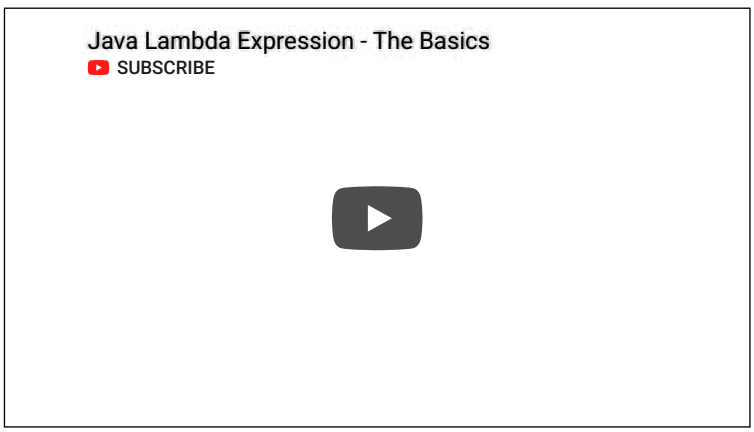
Java Lambda Expressions

Jakob Jenkov
Last update: 2018-05-27

Java lambda expressions are new in Java 8. Java lambda expressions are Java's first step into fu
programming. A Java lambda expression is thus a function which can be created without belongir
class. A Java lambda expression can be passed around as if it was an object and executed on de

Java lambda expressions are commonly used to implement simple event listeners / callbacks, or i
functional programming with the **Java Streams API**.

If you prefer video, I have a video version of this tutorial in this **Java Lambda Expression YouTu
Playlist**. Here is the first video in this playlist:



Java Lambdas and the Single Method Interface

Functional programming is very often used to implement event listeners. Event listeners in Java a
defined as Java interfaces with a single method. Here is a fictive single method interface example

```
public interface StateChangeListener {

    public void onStateChange(State oldState, State newState);

}
```

using callbacks).

In Java 7 you would have to implement this interface in order to listen for state changes. Imagine have a class called `StateOwner` which can register state event listeners. Here is an example:

```
public class StateOwner {

    public void addStateListener(StateChangeListener listener) { ... }

}
```

In Java 7 you could add an event listener using an anonymous interface implementation, like this:

```
StateOwner stateOwner = new StateOwner();

stateOwner.addStateListener(new StateChangeListener() {

    public void onStateChange(State oldState, State newState) {
        // do something with the old and new state.
    }
});
```

First a `StateOwner` instance is created. Then an anonymous implementation of the `StateChangeLis` interface is added as listener on the `StateOwner` instance.

In Java 8 you can add an event listener using a Java lambda expression, like this:

```
StateOwner stateOwner = new StateOwner();

stateOwner.addStateListener(
    (oldState, newState) -> System.out.println("State changed")
);
```

The lambda expressions is this part:

```
(oldState, newState) -> System.out.println("State changed")
```

The lambda expression is matched against the parameter type of the `addStateListener()` metho parameter. If the lambda expression matches the parameter type (in this case the `StateChangeLis` interface) , then the lambda expression is turned into a function that implements the same interfa that parameter.

Java lambda expressions can only be used where the type they are matched against is a single m interface. In the example above, a lambda expression is used as parameter where the parameter was the `StateChangeListener` interface. This interface only has a single method. Thus, the lambda expression is matched successfully against that interface.

Matching Lambdas to Interfaces

A single method interface is also sometimes referred to as a *functional interface*. Matching a Java expression against a functional interface is divided into these steps:

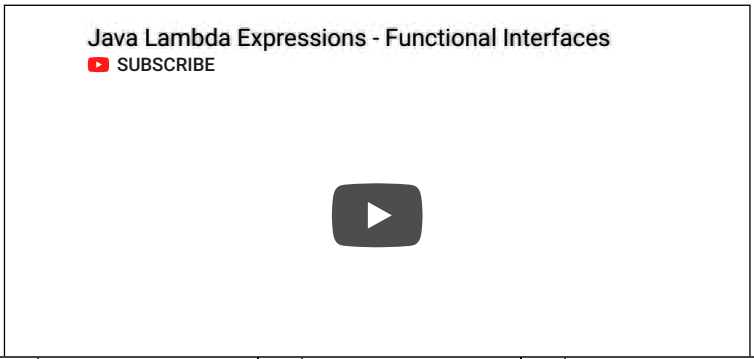
- Does the interface have only one method?
- Does the parameters of the lambda expression match the parameters of the single method?
- Does the return type of the lambda expression match the return type of the single method?

If the answer is yes to these three questions, then the given lambda expression is matched succe against the interface.

Interfaces With Default and Static Methods

From Java 8 a **Java interface** can contain both default methods and static methods. Both default methods and static methods have an implementation defined directly in the interface declaration. means, that a Java lambda expression can implement interfaces with more than one method - as the interface only has a single unimplemented (AKA abstract) method.

In other words, an interface is still a *functional interface* even if it contains default and static metho long as the interface only contains a single unimplemented (abstract) method. Here is a video ver this little section:



The following interface can be implemented with a lambda expression:

```
import java.io.IOException;
import java.io.OutputStream;

public interface MyInterface {

    void printIt(String text);

    default public void printUtf8To(String text, OutputStream outputStream){
        try {
            outputStream.write(text.getBytes("UTF-8"));
        } catch (IOException e) {
            throw new RuntimeException("Error writing String as UTF-8 to OutputStream",
        }
    }

    static void printItToSystemOut(String text){
        System.out.println(text);
    }
}
```

Even though this interface contains 3 methods it can be implemented by a lambda expression, be
only one of the methods is unimplemented. Here is how the implementation looks:

```
MyInterface myInterface = (String text) -> {
    System.out.print(text);
};
```

Lambda Expressions vs. Anonymous Interface Implementati

Even though lambda expressions are close to anonymous interface implementations, there are a
differences that are worth noting.

The major difference is, that an anonymous interface implementation can have state (member var
whereas a lambda expression cannot. Look at this interface:

```
public interface MyEventConsumer {

    public void consume(Object event);

}
```

This interface can be implemented using an anonymous interface implementation, like this:

```
MyEventConsumer consumer = new MyEventConsumer() {
    public void consume(Object event){
        System.out.println(event.toString() + " consumed");
    }
};
```

This anonymous `MyEventConsumer` implementation can have its own internal state. Look at this red

```
MyEventConsumer myEventConsumer = new MyEventConsumer() {
    private int eventCount = 0;
    public void consume(Object event) {
        System.out.println(event.toString() + " consumed " + this.eventCount++ + " time
    }
};
```

Notice how the anonymous `MyEventConsumer` implementation now has a field named `eventCount`.

A lambda expression cannot have such fields. A lambda expression is thus said to be stateless.

Lambda Type Inference

Before Java 8 you would have to specify what interface to implement, when making anonymous i
implementations. Here is the anonymous interface implementation example from the beginning of
text:

```
stateOwner.addStateListener(new StateChangeListener() {

    public void onStateChange(State oldState, State newState) {
        // do something with the old and new state.
    }
});
```

With lambda expressions the type can often be *inferred* from the surrounding code. For instance,
interface type of the parameter can be inferred from the method declaration of the `addStateLister`
method (the single method on the `StateChangeListener` interface). This is called *type inference*. T
compiler infers the type of a parameter by looking elsewhere for the type - in this case the metho
definition. Here is the example from the beginning of this text, showing that the `StateChangeLister`
interface is not mentioned in the lambda expression:

```
    (oldState, newState) -> System.out.println("State changed")
);
```

In the lambda expression the parameter types can often be inferred too. In the example above, th compiler can infer their type from the `onStateChange()` method declaration. Thus, the type of the parameters `oldState` and `newState` are inferred from the method declaration of the `onStateChange` method.

Lambda Parameters

Since Java lambda expressions are effectively just methods, lambda expressions can take param just like methods. The `(oldState, newState)` part of the lambda expression shown earlier specifie parameters the lambda expression takes. These parameters have to match the parameters of the on the single method interface. In this case, these parameters have to match the parameters of th `onStateChange()` method of the `StateChangeListener` interface:

```
public void onStateChange(State oldState, State newState);
```

As a minimum the number of parameters in the lambda expression and the method must match.

Second, if you have specified any parameter types in the lambda expression, these types must m I haven't shown you how to put types on lambda expression parameters yet (it is shown later in th but in many cases you don't need them.

Zero Parameters

If the method you are matching your lambda expression against takes no parameters, then you ca your lambda expression like this:

```
() -> System.out.println("Zero parameter lambda");
```

Notice how the parentheses have no content in between. That is to signal that the lambda takes r parameters.

One Parameter

If the method you are matching your Java lambda expression against takes one parameter, you c the lambda expression like this:

```
(param) -> System.out.println("One parameter: " + param);
```

Notice the parameter is listed inside the parentheses.

When a lambda expression takes a single parameter, you can also omit the parentheses, like this

```
 param -> System.out.println("One parameter: " + param);
```

Multiple Parameters

If the method you match your Java lambda expression against takes multiple parameters, the par need to be listed inside parentheses. Here is how that looks in Java code:

```
(p1, p2) -> System.out.println("Multiple parameters: " + p1 + ", " + p2);
```

Only when the method takes a single parameter can the parentheses be omitted.

Parameter Types

Specifying parameter types for a lambda expression may sometimes be necessary if the compiler infer the parameter types from the functional interface method the lambda is matching. Don't worr compiler will tell you when that is the case. Here is a Java lambda parameter type example:

```
(Car car) -> System.out.println("The car is: " + car.getName());
```

As you can see, the type (`Car`) of the `car` parameter is written in front of the parameter name itself you would when declaring a parameter in a method elsewhere, or when making an anonymous implementation of an interface.

Lambda Function Body

The body of a lambda expression, and thus the body of the function / method it represents, is spe the right of the `->` in the lambda declaration: Here is an example:

```
 (oldState, newState) -> System.out.println("State changed")
```

If your lambda expression needs to consist of multiple lines, you can enclose the lambda function

```
(oldState, newState) -> {
   System.out.println("Old state: " + oldState);
   System.out.println("New state: " + newState);
 }
```

Returning a Value From a Lambda Expression

You can return values from Java lambda expressions, just like you can from a method. You just ad
return statement to the lambda function body, like this:

```
(param) -> {
   System.out.println("param: " + param);
   return "return value";
 }
```

In case all your lambda expression is doing is to calculate a return value and return it, you can sp
return value in a shorter way. Instead of this:

```
(a1, a2) -> { return a1 > a2; }
```

You can write:

```
(a1, a2) -> a1 > a2;
```

The compiler then figures out that the expression `a1 > a2` is the return value of the lambda expres
(hence the name lambda *expressions* - as expressions return a value of some kind).

Lambdas as Objects

A Java lambda expression is essentially an object. You can assign a lambda expression to a varia
pass it around, like you do with any other object. Here is an example:

```
public interface MyComparator {

    public boolean compare(int a1, int a2);

}
```

```
MyComparator myComparator = (a1, a2) -> return a1 > a2;

boolean result = myComparator.compare(2, 5);
```

The first code block shows the interface which the lambda expression implements. The second co
shows the definition of the lambda expression, how the lambda expression is assigned to variable
finally how the lambda expression is invoked by invoking the interface method it implements.

Variable Capture

A Java lambda expression is capable of accessing variables declared outside the lambda functior
under certain circumstances. Java lambdas can capture the following types of variables:

- Local variables
- Instance variables
- Static variables

Each of these variable captures will described in the following sections.

Local Variable Capture

A Java lambda can capture the value of a local variable declared outside the lambda body. To illus
that, first look at this single method interface:

```
public interface MyFactory {
    public String create(char[] chars);
}
```

Now, look this lambda expression that implements the `MyFactory` interface:

```
MyFactory myFactory = (chars) -> {
    return new String(chars);
};
```

Right now this lambda expression is only referencing the parameter value passed to it (`chars`). Bu
change that. Here is an updated version that references a `String` variable declared outside the la
function body:

```
String myString = "Test";

MyFactory myFactory = (chars) -> {
```

As you can see, the lambda body now references the local variable `myString` which is declared ou
lambda body. This is possible if, and only if, the variable being references is "effectively final", mea
does not change its value after being assigned. If the `myString` variable had its value changed late
compiler would complain about the reference to it from inside the lambda body.

Instance Variable Capture

A lambda expression can also capture an instance variable in the object that creates the lambda.
an example that shows that:

```
public class EventConsumerImpl {

    private String name = "MyConsumer";

    public void attach(MyEventProducer eventProducer){
        eventProducer.listen(e -> {
            System.out.println(this.name);
        });
    }
}
```

Notice the reference to `this.name` inside the lambda body. This captures the `name` instance variabl
enclosing `EventConsumerImpl` object. It is even possible to change the value of the instance variab
its capture - and the value will be reflected inside the lambda.

The semantics of `this` is actually one of the areas where Java lambdas differ from anonymous
implementations of interfaces. An anonymous interface implementation can have its own instance
variables which are referenced via the `this` reference. However, an lambda cannot have its own i
variables, so `this` always points to the enclosing object.

Note: The above design of an event consumer is not particularly elegant. I just made it like that to
to illustrate instance variable capture.

Static Variable Capture

A Java lambda expression can also capture static variables. This is not surprising, as static variab
reachable from everywhere in a Java application, provided the static variable is accessible (packa
scoped or public).

Here is an example class that creates a lambda which references a static variable from inside the
body:

```
public class EventConsumerImpl {
    private static String someStaticVar = "Some text";

    public void attach(MyEventProducer eventProducer){
        eventProducer.listen(e -> {
            System.out.println(someStaticVar);
        });
    }
}
```

The value of a static variable is also allowed to change after the lambda has captured it.

Again, the above class design is a bit nonsensical. Don't think too much about that. The class prin
serves to show you that a lambda can access static variables.

Method References as Lambdas

In the case where all your lambda expression does is to call another method with the parameters
to the lambda, the Java lambda implementation provides a shorter way to express the method cal
here is an example single function interface:

```
public interface MyPrinter{
    public void print(String s);
}
```

And here is an example of creating a Java lambda instance implementing the `MyPrinter` interface

```
MyPrinter myPrinter = (s) -> { System.out.println(s); };
```

Because the lambda body only consists of a single statement, we can actually omit the enclosing
brackets. Also, since there is only one parameter for the lambda method, we can omit the enclosi
brackets around the parameter. Here is how the resulting lambda declaration looks:

```
MyPrinter myPrinter = s -> System.out.println(s);
```

Since all the lambda body does is forward the string parameter to the `System.out.println()` meth
can replace the above lambda declaration with a method reference. Here is how a lambda metho
reference looks:

```
MyPrinter myPrinter = System.out::println;
```

Notice the double colons `::`. These signal to the Java compiler that this is a method reference. The method referenced is what comes after the double colons. Whatever class or object that owns the referenced method comes before the double colons.

You can reference the following types of methods:

- Static method
- Instance method on parameter objects
- Instance method
- Constructor

Each of these types of method references are covered in the following sections.

Static Method References

The easiest methods to reference are static methods. Here is first an example of a single function interface:

```
public interface Finder {
    public int find(String s1, String s2);
}
```

And here is a static method that we want to create a method reference to:

```
public class MyClass{
    public static int doFind(String s1, String s2){
        return s1.lastIndexOf(s2);
    }
}
```

And finally here is a Java lambda expression referencing the static method:

```
Finder finder = MyClass::doFind;
```

Since the parameters of the `Finder.find()` and `MyClass.doFind()` methods match, it is possible to a lambda expression that implements `Finder.find()` and references the `MyClass.doFind()` metho

Parameter Method Reference

You can also reference a method of one of the parameters to the lambda. Imagine a single functio interface that looks like this:

```
public interface Finder {
    public int find(String s1, String s2);
}
```

The interface is intended to represent a component able to search `s1` for occurrences of `s2`. Here example of a Java lambda expression that calls `String.indexOf()` to search:

```
Finder finder = String::indexOf;
```

This is equivalent of this lambda definition:

```
Finder finder = (s1, s2) -> s1.indexOf(s2);
```

Notice how the shortcut version references a single method. The Java compiler will attempt to ma referenced method against the first parameter type, using the second parameter type as paramet referenced method.

Instance Method References

Third, it is also possible to reference an instance method from a lambda definition. First, let us loo single method interface definition:

```
public interface Deserializer {
    public int deserialize(String v1);
}
```

This interface represents a component that is capable of "deserializing" a `String` into an `int`.

Now look at this `StringConverter` class:

```
public class StringConverter {
    public int convertToInt(String v1){
        return Integer.valueOf(v1);
    }
}
```

The `convertToInt()` method has the same signature as the `deserialize()` method of the `Deseria` `deserialize()` method. Because of that, we can create an instance of `StringConverter` and refere `convertToInt()` method from a Java lambda expression like this:

```
Deserializer des = stringConverter::convertToInt;
```

The lambda expression created by the second of the two lines references the `convertToInt` metho `StringConverter` instance created on the first line.

Constructor References

Finally it is possible to reference a constructor of a class. You do that by writing the class name fo by `::new`, like this:

```
MyClass::new
```

Too see how to use a constructor as a lambda expression, look at this interface definition:

```
public interface Factory {
    public String create(char[] val);
}
```

The `create()` method of this interface matches the signature of one of the constructors in the `Stri` class. Therefore this constructor can be used as a lambda. Here is an example of how that looks:

```
Factory factory = String::new;
```

This is equivalent to this Java lambda expression:

```
Factory factory = chars -> new String(chars);
```

Next: Java Modules

Jakob Jenkov