JAVA SE

Quiz yourself: Search stream data using the findFirst, findAny, and anyMatch methods (advanced)

Streams are complicated. With the right approach, they can be very efficient too.

by Simon Roberts and Mikalai Zaikin

August 28, 2020

If you have worked on our quiz questions in the past, you know none of them is easy. They model the difficult questions from certification examinations. The “intermediate” and “advanced” designations refer to the exams rather than to the questions, although in almost all cases, “advanced” questions will be harder. We write questions for the certification exams, and we intend that the same rules apply: Take words at their face value and trust that the questions are not intended to deceive you but to straightforwardly test your knowledge of the ins and outs of the language.

Given the following code:

```
List<String> src = List.of("Java 11", "Exam");
```

Which code fragment determines if the word “Java” is present in the `src` list in the most computationally efficient way? Choose one.

A.

```
List<String> res = List.of();
src.stream()
   .peek(v -> { if (v.contains("Java")) res.add(v);
   .count();
var a = (res.size() > 0);
```

The answer is A.

B.

```
var a = src.stream().filter(v -> v.contains("Java"))
```

The answer is B.

C.

```
var a = src.stream().anyMatch(v ->
v.contains("Java"));
```

The answer is C.

D.

```
var a = src.stream()
          .filter(v -> v.contains("Java"))
          .collect(Collectors.toList());
```

The answer is D.

Answer. All the code fragments shown are syntactically valid and compile successfully, and all but one are functionally correct. Thus, the task is to analyze what fragments are correct, and of those, which is the most efficient.

Examining the code in option A, it seems that the approach is to pull the data through a stream pipeline, use `peek` to add any items that contain the word “Java” to a second list, and then count the items in that resulting list.

It’s likely that your instinct says this seems overly complex, but instinct isn’t the best way to answer questions. However, there are indeed several problems with this code.

One problem is that the `List.of` method creates an immutable object. Therefore, if the code tried to add to the list `res`, it would throw an exception. From this, you can determine that option A cannot work and must be incorrect.

Interestingly, however, the `count()` terminal operation has some potential optimizations noted in the documentation:, as follows:

> An implementation may choose to not execute the stream pipeline (either sequentially or in parallel) if it is capable of computing the count directly from the stream source. In such cases no source elements will be traversed and no intermediate operations will be evaluated. Behavioral parameters with side-effects, which are strongly discouraged except for harmless cases such as debugging, may be affected.

In this case, the implementation of `count` in most regular JDKs results in the `count` method immediately returning the size of the list as 2 without ever pulling any data through the pipeline. As a consequence, it’s likely that the `res` list will, in fact, be empty, and the code will assert, wrongly, that the word “Java” never appears. But as already described, if this optimization is not implemented on a given JVM, the code throws a runtime exception and fails anyway.

Another stylistic problem with the code of option A is that the stream pipeline uses visible side effects. That is, it modifies the externally declared `res` list in the pipeline processing. Side effects like this are incompatible with the precepts of functional style. Perhaps more importantly, if a stream such as this worked reliably in sequential mode, it would almost certainly fail unpredictably if it were run in parallel mode, because of uncontrolled concurrent access to the list.

The remaining options are all functionally correct, but let’s investigate the operation of each to determine how they work and which is most

computationally efficient.

Option B builds a two-stage pipeline on the stream. The first stage, the `filter` operation, drops any items that do not contain the word "Java," and the second stage (which is the `findAny` "short-circuiting" terminal operation) determines if the stream contains any items at all, returning a nonempty `Optional<String>` as soon as a value is found or an empty `Optional` if the stream is exhausted without finding anything.

To finally determine if the word "Java" was found in the stream, you'd need to test the `Optional`, probably using the method `isPresent`. This adds one more step to the overall efficiency question. So, the logic and implementation of this option are sound; it uses a two-stage pipeline that will process only one stream item before producing its affirmative result packaged as the `Optional`, which must be tested to determine the required information.

Let's continue the investigation to see if one of the other options might offer a better solution.

Option C uses the `anyMatch` short-circuiting terminal operation, so like option B, it stops as soon as the final result can be determined. Rather than using two pipeline stages, it simply pulls data through the pipe and tests to see if each item satisfies the required criterion in a single stage. It returns a simple Boolean value `true` immediately if such an item is found; it returns `false` if the stream is exhausted without a match. This is exactly what is required and improves on option B in several ways: The pipeline is one stage, rather than two, and it produces a Boolean result directly without requiring subsequent testing. Thus, option C is clearly better than option B. Therefore, option B is incorrect.

Option D filters the original stream looking for elements that contain the word "Java" and then collects the resulting values into a `List`. This is inefficient compared to options B and C because it must process the entire list rather than stopping as soon as a result is known. It's preferable to option A in that it does not fail as a result of the optimization of the `count` method and it does not it throw exceptions. It also avoids the use of side effects, and so it would work in a parallel execution mode. But, although it answers the question from a functional perspective, it is nowhere near as efficient; it processes the entire stream and you must test the size of the resulting list (presumably by testing `a.size() > 0`) before you actually know the answer. Therefore, option D is incorrect.

Based on the requirements, you can see that of the three options that produce a correct answer, option C is the most efficient (and, perhaps more importantly in professional programming, it's arguably the most readable):

- It provides an answer directly as a Boolean, rather than requiring subsequent testing of an intermediate value.
- It benefits from short-circuit behavior and will stop processing the source as soon as the answer can be definitely known.

Therefore, the correct option is C.

Simon Roberts

Simon Roberts joined Sun Microsystems in time to teach Sun's first Java classes in the UK. He created the Sun Certified Java Programmer and Sun Certified Java Developer exams. He wrote several Java certification guides and is currently a freelance educator who publishes recorded and live video

training through Pearson InformIT (available direct and through the O'Reilly Safari Books Online service). He remains involved with Oracle's Java certification projects.

Mikalai Zaikin

Mikalai Zaikin is a lead Java developer at IBA IT Park in Minsk, Belarus. During his career, he has helped Oracle with development of Java certification exams, and he has been a technical reviewer of several Java certification books, including three editions of the famous *Sun Certified Programmer for Java* study guides by Kathy Sierra and Bert Bates.

Share this Page