Chapter 30 Aggregate Operations for Collection Streams

Motivations

```
Double[] numbers = {2.4, 55.6, 90.12, 26.6};
Set<Double> set = new HashSet<>(Arrays.asList(numbers));
int count = 0;
for (double e: set)
  if (e > 60)  count++;
System.out.println("Count is " + count);
```

The code is fine. However, Java provides a better and simpler way for accomplishing the task. Using the aggregate operations, you can rewrite the code as follows:

```
System.out.println("Count is "  +
  set.stream().filter(e -> e > 60).count());
```

Objectives

- To use aggregate operations on collection streams to simplify coding and improve performance (§ 30.1).
- To create a stream pipeline, apply lazy intermediate methods (**skip**, **limit**, **filter**, **distinct**, **sorted**, **map**, and **mapToInt**), and terminal methods (**count**, **sum**, **average**, **max**, **min**, **forEach**, **findFirst**, **firstAny**, **anyMatch**, **allMatch**, **noneMatch**, and **toArray**) on a steam (§ 30.2).
- To process primitive data values using the **IntStream**, **LongStream**, and **DoubleStream** (§ 30.3).
- To create parallel streams for fast execution (§ 30.4).
- To reduce the elements in a stream into a single result using the **reduce** method (§ 30.5).
- To place the elements in a stream into a mutable collection using the **collect** method (§ 30.6).
- To group the elements in a stream and apply aggreate methods for the elements in the groups (§ 30.7).
- To use a variety of examples to demonstrate how to simplify coding using streams (§ 30.8).

Stream

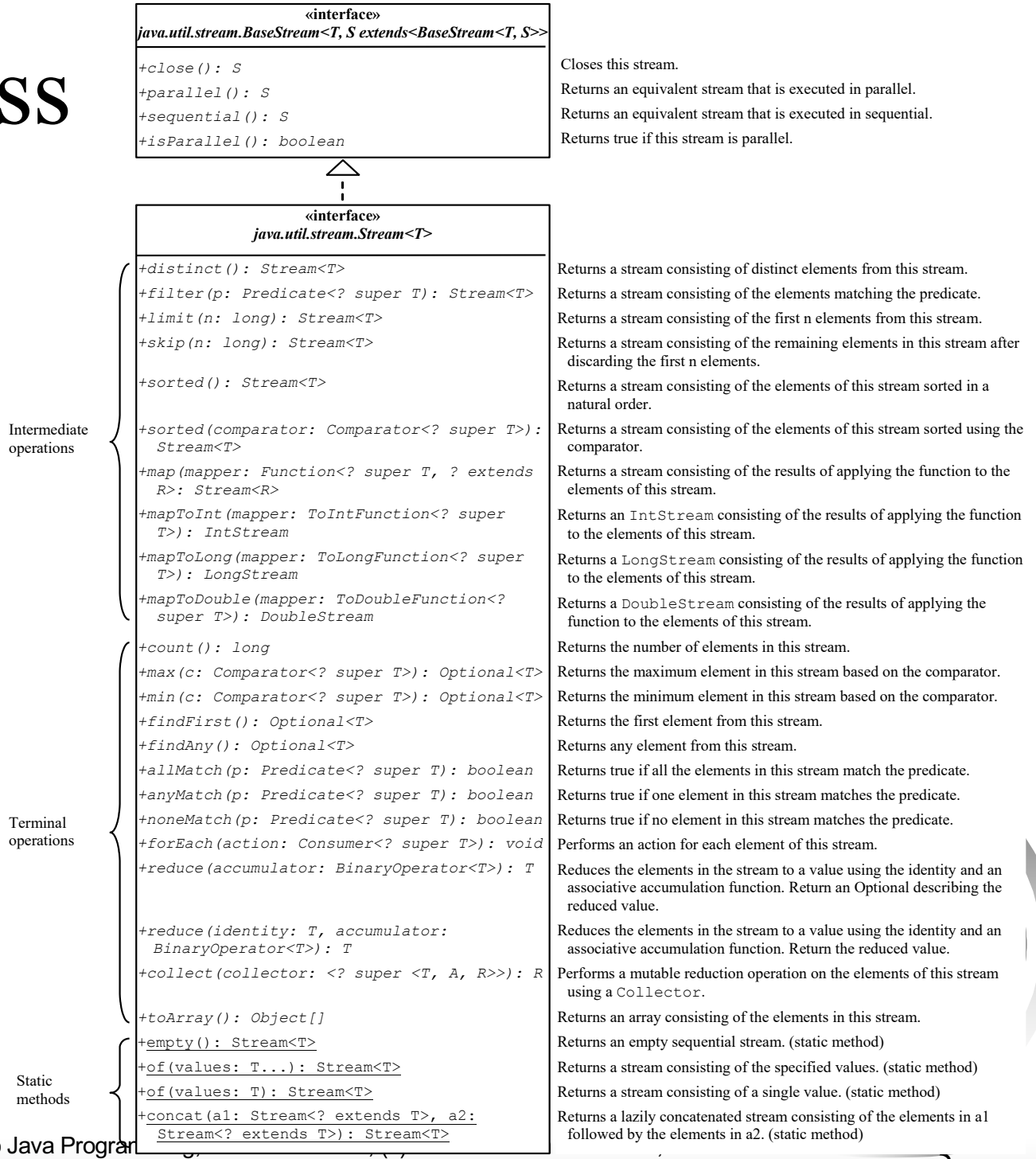
A *collection stream* or simply *stream* is a sequence of elements. The **filter** and **count** are the operations that you can apply on a stream. These operations are known as *aggregate operations*, because they are applied to a collection of data.

Stream Class

An *intermediate method* transfroms a stream into another stream.

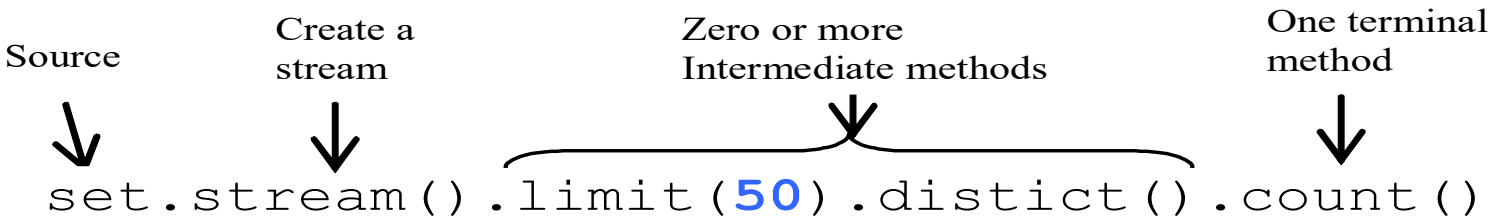
An *terminal method* performs an action and terminates a stream.

A *static method* creates a stream.



Stream Pipeline

A stream pipeline consists of a stream created from a data source, zero or more intermediate methods, and a final terminal method.



Streams are lazy, which means that the computation is performed only when the terminal operation is initiated. This allows the JVM to optimize computation.

StreamDemo Run

Functional Interface Arguments

Most of the arguments for stream methods are instances of functional interfaces. So the arguments can be created using lambda expressions or method references.

forEach Method

```
forEach(e -> System.out.print(e + " "))
```

(a) Using a lambda expression

```
forEach(
  new java.util.function.Consumer<String>() {
    public void accept(String e) {
      System.out.print(e + " ");
    }
  }
)
```

(b) Using an anonymous inner class

The lambda expression not only simplifies the code, but also the concept of the method. You can now simply say that for each element in the stream perform the action as specified in the expression.

The sorted Method

sorted() is to sort the elements in their natural order and sorted(Comparator) sorts using the specified comparator.

```
sorted((e1, e2) ->
  e1.compareToIgnoreCase(e2))
```

(a) Using a lambda expression

```
sorted(String::compareToIgnoreCase)
```

(c) Using a method reference

```
sorted(
  new java.util.Comparator<String>() {
    public int compare(String e1, String e2) {
      return e1.compareToIgnoreCase(e2);
    }
  }
)
```

(b) Using an anonymous inner class

The filter Method

The **filter** method takes an argument of the **Predicate<? super T>** type, which is a functional interface with an abstract method **test(T t)** that returns a Boolean value. The method selects the elements from the stream that satisfies the predicate.

```
filter(e -> e.length() > 4)
```

(a) Using a lambda expression

```
filter(
  new java.util.function.Predicate<String>() {
    public boolean test(String e) {
      return e.length() > 4;
    }
  }
)
```

(b) Using an anonymous inner class

The max and min Methods

The **max** and **min** methods take an argument of the **Comparator<? Super T>** type. This argument specifies how the elements are compared in order to obtain the maximum and minimum element.

The anyMatch, allMatch, and noneMatch Methods

The **anyMatch**, **allMatch**, and **noneMatch** methods take an argument of the **Predicate<? super T>** type to test if the stream contains an element, all elements, or no element that satisfies the predicate.

The map Method

The **map** method returns a new stream by mapping each element in the stream into a new element. The **map** method takes an argument of the **Function<? super T, ? super R>** type to return an instance of the **Stream<R>**. The **Function** is a functional interface with an abstract method **apply(T t)** that maps **t** into a value of the type **R**.

```
map(e -> e.toUpperCase())
```

(a) Using a lambda expression

```
map(String::toUpperCase)
```

(c) Using a method reference

```
map(
  new java.util.function.Function<String, String>() {
    public String apply(String e) {
      return e.toUpperCase();
    }
  }
)
```

(b) Using an anonymous inner class

IntStream, **LongStream**, and **DoubleStream**

Stream represents a sequence of objects. In addition to **Stream**, Java provides **IntStream**, **LongStream**, and **DoubleStream** for representing a sequence of **int**, **long**, and **double** values. These streams are also subinterfaces of **BaseStream**. You can use these streams in the same way like a **Stream**. Additionally, you can use the **sum()**, **average()**, and **summaryStatistics()**methods for returning the sum, average, various statistics of the elements in the stream. You can use the **mapToInt** method to convert a **Stream** to an **IntStream** and use the **map** method to convert any stream including an **IntStream** to a **Stream**.

IntStream, LongStream, and DoubleStream Examples

Parallel Streams

Streams can be executed in parallel mode to improve performance. The **stream()** method in the **Collection** interface returns a sequential stream. To execute operations in parallel, use the **parallelStream()** method in the **Collection** interface to obtain a parallel stream. Any stream can be turned to into a parallel stream by invoking the **parallel()** method defined in tbe **BaseStream** interface. Likewise, you can turn a parallel stream into a sequential stream by invoking the **sequential()** method.

Parallel Streams

Streams can be executed in parallel mode to improve performance. The **stream()** method in the **Collection** interface returns a sequential stream. To execute operations in parallel, use the **parallelStream()** method in the **Collection** interface to obtain a parallel stream. Any stream can be turned to into a parallel stream by invoking the **parallel()** method defined in tbe **BaseStream** interface. Likewise, you can turn a parallel stream into a sequential stream by invoking the **sequential()** method.

Parallel Streams Example

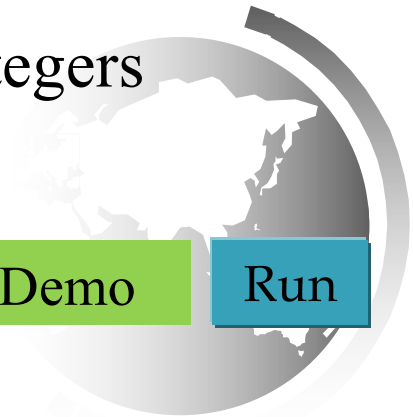
Stream Reduction Using the reduce Method

```
int total = 0;
for (int e: s) {
  total += e;
}
```

```
int sum = s.parallelStream()
   .reduce(0, (e1, e2) -> e1 + e2);
```

The **reduce** method makes the code concise. Moreover, the code is parallelizable, because multiple processors can simultaneously invoke the **applyAsInt** method on two integers repeatedly.

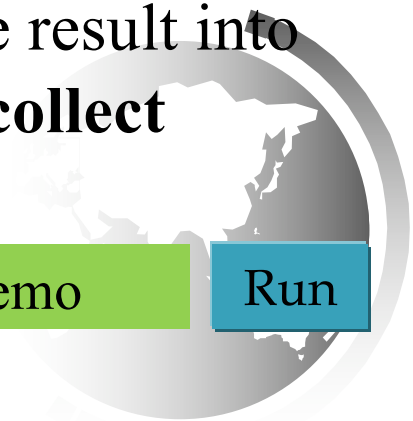
StreamReductionDemo Run



Stream Reduction Using the collect Method

You can use the **collect** method to reduce the elements in a stream into a mutable container.

In the preceding example, the **String**'s **concat** method is used in the **reduce** method for **Stream.of(names).reduce((x, y) -> x + y)**. This operation causes a new string to be created when concatenating two strings, which is very inefficient. A better approach is to use a **StringBuilder** and accumulate the result into a **StringBuilder**. This can be accomplished using the **collect** method.



Grouping Elements Using the groupingby **Collector**

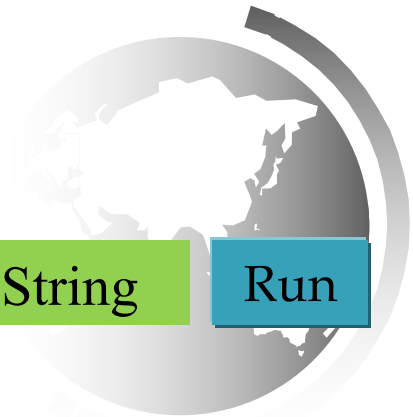
You can use the **groupingBy** collector along with the **collect** method to collect the elements by groups.

The elements in a stream can be divided into groups using the **groupingby** collector and then apply aggregate collectors on each group.

Case Studies: Analyzing Numbers

Case Studies: Counting the Occurrences of Each Letter

Case Studies: Counting the Occurrences of Each Letter in a String



CountOccurrenceOfLettersInAString

Run

Case Studies: Processing All Elements in a Two-Dimensional Array

TwoDimensionalArrayStream

Run

Case Studies: Finding the Directory Size

Case Studies: Counting Keywords

Case Studies: Occurrences of Words