Overview of Java 8 Streams (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

• Understand the structure & functionality of Java 8 streams

Input $x$

Aggregate operation (Function $f$)

Output $f(x)$

Aggregate operation (Function $g$)

Output $g(f(x))$

Aggregate operation (Function $h$)

Output $h(g(f(x)))$
Learning Objectives in this Part of the Lesson

• Understand the structure & functionality of Java 8 streams, e.g.,
  • Fundamentals of streams

\[\text{Input } x\]

Aggregate operation (Function \(f\))

\[\text{Output } f(x)\]

Aggregate operation (Function \(g\))

\[\text{Output } g(f(x))\]

Aggregate operation (Function \(h\))

\[\text{Output } h(g(f(x)))\]
Overview of Java 8 Streams
Overview of Java 8 Streams

- A stream is an ordered pipeline of aggregate operations that process a (conceptually unbounded) sequence of elements

Input x

Aggregate operation (Function f)

Output f(x)

Aggregate operation (Function g)

Output g(f(x))

Aggregate operation (Function h)

Output h(g(f(x)))

Examples of aggregate operations include filter(), map(), forEach(), & collect()
Overview of Java 8 Streams

• A stream is an ordered pipeline of aggregate operations that process a (conceptually unbounded) sequence of elements
  • An aggregate operation performs a function

A function is implemented by a lambda expression or method reference
Overview of Java 8 Streams

- A stream is an ordered pipeline of aggregate operations that process a (conceptually unbounded) sequence of elements
- An aggregate operation performs a function
  - Ideally, a function’s output in a stream depends only on its input arguments

See en.wikipedia.org/wiki/Purely_functional_programming
Overview of Java 8 Streams

- A stream is an ordered pipeline of aggregate operations that process a (conceptually unbounded) sequence of elements
- An aggregate operation performs a function
- A stream holds no non-transient storage

```
Input x

Aggregate operation (Function f)

Output f(x)

Aggregate operation (Function g)

Output g(f(x))

Aggregate operation (Function h)

Output h(g(f(x)))
```
Overview of Java 8 Streams

• A stream is an ordered pipeline of aggregate operations that process a (conceptually unbounded) sequence of elements
  • An aggregate operation performs a function
  • A stream holds no non-transient storage
  • Every stream works essentially the same

![Diagram of Java 8 Streams]

Input x

Aggregate operation (Function f)

Output f(x)

Aggregate operation (Function g)

Output g(f(x))

Aggregate operation (Function h)

Output h(g(f(x)))
Overview of Java 8 Streams

• A stream is an ordered pipeline of aggregate operations that process a (conceptually unbounded) sequence of elements
  • An aggregate operation performs a function
  • A stream holds no non-transient storage
  • Every stream works essentially the same, i.e.,
    • Starts with a source of data

**Diagram:**

1. Input $x$
2. Aggregate operation (Function $f$) → Output $f(x)$
3. Aggregate operation (Function $g$) → Output $g(f(x))$
4. Aggregate operation (Function $h$) → Output $h(g(f(x)))$
Overview of Java 8 Streams

- A stream is an ordered pipeline of aggregate operations that process a (conceptually unbounded) sequence of elements
  - An aggregate operation performs a function
  - A stream holds no non-transient storage
  - Every stream works essentially the same, i.e.,
    - Starts with a source of data
    -Processes the data through a pipeline of intermediate operations
A stream is an ordered pipeline of aggregate operations that process a (conceptually unbounded) sequence of elements.

- An aggregate operation performs a function.
- A stream holds no non-transient storage.
- Every stream works essentially the same, i.e.,
  - Starts with a source of data.
  - Processes the data through a pipeline of intermediate operations.
  - Finishes with a terminal operation.

A terminal operation triggers processing of aggregate operations in a stream.
Overview of Java 8 Streams

- A stream is an ordered pipeline of aggregate operations that process a (conceptually unbounded) sequence of elements
  - An aggregate operation performs a function
  - A stream holds no non-transient storage
  - Every stream works essentially the same
  - By default, each aggregate operation in a stream runs its function sequentially

See radar.oreilly.com/2015/02/java-8-streams-api-and-parallelism.html
A stream is an ordered pipeline of aggregate operations that process a (conceptually unbounded) sequence of elements.

- An aggregate operation performs a function.
- A stream holds no non-transient storage.
- Every stream works essentially the same.
- By default, each aggregate operation in a stream runs its function sequentially.
  - i.e., one after another in the caller’s thread of control.

See radar.oreilly.com/2015/02/java-8-streams-api-and-parallelism.html
Overview of Java 8 Streams

- A stream is an ordered pipeline of aggregate operations that process a (conceptually unbounded) sequence of elements
  - An aggregate operation performs a function
  - A stream holds no non-transient storage
  - Every stream works essentially the same
  - By default, each aggregate operation in a stream runs its function sequentially
- We first cover sequential streams

See [docs.oracle.com/javase/tutorial/collections/streams](docs.oracle.com/javase/tutorial/collections/streams)
Overview of Java 8 Streams

• A stream is an ordered pipeline of aggregate operations that process a (conceptually unbounded) sequence of elements
  • An aggregate operation performs a function
  • A stream holds no non-transient storage
  • Every stream works essentially the same
  • By default, each aggregate operation in a stream runs its function sequentially
  • We first cover sequential streams
    • We cover parallel streams later
Overview of Java 8 Streams

- A Java 8 stream is an implementation of the POSA1 **Pipes & Filters** pattern

Divide an app’s tasks into multiple self-contained data processing steps & connect these steps via intermediate data buffers to form a data processing pipeline

Overview of Java 8 Streams

- Other common implementations of *Pipes & Filters*
Overview of Java 8 Streams

• Other common implementations of *Pipes & Filters*
  • A pipeline in UNIX shells

See en.wikipedia.org/wiki/Pipeline_(Unix)
Overview of Java 8 Streams

- Other common implementations of *Pipes & Filters*
  - A pipeline in UNIX shells
  - System V STREAMS

See [en.wikipedia.org/wiki/STREAMS](en.wikipedia.org/wiki/STREAMS)
End of Overview of Java
8 Streams (Part 1)
Overview of Java 8 Streams (Part 2)

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

- Understand the structure & functionality of Java 8 streams, e.g.,
  - Fundamentals of streams
  - Common stream aggregate operations
Learning Objectives in this Part of the Lesson

- Understand the structure & functionality of Java 8 streams, e.g.,
  - Fundamentals of streams
  - Common stream aggregate operations

These features are applicable to both sequential & parallel streams.
Overview of Common Stream Aggregate Operations
Overview of Common Stream Aggregate Operations

- An aggregate operation processes all elements in a stream

  - Input $x$
  - Aggregate operation (Function $f$)
    - Output $f(x)$
    - Aggregate operation (Function $g$)
      - Output $g(f(x))$
      - Aggregate operation (Function $h$)
        - Output $h(g(f(x)))$
Overview of Common Stream Aggregate Operations

- An aggregate operation processes all elements in a stream

1. **Input** $x$
2. **Stream map** `(Function<> mapper)`
3. **Output** $f(x)$
4. **Stream filter** `(Predicate<> pred)`
5. **Output** $g(f(x))$
6. **R collect** `(Collector<> collector)`
7. **Output** $h(g(f(x)))$
Overview of Common Stream Aggregate Operations

- An aggregate operation processes all elements in a stream

\[ \text{Input } x \]

\[ \text{Stream } \text{map}(\text{Function} \langle \ldots \rangle \text{ mapper}) \]

\[ \text{Output } f(x) \]

\[ \text{Stream } \text{filter}(\text{Predicate} \langle \ldots \rangle \text{ pred}) \]

\[ \text{Output } g(f(x)) \]

\[ R \text{ collect}(\text{Collector} \langle \ldots \rangle \text{ collector}) \]

\[ \text{Output } h(g(f(x))) \]

See docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#map-java.util.function.Function-
Overview of Common Stream Aggregate Operations

• An aggregate operation processes all elements in a stream

Input x

Stream \text{map}(\text{Function}\langle\ldots\rangle \ \text{mapper})

Output f(x)

Stream \text{filter}(\text{Predicate}\langle\ldots\rangle \ \text{pred})

Output g(f(x))

R \text{collect}(\text{Collector}\langle\ldots\rangle \ \text{collector})

Output h(g(f(x)))

The number of output stream elements matches the number of input stream elements

Applies the given function to the elements of the input stream & returns an output stream consisting of the results
Overview of Common Stream Aggregate Operations

- An aggregate operation processes all elements in a stream

Tests the given predicate against each element of the input stream & returns an output stream consisting only of the elements that match the predicate

See docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#filter-java.util.function.Predicate-
Overview of Common Stream Aggregate Operations

- An aggregate operation processes all elements in a stream

Tests the given predicate against each element of the input stream & returns an output stream consisting only of the elements that match the predicate.

The number of output stream elements may be less than the number of input stream elements.
Overview of Common Stream Aggregate Operations

- An aggregate operation processes all elements in a stream

\[
\text{Input } x \\
\downarrow \\
\text{Stream map}(\text{Function}<\ldots> \text{mapper}) \\
\downarrow \\
\text{Output } f(x) \\
\downarrow \\
\text{Stream filter}(\text{Predicate}<\ldots> \text{pred}) \\
\downarrow \\
\text{Output } g(f(x)) \\
\downarrow \\
R \text{ collect}(\text{Collector}<\ldots> \text{collector}) \\
\downarrow \\
\text{Output } h(g(f(x)))
\]

*map() & filter() are intermediate operations that process elements in their input stream & yield an output stream*

Intermediate operations are “lazy” & don’t run until a terminal operator is reached.
Overview of Common Stream Aggregate Operations

- An aggregate operation processes all elements in a stream

```
Input x
Stream map(Function<…> mapper)
Stream filter(Predicate<…> pred)
R collect(Collector<…> collector)
```

A terminal operation that uses a Collector to perform a reduction on the elements of its input stream & returns the results of the reduction.

See docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#collect-java.util.stream.Collector-
Overview of Common Stream Aggregate Operations

- An aggregate operation processes all elements in a stream

```
Input x
Stream map(Function<…> mapper)
Output f(x)
Stream filter(Predicate<…> pred)
Output g(f(x))
R collect(Collector<…> collector)
Output h(g(f(x)))
```

`collect()` also triggers the processing of other aggregate operations in the stream

Each stream **must** have one (& only one) terminal operation
End of Overview of Java
8 Streams (Part 2)
Learning Objectives in this Part of the Lesson

- Understand the structure & functionality of Java 8 streams
- Know how to apply sequential streams to a variant of the SearchStreamGang program

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

- Understand the structure & functionality of Java 8 streams
- Know how to apply sequential streams to a variant of the SearchStreamGang program
- This program is a Java 8 rewrite of the SearchTaskGang program

See github.com/douglascraigschmidt/LiveLessons/tree/master/SearchTaskGang
Applying Sequential Streams to SearchStreamGang (Part 1)
Applying Sequential Streams to SearchStreamGang

- We'll explain aggregate operations in the context of the SearchStreamGang

See github.com/douglascraigschmidt/LiveLessons/tree/master/SearchStreamGang
Applying Sequential Streams to SearchStreamGang

- We’ll explain aggregate operations in the context of the SearchStreamGang
- Finds words in the list of input strings

Input Strings to Search
- "do", "re", "mi", "fa", "so", "la", "ti", "do"

Search Words

Printing results for input file 1 from fastest to slowest
- PARALLEL_STREAM_INPUTS executed in 943 msecs
- COMPLETABLE_FUTURES_INPUTS executed in 1035 msecs
- hardCodedParallelStreamsSolution executed in 1239 msecs
- PARALLEL_STREAMS executed in 2020 msecs
- PARALLEL_STREAM_WORDS executed in 3173 msecs
- SEQUENTIAL_STREAM executed in 3658 msecs
- hardCodedSequentialSolution executed in 3798 msecs
- COMPLETABLE_FUTURES_WORDS executed in 3990 msecs

Printing results for input file 2 from fastest to slowest
- COMPLETABLE_FUTURES_INPUTS executed in 518 msecs
- PARALLEL_STREAM_INPUTS executed in 575 msecs
- PARALLEL_STREAMS executed in 650 msecs
- COMPLETABLE_FUTURES_WORDS executed in 670 msecs

map(word -> searchForWord(...))

filter(result -> result.size() > 0)

collect(toList())
Applying Sequential Streams to SearchStreamGang

- We’ll explain aggregate operations in the context of the SearchStreamGang
- Finds words in the list of input strings
- We’ll examine the processStream() & processInput() methods

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

```java
List<SearchResults> processInput(String inputString) {
    String title = getTitle(inputString);
    String input = inputString.substring(title.length());
    List<SearchResults> results = mWordsToFind.stream()
        .map(word -> searchForWord(word, input, title))
        .filter(result -> result.size() > 0)
        .collect(toList());
    return results;
}
```

See SearchStreamGang/src/main/java/livelessons/streamgangs/SearchWithSequentialStream.java
We’ll explain aggregate operations in the context of the SearchStreamGang.

- Finds words in the list of input strings
- We’ll examine the processStream() & processInput() methods

```java
List<SearchResults> processInput(String inputString) {
    String title = getTitle(inputString);
    String input = inputString.substring(title.length());
    List<SearchResults> results = mWordsToFind
        .stream()
        .map(word -> searchForWord(word, input, title))
        .filter(result -> result.size() > 0)
        .collect(toList());
    return results;
}
```

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

Focus is on the map(), filter(), & collect() aggregate operations.
We’ll explain aggregate operations in the context of the SearchStreamGang

Finds words in the list of input strings

We’ll examine the processStream() & processInput() methods

The emphasis is on aggregate operations for sequential streams
Applying Sequential Streams to SearchStreamGang

- We’ll explain aggregate operations in the context of the SearchStreamGang
  - Finds words in the list of input strings
  - We’ll examine the processStream() & processInput() methods
- The emphasis is on aggregate operations for sequential streams
  - Parallel streams are covered later in this lesson

Input Strings to Search

"do", "re", "mi", "fa", "so", "la", "ti", "do"

Search Words

map(word -> searchForWord(…))

filter(result -> result.size() > 0)

collect(toList())
Applying Sequential Streams to SearchStreamGang

- processStream() sequentially searches for words in lists of input strings

```java
protected List<List<SearchResults>> processStream() {
    return getInput().stream()
        .map(this::processInput)
        .collect(toList());
}
```
Applying Sequential Streams to SearchStreamGang

- `processStream()` sequentially searches for words in lists of input strings

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

Returns a list of lists of search results denoting how many times a search word appeared in each input string
Applying Sequential Streams to SearchStreamGang

- processStream() sequentially searches for words in lists of input strings

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

Returns a list of lists of search results denoting how many times a search word appeared in each input string.
**Applying Sequential Streams to SearchStreamGang**

- processStream() sequentially searches for words in lists of input strings

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

*Method is implemented via a short stream pipe*
Applying Sequential Streams to SearchStreamGang

- processStream() sequentially searches for words in lists of input strings

```java
protected List<List<SearchResults>> processStream() {
    return getInput()
        .stream() // Obtain the list of input strings as a stream
        .map(this::processInput)
        .collect(toList());
}
```
Applying Sequential Streams to SearchStreamGang

- processStream() sequentially searches for words in lists of input strings

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

Obtain the list of input strings as a stream
Applying Sequential Streams to SearchStreamGang

- processStream() sequentially searches for words in lists of input strings

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

Returns an output stream of SearchResults obtained from applying this method reference to each input strings in the stream.
protected List<List<SearchResults>> processStream() {
    return getInput()
        .stream()
        .map(this::processInput)
        .collect(toList());
}

Returns an output stream of SearchResults obtained from applying this method reference to each input strings in the stream

processInput() returns a list of SearchResults—one list for each input string
Applying Sequential Streams to SearchStreamGang

- processStream() sequentially searches for words in lists of input strings

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

Returns an output stream of SearchResults obtained from applying this method reference to each input strings in the stream.
Applying Sequential Streams to SearchStreamGang

- processStream() sequentially searches for words in lists of input strings

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

This terminal operation yields a value & triggers intermediate operation processing
Applying Sequential Streams to SearchStreamGang

- processStream() sequentially searches for words in lists of input strings

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

Returns a list of lists of search results denoting how many times a search word appeared in each input string
Applying Sequential Stream to SearchStreamGang (Part 2)
private List<SearchResults> processInput(String inputString) {
    String title = getTitle(inputString);
    String input = inputString.substring(title.length());
    List<SearchResults> results = mWordsToFind
        .stream()
        .map(word -> searchForWord(word, input, title))
        .filter(result -> result.size() > 0)
        .collect(toList());
    return results;
}
Applying Sequential Streams to SearchStreamGang

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

```java
private List<SearchResults> processInput(String inputString) {
    String title = getTitle(inputString);
    String input = inputString.substring(title.length());

    List<SearchResults> results = mWordsToFind.stream()
        .map(word -> searchForWord(word, input, title))
        .filter(result -> result.size() > 0)
        .collect(toList());

    return results;
}
```

The input is a section of a text file managed by the test driver program
Applying Sequential Streams to SearchStreamGang

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

```java
private List<SearchResults> processInput(String inputString) {
    String title = getTitle(inputString);
    String input = inputString.substring(title.length());

    List<SearchResults> results = mWordsToFind
        .stream()
        .map(word -> searchForWord(word, input, title))
        .filter(result -> result.size() > 0)
        .collect(toList());

    return results;
}
```

The input string is split into two parts.
Applying Sequential Streams to SearchStreamGang

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

```java
private List<SearchResults> processInput(String inputString) {
    String title = getTitle(inputString);
    String input = inputString.substring(title.length());

    List<SearchResults> results = mWordsToFind.stream()
        .map(word -> searchForWord(word, input, title))
        .filter(result -> result.size() > 0)
        .collect(toList());

    return results;
}
```

Convert a list of words into a stream
Applying Sequential Streams to SearchStreamGang

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

```java
private List<SearchResults> processInput(String inputString) {
    String title = getTitle(inputString);
    String input = inputString.substring(title.length());

    List<SearchResults> results = mWordsToFind.stream()
    .map(word -> searchForWord(word, input, title))
    .filter(result -> result.size() > 0)
    .collect(toList());
    return results;
}
```

Returns an output stream of SearchResults obtained from applying this function lambda to all words in the input stream.
private List<SearchResults> processInput(String inputString) {
    String title = getTitle(inputString);
    String input = inputString.substring(title.length());

    List<SearchResults> results = mWordsToFind
        .stream()
        .map(word -> searchForWord(word, input, title))
        .filter(result -> result.size() > 0)
        .collect(toList());

    return results;
}
Applying Sequential Streams to SearchStreamGang

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

```java
private List<SearchResults> processInput(String inputString) {
    String title = getTitle(inputString);
    String input = inputString.substring(title.length());

    List<SearchResults> results = mWordsToFind
        .stream()
        .map(word -> searchForWord(word, input, title))
        .filter(result -> result.size() > 0)
        .collect(toList());

    return results;
}
```

`searchForWord()` is applied to all words in the input stream
Applying Sequential Streams to SearchStreamGang

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

```java
private List<SearchResults> processInput(String inputString) {
    String title = getTitle(inputString);
    String input = inputString.substring(title.length());

    List<SearchResults> results = mWordsToFind
        .stream()
        .map(word -> searchForWord(word, input, title))
        .filter(result -> result.size() > 0)
        .collect(toList());

    return results;
}
```

searchForWord() is applied to all words in the input stream
Applying Sequential Streams to SearchStreamGang

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

```java
private List<SearchResults> processInput(String inputString) {
    String title = getTitle(inputString);
    String input = inputString.substring(title.length());

    List<SearchResults> results = mWordsToFind
        .stream()
        .map(word -> searchForWord(word, input, title))
        .filter(result -> result.size() > 0)
        .collect(toList());

    return results;
}
```

Returns an output stream of SearchResults
Applying Sequential Streams to SearchStreamGang

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

```java
private List<SearchResults> processInput(String inputString) {
    String title = getTitle(inputString);
    String input = inputString.substring(title.length());

    List<SearchResults> results = mWordsToFind
        .stream()
        .map(word -> searchForWord(word, input, title))

        .filter(result -> result.size() > 0)

        .collect(toList());

    return results;
}
```

Returns output stream containing SearchResults from input stream that match predicate lambda
Applying Sequential Streams to SearchStreamGang

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

```java
private List<SearchResults> processInput(String inputString) {
    String title = getTitle(inputString);
    String input = inputString.substring(title.length());
    List<SearchResults> results = mWordsToFind
        .stream()
        .map(word -> searchForWord(word, input, title))
        .filter(result -> result.size() > 0)
        .collect(toList());
    return results;
}
```

Returns output stream containing `SearchResults` from input stream that match predicate lambda.
Applying Sequential Streams to SearchStreamGang

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

```java
private List<SearchResults> processInput(String inputString) {
    String title = getTitle(inputString);
    String input = inputString.substring(title.length());

    List<SearchResults> results = mWordsToFind
        .stream()
        .map(word -> searchForWord(word, input, title))
        .filter(result -> result.size() > 0)
        .collect(toList());
    return results;
}
```

Returns output stream containing `SearchResults` from input stream that match predicate lambda.
Applying Sequential Streams to SearchStreamGang

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

```java
private List<SearchResults> processInput(String inputString) {
    String title = getTitle(inputString);
    String input = inputString.substring(title.length());
    List<SearchResults> results = mWordsToFind
        .stream()
        .map(word -> searchForWord(word, input, title))
        .filter(result -> result.size() > 0)
        .collect(toList());
    return results;
}
```

Returns output stream containing `SearchResults` from input stream that match predicate lambda
processInput() searches an input string for all occurrences of words to find:

```java
private List<SearchResults> processInput(String inputString) {
    String title = getTitle(inputString);
    String input = inputString.substring(title.length());

    List<SearchResults> results = mWordsToFind
        .stream()
        .map(word -> searchForWord(word, input, title))
        .filter(result -> result.size() > 0)
        .collect(toList());

    return results;
}
```

These are both intermediate operations.
Applying Sequential Streams to SearchStreamGang

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

```java
private List<SearchResults> processInput(String inputString) {
    String title = getTitle(inputString);
    String input = inputString.substring(title.length());

    List<SearchResults> results = mWordsToFind
        .stream()
        .map(word -> searchForWord(word, input, title))
        .filter(result -> result.size() > 0)
        .collect(toList());

    return results;
}
```

This terminal operation yields a value & triggers intermediate operation processing
Applying Sequential Streams to SearchStreamGang

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

```java
private List<SearchResults> processInput(String inputString) {
    String title = getTitle(inputString);
    String input = inputString.substring(title.length());

    List<SearchResults> results = mWordsToFind
        .stream()
        .map(word -> searchForWord(word, input, title))
        .filter(result -> result.size() > 0)
        .collect(toList());

    return results;
}
```

This terminal operation yields a value & triggers intermediate operation processing
processInput() searches an input string for all occurrences of words to find

```java
private List<SearchResults> processInput(String inputString) {
    String title = getTitle(inputString);
    String input = inputString.substring(title.length());

    List<SearchResults> results = mWordsToFind
        .stream()
        .map(word -> searchForWord(word, input, title))
        .filter(result -> result.size() > 0)
        .collect(toList());

    return results;
}
```

Returned back to the map() operation in processStream()
Summary of the Sequential Stream Example
Summary of the Sequential Stream Example

- There are several interesting things to note in these implementations

```java
private List<SearchResults> processInput(String inputString) {
    String title = getTitle(inputString);
    String input = inputString.substring(title.length());

    List<SearchResults> results = mWordsToFind
        .stream()
        .map(word -> searchForWord(word, input, title))

        .filter(result -> result.size() > 0)

        .collect(toList());
    return results;
}
```
Summary of the Sequential Stream Example

- There are several interesting things to note in these implementations

```java
private List<SearchResults> processInput(String inputString) {
    String title = getTitle(inputString);
    String input = inputString.substring(title.length());

    List<SearchResults> results = mWordsToFind
        .stream()
        .map(word -> searchForWord(word, input, title))
        .filter(result -> result.size() > 0)
        .collect(toList());
    return results;
}
```

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

Internal iterators shield programmers from knowledge of streams processing
Summary of the Sequential Stream Example

- There are several interesting things to note in these implementations.

```java
private List<SearchResults> processInput(String inputString) {
    String title = getTitle(inputString);
    String input = inputString.substring(title.length());

    List<SearchResults> results = mWordsToFind
        .stream()
        .map(word -> searchForWord(word, input, title))
        .filter(result -> result.size() > 0)
        .collect(toList());

    return results;
}
```

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

Focus on “what” operations to perform, rather than on “how” they’re implemented.
Summary of the Sequential Stream Example

- There are several interesting things to note in these implementations

```java
private List<SearchResults> processInput(String inputString) {
    String title = getTitle(inputString);
    String input = inputString.substring(title.length());

    List<SearchResults> results = mWordsToFind
        .stream()
        .map(word -> searchForWord(word, input, title))
        .filter(result -> result.size() > 0)
        .collect(toList());
    return results;
}
```

These lambda functions have no side-effects
There are several interesting things to note in these implementations:

```java
private List<SearchResults> processInput(String inputString) {
    String title = getTitle(inputString);
    String input = inputString.substring(title.length());

    List<SearchResults> results = mWordsToFind
        .stream()
        .map(word -> searchForWord(word, input, title))
        .filter(result -> result.size() > 0)
        .collect(toList());

    return results;
}
```

These lambda functions have no side-effects. No side-effects makes it easier to reason about behavior & enables optimization.
There are several interesting things to note in these implementations:

```java
private List<SearchResults> processInput(String inputString) {
    String title = getTitle(inputString);
    String input = inputString.substring(title.length());

    List<SearchResults> results = mWordsToFind
        .stream()
        .map(word -> searchForWord(word, input, title))
        .filter(result -> result.size() > 0)
        .collect(toList());
    return results;
}
```

Summary of the Sequential Stream Example

This example only used a few Java 8 aggregate operations.
Many other aggregate operations are part of the Java 8 stream API.
Summary of the Sequential Stream Example

- Many other aggregate operations are part of the Java 8 stream API
- We’ll cover more of them later in this course

See later discussions in the ImageStreamGang app
Summary of the Sequential Stream Example

- Many other aggregate operations are part of the Java 8 stream API
- We’ll cover more of them later in this course
- Other LiveLessons cover Java 8 streams in greater detail

See safari5.bvdep.com/video/programming/java/9780134383644
Summary of the Sequential Stream Example

- Many other aggregate operations are part of the Java 8 stream API
  - We’ll cover more of them later in this course
- Other LiveLessons cover Java 8 streams in greater detail
- Another good source of material to consult is the book *Java 8 in Action*

See [www.manning.com/books/java-8-in-action](http://www.manning.com/books/java-8-in-action)
Many other aggregate operations are part of the Java 8 stream API. We'll cover more of them later in this course. Other LiveLessons cover Java 8 streams in greater detail. Another good source of material to consult is the book Java 8 in Action. There are also good online articles: See www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html
Many other aggregate operations are part of the Java 8 stream API.

- We’ll cover more of them later in this course.
- Other LiveLessons cover Java 8 streams in greater detail.
- Another good source of material to consult is the book *Java 8 in Action*.
- There are also good online articles.

Summary of the Sequential Stream Example

End of Overview of Java 8 Streams (Part 3)