More Expressions, Fewer Statements

We'll cover the following

Languages like Java, C#, and JavaScript have more statements than expressions—`if` statement, `for` statement, `try`, and so on. On the other hand, languages like Ruby, F#, Groovy, Haskell, and many others have more expressions than statements. Let's discuss which is better before we discuss Kotlin's preference.

Drawbacks of statements

While statements are prevalent, they have a dark side—they don't return anything and have side effects. A side effect is a change of state: mutating a variable, writing to a file, updating to a database, sending data to a remote web service, corrupting the hard drive... Expressions are much nicer—they return a result and don't have to modify any state in order to be useful.

Let's look at an example to see the difference. Let's write a piece of Kotlin code as we would in languages like Java and C#:

```
fun canVote(name: String, age: Int): String {
    var status: String
    if (age > 17) {
        status = "yes, please vote"
    } else {
        status = "nope, please come back"
        }
    return "$name, $status"
}
println(canVote("Eve", 12))
```

The `canVote()` method uses `if` like a statement. Since statements don't return

anything, the only way we can get any useful result out of it for further processing

is to set up a mutable variable and modify its value within the branches.

Why Kotlin prefers expressions

In Kotlin, however, `if` is an expression. We can use the result of the call to `if` for further processing. Let's rework the previous code, to use if as an expression instead of a statement:

```
val status = if (age > 17) "yes, please vote" else "nope, please come back"

return "$name, $status"
```

We were able to use `val` instead of `var` since we're not mutating a variable. And we were able to use type inference for status, since the value is known from the `if` expression. The code is less noisy and less error prone as well.

Kotlin also treats `try-catch` as an expression. The last expression within the `try` part becomes the result if there was no exception; otherwise, the last statement within the catch becomes the result. Here's an example of `try-catch-finally` being used as an expression.

```
fun tryExpr(blowup: Boolean): Int {
    return try {
        if (blowup) {
            throw RuntimeException("fail")
        }
        2
    } catch(ex: Exception) {
        4
    } finally {
        //...
    }
}
println(tryExpr(false)) //2
println(tryExpr(true)) //4
```

Assignment as an expression

There's one surprise, though. While Java treats assignment as an expression, Kotlin doesn't. If variables `a`, `b`, and `c` are defined using `var` with some integer values, like, `1`, `2`, and `3` respectively, the following code will fail compilation in Kotlin:

like, `1`, `2`, and `3` respectively, the following code will fail compilation in Kotlin:

```
a = b = c //ERROR
```

One reason for this behavior, of not treating `=` as an expression, is that Kotlin allows you to intercept both gets and sets on variables with delegates, as we'll see later in the chapter. If `=` were treated as an expression, then chaining of assignments may lead to unexpected and complex behavior that may be confusing and turn into a source of errors.

The next lesson concludes the discussion for this chapter.