Java 8 Stream Intermediate And Terminal Operations :

1) The main difference between intermediate and terminal operations is that intermediate operations return a stream as a result and terminal operations return non-stream values like primitive or object or collection or may not return anything.

2) As intermediate operations return another stream as a result, they can be chained together to form a pipeline of operations. Terminal operations can not be chained together.

3) Pipeline of operations may contain any number of intermediate operations, but there has to be only one terminal operation, that too at the end of pipeline.

4) Intermediate operations are lazily loaded. When you call intermediate operations, they are actually not executed. They are just stored in the memory and executed when the terminal operation is called on the stream.

Stateful Intermediate Operations: Stateful intermediate operations are those which maintain information from a previous invocation internally(aka state) to be used again in a future invocation of the method. Intermediate operations such as Stream.distinct(), which needs to remember(or store) the previous elements it encountered, have to store state information from previous passes. This state storage can become huge for instances of infinite streams and hence can potentially affect performance of the whole system. Another example of stateful intermediate operation is Streams.sorted() which requires to store elements in a temporary storage as it sorts them over multiple passes.

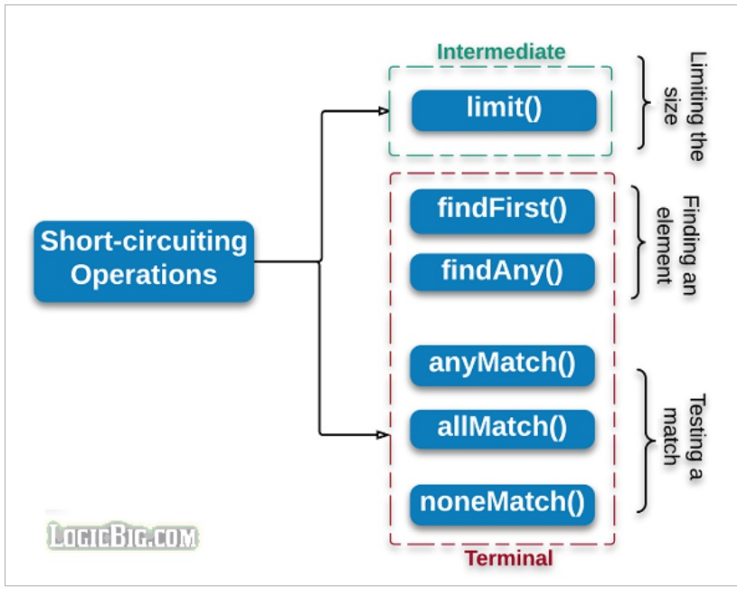
Stateless Intermediate Operations: Stateless intermediate operations are the opposite of stateful and do not store any state across passes. This not only improves the performance of these operations, which include among others filter(), map(), findAny(), it also helps in executing the Stream operation invocations in parallel as there is no information to be shared, or any order to be maintained, between these invocations or passes.

Short Circuit Operations

In boolean short-circuiting logic, for example `firstBoolean && secondBoolean`, if firstBoolean is false then the remaining part of the expression is ignored (the operation is short-circuited) because the remaining evaluation will be redundant. Similarly in `firstBoolean || secondBoolean`, if firstBoolean is true the remaining part is short-circuited.

Java 8 Stream short-circuit operations are not limited to boolean types. There are pre defined short-circuiting operations.

Java 8 stream intermediate and terminal operations both can be short circuiting.



LIMIT()::Returns a new stream created from this stream, truncated to be no longer than maxSize in length.

```
stream.filter(i -> i % 2 == 0)
      .limit(2)
      .forEach(System.out::println);
```

These terminal-short-circuiting methods can finish before transversing all the elements of the underlying stream. A short-circuiting terminal operation, when operating on infinite input data source, may terminate in finite time.

Optional<T> findFirst():

Returns the very first element (wrapped in Optional object) of this stream and before transversing the other.

`boolean anyMatch(Predicate<? super T> predicate)`

Tests whether any elements of this stream match the provided predicate. This terminal method will return as soon as it finds the match and will not transverse all the remaining elements to apply the predicate.

Following example prints 'true':

```
Stream<String> stream = Stream.of("one", "two","three", "four");
boolean match = stream.anyMatch(s -> s.contains("our"));
System.out.println(match);
```

`boolean allMatch(Predicate<? super T> predicate)`

Tests whether all elements match the provided predicate. It may return early with false result when any element doesn't match first.

Following examples outputs false because 'Three' doesn't start with a lower case, at that point short-circuiting happens.

```
Stream<String> stream = Stream.of("one", "two", "Three", "four");
boolean match = stream.allMatch(s -> s.length() > 0 &&
                        Character.isLowerCase(s.charAt(0)));
System.out.println(match);
```

- `boolean noneMatch(Predicate<? super T> predicate)`

  Tests whether no elements of this stream match the provided predicate. It may return early with false result when any element matches the provided predicate first.

  Following example will print 'true' because none of the elements start with an upper case.

  ```
  Stream<String> stream = Stream.of("one", "two", "three", "four");
  boolean match = stream.noneMatch(s -> s.length() > 0 &&
                           Character.isUpperCase(s.charAt(0)));
  System.out.println(match);
  ```

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(), none-Match(), findFirst(), findAny()