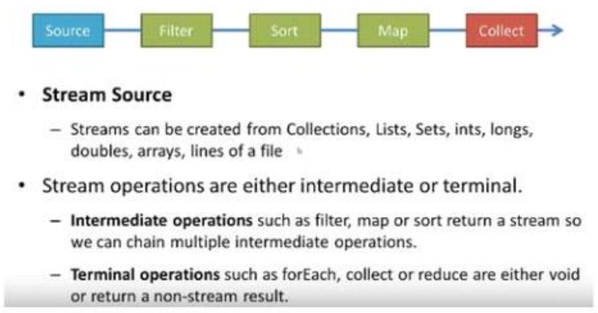
# Java8

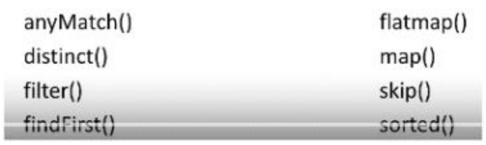
Monday, November 5, 2018 1:17 PM

Streams

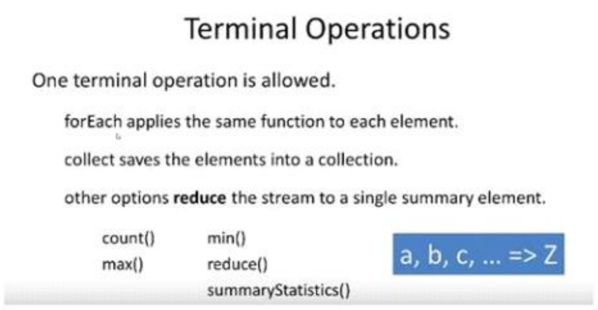

Source
Filter
Sort
Map
Collect

- **Stream Source**
  - Streams can be created from Collections, Lists, Sets, ints, longs, doubles, arrays, lines of a file
- Stream operations are either intermediate or terminal.
  - **Intermediate operations** such as filter, map or sort return a stream so we can chain multiple intermediate operations.
  - **Terminal operations** such as forEach, collect or reduce are either void or return a non-stream result.

Stream source
Intermediate Operations
Terminal Operations



| | |
|---|---|
| anyMatch() | flatmap() |
| distinct() | map() |
| filter() | skip() |
| findFirst() | sorted() |



## Terminal Operations

One terminal operation is allowed.

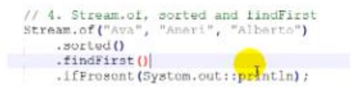
forEach applies the same function to each element.

collect saves the elements into a collection.

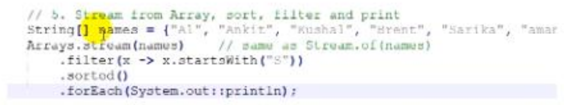
other options **reduce** the stream to a single summary element.

| | |
|---|---|
| count() | min() |
| max() | reduce() |
| | summaryStatistics() |

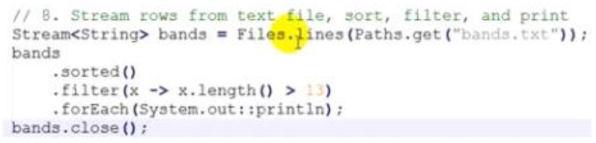
Instream
Stream.of
Arrays.stream
x.stream() -- stream from List, filter and print
Stream<String> -- stream rows from text file, sort, filter and print
.reduce()



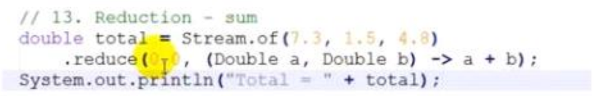
```
// 4. Stream.of, sorted and findFirst
Stream.of("Ava", "Aneri", "Alberto")
    .sorted()
    .findFirst()
    .ifPresent(System.out::println);
```



```
// 5. Stream from Array, sort, filter and print
String[] names = {"Al", "Ankit", "Kushal", "Brent", "Sarika", "amar
Arrays.stream(names)    // same as Stream.of(names)
    .filter(x -> x.startsWith("S"))
    .sorted()
    .forEach(System.out::println);
```



```
// 8. Stream rows from text file, sort, filter, and print
Stream<String> bands = Files.lines(Paths.get("bands.txt"));
bands
    .sorted()
    .filter(x -> x.length() > 13)
    .forEach(System.out::println);
bands.close();
```



```
// 13. Reduction - sum
double total = Stream.of(7.3, 1.5, 4.8)
    .reduce(0.0, (Double a, Double b) -> a + b);
System.out.println("Total = " + total);
```

Java 8 Features
Lambda expressions
Functional Interfaces
    Default & static methods in interface
    Predicate/Function/Consumer [Predefined functional interfaces]
Method and Constructor reference (::)
Streams
Date & Time API [Joda api]

Functional Interfaces
http://tutorials.jenkov.com/java-functional-programming/functional-interfaces.html
A functional interface in Java is an interface that contains only a single abstract (unimplemented) method.

Find smallest integer
int[] arr = new int[]{54,234,1,45,14,54};
int small = Arrays.stream(arr).reduce((x, y) -> x < y ? x : y).getAsInt();

public static void usingRecursion(int number){ if(number > 1){ usingRecursion(number-1); } System.out.println(number); }

Default method
    By using default method, we can provide extra functionality to existing interfaces without impacting implemented classes.
    *sort is added to List interface so that it can be used by ArrayList*
Behaviour Parameterization
    strategy design pattern

8/27/2019

patterns such as filtering, slicing, finding, matching, mapping, and reducing,

*filter*

*sorted*
*map*
*distinct()*
*limit(3)*
*collect*
- *toList()*
- *grouping()*

```
Map<Dish.Type, List<Dish>> dishesByType =
menu.stream().collect(groupingBy(Dish::getType));
```

Finding & Matching
*allMatch*
*anyMatch*
*noneMatch*

*findAny*
*findFirst*

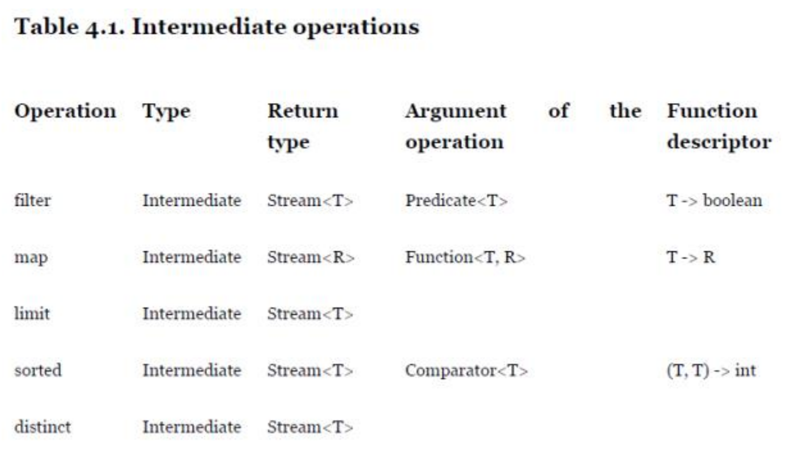


**Table 4.1. Intermediate operations**

| Operation | Type | Return type | Argument of the operation | Function descriptor |
|---|---|---|---|---|
| filter | Intermediate | Stream<T> | Predicate<T> | T -> boolean |
| map | Intermediate | Stream<R> | Function<T, R> | T -> R |
| limit | Intermediate | Stream<T> | | |
| sorted | Intermediate | Stream<T> | Comparator<T> | (T, T) -> int |
| distinct | Intermediate | Stream<T> | | |

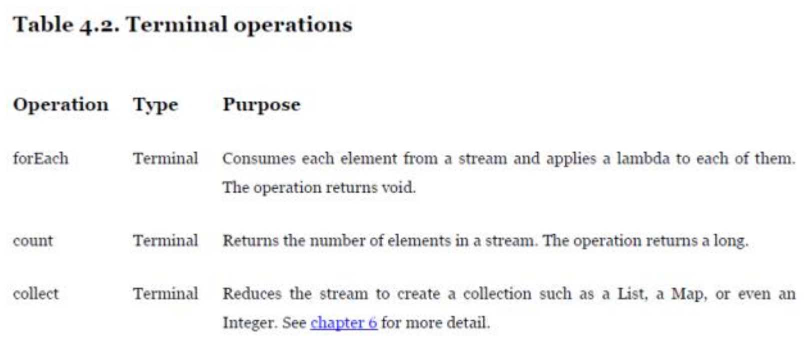


**Table 4.2. Terminal operations**

| Operation | Type | Purpose |
|---|---|---|
| forEach | Terminal | Consumes each element from a stream and applies a lambda to each of them. The operation returns void. |
| count | Terminal | Returns the number of elements in a stream. The operation returns a long. |
| collect | Terminal | Reduces the stream to create a collection such as a List, a Map, or even an Integer. See chapter 6 for more detail. |

## Optional in a nutshell

The Optional<T> class (java.util.Optional) is a container class to represent the existence or absence of a value. In the previous code, it's possible that findAny doesn't find any element. Instead of returning null, which is well known for being error prone, the Java 8 library designers introduced Optional<T>. We won't go into the details of Optional here, because we show in detail in chapter 10 how your code can benefit from using Optional to avoid bugs related to null checking. But for now, it's good to know that there are a few methods available in Optional that force you to explicitly check for the presence of a value or deal with the absence of a value:

- isPresent() returns true if Optional contains a value, false otherwise.
- ifPresent(Consumer<T> block) executes the given block if a value is present. We introduced the Consumer functional interface in chapter 3; it lets you pass a lambda that takes an argument of type T and returns void.
- T get() returns the value if present; otherwise it throws a NoSuchElement-Exception.
- T orElse(T other) returns the value if present; otherwise it returns a default value.

For example, in the previous code you'd need to explicitly check for the presence of a dish in the Optional object to access its name:

```
menu.stream()
    .filter(Dish::isVegetarian)
    .findAny()
    .ifPresent(d -> System.out.println(d.getName());
```

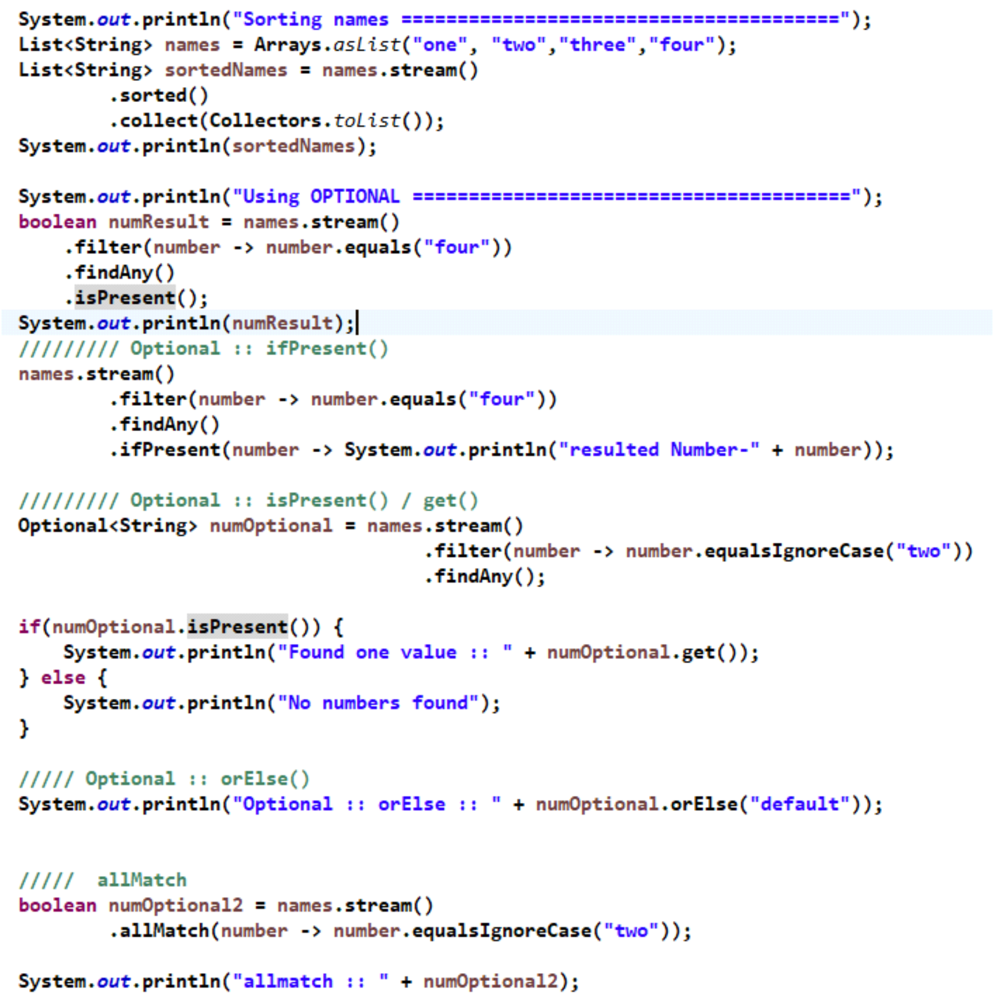
**Returns an Optional<Dish>.**

**If a value is contained, it's printed; otherwise nothing happens.**

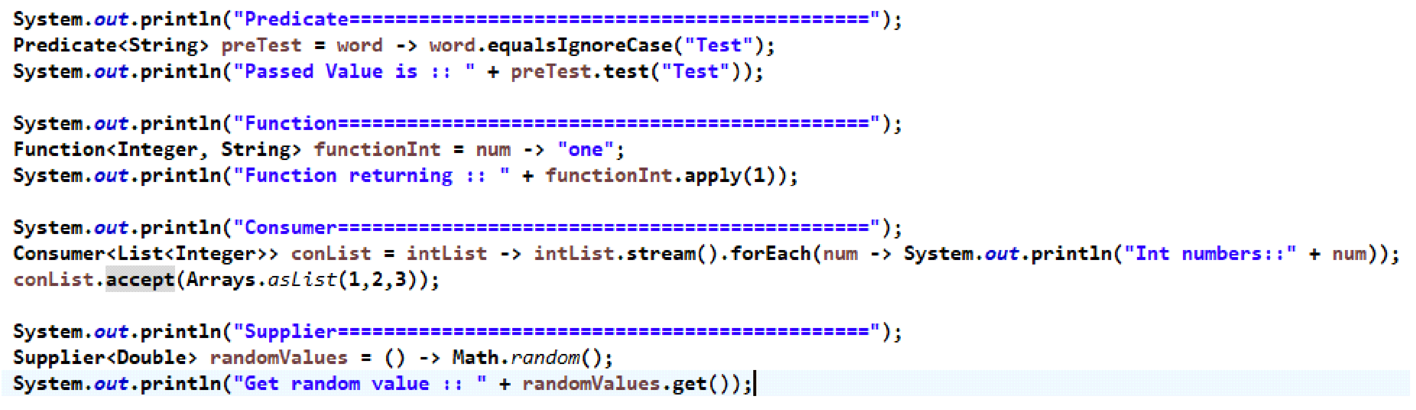
### Table 5.1. Intermediate and terminal operations

| Operation | Type | Return type | Type/functional interface used | Function descriptor |
|---|---|---|---|---|
| filter | Intermediate | Stream<T> | Predicate<T> | T -> boolean |
| distinct | Intermediate (stateful-unbounded) | Stream<T> | | |
| skip | Intermediate (stateful-bounded) | Stream<T> | long | |
| limit | Intermediate (stateful-bounded) | Stream<T> | long | |
| map | Intermediate | Stream<R> | Function<T, R> | T -> R |
| flatMap | Intermediate | Stream<R> | Function<T, Stream<R>> | T -> Stream<R> |
| sorted | Intermediate (stateful-unbounded) | Stream<T> | Comparator<T> | (T, T) -> int |
| anyMatch | Terminal | boolean | Predicate<T> | T -> boolean |
| noneMatch | Terminal | boolean | Predicate<T> | T -> boolean |
| allMatch | Terminal | boolean | Predicate<T> | T -> boolean |
| findAny | Terminal | Optional<T> | | |
| findFirst | Terminal | Optional<T> | | |

| | | | | |
|---|---|---|---|---|
| forEach | Terminal | void | Consumer<T> | T -> void |
| collect | terminal | R | Collector<T, A, R> | |
| reduce | Terminal (stateful-bounded) | Optional<T> | BinaryOperator<T> | (T, T) -> T |
| count | Terminal | long | | |



```
System.out.println("Sorting names ======================================");
List<String> names = Arrays.asList("one", "two","three","four");
List<String> sortedNames = names.stream()
        .sorted()
        .collect(Collectors.toList());
System.out.println(sortedNames);

System.out.println("Using OPTIONAL ======================================");
boolean numResult = names.stream()
    .filter(number -> number.equals("four"))
    .findAny()
    .isPresent();
System.out.println(numResult);
///////// Optional :: ifPresent()
names.stream()
        .filter(number -> number.equals("four"))
        .findAny()
        .ifPresent(number -> System.out.println("resulted Number-" + number));

///////// Optional :: isPresent() / get()
Optional<String> numOptional = names.stream()
                                    .filter(number -> number.equalsIgnoreCase("two"))
                                    .findAny();

if(numOptional.isPresent()) {
    System.out.println("Found one value :: " + numOptional.get());
} else {
    System.out.println("No numbers found");
}

///// Optional :: orElse()
System.out.println("Optional :: orElse :: " + numOptional.orElse("default"));


/////  allMatch
boolean numOptional2 = names.stream()
        .allMatch(number -> number.equalsIgnoreCase("two"));

System.out.println("allmatch :: " + numOptional2);
```



```
System.out.println("Predicate=============================================");
Predicate<String> preTest = word -> word.equalsIgnoreCase("Test");
System.out.println("Passed Value is :: " + preTest.test("Test"));

System.out.println("Function==============================================");
Function<Integer, String> functionInt = num -> "one";
System.out.println("Function returning :: " + functionInt.apply(1));

System.out.println("Consumer==============================================");
Consumer<List<Integer>> conList = intList -> intList.stream().forEach(num -> System.out.println("Int numbers::" + num));
conList.accept(Arrays.asList(1,2,3));

System.out.println("Supplier==============================================");
Supplier<Double> randomValues = () -> Math.random();
System.out.println("Get random value :: " + randomValues.get());
```

Java 9

Publish-Subscribe Framework