

Streams

Terminating the Stream

Terminal Stream Operations

all of the contents of the stream are
combined into a single primitive or Object



Method	When applied on infinite stream	Return value	Reduction
count()	Does not terminate	long	Yes
min() max()	Does not terminate	Optional<T>	Yes
findAny() findFirst()	Terminates	Optional<T>	No
allMatch() anyMatch() noneMatch()	Sometimes terminates	boolean	No
forEach()	Does not terminate	void	No
reduce()	Does not terminate	varies	Yes
collect()	Does not terminate	varies	Yes

```
// counting
```

```
Stream<String> names = Stream.of("John", "George", "Ben");
```

```
System.out.println(names.count());
```

=> 3

terminates the stream and returns long value

```
// for the infinite stream, count() never terminates
```

```
// finding minimum and maximum
```

```
Stream<String> names = Stream.of("John", "George", "Ben");
```

```
Optional<String> min = names.min((s1, s2) -> s1.length() - s2.length());
```

return type is `Optional`

method argument is `Comparator`

```
min.ifPresent(System.out::println);
```

=> Ben

```
// using with empty stream
```

```
Optional<?> minEmpty = Stream.empty().min((s1, s2) -> 0);
```

```
System.out.println(minEmpty.isPresent());
```

=> false

```
// these methods hang if applied to infinite stream
```

// finding a value

```
Stream<String> names = Stream.of("John", "George", "Ben");
```

```
Stream<String> inf = Stream.generate(() -> "Luke");
```

returns Optional

```
names.findAny().ifPresent(System.out::println);
```

=> John (usually the first one)

```
inf.findAny().ifPresent(System.out::println);
```

=> Luke terminates infinite stream

// findFirst() always returns the first element

```
// matching
```

```
var myList = List.of("George", "21", "Ben");
```

```
Stream<String> inf = Stream.generate(() -> "Luke");
```

```
Predicate<String> p = s -> Character.isLetter(s.charAt(0));
```

checks if the name
begins with a letter

```
System.out.println(myList.stream().anyMatch(p));
```

```
=> true
```

```
System.out.println(myList.stream().allMatch(p));
```

```
=> false
```

```
System.out.println(myList.stream().noneMatch(p));
```

```
=> false
```

```
System.out.println(inf.anyMatch(p));
```

```
=> true
```

matching methods terminate infinite streams

```
// iterating
```

```
Stream<String> names = Stream.of("John", "George", "Ben");
```

```
names.forEach(System.out::print);
```

```
=> JohnGeorgeBen
```

method argument is Consumer, return is void

```
// NOTE: you cannot use traditional for loop on the stream!
```

```
Stream<Integer> s = Stream.of(1, 2, 3);
```

```
for (Integer i : s) {
```

```
    // do something
```

```
}
```

```
=> DOES NOT COMPILE
```

```
// forEach() is not really a loop, but rather a terminal operator for streams
```

```
// reducing
```

```
// usually starts with initial value and merge to the next value:
```

```
var myArray = new String[] { "L", "u", "k", "e" };
```

```
var result = "";
```

initial value, called identity

```
for (var s : myArray) result = result + s;
```

accumulator

```
System.out.println(result)
```

=> Luke

```
// same thing using streams
```

```
Stream<String> myStream = Stream.of("L", "u", "k", "e");
```

```
String myName = myStream.reduce("", (s, c) -> s + c);
```

identity

accumulator, passed as BinaryOperator

```
System.out.println(myName);
```

=> Luke


```
// another way
```

```
Stream<String> myStream = Stream.of("L", "u", "k", "e");
```

```
String myName = stream.reduce("", String::concat);
```

```
System.out.println(myName):
```

```
=> Luke
```

```
Stream<Integer> stream = Stream.of(3, 7, 10);
```

```
System.out.println(stream.reduce(1, (a, b) -> a*b));
```

```
=> 210
```

```
// if you omit the identity, Optional will be returned
```

```
// if you omit the identity, Optional will be returned
```

```
BinaryOperator<Integer> op = (a, b) -> a*b;
```

```
Stream<Integer> empty = Stream.empty();
```

```
Stream<Integer> oneElement = Stream.of(7);
```

```
Stream<Integer> threeElements = Stream.of(3, 7, 10);
```

returns Optional

```
empty.reduce(op).ifPresent(System.out::println);
```

=> no output

```
oneElement.reduce(op).ifPresent(System.out::println);
```

=> 7

```
threeElements.reduce(op).ifPresent(System.out::println);
```

=> 210

// when dealing with different types

```
Stream<String> names = Stream.of("John", "George", "Ben");
```

```
int len = names.reduce(0, (i, s) -> i + s.length(), (a, b) -> a + b);
```

initializer *accumulator* *combiner*

```
System.out.println(len);
```

=> 13

```
// collecting (mutable reduction)
```

```
Stream<String> myStream = Stream.of("L", "u", "k", "e");
```

```
StringBuilder myName = myStream.collect(
```

```
    StringBuilder::new, supplier, creates the object that will store the results as we collect data
```

```
    StringBuilder::append, accumulator, passed as BiConsumer
```

```
    StringBuilder::append); combiner, passed as BiConsumer
```

```
System.out.println(myName);
```

```
=> Luke
```

```
// if this was a parallel stream, the order would be unpredictable
```

```
// keeping the order in the collection (sorted)
Stream<String> myStream = Stream.of("L", "u", "k", "e");
TreeSet<String> mySet = myStream.collect(
```

`TreeSet::new,` *supplier*, creates an empty TreeSet

`TreeSet::add,` *accumulator*, adds a single String from Stream to TreeSet

`TreeSet::addAll);` *combiner*, adds all elements of one TreeSet to another

```
System.out.println(mySet);
```

=> [L, e, k, u] (TreeSet automatically sorts the elements in ascending order)

```
// using collectors class
```

```
Stream<String> myStream = Stream.of("L", "u", "k", "e");
```

```
TreeSet<String> mySet = myStream.collect(Collectors.toCollection(TreeSet::new));
```

```
System.out.println(mySet);
```

supplier

```
=> [L, e, k, u]
```

(sorted in ascending order)

```
// if we don't care about the order
```

```
Stream<String> myStream = Stream.of("L", "u", "k", "e");
```

```
Set<String> mySet = myStream.collect(Collectors.toSet());
```

```
System.out.println(mySet);
```

you don't know which implementation

```
=> [u, e, k, L]
```

of Set you will get (most likely HashSet)

(order is unpredictable)