Lambda Expressions

We'll cover the following

A few years ago a conference organizer asked if I'd like to talk about Lambada expressions. Dancing is a form of expression, but I'm far from being qualified to talk about it—I chose lambda expressions instead. Lambdas are also a form of expression—they're concise, expressive, and elegant.

Lambdas are short functions that are used as arguments to higher-order functions. Rather than passing data to functions, we can use lambdas to pass a piece of executable code to functions. Instead of using data to make decisions or perform calculations, the higher-order functions can rely on the lambdas to make decisions or perform calculations. In essence, it's like instead of giving someone a fish, you're giving them a fishing lesson. Let's dive into lambdas.

Structure of lambdas

A lambda expression is a function with no name whose return type is inferred. Generally a function has four parts: name, return type, parameters list, and body. Lambdas carry over only the most essential parts of a function—the parameters list and body. Here's the syntax of lambdas in Kotlin:

```
{ parameter list -> body }
```

A lambda is wrapped within `{}`. The body is separated from the parameter list

A lambda is wrapped within `{}`. The body is separated from the parameter list using a hyphenated arrow (`->`), and the body is generally a single statement or expression, but it may also be multiline.

When passing a lambda to a function as argument, avoid the urge to create multiline lambdas unless it's the last argument. Having multiple lines of code in the middle of the argument list makes the code very hard to read, defeating the benefits of fluency we aim to get from lambdas. In such cases, instead of multiline lambdas, use function references, which we'll see in Using Function References.

Passing lambdas

Thinking functionally and using lambdas takes some effort and time to get comfortable with. Given a problem, we have to think in terms of transformational steps and look for functions that can help us with intermediate steps.

For example, let's implement a function, in the functional style, to tell if a given number is a prime number or not. Let's first formulate the problem in words and then translate that into code. Think of the simplest way to find if a number is a prime number—no fancy optimization for efficiency needed here. A number is a prime number if it is greater than 1 and is not divisible by any number in the range between 2 and the number. Let's reword that: a number is a prime number if greater than one and none (nothing) in the range `2 until n` divides the number. The functional-style code becomes visible with one more coat of polish over those words:

```
fun isPrime(n: Int) = n > 1 && (2 until n).none({ i: Int -> n % i == 0 })
```

Let's discuss how the lambda is passed as a parameter in this code and how, in this example, the functional style is as efficient as an imperative implementation would be. The code snippet `2 until n` returns an instance of the `IntRange` class. This class has two overloaded versions of `none()`, where one of them is a higher-order function that takes a lambda as a parameter. If the lambda returns `true` for any of the values in the range, then `none()` returns `false`; otherwise, it returns `true`. The `none()` function will short-circuit the evaluation—that is, anytime a call to the lambda returns `true`, no further elements will be evaluated, and `none()` will immediately return `false`. In short, `none()` will break out of the iteration upon finding a divisor. This behavior shows that `none()`, which is functional-style code, is as efficient as the equivalent imperative- style code for this problem.

The parameter list of the lambda passed to `none()` specifies the type of the parameter, `i: Int`. You know that Kotlin requires the type for each parameter of a function since there's no type inference for parameters. However, Kotlin doesn't insist on types for lambdas' parameters. It can infer the type for these based on the parameter of the function to which the lambdas are passed. For example, the signature `none(predicate: (Int) -> Boolean): Boolean` of the `none()` method says that the lambda passed should have one parameter of type `Int` when `none()` is called on `IntRange`—the actual method uses parametric type `T` which is specialized to `Int` in this context. Thus, we can drop the type from the lambda's parameter list:

```
fun isPrime(n: Int) = n > 1 && (2 until n).none({ i -> n % i == 0 })
```

That reduced the noise a tad, but we can take this one step further. Since the version of `none()` we're using takes only one parameter, we can drop the parenthesis `()` in the call—we'll discuss this further in Use Lambda as the Last Parameter. Let's get rid of the `()` around the argument passed to `none()`:

```
fun isPrime(n: Int) = n > 1 && (2 until n).none { i -> n % i == 0 }
```

Feel free to leave out the type and also `()` where possible. It's less work for your fingers and less parsing for the eyes of everyone who reads the code.

Using the implicit parameter

If the lambdas passed to a function take only one parameter, like `i` in the previous example, then we can omit the parameter declaration and use a special implicit name `it` instead. Let's change the lambda passed to `none()` to use it:

```
fun isPrime(n: Int) = n > 1 && (2 until n).none { n % it == 0 }
```

For short lambdas with only a single parameter, feel free to leave out the parameter declaration and the arrow, `->`, and use it for the variable name. The only downside is you can't quickly tell if a lambda is a no-parameter lambda or if it takes one parameter that's referenced using `it`. Again, if the lambda is extremely short, this isn't a real concern. But what about long lambdas? Lambdas that have many lines are hard to maintain and should be avoided.

Receiving lambdas

We saw how to pass a lambda to a higher-order function. Let's now look at creating a function that receives a lambda. Here's a function that takes an `Int` and a lambda as parameters. It iterates over values from `1` to the given number and calls the given lambda expression with each value in the range. Let's take a look at the code and then discuss the syntax.

```
// iterate.kts
fun walk1To(action: (Int) -> Unit, n: Int) =
  (1..n).forEach { action(it) }
```

In Kotlin we specify the variable name and then the type, like `n: Int` for parameters. That format is used for lambda parameters as well. In this example, the name of the first parameter is `action`. Instead of the type being something simple like `Int`, it defines a function. The function that `walk1To()` wants to take should take an `Int` and return nothing—that is, return `Unit`. This is specified using the transformational syntax `(types list)` `->` `output type`—in this example, `(Int)` `->` `Unit`. The syntax signifies that the function takes some inputs of the type specified to the left of `->` and returns a result of the type specified on the right of `->`.

Let's call this function passing two arguments: a lambda expression and an `Int`:

```
// iterate.kts
walk1To({ i -> print(i) }, 5) //12345
```

The first argument is a lambda, and the type of the parameter `i` of the lambda is inferred as `Int`. The second argument is `5`, which conforms to the required type `Int`.

That code works, but it's a bit noisy. We can improve the signal-to-noise ratio a bit by rearranging the parameters.

Use a lambda as the last parameter

In the previous example the lambda was the first argument. It's not uncommon for a function to take more than one lambda parameter. In the call we separated the first argument, which is of type lambda, from the second argument with a comma. Both the arguments live within the `()`. With the `{}`, comma, and `()`, that call is a

bit noisy. And if the lambda passed as the first argument is going to be more than a single line, the code will be hard to read, with `}, 5)` hanging on a separate line at the end. At the very least, such code is far from being pleasant to read.

Kotlin makes a concession: it bends the rules for a lambda in the trailing position, like how parents are more lenient toward their last child—I know this well because I wasn't that privileged one. But this feature of Kotlin is a good one; it makes the code less noisy. For this reason, when designing functions with at least one lambda parameter, use the trailing position for a lambda parameter.

Let's rearrange the parameter positions in the previous example:

```
fun walk1To(n: Int, action: (Int) -> Unit) = (1..n).forEach { action(it) }
```

After this change, we may still place the lambda within the parenthesis, but as second argument, like so:

```
walk1To(5, { i -> print(i) })
```

However, we can reduce some noise, get rid of the comma, and place the lambda outside the parenthesis:

```
walk1To(5) { i -> print(i) }
```

This is just a bit less noisy than the previous call. But the difference is significant if we plan to write multiline lambdas. For example, let's split the short lambda into multiple lines to get a feel:

```
walk1To(5) {i ->
  print(i)
}
```

We could go a step further if we choose and use implicit `it` instead of `i` here to reduce the noise even further:

```
walk1To(5) { print(it) }
```

In this example we placed the first argument `5` within the `()`. If the function takes only one parameter of a lambda type, then we don't have to use an empty `()`. We

can follow the method's name with the lambda. You saw this in the call to the `none()` method earlier.

Using function references

We've seen how Kotlin bends the rules a bit to reduce noise. But we can take this noise reduction further if the lambdas are pass-through functions.

Glance at the lambdas passed to the `filter()`, `map()`, and `none()` in the earlier examples. Each of them used the parameter received for some comparison or computation, like `e * 2`, for example. But, the lambdas passed to `forEach()` and to `walk1To()` didn't really do anything with their parameters except to pass them through to some other function. In the case of `forEach()`, the lambda passed the parameter `i` to the lambda action. Likewise, within the lambda passed to `walk1To()`, the parameter was passed to `print()`. We can replace pass-through lambdas with references to the functions the parameter is passed through.

For example, look at the following code:

```
({x -> someMethod(x) })
```

We can replace it with this:

```
(::someMethod)
```

If the pass-through is to another lambda, then we don't need the `::`. Let's modify the previous example to use these Kotlin niceties. While at it, we'll expand the example a little to see a few other variations of function references as well.

First, let's modify the `walk1To()` function. We can replace the lambda that's passed to the `forEach()` method with action—that is, we might have the following code:

```
fun walk1To(n: Int, action: (Int) -> Unit) = (1..n).forEach { action(it) }
```

But we can change it to read like this:

```
fun walk1To(n: Int, action: (Int) -> Unit) = (1..n).forEach(action)
```

We got rid of the middle agent that was merely taking the parameter and sending to `action`. No need for a lambda that doesn't add any value—overall less noise, work, and overhead.

Now, in the call to `walk1To()` we can also get rid of the lambda. But, unlike `action`, the function `print()` isn't a lambda. So we can't replace the lambda with print like we replaced the previous lambda with `action`. If a function is qualified to stand in for a lambda, then it should be prefixed with `::`. Let's take a look at the call to `walk1To()` with the lambda and with the function reference to get a clear view of the change:

```
walk1To(5) { i -> print(i) }
walk1To(5, ::print)
```

Now, suppose we're passing the parameter to `System.out.println()`. We can replace the lambda with a function reference, but, in this case, we can replace the dot with `::`. Let's take a look at an example of this. You might have the following code:

```
walk1To(5) { i -> System.out.println(i) }
```

We change it to this:

```
walk1To(5, System.out::println)
```

In the previous example, `println()` was called on the instance `System.out`, but the reference may be an implicit receiver `this`. In that case, replace the lambda with a function reference on `this`, like in this example where `send()` is called on the implicit receiver `this`:

```
fun send(n: Int) = println(n)
walk1To(5) { i -> send(i) }
walk1To(5, this::send)
```

The same structure is used if the function call is on a singleton, like so:

```
object Terminal {
  fun write(value: Int) = println(value)
}
walk1To(5) { i -> Terminal.write(i) } walk1To(5, Terminal::write)
```

If a lambda is a pass-through lambda, then replace it with either a reference to

another lambda or a function reference, as appropriate.

Functions returning functions

Functions may return functions—that's intriguing. We ran into this accidentally in Functions with Block Body, when type inference was used with `=` in a block-bodied function. We do encounter legitimate situations where we'd want to return functions.

Suppose we have a collection of names and we want to find the first occurrence of a name of length `5` and also of length `4`. We may write something like this:

```
val names = listOf("Pam", "Pat", "Paul", "Paula")

println(names.find {name -> name.length == 5 }) //Paula

println(names.find { name -> name.length == 4 }) //Paul
```

returnlambda.kts

Both the lambdas in this example do the same thing, except for two different lengths. That may not be a big deal, but duplicating code is often a bad taste, and if the code within the lambda has any complexity, we definitely don't want to write that logic twice, or many times, and fall into the Write Every Time (WET) anti-pattern. We can refactor the code to create a function that will take the length as a parameter and return a lambda, a predicate, as the result. Then we can reuse calls to that function, like so:

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

println(names.find(predicateOfLength(5))) //Paula

println(names.find(predicateOfLength(4))) //Paul
```

returnlambda.kts

The parameter passed to `predicateOfLength()` is of type Int, but the return type isn't that simple. The return type is the signature of a function that takes String as

isn't that simple. The return type is the signature of a function that takes String as parameter and returns `Boolean` as output. Within the function we return a lambda

that takes a `String` parameter `input`, and compares its `length` with the value in the `length` parameter passed to `predicateOfLength()`.

In the function, we specified the return type of `predicateOfLength()`. We may also ask Kotlin to infer the type. That's only possible if the function is short, with a non-block body. In this case, it can be, so let's remove the return type details:

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

println(names.find(predicateOfLength(5))) //Paula

println(names.find(predicateOfLength(4))) //Paul
```

returnlambda.kts

Always specify return type for block-body functions, and use type inference only for functions with a non-block body. That's true for functions that return objects and for those that return functions as well.

We've looked at various options Kotlin provides for working with lambdas. In the next lesson, we'll look at saving lambdas into variables for reuse.