Quiz yourself: Lambda expressions

Lambdas are essentially expressions that define a class and instantiate an object from that class. Do you know how to use them?

by Simon Roberts and Mikalai Zaikin

May 10, 2021

The objective of this quiz is to create and use lambda expressions. Given the following code:

```java
@FunctionalInterface public interface SwissKnife {
    default int compare(int i1, int i2) {
        return i1 - i2;
    }
    static void run() {
        System.out.println("Running !");
    }
    String welcomify(String name);
}

SwissKnife sni = (int a, int b) -> a - b;
SwissKnife sni = () -> System.out.print("Running fast !");
SwissKnife sni = (a) -> "Welcome, " + a;
```

Which is the result? Choose one.

A. The following code compiles:

```java
SwissKnife sni = (int a, int b) -> a - b;
```

The answer is A.

B. The following code compiles:

```java
SwissKnife sni = () -> System.out.print("Running fast !");
```

The answer is B.

C. The following code compiles:

```java
SwissKnife sni = (a) -> "Welcome, " + a;
```

The answer is C.

D. Compilation fails at line n1.

The answer is D.

Answer. In the functional programming style, functions can be arguments and can return the values of other functions. Many languages support this idea directly, but most object-oriented programming languages allow passing only objects to and from functions and, of course, you usually call functions in your object system's methods.

To address this seeming limitation, OOP design patterns (such as the Command pattern) suggest creating an object (a function or method, or whatever you want to call it) that contains the desired behavior and passing that around. This works perfectly well, but the syntax tends to be cumbersome, because all the syntactic scaffolding necessary to define a class, and then create and instantiate an object from it, really has no immediate relevance to the point of the source code—which, in such a situation, is simply to describe a function.

To ameliorate this, Java 1.1 provided anonymous classes, which reduce the syntactic scaffolding a little but, more importantly, allow the definition, instantiation, and usage
to be done all in the same place. For example, all of that could be done as a parameter to a method invocation. Therefore, passing an object that implements `SwissKnife` might look like the following when written using the anonymous syntax:

```java
SwissKnife sni = (String a) -> "Welcome, " + a;
```

While the anonymous syntax shown here is consistent with any version of Java from 1.1 onward, the `static` and `default` methods in the `SwissKnife` interface require at least Java 8.

As stated earlier, the behavior definition is colocated with the use, which is good, but the syntax is still very cluttered.

Java 8 addressed this clutter and, in specific situations, allows a better syntax, known as lambdas.

Lambdas are essentially expressions that define a class and instantiate an object from that class, but lambdas do it in a way that allows you to write code that addresses only the definition of a single function. Because of the restriction to define only a single function, the lambda syntax can be used only to implement interfaces that have a single abstract method (SAM).

Java's documentation generally refers to this type of interface as a functional interface, and the annotation with the same name can be used to verify that an interface does indeed have exactly one abstract method.

So, in this example, the `SwissKnife` interface is a functional interface, because it has a SAM named `welcomify`. The other two methods provide implementations, so they are not abstract and don't present a problem for the lambda syntax rules.

The job of the `@FunctionalInterface` annotation is to cause a compiler error if it is attached to an interface that has zero abstract methods or more than one. Because the code given compiles correctly, option D is incorrect.

It might be clear that the relationship between an interface and a lambda expression that implements it centers on the SAM in that interface. The lambda provides an implementation for that method, and the lambda's arguments and return type must match those of the interface's abstract method. By the way, the Java compiler's ability to determine the applicable type for a particular context is called type inferencing.

A lambda that implements `SwissKnife` must take a single parameter of type `String` and it must return a `String`. The only matching lambda is option C, so option C is the correct answer. Options A and B are incorrect because their parameter lists do not match the requirement of `SwissKnife`'s single abstract method.

As a side note, lambda expressions may include or omit the types of all their parameters. It is probably better, in general, to omit parameter types and allow them to be inferred from the context. However, in some situations, perhaps with overloaded methods, the target interface might be ambiguous. In such a situation, providing the argument type might be necessary to allow compilation to succeed. Also, including the argument type might make a program easier to read. Option A includes the argument types, but it fails because the argument list is incorrect in type `int` rather than `String` and in number (two instead of one), not because the argument types are specified. To illustrate that point, the following code would also be correct:

```java
SwissKnife sni = (String a) -> "Welcome, " + a;
```

But of course, that's not one of the options for this quiz.

**Conclusion. The correct answer is option C.**
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