Java 8 idioms: Functional alternatives to the traditional for loop

Three newer methods that cut the fuss out of even complex iterations

Venkat Subramaniam

April 30, 2017

The for loop might be ubiquitous, but it isn't irreplaceable. Learn how range, iterate, and limit cut the fuss out of even complex iterations in Java.

Even with all its moving parts, the `for` loop is so familiar that many developers reach for it without thinking. Starting in Java™ 8, we have several strong new methods that help simplify complex iterations. In this article, you will see how to use `IntStream` methods `range`, `iterate`, and `limit` to iterate through ranges and skip values in a range. You'll also learn about the new `takeWhile` and `dropWhile` methods, coming in Java 9.

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.

The trouble with for

The traditional `for` loop was introduced in the first release of the Java language, and its simpler variation, `for-each`, was introduced in Java 5. Most developers prefer `for-each` for everyday iterations, but will still use `for` for things like iterating through a range or skipping values in a range.

The `for` loop is quite capable, but it has too many moving parts. You can see this in even the simplest task of printing a `get set` prompt:

Listing 1. Complex code for a simple task

```
System.out.print("Get set...");
  for(int i = 1; i < 4; i++) {
    System.out.print(i + "...");
  }
```

In Listing 1, we start the loop index variable `i` at 1 and limit it to a value of less than 4. Note that the `for` loop requires us to tell the loop to increment. In this case we've also elected a pre- versus post-increment.

There isn't a lot of code in Listing 1, but what's there is noisy. Java 8 offers a simpler and quieter alternative: `IntStream`'s `range` method. Here's `range` printing the same `get set` prompt from Listing 1:

Listing 2. Simple code for a simple task

```
System.out.print("Get set...");
  IntStream.range(1, 4)
    .forEach(i -> System.out.print(i + "..."));
```

We haven't significantly reduced the amount of code in Listing 2, but we've reduced its complexity. There are two key reasons why:

1. Unlike `for`, `range` doesn't force us to initialize a mutable variable.
2. Iteration happens automatically, so we don't have to define the increment like we do with the loop index.

Semantically, the variable `i` in the original `for` loop is a *mutated variable*. To appreciate the value of `range` and similar methods, it's helpful to understand the consequences of that design.

Mutables vs parameters

The variable `i`, which we defined in our `for` loop, is a single variable that is mutated through each iteration of the loop. The variable `i` in the `range` example is a parameter to the lambda expression, so it's a brand new variable in each iteration. It's a small difference, but sets the two pieces of code worlds apart. The following examples will help clarify.

The `for` loop in Listing 3 wants to use the index variable in an inner class:

Listing 3. Using an index variable in an inner class

```
ExecutorService executorService = Executors.newFixedThreadPool(10);

      for(int i = 0; i < 5; i++) {
        int temp = i;

        executorService.submit(new Runnable() {
          public void run() {
            //If uncommented the next line will result in an error
            //System.out.println("Running task " + i);
            //local variables referenced from an inner class must be final or effectively final

            System.out.println("Running task " + temp);
          }
        });
      }

      executorService.shutdown();
```

Here we have an anonymous inner class that implements the `Runnable` interface. We want to access the index variable `i` in the `run` method, but the compiler won't permit it.

As a workaround for this restriction, we might create a local temporary variable like `temp`, which is a copy of the index variable. The variable `temp` is created with new each iteration. Prior to Java 8, we would need to mark the variable as `final`. Starting with Java 8, it would be considered effectively final because we are not changing it. Either way, that extra variable in the `for` loop arises solely due to the fact that the index variable is a single mutated variable through the iteration.

Now let's try resolving the same problem using the `range` function.

Listing 4. Using a lambda parameter in an inner class

```
ExecutorService executorService = Executors.newFixedThreadPool(10);

     IntStream.range(0, 5)
       .forEach(i ->
         executorService.submit(new Runnable() {
           public void run() {
             System.out.println("Running task " + i);
           }
         }));

     executorService.shutdown();
```

Received as a parameter to the lambda expression, the index variable `i` does not have the same semantics as the loop index variable. Much like the `temp` that we created by hand in Listing 3, this parameter `i` shows up as a brand new variable through every iteration. It is *effectively final*, since we're not changing its value anywhere. Hence, we can directly use it from within the context of the inner class—no fuss, no muss.

Since `Runnable` is a functional interface, we can easily replace the anonymous inner class with a lambda expression, like so:

Listing 5. Replacing the inner class with a lambda expression

```
IntStream.range(0, 5)
       .forEach(i ->
         executorService.submit(() -> System.out.println("Running task " + i)));
```

It's clear there are benefits to using `range` rather than `for`, for relatively simple iterations, but `for` is especially valued for its ability to handle more complex iteration scenarios. Let's see how `range` and other Java 8 methods compare.

Closed ranges

When creating a `for` loop, we can direct the index variable to close in on a range, like so:

Listing 6. A for loop with a closed range

```
for(int i = 0; i <= 5; i++) {
```

The index variable `i` takes the values `0`, `1`, ...`5`. Rather than use `for`, we could use the `rangeClosed` method. In this case, we'd direct `IntStream` to close over the ending value in the range:

Listing 7. The rangeClosed method

```
IntStream.rangeClosed(0, 5)
```

When we iterate over this range, we'll get a value that includes the boundary value of 5.

Skipping values

The `range` and `rangeClosed` methods are simpler and more fluent alternatives to `for` for basic looping, but what if we want to step over some values? In this case `for`'s demand for upfront effort makes the operation rather easy. In Listing 8, the `for` loop quickly skips two values while iterating:

Listing 8. Skipping values with for

```
int total = 0;
for(int i = 1; i <= 100; i = i + 3) {
  total += i;
}
```

The loop in Listing 8 computes the sum of every third value between 1 and 100—a somewhat complex operation that was rather easy using `for`. Could we also solve this using `range`?

At first, you might consider using the `range` method of `IntStream`, combined with either `filter` or `map`. That would involve more work than using a `for` loop, however. A more likely solution is to combine `iterate` with `limit`:

Listing 9. Iterating with limit

```
IntStream.iterate(1, e -> e + 3)
  .limit(34)
  .sum()
```

The `iterate` method is quite easy to use; it just takes an initial value to start iterating. The lambda expression that is passed as the second argument determines the next value in the iteration. This is similar to Listing 8, where we passed an expression to the `for` loop to increment the value of the index variable. However, in this case, there's a *gotcha*. Unlike `range` and `rangeClosed`, nothing tells the `iterate` method when to stop. If we don't limit the value, the iteration will be unstoppable.

How could we work around this?

We're interested in the values between 1 and 100, and we want to skip two values starting with 1. Doing a little math, we figure out that there are 34 desired values in the given range. So, we pass that number to the `limit` method.

The code works, but the process is too complex: doing the math beforehand isn't fun, and it limits our code. What if we decided to skip three values instead of two? We'd not only have to change the code, but our results would be error prone. There has to be a better way.

The takeWhile method

Coming in Java 9, `takeWhile` is a new method that makes it easier to iterate with limits. Using `takeWhile`, we can directly state that iteration should continue *as long as a desired condition is met*. Here's how the iteration from Listing 9 would look using `takeWhile`:

Listing 10. Iterating with conditions

```
IntStream.iterate(1, e -> e + 3)
     .takeWhile(i -> i <= 100) //available in Java 9
     .sum()
```

Instead of limiting the iteration to a pre-computed number, we dynamically determine when to break out of the iteration, using the condition provided to `takeWhile`. This approach is much easier and less error prone than trying to pre-compute the number of iterations.

The `takeWhile` method, along with its counterpart `dropWhile`, which skips values until a given condition is met, are much needed additions to the JDK. The `takeWhile` method acts like a break, while `dropWhile` acts like a continue. Starting in Java 9, they'll be available for any type of `Stream`.

Iterating in reverse

Iterating in reverse is nearly effortless compared to iterating forward, regardless of whether you use a traditional `for` loop or `IntStream`.

Here's a `for` loop iterating in reverse:

Listing 11. Iterating in reverse using for

```
for(int i = 7; i > 0; i--) {
```

The first argument in `range` or `rangeClosed` can't be greater than the second argument, so we're unable to use either of these methods to iterate in reverse. Instead, we can use the `iterate` method:

Listing 12. Iterating in reverse using iterate

```
IntStream.iterate(7, e -> e - 1)
     .limit(7)
```

We pass a lambda expression as an argument to the `iterate` method, which decrements the given value to move the iteration in the reverse direction. We use the `limit` function to specify how many total values we want to see during the reverse iteration. If necessary, we could also use the `takeWhile` and `dropWhile` methods to dynamically alter the flow of iteration.

Conclusion

While the traditional `for` loop is very powerful, it is also overly complex. Newer methods in Java 8 and Java 9 can help simplify iteration, even for sophisticated iterations. The methods `range`,

`iterate`, and `limit` have fewer moving parts, which will help you code more efficiently. These methods also resolve Java's longstanding requirement that local variables must be declared final in order to be accessed from inner classes. Exchanging a single mutable index variable for an effectively final parameter is a small semantic difference, but it cuts out a lot of garbage variables. The end result is simpler, more elegant code.

Related topics

- Functional Programming in Java: The Pragmatic Bookshelf, 2014
- Java Streams, Part 1: Introducing java.util.stream
- Java Collections: Mutables to watch out for
- Iterating over collections in Java 8
- IBM Code: Java journeys