17.11 `Stream<String>` Manipulations

[This section demonstrates how lambdas and streams can be used to simplify programming tasks that you learned in Chapter 14, Strings, Characters and Regular Expressions.]

So far, we've manipulated only streams of `int` values and `Integer` objects. Figure 17.12 performs similar stream operations on a `Stream<String>`. In addition, we demonstrate *case-insensitive sorting* and sorting in *descending* order. Throughout this example, we use the `String` array `strings` (lines 9–10) that's initialized with color names—some with an initial uppercase letter. Line 13 displays the contents of `strings` *before* we perform any stream processing. We walk through the rest of the code in Sections 17.11.1–17.11.3.

```
   1   // Fig. 17.12: ArraysAndStreams2.java
 2   // Demonstrating lambdas and streams with an arr
       3   import java.util.Arrays;
      4   import java.util.Comparator;
     5   import java.util.stream.Collectors;
                    6
       7   public class ArraysAndStreams2 {
  8      public static void main(String[] args) {
         9         String[] strings =
10          {"Red", "orange", "Yellow", "green", "B
                    11
```

```
        // display original strings
        System.out.printf("Original strings: %s%n"

        // strings in uppercase
        System.out.printf("strings in uppercase: %
           Arrays.stream(strings)
             .map(String::toUpperCase)
           .collect(Collectors.toList()));

        // strings less than "n" (case insensitive
        System.out.printf("strings less than n sor
           Arrays.stream(strings)
           .filter(s -> s.compareToIgnoreCas
           .sorted(String.CASE_INSENSITIVE_O
           .collect(Collectors.toList()));

        // strings less than "n" (case insensitive
        System.out.printf("strings less than n sor
           Arrays.stream(strings)
           .filter(s -> s.compareToIgnoreCas
           .sorted(String.CASE_INSENSITIVE_O
           .collect(Collectors.toList()));
        }
     }
```

```
Original strings: [Red, orange, Yellow, green, Blue,
strings in uppercase: [RED, ORANGE, YELLOW, GREEN, BL
strings less than n sorted ascending: [Blue, green, i
strings less than n sorted descending: [indigo, green
```

Fig. 17.12

Demonstrating lambdas and streams with an array of

`String`s.

17.11.1 Mapping `String`s to Uppercase

The stream pipeline in lines 17–19

```
Arrays.stream(strings)
      .map(String::toUpperCase)
      .collect(Collectors.toList()));
```

displays the `String`s in uppercase letters. To do so, line 17 creates a `Stream<String>` from the array `strings`, then line 18 `map`s each `String` to its uppercase version by calling `String` instance method `toUpperCase` on each stream element.

`Stream` method `map` receives an object that implements the `Function` functional interface, representing a one-parameter method that performs a task with its parameter then returns the result. In this case, we pass to `map` an **unbound instance method reference** of the form *ClassName*`::`*instanceMethodName* (`String::toUpperCase`). "Unbound" means that the method reference does not indicate the specific object on which the method will be called—the compiler converts this to a one-parameter lambda that invokes the instance method on the lambda's parameter, which must have type *ClassName*. In

this case, the compiler converts `String::toUpperCase` into a lambda like

```
s -> s.toUpperCase()
```

which returns the uppercase version of the lambda's argument. Line 19 `collect`s the results into a `List<String>` that we output as a `String`.

17.11.2 Filtering `Strings` Then Sorting Them in Case-Insensitive Ascending Order

The stream pipeline in lines 23–26

```
Arrays.stream(strings)
      .filter(s -> s.compareToIgnoreCase("n") < 0)
      .sorted(String.CASE_INSENSITIVE_ORDER)
      .collect(Collectors.toList())
```

filters and sort the `String`s. Line 23 creates a `Stream<String>` from the array `strings`, then line 24 calls `Stream` method `filter` to locate all the `String`s that are less than `"n"`, using a *case-insensitive* comparison in the `Predicate` lambda. Line 25 sorts the results and line 26 collects them into a `List<String>` that we output as a

`String`.

In this case, line 25 invokes the version of `Stream` method `sorted` that receives a `Comparator` as an argument. A `Comparator` defines a `compare` method that returns a negative value if the first value being compared is less than the second, 0 if they're equal and a positive value if the first value is greater than the second. By default, method `sorted` uses the *natural order* for the type—for `String`s, the natural order is case sensitive, which means that `"Z"` is less than `"a"`. Passing the predefined `Comparator` `String.CASE_INSENSITIVE_ORDER` performs a *case-insensitive* sort.

17.11.3 Filtering `Strings` Then Sorting Them in Case-Insensitive Descending Order

The stream pipeline in lines 30–33

```
   Arrays.stream(strings)
         .filter(s -> s.compareToIgnoreCase("n") < 0)
         .sorted(String.CASE_INSENSITIVE_ORDER.reversed(
         .collect(Collectors.toList()));
```

performs the same tasks as lines 23–26, but sorts the `String`s

in *descending* order. Functional interface `Comparator` contains `default` method `reversed`, which reverses an existing `Comparator`'s ordering. When you apply `reversed` to `String.CASE_INSENSITIVE_ORDER`, `sorted` performs a case-insensitive sort and places the `String`s in *descending* order