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
Streams enhance flexibility by forming a “processing pipeline” that composes 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)](en.wikipedia.org/wiki/Pipeline_(software))
Streams enhance flexibility by forming a “processing pipeline” that composes 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”

Each aggregate operation in the pipeline can filter and/or transform the stream:
- **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()
Applying Streams in Practice

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

**Applying Streams in Practice**

- Array of names
  - 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
Applying Streams in Practice

- Streams enhance flexibility by forming a “processing pipeline” that composes 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
```
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.

```
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
```
Applying Streams in Practice

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

```
Streams enhance flexibility by forming a “processing pipeline” that composes 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

Stream of names starting with "h"

Stream of capitalized names

Stream of sorted names
```
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.

Array

```
["horatio", "laertes", "Hamlet", ...]
```

-> List of names
of(“horatio”, “laertes”, “Hamlet”, …)

-> Stream of names

```
["horatio", "laertes", "Hamlet"]
```

-> Stream of names starting with ‘h’

```
["horatio", "Hamlet"]
```

-> Stream of capitalized names

```
["Horatio", "Hamlet"]
```

-> Stream of sorted names

```
["Hamlet", "Horatio"]
```
Applying Streams in Practice

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

```
Stream<String>
  └── Stream<String>
      └── Stream<String>
          └── Stream<String>
```

1. **Array**
   ```
   ["horatio", "laertes", "Hamlet", ...]
   ```

2. **Stream**
   ```
   Stream<String>
   " horatio "
   " laertes "
   " Hamlet "
   ```

3. **Stream**
   ```
   Stream<String>
   " horatio "
   " Hamlet "
   ```

4. **Stream**
   ```
   Stream<String>
   " Horatio "
   " Hamlet "
   ```

5. **Stream**
   ```
   Stream<String>
   " Hamlet "
   " Horatio "
   ```

- **List of names**
  ```
  of("horatio", "laertes", "Hamlet", ...)
  ```

- **Stream of names**
  ```
  Stream of names
  ```

- **Stream of names starting with ‘h’**
  ```
  filter(s->toLowerCase(s.charAt(0)...) =>
  ```

- **Stream of capitalized names**
  ```
  map(this::capitalize) =>
  ```

- **Stream of sorted names**
  ```
  sorted() =>
  ```
Applying Streams in Practice

- Every stream works very similarly
Every stream works very similarly
Starts with a source of data
e.g., a Java array, collection, generator function, or input channel
Every stream works very similarly

Starts with a source of data

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

```java
List<String> characters = Arrays.asList("horatio", "laertes", "Hamlet", ...);
characters.stream() ...
```
Every stream works very similarly
- Starts with a source of data
- Processes data thru a pipeline of intermediate operations

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

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

```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.
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
• 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

Output $h(g(f(x)))$
Applying Streams in Practice

- A stream can only be traversed once

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

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

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

Duplicate calls are invalid!

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

See blog.jooq.org/2014/06/13/java-8-friday-10-subtle-mistakes-when-using-the-streams-api
A stream can only be traversed once

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

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

Throws `java.lang.IllegalStateException`

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

See [docs.oracle.com/javase/8/docs/api/java/lang/IllegalStateException.html](http://docs.oracle.com/javase/8/docs/api/java/lang/IllegalStateException.html)
• A stream can only be traversed once

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

Aggregate operation (behavior f)

Output f(x)

Aggregate operation (behavior g)

Output g(f(x))

Aggregate operation (behavior h)

Output h(g(f(x)))
End of Java Streams: Applying Streams in Practice