

# 17.10 Stream<Integer> Manipulations

*[This section requires the interface concepts introduced in Sections 10.9–10.10.]*

So far, we've processed `IntStreams`. A `Stream` performs tasks on reference-type objects. `IntStream` is simply an `int`-optimized `Stream` that provides methods for common `int` operations. [Figure 17.11](#) performs *filtering* and *sorting* on a `Stream<Integer>`, using techniques similar to those in prior examples, and shows how to place a stream pipeline's results into a new collection for subsequent processing. We'll work with `Streams` of other reference types in subsequent examples.

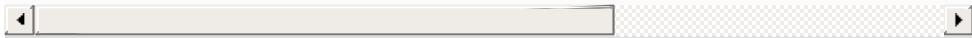
---

```
1 // Fig. 17.11: ArraysAndStreams.java
2 // Demonstrating lambdas and streams with an arr
3 import java.util.Arrays;
4 import java.util.List;
5 import java.util.stream.Collectors;
6
7 public class ArraysAndStreams {
8     public static void main(String[] args) {
9         Integer[] values = {2, 9, 5, 0, 3, 7, 1, 4
10
11         // display original values
12         System.out.printf("Original values: %s%n",
13 }
```

```

14      // sort values in ascending order with str
15      System.out.printf("Sorted values: %s%n",
16          Arrays.stream(values)
17              .sorted()
18          .collect(Collectors.toList()));
19
20      // values greater than 4
21      List<Integer> greaterThan4 =
22          Arrays.stream(values)
23              .filter(value -> value > 4)
24          .collect(Collectors.toList());
25      System.out.printf("Values greater than 4:
26
27      // filter values greater than 4 then sort
28      System.out.printf("Sorted values greater t
29          Arrays.stream(values)
30              .filter(value -> value > 4)
31                  .sorted()
32          .collect(Collectors.toList());
33
34      // greaterThan4 List sorted with streams
35      System.out.printf(
36          "Values greater than 4 (ascending with
37          greaterThan4.stream()
38              .sorted()
39          .collect(Collectors.toList()));
40      }
41  }

```




---

Original values: [2, 9, 5, 0, 3, 7, 1, 4, 8, 6]  
 Sorted values: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
 Values greater than 4: [9, 5, 7, 8, 6]  
 Sorted values greater than 4: [5, 6, 7, 8, 9]  
 Values greater than 4 (ascending with streams): [5, 6]



## Fig. 17.11

Demonstrating lambdas and streams with an array of Integers.

Throughout this example, we use the `Integer` array `values` (line 9) that's initialized with `int` values—the compiler *boxes* each `int` into an `Integer` object. Line 12 displays the contents of `values` before we perform any stream processing. `Arrays` method `asList` creates a `List<Integer>` view of the `values` array. The generic interface `List` (discussed in more detail in [Chapter 16](#)) is implemented by collections like `ArrayList` ([Chapter 7](#)). Line 12 displays the `List<Integer>`'s default `String` representation, which consists of square brackets ([ and ]) containing a comma-separated list of elements—we use this `String` representation throughout the example. We walk through the remainder of the code in [Sections 17.10.1–17.10.5](#).

### 17.10.1 Creating a Stream<Integer>

Class `Arrays` `stream` method can be used to create a `Stream` from an array of objects—for example, line 16 produces a `Stream<Integer>`, because `stream`'s argument is an array of `Integers`. Interface `Stream` (package `java.util.stream`) is a generic interface for performing stream operations on any *reference* type. The types

of objects that are processed are determined by the Stream's source.

Class `Arrays` also provides overloaded versions of method `stream` for creating `Int-Streams`, `LongStreams` and `DoubleStreams` from `int`, `long` and `double` arrays or from ranges of elements in the arrays.

## 17.10.2 Sorting a Stream and Collecting the Results

The stream pipeline in lines 16–18

---

```
Arrays.stream(values)
    .sorted()
    .collect(Collectors.toList())
```

uses stream techniques to sort the `values` array and collect the results in a `List<Integer>`. First, line 16 creates a `Stream<Integer>` from `values`. Next, line 17 calls `Stream` method `sorted` to sort the elements—this results in an intermediate `Stream<Integer>` with the values in *ascending* order. ([Section 17.11.3](#) discusses how to sort in descending order.)

## Creating a New Collection

# Containing a Stream Pipeline's Results

When processing streams, you often create *new* collections containing the results so that you can perform operations on them later. To do so, you can use `Stream`'s terminal operation `collect` (Fig. 17.11, line 18). As the stream pipeline is processed, method `collect` performs a **mutable reduction operation** that creates a `List`, `Map` or `Set` and modifies it by placing the stream pipeline's results into the collection. You may also use the mutable reduction operation `toArray` to place the results in a new array of the `Stream`'s element type.

The version of method `collect` in line 18 receives as its argument an object that implements interface `Collector` (package `java.util.stream`), which specifies how to perform the mutable reduction. Class `Collectors` (package `java.util.stream`) provides `static` methods that return predefined `Collector` implementations.

`Collectors` method `toList` (line 18) returns a `Collector` that places the `Stream<Integer>`'s elements into a `List<Integer>` collection. In lines 15–18, the resulting `List<Integer>` is displayed with an *implicit* call to its `toString` method.

A mutable reduction optionally performs a final data transformation. For example, in Fig. 17.8, we called `IntStream` method `collect` with the object returned by `Collectors` method `joining`. Behind the scenes, this

Collector used a `StringJoiner` (package `java.util`) to concatenate the stream elements' `String` representations, then called the `StringJoiner`'s `toString` method to transform the result into a `String`. We show additional Collectors in [Section 17.12](#). For more predefined Collectors, visit:

---

<https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.html>

### 17.10.3 Filtering a Stream and Storing the Results for Later Use

The stream pipeline in lines 21–24 of [Fig. 17.11](#)

---

```
List<Integer> greaterThan4 =  
    Arrays.stream(values)  
        .filter(value -> value > 4)  
        .collect(Collectors.toList());
```

creates a `Stream<Integer>`, filters the stream to locate all the values greater than 4 and collects the results into a `List<Integer>`. `Stream` method `filter`'s lambda argument implements the functional interface `Predicate` (package `java.util.function`), which represents a one-parameter method that returns a `boolean`

indicating whether the parameter value satisfies the predicate.

We assign the stream pipeline's resulting `List<Integer>` to variable `greaterThan4`, which is used in line 25 to display the values greater than 4 and used again in lines 37–39 to perform additional operations on only the values greater than 4.

## 17.10.4 Filtering and Sorting a Stream and Collecting the Results

The stream pipeline in lines 29–32

---

```
Arrays.stream(values)
    .filter(value -> value > 4)
    .sorted()
    .collect(Collectors.toList())
```



displays the values greater than 4 in sorted order. First, line 29 creates a `Stream<Integer>`. Then line 30 filters the elements to locate all the values greater than 4. Next, line 31 indicates that we'd like the results sorted. Finally, line 32 collects the results into a `List<Integer>`, which is then displayed as a `String`.



## Performance Tip 17.2

*Call `filter` before `sorted` so that the stream pipeline sorts only the elements that will be in the stream pipeline's result.*

### 17.10.5 Sorting Previously Collected Results

The stream pipeline in lines 37–39

---

```
greaterThan4.stream()
    .sorted()
    .collect(Collectors.toList());
```

uses the `greaterThan4` collection created in lines 21–24 to show additional processing on the results of a prior stream pipeline. `List` method `stream` creates the stream. Then we sort the elements and `collect` the results into a new `List<Integer>` and display its `String` representation.