# Java Parallel Streams Internals: Order of Results (Part 3)

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 parallel stream internals, e.g.
  - Know what can change & what can't
    - Splitting, combining, & pooling mechanisms
    - Order of processing
    - Order of results
      - Overview
      - Collections that affect results order
      - Operations that affect results order

# Learning Objectives in this Part of the Lesson

- Understand parallel stream internals, e.g.
  - Know what can change & what can't
    - Splitting, combining, & pooling mechanisms
    - Order of processing
    - Order of results
      - Overview
      - Collections that affect results order
      - Operations that affect results order

```
List<Integer> list =
  Arrays.asList(1, 2, ...);

Integer[] doubledList = list
   .parallelStream()
   .distinct()
   .filter(x -> x % 2 == 0)
   .map(x -> x * 2)
   .limit(sOutputLimit)
   .toArray(Integer[]::new);
```

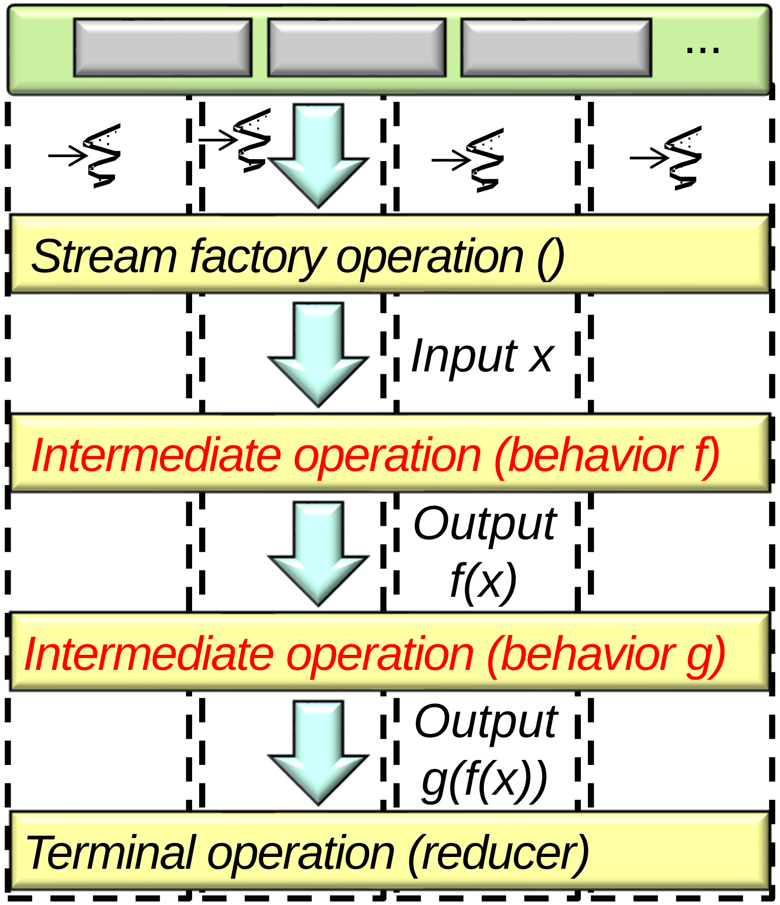
*Multiple examples are analyzed in detail*

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex21

## Operations that Affect Results Order

# Operations that Affect Results Order

- Certain intermediate operations affect ordering behavior


...
Stream factory operation ()
Input x
Intermediate operation (behavior f)
Output f(x)
Intermediate operation (behavior g)
Output g(f(x))
Terminal operation (reducer)

## Operations that Affect Results Order

- Certain intermediate operations affect ordering behavior
  - e.g., sorted(), unordered(), skip(), & limit()

```
List<Integer> list = Arrays
  .asList(2, 3, 1, 4, 2);


Integer[] doubledList = list
    .parallelStream()
    .distinct()
    .filter(x -> x % 2 == 0)
    .map(x -> x * 2)
    .limit(sOutputLimit)
    .toArray(Integer[]::new);
```

See developer.ibm.com/articles/j-java-streams-3-brian-goetz

## Operations that Affect Results Order

- Certain intermediate operations affect ordering behavior
  - e.g., sorted(), unordered(), skip(), & limit()

```
List<Integer> list = Arrays
  .asList(2, 3, 1, 4, 2);
```

*The encounter order is [2, 3, 1, 4, 2] since list is ordered & non-unique*

```
Integer[] doubledList = list
    .parallelStream()
    .distinct()
    .filter(x -> x % 2 == 0)
    .map(x -> x * 2)
    .limit(sOutputLimit)
    .toArray(Integer[]::new);
```

Again, recall that "ordered" isn't the same as "sorted"!

# Operations that Affect Results Order

- Certain intermediate operations affect ordering behavior
  - e.g., sorted(), unordered(), skip(), & limit()

```
List<Integer> list = Arrays
  .asList(2, 3, 1, 4, 2);
```

```
Integer[] doubledList = list
    .parallelStream()
    .distinct()
    .filter(x -> x % 2 == 0)
    .map(x -> x * 2)
    .limit(sOutputLimit)
    .toArray(Integer[]::new);
```

*Remove duplicate elements from the stream (a stateful intermediate operation)*

See [docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#distinct](docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#distinct)

# Operations that Affect Results Order

- Certain intermediate operations affect ordering behavior
  - e.g., sorted(), unordered(), skip(), & limit()

```
List<Integer> list = Arrays
   .asList(2, 3, 1, 4, 2);
```

```
Integer[] doubledList = list
    .parallelStream()
    .distinct()
    .filter(x -> x % 2 == 0)
    .map(x -> x * 2)
    .limit(sOutputLimit)
    .toArray(Integer[]::new);
```

*Only process sOutputLimit elements in the stream (a stateful intermediate operation)*

See [docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#limit](docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#limit)

# Operations that Affect Results Order

- Certain intermediate operations affect ordering behavior
  - e.g., sorted(), unordered(), skip(), & limit()

```
List<Integer> list = Arrays
   .asList(2, 3, 1, 4, 2);
```

```
Integer[] doubledList = list
    .parallelStream()
    .distinct()
    .filter(x -> x % 2 == 0)
    .map(x -> x * 2)
    .limit(sOutputLimit)
    .toArray(Integer[]::new);
```

*The result **must** be [8, 4], but the code is slow due to limit() & distinct() "stateful" semantics in parallel streams*

See [developer.ibm.com/articles/j-java-streams-3-brian-goetz](developer.ibm.com/articles/j-java-streams-3-brian-goetz)

## Operations that Affect Results Order

- Certain intermediate operations affect ordering behavior
  - e.g., sorted(), unordered(), skip(), & limit()

```
List<Integer> list = Arrays
  .asList(2, 3, 1, 4, 2);
```

```
Integer[] doubledList = list
    .parallelStream()
    .unordered()
    .distinct()
    .filter(x -> x % 2 == 0)
    .map(x -> x * 2)
    .limit(sOutputLimit)
    .toArray(Integer[]::new);
```

*This code runs faster since stream is unordered, so therefore limit() & distinct() incur less overhead*

See docs.oracle.com/javase/8/docs/api/java/util/stream/BaseStream.html#unordered

## Operations that Affect Results Order

- Certain intermediate operations affect ordering behavior
  - e.g., sorted(), unordered(), skip(), & limit()

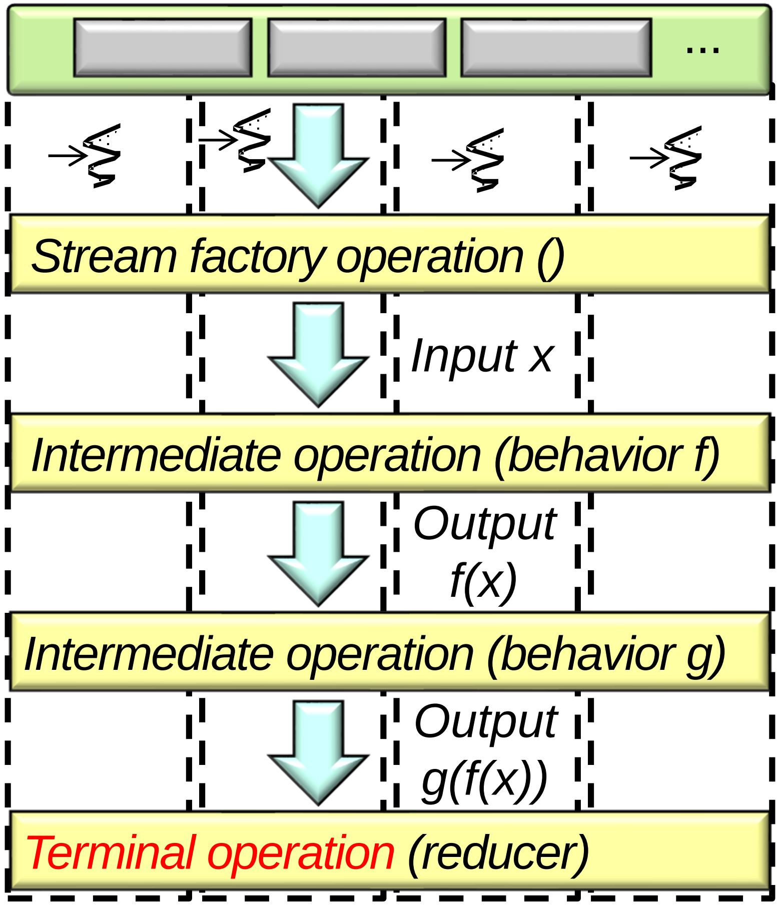
```
List<Integer> list = Arrays
  .asList(2, 3, 1, 4, 2);
```

```
Integer[] doubledList = list
   .parallelStream()
   .unordered()
   .distinct()
   .filter(x -> x % 2 == 0)
   .map(x -> x * 2)
   .limit(sOutputLimit)
   .toArray(Integer[]::new);
```

*Since encounter order need not be maintained the results could be [8, 4] or [4, 8]*

## Operations that Affect Results Order

- Certain terminal operations also affect ordering behavior


...
Stream factory operation ()
Input x
Intermediate operation (behavior f)
Output f(x)
Intermediate operation (behavior g)
Output g(f(x))
Terminal operation (reducer)

# Operations that Affect Results Order

- Certain terminal operations also affect ordering behavior, e.g.
  - forEachOrdered()

*The encounter order is [2, 3, 1, 4, 2] since list is ordered & non-unique.*

```
List<Integer> list = Arrays
  .asList(2, 3, 1, 4, 2);

List<Integer> results =
  new ArrayList<>();

list
  .parallelStream()
  .distinct()
  .filter(x -> x % 2 == 0)
  .map(x -> x * 2)
  .limit(sOutputLimit)
  .forEachOrdered
     (results::add);
```

## Operations that Affect Results Order

- Certain terminal operations also affect ordering behavior, e.g.
  - forEachOrdered()

```
List<Integer> list = Arrays
   .asList(2, 3, 1, 4, 2);

List<Integer> results =
   new ArrayList<>();

list
   .parallelStream()
   .distinct()
   .filter(x -> x % 2 == 0)
   .map(x -> x * 2)
   .limit(sOutputLimit)
   .forEachOrdered
      (results::add);
```

*This list supports unsynchronized insertions & removals of elements*

## Operations that Affect Results Order

- Certain terminal operations also affect ordering behavior, e.g.
  - forEachOrdered()

```
List<Integer> list = Arrays
   .asList(2, 3, 1, 4, 2);

List<Integer> results =
  new ArrayList<>();

list
   .parallelStream()
   .distinct()
   .filter(x -> x % 2 == 0)
   .map(x -> x * 2)
   .limit(sOutputLimit)
   .forEachOrdered
      (results::add);
```

*Results must appear in encounter order (may be slow due to implicit synchronization)*

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

# Operations that Affect Results Order

- Certain terminal operations also affect ordering behavior, e.g.
  - forEachOrdered()
  - forEach()

*Results need not appear in encounter order (will be fast due to absence of synchronization)*

```
List<Integer> list = Arrays
   .asList(2, 3, 1, 4, 2);

List<Integer> results =
  new ArrayList<>();

list
   .parallelStream()
   .distinct()
   .filter(x -> x % 2 == 0)
   .map(x -> x * 2)
   .limit(sOutputLimit)
   .forEach
      (results::add);
```

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

# End of Java Parallel Stream Internals: Order of Results (Part 3)