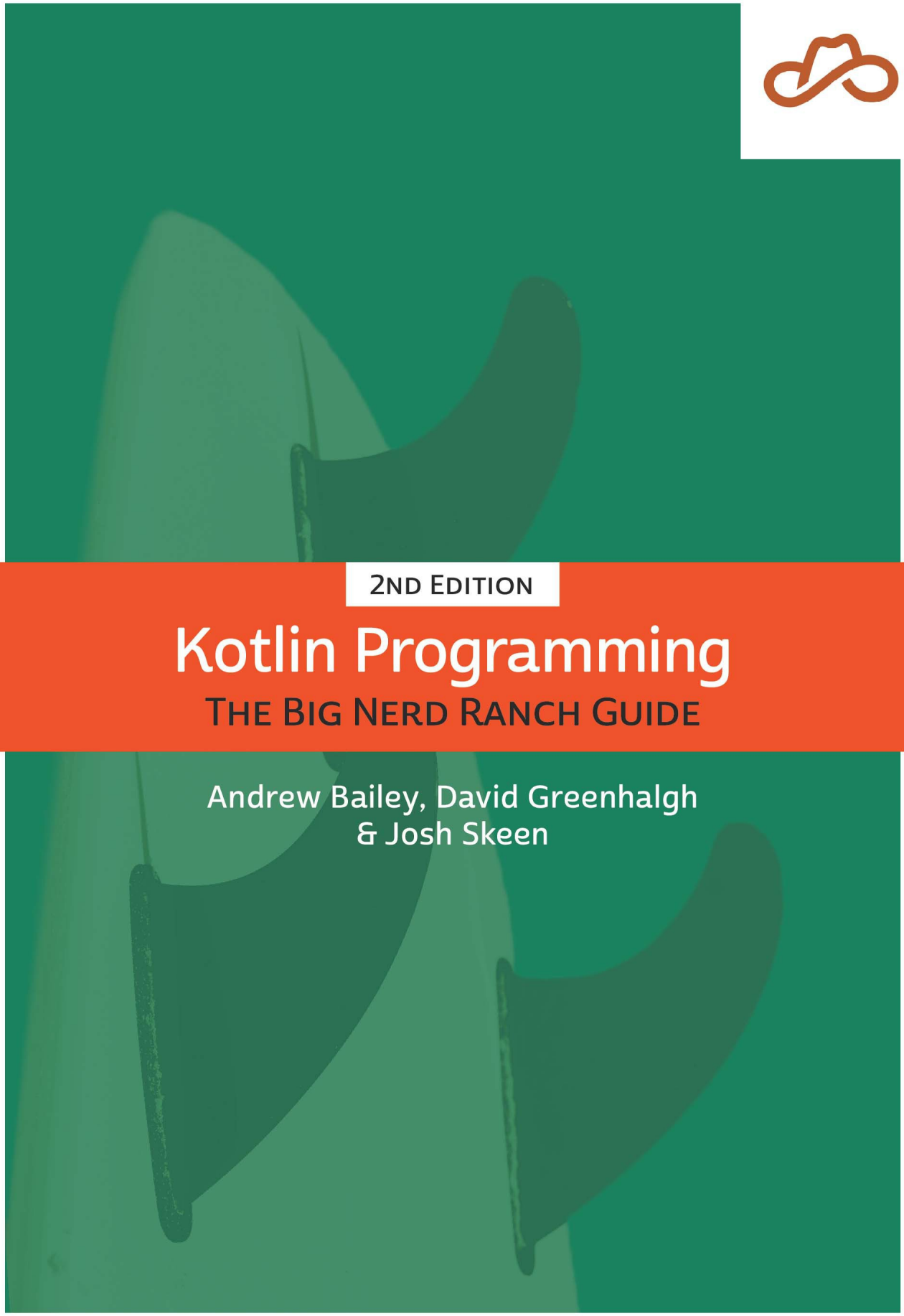

2ND EDITION

Kotlin Programming

THE BIG NERD RANCH GUIDE

Andrew Bailey, David Greenhalgh
& Josh Skeen

Kotlin Programming: The Big Nerd Ranch Guide

by Andrew Bailey, David Greenhalgh and Josh Skeen

Big Nerd Ranch, LLC
200 Arizona Ave NE
Atlanta, GA 30307
(770) 817-6373
http://www.bignerdranch.com/
book-comments@bignerdranch.com

Exclusive worldwide distribution of the English edition of this book by

Pearson Technology Group
800 East 96th Street
Indianapolis, IN 46240 USA
http://www.informit.com

ISBN-10 0136870570
ISBN-13 978-0136870579

Second edition, first printing, September 2021
Release D.2.1.1

Implicit returns

You may have noticed that there is no `return` keyword in the lambda expression you defined:

```
val narrationModifier: () -> String = {
    val numExclamationPoints = 3
    message.uppercase() + "!".repeat(numExclamationPoints)
}
```

However, the function type you specified indicates that the function must return a **`String`**, and the compiler did not complain. And, based on the output, a string is indeed returned: the modified welcome message. Why, then, is there no `return` keyword?

Unlike regular functions, the lambda expression syntax does not require – or even allow, except in rare cases – the `return` keyword to output data. Lambda expressions *implicitly* return the last line of their function definition, allowing you to omit the `return` keyword.

This feature of lambda expressions is both a convenience and a necessity of the syntax. The `return` keyword is prohibited in a lambda expression because it could be ambiguous to the compiler which function the return is from: the lambda expression itself, or the function that called it.

Function arguments

Like other functions, a lambda can accept zero, one, or multiple arguments of any type. The parameters a lambda accepts are indicated by type in the function type definition and then named in the lambda's definition.

Because your narrator's mood should be persistent, the `narrationModifier` function should be declared as a top-level variable instead of inside the **`narrate`** function. Make this refactor, updating the `narrationModifier` variable declaration to accept the message as an argument:

Listing 8.6 Adding a `message` parameter (`Narrator.kt`)

```
val narrationModifier: (String) -> String = { message ->
    val numExclamationPoints = 3
    message.uppercase() + "!".repeat(numExclamationPoints)
}

fun narrate(
    message: String
) {
    val narrationModifier: () -> String = {
        val numExclamationPoints = 3
        message.uppercase() + "!".repeat(numExclamationPoints)
    }

    println(narrationModifier(message))
}
```

Here you specify that the lambda accepts a **`String`** by placing the argument type in the function type's parentheses:

```
    val narrationModifier: (String) -> String = { message ->
```

You name the string parameter within the function, right after the opening curly brace:

```
    val narrationModifier: (String) -> String = { message ->
```

Lambda expressions that provide parameter names in this way separate them from the function body using the arrow operator (–>).

Run `NyetHack.kt` again. The output will not change, but your lambda expression is now taking in the `message` argument itself instead of reading it from **`narrate`**'s argument.

Remember the **`count`** function, which can take a function to check against each character in a string? That function is a *predicate* argument, called `predicate`, of type `(Char) -> Boolean` – in other words, a function that takes a **`Char`** argument and returns a **`Boolean`**. You will see function types and lambdas throughout much of the Kotlin standard library.

The it identifier

When defining lambdas that accept exactly one argument, the `it` identifier is available as a convenient alternative to specifying the parameter name. Both `it` and a named parameter are valid when you have a lambda that has only one parameter.

Delete the parameter name and arrow from the beginning of your `narrationModifier` lambda and use the `it` identifier instead:

Listing 8.7 Using the `it` identifier (`Narrator.kt`)

```
val narrationModifier: (String) -> String = { message ->
    val numExclamationPoints = 3
    message.uppercase() + "!".repeat(numExclamationPoints)
    it.uppercase() + "!".repeat(numExclamationPoints)
}

fun narrate(
    message: String
) {
    println(narrationModifier(message))
}
```

Run `NyetHack.kt` to confirm that it works as before.

`it` is convenient in that it requires no variable naming, but it is not very descriptive about the data it represents. We suggest that when you are working with more complex lambda expressions, or with nested lambdas (lambdas within lambdas), you stick with naming the parameter to preserve future readers' (and your own) sanity.

On the other hand, `it` is great for shorter expressions. For example, `it` would allow the **`count`** function call you saw earlier, to count the s's in "Mississippi," to be written more concisely, like this:

```
    "Mississippi".count({ it == 's' })
```

Because of the simplicity of this example, this logic is clear even without an argument name.

Accepting multiple arguments

While the `it` syntax is available for a lambda that accepts one argument, it is not allowed when there is more than one argument. However, lambdas can certainly accept multiple named arguments.

Suppose you wanted to tweak `narrationModifier` so the tone of the message influences how the narrator relates it. Try this out by running the code in Listing 8.8 in the REPL. (For simplicity's sake, you will not be making this change in NyetHack.)

Listing 8.8 Accepting a second argument (REPL)

```
val loudNarration: (String, String) -> String = { message, tone ->
    when (tone) {
        "excited" -> {
            val numExclamationPoints = 3
            message.uppercase() + "!".repeat(numExclamationPoints)
        }
        "sneaky" -> {
            "$message. The narrator has just blown Madrigal's cover.".uppercase()
        }
        else -> message.uppercase()
    }
}

println(loudNarration("Madrigal cautiously tip-toes through the hallway", "sneaky"))
```

Press Command-Return (Control-Enter) to run this code. You will see this output in the REPL:

```
MADRIGAL CAUTIOUSLY TIP-TOES THROUGH THE HALLWAY. THE NARRATOR HAS JUST BLOWN
MADRIGAL'S COVER.
```

This lambda expression declares two parameters, `message` and `tone`, and accepts two arguments when called. Because there is more than one parameter defined for the expression, the `it` identifier is no longer available.

Unlike regular functions, lambdas cannot have default arguments. The function type is the only information Kotlin retains for this kind of function, and it is not possible to include a default argument in a function's type as you can with a regular function. Named arguments are similarly disallowed when using a lambda.

Type Inference Support

Kotlin's type inference rules apply to function types much like they do with the types you met earlier in this book: If a variable is given a lambda as its value when it is declared, no explicit type definition is needed.

This means that the lambda you wrote earlier that accepted no arguments:

```
    val narrationModifier: () -> String = {
        val numExclamationPoints = 3
        message.uppercase() + "!".repeat(numExclamationPoints)
    }
```

could also have been written with no specified type, like this:

```
    val narrationModifier = {
        val numExclamationPoints = 3
        message.uppercase() + "!".repeat(numExclamationPoints)
    }
```

Type inference is also an option when the lambda accepts one or more arguments, with a slight catch. The compiler needs help to figure out what types the parameters of your lambda expression are. When using type inference, you must provide both the name and the type of each parameter in the lambda expression's definition. This also means that it is not an option to use the `it` shorthand if you want to use type inference.

Update the `narrationModifier` variable to use type inference by including the parameter type.

Listing 8.9 Using type inference for `narrationModifier` (`Narrator.kt`)

```
~~val narrationModifier: (String) -> String = {~~
val narrationModifier = { message: String ->
    val numExclamationPoints = 3
    ~~it.uppercase() + "!".repeat(numExclamationPoints)~~
    message.uppercase() + "!".repeat(numExclamationPoints)
}

fun narrate(
    message: String
) {
    println(narrationModifier(message))
}
```

Run `NyetHack.kt` and confirm that it works just as before.

When combined with an ambiguous implicit return type, type inference may make a lambda expression difficult to read. But when your lambda expressions are simple and clear, type inference is an asset for making your code more concise.

More Effective Lambdas

Take another look at your `Narrator.kt` file:

```
val narrationModifier = { message: String ->
    val numExclamationPoints = 3
    message.uppercase() + "!".repeat(numExclamationPoints)
}

fun narrate(
    message: String
) {
    println(narrationModifier(message))
}
```

Eagle-eyed readers may have noticed a disappointing reality: Everything that you have done so far could have been accomplished without using lambdas. This same logic could have been expressed with a single non-anonymous function, like this:

```
fun narrate(
    message: String
) {
    val numExclamationPoints = 3
    println(message.uppercase() + "!".repeat(numExclamationPoints))
}
```

Fear not: The work you have done so far has not been in vain. Lambdas shine when you need to change a function's behavior. Now that you have the basics of lambdas down, you can flesh out NyetHack's narration to include multiple moods.