### Java 8 Stream API

A **Stream in Java** can be defined as a sequence of elements from a source that supports aggregate operations on them. The source here refers to a [Collections](https://howtodoinjava.com/java-collections/) or [Arrays](https://howtodoinjava.com/java-array/) who provides data to a Stream.

Stream keeps the ordering of the data as it is in the source. The **aggregate operations** or **bulk operations** are operations which allow us to express common manipulations on stream elements easily and clearly.

Before going ahead, it is important to learn that **Java 8 Streams** are designed in such a way that **most of the stream operations returns streams only**. This help us creating chain of the stream operations. This is called as **pipe-lining**. I will use this term multiple times in this post, so keep it in mind.

#### 1. Java Stream vs. Collection

All of us have watch online videos on youtube or some other such website. When you start watching video, a small portion of file is first loaded into your computer and start playing. You don’t need to download complete video before start playing it. This is called streaming. I will try to relate this concept with respect to collections and differentiate with Streams.

At the basic level, the difference between Collections and Streams has to do with when things are computed. A **Collection is an in-memory data structure**, which holds all the values that the data structure currently has—every element in the Collection has to be computed before it can be added to the Collection. A **Stream is a conceptually fixed data structure, in which elements are computed on demand**. This gives rise to significant programming benefits. The idea is that a user will extract only the values they require from a Stream, and these elements are only produced—invisibly to the user—as and when required. This is a form of a producer-consumer relationship.

In java, java.util.Stream represents a stream on which one or more operations can be performed. Stream **operations are either intermediate or terminal**. While **terminal operations return a result of a certain type**, **intermediate operations return the stream itself** so you can chain multiple method calls in a row. Streams are created on a source, e.g. a java.util.Collection like lists or sets (maps are not supported). Stream operations can either be executed sequential or parallel.

Based on above points, if we list down the various characteristics of Stream, they will be as follows:

Not a data structure

Designed for lambdas

Do not support indexed access

Can easily be outputted as arrays or lists

Lazy access supported

Parallelizable

2. Different ways to create streams

Below is the most popular different ways to build streams from collections.

2.1. Stream.of(val1, val2, val3….)

|  |
| --- |
| public class StreamBuilders {       public static void main(String[] args)       {           Stream<Integer> stream = Stream.of(1,2,3,4,5,6,7,8,9);           stream.forEach(p -> System.out.println(p));       }} |

2.2. Stream.of(arrayOfElements)

|  |
| --- |
| public class StreamBuilders  {       public static void main(String[] args)       {           Stream<Integer> stream = Stream.of( new Integer[]{1,2,3,4,5,6,7,8,9} );           stream.forEach(p -> System.out.println(p));       }  } |

2.3. List.stream()

|  |
| --- |
| public class StreamBuilders  {       public static void main(String[] args)       {           List<Integer> list = new ArrayList<Integer>();            for(int i = 1; i< 10; i++){               list.add(i);           }           Stream<Integer> stream = list.stream();           stream.forEach(p -> System.out.println(p));       }  } |

2.4. Stream.generate() or Stream.iterate()

|  |
| --- |
| public class StreamBuilders  {       public static void main(String[] args)       {           Stream<Date> stream = Stream.generate(() -> { return new Date(); });           stream.forEach(p -> System.out.println(p));       }  } |

2.5. String chars or String tokens

|  |
| --- |
| public class StreamBuilders  {       public static void main(String[] args)       {          IntStream stream = "12345\_abcdefg".chars();          stream.forEach(p -> System.out.println(p));                   //OR                   Stream<String> stream = Stream.of("A$B$C".split("\\$"));          stream.forEach(p -> System.out.println(p));       }} |

There are some more ways apart from above list such as using **Stream.Buider** or using intermediate operations. We will learn about them in separate posts time to time.

3. Convert streams to collections

I should rather say converting stream to other data structures.

Please note that it is not a true conversion. It’s just collecting the elements from the stream into a collection or array.

3.1. Convert Stream to List – Stream.collect( Collectors.toList() )

|  |
| --- |
| public class StreamBuilders {       public static void main(String[] args){           List<Integer> list = new ArrayList<Integer>();           for(int i = 1; i< 10; i++){               list.add(i);           }           Stream<Integer> stream = list.stream();           List<Integer> evenNumbersList = stream.filter(i -> i%2 == 0).collect(Collectors.toList());           System.out.print(evenNumbersList);       }} |

3.2. Convert Stream to array – Stream.toArray( EntryType[]::new )

|  |
| --- |
| public class StreamBuilders {       public static void main(String[] args){           List<Integer> list = new ArrayList<Integer>();           for(int i = 1; i< 10; i++){               list.add(i);           }           Stream<Integer> stream = list.stream();           Integer[] evenNumbersArr = stream.filter(i -> i%2 == 0).toArray(Integer[]::new);           System.out.print(evenNumbersArr);       }  } |

There are plenty of other ways also to collect stream into set, map or into multiple ways. Just go through **Collectors** class and try to keep them in mind.

4. Core stream operations

Stream abstraction have a long list of useful functions for you. I am not going to cover them all, but I plan here to list down all most important ones, which you must know first hand.

Before moving ahead, lets build a collection of String beforehand. We will build out example on this list, so that it is easy to relate and understand.

|  |
| --- |
| List<String> memberNames = new ArrayList<>();  memberNames.add("Amitabh");  memberNames.add("Shekhar");  memberNames.add("Aman");  memberNames.add("Rahul");  memberNames.add("Shahrukh");  memberNames.add("Salman");  memberNames.add("Yana");  memberNames.add("Lokesh"); |

These core methods have been divided into 2 parts given below:

4.1. Intermediate operations

**Intermediate operations return the stream itself** so you can chain multiple method calls in a row. Let’s learn important ones.

4.1.1. Stream.filter()

Filter accepts a predicate to filter all elements of the stream. This operation is intermediate which enables us to call another stream operation (e.g. forEach) on the result.

|  |
| --- |
| memberNames.stream().filter((s) -> s.startsWith("A"))                      .forEach(System.out::println);    Output:  Amitabh  Aman |

4.1.2. Stream.map()

The intermediate operation map converts each element into another object via the given function. The following example converts each string into an upper-cased string. But you can also use map to transform each object into another type.

|  |
| --- |
| memberNames.stream().filter((s) -> s.startsWith("A"))                       .map(String::toUpperCase)                       .forEach(System.out::println);  Output:  AMITABH  AMAN |

4.1.2. Stream.sorted()

Sorted is an intermediate operation which returns a sorted view of the stream. The elements are sorted in natural order unless you pass a custom Comparator.

|  |
| --- |
| memberNames.stream().sorted()                      .map(String::toUpperCase)                      .forEach(System.out::println);  Output:  AMAN  AMITABH  LOKESH  RAHUL  SALMAN  SHAHRUKH  SHEKHAR  YANA |

Keep in mind that sorted does only create a sorted view of the stream without manipulating the ordering of the backed collection. The ordering of memberNames is untouched.

4.2. Terminal operations

**Terminal operations return a result of a certain type** instead of again a Stream.

4.2.1. Stream.forEach()

This method helps in iterating over all elements of a stream and perform some operation on each of them. The operation is passed as lambda expression parameter.

|  |
| --- |
| memberNames.forEach(System.out::println); |

4.2.2. Stream.collect()

collect() method used to receive elements from a steam and store them in a collection and mentioned in parameter function.

|  |
| --- |
| List<String> memNamesInUppercase = memberNames.stream().sorted()                              .map(String::toUpperCase)                              .collect(Collectors.toList());    System.out.print(memNamesInUppercase);    Outpout: [AMAN, AMITABH, LOKESH, RAHUL, SALMAN, SHAHRUKH, SHEKHAR, YANA] |

4.2.3. Stream.match()

Various matching operations can be used to check whether a certain predicate matches the stream. All of those operations are terminal and return a boolean result.

|  |
| --- |
| boolean matchedResult = memberNames.stream()                      .anyMatch((s) -> s.startsWith("A"));   System.out.println(matchedResult);   matchedResult = memberNames.stream()  .allMatch((s) -> s.startsWith("A"));   System.out.println(matchedResult);   matchedResult = memberNames.stream().noneMatch((s) -> s.startsWith("A"));   System.out.println(matchedResult);   Output:   true  false  false |

4.2.4. Stream.count()

Count is a terminal operation returning the number of elements in the stream as a long.

|  |
| --- |
| long totalMatched = memberNames.stream()                      .filter((s) -> s.startsWith("A"))   .count();   System.out.println(totalMatched);   Output: 2 |

4.2.5. Stream.reduce()

This terminal operation performs a reduction on the elements of the stream with the given function. The result is an Optional holding the reduced value.

|  |
| --- |
| Optional<String> reduced = memberNames.stream()                      .reduce((s1,s2) -> s1 + "#" + s2);    reduced.ifPresent(System.out::println);   Output: Amitabh#Shekhar#Aman#Rahul#Shahrukh#Salman#Yana#Lokesh |

5. Stream short-circuit operations

Though, stream operations are performed on all elements inside a collection satisfying a predicate, It is often desired to break the operation whenever a matching element is encountered during iteration. In external iteration, you will do with if-else block. In internal iteration, there are certain methods you can use for this purpose. Let’s see example of two such methods:

5.1. Stream.anyMatch()

This will return true once a condition passed as predicate satisfy. It will not process any more elements.

|  |
| --- |
| boolean matched = memberNames.stream()        .anyMatch((s) -> s.startsWith("A"));   System.out.println(matched);   Output: true |

5.2. Stream.findFirst()

It will return first element from stream and then will not process any more element.

|  |
| --- |
| String firstMatchedName = memberNames.stream() .filter((s) -> s.startsWith("L")) .findFirst().get();   System.out.println(firstMatchedName);   Output: Lokesh |

6. Parallelism in Java Steam

With the Fork/Join framework added in Java SE 7, we have efficient machinery for implementing parallel operations in our applications. But implementing this framework is itself a complex task; and if not done right; is a source of complex multi-threading bugs having potential to crash the application. With the introduction of internal iteration, we got the possibility of operations to be done in parallel.

To enable parallelism, all you have to do is to create a parallel stream, instead of sequential stream. And to surprise you, this is really very easy. In any of above listed stream examples, anytime you want to particular job using multiple threads in parallel cores, all you have to call method **parallelStream()** method instead of stream() method.

|  |
| --- |
| public class StreamBuilders {       public static void main(String[] args){          List<Integer> list = new ArrayList<Integer>();           for(int i = 1; i< 10; i++){               list.add(i);           }           //Here creating a parallel stream           Stream<Integer> stream = list.parallelStream();           Integer[] evenNumbersArr = stream.filter(i -> i%2 == 0).toArray(Integer[]::new);           System.out.print(evenNumbersArr);       }} |

A key driver for this work is making parallelism more accessible to developers. While the Java platform provides strong support for concurrency and parallelism already, developers face unnecessary impediments in migrating their code from sequential to parallel as needed. Therefore, it is important to encourage idioms that are both sequential- and parallel-friendly. This is facilitated by shifting the focus towards describing what computation should be performed, rather than how it should be performed.

It is also important to strike the balance between making parallelism easier but not going so far as to make it invisible. Making parallelism transparent would introduce non-determinism and the possibility of data races where users might not expect it.

This is all what I wanted to share regarding basics of **Stream abstraction introduced in java 8**. I will be talking about various other things related to Streams in future posts.

Java Stream filter() example

Learn to use **Stream.filter(Predicate condition)** method to traverse all the elements and filter out all elements which do not match a given condition through [Predicate](https://howtodoinjava.com/java8/how-to-use-predicate-in-java-8/) argument.

1. Stream filter() method

This is a **intermediate** operation.

Returns a stream consisting of the elements of this stream that match the given predicate.

The filter() argument shall be **stateless predicate** to apply to each element to determine if it should be included.

Predicate is a [functional interface](https://howtodoinjava.com/java8/functional-interface-tutorial/). So, we can also pass [lambda expression](https://howtodoinjava.com/java8/lambda-expressions/) also.

It **returns a new stream** so we can use other operations applicable to any stream.

2. Stream.filter() method syntax

The stream() method syntax is as follows:

|  |
| --- |
| Syntax |
| Stream<T> filter(Predicate<? super T> condition) |

Predicate is a functional interface and represents the condition to filter out the non-matching elements from the stream.

3. Java Stream filter() examples

3.1. Find all even numbers – Lambda expression

Java example to iterate over stream of integers and print only even numbers.

|  |
| --- |
| Find even numbers in stream |
| import java.util.Arrays;  import java.util.List;   public class Main  {      public static void main(String[] args)      {          List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);           list.stream()            .filter(n -> n % 2 == 0)            .forEach(System.out::println);    }  } |
| Program output.  Console |
| 2  4  6  8  10 |

3.2. Find all even numbers – Predicate class

Java example to iterate over stream of integers and print only even numbers. This example uses Predicate class in place of lambda expression, though both are same things.

|  |
| --- |
| Find even numbers in stream |
| public class Main  {      public static void main(String[] args)      {          List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);                   Predicate<Integer> condition = new Predicate<Integer>()          {              @Override              public boolean test(Integer n) {                  if (n % 2 == 0) {                      return true;                  }                  return false;              }          };           list.stream().filter(condition).forEach(System.out::println);      }} |

Program output.

|  |
| --- |
| Console |
| 2  4  6  8  10 |

3.3. Filter even numbers and collect into new list

We can use the **collect()** method to collect the stream of even numbers and converts it into a list.

|  |
| --- |
| Find even numbers in stream and collect to a new list |
| public class Main  {      public static void main(String[] args)      {          List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);           List<Integer> evenNumbers = list.stream() .filter(n -> n % 2 == 0)  .collect(Collectors.toList());                   System.out.println(evenNumbers);      }} |

Program output.

|  |
| --- |
| Console |
| [2, 4, 6, 8, 10] |

3.4. Filter even numbers and get squares

We can use the **map()** method to collect the stream of even numbers and then convert each number to it’s square before collecting it to a new list.

|  |
| --- |
| Find even numbers in stream and collect the squares |
| public class Main  {      public static void main(String[] args)      {          List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);          List<Integer> evenNumbers = list.stream().filter(n -> n % 2 == 0).map(n -> n \* n) .collect(Collectors.toList());            System.out.println(evenNumbers);      }} |
| o/p: [4, 16, 36, 64, 100]  Java Stream map() example  Learn to use **Stream map()** method which produces one output value of a different type 'X' for each input value of type 'Y'.  Stream.map() converts Stream<X> to Stream<Y>. For each X, a Y is created and put in new stream.  1. Stream.map() method  1.1. Description  It is an **intermediate** operation and return another stream as method output return value.  Returns a stream consisting of the results of applying the given function to the elements of this stream.  The map operation takes a Function, which is called for each value in the input stream and produces one result value, which is sent to the output stream.  The function used for transformation in map() is a stateless function and returns only a **single value**.  map() method is used when we want to convert a stream of X to stream of Y.  Each mapped stream is closed after its contents have been placed into new output stream.  map() operation does not flatten the stream as flatMap() operation does.  1.2. Syntax  Stream map() method has following syntax.   |  | | --- | | map() method syntax | | <R> Stream<R> map(Function<? super T,? extends R> mapper) |   Stream interface has three more similar methods which produce IntStream, LongStream and DoubleStream respectively after the map operation. If the streams created after map() operations are given return types then consider using these functions directly.   |  | | --- | | Similar methods | | IntStream mapToInt(ToIntFunction<? super T> mapper)  LongStream mapToLong(ToLongFunction<? super T> mapper)  DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper) |   2. Java Stream map() example  2.1. Stream of strings to Stream of Integers  Java 8 example of **Stream.map()** function to convert a stream of strings to stream of integers. Here the function [Integer::valueOf()](https://howtodoinjava.com/java/string/convert-string-to-int/) takes one string from stream at a time, and convert the string to integer and put in another stream of integers, which is then collected using Collectors.toList().   |  | | --- | | Stream map() example | | public class Main  {      public static void main(String[] args)      {          List<String> listOfStrings = Arrays.asList("1", "2", "3", "4", "5");          List<Integer> listOfIntegers = listOfStrings.stream().map(Integer::valueOf).collect(Collectors.toList());          System.out.println(listOfIntegers);      }} |   Program output.   |  | | --- | | Console | | [1, 2, 3, 4, 5] |   2.2. Stream of employees to stream of distinct salaries  Java example to find all possible [distinct](https://howtodoinjava.com/java8/java-stream-distinct-examples/) salaries for a list of employees.   |  | | --- | | Stream map() example 2 | | public class Main  {      public static void main(String[] args)      {          List<Employee> employeesList = Arrays.asList(                                              new Employee(1, "Alex", 100),                                              new Employee(2, "Brian", 100),                                              new Employee(3, "Charles", 200),                                              new Employee(4, "David", 200),                                              new Employee(5, "Edward", 300),                                              new Employee(6, "Frank", 300)                                          );                    List<Double> distinctSalaries = employeesList.stream()                                  .map( e -> e.getSalary() ) .distinct() .collect(Collectors.toList());            System.out.println(distinctSalaries);      }} | | o/p: [100.0, 200.0, 300.0]  Java Stream flatMap() example  Learn to use **Stream flatMap()** method which is used to flatten a stream of collections to a stream of elements combined from all collections.  The **flatMap()** operation has the effect of applying a one-to-many transformation to the elements of the stream, and then flattening the resulting elements into a new stream.  Stream.flatMap() helps in converting Collection<Collection<T>> to Collection<T>.  **flatMap() = map() + Flattening**  1. What is flattening  Flattening is referred by converting several lists of lists, and merge all those lists to create single list containing all the elements from all the lists.   |  | | --- | | Flattening example | | Before flattening   : [[1, 2, 3], [4, 5], [6, 7, 8]]  After flattening    : [1, 2, 3, 4, 5, 6, 7, 8] |   2. Stream.flatMap() method  2.1. Syntax  Stream flatMap() method has following syntax.   |  | | --- | | flatMap() method syntax | | <R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper) |   Stream interface has three more similar methods which produce IntStream, LongStream and DoubleStream respectively after the flatMap operation. If the streams created after flatMap() operations have given return types then consider using these functions directly.   |  | | --- | | Similar methods | | IntStream flatMapToInt(Function<? super T,? extends IntStream> mapper)  LongStream flatMapToLong(Function<? super T,? extends LongStream> mapper)  DoubleStream flatMapToDouble(Function<? super T,? extends DoubleStream> mapper) |   2.2. Description  It is an **intermediate** operation and return another stream as method output return value.  Returns a stream consisting of the results of replacing each element of the given stream with the contents of a mapped stream produced by applying the provided mapping function to each element.  The function used for transformation in flatMap() is a stateless function and returns only a stream of new values.  Each mapped stream is closed after its contents have been placed into new stream.  flatMap() operation flattens the stream; opposite to map() operation which does not apply flattening.  3. Java Stream flatMap() example  3.1. Convert list of lists to single list  Java 8 example of **Stream.flatMap()** function to get a single list containing all elements from a list of lists.   |  | | --- | | Stream flatMap() example | | public class Main  {      public static void main(String[] args)      {          List<Integer> list1 = Arrays.asList(1,2,3);          List<Integer> list2 = Arrays.asList(4,5,6);          List<Integer> list3 = Arrays.asList(7,8,9);            List<List<Integer>> listOfLists = Arrays.asList(list1, list2, list3);            List<Integer> listOfAllIntegers = listOfLists.stream().flatMap(x -> x.stream())                                      .collect(Collectors.toList());            System.out.println(listOfAllIntegers);      }  } |   Program output.   |  | | --- | | Console | | [1, 2, 3, 4, 5, 6, 7, 8, 9] |   3.2. Convert array of arrays to single list  Java 8 example of **Stream.flatMap()** function to get a single list containing all elements from a array of arrays.   |  | | --- | | Stream flatMap() example | | public class Main  {      public static void main(String[] args)      {          String[][] dataArray = new String[][]{{"a", "b"}, {"c", "d"}, {"e", "f"}, {"g", "h"}};                   List<String> listOfAllChars = Arrays.stream(dataArray)                                      .flatMap(x -> Arrays.stream(x)).collect(Collectors.toList());            System.out.println(listOfAllChars);      }} |   Program output.   |  | | --- | | Console | | [a, b, c, d, e, f, g, h] |   Java Stream Distinct Examples  Using Java 8 stream API, you can use stream.distinct() method to filter or collect all [distinct elements](https://howtodoinjava.com/puzzles/java-puzzle-find-all-the-distinct-duplicate-elements/) from a collection. Let’s learn how to **find distinct elements with**[**Java stream API**](https://howtodoinjava.com/java/collections/java-8-tutorial-streams-by-examples/).  Table of Contents  [Filter Distinct Elements](https://howtodoinjava.com/java8/java-stream-distinct-examples/#distinct)  [Filter Distinct Elements by Object Attribute/Property](https://howtodoinjava.com/java8/java-stream-distinct-examples/#distinct-by-property)  1. Find All Distinct Elements  In this example, we have list of strings and we will **find all distinct strings**, collect them into another list using collect() which is java stream terminal operations. After collecting the distinct elements, we will verify it by printing them to console.   |  | | --- | | Find all distinct elements | | Collection<String> list = Arrays.asList("A", "B", "C", "D", "A", "B", "C");    // Get collection without duplicate i.e. distinct only  List<String> distinctElements = list.stream().distinct().collect(Collectors.toList());    //Let's verify distinct elements  System.out.println(distinctElements); |   Output is:  [A, B, C, D]  2. Find distinct objects by object key or attribute  Previous **Java 8 stream distinct example** is very much simple and easy because it deals with **stream of strings**.  In realtime usecases, we will be dealing with **stream of objects** or complex types (representing some system entity) and very less chances are that those will be string constants or **primitive types**.  So let’s extend our example to work with complex types such as classes and apply **filtering on object key**.  2.1. distinctByKey Function  Please note that till date, Java does not have any native API for filtering distinct by object property directly, so we will create a utility function and then use it.   |  | | --- | | Utility function to find distinct by class field | | public static <T> Predicate<T> distinctByKey(Function<? super T, Object> keyExtractor)  {      Map<Object, Boolean> map = new ConcurrentHashMap<>();      return t -> map.putIfAbsent(keyExtractor.apply(t), Boolean.TRUE) == null;  } |   2.2. distinctByKey Example  Above distinctByKey() function uses a [ConcurrentHashMap](https://howtodoinjava.com/java/collections/best-practices-for-using-concurrenthashmap/) instance to find out if there is any existing key with same value- where key is obtained from a function reference.  We will pass the object’s property getter method which will cause the property value to act as the key to map.   |  | | --- | | Java program to find distincts by object key | | public class JavaStreamDistinctExamples  {      public static void main(String[] args)      {          Person lokesh = new Person(1, "Lokesh", "Gupta");          Person brian = new Person(2, "Brian", "Clooney");          Person alex = new Person(3, "Alex", "Kolen");                   //Add some random persons          Collection<Person> list = Arrays.asList(lokesh,brian,alex,lokesh,brian,lokesh);           // Get distinct objects by key          List<Person> distinctElements = list.stream().filter( distinctByKey(p -> p.getId()) )                                              .collect( Collectors.toList() );           // Let's verify distinct elements          System.out.println( distinctElements );      }           //Utility function      public static <T> Predicate<T> distinctByKey(Function<? super T, Object> keyExtractor)      {          Map<Object, Boolean> map = new ConcurrentHashMap<>();          return t -> map.putIfAbsent(keyExtractor.apply(t), Boolean.TRUE) == null;      }}    //Model class  class Person  {      public Person(Integer id, String fname, String lname) {          super();          this.id = id;          this.fname = fname;          this.lname = lname;      }       private Integer id;      private String fname;      private String lname;       //Getters and Setters       @Override      public String toString() {          return "Person [id=" + id + ", fname=" + fname + ", lname=" + lname + "]";      }  } |   Program Output:  [  Person [id=1, fname=Lokesh, lname=Gupta],  Person [id=2, fname=Brian, lname=Clooney],  Person [id=3, fname=Alex, lname=Kolen]  ]  Onve you have identified distinct objects, You can use them to **remove duplicate objects from list**.  Java Stream sorted() method example  Learn to use **Stream sorted()** method to [sort](https://howtodoinjava.com/java-sorting-guide/) a stream of elements in their natural order and also according to the provided [Comparator](https://howtodoinjava.com/java/collections/java-comparator/).  1. Stream sort methods  [Stream](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html) interface provides two methods for sorting the stream elements.  1.1. Stream.sorted()   |  | | --- | | Syntax | | Stream<T> sorted() |   This is a **stateful intermediate operation** which returns a new stream.  Returns a stream consisting of the elements of this stream, sorted according to **natural order**.  If the elements of this stream are not Comparable, a java.lang.ClassCastException may be thrown when the terminal operation is executed.  For ordered streams, the sort is stable.  For unordered streams, no stability guarantees are made.  1.2. Stream.sorted(Comparator comparator)   |  | | --- | | Syntax | | Stream<T> sorted(Comparator<? super T> comparator) |   This is a **stateful intermediate operation** which returns a new stream.  Returns a stream consisting of the elements of this stream, sorted according to the provided Comparator..  For ordered streams, the sort is stable.  For unordered streams, no stability guarantees are made.  2. Java Stream sorted() examples  2.1. Sort stream elements in natural order  Java example to sort a stream of integers in natural order and print.   |  | | --- | | Sort in natural order | | public class Main  {      public static void main(String[] args)      {          List<Integer> list = Arrays.asList(2, 4, 1, 3, 7, 5, 9, 6, 8);           List<Integer> sortedList = list.stream() .sorted() .collect(Collectors.toList());           System.out.println(sortedList);      }} |   Program output.   |  | | --- | | Console | | [1, 2, 3, 4, 5, 6, 7, 8, 9] |   2.2. Sort stream elements using comparator  Java example to sort a stream of integers in **reverse order** using a comparator.   |  | | --- | | Sort in reverse order | | public class Main  {      public static void main(String[] args)      {          List<Integer> list = Arrays.asList(2, 4, 1, 3, 7, 5, 9, 6, 8);           List<Integer> sortedList = list.stream() .sorted(Comparator.reverseOrder())                                      .collect(Collectors.toList());           System.out.println(sortedList);      }} |   Program output.   |  | | --- | | Console | | [9, 8, 7, 6, 5, 4, 3, 2, 1] |   2.3. Sort stream elements using custom comparator  Java example to sort a stream of integers in reverse order using a **custom comparator**.   |  | | --- | | Sort in reverse order | | public class Main  {      public static void main(String[] args)      {          List<Integer> list = Arrays.asList(2, 4, 1, 3, 7, 5, 9, 6, 8);           Comparator<Integer> reverseComparator = new Comparator<Integer>() {              @Override              public int compare(Integer i1, Integer i2) {                  return i2.compareTo(i1);              }          };            List<Integer> sortedList = list.stream() .sorted(reverseComparator) .collect(Collectors.toList());           System.out.println(sortedList);      }  } |   Program output.   |  | | --- | | Console | | [9, 8, 7, 6, 5, 4, 3, 2, 1] |   2.4. Sort stream elements using lambda expression  Java example to sort a stream of integers in reverse order using [lambda expression](https://howtodoinjava.com/java8/lambda-expressions/) to specify the comparison logic.   |  | | --- | | Sort using lambda expression | | public class Main  {      public static void main(String[] args)      {          List<Integer> list = Arrays.asList(2, 4, 1, 3, 7, 5, 9, 6, 8);           List<Integer> sortedList = list.stream() .sorted( (i1, i2) -> i2.compareTo(i1) )                      .collect(Collectors.toList());           System.out.println(sortedList);      }  } |   Program output.   |  | | --- | | Console | | [9, 8, 7, 6, 5, 4, 3, 2, 1] |   Java Stream peek() method example  [Java 8](https://howtodoinjava.com/java-8-tutorial/) [Stream](https://howtodoinjava.com/java8/java-streams-by-examples/) interface has peek(Consumer action) method which returns a new stream consists of all the elements of original stream after applying the method argument [Consumer](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html) action.  Using Stream.peek() without any **terminal operation** does nothing.  1. Stream.peek() – Syntax   |  | | --- | | Method syntax | | Stream<T> peek(Consumer<? super T> action) |   Here action is a **non-interfering** action to perform on the elements as they are consumed from the stream. This method returns a new stream consist of elements of the original stream.  2. Stream.peek() – Description  Stream.peek() method is an intermediate operation.  It returns a stream consisting of the elements of current stream.  It additionally perform the provided action on each element as elements.  For parallel stream pipelines, the action may be called at whatever time and in whatever thread the element is made available by the upstream operation.  If the action modifies shared state, it is itself responsible for providing the required synchronization.  This method exists mainly to support **debugging**, where we want to see the elements as they flow past a certain point in a pipeline.  3. Java Stream peek() example  3.1. Without terminal operation  As mentioned above, Stream.peek() without any terminal operation does nothing.   |  | | --- | | Stream.peek() without terminal operation | | List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);    list.stream()      .peek( System.out::println );       //prints nothing |   Program output.   |  | | --- | | Console | |  |   3.2. With terminal operation  Stream.peek() with the terminal operation perform the debug operation as well.   |  | | --- | | Stream.peek() without terminal operation | | List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);    List<Integer> newList = list.stream()                              .peek(System.out::println)                              .collect(Collectors.toList());    System.out.println(newList); |   Program output.   |  | | --- | | Console | | 1  2  3  4  5  [1, 2, 3, 4, 5] |   4. Conclusion  **Stream.peek()** method can be a useful in visualizing how stream operations behave and understanding the implications and interactions of complex intermediate stream operations.  Though it is entirely possible to alter the inner state of elements in the stream, but it is never recommended and shall be avoided.  Java Stream limit() method example  By Lokesh Gupta | Filed Under: [Java 8](https://howtodoinjava.com/java8/)  We can use **Stream.limit(long maxSize)** method to retrieve elements while they must not be greater than a certain maximum count. limit() method returns a [stream](https://howtodoinjava.com/java8/java-streams-by-examples/) consisting of the elements of this stream, truncated to be no longer than maxSize in length.  limit(N) method returns first N elements in the **encounter order**.  1. Stream.limit() – Syntax   |  | | --- | | Method syntax | | Stream<T> limit(long maxSize) |   Here maxSize the number of elements the stream should be limited to; and method return value is a new stream consist of elements picked from original stream.  2. Stream.limit() – Description  **Stream.limit()** method is [**short-circuiting**](https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html)**intermediate operation**. An intermediate operation is short-circuiting if, when presented with infinite input, it may produce a finite stream as a result. Please note that a terminal operation is short-circuiting if, when presented with infinite input, it may terminate in finite time.  It returns a stream consisting of the maximum elements, no longer than given size in length, of current stream.  Generally, limit() is cheap operation but may sometimes be expensive if maxSize has a large value and stream is parallely processed.  Using an unordered stream source (such as generate(Supplier)) or removing the ordering constraint with [BaseStream.unordered()](https://docs.oracle.com/javase/8/docs/api/java/util/stream/BaseStream.html#unordered--) may result in significant speedups of limit() in parallel pipelines.  limit() returns the first n elements in the encounter order.  3. Stream limit() example  Java example of Stream.limit() method to get first 10 even numbers from **infinite stream of even numbers**.   |  | | --- | | Stream.limit() example | | public class Main {      public static void main(String[] args)      {          Stream<Integer> evenNumInfiniteStream = Stream.iterate(0, n -> n + 2);           List<Integer> newList = evenNumInfiniteStream.limit(10)                                              .collect(Collectors.toList());          System.out.println(newList);      }  } |   Program output.   |  | | --- | | Console | | [0, 2, 4, 6, 8, 10, 12, 14, 16, 18] |   4. Java Stream limit() – Conclusion  **Stream.limit()** method can be a useful in certain cases where we need to get the elements from a stream and count of elements will be determined at runtime. The fact, that it returns the elements in encounter order, makes it very useful for normal business usecases as well.  Java Stream skip() method example  We can use **Stream.skip(long n)** method to skip first 'n' elements from a stream. skip() method returns a [stream](https://howtodoinjava.com/java8/java-streams-by-examples/) consisting of the remaining elements of this stream, after the specified n elements have been skipped.  skip(long n) returns a stream consisting of the remaining elements of the stream after discarding the first 'n' elements of the stream in the **encounter order**.  1. Syntax   |  | | --- | | Method syntax | | Stream<T> skip(long n) |   Here n is the number of leading elements to skip; and method return value is a new stream consist of elements picked from original stream.  The method may throw **IllegalArgumentException** if n is negative.  2. Description  **Stream.skip()** method is [**stateful**](https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html)**intermediate operation**. Stateful operations, such as distinct and sorted, may incorporate state from previously seen elements when processing new elements.  Returns a stream consisting of the remaining elements of the stream after discarding the first n elements of the stream.  If the stream contains fewer than n elements then an empty stream will be returned.  Generally skip() is a cheap operation, it can be quite expensive on ordered parallel pipelines, especially for large values of n.  Using an unordered stream source (such as generate(Supplier)) or removing the ordering constraint with [BaseStream.unordered()](https://docs.oracle.com/javase/8/docs/api/java/util/stream/BaseStream.html#unordered--) may result in significant speedups of skip() in parallel pipelines.  skip() skips the first n elements in the encounter order.  3. Stream skip() example  Java example of Stream.skip() method to skip first 5 even numbers from **infinite stream of even numbers** and then collect the next 10 even numbers.   |  | | --- | | Stream.skip() example | | public class Main {      public static void main(String[] args)      {          Stream<Integer> evenNumInfiniteStream = Stream.iterate(0, n -> n + 2);                   List<Integer> newList = evenNumInfiniteStream.skip(5) .limit(10) .collect(Collectors.toList());          System.out.println(newList);      }  } |   Program output.   |  | | --- | | Console | | [10, 12, 14, 16, 18, 20, 22, 24, 26, 28] |   4. Java Stream skip() – Conclusion  **Stream.skip()** method can be a useful in certain cases where we need to get the elements from a stream but first we need to skip few elements from the stream. The fact, that it returns the elements in encounter order, makes it very useful for normal business usecases as well.  Java Stream.forEach()  Learn to use **Stream.forEach(Consumer action)** method to traverse all the elements of stream and performs an action for each element of this stream.  1. Stream.forEach() method  This is a terminal operation.  After **forEach()** is performed, the stream pipeline is considered consumed, and can no longer be used.  If we need to traverse the same data source again, we must return to the data source to get a new stream.  For parallel stream pipelines, forEach() operation does not guarantee the order of elements in the stream, as doing so would sacrifice the benefit of parallelism.  If the provided action (method argument) accesses shared state between elements, it is responsible for providing the required synchronization.  2. forEach() method syntax  The method syntax is as follows:   |  | | --- | | Syntax | | void forEach(Consumer<? super T> action) |   [Consumer](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/Consumer.html) is a functional interface and action represents the a [**non-interfering action**](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/package-summary.html#NonInterference) to be performed on each element in the stream.  3. Stream forEach() method example  3.1. Traverse and print all elements  Java example to iterate over stream elements and print them.   |  | | --- | | Stream example | | List<Integer> list = Arrays.asList(2, 4, 6, 8, 10);   list.stream().forEach( System.out::println ); |   Program output.   |  | | --- | | Console | | 2  4  6  8  10 |   3.2. Traverse and print all elements in reverse order  Java example to iterate over stream elements and print them in reverse order.   |  | | --- | | Stream example | | List<Integer> list = Arrays.asList(2, 4, 6, 8, 10);   list.stream() .sorted(Comparator.reverseOrder()).forEach(System.out::println); |   Program output.   |  | | --- | | Console | | 10  8  6  4  2  Java Stream forEachOrdered()  Learn to use [**Stream.forEachOrdered(Consumer action)**](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/Stream.html#forEachOrdered(java.util.function.Consumer)) method to traverse all the elements and performs an action for each element of this stream, in the encounter order of the stream if the stream has a defined encounter order.  1. Stream forEachOrdered() method  This is a terminal operation.  After **forEachOrdered()** is performed, the stream pipeline is considered consumed, and can no longer be used.  If we need to traverse the same data source again, we must return to the data source to get a new stream.  Performs an action for each element of this stream, in the **encounter order** of the stream if the stream has a defined encounter order.  Performing the action for one element happens-before performing the action for subsequent elements, but for any given element, the action may be performed in whatever thread the library chooses.  2. forEachOrdered() method syntax  The method syntax is as follows:   |  | | --- | | Syntax | | void forEachOrdered(Consumer<? super T> action) |   [Consumer](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/Consumer.html) is a functional interface and action represents the a **non-interfering action** to be performed on each element in the stream.  3. forEach() vs forEachOrdered()  The behavior of [**forEach()**](https://howtodoinjava.com/java8/java-stream-foreach/) operation is explicitly **non-deterministic**. For parallel stream pipelines, this operation does not guarantee to respect the encounter order of the stream.  While the **forEachOrdered()** operation **respects the the encounter order** of the stream if the stream has a defined encounter order whether it is parallel stream or sequential.  Note that you may lose the benefits of parallelism if you use operations like forEachOrdered() with parallel streams. [[Link](https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html)]   |  | | --- | | Stream forEach() vs forEachOrdered() | | List<Integer> list = Arrays.asList(2, 4, 6, 8, 10);   list.stream().parallel().forEach( System.out::println );        //1   list.stream().parallel().forEachOrdered( System.out::println ); //2 |   In above example, second statement guarantees that output will be 2, 4, 6, 8, 10.   |  | | --- | | Console | | //forEach()  6  10  8  4  2   //forEachOrdered()  2  4  6  8  10 |   4. Stream forEachOrdered() method example  4.1. Traverse and print all elements  Java example to iterate over stream elements and print them.   |  | | --- | | Stream example | | List<Integer> list = Arrays.asList(2, 4, 6, 8, 10);    list.stream().forEachOrdered( System.out::println ); |   Program output.   |  | | --- | | Console | | 2  4  6  8  10 |   4.2. Traverse and print all elements in reverse order  Java example to iterate over stream elements and print them in reverse order.   |  | | --- | | Stream example | | List<Integer> list = Arrays.asList(2, 4, 6, 8, 10);   list.stream() .sorted(Comparator.reverseOrder())          .forEachOrdered(System.out::println); |   Program output.   |  | | --- | | Console | | 10  8  6  4  2 |   Java 8 – Convert stream to array  Learn to **convert stream to array** using **Stream.toArray()** API. This post contains multiple examples for collecting [stream](https://howtodoinjava.com/java8/java-streams-by-examples/) elements to [array](https://howtodoinjava.com/java-array/) under different usecases.  1. Convert stream of strings to array  Java 8 example to **convert stream to array** using using Stream.toArray() method.   |  | | --- | | Convert stream of strings to array | | Stream<String> tokenStream = Arrays.asList("A", "B", "C", "D").stream();  //stream  String[] tokenArray = tokenStream.toArray(String[]::new);   //array  System.out.println(Arrays.toString(tokenArray)); |   Program output.   |  | | --- | | Console | | [A, B, C, D] |   2. Convert infinite stream to array  To convert an [infinite stream](https://howtodoinjava.com/java8/java-infinite-stream/) into array, we must [**limit**](https://howtodoinjava.com/java8/java-stream-limit-method-example/)**the stream** to a finite number of elements.  2.1. Infinite stream to array to ints   |  | | --- | | Convert infinite stream to array | | IntStream infiniteNumberStream = IntStream.iterate(1, i -> i+1);  int[] intArray = infiniteNumberStream.limit(10)                              .toArray();    System.out.println(Arrays.toString(intArray)); |   Program output.   |  | | --- | | Console | | [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] |   2.2. Infinite stream to array to Integers – Boxed stream   |  | | --- | | Convert infinite stream to array | | IntStream infiniteNumberStream = IntStream.iterate(1, i -> i+1);  Integer[] integerArray = infiniteNumberStream.limit(10).boxed() .toArray(Integer[]::new);   System.out.println(Arrays.toString(integerArray)); |   Program output.   |  | | --- | | Console | | [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] |   3. Filter stream and collect elements to array  Sometimes we need to find specific items in stream and then add only those elements to array. Here, we can use [**Stream.filter()**](https://howtodoinjava.com/java8/java-stream-filter-example/) method to pass a [predicate](https://howtodoinjava.com/java8/how-to-use-predicate-in-java-8/) which will return only those elements who match the pre-condition.   |  | | --- | | Convert stream to array - filtered elements | | public class Main  {      public static void main(String[] args)      {          List<Employee> employeeList = new ArrayList<>(Arrays.asList(                              new Employee(1, "A", 100),                              new Employee(2, "B", 200),                              new Employee(3, "C", 300),                              new Employee(4, "D", 400),                              new Employee(5, "E", 500),                              new Employee(6, "F", 600)));                   Employee[] employeesArray = employeeList.stream()                                      .filter(e -> e.getSalary() < 400)                                      .toArray(Employee[]::new);                   System.out.println(Arrays.toString(employeesArray));      }  } |   Program output.   |  | | --- | | Console | | [Employee [id=1, name=A, salary=100.0],  Employee [id=2, name=B, salary=200.0],  Employee [id=3, name=C, salary=300.0]] |   3. Conclusion  We can use Stream’s **toArray()** function is variety of ways to collect stream elements into an array in all usescases.  Java Stream min()  Learn to use **Stream min()** method to select the smallest element in the stream according to the [comparator](https://howtodoinjava.com/java/collections/java-comparator/) provided in its argument.  1. Stream.min() method   |  | | --- | | Method Syntax | | Optional<T> min(Comparator<? super T> comparator) |   This is a **terminal operation**. So stream cannot be used after this method is executed.  Returns the minimum/smallest element of this stream according to the provided Comparator.  This is a special case of a **stream reduction**.  The method argument shall be a non-interfering, stateless Comparator.  The method returns an [Optional](https://howtodoinjava.com/java8/java-8-optionals-complete-reference/) describing the smallest element of this stream, or an empty Optional if the stream is empty.  It may throw [NullPointerException](https://howtodoinjava.com/java/exception-handling/how-to-effectively-handle-nullpointerexception-in-java/) if the smallest element is null.  2. Java Stream min() example  2.1. Smallest element in stream with lambda expression  Java example to find the minimum number from a stream of numbers using comparator as [lambda expression](https://howtodoinjava.com/java8/lambda-expressions/).   |  | | --- | | Select smallest element from stream | | public class Main  {      public static void main(String[] args)      {          List<Integer> list = Arrays.asList(2, 4, 1, 3, 7, 5, 9, 6, 8);          Optional<Integer> minNumber = list.stream().min((i, j) -> i.compareTo(j));          System.out.println(minNumber.get());      }  } |   Program output.   |  | | --- | | Console | | 1 |   2.2. Smallest element in stream with comparator  Java example to find the minimum number from a stream of numbers using [custom comparator](https://howtodoinjava.com/sort/sort-arraylist-objects-comparable-comparator/).   |  | | --- | | Select minimum element from stream | | public class Main  {      public static void main(String[] args)      {          List<Integer> list = Arrays.asList(2, 4, 1, 3, 7, 5, 9, 6, 8);          Comparator<Integer> minComparator = new Comparator<Integer>() {              @Override              public int compare(Integer n1, Integer n2) {                  return n1.compareTo(n2);              }          };            Optional<Integer> minNumber = list.stream().min(minComparator);          System.out.println(minNumber.get());      }} |   Program output.   |  | | --- | | Console | | 1 |   Drop me your questions related to **Stream min() method** in [Java Stream API](https://howtodoinjava.com/java8/java-streams-by-examples/) to **find the smallest element in stream**.  Java Stream max()  Learn to use **Stream max()** method to select the largest element in the stream according to the [comparator](https://howtodoinjava.com/java/collections/java-comparator/) provided in its argument.  1. Stream.max() method   |  | | --- | | Method Syntax | | Optional<T> max(Comparator<? super T> comparator) |   This is a **terminal operation**. So stream cannot be used after this method is executed.  Returns the maximum/largest element of this stream according to the provided Comparator.  This is a special case of a **stream reduction**.  The method argument shall be a non-interfering, stateless Comparator.  The method returns an [Optional](https://howtodoinjava.com/java8/java-8-optionals-complete-reference/) describing the maximum element of this stream, or an empty Optional if the stream is empty.  It may throw [NullPointerException](https://howtodoinjava.com/java/exception-handling/how-to-effectively-handle-nullpointerexception-in-java/) if the maximum element is null.  2. Java Stream max() example  2.1. Largest element in stream with lambda expression  Java example to find the largest number from a stream of numbers using comparator as [lambda expression](https://howtodoinjava.com/java8/lambda-expressions/).   |  | | --- | | Select largest element from stream | | public class Main  {      public static void main(String[] args)      {          List<Integer> list = Arrays.asList(2, 4, 1, 3, 7, 5, 9, 6, 8);          Optional<Integer> maxNumber = list.stream() .max((i, j) -> i.compareTo(j));          System.out.println(maxNumber.get());      }  } |   Program output.   |  | | --- | | Console | | 9 |   2.2. Largest element in stream with comparator  Java example to find the largest number from a stream of numbers using [custom comparator](https://howtodoinjava.com/sort/sort-arraylist-objects-comparable-comparator/).   |  | | --- | | Select largest element from stream | | public class Main  {      public static void main(String[] args)      {          List<Integer> list = Arrays.asList(2, 4, 1, 3, 7, 5, 9, 6, 8);          Comparator<Integer> maxComparator = new Comparator<Integer>() {              @Override              public int compare(Integer n1, Integer n2) {                  return n1.compareTo(n2);              }          };            Optional<Integer> maxNumber = list.stream().max(maxComparator);          System.out.println(maxNumber.get());      }  } |   Program output.   |  | | --- | | Console | | 9 |   Drop me your questions related to **Stream max() method** in [Java Stream API](https://howtodoinjava.com/java8/java-streams-by-examples/) to **find the largest element in stream**.  Java stream count number of elements example  To **count the number of elements in stream in**[**Java 8**](https://howtodoinjava.com/java-8-tutorial/), use either the Stream.count() or Collectors.counting() methods.  1. Java stream count list elements – Stream count() function  The Stream interface has a default method called [count()](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#count--) that returns a **long**. Examples of Stream.count() method. This is a terminal operation.  Java program to count number of elements in a list in java.   |  | | --- | | public static void main(String[] args)  {      long count = Stream.of("how","to","do","in","java").count();      System.out.printf("There are %d elements in the stream %n", count);      count = IntStream.of(1,2,3,4,5,6,7,8,9).count();      System.out.printf("There are %d elements in the stream %n", count);      count = LongStream.of(1,2,3,4,5,6,7,8,9).filter(i -> i%2 == 0).count();      System.out.printf("There are %d elements in the stream %n", count);  } |   Output:   |  | | --- | | Console | | There are 5 elements in the stream  There are 9 elements in the stream  There are 4 elements in the stream |   2. Java stream count list elements – Collectors counting() function  Collectors class has one method [counting()](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html#counting--). It is as a downstream collector. It accept input elements and counts them. If no elements are present, the count is 0.   |  | | --- | | public static void main(String[] args)  {      long count = Stream.of("how","to","do","in","java").collect(Collectors.counting());      System.out.printf("There are %d elements in the stream %n", count);      count = Stream.of(1,2,3,4,5,6,7,8,9).collect(Collectors.counting());      System.out.printf("There are %d elements in the stream %n", count);      count = Stream.of(1,2,3,4,5,6,7,8,9).filter(i -> i%2 == 0).collect(Collectors.counting());      System.out.printf("There are %d elements in the stream %n", count);  } |   Output:   |  | | --- | | Console | | There are 5 elements in the stream  There are 9 elements in the stream  There are 4 elements in the stream |   Java Stream anyMatch() API  Java **Stream anyMatch (Predicate predicate)** is terminal-short-circuiting operation which is used to check if the stream contains any matching element with provided [predicate](https://howtodoinjava.com/java8/how-to-use-predicate-in-java-8/).  1. Stream anyMatch() method  1.1. Syntax   |  | | --- | | Method syntax | | boolean anyMatch(Predicate<? super T> predicate) |   Here predicate a non-interfering, stateless predicate to apply to elements of the stream.  1.2. Description  It is a short-circuiting terminal operation.  It returns whether any elements of this stream match the provided predicate.  May not evaluate the predicate on all elements if not necessary for determining the result. Method returns true as soon as first matching element is encountered.  If the stream is empty then false is returned and the predicate is not evaluated.  2. Java Stream anyMatch() example  Java example of Stream.anyMatch() method to check if any stream element match the method argument predicate.   |  | | --- | | Stream.anyMatch() example | | public class Main  {      public static void main(String[] args)      {          Stream<String> stream = Stream.of("one", "two", "three", "four");          boolean match = stream.anyMatch(s -> s.contains("four"));          System.out.println(match);      //true      }  } |   Program output.   |  | | --- | | Console | | true |   3. Difference between anyMatch() vs contains()  Theoretically, there is no difference between anyMatch() and [contains()](https://howtodoinjava.com/java/collections/arraylist/arraylist-contains/) when we want to check if an element exist in a list.  [Parallelism](https://howtodoinjava.com/java/multi-threading/concurrency-vs-parallelism/) might bring an advantage for really large lists, but we should not casually use the Stream.parallel() every time and there assuming that it may make things faster. In fact, invoking parallel() may bring down the performance for small streams.  4. Conclusion  **Stream.anyMatch()** method can be a useful in certain cases where we need to check if there is at least one element in the stream.  The shorter version list.contains() also does the same thing and can be used instead.  Java Stream allMatch() API  Java **Stream allMatch (Predicate predicate)** is short-circuiting terminal operation which is used to check if all the elements of the stream match the provided [predicate](https://howtodoinjava.com/java8/how-to-use-predicate-in-java-8/).  1. Stream allMatch() method  1.1. Syntax   |  | | --- | | Method syntax | | boolean allMatch(Predicate<? super T> predicate) |   Here predicate a non-interfering, stateless predicate to apply to elements of the stream.  1.2. Description  It is a short-circuiting **terminal** operation.  It returns whether all elements of this stream match the provided predicate.  May not evaluate the predicate on all elements if not necessary for determining the result. Method returns true if all stream elements match the given predicate, else method returns false.  If the stream is empty then true is returned and the predicate is not evaluated.  2. Stream allMatch() example  Java example of Stream.allMatch() method to check if all stream elements does not contain any numeric/digit character.   |  | | --- | | Stream.allMatch() example | | public class Main  {      public static void main(String[] args)      {          Stream<String> stream = Stream.of("one", "two", "three", "four");          boolean match = stream.allMatch(s -> s.contains("\\d+") == false );          System.out.println(match);      //true      }  } |   Program output.   |  | | --- | | Console | | true |   3. Conclusion  **Stream.allMatch()** method can be a useful in certain cases where we need to run a check on all stream elements. For example, we can use **Java Stream allMatch()** function on a stream of employee objects to validate that all employees are above a certain age.  It is **short-circuiting** operation. A terminal operation is short-circuiting if, when presented with infinite input, it may terminate in finite time.  Java Stream noneMatch() API  By Lokesh Gupta | Filed Under: [Java 8](https://howtodoinjava.com/java8/)  Java **Stream noneMatch (Predicate predicate)** is short-circuiting terminal operation which is used to check if no element of the stream match the provided [predicate](https://howtodoinjava.com/java8/how-to-use-predicate-in-java-8/).  1. Stream noneMatch() method  1.1. Syntax   |  | | --- | | Method syntax | | boolean noneMatch(Predicate<? super T> predicate) |   Here predicate a non-interfering, stateless predicate to apply to elements of the stream.  1.2. Description  It is a short-circuiting **terminal** operation.  It returns whether no element of the stream match the provided predicate.  May not evaluate the predicate on all elements if not necessary for determining the result. Method returns true if no stream element match the given predicate, else method returns false.  If the stream is empty then true is returned and the predicate is not evaluated.  It is pretty much opposite to method Stream.allMatch().  2. Stream noneMatch() example  Java example of Stream.noneMatch() method to check if no stream element contain any numeric/digit character.   |  | | --- | | Stream.noneMatch() example | | public class Main  {      public static void main(String[] args)      {          Stream<String> stream = Stream.of("one", "two", "three", "four");          boolean match = stream.noneMatch( s -> s.contains("\\d+") );          System.out.println(match);      //true      }  } |   Program output.   |  | | --- | | Console | | true |   3. Conclusion  **Stream.noneMatch()** method can be a useful in certain cases where we need to run a check on all stream elements. For example, we can use **Java Stream noneMatch()** function on a stream of employee objects to validate that all employees are NOT below a certain age.  It is **short-circuiting** operation. A terminal operation is short-circuiting if, when presented with infinite input, it may terminate in finite time.  Java Stream findFirst() vs findAny() API With Example  Java Stream interface has two methods i.e. **findFirst()** and **findAny()**. Both method looks very much similar but they may behave differently in certain conditions. In this post, learn the **difference between findFirst() and findAny()** methods.  1. Stream findFirst() method  1.1. Description   |  | | --- | | Method syntax | | Optional<T> findFirst() |   This method returns an [Optional](https://howtodoinjava.com/java8/java-8-optionals-complete-reference/) describing the **first element of this stream**. In case of stream has :  **defined encounter order** – first element in encounter order in stream.  **no encounter order** – any element may be returned.  The above theory is vaid for all **sequential and parallel streams** and the behavior of findFirst() will not change.  1.2. Stream findFirst() example   |  | | --- | | Stream.findFirst() API example | | public class Main  {      public static void main(String[] args)      {          //sequential stream          Stream.of("one", "two", "three", "four") .findFirst() .ifPresent(System.out::println);                   //parallel stream          Stream.of("one", "two", "three", "four").parallel() .findFirst() .ifPresent(System.out::println);      }  } |   Program output.   |  | | --- | | Console | | one  one |   2. Stream findAny() method  2.1. Description   |  | | --- | | Method syntax | | Optional<T> findAny() |   This method returns an Optional describing the **any element of this stream**. In case of stream has :  **defined encounter order** – any element may be returned.  **no encounter order** – any element may be returned.  The above theory is vaid for all **sequential and parallel streams** and the behavior of findAny() will not change.  In non-parallel streams, findAny() will return the first element in most of the cases but this behavior is not gauranteed.  **Stream.findAny() method has been introduced for performance gain in case of parallel streams, only.**  2.2. Stream findAny() example   |  | | --- | | Stream.findAny() API example | | import java.util.stream.Stream;   public class Main  {      public static void main(String[] args)      {          //sequential stream          Stream.of("one", "two", "three", "four").findAny().ifPresent(System.out::println);          //parallel stream          Stream.of("one", "two", "three", "four") .parallel() .findAny() .ifPresent(System.out::println);      }  } |   Program output.   |  | | --- | | Console | | one  three |   3. Stream findFirst() vs findAny() – Conclusion  In this post, we learned the difference between findFirst() and findAny() methods in [Java 8 Stream API](https://howtodoinjava.com/java8/java-streams-by-examples/). In non-parallel streams, both may return the first element of the stream in most cases, but findAny() does not offer any guarantee of this behavior.  Use findAny() to get any element from any parallel stream in faster time. Else we can always use findFirst() in most of the cases.  ava Stream findFirst() vs findAny() API With Example  Java Stream interface has two methods i.e. **findFirst()** and **findAny()**. Both method looks very much similar but they may behave differently in certain conditions. In this post, learn the **difference between findFirst() and findAny()** methods.  1. Stream findFirst() method  1.1. Description   |  | | --- | | Method syntax | | Optional<T> findFirst() |   This method returns an [Optional](https://howtodoinjava.com/java8/java-8-optionals-complete-reference/) describing the **first element of this stream**. In case of stream has :  **defined encounter order** – first element in encounter order in stream.  **no encounter order** – any element may be returned.  The above theory is vaid for all **sequential and parallel streams** and the behavior of findFirst() will not change.  1.2. Stream findFirst() example   |  | | --- | | Stream.findFirst() API example | | public class Main  {      public static void main(String[] args)      {          //sequential stream          Stream.of("one", "two", "three", "four") .findFirst() .ifPresent(System.out::println);          //parallel stream          Stream.of("one", "two", "three", "four").parallel().findFirst() .ifPresent(System.out::println);      }  } |   Program output.   |  | | --- | | Console | | one  one |   2. Stream findAny() method  2.1. Description   |  | | --- | | Method syntax | | Optional<T> findAny() |   This method returns an Optional describing the **any element of this stream**. In case of stream has :  **defined encounter order** – any element may be returned.  **no encounter order** – any element may be returned.  The above theory is vaid for all **sequential and parallel streams** and the behavior of findAny() will not change.  In non-parallel streams, findAny() will return the first element in most of the cases but this behavior is not gauranteed.  **Stream.findAny() method has been introduced for performance gain in case of parallel streams, only.**  2.2. Stream findAny() example   |  | | --- | | Stream.findAny() API example | | public class Main  {      public static void main(String[] args)      {          //sequential stream          Stream.of("one", "two", "three", "four") .findAny().ifPresent(System.out::println);          //parallel stream          Stream.of("one", "two", "three", "four") .parallel().findAny() .ifPresent(System.out::println);      }  } |   Program output.   |  | | --- | | Console | | one  three |   3. Stream findFirst() vs findAny() – Conclusion  In this post, we learned the difference between findFirst() and findAny() methods in [Java 8 Stream API](https://howtodoinjava.com/java8/java-streams-by-examples/). In non-parallel streams, both may return the first element of the stream in most cases, but findAny() does not offer any guarantee of this behavior.  Use findAny() to get any element from any parallel stream in faster time. Else we can always use findFirst() in most of the cases.  Java Boxed Stream Example  In [Java 8](https://howtodoinjava.com/java-8-tutorial/), if you want to convert stream of objects to collection, then you can use one of the static methods in the [Collectors](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html) class.   |  | | --- | | //It works perfect !!  List<String> strings = Stream.of("how", "to", "do", "in", "java")                      .collect(Collectors.toList()); |   The same process doesn’t work on streams of primitives, however.   |  | | --- | | //Compilation Error !!  IntStream.of(1,2,3,4,5)      .collect(Collectors.toList()); |   To convert a stream of primitives, you must first [box](https://docs.oracle.com/javase/8/docs/api/java/util/stream/IntStream.html#boxed--) the elements in their wrapper class and then collect them. This type of stream in called **boxed stream**.  1. IntStream – stream of ints  Example to **convert int stream to List of Integers**.   |  | | --- | | //Get the collection and later convert to stream to process elements  List<Integer> ints = IntStream.of(1,2,3,4,5) .boxed() .collect(Collectors.toList());  System.out.println(ints);  //Stream operations directly  Optional<Integer> max = IntStream.of(1,2,3,4,5) .boxed()  .max(Integer::compareTo);  System.out.println(max); |   Program Output:   |  | | --- | | [1, 2, 3, 4, 5]  5 |   2. LongStream – stream of longs  Example to **convert long stream to List of Longs**.   |  | | --- | | List<Long> longs = LongStream.of(1l,2l,3l,4l,5l).boxed() .collect(Collectors.toList());  System.out.println(longs);  Output:  [1, 2, 3, 4, 5] |   3. DoubleStream – stream of doubles  Example to **convert double stream to List of Doubles**.   |  | | --- | | List<Double> doubles = DoubleStream.of(1d,2d,3d,4d,5d)  .boxed()                  .collect(Collectors.toList());  System.out.println(doubles);  Output:  [1.0, 2.0, 3.0, 4.0, 5.0] | | | |

Lambda expressions

A very new and exciting feature, Java 8 ship with it, is **java lambda expressions**. They are not unknown to many of us who have worked on advanced languages like scala.

In fact, if you look at history and try to find out any language improvement in Java in last 2 decades, you will not be able to recall many exciting things. Only few concurrent classes, generics and if you agree then annotations as well, are remarkable additions in java in last decade. Lambda expressions break this drought and feels like a pleasant gift.

1. What is a lambda expression in Java?

In programming, a **Lambda expression (or function) is just an anonymous function, i.e., a function with no name and without being bounded to an identifier**.

In other words, **lambda expressions are nameless functions given as constant values, and written exactly in the place where it’s needed, typically as a parameter to some other function**.

The most important features of Lambda Expressions is that **they execute in the context of their appearance**. So, a similar lambda expression can be executed differently in some other context (i.e. logic will be same but results will be different based on different parameters passed to function).

Above definition is full of keywords and can be understood only when you already know in deep that what is lambda expression. So, once you got better understanding on lambda expressions in next sections, I advise you to re-read above paragraph.

So, it is clear that lambda is some kind of function without name and identifiers. Well, what’s big deal? Why everyone is so excited?

The answer lies in the benefits involved in functional programming over object oriented programming (OOP). Most OOP languages evolve around objects and instances and treat only them their first class citizens. Another important entity i.e. functions take back seat. This is specially true in java, where functions can’t exist outside an object. A function itself does not mean anything in java, until it is related to some object or instance.

But in functional programming, you can define functions, give them reference variables and pass them as method arguments and much more. JavaScript is a good example of this where you can pass callback methods e.g. to Ajax calls. It’s very useful feature and has been lacking in java from beginning. Now with java 8, we can also use these lambda expressions.

1.1. Lambda Syntax

A typical lambda expression syntax will be like this:

|  |
| --- |
| (x, y) -> x + y  //This function takes two parameters                      //and return their sum. |

Now based on type of x and y, method may be used in multiple places. Parameters can match to int, or Integer or simply String also. Based on context, it will either add two integers or concat two strings.

Syntax:

The other possible syntaxes of a lambda expression are:

|  |
| --- |
| either   (parameters) -> expression           //1   or   (parameters) -> { statements; }  //2   or  () -> expression                     //3 |

1.2. Lambda examples

Let’s see some examples of lambda expressions as well:

|  |
| --- |
| (int a, int b) ->    a \* b               // takes two integers and returns their multiplication    (a, b)          ->   a - b               // takes two numbers and returns their difference    () -> 99                                // takes no values and returns 99    (String a) -> System.out.println(a)     // takes a string, prints its value to the console, and returns nothing    a -> 2 \* a                               // takes a number and returns the result of doubling it    c -> { //some complex statements }   // takes a collection and do some procesing |

1.3. Features of Lambda Expressions

Let’s identify some **features of lambda expression**:

A lambda expression can have zero, one or more parameters.

The type of the parameters can be explicitly declared or it can be inferred from the context.

Multiple parameters are enclosed in mandatory parentheses and separated by commas. Empty parentheses are used to represent an empty set of parameters.

When there is a single parameter, if its type is inferred, it is not mandatory to use parentheses. e.g. a -> return a\*a.

The body of the lambda expressions can contain zero, one or more statements.

If body of lambda expression has single statement curly brackets are not mandatory and the return type of the anonymous function is the same as that of the body expression. When there is more than one statement in body than these must be enclosed in curly brackets.

So, we got a brief overview of what the heck is lambda expression. Please have patience if you feel lost and not able to relate, how it can be used in java programming language. We will make out everything in next 30 minutes. So let’s get going.

Before going deep in relation between lambda expressions and java programming, you must know about **functional interfaces** as well. It is just too important.

2. Java 8 functional interface

**Single Abstract Method interfaces** (SAM Interfaces) is not a new concept. It means **interfaces with only one single method**. In java, we already have many examples of such SAM interfaces. From java 8, they will also be **referred as functional interfaces as well**. Java 8, enforces the rule of single responsibility by marking these interfaces with a new annotation i.e. **@FunctionalInterface**.

For example, new definition of Runnable interface is like this:

|  |
| --- |
| @FunctionalInterface  public interface Runnable {      public abstract void run();  } |

If you try to add a new method in any functional interface, compiler would not allow you to do this and will throw compile time error.

So far so good. But, **how they are related to Lambda expressions?** Let’s find out the answer.

We know that Lambda expressions are anonymous functions with no name and they are passed (mostly) to other functions as parameters. Well, in java method parameters always have a type and this type information is looked for to determine which method needs to be called in case of method overloading or even simple method calling. So, basically every lambda expression also must be convertible to some type to be accepted as method parameters. Well, that **type in which lambda expressions are converted, are always of functional interface type**.

Let’s understand it with an example. If we have to write a thread which will print “howtodoinjava” in console then simplest code will be:

|  |
| --- |
| new Thread(new Runnable() {      @Override      public void run() {          System.out.println("howtodoinjava");      }  }).start(); |

If we use the lambda expression for this task then code will be :

|  |
| --- |
| new Thread(              () ->   {                          System.out.println("My Runnable");                      }           ).start(); |

We have also see that Runnable is an functional interface with single method run(). So, when you pass lambda expression to constructor of Thread class, compiler tries to convert the expression into equivalent Runnable code as shown in first code sample. If compiler succeed then everything runs fine, if compiler is not able to convert the expression into equivalent implementation code, it will complain. Here, in above example, lambda expression is converted to type Runnable.

In simple words, a lambda expression is an instance of a functional interface. But a lambda expression itself does not contain the information about which functional interface it is implementing; that information is deduced from the context in which it is used.

3. Java 8 lambda expression examples

I am listing out some code samples which you can read and analyze to how a lambda expression can be used in day to day programming.

**1) Iterating over a List and perform some operations**

|  |
| --- |
| List<String> pointList = new ArrayList();  pointList.add("1");  pointList.add("2");    pointList.forEach(p ->  {                              System.out.println(p);                              //Do more work                          }                   ); |

**2) Create a new runnable and pass it to thread**

|  |
| --- |
| new Thread(      () -> System.out.println("My Runnable");  ).start(); |

**3) Sorting employees objects by their name**

|  |
| --- |
| public class LambdaIntroduction {      public static void main (String[] ar){            Employee[] employees  = {                new Employee("David"),                new Employee("Naveen"),                new Employee("Alex"),                new Employee("Richard")};              System.out.println("Before Sorting Names: "+Arrays.toString(employees));            Arrays.sort(employees, Employee::nameCompare);            System.out.println("After Sorting Names "+Arrays.toString(employees));        }  }    class Employee {  String name;      Employee(String name) {      this.name = name;    }    public static int nameCompare(Employee a1, Employee a2) {      return a1.name.compareTo(a2.name);    }    public String toString() {      return name;    }}  Output:  Before Sorting Names: [David, Naveen, Alex, Richard]  After Sorting Names [Alex, David, Naveen, Richard] |

**4) Adding an event listener to a GUI component**

|  |
| --- |
| JButton button =  new JButton("Submit");  button.addActionListener((e) -> {      System.out.println("Click event triggered !!");  }); |

Java 8 – Functional Interfaces

Learn about Java 8 functional interfaces and the rules around one abstract method permitted in one interface. Learn to add more methods via default methods in functional interfaces.

Table of Contents

[1. What is functional interface](https://howtodoinjava.com/java8/functional-interface-tutorial/#1)

[2. Do's and Don't's in functional interfaces](https://howtodoinjava.com/java8/functional-interface-tutorial/#2)

1. What is functional interface

Functional interfaces are new additions in [**java 8**](https://howtodoinjava.com/category/java8/) which **permit exactly one abstract method inside them**. These interfaces are also called **Single Abstract Method interfaces (SAM Interfaces)**.

In Java 8, functional interfaces can be represented using lambda expressions, method reference and constructor references as well.

Java 8 introduces an annotation i.e. **@FunctionalInterface** too, which can be used for compiler level errors when the interface you have annotated violates the contracts of exactly one abstract method.

Let’s build our first functional interface:

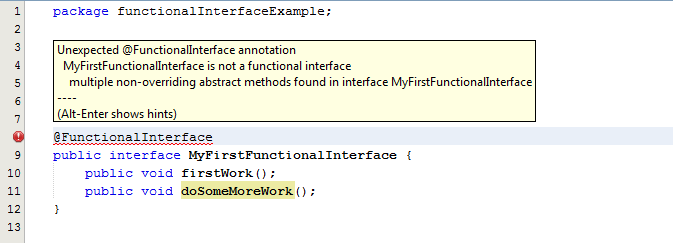
|  |
| --- |
| Functional Interface Definition |
| @FunctionalInterface  public interface MyFirstFunctionalInterface  {      public void firstWork();  } |

Let’s try to add another abstract method:

|  |
| --- |
| @FunctionalInterface  public interface MyFirstFunctionalInterface  {      public void firstWork();      public void doSomeMoreWork();   //error  } |

Above will result into compiler error as given below:

|  |
| --- |
| Console |
| Unexpected @FunctionalInterface annotation  @FunctionalInterface ^ MyFirstFunctionalInterface is not a functional interface  multiple non-overriding abstract methods found in interface MyFirstFunctionalInterface |



Read More : [Generic functional interfaces](https://howtodoinjava.com/java8/generic-functional-interfaces/)

2. Do’s and Don’t’s in functional interfaces

Below is list of things which are allowed and which are not in a functional interface.

As discussed above, ***only one abstract method is allowed*** in any functional interface. Second abstract method is not not permitted in a functional interface. If we remove **@FunctionInterface** annotation then we are allowed to add another abstract method, but it will make the interface non-functional interface.

A functional interface is ***valid even if the @FunctionalInterface annotation would be omitted***. It is only for informing the compiler to enforce single [abstract method](https://howtodoinjava.com/object-oriented/exploring-interfaces-and-abstract-classes-in-java/) inside interface.

Conceptually, a functional interface has exactly one abstract method. Since [**default methods**](https://howtodoinjava.com/java8/default-methods-in-java-8/) have an implementation, they are not abstract. Since default methods are not abstract you’re ***free to add default methods to your functional interface as many as you like***.

Below is valid functional interface:

|  |
| --- |
| @FunctionalInterface  public interface MyFirstFunctionalInterface  {    public void firstWork();       default void doSomeMoreWork1(){      //Method body      }      default void doSomeMoreWork2(){      //Method body      }  } |

If an interface declares an ***abstract method overriding one of the public methods of java.lang.Object, that also does not count toward the interface’s abstract method count*** since any implementation of the interface will have an implementation from java.lang.Object or elsewhere. e.g. [**Comparator**](https://howtodoinjava.com/search-sort/when-to-use-comparable-and-comparator-interfaces-in-java/) is a functional interface even though it declared two abstract methods. Why? Because one of these abstract methods “equals()” which has signature equal to public method in Object class.

e.g. Below interface is a valid functional interface.

|  |
| --- |
| @FunctionalInterface  public interface MyFirstFunctionalInterface  {    public void firstWork();       @Override      public String toString();                //Overridden from Object class       @Override      public boolean equals(Object obj);        //Overridden from Object class  } |

1. Types of method references

Java 8 allows four types of method references.

|  |  |  |
| --- | --- | --- |
| METHOD REFERENCE | DESCRIPTION | METHOD REFERENCE EXAMPLE |
| Reference to **static method** | Used to refer static methods from a class | Math::max equivalent to Math.max(x,y) |
| Reference to **instance method from instance** | Refer to an instance method using a reference to the supplied object | System.out::println equivalent to System.out.println(x) |
| Reference to **instance method from class type** | Invoke the instance method on a reference to an object supplied by the context | String::length equivalent to str.length() |
| Reference to **constructor** | Reference to a constructor | ArrayList::new equivalent to new ArrayList() |

2. Method reference to static method – Class::staticMethodName

An example to use Math.max() which is static method.

|  |
| --- |
| List<Integer> integers = Arrays.asList(1,12,433,5);  Optional<Integer> max = integers.stream().reduce( Math::max );  max.ifPresent(value -> System.out.println(value)); |

Output:

433

3. Method reference to instance method from instance – ClassInstance::instanceMethodName

In above example, we use System.out.println(value) to print the max value found. We can use System.out::println to print the value.

|  |
| --- |
| List<Integer> integers = Arrays.asList(1,12,433,5);    Optional<Integer> max = integers.stream().reduce( Math::max );    max.ifPresent( System.out::println ); |

Output:

433

4. Method reference to instance method from class type – Class::instanceMethodName

In this example, s1.compareTo(s2) is referred as String::compareTo.

|  |
| --- |
| List<String> strings = Arrays .asList("how", "to", "do", "in", "java", "dot", "com");   List<String> sorted = strings .stream().sorted((s1, s2) -> s1.compareTo(s2)) .collect(Collectors.toList());   System.out.println(sorted);   List<String> sortedAlt = strings.stream().sorted(String::compareTo).collect(Collectors.toList());   System.out.println(sortedAlt); |

Output:

[com, do, dot, how, in, java, to]

[com, do, dot, how, in, java, to]

5. Reference to constructor – Class::new

The first method can be updated to create a list of integers from 1 to 100. Using [lambda expression](https://howtodoinjava.com/java8/complete-lambda-expressions-tutorial-in-java/) is rather easy. To create a new instance of ArrayList, we have use ArrayList::new.

|  |
| --- |
| List<Integer> integers = IntStream.range(1, 100).boxed().collect(Collectors.toCollection( ArrayList::new ));  Optional<Integer> max = integers.stream().reduce(Math::max);   max.ifPresent(System.out::println); |

Output:

99

Java Default Methods Tutorial

In previous post, we learned about [**Lambda expressions and functional interfaces**](https://howtodoinjava.com/java8/complete-lambda-expressions-tutorial-in-java/). Now, let’s move on the discussion and talk about another related feature i.e. **default methods**. Well, this is truly revolutionary for java developers. Till java 7, we have learned a lot of things about interfaces and all those things have been in our mind whenever we wrote code or designed the applications. Some of these concepts are going to change drastically from java 8, after introduction of default methods.

What are default methods in java 8?

Default methods enable you to add new functionality to the interfaces of your libraries and ensure binary compatibility with code written for older versions of those interfaces.

As name implies, default methods in java 8 are simply default. If you do not override them, they are the methods which will be invoked by caller classes. They are defined in interfaces.

Let’s understand with an example:

|  |
| --- |
| public interface Moveable {      default void move(){          System.out.println("I am moving");      }} |

Moveable interface defines a method move(); and provided a default implementation as well. If any class implements this interface then it need not to implement it’s own version of move() method. It can directly call instance.move();

|  |
| --- |
| public class Animal implements Moveable{      public static void main(String[] args){          Animal tiger = new Animal();          tiger.move();      }}  Output: I am moving |

And if class willingly wants to customize the behavior then it can provide it’s own custom implementation and override the method. Now it’s own custom method will be called.

|  |
| --- |
| public class Animal implements Moveable{          public void move(){          System.out.println("I am running");      }      public static void main(String[] args){          Animal tiger = new Animal();          tiger.move();      }  }  Output: I am running |

This is not all done here. Best part comes as following benefits:

Static default methods: You can define static default methods in interface which will be available to all instances of class which implement this interface. This makes it easier for you to organize helper methods in your libraries; you can keep static methods specific to an interface in the same interface rather than in a separate class. This enables you to define methods out of your class and yet share with all child classes.

They provide you an highly desired capability of adding a capability to number of classes without even touching their code. Simply add a default method in interface which they all implement.

Why default methods were needed in java 8?

This is a good candidate for your next [**interview question**](https://howtodoinjava.com/java-interview-questions/). **Simplest answer is to enable the functionality of lambda expression in java.** Lambda expression are essentially of type of functional interface. To support lambda expressions seamlessly, all core classes have to be modified. But these core classes like java.util.List are implemented not only in JDK classes, but also in thousands of client code as well. Any incompatible change in core classes will back fire for sure and will not be accepted at all.

Default methods break this deadlock and allow adding support for functional interface in core classes. Let’s see an example. Below is a method which has been added to java.lang.Iterable.

|  |
| --- |
| default void forEach(Consumer<? super T> action) {      Objects.requireNonNull(action);      for (T t : this) {          action.accept(t);      }  } |

Before java 8, if you had to iterate on a java collection then your would get an iterator instance and call it’s next method until hasNext() returns false. This is common code and have been used thousands of time in day to day programming by us. Syntax is also always same. So can we make it compact so that it takes only single line of code and still do the job for us as before. Above function does that.

Now to iterate and perform some simple operation on every item in list, all you need to do is:

|  |
| --- |
| public class Animal implements Moveable{      public static void main(String[] args){          List<Animal> list = new ArrayList();          list.add(new Animal());          list.add(new Animal());          list.add(new Animal());          //Iterator code reduced to one line          list.forEach((Moveable p) -> p.move());      }  } |

So here, an additional method has been added to List without breaking any custom implementations of it. It has been very desired feature in java since long. Now it’s with us.

How conflicts are resolved while calling default methods?

So far so good. We have got all basics well. Now move to complicated things. In java, a class can implement N number of interface. Additionally, a interface can also extend another interface as well. An if any default method is declared in two such interfaces which are implemented by single class. then obviously class will get confused which method to call.

**Rules for this conflict resolution are as follows:**

**1)** Most preferred are the overridden methods in classes. They will be matched and called if found before matching anything.  
**2)** The method with the same signature in the “most specific default-providing interface” is selected. This means if class Animal implements two interfaces i.e. Moveable and Walkable such that Walkable extends Moveable. Then Walkable is here most specific interface and default method will be chosen from here if method signature is matched.  
**3)** If Moveable and Walkable are independent interfaces then a serious conflict condition happen, and compiler will complain then it is unable to decide. The you have to help compiler by providing extra info that from which interface the default method should be called. e.g.

|  |
| --- |
| Walkable.super.move();  //or  Moveable.super.move(); |

Java 8 – Read file line by line

In this [Java 8 tutorial](https://howtodoinjava.com/java-8-tutorial/), learn to **read a file line by line** using [stream api](https://howtodoinjava.com/java8/java-streams-by-examples/). Also learn to iterate through lines and filter the file content based on some conditions.

1. Java 8 read file – line by line

In this example, I will read the file content in lines as stream and fetch each line one at a time and check it for word "password".

|  |
| --- |
| Path filePath = Paths.get("c:/temp", "data.txt");   //try-with-resources  try (Stream<String> lines = Files.lines( filePath ))  {      lines.forEach(System.out::println);  }  catch (IOException e)  {      e.printStackTrace();  } |

The above program output will print the content of the file in the console line by line.

|  |
| --- |
| Console |
| Never  store  password  except  in mind. |

2. Java 8 read file – filtering stream of lines

In this example, we will read the file content as stream of lines as. Then we will filter all lines which have the word "password" in it.

|  |
| --- |
| Path filePath = Paths.get("c:/temp", "data.txt");   try (Stream<String> lines = Files.lines(filePath))  {        List<String> filteredLines = lines .filter(s -> s.contains("password")).collect(Collectors.toList());       filteredLines.forEach(System.out::println);  } catch (IOException e) {      e.printStackTrace();  } |

Program output.

|  |
| --- |
| Console |
| password |

We will read the content of the given file and check if any line contains word "password" then print it.

3. Java 7 – Read file using FileReader

Till java 7, we could read a file using [FileReader](https://docs.oracle.com/javase/7/docs/api/java/io/FileReader.html) in various ways.

|  |
| --- |
| private static void readLinesUsingFileReader() throws IOException  {      File file = new File("c:/temp/data.txt");      FileReader fr = new FileReader(file);      BufferedReader br = new BufferedReader(fr);       String line;      while((line = br.readLine()) != null)      {          if(line.contains("password")){              System.out.println(line);          }      }      br.close();      fr.close();  } |

Java Comparator with Lambda

[Comparator](https://docs.oracle.com/javase/10/docs/api/java/util/Comparator.html) is used when we want to sort a [collection](https://howtodoinjava.com/java/collections/useful-java-collection-interview-questions/) of objects which can be compared with each other. This comparison can be done using Comparable interface as well, but it restrict you compare these objects in a single particular way only. If you want to sort this collection, based on multiple criterias/fields, then you have to use Comparator only.

**Quick Reference:**

|  |
| --- |
| //Compare by Id  Comparator<Employee> compareById\_1 = Comparator.comparing(e -> e.getId());  Comparator<Employee> compareById\_2 = (Employee o1, Employee o2) -> o1.getId().compareTo( o2.getId() );   //Compare by firstname  Comparator<Employee> compareByFirstName = Comparator.comparing(e -> e.getFirstName());  //how to use comparator  Collections.sort(employees, compareById); |

1) Overview

To demo the concept, I am using class Employee with four attributes. We will use it to understand various use cases.

|  |
| --- |
| public class Employee {      private Integer id;      private String firstName;      private String lastName;      private Integer age;      public Employee(Integer id, String firstName, String lastName, Integer age){          this.id = id;          this.firstName = firstName;          this.lastName = lastName;          this.age = age;      }      //Other getter and setter methods        @Override      public String toString() {          return "\n["+this.id+","+this.firstName+","+this.lastName+","+this.age+"]";      }  } |

Additionally I have written one method which always return a list of Employees in unsorted order.

|  |
| --- |
| private static List<Employee> getEmployees(){      List<Employee> employees  = new ArrayList<>();      employees.add(new Employee(6,"Yash", "Chopra", 25));      employees.add(new Employee(2,"Aman", "Sharma", 28));      employees.add(new Employee(3,"Aakash", "Yaadav", 52));      employees.add(new Employee(5,"David", "Kameron", 19));      employees.add(new Employee(4,"James", "Hedge", 72));      employees.add(new Employee(8,"Balaji", "Subbu", 88));      employees.add(new Employee(7,"Karan", "Johar", 59));      employees.add(new Employee(1,"Lokesh", "Gupta", 32));      employees.add(new Employee(9,"Vishu", "Bissi", 33));      employees.add(new Employee(10,"Lokesh", "Ramachandran", 60));      return employees;  } |

2) Sort by first name

Basic usecase where list of employees will be sorted based on their first name.

|  |
| --- |
| List<Employee> employees  = getEmployees();      //Sort all employees by first name      employees.sort(Comparator.comparing(e -> e.getFirstName()));      //OR you can use below      employees.sort(Comparator.comparing(Employee::getFirstName));      //Let's print the sorted list      System.out.println(employees);  Output: //Names are sorted by first name  [      [3,Aakash,Yaadav,52],      [2,Aman,Sharma,28],      [8,Balaji,Subbu,88],      [5,David,Kameron,19],      [4,James,Hedge,72],      [7,Karan,Johar,59],      [1,Lokesh,Gupta,32],      [10,Lokesh,Ramachandran,60],      [9,Vishu,Bissi,33],      [6,Yash,Chopra,25]  ] |

3) Sort by first name – ‘reverse order’

What if we want to sort on first name but in revered order. It’s really very easy; use reversed() method.

|  |
| --- |
| List<Employee> employees  = getEmployees();        //Sort all employees by first name; And then reversed      Comparator<Employee> comparator = Comparator.comparing(e -> e.getFirstName());      employees.sort(comparator.reversed());      //Let's print the sorted list      System.out.println(employees);  Output: //Names are sorted by first name  [[6,Yash,Chopra,25],  [9,Vishu,Bissi,33],  [1,Lokesh,Gupta,32],  [10,Lokesh,Ramachandran,60],  [7,Karan,Johar,59],  [4,James,Hedge,72],  [5,David,Kameron,19],  [8,Balaji,Subbu,88],  [2,Aman,Sharma,28],  [3,Aakash,Yaadav,52]] |

4) Sort by last name

We can use similar code to sort on last name as well.

|  |
| --- |
| List<Employee> employees  = getEmployees();      //Sort all employees by first name      employees.sort(Comparator.comparing(e -> e.getLastName()));      //OR you can use below      employees.sort(Comparator.comparing(Employee::getLastName));      //Let's print the sorted list      System.out.println(employees);  Output: //Names are sorted by first name  [[9,Vishu,Bissi,33],  [6,Yash,Chopra,25],  [1,Lokesh,Gupta,32],  [4,James,Hedge,72],  [7,Karan,Johar,59],  [5,David,Kameron,19],  [10,Lokesh,Ramachandran,60],  [2,Aman,Sharma,28],  [8,Balaji,Subbu,88],  [3,Aakash,Yaadav,52]] |

5) Sort on multiple fields – thenComparing()

Here we are sorting list of employees first by their first name, and then sort again the list of last name. Just as we apply sorting on SQL statements. It’s actually a very good feature.

Now you don’t need to always use sorting on multiple fields in SQL select statements, you can sort them in java as well.

|  |
| --- |
| List<Employee> employees  = getEmployees();  //Sorting on multiple fields; Group by.  Comparator<Employee> groupByComparator = Comparator.comparing(Employee::getFirstName)                                                      .thenComparing(Employee::getLastName);  employees.sort(groupByComparator);  System.out.println(employees);  Output:   [3,Aakash,Yaadav,52],  [2,Aman,Sharma,28],  [8,Balaji,Subbu,88],  [5,David,Kameron,19],  [4,James,Hedge,72],  [7,Karan,Johar,59],  [1,Lokesh,Gupta,32],         //These both employees are  [10,Lokesh,Ramachandran,60], //sorted on last name as well  [9,Vishu,Bissi,33],  [6,Yash,Chopra,25] |

5) Parallel sort (with multiple threads)

You can **sort the collection of objects in parallel** using multiple threads as well. It is going to be very fast if collection is big enough having thousands of objects. For small collection of objects, normal sorting is good enough and recommended.

|  |
| --- |
| //Parallel Sorting  Employee[] employeesArray = employees.toArray(new Employee[employees.size()]);  //Parallel sorting  Arrays.parallelSort(employeesArray, groupByComparator);  System.out.println(employeesArray);  Output:  [3,Aakash,Yaadav,52],  [2,Aman,Sharma,28],  [8,Balaji,Subbu,88],  [5,David,Kameron,19],  [4,James,Hedge,72],  [7,Karan,Johar,59],  [1,Lokesh,Gupta,32],         //These both employees are  [10,Lokesh,Ramachandran,60], //sorted on last name as well  [9,Vishu,Bissi,33],  [6,Yash,Chopra,25] |