

17.6 Method References

[This section demonstrates how streams can be used to simplify programming tasks that you learned in [Chapter 6, Methods: A Deeper Look](#).]

For a lambda that simply calls another method, you can replace the lambda with that method's name—known as a **method reference**. The compiler converts a method reference into an appropriate lambda expression.

Like [Fig. 6.6](#), [Fig. 17.8](#) uses `SecureRandom` to obtain random numbers in the range 1–6. The program uses streams to create the random values and method references to help display the results. We walk through the code in [Sections 17.6.1–17.6.4](#).

```
 1 // Fig. 17.8: RandomIntegers.java
 2 // Shifted and scaled random integers.
 3 import java.security.SecureRandom;
 4 import java.util.stream.Collectors;
 5
 6 public class RandomIntegers {
 7     public static void main(String[] args) {
 8         SecureRandom randomNumbers = new SecureRan
 9
10         // display 10 random integers on separate
11         System.out.println("Random numbers on sepa
12             randomNumbers.ints(10, 1, 7)
13                 .forEach(System.out::println)
```

```
        14  
15      // display 10 random integers on the same  
16          String numbers =  
17          randomNumbers.ints(10, 1, 7)  
18              .mapToObj(String::valueOf)  
19              .collect(Collectors.joining(""  
20          System.out.printf("%nRandom numbers on one  
21  
22      }  
23  }
```



Random numbers on separate lines:

```
4  
3  
4  
5  
1  
5  
5  
3  
6  
5
```

Random numbers on one line: 4 6 2 5 6 4 3 2 4 1

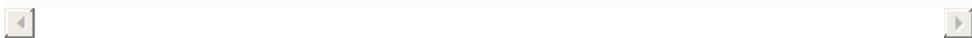


Fig. 17.8

Shifted and scaled random integers.

17.6.1 Creating an

IntStream of Random Values

Class `SecureRandom`'s `ints` method returns an `IntStream` of random numbers. In the stream pipeline of lines 12–13

```
randomNumbers.ints(10, 1, 7)
```



creates an `IntStream` data source with the specified number of random `int` values (10) in the range starting with the first argument (1) up to, but not including, the second argument (7). So, line 12 produces an `IntStream` of 10 random integers in the range 1–6.

17.6.2 Performing a Task on Each Stream Element with `forEach` and a Method Reference

Next, line 13 of the stream pipeline uses `IntStream` method `forEach` (a terminal operation) to perform a task on each stream element. Method `forEach` receives as its argument a method that takes one parameter and performs a task using the parameter's value.

The argument to `forEach`

```
System.out::println
```



in this case is a method reference—a shorthand notation for a lambda that calls the specified method. A method reference of the form

```
objectName::instanceMethodName
```



is a **bound instance method reference**—“bound” means the *specific* object to the left of the `::` (`System.out`) *must* be used to call the instance method to the right of the `::` (`println`). The compiler converts `System.out::println` into a one-parameter lambda like

```
x -> System.out.println(x)
```



that passes the lambda’s argument—the current stream element (represented by `x`)—to the `System.out` object’s `println` instance method, which implicitly outputs the `String` representation of the argument. The stream pipeline of lines 12–13 is equivalent to the following `for` loop:

```
for (int i = 1; i <= 10; i++) {  
    System.out.println(1 + randomNumbers.nextInt(6));  
}
```

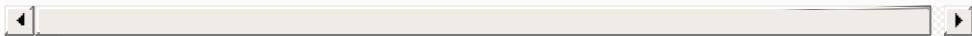




17.6.3 Mapping Integers to String Objects with mapToObj

The stream pipeline in lines 16–19

```
String numbers =
    randomNumbers.ints(10, 1, 7)
        .mapToObj(String::valueOf)
        .collect(Collectors.joining(" "));
```



creates a `String` containing 10 random integers in the range 1–6 separated by spaces. The pipeline performs three chained method calls:

- Line 17 creates the data source—an `IntStream` of 10 random integers from 1–6.
- Line 18 maps each `int` to its `String` representation, resulting in an intermediate stream of `Strings`. The `IntStream` method `map` that we've used previously returns another `IntStream`. To map to `Strings`, we use instead the `IntStream` method `mapToObj`, which enables you to map from `ints` to a stream of reference-type elements. Like `map`, `mapToObj` expects a one-parameter method that returns a result. In this example, `mapToObj`'s argument is a **static method reference** of the form `ClassName::staticMethodName`. The compiler converts `String::valueOf` (which returns its argument's `String` representation) into a one-parameter lambda that calls `valueOf`, passing the current stream element as an argument, as in

```
x -> String.valueOf(x)
```

- Line 19, which we discuss in more detail in [Section 17.6.4](#), uses the `Stream` terminal operation `collect` to concatenate all the `String`s, separating each from the next with a space. Method `collect` is a form of reduction because it returns one object—in this case, a `String`.

Line 20 then displays the resulting `String`.

17.6.4 Concatenating Strings with `collect`

Consider line 19 of [Fig. 17.8](#). The `Stream` terminal operation `collect` uses a *collector* to gather the stream's elements into a single object—often a collection. This is similar to a reduction, but `collect` returns an object containing the stream's elements, whereas `reduce` returns a single value of the stream's element type. In this example, we use a predefined collector returned by the `static Collectors` method `joining`. This collector creates a concatenated `String` representation of the stream's elements, appending each element to the `String` separated from the previous element by the `joining` method's argument (in this case, a space). Method `collect` then returns the resulting `String`. We discuss other collectors throughout this chapter.