Zach Klippenstein
Posted on 6 Apr 2021 • Updated on 24 May 2021

60 14

remember { mutableStateOf() } – A cheat sheet

#android #compose #kotlin #cheatsheet

Compose state explained (7 Part Series)

If you've ever read any Compose code, you've probably seen this at least a million times:

```
var text by remember { mutableStateOf("") }
```

What the heck is that? There's a lot going on. The goal of this post is to pull the pieces apart, show how which parts do what, and how they all fit together.

---

tl;dr

Really quick cheat sheet (more of a cheat card):

- `by` has nothing to do with Compose. It declares a delegated property.
- `remember` and `mutableStateOf` are completely independent concepts. They just happen to be used together a lot. Just like how databases and RxJava are completely independent but fit nicely together for certain use cases.
- `remember` keeps a value (any value) consistent across recompositions.
- `mutableStateOf` returns a `MutableState`. Think `mutableListOf`.
- `MutableState` is just a thing that holds a value, where Compose will automatically observe changes to the value. Think `MutableLiveData`, but you don't need to call `observe` yourself.

⚠ Gotcha!

You may need to manually import a couple functions to make the property delegate compile. This is a known IDE bug, hopefully it gets fixed before 1.0.

```
import androidx.compose.runtime.getValue
import androidx.compose.runtime.setValue // only if using var
```

"State": it's a thing

Probably one of the most overloaded terms in UI programming, certainly in Compose, is "state".

In its most general, "state" refers to any data that is held in memory (either RAM or on disk). State can change over time, but those changes are persistent for some period of time. State can live in variables in a function, or properties/fields in a class.

In other words, something that is "state" has two properties:

1. It exists over time. This is what `remember` does.
2. It can change. This is what `mutableStateOf` does.

Property delegation

```
var foo: T by bar
```

This is property delegation. Property delegation is a Kotlin feature. It has nothing to do with Compose. It's been a core feature of the language for... ever, I think.

Remember that a property in Kotlin is effectively just a pair of getter and setter methods. In the above code, the compiler generates `fun getFoo(): T` and `fun setFoo(value: T)`.

A property delegate is an object that has special getter and setter methods (or extension functions). When you use the object on the right side of the `by` in code like above, the `getFoo` and `setFoo` methods call (delegate to) the delegate's getter and setter, respectively.

Delegated properties can be both class properties and local properties (i.e. vals and vars defined in a function).

`mutableStateOf` supports a variety of ways to interact with it. Property delegation is one option, but you can also just treat it as a regular object.

`remember`: Keeps a value over time

`remember` is a composable function that "memoizes" the value returned from the function (lambda) you pass to it then returns that value, allowing you to create state that persists across recompositions. "Recomposition" means when a composable function is called multiple times to update the UI.

The first time a composable runs that calls `remember`, it executes the lambda to get the value. It then stores (memoizes) that value in the composition. The next time that composable is recomposed, the call to `remember` will see that there was a value stored from the last call and just return that. You can remember any type of value – primitives, your own classes, anything. One of the things you can remember is the `MutableState` type, but there's nothing special about it.

Note that you only need to use `remember` inside composable functions (and the compiler will complain if you try to use it anywhere else). Outside of composables there are other ways to store state. For example, if you're writing a class, you'll probably put your state in properties in that class.

`mutableStateOf`: Observable state holder

State is only really useful if it can change over time. We need a way to change it, and we need a way to react when it changes. In that `var text by remember { mutableStateOf("") }` line, a whole bunch of pieces are in play:

`State<T>`

A type that holds a read-only `value: T` and notifies the composition (and other "snapshot observers", like layouts) when `value` changes.

- A "read" is tracked whenever code calls the `value` getter, even if there are multiple call stack frames (function calls) between the actual property and the thing reading the state.
- This means that if you have a `foo: State<T>`, and a function `fun getFoo(): T = foo.value`, both reading `foo` directly and calling `getFoo()` will be considered "reading foo" by a snapshot observer and cause the observer to be notified when `foo.value` changes.

`MutableState<T>`

A subtype of `State<T>` that allows setting `value`. When the value is changed, any snapshot observers that read `value` will be notified when the snapshot is applied (i.e. when preparing the next frame).

- It’s a common pattern to hold a `MutableState<T>` as a private class property, then expose it publicly either as a `State<T>` or as a regular property that has the type `T`.
- `State` is to `MutableState` as `List` is to `MutableList`.

`State<T>.getValue()`

An extension function on `State<T>` that allows instances to act as property delegates for read-only properties (vals), via the `by` syntax.

`MutableState<T>.setValue()`

An extension function on `MutableState<T>` that allows instances to act as property delegates for mutable properties (vars).

`mutableStateOf(): MutableState<T>`

A function that creates a `MutableState<T>` given an initial value.

- This function is analogous to `mutableListOf()`.
- This is not a composable function – you can call it from regular code. In fact it’s quite common to use the function to create properties or property delegates for regular classes.

Putting the pieces together

These snippets all do the same thing.

```
@Composable fun MyField() {
  var text by remember { mutableStateOf("") }
  TextField(text, { text = it })
}
```

Without using the delegate (notice `by` -> `=`, `var` -> `val`):

```
@Composable fun MyField() {
  val text = remember { mutableStateOf("") }
  TextField(text.value, { text.value = it })
}
```

Pulling state into a separate class, using a property delegate. Notice `mutableStateOf` is *not* directly wrapped with a `remember` here, because it's not being called from a composable function. Instead, the thing being remembered is the class instance itself – `remember` instantiates the class.

```
class MyFieldState() {
  var text by mutableStateOf("")
}

@Composable fun MyField() {
  val state = remember { MyFieldState() }
  TextField(state.text, { state.text = it })
}
```

Giving the class a more fancy API:

```
class MyFieldState() {
  var text by mutableStateOf("")
    private set
  fun setText(text: String) {
    this.text = text
  }
}

@Composable fun MyField() {
  val state = remember { MyFieldState() }
  TextField(state.text, state::setText)
}
```

Dive deeper

We've just scratched the surface. Here's a teaser:

- `remember` can be given keys to trigger re-calculation of its value.
- `MutableState` supports destructing.
- `MutableState` can be told how to conflate values, and even how to merge changes made from different threads.
- You can create `Flow`s that observe `State` values.
- `remember`ed values can be notified when they enter and leave a composition.

The official docs go into a lot more detail.

- Delegated properties in the Kotlin language spec
- `remember` reference
- `MutableState` reference
- State and Jetpack Compose article
- Using State in Jetpack Compose codelab
- Jetpack Compose: State video

Thanks to Colton Idle for requesting and giving feedback on this post.

Top comments (7)

Louis CAD • 9 Apr 21

If I make an interface of `val` properties, and implement it with a class that delegates these to a Compose `State`/`MutableState`, I'll not get the observability for Composable functions because, and the only way through is to expose `State<T>` instead of `T`, right?

Zach Klippenstein • 9 Apr 21

Nope, that still works. Reads are tracked by actual calls to the getter of a State.value, no matter where or how it is called. Also, there's no compiler magic here. I believe under the hood there is actually a thread local installed that points to the current snapshot and reads record themselves there. So it doesn't matter what your call stack looks like.

Louis CAD • 10 Apr 21

Oh, that's very neat!
Thanks for the info. I hope this knowledge makes it into the official docs.

Hakanai • 16 Nov 22 • Edited on 16 Nov

There's also this form.

```
@Composable
fun MyField() {
    val (text, setText) = remember { mutableStateOf("") }
    TextField(text, setText)
}
```

It really makes me wish they'd have an overload which just takes `MutableState<String>` so that I can just pass the state object itself, or some kind of adapter.

For whatever reason, Jetpack Compose seem to really love passing the value and the way to update the value as two different thingies, when they could have been bundled together as a single concept.

Daniel Rendox • 7 Jul

So what actually happens when the following code is executed:

```
var text by remember { mutableStateOf("") }
```

1) A new object of the type `MutableState<String>` is created. This object holds the value (in our case empty string). And whenever the value changes, `MutableState` tells Android to recompose the UI.
2) Thanks to `remember`, the object of the `MutableState` is cashed and therefore will not get recreated every recomposition.
3) `by` tells the compiler to "link" `var text` with the value that the `MutableState` object holds. Whenever our `text` is updated, that value will be updated either automatically.

🙏 Correct me if I'm wrong

Uli • 12 Jan 22

"remembered values can be notified when they enter and leave a composition": can you give a pointer what you are referring to?

Zach Klippenstein • 2 Feb 22

developer.android.com/reference/ko...

Code of Conduct • Report abuse

DEV Community

Presenting the... Warm Welcome badge!

This unique badge is awarded to our distinguished online participants and devoted community members who are consistently active in our Welcome Thread.

Learn more about this badge:

Introducing the Warm Welcome Badge! 🌟🎉
dev.to staff for The DEV Team · Jul 25
#meta #design #welcome

Check out our Welcome Thread:

Welcome Thread - v235
Sloan the DEV Moderator for The DEV Team · Jul 26
#welcome

Zach Klippenstein

Opinions are my own. He/him.

LOCATION
San Francisco, CA

EDUCATION
University of Manitoba, Canada

PRONOUNS
he/him

WORK
Software Engineer at Google (previously Square, Amazon)

JOINED

13 Jul 2020

More from Zach Klippenstein

Implementing snapshot-aware data structures
#programming #compose #kotlin #computerscience

Compose Hackathon: Day… the rest
#programming #compose #kotlin

Compose Hackathon: Day 2.5
#programming #compose #kotlin

The DEV Team PROMOTED •••

YOUR AD HERE
YOUR AD HERE