**Java 8 Streams API:**

**What Are Streams?**

Streams can be defined as a sequence of elements from a source which support data processing operations. You can treat streams as operations on data.

Why Streams?

Almost every Java application use Collections API to store and process the data. Despite being the most used Java API, it is not easy to write the code for even some common data processing operations like filtering, finding, matching, sorting, mapping etc using Collections API . So, there needed Next-Gen API to process the data. So, Java API designers have come with Java 8 Streams API to write more complex data processing operations with much of ease.

Characteristics Of Java 8 Streams

1) Streams are not the data structures

Streams doesn’t store the data. You can’t add or remove elements from streams. Hence, they are not the data structures. They are the just operations on data.

2)Stream Consumes a data source

Stream consumes a source, performs operations on it and produces the result. Source may be a collection or an array or an I/O resource. Remember, stream doesn’t modify the source.

3)**Intermediate And Terminal Operations**

Most of the stream operations return another new stream and they can be chained together to form a pipeline of operations.

The operations which return stream themselves are called intermediate operations. For example – filter(), distinct(), sorted() etc.

The operations which return other than stream are called terminal operations. count(). min(), max() are some terminal operations.

4) **Pipeline Of Operations:**

A pipeline of operations consists of three things – a source, one or more intermediate operations and a terminal operation. Pipe-lining of operations let you to write database-like queries on a data source. In the below example, int array is the source, filter() and distinct() are intermediate operations and forEach() is a terminal operation.

import java. util. stream.IntStream;

public class TestStream {

public static void main(String[] args) {  
IntStream.of(new int[] {4, 7, 1, 8, 3, 9, 7}).filter((int i) -> i > 5).distinct().forEach(System.out::println);

// IntStream : A sequence of primitive int-valued elements

}}

5**) Streams are traversable only once**

You can’t traverse the streams more than once just like iterators. If you traverse the stream first time, it is said to be consumed.

List<String> nameList = Arrays.asList("Dinesh", "Ross", "Kagiso", "Steyn");

Stream<String> stream = nameList.stream();

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

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

//Error : stream has already been operated upon or closed

6) Short Circuiting Operations

Short circuiting operations are the operations which don’t need the whole stream to be processed to produce a result. For example – findFirst(), findAny(), limit() etc.

 7) Execution

To gain the performance while processing the large amount of data, you have to process it in parallel and use multi core architectures. Java 8 Streams can be processed in parallel without writing any multi threaded code. For example, to process the collections in parallel, you just use *parallelStream()* method instead of *stream()* method.

//Normal Execution

List<String> names = **new** ArrayList<>();

names.add("David");

names.add("Johnson");

names.add("Samontika");

names.add("Brijesh");

names.add("John")

names.stream().filter((String name) -> name.length() > 5).skip(2).forEach(System.out::println);

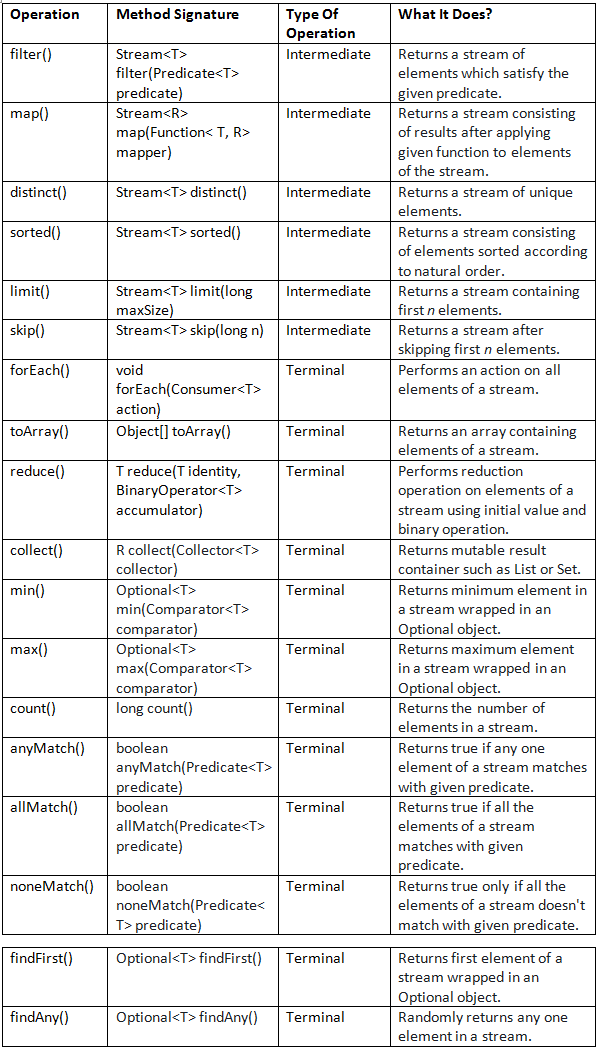
//Parallel Execution: no order bcoz execute parallely

System.out.println("parallel stream--------");

names.parallelStream().filter((String name) -> name.length() > 5).skip(2).forEach(System.out::println);

**java.util.stream.Stream**

java.util.stream.Stream interface is the center of Java 8 Streams API. This interface contains all the stream operations. Below table shows frequently used Stream methods with description.



### **Stream Creation Operations:**

### **1)empty() : Creates an empty stream**

Method Signature : public static<T> Stream<T> empty()

Type Of Method : Static Method

What It Does? : Returns an empty stream of type T.

#### **2) Creates a stream of single element of type T**

Method Signature: public static<T> Stream<T> of(T t)

Type Of Method : Static Method

What It Does? : Returns a single element stream of type T.

Stream<Student> singleElementStream = Stream.of(**new** Student());

System.out.println(singleElementStream.count());

#### **3)Creates a stream from values:**

Method Signature : public static<T> Stream<T> of(T… values)

Type Of Method : Static Method

What It does? : Returns a stream consisting of supplied values as elements.

Stream<Integer> streamOfNumbers = Stream.of(7, 2, 6, 9, 4, 3, 1);

System.out.println(streamOfNumbers.count());

#### **4)Creating streams from collections**

From Java 8, every collection type will have a method called stream() which returns the stream of respective collection type.

List<String> listOfStrings = **new** ArrayList<>();

listOfStrings.add("One");

listOfStrings.add("Two");

listOfStrings.add("Three");

listOfStrings.stream().forEach(System.out::println);

### **Selection Operations**

***1)***

***filter()* : Selecting with a predicate**

Method Signature : Stream<T> filter(Predicate<T> predicate)

Type Of Operation : Intermediate Operation

What it does? : Returns a stream of elements which satisfy the given predicate.

List<String> names = new ArrayList<>();

names.add("David");

names.add("Johnson");

names.add("Samontika");

names.add("Brijesh");

names.add("John");

//Selecting names containing more than 5 characters

names.stream().filter((String name) -> name.length() > 5).forEach(System.out::println);

***names.stream().filter((String n)->n.length()>5).forEach(System.out::println);***

***2)***

***distinct()* : Selects only unique elements**

Method Signature : Stream<T> distinct()

Type Of Operation : Intermediate Operation

What It Does? : Returns a stream of unique elements.

List<String> names = **new** ArrayList<>();

names.add("David");

names.add("Johnson");

names.add("Samontika");

names.add("Brijesh");

names.add("John");

names.add("David");

names.add("Brijesh");

//Selecting only unique names

names.stream().distinct().forEach(System.out::println);

3)

#### limit() : Selects first n elements

Method Signature : Stream<T> limit(long maxSize)

Type Of Operation : Intermediate Operation

What It Does? : Returns a stream containing first n elements.

List<String> names = new ArrayList<>();

names.add("David");

names.add("Johnson");

names.add("Samontika");

names.add("Brijesh");

names.add("John");

names.add("David");

names.add("Brijesh");

//Selecting first 4 names

names.stream().limit(4).forEach(System.out::println);

4)

skip

Method Signature : Stream<T> skip(long n)

Type Of Operation : Intermediate Operation

What It Does? : Returns a stream after skipping first n elements.

List<String> names = **new** ArrayList<>();

names.add("David");

names.add("Johnson");

names.add("Samontika");

names.add("Brijesh");

names.add("John");

names.add("David");

names.add("Brijesh");

//Skipping first 4 names

names.stream().skip(4).forEach(System.out::println);

### **Mapping Operations**

#### map() : Applies a function

Method Signature : Stream<R> map(Function<T, R> mapper);

Type Of Operation : Intermediate Operation

What It Does? : Returns a stream consisting of results after applying given function to elements of the stream.

List<String> names = new ArrayList<>();

names.add("David");

names.add("Johnson");

names.add("Samontika");

names.add("Brijesh");

names.add("John");

//Returns length of each name

names.stream().map(String::length).forEach(System.out::println);

**Sorting operations:**

#### 1)

#### sorted() : Sorting according to natural order

Method Signature : Stream<T> sorted()

Type Of Operation : Intermediate Operation

What It Does? : Returns a stream consisting of elements sorted according to natural order.

List<String> names = new ArrayList<>();

names.add("David");

names.add("Johnson");

names.add("Samontika");

names.add("Brijesh");

names.add("John");

//Sorting the names according to natural order

names.stream().sorted().forEach(System.out::println);

2)

#### sorted(Comparator) : Sorting according to supplied comparator

Method Signature : Stream<T> sorted(Comparator<T> comparator)

Type Of Operation : Intermediate Operation

What It Does? : Returns a stream consisting of elements sorted according to supplied Comparator.

List<String> names = **new** ArrayList<>();

names.add("David");

names.add("Johnson");

names.add("Samontika");

names.add("Brijesh");

names.add("John");

//Sorting the names according to their length

names.stream().sorted((String name1, String name2) -> name1.length() - name2.length()).forEach(System.out::println);

### **Reducing Operations**

Reducing operations are the operations which combine all the elements of a stream repeatedly to produce a single value. For example, counting number of elements, calculating average of elements, finding maximum or minimum of elements etc.

reduce () : Produces a single value

Method Signature : T reduce(T identity, BinaryOperator<T> accumulator);

Type Of Operation: Terminal Operation

What It Does? : This method performs reduction operation on elements of a stream using initial value and binary operation.

int sum = Arrays.stream(new int[] {7, 5, 9, 2, 8, 1}).reduce(0, (a, b) -> a+b);

//Output : 32

#### min() : Finding the minimum

Method Signature : Optional<T> min(Comparator<T> comparator)

Type Of Operation : Terminal Operation

What It Does? : It returns minimum element in a stream wrapped in an Optional object.

OptionalInt min = Arrays.stream(new int[] {7, 5, 9, 2, 8}).min();

System.out.println(min);

***max()* : Finding the maximum**

Method Signature : Optional<T> max(Comparator<T> comparator)

Type Of Operation : Terminal Operation

What It Does? : It returns maximum element in a stream wrapped in an Optional object.

OptionalInt max = Arrays.stream(**new** **int**[] {7, 5, 9, 2, 8, 1}).max();

#### **count() : Counting the elements**

Method Signature : long count()

Type Of Operation : Terminal Operation

What It Does? : Returns the number of elements in a stream.

//Counting the names with length > 5

// long noOfBigNames = names1.stream().filter((String name) -> name.length() > 5).count();

**long** noOfBigNames = names1.stream().count();

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

**Finding And Matching Operations:**

#### **anyMatch()** : Any one element matches

Method Signature : boolean anyMatch(Predicate<T> predicate)

Type Of Operation : Short-circuiting Terminal Operation

What It Does? : Returns true if any one element of a stream matches with given predicate. This method may not evaluate all the elements of a stream. Even if the first element matches with given predicate, it ends the operation.

#### **allMatch()** : All elements matches

Method Signature : boolean allMatch(Predicate<T> predicate)

Type Of Operation : Terminal Operation

What It Does? : This method returns true if all the elements of a stream matches with given predicate. Otherwise returns false.

List<String> names2 = new ArrayList<>();

names2.add("Sampada");

names2.add("Johnson");

names2.add("Samontika");

names2.add("Brijesh");

if(names2.stream().allMatch((String name) -> name.length() > 5))

{

System.out.println("All are big names");

}

#### findFirst() : Finding first element

Method Signature : Optional<T> findFirst()

Type Of Operation : Short-circuiting Terminal Operation

What It Does? : Returns first element of a stream wrapped in an Optional object.

Optional<String> firstElement = Stream.*of*("First", "Second", "Third", "Fourth").findFirst();

#### findAny() : Finding any element

Method Signature : Optional<T> findAny()

Type Of Operation : Short-circuiting Terminal operation

What It Does? : Randomly returns any one element in a stream. The result of this operation is unpredictable. It may select any element in a stream. Multiple invocations on the same source may not return same result.

Optional<String> anyElement = Stream.of("First", "Second", "Third", "Fourth").findAny(); System.out.println(anyElement);

Other operations:

#### forEach() :

Method Signature : void forEach(Consumer<T> action)

Type Of Operation : Terminal Operation

What It Does? : Performs an action on all elements of a stream.

Stream.of("First", "Second", "Second", "Third", "Fourth").limit(3).distinct().forEach(System.out::println);

#### toArray() : Stream to array

Method Signature : Object[] toArray()

Type Of Operation : Terminal Operation

What It Does? : Returns an array containing elements of a stream.

List<String> names = **new** ArrayList<>();

names.add("David");

names.add("Johnson");

names.add("Samontika");

names.add("Brijesh");

names.add("John");

//Storing first 3 names in an array

Object[] streamArray = names.stream().limit(3).toArray();

System.out.println(Arrays.toString(streamArray));

Note:

**Example-1**  
Let us create the stream of integers.

Stream<Integer> mystream = Stream.of(10, 12, 14, 16);

Suppose we iterate and print the stream elements.

mystream.forEach(e -> System.out.println(e));

The output will be 10 12 14 16.

**Example-2**  
Let us create stream of string.

Stream<String> mystream = Stream.of("AA", "BB", "CC", "DD");

mystream.forEach(e -> System.out.println(e));

The output will be AA BB CC DD.  
  
**Example-3**  
Find the example to create the stream of objects.  
**StreamOfObjDemo.java**

package com.concretepage;

import java.util.stream.Stream;

public class StreamOfObjDemo {

public static void main(String[] args) {

Stream<User> userStream = Stream.of(

new User("Mahesh", 22),

new User("Krishn", 20),

new User("Suresh", 25)

);

userStream.forEach(u -> System.out.println(u.getUserName()));

}

}

class User {

private String userName;

private int age;

public User(String userName, int age) {

this.userName = userName;

this.age = age;

}

//Sets and Gets

}

Example-4

To create IntStream, we use IntStream.of method.

To create LongStream, we use LongStream.of method.

To create DoubleStream, we use DoubleStream.of method.

Find the examples.

**StreamOfDemo.java**

package com.concretepage;

import java.util.stream.DoubleStream;

import java.util.stream.IntStream;

import java.util.stream.LongStream;

public class StreamOfDemo {

public static void main(String[] args) {

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

IntStream intStream = IntStream.of(12, 14, 16);

intStream.forEach(e -> System.out.println(e));

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

LongStream longStream = LongStream.of(154L, 236L, 306L);

longStream.forEach(e -> System.out.println(e));

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

DoubleStream doubleStream = DoubleStream.of(123.56, 456.87, 784.65);

doubleStream.forEach(e -> System.out.println(e));

}

}

## Creating Streams

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

### 2.1. 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));

### 2.2. 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](https://howtodoinjava.com/java/collections/arraylist/java-arraylist/). 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));

### 2.4. 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);

### 2.5. 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.

## 3. 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](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html) methods.

### 3.1. 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);

### 3.2. 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.

## 4. 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:

### 4.1. Intermediate Operations

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

#### 4.1.1. Stream.filter()

The filter() method accepts a [Predicate](https://howtodoinjava.com/java8/how-to-use-predicate-in-java-8/) to filter all elements of the stream. This operation is intermediate, enabling us to call another stream operation (e.g. [forEach()](https://howtodoinjava.com/java8/java-stream-foreach/)) on the result.

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

.forEach(System.out::println);

Program Output:

Amitabh

Aman

#### 4.1.2. 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

#### 4.1.2. 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](https://howtodoinjava.com/java/collections/java-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.

### 4.2. 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.

#### 4.2.1. 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);

#### 4.2.2. 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]

#### 4.2.3. 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*

#### 4.2.4. 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*

#### 4.2.5. Stream.reduce()

The reduce() method performs a reduction on the elements of the stream with the given function. The result is an [Optional](https://howtodoinjava.com/java8/java-8-optionals-complete-reference/) 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*

## 5. 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](https://howtodoinjava.com/java/flow-control/if-else-statement-in-java/). In the internal iterations such as in streams, there are certain methods we can use for this purpose.

### 5.1. 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*

### 5.2. 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*

## 6. Parallel Streams

With the [Fork/Join framework](https://howtodoinjava.com/java/forkjoin-framework-tutorial-forkjoinpool-example/) 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 that have the potential to crash the application. With the introduction of [internal iterations](https://howtodoinjava.com/java8/internal-vs-external-iteration/), 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 our 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](https://howtodoinjava.com/series/java-concurrency/) and [parallelism](https://howtodoinjava.com/java/multi-threading/concurrency-vs-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.

## 7. Stream Methods

### 7.1 Creating Streams

* concat()
* empty()
* generate()
* iterate()
* of()

### 7.2 Intermediate Operations

* [filter()](https://howtodoinjava.com/java8/java-stream-filter-example/)
* [map()](https://howtodoinjava.com/java8/stream-map-example/)
* [flatMap()](https://howtodoinjava.com/java8/stream-flatmap-example/)
* [distinct()](https://howtodoinjava.com/java8/java-stream-distinct-examples/)
* [sorted()](https://howtodoinjava.com/java8/stream-sorted-method/)
* [peek()](https://howtodoinjava.com/java8/java-stream-peek-example/)
* [limit()](https://howtodoinjava.com/java8/java-stream-limit-method-example/)
* [skip()](https://howtodoinjava.com/java8/stream-skip-example/)

### 7.3. Terminal Operations

* [forEach()](https://howtodoinjava.com/java8/java-stream-foreach/)
* [forEachOrdered()](https://howtodoinjava.com/java8/java-stream-foreachordered/)
* [toArray()](https://howtodoinjava.com/java8/convert-stream-to-array/)
* reduce()
* collect()
* [min()](https://howtodoinjava.com/java8/java-stream-min/)
* [max()](https://howtodoinjava.com/java8/java-stream-max/)
* [count()](https://howtodoinjava.com/java8/stream-count-elements-example/)
* [anyMatch()](https://howtodoinjava.com/java8/stream-anymatch-example/)
* [allMatch()](https://howtodoinjava.com/java8/stream-allmatch-example/)
* [noneMatch()](https://howtodoinjava.com/java8/stream-nonematch-example/)
* [findFirst()](https://howtodoinjava.com/java8/stream-findfirst-findany/)
* [findAny()](https://howtodoinjava.com/java8/stream-findfirst-findany/)