Prefer val over var

We'll cover the following

When to use `val`

To define an immutable variable–that is, a constant or a value—use `val`, like so:

```
val pi: Double = 3.14
```

Unlike Java, where you'd place the type before the name of the variable, in Kotlin you place the name of the variable first, then a colon, followed by the type. Kotlin considers the sequence Java requires as "placing the cart before the horse" and places a greater emphasis on variable names than variable types.

Since the type of the variable is obvious in this context, we may omit the type specification and ask Kotlin to use type inference:

```
val pi = 3.14
```

Either way, the value of `pi` can't be modified; `val` is like `final` in Java. Any attempt to change or reassign a value to variables defined using val will result in a compilation error. For example, the following code isn't valid:

```
val pi = 3.14
pi = 3.14 //ERROR: val cannot be reassigned
```

When to use `var`

What if we want to be able to change the value of a variable? For that, Kotlin has `var`—also known as "keyword of shame." Variables defined using var may be mutated at will.

Here's a script that creates a mutable variable and then modifies its value:

```
var score = 10
//or var score: Int = 10
println(score) //10 score = 11
println(score) //11
```

Mutating variables is a way of life in imperative style of programming. But that's a taboo in functional programming. In general, it's better to prefer immutability—that is, val over var. Here's an example to illustrate why that's better:

```
var factor = 2

fun doubleIt(n: Int) = n * factor

factor = 0

println(doubleIt(2))
```

mutate.kts

Don't run the code; instead eyeball it, show it to a few of your colleagues and ask what the output of the code will be. Take a poll. The output will be equal to what most people said it will be—just kidding. Program correctness is not a democratic process; thankfully, I guess.

You probably got three responses to your polling:

- The output is 4.
- The output is 0, I think.
- WHAT—the response the code evoked on someone recently.

The output of the above code is `0`—maybe you guessed that right, but guessing is not a pleasant activity when coding.

Why `val` and not `var`?

Mutability makes code hard to reason. Code with mutability also has a higher chance of errors. And code with mutable variables is harder to parallelize. In general, try really hard to use `val` as much as possible instead of `var`. You'll see

later on that Kotlin defaults toward `val` and immutability, as well, in different instances.

Whereas `val` in Kotlin is much like Java's `final`, Kotlin—unlike Java—insists on marking mutable variables with `var`. That makes it easier to search for the presence of var in Kotlin than to search for the absence of `final` in Java. So in Kotlin, it's easier to scrutinize code for potential errors that may arise from mutability.

A word of caution with `val`, however—it only makes the variable or reference a constant, not the object referenced. So `val` only guarantees immutability of the reference and doesn't prevent the object from changing. For example, String is immutable but `StringBuilder` is not. Whether you use `val` or `var`, an instance of `String` is safe from change, but an instance of `StringBuilder` isn't. In the following code, the variable `message` is immutable, but the object it refers to is modified using that variable.

```
val message = StringBuilder("hello ")
//message = StringBuilder("another") //ERROR
message.append("there")
```

In short, `val` only focuses on the variable or reference at hand, not what it refers to. Nevertheless, prefer `val` over `var` where possible.

QUIZ

1 To define a mutable variable, we'll use what?

2

Will an error be generated by the following code snippet?

```
var a = 2
a = a + 1
```

Retake Quiz

In the next lesson, we’ll learn about equality checks in Kotlin.