developerWorks®

Java 8 idioms: Why the perfect lambda expression is just one line

Write single-line lambdas for code that is easier to read, test, and reuse

Venkat Subramaniam

August 02, 2017

Short, concise lambda expressions support code readability, which is one of the key benefits of programming in the functional style. In addition to being hard to read, multiline lambdas are also hard to test and reuse, which can lead to duplication of effort and poor code quality. In this installment, learn to write single-line lambdas for code that is easier to read, test, and reuse.

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.

As you've learned in this series so far, a major benefit of function composition is that it results in code that is highly expressive. Writing short, concise lambda expressions is key to that expressiveness, but is often much easier said than done. This article deepens what you've learned so far about the art and science of crafting lambdas. By studying the structure and benefits of function composition, you will soon be on your way to mastering the perfect lambda—one that unfolds across a short, single line.

Two ways to write a lambda expression

Lambda expressions are, as you know, anonymous functions. They are also inherently concise. A normal function or method typically has four elements:

- A name
- The return type
- Parameter list
- Body

A lambda expression has only the last two of those four, as you see here:

```
(parameter list) -> body
```

The `->` separates the list of parameters from the body of the function, which is intended to do something with the given parameters. The body of the function may be a single expression or a statement. Here's an example:

```
(Integer e) -> e * 2
```

In this code, the body is just one line: an expression that returns twice the given parameter. The signal-to-noise ratio is high, with no semicolons and no need for the `return` keyword. This is an ideal lambda expression.

Multiline lambdas

In Java™, it is also possible for the body of a lambda to be a complex expression or statement; that is, a lambda expression consisting of more than one line. In that case, semicolons are necessary. If the lambda is returning a result, the `return` keyword is also required. Here's an example:

```
(Integer e) -> {
  double sqrt = Math.sqrt(e);
  double log = Math.log(e);

  return sqrt + log;
}
```

The lambda in this example returns the sum of `sqrt` and a `log` of the given parameter. Because the body contains multiple lines, the brackets (`{}`), semicolons (`;`), and `return` keyword are all required.

It feels like Java is punishing us for writing a multiline lambda—and perhaps we should take the hint.

The power of function composition

Coding in the functional style leverages the expressive power of functional composition. It's easy to see the benefits of expressiveness when comparing two pieces of code. Here's the first, written in the imperative style:

```
int result = 0;
for(int e : values) {
  if(e > 3 && e % 2 == 0) {
    result = e * 2;
    break;
  }
}
```

Now consider the same code written in the functional style:

```
int result = values.stream()
  .filter(e -> e > 3)
  .filter(e -> e % 2 == 0)
  .map(e -> e * 2)
  .findFirst()
  .orElse(0);
```

Both code snips achieve the same result. In the imperative-style code, we have to read into the `for` loop and follow the flow through the branch and the break. The second code snip, with function composition, is much easier to read. Because it flows from top to bottom, we only need to pass through the code once.

In essence, the second code snip reads like a problem statement: *Given the values, select only those greater than 3. From these, choose only even values, and double them. Finally, pick the first result. If no value exist, return zero.*

Not only is this code elegant but it does *no more work* than its imperative counterpart. Thanks to `Stream`'s lazy evaluation capabilities, there are no wasted computations.

The expressive power of function composition relies greatly on the conciseness of each lambda expression. If your lambda unfolds over multiple lines (even two lines may be too many) you are likely missing a key point of functional-style programming.

The perilously long lambda

To better understand the benefits of writing short, concise lambda expressions, consider the opposite: a sprawling lambda that unfolds over several lines of code:

```
System.out.println(
values.stream()
  .mapToInt(e -> {
    int sum = 0;
    for(int i = 1; i <= e; i++) {
      if(e % i == 0) {
        sum += i;
      }
    }

    return sum;
  })
  .sum());
```

Even though this code is written in the functional style, it misses the benefits of functional-style programming. Let's consider the reasons why.

1. It's hard to read

Good code should be inviting to read. This code takes mental effort to read: your eyes strain to find the beginning and end of the different parts.

2. Its purpose isn't clear

Good code should read like a story, not like a puzzle. A long, anonymous piece of code like this one hides the details of its purpose, costing the reader time and effort. Wrapping this piece of code into a named function would make it modular, while also bringing out its purpose through the associated name.

3. Poor code quality

Whatever your code does, it's likely that you'll want to reuse it sometime. The logic in this code is embedded within the lambda, which in turn is passed as an argument to another function, `mapToInt`. If we needed the code elsewhere in our program, we might be tempted to rewrite it, thus introducing inconsistencies in our code base. Alternatively, we might just copy and paste the code. Neither option would result in good code or quality software.

4. It's hard to test

Code always does what was typed and not necessarily what was intended, so it stands that any nontrivial code must be tested. If the code within the lambda expression can't be reached as a unit, it can't be unit tested. You could run integration tests, but that is no substitute for unit testing, especially when that code does significant work.

5. Poor code coverage

A student in a Java 8 course recently quipped that they hated lambda expressions. When I asked why, they showed me a colleague's work, which included lambda expressions running into hundreds of lines of code. Lambdas that were embedded in arguments were not easily extracted as units, and many showed up red on the coverage report. With no insight, the team simply had to assume that those pieces worked.

Lambdas as glue code

The way to resolve all of these issues is to make your lambdas highly concise. As a first and powerful step, simply avoid using brackets in your lambdas. Consider how easily we rework the previous, sprawling lambda using just this technique:

```
System.out.println(
values.stream()
  .mapToInt(e -> sumOfFactors(e))
  .sum());
```

This code is concise, even if it is incomplete. It is also highly readable. It says: *Given a collection of values, transform the list to sums of factors of each number, then compute the sum of the resulting collection.* Explicitly naming the body of code that was previously contained in brackets makes this code much easier to read and understand. The function pipeline is now crisp and easy to follow.

There is no need to solve puzzles because this code readily reveals its purpose. The code that computes the sum of factors has been modularized into a separate method named `sumOfFactors`, which can be reused. Since it's in a standalone method, the logic is also easy to unit test. Because this code is so testable, you can be assured of good code coverage.

In short, the once sprawling lambda is now *glue code*—instead of taking on a large chunk of responsibility, it merely glues the named function to the `mapToInt` function.

Fine-tuning with method references

As you've seen elsewhere in this series, it's possible to make the above code a tad more expressive by replacing the lambda expression with a method reference (where `sumOfFactors` is a method of a class named `Sample`):

```
System.out.println(
values.stream()
  .mapToInt(Sample::sumOfFactors)
  .sum());
```

Here's the rewritten `sumOfFactors` method:

```
public static int sumOfFactors(int number) {
  return IntStream.rangeClosed(1, number)
    .filter(i -> number % i == 0)
    .sum();
}
```

Now that is one short method. The lambda expression in the method is also concise: a single line with no excess ceremony or noise.

Conclusion

Short, concise lambda expressions support code readability, which is one of the key benefits of programming in the functional style. Lambda expressions that unfold over multiple lines have the opposite effect, of making code noisy and hard to read. Multiline lambdas are also hard to test and reuse, which can lead to duplication of effort and poor code quality. Fortunately, it's easy to avoid these issues by moving the body of a multiline lambda to a named function, then invoking the function from within the lambda. I also recommend replacing lambda expressions with method references wherever possible.

In short, I recommend avoiding multiline lambda expressions, except when illustrating why they are bad.

Related topics

- Using method references in Java 8
- Java programming with lambda expressions
- Java 8 language changes
- Functional Programming in Java: The Pragmatic Bookshelf, 2014
- IBM Code: Java journeys