Important Point :

i -> System.out.print(i) <=> System.out::print

**Stream**

1. **Function and BiFunction**: Function represents a function that takes one type of argument and returns another type of argument. Function<T, R> is the generic form where T is the type of the input to the function and R is the type of the result of the function.

For handling primitive types, there are specific Function interfaces – ToIntFunction, ToLongFunction, ToDoubleFunction, ToIntBiFunction, ToLongBiFunction, ToDoubleBiFunction, LongToIntFunction, LongToDoubleFunction, IntToLongFunction, IntToDoubleFunction etc.

Some of the Stream methods where Function or it’s primitive specialization is used are:

* + <R> Stream<R> map(Function<? super T, ? extends R> mapper)
  + IntStream mapToInt(ToIntFunction<? super T> mapper) – similarly for long and double returning primitive specific stream.
  + IntStream flatMapToInt(Function<? super T, ? extends IntStream> mapper) – similarly for long and double
  + <A> A[] toArray(IntFunction<A[]> generator)
  + <U> U reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner)

1. **Predicate and BiPredicate**: It represents a predicate against which elements of the stream are tested. This is used to filter elements from the java stream. Just like Function, there are primitive specific interfaces for int, long and double.

Some of the Stream methods where Predicate or BiPredicate specializations are used are:

* + Stream<T> filter(Predicate<? super T> predicate)
  + boolean anyMatch(Predicate<? super T> predicate)
  + boolean allMatch(Predicate<? super T> predicate)
  + boolean noneMatch(Predicate<? super T> predicate)

1. **Consumer and BiConsumer**: It represents an operation that accepts a single input argument and returns no result. It can be used to perform some action on all the elements of the java stream.

Some of the Java 8 Stream methods where Consumer, BiConsumer or it’s primitive specialization interfaces are used are:

* + Stream<T> peek(Consumer<? super T> action)
  + void forEach(Consumer<? super T> action)
  + void forEachOrdered(Consumer<? super T> action)

1. **Supplier**: Supplier represent an operation through which we can generate new values in the stream. Some of the methods in Stream that takes Supplier argument are:
   * public static<T> Stream<T> generate(Supplier<T> s)
   * <R> R collect(Supplier<R> supplier,BiConsumer<R, ? super T> accumulator,BiConsumer<R, R> combiner)

**java.util.Optional**

[Java Optional](https://www.journaldev.com/16709/java-optional) is a container object which may or may not contain a non-null value. If a value is present, isPresent() will return true and get() will return the value. Stream terminal operations return Optional object. Some of these methods are:

* Optional<T> reduce(BinaryOperator<T> accumulator)
* Optional<T> min(Comparator<? super T> comparator)
* Optional<T> max(Comparator<? super T> comparator)
* Optional<T> findFirst()
* Optional<T> findAny()

**java.util.Spliterator**

For supporting parallel execution in Java 8 Stream API, Spliterator interface is used. Spliterator trySplit method returns a new Spliterator that manages a subset of the elements of the original Spliterator.

**Java Stream Intermediate and Terminal Operations**

Java Stream API operations that returns a new Stream are called intermediate operations. Most of the times, these operations are lazy in nature, so they start producing new stream elements and send it to the next operation. Intermediate operations are never the final result producing operations. Commonly used intermediate operations are filter and map.

Java 8 Stream API operations that returns a result or produce a side effect. Once the terminal method is called on a stream, it consumes the stream and after that we can’t use stream. Terminal operations are eager in nature i.e they process all the elements in the stream before returning the result. Commonly used terminal methods are forEach, toArray, min, max, findFirst, anyMatch, allMatch etc. You can identify terminal methods from the return type, they will never return a Stream.

**Java Stream Short Circuiting Operations**

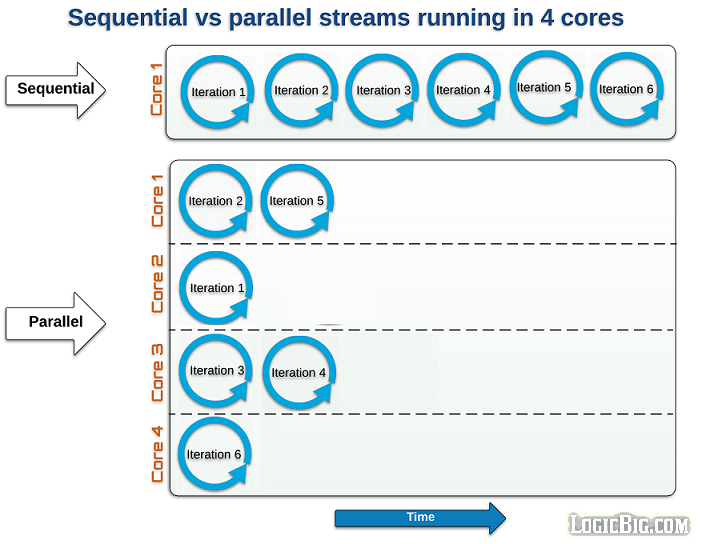
An intermediate operation is called short circuiting, if it may produce finite stream for an infinite stream. For example limit() and skip() are two short circuiting intermediate operations.

A terminal operation is called short circuiting, if it may terminate in finite time for infinite stream. For example anyMatch, allMatch, noneMatch, findFirst and findAny are short circuiting terminal operations.

# Java 8 Streams - Sequential vs Parallel streams

Parallel streams divide the provided task into many and run them in different threads, utilizing multiple cores of the computer. On the other hand sequential streams work just like for-loop using a single core.

The tasks provided to the streams are typically the iterative operations performed on the elements of a collection or array or from other dynamic sources. Parallel execution of streams run multiple iterations simultaneously in different available cores.



In parallel execution, if number of tasks are more than available cores at a given time, the remaining tasks are queued waiting for currently running task to finish.

It is also important to know that iterations are only performed at a terminal operation. The streams are designed to be lazy.

### Example

Let's test sequential and parallel behavior with an example.

import java.time.LocalTime;

import java.util.Arrays;

import java.util.stream.Stream;

public class SequentialParallelComparison {

public static void main (String[] args) {

String[] strings = {"1", "2", "3", "4", "5", "6", "7", "8", "9", "10"};

System.out.println("-------\nRunning sequential\n-------");

run(Arrays.stream(strings).sequential());

System.out.println("-------\nRunning parallel\n-------");

run(Arrays.stream(strings).parallel());

}

public static void run (Stream<String> stream) {

stream.forEach(s -> {

System.out.println(LocalTime.now() + " - value: " + s +

" - thread: " + Thread.currentThread().getName());

try {

Thread.sleep(200);

} catch (InterruptedException e) {

e.printStackTrace();

}

});

}

}

In above example we are printing various information, i.e. time, collection element value and thread name. We are doing that in forEach() terminal function. Other than parallel() and sequential(), we are not using any other intermediate operations, but that doesn't matter if we use the same intermediate operations for the both. We are also making each iteration to sleep for 200ms so that we can clearly compare the time taken by sequential and parallel invocations.

#### Output:

Following is the output, on an 8 logical processors (4 Core) machine.

-------

Running sequential

-------

02:29:02.817 - value: 1 - thread: main

02:29:03.022 - value: 2 - thread: main

02:29:03.223 - value: 3 - thread: main

02:29:03.424 - value: 4 - thread: main

02:29:03.624 - value: 5 - thread: main

02:29:03.824 - value: 6 - thread: main

02:29:04.025 - value: 7 - thread: main

02:29:04.225 - value: 8 - thread: main

02:29:04.426 - value: 9 - thread: main

02:29:04.626 - value: 10 - thread: main

-------

Running parallel

-------

02:29:04.830 - value: 7 - thread: main

02:29:04.830 - value: 3 - thread: ForkJoinPool.commonPool-worker-1

02:29:04.830 - value: 8 - thread: ForkJoinPool.commonPool-worker-4

02:29:04.830 - value: 2 - thread: ForkJoinPool.commonPool-worker-3

02:29:04.830 - value: 9 - thread: ForkJoinPool.commonPool-worker-2

02:29:04.830 - value: 5 - thread: ForkJoinPool.commonPool-worker-5

02:29:04.830 - value: 1 - thread: ForkJoinPool.commonPool-worker-6

02:29:04.831 - value: 10 - thread: ForkJoinPool.commonPool-worker-7

02:29:05.030 - value: 4 - thread: ForkJoinPool.commonPool-worker-3

02:29:05.030 - value: 6 - thread: ForkJoinPool.commonPool-worker-2

This clearly shows that in sequential stream, each iteration waits for currently running one to finish, whereas, in parallel stream, eight threads are spawn simultaneously, remaining two, wait for others. Also notice the name of threads. In parallel stream, Fork and Join framework is used in the background to create multiple threads. Parallel streams create ForkJoinPool instance via static ForkJoinPool.commonPool() method.

**Optional API**

* Useful link : <https://www.journaldev.com/16709/java-optional>
* One of the most frequently [exception](https://www.journaldev.com/1696/exception-handling-in-java) in java programming is [NullPointerException](https://www.journaldev.com/14544/java-lang-nullpointerexception). A Null value often represents an absence of value which has to be handled before proceeding with the usual business logic, which leads to unnecessary null checks.
* To handle such boiler plate code for null check situations, Java 8 introduced Optionalclass. In this article, we’ll look into details how Java 8 Optional class API helps us to deal with null values.
* [Java 8 stream](https://www.journaldev.com/2774/java-8-stream) API and collection methods can return Optional objects. It may or may not contain a non-null value. There are various methods available in the API to deal with the Optional value in a convenient and reliable manner.
* Java Optional is a final class.

How to create an empty Optional object?

Using static empty() of Optional class

* Optional<String> empty = Optional.*empty*();