Creating Infinite Sequences

We'll cover the following

Since coroutines are useful for creating cooperating tasks, we can use them to create an infinite series or unbounded values and process the generated values at the same time. A function may create a value in the series and yield it to the code that is expecting the value. Upon consuming the value, the calling code can come back asking for the next value in the series. These steps can continue in tandem until either the code that produces the series exits or the caller doesn't bother asking for another value in the series. We'll take a look at two different functions available in Kotlin to create infinite series.

Using sequence

The Kotlin library has a `sequence` function that's readily available for creating a series of values. We'll use that to create an infinite series of prime numbers, starting from a given number.

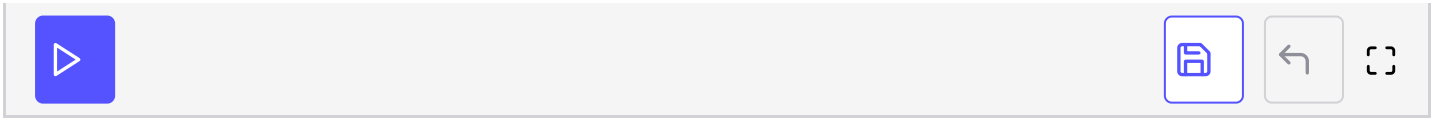
Here's a `primes()` function that takes a starting number and returns a `Sequence<Int>`. The Sequence acts as a continuation, yielding values for iteration.

```
fun primes(start: Int): Sequence<Int> = sequence {
  println("Starting to look")
  var index = start

  while (true) {
    if (index > 1 && (2 until index).none { i -> index % i == 0 }) {
      yield(index)
      println("Generating next after $index")
    }

    index++
  }
}
```

primes.kts

Within the lambda passed to the `sequence()` function, we look for the next prime value and yield it using a `yield()` method—this one is part of the standard library and is different from the `yield()` we used from the `kotlinx.coroutines` library. We can iterate over the values generated by a Sequence, much like the way we iterate over `List`, `Set`, and so on.

```
for (prime in primes(start = 17)) {
  println("Received $prime")
  if (prime > 30) break
}
```



primes.kts

The fact that coroutines and continuations are involved here isn't visible in the code. We iterate over the values, one at a time, print a prime received from `primes()`, and decide to get the next element or break. At every step of the iteration, the flow of control jumps right into the `primes()` function, right back to where it left the last time. We can see this from the output, which has extra print messages to illustrate this behavior.

```
Starting to look
Received 17
Generating next after 17
Received 19
Generating next after 19
Received 23
Generating next after 23
Received 29
Generating next after 29
Received 31
```

The `sequence()` function offers three benefits. First, you don't have to create a collection of values ahead of time, so you don't have to know how many values to compute—you can create the values on the fly. Second, you can amortize the cost of creating the values over time and let the values generated so far be used. Third, since the creation of a value in the series happens only on demand—that is, lazily—we can avoid creating values that may never be used. That results in more

efficient code. It's also easier to apply this technique to write custom Iterable/Iterator classes.

In addition to creating a `Sequence<T>`, you may also easily implement an `Iterator<T>` using Kotlin's `iterator()` function. Let's take a look at that next.

Using `iterator()`

Back in Injecting into Third-Party Classes, we added an extension function into the `ClosedRange<String>` class to iterate over a range of `String` values. For convenience, that code is repeated here:

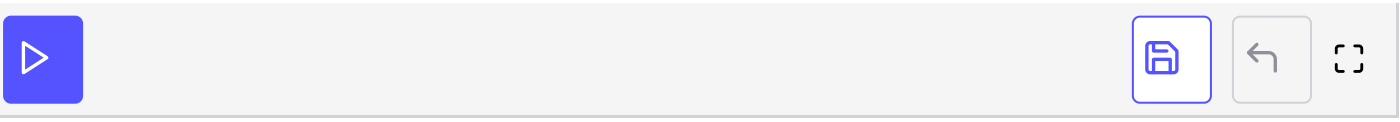
```
operator fun ClosedRange<String>.iterator() =
  object: Iterator<String> {
    private val next = StringBuilder(start)
    private val last = endInclusive

    override fun hasNext() =
      last >= next.toString() && last.length >= next.length

    override fun next(): String {
      val result = next.toString()

      val lastCharacter = next.last()

      if (lastCharacter < Char.MAX_VALUE) {
        next.setCharAt(next.length - 1, lastCharacter + 1)
      } else {
        next.append(Char.MIN_VALUE)
      }

      return result
    }
  }
```

forstringrange.kts

The `iterator()` function of `ClosedRange<String>` is returning an object that implements the `Iterator<String>` interface. The implementation has a couple of properties and two methods. The `hasNext()` tells us if there's another value for iteration, and the `next()` function returns the next value when called. If we step back from the implementation and look at the overall goal, what we want here is an iterator that *yields* one value at a time until it reaches some termination value. This fits the bill of coroutines well.

What we need is an iterator that will run as a coroutine. That's the purpose of the method `iterator()` in the Kotlin standard library. Let's rewrite the above function using `iterator()` and then call it to see its behavior.

```
operator fun ClosedRange<String>.iterator(): Iterator<String> = iterator {
  val next = StringBuilder(start)
  val last = endInclusive

  while (last >= next.toString() && last.length >= next.length) {
    val result = next.toString()

    val lastCharacter = next.last()

    if (lastCharacter < Char.MAX_VALUE) {
      next.setCharAt(next.length - 1, lastCharacter + 1)
    } else {
      next.append(Char.MIN_VALUE)
    }

    yield(result)
  }
}

for (word in "hell".."help") { print("$word, ") }
```



forstringrange.kts

Unlike the `sequence` function which returned a `Sequence<T>`, the `iterator()` function returns an `Iterator<T>`. Within the lambda passed to the `iterator()` function, we loop through to generate the next `String` in the sequence and call the `yield()` function, much like the way we did in the example that used `sequence()`, to return a generated `String` to the caller. In this latest version, the `hasNext()` function disappeared and instead of return, which we used in the `next()` method, we use `yield()` in the `iterator()` function.

In the last line we exercise the `ClosedRange<String>`'s `iterator()` function using the `for` loop to iterate over a range of values. The output of this code is the same as the output of the other version this code replaces.

Between the `sequence()` function and the `iterator()` function, Kotlin has you covered for creating coroutines to generate unbounded values with highly expressive code.

The next lesson concludes the discussion for this chapter.