Lambdas and Anonymous Functions

Lambdas are often passed as arguments to functions, but if the same lambda is needed on multiple calls, that may lead to code duplication. We can avoid that in a couple of ways. One is to store a lambda into a variable for reuse. Or we may create anonymous functions instead of lambdas—but consequences follow that design decision. Let’s explore these two options and discuss when to use each.

Using lambdas

In the previous example, we wrote a function to generate the lambdas. If the same lambda is used multiple times, we may save it into a variable and reuse it. There’s one catch, though. When a lambda is passed as argument to a function, Kotlin can infer the type of the parameters. But if we define a variable to store a lambda, Kotlin doesn’t have any context about the types. So, in this case, we need to provide sufficient type information.

Let’s create a variable to store one of the lambdas we passed earlier to the `find()` method:

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

val checkLength5 = { name: String -> name.length == 5 }
println(names.find(checkLength5)) //Paula
```

savelambdas.kts

The variable `checkLength5` refers to a lambda that takes a `String` parameter. The return type of this lambda is inferred by Kotlin because there are enough details to deduce the type information. We then pass the variable as an argument to the

function where a lambda, of the appropriate type, is expected. We may reuse that variable any number of times in other suitable calls.

In this example, we specified the type of the lambda's parameter, and Kotlin inferred the type of the variable to be `(String) -> Boolean`. Alternatively, we may ask Kotlin to infer in the opposite direction; we can specify the type of the variable and ask it to infer the type of the lambda's parameter. Let's give that a try:

```
val checkLength5: (String) -> Boolean = { name -> name.length == 5 }
```

In this case, if the lambda doesn't return the expected type that's specified in the variable declaration, then the compiler will complain.

Another undesirable alternative is that one may be tempted to specify both the type of the variable and of the lambda parameters, as here for example:

```
val checkLength5: (String) -> Boolean = { name: String -> name.length == 5 }
//Not Preferred
```

This feature is used only by programmers at the Department of Redundancy— the rest of us should avoid it.

Specify the type of the variable if you'd like to convey and enforce the return type of the lambda. If you want the return type to be inferred, then specify the type of the lambda parameters.

If you define the type of the variable to which a lambda is assigned, you should specify the return type. If you only specify the lambda parameter's type, then the return type is inferred. Another option to specify the return type while inferring the variable's type is *anonymous functions*.

Using anonymous functions

An anonymous function is written like a regular function, so the rules of specifying the return type—no type inference for block-body, `return` required for block-body, and so on—apply, with one difference: the function doesn't have a name. Let's convert the assignment to the variable in the previous example to use an anonymous function instead of a lambda.

```kotlin
val checkLength5 = fun(name: String): Boolean { return name.length == 5 }
```

Instead of storing an anonymous function in a variable, you may use it directly as an argument in a function call, in place of a lambda. For instance, here's an example to pass an anonymous function to the `find()` method:

```kotlin
names.find(fun(name: String): Boolean { return name.length == 5 })
```

That's a lot more verbose than passing a lambda, so there's no good reason to do this except in some rare situations.

Some restrictions apply when an anonymous function is used instead of a lambda. The `return` keyword is required for block-body anonymous functions that return a value. The return will always return from the anonymous function, and not from the encompassing function—we'll discuss `return` and lambdas in Non-Local and Labeled return. Also, if the lambda parameter is in the trailing position, then you may pass the lambda outside of the `()`—we discussed this in Use Lambda as the Last Parameter. However, anonymous functions are required to be within the `()`. In short, the following isn't allowed:

```kotlin
names.find { fun(name: String): Boolean { return name.length == 5 } } //ERROR
```

Prefer lambdas over anonymous functions where possible, and use anonymous functions selectively only in those rare occasions when they are suitable instead of lambdas—one such situation we'll see in Chapter 17, Asynchronous Programming.

QUIZ

1 Which of the following statements is true?

2

Why are anonymous functions used?

Retake Quiz

In the next lesson, we’ll see how lambdas relate to the concept of closures.