Inlining Functions with Lambdas

We'll cover the following

Lambdas are elegant, and it's convenient to pass functions to functions, but there's a catch—performance. Kotlin provides the `inline` keyword to eliminate the call overhead in order to improve performance, to provide non-local control flow such as a return from within `forEach()`, and to pass reified type parameters as we saw in Reified Type Parameters.

Before we delve into ways to improve performance of functions that use lambdas, let's set some context. Every higher-order function we write doesn't need the solutions we'll see in this section. A good amount of code we write will enjoy a reasonable performance and need nothing special. But in some situations—such as when a higher-order function contains a loop and excessively calls a lambda expression from within the loop, for example—the overhead of calling the higher-order function and the lambdas within it may be measurable. In that case, and only in that case, measure the performance first, and then consider these added complexities to improve the performance where necessary.

No inline optimization by default

To learn about `inline`, let's create an `invokeTwo()` function that takes an `Int` and two lambdas. It also returns a lambda. We'll modify this function a few times in this section, but the following is a good starting point without any `inline`:

```
fun invokeTwo(
fun invokeTwo(
  n: Int,
  action1: (Int) -> Unit,
  action2: (Int) -> Unit
  ): (Int) -> Unit {

  println("enter invokeTwo $n")

  action1(n)
  action2(n)

  println("exit invokeTwo $n")
  return { _: Int -> println("lambda returned from invokeTwo") }
}
```

noinline.kts

The short function invokes the two lambdas given and returns a lambda created within the function. The lambda that's returned ignores the input parameter and merely prints a message. Let's call this function from within another function named `callInvokeTwo()`. And, right after defining that function, let's call `callInvokeTwo()`.

```
// noinline.kts
fun callInvokeTwo() {
  invokeTwo(1, { i -> report(i) }, { i -> report(i) })
}

callInvokeTwo()
```

Within the `callInvokeTwo()` function we pass the value `1` as the first argument to `invokeTwo()`. For the second and third arguments we pass two identical lambdas that call a function named `report`. That function doesn't exist yet, but we'll write it now to print the parameter it receives along with the depth of the call stack.

```
fun report(n: Int) {
  println("")
  print("called with $n, ")

  val stackTrace = RuntimeException().getStackTrace()

  println("Stack depth: ${stackTrace.size}")
  println("Partial listing of the stack:")
  stackTrace.take(3).forEach(::println)
}
```

noinline.kts

The function reports the number of levels of call stack below the current execution of report(). Let's run the code to take a look at the calls and the number of levels in the call stack:

```
enter invokeTwo 1
called with 1, Stack depth: 31
Partial listing of the stack:
Noinline.report(noinline.kts:31)
Noinline$callInvokeTwo$1.invoke(noinline.kts:20)
Noinline$callInvokeTwo$1.invoke(noinline.kts:1)

called with 1, Stack depth: 31
Partial listing of the stack:
Noinline.report(noinline.kts:31)
Noinline$callInvokeTwo$2.invoke(noinline.kts:20)
Noinline$callInvokeTwo$2.invoke(noinline.kts:1)
exit invokeTwo 1
```

The call to `callInvokeTwo()` results in a call to `invokeTwo()`. That function call in turn results in a call to `action1()`, the first lambda passed as parameter. The lambda calls `report()`. Likewise, when `invokeTwo()` calls the second lambda, `action2()`, it calls `report()`. Between the place of call to `invokeTwo()` and within each call to `report()`, we have three levels of stack. That's the top three out of the depth of `31`.

Inline optimization

You may improve performance of functions that receive lambdas using the `inline` keyword. If a function is marked as `inline`, then instead of making a call to the function, the bytecode for that function will be placed inline at the call location. This will eliminate the function call overhead, but the bytecode will be larger since the inlining will happen at every location where the function is called. It's usually a bad idea to inline long functions.

Though you may annotate any non-recursive function with `inline`, Kotlin will give you a warning if it sees no benefit to inlining, for example, if the function isn't receiving any lambda parameters.

Let's optimize the `invokeTwo()` function using `inline`:

```
inline fun invokeTwo(
  n: Int,
  action1: (Int) -> Unit,
  action2: (Int) -> Unit
  ): (Int) -> Unit {
```

The function’s body has no change; the `inline` annotation prefixes the function declaration—that’s enough to tell the compiler to optimize the call.

Let’s run the code after this change and take a look at the depth of the call stack.

```
enter invokeTwo 1

called with 1, Stack depth: 28
Partial listing of the stack:
Inlineoptimization.report(inlineoptimization.kts:31)
Inlineoptimization.callInvokeTwo(inlineoptimization.kts:20)
Inlineoptimization.<init>(inlineoptimization.kts:23)

called with 1, Stack depth: 28
Partial listing of the stack:
Inlineoptimization.report(inlineoptimization.kts:31)
Inlineoptimization.callInvokeTwo(inlineoptimization.kts:20)
Inlineoptimization.<init>(inlineoptimization.kts:23)
exit invokeTwo 1
```

The three top levels of call stack we discussed earlier, before adding the `inline` annotation, are gone. Within the `callInvokeTwo()` function the compiler expands the bytecode for the `invokeTwo()` function. And within the `invokeTwo()` functions body, the compiler inlines or expands the bytecode for the two lambdas, instead of making the calls. That optimization continues to eliminate the call overhead to `report()` as well.

By using the `inline` annotation you can eliminate the call overhead. But if the function being inlined is very large and if it’s called from a lot of different places, the bytecode generated may be much larger than when `inline` isn’t used. Measure and optimize—don’t optimize blindly.

Selective noinline of parameter

If for some reason we don’t want to optimize the call to a lambda, we can ask that optimization to be eliminated by marking the lambda parameter as `noinline`. We can use that keyword only on parameters when the function itself is marked as `inline`.

Let’s ask the compiler to inline the `invokeTwo()` function, and as a result inline the call to `action1()`, as well, but specifically exclude the optimization for `action2()` call, using `noinline` on that parameter:

```
inline fun invokeTwo(
  n: Int,
  action1: (Int) -> Unit,
  noinline action2: (Int) -> Unit
  ): (Int) -> Unit {
```

Kotlin won't allow us to hold a reference to `action1` since it's inlined, but we may create a reference to `action2` within the `invokeTwo()` function, if we like, since `action2` is defined as `noinline`.

Also, since the `action2` parameter is marked with `noinline`, there'll be no optimization to its call. Thus, the second call to `report()`, from within the lambda passed to `action2` will be deeper than the call to `report()` from within the lambda passed to `action1`. We can see this in the output of the code we used:

```
enter invokeTwo 1
called with 1, Stack depth: 28
Partial listing of the stack:
Noinlineoptimization.report(noinlineoptimization.kts:31)
Noinlineoptimization.callInvokeTwo(noinlineoptimization.kts:20)
Noinlineoptimization.<init>(noinlineoptimization.kts:23)

called with 1, Stack depth: 30
Partial listing of the stack:
Noinlineoptimization.report(noinlineoptimization.kts:31)
Noinlineoptimization$callInvokeTwo$2.invoke(noinlineoptimization.kts:20)
Noinlineoptimization$callInvokeTwo$2.invoke(noinlineoptimization.kts:1) exit i
nvokeTwo 1
```

In addition to inlining the code, the `inline` keyword also makes it possible for lambdas called from inlined functions to have non-local `return`. We saw this in the context of `forEach()` earlier. Let's revisit that for our `invokeTwo()` function.

Non-local return permitted in inlined lambdas

In the previous example, the `invokeTwo()` function has the inline annotation and, as a result, the first lambda `action1()` will also be inlined. However, the second lambda `action2()` is marked as `noinline`. Thus, Kotlin will permit non-local `return` and labeled `return` from within the lambda passed as an argument for the `action1` parameter. But, from within the lambda passed as the argument for the action2 parameter, only labeled return is permitted. This is because, whereas an inlined lambda expands within a function, the non-inlined lambda will be a separate function call. While the return from the former will exit the function, the return from the latter won't do the same since it's in a more nested level of stack.

Let's see this behavior in action.

```
fun callInvokeTwo() {
    invokeTwo(1, { i ->
        if (i == 1) { return }

        report(i)
    }, { i ->
        //if (i == 2) { return }| //ERROR, return not allowed here
        report(i)
    })
}
```

Within the first lambda passed to invokeTwo(), we call return if the value of the parameter i == 1. This is a non-local return and will result in the exit from the function being defined—that is, callInvokeTwo(). We can verify this in the output that follows. On the other hand, within the second lambda passed to invokeTwo(), the Kotlin compiler won't permit using non-local return. Any attempt to uncomment line 7 will result in compilation failure.

```
enter invokeTwo 1
```

In addition to annotating functions with inline, you may also mark methods and properties of classes with inline if you choose. When using inline you can not only eliminate the function call overhead, but also gain the ability to place a non-local return from within the inlined lambdas. Any lambda that is not inlined can't have a non-local return. That's good, but what if a lambda that's intended to be inlined can't really be inlined? Let's discuss next how this may happen and how Kotlin lets us know about this situation.

crossinline parameters

If a function is marked inline, then the lambda parameters not marked with noinline are automatically considered to be inlined. At the location where a lambda is invoked within the function, the body of the lambda will be inlined. But there's one catch. What if instead of calling the given lambda, the function passes on the lambda to yet another function, or back to the caller? Tricky, you can't on the lambda to yet another function, or back to the caller? Tricky, you can't inline what is not being called.

In the case where the lambda is passed on instead of being called, not placing any annotation on the lambda parameter makes no sense. One solution is to mark it as `noinline`. But what if you want the lambda to be inlined wherever it may be called. You can ask the function to pass on your request for inlining across to the caller; that's what `crossinline` is for.

Let's understand this scenario and how `crossinline` helps with an example. Let's make two changes to the `invokeTwo()` function. First, let's remove the `noinline` annotation of the `action2` parameter. Second, let's modify the lambda returned in the end of `invokeTwo()` to call `action2`—that is, `invokeTwo()` passes on `action2` so that it may be called eventually by the caller of `invokeTwo()`.

```
inline fun invokeTwo(
    n: Int,
    action1: (Int) -> Unit,
    action2: (Int) -> Unit //ERROR
    ): (Int) -> Unit {

    println("enter invokeTwo $n")

    action1(n)

    println("exit invokeTwo $n")
    return { input: Int -> action2(input) }
}
```

When `invokeTwo()` is inlined, the internal call `action1(n)` can be inlined. But since `invokeTwo()` isn't directly calling `action2`, the `action2(input)` call embedded within the lambda on the last line can't be inlined. Since there is no `noinline` annotation on the second parameter to `invokeTwo()`, we're in a conflict situation and the compiler will give us an error.

Besides the error, we need to document for the programmers using `invokeTwo()` that they can't use a non-local `return` from the second lambda passed to `invokeTwo()`. We can achieve this goal and resolve the compilation error in one of two ways:

- Mark the second parameter as `noinline`. In this case, the call to `action2` won't be inlined, period. There'll be no performance benefit and a non-local `return` won't be permitted within the lambda passed for `action2`.

- Mark the second parameter as `crossinline`. In this case, the call to `action2` will be inlined, not within the `invokeTwo()` function but wherever it is called.

You're not allowed to place a non-local `return` within a lambda passed to the parameter marked with `crossinline`. The reason is that by the time the lambda is executed, you would have exited from the function to which it is passed as a parameter; no point trying to return from a function that has already completed.

Let's modify the above code so it will pass compilation, by using `crossinline`:

```
inline fun invokeTwo(
  n: Int,
  action1: (Int) -> Unit,
  crossinline action2: (Int) -> Unit
  ): (Int) -> Unit {
```

Now that we marked `action2` as `crossinline`, the compiler is happy that we understood the consequences.

In summary,

- `inline` performs inline optimization, to remove function call overhead.
- `crossinline` also performs inline optimization, not within the function to which the lambda is passed, but wherever it is eventually called.
- Only lambdas passed for parameters not marked `noinline` or `crossinline` can have non-local `return`.

Good practices for inline and returns

The concepts related to `inline`, `return` from lambda, and non-local returns is not trivial and can get overwhelming. Take some time to review, practice the examples, and try out your own code examples to get a better grip of the concepts.

Here's a summary and some good practices related to returns and inline:

- Unlabeled `return` is always a return from a function and not from a lambda.
- Unlabeled `returns` are not permitted in non-inlined lambdas.
- Function names are the default labels, but don't rely on them, always provide custom names if you choose to use labeled `returns`.

- Measure performance before deciding to optimize code; this is true in general, and in particular for code that uses lambdas.
- Use `inline` only when you see measurable performance improvements.

The next lesson concludes the discussion for this chapter.