStream() {
    List<CharSequence> inputList = getInput();
    return inputList.stream()
        .map(this::processInput)
        .collect(toList());
}

Implementing processStream() as a Sequential Stream

• processStream() sequentially searches for phrases in lists of input “strings”

Applying processInput() method reference to each input in the stream
Implementing `processStream()` as a Sequential Stream

- `processStream()` sequentially searches for phrases in lists of input “strings”

```java
protected List<List<SearchResults>> processStream() {
    List<CharSequence> inputList = getInput();

    return inputList.stream()
        .map(this::processInput)
        .collect(toList());
}
```

`processInput()` returns a list of `SearchResults`—one list for each input string
Implementing `processStream()` as a Sequential Stream

- `processStream()` sequentially searches for phrases in lists of input “strings”

```java
protected List<List<SearchResults>> processStream() {
    List<CharSequence> inputList = getInput();

    return inputList
        .stream()
        .map(this::processInput)
        .collect(toList());
}
```

This terminal operation triggers intermediate operation processing
Implementing processStream() as a Sequential Stream

- processStream() sequentially searches for phrases in lists of input “strings”

```java
protected List<List<SearchResults>> processStream() {
    List<CharSequence> inputList = getInput();

    return inputList
        .stream()
        .map(this::processInput)
        .collect(toList());
}
```

Yields a list (of lists) of search results
Implementing processStream() as a Sequential Stream

- processStream() sequentially searches for phrases in lists of input “strings”

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

Returns a list of lists of search results denoting how many times a search phrase appeared in each input string
Implementing `processInput()` as a Sequential Stream
Implementing `processInput()` as a Sequential Stream

- `processInput()` searches an input string for all occurrences of phrases to find

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

List <SearchResults>

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

45,000+ phrases

Search Phrases

Hamlet
Implementing `processInput()` as a Sequential Stream

- `processInput()` searches an input string for all occurrences of phrases to find.

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

    List<SearchResults> results = mPhrasesToFind
        .stream()
        .map(phrase -> searchForPhrase(phrase, input, title, false))
        .filter(not(SearchResults::isEmpty))
        .collect(toList());
    return results;
}
```
Implementing `processInput()` as a Sequential Stream

- `processInput()` searches an input string for all occurrences of phrases to find.

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

    List<SearchResults> results = mPhrasesToFind
        .stream()
        .map(phrase
             -> searchForPhrase(phrase, input,
                               .filter(not(SearchResults::isEmpty)))
             .collect(toList()));
    return results;
}
```

The input is a section of a text file managed by the test driver program.
Implementing `processInput()` as a Sequential Stream

- `processInput()` searches an input string for all occurrences of phrases to find:

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

    List<SearchResults> results = mPhrasesToFind
        .stream()
        .map(phrase -> searchForPhrase(phrase, input, title, false))
        .filter(not(SearchResults::isEmpty))
        .collect(toList());

    return results;
}
```

The input string is split into two parts.
Implementing `processInput()` as a Sequential Stream

- `processInput()` searches an input string for all occurrences of phrases to find.

```java
List<SearchResults> processInput(CharSequence inputSeq) {
    String title = getTitle(inputSeq);
    CharSequence input = inputSeq.subSequence(...);
    List<SearchResults> results = mPhrasesToFind.stream()
        .map(phrase -> searchForPhrase(phrase, input, title, false))
        .filter(not(SearchResults::isEmpty))
        .collect(toList());
    return results;
}
```

`subSequence()` is used to avoid memory copying overhead for substrings.

See `SearchStreamGang/src/main/java/livelessons/utils/SharedString.java`
Implementing processInput() as a Sequential Stream

- processInput() searches an input string for all occurrences of phrases to find.

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

    List<SearchResults> results = mPhrasesToFind.stream()
        .map(phrase -> searchForPhrase(phrase, input, title, false))
        .filter(not(SearchResults::isEmpty))
        .collect(toList());
    return results;
}
```

Convert a list of phrases into a stream
Implementing processInput() as a Sequential Stream

- processInput() searches an input string for all occurrences of phrases to find

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

    List<SearchResults> results = mPhrasesToFind.stream()
                                      .map(phrase -> searchForPhrase(phrase, input, title, false))
                                      .filter(not(SearchResults::isEmpty))
                                      .collect(toList());

    return results;
}
```

Part 2 of this lesson shows the implementation of this method
Implementing `processInput()` as a Sequential Stream

- `processInput()` searches an input string for all occurrences of phrases to find.

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

    List<SearchResults> results = mPhrasesToFind.stream()
        .map(phrase -> searchForPhrase(phrase, input, title, false))
        .filter(not(SearchResults::isEmpty))
        .collect(toList());
    return results;
}
```

Returns output stream containing non-empty `SearchResults` from input stream.
Implementing `processInput()` as a Sequential Stream

- `processInput()` searches an input string for all occurrences of phrases to find

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

    List<SearchResults> results = mPhrasesToFind
        .stream()
        .map(phrase
              -> searchForPhrase(phrase, input, title, false))
        .filter(not(SearchResults::isEmpty))
        .collect(toList());

    return results;
}
```

*Note use of a method reference & a negator predicate lambda*

See [SearchStreamGang/src/main/java/livelessons/utils/StreamsUtils.java](SearchStreamGang/src/main/java/livelessons/utils/StreamsUtils.java)
Implementing `processInput()` as a Sequential Stream

- `processInput()` searches an input string for all occurrences of phrases to find:

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

    List<SearchResults> results = mPhrasesToFind
        .stream()
        .map(phrase -> searchForPhrase(phrase, input, title, false))
        .filter(result -> result.size() > 0)
        .collect(toList());
    return results;
}
```

Another approach using a lambda expression
Implementing processInput() as a Sequential Stream

- processInput() searches an input string for all occurrences of phrases to find

```java
List<SearchResults> processInput(CharSequence inputSeq) {
    String title = getTitle(inputSeq);
    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 are both intermediate operations
Implementing `processInput()` as a Sequential Stream

- `processInput()` searches an input string for all occurrences of phrases to find.

```
List<SearchResults> processInput(CharSequence inputSeq) {
    String title = getTitle(inputSeq);
    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 terminal operation triggers intermediate operation processing & yields a list result.
Implementing `processInput()` as a Sequential Stream

- `processInput()` searches an input string for all occurrences of phrases to find

```java
List<SearchResults> processInput(CharSequence inputSeq) {
    String title = getTitle(inputSeq);
    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 terminal operation triggers intermediate operation processing & yields a list result
Implementing `processInput()` as a Sequential Stream

- `processInput()` searches an input string for all occurrences of phrases to find

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

    List<SearchResults> results = mPhrasesToFind
        .stream()
        .map(phrase
             -> searchForPhrase(phrase, input, title, false))
        .filter(not(SearchResults::isEmpty))
        .collect(toList());
    return results;
}
```

The list result is returned back to the `map()` operation in `processStream()`
End of Java 8 Sequential SearchStreamGang Example (Part 1)