Java 8 idioms: Cascading lambdas

Reusable functions help make your code highly concise, but does conciseness ever go too far?

Venkat Subramaniam

November 07, 2017

Venkat explains the mysterious origins of cascading lambdas, a type of syntax that arises from functions returning functions in highly concise code.

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 functional-style programming, functions may both receive and return other functions. Instead of a function being simply a *factory* or *producer* of an object, as in traditional object-oriented programming, it is also able to create and return another function. Functions returning functions can result in *cascading lambdas*, especially in highly concise code. While this syntax may look quite strange at first, it has its uses. This article will help you recognize cascading lambdas and understand their nature and purpose in code.

Mysterious syntax

Have you ever seen a piece of code that looks like this?

```
x -> y -> x > y
```

You aren't alone if you are wondering, "What in the world is that?" For developers new to programming with lambdas, this syntax might look like something that has fallen out of a fast-moving truck.

Fortunately we don't see them very often, but understanding how cascading lambdas are created and how to read them in code will save you a lot of grief.

Higher-order functions

Before we can talk about cascading lambdas, it is important to understand how they are made. For that, we will need to revisit higher-order functions (introduced in the first article in this series) and

their role in *functional decomposition*, which is a way of breaking complex processes into smaller, simpler parts.

To get started, consider the rules separating higher-order functions from regular ones:

A regular function

- May receive objects
- May create objects
- May return objects

A higher-order function

- May receive functions
- May create functions
- May return functions

Developers pass anonymous functions or lambda expressions to higher-order functions to make code concise and expressive. Let's look at two examples of these higher-order functions.

Example 1: A function that receives a function

In Java™, we use functional interfaces to reference lambda expressions and method references. Here's a function that receives an object and a function:

```
public static int totalSelectedValues(List<Integer> values,
  Predicate<Integer> selector) {

  return values.stream()
    .filter(selector)
    .reduce(0, Integer::sum);
}
```

The first parameter to `totalSelectedValues` is the collection object, while the second parameter is the `Predicate` functional interface. Because the parameter type is a functional interface (`Predicate`), we may now pass a lambda expression as the second argument to `totalSelectedValues`. For example, if we wanted to total *only even values* in a `numbers` list, we could call `totalSelectedValues`, like so:

```
totalSelectedValues(numbers, e -> e % 2 == 0);
```

Suppose we now have a `static` method named `isEven` in a `Util` class. In this case, we could use `isEven` as an argument to `totalSelectedValues`, instead of passing the lambda expression:

```
totalSelectedValues(numbers, Util::isEven);
```

As a rule, whenever a functional interface appears as the *type* of the parameter to a function, you are looking at a higher-order function.

Example 2: A Function that returns a function

A function can receive a function, a lambda expression, or a method reference as a parameter. In a similar way, a function may also return a lambda expression or a method reference. In this case, the return type will be a functional interface.

Let's start with a function that creates and returns a `Predicate` to verify if a given value is an odd number:

```
public static Predicate<Integer> createIsOdd() {
  Predicate<Integer> check = (Integer number) -> number % 2 != 0;
  return check;
}
```

In order to return a function, we must provide a functional interface as the return type. In this case, our functional interface is `Predicate`. While the above code is syntactically correct, it could be more concise. We improve the code by using type inference and removing the temporary variable:

```
public static Predicate<Integer> createIsOdd() {
  return number -> number % 2 != 0;
}
```

Here's an example of the `createIsOdd` method in use:

```
Predicate<Integer> isOdd = createIsOdd();

isOdd.test(4);
```

Note that calling `test` on `isOdd` returns `false`. We may call `test` on `isOdd` with more values as well; it isn't limited to a single use.

Creating reusable functions

Now that you understand a bit about higher-order functions and how to spot them in code, we can consider using them to make code more concise.

Imagine we have two lists, `numbers1` and `numbers2`. Suppose we want to extract only numbers greater than 50 from the first list, and then we want to extract *and double* values greater than 50 from the second list.

We could achieve these goals with the following code:

```
List<Integer> result1 = numbers1.stream()
  .filter(e -> e > 50)
  .collect(toList());

List<Integer> result2 = numbers2.stream()
  .filter(e -> e > 50)
  .map(e -> e * 2)
  .collect(toList());
```

This code is good but do you notice the redundancy? We've duplicated the lambda expression that checks whether a number is greater than 50. We can remove the duplication and make the code more expressive by creating and reusing a `Predicate`:

```
Predicate<Integer> isGreaterThan50 = number -> number > 50;

List<Integer> result1 = numbers1.stream()
  .filter(isGreaterThan50)
  .collect(toList());

List<Integer> result2 = numbers2.stream()
  .filter(isGreaterThan50)
  .map(e -> e * 2)
  .collect(toList());
```

Storing a lambda expression in a reference enables us to reuse it, which is how we avoid duplicating lambda expressions. If we want to reuse the lambda expression across methods, we may also tuck that reference into a separate method instead of a local variable reference.

Now suppose we want to extract values greater than 25, 50, and 75 from list `numbers1`. We might start by writing three separate lambdas:

```
List<Integer> valuesOver25 = numbers1.stream()
  .filter(e -> e > 25)
  .collect(toList());

List<Integer> valuesOver50 = numbers1.stream()
  .filter(e -> e > 50)
  .collect(toList());

List<Integer> valuesOver75 = numbers1.stream()
  .filter(e -> e > 75)
  .collect(toList());
```

While each of the above lambdas compares input with a different value, they are all pretty much doing the same thing. How could we rewrite this code with less duplication?

Creating and reusing lambda expressions

While the two lambdas in the previous example were identical, there are slight variations in the three above. Creating a `Function` that returns a `Predicate` solves the problem at hand.

First, the functional interface `Function<T, U>` transforms an input of type `T` to an output of type `U`. For instance, the following example transforms a given value to its square root:

```
Function<Integer, Double> sqrt = value -> Math.sqrt(value);
```

Here, the return type of `U` could be something simple, like `Double`, `String`, or `Person`. Or it could be something more complex, like another functional interface such as `Consumer` or `Predicate`.

In this case, we want a `Function` to create a `Predicate`. So, here it is:

```
Function<Integer, Predicate<Integer>> isGreaterThan = (Integer pivot) -> {
  Predicate<Integer> isGreaterThanPivot = (Integer candidate) -> {
    return candidate > pivot;
  };

  return isGreaterThanPivot;
};
```

The reference `isGreaterThan` refers to a lambda expression that represents a `Function<T, U>`—or more precisely, `Function<Integer, Predicate<Integer>>`. The input is an `Integer` and the output is a `Predicate<Integer>`.

Within the body of the lambda expression (the outer `{}`) we created another reference, `isGreaterThanPivot`, that holds a reference to another lambda expression. This time the reference is to a `Predicate` instead of a `Function`. Finally, we return that reference.

`isGreaterThan` is a reference to a lambda expression that returns *another* lambda expression when invoked—in other words, there's a cascade of lambdas hidden in there.

Now we can use our newly created outer lamba expression to resolve the duplication in our code:

```
List<Integer> valuesOver25 = numbers1.stream()
  .filter(isGreaterThan.apply(25))
  .collect(toList());

List<Integer> valuesOver50 = numbers1.stream()
  .filter(isGreaterThan.apply(50))
  .collect(toList());

List<Integer> valuesOver75 = numbers1.stream()
  .filter(isGreaterThan.apply(75))
  .collect(toList());
```

The call to `apply` on `isGreaterThan` returns a `Predicate`, which is then passed in as argument to the `filter` method.

While all of this is fairly simple (for the sake of example) being able to abstract into a function is especially useful for scenarios where the predicate is more complex.

A recipe for conciseness

We've successfully removed the duplicate lambda expressions from our code, but the definition of `isGreaterThan` still looks noisy. Fortunately, we can combine several Java 8 conventions to reduce noise and make the code more concise.

We'll start by refactoring the following:

```
Function<Integer, Predicate<Integer>> isGreaterThan = (Integer pivot) -> {
  Predicate<Integer> isGreaterThanPivot = (Integer candidate) -> {
    return candidate > pivot;
  };

  return isGreaterThanPivot;
};
```

We can use type inference to remove type details from the parameters of both the outer and the inner lambda expressions:

```
Function<Integer, Predicate<Integer>> isGreaterThan = (pivot) -> {
  Predicate<Integer> isGreaterThanPivot = (candidate) -> {
    return candidate > pivot;
  };

  return isGreaterThanPivot;
};
```

So far we've removed two words from the code, which isn't much of an improvement.

Next, we remove the redundant `()`, as well as the unnecessary temporary reference within the outer lambda expression:

```
Function<Integer, Predicate<Integer>> isGreaterThan = pivot -> {
  return candidate -> {
    return candidate > pivot;
  };
};
```

The code is less noisy by a few decibels, but it still looks cluttered.

Seeing that the body of the inner lambda is only one line, it's clear that `{}` and `return` are redundant. Let's get rid of those:

```
Function<Integer, Predicate<Integer>> isGreaterThan = pivot -> {
  return candidate -> candidate > pivot;
};
```

Now we can see that the outer lambda's body is *also* a single line, and so the `{}` and `return` are redundant there, too. Here we apply the final refactoring:

```
Function<Integer, Predicate<Integer>> isGreaterThan =
  pivot -> candidate -> candidate > pivot;
```

And now you see it—there's our cascading lambda.

Understanding cascading lambdas

We arrived at our final code, the cascading lambda, through a process of refactoring that made sense at each stage. The outer lambda in this case receives `pivot` as a parameter, and the inner lambda receives `candidate` as a parameter. The body of the inner lambda uses both the parameter it receives (`candidate`) and the parameter from the outer scope. That is, the body of the inner lambda relies on both its parameter and its *lexical scope* or *defining scope*.

The cascading lambda makes perfect sense for the person who wrote it. But what about the reader?

When you see a lambda expression with a single right-arrow (`->`), you should know that you are looking at an anonymous function that takes parameters (possibly empty) and either performs an action or returns a result value.

When you see a lambda expression with two right arrows (`->`), you are looking at an anonymous function that takes parameters (possibly empty) and returns another lambda expression. The returned lambda expression may take its own parameters or it may be empty. It may perform an action or return a value. It may even return yet another lambda expression, but that's usually overkill and best avoided.

In essence, when you see two right-arrows, treat everything to the right of the first arrow as a black box: a lambda expression returned by the outer lambda expression.

Conclusion

Cascading lambdas are not too common but you should know how to recognize and read them in your code. When a lambda expression returns another lambda expression, instead of taking an action or returning a value, you will see two arrows. This code is highly concise but can be quite intimidating upon first encounter. Once you have learned to recognize this functional-style syntax, however, it becomes much easier to understand and manage.

Related topics

- Java programming with lambda expressions
- Java 8 language changes
- Functional Programming in Java: The Pragmatic Bookshelf, 2014