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 (behavior $f$)

Output $f(x)$

Aggregate operation (behavior $g$)

Output $g(f(x))$

Aggregate operation (behavior $h$)

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

```
Input x

Aggregate operation (behavior f)

Output f(x)

Aggregate operation (behavior g)

Output g(f(x))

Aggregate operation (behavior 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
  - We’ll use an example program to illustrate key concepts

See [github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex12](https://github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex12)
Overview of Java 8 Streams
Overview of Java 8 Streams

- Java 8 streams are an addition to the Java library that provide programs with several key benefits.

See docs.oracle.com/javase/tutorial/collections/streams
Overview of Java 8 Streams

- Java 8 streams are an addition to the Java library that provide programs with several key benefits
- Manipulate flows of data in a declarative way

This stream expresses what operations to perform, not how to perform them

Java 8 streams are an addition to the Java library that provide programs with several key benefits:

- Manipulate flows of data in a declarative way.
- Enable transparent parallelization without the need to write any multi-threaded code.

The data elements in this stream are automatically mapped to processor cores.

See [docs.oracle.com/javase/tutorial/collections_streams/parallelism.html](docs.oracle.com/javase/tutorial/collections_streams/parallelism.html)
Overview of Java 8 Streams

- A stream is a pipeline of aggregate operations that process a sequence of elements (aka, “values”)

Input $x$

\[ \text{Aggregate operation (behavior } f \text{)} \]

Output $f(x)$

\[ \text{Aggregate operation (behavior } g \text{)} \]

Output $g(f(x))$

\[ \text{Aggregate operation (behavior } h \text{)} \]

See docs.oracle.com/javase/tutorial/collections/streams
Overview of Java 8 Streams

- A stream is a pipeline of aggregate operations that process a sequence of elements (aka, “values”)

A stream is conceptually unbounded, though they are typically bounded by practical constraints.
Overview of Java 8 Streams

- A stream is a pipeline of aggregate operations that process a sequence of elements (aka, “values”)

```java
Stream.of("horatio",
    "laertes",
    "Hamlet", ...)
    .filter(s -> toLowerCase(s.charAt(0)) == 'h')
    .map(this::capitalize)
    .sorted()
    .forEach(System.out::println);
```

See [github.com/douglasraigschmidt/LiveLessons/tree/master/Java8/ex12](https://github.com/douglasraigschmidt/LiveLessons/tree/master/Java8/ex12)
Overview of Java 8 Streams

- A stream is created via a factory method

Stream
  .of("horatio",
   "laertes",
   "Hamlet",
   ...
  )

Input x

Aggregate operation (behavior f)

Output f(x)

Aggregate operation (behavior g)

Output g(f(x))

Aggregate operation (behavior h)

See [en.wikipedia.org/wiki/Factory_method_pattern](en.wikipedia.org/wiki/Factory_method_pattern)
Overview of Java 8 Streams

- A stream is created via a factory method

Stream
.of("horatio",
"laertes",
"Hamlet", ...) ...

Input x
Aggregate operation (behavior f)
Output f(x)
Aggregate operation (behavior g)
Output g(f(x))
Aggregate operation (behavior h)

The of() factory method converts an array of T into a stream of T

See docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#of
Overview of Java 8 Streams

- A stream is created via a factory method
  
  ```java
collection.stream()
collection.parallelStream()
Pattern.compile(...) .splitAsStream()
Stream.of(value1, ... ,valueN)
Arrays.stream(array)
Arrays.stream(array, start, end)
Files.lines(file_path)
"string".chars()
Stream.builder().add(...).build()
Stream.generate(generate_expression)
Files.list(file_path)
Files.find(file_path, max_depth, matcher)
Stream.generate(iterator::next)
Stream.iterate(init_value, generate_expression)
StreamSupport.stream(iterable.spliterator(), false)
...
```

There are many other factory methods that create streams
Overview of Java 8 Streams

- An aggregate operation performs a *behavior* on each element in a stream

\[ \text{Input } x \rightarrow \text{Aggregate operation (behavior } f) \]

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

- An aggregate operation performs a behavior on each element in a stream

```java
Stream.of("horatio", "laertes", "Hamlet", ...)
    .filter(s -> toLowerCase(s.charAt(0)) == 'h')
    .map(this::capitalize)
    .sorted()
    .forEach(System.out::println);
```

See [github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex12](github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex12)
Overview of Java 8 Streams

- An aggregate operation performs a behavior on each element in a stream.
- Ideally, a behavior’s output in a stream depends only on its input arguments.

See [en.wikipedia.org/wiki/Side_effect_(computer_science)](en.wikipedia.org/wiki/Side_effect_(computer_science))
Overview of Java 8 Streams

- An aggregate operation performs a *behavior* on each element in a stream.
- Ideally, a behavior’s output in a stream depends only on its input arguments.

```java
String capitalize(String s) {
    if (s.length() == 0)
        return s;
    return s.substring(0, 1)
        .toUpperCase()
        + s.substring(1)
        .toLowerCase();
}
```

See [github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex12](https://github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex12)
An aggregate operation performs a behavior on each element in a stream.

Ideally, a behavior’s output in a stream depends only on its input arguments.

Behaviors with side-effects likely incur race conditions in parallel streams.
Overview of Java 8 Streams

- An aggregate operation performs a behavior on each element in a stream.
- Ideally, a behavior’s output in a stream depends only on its input arguments.
- Behaviors with side-effects likely incur race conditions in parallel streams.

Only you can prevent race conditions!

In Java you must avoid race conditions, i.e., the compiler & JVM won’t save you.
Overview of Java 8 Streams

- Streams enhance flexibility by forming a “processing pipeline” that chains multiple aggregate operations together

\[
\text{Input } x \quad \rightarrow \quad \text{Aggregate operation (behavior } f)\quad \rightarrow \quad \text{Output } f(x) \quad \rightarrow \quad \text{Aggregate operation (behavior } g)\quad \rightarrow \quad \text{Output } g(f(x)) \quad \rightarrow \quad \text{Aggregate operation (behavior } h)\quad \rightarrow \quad \text{Output } h(g(f(x)))
\]

See [en.wikipedia.org/wiki/Pipeline_(software)](en.wikipedia.org/wiki/Pipeline_(software))
Overview of Java 8 Streams

- Streams enhance flexibility by forming a “processing pipeline” that chains multiple aggregate operations together.

Each aggregate operation in the pipeline can filter and/or transform the stream.
Overview of Java 8 Streams

- A stream holds no non-transient storage
Overview of Java 8 Streams

• Every stream works very similarly
Overview of Java 8 Streams

- Every stream works very similarly
- Starts with a source of data

```
Stream.of("horatio",
    "laertes",
    "Hamlet", ...)
```

E.g., a Java array, collection, generator function, or input channel
Overview of Java 8 Streams

• Every stream works very similarly
• Starts with a source of data

```
List<String> characters = Arrays.asList("horatio", "laertes", "Hamlet", ...);

characters.stream()
...
```

e.g., a Java array, collection, generator function, or input channel
Overview of Java 8 Streams

- Every stream works very similarly
  - Starts with a source of data
  - Processes the data through a pipeline of intermediate operations

Example of a stream:

```java
Stream.of("horatio", "laertes", "Hamlet", ...)
  .filter(s -> toLowerCase(s.charAt(0)) == 'h')
  .map(this::capitalize)
  .sorted()
...
```

Examples of intermediate operations include `filter()`, `map()`, & `flatMap()`
Overview of Java 8 Streams

• Every stream works very similarly
  • Starts with a source of data
  • Processes the data through a pipeline of intermediate operations
  • Finishes with a terminal operation that yields a non-stream result

```java
... .filter(s -> toLowerCase(s.charAt(0)) == 'h') .map(this::capitalize) .sorted() .forEach(System.out::println);
```
Overview of Java 8 Streams

- Every stream works very similarly
  - Starts with a source of data
  - Processes the data through a pipeline of intermediate operations
  - Finishes with a terminal operation that yields a non-stream result

```java
... .filter(s -> toLowerCase(s.charAt(0)) == 'h') .map(this::capitalize) .sorted() .forEach(System.out::println);
```

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

- Every stream works very similarly
  - Starts with a source of data
  - Processes the data through a pipeline of intermediate operations
  - Finishes with a terminal operation that yields a non-stream result, e.g.
    - no value at all

```java
void runForEach() {
    Stream.of("horatio",
            "laertes",
            "Hamlet",
            ...)
        .filter(s -> toLowerCase(s.charAt(0)) == 'h')
        .map(this::capitalize)
        .sorted()
        .forEach(System.out::println);
}
```

See [docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#forEach](docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#forEach)
Overview of Java 8 Streams

- Every stream works very similarly
  - Starts with a source of data
  - Processes the data through a pipeline of intermediate operations
  - Finishes with a terminal operation that yields a non-stream result, e.g.
    - no value at all
    - a collection

```java
void runCollect() {
    List<String> characters =
        Arrays.asList("horatio", "laertes", "Hamlet", ...);
    List<String> results =
        characters
            .stream()
            .filter(s -> toLowerCase(...) == 'h')
            .map(this::capitalize)
            .sorted()
            .collect(toList()); ...
}
```
Overview of Java 8 Streams

- Every stream works very similarly
  - Starts with a source of data
  - Processes the data through a pipeline of intermediate operations
  - Finishes with a terminal operation that yields a non-stream result, e.g.
    - no value at all
    - a collection

```java
void runCollect() {
    List<String> characters = Arrays.asList("horatio", "laertes", "Hamlet", ...);
    Map<String, Long> results =
        characters
            .collect
            (groupingBy
                (identity(),
                TreeMap::new,
                summingLong
                (String::length)));
    ...
}
```

`collect()` can be used with a range of powerful collectors, e.g., to group by name & length of name

See docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html
Overview of Java 8 Streams

- Every stream works very similarly
  - Starts with a source of data
  - Processes the data through a pipeline of intermediate operations
  - Finishes with a terminal operation that yields a non-stream result, e.g.
    - no value at all
    - a collection

```java
void runCollect() {
    List<String> characters = Arrays.asList("horatio", "laertes", "Hamlet", ...);
    Map<String, Long> results = ... .collect
        (groupingBy
            (identity(),
             TreeMap::new,
             summingLong
             (String::length)));
    ...
}
```

See docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html#groupingBy
Overview of Java 8 Streams

- Every stream works very similarly
  - Starts with a source of data
  - Processes the data through a pipeline of intermediate operations
  - Finishes with a terminal operation that yields a non-stream result, e.g.
    - no value at all
    - a collection
    - a primitive value

```
void runCollectReduce() {
    Map<String, Long> matchingCharactersMap =
        Pattern.compile",").splitAsStream
            .splitAsStream("horatio,Hamlet,...")
    ...
    long countOfNameLengths =
        matchingCharactersMap
            .values()
            .stream()
            .reduce(0L,
                    (x, y) -> x + y);
    // Could use .sum()
}
```
Overview of Java 8 Streams

• Every stream works very similarly
  • Starts with a source of data
  • Processes the data through a pipeline of intermediate operations
  • Finishes with a terminal operation that yields a non-stream result, e.g.
    • no value at all
    • a collection
    • a primitive value

0 is the “identity,” i.e., the initial value of the reduction & the default result if there are no elements in the stream

```java
void runCollectReduce() {
    Map<String, Long>
        matchingCharactersMap = Pattern.compile","
            .splitAsStream
            ("horatio, Hamlet,...")
    ...
    long countOfNameLengths = matchingCharactersMap
        .values()
        .stream()
        .reduce(0L,  
            (x, y) -> x + y);  
    // Could use .sum()
```
Overview of Java 8 Streams

- Every stream works very similarly
  - Starts with a source of data
  - Processes the data through a pipeline of intermediate operations
  - Finishes with a terminal operation that yields a non-stream result, e.g.
    - no value at all
    - a collection
    - a primitive value

This lambda is the “accumulator,” which is a stateless function that combines two values

```java
void runCollectReduce() {
  Map<String, Long>
      matchingCharactersMap =
        Pattern.compile("","")
          .splitAsStream
            ("horatio,Hamlet,...")
      ...
  long countOfNameLengths =
    matchingCharactersMap
      .values()
      .stream()
      .reduce(0L,
        (x, y) -> x + y);
  // Could use .sum()
```
Overview of Java 8 Streams

- Every stream works very similarly
  - Starts with a source of data
  - Processes the data through a pipeline of intermediate operations
  - Finishes with a terminal operation that yields a non-stream result, e.g.
    - no value at all
    - a collection
    - a primitive value

There's a 3 parameter "map/reduce" version of reduce() that's used in parallel streams

void runCollectReduce() {
    Map<String, Long>
        matchingCharactersMap = 
          Pattern.compile("","")
            .splitAsStream
              ("horatio,Hamlet,...")
    ...
    long countOfNameLengths =
        matchingCharactersMap
          .values()
          .stream()
          .reduce(0L,
              (x, y) -> x + y,
              (x, y) -> x + y);

See www.youtube.com/watch?v=oWIWEKNM5Aw
Overview of Java 8 Streams

• Every stream works very similarly
  • Starts with a source of data
  • Processes the data through a pipeline of intermediate operations
  • Finishes with a terminal operation that yields a non-stream result

Each stream must have one (and only one) terminal operation
Overview of Java 8 Streams

• Each aggregate operation in a stream runs its behavior sequentially by default

See radar.oreilly.com/2015/02/java-8-streams-api-and-parallelism.html
Each aggregate operation in a stream runs its behavior sequentially by default
- i.e., one at a time in a single thread

We’ll cover sequential streams first

See docs.oracle.com/javase/tutorial/collections/streams
Overview of Java 8 Streams

- A Java 8 parallel stream splits its elements into multiple chunks & uses a common fork-join pool to process the chunks independently.

We’ll cover parallel streams shortly.

See docs.oracle.com/javase/tutorial/collections/streams/parallelism.html
End of Overview of Java 8 Streams (Part 1)