Java 8 idioms: Using closures to capture state

Lambdas are stateless, but your programs don't have to be

Venkat Subramaniam

December 06, 2017

Whereas lambda expressions rely on internal parameters and constants, closures look to variables for additional information. Find out how to use closures to carry state from a defining context to the point of execution in your programs.

About this series
Java 8 is the most significant update to the Java language since its inception—packed so full of new features that you might wonder where to start. In this series, author and educator Venkat Subramaniam offers an idiomatic approach to Java 8: short explorations that invite you to rethink the Java conventions you've come to take for granted, while gradually integrating new techniques and syntax into your programs.

In Java™ programming, we loosely use the term *lambda expression* for both lambda expressions and closures. But in some cases it's important to understand the distinction. Whereas lambda expressions are stateless, closures carry state with them. Replacing lambda expressions with closures is an elegant way to manage state in functional-style programs.

The stateless life

We've worked with lambdas enough in this series that you are getting to know them very well. They are cute little anonymous functions that take optional parameters, perform some computation or action, and may return a result. Lambdas are also stateless, which can have interesting effects in your code.

Let's start with a simple example that uses a lambda expression. Suppose we want to double each of the even elements in a collection of numbers. One option is to create a functional pipeline, using `Stream` and lambdas, like so:

```
numbers.stream()
  .filter(e -> e % 2 == 0)
  .map(e -> e * 2)
  .collect(toList());
```

The lambda expression we've passed in to `filter` stands in for the `Predicate` functional interface. It takes a number and returns `true` if the number is even or `false` if not. The lambda expression

that we passed to `map`, on the other hand, stands in for the `Function` functional interface: it takes any number and returns twice its value.

Both of these lambdas rely on the parameter passed in and constant literals. Both are self-contained, meaning that they do not have any external dependencies. Because they rely on parameters passed in, and maybe some constants, lambdas are stateless. They are cute and calm, like napping babies.

Why we need state

Now let's look closer at the lambda expression that we passed to the `map` method. What if we wanted to triple or quadruple the given value? We could turn the constant `2` into a variable (say, `factor`), but we would still need a way for the lambda to get that variable.

We might reason that the lambda expression could receive `factor` in the same way it received the parameter `e`, like so:

```
.map((e, factor) -> e * factor)
```

Not bad, but sadly it won't work. The method `map` expects as argument an implementation of the functional interface `Function<T, R>`. The `map` won't budge if we pass in anything but that interface —like a `BiFunction<T, U, R>`, for example. There has to be another way to get `factor` to our lambda expression.

Lexical scoping

Functions expect variables to be in scope. Because they are anonymous functions at heart, lambdas expect variables referenced to be in scope, as well. Some variables are received as parameters to functions or lambdas. Some variables are defined locally. And some variables come from outside the function, in what's called the *lexical scope*.

Here's an example of lexical scoping.

```
public static void print() {
  String location = "World";

  Runnable runnable = new Runnable() {
    public void run() {
      System.out.println("Hello " + location);
    }
  };

  runnable.run();
}
```

In the `print` method, `location` is a local variable. However, the `run` method of `Runnable` also refers to a `location` that is not a local variable or a parameter to the `run` method. The reference to `location` next to `Hello` binds to the `print` method's `location` variable.

A *lexical scope* is the defining scope of a function. In turn, it may also be the defining scope of the defining scope, and so on.

In the previous code, the method `run` does not define `location` or receive it as a parameter. The defining scope of `run` is the anonymous inner object of `Runnable`. Because `location` is not defined as a field in that instance, the search continues to the defining scope of the anonymous inner object—in this case the local scope of the method `print`.

If `location` were not present in the scope, the compiler's search would continue in the defining scope of `print`, continuing until either the variable was found or the search failed.

Lexical scoping in a lambda expression

Now let's see what happens when we rewrite the previous code using a lambda expression:

```
public static void print() {
  String location = "World";

  Runnable runnable = () -> System.out.println("Hello " + location);

  runnable.run();
}
```

The code is more concise thanks to the lambda expression, but the scoping and binding for `location` have not changed. The variable `location` within the lambda expression is bound to the variable `location` in the lambda expression's lexical scope. Strictly speaking, the lambda expression in this code is a *closure*.

How closures carry state

Lambda expressions do not depend on anything external; instead, they are content relying on their own parameters and constants. Closures, on the other hand, depend on both parameters and constants, and variables in their lexical scope.

Logically speaking, closures bind to variables in their lexical scope. But while that is logically true, it's not always literally the case. At times, such binding is impossible.

Two scenarios will illustrate this point.

First, here's a piece of code that passes a lambda expression or a closure to a `call` method:

```
class Sample {
  public static void call(Runnable runnable) {
    System.out.println("calling runnable");

    //level 2 of stack
    runnable.run();
  }

  public static void main(String[] args) {
    int value = 4;  //level 1 of stack
    call(
      () -> System.out.println(value) //level 3 of stack
    );
  }
}
```

The closure in this code uses the variable `value` from its lexical scope. If the execution of `main` is in stack level 1, then the execution of the body of the `call` method will be in stack level 2. Since the `run` method of `Runnable` is called from within `call`, the body of the closure runs in level 3. If the `call` method were to pass the closure to yet another method (thus deferring the place of call) then the stack level of execution could be higher than 3.

You might now be wondering how in the world an execution in one level of stack is able to reach for a variable in another, previous level of the stack—especially without context being passed through the calls. The short answer is that it can't.

Now consider another example:

```
class Sample {
  public static Runnable create() {
    int value = 4;
    Runnable runnable = () -> System.out.println(value);

    System.out.println("exiting create");
    return runnable;
  }

  public static void main(String[] args) {
    Runnable runnable = create();

    System.out.println("In main");
    runnable.run();
  }
}
```

In this example, the `create` method has a local variable, `value`, which has a pretty short life: it will disappear as soon as we exit `create`. The closure created within the `create` refers to this variable in its lexical scope. The `create` method returns the closure to the caller in `main`, after the method has finished. In that process it removes the variable `value` from its stack, and the lambda expression is executed. Here is the resulting output:

```
exiting create
In main
4
```

We know that the `value` within `create` is dead by the time `run` is called in `main`. While we might have assumed that the `value` within the lambda expression would directly bind to the variable in its lexical scope, that assumption doesn't hold up.

An analogy will explain how this works.

Closure lunch break

Suppose my office is about 10 miles from my home (that's 16 KM for those using evolved units of measure), and I leave for work at 8 a.m. Right about noon I have a short time for lunch, but being health conscious I'd rather have a home-cooked meal. Given my short break, that's only possible if I carried lunch with me when I left home.

That is exactly what closures do: they carry their lunch (or state) with them.

To further clarify, let's look again at the lambda expression within `create`:

```
Runnable runnable = () -> System.out.println(value);
```

The lambda we wrote does not take any arguments but needs its `value`. Compile the class `Sample` and run `javap -c -p Sample.class` to examine the bytecode. You will notice that the compiler has created a method for the closure, where that method takes an `int` parameter:

```
private static void lambda$create$0(int);
    Code:
       0: getstatic     #3                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: iload_0
       4: invokevirtual #9                  // Method java/io/PrintStream.println:(I)V
       7: return
}
```

Now look at the bytecode generated for the `create` method:

```
0: iconst_4
1: istore_0
2: iload_0
3: invokedynamic #2,  0              // InvokeDynamic #0:run:(I)Ljava/lang/Runnable;
```

The value `4` is stored into a variable, which is then loaded and passed to the function created for the closure. The closure holds on to a copy of `value` in this case.

That is how closures carry state.

Using closures

Now we will revisit the example from the beginning of this article. Instead of doubling the value of even numbers in a collection, what if we wanted to triple or quadruple them? We can do it by turning our original lambda expression into a closure.

Here's the stateless code we saw earlier:

```
numbers.stream()
  .filter(e -> e % 2 == 0)
  .map(e -> e * 2)
  .collect(toList());
```

Using a closure instead of a lambda, the code becomes:

```
int factor = 3;

numbers.stream()
  .filter(e -> e % 2 == 0)
  .map(e -> e * factor)
  .collect(toList());
```

Instead of taking a lambda expression, the `map` method now takes a closure. We know that this closure is receiving a parameter, `e`, but it also captures and carries the state of the `factor` variable.

This variable is *in the lexical scope* of the closure. It could be a local variable in the function defining the lambda expression; it could come in as a parameter to that outer function; it could be anywhere in the defining scope of the closure (or in the defining scope of the defining scope, and so on). Regardless, the closure carries state from the code defining that closure to the point of execution where the variable is needed.

Conclusion

Closures differ from lambda expressions in that they rely on their lexical scope for some variables. As a result, closures can capture and carry state with them. While lambdas are stateless, closures are stateful. You can use them in your programs to carry state from a defining context to the point of execution.

Related topics

- Java programming with lambda expressions
- Are Java 8 lambdas closures?
- Functional Programming in Java: The Pragmatic Bookshelf, 2014