

Google Home Mini
Best Smart Speaker*
Available in Hindi

*Source: Latest Logic Ventures Smart Spe

What are Java 8 streams?

[Updated: Feb 7, 2017, Created: Aug 15, 2016]

Java stream API defines functional-style operations on discrete sequence of data elements. These data elements are usually provided by some standard data structures (**ADT**), for example the ones provided in `java.util.collection`. These data structures are typically called stream sources.

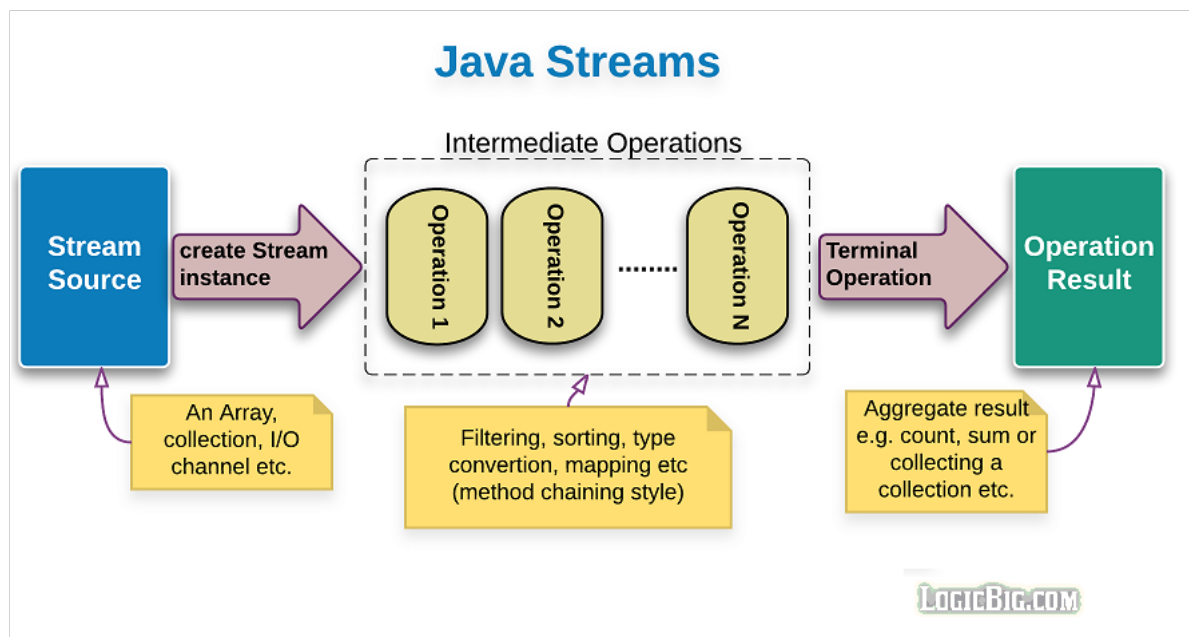
Java 8 enhanced the existing API like collections, arrays etc to add new methods to create **stream object** instances. This API itself provides some static method to generate finite/infinite stream of data elements.

Streams are functional, they operate on the provided source and produce results rather than modifying the source.

Stream operations

A stream life cycle can be divided into three types of operation:

1. Obtaining the instance of Stream from a source. A source might be an array, a collection, a generator function, an I/O channel, etc
2. Zero or more intermediate operations which transform a stream into another stream, such as filtering, sorting, element transformation (mapping)
3. A terminal operation which produces a result, such as count, sum or a new collection.



In this series of tutorials we will be exploring stream operations and the characteristics associated with them.

What is Stream Pipeline?

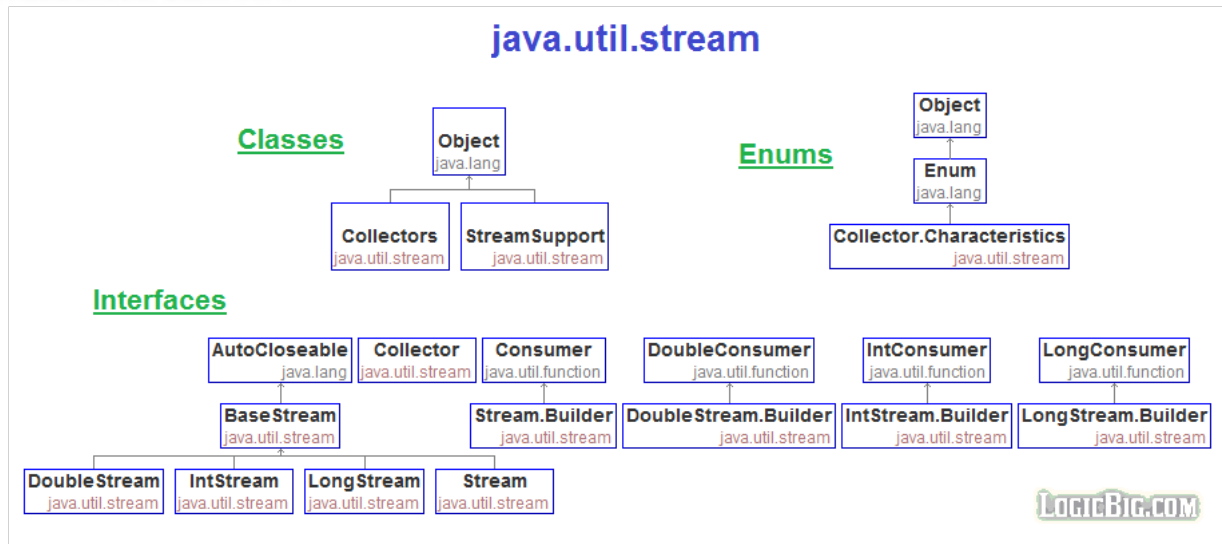
A stream pipeline is nothing but combined intermediate and terminal operations

Many stream operations return a stream themselves. This allows operations to be chained to form a larger pipeline. This enables certain optimizations, such as laziness and short-circuiting, which we explore later.

According to the [API docs](#) :

Stream operations are divided into intermediate and terminal operations, and are combined to form stream pipelines. A stream pipeline consists of a source (such as a Collection, an array, a generator function, or an I/O channel); followed by zero or more intermediate operations such as Stream.filter or Stream.map; and a terminal operation such as Stream.forEach or Stream.reduce.

java.util.stream API



Obtaining stream instances from the source

Java 8 has modified the existing collection and other data structures API to create/generate stream instances (please see the next section on stream instance). For example:

- Collection interface has added new default methods like stream() and parallelStream() which return a Stream instance .

```
List<String> list = Arrays.asList("1", "2", "3");
Stream<String> stream = list.stream();
```

- New method which generate stream instances have been added in Arrays class.

```
String[] strs = {"1", "2", "3"};
Stream<String> stream = Arrays.stream(strs);
```

The stream interfaces

This API defines Stream interface for object elements. For primitives it defines IntStream , LongStream and DoubleStream interfaces. For example if we modify the above code to define array of int rather than array of String:

```
int[] ints = {1, 2, 3};
IntStream stream = Arrays.stream(ints);
```

The extra primitive streams are provided for efficiency because wrapping primitives to Number objects and auto-boxing are relatively costly process.

The methods define by these interfaces can be divided into following two categories:

- **Intermediate operations:**
 - These methods usually accept functional interfaces as parameters and always return a new stream.

Functional interface parameter!! that means, We can take full advantage of lambda expressions here.

In fact using lambda expressions which can participate an 'internal iteration', was the sole motivation for introducing this API.

What is internal iteration:

The internal iterator, is kind of iterator where the library API takes care of iteration rather than application code has to do iteration itself (external iterator).

- These methods store the provided functions rather than evaluating them at the same time, associate them with 'this' stream instance for later use (at the actual traversal step in the terminal method) and then return a new instance of the Stream
- **Terminal operations:** These methods produce some result e.g. `count()`, `max(..)`, `toArray(..)`, `collect(..)` .

Data Science Certification

Ad Learn Big Data, Machine Learning, Data Analytics Science using Python & R.

Amity Future Academy

OPEN

Imperative vs Declarative styles of coding

In **imperative programming**, we have to write code line by line to give instructions to the computer about what we want to do to achieve a result. For example iterating through a collection of integer using 'for loop' to calculate sum is an imperative style. We have to create local variables and maintain the loop ourselves rather than focusing on what result we want to achieve. It's like repeating the same logic regarding looping and calculating the sum every time.

In **declarative programming**, we just have to focus on what we want to achieve without repeating the low level logic (like looping) every time. Stream API achieve this by using internal iterator constructs along with lambda expressions. They not only take care of iteration but also provide intermediate and terminal operations to customize the outcome, and additionally abstract away, the well-tested algorithms and parallelism, hence giving the best performance.

Examples

In following examples we will compare the old imperative style with new declarative style.

Old Imperative Style	New Declarative Style
Internal iteration, using new default method	
<div> <div> <div>Iterable#forEach(Consumer<? super T> action)</div> </div> </div>	
<pre>List<String> list = Arrays.asList("Apple", "Orange", "Banana"); for (String s : list) { System.out.println(s); }</pre>	<pre>List<String> list = Arrays.asList("Apple", "Orange", "Banana"); //using Lambda expression list.forEach(s -> System.out.println(s)); //or using method reference on System.out instance list.forEach(System.out::println);</pre>
Counting even numbers in a list, using	
<div> <div>Collection#stream()</div> <div>and</div> <div>java.util.stream.Stream</div> </div>	



<pre>List<Integer> list = Arrays.asList(3, 2, 12, 5, 6, 11, 13); int count = 0; for (Integer i : list) { if (i % 2 == 0) { count++; } } System.out.println(count);</pre>	<pre>List<Integer> list = Arrays.asList(3, 2, 12, 5, 6, 11, 13); long count = list.stream() .filter(i -> i % 2 == 0) .count(); System.out.println(count);</pre>
Retrieving even number list	
<pre>List<Integer> list = Arrays.asList(3, 2, 12, 5, 6, 11, 13); List<Integer> evenList = new ArrayList<>(); for (Integer i : list) { if (i % 2 == 0) { evenList.add(i); } } System.out.println(evenList);</pre>	<pre>List<Integer> list = Arrays.asList(3, 2, 12, 5, 6, 11, 13); List<Integer> evenList = list.stream() .filter(i -> i % 2 == 0) .collect(Collectors.toList()); System.out.println(evenList);</pre> <p>Or if we are only interested in printing:</p> <pre>List<Integer> list = Arrays.asList(3, 2, 12, 5, 6, 11, 13); list.stream().filter(i -> i % 2 == 0) .forEach(System.out::println);</pre>
Finding sum of all even numbers	
<pre>List<Integer> list = Arrays.asList(3, 2, 12, 5, 6, 11, 13); int sum = 0; for (Integer i : list) { if (i % 2 == 0) { sum += i; } } System.out.println(sum);</pre>	<pre>List<Integer> list = Arrays.asList(3, 2, 12, 5, 6, 11, 13); int sum = list.stream() .filter(i -> i % 2 == 0) .mapToInt(Integer::intValue) .sum(); System.out.println(sum);</pre> <p>Alternatively</p> <pre>List<Integer> list = Arrays.asList(3, 2, 12, 5, 6, 11, 13); int sum = list.stream() .filter(i -> i % 2 == 0) .reduce(0, (i, c) -> i + c); System.out.println(sum);</pre>
Finding whether all integers are less than 10 in the list	
<pre>List<Integer> list = Arrays.asList(3, 2, 12, 5, 6, 11, 13); boolean b = true; for (Integer i : list) { if (i >= 10) { b = false; break; } } System.out.println(b);</pre>	<pre>List<Integer> list = Arrays.asList(3, 2, 12, 5, 6, 11, 13); boolean b = list.stream() .allMatch(i -> i < 10); System.out.println(b);</pre> <p>Also look at <code>Stream#anyMatch(...)</code> method</p>
Finding all sub-directory names in a directory. Using new static methods, <code>Arrays#stream(T[] array)</code>	
<pre>List<String> allDirNames = new ArrayList<>(); for (File file : new File("d:\\").listFiles()) { if(file.isDirectory()){ allDirNames.add(file.getName()); } }</pre>	<pre>List<String> allDirNames = Arrays.stream(new File("d:\\") .listFiles()) .filter(File::isDirectory)</pre>



Modern Java
\$44.95



OCP: Oracle
Professional

\$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](#)




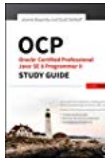



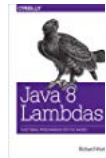
```
}  
}  
System.out.println(allDirNames);
```

```
.map(File::getName)  
.collect(Collectors.toList());  
System.out.println(allDirNames);
```



Discover our latest offers

BOOK NOW

 <p>Modern Java in Action: Lambdas, streams, functional and reactive progra...</p> <p>\$44.05 \$64.00</p> <p>(4)</p>	 <p>Java 8 in Action: Lambdas, Streams, and functional-style programming</p> <p>\$83.69</p> <p>(84)</p>	 <p>Effective Java</p> <p>\$43.86 \$64.00</p> <p>(79)</p>	 <p>OCP: Oracle Certified Prof SE 8 Programmer II Study</p> <p>\$37.53 \$60.00</p> <p>(66)</p>
 <p>OCP Java SE 8 Programmer II Exam Guide (Exam 1Z0-809)</p> <p>\$35.89 \$60.00</p> <p>(13)</p>	 <p>Oracle Certified Professional Java SE 8 Programmer Exam 1Z0-809: A Compre...</p> <p>\$38.24 \$44.00</p> <p>(16)</p>	 <p>Java Pocket Guide: Instant Help for Java Programmers</p> <p>\$13.43 \$49.00</p> <p>(6)</p>	 <p>Java 8 Lambdas: Pragmati Programming</p> <p>\$18.35</p> <p>(33)</p>

See Also

- [Java Collections](#)
- [Java 8 Enhancements](#)
- [Java 8 Stream operations cheat sheet](#)
- [Sequential vs Parallel streams](#)
- [Lazy evaluation](#)
- [Short circuiting operations](#)

- [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 Support](#)
- [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.:](#)

