Java 8 idioms: An alternative to passing through

Learn how to recognize and replace pass-through lambdas with method references in your Java code

Venkat Subramaniam

July 07, 2017

Pass-through lambda expressions come in a few varieties, but they are almost always more trouble than they're worth. Learn how to identify common varieties of pass-throughs in your code, then see what happens when you replace each one with a simple and expressive method reference.

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.

Lambda expressions are extensively used in functional-style programming, but they can be challenging to read and understand. In many cases where the lambda expression exists *solely* to pass through one or more parameters, you would be better off replacing it with a method reference. In this article you'll learn how to recognize pass-through lambdas in your code, and how to replace them with corresponding method references. While there is a learning curve to using method references, the long-term benefits will soon outweigh your initial effort.

What is a pass-through lambda?

In functional-style programming, it is common to pass lambda expressions as anonymous functions, using the lambda as an argument to a higher-order function. In Listing 1, for instance, we pass a lambda expression to the `filter` method:

Listing 1. A pass-through lambda expression

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

numbers.stream()
  .filter(e -> e % 2 == 0)
  .forEach(e -> System.out.println(e));
```

Note that in this code we've also passed a lambda expression to the `forEach` method. While the two lambda expressions obviously do different things, there is another important, if subtle difference between them: the first lambda expression actually does some work, while the second one does not.

The lambda expression passed to the `forEach` method is what we call a *pass-through lambda expression*. The expression `e -> System.out.println(e)` *passes its parameter through* as an argument to the `println` method of the `PrintStream` class, which is the `System.out` instance.

While there isn't anything wrong with the second lambda expression in Listing 1, its syntax is more complex than necessary for the task at hand. In order to understand the purpose of `(parameters) -> body`, we have to step into the *body* (on the right side of `->`) to see what is happening with the parameter. If the lambda expression isn't actually doing anything with the parameter, that effort is wasted.

In this case it would be advantageous to replace the pass-through lambda with a method reference. Unlike a call to a method, a method reference names the method to which we are passing the parameter. Using a method reference also opens up a variety of options for the way that parameters are passed through.

Try rewriting the previous code like so:

Listing 2. Passing a parameter with a method reference

```
numbers.stream()
  .filter(e -> e % 2 == 0)
  .forEach(System.out::println);
```

Using the method reference reduces the effort of understanding the code. While this might appear a rather small benefit at first, it multiplies as we write and read more code.

Passing a parameter as an argument

In the next sections we'll explore variations of pass-through lambdas. I'll show you how to replace each one with a method reference.

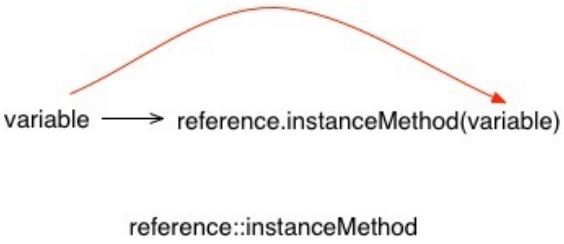
Argument to an instance method

It's very common for a lambda expression to pass its parameter as an argument to an instance method. You saw this in Listing 1, where the parameter `e` was passed as an argument to the `println` method, which is itself an instance method of `System.out`.

In Listing 2, we replaced this lambda expression with the method reference `System.out::println`, using the format: *referenceToInstance::methodName*.

You can see this in Figure 1, which shows the structure of a lambda expression, where the parameter is passed through as an argument to an instance method:

Figure 1. From parameter to argument of an instance method



If you are new to method references, seeing the lambda expression like this could help you to understand its structure and where the parameter is passed through. To change the lambda expression to a method reference, you would simply remove the common parts, the parameter, and the argument, and replace the dot with a colon on the method call.

Argument to a method on `this`

A special case of the previous pass-through is when the instance method is called on a *context* instance of the current method.

Suppose we have a class named `Example` with an instance method of `increment`:

Listing 3. A class with an instance method

```
public class Example {
  public int increment(int number) {
    return number + 1;
  }

  //...
}
```

Now suppose we have another instance method, where we have created a lambda expression and passed it through the `Stream`'s `map` method, like so:

Listing 4. A lambda expression passing through the parameter to an instance method

```
.map(e -> increment(e))
```

It may not be immediately obvious, but this code is very similar in structure to the previous example—in both cases, we've passed the parameter as argument to an instance method. Rewriting this code just slightly makes the similarity more obvious:

Listing 5. The pass-through revealed

```
.map(e -> this.increment(e))
```

Introducing the redundant `this` as the target of the call to `increment` makes the structure of the pass-through clear. Now we can easily resolve the redundancy with a method reference:

Listing 6. A method reference resolves redundancy

```
.map(this::increment)
```

Much like we replaced `e -> System.out.println(e)` with `System.out::println`, we can replace the lambda expression `e -> increment(e)` (or more precisely `e -> this.increment(e)`) with `this::increment`. In both cases, the code is clearer.

Argument to a static method

In the previous two examples, we replaced a lambda expression that passed a parameter as an argument to an instance method. It is also possible to replace a lambda expression that passes a parameter to a static method.

Listing 7. Passing a parameter to a static method

```
.map(e -> Integer.valueOf(e))
```

Here, the lambda expression is passing the parameter as an argument to the `valueOf` method of the `Integer` class. The structure of the code is the same as what you saw in Figure 1. The only difference is that, in this case, the method being called is a *static method* rather than an instance method. Just like the two previous examples, we replace this lambda with a method reference. Instead of placing the method reference on an instance, we will place it on a class, as shown in Listing 8.

Listing 8. Method reference to a static method

```
.map(Integer::valueOf)
```

To summarize: If the purpose of the lambda expression is solely to pass a parameter to an instance method, then you may replace it with a method reference on the instance. If the pass-through is to a static method, then you may replace it with a method reference on the class.

Passing a parameter to a target

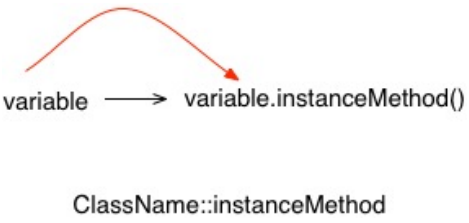
There are two different scenarios where you may use the *ClassName::methodName* form. You've just seen the first scenario, where the parameter is passed through as an argument to a static method. Now let's consider a variation where the parameter is the target of a method call.

Listing 9. Using the parameter as a target

```
.map(e -> e.doubleValue())
```

In this example, the parameter `e`, whose inferred type we assume to be `Integer`, is the target of the call to the `doubleValue` method. The structure of this type of pass-through lambda is shown in Figure 2.

Figure 2. From parameter to target



Even though the parameters in the two previous lambda expressions were passed through differently—one as the argument to a static method and the other as the target of an instance method call—the format of the method reference is exactly the same: *ClassName::methodName*.

Ambiguity and method references

Looking at a method reference, it isn't easy to tell whether the parameter is passed to a static method or is being used as a target. To know the difference, we would need to know whether the method was a static or an instance method. From a code-readability perspective that doesn't matter very much, but knowing the difference is vital for successful compilation.

If a class has both a static method and a compatible instance method with the same name, and we use a method reference, the compiler will complain about the ambiguity of the call. So, for example, we couldn't replace the lambda expression `(Integer e) -> e.toString()` with the method reference `Integer::toString`, since the `Integer` class has both the static method `public static String toString(int i)` and the instance method `public String toString()`.

You or your IDE might suggest using `Object::toString` to resolve this particular issue, since there is no `statictoString` method in `Object`. While that solution might compile, such cleverness is generally not helpful. You must be able to verify that the method reference is calling the intended method. When in doubt, it's best to use the lambda expression, to avoid any confusion or possible errors.

Passing constructor calls

In addition to static and instance methods, method references may be used to represent calls to the constructor. Consider a constructor call made from within a `Supplier`, as presented to the `toCollection` method shown below.

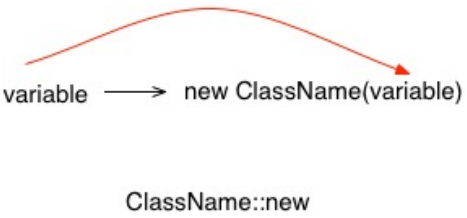
Listing 10. A constructor call

```
.collect(toCollection(() -> new LinkedList<Double>()));
```

The code in Listing 10 intends to take a `Stream` of data and reduce or collect it into a `LinkedList`. The `toCollection` method takes a `Supplier` as its argument. The supplier takes no parameters, hence the empty `()`. It returns an instance of a `Collection`, which in this case is `LinkedList`.

Although the parameter list is empty, the general structure of the code is as follows:

Figure 3. From parameter to constructor argument



The parameter received, which may be empty, is passed as an argument to the constructor. In such a case, we could replace the lambda expression with a method reference to `new`. Here's how:

Listing 11. Replacing a constructor call with a method reference

```
.collect(toCollection(LinkedList::new));
```

The code with the method reference is much less noisy than the original code with the lambda expression, and therefore easier to read and understand.

Passing multiple arguments

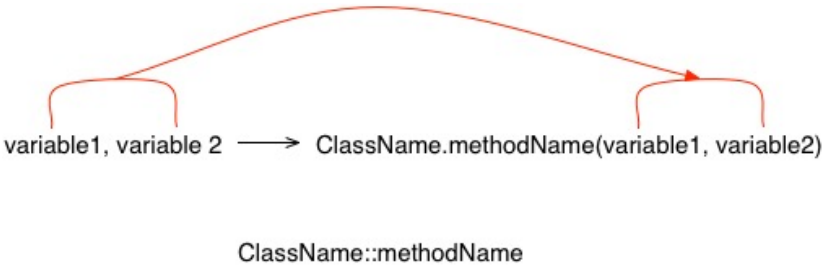
So far you've seen examples of a single parameter and an empty parameter being passed through. But method references aren't limited to zero or one parameter: they work just as well with multiple arguments. Consider the following `reduce` operation:

Listing 12. reduce() with a lambda expression

```
.reduce(0, (total, e) -> Integer.sum(total, e)));
```

The `reduce` method is called on a `Stream<Integer>`, and sums up the values in the stream using `Integer`'s `sum` method. The lambda expression in this case takes two parameters, which are passed (in exactly the same order) as arguments to the `sum` method. Figure 4 shows the structure of this lambda expression.

Figure 4. Passing two parameters as arguments



We could replace the lambda expression with a method reference, like so:

Listing 13. Replacing a lambda expression that takes two parameters

```
.reduce(0, Integer::sum));
```

If the `static` method call takes as its argument the given parameters to the lambda expression, in exactly the same order as they appear in the parameter list, then it is possible to replace the lambda expression with a method reference pointing to the `static` method.

Pass through as target and argument

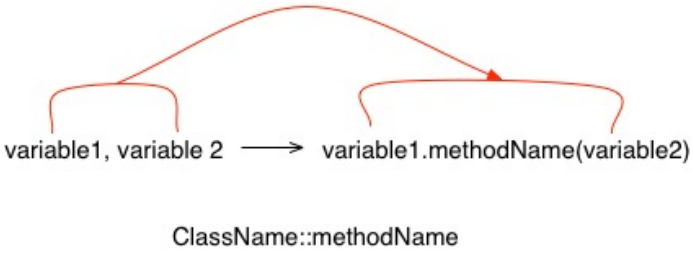
Instead of passing all parameters as arguments to a `static` method, a lambda might use one of the parameters as the target of an instance method call. If the first parameter is used as a target, then you could replace the lambda expression with a method reference. Consider Listing 14.

Listing 14. reduce() with a lambda expression that uses a parameter as a target

```
.reduce("", (result, letter) -> result.concat(letter)));
```

In this example, the `reduce` method is called on a `Stream<String>`. The lambda expression uses the `concat` instance method of `String` to concatenates the strings. The structure of the pass-through in this lambda expression is different from what you saw in the previous `reduce` example:

Figure 5. First parameter passed though as the target of a call



The first parameter of the lambda expression is used as the target of an instance method call. The second parameter is used as an argument to that method. Given this order, you could replace the lambda expression with a method reference, as shown in Listing 15:

Listing 15. A method reference with the first parameter as target

```
.reduce("", String::concat));
```

Note that once again you use the class name even though the lambda expression is invoking an instance method. In other words, the method reference looks the same regardless of whether you call a static method or invoke an instance method with the first parameter as a target. As long as there is no ambiguity, there is no problem.

Better with method references

It takes some time and effort to grasp the variations and structure of pass-through lambda expressions and the method references that replace them. It took me a couple of weeks to allow the concepts and syntax to sink in. After that, using method references instead of pass-throughs started feeling more natural.

Even more so than lambdas, method references make your code very concise and expressive, which can greatly reduce the effort of reading your code. To further drive home this point, consider one last example.

Listing 16. An example using lambda expressions

```
List<String> nonNullNamesInUpperCase =
    names.stream()
      .filter(name -> Objects.nonNull(name))
      .map(name -> name.toUpperCase())
      .collect(collectingAndThen(toList(), list -> Collections.unmodifiableList(list)));
```

Given a `List<String> names`, the code above removes any `null` values in the list, converts each name to uppercase, and collects the results into an unmodifiable list.

Now let's rewrite the above code using method references. In this case, each lambda expression is a pass-through, either to a static method or an instance method. Thus, we replace each lambda expression with a method reference:

Listing 17. Using method references

```
List<String> nonNullNamesInUpperCase =
    names.stream()
      .filter(Objects::nonNull)
      .map(String::toUpperCase)
      .collect(collectingAndThen(toList(), Collections::unmodifiableList));
```

Comparing the two listings, it's easy to see that the code using method references is more fluent and easier to read. It simply says: *given the names, filter non-nulls, map to uppercase, and collect to an unmodifiable list*.

Conclusion

Whenever you see a lambda expression whose sole purpose is to pass parameters to one or more other functions, consider whether that lambda would be better replaced by a method reference. The deciding factor is that no real work is being done inside the lambda expression. In that case, the lambda expression is a *pass-through*, and the syntax is probably too complex for the task at hand.

For most developers, it will take a bit of effort to learn how to use method references. Once you are familiar with them, however, you will find that using method references makes your code more fluent and expressive than the same code using lambda expressions.

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