**Java 8 Streams**

A *Stream in Java* can be defined as a **sequence of elements from a source**. The source of elements here refers to a [Collection](https://howtodoinjava.com/java/collections/into-to-java-collection/) or [Array](https://howtodoinjava.com/series/java-arrays/) that provides data to the Stream.

* Java streams are designed in such a way that most of the stream operations (called **intermediate operations**) return a Stream. This helps to create a chain of stream operations. This is called **stream pipe-lining**.
* Java streams also support the **aggregate or terminal operations** on the elements. The aggregate operations are operations that allow us to express common manipulations on stream elements quickly and clearly, for example, finding the max or min element, finding the first element matching giving criteria, and so on.
* Not that a ***stream maintains the same ordering* of the elements *as the ordering in the stream source***.

**Difference between Collection & Stream**

A Collection is an in-memory data structure that holds all the data structure’s values.

Every element in the Collection has to be computed before it can be added to the Collection. While a Stream is conceptually a pipeline in which elements are computed on demand.

This concept gives rise to significant programming benefits. The idea is that a user will extract only the values they require from a Stream

**Empty Stream**

We should use the ***empty()*** method in case of the creation of an empty stream:

Stream<String> streamEmpty = Stream.empty();

We often use the *empty()*method upon creation to avoid returning *null* for streams with no element:

### ****Stream of****Collection

We can also create a stream of any type of Collection (Collection, List, Set):

**Collection<String> collection = Arrays.asList("a", "b", "c");**

**Stream<String> streamOfCollection = collection.stream();**

### ****Stream of Array****

An array can also be the source of a stream:

**Stream<String> streamOfArray = Stream.of("a", "b", "c");**

We can also create a stream out of an existing array or of part of an array:

**String[] arr = new String[] {"a", "b", "c"};**

**Stream<String> streamOfArrayFull = Arrays.stream(arr);**

**Stream<String> streamOfArrayPart = Arrays.stream(arr, 1, 3);**

### Stream.builder()

**When builder is used, the desired type should be additionally specified in the right part of the statement,** otherwise the build() method will create an instance of the Stream<Object>:

**Stream<String> streamBuilder =**

**Stream.<String>builder().add("a").add("b").add("c").build();**

## Creating Streams

## The given below ways are the most popular different ways to build streams from collections.

## Stream.of()

## In the given example, we are creating a stream of a fixed number of integers.

## Stream<Integer> stream = Stream.of(1,2,3,4,5,6,7,8,9);

## stream.forEach(p -> System.out.println(p));

## Stream.of(array)

## In the given example, we are creating a stream from the array. The elements in the stream are taken from the array.

## Stream<Integer> stream = Stream.of( new Integer[]{1,2,3,4,5,6,7,8,9} );

## stream.forEach(p -> System.out.println(p));

## List.stream()

## In the given example, we are creating a stream from the List. The elements in the stream are taken from the List.

## 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));

## ****Stream.generate() or Stream.iterate()****

## **In the given example, we are creating a stream from generated elements. This will produce a stream of 20 random numbers. We have restricted the elements count using limit() function.**

## ****Stream<Integer> randomNumbers = Stream.generate( () -> (new Random() ).nextInt(100));****

## ****randomNumbers.limit(20).forEach(System.out::println);****

## ****Stream of String chars or tokens****

## **In the given example, first, we create a stream from the characters of a given string. In the second part, we are creating the stream of tokens received from splitting from a string.**

## ****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 also such as using Stream.Buider or using intermediate operations. We will learn about them in separate posts from time to time.**

## ****Stream Collectors****

## **After performing the intermediate operations on elements in the stream, we can collect the processed elements again into a Collection using the stream Collector methods.**

## ****Collect Stream elements to a List****

## **In the given example, first, we create a stream on integers 1 to 10. Then we process the stream elements to find all even numbers.**

## **At last, we are collecting all even numbers into a List.**

## **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);**

## ****Collect Stream elements to an Array****

## **The given example is similar to the first example shown above. The only difference is that we are collecting even numbers in an Array.**

## ****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 a Set, Map or into multiple ways. Just go through Collectors class and try to keep them in mind.**

## ****Stream Operations****

## **Stream abstraction has a long list of useful functions. Let us look at a few of them.**

## **Before moving ahead, let us build a List of strings beforehand. We will build our examples 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:**

## ****Intermediate Operations****

## ****Intermediate operations return the stream itself so you can chain multiple methods calls in a row.****

## **Let’s learn important ones.**

## ****Stream.filter()****

## **The filter() method accepts a Predicate to filter all elements of the stream. This operation is intermediate, enabling us to call another stream operation (e.g. forEach()) on the result.**

## **memberNames.stream().filter((s) -> s.startsWith("A"))**

## **.forEach(System.out::println);**

## **Program Output:**

## **Amitabh**

## **Aman**

## ****Stream.map()****

## **The map() intermediate operation converts each element in the stream into another object via the given function.**

## **The following example converts each string into an UPPERCASE string. But we can use map() to transform an object into another type as well.**

## ****memberNames.stream()****

## ****.filter((s) -> s.startsWith("A"))****

## ****.map(String::toUpperCase)****

## ****.forEach(System.out::println);****

## **Program Output:**

## **AMITABH**

## **AMAN**

## ****Stream.sorted()****

## **The sorted() method is an intermediate operation that returns a sorted view of the stream. The elements in the stream are sorted in natural order unless we pass a custom Comparator.**

## ****memberNames.stream().sorted()****

## ****.map(String::toUpperCase)****

## ****.forEach(System.out::println);****

## **Program Output:**

## **AMAN**

## **AMITABH**

## **LOKESH**

## **RAHUL**

## **SALMAN**

## **SHAHRUKH**

## **SHEKHAR**

## **YANA**

## **Please note that the sorted() method only creates a sorted view of the stream without manipulating the ordering of the source Collection. In this example, the ordering of string in the memberNames is untouched.**

## ****IntStream:****

**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**

## ****Terminal operations****

## ****Terminal operations return a result of a certain type after processing all the stream elements.****

## **Once the terminal operation is invoked on a Stream, the iteration of the Stream and any of the chained streams will get started.**

## **Once the iteration is done, the result of the terminal operation is returned.**

## ****Stream.forEach()****

## **The forEach() method helps iterate over all stream elements and perform some operation on each of them. The operation to be performed is passed as the lambda expression.**

## ****memberNames.forEach(System.out::println);****

## ****Stream.collect()****

## **The collect() method is used to receive elements from steam and store them in a collection.**

## ****List<String> memNamesInUppercase = memberNames.stream().sorted()****

## ****.map(String::toUpperCase)****

## ****.collect(Collectors.toList());****

## ****System.out.print(memNamesInUppercase);****

## ****Program Output:****

## ****[AMAN, AMITABH, LOKESH, RAHUL, SALMAN, SHAHRUKH, SHEKHAR, YANA]****

## ****Stream.match()****

## **Various matching operations can be used to check whether a given predicate matches the stream elements. All of these matching operations are terminal and return a boolean result.**

## ****boolean matchedResult = memberNames.stream()****

## ****.anyMatch((s) -> s.startsWith("A"));****

## 

## ****System.out.println(matchedResult); //true****

## 

## ****matchedResult = memberNames.stream()****

## ****.allMatch((s) -> s.startsWith("A"));****

## 

## **System.out.println(matchedResult); //false**

## 

## **matchedResult = memberNames.stream()**

## **.noneMatch((s) -> s.startsWith("A"));**

## 

## **System.out.println(matchedResult); //false**

## ****Stream.count()****

## **The count() is a terminal operation returning the number of elements in the stream as a long value.**

## ****long totalMatched = memberNames.stream()****

## ****.filter((s) -> s.startsWith("A"))****

## ****.count();****

## 

## ****System.out.println(totalMatched); //2****

## ****Stream.reduce()****

## **The reduce() method performs a reduction on the elements of the stream with the given function. The result is an Optional holding the reduced value.**

## **In the given example, we are reducing all the strings by concatenating them using a separator #.**

## ****Optional<String> reduced = memberNames.stream()****

## ****.reduce((s1,s2) -> s1 + "#" + s2);****

## ****reduced.ifPresent(System.out::println);****

## **Program Output:**

## **Amitabh#Shekhar#Aman#Rahul#Shahrukh#Salman#Yana#Lokesh**

## ****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, we will do with the if-else block. In the internal iterations such as in streams, there are certain methods we can use for this purpose.**

## ****Stream.anyMatch()****

## **The anyMatch() will return true once a condition passed as predicate satisfies. Once a matching value is found, no more elements will be processed in the stream.**

## **In the given example, as soon as a String is found starting with the letter 'A', the stream will end and the result will be returned.**

## ****boolean matched = memberNames.stream()****

## ****.anyMatch((s) -> s.startsWith("A"));****

## 

## ****System.out.println(matched); //true****

## ****Stream.findFirst()****

## **The findFirst() method will return the first element from the stream and then it will not process any more elements.**

## ****String firstMatchedName = memberNames.stream()****

## ****.filter((s) -> s.startsWith("L"))****

## ****.findFirst().get();****

## ****System.out.println(firstMatchedName); //Lokesh****

## ****Parallel Streams****

## **With the Fork/Join framework added in Java SE 7, we have efficient machinery for implementing parallel operations in our applications.**

## **But implementing a fork/join framework is a complex task, and if not done right; it is a source of complex multi-threading bugs having the potential to crash the application. With the introduction of internal iterations, we got the possibility of operations to be done in parallel more efficiently.**

## **To enable parallelism, all we have to do is to create a parallel stream, instead of a sequential stream. And to surprise, this is really very easy.**

## **In any of the above-listed stream examples, anytime we want to do a particular job using multiple threads in parallel cores, all we have to call parallelStream() method instead of stream() method.**

## ****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 Stream APIs 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 and 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.**

## ****Working with Streams****

## ****Creating Streams****

## ****concat()****

## ****empty()****

## ****generate()****

## ****iterate()****

## ****of()****

## ****Intermediate Operations****

## ****filter()****

## ****map()****

## ****flatMap()****

## ****distinct()****

## ****sorted()****

## ****peek()****

## ****limit()****

## ****skip()****

## ****Terminal Operations****

## ****forEach()****

## ****forEachOrdered()****

## ****toArray()****

## ****reduce()****

## ****collect()****

## ****min()****

## ****max()****

## ****count()****

## ****anyMatch()****

## ****allMatch()****

## ****noneMatch()****

## ****findFirst()****

## ****findAny()****

<https://howtodoinjava.com/java/stream/java-streams-by-examples/>

## ****What is a Functional Interface?****

## **Only one abstract method is allowed**

## **Functional interfaces are new additions in Java 8. As a rule, a functional interface can contain exactly one abstract method. These functional interfaces are also called Single Abstract Method interfaces (SAM Interfaces).**

## **Apart from one abstract method, a functional interface can also have the following methods that do not count for defining it as a functional interface.**

## ****Default methods****

## ****Static methods****

## ****Public methods inherited from the Object class****

## ****Implemented by Lambda Expressions****

## **In Java, lambda expressions can be used to represent an instance of a functional interface. For example, Comparator interface is a functional interface.**

## ****@FunctionalInterface****

## ****public interface Comparator<T> {****

## ****int compare(T o1, T o2);****

## ****boolean equals(Object obj);****

## ****//and multiple default methods...****

## ****}****

## **Comparator interface has only two abstract methods compare() and equals(). But equals() has been inherited from the Object class, so it is not counted. Other than these two methods, all other methods are default methods. So Comparator is qualified to be declared as a functional interface.**

## ****Java program to implement Comparator using a lambda expression.****

## ****//Compare by Id****

## ****Comparator<Employee> compareById = Comparator.comparing(e -> e.getId());****

## ****Comparator<Employee> compareByFirstName = Comparator.comparing(e -> e.getFirstName());****

## ****@FunctionalInterface Annotation****

## **Java 8 introduced the annotation @FunctionalInterface to mark an interface as a functional interface. The primary use of this annotation is for compiler-level errors when the interface violates the contracts of precisely one abstract method.**

## ****Note that using the annotation @FunctionalInterface is optional.****

## **If the interface has one abstract method and does not have @FunctionalInterface annotation, the interface is still a functional interface, and it can be the target type for lambda expressions.**

## **The presence of the annotation protects us from inadvertently changing a functional interface into a non-functional interface, as the compiler will catch it.**

## **Let’s build our first functional interface. Note that methods in an interface are, by default, abstract.**

## ****@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****

## ****}****

## ****The above code will result in a compiler error:****

## ****Functional Interfaces in JDK****

## ****The following is a list of Java’s most commonly used functional interfaces.****

## **Runnable: contains only the run() method.**

## **Comparable: contains only the compareTo() method.**

## **ActionListener: contains only the actionPerformed() method.**

## **Callable: contains only the call() method.**

## **Predicate: a boolean-valued function that takes an argument and returns true or false.**

## **BiPredicate: a predicate with two arguments.**

## **Consumer: an operation that takes an argument, operates on it, and returns no result.**

## **BiConsumer: a consumer with two arguments.**

## **Supplier: a supplier that returns a value.**

## **Function<T, R>: takes an argument of type T and returns a result of type R.**

## **BiFunction<T, U, R>: takes two arguments of types T and U and returns a result of type R.**

## ****Demo****

## **Let’s see a quick example of creating and using functional interfaces in Java.**

## **We are using a functional interface Function to create the formula for mathematical squares.**

## ****Function<Integer, Integer> square = x -> x \* x;****

## **The Function interface has one abstract method apply() that we have implemented above. we can execute the above method as follows:**

## ****System.out.println( square.apply(5) ); //Prints 25****

## ****5. Conclusion****

## **In this tutorial, we learned to create and manage functional interfaces in Java. We learned that a functional interface has only one abstract method and they can be implemented by the lambda expressions.**

## **We also saw the JDK provided existing functional interfaces, and finally how to create an use a functional interface.**

## ****What are Lambda Expressions?****

## **In general programming language, a Lambda expression (or function) is an anonymous function, i.e., a function without any name or identifier, and with a list of formal parameters and a body. An arrow (->) is used to separate the list of parameters and the body.**

## **In Java, a lambda expression is an expression that represents an instance of a functional interface.**

## **Similar to other types in Java, lambda expressions are also typed, and their type is a functional interface type. To infer the type, the compiler looks at the left side of the assignment in a lambda expression.**

## **Note that the lambda expression itself does not contain the information about which functional interface it is implementing. This information is deduced from the context in which expression is used.**

**Lambda Expression Example**

A typical lambda expression syntax will be like this:

**(parameters) -> expression**

For example, the below-given lambda expression takes two parameters and returns their addition. Based on the type of x and y, the expression will be used differently.

If the parameters match to Integer the expression will add the two numbers.

If the parameters of type String the expression will concat the two strings.

(x, y) -> x + y

For example, we have the following functional interface Operator. It has one method process() that takes two parameters and returns a value.

@FunctionalInterface

interface Operator<T> {

T process(T a, T b);}

We can create lambda expressions for this functional interface in the following manner. Notice we are able to create the method implementations and immediately use them. We do not need to create a concrete class OperatorImpl that implements Operator interface.

**Operator<Integer> addOperation = (a, b) -> a + b;**

**System.out.println(addOperation.process(3, 3)); //Prints 6**

**Operator<String> appendOperation = (a, b) -> a + b;**

**System.out.println(appendOperation.process("3", "3")); //Prints 33**

**Operator<Integer> multiplyOperation = (a, b) -> a \* b;**

**System.out.println(multiplyOperation.process(3, 3)); //Prints 9**

Two good examples of functional interface types are Consumer and BiConsumer interfaces that are heavily used in Stream API for creating lambda expressions.

**Features of Lambda Expressions**

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

(x, y) -> x + y

(x, y, z) -> x + y + z

The body of the lambda expressions can contain zero, one or more statements. If the body of lambda expression has a 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 the body then these must be enclosed in curly brackets.

(parameters) -> { statements; }

The type of the parameters can be explicitly declared or it can be inferred from the context. In previous example, the type of addOperation and appendOperation is derived from context.

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

() -> expression

When there is a single parameter, if its type is inferred, it is not mandatory to use parentheses.

a -> return a \* a;

A lambda expression cannot have a throws clause. It is inferred from the context of its use and its body.

Lambda expressions cannot be generic i.e. they cannot declare type parameters.

**More Examples**

We are listing out some code samples which you can read and analyze how a lambda expression can be used in the day-to-day programming.

Example 1: Using lambda expression to iterate over a List and perform some action on list items

In the given example, we are iterating over the list and printing all the list elements in the standard output. We can perform any desired operation in place of printing them.

List<String> pointList = new ArrayList();

pointList.add("1");

pointList.add("2");

pointList.forEach( p -> { System.out.println(p); } );

Example 2: Using lambda expression to create and start a Thread in Java

In given example, we are passing the instance of Runnable interface into the Thread constructor.

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

Example 3: Java comparator

Comparator<User> firstNameComparator = (u1, u2) -> u1.firstName().compareTo(u2.firstName());

Example 4: Using lambda expression for adding an event listener to a GUI component

JButton button = new JButton("Submit");

button.addActionListener((e) -> {

System.out.println("Click event triggered !!");

});

Above are very basic examples of lambda expressions in java 8. I will be coming up with more useful examples and code samples from time to time.

**Advantages of Lambda Expressions**

Lambda expressions enable many benefits of functional programming to Java. Like most OOP languages, Java is built around classes and objects and treats only the classes as their first-class citizens. The other important programming entities, such as functions, take the back seat.

But in functional programming, we can define functions, give them reference variables, and pass them as method arguments and much more. JavaScript is a good example of functional programming where we can pass callback methods to Ajax calls and so on.

Note that we were able to do everything prior to Java 8 using anonymous classes that we can do with lambda expressions, but they use a very concise syntax to achieve the same result. Let us see the comparison of the same method implementation using both techniques.

//Using lambda expression

Operator<Integer> addOperation = (a, b) -> a + b;

//Using anonymous class

Operator<Integer> addOperation = new Operator<Integer>() {

@Override

public Integer process(Integer a, Integer b) {

return a + b;

}

};

Lambda expression is a very useful feature and has been lacking in Java from the beginning. Now with Java 8, we can also use functional programming concepts with the help of this.

## ****Method References****

Method reference can be used as a shorter and more readable alternative for a lambda expression that only calls an existing method. There are four variants of method references.

**Reference to a Static Method**

The reference to a static method holds the syntax **ContainingClass::methodName.**

We'll try to count all empty strings in the List<String> with the help of Stream API:

**boolean isReal = list.stream().anyMatch(u -> User.isRealUser(u));**

Let's take a closer look at lambda expression in the anyMatch() method. It just makes a call to a static method isRealUser(User user) of the User class.

So, it can be substituted with a reference to a static method:

**boolean isReal = list.stream().anyMatch(User::isRealUser);**

This type of code looks much more informative.

**Reference to an Instance Method**

The reference to an instance method holds the syntax **containingInstance::methodName.**

The following code calls method isLegalName(String string) of type User, which validates an input parameter:

**User user = new User();**

**boolean isLegalName = list.stream().anyMatch(user 🡪 user.isLegalName());**

**User user = new User();**

**boolean isLegalName = list.stream().anyMatch(user::isLegalName);**

**Reference to an Instance Method of an Object of a Particular Type**

This reference method takes the syntax ContainingType::methodName.

Let's look at an example:

long count = list.stream().filter(String::isEmpty).count();

try writing it using lambda syntax to understand difference in ease.

**Reference to a Constructor**

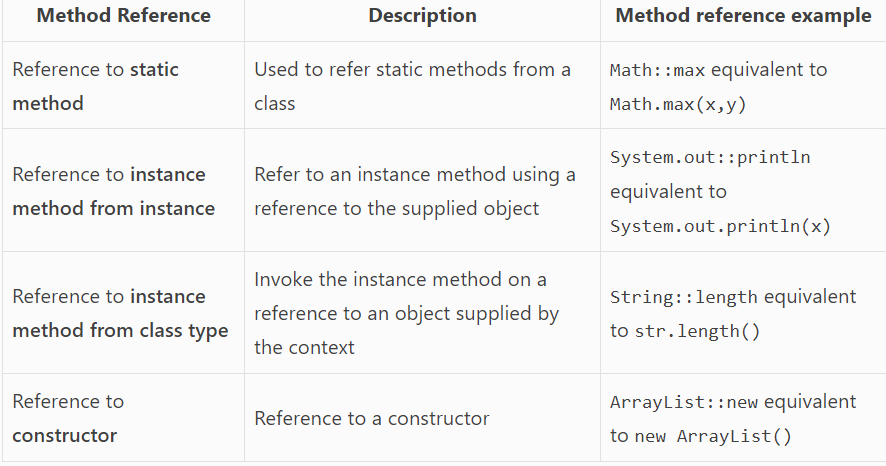
A reference to a constructor takes the syntax **ClassName::new.**

As constructor in Java is a special method, method reference could be applied to it too, with the help of new as a method name:

**Stream<User> stream = list.stream().map(User::new);**

**Types of method references**

**Java 8 allows four types of method references.**

****

**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 );**

**Optional<Integer> max = integers.stream().reduce(Math.max(x,y));**

**max.ifPresent(value -> System.out.println(value));**

**max.ifPresent( System.out::println );**

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);**

The first method can be updated to create a list of integers from 1 to 100. Using lambda expression 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**