Closures and Lexical Scoping

We'll cover the following

- Understanding closures and lexical scoping
- Mutable variables

Functional programmers talk about lambdas and closures. Many programmers use those two terms interchangeably, which is acceptable, as long as we know the difference and can discern which one we're using based on the context. Let's take a closer look at how lambdas bind to variables and how that relates to the concept of closures.

Understanding closures and lexical scoping

A lambda is stateless; the output depends on the values of the input parameters. For example, the output of the following lambda is twice the value of the given parameter:

```
// closures.kts
val doubleIt = { e: Int -> e * 2 }
```

Sometimes we want to depend on external state. Such a lambda is called a *closure*—that's because it closes over the defining scope to bind to the properties and methods that aren't local. Let's turn the previous lambda into a closure.

```
// closures.kts
val factor = 2

val doubleIt = { e: Int -> e * factor }
```

In this version, `e` is still the parameter. But within the body, the variable or property `factor` isn't local. The compiler has to look in the defining scope of the closure—that is, where the body of the closure is defined—for that variable. If it doesn't find it there, the compiler will have to continue the search in the defining

scope of the defining scope, and so on. This is called *lexical scoping*.

In our example, the compiler binds the variable factor within the body of the closure to the variable right above the closure—that is, `val factor = 2`.

We used lexical scoping earlier in Function Returning Functions, and it felt so natural that it may have gone unnoticed. Here's the relevant code from that section, repeated for your convenience:

```
fun predicateOfLength(length: Int): (String) -> Boolean {
  return { input: String -> input.length == length }
}
```

The lambda being returned has a parameter named `input`. Within the body of the lambda we compare the value of the `length` property of this `input` parameter with the value in the `length` variable. But, the variable `length` isn't part of the lambda—it's from the closure's lexical scope, which happens to be the parameter passed to the `predicateOfLength()` function.

Mutable variables

Mutability is taboo in functional programming. However, Kotlin doesn't complain if from within a closure we read or modify a mutable local variable. In the earlier example, the variable `factor` is immutable since it's defined as a `val`. If we change it to a `var`—that is, turn it into a mutable variable—then potentially we may modify `factor` from within the closure. The Kotlin compiler won't warn in this case, but the result may surprise or, at the very least, confuse the reader.

In the following code we transform values in two collections, one created using `listOf()` and the other created using `sequenceOf()`. After that, we modify the value in the variable factor and, finally, print the values of the transformed collections. Without running the code, try guessing the output of the code.

```
var factor = 2

val doubled = listOf(1, 2).map { it * factor }
val doubledAlso = sequenceOf(1, 2).map { it * factor }

factor = 0

doubled.forEach { println(it) }
doubledAlso.forEach { println(it) }
```

mutable.kts

Is it `2, 4, 2, 4` on separate lines, `0, 0, 0, 0`, or `2, 4, 0, 0`?

It's certainly not worth the trouble thinking through the different behaviors of list vs. sequence (which we'll cover in Chapter 12, Internal Iteration and Lazy Evaluation), even though the structure of the code to transform the two collections are so similar. Using mutable variables from within a closure is often a source of error and should be avoided. Keep closure as pure functions to avoid confusion and to minimize errors—see Prefer val over var.

QUIZ

Q

Can the following lambda be considered a closure?

```
val value = 2

val doubleIt = { i: Int -> i * 2 }
```

Retake Quiz

In the next lesson, we'll learn about the return keyword in lambdas.