**STREAM API**

1. Stream Creation
2. Collectors
3. Intermediate operations
4. Terminal operations
5. Real time examples
6. **Stream Creation**

We can create stream from

1. **Number of elements**

//Stream.of

Stream<Integer> numbers = Stream.*of*(1,2,3,4,5,6);

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

1. **Array**

//stream from array

String[] cities = **new** String[] {"Pune","Banglore","Hyderabad"};

Stream.*of*(cities).forEach(System.***out***::println);

1. **List collection**

//stream form list collection

LinkedList<Integer> nums=**new** LinkedList();

nums.add(54);

nums.add(98);

nums.stream().forEach(System.***out***::println);

1. **Generate() of Stream**

//generate method

Stream<Integer> rollNumbers=Stream.*generate*(()->**new** Random().nextInt());

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

System.***out***.println("---------------");

//stream for chars of string

IntStream stream = "12345\_abcdefg".chars();

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

Stream<String> streams = Stream.*of*("A$B$C".split("\\$"));

streams.forEach(p -> System.***out***.println(p));

1. **Collectors**

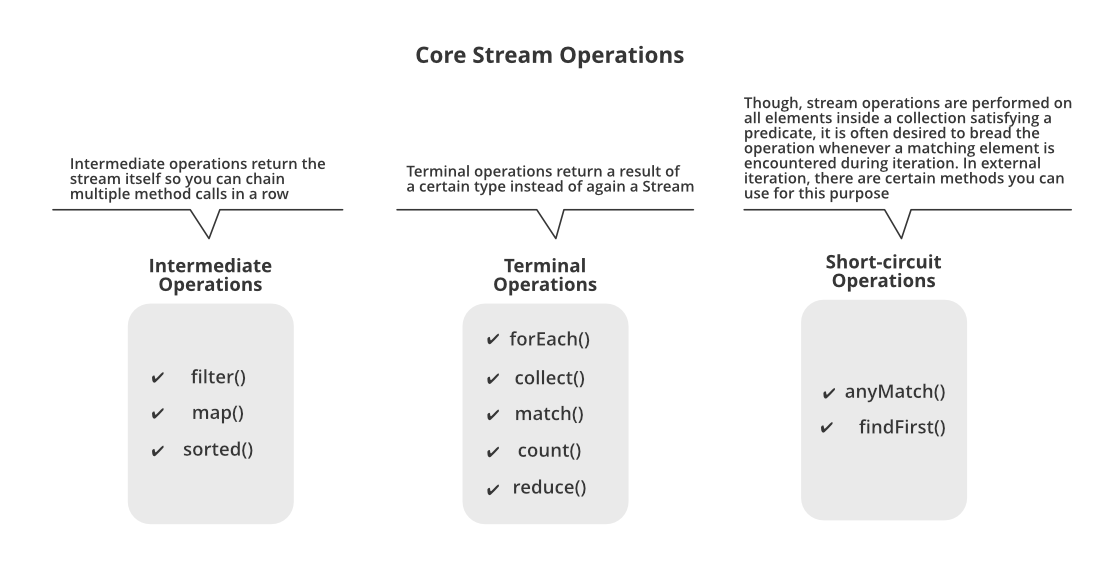
We can collect the stream into list, set, map, array etc. Along with this, we can don several operations like

**6)** **Intermediate Operations :**

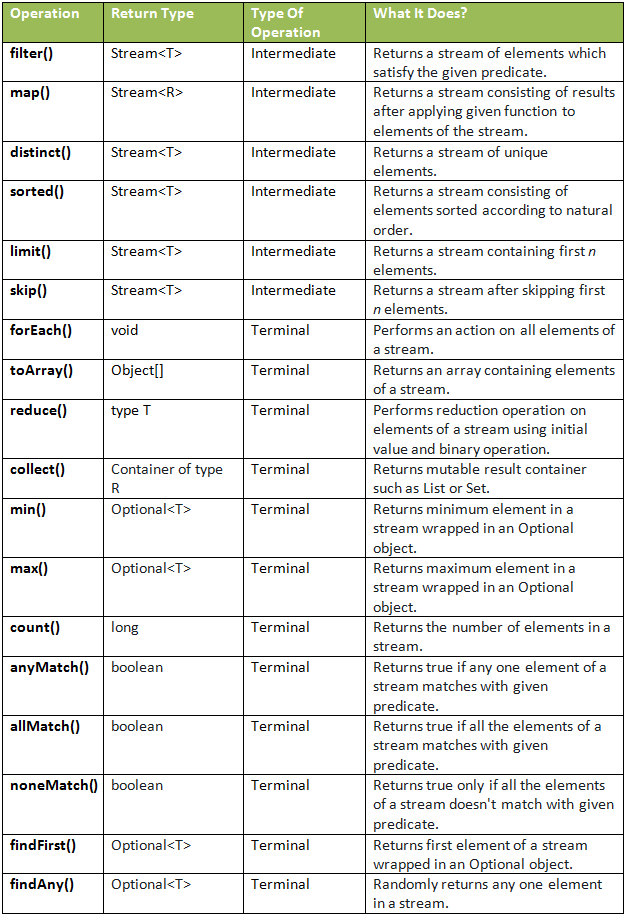
map(), filter(), distinct(), sorted(), limit(), skip()

**Terminal Operations :**

forEach(), toArray(), reduce(), collect(), min(), max(), count(), anyMatch(), allMatch(), noneMatch(), findFirst(), findAny()



Below is the list of intermediate and terminal operations.



Java 8 Stream Intermediate Vs Terminal Operations

|  |  |
| --- | --- |
| **Intermediate Operations** | **Terminal Operations** |
| They return stream. | They return non-stream values. |
| They can be chained together to form a pipeline of operations. | They can’t be chained together. |
| Pipeline of operations may contain any number of intermediate operations. | Pipeline of operations can have maximum one terminal operation, that too at the end. |
| Intermediate operations are lazily loaded. | Terminal operations are eagerly loaded. |
| They don’t produce end result. | They produce end result. |
| Examples : filter(), map(), distinct(), sorted(), limit(), skip() | Examples : forEach(), toArray(), reduce(), collect(), min(), max(), count(), anyMatch(), allMatch(), noneMatch(), findFirst(), findAny() |

forEach methods different for different collection. ***forEach used for various collections*** *is given below.*

*Refer* [*https://beginnersbook.com/2017/10/java-8-foreach/*](https://beginnersbook.com/2017/10/java-8-foreach/)

* foreach is used for following

1. iterate map

2.iterate list

3.iterate tream

* inside foreach loop, we can use lambda expression, mehod reference, function

1. **map:**The map method is used to returns a stream consisting of the results of applying the given function to the elements of this stream.  
   List number = Arrays.asList(2,3,4,5);  
   List square = number.stream().map(x->x\*x).collect(Collectors.toList());
2. **filter:** The filter method is used to select elements as per the Predicate passed as argument.  
   List names = Arrays.asList("Reflection","Collection","Stream");  
   List result = names.stream().filter(s->s.startsWith("S")).collect(Collectors.toList());
3. **sorted:** The sorted method is used to sort the stream.  
   List names = Arrays.asList("Reflection","Collection","Stream");  
   List result = names.stream().sorted().collect(Collectors.toList());

**Terminal Operations:**

1. **collect:** The collect method is used to return the result of the intermediate operations performed on the stream.  
   List number = Arrays.asList(2,3,4,5,3);  
   Set square = number.stream().map(x->x\*x).collect(Collectors.toSet());
2. **forEach:** The forEach method is used to iterate through every element of the stream.  
   List number = Arrays.asList(2,3,4,5);  
   number.stream().map(x->x\*x).forEach(y->System.out.println(y));
3. **reduce:** The reduce method is used to reduce the elements of a stream to a single value.  
   The reduce method takes a BinaryOperator as a parameter.

List number = Arrays.asList(2,3,4,5);  
int even = number.stream().filter(x->x%2==0).reduce(0,(ans,i)->ans+i);

Here ans variable is assigned 0 as the initial value and i is added to it .

Refer: <https://www.geeksforgeeks.org/stream-in-java/>

**Min() and Max()**

Integer[] intArr = { 1, 23, 2, 55, 65, 64, 26, 5, 56 };

System.out.println(Stream.of(intArr).min(***(o1, o2) -> o1 > o2 ? o2 : o1***).get());

System.out.println(Stream.of(intArr).max(**Integer::compareTo**).get());

System.out.println(Stream.of(intArr).max(**(a, b) -> a.compareTo(b)**).get());

# Method reference

Java provides a new feature called method reference in Java 8. Method reference is used to refer method of functional interface. It is compact and easy form of lambda expression. Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference.

## **Four types of method references**

1. Method reference to an instance method of an object – object::instanceMethod  
2. Method reference to a static method of a class – Class::staticMethod  
3. Method reference to an instance method of an arbitrary object of a particular type – Class::instanceMethod  
4. Method reference to a constructor – Class::new

Given example will show you the difference between lambda interface and method reference

@FunctionalInterface

**public** **interface** PersonFunctionalInterface {

String getInstanceMethod(**int** age);

}

We can implement above interface using lambda expression as given below

**PersonFunctionalInterface person=(age)-> "Ankur's age is "+age;**

**System.out.println(person.getInstanceMethod(52));**

And same implementation you can do using method reference i. e. referring to interface method to instance or static method of class like below.

**public** **class** PersonDetails {

**public** String getAge(**int** age) {

**return** "Ankur's age is" + age;

}

}

**PersonFunctionalInterface personI=new PersonDetails()::getAge;**

**System.out.println(personI.getInstanceMethod(5));**

Method reference is a shorthand notation of a lambda expression to call a method. For example:  
If your lambda expression is like this:

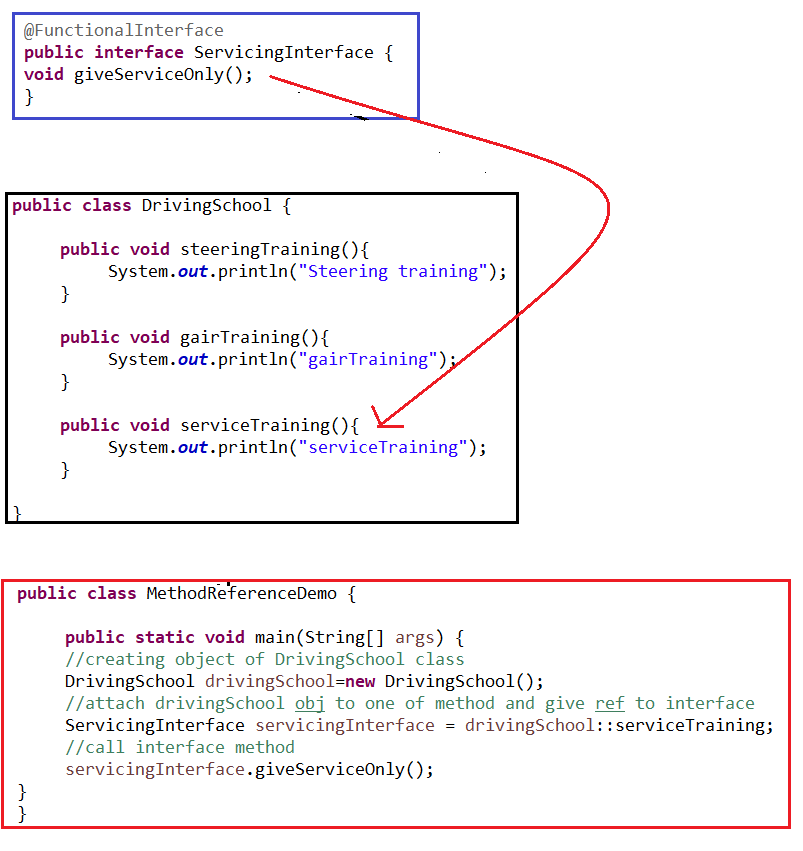
str ->System.out.println(str)

then you can replace it with a method reference like this:

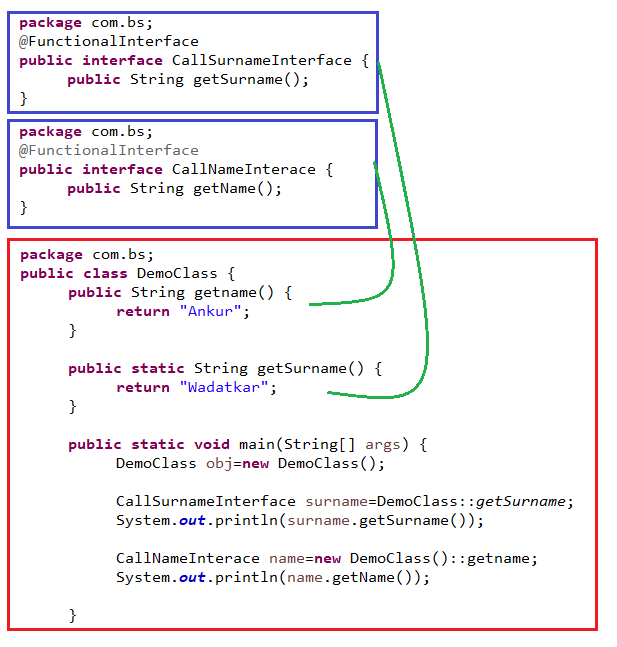
System.out::println

The :: operator is used in method reference to separate the class or object from the method name(we will learn this with the help of examples).

1. Method reference to an instance method of an object – **object::instanceMethod**

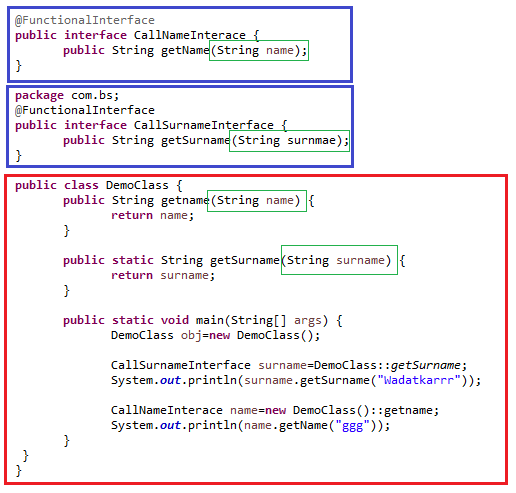


2. Method reference to an static method of a class– **ClassName::staticMethod**



**Method reference with arguments**

While referring to method no need to give an argument list, but when we will call interface method that time we should provide arguments. Note that the functional interface and class method both should have method arguments defined otherwise we may get compile time error. See below example.



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()** |

## Instance Method Reference of an Arbitrary Object

When you refer to the instance method of a class with the ClassName , you will get the instance method reference of an arbitrary object of a particular type, such as***ClassName::someInstanceMethod***;

Comparator<String>stringIgnoreCase = String::compareToIgnoreCase;

//this is equivalent to

Comparator<String>stringComparator = (first, second) ->first.compareToIgnoreCase(second);

# ForEach loop in java 8

**Definition of forEach**default void forEach(Consumer<? super T> action)

*Consumer* is functional interface and as *accept* method

**publicclass**ForEachDemo {

**publicstaticvoid**main(String[] args) {

List<Employee>employees= **new**UtilClass().getAllEmployees();

employees.forEach(**new**MyConsmer());

}

}

**class**MyConsmer**implements** Consumer<Employee>{

s@Override

**publicvoid**accept(Employee t) {

System.***out***.println(t);

}

}

We can use lambda expression here for calling consumer method

employees.forEach((emp)->{

System.***out***.println(emp);

});

Or we can use method reference

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

Examples:

1. To print even numbers.
2. To find distinct element of stream
3. To find max and min of the stream

[1. What is a Stream? Stream vs Collection?](https://howtodoinjava.com/java8/java-streams-by-examples/#1-what-is-a-stream-stream-vs-collection)

[2. Creating Streams](https://howtodoinjava.com/java8/java-streams-by-examples/#2-creating-streams)

[2.1. Stream.of()](https://howtodoinjava.com/java8/java-streams-by-examples/#21-streamof)

[2.2. Stream.of(array)](https://howtodoinjava.com/java8/java-streams-by-examples/#22-streamofarray)

[2.3. List.stream()](https://howtodoinjava.com/java8/java-streams-by-examples/#23-liststream)

[2.4. Stream.generate() or Stream.iterate()](https://howtodoinjava.com/java8/java-streams-by-examples/#24-streamgenerate-or-streamiterate)

[2.5. Stream of String chars or tokens](https://howtodoinjava.com/java8/java-streams-by-examples/#25-stream-of-string-chars-or-tokens)

[3. Stream Collectors](https://howtodoinjava.com/java8/java-streams-by-examples/#3-stream-collectors)

[3.1. Collect Stream elements to a List](https://howtodoinjava.com/java8/java-streams-by-examples/#31-collect-stream-elements-to-a-list)

* 1. [Collect Stream elements to an Array](https://howtodoinjava.com/java8/java-streams-by-examples/#32-collect-stream-elements-to-an-array)
  2. Collect stream to a set

[4. Stream Operations](https://howtodoinjava.com/java8/java-streams-by-examples/#4-stream-operations)

[4.1. Intermediate Operations](https://howtodoinjava.com/java8/java-streams-by-examples/#41-intermediate-operations)

[4.2. Terminal operations](https://howtodoinjava.com/java8/java-streams-by-examples/#42-terminal-operations)

[5. Short-circuit Operations](https://howtodoinjava.com/java8/java-streams-by-examples/#5-short-circuit-operations)

[5.1. Stream.anyMatch()](https://howtodoinjava.com/java8/java-streams-by-examples/#51-streamanymatch)

[5.2. Stream.findFirst()](https://howtodoinjava.com/java8/java-streams-by-examples/#52-streamfindfirst)

[6. Parallel Streams](https://howtodoinjava.com/java8/java-streams-by-examples/#6-parallel-streams)

[7. Stream Operations](https://howtodoinjava.com/java8/java-streams-by-examples/#7-stream-operations)

[7.1 Creating Streams](https://howtodoinjava.com/java8/java-streams-by-examples/#71-creating-streams)

[7.2 Intermediate Operations](https://howtodoinjava.com/java8/java-streams-by-examples/#72-intermediate-operations)

[7.3. Terminal Operations](https://howtodoinjava.com/java8/java-streams-by-examples/#73-terminal-operations)

[8. Java Stream Examples](https://howtodoinjava.com/java8/java-streams-by-examples/#8-java-stream-examples)

# Functional interfaces

1. Predicate & BiPredicate
2. Function & BiFunction
3. Consumer & supplier

**Predicate**

* A predicate is a function with a single argument and returns boolean value.
* It’s a functional interface and it contains only one method i.e., **test()** , it accepts one argument and return Boolean value.
* We can join multiple predicates by using and(), or() predicates also.

|  |  |
| --- | --- |
| Methods | Description |
| boolean **test**(T t) | It evaluates this predicate on the given argument. |
| default Predicate<T> **and**(Predicate<? super T> other) | It returns a composed predicate that represents a short-circuiting logical AND of this predicate and another. When evaluating the composed predicate, if this predicate is false, then the other predicate is not evaluated. |
| default Predicate<T> **negate**() | It returns a predicate that represents the logical negation of this predicate. |
| default Predicate<T> **or**(Predicate<? super T> other) | It returns a composed predicate that represents a short-circuiting logical OR of this predicate and another. When evaluating the composed predicate, if this predicate is true, then the other predicate is not evaluated. |
| static <T> Predicate<T> **isEqual**(Object targetRef) | It returns a predicate that tests if two arguments are equal according to Objects.equals(Object, Object). |

**Function**

* It is same as predicate only the difference is it accepts two argument and returns value of any type.
* It’s a functional interface and it contains only one method i.e., **apply()**

|  |  |
| --- | --- |
| **Method** | **Description** |
| default <V> Function<T,V> **andThen**(Function<? super R,? extends V> after) | It returns a composed function that first applies this function to its input, and then applies the after function to the result. If evaluation of either function throws an exception, it is relayed to the caller of the composed function. |
| static <T> Function<T,T> identity() | It returns a function that always returns its input argument. |
| R **apply**(T t) | It applies this function to the given argument. |
| default <V> Function<V,R> compose(Function<? super V,? extends T> before) | It Returns a composed function that first applies the before function to its input, and then applies this function to the result. If evaluation of either function throws an exception, it is relayed to the caller of the composed function. |

# Collection Sorting

We can use sort() of collection or we can use sorted() of java 8 to sort the collection object.

We can sort collection ascending by sort()/sorted() method but if we have to use custom sorting, we should use **Comparator.**

Comparator interface has a static method called **comparing** which accepts **Function.** In the function we have to return the value by which we have to sort our collection.

*The function we are using in comparing method can be implemented by* ***method reference or lambda expression.***

If we have to sort list of employee by firstName, then comparator will be

// using method reference

Comparator<Course> compareByFistName = Comparator.comparing(**Course::getFirstName**);

//using lambda ex

Comparator<Course> compareByFistName = Comparator.comparing(**object->object. getFirstName ()**);

Collection sort mehod : <https://howtodoinjava.com/java/sort/collections-sort/>

# Default methods

* Default methods in interfaces help us to **introduce new functionality without breaking the contract of the implementing classes**.
* Helps to **solve diamond problem** which occurs in multiple inheritance.
* No compulsion to override the default method
* We can override based on requirement.

# Optional class

* deal with NullPointerException in Java application
* It provides methods which are used to check the presence of value for particular variable.

Reference links

<https://dzone.com/articles/java-8-method-references>

<https://howtodoinjava.com/java8/java-streams-by-examples/#3-stream-collectors>