Quiz Yourself: Lambda Expressions (Advanced)

The subtleties of using var in lambda expressions

by Simon Roberts and Mikalai Zaikin

February 27, 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.

The objective here is to test your knowledge of creating and using lambda expressions with local variables.

Assume you are writing software that generates PDF reports with user information. To calculate the label length for the document header, you create a lambda expression that takes the username and a title as arguments and returns the length of the text. This lambda expression implements the functional interface `ToIntBiFunction`, as shown here:

```java
ToIntBiFunction<Name, String> l = ... // missing code
```

You are using a framework that invokes this behavior and that uses annotations on method arguments to ensure the validity of the arguments to be supplied. For example, the framework can use this feature to identify missing values (null pointers), out-of-range values, and similar issues. In implementing your lambda expression, you must apply the annotation `@ValidName` to the first parameter (the one of type `Name`).

**Which lambda expressions comply with these requirements?**

Choose two.

A. 
```
@ValidName Name n, String t) ->
n.firstName.length() + n.lastName.length() + t.length()
```

B. 
```
@ValidName var n, String t) ->
n.firstName.length() + n.lastName.length() + t.length()
```
Answer. Java 10 introduced `var` as a reserved type name that can be applied to method local variables. Using this (under specified circumstances), the type of a local variable may be inferred from the expression used to initialize it. The proposal is documented in JEP 286. Java 11 further expanded this facility through JEP 323 so that `var` may be used for lambda expression parameters.

In the original form of lambda expression arguments, there was a choice between providing the full type of every argument or allowing all the arguments to have their type inferred. The type-inferred form might look like this:

```java
(x) -> x.clear();
```

The explicit form might look like the example below. (Note that the type can sometimes be quite complex, which—in situations where explicit knowledge of the type isn't very helpful for understanding the code—is one of the benefits of type inference.)

```java
(Map<String, List<Map.Entry<String, Long>>> x) -> x
```

In the second form, but not the first, it's also possible to add an annotation to the type declaration. (Annotations in this context are applied to the parameter type, not to the parameter.) Therefore, assuming that a `@ValidMap` annotation exists, the example in this question might look like this:

```java
(@ValidMap Map<String, List<Map.Entry<String, Long>>> x) -> x
```

However, the type-inferred form cannot support the annotation, so this would be invalid syntax:

```java
(@ValidMap x) -> x.clear(); // Missing type, does not compile
```

The enhancement of JEP 323 that was brought by Java 11 allows the pseudo-type `var` to be used for lambda expression arguments. This capability provides a brief, and type-inferred, syntax that permits the use of an annotation. So, without an annotation, the code might look like this:

```java
(var x) -> x.clear();
```
But it could also support the annotation and look like this:

```java
(@ValidMap var x) -> x.clear();
```

Let's compare the original form and this last form directly:

```java
(x) -> x.clear();
(@ValidMap var x) -> x.clear();
```

Notice that in both cases, the type of `x` is inferred; the explicit type is not provided and the code has the potential benefit of brevity. However in the first case, no annotation is permitted, whereas in the second case—using the `var` pseudo-type—it is permissible to use an annotation because there is a “type” element in the syntax to attach it to.

Now you know that the annotation can be used on `var` or on a real type, but it cannot be used where neither is present. Because option D attempts to use an annotation with neither a type nor `var`, you can conclude that option D is incorrect.

Of course, there’s more to this question than just that. There is another rule with lambda expressions that has existed since their introduction in Java 8. That rule is that for any one lambda expression, if *any* types are explicitly specified in the argument list, then all the arguments must have explicit types (and, conversely, if you wish to leave out any type, you must leave them all out).

This rule still applies, but it is extended to cover the use of `var`. Specifically, you must choose between all the arguments being given explicit types, all the arguments using `var` to indicate type inferencing, or all the arguments having neither an explicit type nor `var`.

From this rule, you know you cannot mix `var` on one argument with an explicit type on any other argument in a single lambda declaration. This makes options B and E incorrect.

You also know that, in a single lambda declaration, if all arguments use the same type mechanism (that is, they all have an explicit type), or they all use `var`, or they all use “naked” inferencing, then the code will be correct (at least in that respect). You can see that option A and option C are consistent in this way: Option A applies an explicit type to all its arguments, and option C applies `var` to all its arguments. Therefore, you can conclude that option A and option C are correct.

Two side notes might be useful additions to the discussion introduced by this question. The first is that option A was valid in Java 8, whereas option C is valid only as of Java 11. The second is that there is a special case for a lambda that takes a single argument that has no type specification at all. That would be something like this:

```java
(x) -> x.doSomething(); // OK. Regular syntax for lambda
```

In this situation, and this situation alone, it's permitted to omit the parentheses, so this is valid:

```java
x -> x.doSomething(); // OK. Special syntax for single-argument lambda
```
However, this rule does not allow either of these forms:

```java
type x -> x.doSomething(); // NOT OK--Requires parentheses
var x -> x.doSomething();  // NOT OK--Requires parentheses
```

Instead, if you desire to provide a type or `var`, you must use parentheses, like this:

```java
(SomeType x) -> x.doSomething();  // OK
(var x) -> x.doSomething();        // OK
```

The correct answers are options A and 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.