Understand Java Streams: Overview

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 Java streams structure & functionality

Learning Objectives in this Part of the Lesson

- Understand Java streams structure & functionality, e.g.
  - Fundamentals of streams

Learning Objectives in this Part of the Lesson

- Understand Java streams structure & functionality, e.g.
  - Fundamentals of streams
  - Three streams phases

Overview of Java Streams

Overview of Java Streams

- Java streams are a framework first introduced into the Java class library in Java 8

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 Streams

- A stream is a pipeline of aggregate operations that process a sequence of elements (aka, "values" or "data")

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

Overview of Java Streams

- A stream is a pipeline of aggregate operations that process a sequence of elements (aka, "values" or "data")

An aggregate operation is a higher-order function that applies a "behavior" param to every element in a stream.

See en.wikipedia.org/wiki/Higher-order_function

Overview of Java Streams

- A stream is a pipeline of aggregate operations that process a sequence of elements (aka, "values" or "data")

Behavior parameterization simplifies coping with changing requirements.

See blog.indrek.io/articles/java-8-behavior-parameterization

Overview of Java Streams

- A stream is a pipeline of aggregate operations that process a sequence of elements (aka, "values" or "data")

Input x

Aggregate operation (behavior f)

Output f(x)

Aggregate operation (behavior g)

Output g(f(x))

Aggregate operation (behavior h)

A stream is conceptually unbounded, though it's often bounded by practical constraints.

Overview of Java Streams

- We use this stream as a case study example throughout this introduction

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

Print each character in Hamlet that starts with 'H' or 'h' in consistently capitalized & sorted order.

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

Overview of Java Streams

- Java streams have evolved a bit over time

Overview of Java Streams

- Java streams have evolved a bit over time
  - e.g., Later versions of Java added some new operations

See www.baeldung.com/java-9-stream-api & blog.codefx.org/java/teeing-collector

Overview of Java Streams

- Java 9 also added a new API that implements the reactive streams specification

See www.reactive-streams.org

Overview of Java Streams

- Java 9 also added a new API that implements the reactive streams specification
  - Reactive streams is covered later in this course

Project Reactor

See upcoming lessons on RxJava & Project Reactor

Overview of Stream Phases

Overview of Stream Phases

- Streams usually have three phases

See www.jstatsoft.org/article/view/v040i01/v40i01.pdf

Overview of Stream Phases

- Streams usually have three phases, i.e.
  - **Split** – start with a source of data

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

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

Overview of Stream Phases

- Streams usually have three phases, i.e.
  - **Split** – start with a source of data

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

characters
  .stream()
  ...
```

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

Overview of Stream Phases

- Streams usually have three phases, i.e.
  - **Split** – start with a source of data
  - **Apply** – process data through a pipeline of intermediate operations

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

Each operation maps an input stream to an output stream.

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

Overview of Stream Phases

- Streams usually have three phases, i.e.
  - **Split** – start with a source of data
  - **Apply** – process data through a pipeline of intermediate operations
    - Processing often involves transforming

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

Each operation maps an input stream to an output stream.

Overview of Stream Phases

- Streams usually have three phases, i.e.
  - **Split** – start with a source of data
  - **Apply** – process data through a pipeline of intermediate operations
    - Processing often involves transforming

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

Each operation maps an input stream to an output stream.

Some transformations are stateless & some are stateful

Overview of Stream Phases

- Streams usually have three phases, i.e.
  - **Split** – start with a source of data
  - **Apply** – process data through a pipeline of intermediate operations
    - Processing often involves transforming

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

Each operation maps an input stream to an output stream.

Some transformations are stateless & some are stateful

Overview of Stream Phases

- Streams usually have three phases, i.e.
  - **Split** – start with a source of data
  - **Apply** – process data through a pipeline of intermediate operations
  - **Combine** – finish 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);
```

Overview of Stream Phases

- Streams usually have three phases, i.e.
  - **Split** – start with a source of data
  - **Apply** – process data through a pipeline of intermediate operations
  - **Combine** – finish 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);
```

A terminal operation triggers processing of intermediate operations in a stream

Overview of Stream Phases

- Streams usually have three phases, i.e.
  - **Split** – start with a source of data
  - **Apply** – process data through a pipeline of intermediate operations
  - **Combine** – finish with a terminal operation that yields a non-stream result

A stream only runs if it has one (& only one) terminal operation

End of Understand Java Streams: Overview