State of the Lambda: Libraries Edition

September 2013

Java SE 8 Edition

This is an informal overview of the major library enhancements in Java SE 8 to take advantage of new language features, primarily lambda expressions and default methods, specified by JSR 335 and implemented in the OpenJDK Lambda Project. It refines the previous iteration posted in November 2012. The new language features for Java SE 8 are described in State of the Lambda, which should be read first.

Background

Had lambda expressions (closures) been part of the Java language from the beginning, our Collections APIs would certainly look different than they do today. As the Java language acquires lambda expressions as part of JSR 335, this has the unfortunate side effect of making our Collections interfaces look even more out of date! While it might be tempting to start from scratch and build a replacement Collections framework ("Collections II"), replacing the Collections framework would be a major task, as the Collections interfaces permeate the entire Java ecosystem, and the adoption lag would be many years. Instead, we pursue an evolutionary strategy of adding extension methods to existing interfaces (such as **`Collection`**, **`List`**, or **`Iterable`**), and adding a *stream* abstraction (e.g., **`java.util.stream.Stream`**) for performing aggregate operations on datasets, and retrofitting existing classes to provide stream views, enabling many of the desired idioms without making people trade in their trusty **`ArrayList`**s and **`HashMap`**s. (This is not to say that Collections will never be replaced; clearly there are limitations beyond simply not being designed for lambdas. A more modern collections framework may be considered for a future version of the JDK.)

A key driver for this work is making parallelism more accessible to developers. While the Java platform provides strong support for concurrency and parallelism already, developers face unnecessary impediments in migrating their code from sequential to parallel as needed. Therefore, it is important to encourage idioms that are *both* sequential- and parallel-friendly. This is facilitated by shifting the focus towards describing *what* computation should be performed, rather than *how* it should be performed. It is also important to strike the balance between making parallelism *easier* but not going so far as to make it *invisible*; our goal is *explicit but unobstrusive* parallelism. (Making parallelism transparent would introduce nondeterminism and the possibility of data races where users might not expect it.)

Internal vs external iteration

The Collections framework relies on the concept of *external iteration*, where a **`Collection`** provides, by implementing **`Iterable`**, a means to enumerate its elements, and clients use this to step sequentially through the elements of a collection. For example, if we wanted to set the color of every shape in a collection of shapes to red, we would write:

```
for (Shape s : shapes) {
    s.setColor(RED);
}
```

This example illustrates external iteration; the for-each loop calls the **`iterator()`** method of **`shapes`**, and steps through the collection one by one. External iteration is straightforward enough, but it has several problems:

- Java's for-loop is inherently sequential, and must process the elements in the order specified by the collection.
- It deprives the library method of the opportunity to manage the control flow, which might be able to provide better peformance by exploiting reordering of the data, parallelism, short-circuiting, or laziness.

Sometimes the strong guarantees of the for-each loop (sequential, in-order) are desirable, but often are just an impediment to performance.

The alternative to external iteration is *internal iteration*, where instead of controlling the iteration, the client delegates that to the library and passes in snippets of code to execute at various points in the computation.

The internal-iteration equivalent of the previous example is:

```
shapes.forEach(s -> s.setColor(RED));
```

This may appear to be a small syntactic change, but the difference is significant. The control of the operation has been handed off from the client code to the library code, allowing the libraries not only to abstract over common control flow operations, but also enabling them to potentially use laziness, parallelism, and out-of-order execution to improve performance. (Whether an implementation of **`forEach`** actually does any of these things is for the implementation to determine, but with internal iteration they are at least possible, whereas with external iteration, they are not.)

Whereas external iteration mixes the *what* (color the shapes red) and the *how* (get an **`Iterator`** and iterate it

sequentially), internal iteration lets the client dictate the what but lets the library control the how. This offers several potential benefits: client code can be clearer, since it need only focus on stating the problem, not the details of how to go about solving it, and we can move complex optimization code into libraries where it can benefit all users.

Streams

The key new library abstraction introduced in Java SE 8 is a *stream*, defined in package `java.util.stream`. (There are several stream types; `Stream<T>` represents a stream of object references, and there are specializations such as `IntStream` to describe streams of primitives.) A stream represents a sequence of values, and exposes a set of aggregate operations that allow us to express common manipulations on those values easily and clearly. The libraries provide convenient ways to obtain stream views of collections, arrays, and other data sources.

Stream operations are chained together into *pipelines*. For example, if we wanted to color only the blue shapes red, we could say:

```
shapes.stream()
      .filter(s -> s.getColor() == BLUE)
      .forEach(s -> s.setColor(RED));
```

The `stream()` method on `Collection` produces a stream view of the elements of that collection; the `filter()` operation then produces a stream containing the shapes that are blue, and these elements are then made red by `forEach()`.

If we wanted to collect the blue shapes into a new `List`, we could say:

```
List<Shape> blue = shapes.stream()
                         .filter(s -> s.getColor() == BLUE)
                         .collect(Collectors.toList());
```

The `collect()` operation collects the input elements into an aggregate (such as a `List`) or a summary description; the argument to `collect()` is a recipe for how to do this aggregation. In this case, we use `toList()`, which is a simple recipe that accumulates the elements into a `List`. (More detail on `collect()` can be found in the "Collectors" section.)

If each shape were contained in a `Box`, and we wanted to know which boxes contained at least one blue shape, we could say:

```
Set<Box> hasBlueShape = shapes.stream()
                              .filter(s -> s.getColor() == BLUE)
                              .map(s -> s.getContainingBox())
                              .collect(Collectors.toSet());
```

The `map()` operation produces a stream whose values are the result of applying a mapping function (here, one that takes a shape and returns its containing box) to each element in its input stream.

If we wanted to add up the total weight of the blue shapes, we could express that as:

```
int sum = shapes.stream()
                .filter(s -> s.getColor() == BLUE)
                .mapToInt(s -> s.getWeight())
                .sum();
```

So far, we haven't provided much detail about the specific signatures of the Stream operations shown; these examples simply illustrate the types of problems that the Streams framework is designed to address.

Streams vs Collections

Collections and streams, while bearing some superficial similarities, have different goals. Collections are primarily concerned with the efficient management of, and access to, their elements. By contrast, streams do not provide a means to directly access or manipulate their elements, and are instead concerned with declaratively describing the computational operations which will be performed in aggregate on that source. Accordingly, streams differ from Collections in several ways:

- No storage. Streams don't have storage for values; they carry values from a source (which could be a data structure, a generating function, an I/O channel, etc) through a pipeline of computational steps.
- Functional in nature. Operations on a stream produce a result, but do not modify its underlying data source.
- Laziness-seeking. Many stream operations, such as filtering, mapping, sorting, or duplicate removal) can be implemented lazily. This facilitates efficient single-pass execution of entire pipelines, as well as facilitating efficient implementation of short-circuiting operations.
- Bounds optional. There are many problems that are sensible to express as infinite streams, letting clients consume values until they are satisfied. (If we were enumerating perfect numbers, it is easy to express this as a filtering operation on the stream of all integers.) While a Collection is constrained to be finite, a stream is not. (To terminate in finite time, a stream pipeline with an infinite source can use short-circuiting operations; alternately, you can request an `Iterator` from a `Stream` and traverse it manually.)

As an API, Streams is completely independent from Collections. While it is easy to use a collection as the source for a stream (`Collection` has `stream()` and `parallelStream()` methods) or to dump the elements of a stream into a collection (using the `collect()` operation as shown earlier), aggregates other than `Collection` can be used as sources for streams as well. Many JDK classes, such as `BufferedReader`, `Random`, and `BitSet`, have been retrofitted to act as sources for streams, and `Arrays.stream()` provides stream view of arrays. In fact, anything that can be described with an `Iterator` can be used as a stream source, though if more information is available (such as size or metadata about stream contents like sortedness), the library can provide an optimized execution.

Laziness

Operations like filtering or mapping, can be performed *eagerly* (where the filtering is performed on all elements before the `filter` method returns), or *lazily* (where the `Stream` representing the filtered result only applies the filter to elements from its source as needed.) Performing computations lazily, where practical, can be beneficial. For example, if we perform filtering lazily, we can fuse the filtering with other operations later in the pipeline, so as not to require multiple passes on the data. Similarly, if we are searching a large collection for the first element that matches a given criteria, we can stop once we find one, rather than processing the entire collection. (This is especially important for infinite sources; whereas laziness is merely an optimization for finite sources, it makes operations on infinite sources possible, whereas an eager approach would never terminate.)

Operations such as filtering and mapping can be thought of as naturally lazy, whether or not they are implemented as such. On the other hand, value-producing operations such as `sum()`, or side-effect-producing operations such as `forEach()`, are "naturally eager", because they must produce a concrete result.

In a pipeline such as:

```
int sum = shapes.stream()
                .filter(s -> s.getColor() == BLUE)
                .mapToInt(s -> s.getWeight())
                .sum();
```

the filtering and mapping operations are lazy. This means that we don't start drawing elements from the source until we start the `sum` operation, and when we do perform the `sum` operation, we fuse filtering, mapping, and addition into a single pass on the data. This minimizes the bookkeeping costs required to manage intermediate elements.

Many loops can be restated as aggregate operations drawing from a data source (array, collection, generator function, I/O channel), doing a series of lazy operations (filtering, mapping, etc), and then doing a single eager operation (`forEach`, `toArray`, `collect`, etc) -- such as filter-map-accumulate, filter-map-sort-foreach, etc. The naturally lazy operations tend to be used to compute temporary intermediate results, and we exploit this property in our API design. Rather than have the `filter` and `map` return a collection, we instead have them return a new stream. In the Streams API, operations that return a stream are lazy, and operations that return a non-stream result (or return no result, such as `forEach()`) are eager. In most cases where potentially-lazy operations are being applied to aggregates, this turns out to be exactly what we want -- each stage takes a stream of input values, performs some transformation on it, and passes the values to the next stage in the pipeline.

Conveniently, when used in a source-lazy-lazy-eager pipeline, the laziness is mostly invisible, as the computation is "sandwiched" with a source at one end (often a collection), and an operation that produces the desired result (or side-effect) at the other end. This turns out to yield good usability and performance in an API with a relatively small surface area.

Methods like `anyMatch(Predicate)` or `findFirst()`, while eager, can use *short-circuiting* to stop processing once they can determine the final result. Given a pipeline like:

```
Optional<Shape> firstBlue = shapes.stream()
                                  .filter(s -> s.getColor() == BLUE)
                                  .findFirst();
```

Because the filter step is lazy, the `findFirst` implementation will only draw from upstream until it gets an element, which means we need only apply the predicate to input elements until we find one for which the predicate is true, rather than all of them. The `findFirst()` method returns an `Optional`, since there might not be any elements matching the desired criteria. `Optional` provides a means to describe a value that might or might not be present.

Note that the user didn't have to ask for laziness, or even think about it very much; the right thing happened, with the library arranging for as little computation as it could.

Parallelism

Stream pipelines can execute either in serial or parallel; this choice is a property of the stream. Unless you explicitly ask for a parallel stream, the JDK implementations always return sequential streams (a sequential stream may be converted into a parallel one with the `parallel()` method.)

While parallelism is always explicit, it need not be intrusive. Our sum-of-weights example can be executed in parallel simply by invoking the `parallelStream()` method on the source collection instead of `stream()`:

```
int sum = shapes.parallelStream()
                .filter(s -> s.getColor() == BLUE)
                .mapToInt(s -> s.getWeight())
                .sum();
```

The result is that the serial and parallel expressions of the same computation look similar, but parallel executions are still clearly identified as parallel (without the parallel machinery overwhelming the code).

Because the stream source might be a mutable collection, there is the possibility for interference if the source is modified while it is being traversed. The stream operations are intended to be used while the underlying source is held constant for the duration of the operation. This condition is generally easy to maintain; if the collection is confined to the current thread, simply ensure that the lambda expressions passed to stream operations do not mutate the stream source. (This condition is not substantially different from the restrictions on iterating Collections today; if a Collection is modified while being iterated, most implementations throw **`ConcurrentModificationException`**.) We refer to this requirement as *non-interference*.

It is best to avoid any side-effects in the lambdas passed to stream methods. While some side-effects, such as debugging statements that print out values are usually safe, accessing mutable state from these lambdas can cause data races or surprising behavior since lambdas may be executed from many threads simultaneously, and may not see elements in their natural encounter order. Non-interference includes not only not interfering with the source, but not interfering with other lambdas; this sort of interference can arise when one lambda modifies mutable state and another lambda reads it.

As long as the non-interference requirement is satisfied, we can execute parallel operations safely and with predictable results even on non-thread-safe sources such as **`ArrayList`**.

Examples

Below is an fragment from the JDK class **`Class`** (the **`getEnclosingMethod`** method), which loops over all declared methods, matching method name, return type, and number and type of parameters. Here is the original code:

```
for (Method m : enclosingInfo.getEnclosingClass().getDeclaredMethods()) {
    if (m.getName().equals(enclosingInfo.getName()) ) {
        Class<?>[] candidateParamClasses = m.getParameterTypes();
        if (candidateParamClasses.length == parameterClasses.length) {
            boolean matches = true;
            for(int i = 0; i < candidateParamClasses.length; i++) {
                if (!candidateParamClasses[i].equals(parameterClasses[i])) {
                    matches = false;
                    break;
                }
            }

            if (matches) { // finally, check return type
                if (m.getReturnType().equals(returnType) )
                    return m;
            }
        }
    }
}

throw new InternalError("Enclosing method not found");
```

Using streams, we can eliminate all the temporary variables and move the control logic into the library. We fetch the list of methods via reflection, turn it into a **`Stream`** with **`Arrays.stream`**, and then use a series of filters to reject the ones that don't match name, parameter types, or return type. The result of **`findFirst`** is an **`Optional<Method>`**, and we then either fetch and return the resulting method or throw an exception.

```
return Arrays.stream(enclosingInfo.getEnclosingClass().getDeclaredMethods())
             .filter(m -> Objects.equals(m.getName(), enclosingInfo.getName())
             .filter(m ->  Arrays.equals(m.getParameterTypes(), parameterClasses))
             .filter(m -> Objects.equals(m.getReturnType(), returnType))
             .findFirst()
             .orElseThrow(() -> new InternalError("Enclosing method not found");
```

This version of the code is more compact, more readable, and less error-prone.

Stream operations are very effective for ad-hoc queries over collections. Consider a hypothetical "music library" application, where a library has a list of albums, an album has a title and a list of tracks, and a track has a name, artist, and rating.

Consider the query "find the names of albums that have at least one track rated four or higher, sorted by name." To construct this set, we might write:

```
List<Album> favs = new ArrayList<>();
```

```
for (Album a : albums) {
    boolean hasFavorite = false;
    for (Track t : a.tracks) {
        if (t.rating >= 4) {
            hasFavorite = true;
            break;
        }
    }
    if (hasFavorite)
        favs.add(a);
}
Collections.sort(favs, new Comparator<Album>() {
                           public int compare(Album a1, Album a2) {
                               return a1.name.compareTo(a2.name);
                           }});
```

We can use the stream operations to simplify each of the three major steps -- identification of whether any track in an album has a rating of at least for (`anyMatch`), the sorting, and the collection of albums matching our criteria into a `List`:

```
List<Album> sortedFavs =
  albums.stream()
        .filter(a -> a.tracks.anyMatch(t -> (t.rating >= 4)))
        .sorted(Comparator.comparing(a -> a.name))
        .collect(Collectors.toList());
```

The `Comparator.comparing()` method takes a function that extracts a `Comparable` sort key, and returns a `Comparator` that compares on that key (see section "Comparator factories", below.)

Collectors

In the examples so far, we've used the `collect()` method to gather the elements of a stream into a `List` or `Set`. The argument to `collect()` is a `Collector`, which embodies a recipe for folding elements into a data structure or summary. The `Collectors` class contains factories for many common collectors; `toList()` and `toSet()` are among the most commonly used, but there are many more that can be used to perform sophisticated transforms on the data.

A `Collector` is parameterized by its input and output types. The `toList()` collector has an input type of some `T` and an output type of `List<T>`. A slightly more complicated `Collector` is `toMap`, of which there are several versions. The simplest version takes a pair of functions, one which maps input elements to map keys, and the other to map values. It takes a `T` as input and produces a `Map<K,V>`, where `K` and `V` are the result types of the key and value mapping functions. (More complex versions allow you to customize the type of the resulting map, or to resolve duplicates when multiple elements map to the same key.) For example, to create a reverse index on a known unique key such as catalog number:

```
Map<Integer, Album> albumsByCatalogNumber =
    albums.stream()
          .collect(Collectors.toMap(a -> a.getCatalogNumber(), a -> a));
```

Related to `toMap` is `groupingBy`. Let's say we wanted to tabulate our favorite tracks by artist. We want a `Collector` that takes as input `Track` and produces a `Map<Artist,List<Track>>`. This exactly matches the behavior of the simplest form of the `groupingBy` collector, which takes a classification function and produces a map keyed by that function, whose corresponding values are a list of input elements who correspond to that key.

```
Map<Artist, List<Track>> favsByArtist =
    tracks.stream()
          .filter(t -> t.rating >= 4)
          .collect(Collectors.groupingBy(t -> t.artist));
```

Collectors can be composed and reused to produce more complex collectors. The simple form of the `groupingBy` collector organized elements into buckets according to the classification function (here, the track's artist), and put all elements that map to the same bucket into a `List`. There is a more general version that lets you use *another* collector to organize the elements within a bucket; this version takes a classifying function and a downstream collector as arguments, and all elements mapped into the same bucket by the classifying function are passed to the downstream collector. (The one-argument version of `groupingBy` implicitly uses `toList()` as its downstream collector.) For example, if we want to collect the tracks associated with each artist into a `Set` instead of a `List`, we could combine this with the `toSet()` collector:

```
Map<Artist, Set<Track>> favsByArtist =
    tracks.stream()
          .filter(t -> t.rating >= 4)
          .collect(Collectors.groupingBy(t -> t.artist,
                                         Collectors.toSet()));
```

If we wanted to categorize tracks by artist and rating to create a multi-level map, we could do:

```
Map<Artist, Map<Integer, List<Track>>> byArtistAndRating =
    tracks.stream()
          .collect(groupingBy(t -> t.artist,
                              groupingBy(t -> t.rating)));
```

As a final example, let's say we wanted to create a frequency distribution of words that appear in track titles. We first use **`Stream.flatMap()`** and **`Pattern.splitAsStream`** to take a stream of tracks and explode each track into the words in that track's name, producing a stream of words in all the names of all the tracks. We can then use **`groupingBy`** using **`String.toUpperCase`** as the classifier function (so all words that are the same word, ignoring case, are considered the same and therefore appear in the same bucket) and use the **`counting()`** collector as the downstream collector to count the appearances of each word (without having to create an intermediate collection):

```
Pattern pattern = Pattern.compile(\\s+");
Map<String, Integer> wordFreq =
    tracks.stream()
          .flatMap(t -> pattern.splitAsStream(t.name)) // Stream<String>
          .collect(groupingBy(s -> s.toUpperCase(),
                              counting()));
```

The **`flatMap`** method takes as its argument a function that maps an input element (here, a track) to a stream of something (here, words in the track name). It applies this mapping function to every element of the stream, replacing each element with the contents of the resulting stream. (Think of this as two operations, first mapping every element to a stream of zero or more other elements, and then flattening out the contents of the resulting streams into a single stream.) So here, the result of the **`flatMap`** operation is a stream containing all the words in all the track names. We then group the words together into buckets containing occurrences of words which are identical modulo case, and use the **`counting()`** collector to count the number of words in each bucket.

The **`Collectors`** class has lots of methods for constructing collectors that can be used for all sorts of common queries, roll-ups, and tabulations, and you can implement your own **`Collector`** as well.

Parallelism under the hood

With the Fork/Join framework added in Java SE 7, the JDK has an API for efficiently implementing parallel computations. However, parallel code with Fork/Join looks very different from (and much bigger than) the equivalent serial code, which acts as a barrier to parallelization. By supporting the exact same set of operations on sequential and parallel streams, users can switch between serial and parallel execution without rewriting their code, removing this barrier and making parallelism more accessible and less error-prone.

The steps involved in implementing a parallel computation via recursive decomposition are: dividing a problem into subproblems, solving a subproblem sequentially to produce a partial result, and combining the partial results of two subproblems. The Fork/Join machinery is designed to automate this process.

In order to support the full set of operations on any stream source, we model the stream source with an abstraction called **`Spliterator`**, which is a generalization of a traditional iterator. In addition to supporting sequential access to the data elements, a spliterator also supports decomposition: just as an **`Iterator`** lets you carve off a single element and leave the rest described by the **`Iterator`**, a **`Spliterator`** lets you carve off a larger chunk (ideally, half) of the input elements into a new **`Spliterator`**, and leave the rest of the data to be described by the original **`Spliterator`**. (Both spliterators can then be decomposed further.) Additionally, a spliterator can provide source metadata such as the number of elements (if known) and a set of boolean characteristics (such as "the elements are sorted") that can be used by the Streams framework to optimize execution.

This approach separates the structural properties of recursive decomposition from the algorithms that can be executed in parallel on decomposible data structures. The author of a data structure need only provide the decomposition logic, and then immediately gets the benefit of parallel execution of stream operations.

Most users won't ever have to implement a **`Spliterator`**; they'll just use the **`stream()`** methods on existing collections. But, if you ever are implementing a collection or other stream source, you might want to consider providing a custom **`Spliterator`**. The API for **`Spliterator`** is shown below:

```
public interface Spliterator<T> {
    // Element access
    boolean tryAdvance(Consumer<? super T> action);
    void forEachRemaining(Consumer<? super T> action);

    // Decomposition
    Spliterator<T> trySplit();

    // Optional metadata
    long estimateSize();
    int characteristics();
    Comparator<? super T> getComparator();
```

```
}
```

Base interfaces such as **`Iterable`** and **`Collection`** provide correct but low-performance **`spliterator()`** implementations, but sub-interfaces (like **`Set`**) and concrete implementations (like **`ArrayList`**) override these with higher-quality spliterators that take advantage of information not available to the base type. The quality of a spliterator implementation will affect performance of stream execution; returning well-balanced splits from the **`split()`** method will improve CPU utilization, and providing the correct characteristics and size metadata will enable many other optimizations.

Encounter order

Many data sources, such as lists, arrays, and I/O channels, have a natural *encounter order*, which means the order in which the elements appear has significance. Others, such as **`HashSet`**, have no defined encounter order (and therefore an **`Iterator`** for a **`HashSet`** is permitted to serve up the elements in any order it likes.)

One of the characteristics tracked by **`Spliterator`**, and used by stream implementations, is whether the stream has a defined encounter order. With a few exceptions (such as **`Stream.forEach()`** or **`Stream.findAny()`**), parallel operations are constrained by encounter order. This means that in a stream pipeline like

```
List<String> names = people.parallelStream()
                           .map(Person::getName)
                           .collect(toList());
```

the names must appear in the same order as the corresponding people did in the stream source. Usually, this is what we want, and for many stream operations, this is not prohibitively expensive to preserve. On the other hand, if the source were a **`HashSet`**, then the names could appear in any order, and might appear in a different order across multiple executions.

Streams and lambdas in the JDK

Having exposed **`Stream`** as a top-level abstraction, we want to ensure that the features of **`Stream`** are available as widely throughout the JDK as possible. **`Collection`** has been augmented with **`stream()`** and **`parallelStream()`** methods for converting collections into streams; arrays can be converted into streams with **`Arrays.stream()`**.

Additionally, there are static factory methods in **`Stream`** (and the associated primitive specializations) for creating streams, such as **`Stream.of`**, **`Stream.generate`**, and **`IntStream.range`**. Many other classes have acquired new stream-bearing methods, such as **`String.chars`**, **`BufferedReader.lines`**, **`Pattern.splitAsStream`**, **`Random.ints`**, and **`BitSet.stream`**.

Finally, we provide a set of APIs for constructing streams, to be used by library writers who wish to expose stream functionality on non-standard aggregates. The minimal information needed to create a **`Stream`** is an **`Iterator`**, but if the creator has additional metadata (such as knowing the size), the library can provide a more efficient implementation by implementing a **`Spliterator`** (as all of the JDK collections have).

Comparator factories

The **`Comparator`** class has acquired a number of new methods that are useful for building comparators.

The static method **`Comparator.comparing()`** takes a function that extracts a **`Comparable`** sort key and produces a **`Comparator`**. Its implementation is very simple:

```
public static <T, U extends Comparable<? super U>> Comparator<T> comparing(
        Function<? super T, ? extends U> keyExtractor) {
    return (c1, c2)
        -> keyExtractor.apply(c1).compareTo(keyExtractor.apply(c2));
}
```

Methods like this are an example of *higher order functions* -- functions who take as arguments functions or return new functions. Methods like this simplify user code by reducing duplication:

```
List<Person> people = ...
people.sort(comparing(p -> p.getLastName()));
```

This is much cleaner than the "old way", which usually involved an anonymous class instance that implemented **`Comparator`**. But the real power of this approach is its improved composability. For example, **`Comparator`** has a default method for reversing its direction. So, to sort the people by last name in reverse order, we can simply create the comparator as before, and then ask it to reverse itself:

```
people.sort(comparing(p -> p.getLastName()).reversed());
```

Similarly, the default method **`thenComparing`** allows you to take a **`Comparator`** and refine its behavior when the initial comparator views two elements as equal. To sort the people by last name then first name, we would do:

```
Comparator<Person> c = Comparator.comparing(p -> p.getLastName())
                                 .thenComparing(p -> p.getFirstName());
people.sort(c);
```

Mutative collection operations

Stream operations on collections produce a new value, collection, or side-effect. However, sometimes we do want to mutate the collection in-place, and some new methods have been added to **`Collection`**, **`List`**, and **`Map`** to take advantage of lambdas, such as **`Iterable.forEach(Consumer)`**, **`Collection.removeAll(Predicate)`**, **`List.replaceAll(UnaryOperator)`**, **`List.sort(Comparator)`**, and **`Map.computeIfAbsent()`**. Additionally, non-atomic versions of the methods from **`ConcurrentMap`**, such as **`replace`** and **`putIfAbsent`** have been pulled up into **`Map`**.

Summary

While adding lambda expressions to the language is a huge step forward, developers get their work done every day by using the core libraries, so the language evolution effort was paired with a library evolution effort so that users could start using the new features on day one. The centerpiece of the new library features is the **`Stream`** abstraction, which provides powerful facilities for aggregate operations on data sets, and has been deeply integrated with the existing collection classes as well as other JDK classes.