Chapter 17
Java SE 8 Lambdas and Streams

Java How to Program, 10/e

OBJECTIVES

In this chapter you'll:

- Learn what functional programming is and how it complements object-oriented programming.
- Use functional programming to simplify programming tasks you've performed with other techniques.
- Write lambda expressions that implement functional interfaces.
- Learn what streams are and how stream pipelines are formed from stream sources, intermediate operations and terminal operations.
- Perform operations on `IntStream`s, including `forEach`, `count`, `min`, `max`, `sum`, `average`, `reduce`, `filter` and `sorted`.
- Perform operations on `Stream`s, including `filter`, `map`, `sorted`, `collect`, `forEach`, `findFirst`, `distinct`, `mapToDouble` and `reduce`.
- Create streams representing ranges of `int` values and random `int` values.

17.1 Introduction

- Prior to Java SE 8, Java supported three programming paradigms—procedural programming, object-oriented programming and generic programming. Java SE 8 adds functional programming.
- The new language and library capabilities that support functional programming were added to Java as part of Project Lambda.
- This chapter presents many examples of functional programming, often showing simpler ways to implement tasks that you programmed in earlier chapters (Fig. 17.1).

Pre-Java-SE-8 topics	Corresponding Java SE 8 discussions and examples
Chapter 7, Arrays and ArrayLists	Sections 17.3–17.4 introduce basic lambda and streams capabilities that process one-dimensional arrays.
Chapter 10, Object-Oriented Programming: Polymorphism and Interfaces	Section 10.10 introduced the new Java SE 8 interface features (`default` methods, `static` methods and the concept of functional interfaces) that support functional programming.
Chapter 12, GUI Components: Part 1	Section 17.9 shows how to use a lambda to implement a Swing event-listener functional interface.
Chapter 14, Strings, Characters and Regular Expressions	Section 17.5 shows how to use lambdas and streams to process collections of `String` objects.
Chapter 15, Files, Streams and Object Serialization	Section 17.7 shows how to use lambdas and streams to process lines of text from a file.

Fig. 17.1 | Java SE 8 lambdas and streams discussions and examples.

Pre-Java-SE-8 topics	Corresponding Java SE 8 discussions and examples
Chapter 22, GUI Components: Part 2	Discusses using lambdas to implement Swing event-listener functional interfaces.
Chapter 23, Concurrency	Shows that functional programs are easier to parallelize so that they can take advantage of multi-core architectures to enhance performance. Demonstrates parallel stream processing. Shows that `Arrays` method `parallelSort` improves performance on multi-core architectures when sorting large arrays.
Chapter 25, JavaFX GUI: Part 1	Discusses using lambdas to implement JavaFX event-listener functional interfaces.

Fig. 17.1 | Java SE 8 lambdas and streams discussions and examples.

17.2 Functional Programming Technologies Overview

- Prior to functional programming, you typically determined what you wanted to accomplish, then specified the precise steps to accomplish that task.
- External iteration
  - Using a loop to iterate over a collection of elements.
  - Requires accessing the elements sequentially.
  - Requires mutable variables.

17.2 Functional Programming Technologies Overview (Cont.)

- Functional programming
  - Specify what you want to accomplish in a task, but not how to accomplish it
- Internal iteration
  - Let the library determine how to iterate over a collection of elements is known as.
  - Internal iteration is easier to parallelize.
- Functional programming focuses on immutability—not modifying the data source being processed or any other program state.

17.2.1 Functional Interfaces

- Functional interfaces are also known as single abstract method (SAM) interfaces.
- Package `java.util.function`
  - Six basic functional interfaces
  - Figure 17.2 shows the six basic generic functional interfaces.
- Many specialized versions of the basic functional interfaces
  - Use with `int`, `long` and `double` primitive values.
- Also generic customizations of `Consumer`, `Function` and `Predicate`
  - for binary operations—methods that take two arguments.

Interface	Description
`BinaryOperator<T>`	Contains method `apply` that takes two `T` arguments, performs an operation on them (such as a calculation) and returns a value of type `T`. You'll see several examples of `BinaryOperators` starting in Section 17.3.
`Consumer<T>`	Contains method `accept` that takes a `T` argument and returns `void`. Performs a task with it's `T` argument, such as outputting the object, invoking a method of the object, etc. You'll see several examples of `Consumers` starting in Section 17.3.
`Function<T,R>`	Contains method `apply` that takes a `T` argument and returns a value of type `R`. Calls a method on the `T` argument and returns that method's result. You'll see several examples of `Functions` starting in Section 17.5.
`Predicate<T>`	Contains method `test` that takes a `T` argument and returns a `boolean`. Tests whether the `T` argument satisfies a condition. You'll see several examples of `Predicates` starting in Section 17.3.

Fig. 17.2 | The six basic generic functional interfaces in package `java.util.function`.

Interface	Description
`Supplier<T>`	Contains method `get` that takes no arguments and produces a value of type `T`. Often used to create a collection object in which a stream operation's results are placed. You'll see several examples of `Suppliers` starting in Section 17.7.
`UnaryOperator<T>`	Contains method `get` that takes no arguments and returns a value of type `T`. You'll see several examples of `UnaryOperators` starting in Section 17.3.

Fig. 17.2 | The six basic generic functional interfaces in package `java.util.function`.

17.2.2 Lambda Expressions

- Lambda expression
  - anonymous method
  - shorthand notation for implementing a functional interface.
- The type of a lambda is the type of the functional interface that the lambda implements.
- Can be used anywhere functional interfaces are expected.

17.2.2 Lambda Expressions (Cont.)

- A lambda consists of a parameter list followed by the arrow token and a body, as in:
  - (*parameterList*) -> {*statements*}
- For example, the following lambda receives two `int`s and returns their sum:
  - (`int` x, `int` y) -> {`return` x + y;}
- This lambda's body is a statement block that may contain one or more statements enclosed in curly braces.
- A lambda's parameter types may be omitted, as in:
  - (x, y) -> {`return` x + y;}
- in which case, the parameter and return types are determined by the lambda's context.

17.2.2 Lambda Expressions (Cont.)

- A lambda with a one-expression body can be written as:
  - (x, y) -> x + y
  - In this case, the expression's value is implicitly returned.
- When the parameter list contains only one parameter, the parentheses may be omitted, as in:
  - value -> System.out.printf("%d ", value)
- A lambda with an empty parameter list is defined with `()` to the left of the arrow token (`->`), as in:
  - () -> System.out.println("Welcome to lambdas!")
- There are also specialized shorthand forms of lambdas that are known as method references.

17.2.3 Streams

- Streams are objects that implement interface `Stream` (from the package `java.util.stream`
  - Enable you to perform functional programming tasks
- Specialized stream interfaces for processing `int`, `long` or `double` values
- Streams move elements through a sequence of processing steps—known as a stream pipeline
  - Pipeline begins with a data source, performs various intermediate operations on the data source's elements and ends with a terminal operation.
- A stream pipeline is formed by chaining method calls.

17.2.3 Streams (Cont.)

- Streams do not have their own storage
  - Once a stream is processed, it cannot be reused, because it does not maintain a copy of the original data source.
- An intermediate operation specifies tasks to perform on the stream’s elements and always results in a new stream.
- Intermediate operations are lazy—they aren’t performed until a terminal operation is invoked.
  - Allows library developers to optimize stream-processing performance.

17.2.3 Streams (Cont.)

- Terminal operation
  - initiates processing of a stream pipeline's intermediate operations
  - produces a result
  - Terminal operations are eager—they perform the requested operation when they are called.
- Figure 17.3 shows some common intermediate operations.
- Figure 17.4 shows some common terminal operations.

Intermediate `Stream` operations	
`filter`	Results in a stream containing only the elements that satisfy a condition.
`distinct`	Results in a stream containing only the unique elements.
`limit`	Results in a stream with the specified number of elements from the beginning of the original stream.
`map`	Results in a stream in which each element of the original stream is mapped to a new value (possibly of a different type)—e.g., mapping numeric values to the squares of the numeric values. The new stream has the same number of elements as the original stream.
`sorted`	Results in a stream in which the elements are in sorted order. The new stream has the same number of elements as the original stream.

Fig. 17.3 | Common intermediate `Stream` operations.

Terminal `Stream` operations	
`forEach`	Performs processing on every element in a stream (e.g., display each element).
Reduction operations—***Take all values in the stream and return a single value***	
`average`	Calculates the *average* of the elements in a numeric stream.
`count`	Returns the *number of elements* in the stream.
`max`	Locates the *largest* value in a numeric stream.
`min`	Locates the *smallest* value in a numeric stream.
`reduce`	Reduces the elements of a collection to a *single value* using an associative accumulation function (e.g., a lambda that adds two elements).
Mutable reduction operations—***Create a container (such as a collection or `StringBuilder`)***	
`collect`	Creates a *new collection* of elements containing the results of the stream's prior operations.
`toArray`	Creates an *array* containing the results of the stream's prior operations.

Fig. 17.4 | Common terminal `Stream` operations.

Terminal `Stream` operations	
Search operations	
`findFirst`	Finds the *first* stream element based on the prior intermediate operations; immediately terminates processing of the stream pipeline once such an element is found.
`findAny`	Finds *any* stream element based on the prior intermediate operations; immediately terminates processing of the stream pipeline once such an element is found.
`anyMatch`	Determines whether *any* stream elements match a specified condition; immediately terminates processing of the stream pipeline if an element matches.
`allMatch`	Determines whether *all* of the elements in the stream match a specified condition.

Fig. 17.4 | Common terminal `Stream` operations.

17.3 `IntStream` Operations

- Figure 17.5 demonstrates operations on an `IntStream` (package `java.util.stream`)—a specialized stream for manipulating `int` values.
- The techniques shown in this example also apply to `LongStream`s and `DoubleStream`s for `long` and `double` values, respectively.

```java
// Fig. 17.5: IntStreamOperations.java
// Demonstrating IntStream operations.
import java.util.Arrays;
import java.util.stream.IntStream;

public class IntStreamOperations
{
   public static void main(String[] args)
   {
      int[] values = {3, 10, 6, 1, 4, 8, 2, 5, 9, 7};

      // display original values
      System.out.print("Original values: ");
      IntStream.of(values)
               .forEach(value -> System.out.printf("%d ", value));
      System.out.println();

```

Fig. 17.5 | Demonstrating `IntStream` operations. (Part 1 of 5.)

```
        // count, min, max, sum and average of the values
        System.out.printf("%nCount: %d%n", IntStream.of(values).count());
        System.out.printf("Min: %d%n",
           IntStream.of(values).min().getAsInt());
        System.out.printf("Max: %d%n",
           IntStream.of(values).max().getAsInt());
        System.out.printf("Sum: %d%n", IntStream.of(values).sum());
        System.out.printf("Average: %.2f%n",
           IntStream.of(values).average().getAsDouble());

        // sum of values with reduce method
        System.out.printf("%nSum via reduce method: %d%n",
           IntStream.of(values)
                    .reduce(0, (x, y) -> x + y));

        // sum of squares of values with reduce method
        System.out.printf("Sum of squares via reduce method: %d%n",
           IntStream.of(values)
                    .reduce(0, (x, y) -> x + y * y));
```

Fig. 17.5 | Demonstrating `IntStream` operations. (Part 2 of 5.)

```java

      // product of values with reduce method
      System.out.printf("Product via reduce method: %d%n",
         IntStream.of(values)
                  .reduce(1, (x, y) -> x * y));

      // even values displayed in sorted order
      System.out.printf("%nEven values displayed in sorted order: ");
      IntStream.of(values)
               .filter(value -> value % 2 == 0)
               .sorted()
               .forEach(value -> System.out.printf("%d ", value));
      System.out.println();

      // odd values multiplied by 10 and displayed in sorted order
      System.out.printf(
         "Odd values multiplied by 10 displayed in sorted order: ");
      IntStream.of(values)
               .filter(value -> value % 2 != 0)
               .map(value -> value * 10)
               .sorted()
               .forEach(value -> System.out.printf("%d ", value));
      System.out.println();

```

Fig. 17.5 | Demonstrating `IntStream` operations. (Part 3 of 5.)

```
        // sum range of integers from 1 to 10, exlusive
        System.out.printf("%nSum of integers from 1 to 9: %d%n",
           IntStream.range(1, 10).sum());

        // sum range of integers from 1 to 10, inclusive
        System.out.printf("Sum of integers from 1 to 10: %d%n",
           IntStream.rangeClosed(1, 10).sum());
     }
  } // end class IntStreamOperations
```

Fig. 17.5 | Demonstrating `IntStream` operations. (Part 4 of 5.)

```
Original values: 3 10 6 1 4 8 2 5 9 7

Count: 10
Min: 1
Max: 10
Sum: 55
Average: 5.50

Sum via reduce method: 55
Sum of squares via reduce method: 385
Product via reduce method: 3628800

Even values displayed in sorted order: 2 4 6 8 10
Odd values multiplied by 10 displayed in sorted order: 10 30 50 70 90

Sum of integers from 1 to 9: 45
Sum of integers from 1 to 10: 55
```

Fig. 17.5 | Demonstrating `IntStream` operations. (Part 5 of 5.)

17.3.1 Creating an `IntStream` and Displaying Its Values with the `forEach` Terminal Operation

- `IntStream` `static` method `of` receives an `int` array as an argument and returns an `IntStream` for processing the array's values.
- `IntStream` method `forEach` (a terminal-operation) receives as its argument an object that implements the `IntConsumer` functional interface (package `java.util.function`). This interface's `accept` method receives one `int` value and performs a task with it.

17.3.1 Creating an `IntStream` and Displaying Its Values with the `forEach` Terminal Operation (Cont.)

- Compiler can infer the types of a lambda's parameters and the type returned by a lambda from the context in which the lambda is used.
  - Determined by the lambda's target type—the functional interface type that's expected where the lambda appears in the code.
- Lambdas may use `final` local variables or effectively `final` local variables.
- A lambda that refers to a local variable in the enclosing lexical scope is known as a capturing lambda.

17.3.1 Creating an `IntStream` and Displaying Its Values with the `forEach` Terminal Operation (Cont.)

- A lambda can use the outer class's `this` reference without qualifying it with the outer class's name.
- The parameter names and variable names that you use in lambdas cannot be the same as any other local variables in the lambda's lexical scope; otherwise, a compilation error occurs.

17.3.2 Terminal Operations `count`, `min`, `max`, `sum` and `average`

- Class `IntStream` provides terminal operations for common stream reductions
  - `count` returns the number of elements
  - `min` returns the smallest `int`
  - `max` returns the largest `int`
  - `sum` returns the sum of all the `int`s
  - `average` returns an `OptionalDouble` (package `java.util`) containing the average of the `int`s as a value of type `double`
- Class `OptionalDouble`’s `getAsDouble` method returns the `double` in the object or throws a `NoSuchElementException`.
  - To prevent this exception, you can call method `orElse`, which returns the `OptionalDouble`’s value if there is one, or the value you pass to `orElse`, otherwise.

17.3.2 Terminal Operations count, min, max, sum and average (Cont.)

- IntStream method summaryStatistics performs the count, min, max, sum and average operations in one pass of an IntStream's elements and returns the results as an IntSummaryStatistics object (package java.util).

17.3.3 Terminal Operation reduce

- You can define your own reductions for an `IntStream` by calling its `reduce` method.
  - First argument is a value that helps you begin the reduction operation
  - Second argument is an object that implements the `IntBinaryOperator` functional interface
- Method `reduce`'s first argument is formally called an identity value—a value that, when combined with any stream element using the `IntBinaryOperator` produces that element's original value.

17.3.4 Intermediate Operations: Filtering and Sorting `IntStream` Values

- Filter elements to produce a stream of intermediate results that match a predicate.
- `IntStream` method `filter` receives an object that implements the `IntPredicate` functional interface (package `java.util.function`).
- `IntStream` method `sorted` (a lazy operation) orders the elements of the stream into ascending order (by default).
  - All prior intermediate operations in the stream pipeline must be complete so that method `sorted` knows which elements to sort.

17.3.4 Intermediate Operations: Filtering and Sorting `IntStream` Values (Cont.)

- Method `filter` a stateless intermediate operation—it does not require any information about other elements in the stream in order to test whether the current element satisfies the predicate.
- Method `sorted` is a stateful intermediate operation that requires information about all of the other elements in the stream in order to sort them.
- Interface `IntPredicate`’s `default` method `and` performs a logical AND operation with short-circuit evaluation between the `IntPredicate` on which it’s called and its `IntPredicate` argument.

17.3.4 Intermediate Operations: Filtering and Sorting `IntStream` Values (Cont.)

- Interface `IntPredicate`'s `default` method `negate` reverses the `boolean` value of the `IntPredicate` on which it's called.
- Interface `IntPredicate` `default` method `or` performs a logical OR operation with short-circuit evaluation between the `IntPredicate` on which it's called and its `IntPredicate` argument.
- You can use the interface `IntPredicate` `default` methods to compose more complex conditions.

17.3.5 Intermediate Operation: Mapping

- Mapping is an intermediate operation that transforms a stream's elements to new values and produces a stream containing the resulting (possibly different type) elements.
- `IntStream` method `map` (a stateless intermediate operation) receives an object that implements the `IntUnaryOperator` functional interface (package `java-.util.function`).

17.3.6 Creating Streams of `int`s with `IntStream` Methods `range` and `rangeClosed`

- `IntStream` methods `range` and `rangeClosed` each produce an ordered sequence of `int` values.
  - Both methods take two `int` arguments representing the range of values.
  - Method `range` produces a sequence of values from its first argument up to, but not including, its second argument.
  - Method `rangeClosed` produces a sequence of values including both of its arguments.

17.4 `Stream<Integer>` Manipulations

- Class `Array`'s `stream` method is used to create a `Stream` from an array of objects.

```java
// Fig. 17.6: ArraysAndStreams.java
// Demonstrating lambdas and streams with an array of Integers.
import java.util.Arrays;
import java.util.Comparator;
import java.util.List;
import java.util.stream.Collectors;

public class ArraysAndStreams
{
   public static void main(String[] args)
   {
      Integer[] values = {2, 9, 5, 0, 3, 7, 1, 4, 8, 6};

      // display original values
      System.out.printf("Original values: %s%n", Arrays.asList(values));

      // sort values in ascending order with streams
      System.out.printf("Sorted values: %s%n",
         Arrays.stream(values)
               .sorted()
               .collect(Collectors.toList()));

```

Fig. 17.6 | Demonstrating lambdas and streams with an array of `Integers`. (Part 1 of 3.)

```java
        // values greater than 4
        List<Integer> greaterThan4 =
           Arrays.stream(values)
                 .filter(value -> value > 4)
                 .collect(Collectors.toList());
        System.out.printf("Values greater than 4: %s%n", greaterThan4);

        // filter values greater than 4 then sort the results
        System.out.printf("Sorted values greater than 4: %s%n",
           Arrays.stream(values)
                 .filter(value -> value > 4)
                 .sorted()
                 .collect(Collectors.toList()));

        // greaterThan4 List sorted with streams
        System.out.printf(
           "Values greater than 4 (ascending with streams): %s%n",
           greaterThan4.stream()
                 .sorted()
                 .collect(Collectors.toList()));
     }
  } // end class ArraysAndStreams
```

Fig. 17.6 | Demonstrating lambdas and streams with an array of `Integers`. (Part 2 of 3.)

```
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, 7, 8, 9]
```

Fig. 17.6 | Demonstrating lambdas and streams with an array of `Integers`. (Part 3 of 3.)

17.4.1 Creating a Stream<Integer>

- Interface `Stream` (package `java.util.stream`) is a generic interface for performing stream operations on objects. The types of objects that are processed are determined by the `Stream`'s source.
- Class `Arrays` provides overloaded `stream` methods for creating `IntStream`s, `LongStream`s and `DoubleStream`s from `int`, `long` and `double` arrays or from ranges of elements in the arrays.

17.4.2 Sorting a Stream and Collecting the Results

- `Stream` method `sorted` sorts a stream's elements into ascending order by default.
- To create a collection containing a stream pipeline's results, you can use `Stream` method `collect` (a terminal operation).
  - As the stream pipeline is processed, method `collect` performs a mutable reduction operation that places the results into an object, such as a `List`, `Map` or `Set`.
- Method `collect` with one argument receives an object that implements interface `Collector` (package `java.util.stream`), which specifies how to perform the mutable reduction.

17.4.2 Sorting a Stream and Collecting the Results (Cont.)

- Class `Collectors` (package `java.util.stream`) provides `static` methods that return predefined `Collector` implementations.
- `Collectors` method `toList` transforms a `Stream<T>` into a `List<T>` collection.

17.4.3 Filtering a Stream and Storing the Results for Later Use

- `Stream` method `filter` receives a `Predicate` and results in a stream of objects that match the `Predicate`.
- `Predicate` method `test` returns a `boolean` indicating whether the argument satisfies a condition. Interface `Predicate` also has methods `and`, `negate` and `or`.

17.4.4 Sorting Previously Collected Results

- Once you place the results of a stream pipeline into a collection, you can create a new stream from the collection for performing additional stream operations on the prior results.

17.5 `Stream<String>` Manipulations

- Figure 17.7 performs some of the same stream operations you learned in – but on a `Stream<String>`.

```java
// Fig. 17.7: ArraysAndStreams2.java
// Demonstrating lambdas and streams with an array of Strings.
import java.util.Arrays;
import java.util.Comparator;
import java.util.stream.Collectors;

public class ArraysAndStreams2
{
   public static void main(String[] args)
   {
      String[] strings =
         {"Red", "orange", "Yellow", "green", "Blue", "indigo", "Violet"};

      // display original strings
      System.out.printf("Original strings: %s%n", Arrays.asList(strings));

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

```

Fig. 17.7 | Demonstrating lambdas and streams with an array of `Strings`. (Part 1 of 2.)

```
      // strings less than "n" (case insensitive) sorted ascending
      System.out.printf("strings greater than m sorted ascending: %s%n",
         Arrays.stream(strings)
               .filter(s -> s.compareToIgnoreCase("n") < 0)
               .sorted(String.CASE_INSENSITIVE_ORDER)
               .collect(Collectors.toList()));

      // strings less than "n" (case insensitive) sorted descending
      System.out.printf("strings greater than m sorted descending: %s%n",
         Arrays.stream(strings)
               .filter(s -> s.compareToIgnoreCase("n") < 0)
               .sorted(String.CASE_INSENSITIVE_ORDER.reversed())
               .collect(Collectors.toList()));
   }
} // end class ArraysAndStreams2
```

```
Original strings: [Red, orange, Yellow, green, Blue, indigo, Violet]
strings in uppercase: [RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET]
strings greater than m sorted ascending: [orange, Red, Violet, Yellow]
strings greater than m sorted descending: [Yellow, Violet, Red, orange]
```

Fig. 17.7 | Demonstrating lambdas and streams with an array of `Strings`. (Part 2 of 2.)

17.5.1 Mapping `String`s to Uppercase Using a Method Reference

- `Stream` method `map` maps each element to a new value and produces a new stream with the same number of elements as the original stream.
- A method reference is a shorthand notation for a lambda expression.
- *ClassName*`::`*instanceMethodName* represents a method reference for an instance method of a class.
  - Creates a one-parameter lambda that invokes the instance method on the lambda's argument and returns the method's result.

17.5.1 Mapping Strings to Uppercase Using a Method Reference (Cont.)

- *objectName*`::`*instanceMethodName* represents a method reference for an instance method that should be called on a specific object. Creates a one-parameter lambda that invokes the instance method on the specified object—passing the lambda's argument to the instance method—and returns the method's result.
- *ClassName*`::`*staticMethodName* represents a method reference for a `static` method of a class. Creates a one-parameter lambda in which the lambda's argument is passed to the specified a `static` method and the lambda returns the method's result.

17.5.1 Mapping `String`s to Uppercase Using a Method Reference (Cont.)

- *ClassName*`::new` represents a constructor reference.
  - Creates a lambda that invokes the no-argument constructor of the specified class to create and initialize a new object of that class.
- Figure 17.8 shows the four method reference types.

Lambda	Description
`String::toUpperCase`	Method reference for an instance method of a class. Creates a one-parameter lambda that invokes the instance method on the lambda's argument and returns the method's result. Used in Fig. 17.7.
`System.out::println`	Method reference for an instance method that should be called on a specific object. Creates a one-parameter lambda that invokes the instance method on the specified object—passing the lambda's argument to the instance method—and returns the method's result. Used in Fig. 17.10.
`Math::sqrt`	Method reference for a `static` method of a class. Creates a one-parameter lambda in which the lambda's argument is passed to the specified a `static` method and the lambda returns the method's result.
`TreeMap::new`	Constructor reference. Creates a lambda that invokes the no-argument constructor of the specified class to create and initialize a new object of that class. Used in Fig. 17.17.

Fig. 17.8 | Types of method references.

17.5.2 Filtering Strings Then Sorting Them in Case-Insensitive Ascending Order

- `Stream` method `sorted` can receive a `Comparator` as an argument to specify how to compare stream elements for sorting.
- By default, method `sorted` uses the natural order for the stream's element 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.

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

- Functional interface `Comparator`'s `default` method `reversed` reverses an existing `Comparator`'s ordering.

17.6 Stream<Employee> Manipulations

- The example in Figs. 17.9–17.16 demonstrates various lambda and stream capabilities using a `Stream<Employee>`.
- Class `Employee` (Fig. 17.9) represents an employee with a first name, last name, salary and department and provides methods for manipulating these values.

```java
// Fig. 17.9: Employee.java
// Employee class.
public class Employee
{
   private String firstName;
   private String lastName;
   private double salary;
   private String department;

   // constructor
   public Employee(String firstName, String lastName,
      double salary, String department)
   {
      this.firstName = firstName;
      this.lastName = lastName;
      this.salary = salary;
      this.department = department;
   }

   // set firstName
   public void setFirstName(String firstName)
   {
      this.firstName = firstName;
   }
```

Fig. 17.9 | Employee class for use in Figs. 17.10–17.16. (Part 1 of 4.)

```java

    // get firstName
    public String getFirstName()
    {
       return firstName;
    }

    // set lastName
    public void setLastName(String lastName)
    {
       this.lastName = lastName;
    }

    // get lastName
    public String getLastName()
    {
       return lastName;
    }

    // set salary
    public void setSalary(double salary)
    {
       this.salary = salary;
    }
```

Fig. 17.9 | `Employee` class for use in Figs. 17.10–17.16. (Part 2 of 4.)

```java

    // get salary
    public double getSalary()
    {
       return salary;
    }

    // set department
    public void setDepartment(String department)
    {
       this.department = department;
    }

    // get department
    public String getDepartment()
    {
       return department;
    }

    // return Employee's first and last name combined
    public String getName()
    {
       return String.format("%s %s", getFirstName(), getLastName());
    }
```

Fig. 17.9 | Employee class for use in Figs. 17.10–17.16. (Part 3 of 4.)

```java

   // return a String containing the Employee's information
   @Override
   public String toString()
   {
      return String.format("%-8s %-8s %8.2f   %s",
         getFirstName(), getLastName(), getSalary(), getDepartment());
   } // end method toString
} // end class Employee
```

Fig. 17.9 | Employee class for use in Figs. 17.10–17.16. (Part 4 of 4.)

17.6.1 Creating and Displaying a `List<Employee>`

- When the instance method reference `System.out::println` is passed to `Stream` method `forEach`, it's converted by the compiler into an object that implements the `Consumer` functional interface.
  - This interface's `accept` method receives one argument and returns `void`. In this case, the `accept` method passes the argument to the `System.out` object's `println` instance method.
- Class `ProcessingEmployees` (Figs. 17.10–17.16) is split into several figures so we can show you the lambda and streams operations with their corresponding outputs.

17.6.1 Creating and Displaying a `List<Employee>` (Cont.)

- Figure 17.10 creates an array of `Employee`s and gets its `List` view.

```java
// Fig. 17.10: ProcessingEmployees.java
// Processing streams of Employee objects.
import java.util.Arrays;
import java.util.Comparator;
import java.util.List;
import java.util.Map;
import java.util.TreeMap;
import java.util.function.Function;
import java.util.function.Predicate;
import java.util.stream.Collectors;

public class ProcessingEmployees
{
   public static void main(String[] args)
   {
      // initialize array of Employees
      Employee[] employees = {
         new Employee("Jason", "Red", 5000, "IT"),
         new Employee("Ashley", "Green", 7600, "IT"),
         new Employee("Matthew", "Indigo", 3587.5, "Sales"),
         new Employee("James", "Indigo", 4700.77, "Marketing"),
         new Employee("Luke", "Indigo", 6200, "IT"),
         new Employee("Jason", "Blue", 3200, "Sales"),
         new Employee("Wendy", "Brown", 4236.4, "Marketing")};
```

Fig. 17.10 | Creating an array of `Employees`, converting it to a `List` and displaying the `List`. (Part 1 of 2.)

```java

        // get List view of the Employees
        List<Employee> list = Arrays.asList(employees);

        // display all Employees
        System.out.println("Complete Employee list:");
        list.stream().forEach(System.out::println);

```

```
Complete Employee list:
Jason     Red       5000.00   IT
Ashley    Green     7600.00   IT
Matthew   Indigo    3587.50   Sales
James     Indigo    4700.77   Marketing
Luke      Indigo    6200.00   IT
Jason     Blue      3200.00   Sales
Wendy     Brown     4236.40   Marketing
```

Fig. 17.10 | Creating an array of `Employees`, converting it to a `List` and displaying the `List`. (Part 2 of 2.)

17.6.2 Filtering Employees with Salaries in a Specified Range

- Figure 17.11 demonstrates filtering `Employee`s with an object that implements the functional interface `Predicate<Employee>`, which is defined with a lambda
- To reuse a lambda, you can assign it to a variable of the appropriate functional interface type.
- The `Comparator` interface's `static` method `comparing` receives a `Function` that's used to extract a value from an object in the stream for use in comparisons and returns a `Comparator` object.

```java
      // Predicate that returns true for salaries in the range $4000-$6000
      Predicate<Employee> fourToSixThousand =
         e -> (e.getSalary() >= 4000 && e.getSalary() <= 6000);

      // Display Employees with salaries in the range $4000-$6000
      // sorted into ascending order by salary
      System.out.printf(
         "%nEmployees earning $4000-$6000 per month sorted by salary:%n");
      list.stream()
          .filter(fourToSixThousand)
          .sorted(Comparator.comparing(Employee::getSalary))
          .forEach(System.out::println);

      // Display first Employee with salary in the range $4000-$6000
      System.out.printf("%nFirst employee who earns $4000-$6000:%n%s%n",
         list.stream()
             .filter(fourToSixThousand)
             .findFirst()
             .get());

```

Fig. 17.11 | Filtering `Employees` with salaries in the range $4000–$6000. (Part 1 of 2.)

```
Employees earning $4000-$6000 per month sorted by salary:
Wendy    Brown     4236.40   Marketing
James    Indigo    4700.77   Marketing
Jason    Red       5000.00   IT

First employee who earns $4000-$6000:
Jason    Red       5000.00   IT
```

Fig. 17.11 | Filtering `Employees` with salaries in the range $4000–$6000. (Part 2 of 2.)

17.6.2 Filtering Employees with Salaries in a Specified Range (Cont.)

- A nice performance feature of lazy evaluation is the ability to perform short circuit evaluation—that is, to stop processing the stream pipeline as soon as the desired result is available.
- `Stream` method `findFirst` is a short-circuiting terminal operation that processes the stream pipeline and terminates processing as soon as the first object from the stream pipeline is found.
  - Returns an `Optional` containing the object that was found, if any.

17.6.3 Sorting Employees By Multiple Fields

- Figure 17.12 shows how to use streams to sort objects by *multiple* fields.
- To sort objects by two fields, you create a `Comparator` that uses two `Function`s.
- First you call `Comparator` method `comparing` to create a `Comparator` with the first `Function`.
- On the resulting `Comparator`, you call method `thenComparing` with the second `Function`.
- The resulting `Comparator` compares objects using the first `Function` then, for objects that are equal, compares them by the second `Function`.

```java
      // Functions for getting first and last names from an Employee
      Function<Employee, String> byFirstName = Employee::getFirstName;
      Function<Employee, String> byLastName = Employee::getLastName;

      // Comparator for comparing Employees by first name then last name
      Comparator<Employee> lastThenFirst =
         Comparator.comparing(byLastName).thenComparing(byFirstName);

      // sort employees by last name, then first name
      System.out.printf(
         "%nEmployees in ascending order by last name then first:%n");
      list.stream()
          .sorted(lastThenFirst)
          .forEach(System.out::println);

      // sort employees in descending order by last name, then first name
      System.out.printf(
         "%nEmployees in descending order by last name then first:%n");
      list.stream()
          .sorted(lastThenFirst.reversed())
          .forEach(System.out::println);

```

Fig. 17.12 | Sorting Employees by last name then first name. (Part 1 of 2.)

```
Employees in ascending order by last name then first:
Jason     Blue      3200.00   Sales
Wendy     Brown     4236.40   Marketing
Ashley    Green     7600.00   IT
James     Indigo    4700.77   Marketing
Luke      Indigo    6200.00   IT
Matthew   Indigo    3587.50   Sales
Jason     Red       5000.00   IT
```

```
Employees in descending order by last name then first:
Jason     Red       5000.00   IT
Matthew   Indigo    3587.50   Sales
Luke      Indigo    6200.00   IT
James     Indigo    4700.77   Marketing
Ashley    Green     7600.00   IT
Wendy     Brown     4236.40   Marketing
Jason     Blue      3200.00   Sales
```

Fig. 17.12 | Sorting `Employees` by last name then first name. (Part 2 of 2.)

17.6.4 Mapping `Employees` to Unique Last Name `Strings`

- Figure 17.13 shows how to map objects of one type (`Employee`) to objects of a different type (`String`).
- You can map objects in a stream to different types to produce another stream with the same number of elements as the original stream.
- `Stream` method `distinct` eliminates duplicate objects in a stream.

```java
        // display unique employee last names sorted
        System.out.printf("%nUnique employee last names:%n");
        list.stream()
            .map(Employee::getLastName)
            .distinct()
            .sorted()
            .forEach(System.out::println);

        // display only first and last names
        System.out.printf(
           "%nEmployee names in order by last name then first name:%n");
        list.stream()
            .sorted(lastThenFirst)
            .map(Employee::getName)
            .forEach(System.out::println);

```

Fig. 17.13 | Mapping `Employee` objects to last names and whole names. (Part 1 of 2.)

```
Unique employee last names:
Blue
Brown
Green
Indigo
Red

Employee names in order by last name then first name:
Jason Blue
Wendy Brown
Ashley Green
James Indigo
Luke Indigo
Matthew Indigo
Jason Red
```

Fig. 17.13 | Mapping `Employee` objects to last names and whole names. (Part 2 of 2.)

17.6.5 Grouping Employees By Department

- Figure 17.14 uses `Stream` method `collect` to group `Employees` by department.
- `Collectors` `static` method `groupingBy` with one argument receives a `Function` that classifies objects in the stream—the values returned by this function are used as the keys in a `Map`.
  - The corresponding values, by default, are `List`s containing the stream elements in a given category.
- `Map` method `forEach` performs an operation on each key–value pair.
  - Receives an object that implements functional interface `BiConsumer`.
  - `BiConsumer`'s `accept` method has two parameters.
  - For `Map`s, the first represents the key and the second the corresponding value.

```java
        // group Employees by department
        System.out.printf("%nEmployees by department:%n");
        Map<String, List<Employee>> groupedByDepartment =
           list.stream()
               .collect(Collectors.groupingBy(Employee::getDepartment));
        groupedByDepartment.forEach(
           (department, employeesInDepartment) ->
           {
              System.out.println(department);
              employeesInDepartment.forEach(
                 employee -> System.out.printf("   %s%n", employee));
           }
        );

```

Fig. 17.14 | Grouping `Employees` by department. (Part 1 of 2.)

```
Employees by department:
Sales
   Matthew  Indigo    3587.50   Sales
   Jason    Blue      3200.00   Sales
IT
   Jason    Red       5000.00   IT
   Ashley   Green     7600.00   IT
   Luke     Indigo    6200.00   IT
Marketing
   James    Indigo    4700.77   Marketing
   Wendy    Brown     4236.40   Marketing
```

Fig. 17.14 | Grouping `Employees` by department. (Part 2 of 2.)

17.6.6 Counting the Number of Employees in Each Department

- Figure 17.15 once again demonstrates `Stream` method `collect` and `Collectors` `static` method `groupingBy`, but in this case we count the number of `Employee`s in each department.
- `Collectors` `static` method `groupingBy` with two arguments receives a `Function` that classifies the objects in the stream and another `Collector` (known as the downstream `Collector`).
- `Collectors` `static` method `counting` returns a `Collector` that counts the number of objects in a given classification, rather than collecting them into a `List`.

```java
      // count number of Employees in each department
      System.out.printf("%nCount of Employees by department:%n");
      Map<String, Long> employeeCountByDepartment =
         list.stream()
             .collect(Collectors.groupingBy(Employee::getDepartment,
                Collectors.counting()));
      employeeCountByDepartment.forEach(
         (department, count) -> System.out.printf(
            "%s has %d employee(s)%n", department, count));

```

```
Count of Employees by department:
IT has 3 employee(s)
Marketing has 2 employee(s)
Sales has 2 employee(s)
```

Fig. 17.15 | Counting the number of `Employees` in each department.

17.6.7 Summing and Averaging Employee Salaries

- Figure 17.16 demonstrates `Stream` method **`mapToDouble`**, which maps objects to `double` values and returns a `DoubleStream`.
- `Stream` method `mapToDouble` maps objects to `double` values and returns a `DoubleStream`. The method receives an object that implements the functional interface `ToDoubleFunction` (package `java.util.function`).
  - This interface's `applyAsDouble` method invokes an instance method on an object and returns a `double` value.

```java
        // sum of Employee salaries with DoubleStream sum method
        System.out.printf(
           "%nSum of Employees' salaries (via sum method): %.2f%n",
           list.stream()
               .mapToDouble(Employee::getSalary)
               .sum());

        // calculate sum of Employee salaries with Stream reduce method
        System.out.printf(
           "Sum of Employees' salaries (via reduce method): %.2f%n",
           list.stream()
               .mapToDouble(Employee::getSalary)
               .reduce(0, (value1, value2) -> value1 + value2));

        // average of Employee salaries with DoubleStream average method
        System.out.printf("Average of Employees' salaries: %.2f%n",
           list.stream()
               .mapToDouble(Employee::getSalary)
               .average()
               .getAsDouble());
     } // end main
  } // end class ProcessingEmployees
```

Fig. 17.16 | Summing and averaging `Employee` salaries. (Part 1 of 2.)

```
Sum of Employees' salaries (via sum method): 34524.67
Sum of Employees' salaries (via reduce method): 34525.67
Average of Employees' salaries: 4932.10
```

Fig. 17.16 | Summing and averaging `Employee` salaries. (Part 2 of 2.)

17.7 Creating a `Stream<String>` from a File

- Figure 17.17 uses lambdas and streams to summarize the number of occurrences of each word in a file then display a summary of the words in alphabetical order grouped by starting letter.
- Figure 17.18 shows the program's output.
- `Files` method `lines` creates a `Stream<String>` for reading the lines of text from a file.
- `Stream` method `flatMap` receives a `Function` that maps an object into a stream—e.g., a line of text into words.
- `Pattern` method `split-AsStream` uses a regular expression to tokenize a `String`.

```java
// Fig. 17.17: StreamOfLines.java
// Counting word occurrences in a text file.
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.Map;
import java.util.TreeMap;
import java.util.regex.Pattern;
import java.util.stream.Collectors;

public class StreamOfLines
{
   public static void main(String[] args) throws IOException
   {
      // Regex that matches one or more consecutive whitespace characters
      Pattern pattern = Pattern.compile("\\s+");

      // count occurrences of each word in a Stream<String> sorted by word
      Map<String, Long> wordCounts =
         Files.lines(Paths.get("Chapter2Paragraph.txt"))
              .map(line -> line.replaceAll("(?!')\\p{P}", ""))
              .flatMap(line -> pattern.splitAsStream(line))
              .collect(Collectors.groupingBy(String::toLowerCase,
                 TreeMap::new, Collectors.counting()));
```

Fig. 17.17 | Counting word occurrences in a text file. (Part 1 of 2.)

```java

      // display the words grouped by starting letter
      wordCounts.entrySet()
         .stream()
         .collect(
            Collectors.groupingBy(entry -> entry.getKey().charAt(0),
               TreeMap::new, Collectors.toList()))
         .forEach((letter, wordList) ->
            {
               System.out.printf("%n%C%n", letter);
               wordList.stream().forEach(word -> System.out.printf(
                  "%13s: %d%n", word.getKey(), word.getValue()));
            });
   }
} // end class StreamOfLines
```

Fig. 17.17 | Counting word occurrences in a text file. (Part 2 of 2.)

```
A                          I                          R
            a: 2               inputs: 1             result: 1
          and: 3             instruct: 1            results: 2
  application: 2           introduces: 1                run: 1
   arithmetic: 1
                           J                          S
B                                java: 1               save: 1
        begin: 1                  jdk: 1             screen: 1
                                                       show: 1
C                          L                            sum: 1
   calculates: 1                 last: 1
 calculations: 1                later: 1              T
      chapter: 1                learn: 1               that: 3
     chapters: 1                                        the: 7
  commandline: 1           M                          their: 2
     compares: 1                 make: 1               then: 2
   comparison: 1             messages: 2               this: 2
      compile: 1                                         to: 4
     computer: 1           N                          tools: 1
```

Fig. 17.18 | Output for the program of Fig. 17.17 arranged in three columns.

```
D
    decisions: 1
 demonstrates: 1
      display: 1
     displays: 2

E
      example: 1
     examples: 1

F
          for: 1
         from: 1

H
          how: 2
```

```
      numbers: 2

O
      obtains: 1
           of: 1
           on: 1
       output: 1

P
      perform: 1
      present: 1
      program: 1
  programming: 1
     programs: 2
```

```
          two: 2

U
          use: 2
         user: 1

W
           we: 2
         with: 1

Y
       you'll: 2
```

Fig. 17.18 | Output for the program of Fig. 17.17 arranged in three columns.

17.7 Creating a `Stream<String>` from a File (Cont.)

- `Collectors` method `groupingBy` with three arguments receives a classifier, a `Map` factory and a downstream `Collector`.
  - The classifier is a `Function` that returns objects which are used as keys in the resulting `Map`.
  - The `Map` factory is an object that implements interface `Supplier` and returns a new `Map` collection.
  - The downstream `Collector` determines how to collect each group's elements.
- `Map` method `entrySet` returns a `Set` of `Map.Entry` objects containing the `Map`'s key–value pairs.
- `Set` method `stream` returns a stream for processing the `Set`'s elements.

17.8 Generating Streams of Random Values

- In Fig. 6.7, we demonstrated rolling a six-sided die 6,000,000 times and summarizing the frequencies of each face using *external iteration* (a `for` loop) and a `switch` statement that determined which counter to increment.
- We then displayed the results using separate statements that performed external iteration.

17.8 Generating Streams of Random Values (Cont.)

- In Fig. 7.7, we reimplemented Fig. 6.7, replacing the entire `switch` statement with a single statement that incremented counters in an array—that version of rolling the die still used external iteration to produce and summarize 6,000,000 random rolls and to display the final results.
- Both prior versions of this example, used mutable variables to control the external iteration and to summarize the results.

17.8 Generating Streams of Random Values (Cont.)

- Figure 17.19 reimplements those programs with a *single statement* that does it all, using lambdas, streams, internal iteration and no mutable variables to roll the die 6,000,000 times, calculate the frequencies and display the results.

```java
// Fig. 17.19: RandomIntStream.java
// Rolling a die 6,000,000 times with streams
import java.security.SecureRandom;
import java.util.Map;
import java.util.function.Function;
import java.util.stream.IntStream;
import java.util.stream.Collectors;

public class RandomIntStream
{
   public static void main(String[] args)
   {
      SecureRandom random = new SecureRandom();

      // roll a die 6,000,000 times and summarize the results
      System.out.printf("%-6s%s%n", "Face", "Frequency");
      random.ints(6_000_000, 1, 7)
            .boxed()
            .collect(Collectors.groupingBy(Function.identity(),
               Collectors.counting()))
            .forEach((face, frequency) ->
               System.out.printf("%-6d%d%n", face, frequency));
   }
} // end class RandomIntStream
```

Fig. 17.19 | Rolling a die 6,000,000 times with streams. (Part 1 of 2.)

```
Face   Frequency
1      999339
2      999937
3      1000302
4      999323
5      1000183
6      1000916
```

Fig. 17.19 | Rolling a die 6,000,000 times with streams. (Part 2 of 2.)

17.8 Generating Streams of Random Values (Cont.)

- Class `SecureRandom`'s methods `ints`, `longs` and `doubles` (inherited from class `Random`) return `IntStream`, `LongStream` and `DoubleStream`, respectively, for streams of random numbers.
- Method `ints` with no arguments creates an `IntStream` for an infinite stream of random `int` values.
- An infinite- stream is a stream with an unknown number of elements—you use a short-circuiting terminal operation to complete processing on an infinite stream.

17.8 Generating Streams of Random Values (Cont.)

- Method `ints` with a `long` argument creates an `IntStream` with the specified number of random `int` values.
- Method `ints` with two `int` arguments creates an `IntStream` for an infinite stream of random `int` values in the range starting with the first argument and up to, but not including, the second.
- Method `ints` with a `long` and two `int` arguments creates an `IntStream` with the specified number of random `int` values in the range starting with the first argument and up to, but not including, the second.

17.8 Generating Streams of Random Values (Cont.)

- To convert an `IntStream` to a `Stream<Integer>` call `IntStream` method `boxed`.
- `Function` `static` method `identity` creates a `Function` that simply returns its argument.

17.9 Lambda Event Handlers

- Some event-listener interfaces are functional interfaces. For such interfaces, you can implement event handlers with lambdas.
- For a simple event handler, a lambda significantly reduces the amount of code you need to write.

17.10 Additional Notes on Java SE 8 Interfaces

- Functional interfaces must contain only one `abstract` method, but may also contain `default`-methods and `static` methods that are fully implemented in the interface declarations.
- When a class implements an interface with `default` methods and does not override them, the class inherits the `default` methods' implementations. An interface's designer can now evolve an interface by adding new `default` and `static` methods without breaking existing code that implements the interface.

17.10 Additional Notes on Java SE 8 Interfaces (Cont.)

- If one class inherits the same `default` method from two interfaces, the class must override that method; otherwise, the compiler will generate a compilation error.
- You can create your own functional interfaces by ensuring that each contains only one `abstract-` method and zero or more `default` or `static` methods.
- You can declare that an interface is a functional interface by preceding it with the `@FunctionalInterface` annotation. The compiler will then ensure that the interface contains only one `abstract-` method; otherwise, it'll generate a compilation error.