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

Multiple examples are analyzed in detail

See [github.com/douglasraigschmidt/LiveLessons/tree/master/Java8/ex21](https://github.com/douglasraigschmidt/LiveLessons/tree/master/Java8/ex21)

```java
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);
```
Operations that Affect Results Order
Operations that Affect Results Order

- Certain intermediate operations affect ordering behavior
Operations that Affect Results Order

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

```java
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](https://developer.ibm.com/articles/j-java-streams-3-brian-goetz)
Again, recall that “ordered” isn’t the same as “sorted”!

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

```java
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);
```
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
Operations that Affect Results Order

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

```java
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)
Certain intermediate operations affect ordering behavior

- e.g., sorted(), unordered(), skip(), & limit()

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

```java
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()

```java
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](docs.oracle.com/javase/8/docs/api/java/util/stream/BaseStream.html#unordered)
• Certain intermediate operations affect ordering behavior
  
  • e.g., sorted(), unordered(), skip(), & limit()

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

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);
Operations that Affect Results Order

- Certain terminal operations also affect ordering behavior

Diagram:
- 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()

```java
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);

The encounter order is [2, 3, 1, 4, 2] since list is ordered & non-unique.
```
Operations that Affect Results Order

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

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

```java
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](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#forEachOrdered)
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)

```java
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](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#forEach)
End of Java Parallel Stream Internals: Order of Results (Part 3)