Kotlin Var and Val Getter and Setter Problem

Asked 7 months ago Modified 7 months ago Viewed 163 times

0

I am new to Kotlin Programming Language. (Java -> Kotlin)

```
var x
    get()= 4; // Showing error Property must be initialized

val y
    get()= 4; // No error
```

Why is this happening ?

android kotlin val

Share Edit Follow

asked Nov 27, 2022 at 15:43
Jithin Murali
33 ● 6

The first case is a `var`, so it has a setter. What would you expect the setter to do in this case? The getter would always return 4, but the setter should be able to update some value. Do you have a real-life example for this? – Joffrey Nov 27, 2022 at 15:52

@Joffrey I think there is no setter given. – Jithin Murali Nov 27, 2022 at 16:23

You didn't provide a custom setter implementation, but using `var` makes Kotlin generate a default one. Please refer to the current answer, it explains in details why the default setter makes this code problematic. However, I'd be interested in knowing whether you actually faced this problem in real life – Joffrey Nov 27, 2022 at 16:30

1 Answer

Sorted by: Highest score (default)

6

`val y = 4` is a *val*, which means it's *read-only*. The value you're initialising it with, *4*, is stored in a *backing field*. A bit of memory where the value of that variable is stored.

`var y = 4` is a *var*, which means *you can write to it*. Again, there's a backing field storing the current value - it's just that you can change it now.

This is probably all stuff you're used to - but Kotlin also allows you to define *getter and setter functions*, depending on whether it's a `val` (getter only) or `var`. In reality, default getters and setters already exist - they just read from and write to the backing field. If you don't define your own, it just uses those.

`val y get() = 4` is a *val*, which means it's *read-only*. Because you've added a getter function, the `4` value isn't actually stored in a backing field - so Kotlin doesn't create one! There's nothing to initialise with a value.

`var y get() = 4` is a *var*, so you have the ability to set it. You haven't defined a setter function here, so it's using the implicit default one, which writes to the backing field.

And that's where your problem is - Kotlin won't create a backing field if it doesn't need one:

```
// stores any value passed to the setter in the variable's backing field
var something
    get() = "hi"
    set(value) { field = value }

// does nothing with the setter value, no field is used anywhere, so it doesn't create
one
var somethingElse
    get() = "hi"
    set(value) { }
```

(It's not a good example because generally you wouldn't have a fixed value with a setter, because what's the point? But you could have, say, a getter/setter pair that stores and reads from a map or something, which doesn't require a backing field since it's getting and storing the data elsewhere)

So when you do `var y get() = 4` you're effectively doing this:

```
var y
    get() = 4
    set(value) { field = value } // default setter
```

and that's creating a backing field through the setter. And that field needs to have *some* initial value! Kotlin's not like Java that way, where things can be left unassigned and default to a value.

So you just have to assign a value:

```
var y = 1
    get() = 4
```

Now your backing field has a value so the compiler's happy. You never actually read from it, so it's useless, and that's a good sign you don't need a `var` at all!

Have a look at the properties documentation, it explains all this stuff (and some other things about visibility etc) but hopefully that makes sense, and you can see why your example is actually a bit strange and something you'd avoid.

Share Edit Follow

answered Nov 27, 2022 at 16:24

cactu: cactustictacs
17.6k ● 2 ● 13 ● 24