Lambda Expression : It is an anonymous function

A function contains 4 things :

1. name -------------🡪anonymous function

**2. parameter list -🡪 Most important parts**

**3. body -🡪 Most important parts**

4. return type ------------🡪inferred

The beauty of java is backward compatibility. that means new feature like Lambda expression we can pass in Thread constructor.

* Making more anonymous inner class is bad 🡪 The more you will have more .class file and the jar file size will be very big and results it will take large memory => more garbage collection operation 🡺
* Using Lambda expression no more inner class object is being created i.e no more .class file is created i.e jar file size is reduced
* If we invoke lambda expression it simply becomes **invokedynamic**
* public class Main {  
    
   public static void main(String[] args) {  
   System.*out*.println("Hiii java 8");  
   List<Integer> numbers = Arrays.*asList*(1,2,3,4,5,6,7,8,9,10);  
   //External Iterator  
   /\* for (int i=0;i<numbers.size();i++){  
   System.out.println(numbers.get(i));  
   }\*/  
    
   //External Iterator too  
   /\*for (int i:numbers) {  
   System.out.println(i);  
    
   }\*/  
    
   //Internal Iterator  
    
   /\*numbers.forEach(new Consumer<Integer>() {  
   @Override  
   public void accept(Integer value) {  
   System.out.println(value);  
   }  
   });\*/  
    
   /\* numbers.forEach((Integer value) -> System.out.println(value));\*/  
    
   //Java 8 has type inference finally but only for lambda expression  
   numbers.forEach((value) -> System.*out*.println(value));  
   }

numbers.forEach(System.out::println)

Note : Lambda expression is cute so its our responsibility to make it cue it means we write lambda expression for small logic not for 20 lines of code logic. Two lines may be too many, otherwise it will be very difficult to understand that code , the code will become very noisy and ugly.It will be **AntiPattern.**

**The primary purpose of Functional Interfaces**  
One of the most important uses of Functional Interfaces is that implementations of their abstract method can be passed around as [lambda expressions](https://www.javabrahman.com/java-8/lambda-expressions-java-8-explained-examples/).

**Stream Api :** java.util.stream.Stream is an interface which represents a sequence of elements. The computation operations are composed into stream pipeline which consists a **source, intermediate operations and a terminal operation**.

Streams are lazy and the operations on streams are performed only when terminal operation is initiated and source elements are consumed only if needed.

In most of the stream operations we need to pass lambda expression that must be non-interfering and stateless. Non-interfering means that computational operations do not modify source stream and being stateless means that result should not depend on any state that can change in stream pipeline execution.

While invoking intermediate or terminal operation, stream should be operated on only once and if it is being reused then it will throw IllegalStateException. Streams implement AutoCloseable and need not to close after use.

Streams can execute either sequentially or in parallel and this choice is made while initially creating streams.

Streams vs Collections

Streams and Collections have some similarity but they differ in many ways. Find some points.   
  
1. Collections efficiently manage and allow access to elements whereas Streams do not allow direct manipulation or access to elements. Streams are computed using intermediate and terminal operation that creates new stream.   
  
2. Streams do not store data. They only allow passing the elements through a computational pipeline. The sources of elements in stream are array, list and map.   
  
3. Streams are functional in nature. The function is applied on each element of the stream and produces the result but the source elements are not modified.   
  
4. Stream operations are always divided into intermediate operations and terminal operations. Intermediate operations are always lazy.   
  
5. Streams are unbounded whereas collections can have finite size. The infinite elements can be computed within finite time using streams.   
  
6. While computation the elements of stream are visited only once during life. The elements can be revisited in another instance of stream which will be the output of computation on previous stream instance.

### Parallel and Sequential Stream

Java 8 streams can be computed in parallel and sequential way. Sequential computation is performed one by one in an order. In parallel processing computations are processed simultaneously. In stream parallel processing computations are performed as a pipeline of aggregate operations whereas sequential stream operation is performed as imperative operations. To work with parallel and sequential stream, we need to instantiate stream as parallel and sequential and after that both will be same in coding. We can instantiate stream as follows.

List<String> list = Arrays.asList("A", "B", "C");

list.stream(); //Sequential Stream

list.parallelStream(); //Parallel stream

Collection has also introduced new methods

i.e Collection.stream() and Collection.parallelStream() that is used to obtain sequential and parallel streams in our code.

Ways to obtain Streams Instance

For the streams of primitive data type java 8 provides IntStream, LongStream and DoubleStream class and for the streams of objects, java 8 provide Stream class. There are many ways to obtain the instance of these streams.   
  
1. Using stream() and parallelStream() methods of Collection which is extended by List, Queue , Set etc. For example suppose we have a List, then we can use methods as foolow.   
List.stream() and   
List.parallelStream()   
  
2. In case of Map, streams are obtained as follows   
Map.entrySet().stream() and   
Map.entrySet().parallelStream()   
  
3. Using Arrays.stream method. We can pass array of primitive data types or objects to this method such asArrays.stream(int[] array) or Arrays.stream(Object[] array) etc.   
  
4. Using Stream.of(Object[] array). Here of() is a static method of Stream.   
  
5. The primitive data type stream classes also provide the method to obtain stream such as IntStream.range(int, int) etc.

java.util.stream.Stream API

Now we will discuss here the usability of java.util.stream.Stream API. The methods of Stream class accept Function instance or a lambda expression as parameters. The computation on streams can be performed sequentially or in parallel. Within a single line of code we can perform aggregate operations on the stream of elements.

[Home](https://www.concretepage.com/)  >  [Java 8](https://www.concretepage.com/java/java-8/)

# **Java 8 Stream Tutorial with Example**

By Arvind Rai, October 12, 2016

This page will walk through java 8 Stream tutorial with example. Java 8 has introduced a package java.util.stream that consists the classes that supports functional-style operations on streams of elements. The basic classes of this package are Stream for objects and IntStream, LongStream, DoubleStream for primitive data type integer, long and double respectively. java.util.stream.Stream is an interface which represents a sequence of elements. It supports sequential and aggregate operations. The computation operations are composed into stream pipeline which consists a source, intermediate operations and a terminal operation. Streams are lazy and the operations on streams are performed only when terminal operation is initiated and source elements are consumed only if needed. In most of the stream operations we need to pass lambda expression that must be non-interfering and stateless. Non-interfering means that computational operations do not modify source stream and being stateless means that result should not depend on any state that can change in stream pipeline execution. The parameter passed in stream operation could be an instance of java 8 Function or a lambda expression. While invoking intermediate or terminal operation, stream should be operated on only once and if it is being reused then it will throw IllegalStateException. Streams implement AutoCloseable and need not to close after use but if stream source is IO channel then we need to close it. Streams are backed by collections, arrays or generating functions. Streams can execute either sequentially or in parallel and this choice is made while initially creating streams.

##### Contents

* [Streams vs Collections](https://www.concretepage.com/java/jdk-8/java-8-stream-tutorial-with-example#streams-vs-collections)
* [Parallel and Sequential Stream](https://www.concretepage.com/java/jdk-8/java-8-stream-tutorial-with-example#parallel-and-sequential)
* [Ways to obtain Streams Instance](https://www.concretepage.com/java/jdk-8/java-8-stream-tutorial-with-example#obtain-streams)
* [java.util.stream.Stream API](https://www.concretepage.com/java/jdk-8/java-8-stream-tutorial-with-example#stream-api)

### Streams vs Collections

Streams and Collections have some similarity but they differ in many ways. Find some points.   
  
1. Collections efficiently manage and allow access to elements whereas Streams do not allow direct manipulation or access to elements. Streams are computed using intermediate and terminal operation that creates new stream.   
  
2. Streams do not store data. They only allow passing the elements through a computational pipeline. The sources of elements in stream are array, list and map.   
  
3. Streams are functional in nature. The function is applied on each element of the stream and produces the result but the source elements are not modified.   
  
4. Stream operations are always divided into intermediate operations and terminal operations. Intermediate operations are always lazy.   
  
5. Streams are unbounded whereas collections can have finite size. The infinite elements can be computed within finite time using streams.   
  
6. While computation the elements of stream are visited only once during life. The elements can be revisited in another instance of stream which will be the output of computation on previous stream instance.

### Parallel and Sequential Stream

Java 8 streams can be computed in parallel and sequential way. Sequential computation is performed one by one in an order. In parallel processing computations are processed simultaneously. In stream parallel processing computations are performed as a pipeline of aggregate operations whereas sequential stream operation is performed as imperative operations. To work with parallel and sequential stream, we need to instantiate stream as parallel and sequential and after that both will be same in coding. We can instantiate stream as follows.

List<String> list = Arrays.asList("A", "B", "C");

list.stream(); //Sequential Stream

list.parallelStream(); //Parallel stream

Collection has also introduced new methods i.e Collection.stream() and Collection.parallelStream() that is used to obtain sequential and parallel streams in our code.

### Ways to obtain Streams Instance

For the streams of primitive data type java 8 provides IntStream, LongStream and DoubleStream class and for the streams of objects, java 8 provide Stream class. There are many ways to obtain the instance of these streams.   
  
1. Using stream() and parallelStream() methods of Collection which is extended by List, Queue , Set etc. For example suppose we have a List, then we can use methods as foolow.   
List.stream() and   
List.parallelStream()   
  
2. In case of Map, streams are obtained as follows   
Map.entrySet().stream() and   
Map.entrySet().parallelStream()   
  
3. Using Arrays.stream method. We can pass array of primitive data types or objects to this method such asArrays.stream(int[] array) or Arrays.stream(Object[] array) etc.   
  
4. Using Stream.of(Object[] array). Here of() is a static method of Stream.   
  
5. The primitive data type stream classes also provide the method to obtain stream such as IntStream.range(int, int) etc.   
  
6. Using Stream.iterate(T seed, UnaryOperator<T> f) where **T** is the type of elements and **f** is a function that is applied to the previous element to get new element.   
  
7. Using BufferedReader.lines(). It returns the stream of string.   
  
8. Using java 8 methods of Files such as find(), lines(), walk(). These methods return stream.   
  
9. Using Random class we can obtain streams of random number for primitive data type. The methods ints(), longs() and doubles() of Random class return IntStream, LongStream and DoubleStream respectively.   
  
10. Using BitSet.stream() we obtain stream of indices as IntStream.   
  
11. Using Pattern.splitAsStream(CharSequence input) we obtain the stream of string. This method creates stream for the given input sequence around matches of the pattern.   
  
12. JarFile.stream() returns an ordered Stream over the ZIP file entries.

### java.util.stream.Stream API

Now we will discuss here the usability of java.util.stream.Stream API. The methods of Stream class accept Function instance or a lambda expression as parameters. The computation on streams can be performed sequentially or in parallel. Within a single line of code we can perform aggregate operations on the stream of elements. Find the examples of streams methods.

### Stream.allMatch(), Stream.anyMatch() and Stream.noneMatch()

allMatch(): It returns true if all elements of stream matches the given Predicate.   
anyMatch(): It returns true if any element of stream matches the given Predicate.   
noneMatch(): It returns true if none of the elements of stream matches the given Predicate.   
  
Now find the example.   
**MatchElement.java**

package com.concretepage;

import java.util.Arrays;

import java.util.List;

import java.util.function.Predicate;

public class MatchElement {

public static void main(String[] args) {

Predicate<Integer> p = num -> num % 2 == 0;

List<Integer> list = Arrays.asList(3,5,6);

System.out.println("allMatch:" + list.stream().allMatch(p));

System.out.println("anyMatch:" + list.stream().anyMatch(p));

System.out.println("noneMatch:" + list.stream().noneMatch(p));

}

}

**Output**

allMatch:false

anyMatch:true

noneMatch:false

### Stream.collect()

It performs mutable reduction operation with java 8 Collector. Find the example to sum the integers in a list.   
**StreamCollect.java**

package com.concretepage;

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class StreamCollect {

public static void main(String[] args) {

List<Integer> list = Arrays.asList(3,5,6);

int sum = list.stream().collect(Collectors.summingInt(i->i));

System.out.println("Sum: "+ sum);

}

}

**Output**

Sum: 14

### Stream.concat()

It creates a lazily concatenated stream including all the elements of first stream and followed by next stream.   
**StreamConcat.java**

package com.concretepage;

import java.util.Arrays;

import java.util.List;

import java.util.stream.Stream;

public class StreamConcat {

public static void main(String[] args) {

List<Integer> list1 = Arrays.asList(1,2,3);

List<Integer> list2 = Arrays.asList(4,5,6);

Stream<Integer> resStream = Stream.concat(list1.stream(), list2.stream());

resStream.forEach(s->System.out.print(s+" "));

}

}

**Output**

1 2 3 4 5 6

### Stream.count()

It returns the number of elements in stream.   
**StreamCount.java**

package com.concretepage;

import java.util.Arrays;

import java.util.List;

import java.util.function.Predicate;

public class StreamCount {

public static void main(String[] args) {

Predicate<Integer> p = num -> num % 2 == 0;

List<Integer> list = Arrays.asList(3,4,6);

System.out.println("Count: " + list.stream().filter(p).count());

}

}

**Output**

Count: 2

### Stream.distinct()

It returns stream with distinct elements.   
**StreamDistinct.java**

package com.concretepage;

import java.util.Arrays;

import java.util.List;

public class StreamDistinct {

public static void main(String[] args) {

List<Integer> list = Arrays.asList(3,4,6,6,4);

System.out.println("Distinct Count: " + list.stream().distinct().count());

}

}

**Output**

Distinct Count: 3

### Stream.filter()

It returns the stream with the elements that matches the given Predicate.   
**StreamFilter.java**

package com.concretepage;

import java.util.Arrays;

import java.util.List;

import java.util.function.Predicate;

public class StreamFilter {

public static void main(String[] args) {

Predicate<Integer> p = num -> num % 2 == 0;

List<Integer> list = Arrays.asList(3,4,6);

list.stream().filter(p).forEach(e -> System.out.print(e+" "));

}

}

**Output**

4 6

### Stream.findAny() and Stream.findFirst()

findAny(): It can return any element of the stream.   
findFirst(): It returns first element of the stream and if stream has defined no encounter order, then it may return any element.   
**StreamFindAnyFindFirst.java**

package com.concretepage;

import java.util.Arrays;

import java.util.List;

public class StreamFindAnyFindFirst {

public static void main(String[] args) {

List<String> list = Arrays.asList("G","B","F","E");

String any = list.stream().findAny().get();

System.out.println("FindAny: "+ any);

String first = list.stream().findFirst().get();

System.out.println("FindFirst: "+ first);

}

}

**Output**

FindAny: G

FindFirst: G

### Stream.flatMap()

It returns a stream of object after applying mapping function on each element and then flattens the result.   
**StreamFlatMap.java**

package com.concretepage;

import java.util.Arrays;

public class StreamFlatMap {

public static void main(String[] args) {

Integer[][] data = {{1,2},{3,4},{5,6}};

Arrays.stream(data).flatMap(row -> Arrays.stream(row)).filter(num -> num%2 == 1)

.forEach(s -> System.out.print(s+" "));

}

}

**Output**

1 3 5

flatMapToInt(): It is used with primitive data type int and returns IntStream.   
flatMapToLong(): It is used with primitive data type long and returns LongStream.   
flatMapToDouble(): It is used with primitive data type double and returns DoubleStream .

Note :

public class StreamFlatMap {  
 public static void main(String args []) {  
 int [][] data = {{1,2},{3,4},{5,6},{7,8},{9,10}};  
 Arrays.*stream*(data).flatMapToInt(row -> Arrays.*stream*(row))  
 .filter(n -> n%2 ==0)  
 .forEach(s -> System.*out*.print(s+" "));  
  
 }

### Stream.forEach() and Stream.forEachOrdered()

forEach(): It performs an action on each element of stream.   
forEachOrdered (): It also performs an action on each element of the stream but in encountered order of the stream if defined any.   
**StreamForEach.java**

package com.concretepage;

import java.util.Arrays;

public class StreamForEach {

public static void main(String[] args) {

Integer[] data = {1,2,3,4,5,6,7};

System.out.println("---forEach Demo---");

Arrays.stream(data).filter(num -> num%2 == 1)

.forEach(s -> System.out.print(s+" "));

System.out.println("\n---forEachOrdered Demo---");

Arrays.stream(data).filter(num -> num%2 == 1)

.forEachOrdered(s -> System.out.print(s+" "));

}

}

**Output**

---forEach Demo---

1 3 5 7

---forEachOrdered Demo---

1 3 5 7

### Stream.map()

It returns a stream after applying given function to each element of the stream.   
**StreamMap.java**

package com.concretepage;

import java.util.Arrays;

import java.util.List;

public class StreamMap {

public static void main(String[] args) {

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

list.stream().map(i -> i\*i)

.forEach(s->System.out.print(s+" "));

}

}

**Output**

1 4 9 16

mapToInt(): It returns IntStream after applying the given function.   
mapToLong(): It returns LongStream after applying the given function.   
mapToDouble(): It returns DoubleStream after applying the given function.

### Stream.max() and Stream.min()

max(): It finds maximum element for the given Comparator.   
min(): It finds minimum element for the given Comparator.   
**StreamMaxMin.java**

package com.concretepage;

import java.util.Arrays;

import java.util.Comparator;

import java.util.List;

public class StreamMaxMin {

public static void main(String[] args) {

List<String> list = Arrays.asList("G","B","F","E");

String max = list.stream().max(Comparator.comparing(String::valueOf)).get();

System.out.println("Max:"+ max);

String min = list.stream().min(Comparator.comparing(String::valueOf)).get();

System.out.println("Min:"+ min);

}

}

**Output**

Max:G

Min:B

### Stream.peek()

It is an intermediate operation. It returns a new stream which consists all the elements of stream after applying the Consumer.   
**StreamPeek.java**

package com.concretepage;

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

public class StreamPeek {

public static void main(String[] args) {

List<String> list = Arrays.asList("A","B","C");

list.stream().peek(s->System.out.println(s+s)).collect(Collectors.toList());

}

}

**Output**

AA

BB

CC

### Stream.reduce()

It performs reduction on stream elements using a start value and accumulation function.   
**StreamReduce.java**

package com.concretepage;

import java.util.Arrays;

public class StreamReduce {

public static void main(String[] args) {

int[] array = {3,5,10,15};

int sum = Arrays.stream(array).reduce(0, (x,y) -> x+y);

System.out.println("Sum:"+ sum);

}

}

**Output**

Sum:33

### Stream.sorted()

It returns a stream sorted with given Comparator.   
**StreamSorted.java**

package com.concretepage;

import java.util.Comparator;

import java.util.HashMap;

import java.util.Map;

public class StreamSorted {

public static void main(String[] args) {

Map<Integer, String> map = new HashMap<>();

map.put(1, "BBBB");

map.put(2, "AAAA");

map.put(3, "CCCC");

System.out.println("---Sort by Map Value---");

map.entrySet().stream().sorted(Comparator.comparing(Map.Entry::getValue))

.forEach(e -> System.out.println("Key: "+ e.getKey() +", Value: "+ e.getValue()));

}

}

**Output**

---Sort by Map Value---

Key: 2, Value: AAAA

Key: 1, Value: BBBB

Key: 3, Value: CCCC

import java.util.Comparator;  
import java.util.HashMap;  
import java.util.Map;  
  
public class StreamSorted {  
 public static void main(String args []) {  
 Map<Integer,String> map = new HashMap<>();  
 map.put(10,"AAA");  
 map.put(15,"Deepak");  
 map.put(15,"Deepak");  
 map.put(20,"Dksi");  
 map.put(13,"bharat");  
  
 System.*out*.println("------ order by key -----");  
 map.entrySet().stream().sorted(Comparator.*comparing*(Map.Entry::getKey))  
 .forEach(e -> System.*out*.print("Key is : " + e.getKey() + "Value is : " + e.getValue()));  
  
 System.*out*.println("------ order by value -----");  
 map.entrySet().stream().sorted(Comparator.*comparing*(Map.Entry::getValue))  
 .forEach(e -> System.*out*.print("Key is : " + e.getKey() + "Value is : " + e.getValue()));  
 }  
}

### O/P : ------ order by key -----

### Key is : 10 Value is : AAA

### Key is : 13 Value is : bharat

### Key is : 15 Value is : Deepak

### Key is : 20 Value is : Dksi

### ------ order by value -----

### Key is : 10 Value is : AAA

### Key is : 15 Value is : Deepak

### Key is : 20 Value is : Dksi

### Key is : 13 Value is : bharat

### Stream.toArray()

It returns an array containing the elements of stream.   
**StreamToArray**

package com.concretepage;

import java.util.Arrays;

import java.util.List;

public class StreamToArray {

public static void main(String[] args) {

List<String> list = Arrays.asList("A", "B", "C", "D");

Object[] array = list.stream().toArray();

System.out.println("Length of array: "+array.length);

}

}

**Output**

Length of array: 4