Sequences for Lazy Evaluation

We'll cover the following

- Improve performance with sequences
- Infinite sequences

Simply put, collections are eager and sequences are lazy. Sequences are optimized wrappers around collections intended to improve performance. In this section, we'll look at what that means and how to decide when to use collections and when to use sequences.

Unlike Java, the methods like `filter()`, `map()`, and so on are available in Kotlin directly on collections like `List<T>`, instead of only on a `Stream<T>`. The designers of Java made a decision against offering these methods on collections for performance reasons. The designers of Kotlin, on the other hand, decided to lean toward convenience, and they expect us to choose wisely.

Use the internal iterators on collections directly in Kotlin when the collection size is small. For larger collections, use the internal iterators through a sequence. The reason for this is that unlike operations on collections that are evaluated eagerly, the function calls on sequences are evaluated lazily. Laziness defers execution of code to the point where they may be eliminated if unnecessary. That optimization can save time and resources that may otherwise be spent on computations whose results may never be used. And laziness also makes it possible to create infinite sequences that are evaluated on demand. We'll explore these ideas and discuss their benefits here.

Improve performance with sequences

Let's revisit the example from the previous section where we obtained the first adult in a given list. We'll use the same `Person` class and the `people` list we created earlier. Using the internal iterators `filter()`, `map()`, and `first()`, we get the name of the first adult in the group. Instead of passing lambdas to `filter()` and `map()`, we'll pass function references. The `filter()` method takes a reference to an

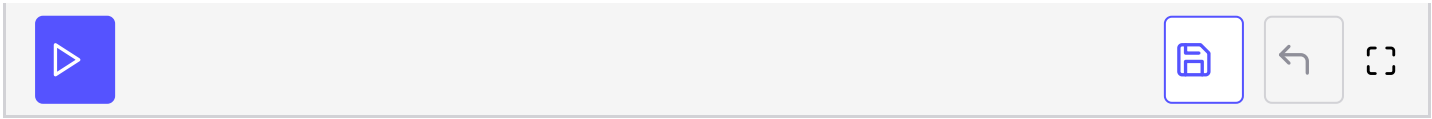
we'll pass function references. The `filter()` method takes a reference to an

`isAdult()` function, and the `map()` method takes a reference to the `fetchFirstName()` function.

```
fun isAdult(person: Person): Boolean {
    println("isAdult called for ${person.firstName}")
    return person.age > 17
}
fun fetchFirstName(person: Person): String {
    println("fetchFirstName called for ${person.firstName}")
    return person.firstName
}

val nameOfFirstAdult = people
    .filter(::isAdult)
    .map(::fetchFirstName)
    .first()

println(nameOfFirstAdult)
```



As you would expect, the value stored in `nameOfFirstAdult` is **Jill**, no surprises there. But the effort to get that result is significant. The `filter()` method executes eagerly to call the `isAdult()` function and creates a list of adults. Then the `map()` method also executes eagerly to call the `fetchFirstName()` function and creates yet another list of names of adults. Finally, the two intermediate temporary lists are discarded when `first()` gets just the first element from the list returned by `map()`. Although only one value is expected as the final result, the execution ends up performing a lot of work, as we can see from the output:

```
isAdult called for Sara
isAdult called for Jill
isAdult called for Paula
isAdult called for Paul
isAdult called for Mani
isAdult called for Jack
isAdult called for Sue
fetchFirstName called for Jill
fetchFirstName called for Paula
fetchFirstName called for Paul
fetchFirstName called for Jack
Jill
```

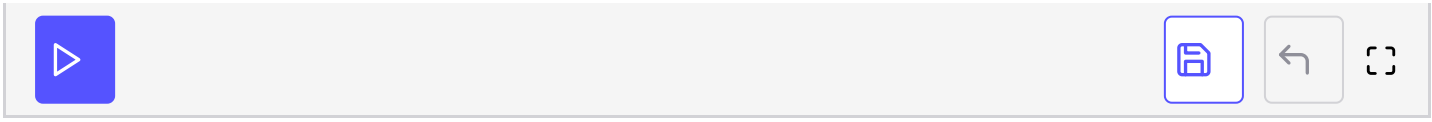
This was a small collection, but what if we had hundreds of thousands of elements

This was a small collection, but what if we had hundreds of thousands of elements in the collection. That would result in a lot of computations whose results aren't used in the end—what a waste.

This is where the lazy evaluation capabilities of *sequences* come in. We can wrap a collection into a sequence using the `asSequence()` method and then apply the same internal iterator methods that we used on the collection, but this time on the sequence. Let's change the previous code to do that.

```
val nameOfFirstAdult = people.asSequence()
    .filter(::isAdult)
    .map(::fetchFirstName)
    .first()

println(nameOfFirstAdult)
```



The only difference between the previous code and this one is the call to `asSequence()` before the `filter()` call. A small change but a huge gain in performance, as we see from the output:

```
isAdult called for Sara
isAdult called for Jill
fetchFirstName called for Jill
Jill
```

Instead of eagerly creating a list of adults, the `filter()` method, when called on a sequence, returns another sequence. Likewise, the call to `map()` on a sequence returns yet another sequence. But the lambdas passed to `filter()` or `map()` haven't been called yet. When the `first()` method is called, the actual evaluation that was deferred so long is triggered. Unlike the other methods on sequence that return a sequence, the terminal methods like `first()` return a result of executing the pipeline of operations. By nature, sequences defer evaluation until a terminal method is called and then minimally perform operations to get the desired result.

Both the direct use of internal iterators on collections and their use via sequences lead to elegant functional-style code. Whereas the computations are performed eagerly when internal iterators are run on collections, the same operations execute lazily when run on sequences.

Since sequences do less work, should we use them all the time instead of calling

Since sequences do less work, should we use them all the time instead of calling the internal iterators on collections directly? The short answer is no. If the collection is very small, the difference in performance will be rather negligible. Eager evaluation, which is easier to debug and easier to reason, may be better in that case. However, if the collection size is large, in the hundreds of thousands of elements, then using sequence will remove the significant overhead of creating intermediate collections and eliminated computations.

Infinite sequences

Performance isn't the only benefit of laziness. Laziness can also help perform on-demand computation and that, in turn, can help to create infinite or unbounded sequences of elements. An infinite sequence starts with a value and is followed by a sequence of numbers that follow a particular pattern of occurrence. For example, the Fibonacci sequence `1, 1, 2, 3, 5, 8,...` follows the pattern that the value at a position is equal to the sum of values at the previous two positions. The numbers `2, 4, 6, 8, 10,...` form an infinite sequence of even numbers that start with `2`.

Kotlin provides a few different ways to create an infinite sequence. The `generateSequence()` function is one way. Let's use that function to create an infinite sequence of prime numbers as a way to illustrate the power of sequences to create unbounded sequences of values. We'll start with some convenience functions to lead up to the use of `generateSequence()`.

Given a number `n``, the` isPrime()`function returns true if the number is prime and`false` otherwise:

```
// primes.kts
fun isPrime(n: Long) = n > 1 && (2 until n).none { i -> n % i == 0L }
```

The `nextPrime()` function takes a number n and returns the prime number after that value. For example, if we provide `5` or `6` as input, it will return the next prime number `7`. The method is marked as `tailrec` to prevent the possibility of `StackOverflowError`—see Chapter 15, Programming Recursion and Memoization, for a discussion of tail call optimization.

```
// primes.kts
tailrec fun nextPrime(n: Long): Long =
  if (isPrime(n + 1)) n + 1 else nextPrime(n + 1)
```

Given the above two functions, we can create an infinite sequence of prime numbers starting with any prime number, using the `generateSequence()` function, like so:

```
// primes.kts
val primes = generateSequence(5, ::nextPrime)
```

One overloaded version of the `generateSequence()` function takes a seed value as the first parameter and a function, a lambda expression, as the second parameter. The lambda takes in a value and returns a result value. In this example, we use a function reference to the `nextPrime()` function which, given a number, returns the next prime number. Thus, `primes` holds an infinite sequence of prime numbers starting with `5`.

In the call to `generateSequence()`, if the second argument, which is a call to the `nextPrime()`, were executed immediately and repeatedly, then we'd end up with an infinite collection of values. Such execution doesn't make sense both from the time and space points of view. The trick here is that `generateSequence()` is lazy and it doesn't execute the `nextPrime()` function until we ask for the values. We can ask for any number of values using the `take()` method. This method gives us a view into the regions of the infinite sequence that we're interested in viewing or using. Let's use the `take()` method to get `6` values from the `primes` infinite series of prime numbers.

```
// primes.kts
System.out.println(primes.take(6).toList()) //[5, 7, 11, 13, 17, 19]
```

The call to `toList()` triggers the evaluation of the elements in the sequence, but `take()` produces only the given number of values.

With the ability to create infinite sequences, we can model problems with lazy evaluation to create sequence of values whose bounds aren't known ahead of time.

Instead of writing the `nextPrime()` recursive function and then using the `generateSequence()` function, we can also use the `sequence()` function. This function takes a lambda that runs as a continuation, which is an advanced and relatively new topic in Kotlin. We'll dive extensively into continuations and coroutines later in this book. For now, when you see the `yield()` call, read it as

coroutines later in this book. For now, when you see the `yield()` call, read it as "return a value to the caller and then continue executing the next line of code." In other words, continuations give an illusion of writing a function with multiple return points.

Let's replace the `nextPrime()` function and the call to `generateSequence()` in the previous example with the following:

```
val primes = sequence {
    var i: Long = 0
        while (true) {
            i++
            if (isPrime(i)) {
                yield(i)
        }
    }
}
```

The result of the call to `sequence()` is an instance that implements the `Sequence` interface. Within the lambda provided to the `sequence()` function, we have an infinite loop that yields prime numbers. The code within the lambda is executed on demand, only when a value is requested or taken from the sequence. The iteration starts with the value `i` being `0` and the first prime number yielded by the code is `2`. We can take as many values as we like from this infinite sequence, but we can also skip or drop some values. Since the sequence of primes starts with `2`, let's use `drop()` to throw away a couple of initial values and then take six values after that:

```
println(primes.drop(2).take(6).toList()) //[5, 7, 11, 13, 17, 19]
```

If you have a separate function, like `nextPrime()`, that can produce the next value in a sequence, then use that function with `generateSequence()` to generate an infinite sequence. On the other hand, if you want to combine the code to generate the next value in a sequence with the code to create the infinite sequence, then use the `sequence()` function.

The next lesson concludes the discussion for this chapter.