

Ok Google,
पुराने हिंदी गाने सुनाओ



Java 8 Streams - Ordering

[Updated: Nov 2, 2018, Created: Oct 28, 2016]

The order in which Java 8 streams process the elements depends on following factors.

Encounter order

This is the order in which the stream source makes its elements available. The source elements can be in a particular order or it might not have any meanings of order. If there's no encounter order from source then we should not care whether the stream result will be ordered. If we have an ordered source and perform intermediate and terminal operations which guarantee to maintain the order, it does not matter whether the stream is processed in parallel or sequential, the order will be maintained.

Splitterator#ORDERED characteristic:

If stream is created with **Splitterator#ORDERED** characteristic (please check out our [Splitterator tutorial](#)) then the implementation logic of operations like `limit()`, `skip()`, `distinct()` and `findFirst()` will be tied to encounter order regardless if it's sequential or parallel stream. We will discuss these operations in the next sections.

Collection sources:

Streams created via **Collection#stream()** or **Collection#parallelStream()** will be ordered depending on the underlying collection. For example List and arrays are ordered. HashSet is not ordered, whereas LinkedHashSet is ordered (Check out our [Java collection cheat sheet](#)).

Other sources:

All stream sources created from static methods of stream classes e.g. **Stream#of()**, **Stream#iterate()**, **Stream#builder()** (and other corresponding `IntStream`, `LongStream`, `DoubleStream` methods) are ordered. (It's good to see our [stream operation cheat sheet](#) whenever we are comparing stream operations)

Stream#concat methods create a new stream of the first stream followed by all the elements of the second stream. The resulting stream is ordered if both of the input streams are ordered.

Stream#generate(), **IntStream#generate()** and other Long, double generate functions create the streams without **Splitterator#ORDERED** characteristic, so it's unordered.

Intermediate operations and encounter order

Following are the intermediate methods which either effect the encounter order or effect the performance in the presence of encounter order.

Stream#sorted()

This method returns a stream whose elements are sorted in natural order, thus ignoring any initial encounter order and imposing a new sorted order.

We can also use overloaded **sorted(Comparator<? super T> comparator)** to sort elements per provided comparator.

```
Set<Integer> list = new HashSet<>(Arrays.asList(2, 1, 3));
Object[] objects = list.stream().sorted().toArray();
System.out.println(Arrays.toString(objects))
```

Output:

```
[1, 2, 3]
```

Shopp
Never M
You Als
Yahoo :

Dis

For ordered streams, the sort is stable. For unordered streams, no stability guarantees are made. A [stable sort](#) is the one where two objects with same compare keys appear in the same encounter order in sorted output as they appear in the input source.

BaseStream#unordered()

Returns an unordered stream if it initially has encounter order, no change otherwise.

This method doesn't take any actions to explicitly unordered the stream, it just removes the ORDERED constraint on the stream which will optimize operations like **skip()**, **limit()**, **distinct()** and the terminal operation **findFirst**.

Stream#skip(long n)

This method returns a new stream which will discard the first n elements of the stream. For ordered parallel stream it's a comparatively expensive operation, that's because multiple threads have to wait for each other to figure out the result which should respect the encountered order. Using an unordered stream source (such as **generate(Supplier)**) or removing the ordering constraint by using **BaseStream#unordered()** may result in performance improvement for parallel pipelines, we should only do that if we don't care about the ordered result.

In following examples we are going to use our PerformanceTestUtil.runTest method to record execution time, this method also runs each test twice, first time silently to avoid cold-start (class loading etc) extra time. The complete code is included in the project browser below.

```
public class SkipExample {

    public static void main (String[] args) {

        PerformanceTestUtil.runTest("unordered parallel skip", () -> {
            IntStream intStream = IntStream.range(1, 100000000);
            intStream.unordered().parallel().skip(1000).toArray();
        });

        PerformanceTestUtil.runTest("ordered parallel skip", () -> {
            IntStream intStream = IntStream.range(1, 100000000);
            intStream.parallel().skip(1000).toArray();
        });
    }
}
```

Output on my intel i7-2.70GHz, 4 core, 8 logical processor, 16Gb ram machine:

```
unordered parallel skip time taken: 123.6 milliseconds
ordered parallel skip time taken: 166.5 milliseconds
```

Note that the result might vary from machine to machine.

Discover insights and drive innovation in an organization with analytics and Big Data.

Ad Data-Driven Business, Statistics & programming, Data Visualization & exploitation, ...

Barcelona Tech School

Apply Now

Stream#limit(long maxSize)

This method returns a new stream truncated to be no longer than maxSize. For ordered parallel stream it's a comparatively expensive operation, that's because at terminal operation multiple threads have to wait for each other to find out specified number of elements in encounter order rather than the ones which just finish first in time. Using an unordered stream source or removing the ordering constraint by using **BaseStream#unordered()** may result in performance improvements.



```

public class LimitExample {
    public static void main (String[] args) {

        PerformanceTestUtil.runTest("unordered parallel stream limit test", () -> {
            IntStream stream = IntStream.range(0, 1000000000);
            stream.unordered().parallel().filter(i -> i % 2 == 0)
                .limit(100000000).count();
        });

        PerformanceTestUtil.runTest("ordered parallel stream limit test", () -> {
            IntStream stream = IntStream.range(0, 1000000000);
            stream.parallel().filter(i -> i % 2 == 0)
                .limit(100000000).count();
        });
    }
}

```

Output:

```

unordered parallel skip time taken: 148.7 milliseconds
ordered parallel skip time taken: 160.5 milliseconds

```

Stream#distinct(), IntStream#distinct(), LongStream#distinct() and DoubleStream#distinct()

These methods return a new stream consisting of the distinct elements of this stream. The distinct elements are select based on equal and hashCode methods.

```

public class DistinctExample {
    public static void main (String[] args) {
        PerformanceTestUtil.runTest("unordered stream", () -> {
            IntStream stream = IntStream.range(0, 1000000);
            stream.unordered().parallel().distinct().count();
        });

        PerformanceTestUtil.runTest("ordered stream", () -> {
            IntStream stream = IntStream.range(0, 1000000);
            stream.parallel().distinct().count();
        });
    }
}

```

Output:

```

unordered stream time taken: 32.43 milliseconds
ordered stream time taken: 134.6 milliseconds

```

As seen in the output, distinct operation with ordered parallel stream is noticeably slower than the unordered parallel stream. The reason is: **ordered parallel stream** preserves the stability whereas, **unordered parallel stream** does not.

The distinct operation on ordered stream preserves the encounter order and the element appearing first in the order are selected for the final result. For unordered stream no stability is maintained, the duplicates are just ignored which come later in time rather than respecting encounter order.

distinct() is full barrier operation with a lot of memory utilization.

Generally full barrier operation means all operations appearing first in sequence must be performed completely before starting the next operation.

If order is not important we should always choose unordered data source or use the method **unordered()** to remove order constraint.



Following example shows what stability means in ordered parallel streams:

```
public class MyObject {
    private static int c = 0;
    private int id = ++c;
    private String str;

    public MyObject (String str) {
        this.str = str;
    }

    @Override
    public boolean equals (Object o) {
        if (this == o)
            return true;
        if (o == null || getClass() != o.getClass())
            return false;

        MyObject myObject = (MyObject) o;

        return str != null ? str.equals(myObject.str) : myObject.str == null;
    }

    @Override
    public int hashCode () {
        return str != null ? str.hashCode() : 0;
    }

    @Override
    public String toString () {
        return "MyObject{id=" + id + ", str='" + str + "'}";
    }
}
```

```
public class DistinctStabilityExample {
    Object[] myObjects = createStream().parallel().distinct().toArray();
    System.out.printf("ordered distinct result 1: %s\n",
        Arrays.toString(myObjects));

    MyObject.c = 0;
    myObjects = createStream().parallel().distinct().toArray();
    System.out.printf("ordered distinct result 2: %s\n",
        Arrays.toString(myObjects));

    MyObject.c = 0;
    myObjects = createStream().unordered().parallel().distinct().toArray();
    System.out.printf("unordered distinct result 1: %s\n",
        Arrays.toString(myObjects));

    MyObject.c = 0;
    myObjects = createStream().unordered().parallel().distinct().toArray();
    System.out.printf("unordered distinct result 2: %s\n",
        Arrays.toString(myObjects));
}
```

```
ordered distinct result 1: [MyObject{id=1, str='a'}, MyObject{id=2, str='b'}, MyObject{id=3, str='c'}]
ordered distinct result 2: [MyObject{id=1, str='a'}, MyObject{id=2, str='b'}, MyObject{id=3, str='c'}]
unordered distinct result 1: [MyObject{id=7, str='a'}, MyObject{id=4, str='b'}, MyObject{id=5, str='c'}]
unordered distinct result 2: [MyObject{id=1, str='a'}, MyObject{id=4, str='b'}, MyObject{id=5, str='c'}]
```



Modern Java 8 with Lambdas
\$44.65



OCP: Oracle Professional Java 8 Programmer's Certification Study Guide
\$37.53

Java 11

[Java 11](#)

Java 10

[Java 10](#)

Java 9

[Java 9](#)

[Java 9](#)

[Java 9](#)

Recent

[Spring Local C](#)

[Spring Listene](#)

[Spring Default](#)

[Spring request](#)

[Spring configu](#)

[Spring Configu](#)

[Spring failure](#)

[Spring](#)

[TypeSc Suppor](#)

[JavaSci Promisi](#)

[TypeSc](#)

[JPA Cri Objects](#)

[JPA Cri](#)

[JPA Cri](#)

[JPA Cri Criteria](#)

[JPA Cri Criteria](#)

[Java 12 files wi getCho](#)

Terminal operations and encounter order

Following are the terminal operations which either ignore the encounter order or effect the performance in the presence of ORDERED constraint.

Stream#forEach(Consumer<? super T> action)

For parallel stream pipelines, Stream#forEach() operation does not guarantee to respect the encounter order of the stream, as doing will undermine parallelism. For parallel streams, we should not use this method if encounter order is important.

Stream#forEachOrdered(Consumer<? super T> action)

This method guarantees the encounter order of the stream. In case of parallel processing, Stream#forEachOrdered() processes the elements one by one. Each consumer action establishes [happens-before](#) relation with the subsequent elements action. We may lose the parallel stream performance gain by using this method.

Here's the comparison between **forEach()** and **forEachOrdered()**

```
public class ForEachExample {
    public static void main (String[] args) {

        final int[] ints = IntStream.range(0, 5).toArray();
        PerformanceTestUtil.runTest("forEach() method", () -> {
            Arrays.stream(ints).parallel().forEach(i -> doSomething(i));
        });

        PerformanceTestUtil.runTest("forEachOrdered() method", () -> {
            Arrays.stream(ints).parallel().forEachOrdered(i -> doSomething(i));
        });
    }

    private static void doSomething (int i) {
        try {
            Thread.sleep(10);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.printf("%s, ", i);
    }
}
```

Output:

```
3, 1, 4, 0, 2, forEach() method time taken: 10.08 milliseconds
0, 1, 2, 3, 4, forEachOrdered() method time taken: 50.18 milliseconds
```

It's clear from the output forEachOrdered maintains the encounter order but is slower.

Stream#peek()

This method and other peek methods defined in IntStream, LongStream and DoubleStream do not respect encounter order. Also these methods are for debugging purpose only and shouldn't be used to implement application logic.

```
public class PeekExample {
    public static void main (String[] args) {
        IntStream.range(0, 5).parallel().peek(System.out::println).
                                                count();
    }
}
```

[Java 12](#)

[excepti](#)

[Java 12](#)

[Java 12](#)

[Collect](#)

[Java 12](#)

[Java 12](#)

[Java 12](#)

[Git - Ur](#)

[Git - Cr](#)

['branch](#)

[TypeSc](#)

[TypeSc](#)

[TypeSc](#)

[Jackson](#)

[Installi](#)

[started](#)

[Mongol](#)

[JPA Cri](#)

[Operati](#)

[JPA Cri](#)

[index\(\)](#)

[JPA Cri](#)

[Operati](#)

[JPA Cri](#)

[Manipu](#)

[TypeSc](#)

[Obtaini](#)

[Object.](#)

[Git - Hc](#)

[over SS](#)

[Groovy](#)

[Groovy](#)

[Groovy](#)

[private](#)

[Groovy](#)

[Groovy](#)

[Groovy](#)

[Groovy](#)

[Getting](#)

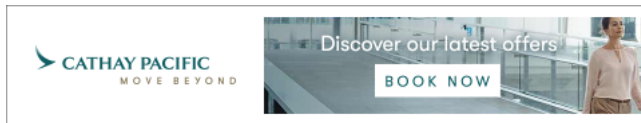
[Aparap](#)

[Extract](#)

[Mounti](#)

Output:

```
4
1
0
2
3
```



Dependencies and Technologies Used:

- JDK 1.8
- Maven 3.0.4

Stream Ordering Examples

stream-ordering-examples

src

main

java

com

logicbig

example

DistinctExample.java

DistinctStabilityExample.java

ForEachExample.java

LimitExample.java

PeekExample.java

PerformanceTestUtil.java

SkipExample.java

SortedExample.java

pom.xml

```
import java.util.stream.IntStream;
import java.util.stream.LongStream;

public class ForEachExample {

    public static void main (String[] args) {

        final int[] ints = IntStream.range(0, 5).toArray();
        PerformanceTestUtil.runTest("forEach() method", () -> {
            Arrays.stream(ints).parallel().forEach(i -> doSomething(i));
        });

        PerformanceTestUtil.runTest("forEachOrdered() method", () -> {
            Arrays.stream(ints).parallel().forEachOrdered(i -> doSomething(i));
        });
    }

    private static void doSomething (int i) {
        try {
            Thread.sleep(10);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.printf("%s, ", i);
    }
}
```

Data Science Certification

Learn Big Data, Machine Learning, Data Analytics, Data Science using Python & R. Amity Future Academy

OPEN

Project Structure

```
stream-ordering-examples
src
main
java
com
logicbig
example
    DistinctExample.java
    DistinctStabilityExample.java
    ForEachExample.java
    LimitExample.java
    PeekExample.java
    PerformanceTestUtil.java
    SkipExample.java
    SortedExample.java
pom.xml
```

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.logicbig.example</groupId>
  <artifactId>stream-ordering-examples</artifactId>
  <version>1.0-SNAPSHOT</version>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.5.1</version>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
          <encoding>UTF-8</encoding>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

```
package com.logicbig.example;

import java.util.stream.IntStream;

public class DistinctExample {

    public static void main (String[] args) {
        PerformanceTestUtil.runTest("unordered stream", () -> {
            IntStream stream = IntStream.range(0, 1000000);
            stream.unordered().parallel().distinct().count();
        });

        PerformanceTestUtil.runTest("ordered stream", () -> {
            IntStream stream = IntStream.range(0, 1000000);
            stream.parallel().distinct().count();
        });
    }
}
```

```
package com.logicbig.example;

import java.util.Arrays;
import java.util.stream.Stream;

public class DistinctStabilityExample {
    public static void main (String[] args) {

        Object[] myObjects = createStream().parallel().distinct().toArray();
        System.out.printf("ordered distinct result 1: %s\n",
            Arrays.toString(myObjects));

        MyObject.c = 0;
        myObjects = createStream().parallel().distinct().toArray();
        System.out.printf("ordered distinct result 2: %s\n",
```

```

        Arrays.toString(myObjects));

    MyObject.c = 0;
    myObjects = createStream().unordered().parallel().distinct().toArray();
    System.out.printf("unordered distinct result 1: %s%n",
        Arrays.toString(myObjects));

    MyObject.c = 0;
    myObjects = createStream().unordered().parallel().distinct().toArray();
    System.out.printf("unordered distinct result 2: %s%n",
        Arrays.toString(myObjects));

}

private static Stream<MyObject> createStream () {
    return Stream.of(new MyObject("a"), new MyObject("b"),
        new MyObject("c"), new MyObject("b"),
        new MyObject("c"), new MyObject("c"),
        new MyObject("a"));
}

private static class MyObject {
    private static int c = 0;
    private int id = ++c;
    private String str;

    public MyObject (String str) {
        this.str = str;
    }

    @Override
    public boolean equals (Object o) {
        if (this == o)
            return true;
        if (o == null || getClass() != o.getClass())
            return false;

        MyObject myObject = (MyObject) o;

        return str != null ? str.equals(myObject.str) : myObject.str == null;
    }

    @Override
    public int hashCode () {
        return str != null ? str.hashCode() : 0;
    }

    @Override
    public String toString () {
        return "MyObject{id=" + id + ", str='" + str + "'}";
    }
}
}

```

```

package com.logicbig.example;

import java.util.Arrays;
import java.util.stream.DoubleStream;
import java.util.stream.IntStream;
import java.util.stream.LongStream;

public class ForEachExample {

```



```
public static void main (String[] args) {
    final int[] ints = IntStream.range(0, 5).toArray();
    PerformanceTestUtil.runTest("forEach() method", () -> {
        Arrays.stream(ints).parallel().forEach(i -> doSomething(i));
    });

    PerformanceTestUtil.runTest("forEachOrdered() method", () -> {
        Arrays.stream(ints).parallel().forEachOrdered(i -> doSomething(i));
    });
}

private static void doSomething (int i) {
    try {
        Thread.sleep(10);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.printf("%s, ", i);
}
}
```

```
package com.logicbig.example;

import java.util.stream.IntStream;

public class LimitExample {

    public static void main (String[] args) {

        PerformanceTestUtil.runTest("unordered parallel stream limit test", () -> {
            IntStream stream = IntStream.range(0, 1000000000);
            stream.unordered()
                .parallel()
                .filter(i -> i % 2 == 0)
                .limit(100000000)
                .count();
        });

        PerformanceTestUtil.runTest("ordered parallel stream limit test", () -> {
            IntStream stream = IntStream.range(0, 1000000000);
            stream.parallel()
                .filter(i -> i % 2 == 0)
                .limit(100000000)
                .count();
        });
    }
}
```

```
package com.logicbig.example;

import java.util.stream.IntStream;

public class PeekExample {
    public static void main (String[] args) {
        IntStream.range(0, 5).parallel().peek(System.out::println).
            count();
    }
}
```

```

package com.logicbig.example;

import java.io.IOException;
import java.io.OutputStream;
import java.io.PrintStream;
import java.util.Optional;
import java.util.concurrent.TimeUnit;
import java.util.stream.Stream;

import static java.util.concurrent.TimeUnit.*;

public class PerformanceTestUtil {

    public static void runTest (String msg, Runnable testRunner) {
        setPrintStreamDisabled(true);
        //run test quietly first time to avoid cold start false-positive result
        testRunner.run();
        setPrintStreamDisabled(false);

        long startTime = getTimeElapsed(0);
        testRunner.run();
        System.out.printf("%s time taken: %s%n", msg, timeToString(getTimeElapsed(startTime)));
    }

    private static String timeToString (long nanos) {

        Optional<TimeUnit> first = Stream.of(DAYS, HOURS, MINUTES, SECONDS, MILLISECONDS,
            MICROSECONDS).filter(u -> u.convert(nanos, NANOSECONDS) > 0)
            .findFirst();
        TimeUnit unit = first.isPresent() ? first.get() : NANOSECONDS;

        double value = (double) nanos / NANOSECONDS.convert(1, unit);
        return String.format("%.4g %s", value, unit.name().toLowerCase());
    }

    private static long getTimeElapsed (long startTime) {
        return System.nanoTime() - startTime;
    }

    private static void setPrintStreamDisabled (boolean b) {
        if (b) {
            System.setOut(blankPrintStream);
        } else {
            System.setOut(originalPrintStream);
        }
    }

    private static final PrintStream originalPrintStream = System.out;
    private static PrintStream blankPrintStream = new PrintStream(new OutputStream() {
        public void write (int b) {
        }

        @Override
        public void write (byte[] b, int off, int len) throws IOException {
        }
    });
}

```

```

package com.logicbig.example;

import java.util.stream.IntStream;

public class SkipExample {

```

```

public static void main (String[] args) {
    PerformanceTestUtil.runTest("unordered parallel skip", () -> {
        IntStream intStream = IntStream.range(1, 100000000);
        intStream.unordered().parallel().skip(1000).toArray();
    });

    PerformanceTestUtil.runTest("ordered parallel skip", () -> {
        IntStream intStream = IntStream.range(1, 100000000);
        intStream.parallel().skip(1000).toArray();
    });
}
}

```

```



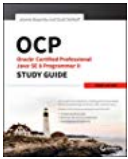





package com.logicbig.example;

import java.util.Arrays;
import java.util.HashSet;
import java.util.Set;
import java.util.stream.IntStream;

public class SortedExample {

    public static void main (String[] args) {
        Set<Integer> list = new HashSet<>(Arrays.asList(2, 1, 3));
        Object[] objects = list.stream().sorted().toArray();
        System.out.println(Arrays.toString(objects));
    }
}

```

 <p>Modern Java in Action: Lambdas, streams, functional and reactive programming</p> <p>\$44.05 \$54.00</p> <p>(4)</p>	 <p>Java 8 in Action: Lambdas, Streams, and functional-style programming</p> <p>\$83.69</p> <p>(84)</p>	 <p>OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide: Exam 1Z0-809</p> <p>\$37.53 \$50.00</p> <p>(66)</p>	 <p>Effective Java</p> <p>\$43.86 \$54.00</p> <p>(79)</p>
 <p>Java Pocket Guide: Instant Help for Java Programmers</p> <p>\$13.43 \$40.00</p> <p>(6)</p>	 <p>OCP Java SE 8 Programmer II Exam Guide (Exam 1Z0-809)</p> <p>\$35.89 \$60.00</p> <p>(13)</p>	 <p>Java 8 Lambdas: Pragmatic Functional Programming</p> <p>\$18.35</p> <p>(33)</p>	 <p>Oracle Certified Professional Java SE 8 Programmer Exam 1Z0-809</p> <p>\$38.24 \$44.00</p> <p>(16)</p>

See Also

[Short circuiting operations](#)

[Lazy evaluation](#)

[Sequential vs Parallel streams](#)

[Java 8 Stream operations cheat sheet](#)

[What are Java 8 streams?](#)

[Stateful vs Stateless behavioral parameters](#)

[Side Effects](#)

[Reduction](#)

[Mutable Reduction](#)

[Java 8 Streams quick examples](#)

Java 11 Tutorials

[Java 11 Features](#)