Quiz Yourself: Lambda Types (Advanced)

Can a lambda return a lambda?

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 the proper use of interfaces from the java.util.function package.

Given the following:

```java
public void doIt() {
    /* insert here */ v1 = () -> a -> {};
}
```

Which of the following correctly completes the code when used to entirely replace the comment /* insert here */? Choose one.

A. var
B. Supplier>Consumer>??
C. Function>Consumer>??,??
D. Function>Predicate>??,??
E. Consumer>Function>??,??

This question investigates aspects of lambda expressions, the complexities of nested lambdas, declaring and initializing variables of a functional interface type with lambda expressions, and the nature of functional interfaces in the java.util.function package.

Code with lambda expressions can be hard to understand when the concepts are new, and in some situations it can be hard to understand even if you have considerable experience. One of the tricks that can help
is to break things down into smaller chunks. Every lambda expression consists of an argument list and a method body. Sometimes the method body is simply an expression, and sometimes it's a full method body in curly braces, but either way, it's always present.

So, let's start by ignoring the type declarations listed in the options and simply looking at the expression that's proposed to initialize the variable *v1*. That is the following expression:

```
() -< a -< {}
```

This syntax might be entirely comfortable if you're into Haskell, but since this is a *Java Magazine* article discussing Java syntax in the context of a Java exam, it's safe to assume that for many, it's not comfortable at all.

So, how do you break this down? The first observation is that every lambda has three parts: an argument list, an arrow, and a method body. Let's see where these fit here:

```
() -< a -< {}
```

A reasonable question would be how do you know the second arrow isn’t the arrow of this lambda? There are two reasons. First, you can’t have an arrow in an argument list. Second, and more abstractly, like most Java operators, the arrow groups left to right.

From what you have so far, you know that this whole mess is a lambda expression that takes zero arguments. That by itself tells you a lot about it, but leave that idea on the side for a little longer and look at the body.

In this case, the body is another lambda expression. Actually, however, this is the short form of the body. Two forms of lambda expression exist: One form has curly braces on the right side, and the other form does not. The form that does not is really just a special case of the form that does. Consider this lambda, which uses a block:

```
(a, b) -< { return a + b; }
```

Such a block in a lambda is, in fact, a full method body (and it must conform in every way to those requirements). However, this one is very simple in that all it does is evaluate an expression and return the resulting value. In this situation—specifically, when a method body does nothing more than evaluate an expression and return the result—you're allowed to remove the curly braces, the `return` keyword, and the semicolon associated with the `return` keyword. All that's on the right side is the expression. So the previous example is identical to this shorter form:

```
(a, b) -< a + b
```

However, lambda expressions, in Java at least, are expressions representing instances of objects (they might be implemented differently "under the hood," but they will behave in every way like an object from the source-code perspective). So, whatever the first lambda is, it returns an object. The type of the object is nontrivial, but it's an object that implements some interface.
So, let's break down the returned value:

\[ a \rightarrow \{ \} \]

**Argument list**  
arrow  
**method body**

This time, you can see that the lambda takes exactly one argument, and that the body is in “block lambda” form with curly braces surrounding a full method body—even though that method body is empty. Given that the body is an empty block, you can tell this lambda has a `void` return type.

So, if you revisit the overall expression, you’ll see that you have a lambda that takes zero arguments and returns a lambda that takes one argument and returns `void`.

It’s perhaps worth considering how this would look if the outer lambda also used a block form:

\[
() \rightarrow \{ \text{return } a \rightarrow \{ \}; \}
\]

But if you do the same breakdown here, you get the same elements:

\[
() \rightarrow \{ \text{return } a \rightarrow \{ \}; \}
\]

**Argument list**  
arrow  
**method body**

Which tells you the expression is a lambda that takes zero arguments and returns “something.”

Investigating the “something” that’s returned produces the same result as before:

\[ a \rightarrow \{ \} \]

**Argument list**  
arrow  
**method body**

Which is right where you were before…

Let’s get back to the original problem. What kind of declarations might be acceptable for this complex, nested lambda expression?

Let’s consider option A first. To use the `var` pseudo-type with a variable declaration like this, it must be possible to determine the type of the variable from the expression that is used to initialize that variable (that’s the lambda, in this case). However, of course, a lambda expression determines the functional interface type that it implements from the context to which it is assigned. In other words, to decide the type implied by `var`, you must determine the type of the lambda, but to decide the type of the lambda, you must know the type to which the lambda is being assigned—which is the type of the `var`. This *a is what b is, but b is what a is* circular dependency is not logically resolvable, and so you are not permitted to use the `var` pseudo-type with a lambda expression in the form shown in option A, and so you can determine that option A is incorrect.

For the remaining options, you’ll be looking to see which combination of the core functional interfaces can correctly describe the lambda. You need to consider the type requirements of the four functional interfaces discussed in the question. There are `Supplier`, `Consumer`, `Function`, and `Predicate`. Looking at the API documentation, you can see that
they have the following forms with respect to the abstract method they declare:

```java
public interface Supplier<T> { T get(); }
public interface Consumer<T> { void accept(T t); }
public interface Function<T, R> { R apply(T t); }
public interface Predicate<T> { boolean test(T t); }
```

For a lambda to be valid, the argument list must be consistent with the argument list required by whichever of these interfaces is being implemented by that lambda expression. Additionally, the return type of the lambda expression’s behavior must be conformant too. (The word conformant is intended to allow for the fact that some automatic conversions can happen.)

As a reminder, you’re looking for a type that represents a lambda that takes zero arguments and returns a lambda that takes one argument and returns void. So, without considering the return type, which of the four are compatible with a lambda that takes zero arguments? Only one is, and it’s the Supplier. This immediately indicates that options C, D, and E must be incorrect. Of course, that leaves only option B, but you should determine if that option is fully correct before you stop.

Option B defines a Supplier<Consumer<?>>. It’s been established that the lambda must, and does, take zero arguments. So far so good. Now let’s consider the return type. Option B says that the return type must be a Consumer (strictly speaking, Consumer<?, but this merely means Consumer of something that can be assigned to Object, which means you know nothing about the argument to the Consumer). The returned value in this Supplier is (a) -> {}. This lambda takes one argument and returns void. That’s consistent with Consumer, and from that you can conclude that the code in option B is fully correct and option B is indeed the correct answer.

The correct answer is option B.

---

**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.