Learn how to use lambda expressions to greatly reduce code clutter.

by Venkat Subramaniam

May 28, 2021

Collections of numbers, strings, and objects are used so commonly in Java that removing even a small amount of ceremony from coding them can reduce code clutter greatly. In this two-part article, I demonstrate how to use lambda expressions to take advantage of the functional style of programming to create more-expressive and concise code with less mutability and fewer errors.

After you read this article, your Java code to manipulate collections might never be the same—it'll be concise, expressive, elegant, and more extensible than ever before.

Iterating through a list

Iterating through a list is a basic operation on a collection, but over the years that operation has gone through a few significant changes. I’ll begin with the old and evolve an example—enumerating a list of names—to the elegant style.

You can easily create an immutable collection of a list of names with the following code:

```java
final List<String> friends = Arrays.asList("Brian", "Nate", "Neal", "Raju")
```
Here’s the habitual, but not so desirable, way to iterate and print each of the elements.

```java
for(int i = 0; i < friends.size(); i++) { System.out.print...
```

I call this style the self-inflicted wound pattern—it’s verbose and error-prone. You have to stop and wonder, “Is it i < or i <=?” This is useful only if you need to manipulate elements at a particular index in the collection, but even then, you could opt to use a functional style that favors immutability, as will be discussed soon.

Java offers a construct that is a bit more civilized than the good old for loop.

```java
for(String name : friends) { System.out.print...
```

Under the hood, this form of iteration uses the `Iterator` interface and calls into its `hasNext()` and `next()` methods.

Both these versions are external iterators, which mix how you do it with what you would like to achieve. You explicitly control the iteration with them, indicating where to start and where to end; the second version does that under the hood using the `Iterator` methods. With explicit control, the `break` and `continue` statements help manage the iteration’s flow of control.

The second construct has less ceremony than the first and is better than the first if you don’t intend to modify the collection at a particular index. Both styles, however, are imperative and you can dispense with them in modern Java by using a functional approach.

There are quite a few reasons to favor the change to the functional style.

- `for` loops are inherently sequential and are quite difficult to parallelize.
- Such loops are nonpolymorphic: You get exactly what you ask for. You passed the collection to `for` instead of invoking a method (a polymorphic operation) on the collection to perform the task.
- At the design level, the code fails the “Tell, don’t ask” principle. You ask for a specific iteration to be performed instead of leaving the details of the iteration to the underlying libraries.

It’s time to trade in the old imperative style for the more elegant functional-style version of internal iteration. With an internal iteration, you willfully turn over the how to the underlying library so you can focus on the essential what. The underlying function will take care of managing the iteration.
Here, you will use an internal iterator to enumerate the names. The `Iterable` interface has been enhanced (beginning in JDK 8) with a special method named `forEach()`, which accepts a parameter of type `Consumer`. As the name indicates, an instance of `Consumer` will consume, through its `accept()` method, which is what's given to it. Use the `forEach()` method with the anonymous inner class syntax.

```java
friends.forEach(new Consumer<String>() {
    public void accept(final String name) {
        System.out.println(name);
    }
});
```

You have invoked `forEach()` on the `friends` collection and passed an anonymous instance of `Consumer` to it. The `forEach()` method will invoke the `accept()` method of the given `Consumer` for each element in the collection and perform a specified action. In this example, the action merely prints the given value, which is a name.

Look at the output from this version, which is the same as the output from the two previous versions.

```
Brian
Nate
Neal
Raju
Sara
Scott
```

You changed merely one thing, trading in the old `for` loop for the new internal iterator `forEach()`. As for the benefit, you went from specifying how to iterate to focusing on what you want to do for each element. The bad news is the code looks a lot more verbose, so much that it can drain away any excitement about the new style of programming. Thankfully, you can fix that quickly. This is where lambda expressions and the compiler magic come in. Let's make one change, replacing the anonymous inner class with a lambda expression.

```java
friends.forEach((final String name) -> System.out.println(name));
```

That's a lot better, and not only because it is shorter. The `forEach()` is a higher-order function that accepts a lambda expression or block of code to execute in the context of each element in the list. The variable `name` is bound to each element of the collection during the call. The underlying library takes control of how any lambda expressions are evaluated. It can decide to perform them lazily, in any order, and exploit parallelism as it sees fit.
This version produces the same output as the previous versions. The internal-iterator version is more concise than the other ones. In addition, it helps focus your attention on what you want to achieve for each element rather than how to sequence through the iteration—it’s declarative.

The better-code version has a limitation, however. Once the `forEach()` method starts, unlike in the other two versions, you can’t break out of the iteration. (There are facilities to handle this limitation.) Consequently, this style is useful where you want to process each element in a collection. Later I’ll show other functions that offer more control over the path of iteration.

The standard syntax for lambda expressions expects the parameters to be enclosed in parentheses, with the type information provided and comma separated. The Java compiler also offers some lenience and can infer the types. Leaving out the type is convenient, requires less effort, and is less noisy. Here’s the previous code without the type information.

```java
friends.forEach((name) -> System.out.println(name));
```

In this case, the Java compiler determines that the `name` parameter is a `String` type, based on the context. It looks up the signature of the called method—`forEach()`, in this example—and analyzes the functional interface it takes as a parameter. It then looks at that interface’s abstract method to determine the expected number of parameters and their types. You can also use type inference if a lambda expression takes multiple parameters, but in that case, you must leave out the type information for all the parameters; you have to specify the type for none or for all of the parameters in a lambda expression.

The Java compiler treats single-parameter lambda expressions as special. You can leave off the parentheses around the parameter if the parameter’s type is inferred.

```java
friends.forEach(name -> System.out.println(name));
```

There’s one caveat: Inferred parameters are non-`final`. The previous example, which explicitly specified the type, also marked the parameter as `final`. This prevents modifying the parameter within the lambda expression. In general, modifying parameters is in poor taste and leads to errors, so marking them `final` is a good practice. Unfortunately, when you favor type inference, you have to practice extra discipline not to modify the parameter, because the compiler will not protect you.

This example has reduced the code quite a bit. One last step will tease out another ounce of conciseness.
This code uses a method reference. Java lets you simply replace the body of code with the method name of your choice. I will dig into this further in the next section, but for now let's reflect on the wise words of Antoine de Saint-Exupéry: “Perfection is achieved not when there is nothing more to add, but when there is nothing left to take away.”

Lambda expressions helped you concisely iterate over a collection. Next, you’ll see how they help remove mutability and make the code even more concise when you transform collections.

**Transforming a list**

Manipulating a collection to produce another result is as easy as iterating through the elements of a collection. Suppose the task is to convert a list of names to all capital letters. What are some options?

Java’s `String` is immutable, so instances can’t be changed. You could create new strings in all caps and replace the appropriate elements in the collection. However, the original collection would be lost; also, if the original list is immutable, as it is when it’s created with `Arrays.asList()`, the list can’t change. It would also be hard to parallelize the work.

Creating a new list that has the elements in all uppercase is better.

That suggestion may seem quite naive at first; performance is an obvious concern for everyone. You’re likely to find, however, that the functional approach often yields surprisingly better performance than the imperative approach. Start by creating a new collection of uppercase names from the given collection.

```java
final List<String> uppercaseNames = new ArrayList<String>();
for(String name : friends) {
    uppercaseNames.add(name.toUpperCase());
}
```

In this imperative style, this code created an empty list and then populated it with uppercase names, one element at a time, while iterating through the original list. As a first step to move toward a functional style, use the internal iterator `forEach()` method to replace the `for` loop, as follows:

```java
final List<String> uppercaseNames = new ArrayList<String>();
friends.forEach(name ->
    uppercaseNames.add(name.toUpperCase());
}
This code used the internal iterator, but that still required the empty list and the effort to add elements to it.

Going the next step, the map() method of a new Stream interface can help avoid mutability and make the code concise. A Stream is much like an iterator on a collection of objects and provides some nice fluent functions. Using the methods of this interface, you can compose a sequence of calls, so the code reads and flows in the same way you’d state the problem, making it easier to read.

The map() method of Stream can map or transform an input sequence to an output sequence. This will fit quite well for the task at hand.

The stream() method is available on all collections since JDK 8, and it wraps the collection into an instance of Stream. The map() method applies the given lambda expression or block of code within the parentheses on each element in the Stream. The map() method is quite unlike the forEach() method, which simply runs the block in the context of each element in the collection. In addition, the map() method collects the result of running the lambda expression and returns the resulting collection. Finally, the code prints the elements in this result using the forEach() method. The names in the new collection are in all capital letters.

The map() method is very useful for mapping or transforming an input collection into a new output collection. This method will ensure that the same number of elements exists in the input and the output sequence. However, element types in the input don’t have to match the element types in the output collection.

In this example, both the input and the output are a collection of strings. You could have passed to the map() method a block of code that returned, for example, the number of characters in a given name. In this case, the input would still be a sequence of strings, but the output would be a sequence of numbers, as in the next example.
The result is a count of the number of letters in each name.

The versions using the lambda expressions have no explicit mutation; they're concise. These versions also didn't need any initial empty collection or garbage variable; that variable quietly receded into the shadows of the underlying implementation.

Using method references

You can make the code be just a bit more concise by using a feature called method reference. The Java compiler will take either a lambda expression or a reference to a method where an implementation of a functional interface is expected. With this feature, a short `String::toUpperCase` can replace

```
name -> name.toUpperCase()
```

as follows:

```
friends.stream()
    .map(String::toUpperCase)
    .forEach(name -> System.out.println(name))
```

Java knows to invoke the `String` class's given method `toUpperCase()` on the parameter passed in to the synthesized method—the implementation of the functional interface's abstract method. That parameter reference is implicit here. In simple situations such as the previous example, you can substitute method references for lambda expressions; I'll explain that in a moment.

In the preceding example, the method reference was for an instance method. Method references can also refer to `static` methods and methods that take parameters. I'll show examples of these later.

Lambda expressions helped enumerate a collection and transform it into a new collection. Lambdas can also help you concisely pick an element from a collection, coming up next.

**When should you use method references?** I typically use lambda expressions much more often than method references when programming in Java. That doesn't mean method references are unimportant or less useful, though. They are nice replacements when the lambda expressions are short and make simple, direct calls to either an instance method or a static method. In other words, if lambda expressions merely pass their
parameters through, you can replace them with method
references.

These candidate lambda expressions are much like Tom
Smykowski, in the movie *Office Space*, whose job is to “take
specifications from the customers and bring them down to the
software engineers.” For this reason, I call the refactoring of
lambdas to method references the *office-space pattern*.

In addition to conciseness, with method references you gain the
ability to use more directly the names already chosen for these
methods.

There’s quite a bit of compiler magic taking place under the hood
with method references. The method reference's target object
and parameters are derived from the parameters passed to the
synthesized method. This makes the code with method
references much more concise than the code with lambda
expressions. However, you can’t use this convenience if the
application logic requires manipulating parameters before
sending them as arguments or tinkering with the call’s results
before returning them.

**Finding elements in a collection**

The elegant methods used to traverse and transform collections
will not directly help you pick elements from a collection. The
`filter()` method is designed for that purpose.

Imagine that from a list of names, you need to pick the ones that
start with the letter *N*. Because there may be zero matching
names in the list, the result may be an empty list. First, here’s
how to code it using the old approach.

```java
final List<String> startsWithN =
    new ArrayList<String>();
for(String name : friends) {
    if(name.startsWith("N")) {
        startsWithN.add(name);
    }
}
```

That’s a chatty piece of code for a simple task: it created a
variable and initialized it to an empty collection. Then it looped
through the collection, looking for a name that starts with the
desired letter. If found, it added the element to the collection.

Here is how to refactor this code to use the `filter()` method
and see how it changes things.

```java
final List<String> startsWithN =
    friends.stream()
        .filter(name -> name.startsWith("N"))
        .collect(Collectors.toList());
```
The `filter()` method expects a lambda expression that returns a boolean result. If the lambda expression returns `true`, the element in context while executing that lambda expression is added to a result collection; it's skipped otherwise. Finally, the method returns a stream with only elements for which the lambda expression yielded `true`. In the end, you transformed the result into a `List` using the `collect()` method.

Here's how to print the number of elements in the result collection.

```java
System.out.println(
    String.format(
        "Found %d names", startsWithN.size()));
```

From the following output, it's clear that the method picked up the proper number of elements from the input collection:

```
Found 2 names
```

The `filter()` method returns an iterator just like the `map()` method does, but the similarity ends there. Whereas the `map()` method returns a collection of the same size as the input collection, the `filter()` method might not. It might yield a result collection with a number of elements ranging from zero to the maximum number of elements in the input collection. However, unlike `map()`, the elements in the result collection that `filter()` returned are a subset of the elements in the input collection.

**Conclusion**

The conciseness achieved by using lambda expressions so far is nice, but code duplication might sneak in quickly if you're not careful. I'll address this concern in the second part of this article, “Functional programming in Java, Part 2: Lambda reuse, lexical scoping and closures, and reduce().”

**Dig deeper**

- [Behind the scenes: How do lambda expressions really work in Java?](#)
- [Venkat Subramaniam: Java is changing in a responsible manner](#)
- [The Java tutorial for lambda expressions](#)
- [The Java tutorial for method references](#)
- [Java 8: Lambdas, Part 1](#)
- [Java 8: Lambdas, Part 2](#)
Venkat Subramaniam

Venkat Subramaniam is an award-winning author, founder of Agile Developer, Inc., and an adjunct faculty at the University of Houston. Subramaniam is author and co-author of multiple books, including the 2007 Jolt Productivity award winning book *Practices of an Agile Developer*. Subramaniam has trained and mentored thousands of software developers in the US, Canada, Europe, and Asia, and is a regularly-invited speaker at several international conferences. His latest book is *Pragmatic Scala: Create Expressive, Concise, and Scalable Applications*.