JAVA SE

Quiz yourself: The correct syntax of Java lambda parameters

Be sure to know the difference between identifiers and specifiers.

by Mikalai Zaikin and Simon Roberts

March 29, 2021

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. 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.

It's time for a quiz about lambdas. Given the following functional interface:

```
@FunctionalInterface
interface ConverterFunction<T, U, R> {
  R apply(T t, U u);
}
```

And the following class fragment:

```
public class Converter {
  String doConversion(ConverterFunction<Strin
    // … valid implementation here
  }
}
```

Which lambda expression can be used as an inline parameter for the `doConversion` method? Choose one.

A. `(a, Integer b) -> a.repeat(b)` The answer is A.

B. `(var a, var b) -> a.repeat(b)` The answer is B.

C. `(a, var b) -> a.repeat(b)` The answer is C.

D.
```
(a, b, c) -> { c = a.repeat(b);
return c; }
```
The answer is D.

E.
```
(String a, int b) ->
a.repeat(b)
```
The answer is E.

F. None of the above The answer is F.

Answer. This question tests the syntax of lambda parameters, which is documented in *Java Language Specification* section 15.27.1, "Lambda parameters."

The formal parameter list of a lambda can be empty, in which case the lambda has empty parentheses to the left of the arrow or is a comma-separated list. The items in the list may be simple identifiers, or they must be specifiers. The difference is that an *identifier* is a plain variable name, and a *specifier* contains additional information in the manner of a normal method formal parameter.

Like a local variable does, a specifier will contain a type indication, which could be an explicit Java type or might be the pseudotype `var`. A specifier can also carry certain modifiers, such as annotations or the keyword `final`.

In the cases of parameter lists containing identifiers and of those containing the pseudotype `var`, the types of the parameters are inferred.

The next detail is that the specification explicitly prohibits mixing identifiers with specifiers and mixing `var` with explicit types. In other words

- If *any* of the parameters is a simple identifier, then *all* must be simple identifiers.
- If *any* of the parameters uses `var`, then *all* must use `var`.
- If *any* of the parameters uses explicit types, then *all* must use explicit types.

Based on this, you know that both options A and C are incorrect.

Option B presents a lambda with a correctly formed formal parameter list. The number of the parameters is also correct, and the types of `a` and `b` will be inferred to be `String` and `Integer`; thus the invocation of `a.repeat(b)` is valid.

Note that `b` will be automatically unboxed to become an `int` for the argument to the `repeat` method. Finally, the expression `a.repeat(b)` has type `String`, which is the required return type for the lambda. From this you can see that option B is

correct (and by inference, you can determine that option F is incorrect).

In option D, the lambda declares three formal parameters, but the `apply(...)` method declared in the `ConverterFunction` interface requires two formal parameters. This means that the lambda is not a valid implementation of the interface and is not valid as the parameter to the `doConversion` method. From this it's clear that option D is incorrect.

Option E looks promising because the formal parameter list is fully typed and declares the correct number of parameters. Also, the return type of the lambda expression is `String`, which is correct. However, this option also fails because the lambda's second formal parameter is of type `int`. It should be `Integer`.

It's tempting to think that autoboxing would fix this mismatch, but that's not the case. Given that the type-sequence of the parameters differ, this would be an overloading method, not an overriding one. Autoboxing works as the call site but does not turn an overload into an override.

This is likely clearer if you consider the following situation:

```
interface I {
    void doIt(Integer i);
}
class C implements I {
    @Override
    // compilation fails; doIt is overloaded,
    public void doIt(int i) {}
}
```

You can see that option E presents a lambda that cannot be used as the parameter for the `doConversion` method and, therefore, option E is incorrect.

Note: The `repeat(int n)` method might be unfamiliar. This method was added to the `String` class in Java 11 and creates a `String` by repeating the invocation instance `n` times. It's very convenient if you need, for example, to pad some string with spaces to some predefined length.

Conclusion: The correct answer is option B.

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.

Share this Page