Implicit Receivers

We'll cover the following

Unlike the `let()` and `also()` methods, the `run()` and `apply()` methods executed their lambdas in the context of their own target. What's cool about Kotlin is that the language and library implementors didn't keep that as a privileged execution for themselves. Instead, they made it accessible very elegantly to every one using the language. If you're eager to learn and execute such methods for your own code, the wait is over. And this technique is one of the most essential to create fluent syntax for DSLs, so it's all the more exciting.

Passing a receiver

In JavaScript, functions may take zero or more parameters, but you may also pass a context object—a receiver—using either the `call()` method or `apply()` function. Kotlin's ability to tie a receiver to a lambda expression is heavily influenced by that JavaScript capability.

Before we dive into dealing with receivers, let's look at a regular lambda expression for a moment.

```
var length = 100

val printIt: (Int) -> Unit = { n: Int ->
  println("n is $n, length is $length")
}

printIt(6)
```

lambda.kts

In this code, `printIt` refers to a lambda, which receives an `Int` as its parameter and returns nothing. Within the lambda we print a property named `length` and the value of the parameter `n`. It's clear what `n` is, but where does `length` come from? The variable `length` isn't in the internal scope of the lambda. Thus, the compiler binds `length` to the variable in the lexical scope—that is, from above the lambda expression. Let's confirm that from the output:

```
n is 6, length is 100
```

Kotlin gives us a nice way to set a receiver for lambdas. To do this we have to change the signature of the lambda a little. Let's do that in the next example:

```
var length = 100

val printIt: String.(Int) -> Unit = { n: Int ->
  println("n is $n, length is $length")
}

printIt("Hello", 6)
```

lambdareceiver.kts

We still have the `length` property in the scope of the script. The only change in the definition of the lambda is in the parameter signature: `String.(Int) -> Unit` instead of `(Int) -> Unit`. The syntax `String.(Int)` says that the lambda will execute in the context of an instance of String. If you have more than one parameter for the lambda, for example, (`Int`, `Double`), then to say that the lambda will get a receiver during a call, write it as `Type.(Int, Double)`, where `Type` is the type of the receiver. When resolving the scope of variables, the compiler will first look for the variable in the scope of the receiver, if present. If the receiver isn't present or if the variable isn't in the receiver, then the compiler looks for the variable in the lexical scope.

When calling a lambda that expects a receiver, we need to pass an additional argument—that is, the context or the receiver that will be bound to `this` inside the lambda. Here's one way, though not the most elegant way, to achieve that goal:

```
// lambdareceiver.kts
printIt("Hello", 6)
```

The receiver is passed in as the first argument, and the actual parameters of the lambda follow that. That'll work, but it's not the preferred syntax in Kotlin. That's like, in JavaScript, doing `func.call(context, value)` instead of `context.func(value)`. You can invoke the lambda like it's a member function, a method, of the receiver, like so:

```
"Hello".printIt(6)
```

That's wicked cool.

The lambda acts like it's an extension function for the receiver. In fact, that's exactly what Kotlin did. It treated the lambda as an extension function of the receiver.

Whether we pass the receiver as a parameter (and risk being chided as boring) or use it as a target of the call, the `this` within the lambda now refers to the receiver passed instead of the lexical `this`. As a result, the access to property `length` will be the length of the target receiver "Hello" and not the `length` property defined above the lambda. Let's verify this by running the modified version of the lambda with the receiver.

```
n is 6, length is 5
```

The output shows the `length` of the receiver `String` and not the value from the lexical scope. The receiver may be an object of any type, not only a `String`. We'll exploit this feature when building DSLs in the next chapter.

Multiple scopes with receivers

Lambda expressions may be nested within other lambda expressions. In this case, the inner lambda expression may appear to have multiple receivers: its direct receiver and the receiver of its parent. If the parent itself were nested into another lambda expression, then the innermost lambda expression may appear to have three or more receivers. In reality, a lambda has only one receiver but may have multiple scopes for binding to variables, depending on the levels of nesting. This concept of perceived multiple receivers for lambdas is much like the use of receivers we saw for extension functions defined within classes in Injecting from

within a Class, and the receivers of an inner class that we looked at in Nested and Inner Classes. In both of those cases, we were able to refer to the outer receiver using the syntax `this@OuterClassName`. In the case of lambda expressions, though, the outer is a function instead of a class. Thus, we can use the syntax `this@Outer-FunctionName` to refer to the outer scope. Let's examine this with an example.

We'll define two functions, `top()` and `nested()`, that will receive a parameter that's a lambda expression with a receiver. Each of the functions will invoke the lambda expressions provided to them, in the context of a target or receiver object. We'll then examine the receivers by nesting the calls.

```
fun top(func: String.() -> Unit) = "hello".func()

fun nested(func: Int.() -> Unit) = (-2).func()

top {
  println("In outer lambda $this and $length")

  nested {
    println("in inner lambda $this and ${toDouble()}")
    println("from inner through receiver of outer: ${length}")
    println("from inner to outer receiver ${this@top}")
  }
}
```

multiplereceivers.kts

The `top()` function invokes the lambda expression provided with an arbitrary `String` instance "hello" as the receiver. The `nested()` function does the same for its parameter with an arbitrary `Int` value of `-2`. In the call to the `top()` function, we pass a lambda expression that prints its receiver, `this`, and the `length` property of the receiver. That will be the `String` "hello" and the length of 5. Then, within that lambda expression we're calling the `nested()` function, again providing a lambda expression to it.

Within this nested lambda expression we print its receiver, also referenced using `this` and the result of call to the `toDouble()` method. Since the `nested()` function is using `Int` `-2` as the receiver, this here refers to the `Int`, and `toDouble()` is invoked on the `Int`. In the next line, when we read the `length` property, since `Int` doesn't have that property, the call is quietly routed to the receiver of the parent—that is, the nesting, lambda expression. In the event of a property collision between the inner receiver and the parent's receiver or if you want to explicitly refer to the

receiver of the parent, we can use the `@` syntax like in the last line: `this@top`.

Work through the example by hand and then compare the output from your understanding to the output from execution of the following code:

```
In outer lambda hello and 5
in inner lambda -2 and -2.0
from inner through receiver of outer: 5 from inner to outer receiver hello
```

The ability to access both the inner or closest receiver and the outer or parent receiver is powerful and is consistent when working with inner classes, method extensions, and lambdas with receivers.

The next lesson concludes the discussion for this chapter.