Java Sequential SearchStreamGang

Example: Implementing Hook Methods

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
- Understand the SearchStreamGang process Stream() & processInput() hook methods
Implementing `processStream()` as a Sequential Stream
Implementing `processStream()` as a Sequential Stream

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

```
List<String> inputStrings = ...;
Stream<List<SearchResults>> stream = inputStrings.stream();
```

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

**Complete works of Shakespeare**

- `inputStrings`:
  - Stream of CharSequences
  - List of CharSequences

- `stream`:
  - Stream of Lists of SearchResults
  - List of Lists of SearchResults

- `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`
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());
}
```

Stores # of times a phrase appeared in an input string

See `livelessons/utils/SearchResults.java`
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*

`stream()` uses `StreamSupport.stream(spliterator(),false)`
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());
}

The `processInput()` method reference is applied 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

Collect() allocates memory for results, which is less error-prone than OO version!
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>
Stream <SearchResults>
List <SearchResults>
```

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

Search Phrases
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;
}
```
• 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 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

```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 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;
}
```

Apply this function lambda to all phrases in input stream & return an output stream of SearchResults

See upcoming lesson on "Java Sequential SearchStreamGang Example: Applying Spliterator"
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;
}
```

This approach uses a method reference along with a negator predicate lambda

See [SearchStreamGang/src/main/java/livelessons/utils/StreamsUtils.java](https://example.com/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(((Predicate<String>) SearchResults::isEmpty).negate())
        .collect(toList());
    return results;
}
```

Another approach uses a composed predicate

See [docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html#negate](https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html#negate)
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;
  }
  ```

Yet another approach uses 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.

There are no control constructs in this code, which makes it easier to read!
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

Again, `collect()` allocates memory, which is less error-prone than OO version!
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 Sequential SearchStreamGang Example: Implementing Hook Methods