Java Streams:

Applying Streams in Practice

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

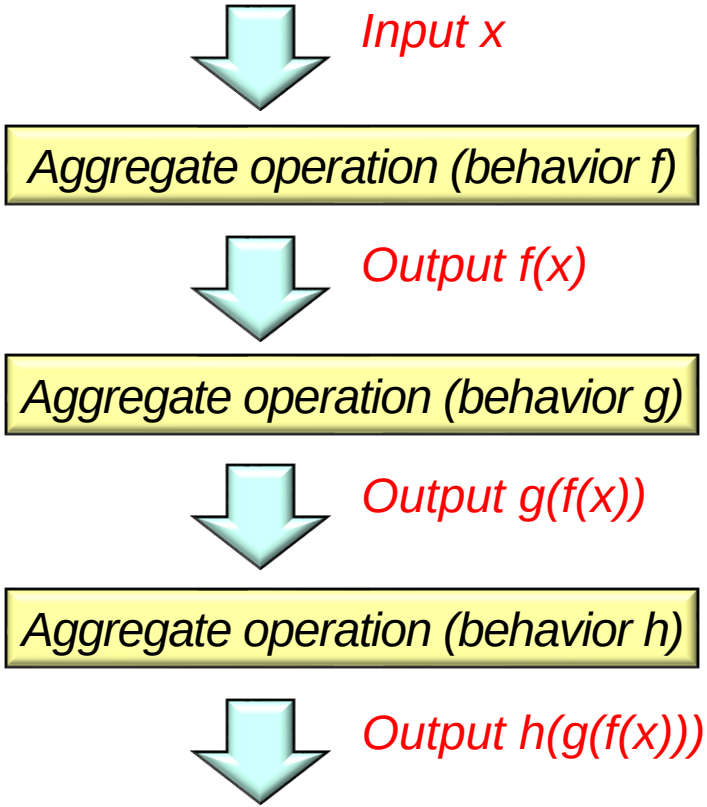
Learning Objectives in this Part of the Lesson

- Understand the structure & functionality of Java streams, e.g.,
  - Fundamentals of streams
  - Benefits of streams
  - Operations that create a stream
  - Aggregate operations in a stream
  - Applying streams in practice

Applying Streams in Practice

Applying Streams in Practice

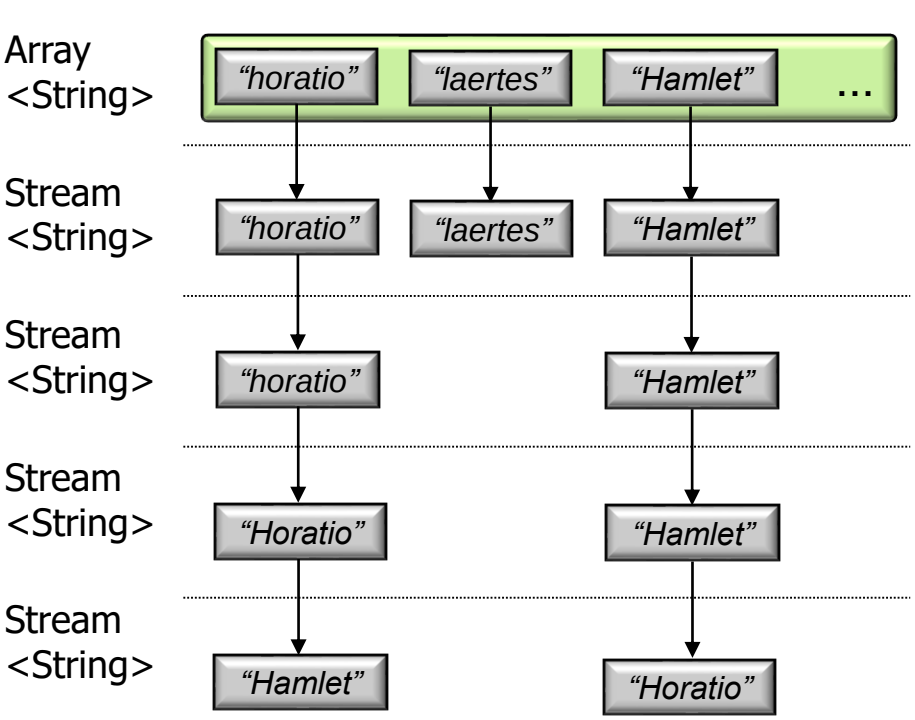
- Streams enhance flexibility by forming a "processing pipeline" that composes multiple aggregate operations together

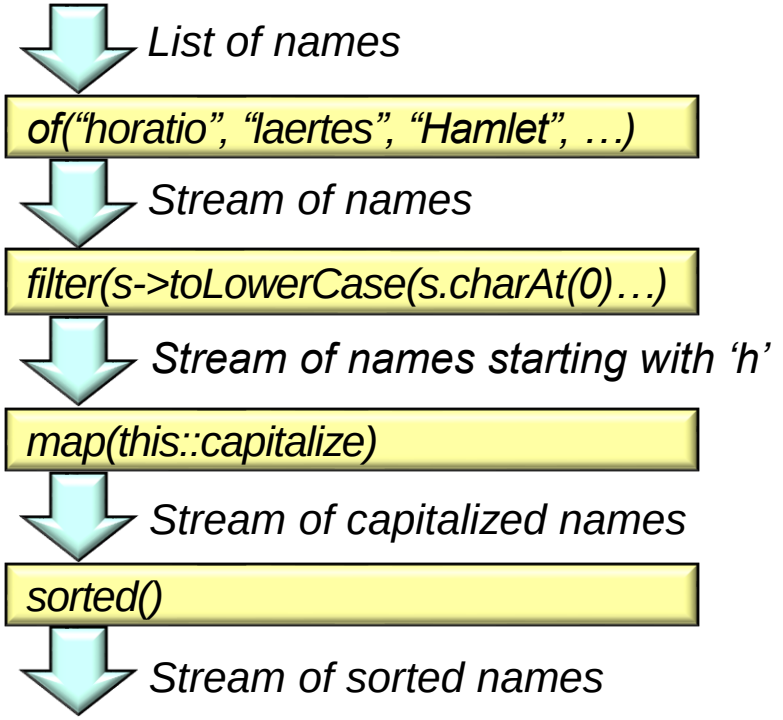


See en.wikipedia.org/wiki/Pipeline_(software)

Applying Streams in Practice

- Streams enhance flexibility by forming a "processing pipeline" that composes multiple aggregate operations together

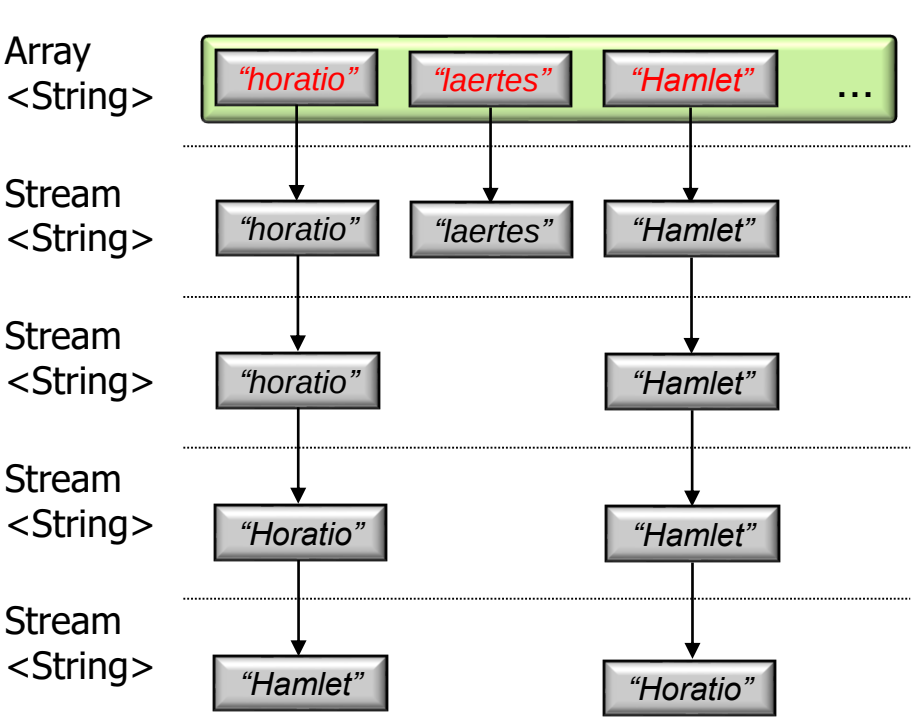


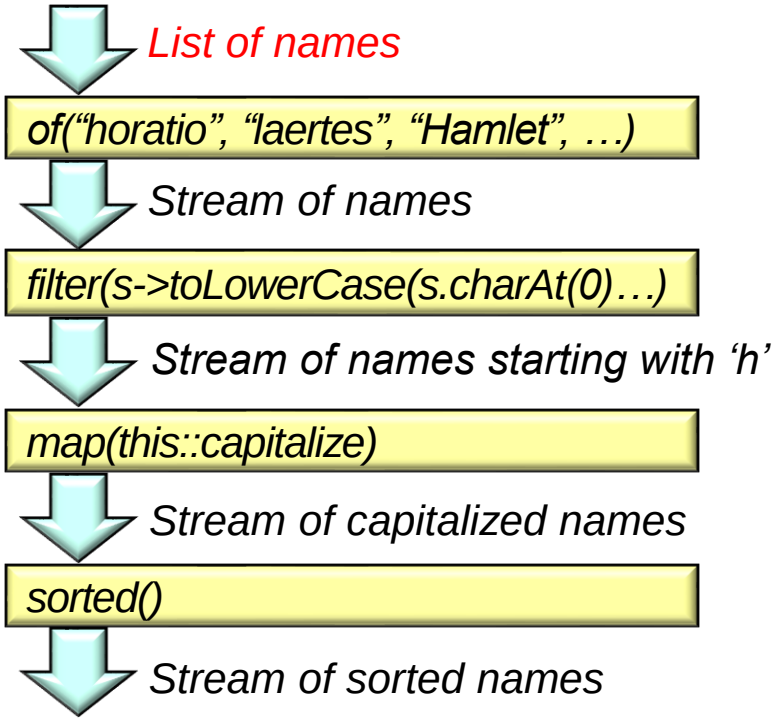


Each aggregate operation in the pipeline can filter and/or transform the stream

Applying Streams in Practice

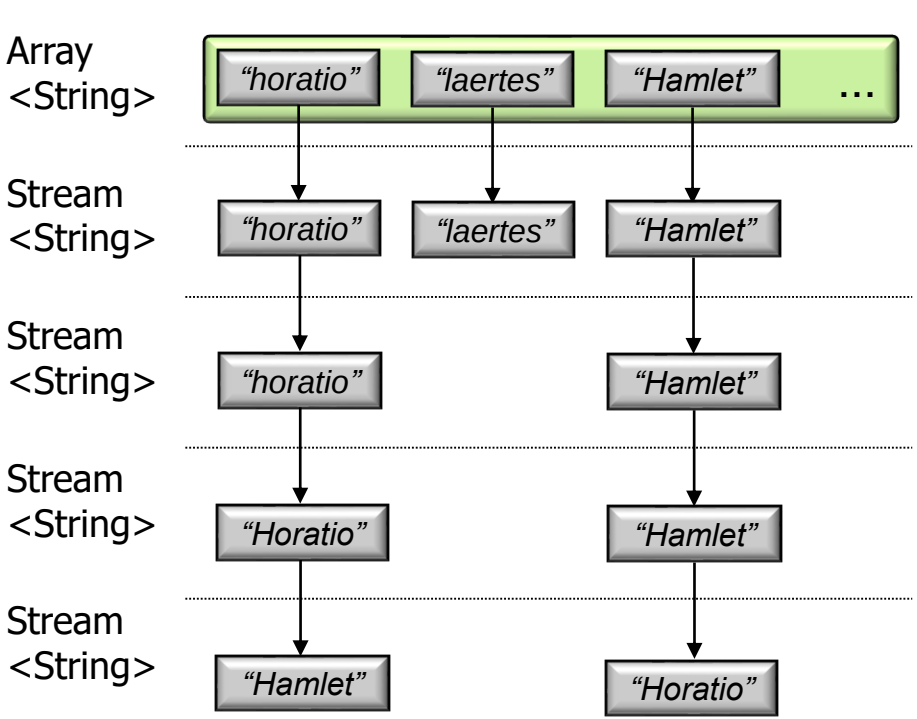
- Streams enhance flexibility by forming a "processing pipeline" that composes multiple aggregate operations together

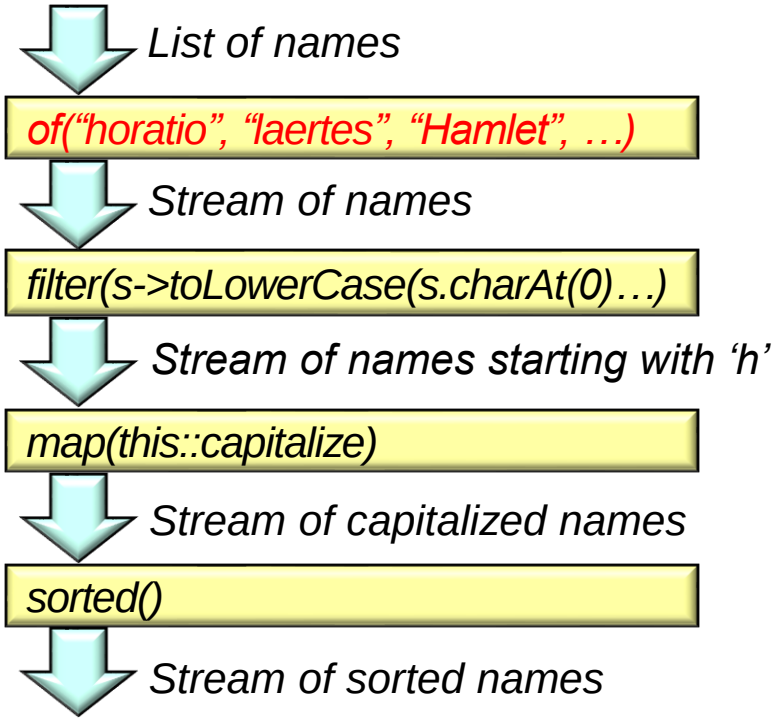




Applying Streams in Practice

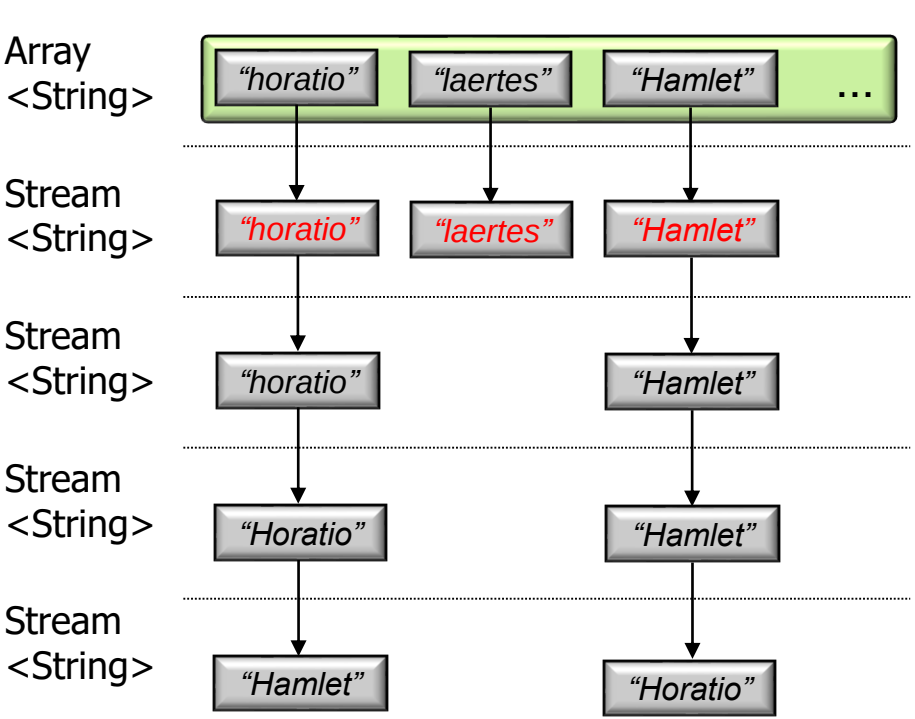
- Streams enhance flexibility by forming a "processing pipeline" that composes multiple aggregate operations together

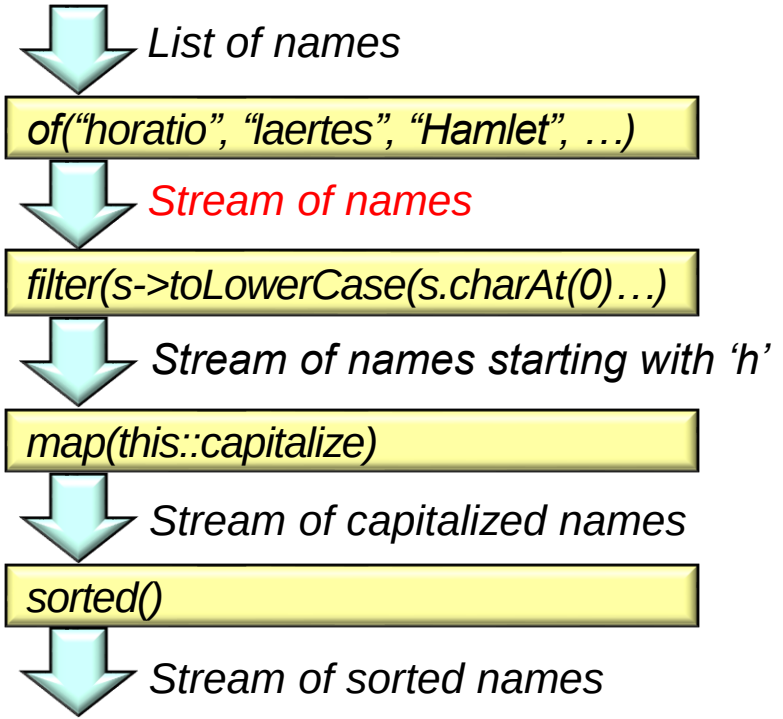




Applying Streams in Practice

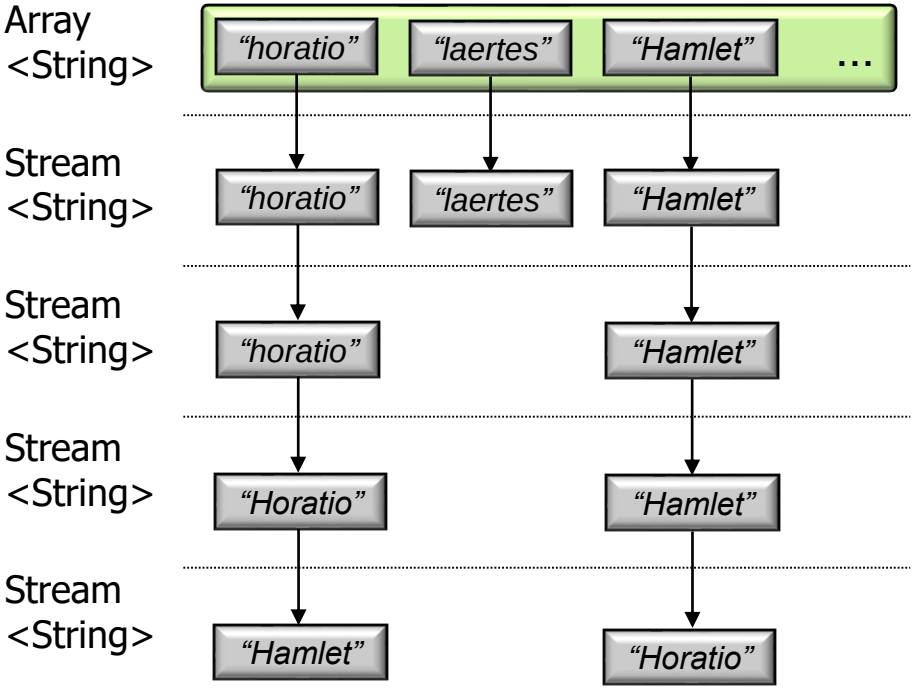
- Streams enhance flexibility by forming a "processing pipeline" that composes multiple aggregate operations together

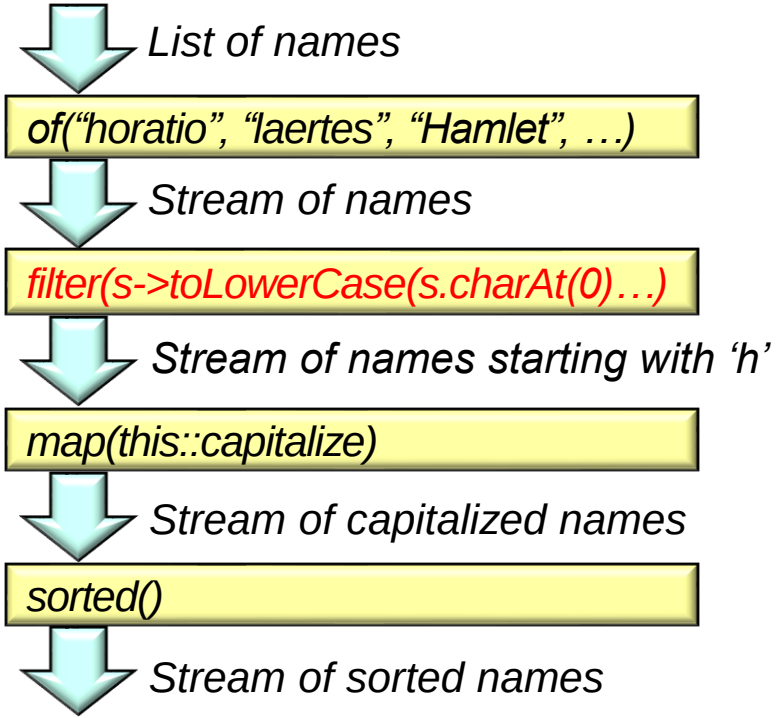




Applying Streams in Practice

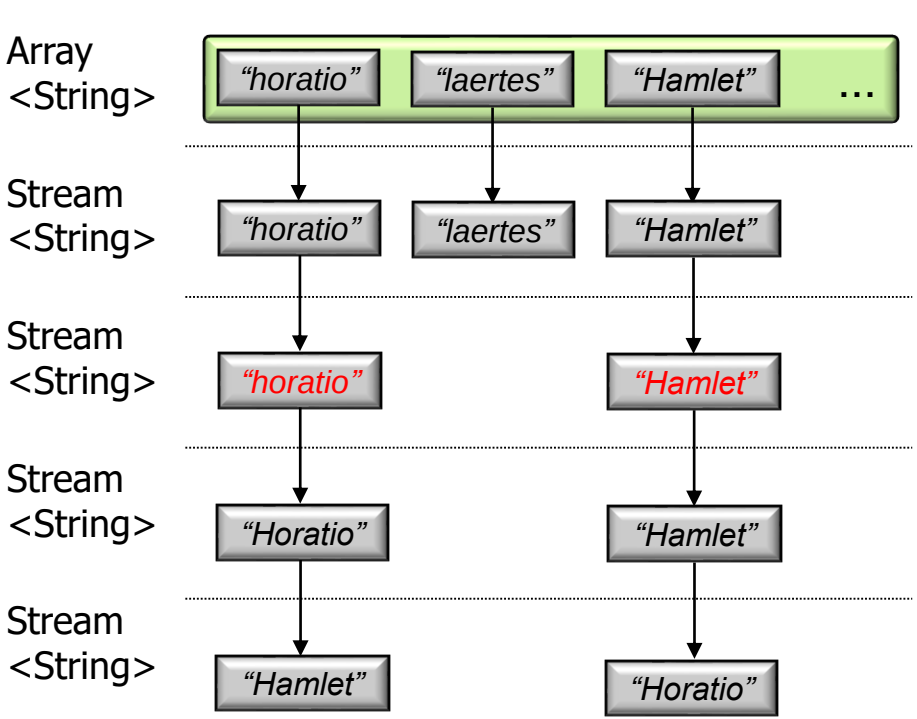
- Streams enhance flexibility by forming a "processing pipeline" that composes multiple aggregate operations together

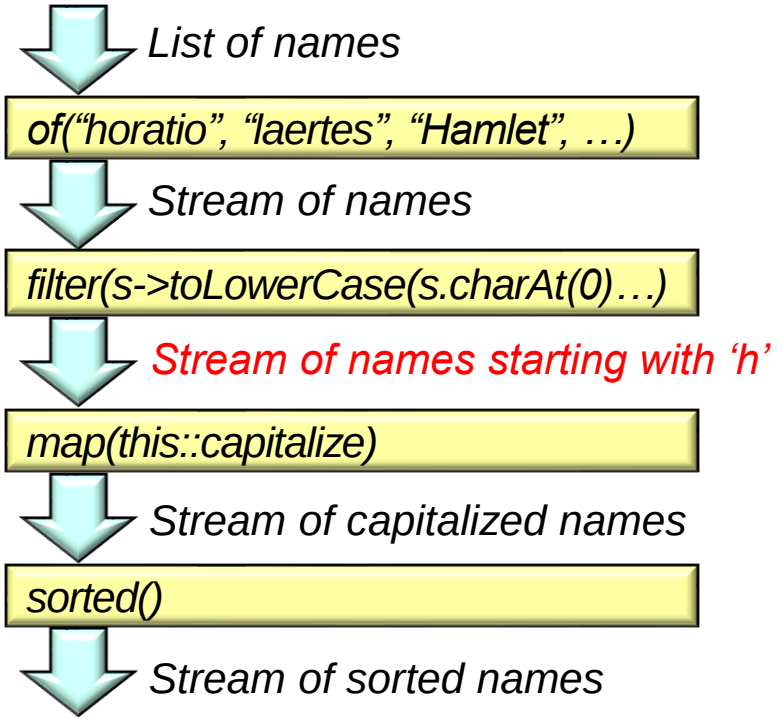




Applying Streams in Practice

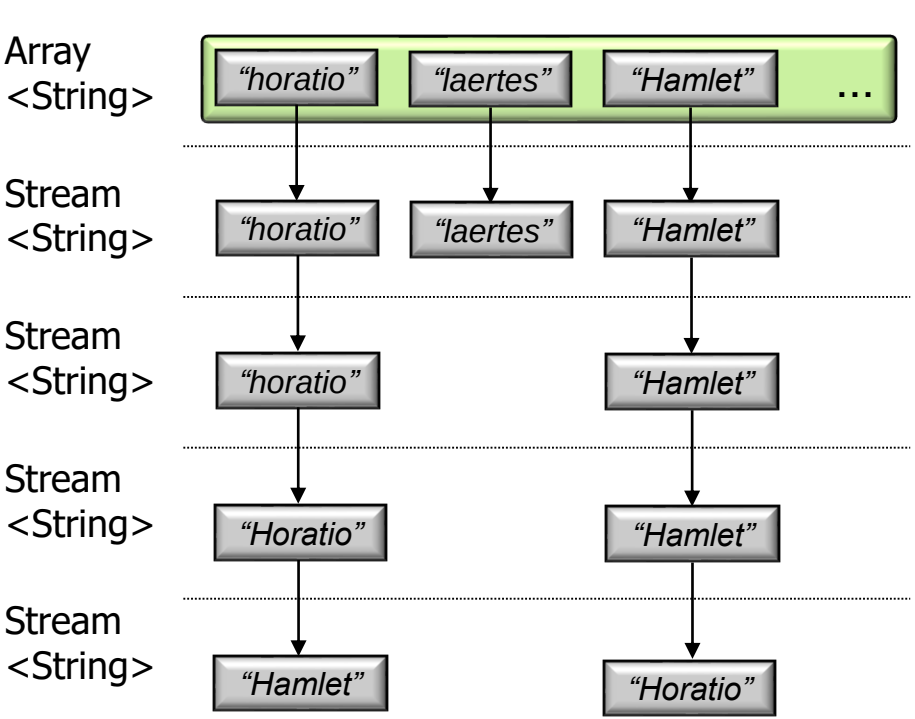
- Streams enhance flexibility by forming a "processing pipeline" that composes multiple aggregate operations together

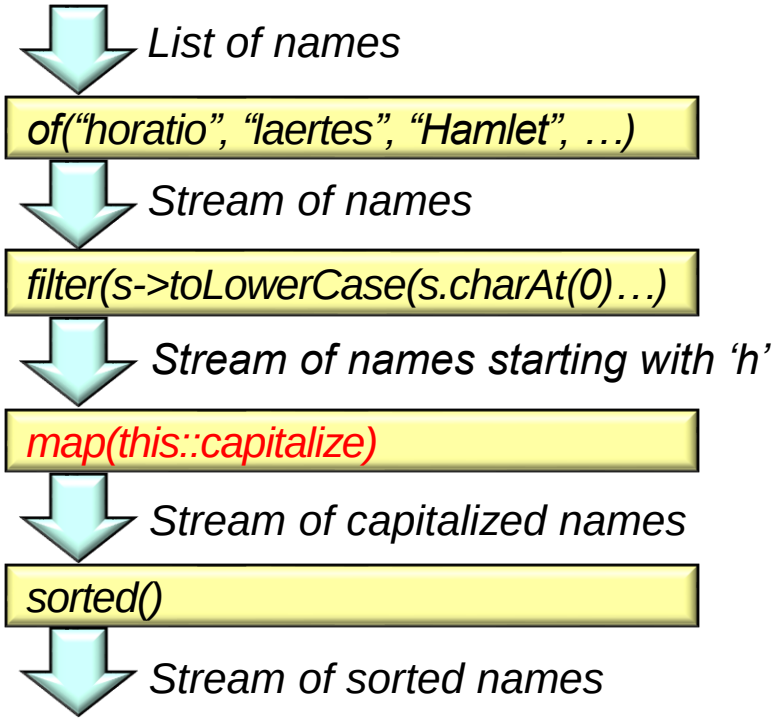




Applying Streams in Practice

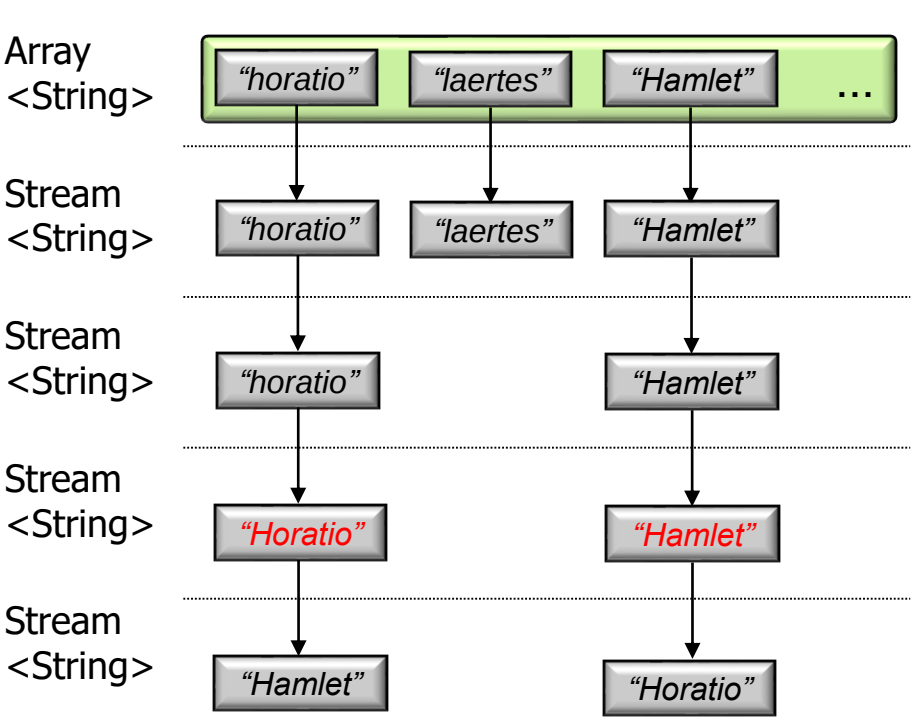
- Streams enhance flexibility by forming a "processing pipeline" that composes multiple aggregate operations together

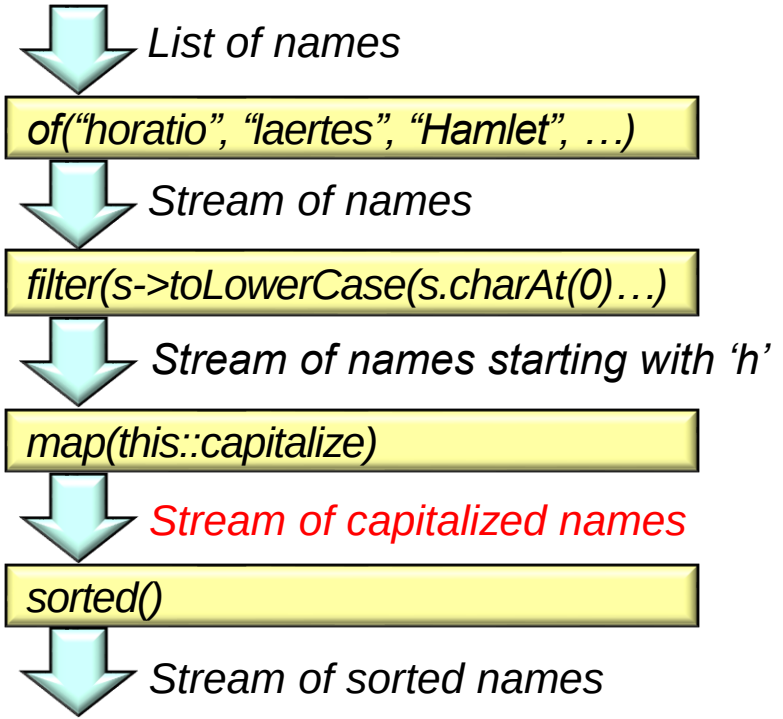




Applying Streams in Practice

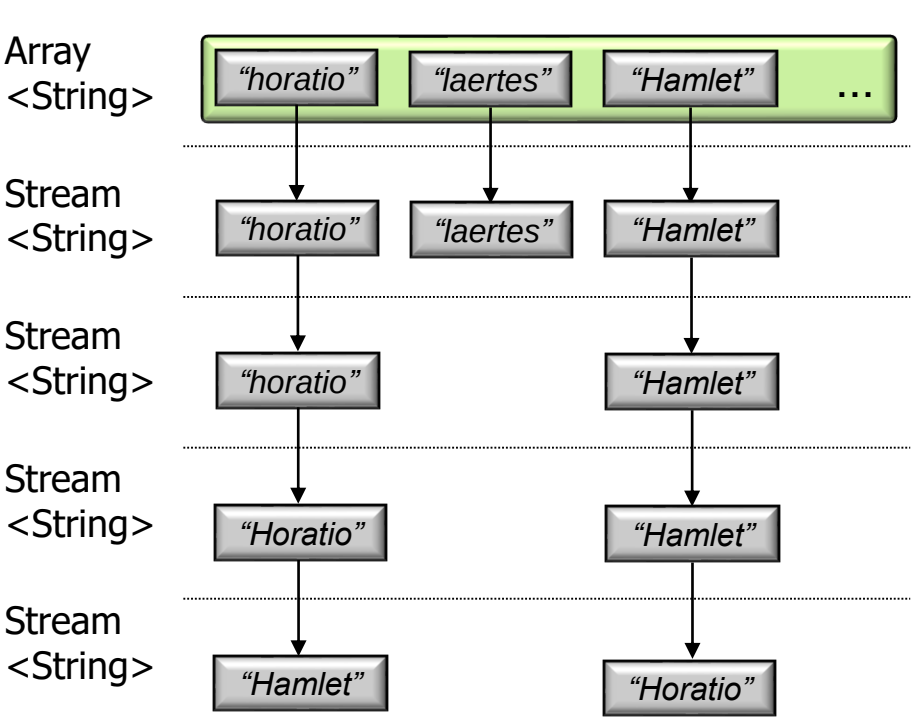
- Streams enhance flexibility by forming a "processing pipeline" that composes multiple aggregate operations together

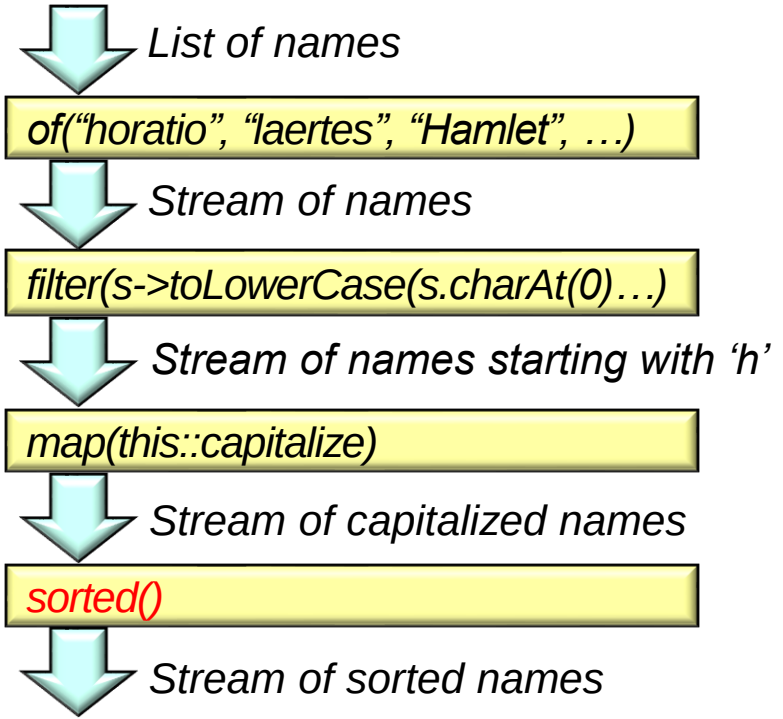




Applying Streams in Practice

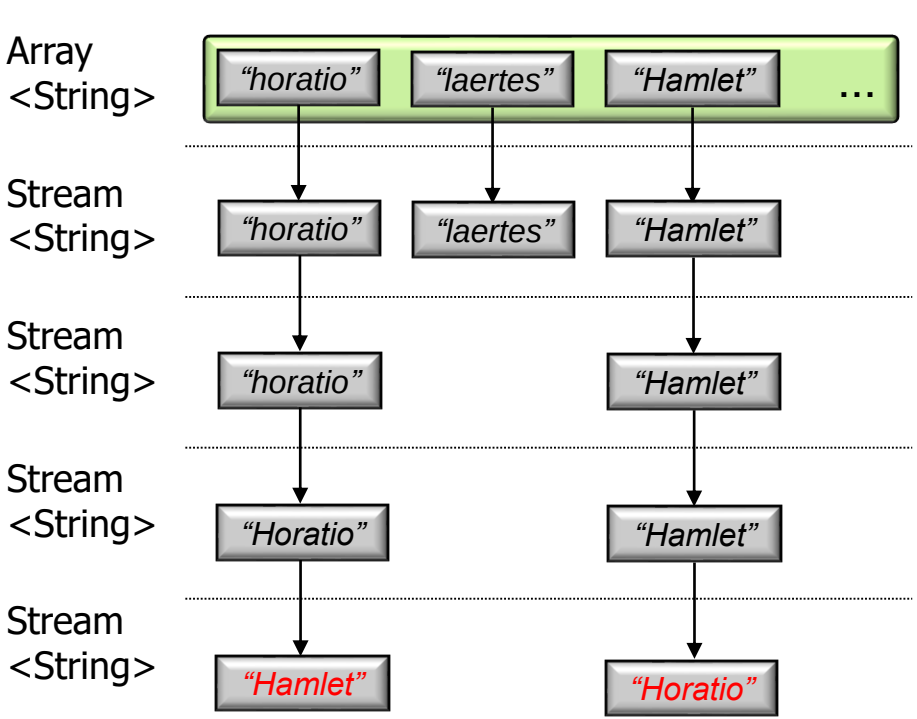
- Streams enhance flexibility by forming a "processing pipeline" that composes multiple aggregate operations together

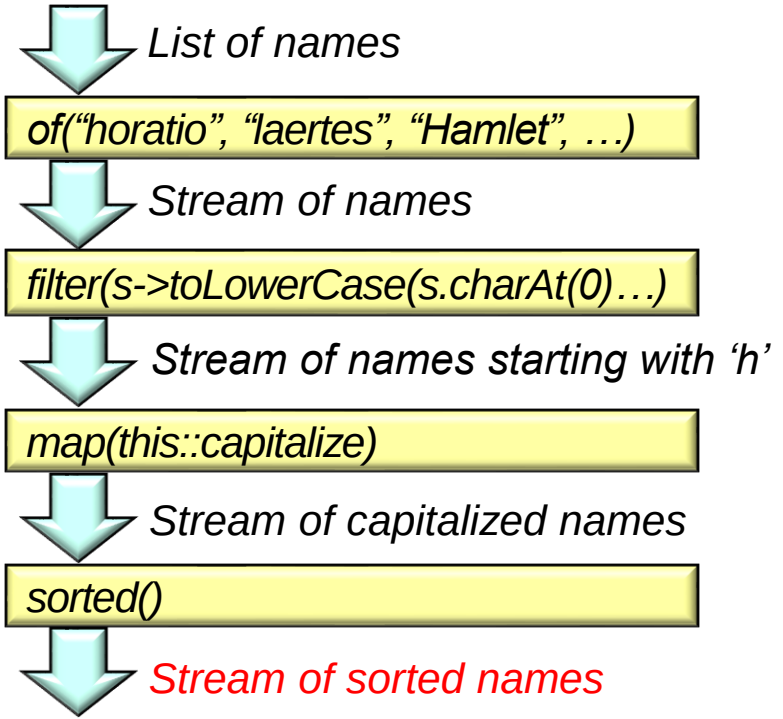




Applying Streams in Practice

- Streams enhance flexibility by forming a "processing pipeline" that composes multiple aggregate operations together





Applying Streams in Practice

- Every stream works very similarly

Applying Streams in Practice

- 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

Applying Streams in Practice

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

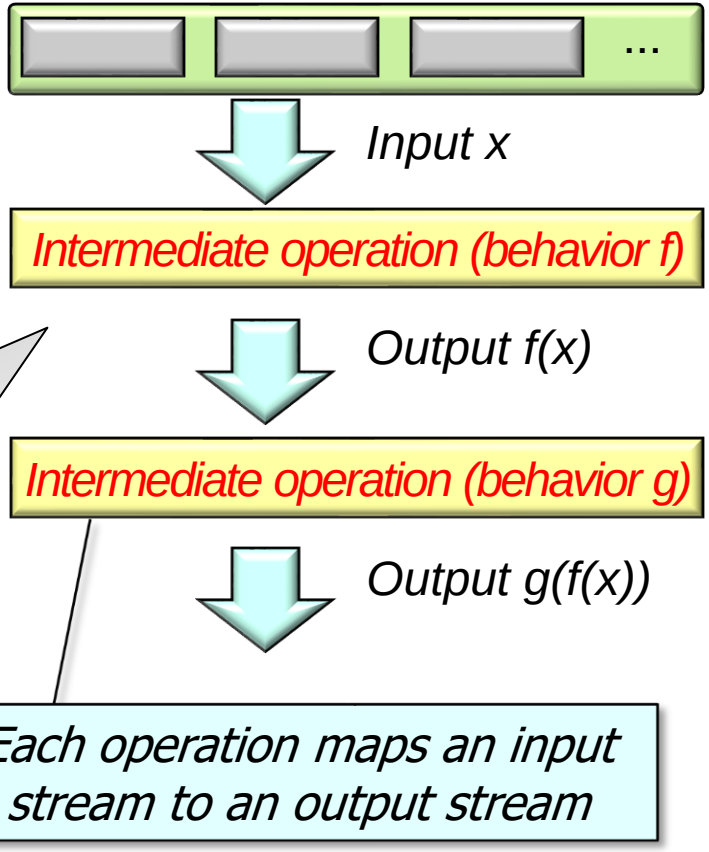

```
List<String> characters =
  Arrays.asList("horatio",
                "laertes",
                "Hamlet", ...);

characters
  .stream()
  ...
```

e.g., a Java array, collection, generator function, or input channel

Applying Streams in Practice

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

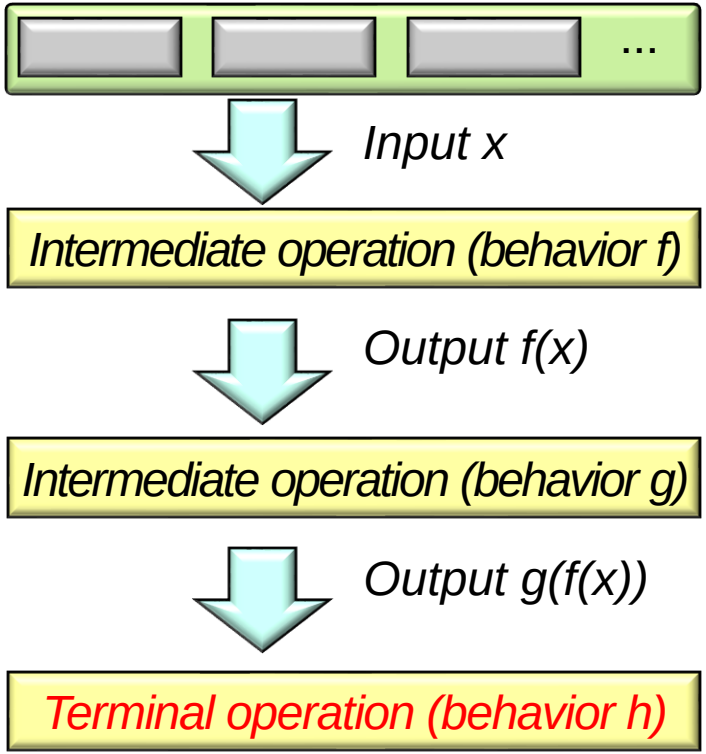



```
Stream
   .of("horatio", "laertes",
       "Hamlet", ...)
   .filter(s -> toLowerCase
               (s.charAt(0)) == 'h')
   .map(this::capitalize)
   .sorted()
   ...
```

Examples of intermediate operations include filter(), map(), & sorted()

Applying Streams in Practice

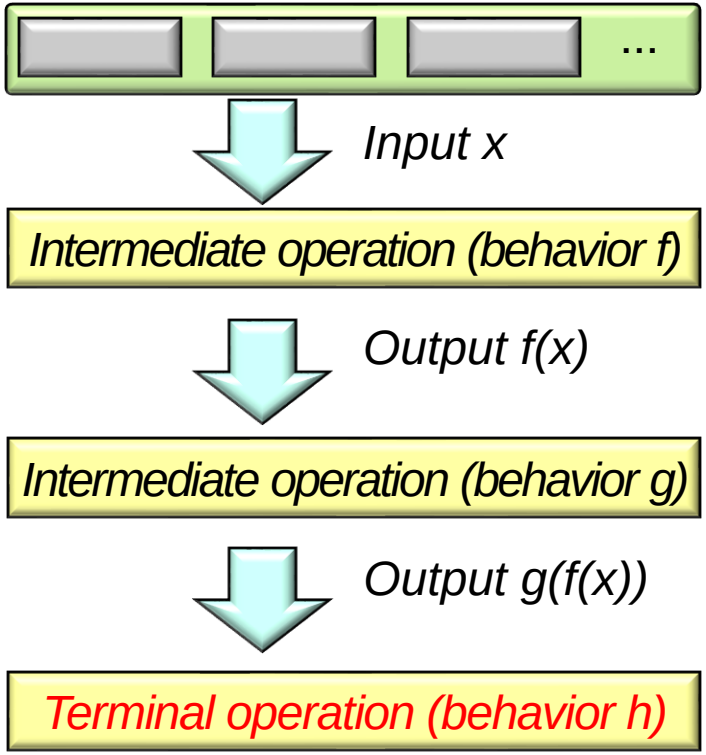
- Every stream works very similarly
  - Starts with a source of data
  - Processes data thru a pipeline of intermediate operations
  - Finishes w/a terminal operation that yields a non-stream result




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

Applying Streams in Practice

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




```
...
.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

Applying Streams in Practice

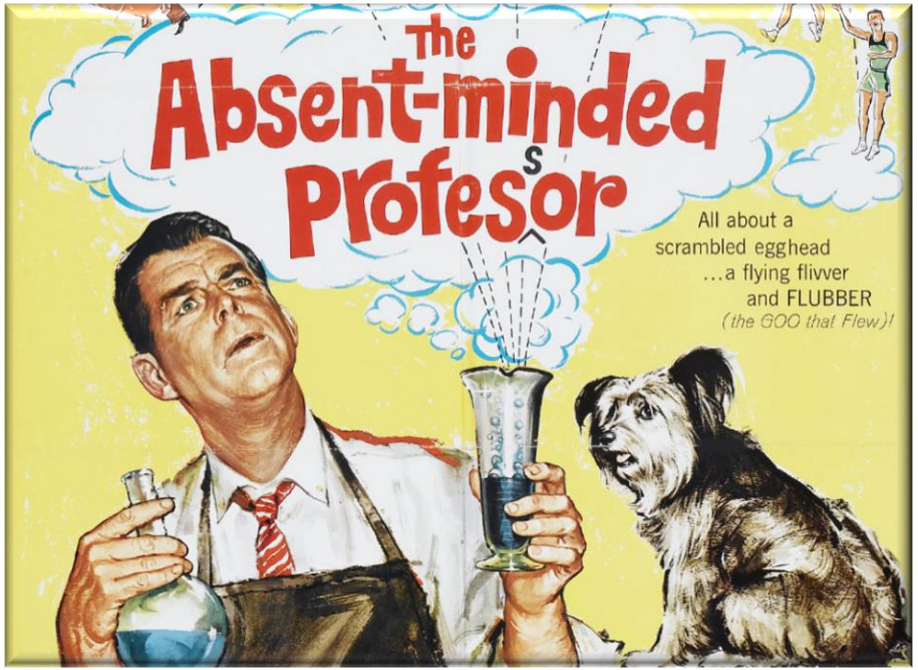
- Every stream works very similarly
  - Starts with a source of data
  - Processes data thru a pipeline of intermediate operations
  - Finishes w/a terminal operation that yields a non-stream result

Each stream *must* have one (& only one) terminal operation

Applying Streams in Practice

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

```
Stream
   .of("horatio",
       "laertes",
       "Hamlet", ...)
   .filter(s -> ...)
   .map(this::capitalize)
   .sorted();
```

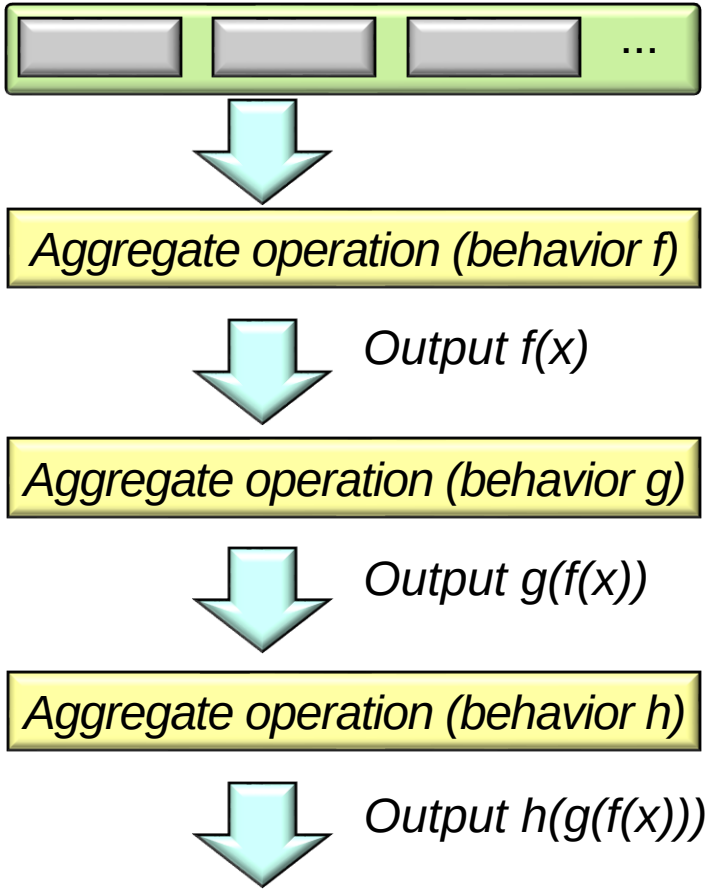


A common "beginner mistake" is to forget the terminal operation

See blog.jooq.org/2014/06/13/java-8-friday-10-subtle-mistakes-when-using-the-streams-api

Applying Streams in Practice

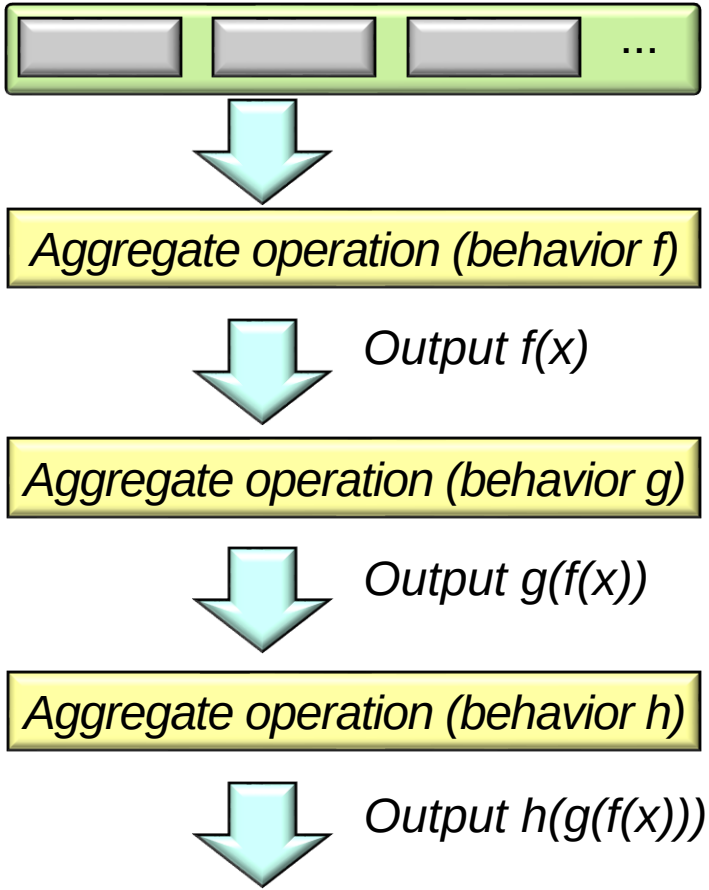
- A stream holds no non-transient storage



Apps are thus responsible for persisting any data that must be preserved

Applying Streams in Practice

- A stream can only be traversed once



Applying Streams in Practice

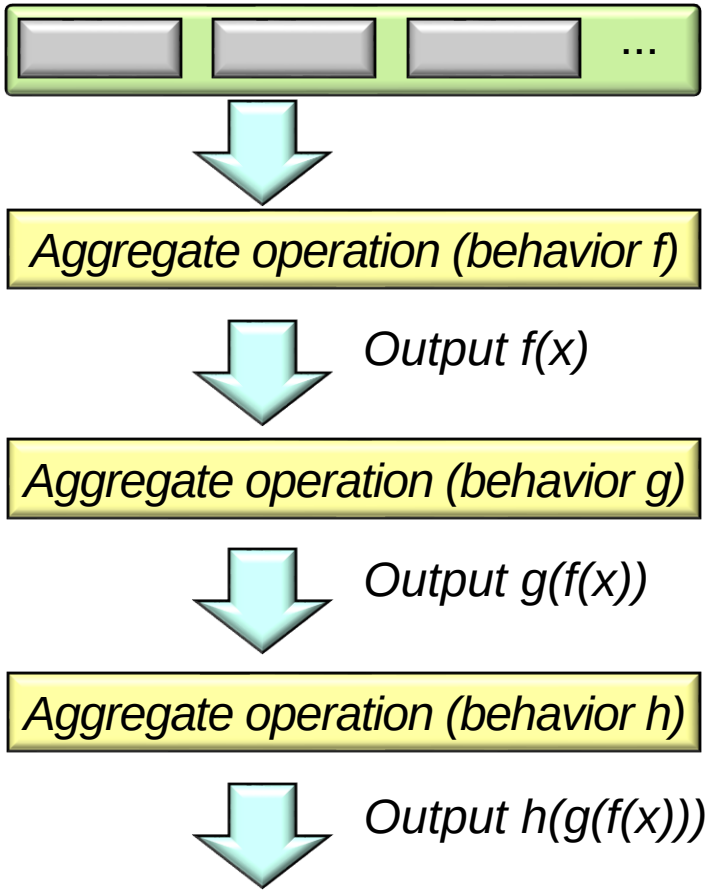
- A stream can only be traversed once

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

Stream<String> s = characters
   .stream()
   .filter(s -> ...)
   .map(this::capitalize)
   .sorted();

s.forEach(System.out::println);
s.forEach(System.out::println);
```

Duplicate calls are invalid!



See blog.jooq.org/2014/06/13/java-8-friday-10-subtle-mistakes-when-using-the-streams-api

Applying Streams in Practice

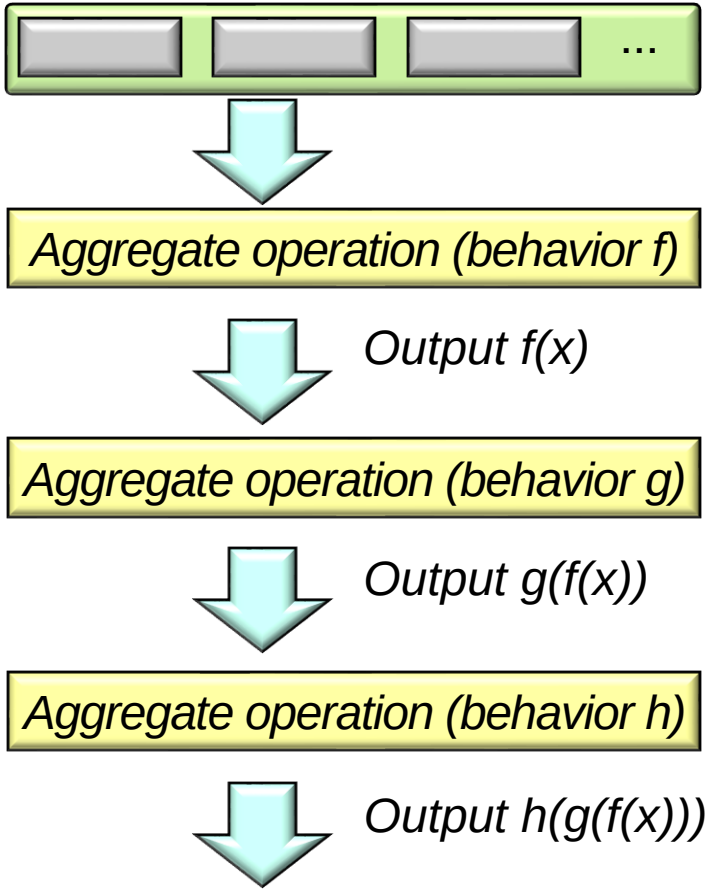
- A stream can only be traversed once

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

Stream<String> s = characters
   .stream()
   .filter(s -> ...)
   .map(this::capitalize)
   .sorted();

s.forEach(System.out::println);
s.forEach(System.out::println);
```

Throws java.lang.IllegalStateException

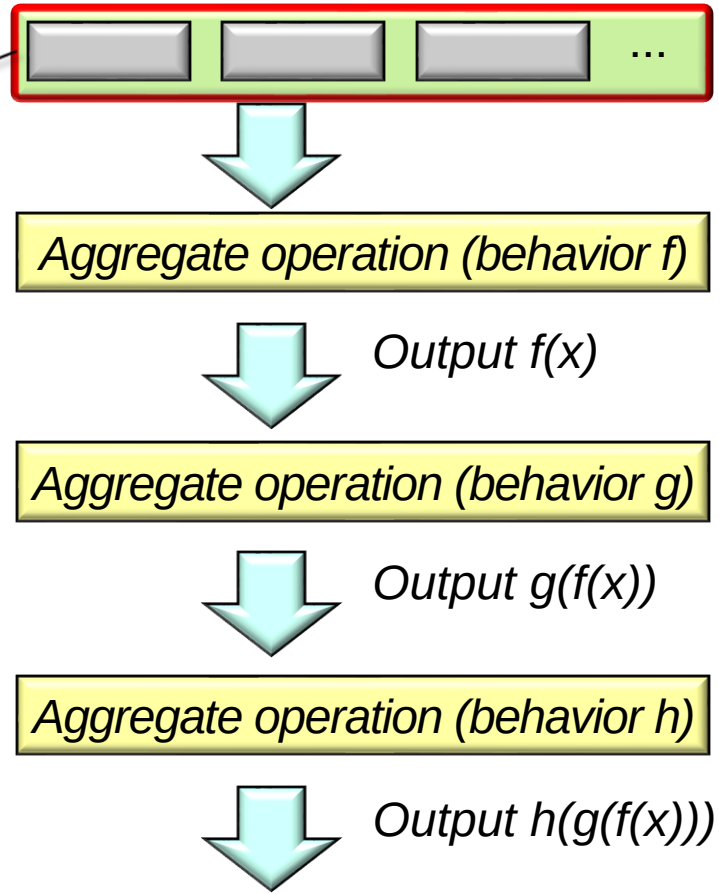


See docs.oracle.com/javase/8/docs/api/java/lang/IllegalStateException.html

Applying Streams in Practice

- A stream can only be traversed once

To traverse a stream again you need to get a new stream from the data source



End of Java Streams: Appying Streams in Practice