Data streaming and functional programming in Java

Learn how to use the stream API and functional programming constructs in Java 8.

By Marty Kalin

January 2, 2020 | 3 Comments | 17 min read

119 readers like this.

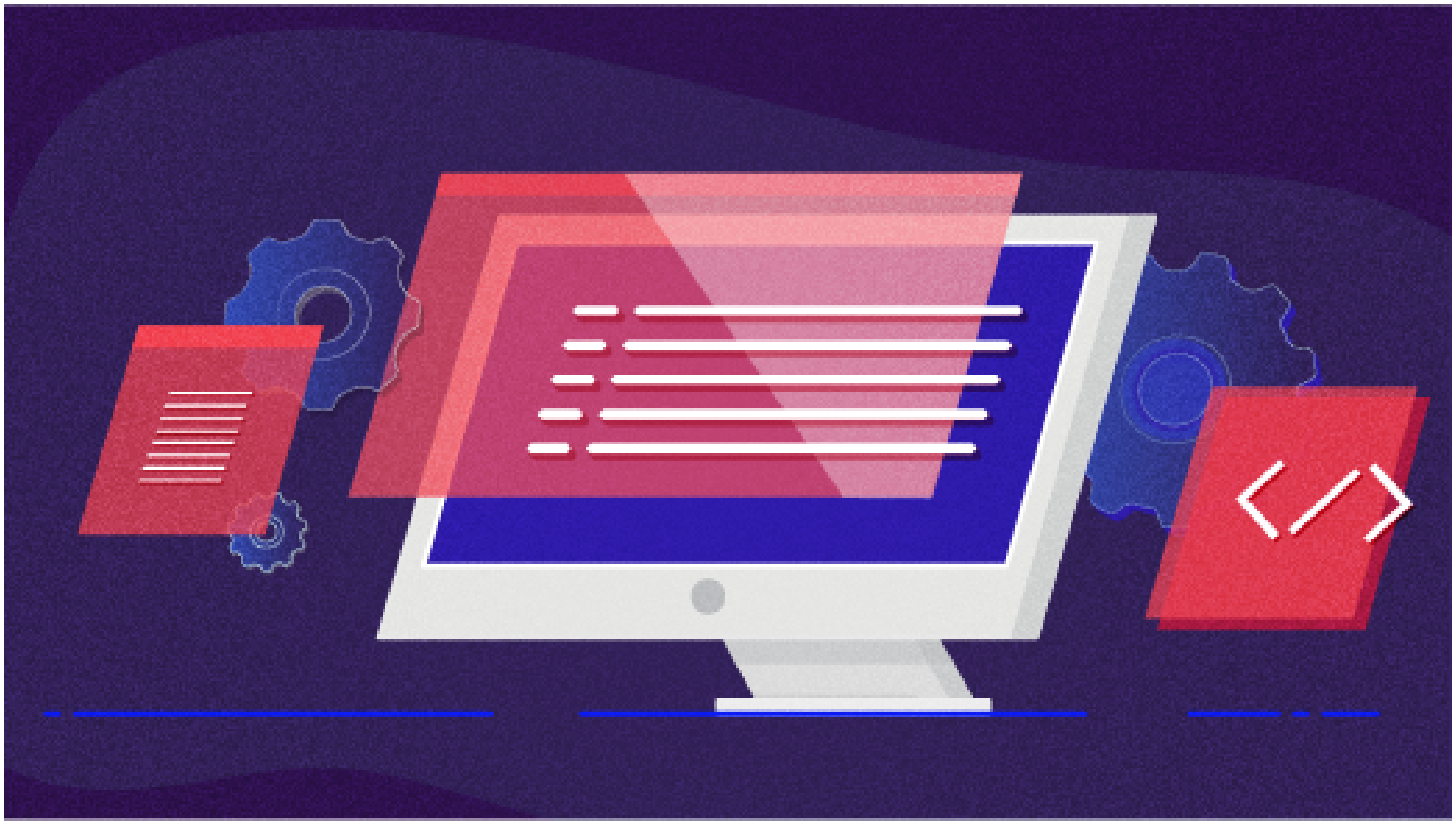

Image by: *Opensource.com*

When Java SE 8 (aka core Java 8) was introduced in 2014, it introduced changes that fundamentally impact programming in it. The changes have two closely linked parts: the stream API and the functional programming constructs. This article uses code examples, from the basics through advanced features, to introduce each part and illustrate the interplay between them.

The basics

The stream API is a concise and high-level way to iterate over the elements in a data sequence. The packages **java.util.stream** and **java.util.function** house the new libraries for the stream API and related functional programming constructs. Of course, a code example is worth a thousand words.

The code segment below populates a **List** with about 2,000 random integer values:

```
Random rand = new Random();
List<Integer> list = new ArrayList<Integer>();           // empty list
for (int i = 0; i < 2048; i++) list.add(rand.nextInt()); // populate it
```

Another **for** loop could be used to iterate over the populated list to collect the even values into another list. The stream API is a cleaner way to do the same:

```
List <Integer> evens = list
   .stream()                      // streamify the list
   .filter(n -> (n & 0x1) == 0)   // filter out odd values
   .collect(Collectors.toList()); // collect even values
```

The example has three functions from the stream API:

The **stream** function can turn a **Collection** into a stream, which is a conveyor belt of values accessible one at a time. The streamification is lazy (and therefore efficient) in that the values are produced as needed rather than all at once.

The **filter** function determines which streamed values, if any, get through to the next stage in the processing pipeline, the **collect** stage. The **filter** function is *higher-order* in that its argument is a function—in this example, a lambda, which is an unnamed function and at the center of Java's new functional programming constructs.

The lambda syntax departs radically from traditional Java:

```
n -> (n & 0x1) == 0
```

The arrow (a minus sign followed immediately by a greater-than sign) separates the argument list on the left from the function's body on the right. The argument **n** is not explicitly typed, although it could be; in any case, the compiler figures out that **n** is an **Integer**. If there were multiple arguments, these would be enclosed in parentheses and separated by commas.

The body, in this example, checks whether an integer's lowest-order (rightmost) bit is a zero, which indicates an even value. A filter should return a boolean value. There is no explicit **return** in the function's body, although there could be. If the body has no explicit **return**, then the body's last expression is the returned value. In this example, written in the spirit of lambda programming, the body consists of the single, simple boolean expression **(n & 0x1) == 0**.

The **collect** function gathers the even values into a list whose reference is **evens**. As an example below illustrates, the **collect** function is thread-safe and, therefore, would work correctly even if the filtering operation was shared among multiple threads.

Convenience functions and easy multi-threading

In a production environment, a data stream might have a file or a network connection as its source. For learning the stream API, Java provides types such as **IntStream**, which can generate streams with elements of various types. Here is an **IntStream** example:

```
IntStream                         // integer stream
   .range(1, 2048)                // generate a stream of ints in this range
   .parallel()                    // partition the data for multiple threads
   .filter(i -> ((i & 0x1) > 0))  // odd parity? pass through only odds
   .forEach(System.out::println); // print each
```

The **IntStream** type includes a **range** function that generates a stream of integer values within a specified range, in this case, from 1 through 2,048, with increments of 1. The **parallel** function automatically partitions the work to be done among multiple threads, each of which does the filtering and printing. (The number of threads typically matches the number of CPUs on the host system.) The argument to the **forEach** function is a *method reference*, in this case, a reference to the **println** method encapsulated in **System.out**, which is of type **PrintStream**. The syntax for method and constructor references will be discussed shortly.

Because of the multi-threading, the integer values are printed in an arbitrary order overall but in sequence within a given thread. For example, if thread T1 prints 409 and 411, then T1 does so in the order 409–411, but some other thread might print 2,045 beforehand. The threads behind the **parallel** call execute concurrently, and the order of their output is therefore indeterminate.

The map/reduce pattern

The *map/reduce* pattern has become popular in processing large datasets. A map/reduce macro operation is built from two micro-operations. The data first are scattered (*mapped*) among various workers, and the separate results then are gathered together—perhaps as a single value, which would be the *reduction*. Reduction can take different forms, as the following examples illustrate.

Instances of the **Number** class below represent integer values with either **EVEN** or **ODD** parity:

```
public class Number {
    enum Parity { EVEN, ODD }
    private int value;
    public Number(int n) { setValue(n); }
    public void setValue(int value) { this.value = value; }
    public int getValue() { return this.value; }
    public Parity getParity() {
        return ((value & 0x1) == 0) ? Parity.EVEN : Parity.ODD;
    }
    public void dump() {
        System.out.format("Value: %2d (parity: %s)\n", getValue(),
                          (getParity() == Parity.ODD ? "odd" : "even"));
    }
}
```

The following code illustrates map/reduce with a **Number** stream, thereby showing that the stream API can handle not only primitive types such as **int** and **float** but programmer-defined class types as well.

In the code segment below, a list of random integer values is streamified using the **parallelStream** rather than the **stream** function. The **parallelStream** variant, like the **parallel** function introduced earlier, does automatic multithreading.

```
final int howMany = 200;
Random r = new Random();
Number[ ] nums = new Number[howMany];
for (int i = 0; i < howMany; i++) nums[i] = new Number(r.nextInt(100));
List<Number> listOfNums = Arrays.asList(nums);  // listify the array

Integer sum4All = listOfNums
   .parallelStream()           // automatic multi-threading
   .mapToInt(Number::getValue) // method reference rather than lambda
   .sum();                     // reduce streamed values to a single value
System.out.println("The sum of the randomly generated values is: " + sum4All);
```

The higher-order **mapToInt** function could take a lambda as an argument, but in this case, it takes a method reference instead, which is **Number::getValue**. The **getValue** method expects no arguments and returns its **int** value for a given **Number** instance. The syntax is uncomplicated: the class name **Number** followed by a double colon and the method's name. Recall the earlier **System.out::println** example, which has the double colon after the **static** field **out** in the **System** class.

The method reference **Number::getValue** could be replaced by the lambda below. The argument **n** is one of the **Number** instances in the stream:

```
mapToInt(n -> n.getValue())
```

In general, lambdas and method references are interchangeable: if a higher-order function such as **mapToInt** can take one form as an argument, then this function could take the other as well. The two functional programming constructs have the same purpose—to perform some customized operation on data passed in as arguments. Choosing between the two is often a matter of convenience. For example, a lambda can be written without an encapsulating class, whereas a method cannot. My habit is to use a lambda unless the appropriate encapsulated method is already at hand.

The **sum** function at the end of the current example does the reduction in a thread-safe manner by combining the partial sums from the **parallelStream** threads. However, the programmer is responsible for ensuring that, in the course of the multi-threading induced by the **parallelStream** call, the programmer's own function calls (in this case, to **getValue**) are thread-safe.

The last point deserves emphasis. Lambda syntax encourages the writing of *pure functions*, which are functions whose return values depend only on the arguments, if any, passed in; a pure function has no side effects such as updating a **static** field in a class. Pure functions are thereby thread-safe, and the stream API works best if the functional arguments passed to higher-order functions, such as **filter** and **map**, are pure functions.

For finer-grained control, there is another stream API function, named **reduce**, that could be used for summing the values in the **Number** stream:

```
Integer sum4AllHarder = listOfNums
   .parallelStream()                           // multi-threading
   .map(Number::getValue)                      // value per Number
   .reduce(0, (sofar, next) -> sofar + next);  // reduction to a sum
```

This version of the **reduce** function takes two arguments, the second of which is a function:

> The first argument (in this case, zero) is the *identity* value, which serves as the initial value for the reduction operation and as the default value should the stream run dry during the reduction.
>
> The second argument is the *accumulator*, in this case, a lambda with two arguments: the first argument (**sofar**) is the running sum, and the second argument (**next**) is the next value from the stream. The running sum and next value then are added to update the accumulator. Keep in mind that both the **map** and the **reduce** functions now execute in a multi-threaded context because of the **parallelStream** call at the start.

In the examples so far, stream values are collected and then reduced, but, in general, the **Collectors** in the stream API can accumulate values without reducing them to a single value. The collection activity can produce arbitrarily rich data structures, as the next code segment illustrates. The example uses the same **listOfNums** as the preceding examples:

```
Map<Number.Parity, List<Number>> numMap = listOfNums
   .parallelStream()
   .collect(Collectors.groupingBy(Number::getParity));

List<Number> evens = numMap.get(Number.Parity.EVEN);
List<Number> odds = numMap.get(Number.Parity.ODD);
```

The **numMap** in the first line refers to a **Map** whose key is a **Number** parity (**ODD** or **EVEN**) and whose value is a **List** of **Number** instances with values having the designated parity. Once again, the processing is multi-threaded through the **parallelStream** call, and the **collect** call then assembles (in a thread-safe manner) the partial results into the single **Map** to which **numMap** refers. The **get** method then is called twice on the **numMap**, once to get the **evens** and a second time to get the **odds**.

The utility function **dumpList** again uses the higher-order **forEach** function from the stream API:

```
private void dumpList(String msg, List<Number> list) {
   System.out.println("\n" + msg);
   list.stream().forEach(n -> n.dump()); // or: forEach(Number::dump)
}
```

Here is a slice of the program's output from a sample run:

```
The sum of the randomly generated values is: 3322
The sum again, using a different method:     3322

Evens:

Value: 72 (parity: even)
Value: 54 (parity: even)
...
Value: 92 (parity: even)

Odds:

Value: 35 (parity: odd)
Value: 37 (parity: odd)
...
Value: 41 (parity: odd)
```

Functional constructs for code simplification

Functional constructs, such as method references and lambdas, fit nicely into the stream API. These constructs represent a major simplification of higher-order functions in Java. Even in the bad old days, Java technically supported higher-order functions through the **Method** and **Constructor** types, instances of which could be passed as arguments to other functions. These types were used—but rarely in production-grade Java precisely because of their complexity. Invoking a **Method**, for example, requires either an object reference (if the method is non-**static**) or at least a class identifier (if the method is **static**). The arguments for the invoked **Method** then are passed to it as **Object** instances, which may require explicit downcasting if polymorphism (another complexity!) is not in play. By contrast, lambdas and method references are easy to pass as arguments to other functions.

The new functional constructs have uses beyond the stream API, however. Consider a Java GUI program with a button for the user to push, for example, to get the current time. The event handler for the button push might be written as follows:

```
JButton updateCurrentTime = new JButton("Update current time");
updateCurrentTime.addActionListener(new ActionListener() {
   @Override
   public void actionPerformed(ActionEvent e) {
      currentTime.setText(new Date().toString());
   }
});
```

This short code segment is a challenge to explain. Consider the second line in which the argument to the method **addActionListener** begins as follows:

```
new ActionListener() {
```

This seems wrong in that **ActionListener** is an **abstract** interface, and **abstract** types cannot be instantiated with a call to **new**. However, it turns out that something else entirely is being instantiated: an unnamed inner class that implements this interface. If the code above were encapsulated in a class named **OldJava**, then this unnamed inner class would be compiled as **OldJava$1.class**. The **actionPerformed** method is overridden in the unnamed inner class.

Now consider this refreshing change with the new functional constructs:

```
updateCurrentTime.addActionListener(e -> currentTime.setText(new Date().toString()));
```

The argument **e** in the lambda is an **ActionEvent** instance, and the lambda's body is a simple call to **setText** on the button.

Functional interfaces and composition

The lambdas used so far have been written in place. For convenience, however, there can be references to lambdas just as there are to encapsulated methods. The following series of short examples illustrate this.

Consider this interface definition:

```
@FunctionalInterface // optional, usually omitted
interface BinaryIntOp {
    abstract int compute(int arg1, int arg2); // abstract could be dropped
}
```

The annotation **@FunctionalInterface** applies to any interface that declares a *single* abstract method; in this case, **compute**. Several standard interfaces (e.g., the **Runnable** interface with its single declared method, **run**) fit the bill. In this example, **compute** is the declared method. The interface can be used as the target type in a reference declaration:

```
BinaryIntOp div = (arg1, arg2) -> arg1 / arg2;
div.compute(12, 3); // 4
```

The package **java.util.function** provides various functional interfaces. Some examples follow.

The code segment below introduces the parameterized **Predicate** functional interface. In this example, the type **Predicate<String>** with parameter **String** can refer to either a lambda with a **String** argument or a **String** method such as **isEmpty**. In general, a *predicate* is a function that returns a boolean value.

```
Predicate<String> pred = String::isEmpty; // predicate for a String method
String[ ] strings = {"one", "two", "", "three", "four"};
Arrays.asList(strings)
   .stream()
   .filter(pred)                  // filter out non-empty strings
   .forEach(System.out::println); // only the empty string is printed
```

The **isEmpty** predicate evaluates to **true** just in case a string's length is zero; hence, only the empty string makes it through to the **forEach** stage in the pipeline.

The next code segments illustrate how simple lambdas or method references can be composed into richer ones. Consider this series of assignments to references of the **IntUnaryOperator** type, which takes an integer argument and returns an integer value:

```
IntUnaryOperator doubled = n -> n * 2;
IntUnaryOperator tripled = n -> n * 3;
IntUnaryOperator squared = n -> n * n;
```

IntUnaryOperator is a **FunctionalInterface** whose single declared method is **applyAsInt**. The three references **doubled**, **tripled**, and **squared** now can be used standalone or in various compositions:

```
int arg = 5;
doubled.applyAsInt(arg); // 10
tripled.applyAsInt(arg); // 15
squared.applyAsInt(arg); // 25
```

Here are some sample compositions:

```
int arg = 5;
doubled.compose(squared).applyAsInt(arg); // doubled-the-squared: 50
tripled.compose(doubled).applyAsInt(arg); // tripled-the-doubled: 30
doubled.andThen(squared).applyAsInt(arg); // doubled-andThen-squared: 100
squared.andThen(tripled).applyAsInt(arg); // squared-andThen-tripled: 75
```

Compositions could be done with in-place lambdas, but the references make the code cleaner.

Constructor references

Constructor references are yet another of the functional programming constructs, but these references are useful in more subtle contexts than lambdas and method references. Once again, a code example seems the best way to clarify.

Consider this POJO class:

```
public class BedRocker { // resident of Bedrock
    private String name;
    public BedRocker(String name) { this.name = name; }
    public String getName() { return this.name; }
    public void dump() { System.out.println(getName()); }
}
```

The class has a single constructor, which requires a **String** argument. Given an array of names, the goal is to generate an array of **BedRocker** elements, one per name. Here is the code segment that uses functional constructs to do so:

```
String[ ] names = {"Fred", "Wilma", "Peebles", "Dino", "Baby Puss"};

Stream<BedRocker> bedrockers = Arrays.asList(names).stream().map(BedRocker::new);
BedRocker[ ] arrayBR = bedrockers.toArray(BedRocker[]::new);

Arrays.asList(arrayBR).stream().forEach(BedRocker::dump);
```

At a high level, this code segment transforms names into **BedRocker** array elements. In detail, the code works as follows. The **Stream** interface (in the package **java.util.stream**) can be parameterized, in this case, to generate a stream of **BedRocker** items named **bedrockers**.

The **Arrays.asList** utility again is used to streamify an array, **names**, with each stream item then passed to the **map** function whose argument now is the constructor reference **BedRocker::new**. This constructor reference acts as an object factory by generating and initializing, on each call, a **BedRocker** instance. After the second line executes, the stream named **bedrockers** consists of five **BedRocker** items.

The example can be clarified further by focusing on the higher-order **map** function. In a typical case, a mapping transforms a value of one type (e.g., an **int**) into a different value of the *same* type (e.g., an integer's successor):

```
map(n -> n + 1) // map n to its successor
```

In the **BedRocker** example, however, the transformation is more dramatic because a value of one type (a **String** representing a name) is mapped to a value of a *different* type, in this case, a **BedRocker** instance with the string as its name. The transformation is done through a constructor call, which is enabled by the constructor reference:

```
map(BedRocker::new) // map a String to a BedRocker
```

The value passed to the constructor is one of the names in the **names** array.

The second line of this code example also illustrates the by-now-familiar transformation of an array first into a **List** and then into a **Stream**:

```
Stream<BedRocker> bedrockers = Arrays.asList(names).stream().map(BedRocker::new);
```

The third line goes the other way—the stream **bedrockers** is transformed into an array by invoking the **toArray** method with the *array* constructor reference **BedRocker[]::new**:

```
BedRocker[ ] arrayBR = bedrockers.toArray(BedRocker[]::new);
```

This constructor reference does not create a single **BedRocker** instance, but rather an entire array of these: the constructor reference is now **BedRocker[]::new** rather than **BedRocker::new**. For confirmation, the **arrayBR** is transformed into a **List**, which again is streamified so that **forEach** can be used to print the **BedRocker** names:

```
Fred
Wilma
Peebles
Dino
Baby Puss
```

The example's subtle transformations of data structures are done with but few lines of code, underscoring the power of various higher-order functions that can take a lambda, a method reference, or a constructor reference as an argument

Currying

To *curry* a function is to reduce (typically by one) the number of explicit arguments required for whatever work the function does. (The term honors the logician Haskell Curry.) In general, functions are easier to call and are more robust if they have fewer arguments. (Recall some nightmarish function that expects a half-dozen or so arguments!) Accordingly, currying should be seen as an effort to simplify a function call. The interface types in the **java.util.function** package are suited for currying, as the next example shows.

References of the **IntBinaryOperator** interface type are for functions that take two integer arguments and return an integer value:

```
IntBinaryOperator mult2 = (n1, n2) -> n1 * n2;
mult2.applyAsInt(10, 20); // 200
mult2.applyAsInt(10, 30); // 300
```

The reference name **mult2** underscores that two explicit arguments are required, in this example, 10 and 20.

The previously introduced **IntUnaryOperator** is simpler than an **IntBinaryOperator** because the former requires just one argument, whereas the latter requires two arguments. Both return an integer value. The goal, therefore, is to curry the two-argument **IntBinraryOperator** named **mult2** into a one-argument **IntUnaryOperator** version **curriedMult2**.

Consider the type **IntFunction<R>**. A function of this type takes an integer argument and returns a result of type **R**, which could be another function—indeed, an **IntBinaryOperator**. Having a lambda return another lambda is straightforward:

```
arg1 -> (arg2 -> arg1 * arg2) // parentheses could be omitted
```

The full lambda starts with **arg1,** and this lambda's body—and returned value—is another lambda, which starts with **arg2**. The returned lambda takes just one argument (**arg2**) but returns the product of two numbers (**arg1** and **arg2**). The following overview, followed by the code, should clarify.

Here is an overview of how **mult2** can be curried:

> A lambda of type **IntFunction<IntUnaryOperator>** is written and called with an integer value such as 10. The returned **IntUnaryOperator** caches the value 10 and thereby becomes the curried version of **mult2**, in this example, **curriedMult2**.
>
> The **curriedMult2** function then is called with a single explicit argument (e.g., 20), which is multiplied with the cached argument (in this case, 10) to produce the product returned.

Here are the details in code:

```
// Create a function that takes one argument n1 and returns a one-argument
// function n2 -> n1 * n2 that returns an int (the product n1 * n2).
IntFunction<IntUnaryOperator> curriedMult2Maker = n1 -> (n2 -> n1 * n2);
```

Calling the **curriedMult2Maker** generates the desired **IntUnaryOperator** function:

```
// Use the curriedMult2Maker to get a curried version of mult2.
// The argument 10 is n1 from the lambda above.
IntUnaryOperator curriedMult2 = curriedMult2Maker2.apply(10);
```

The value 10 is now cached in the **curriedMult2** function so that the explicit integer argument in a **curriedMult2** call will be multiplied by 10:

```
curriedMult2.applyAsInt(20); // 200 = 10 * 20
curriedMult2.applyAsInt(80); // 800 = 10 * 80
```

The cached value can be changed at will:

```
curriedMult2 = curriedMult2Maker.apply(50); // cache 50
curriedMult2.applyAsInt(101);               // 5050 = 101 * 50
```

Of course, multiple curried versions of **mult2**, each an **IntUnaryOperator**, can be created in this way.

Currying takes advantage of a powerful feature about lambdas: a lambda is easily written to return whatever type of value is needed, including another lambda.

Wrapping up

Java remains a class-based object-oriented programming language. But with the stream API and its supporting functional constructs, Java takes a decisive (and welcomed) step toward functional languages such as Lisp. The result is a Java better suited to process the massive data streams so common in modern programming. This step in the functional direction also makes it easier to write clear, concise Java in the pipeline style highlighted in previous code examples:

```
dataStream
   .parallelStream() // multi-threaded for efficiency
   .filter(...)      // stage 1
   .map(...)         // stage 2
   .filter(...)      // stage 3
   ...
   .collect(...);    // or, perhaps, reduce: stage N
```

The automatic multi-threading, illustrated with the **parallel** and **parallelStream** calls, is built upon Java's fork/join framework, which supports *task stealing* for efficiency. Suppose that the thread pool behind a **parallelStream** call consists of eight threads and that the **dataStream** is partitioned eight ways. Some thread (e.g., T1) might work faster than another (e.g., T7), which means that some of T7's tasks ought to be moved into T1's work queue. This happens automatically at runtime.

The programmer's chief responsibility in this easy multi-threading world is to write thread-safe functions passed as arguments to the higher-order functions that dominate in the stream API. Lambdas, in particular, encourage the writing of pure—and, therefore, thread-safe—functions.