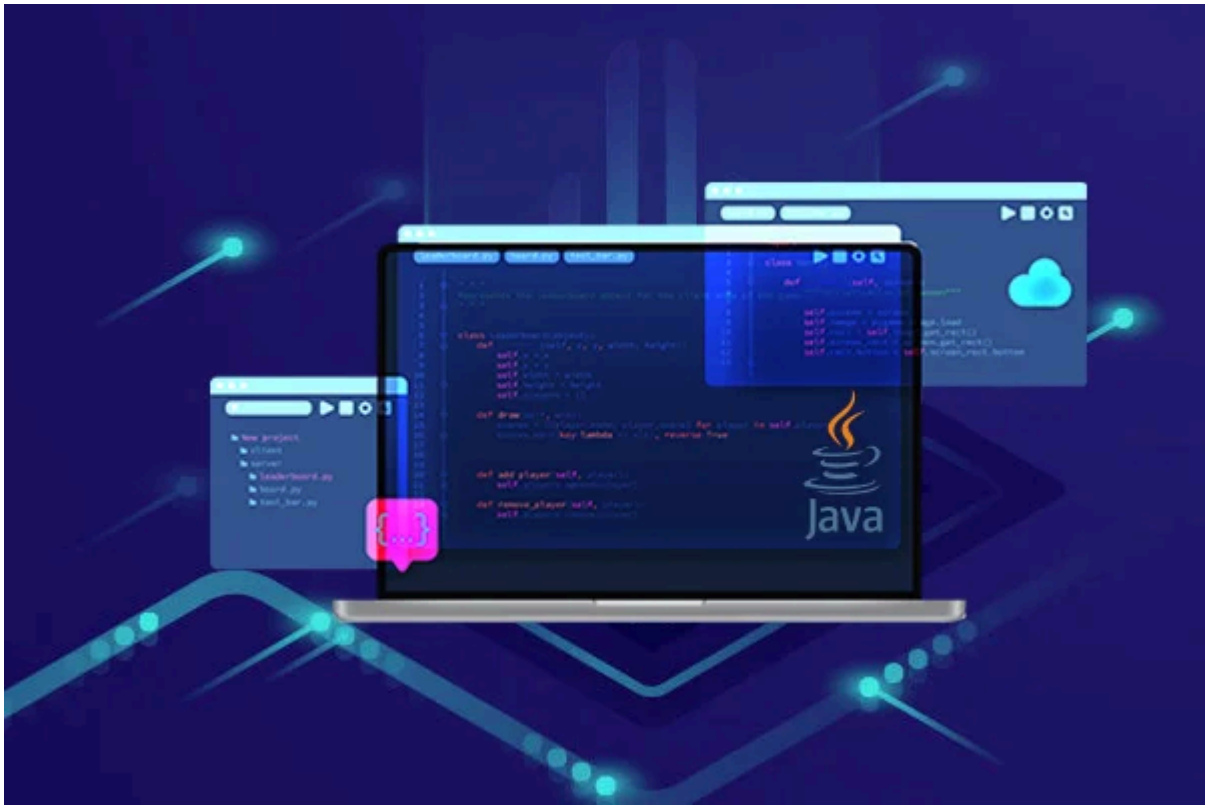

September 24, 2024

Using Java Streams in Java 8 and Beyond

JAVA TOOLS | JAVA APPLICATION DEVELOPMENT

Java 8 has been out since 2014, but many of the original features have remained relevant today. One of the most prominent of these is Java streams. In this blog, we answer what are Java streams, explain when to use them, troubleshoot Java streams issues, and give you a brief rundown of common Java streams operations.

Table of Contents

Table of Contents

Back to top

What is a Java Stream?

> A Java stream is a pipeline of functions that can be evaluated. Java streams are not a data structure and cannot mutate data; they can only transform data.

Java streams enable functional-style operations on streams of elements. A stream is an abstraction of a non-mutable collection of functions applied in some order to the data. A stream is not a collection where you can store elements.

The most important difference between a stream and a structure is that a Java stream doesn't hold the data. For example, you cannot point to a location in the stream where a certain element exists. You can only specify the functions that operate on that data. And when performing operations on a stream, it will affect the original stream.

As a note, the streams in this blog are not to be confused with streams in Java I/O package like InputStream, OutputStream, etc.

Want to dive deeper? Check out our Java Streams Cheat Sheet (https://www.jrebel.com/blog/java-streams-cheat-sheet)

Back to top

Are Java Streams Still Relevant in 2024?

While Java streams first debuted in Java 8, they are still relevant in 2024 for several reasons, including conciseness, parallel processing, and efficiency. Specifically, parallel processing has enabled microservices and remote development environments (which primarily rely upon microservices environments).

Conciseness: Streams are more consise and readable than Java alternatives
Parallel Processing: You are able to take advantage of multi-core processors and expedite processing tasks
Efficient: Highly performant option that only evaluates when computed (aka lazy evaluation)

Back to top

When to Use Java Streams

Java streams represent a pipeline through which data will flow and the functions to operate on the data. A pipeline in this instance consists of a stream source, followed by zero or more intermediate operations, and a terminal operation.

As such, streams can be used in any number of applications that involve data-driven functions. In the example below, the Java stream is used as a fancy iterator:

```
List numbers = Arrays.asList(1, 2, 3, 4);
List result = numbers.stream()
.filter(e -> (e % 2) == 0)
.map(e -> e * 2)
.collect(toList());
```

In this example we select only even values, by using the *filter* method and doubled them by *mapping* the function that doubles the input. What does this provide us? The streams API gives us the power to specify a sequence of operations on the data in individual steps. We don't specify any conditional processing code, we are not tempted to write large complex functions, we don't care about the data flow.

In fact, we only bother ourselves with one data processing step at a time: we compose the functions and the data flows through the functions by itself by the power of the streams framework. The example above shows one of the most important pattern you'll end up using with the streams:

- Raise a collection to a stream.
- Ride the stream: filter values, transform values, limit the output.
- Compose small individual operations.
- Collect the result back into a concrete collection.

Back to top

Common Operations in Java Streams

In Java 8 and up, you can easily obtain a stream from any collection by calling the stream() method. Beyond that simple starting point, there are a couple of fundamental functions that you'll encounter all the time.

Here are some common operations in Java streams. You won't use all of these functions every time you encounter a stream, but you have them available to use at will:

- **Filter:** Returns a new stream that contains some of the elements of the original. It accepts the predicate to compute which elements should be returned in the new stream and removes the rest. In the imperative code we would employ the conditional logic to specify what should happen *if* an element satisfies the condition. In the functional style we don't bother with *ifs*, we filter the stream and work only on the values we require.
- **Map:** Transforms the stream elements into something else, it accepts a function to apply to each and every element of the stream and returns a stream of the values the parameter function produced. This is the bread and butter of the Java streaming API. Map allows you to perform a computation on the data inside a stream.
- **Reduce:** (also sometimes called a fold) Performs a reduction of the stream to a single element. If you want to sum all the integer values in the stream, you want to use the reduce function. If you want to find the maximum in the stream, reduce is your friend.
- **Collect:** This is the way to get out of the streams world and obtain a concrete collection of values, like a list in the example above.

Function	Preserves count	Preserves type	Preserves order
map	✅	❌	✅
filter	❌	✅	✅
distinct	❌	✅	✅
sorted	✅	✅	❌
peek	✅	✅	✅

Back to top

Java Stream Examples and Potential Issues to Avoid

There are some caveats of using the Java streaming API though, and sometimes stream processing can get out of hand. Imagine we have the follow class Person:

```
class Person {
Gender gender; String name;
public Gender getGender() { return gender; }
public String getName() { return name; }
}
enum Gender { MALE, FEMALE, OTHER }
```

This is a typical Java bean with some getters on the fields. Now, suppose we have a list of these persons and want to get the list of uppercase names of all the “FEMALE” people in that list. Sounds easy, right?

```
List names = new ArrayList();
List people = …
people.stream()
.filter(p -> p.getGender() == Gender.FEMALE)
.map(Person::getName)
.map(String::toUpperCase)
.forEach(name -> names.add(name));
```

We follow the specification of what we must do at every step, but the problem is in the mutation of the shared state. We know nothing of the nature of the stream and if the stream is parallel, the concurrent addition of the elements into the stream can lead to errors.

Another gotcha to avoid: If only an intermediate operation is specified and the terminal operation is inadvertently missed out, then nothing is displayed in the output. This is because intermediate operations are executed only when a terminal operation is present.

For example, the following snippet does not print any output:

```
Stream.of("Cathy","Alba","Beth")
.filter(s -> {
System.out.println("filter: " + s);
return true;
});
```

The ordering of output is somewhat surprising as each element passes through all the operations (unlike what some programmers may assume - that all elements are first passed through filter () and then forEach()). For example:

```
Stream.of("Cathy", "Alba", "Beth").
filter(s -> {
System.out.println("filter: " + s);
return true;
})
.forEach(s -> System.out.println("forEach: " + s));
```

The output is:

```
filter: Cathy
forEach: Cathy
filter: Alba
forEach: Alba
filter: Beth
forEach: Beth
```

Using Java Stream Collect to Avoid Concurrency Errors

Instead, we should have *collected* the stream into the resulting list, so that concurrency and mutability is the responsibility of the streams framework. Here's the example of how to do so:

```
List people = …
List names = people.stream()
.filter(p -> p.getGender() == Gender.FEMALE)
.map(Person::getName)
.map(String::toUpperCase)
.collect(Collectors.toList());
```

In general, the Collectors class provides almost all necessary primitives to transform a stream into a concrete collection. One example is the *toMap()* collector. You might be confused about how can an element be transformed into a key-value pair required for the map.

To do this, you specify a function that turns the element into the key and another function that creates the value. Here's an example that collects the same stream of people into a map:

```
List people = …
Map<String, Person> names = people.stream()
.collect(Collectors.toMap(p -> p.getName(), p -> p));
```

The first function given to the *toMap* method transforms the element into the key and the second to the value for the map.

Back to top

Java Stream Intermediate and Terminal Operations

One of the virtues of Java streams is that they are lazily evaluated. Some operations on the streams, particularly the functions that return an instance of the stream: filter, map, are called **intermediate**. This means they aren't evaluated when they are specified. This means that they won't be evaluated when they are specified. Instead, the computation will happen when the result of that operation is necessary.

This means that if we just specify the code like:

```
Stream names = people.stream()
.filter(p -> p.getGender() == Gender.FEMALE)
.map(Person::getName)
.map(String::toUpperCase);
```

None of the names will be immediately collected and made into the upper case. So, when does the computation occur? When a **terminal** operation is called.

All operations that return something other than a stream are terminal. Operations like forEach, collect, reduce are terminal. This makes streams particularly efficient at handling large amounts of data.

From a performance perspective, the ordering of intermediate operations is critical.

For example, if map() is specified before filter() then map() will be called multiple times. However, if filter() is specified before map() then map() is called only once, leading to high performance.

```
Stream.of("Cathy", "Alba", "Beth")
.map(s -> {
System.out.println("map: " + s);
return s.toUpperCase();
})
.filter(s -> {
System.out.println("filter: " + s);
return s.startsWith("A");
})
.forEach(s -> System.out.println("forEach: " + s));
```

Here map() is called multiple times, which is sub-optimal. However, in this example, map() is called only once, leading to improved performance.

```
Stream.of("Cathy", "Alba", "Beth")
.filter(s -> {
System.out.println("filter: " + s);
return s.startsWith("A");
})
.map(s -> {
System.out.println("map: " + s);
return s.toUpperCase();
})
.forEach(s -> System.out.println("forEach: " + s));
```

On top of that, one can almost always try to parallelize the stream processing by converting the stream into a parallel stream by calling the *parallel()* method. Note, that although the stream doesn't have to be parallelizable, the method to parallelize it is always there. So depending on the internal nature of the stream you can get the performance benefits.

Back to top

Final Thoughts on Java Streams API

There are pitfalls of running every stream operation in parallel; most streams implementations use the default ForkJoinPool to perform the operations in background. Thus, you can easily make the particular stream processing a bit faster, but instead sacrifice the performance of the whole JVM without even realizing it.

Solving the problems using functional programming requires a different way of thinking. But with a bit of experimentation, you can get the hang of it.

And often, you may struggle to find a functional solution, but once you get it, you realize that it's not particularly complicated. And then the next time solving a similar problem will be much easier.

JRebel for All Java Versions

Whether you are working in Java 8 or Java 23 (https://www.jrebel.com/blog/whats-new-java-23), JRebel can save you time and boost performance. JRebel helps developers to save time during application development by skipping rebuilds and redeploys while maintaining application state.

Try JRebel for free for 14 days and see how much Java development time your team could save.

TRY JREBEL FOR FREE (https://jrebel.com/software/jrebel/trial/)

This blog was originally published in 2022 and has been edited for relevancy and clarity.

Back to top

SEND FEEDBACK

JRebel by Perforce (https://www.perforce.com)

SEND FEEDBACK