Overview of Java Streams

Terminal Operations

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 terminal operations
• Terminal operations start the internal iteration of stream elements, trigger the intermediate operations, & produce some result
Learning Objectives in this Part of the Lesson

- Understand the structure & functionality of Java Streams terminal operations
- Terminal operations start the internal iteration of stream elements, trigger the intermediate operations, & produce some result

These operations also apply to both sequential & parallel streams
Learning Objectives in this Part of the Lesson

- Understand the structure & functionality of Java Streams terminal operations
- Terminal operations start the internal iteration of stream elements, trigger the intermediate operations, & produce some result

We continue to showcase the “Hamlet” program
Overview of Terminal Operations
Every stream finishes with a terminal operation that (typically) yields a non-stream result.

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex12
Overview of Common Stream Terminal Operations

- Every stream finishes with a terminal operation that (typically) yields a non-stream result, e.g.
  - No value at all
  - e.g., forEach() & forEachOrdered()

Java Stream : forEachOrdered() vs forEach()
By Arvind Rai, June 13, 2020

On this page we will provide differences between Stream.forEachOrdered() and Stream.forEach() methods. Both methods perform an action as Consumer. The difference between forEachOrdered() and forEach() methods is that forEachOrdered() will always perform given action in encounter order of elements in stream whereas forEach() method is non-deterministic. In parallel stream forEach() method may not necessarily respect the order whereas forEachOrdered() will always respect the order. In sequential stream both methods respect the order. So we should use forEachOrdered() method, if we want action to be perform in encounter order in every case whether the stream is sequential or parallel. If the stream is sequential, we can use any method to respect order. But if stream can be parallel too, then we should use forEachOrdered() method to respect the order.

These terminal operations both "run-to-completion"

See www.concretepage.com/java/java-8/java-stream-foreachordered-vs-foreach
Overview of Common Stream Terminal Operations

- Every stream finishes with a terminal operation that (typically) yields a non-stream result, e.g.
  - No value at all
    - e.g., forEach() & forEachOrdered()

```java
forEach() & forEachOrdered() only have side-effects!
```
Overview of Common Stream Terminal Operations

- Every stream finishes with a terminal operation that (typically) yields a non-stream result, e.g.
  - No value at all
  - e.g., forEach() & forEachOrdered()

```java
Stream.of("horatio", "laertes", "Hamlet", ...)
  .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.
Overview of Common Stream Terminal Operations

- Every stream finishes with a terminal operation that (typically) yields a non-stream result, e.g.
  - No value at all
  - The result of a reduction operation
    - e.g., collect() & reduce()

See docs.oracle.com/javase/tutorial/collections/streams/reduction.html
Every stream finishes with a terminal operation that (typically) yields a non-stream result, e.g.

- No value at all
- The result of a reduction operation
  - e.g., collect() & reduce()

`collect() & reduce()` terminal operations work seamlessly with parallel streams.

See [docs.oracle.com/javase/tutorial/collections/streams/parallelism.html](docs.oracle.com/javase/tutorial/collections/streams/parallelism.html)
Overview of Common Stream Terminal Operations

- Every stream finishes with a terminal operation that (typically) yields a non-stream result, e.g.
  - No value at all
  - The result of a reduction operation
  - An Optional or boolean value
    - e.g., `findAny()`, `findFirst()`, `noneMatch()`, `allMatch()`, etc.

```java
List<String> countries = Arrays.asList("france", "india", "china", "usa");
	print(countries.stream()
		.filter(country -> country.contains("i"))
		.findFirst().get());
	print(countries.stream()
		.filter(country -> country.contains("i"))
		.findAny().get());
	print(countries.stream()
		.noneMatch(country -> country.contains("z")));
```

See [dzone.com/articlescollectors-part-1-%E2%80%93-reductions](dzone.com/articlescollectors-part-1-%E2%80%93-reductions)
Every stream finishes with a terminal operation that (typically) yields a non-stream result, e.g.

- No value at all
- The result of a reduction operation
- An Optional or boolean value
  - e.g., `findAny()`, `findFirst()`, `noneMatch()`, `allMatch()`, etc.

Overview of Common Stream Terminal Operations

```java
List<String> countries = Arrays.asList("france", "india", "china", "usa");

print(countries.stream()
    .filter(country -> country.contains("i"))
    .findFirst().get());

print(countries.stream()
    .filter(country -> country.contains("i"))
    .findAny().get());

print(countries.stream()
    .noneMatch(country -> country.contains("z")));
```

These terminal operations are "short-circuiting"
Overview of the collect() Terminal Operation

- A terminal operation also triggers all the ("lazy") intermediate operation processing

```
stream()

Input x

Intermediate operation (behavior f)

Output f(x)

Intermediate operation (behavior g)

Output g(f(x))

Terminal operation (behavior h)
```
End of Overview of Java Streams Terminal Operations