# Java 8 Streams - Short circuiting operations

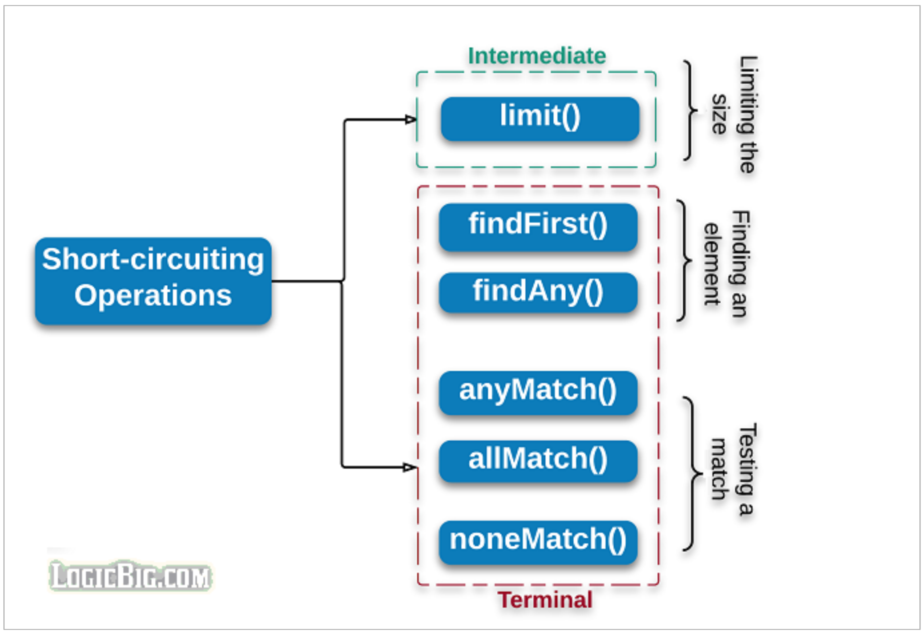
[Updated: Feb 7, 2017, Created: Sep 10, 2016]

Java 8 short-circuiting operations are just like boolean short-circuit evaluations in Java.

In boolean short-circuiting logic, for example `firstBoolean && secondBoolean`, if firstBoolean is false then the remaining part of the expression is ignored (the operation is short-circuited) because the remaining evaluation will be redundant. Similarly in `firstBoolean || secondBoolean`, if firstBoolean is true the remaining part is short-circuited.

Java 8 Stream short-circuit operations are not limited to boolean types. There are pre defined short-circuiting operations.

Java 8 stream intermediate and terminal operations both can be short circuiting.



## Intermediate short-circuiting methods

Intermediate short-circuiting methods cause a stream to operate on a reduced size.

No actual iteration of the elements occur on calling these intermediate methods until the terminal operation is performed.

For example, if some intermediate short-circuiting method reduce the stream size to 0, the pipeline won't stop processing until the terminal step is executed and that's where intermediate lambdas are evaluated to yield a terminal result.

Following is the only one intermediate-short-circuiting method currently defined in Stream interface:

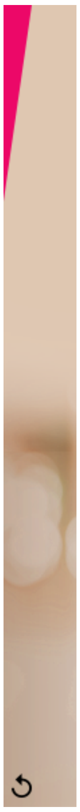
- `Stream<T> limit(long maxSize)`

  Returns a new stream created from this stream, truncated to be no longer than maxSize in length.

  Following example shows the difference between using limit() and not using limit():

```
package com.logicbig.example;

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

```
public class LimitExample {

    public static void main (String[] args) {
        int[] ints = {1, 2, 3, 4, 5, 6};

        System.out.printf("Source: %s%n", Arrays.toString(ints));
        System.out.println("Finding even numbers.");
        runWithoutLimit(Arrays.stream(ints));
        runWithLimit(Arrays.stream(ints));
    }

    private static void runWithoutLimit (IntStream stream) {
        System.out.println("Running without limit()");

        //filter even numbers
        stream.filter(i -> i % 2 == 0)
              .forEach(System.out::println);
    }

    private static void runWithLimit (IntStream stream) {
        System.out.println("Running with limit(2)");

        //filter even numbers
        stream.filter(i -> i % 2 == 0)
              .limit(2)
              .forEach(System.out::println);
    }
}
```



**Output:**

```
Source: [1, 2, 3, 4, 5, 6]
Finding even numbers.
Running without limit()
2
4
6
Running with limit(2)
2
4
```

Note: In above example we are creating and passing new stream instance from the same source to the methods, runWithLimit(..) and runWithoutLimit(..).

A stream cannot be reused after a terminal operation is called.

### Infinite streams and limit() method

limit(..) method is typically used, when there's an infinite input, e.g. when a stream created with static methods like `Stream<T> generate(Supplier<T> s)` or `Stream<T> iterate(T seed, UnaryOperator<T> f)` . Calling limit() method produces a finite stream as a result.

Following example will terminate after 5 elements.

```
Stream<Integer> stream = Stream.iterate(1, i -> i + 1);
stream.filter(i -> i % 2 == 0)
      .limit(5)
      .forEach(System.out::println);
```

**Output:**

```
2
4
6
8
10
```

If we remove limit(5) part, it will be printing even numbers forever.

An infinite streams is desirable where size of the data source is not known in advance, for example, data coming as messages from a remote location or data generated to show some GUI animations.

In above example, what if we use the limit(5) first and then apply the filter later? It's not relevant to the current topic but just see the outcome:

```
Stream<Integer> stream = Stream.iterate(1, i -> i + 1);
stream.limit(5)
      .filter(i -> i % 2 == 0)
      .forEach(System.out::println);
```

**Output:**

```
2
4
```

That shows that order matters while performing intermediate operations.

## Terminal short-circuiting methods

These terminal-short-circuiting methods can finish before transversing all the elements of the underlying stream.

A short-circuiting terminal operation, when operating on infinite input data source, may terminate in finite time.

Following are the terminal-short-circuiting methods defined in Stream interface:

- `Optional<T> findFirst()`:

  Returns the very first element (wrapped in Optional object) of this stream and before transversing the other.

  Following example shows a InStream of predefined elements which is meant to return only multiples of 3 results. The terminal operation terminated on finding the first element, hence short-circuited.

```
IntStream stream = IntStream.of(1, 2, 3, 4, 5, 6);
stream = stream.filter(i -> i % 3 == 0);
OptionalInt opt = stream.findFirst();
if (opt.isPresent()) {
    System.out.println(opt.getAsInt());
}
```

  **Output:**

```
3
```

  Please note that, in above example IntStream.of creates a sequential ordered stream. If source is not initially ordered then we cannot predict findFirst will still return '3'.

  For parallel ordered streams it's guaranteed that findFirst will return the very first element which will correspond to the first original source element (regardless it's been mapped to something else). The operation will wait for necessary parallel results arrival before short-circuiting.

- `Optional<T> findAny()`

  Returns an Optional instance which wraps any and only one element of the stream.

According to Java doc:

> The behavior of this operation is explicitly nondeterministic; it is free to select any element in the stream. This is to allow for maximal performance in parallel operations; the cost is that multiple invocations on the same source may not return the same result. (If a stable result is desired, use findFirst() instead.

A deterministic operation will always produces the same output for a given input, regardless of we use parallel or sequential pipeline.

Except for operations identified as explicitly nondeterministic, such as findAny(), whether a stream executes sequentially or in parallel should not change the result of the computation.

For a sequential stream there won't be any difference between 'findFirst' and 'findAny'. But for a parallel stream findAny will return 'any' element rather than waiting for the 'first' element.

Following example uses a parallel stream. I'm getting output of '6' instead of '2' on my machine. The result might also vary on multiple executions.

```
IntStream stream = IntStream.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
                            .parallel();
stream = stream.filter(i -> i % 2 == 0);
OptionalInt opt = stream.findAny();
if (opt.isPresent()) {
    System.out.println(opt.getAsInt());
}
```

**Output:**

```
6
```

- `boolean anyMatch(Predicate<? super T> predicate)`

  Tests whether any elements of this stream match the provided predicate. This terminal method will return as soon as it finds the match and will not transverse all the remaining elements to apply the predicate.

  Following example prints 'true':

```
  Stream<String> stream = Stream.of("one", "two","three", "four");
  boolean match = stream.anyMatch(s -> s.contains("our"));
  System.out.println(match);
```

- `boolean allMatch(Predicate<? super T> predicate)`

  Tests whether all elements match the provided predicate. It may return early with false result when any element doesn't match first.

  Following examples outputs false because 'Three' doesn't start with a lower case, at that point short-circuiting happens.

```
Stream<String> stream = Stream.of("one", "two", "Three", "four");
boolean match = stream.allMatch(s -> s.length() > 0 &&
                          Character.isLowerCase(s.charAt(0)));
System.out.println(match);
```

- `boolean noneMatch(Predicate<? super T> predicate)`

  Tests whether no elements of this stream match the provided predicate. It may return early with false result when any element matches the provided predicate first.

  Following example will print 'true' because none of the elements start with an upper case.

Modern J
Lambdas
$44.05 $

OCP: Or
Professio
$37.53 $

Java 1

Java 1

Java 1

Java 1

Java 9

Java 9

Java 9

Java 9

Recent

Spring
Local (

Spring
Listen

Spring
Defaul

Spring
reques

Spring
config

Spring
Config

Spring
failure

Spring

TypeS
Suppo

```
Stream<String> stream = Stream.of("one", "two", "three", "four");
boolean match = stream.noneMatch(s -> s.length() > 0 &&
                              Character.isUpperCase(s.charAt(0)));
System.out.println(match);
```

Dependencies and Technologies Used:

- JDK 1.8
- Maven 3.0.4

**Stream Short Circuiting Example**

```
stream-short-circuiting
    src
        main
            java
                com
                    logicbig
                        example
                            AllMatchExample.java
                            AnyMatchExample.java
                            FindAnyExample.java
                            FindFirstExample.java
                            LimitExample.java
                            LimitingInfiniteStream.ja
                            LimitingInfiniteStream2.ja
                            NoneMatchExample.java
    pom.xml
```

```
package com.logicbig.example;

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

public class LimitExample {

    public static void main (String[] args) {
        int[] ints = {1, 2, 3, 4, 5, 6};

        System.out.printf("Source: %s%n", Arrays.toString(ints));
        System.out.println("Finding even numbers.");
        runWithoutLimit(Arrays.stream(ints));
        //Note: creating and passing new stream because it
        // cannot be reused after a terminal operation is called.
        runWithLimit(Arrays.stream(ints));
    }

    private static void runWithoutLimit (IntStream stream) {
        System.out.println("Running without limit()");

        //filter even numbers
        stream.filter(i -> i % 2 == 0)
              .forEach(System.out::println);
    }
```

**Project Structure**

```
stream-short-circuiting
    src
        main
            java
                com
                    logicbig
                        example
                            AllMatchExample.java
                            AnyMatchExample.java
                            FindAnyExample.java
                            FindFirstExample.java
                            LimitExample.java
                            LimitingInfiniteStream.java
                            LimitingInfiniteStream2.java
```

NoneMatchExample.java

pom.xml

```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-short-circuiting</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>
```

```java
package com.logicbig.example;

import java.util.stream.Stream;

public class AllMatchExample {
    public static void main (String[] args) {
        Stream<String> stream = Stream.of("one", "two", "Three", "four");
        boolean match = stream.allMatch(s -> s.length() > 0 &&
                                    Character.isLowerCase(s.charAt(0)));
        System.out.println(match);
    }
}
```

```java
package com.logicbig.example;

import java.util.stream.Stream;


public class AnyMatchExample {

    public static void main(String[] args) {
        Stream<String> stream = Stream.of("one", "two", "three", "four");
        boolean match = stream.anyMatch(s -> s.contains("our"));
        System.out.println(match);
    }
}
```

```java
package com.logicbig.example;

import java.util.OptionalInt;
import java.util.stream.IntStream;

public class FindAnyExample {

    public static void main (String[] args) {
        IntStream stream = IntStream.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
                                    .parallel();
        stream = stream.filter(i -> i % 2 == 0);

        OptionalInt opt = stream.findAny();
        if (opt.isPresent()) {
            System.out.println(opt.getAsInt());
        }
    }
}
```

```java
package com.logicbig.example;

import java.util.OptionalInt;
import java.util.stream.IntStream;

public class FindFirstExample {

    public static void main (String[] args) {
        IntStream stream = IntStream.of(1, 2, 3, 4, 5, 6);
        stream = stream.filter(i -> i % 3 == 0);
        OptionalInt opt = stream.findFirst();
        if (opt.isPresent()) {
            System.out.println(opt.getAsInt());
        }
    }
}
```

```java
package com.logicbig.example;

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

public class LimitExample {

    public static void main (String[] args) {
        int[] ints = {1, 2, 3, 4, 5, 6};

        System.out.printf("Source: %s%n", Arrays.toString(ints));
        System.out.println("Finding even numbers.");
        runWithoutLimit(Arrays.stream(ints));
        //Note: creating and passing new stream because it
        // cannot be reused after a terminal operation is called.
        runWithLimit(Arrays.stream(ints));
    }

    private static void runWithoutLimit (IntStream stream) {
        System.out.println("Running without limit()");

        //filter even numbers
        stream.filter(i -> i % 2 == 0)
              .forEach(System.out::println);
    }

    private static void runWithLimit (IntStream stream) {
        System.out.println("Running with limit(2)");
        //filter even numbers
        stream.filter(i -> i % 2 == 0)
              .limit(2)
              .forEach(System.out::println);
    }
}
```

```
package com.logicbig.example;

import java.util.stream.Stream;

public class LimitingInfiniteStream {
    public static void main (String[] args) {
        Stream<Integer> stream = Stream.iterate(1, i -> i + 1);
        stream.filter(i -> i % 2 == 0)
              .limit(5)
              .forEach(System.out::println);
    }
}
```

```
package com.logicbig.example;

import java.util.stream.Stream;

public class LimitingInfiniteStream2 {
    public static void main (String[] args) {
        Stream<Integer> stream = Stream.iterate(1, i -> i + 1);
        stream.limit(5)
              .filter(i -> i % 2 == 0)
              .forEach(System.out::println);
    }
}
```

```
package com.logicbig.example;


import java.util.stream.Stream;

public class NoneMatchExample {
    public static void main (String[] args) {
        Stream<String> stream = Stream.of("one", "two", "three", "four");
        boolean match = stream.noneMatch(s -> s.length() > 0 &&
                                Character.isUpperCase(s.charAt(0)));
        System.out.println(match);
    }
}
```

## See Also

Lazy evaluation
Sequential vs Parallel streams
Java 8 Stream operations cheat sheet
What are Java 8 streams?
Ordering
Stateful vs Stateless behavioral parameters
Side Effects
Reduction
Mutable Reduction
Java 8 Streams quick examples

## Java 11 Tutorials

Java 11 Features

## Java 10 Tutorials

Java 10 Features

## Java 9 Tutorials

Java 9 Module System
Java 9 Misc Features
Java 9 JShell

## Recent Tutorials

Spring Cloud - Hystrix CircuitBreaker, Thread Local Context Propagation
Spring Cloud - Circuit Breaker Hystrix Event Listener
Spring Cloud - Circuit Breaker Hystrix, Changing Default Thread Pool Properties
Spring Cloud - Circuit Breaker Hystrix, concurrent requests and default thread pool size
Spring Cloud - Circuit Breaker, Specifying Hystrix configuration in application.properties file
Spring Cloud - Hystrix Circuit Breaker, Setting Configuration Properties Using @HystrixProperty
Spring Cloud - Hystrix Circuit Breaker, getting failure exception in fallback method
Spring Cloud - Circuit Breaker Hystrix Basics
TypeScript - Standard JavaScript's built-in objects Supp
JavaScript - Async Processing with JavaScript Promise
TypeScript - Applying Mixins
JPA Criteria API - Modifying and Reusing Query Objects
JPA Criteria API - Delete Operations
JPA Criteria API - Update Operations
JPA Criteria API - Case Expressions with CriteriaBuilder.

## Share