Range and Iteration

We'll cover the following

- Range classes
- Forward iteration
- Reverse iteration
- Skipping values in a range

Imagine telling someone to count from one to five by uttering “set `i` equal to `1` but while keeping `i` less than `6`, increment `i` and report the value.” If we had to communicate with a fellow human that way, it would have ended civilization a long time ago. Yet that’s how programmers have been writing code in many C-like languages. But we don’t have to, at least not in Kotlin.

Range classes

Kotlin raises the level of abstraction to iterate over a range of values with specialized classes. For instance, here’s a way to create a range of numbers from *1 to 5*.

```
// ranges.kts
val oneToFive: IntRange = 1..5
```

The type `IntRange`, which is part of the `kotlin.ranges` package, is provided for clarity, but you may leave it out and let type inference figure out the variable’s type.

If you want a range of letters in the English alphabet, the process is the same:

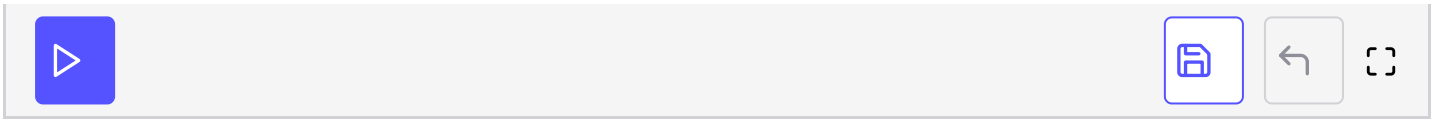
```
// ranges.kts
val aToE: CharRange = 'a'..'e'
```

You’re not limited to primitives like `int`, `long`, and `char`. Here’s a range of strings:

```
// ranges.kts
val seekHelp: ClosedRange<String> = "hell".."help"
```

That's nifty. Take the initial value, place the `..` operator, followed by the last value in the range. The range includes both the values before and after the `..` operator. Let's quickly check if a couple of values exist in that range.

```
println(seekHelp.contains("helm")) //true
println(seekHelp.contains("helq")) //false
```



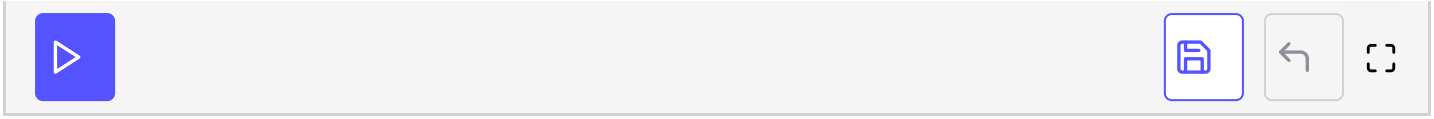
ranges.kts

The first call verifies that the range includes the value "helm", which is in the lexical order of words between the values "hell" and "help". The second call confirms that the value "helq" is not in the range.

Forward iteration

Once we create a range, we can iterate over it using the `for (x in ..)` syntax. Let's iterate over a range of values *1 to 5*.

```
for (i in 1..5) { print("$i, ") } //1, 2, 3, 4, 5,
```



ranges.kts

Not only is the syntax elegant, it's safe as well. As you may have guessed—even though we didn't explicitly say it—the variable `i` is a `val` and not a `var`; that is, we can't mutate the variable `i` within the loop. And, of course, the variable i's scope is limited; it's not visible outside of the loop.

Likewise we can iterate over the range of characters:

```
// ranges.kts
for (ch in 'a'..'e') { print(ch) } //abcde
```

All that went well, but the iteration from "hell" to "help", like the following, will run into issues:

```
for (word in "hell".."help") { print("$word, ") }
//ERROR //for-loop range must have an 'iterator()' method
```

The reason for the failure is whereas classes like `IntRange` and `CharRange` have an `iterator()` function/operator, their base class `ClosedRange<T>` doesn't. But brave-at-heart programmers won't sulk and take no for an answer. Stay tuned—we'll create an extension function to iterate over elements of `ClosedRange<String>` in Injecting into Third-Party Classes.

Reverse iteration

The previous examples showed iterating forward, but we should be able to iterate over the range in reverse just as easily. Creating a range of values `5..1` won't cut it, though. This is where `downTo` comes in.

Let's iterate from 5 down to 1:

```
for (i in 5.downTo(1)) { print("$i, ") } //5, 4, 3, 2, 1,
```

reverse.kts

The call to the `downTo()` method creates an instance of `IntProgression`, which is also part of the `kotlin.ranges` package. That works, but it's a bit noisy. We can remove the dot and parenthesis near `downTo` with the infix notation to make the code easier to read—we'll see in Function Fluency with infix, how to make dot and parenthesis optional for our own code. Let's rewrite the iteration with less noise:

```
// reverse.kts
for (i in 5 downTo 1) { print("$i, ") } //5, 4, 3, 2, 1,
```

Both `..` and `downTo()` produced a range of every single value from the start to the end value. It's not uncommon to skip some values in the range, and there are other methods to achieve that.

Skipping values in a range

When iterating over a range of numbers, you can skip the ending value in the range by using `until()`. Unlike `..`, the `until()` method will stop one value shy of the end value. Just like the `downTo()` method, we can drop the dot and parenthesis

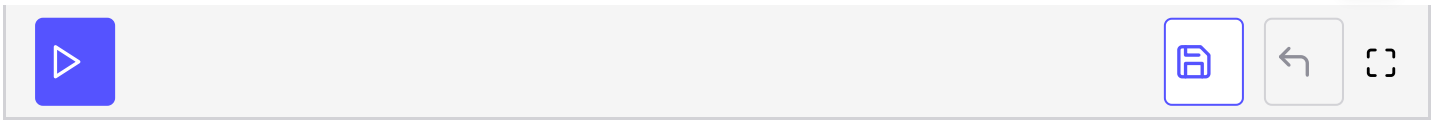
when using the `until()` method. Let's give that a try:

```
// skipvalues.kts
for (i in 1 until 5) { print("$i, ") } //1, 2, 3, 4,
```

This iteration created using until() didn't include the ending value 5, whereas the range we created previously using ... did.

In the traditional `for-loop` in C-like languages we can skip some values by using, for example, `i = i + 3`, but that's mutating `i` and we know that's a *no-no* in Kotlin. To skip some values during iteration, Kotlin provides a `step()` method—that's definitely a step in the right direction for fluency and fewer errors. Let's use the `step()` method, again with the infix notation for fluency:

```
for (i in 1 until 10 step 3) { print("$i, ") } //1, 4, 7,
```



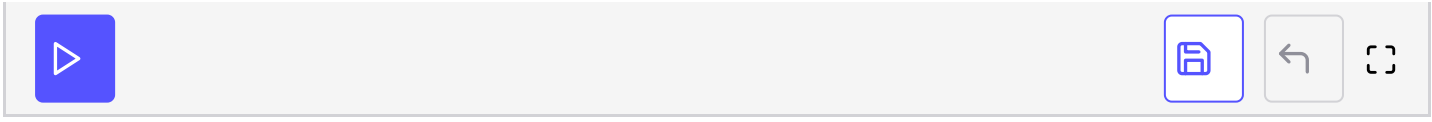
skipvalues.kts

The `step()` method transforms an `IntRange` or `IntProgression` created using `..`, until, `downTo`, etc. into an `IntProgression` that skips some values. Let's use `step()` to iterate in reverse order while skipping some values:

```
// skipvalues.kts
for (i in 10 downTo 0 step 3) { print("$i, ") } //10, 7, 4, 1,
```

That was an easy way to methodically skip some values, but there are other methods to skip values that don't fall into a rhythm. For example, if you want to iterate over all values divisible by 3 and 5, use the `filter()` method:

```
for (i in (1..9).filter { it % 3 == 0 || it % 5 == 0 }) {
    print("$i, ") //3, 5, 6, 9,
}
```



skipvalues.kts

The `filter()` method takes a predicate—a lambda expression—as argument. We'll discuss lambdas and functional style later.

discuss lambdas and functional style later.

QUIZ

1

What will be the output of the following code snippet?

```
val compiler: ClosedRange<String> = "kotlic".."kotlin"

println(compiler.contains("kotlinc"))
println(compiler.contains("kotlin"))
```

2

What will be the output of the following code snippet?

```
for (letter in "a".."e") { print("$letter, ") }
```

Retake Quiz

So far you've seen rudimentary iteration over a range of values. In the next lesson,

let’s take a look at iterating over a collection of values.