Java Magazine

Java SE, Quiz

Quiz yourself: Streams and flatMap operations in Java

Mikalai Zaikin

Simon Roberts

November 8, 2021

Text Size 100%:

The powerful peek() function can be tricky to use correctly in Java streams.

Given the code fragment

```
String[][] arr = {{"a", "d"}, {"n", "d"}, {"a", "x"}};
Arrays.stream(arr)
  .peek(v -> v.equals(new String[] {"a", "d"}))
  .flatMap(u -> Arrays.stream(u))
  .forEach(System.out::print);
```

Copy code snippet

What is the output? Choose one.

A. `ad` The answer is A.

B. `[a,d]` The answer is B.

C. `anaddx` The answer is C.

D. `adndax` The answer is D.

E. `[[a,d][n,d][a,x]]` The answer is E.

F. Compilation fails due to the argument to `peek`. The answer is F.

Answer. This question investigates the behavior of arrays and the `flatMap` operation in streams. Notice first that the array `arr` is an array containing arrays of `String`s. When an array is passed to `Arrays.stream`, the resulting stream will contain the elements of that array in the order of low to high index values. So, the initial stream contains *three* elements, `{"a", "d"}`, `{"n", "d"}`, and `{"a", "x"}` in left-to-right order.

The `peek` method itself *never* alters the stream, although it's possible for the argument to do so if it includes a side effect. In this question case, the argument has no side effects and does not modify anything in the stream. In fact, the argument does nothing useful whatsoever. Don't confuse `peek` with a `filter` operation merely because the argument is a lambda that yields a `Boolean` result. Of course, a `filter` operation would potentially remove some elements, but that's not what is shown here.

Does the lambda passed to `peek` cause a compilation error? The `peek` method requires an argument that is a `Consumer` of the stream element type. The lambda provided as argument is correctly formed but appears to return a `Boolean` result from the `equals` method. Incidentally, that return will always be `false`. However, it is acceptable to return a value in this way in a void-compatible lambda; the returned value is simply ignored. This is closely parallel to calling the `add` method on a `list` object but ignoring the `Boolean` value returned by that method. Thus, option F is incorrect because this code does not cause a compilation error.

The `flatMap` invocation has a lambda expression that expands the `String[]` elements to `Stream<String>` elements. The effect of a `flatMap` is to take the elements of each stream that the argument lambda returns and concatenate them into a single `Stream<String>`. Again, the substreams returned from the `Arrays.stream` invocations will be processed from low to high index values. Consequently, the result will be the sequence `adndax`. From this, you can see that option D is correct, and the remaining options—A, B, C, and E—are incorrect.

Conclusion. The correct answer is option D.

Related quizzes

Quiz yourself: Use Java streams to filter, transform, and process data

Quiz yourself: Reductions with Java streams using collectors

Quiz yourself: HashSet and TreeSet sources in Java streams

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.

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.

Previous Post

Vector math made easy: John Rose and Paul Sandoz on Java’s Vector API

Justine Kavanaugh-Brown | 9 min read