Non-Local and Labeled return

We'll cover the following

- return not allowed by default
- Labeled return
- Non-Local return

By default lambdas aren't allowed to have the `return` keyword, even if they return a value. This is a significant difference between lambdas and anonymous functions—the latter is required to have `return` if returning values and it signifies only a `return` from the immediate lambda and not the outer calling function. In this section, we'll look at why `return` is not allowed by default, how to use labeled `return`, and when non-local `return` is permitted.

`return` not allowed by default

`return` is not valid in lambdas by default, but you can use it under some special situations. This can be quite confusing if you don't take the time to fully understand the context and the consequences. Take this topic a bit slow for the details to sink in. We'll use a function to illustrate the concepts in this section. This function takes two parameters, an Int and a lambda, and returns `Unit`.

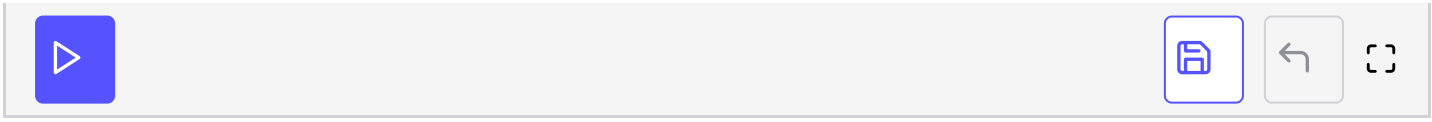
```
fun invokeWith(n: Int, action: (Int) -> Unit) {
  println("enter invokeWith $n")
  action(n)
  println("exit invokeWith $n")
}
```

noreturn.kts

Within the `invokeWith()` function, we pass the given `Int` parameter to the given lambda referenced using `action()`. Quite simple, nothing exciting yet.

Let's call `invokeWith()` from within a `caller()` function and then call the `caller()` function.

```
fun caller() {
  (1..3).forEach { i ->

    invokeWith(i) {
      println("enter for $it")

      if (it == 2) { return } //ERROR, return is not allowed here

      println("exit for $it")
    }
  }

  println("end of caller")
}

caller()
println("after return from caller")
```



noreturn.kts

The `caller()` function iterates over values `1` to `3` and calls the `invokeWith()` function for each value. The lambda attached to the `invokeWith()` function prints a message when we enter it and then again when we exit it. If the value of the lambda's parameter, referenced using the implicit reference `it`, is equal to `2` we request an immediate return using the `return` keyword.

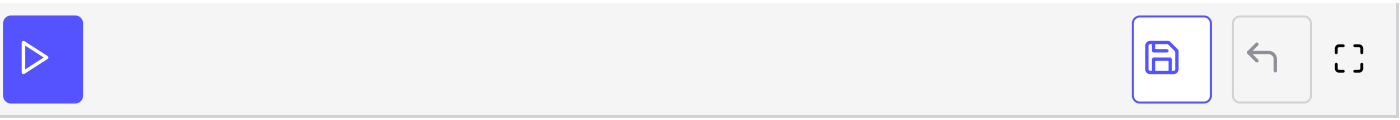
The line with return will fail compilation. The reason for this becomes clear if we look at that line in the overall context of the entire code. When we call `return` on line 6, Kotlin doesn't know if we mean (1) to exit the immediate lambda and continue executing code within `invokeWith()` right after the call to `action(n)`, or (2) we mean to exit the for loop we entered on line 2, or (3) exit the function `caller()` we entered on line 1. To avoid this confusion, Kotlin doesn't permit `return` by default. However, Kotlin makes two exceptions to this rule—labeled `return` and non-local `return`. We'll see these next, one at a time.

Labeled return

If you want to exit the current lambda immediately, then you may use a labeled `return`, that is `return@label` where label is some label you can create using the syntax `label@`. Let's use a labeled `return` in a modified version of the `caller()`.

```
fun caller() {
    (1..3).forEach { i ->
        invokeWith(i) here@ {
            println("enter for $it")
```

```
            if (it == 2) {
                return@here
            }

            println("exit for $it")
        }
    }

    println("end of caller")
}

caller()
println("after return from caller")
```



labeledreturn.kts

We replaced the `return` with `return@here` on line 7 and, in addition, added a label `here@` on line 3. The compiler won't complain at this use of `return`.

The labeled `return` causes the control flow to jump to the end of the labeled block—that is, exit the lambda expression. The behavior here is equivalent to the `continue` statements used in imperative-style loops where the control skips to the end of the loop. In this case, it skips to the end of the lambda expression. We can observe this behavior from the output of the previous code, shown here:

```
enter invokeWith 1
enter for 1
exit for 1
exit invokeWith 1
enter invokeWith 2
enter for 2
exit invokeWith 2
enter invokeWith 3
enter for 3
exit for 3
exit invokeWith 3
end of caller
after return from caller
```

Instead of using an explicit label, like `@here`, we can use an implicit label that is the name of the function to which the lambda is passed. We can thus replace `return@here` with `return@invokeWith` and remove the label `here@`, like so:

```
fun caller() {
  (1..3).forEach { i ->

    invokeWith(i) {
      println("enter for $it")

      if (it == 2) {
        return@invokeWith
      }

      println("exit for $it")
    }
  }

  println("end of caller")
}
```

Even though Kotlin permits the use of method names as labels, prefer explicit labels instead. That makes the intention clearer and the code easier to understand.

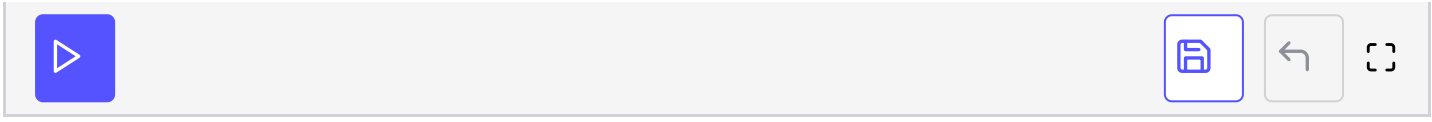
The compiler won't permit labeled return to arbitrary outer scope—you can only return out of the current encompassing lambda. If you want to exit out of the current function being defined, then you can't do that by default, but you can if the function to which the lambda is passed is inlined. We looked at the default behavior in the previous subsection. Let's explore the inlining option next.

Non-Local return

The `return` keyword is not allowed by default in lambdas. You may use a labeled return and that restricts the control flow to only return out of the current lambda. Non-local return is useful to break out of the current function that's being implemented, right from within a lambda. Let's modify the `caller()` function once again to see this feature.

```
fun caller() {
  (1..3).forEach { i ->

    println("in forEach for $i")
    if (i == 2) { return }

    invokeWith(i) {
      println("enter for $it")

      if (it == 2) { return@invokeWith }

      println("exit for $it")
    }
```

```
  }
  println("end of caller")
}


caller()
println("after return from caller")
```



In the `caller()` function, within the lambda passed to `forEach()` we return `if i == 2` on line 5. We know that on line 10, Kotlin doesn't permit `return` but only a labeled `return`. But, in line 5 it doesn't have any qualms about it. Curious—before we discuss the why, let's address the what; that is, let's understand the behavior of this `return`. Unlike the labeled return which exits the encompassing lambda, this `return` on line 5 will exit the encompassing function being defined—`caller()` in this example. Thus, it's called the non-local `return`.

Let's verify this behavior by executing the `caller()` function:

```
in forEach for 1
enter invokeWith 1
enter for 1
exit for 1
exit invokeWith 1
in forEach for 2
after return from caller
```

When the execution hits the `return` statement, it bails all the way out of the `caller()`, and the code following the call to `caller()` is executed after `return`.

Now to the question: Why did Kotlin disallow `return`, without label, of course, within the lambda we passed to `invokeWith()`, but didn't flinch at the `return` within the lambda passed to `forEach()`? The answer is hidden in the definition of the two functions.

We defined `invokeWith()` as the following:

```
fun invokeWith(n: Int, action: (Int) -> Unit) {
```

On the other hand, forEach() is defined, in the Kotlin standard library, like this:

```
inline fun <T> Iterable<T>.forEach(action: (T) -> Unit): Unit {
```

The answer lies in the keyword `inline`. We'll focus on this keyword in the next section—let's recap the behavior of `return` before we move forward:

- `return` is not allowed by default within lambdas.
- You may use a labeled `return` to step out of the encompassing lambda.
- Use of non-local `return` to exit from the encompassing function being defined is possible only if the function to which the lambda is passed is defined with `inline`.

Here are some ways to deal with the complexity associated with the behavior of `return`:

- You can use labeled `return` anytime to return out of the lambda.
- If you're able to use `return`, remember that it will exit out of the encompassing function that's being defined and not merely from within the encompassing lambda or from within the caller of the lambda.
- If you're not allowed to use `return`, don't worry; Kotlin will let you know in no uncertain terms.

QUIZ

1 Why is `return` not allowed by default in lambdas?

2 Which statement does the labeled return in lambdas behave like?

Retake Quiz

In the next lesson, we'll see how to improve the performance of functions that use lambdas.