Overview of Java 8 Streams (Part 1)

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

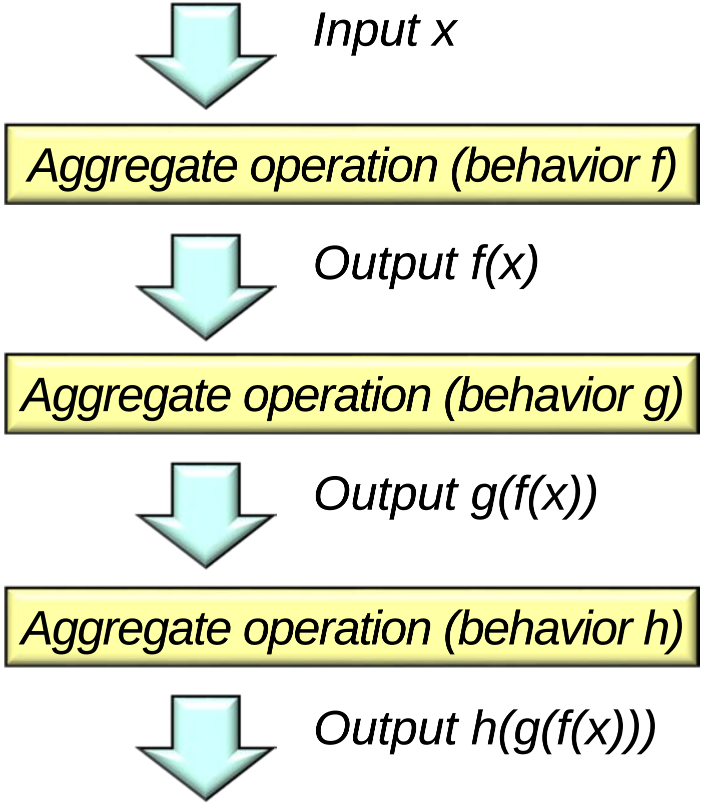
Professor of Computer Science

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)

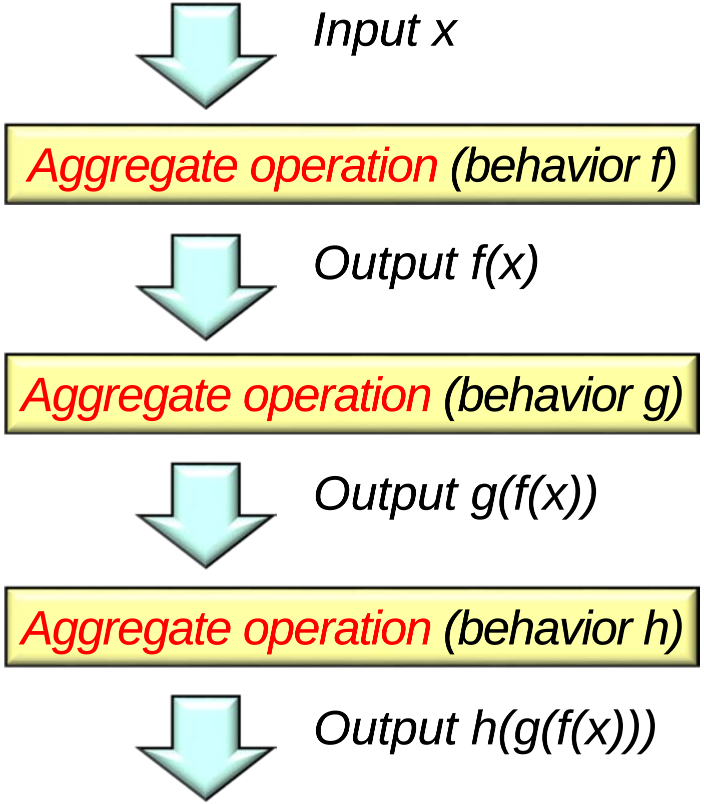
Output g(f(x))

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))

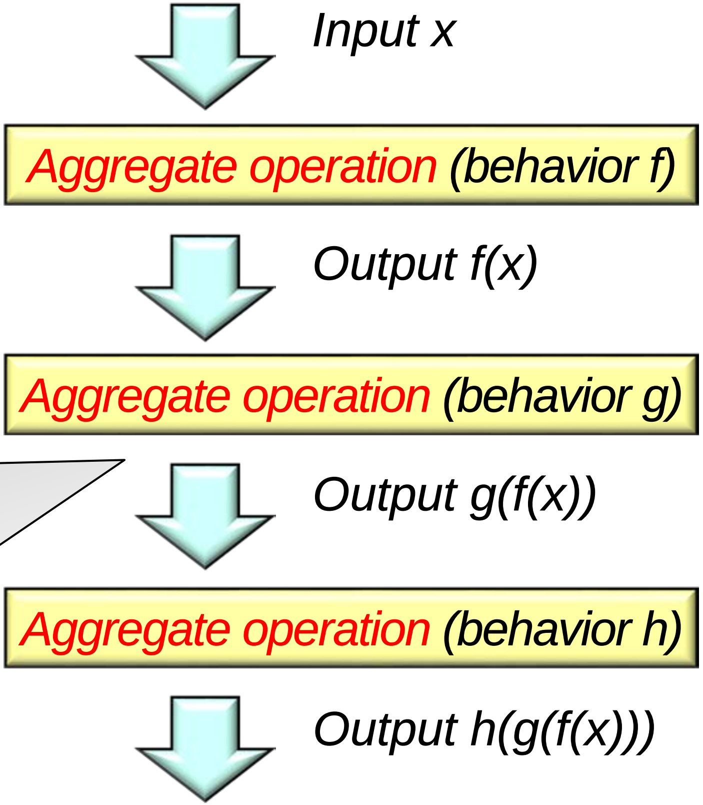
Aggregate operation (behavior h)

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

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



Input x

Aggregate operation (behavior f)

Output f(x)

Aggregate operation (behavior g)

Output g(f(x))

Aggregate operation (behavior h)

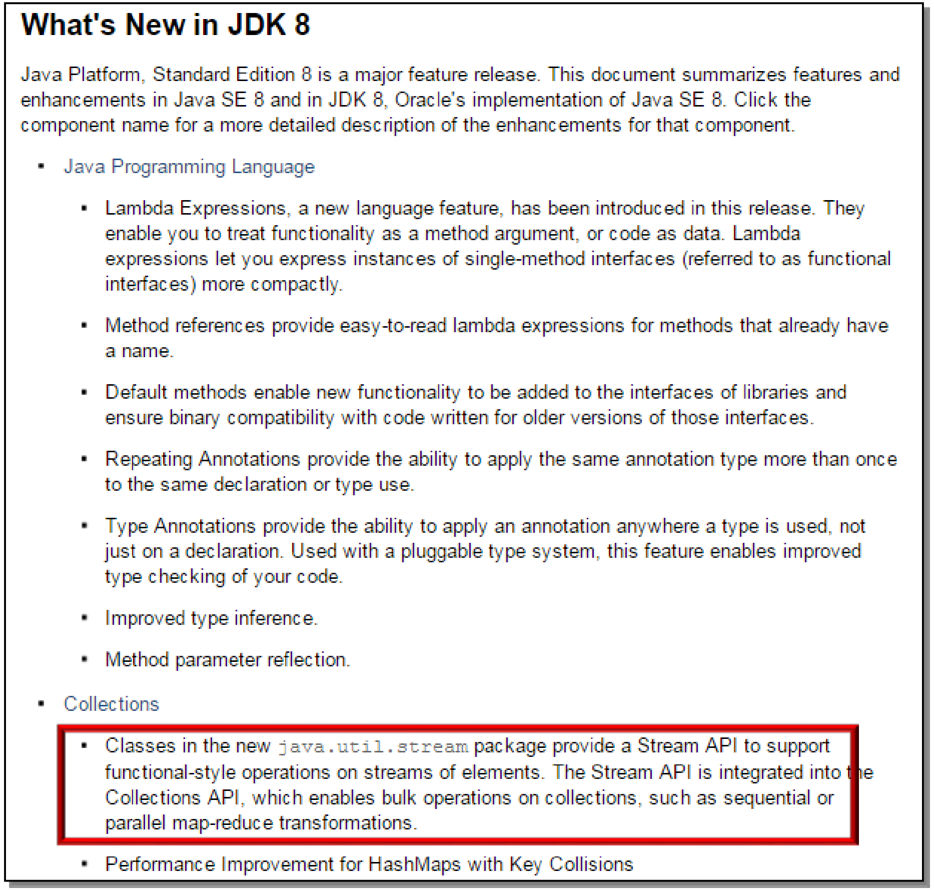
Output h(g(f(x)))

See 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



What's New in JDK 8

Java Platform, Standard Edition 8 is a major feature release. This document summarizes features and enhancements in Java SE 8 and in JDK 8, Oracle's implementation of Java SE 8. Click the component name for a more detailed description of the enhancements for that component.

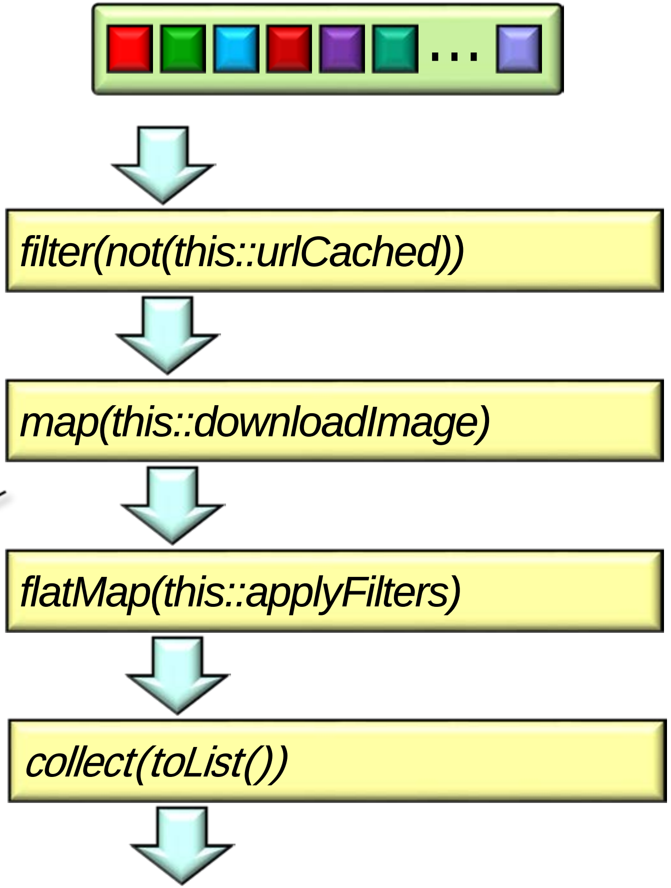
- Java Programming Language
  - Lambda Expressions, a new language feature, has been introduced in this release. They enable you to treat functionality as a method argument, or code as data. Lambda expressions let you express instances of single-method interfaces (referred to as functional interfaces) more compactly.
  - Method references provide easy-to-read lambda expressions for methods that already have a name.
  - Default methods enable new functionality to be added to the interfaces of libraries and ensure binary compatibility with code written for older versions of those interfaces.
  - Repeating Annotations provide the ability to apply the same annotation type more than once to the same declaration or type use.
  - Type Annotations provide the ability to apply an annotation anywhere a type is used, not just on a declaration. Used with a pluggable type system, this feature enables improved type checking of your code.
  - Improved type inference.
  - Method parameter reflection.
- Collections
  - Classes in the new `java.util.stream` package provide a Stream API to support functional-style operations on streams of elements. The Stream API is integrated into the Collections API, which enables bulk operations on collections, such as sequential or parallel map-reduce transformations.
  - Performance Improvement for HashMaps with Key Collisions

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

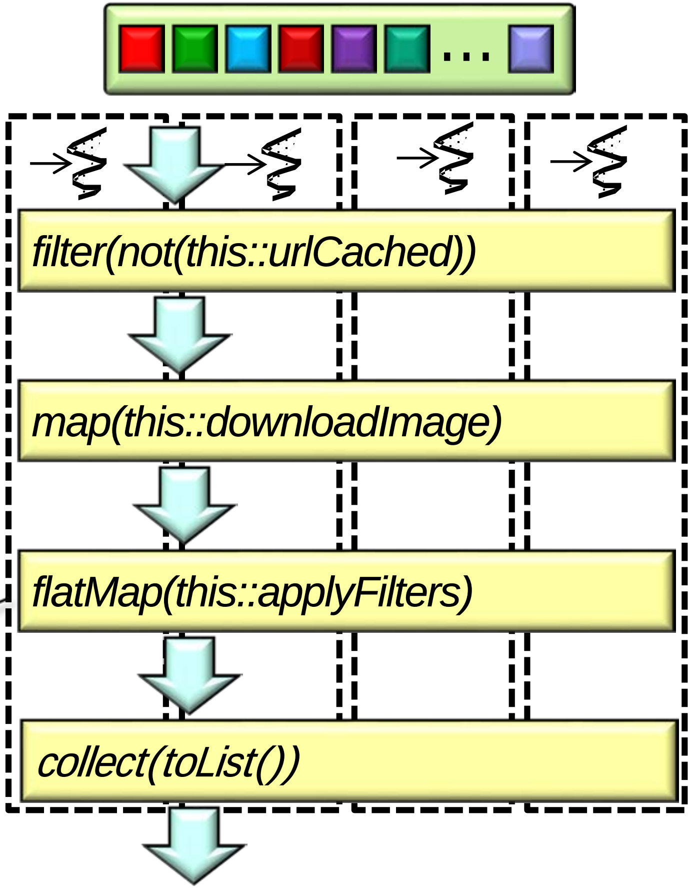
*This stream expresses **what** operations to perform, not **how** to perform them*



See github.com/douglascraigschmidt/LiveLessons/tree/master/ImageStreamGang

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
  - Enable transparent parallelization without the need to write any multi-threaded code

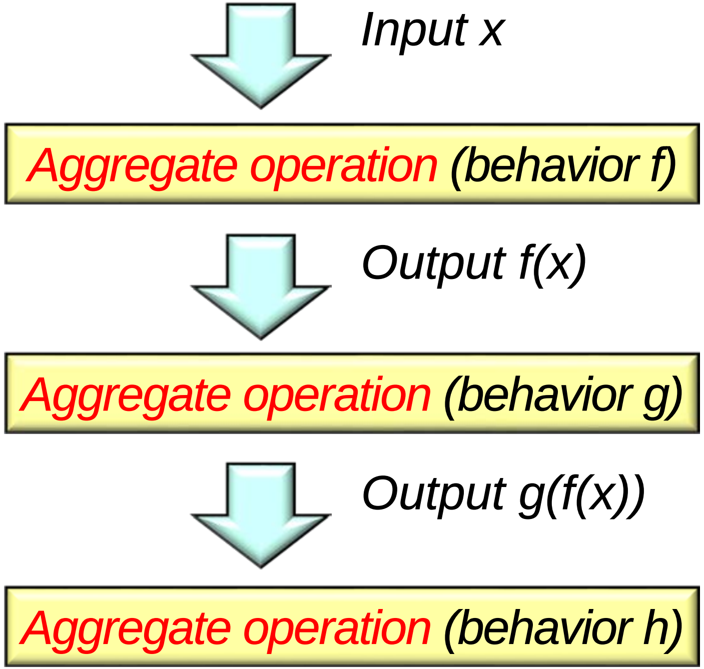


The data elements in this stream are automatically mapped to processor cores

See docs.oracle.com/javase/tutorial/collections/streams/parallelism.html

Overview of Java 8 Streams

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



Input x

Aggregate operation (behavior f)

Output f(x)

Aggregate operation (behavior g)

Output g(f(x))

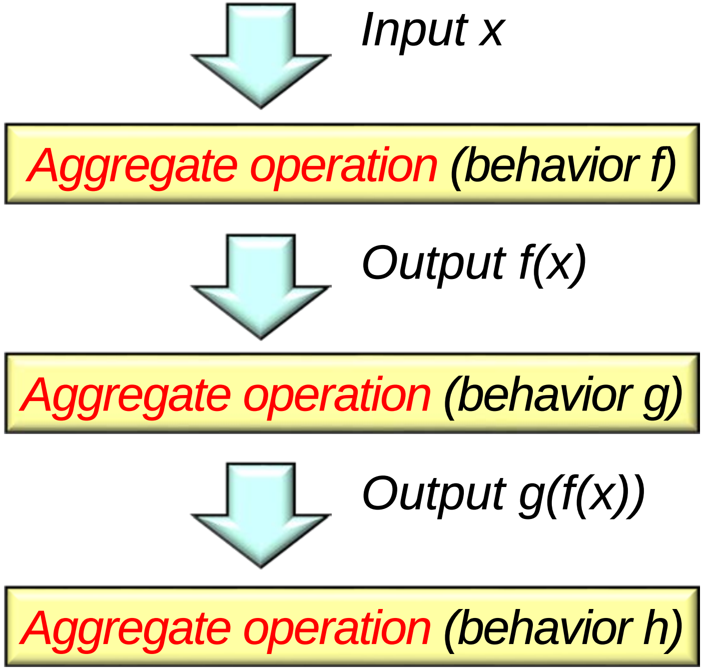
Aggregate operation (behavior h)

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



Input x

Aggregate operation (behavior f)

Output f(x)

Aggregate operation (behavior g)

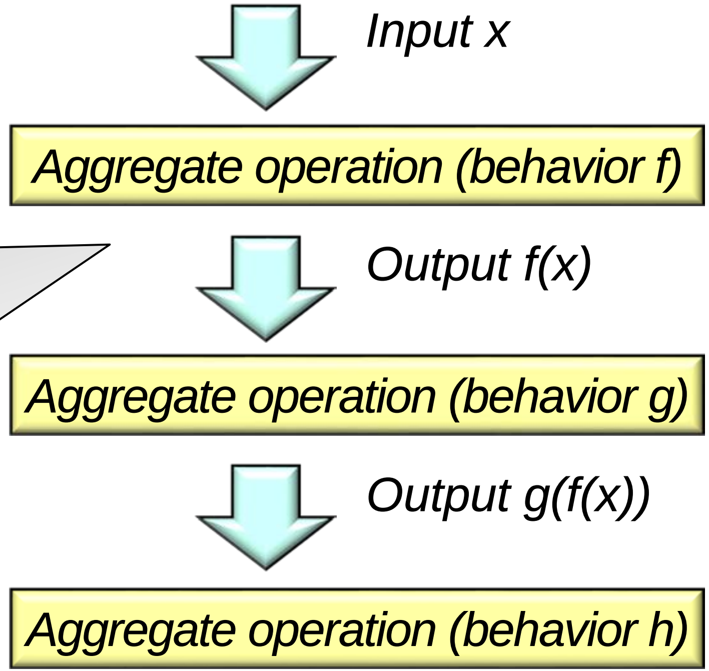
Output g(f(x))

Aggregate operation (behavior h)

Overview of Java 8 Streams

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

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



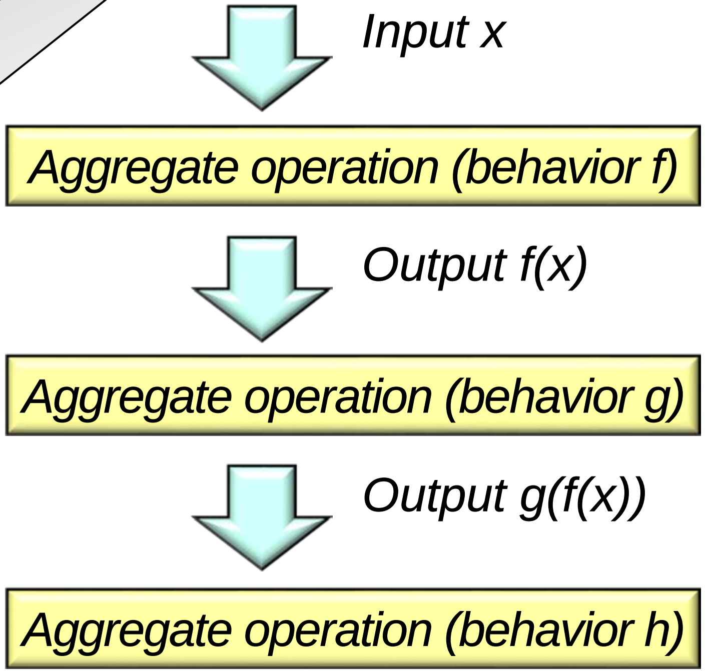
Input x
Aggregate operation (behavior f)
Output f(x)
Aggregate operation (behavior g)
Output g(f(x))
Aggregate operation (behavior h)

See github.com/douglascraigschmidt/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))

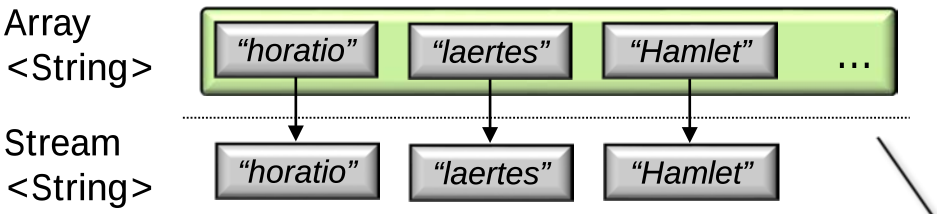
Aggregate operation (behavior h)

See en.wikipedia.org/wiki/Factory_method_pattern

Overview of Java 8 Streams

- A stream is created via a factory method

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



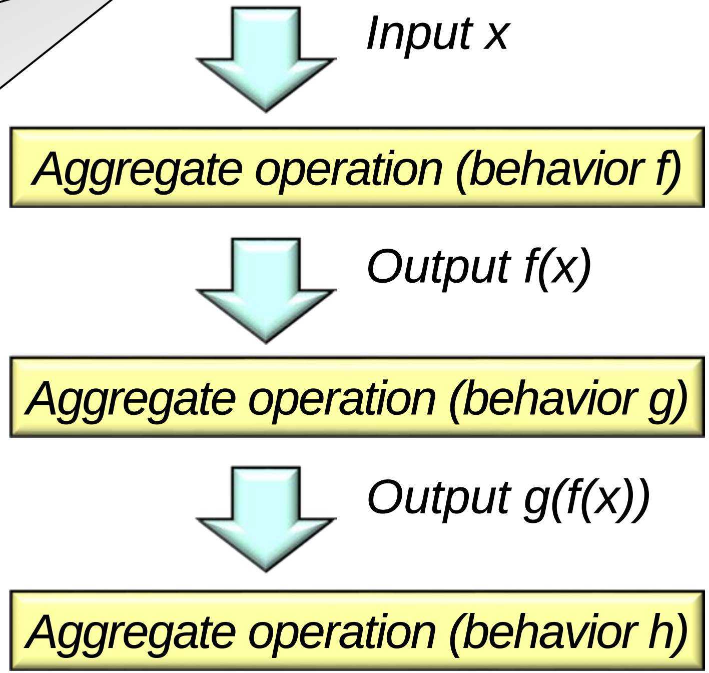
Array <String>

"horatio" "laertes" "Hamlet" …

Stream <String>

"horatio" "laertes" "Hamlet"

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



Input x

Aggregate operation (behavior f)

Output f(x)

Aggregate operation (behavior g)

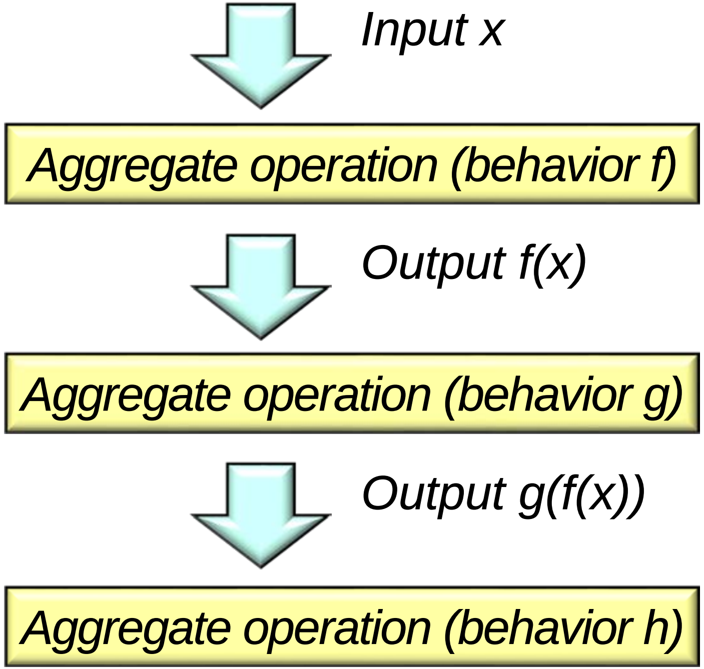
Output g(f(x))

Aggregate operation (behavior h)

Overview of Java 8 Streams

- A stream is created via a factory method

```
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, mathcher)
Stream.generate(iterator::next)
Stream.iterate(init_value, generate_expression)
StreamSupport.stream(iterable.spliterator(), false)
...
```


Input x

Aggregate operation (behavior f)

Output f(x)

Aggregate operation (behavior g)

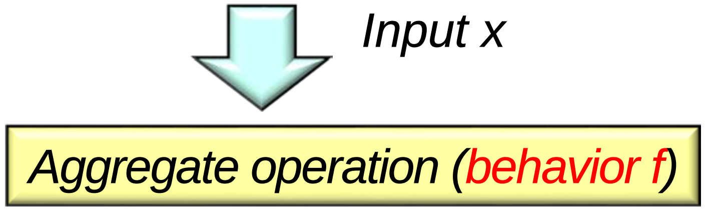
Output g(f(x))

Aggregate operation (behavior h)

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



Input x

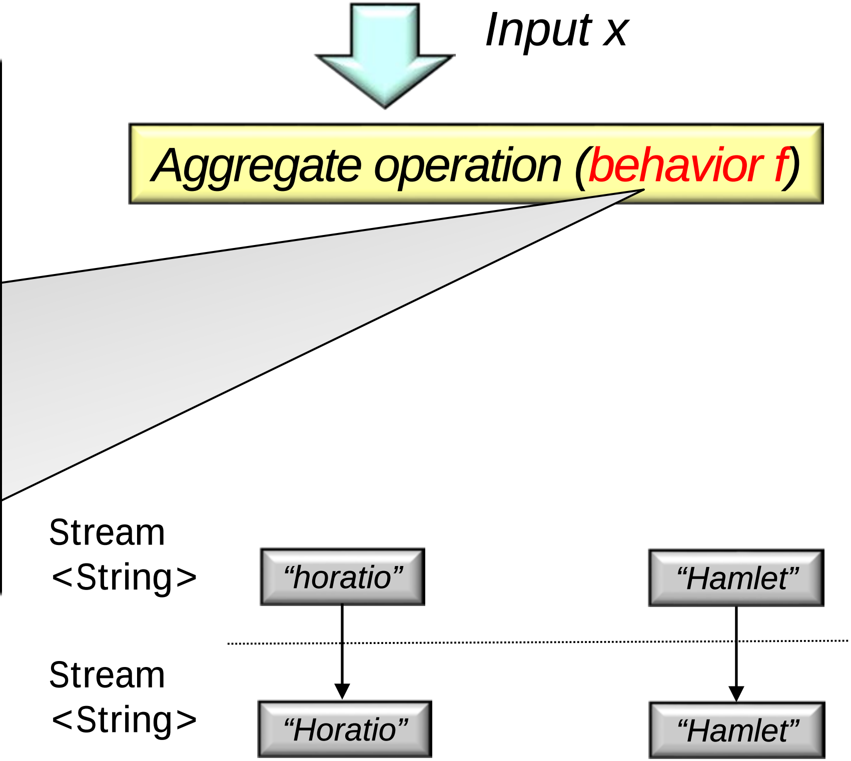
Aggregate operation (behavior f)

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

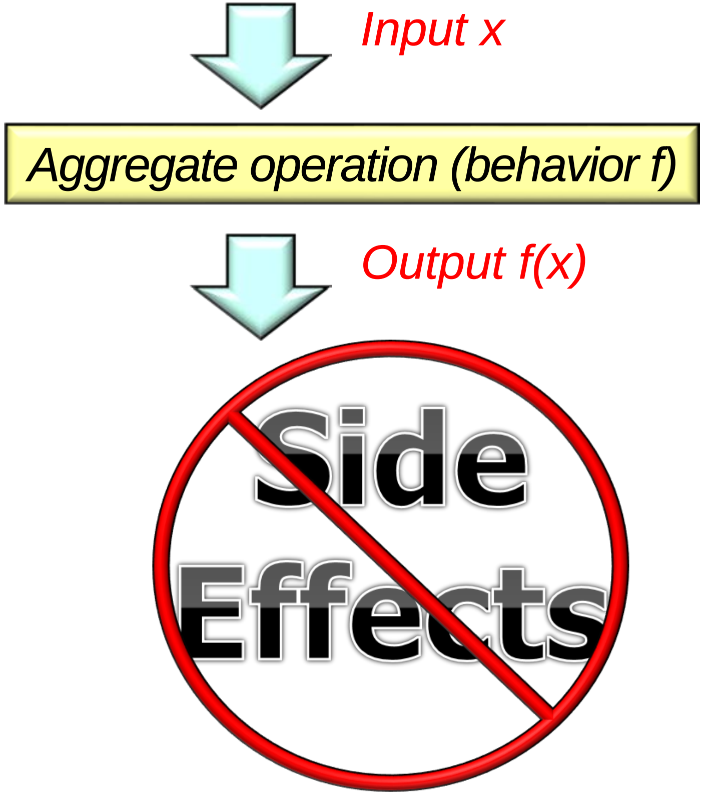
```
Stream
   .of("horatio",
       "laertes",
       "Hamlet", ...)
   .filter(s -> toLowerCase
                  (s.charAt(0)) == 'h')
   .map(this::capitalize)
   .sorted()
   .forEach(System.out::println);
```



See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex12

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



Input x

Aggregate operation (behavior f)

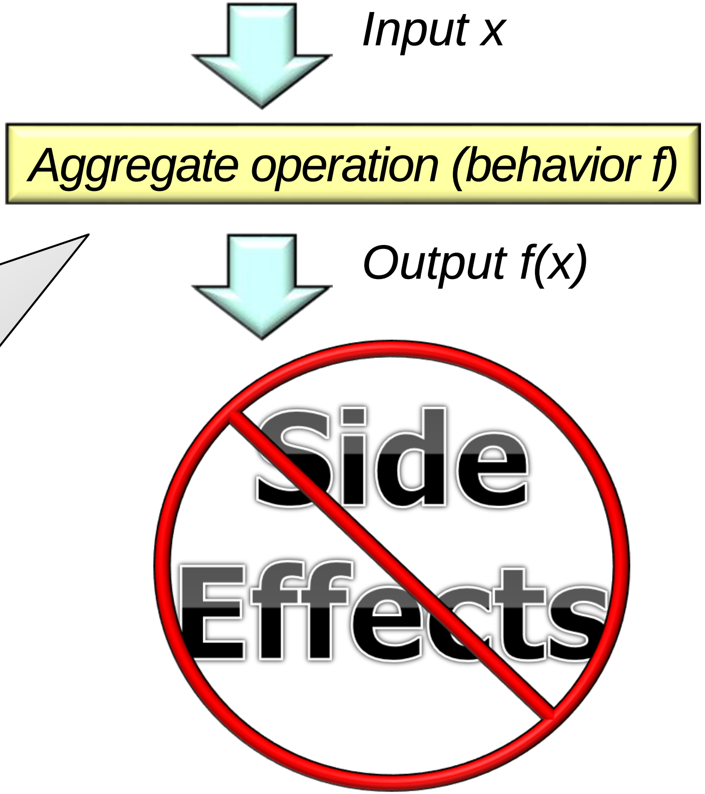
Output f(x)

See 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

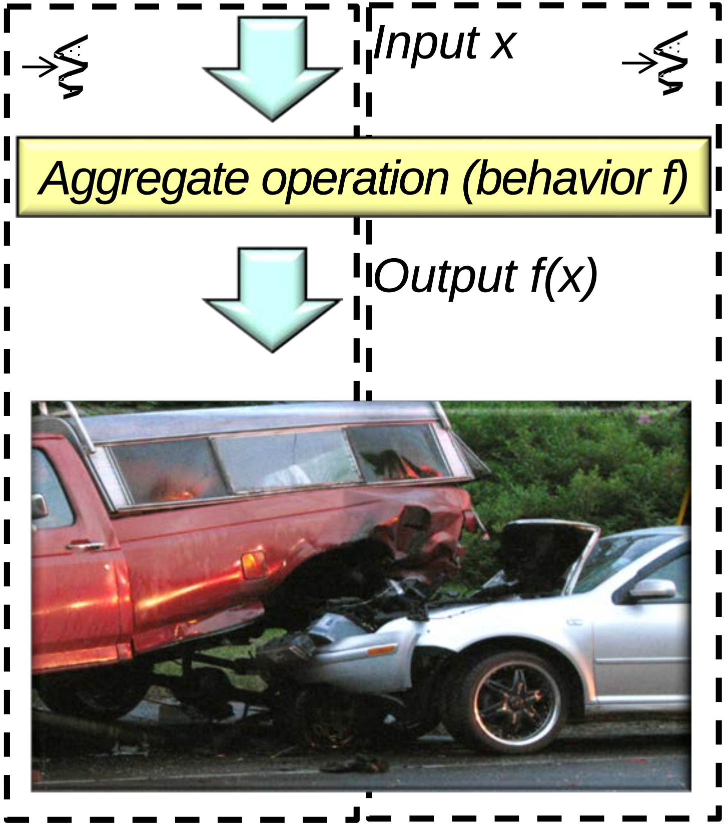
```
String capitalize(String s) {
  if (s.length() == 0)
    return s;
  return s.substring(0, 1)
          .toUpperCase()
         + s.substring(1)
            .toLowerCase();
```



See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex12

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



Input x

Aggregate operation (behavior f)

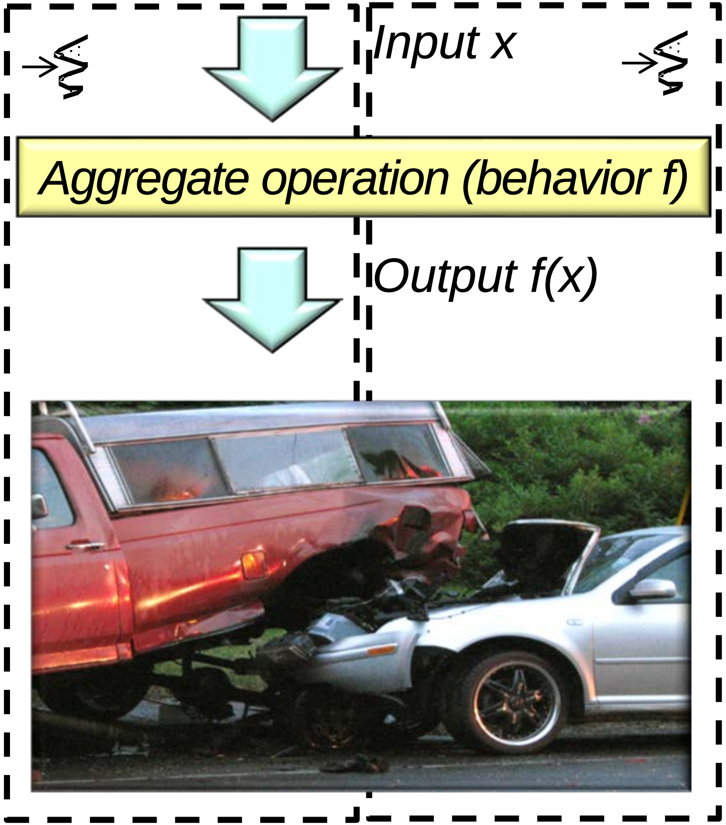
Output f(x)

In Java *you* must avoid race conditions, i.e., the compiler & JVM won't save you..

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!



Input x

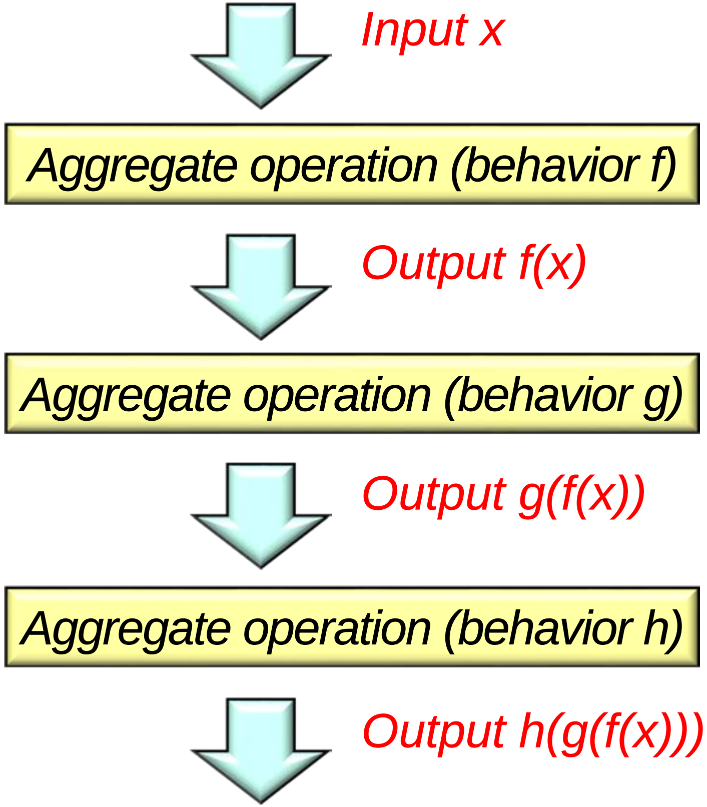
Aggregate operation (behavior f)

Output f(x)

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



Input x

Aggregate operation (behavior f)

Output f(x)

Aggregate operation (behavior g)

Output g(f(x))

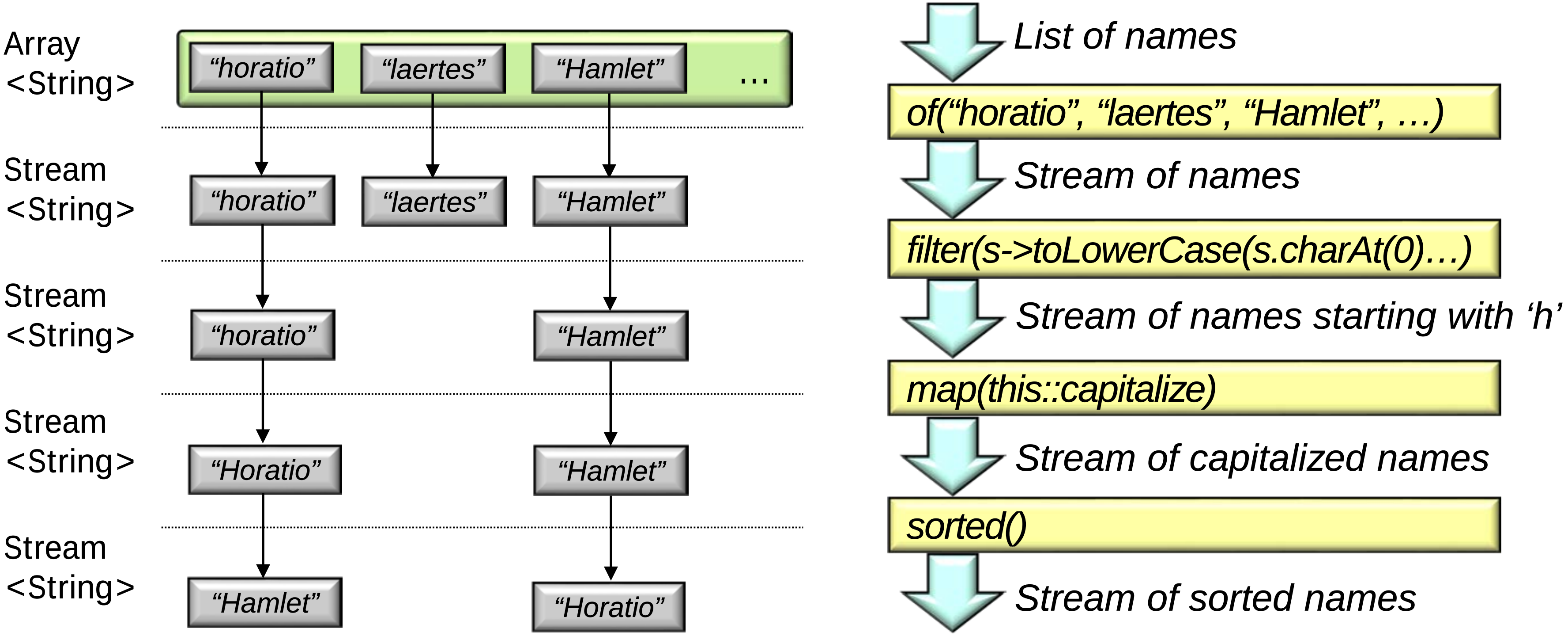
Aggregate operation (behavior h)

Output h(g(f(x)))

See 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



Array <String>

“horatio” “laertes” “Hamlet” …

Stream <String>

“horatio” “laertes” “Hamlet”

Stream <String>

“horatio” “Hamlet”

Stream <String>

“Horatio” “Hamlet”

Stream <String>

“Hamlet” “Horatio”

List of names

of(“horatio”, “laertes”, “Hamlet”, …)

Stream of names

filter(s->toLowerCase(s.charAt(0)…)

Stream of names starting with ‘h’

map(this::capitalize)

Stream of capitalized names

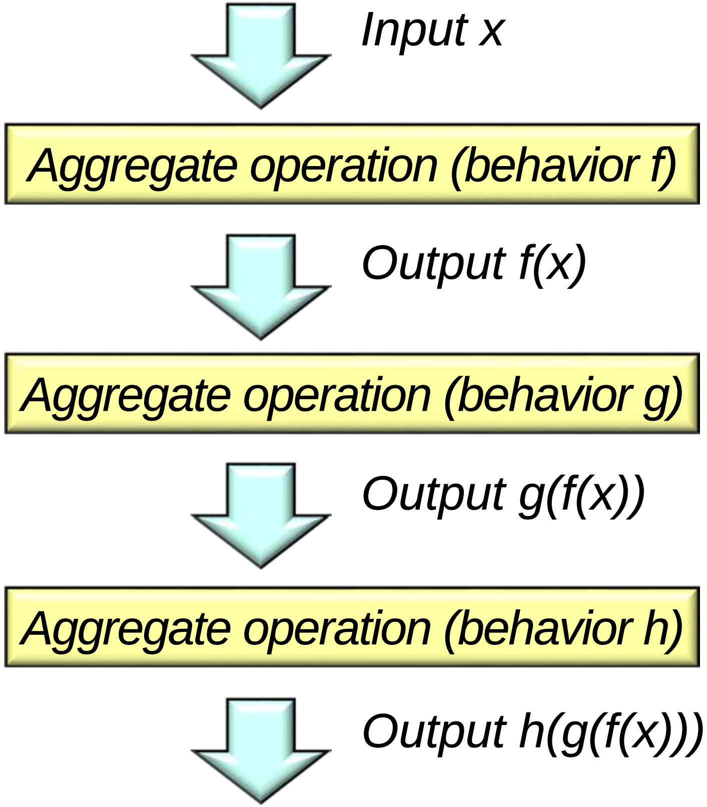
sorted()

Stream of sorted names

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



Input x

Aggregate operation (behavior f)

Output f(x)

Aggregate operation (behavior g)

Output g(f(x))

Aggregate operation (behavior h)

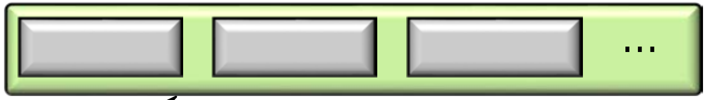
Output h(g(f(x)))

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

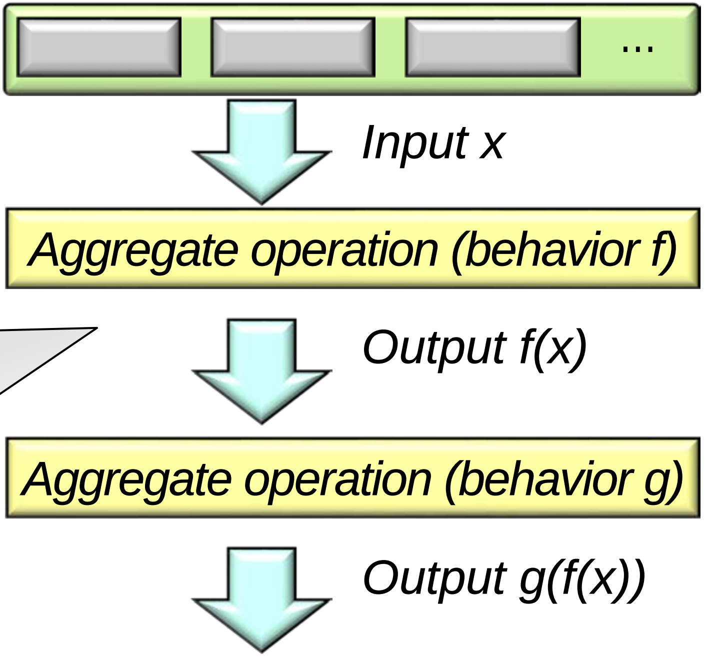
```
List<String> characters =
  Arrays.asList("horatio",
                "laertes",
                "Hamlet", ...);

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

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



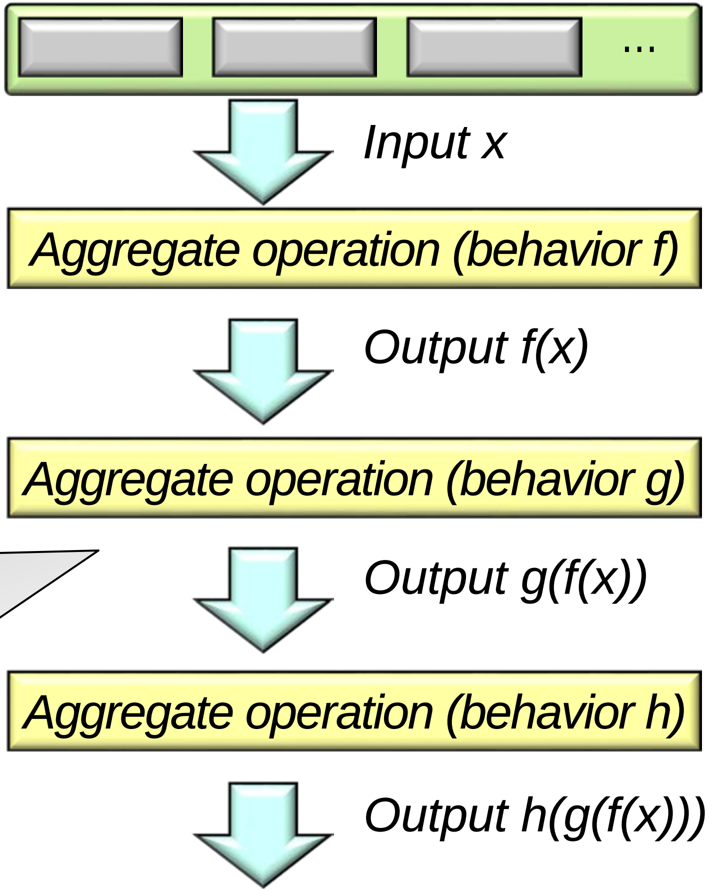
...
Input x
Aggregate operation (behavior f)
Output f(x)
Aggregate operation (behavior g)
Output g(f(x))

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

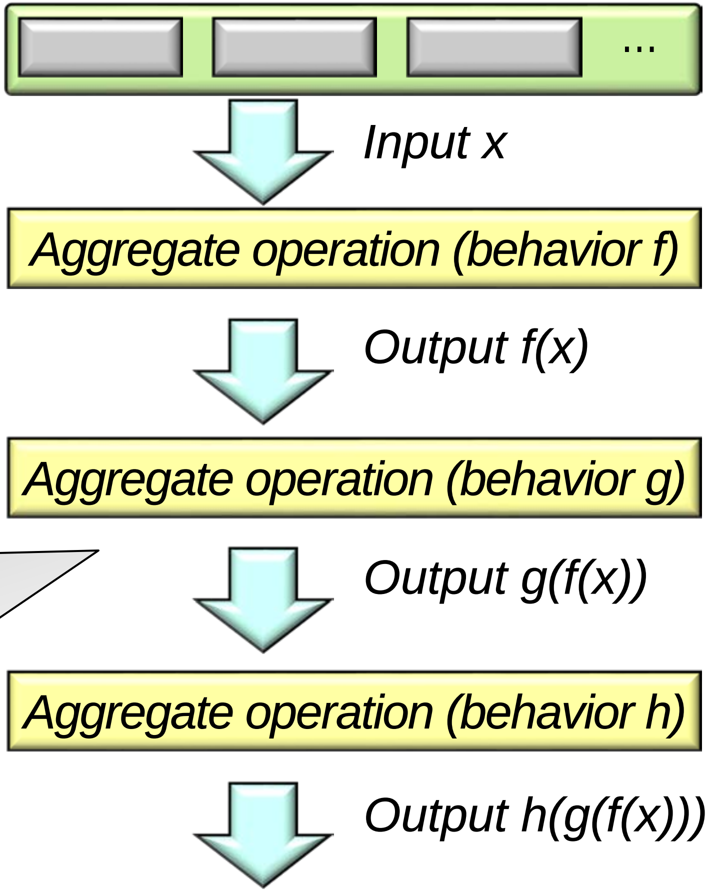
```
...
.filter(s -> toLowerCase
          (s.charAt(0)) == 'h')
.map(this::capitalize)
.sorted()
.forEach(System.out::println);
```


...
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)))

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

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


...
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)))

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

```
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

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

```
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()); ...
```

See docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#collect

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

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

collect() can be used with a wide range of powerful collectors

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

```
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
        ("horatio,Hamlet,…")
    ...
  long countOfNameLengths =
    matchingCharactersMap
      .values()
      .stream()
      .reduce(0L,
           (x, y) -> x + y);
      // Could use .sum()
```

See docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#reduce

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
        ("horatio,Hamlet,…")
    ...
  long countOfNameLengths =
    matchingCharactersMap
      .values()
      .stream()
      .reduce(0L,
           (x, y) -> x + y);
      // Could use .sum()
```

0 is the "identity," i.e., the initial value of the reduction & the default result if there are no elements in the 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
    - a collection
    - a primitive value

```
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()
```

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

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
    ("horatio,Hamlet,…")
  ...
 long countOfNameLengths =
  matchingCharactersMap
   .values()
   .stream()
   .reduce(0L,
      (x, y) -> x + y,
      (x, y) -> x + y);
```

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

See www.youtube.com/watch?v=oWlWEKNM5Aw

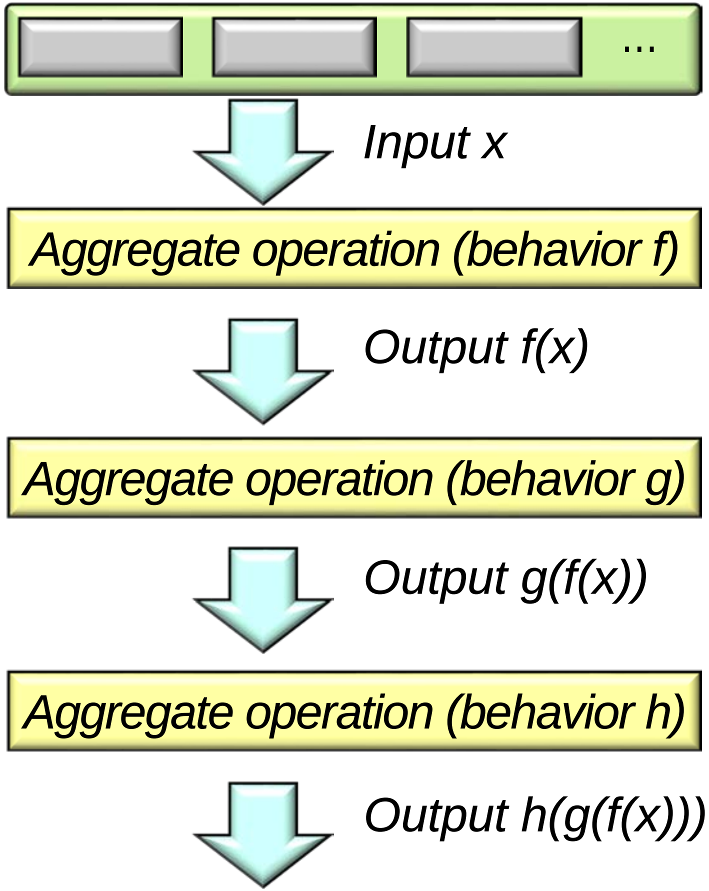
Overview of Java 8 Streams

- 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 (& only one) terminal operation

Overview of Java 8 Streams

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



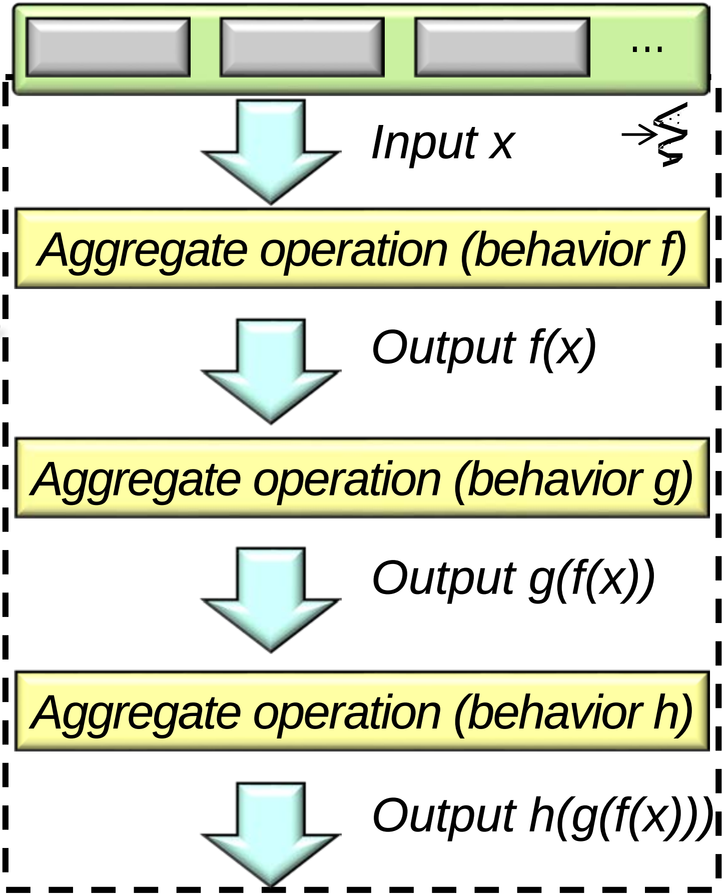
...
Input x
Aggregate operation (behavior f)
Output f(x)
Aggregate operation (behavior g)
Output g(f(x))
Aggregate operation (behavior h)
Output h(g(f(x)))

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

Overview of Java 8 Streams

- Each aggregate operation in a stream runs its behavior sequentially by default
  - i.e., one at a time in a single thread

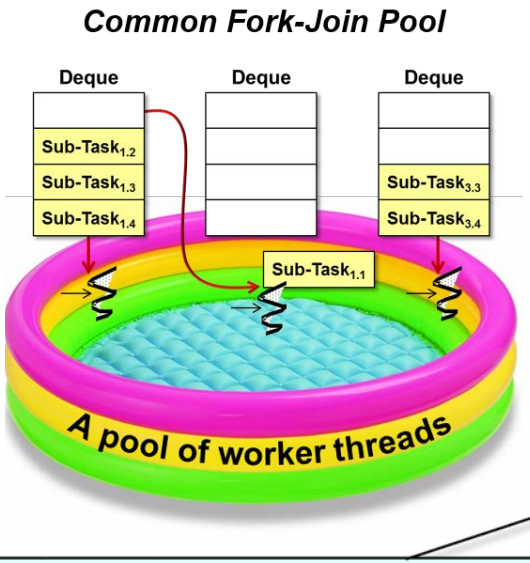
We'll cover sequential streams first


...
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)))

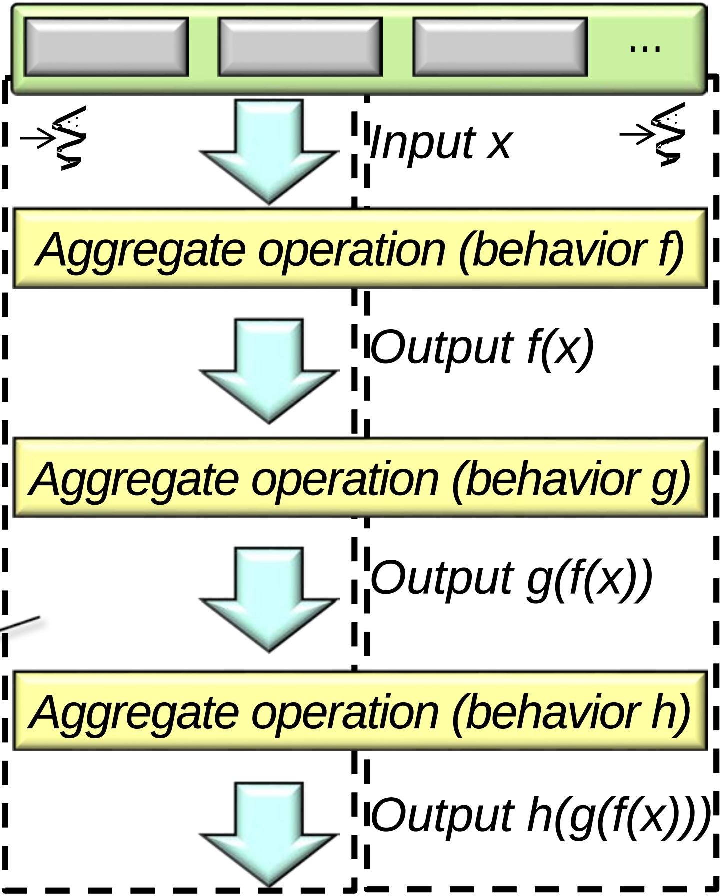
See docs.oracle.com/javase/tutorial/collections/streams

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


...
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)))

See docs.oracle.com/javase/tutorial/collections/streams/parallelism.html

End of Overview of Java 8 Streams (Part 1)