

# Jetpack Compose Internals

JORGE CASTILLO

ANDREI SHIKOV

Jetpack Compose and Android are trademarks of Google LLC  
and this book is not endorsed by or affiliated with Google in any way.

# Jetpack Compose internals

Jorge Castillo

This book is for sale at <http://leanpub.com/composeinternals>

This version was published on 2022-11-20



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2021 - 2022 Jorge Castillo

## **Tweet This Book!**

Please help Jorge Castillo by spreading the word about this book on [Twitter](#)!

The suggested hashtag for this book is [#composeinternals](#).

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

[#composeinternals](#)

*This book is dedicated to all the people that helped along the way. I want to say thank you to Manuel Vivo, Joe Birch, Antonio Leiva, Enrique López-Mañas, Andrei Shikov, Leland Richardson, Chuck Jazdzewski, Andrey Kulikov, and Adam Powell for reviewing parts of this book and providing very valuable feedback.*

*I also want to give special thanks to the Google Jetpack Compose team/s for the incredible work and amount of effort they have put on this ecosystem of libraries, and for embracing the book in such a positive way since day one. Jetpack Compose is already a game changer for the future of Android and Kotlin Multiplatform UI.*

*Shout-out to my mate Andrei Shikov, who wrote one of the chapters (probably the most interesting one!). Thanks Andrei for being an active part of this, your help is really appreciated.*

*I must also mention Snapp Mobile (<https://snappmobile.io/>), Snapp Automotive for their proactiveness sponsoring the book and designing the book cover during its earlier stages. Ana Silva is the designer behind such an incredible cover. Outstanding work, we truly love it! Big thanks to Stream (<https://getstream.io/>) also for their sponsorship, they helped to make this book a reality.*

*Finally, I really want to thank my wife, María Isabel, and my beloved daughter, Julia, for giving me all the strength and motivation to face such a complicated challenge during these particularly hard times. Having a family like the one I have is like having a superpower. I am so lucky to have you. I love you both.*

# Contents

<b>Prelude</b> . . . . .	<b>1</b>
Why to read this book . . . . .	1
What this book is not about . . . . .	1
Why to write about internals . . . . .	1
Keep the sources close . . . . .	2
Code snippets and examples . . . . .	2
<b>1. Composable functions</b> . . . . .	<b>3</b>
The meaning of Composable functions . . . . .	3
Properties of Composable functions . . . . .	4
Calling context . . . . .	4
Idempotent . . . . .	6
Free of uncontrolled side effects . . . . .	6
Restartable . . . . .	8
Fast execution . . . . .	9
Positional memoization . . . . .	9
Similarities with suspend functions . . . . .	12
The color of Composable functions . . . . .	13
Composable function types . . . . .	15
<b>2. The Compose compiler</b> . . . . .	<b>17</b>
A Kotlin compiler plugin . . . . .	17
Compose annotations . . . . .	18
Registering Compiler extensions . . . . .	24
Kotlin Compiler version . . . . .	25
Static analysis . . . . .	25
Static Checkers . . . . .	26
Call checks . . . . .	26
Type checks . . . . .	28
Declaration checks . . . . .	28
Diagnostic suppression . . . . .	29
Runtime version check . . . . .	31
Code generation . . . . .	31
The Kotlin IR . . . . .	31

## CONTENTS

Lowering . . . . .	32
Inferring class stability . . . . .	33
Enabling live literals . . . . .	36
Compose lambda memoization . . . . .	37
Injecting the Composer . . . . .	40
Comparison propagation . . . . .	42
Default parameters . . . . .	44
Control flow group generation . . . . .	44
Klib and decoy generation . . . . .	49
<b>3. The Compose runtime . . . . .</b>	<b>50</b>
The slot table and the list of changes . . . . .	51
The slot table in depth . . . . .	51
The list of changes . . . . .	54
The Composer . . . . .	55
Feeding the Composer . . . . .	55
Modeling the Changes . . . . .	57
Optimizing when to write . . . . .	58
Writing and reading groups . . . . .	58
Remembering values . . . . .	59
Recompose scopes . . . . .	59
SideEffects in the Composer . . . . .	60
Storing CompositionLocals . . . . .	61
Storing source information . . . . .	61
Linking Compositions via CompositionContext . . . . .	61
Accessing the current State snapshot . . . . .	61
Navigating the nodes . . . . .	62
Keeping reader and writer in sync . . . . .	62
Applying the changes . . . . .	62
Performance when building the node tree . . . . .	63
How changes are applied . . . . .	65
Attaching and drawing the nodes . . . . .	66
Composition . . . . .	67
Creating a Composition . . . . .	68
The initial Composition process . . . . .	70
Applying changes after initial Composition . . . . .	72
Additional information about the Composition . . . . .	72
The Recomposer . . . . .	73
Spawning the Recomposer . . . . .	73
Recomposition process . . . . .	77
Concurrent recomposition . . . . .	78
Recomposer states . . . . .	78

## CONTENTS

<b>4. Compose UI . . . . .</b>	<b>80</b>
Integrating UI with the Compose runtime . . . . .	80
Mapping scheduled changes to actual changes to the tree . . . . .	81
Composition from the point of view of Compose UI . . . . .	81
Subcomposition from the point of view of Compose UI . . . . .	85
Reflecting changes in the UI . . . . .	88
Different types of Appliers . . . . .	88
Materializing a new LayoutNode . . . . .	92
Closing the circle . . . . .	96
Materializing a change to remove nodes . . . . .	97
Materializing a change to move nodes . . . . .	98
Materializing a change to clear all the nodes . . . . .	98
Measuring in Compose UI . . . . .	98
Measuring policies . . . . .	101
Intrinsic measurements . . . . .	107
Layout Constraints . . . . .	110
LookaheadLayout . . . . .	112
Modeling modifier chains . . . . .	125
Setting modifiers to the LayoutNode . . . . .	126
How LayoutNode ingests new modifiers . . . . .	130
Drawing the node tree . . . . .	132
Semantics in Jetpack Compose . . . . .	134
Notifying about semantic changes . . . . .	136
Merged and unmerged semantic trees . . . . .	137
<b>5. State snapshot system . . . . .</b>	<b>139</b>
What snapshot state is . . . . .	139
Concurrency control systems . . . . .	141
Multiversion concurrency control (MCC or MVCC) . . . . .	142
The Snapshot . . . . .	143
The snapshot tree . . . . .	146
Snapshots and threading . . . . .	147
Observing reads and writes . . . . .	147
MutableSnapshots . . . . .	149
GlobalSnapshot and nested snapshots . . . . .	152
StateObjects and StateRecords . . . . .	153
Reading and writing state . . . . .	157
Removing or reusing obsolete records . . . . .	159
Change propagation . . . . .	160
Merging write conflicts . . . . .	162
<b>6. Effects and effect handlers . . . . .</b>	<b>165</b>
Introducing side effects . . . . .	165

## CONTENTS

Side effects in Compose . . . . .	166
What we need . . . . .	168
Effect Handlers . . . . .	168
Non suspended effects . . . . .	169
Suspended effects . . . . .	172
Third party library adapters . . . . .	174
<b>7. Advanced Compose Runtime use cases . . . . .</b>	<b>177</b>
Compose runtime vs Compose UI . . . . .	177
(Re-) Introducing composition . . . . .	179
Composition of vector graphics . . . . .	181
Building vector image tree . . . . .	183
Integrating vector composition into Compose UI . . . . .	186
Managing DOM with Compose . . . . .	189
Standalone composition in the browser . . . . .	193
Conclusion . . . . .	196

# Prelude

## Why to read this book

Jetpack Compose has officially become the new standard for UI development in the Android platform. Even if lots of apps still rely on the View system for their already existing UI, new screens and components are very frequently coded using Compose, so it is becoming an unavoidable technology to learn. My strong suggestion is to dedicate some time to learn about its internals in detail, since that will make you gain powerful knowledgeability and skills to write correct, modern, and efficient Android apps.

In the other hand, if you happen to be interested in other use cases of Jetpack Compose rather than Android, you'll be happy to know that this book has got you covered also. Jetpack Compose internals is very focused on the compiler and runtime details, making the overall experience very agnostic of the target platform. Having an Android background is not a requirement for reading this book. The book also contains a chapter dedicated to different use cases of the library, which exposes a few really interesting examples over code.

Finally, if you are a curious engineer this might be a great chance to learn something new, and a perfect challenge for you.

## What this book is not about

This book does not aim to replicate the Jetpack Compose official documentation, which is quite good already and the source of truth for anyone starting with it. For that reason, you will not find here any listings or catalogues of all the existing components or APIs that the library provides.

If you are new to Compose, I can recommend a sibling book to have on your desk along this one: “[Practical Jetpack Compose](#)” book by Joe Birch<sup>1</sup>. Joe’s book is full of interesting examples and detailed explanations about all the relevant components and APIs available. The book showcases a set of applications to teach you Compose over code.

## Why to write about internals

As an Android developer and over the years, I have grown a feeling of how astoundingly important can become to learn the internals of the platform you work with every day. It is a game changer. Working on this knowledge base regularly helps me to understand what code I want to write, and

---

<sup>1</sup><https://practicaljetpackcompose.com/>

why. It enables me to write performant code that complies with the platform expectations, and to understand why things work the way they do. To me, this is probably one of the biggest differences between not very experienced and experienced Android developers.

My personal goal as the author of this book is to give you all the tools to achieve a big leap forward on this field.

## Keep the sources close

If you ask me, I'd say that reading sources is one of the most convenient skills we can grow as software developers, no matter who wrote them. I strongly recommend anyone reading this book to keep the sources as close as possible while reading it, and explore even further. You can find everything in [cs.android.com](https://cs.android.com/)<sup>2</sup>. Sources are also indexed in Android Studio, so you should be able to navigate those. Having a playground project with Compose around can also be desirable.

## Code snippets and examples

One of the things we will learn in this book is that Jetpack Compose can be used not only to represent UI trees but essentially any large call graphs with any types of nodes. However, many of the code snippets and examples you'll find in this book will be UI oriented for an easier mental mapping, since that is what most developers are used to at this point.

---

Welcome to Jetpack Compose Internals. Grab a coffee, have a seat, and enjoy the reading.

Jorge.

---

<sup>2</sup><https://cs.android.com/>

# 1. Composable functions

## The meaning of Composable functions

Probably the most adequate way to start a book about Jetpack Compose internals would be by learning about Composable functions, given those are the atomic building blocks of Jetpack Compose, and the construct we will use to write our composable trees. I intentionally say “trees” here, since composable functions can be understood as nodes in a larger tree that the Compose runtime will represent in memory. We will get to this in detail when the time comes, but it is good to start growing the correct mindset from the very beginning.

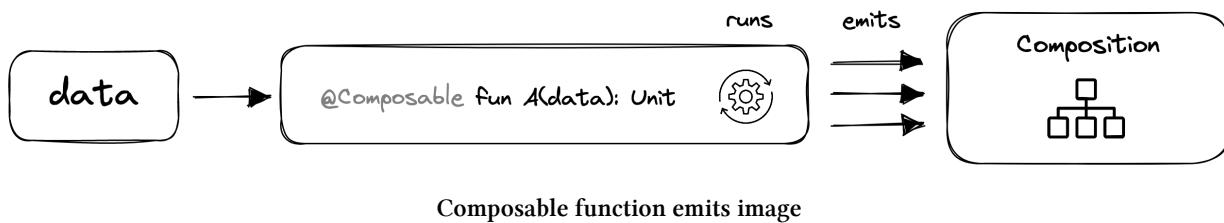
If we focus on plain syntax, any standard Kotlin function can become a Composable function just by annotating it as `@Composable`:

```
1 @Composable
2 fun NamePlate(name: String) {
3     // Our composable code
4 }
```

By doing this we are essentially telling the compiler that the function intends to convert some data into a node to register in the composable tree. That is, if we read a Composable function as `@Composable (Input) -> Unit`, the input would be the data, and the output would not be a value returned from the function as most people would think, but an action registered to insert the element into the tree. We could say that this happens as a side effect of executing the function.

Note how returning `Unit` from a function that takes an input means we are likely consuming that input somehow within the body of the function.

The described action is usually known as “emitting” in the Compose jargon. Composable functions emit when executed, and that happens during the composition process. We will learn every detail about this process in the upcoming chapters. For the time being, every time we read something about “composing” a Composable function, let’s simply think of it as an equivalent of “executing” it.



The only purpose of executing our Composable functions is to build or update the in-memory representation of the tree. That will keep it always up to date with the structure it represents, since Composable functions will re-execute whenever the data they read changes. To keep the tree updated, they can emit actions to insert new nodes as explained above, but they can also remove, replace, or move nodes around. Composable functions can also read or write state from/to the tree.

## Properties of Composable functions

There are other relevant implications of annotating a function as Composable. The `@Composable` annotation effectively **changes the type of the function** or expression that it is applied to, and as any other type, it imposes some constraints or properties over it. These properties are very relevant to Jetpack Compose since they unlock the library capabilities.

The Compose runtime expects Composable functions to comply to the mentioned properties, so it can assume certain behaviors and therefore exploit different runtime optimizations like parallel composition, arbitrary order of composition based on priorities, smart recomposition, or positional memoization among others. But please, don't feel overwhelmed about all these new concepts yet, we will dive into every single one in depth at the right time.

Generically speaking, runtime optimizations are only possible when a runtime can have some certainties about the code it needs to run, so it can assume specific conditions and behaviors from it. This unlocks the chance to execute, or in other words "consume" this code following different execution strategies or evaluation techniques that take advantage of the mentioned certainties.

An example of these certainties could be the relation between the different elements in code. Are they dependant on each other or not? Can we run them in parallel or different order without affecting the program result? Can we interpret each atomic piece of logic as a completely isolated unit?

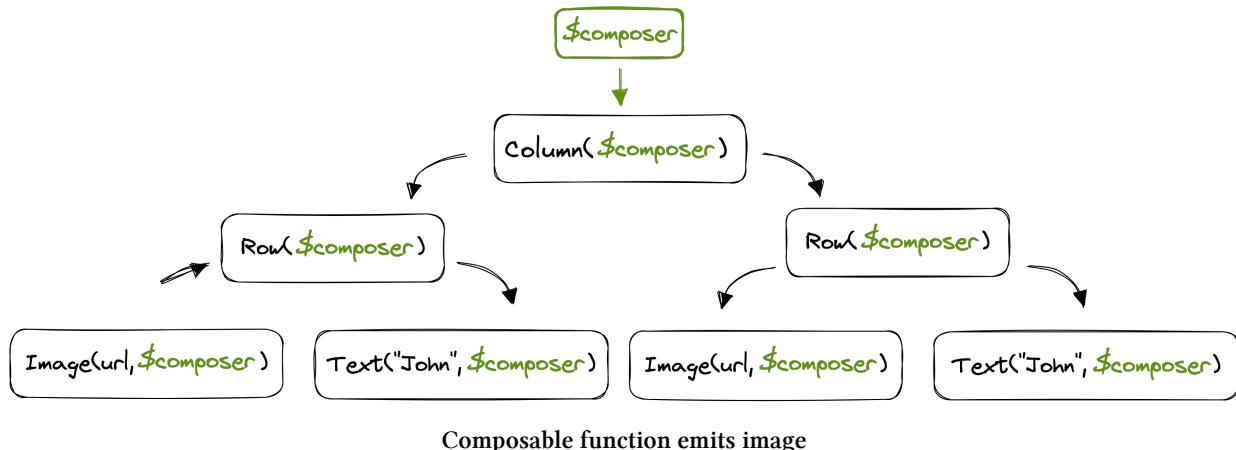
Let's learn about the properties of Composable functions.

## Calling context

Most of the properties of Composable functions are enabled by the Compose compiler. Since it is a Kotlin compiler plugin, it runs during the normal compiler phases, and has access to all the

information that the Kotlin compiler has access to. This allows it to intercept and transform the IR (intermediate representation) of all the Composable functions from our sources in order to add some extra information to them.

One of the things added to each Composable function is a new parameter, at the end of the parameters list: The Composer. This parameter is implicit, the developer remains agnostic of it. An instance of it is injected at runtime, and forwarded to all the child Composable calls so it is accessible from all levels of the tree.



In code, let's say we have the following Composable:

```

1 @Composable
2 fun NamePlate(name: String, lastname: String) {
3     Column(modifier = Modifier.padding(16.dp)) {
4         Text(text = name)
5         Text(text = lastname, style = MaterialTheme.typography.subtitle1)
6     }
7 }
```

The Compiler will transform it into something like this:

```

1 fun NamePlate(name: String, lastname: String, $composer: Composer<*>) {
2     ...
3     Column(modifier = Modifier.padding(16.dp), $composer) {
4         Text(
5             text = name,
6             $composer
7         )
8         Text(
9             text = lastname,
10            style = MaterialTheme.typography.subtitle1,
```

```
11     $composer
12   )
13 }
14 ...
15 }
```

As we can see, the `Composer` is forwarded to all the Composable calls within the body. On top of this, the Compose compiler imposes a strict rule to Composable functions: They can only be called from other Composable functions. This is the actual **calling context** required, and it ensures that the tree is conformed of only Composable functions, so the `Composer` can be forwarded down.

The `Composer` is the connection between the Composable code we write as developers, and the Compose runtime. Composable functions will use it to emit their changes for the tree and therefore inform the runtime about its shape in order to build its in-memory representation or update it.

## Idempotent

Composable functions are expected to be idempotent relative to the node tree they produce. Re-executing a Composable function multiple times using the same input parameters should result in the same tree. The Jetpack Compose runtime relies on this assumption for things like recomposition.

In Jetpack Compose, **recomposition** is the action of re-executing Composable functions when their inputs vary, so they can emit updated information and update the tree. The runtime must have the ability to recompose our Composable functions at arbitrary times, and for diverse reasons.

The recomposition process traverses down the tree checking which nodes need to be recomposed (re-executed). Only the nodes with varying inputs will recompose, and the rest will be **skipped**. Skipping a node is only possible when the Composable function representing it is idempotent, since the runtime can assume that given the same input, it will produce the same results. Those results are already in-memory, hence Compose does not need to re-execute it.

## Free of uncontrolled side effects

A side effect is any action that escapes the control of the function where it is called in order to do something unexpected on the side. Things like reading from a local cache, making a network call, or setting a global variable are considered side effects. They make the calling function dependant on external factors that might influence its behavior: external state that might be written from other threads, third party apis that might throw, etc. In other words, the function does not depend on its inputs only to produce a result.

Side effects are a **source of ambiguity**. That is not great for Compose, since the runtime expects Composable functions to be predictable (deterministic), so they can be re-executed multiple times

safely. If a Composable function ran side effects, it could produce a different program state on every execution, making it not idempotent.

Let's imagine that we ran a network request directly from the body of a Composable function, like this:

```
1 @Composable
2 fun EventsFeed(networkService: EventsNetworkService) {
3     val events = networkService.loadAllEvents()
4
5     LazyColumn {
6         items(events) { event ->
7             Text(text = event.name)
8         }
9     }
10 }
```

This would be very risky, since this function might get re-executed multiple times in a short period of time by the Compose runtime, making the network request trigger multiple times and spiral out of control. It is actually worse than that, since those executions might happen from different threads without any coordination.

The Compose runtime reserves the right to pick the execution strategies for our Composable functions. It can offload recompositions to different threads to take advantage of multiple cores, or run them in any arbitrary order based on its own needs or priorities (E.g: Composables not showing on screen could get assigned a lower priority).

Another common caveat of side effects is that we could make a Composable function depend on the result of another Composable function, imposing a relation of order. That should be avoided at all cost. An example of this:

```
1 @Composable
2 fun MainScreen() {
3     Header()
4     ProfileDetail()
5     EventList()
6 }
```

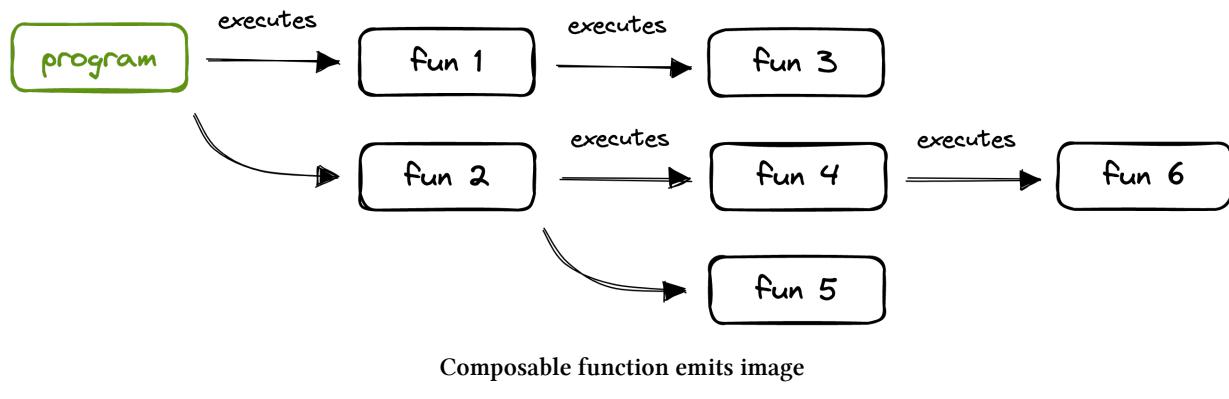
In this snippet, `Header`, `ProfileDetail` and `EventList` might execute in any order, or even in parallel. We should not write logics that assume any specific execution order, like reading an external variable from `ProfileDetail` that is expected to be written from `Header`.

Generically speaking, side effects are not ideal in Composable functions. We must try making all our Composable functions stateless, so they get all their inputs as parameters, and only use them to produce a result. This makes Composables simpler, dumber, and highly reusable. However, side effects are needed to write stateful programs, so at some level we will need to run them (frequently at the root of our Composable tree). Programs need to run network requests, persist information in databases, use memory caches, etc. For this reason, Jetpack Compose offers mechanisms to call effects from Composable functions safely and within a controlled environment: The **effect handlers**.

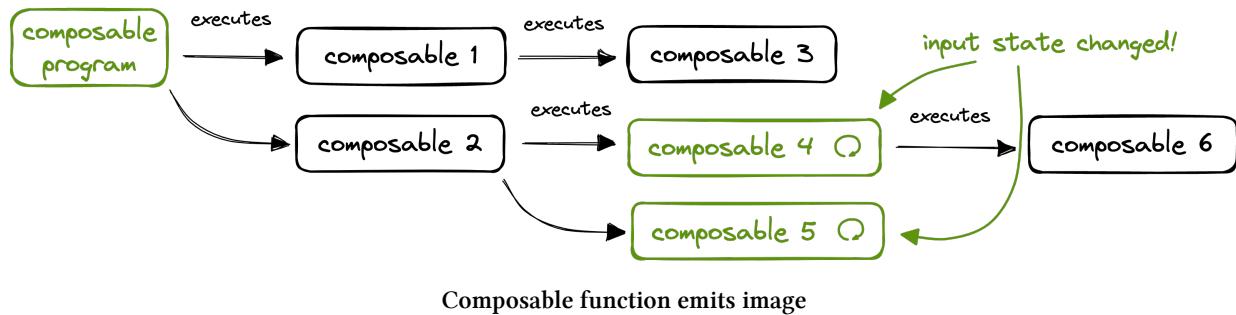
Effect handlers make side effects aware of the Composable lifecycle, so they can be constrained/-driven by it. They allow effects to be automatically disposed/canceled when the Composable leaves the tree, re-triggered if the effect inputs change, or even span the same effect across executions (recompositions) so it is only called once. We will cover effect handlers in detail in later chapters. They will allow us to avoid calling effects directly from the Composable's body without any control.

## Restartable

We have mentioned this a few times already. Composable functions can recompose, so they are not like standard functions, in the sense that they will not be called only once as part of a call stack. This is how a normal call stack would look. Each function gets called once, and it can call one or multiple other functions.



On the other hand, Composable functions can be restarted (re-executed, recomposed) multiple times, so the runtime keeps a reference to them in order to do so. Here is how a Composable call tree could look:



Composables 4 and 5 are re-executed after their inputs change.

Compose is selective about which nodes of the tree to restart in order to keep its in-memory representation always up to date. Composable functions are designed to be reactive and re-executed based on changes in the state they observe.

The Compose compiler finds all Composable functions that read some state and generates the code required to teach the runtime how to restart them. Composables that don't read state don't need to be restarted, so there is no reason to teach the runtime how to do so.

## Fast execution

We can think of Composable functions and the Composable function tree as a fast, declarative, and lightweight approach to build a description of the program that will be retained in memory and interpreted / materialized in a later stage.

Composable functions don't build and return UI. They simply emit data to build or update an in-memory structure. That makes them blazing fast, and allows the runtime to execute them multiple times without fear. Sometimes it happens very frequently, like for every frame of an animation.

Developers must fulfill this expectation when writing code. Any cost heavy computation should be offloaded to coroutines and always wrapped into one of the lifecycle aware effect handlers that we will learn about ahead in this book.

## Positional memoization

Positional memoization is a form of function memoization. Function memoization is the ability of a function to cache its result based on its inputs, so it does not need to be computed again every time the function is called for the same inputs. As we already learned, that is only possible for pure (**deterministic**) functions, since we have the certainty that they will always return the same result for the same inputs, hence we can cache and reuse the value.

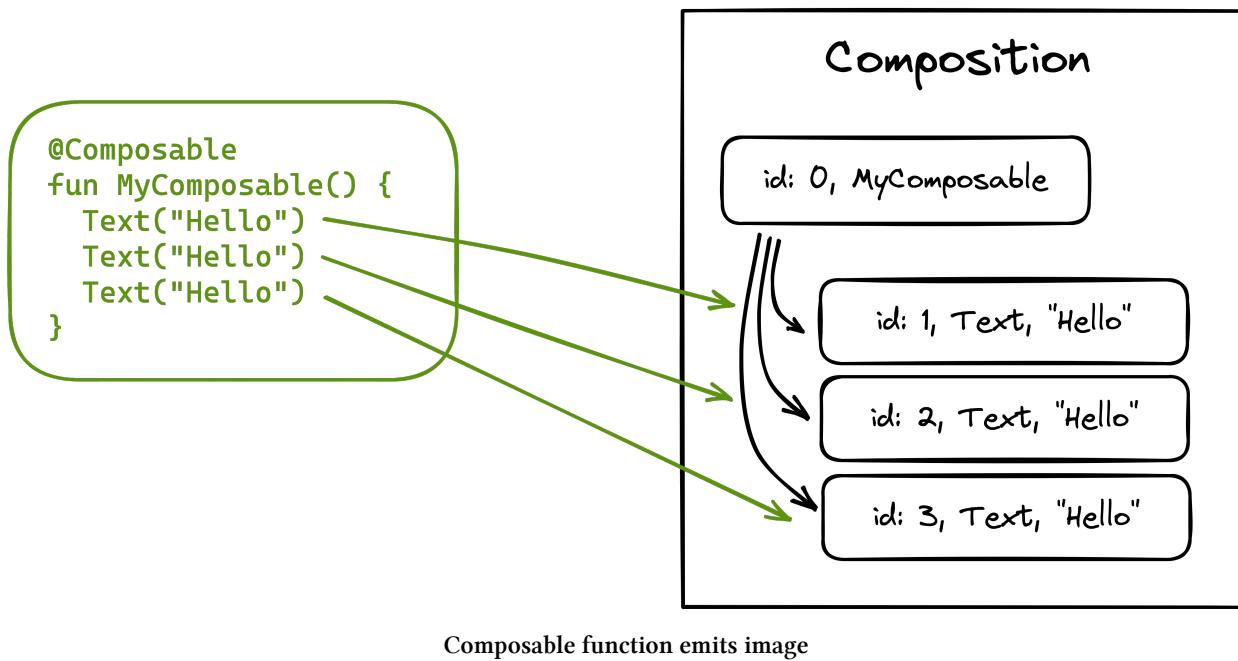
Function memoization is a technique widely known in the Functional Programming paradigm, where programs are defined as a composition of pure functions.

In function memoization, a function call can be identified through a combination of its name, type, and parameter values. A unique key can be created using those elements, and used to store/index/read the cached result in later calls. In Compose, an additional element is considered: Composable functions have constant knowledge about **their location in the sources**. The runtime will generate different ids (unique within the parent) when the same function is called with the same parameter values but from different places:

```

1 @Composable
2 fun MyComposable() {
3     Text("Hello") // id 1
4     Text("Hello") // id 2
5     Text("Hello") // id 3
6 }
```

The in-memory tree will store three different instances of it, each one with a different identity.



Composable identity is preserved across recompositions, so the runtime can appeal to this structure to know whether a Composable was called previously, and skip it if possible.

Sometimes assigning unique identities can be hard for the Compose runtime. One example is lists of Composables generated from a loop:

```

1 @Composable
2 fun TalksScreen(talks: List<Talk>) {
3     Column {
4         for (talk in talks) {
5             Talk(talk)
6         }
7     }
8 }
```

In this case, `Talk(talk)` is called from the same position every time, but each talk represents a different item on the list, and therefore a different node on the tree. In cases like this, the Compose runtime relies on the **order of calls** to generate the unique id, and still be able to differentiate them. This works nicely when adding a new element to the end of the list, since the rest of the calls stay in the same position as before. But what if we added elements to the top, or somewhere in the middle? The runtime would recompose all the `Talks` below that point since they shifted their position, even if their inputs have not changed. This is highly inefficient (esp. for long lists), since those calls should have been skipped.

To solve this, Compose provides the `key` Composable, so we can assign an explicit key to the call manually:

```

1 @Composable
2 fun TalksScreen(talks: List<Talk>) {
3     Column {
4         for (talk in talks) {
5             key(talk.id) { // Unique key
6                 Talk(talk)
7             }
8         }
9     }
10 }
```

In this example we are using the talk id (likely unique) as the key for each `Talk`, which will allow the runtime to preserve the identity of all the items on the list **regardless of their position**.

Positional memoization allows the runtime to remember Composable functions by design. Any Composable function inferred as restartable by the Compose compiler should also be skippable, hence **automatically remembered**. Compose is built on top of this mechanism.

Sometimes developers need to appeal to this in-memory structure in a more granular way than the scope of a Composable function. Let's say we wanted to cache the result of a heavy calculation that takes place within a Composable function. The Compose runtime provides the `remember` function for that matter:

```
1 @Composable
2 fun FilteredImage(path: String) {
3     val filters = remember { computeFilters(path) }
4     ImageWithFiltersApplied(filters)
5 }
6
7 @Composable
8 fun ImageWithFiltersApplied(filters: List<Filter>) {
9     TODO()
10 }
```

Here, we use `remember` to cache the result of an operation to precompute the filters of an image. The key for indexing the cached value will be based on the call position in the sources, and also the function input, which in this case is the file path. The `remember` function is just a Composable function that knows how to read from and write to the in-memory structure that holds the state of the tree. It only exposes this “positional memoization” mechanism to the developer.

In Compose, memoization is not application-wide. When something is memoized, it is done within the context of the Composable calling it. In the example from above, it would be `FilteredImage`. In practice, Compose will go to the in-memory structure and look for the value in the range of slots where the information for the enclosing Composable is stored. This makes it be more like a **singleton within that scope**. If the same Composable was called from a different parent, a new instance of the value would be returned.

## Similarities with suspend functions

Kotlin suspend functions can only be called from other suspend functions, so they also require a calling context. This ensures that suspend functions can only be chained together, and gives the Kotlin compiler the chance to inject and forward a runtime environment across all the computation levels. This runtime is added to each suspend function as an extra parameter at the end of the parameters list: `TheContinuation`. This parameter is also implicit, so developers can remain agnostic of it. `TheContinuation` is used to unlock some new powerful features in the language.

Sounds familiar, right?

In the Kotlin coroutine system, a `Continuation` is like a callback. It tells the program how to continue the execution.

Here is an example. A code like the following:

```
1 suspend fun publishTweet(tweet: Tweet): Post = ...
```

Is replaced by the Kotlin compiler with:

```
1 fun publishTweet(tweet: Tweet, callback: Continuation<Post>): Unit
```

The `Continuation` carries all the information that the Kotlin runtime needs to suspend and resume execution from the different suspension points in our program. This makes `suspend` another good example of how requiring a calling context can serve as a means for carrying implicit information across the execution tree. Information that can be used at runtime to enable advanced language features.

In the same way, we could also understand `@Composable` as a language feature. It makes standard Kotlin functions restartable, reactive, etc.

A fair question to make at this point is why the Jetpack Compose team didn't use `suspend` for achieving their wanted behavior. Well, even if both features are really similar in the pattern they implement, both are enabling completely different features in the language.

The `Continuation` interface is very specific about suspending and resuming execution, so it is modeled as a callback interface, and Kotlin generates a default implementation for it with all the required machinery to do the jumps, coordinate the different suspension points, share data between them, and so on. The Compose use case is very different, since its goal is to create an in memory representation of a large call graph that can be optimized at runtime in different ways.

Once we understand the similarities between Composable and `suspend` functions, it can be interesting to reflect on the idea of “function coloring”.

## The color of Composable functions

Composable functions have different limitations and capabilities than standard functions. They have a different type (more on this later), and model a very specific concern. This differentiation can be understood as a form of “function coloring”, since somehow they represent a separate **category of functions**.

“Function coloring” is a concept explained by Bob Nystrom from the Dart team at Google in a blockpost called “[What color is your function?](#)”<sup>3</sup>, written in 2015. He explained how `async` and `sync` functions don't compose well together, since you cannot call `async` functions from `sync` ones, unless you make the latter also `async`, or provide an awaiting mechanism that allows to call `async` functions and await for their result. This is why Promises and `async/await` were introduced by some libraries

<sup>3</sup><https://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/>

and languages. It was an attempt to bring composability back. Bob refers to these two function categories as two different “function colors”.

In Kotlin, suspend aims to solve the same problem. However, suspend functions are also colored, since we can only call suspend functions from other suspend functions. Composing programs with a mix of standard and suspend functions requires some ad-hoc integration mechanism (coroutine launch points). The integration is not transparent to the developer.

Overall, this limitation is expected. We are modeling two categories of functions that represent concepts of a very different nature. It’s like speaking two different languages. We have operations that are meant to calculate an immediate result (sync), and operations that unfold over time and eventually provide a result (async), which will likely take longer to complete.

In Jetpack Compose, the case of Composable functions is equivalent. We cannot call Composable functions from standard functions transparently. If we want to do that, an integration point is required (e.g: `Composition.setContent`). Composable functions have a completely different goal than standard functions. They are not designed to write program logics, but to describe changes for a node tree.

It might seem that I am tricking a bit here. One of the benefits of Composable functions is that you can declare UI using logics, actually. That means sometimes we need to call Composable functions from standard functions. For example:

```
1 @Composable
2 fun SpeakerList(speakers: List<Speaker>) {
3     Column {
4         speakers.forEach {
5             Speaker(it)
6         }
7     }
8 }
```

The Speaker Composable is called from the `forEach` lambda, and the compiler does not seem to complain. How is it possible to mix function colors this way then?

The reason is `inline`. Collection operators are declared as `inline`, so they inline their lambdas into their callers making it effectively as if there was no extra indirection. In the above example, the Speaker Composable call is inlined within the `SpeakerList` body, and that is allowed since both are Composable functions. By leveraging `inline` we can bypass the problem of function coloring to write the logic of our Composables. Our tree will be comprised of Composable functions only.

But, is coloring really a problem?

Well, it might be if we needed to combine both types of functions and jump from one to the other all the time. However, that is not the case either for `suspend` or `@Composable`. Both mechanisms require an integration point, and therefore we gain a completely colored call stack beyond that point (everything `suspend`, or Composable). This is actually an advantage, since it allows the compiler and

runtime to treat colored functions differently, and enable more advanced language features that were not possible with standard functions.

In Kotlin, `suspend` allows to model async non-blocking programs in a very idiomatic and expressive manner. The language gains the ability to represent a very complex concept in an extremely simple way: adding a `suspend` modifier to our functions. On the other hand, `@Composable` makes standard functions become restartable, skippable, and reactive, which are capabilities that standard Kotlin functions do not have.

## Composable function types

The `@Composable` annotation effectively changes the type of the function at compile time. From a syntax perspective, the type of a Composable function is `@Composable (T) -> A`, where `A` can be `Unit`, or any other type if the function returns a value (e.g: `remember`). Developers can use that type to declare Composable lambdas as one would declare any standard lambda in Kotlin.

```
1 // This can be reused from any Composable tree
2 val textComposable: @Composable (String) -> Unit = {
3     Text(
4         text = it,
5         style = MaterialTheme.typography.subtitle1
6     )
7 }
8
9 @Composable
10 fun NamePlate(name: String, lastname: String) {
11     Column(modifier = Modifier.padding(16.dp)) {
12         Text(
13             text = name,
14             style = MaterialTheme.typography.h6
15         )
16         textComposable(lastname)
17     }
18 }
```

Composable functions can also have the type `@Composable Scope.() -> A`, frequently used for scoping information to a specific Composable only:

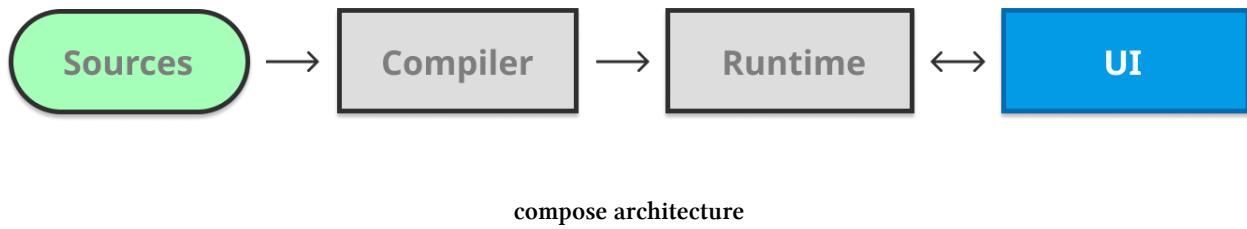
```
1 inline fun Box(  
2     ...,  
3     content: @Composable BoxScope.() -> Unit  
4 ) {  
5     // ...  
6     Layout(  
7         content = { BoxScopeInstance.content() },  
8         measurePolicy = measurePolicy,  
9         modifier = modifier  
10    )  
11 }
```

From a language perspective, types exist to provide information to the compiler in order to perform quick static validation, sometimes generate some convenient code, and to delimit/refine how the data can be used at runtime. The `@Composable` annotation changes how a function is validated and used at runtime, and that is also why they are considered to have a different type than normal functions.

## 2. The Compose compiler

Jetpack Compose is comprised of a series of libraries, but this book is going to heavily focus on three specific ones: The Compose compiler, the Compose runtime, and Compose UI.

The Compose compiler and the runtime are the pillars of Jetpack Compose. Compose UI is not technically part of the Compose architecture, since the runtime and the compiler are designed to be generic and consumed by any client libraries that comply to their requirements. Compose UI only happens to be one of the available clients. There are also other client libraries in the works, like the ones for desktop and web by JetBrains. That said, going over Compose UI will help us to understand how Compose feeds the runtime in-memory representation of the Composable tree, and how it eventually materializes real elements from it.



In a first approach to Compose one might feel a bit confused about what's the exact order of things. Up to this point in the book we've been told that the compiler and the runtime work together to unlock all the library features, but that probably still remains a bit too abstract if we are not familiar with it already. We'd likely welcome deeper explanations on what actions the Compose compiler takes to make our code comply with the runtime requirements, how the runtime works, when initial composition and further recompositions are triggered, how the in-memory representation of the tree is feeded, how that information is used for further recompositions... and much more. Grasping concepts like these can help us to grow an overall sense on how the library works and what to expect from it while we are coding.

Let's go for it, and let's start by understanding the compiler.

### A Kotlin compiler plugin

Jetpack Compose relies a bit on code generation. In the world of Kotlin and the JVM, the usual way to go for this is annotation processors via **kapt**, but Jetpack Compose is different. The Compose compiler is actually a Kotlin compiler plugin instead. This gives the library the ability to embed its compile time work within the Kotlin compilation phases, gaining access to more relevant

information about the shape of code, and speeding up the overall process. While kapt needs to run prior to compilation, a compiler plugin is directly **inlined in the compilation process**.

Being a Kotlin compiler plugin also gives the chance to report diagnostics in the frontend phase of the compiler, providing a very fast feedback loop. However, these diagnostics will not get reported in the IDE, since IDEA is not directly integrated with the plugin. Any IDEA-level inspections we can find in Compose today have been added via a separate IDEA plugin which doesn't share any code with the Compose compiler plugin. That said, frontend diagnostics will be reported as soon as we hit the compile button. Improving the feedback loop is the ultimate benefit of static analysis in the frontend phase of a Kotlin compiler, and the Jetpack Compose compiler makes good use of that.

Another big advantage of Kotlin compiler plugins is that they can tweak the existing sources at will (not only add new code, like annotation processors do). They are able to modify the output IR for those elements before it gets lowered to more atomic terms that then can be translated to primitives supported by the target platforms –remember Kotlin is multiplatform–. We will dive a bit more into this in this chapter, but this will give the Compose compiler the ability to **transform the Composable functions** as the runtime requires.

Compiler plugins have a bright future in Kotlin. Many known annotation processors out there will likely be migrated gradually to become compiler plugins or “lightweight” compiler plugins via KSP (Kotlin Symbol Processing library. See the blockquote below).

If you are particularly interested in Kotlin compiler plugins I would highly recommend checking [KSP \(Kotlin Symbol Processing\)<sup>a</sup>](#), a library that Google is proposing as a replacement for Kapt. KSP proposes a normalized DSL for “writing lightweight compiler plugins” that any libraries can rely on for metaprogramming. Make sure to give a read to the [“Why KSP” section<sup>b</sup>](#) in the KSP repository.

Also, note that the Jetpack Compose compiler relies a lot on IR transformation and that might be dangerous if used as a widespread practice around meta-programming. If all annotation processors out there were translated to compiler plugins we might end up with too many IR transforms, and something like that might destabilize the language. Tweaking/Augmenting the language always comes with a risk. That is why KSP is probably a better pick, overall.

<sup>a</sup><https://github.com/google/ksp>

<sup>b</sup><https://github.com/google/ksp/blob/main/docs/why-ksp.md>

## Compose annotations

Back to the order of things. The first thing we need to look at is how we annotate our code so the compiler can scan for the required elements and do its magic. Let's start by learning about the Compose annotations available.

Even if compiler plugins can do quite more than annotation processors, there are some things that both have in common. One example of this is their frontend phase, frequently used for static analysis

and validation.

The Compose compiler utilizes hooks/extension points in the kotlin compiler's frontend to verify that the constraints it would like to enforce are met and that the type system is properly treating `@Composable` functions, declarations, or expressions, different from non-`Composable` ones.

Apart from that, Compose also provides other complementary annotations meant to unlock additional checks and diverse runtime optimizations or “shortcuts” under some certain circumstances. All the annotations available are provided by the Compose runtime library.

Let's start by making a deep dive into the most relevant annotations.

All the Jetpack Compose annotations are provided by the Compose runtime, since it is both the compiler and the runtime the modules making good use of those.

## **@Composable**

This annotation was already covered in depth in chapter 1. That said, it likely requires a standalone section since it is clearly the most important one. That is why it will be the first on my list.

The biggest difference between the Compose compiler and an annotation processor is that Compose effectively **changes** the declaration or expression that is annotated. Most annotation processors can't do this, they have to produce additional / sibling declarations. That is why the Compose compiler uses IR transforms. The `@Composable` annotation actually **changes the type** of the thing, and the compiler plugin is used to enforce all kinds of rules in the frontend to ensure that Composable types aren't treated the same as their non-composable-annotated equivalents.

Changing the type of the declaration or expression via `@Composable` gives it a “memory”. That is the ability to call `remember` and utilize the Composer/slot table. It also gives it a lifecycle that effects launched within its body will be able to comply with. –E.g: spanning a job across recompositions.– Composable functions will also get assigned an identity that they will preserve, and will have a position in the resulting tree, meaning that they can emit nodes into the Composition and address `CompositionLocals`.

Small recap: A Composable function represents mapping from data to a node that is emitted to the tree upon execution. This node can be a UI node, or a node of any other nature, depending on the library we are using to consume the Compose runtime. The Jetpack Compose runtime works with **generic types of nodes** not tied to any specific use case or semantics. We will cover that topic in detail towards the end of this book, when our understanding of Compose is more diverse and much richer.

## @ComposeCompilerApi

This annotation is used by Compose to flag some parts of it that are only meant to be used by the compiler, with the only purpose of informing potential users about this fact, and to let them know that it should be used with caution.

## @InternalComposeApi

Some APIs are flagged as internal in Compose since they are expected to vary internally even if the public API surface remains unchanged and frozen towards a stable release. This annotation has a wider scope than the language `internal` keyword, since it allows usage across modules, which is a concept that Kotlin does not support.

## @DisallowComposableCalls

Used to prevent composable calls from happening inside of a function. This can be useful for `inline` lambda parameters of Composable functions that cannot safely have composable calls in them. It is best used on lambdas which will **not** be called on every recomposition.

An example of this can be found in the `remember` function, part of the Compose runtime. This Composable function remembers the value produced by the `calculation` block. This block is only evaluated during the initial composition, and any further recompositions will always return the already produced value.

### Composables.kt

---

```
1 @Composable
2 inline fun <T> remember(calculation: @DisallowComposableCalls () -> T): T =
3     currentComposer.cache(false, calculation)
```

---

Composable calls are forbidden within the `calculation` lambda thanks to this annotation. If those were allowed otherwise, they would take space in the slot table when invoked (emitting), and that would then be disposed of after the first composition, since the lambda isn't invoked anymore.

This annotation is best used on inline lambdas that are called conditionally as an implementation detail, but should not be “alive” like composable functions are. This is only needed because inline lambdas are special in that they “inherit” the composable abilities of their parent calling context. For example, the lambda of a `forEach` call is not marked as `@Composable`, but you can call composable functions if the `forEach` itself is called within a composable function. This is desired in the case of `forEach` and many other inline APIs, but it is **not** desirable in some other cases like `remember`, which is where this annotation comes in.

Also note that this annotation is “contagious” in the sense that if you invoke an inline lambda inside of an inline lambda marked as `@DisallowComposableCalls`, the compiler will require that you mark that lambda as `@DisallowComposableCalls` as well.

As you probably guessed, this is likely an annotation you might not ever use in any client projects, but could become more relevant if you are using Jetpack Compose for a different use case than Compose UI. In that case you'll likely need to write your own client library for the runtime, and that will require you to comply with the runtime constraints.

## @ReadOnlyComposable

When applied over a Composable function it means we know that the body of this Composable will not write to the composition ever, only read from it. That also must remain true for all nested Composable calls in the body. This allows the runtime to avoid generating code that will not be needed if the Composable can live up to that assumption.

For any Composable that writes to the composition inside, the compiler generates a “group” that wraps its body, so the whole group is emitted at runtime instead. These emitted groups provide the required information about the Composable to the composition, so it knows how to cleanup any written data later when a recomposition needs to override it with the data of a different Composable, or how to move that data around by preserving the identity of the Composable. There are different types of groups that can be generated: E.g: restartable groups, movable groups... etc.

For more context on what exactly a “group” is, imagine a couple of pointers at the start and end of a given span of selected text. All groups have a source position key, which is used to store the group, and therefore what unlocks positional memoization. That key is also how it knows different identity between `if` or `else` branches of conditional logics like:

ConditionalTexts.kt

---

```
1 if (condition) {  
2     Text("Hello")  
3 } else {  
4     Text("World")  
5 }
```

---

These are both `Text`, but they have different identity since they represent different ideas to the caller. Movable groups also have a semantic identity key, so they can reorder within their parent group.

When our composable does not write to the composition, generating those groups does not provide any value, since its data is not going to be replaced or moved around. This annotation helps to avoid it.

Some examples of read only Composables within the Compose libraries could be many `CompositionLocal` defaults or utilities that delegate on those, like the `Material Colors`, `Typography`, the `isSystemInDarkTheme()` function, the `LocalContext`, any calls to obtain application resources of any type –since they rely on the `LocalContext`–, or the `LocalConfiguration`. Overall, it is about things that are only set once when running our program and are expected to stay the same and be available to be read from Composables on the tree.

## @NonRestartableComposable

When applied on a function or property getter, it basically makes it be a non-restartable Composable. (Note that not all Composables are restartable by default either, since inline Composables or Composables with non-Unit return types are not restartable).

When added, the compiler does not generate the required boilerplate needed to allow the function to recompose or be skipped during recomposition. Please keep in mind that this has to be used very sparingly, since it might only make sense for very small functions that are likely getting recomposed (restarted) by another Composable function that calls them, since they might contain very little logic so it doesn't make much sense for them to self invalidate. Their invalidation / recomposition will be essentially driven by their parent/enclosing Composable, in other words.

This annotation should rarely/never be needed for “correctness”, but can be used as a very slight performance optimization if you know that this behavior will yield better performance, which is sometimes the case.

## @StableMarker

The Compose runtime also provides some annotations to denote the stability of a type. Those are the `@StableMarker` meta-annotation, and the `@Immutable` and `@Stable` annotations. Let's start with the `@StableMarker` one.

`@StableMarker` is a meta-annotation that annotates other annotations like `@Immutable` and `@Stable`. This might sound a bit redundant, but it is meant for reusability, so the implications it has also apply over all the annotations annotated with it.

`@StableMarker` implies the following requirements related to data stability for the ultimately annotated type:

- The result of calls to `equals` will always be the same for the same two instances.
- Composition is always notified when a public property of the annotated type changes.
- All the public properties of the annotated type are also stable.

Any types annotated with `@Immutable` or `@Stable` will also need to imply these requirements, since both annotations are flagged as a `@StableMarker`, or in other words, as markers for stability.

Note how these are promises we give to the compiler so it can make some assumptions when processing the sources, but **they are not validated at compile time**. That means it is up to you, the developer, to decide when all the requirements are met.

That said, the Compose compiler will do its best to infer when certain types meet the requirements stated above and treat the types as stable without being annotated as such. In many cases this is preferred as it is guaranteed to be correct, however, there are two cases when annotating them directly is important:

- When it's a required contract/expectation of an interface or abstract class. This annotation becomes not only a promise to the compiler but a **requirement** to the implementing declaration (and unfortunately, one that is not validated in any way).
- When the implementation is mutable, but is implemented in a way where the mutability is safe under the assumptions of stability. The most common example of this is if the type is mutable because it has an internal cache of some sort, but the corresponding public API of the type is independent of the state of the cache.

Class stability inference will be explained later in this chapter with much deeper detail.

## @Immutable

This annotation is applied over a class as a strict promise for the compiler about all the publicly accessible class properties and fields remaining unchanged after creation. Note that this is a **stronger promise** than the language `val` keyword, since `val` only ensures that the property can't be reassigned via setter, but it can point to a mutable data structure for example, making our data mutable even if it only has `val` properties. That would break the expectations of the Compose runtime. In other words, this annotation is needed by Compose essentially because the Kotlin language does not provide a mechanism (a keyword or something) to ensure when some data structure is immutable.

Based on the assumption that the value reads from the type will never change after initialized, the runtime can apply optimizations to the smart recomposition and skipping recomposition features.

One good example of a class that could safely be flagged as `@Immutable` would be a data class with `val` properties only, where none of them have custom getters –that would otherwise get computed on every call and potentially return different results every time, making it become a non-stable api to read from– and all of them are either primitive types, or types also flagged as `@Immutable`.

`@Immutable` is also a `@StableMarker` as explained above, so it also inherits all the implications from it. A type that is considered immutable always obeys the implications stated for a `@StableMarker`, since its public values will never change. `@Immutable` annotation exists to flag immutable types as stable.

An extra pointer regarding last paragraph: It is worth noting that immutable types don't notify composition of their values changing, which is one of the requirements listed in `@StableMarker`, but they **don't really have to**, because their values don't change, so it satisfies the constraint anyway.

## @Stable

This one might be a bit of a lighter promise than `@Immutable`. It has different meaning depending on what language element it is applied to.

When this annotation is applied to a type, it means the type is **mutable** –(We'd use `@Immutable` otherwise)– and it will only have the implications inherited by `@StableMarker`. Feel free to read them once again to refresh your memory.

When `@Stable` annotation is applied to a function or a property instead, it tells the compiler that the function will always return the same result for the same inputs (pure). This is only possible when the parameters of the function are also `@Stable`, `@Immutable` or primitive types (those are considered stable).

There is a nice example of how relevant this is for the runtime in docs: *When all types passed as parameters to a Composable function are marked as stable then the parameter values are compared for equality based on positional memoization and the call is skipped if all the values are the equal to the previous call.*

An example of a type that could be flagged as `@Stable` is an object whose public properties do not change but cannot be considered immutable. For example, it has private mutable state, or it uses property delegation to a `MutableState` object, but is otherwise immutable in terms of how it is used from the outside.

One more time, the implications of this annotation are used by the compiler and the runtime to make assumptions over how our data will evolve (or not evolve) and take shortcuts where required. And once again, this annotation should never be used unless you are completely sure its implications are fulfilled. Otherwise we'd be giving incorrect information to the compiler and that would easily lead to runtime errors. This is why all these annotations are recommended to be used sparingly.

Something interesting to highlight is that even if both `@Immutable` and `@Stable` annotations are different promises with different meaning, today the Jetpack Compose compiler **treats both the same way**: To enable and optimize smart recomposition and skipping recompositions. Still both exist to leave the door open for different semantics to impose a differentiation that the compiler and the runtime might want to leverage in the future.

## Registering Compiler extensions

Once we have peeked into the most relevant available annotations **provided by the runtime**, it's time to understand how the Compose compiler plugin works and how it makes use of those annotations.

The first thing the Compose compiler plugin does is registering itself into the Kotlin compiler pipeline using a `ComponentRegistrar`, which is the mechanism that the Kotlin compiler provides for this matter. The `ComposeComponentRegistrar` registers a series of compiler extensions for different purposes. These extensions will be in charge of easing the use of the library and generating the required code for the runtime. All the registered extensions will run along with the Kotlin compiler.

The Compose compiler also registers a few extensions depending on the compiler flags enabled. Developers using Jetpack Compose have the chance to enable a few specific compiler flags that allow them to enable features like live literals, including source information in the generated code so Android Studio and other tools can inspect the composition, optimizations for remember functions, suppressing Kotlin version compatibility checks, and/or generating decoy methods in the IR transformation.

If we are interested on digging deeper on how compiler extensions are registered by the compiler plugin, or other further explorations, remember that we can always browse the sources on [cs.android.com](https://cs.android.com/)<sup>a</sup>.

<sup>a</sup><https://cs.android.com/androidx/platform/frameworks/support/+/androidx-main:compose/compiler/compiler-hosted/src/main/java/androidx/compose/compiler/plugins/kotlin/ComposePlugin.kt>

## Kotlin Compiler version

The Compose compiler requires a very specific version of Kotlin, so it checks whether the Kotlin compiler version used matches the required one. This is the first check happening since it is a big blocker if not fulfilled.

There is the chance to bypass this check by using the `suppressKotlinVersionCompatibilityCheck` compiler argument, but that is at our own risk, since then we become able to run Compose with any version of Kotlin, which could easily lead to important inconsistencies. Even more if we think about the evolution of the Kotlin compiler backends in latest Kotlin versions. This parameter was probably added to allow running and testing Compose against experimental Kotlin releases and the like.

## Static analysis

Following the standard behavior of an average compiler plugin, the first thing that happens is **linting**. Static analysis is done by scanning the sources searching for any of the library annotations and then performing some important checks to make sure they are used correctly. And by correctly I mean the way the runtime expects. Here, relevant warnings or errors are reported via the context trace, which compiler plugins have access to. This integrates well with **idea**, since it is already prepared to display those warnings or errors inline while the developer is still typing. As mentioned earlier, all these validations take place in the frontend phase of the compiler, helping Compose to provide the fastest possible feedback loop for developers.

Let's have a look to some of the most important static checks performed.

## Static Checkers

Some of the registered extensions come in the form of static checkers that will guide developers while coding. Checkers for calls, types and declarations are registered as extensions by Jetpack Compose. They will ensure the correct use of the library, and are obviously opinionated towards the problem the library wants to solve. Things like requirements for Composable functions like the ones we learned in chapter 1 are validated here and reported when violated.

In the world of Kotlin compilers there are different types of analyzers available depending on the element we want to check. There are checkers for class instantiation, types, function calls, deprecated calls, contracts, capturing in closure, infix calls, coroutine calls, operator calls and many more that allow compiler plugins to analyze all the corresponding elements from the input sources and report information, warnings, or errors where needed.

Given all registered checkers run in the frontend phase of the Kotlin compiler, they are expected to be very fast and not contain very cpu consuming operations. That is a responsibility of the developer, so keeping always in mind that these checks will run while the developer types and that we do not want to create a janky user experience is key. We want to implement lightweight checkers.

## Call checks

One of the different kinds of checkers registered by Compose are the ones used to validate calls. The Compose compiler has static call checks in place for validating composable function calls in many different contexts, like when they are done under the scope of `@DisallowComposableCalls` or `@ReadOnlyComposable`.

A Call checker is a compiler extension used to perform static analysis on all calls across our codebase, so it provides a `check` function that is recursively called for visiting all the PSI elements that are considered calls in our sources. Or in other words: All nodes on the PSI tree. It's an implementation of the visitor pattern.

Some of these checks require a wider context than the current language element they are visiting, since they might need to know things like from where the Composable is called, for example. This means that analyzing a single PSI node is not enough. Gathering such information requires recording smaller bits of information from different elements visited, like a breadcrumb to build a story as a whole, and perform more complex validations on further passes. To do this, the compiler can record that information conveniently in the context `trace`. This allows to widen the scope for the checks in place, and be able to look for enclosing lambda expressions, try / catch blocks, or similar things that might be relevant.

Here is an example of a compiler call that records relevant information to the trace and also uses it to report an error when a Composable call is done within a context flagged with `@DisallowComposableCalls`:

### ContextTraceExamples.kt

```

1 if (arg?.type?.hasDisallowComposableCallsAnnotation() == true) {
2     context.trace.record(
3         ComposeWritableSlices.LAMBDA_CAPABLE_OF_COMPOSER_CAPTURE,
4         descriptor, // reference to the function literal
5         false
6     )
7     context.trace.report(
8         ComposeErrors.CAPTURED_COMPOSABLE_INVOCATION.on(
9             reportOn,
10            arg,
11            arg.containingDeclaration
12        )
13    )
14 }
15 }
```

The context and therefore the context trace are available from every call to the `check` function, and indeed it is the same trace we can also use to report errors, warnings or information messages. We can understand the trace as a mutable structure we can fill up with relevant information to carry across the overall analysis.

Other checks are simpler and only require the information available on the current element visited, so they perform their action and return. On every `check` call, the plugin will match the element type for the current node, and depending on it, it simply performs a check and returns –if everything is correct–, reports an error –if needed–, records relevant information to the context trace, or recurses again to the parent of this node to keep visiting more nodes and gather more information. Different checks for different annotations are performed along the way.

One thing that the Compose compiler checks is that Composables are not called from disallowed places, like from within a `try/catch` block (that is not supported), from a function not annotated as Composable also, or from lambdas annotated with `@DisallowComposableCalls`. –Remember that annotation was used to avoid composable annotations within inline lambdas.–

For each composable call, the compiler visits up the PSI tree checking its callers, the callers of its callers, and so on, to confirm that all the requirements for this call are fulfilled. All scenarios are taken into account, since parents can be lambda expressions, functions, properties, property accessors, `try/catch` blocks, classes, files, and more.

The PSI models the structure of the language for the frontend compiler phases, hence we must keep in mind that its way to understand code is completely syntactical and static.

It is also important for these checks to take `inline` functions into account, since it must be possible to call a Composable function from an inline lambda **as long as the callers of the inline lambda are also Composable**. The compiler checks that any inline lambdas calling Composable functions are also enclosed by a Composable function at some level up the call stack.

Another call check performed is the one detecting the potentially missing Composable annotations where they would be required or expected, so it can conveniently ask the developer to add those.  
-E.g: If a Composable function is being called within a lambda, compiler will friendly suggest to also add the Composable annotation to that lambda.- Static analysis checks exist to guide the developer while writing code, so it is not all about forbidding, sometimes they can infer and suggest what is needed or tell us how to improve our code.

There are also static call checks in place for Composable functions annotated as `@ReadOnlyComposable`. Those can only call other read only Composables, since otherwise we would be breaking the contract for the optimization, where a read only composable can only read from the composition, never write to it. Given this must be fulfilled at all depth levels within the Composable, the visitor pattern will come handy.

Another check we can find is the one to disallow the use of Composable function references, since that is not supported by Jetpack Compose at this point.

## Type checks

Sometimes we annotate types as Composable, not only functions. For that, the Compose compiler has a check related to type inference, so it can report when a type annotated with `@Composable` was expected but a non-annotated type was found instead. Similar to the check for function calls mentioned above. The error will print the inferred and expected types along with their annotations to make the difference more obvious.

## Declaration checks

Checks for call sites and types are needed, but not enough. Declaration sites are also part of any Compose codebase. Things like properties, property accessors, function declarations or function parameters need to be analyzed.

Properties, property getters and functions can be overridden, even when they are annotated as Composable. The Composer compiler has a check in place to ensure that any overrides of any of those KtElements is also annotated as Composable to keep coherence.

Another declaration check available is the one to ensure that Composable functions are not suspend, since that is not supported. As explained in chapter 1, `suspend` has a different meaning than `@Composable`, and even if both could be understood as language primitives somehow, they are designed to represent completely different things. Both concepts are not supported together as of today.

Things like Composable `main` functions or backing fields on Composable properties are also forbidden via declaration checks.

## Diagnostic suppression

Compiler plugins can register diagnostic suppressors as extensions so they can basically mute diagnostics for some specific circumstances –e.g: errors notified by static checks–. This is usual when compiler plugins generate or support code that the Kotlin compiler wouldn't normally accept, so that it can bypass the corresponding checks and make it work.

Compose registers a `ComposeDiagnosticSuppressor` to bypass some language restrictions that would otherwise fail compilation, so that can unleash some specific use cases.

One of these restrictions goes for inline lambdas annotated with “non-source annotations” on call sites. That is annotations with retention `BINARY` or `RUNTIME`. Those annotations survive until the output binaries, not like `SOURCE` annotations. Given inline lambdas are effectively inlined into their callers at compile time, they're not gonna be stored anywhere, so there will not be anything to annotate anymore at that point. That is why Kotlin forbids this and reports the following error:

*“The lambda expression here is an inlined argument so this annotation cannot be stored anywhere.”*

Here is an example on a piece of code that would trigger the error using plain Kotlin:

AnnotatedInlineLambda.kt

---

```

1 @Target(AnnotationTarget.FUNCTION)
2 annotation class FunAnn
3
4 inline fun myFun(a: Int, f: (Int) -> String): String = f(a)
5
6 fun main() {
7   myFun(1) @FunAnn { it.toString() } // Call site annotation
8 }
```

---

The Compose compiler suppresses this check only for cases where the annotation used is `@Composable`, so we can write code like the following:

**AnnotatedComposableInlineLambda.kt**


---

```

1  @Composable
2  inline fun MyComposable(@StringRes nameResId: Int, resolver: (Int) -> String) {
3      val name = resolver(nameResId)
4      Text(name)
5  }
6
7  @Composable
8  fun Screen() {
9      MyComposable(nameResId = R.string.app_name) @Composable {
10         LocalContext.current.resources.getString(it)
11     }
12 }
```

---

This allows to annotate our lambda parameters as `@Composable` on call sites, so we don't necessarily have to do it on the function declaration. This allows the function to have a more flexible contract.

Another language restriction that gets bypassed with the suppressor is related to allowing named arguments in places the Kotlin compiler would not support, but only in case the function they belong to is annotated as `@Composable`.

One example is function types. Kotlin does not allow named arguments on those, but Compose makes it possible if the function is annotated as `Composable`:

**NamedParamsOnFunctionTypes.kt**


---

```

1  interface FileReaderScope {
2      fun onFileOpen(): Unit
3      fun onFileClosed(): Unit
4      fun onLineRead(line: String): Unit
5  }
6
7  object Scope : FileReaderScope {
8      override fun onFileOpen() = TODO()
9      override fun onFileClosed() = TODO()
10     override fun onLineRead(line: String) = TODO()
11 }
```

---

```

13  @Composable
14  fun FileReader(path: String, content: @Composable FileReaderScope.(path: String) -> \
15  Unit) {
16      Column {
17          //...
18          Scope.content(path = path)
```

```
19     }  
20 }
```

---

If we remove the `@Composable` annotation we'll get an error like:

*“Named arguments are not allowed for function types.”*

Same requirement is suppressed in other cases like members of expected classes. Remember Jetpack Compose aims to be multiplatform, so the runtime should definitely accept expect functions and properties flagged as Composable.

## Runtime version check

We already have all the static checkers and the diagnostic suppressor installed. We can move on to more interesting things. The first thing happening right before code generation is a check for the Compose runtime version used. The Compose compiler requires a minimum version of the runtime so it has a check in place to ensure that the runtime is not outdated. It is able to detect both when the runtime is missing and when it is outdated.

A single Compose compiler version can support multiple runtime versions as long as they are higher than the minimum supported one.

This is the second version check in place. There is one for the Kotlin compiler version, and then this other one for the Jetpack Compose runtime.

## Code generation

Finally, it's time for the Compiler to move on to the code generation phase. That is another thing annotation processors and compiler plugins have in common, since both are frequently used to synthesize convenient code that our runtime libraries will consume.

## The Kotlin IR

As explained previously, compiler plugins have the ability to modify the sources, not only generate new code, since they have access to the **intermediate representation** of the language (IR) before it yields the ultimate code for the target platform/s. That means the compiler plugin can sneak in and replace parameters, add new ones, reshape the structure of code before “committing” it. This takes place in the backend phases of the Kotlin compiler. And as you probably guessed, this is what Compose does for “injecting” the implicit extra parameter, the Composer, to each Composable call.

Compiler plugins have the ability to generate code in different formats. If we were only targeting the JVM we could think of generating Java compatible bytecode, but following the latest plans and

refactors from the Kotlin team towards stabilizing all the IR backends and normalizing them into a single one for all platforms, it makes much more sense to generate IR. Remember that the IR exists as a representation of the language elements that **remains agnostic of the target platform** –an “intermediate representation”–. That means generating IR will potentially make Jetpack Compose generated code multiplatform.

The Compose compiler plugin generates IR by registering an implementation of the `IrGenerationExtension`, which is an extension provided by the Kotlin compiler common IR backend.

If you want to learn Kotlin IR in depth I highly recommend to check [these series<sup>a</sup>](#) by Brian Norman that covers the Kotlin IR, and the compiler plugin creation topic, overall. It helped me to learn a lot of interesting things. Learning IR in depth is necessarily out of the scope of this book.

<sup>a</sup><https://blog.bnrm.dev/writing-your-second-compiler-plugin-part-2>

## Lowering

The term “lowering” refers to the translation compilers can do from higher level or more advanced programming concepts to a combination of lower level more atomic ones. This is pretty common in Kotlin, where we have an intermediate representation (IR) of the language that is able to express pretty advanced concepts that then need to get translated to lower level atomics before transforming them to JVM byte code, Javascript, LLVM’s IR, or whatever platform we target. The Kotlin compiler has a lowering phase for this matter. Lowering can also be understood as a form of normalization.

The Compose compiler needs to lower some of the concepts the library supports, so they are normalized to a representation that the runtime can understand. The process of lowering is the actual code generation phase for the Compose compiler plugin. This is where it visits all the elements from the IR tree and tweaks the IR at will for those required based on the runtime needs.

Here is a brief summary of a few meaningful examples of things happening during the lowering phase that we are covering in this section:

- Inferring class stability and adding the required metadata to understand it at runtime.
- Transforming live literal expressions so they access a mutable state instance instead so it is possible for the runtime to reflect changes in the source code without recompiling (live literals feature).
- Injecting the implicit `Composer` parameter on all Composable functions and forwarding it to all Composable calls.

- Wrapping Composable function bodies for things like:
  - Generating different types of groups for control-flow (replaceable groups, movable groups...).
  - Implementing default argument support, so they can be executed within the scope of the generated group of the function –instead of relying on Kotlin default param support–.
  - Teach the function to skip recompositions.
  - Propagating relevant information regarding state changes down the tree so they can automatically recompose when they change.

Let's learn all kinds of lowering applied by the Jetpack Compose compiler plugin.

## Inferring class stability

Smart recomposition means skipping recomposition of Composables when their inputs have not changed **and those inputs are considered stable**. Stability is a very relevant concept in this sense, since it means that the Compose runtime can safely read and compare those inputs to skip recomposition when needed. The ultimate goal of stability is to help the runtime.

Following this line of thought, let's recap over the properties that a stable type must fulfill:

- Calls to `equals` for two instances always return the same result for the same two instances. That means comparison is coherent, so the runtime can rely on it.
- Composition is always notified when a public property of the type changes. Otherwise, we could end up with desynchronization between the input of our Composable and the latest state they reflect when materialized. To make sure this doesn't happen, recomposition is always triggered for cases like this one. **Smart recomposition can't rely on this input**.
- All public properties have primitive types or types that are also considered stable.

All primitive types are considered stable by default, but also `String`, and all the function types. That is because they're immutable by definition. Since immutable types do not change they do not need to notify the composition either.

We also learned that there are types that are not immutable but can be assumed as stable by Compose –they can be annotated with `@Stable`–. One example of this is `MutableState`, since Compose is notified every time it changes, hence it's safe to rely on it for smart recomposition.

For custom types that we create in our code, we can determine if they comply with the properties listed above, and flag them manually as stable using the `@Immutable` or `@Stable` annotations conveniently. But relying on developers to keep that contract fulfilled can be quite risky and hard to maintain over time. Inferring class stability automatically would be desirable instead.

Compose does this. The algorithm to infer stability is in constant evolution, but it goes along the lines of visiting every class and synthetizing an annotation called `@StabilityInferred` for it. It also

adds a synthetic `static final int $stable` value that encodes the relevant stability information for the class. This value will help the compiler to generate extra machinery in later steps to determine the stability of the class at runtime, so Compose can determine if our Composable functions that depend on this class need to recompose or not.

Truth be told, it's not literally every class, but only every eligible class. That is every public class that is not an enum, an enum entry, an interface, an annotation, an anonymous object, an `expect` element, an inner class, a companion, or `inline`. Also not if it's already flagged as stable with the annotations mentioned above, as you probably guessed. So, overall, it's just classes, data classes, and the like that have not been annotated as stable already. It makes sense, given that is what we'll use to model the inputs of our Composable functions.

To infer the stability of a class, Compose has different things into account. A type is inferred stable when all of the fields of the class are readonly and stable. Referring to “field” as in terms of the resulting JVM bytecode. Classes like `class Foo`, or `class Foo(val value: Int)` will be inferred as stable, since they have no fields or stable fields only. Then things like `class Foo(var value: Int)` will be inferred as unstable right away.

But things like the class generic type parameters might also affect class stability, e.g:

#### FooWithTypeParams.kt

---

```
1 class Foo<T>(val value: T)
```

---

In this case, `T` is used for one of the class parameters and therefore stability of `Foo` will rely on stability of the type passed for `T`. But given `T` is not a reified type, it will be unknown until the runtime. Therefore, it needs to exist some machinery to determine stability of a class at runtime, once the type passed for `T` is known. To solve this, the Compose compiler calculates and puts a bitmask into the `StabilityInferred` annotation that indicates that calculating the stability of this class at runtime should depend on the stability of the corresponding type parameter/s.

But having generic types does not necessarily mean unstable. The compiler knows that for example, code like: `class Foo<T>(val a: Int, b: T) { val c: Int = b.hashCode() }` is stable since `hashCode` is expected to always return the same result for the same instance. It's part of the contract.

For classes that are composed of other classes, like `class Foo(val bar: Bar, val bazz: Bazz)`, stability is inferred as the combination of the stability of all the arguments. This is resolved recursively.

Things like internal mutable state also make a class unstable. One example of this could be the following:

**Counter.kt**


---

```

1 class Counter {
2     private var count: Int = 0
3     fun getCount(): Int = count
4     fun increment() { count++ }
5 }
```

---

This state mutates over time even if it's mutated internally by the class itself. That means the runtime can't really trust on it being consistent.

Overall the Compose compiler considers a type stable only when it can prove it. E.g: an interface is assumed to be unstable, because Compose doesn't know how it's going to be implemented.

**MyListOfItems.kt**


---

```

1 @Composable
2 fun <T> MyListOfItems(items: List<T>) {
3     // ...
4 }
```

---

In this example we get a `List` as an argument, which can be implemented in mutable ways –see: `ArrayList`. It is not safe for the compiler to assume we'll be using immutable implementations only, and inferring that is not that easy, so it will assume it's unstable.

Another example is types with mutable public properties whose implementation could be immutable. Those are also considered not stable by default, since the compiler is not able to infer that much.

This is a bit of a con, since many times those things could be implemented to be immutable and for what is worth for the Compose runtime that should be enough. For that reason, if a model that we are using as input for our Composable functions is considered unstable by Compose, we can still flag it as `@Stable` explicitly if we have more information in our hand and its under our control. The official docs give this example, which is pretty meaninful:

**UiState.kt**


---

```

1 // Marking the type as stable to favor skipping and smart recompositions.
2 @Stable
3 interface UiState<T : Result<T>> {
4     val value: T?
5     val exception: Throwable?
6
7     val hasError: Boolean
8         get() = exception != null
9 }
```

---

There are more cases covered by the class stability inference algorithm. For all the cases covered by this feature I definitely suggest you to [read the library tests<sup>4</sup>](#) for the `ClassStabilityTransform`.

Keep in mind internals of how stability is inferred by the compiler can likely vary and get improved over time. The good point is that it will always be transparent for the library users.

## Enabling live literals

**Disclaimer:** This section is quite close to implementation details and is in constant evolution. It might change again multiple times in the future and evolve in different ways as more efficient implementations are figured out.

One of the flags we can pass to the compiler is the live literals one. Over time there have been two implementations of this feature, so you can enable either one or the other using the `liveLiterals` (v1) or `liveLiteralsEnabled` (v2) flags.

Live literals is this feature where the Compose tools are able to reflect changes live in the preview without the need for recompilation. What the compose compiler does is replacing those expressions by new versions of them that read their values from a `MutableState` instead. That allows the runtime to be notified about changes instantly, without the need for recompiling the project. As the library kdocs expose:

*“This transformation is intended to improve developer experience and should never be enabled in a release build as it will significantly slow down performance-conscious code”*

The Compose compiler will generate unique ids for each single constant expression in our codebase, then it transforms all those constants into property getters that read from some `MutableState` that is held into a generated singleton class per file. At runtime, there are apis to obtain the value for those constants using the generated key.

Here is an example extracted from the library kdocs. Let's say we start with this Composable:

---

<sup>4</sup><https://cs.android.com/androidx/platform/frameworks/support/+/androidx-main:compose/compiler/compiler-hosted/integration-tests/src/test/java/androidx/compose/compiler/plugins/kotlin/ClassStabilityTransformTests.kt>

**LiveLiterals1.kt**


---

```

1 @Composable
2 fun Foo() {
3     print("Hello World")
4 }
```

---

Will get transformed to the following.

**LiveLiterals2.kt**


---

```

1 // file: Foo.kt
2 @Composable
3 fun Foo() {
4     print(LiveLiterals$FooKt.`getString$arity-0$call-print$fun-Foo`())
5 }
6
7 object LiveLiterals$FooKt {
8     var `String$arity-0$call-print$fun-Foo`: String = "Hello World"
9     var `State$String$arity-0$call-print$fun-Foo`: MutableState<String>? = null
10    fun `getString$arity-0$call-print$fun-Foo`(): String {
11
12        val field = this.`String$arity-0$call-print$fun-Foo`
13
14        val state = if (field == null) {
15            val tmp = liveLiteral(
16                "String$arity-0$call-print$fun-Foo",
17                this.`String$arity-0$call-print$fun-Foo`
18            )
19            this.`String$arity-0$call-print$fun-Foo` = tmp
20            tmp
21        } else field
22
23        return field.value
24    }
25 }
```

---

We can see how the constant is replaced by a getter that reads from the `MutableState` held into the generated singleton for the corresponding file.

## Compose lambda memoization

This step generates convenient IR to teach the runtime how to optimize the execution of lambdas passed to Composable functions. This work is done for two types of lambdas:

- **Non-composable lambdas:** The compiler generates IR for memoizing these by wrapping each lambda into a `remember` call. Think of a callback we pass to a Composable function, for example. Here, `remember` allows to appeal to the slot table to store and read these lambdas later.
- **Composable lambdas:** The compiler generates IR to wrap them and add relevant information to teach the runtime how to store and read the expression to/from the Composition. This has the same final goal than using `remember`, but it is not using it. An example of this can be the content Composable lambdas we pass to our Compose UI nodes when calling them.

## Non-composable lambdas

This action optimizes lambda calls passed to Composable functions so they can be reused. Kotlin already optimizes lambdas when they don't capture any values by modeling them as singletons, so there is a single reusable instance for the complete program. That said, this optimization is not possible when lambdas capture values, since those values might vary per call making it unique, and therefore a different instance per lambda will be needed. Compose is smarter for this specific case. Let's explore this using an example:

NamePlateClickLambda.kt

```
1 @Composable
2 fun NamePlate(name: String, onClick: () -> Unit) {
3     // ...
4     // onClick()
5     // ...
6 }
```

Here, `onClick` is a standard Kotlin lambda that is passed to a Composable function. If the lambda we pass to it from the call site captures any values, Compose has the ability to teach the runtime how to memoize it. That basically means wrapping it into a call to `remember`. This is done via the generated IR. This call will remember the lambda expression based on the values it captures, as long as these values are stable. This allows the runtime to reuse lambdas that already exist instead of creating new ones, as long as the values they capture match (input parameters included).

The reason to require the captured values to be stable is that they will be used as condition arguments for the `remember` call, so they must be reliable for comparison.

Note that memoized lambdas **cannot be inline**, since otherwise there would be nothing to remember after they are inlined on their callers at compile time.

This optimization only makes sense for lambdas that capture. If they don't, Kotlin's default optimization –representing those as singletons– is sufficient.

As explained above, memoization is done based on the lambda captured values. When generating the IR for the expression, the compiler will prepend a call to `remember` with the return type matching the type of the memoized expression, then it will add the generic type argument to the call – `remember<T>...` – to match the expression return type. Right after, it will add all the values captured by the lambda as condition arguments –`remember<T>(arg1, arg2...)`– so they can be used for comparison, and finally, the lambda for the expression `remember<T>(arg1, arg2..., expression)` so it can work as a trailing lambda–.

Using the captured values as condition arguments will ensure that they are used as keys for remembering the result of the expression, so it will be invalidated whenever those vary.

**Automatically remembering lambdas** passed to Composable functions unlocks reusability when recomposition takes place.

## Composable Lambdas

The Compose compiler is also able to memoize Composable lambdas. Implementation details just happen to be different for this case given the “special” way Composable lambdas are implemented. But the ultimate goal is the same: Store and read these lambdas to/from the slot table.

Here is an example of a Composable lambda that will get memoized:

Container.kt

---

```

1 @Composable
2 fun Container(content: @Composable () -> Unit) {
3     // ...
4     // content()
5     // ...
6 }
```

---

To do it, the IR of the lambda expression is tweaked so it calls a **composable factory function** with some specific parameters first: `composableLambda(...)`.

The first parameter added will be the current `$composer`, so it is forwarded as expected. `composableLambda($composer, ...)`.

Then it will add the `key` parameter, obtained from a combination of the `hashcode` from the fully qualified name of the Composable lambda, and the **expression start offset**, which is essentially where it is located in the file, to make sure the key is unique –positional memoization–. `composableLambda($composer, $key, ...)`.

After the key, a boolean parameter `shouldBeTracked` is added. This parameter determines whether this Composable lambda call needs to be tracked or not. When lambdas have no captures, Kotlin turns them into singleton instances, because they will never change. That also means they do not need to be tracked by Compose. `composableLambda($composer, $key, $shouldBeTracked, ...)`.

An optional parameter about the arity of the expression can also be added, only needed when it has more than 22 parameters (magic number). `composableLambda($composer, $key, $shouldBeTracked, $arity, ...)`.

Finally, it adds the lambda expression itself as the final parameter of the wrapper (the block, as a trailing lambda). `composableLambda($composer, $key, $shouldBeTracked, $arity, expression)`.

The purpose of the Composable factory function is straightforward: **Adding a replaceable group to the composition to store the lambda expression using the generated key.** This where Compose teaches the runtime how to store and retrieve the Composable expression.

On top of this wrapping, Compose can also optimize Composable lambdas that do not capture values, the same way Kotlin does: by representing those using singletons. For this, it generates a synthetic “ComposableSingletons” `internal object` per file where Composable lambdas were found. This object will retain (memoize) static references to those Composable lambdas and also include getters for those so they can be retrieved later.

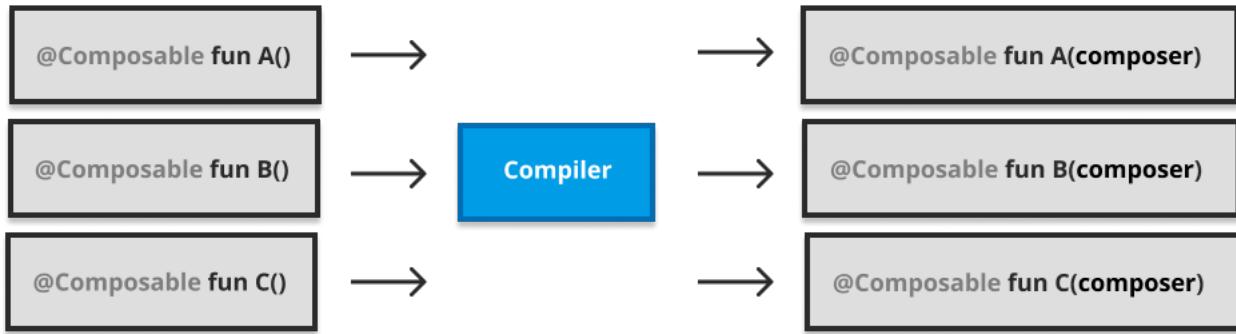
Composable lambdas have a final optimization provided by the way they are implemented: Similarly to `MutableState`. We can think of a Composable lambda like `@Composable (A, B) -> C` as being equivalently implemented as `State<@Composable (A, B) -> C`. Callsites where the lambda is invoked (`lambda(a, b)`) can then be replaced with the equivalent `lambda.value.invoke(a, b)`.

This is an optimization. It creates a snapshotted state object for all Composable lambdas, which allows Compose to more intelligently recompose sub-hierarchies based on lambdas changing. This was originally called “donut-hole skipping”, because it allows for a lambda to be updated “high” in the tree, and for compose to only need to recompose at the very “low” portion of the tree where this value is actually read. This ends up being a good tradeoff for lambdas in particular since their natural usage results in the instance being passed around a lot and often into “lower” portions of the hierarchy without ever actually “reading” their value (invoking them).

## Injecting the Composer

This is the step where the Compose compiler replaces all Composable functions by new versions of them with an extra Composer synthetic parameter added. This parameter is also forwarded to every Composable call in code to ensure it is always available at any point of the tree. That also includes calls to Composable lambdas.

This also requires some type remapping work, since the function type varies when the compiler plugin adds extra parameters to it.



### Composer injection

This effectively makes the Composer available for any subtree, providing all the information required to materialize the Composable tree and keep it updated.

Here is an example of it.

#### NamePlate.kt

---

```

1 fun NamePlate(name: String, lastname: String, $composer: Composer) {
2     $composer.start(123)
3     Column(modifier = Modifier.padding(16.dp), $composer) {
4         Text(
5             text = name,
6             $composer
7         )
8         Text(
9             text = lastname,
10            style = MaterialTheme.typography.subtitle1,
11            $composer
12        )
13    }
14    $composer.end()
15 }
```

---

Inline lambdas that are not Composable are intentionally not transformed, since they'll disappear at compile time when they're inlined on their callers. Also, expect functions are not transformed either. Those functions are resolved to the actual ones on type resolution, meaning it's the latter the ones that would be transformed in any case.

## Comparison propagation

We have learned about how the compiler injects the `$composer` extra parameter and forwards it to all composable calls. There is some extra pieces of metadata that are also generated and added to every Composable. One of them is the `$changed` parameter. This parameter is used to bring clues about whether the input parameters of the current Composable might have changed since the previous composition. This allows to skip recompositions.

SyntheticChangedParam1.kt

---

```
1 @Composable
2 fun Header(text: String, $composer: Composer<*>, $changed: Int)
```

---

This parameter is synthesized as a combination of bits that represent this condition for **each one** of the function input parameters –There’s a single `$changed` param that encodes this condition for every  $n$  input params (10 or so), which is limited by the amount of bits used. If the composable happens to have more params, 2 or more flags are added–. The reason for using bits is that processors are very good at bit processing by design.

Carrying this information allows certain optimizations for the runtime:

- It can skip `equals` comparisons to check whether an input parameter has changed from its latest stored value –from previous compositions–. This happens for cases where the input parameter is known to be static. The `$changed` bitmask provides this information. Let’s say this parameter is a `String` literal like in the snippet above, a constant, or something similar. The bits on this flag will tell the runtime that the value is **known at compile time**, hence it will never change at runtime, and therefore the runtime can **avoid comparing it ever**.
- There are also cases where the parameter will have **always** either not changed since the last composition, or if changed, its comparison is guaranteed to have been done already by a parent Composable in the tree. That means there is no need to recompute it. In this case, the state of the parameter is considered “certain”.
- For any other cases, the state is considered “uncertain”, so the runtime can just go ahead, compare it –using `equals`–, and store it in the slot table, so that we can always find the latest result later on. The bit value for this case is `0`, which is the default case. When `0` is passed for the `$changed` parameter we are telling the runtime to do all the work (not take any shortcuts).

Here is an example of how a Composable function body looks after injecting the `$changed` param and the required logic to handle it:

### SyntheticChangedParam2.kt

---

```

1  @Composable
2  fun Header(text: String, $composer: Composer<*>, $changed: Int) {
3      var $dirty = $changed
4      if ($changed and 0b0110 === 0) {
5          $dirty = $dirty or if ($composer.changed(text)) 0b0010 else 0b0100
6      }
7      if ($dirty and 0b1011 xor 0b1010 !== 0 || !$composer.skipping) {
8          f(text) // executes body
9      } else {
10         $composer.skipToGroupEnd()
11     }
12 }
```

---

There is some bit dance in there, but trying to stay agnostic of low level details, we can see how a local variable `$dirty` is used. This variable stores whether the param changed or not, and that is determined by both the `$changed` param bitmask and in case it is needed, the value previously stored in the slot table. If the value is considered “dirty” (has changed), the function body gets called (recomposed). Otherwise the Composable will skip recomposition.

Given recomposition can happen lots of times, carrying information about how the input state evolves can potentially save quite a bit of computation time and also lots of space. Often times parameters are passed through many composable functions and Compose does not want to store and compare them each time, as each time it does, it will take up slot-table space.

The same way that our Composable gets the `$changed` parameter passed by the caller, this Composable also has the responsibility to forward any information it has about any parameters passed down the tree. This is called “comparison propagation”. This is information we have available in the body –during composition–, so if we already know that an input has changed, is static, or whatever, we can forward that information to the `$changed` parameter of any child Composable that happens to reuse that parameter.

The `changed` parameter also encodes information about whether or not the argument passed into the function is known to be stable or unstable. this allows for a function accepting a broader type (say `List<T>`) to skip if the parameter is stable based on the input argument being something inferred as such (like say the expression `listOf(1, 2)`).

If you want to go more in depth about this, there are [some nice videos<sup>a</sup>](#) explaining the foundations of this plus stability inference by Leland Richardson that you could want to watch.

<sup>a</sup>[https://www.youtube.com/watch?v=bg0R9-AUXQM&ab\\_channel=LelandRichardson](https://www.youtube.com/watch?v=bg0R9-AUXQM&ab_channel=LelandRichardson)

## Default parameters

Another extra piece of metadata that is added to each Composable function at compile time is/are the `$default` parameter/s.

The default argument support by Kotlin is not usable for arguments of Composable functions, since Composable functions have the need to execute the default expressions for their arguments inside the scope (generated group) of the function. To do this Compose provides an alternative implementation of the default argument resolution mechanism.

Compose represents default arguments using a `$default` bitmask parameter that maps each parameter index to a bit on the mask. Kind of like what its done for the `$changed` parameter/s. There is also one `$default` param every `n` input parameters with default values. This bitmask provides information about whether the parameters have a value provided at the call site or not, to determine if the default expression must be used.

I've extracted this example from the library docs that shows very clearly how a Composable function looks before and after the `$default` bitmask is injected, plus the code to read it and use the default parameter value if required.

### DefaultParam.kt

---

```

1 // Before compiler (sources)
2 @Composable fun A(x: Int = 0) {
3     f(x)
4 }
5
6 // After compiler
7 @Composable fun A(x: Int, $changed: Int, $default: Int) {
8     // ...
9     val x = if ($default and 0b1 != 0) 0 else x
10    f(x)
11    // ...
12 }
```

---

Once again there is some bit dance, but the comparison simply checks the `$default` bitmask to default to 0 or keep the value passed for `x`.

## Control flow group generation

The Compose compiler also inserts a **group** on the body of each Composable function. There are different types of groups that can be generated depending on the control flow structures found within the body:

- Replaceable groups.
- Movable groups.
- Restartable groups.

Composable functions end up emitting groups at runtime, and those groups wrap all the relevant information about the current state of the Composable call. This allows the Composition to know how to cleanup any written data when the group needs to be replaced (replaceable groups), move the data around by preserving the identity of the Composable all the time (movable groups), or restart the function during recomposition (restartable groups).

At the end of the day, the runtime needs to know how to deal with control-flow based on the information that the Composition has stored in memory.

Groups also carry information about the position of the call in the sources. They wrap a span of text in the sources and have a key generated using the position of the call as one of its factors. That allows to store the group, and unlocks positional memoization.

## Replaceable groups

A few sections ago we explained that the body of a Composable lambda is automatically wrapped by inserting a call to a Composable function factory that gets passed information like the `$composer`, the generated `$key`, and the actual Composable lambda expression, among other things.

This is how that factory function looks in code:

ReplaceableGroup.kt

---

```

1 fun composableLambda(
2     composer: Composer,
3     key: Int,
4     tracked: Boolean,
5     block: Any
6 ): ComposableLambda {
7     composer.startReplaceableGroup(key)
8     val slot = composer.rememberedValue()
9     val result = if (slot === Composer.Empty) {
10         val value = ComposableLambdaImpl(key, tracked)
11         composer.updateRememberedValue(value)
12         value
13     } else {
14         slot as ComposableLambdaImpl
15     }
16     result.update(block)
17     composer.endReplaceableGroup()
18     return result
19 }
```

---

This factory function is called for Composable lambdas, like the ones used for the content of our Composable functions. If we look at it carefully, we'll notice that it starts a replaceable group with the key first, and closes the group at the end, wrapping all the text span in the middle. In between the start and end calls, it updates the composition with the relevant information. For this specific case, that is the lambda expression we are wrapping.

That is for Composable lambdas, but it happens the same way for other Composable calls. Here is an example of how the code for an average Composable function is transformed when it is flagged as non-restartable:

#### ReplaceableGroup2.kt

---

```

1 // Before compiler (sources)
2 @NonRestartableComposable
3 @Composable
4 fun Foo(x: Int) {
5     Wat()
6
7 // After compiler
8 @NonRestartableComposable
9 @Composable
10 fun Foo(x: Int, %composer: Composer?, %changed: Int) {
11     %composer.startReplaceableGroup(<>)
12     Wat(%composer, 0)
13     %composer.endReplaceableGroup()
14 }
```

---

The Composable call will also emit a replaceable group that will be stored in the Composition.

Groups are like a tree. Each group can contain any number of children groups. If that call to `Wat` is also a Composable, the compiler will also insert a group for it.

Some section ago we used the following example to showcase how identity can be preserved also based in position of a Composable call, so the runtime can understand these two calls to `Text` as different:

#### ConditionalTexts.kt

---

```

1 if (condition) {
2     Text("Hello")
3 } else {
4     Text("World")
5 }
```

---

A Composable function that does some conditional logic like this will also emit a replaceable group, so it stores a group that can be replaced later on when the condition toggles.

## Movable groups

These are groups that can be reordered without losing identity. Those are only required for the body of key calls at this point. Recapping a bit on an example we used in previous chapter:

### MovableGroup.kt

---

```

1 @Composable
2 fun TalksScreen(talks: List<Talk>) {
3     Column {
4         for (talk in talks) {
5             key(talk.id) { // Unique key
6                 Talk(talk)
7             }
8         }
9     }
10 }
```

---

Wrapping our `Talk` into the `key` composable ensures it is given a unique identity without exception. When we wrap Composables with `key`, a movable group is generated. That will help to reorder any of the calls without risk of losing the items' identity.

Here is an example of how a Composable using `key` is transformed:

### MovableGroup2.kt

---

```

1 // Before compiler (sources)
2 @Composable
3 fun Test(value: Int) {
4     key(value) {
5         Wrapper {
6             Leaf("Value ${'$'}value")
7         }
8     }
9 }
10
11 // After
12 @Composable
13 fun Test(value: Int, %composer: Composer?, %changed: Int) {
14     // ...
15     %composer.startMovableGroup(<>, value)
16     Wrapper(composableLambda(%composer, <>, true) { %composer: Composer?, %changed: In\
17     t ->
18         Leaf("Value %value", %composer, 0)
19     }, %composer, 0b0110)
```

```

20    %composer.endMovableGroup()
21    // ...
22 }
```

---

## Restartable groups

Restartable groups are the most interesting ones probably. Those are inserted only for restartable Composable functions. They also wrap the corresponding Composable call, but on top of it, they expand the end call a bit so it returns a nullable value. This value will be null only when the body of the Composable call doesn't read any states that might vary, hence **recomposition will never be required**. In that sense, there is no need to teach the runtime how to recompose this Composable. Otherwise, if it returns a non-null value, the compiler generates a lambda that teaches the runtime how to “restart” (re-execute) the Composable and therefore update the Composition.

This is how it looks in code:

RestartableGroup.kt

```

1 // Before compiler (sources)
2 @Composable fun A(x: Int) {
3     f(x)
4 }
5
6 // After compiler
7 @Composable
8 fun A(x: Int, $composer: Composer<*>, $changed: Int) {
9     $composer.startRestartGroup()
10    // ...
11    f(x)
12    $composer.endRestartGroup()?.updateScope { next ->
13        A(x, next, $changed or 0b1)
14    }
15 }
```

---

See how the update scope for recomposition contains a new call to the same Composable.

This is the type of group generated for all Composable functions that read from a state.

Before wrapping this section I want to add some extra reasoning applied by the Compiler to executable blocks found in order to generate the different types of groups. This was extracted from the official docs:

- If a block executes exactly 1 time always, no groups are needed.

- If a set of blocks are such that exactly one of them is executed at once (for example, the result blocks of an if statement, or a when clause), then we insert a replaceable group around each block. We have conditional logic.
- A movable group is only needed for the content lambda of key calls.

## Klib and decoy generation

There is specific support for .klib (multiplatform) and Kotlin/JS added to the Compose compiler. It was needed due to how the IR for dependencies is deserialized in JS, since it will not be able to match the type signatures once the IR is transformed –remember Compose adds extra synthetic parameters to Composable function declarations and Composable function calls–.

For this matter, Compose avoids replacing the IR for functions in Kotlin/JS –as it does for JVM– and creates copies instead. It will keep the original function declaration around to have a connection between each function in the Kotlin metadata and its IR, and all references in code can still resolve just fine, and then the IR for the copy will be tweaked by Compose as required. To differentiate both at runtime a `$composable` suffix will be added to the name.

### KlibAndDecoys.kt

---

```

1 // Before compiler (sources)
2 @Composable
3 fun Counter() {
4     ...
5 }
6
7 // Transformed
8 @Decoy(...)
9 fun Counter() { // signature is kept the same
10     illegalDecoyCallException("Counter")
11 }
12
13 @DecoyImplementation(...)
14 fun Counter$composable( // signature is changed
15     $composer: Composer,
16     $changed: Int
17 ) {
18     ...transformed code...
19 }
```

---

If you want to dig more into this support for .klib and Kotlin/JS I recommend reading [this awesome post by Andrei Shikov<sup>5</sup>](#).

---

<sup>5</sup><https://dev.to/shikasd/kotlin-compiler-plugins-and-binaries-on-multiplatform-139e>

# 3. The Compose runtime

Not long ago I [tweeted a summary about how the Compose architecture works internally<sup>6</sup>](#), covering the communication between the UI, the compiler, and the runtime.

The screenshot shows a tweet from Jorge Castillo (@JorgeCastilloPr). The tweet contains two paragraphs of text and a small icon. The first paragraph discusses Composables emitting changes to the runtime's slot table via the Composer. The second paragraph explains that this representation is interpreted later to "materialize" UI, accompanied by a small icon of a spool of thread.

Jorge Castillo  
@JorgeCastilloPr

When thinking about Compose it's good to notice that Composables don't yield actual UI, but "emit" changes to the in-memory structure managed by the runtime (slot table) via the Composer.

That representation has to be interpreted later on to "materialize" UI from it

Compose architecture tweet

This Twitter thread can work great as an intro for this chapter, since it gave a birdeye perspective over some of the most important points we need to understand. This particular chapter is focused on the Jetpack Compose runtime, but it will reinforce our mental mapping about how the different parts of Compose communicate and work together. If you feel curious, I'd recommend reading the Twitter thread before moving on.

This thread explained how Composable functions **emit** changes to the Composition, so the Composition can be updated with all the relevant information, and how that takes place via the current injected `$composer` instance thanks to the compiler (see chapter 2). The call to obtain the current Composer instance and the Composition itself are part of the Jetpack Compose runtime.

The thread intentionally stayed a bit over the surface, since Twitter is probably not be the best place to dive deep into a topic like this one. This book is a very good chance to do it.

So far we have referenced the state maintained in memory by the runtime as “the Composition”. That is an intentionally superficial concept. Let’s start by learning about the data structures used to store and update the Composition state.

---

<sup>6</sup><https://twitter.com/JorgeCastilloPr/status/1390928660862017539>

## The slot table and the list of changes

I've spotted some confusion floating around about the difference between these two data structures, probably due to the current lack of literature about Compose internals. As of today, I consider it necessary to clarify this first.

The slot table is an optimized in-memory structure that the runtime uses to store the **current state of the Composition**. It is filled with data during initial composition, and gets updated with every recomposition. We can think of it as a trace of all the Composable function calls, including their location in sources, parameters, remembered values, `CompositionLocals`, and more. Everything that has happened during composition is there. All this information is used later by the Composer to produce the next list of changes, since any changes to the tree will always depend on the current state of the Composition.

While the slot table records the state, the change list is what makes the actual changes to the node tree. It can be understood as a patch file that once applied, it updates the tree. All the changes to make need to be recorded, and then applied. Applying the changes on the list is a responsibility of the `Applier`, which is an abstraction that the runtime relies on to ultimately materialize the tree. We will get back to all this in deep detail later.

Finally, the Recomposer coordinates all this, deciding when and on what thread to recompose, and when and on what thread to apply the changes. Also more on this later.

## The slot table in depth

Let's learn how the state of the Composition is stored. The slot table is a data structure optimized for rapid linear access. It is based on the idea of a "gap buffer", very common in text editors. It stores the data in two linear arrays for that reason. One of these arrays keeps the information about the **groups** available in the Composition, the other one stores the **slots** that belong to each group.

`LinearStructures.kt`

---

```

1 var groups = IntArray(0)
2     private set
3
4 var slots = Array<Any?>(0) { null }
5     private set

```

---

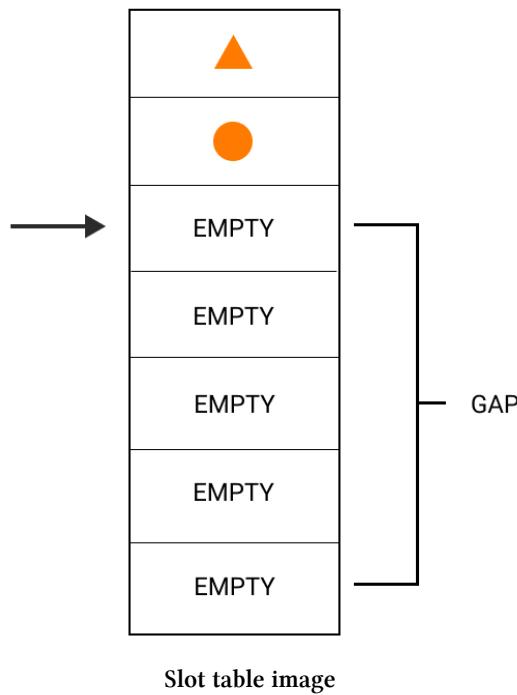
In chapter 2 we learned how the compiler wraps Composable functions bodies to make them emit groups instead. Those groups will give identity to the Composable once it is stored in memory (unique key), so that it can be identified later. Groups wrap all the relevant information for the Composable call and its children, and provide information about how to treat the Composable (as a group). Groups can have a different type depending on the control flow patterns found inside the

Composable body: Restartable groups, moveable groups, replaceable groups, reusable groups...

The groups array uses Int values since it will only store “group fields”, which represent meta-data for the groups. Parent and child groups are stored in the form of group fields. Given it’s a linear data structure, the group fields of a parent group will always come first, and the group fields of all its children will follow. This is a linear way to model a group tree, and it favors a linear scan of the children. Random access is expensive unless it is done through a group anchor. Anchors are like pointers that exist for this reason.

In the other hand, the slots array stores the relevant data for each one of those groups. It stores values of any type (Any?), since it is meant to store any type of information. This is where the actual Composition data is stored. Each group stored in the groups array describes how to find and interpret its slots in the slots array, since a group is always linked to a range of slots.

The slot table relies on a gap to read and write. Think of it as a range of positions from the table. This gap moves around and determines where the data is read from and written to the arrays when the time comes. The gap has a pointer to indicate where to start writing, and can shift its start and end positions, so the data in the table can also be overwritten.



Slot table image

Imagine some conditional logic like this one:

**ConditionalNonRestartable.kt**


---

```

1 @Composable
2 @NonRestartableComposable
3 fun ConditionalText() {
4     if (a) {
5         Text(a)
6     } else {
7         Text(b)
8     }
9 }
```

---

Given this Composable is flagged as **non-restartable**, a replaceable group will be inserted (instead of a restartable one). The group will store data in the table for the current “active” child. That will be `Text(a)`, in case `a` is true. When the condition toggles, the gap will move back to the start position of the group, and it will start writing from there, overriding all those slots with the data for `Text(b)`.

To read from and write to the table, we have `SlotReader` and `SlotWriter`. The slot table can have multiple active readers but a **single active writer**. After each read or write operation, the corresponding reader or writer gets closed. Any number of readers can be open, but the table can only be read **while it’s not being written**, for safety. The `SlotTable` remains invalid until the active writer is closed, since it will be modifying groups and slots directly and that could lead to race conditions if we try to read from it at the same time.

A reader works as a **visitor**. It keeps track of the current group being read from the groups array, its beginning and end positions, its parent (stored right before), the current slot from the group being read, the amount of slots the group has... etc. The reader can reposition, skip groups, read the value from the current slot, read values from specific indexes, and other things of the like. In other words, it is used to read information about the groups and their slots from the arrays.

The writer, in the other hand, is used for writing groups and slots to the arrays. As explained above, it can write data of any type –`Any`?– to the table. The `SlotWriter` relies on the **gaps** mentioned above for groups and slots, so it uses them to determine where to write (positions) within the arrays.

Think of a gap as a **slidable and resizable span of positions** for a linear array. The writer keeps track of the start and end positions, and length of each gap. It can move the gap around by updating its start and end positions.

The writer is able to add, replace, move and remove groups and slots. Think of adding a new Composable node to the tree, or Composables under conditional logics that might need to be replaced when condition toggles, for instance.

The writer can skip groups and slots, advance by a given amount of positions, seek to a position determined by an `Anchor`, and many other similar operations.

It keeps track of a list of `Anchors` pointing to specific indexes for rapid access through the table. Position of each group –also called group index– in the table is also tracked via an `Anchor`. The

Anchor is updated when groups are moved, replaced, inserted, or removed before the position the Anchor is pointing to.

The slot table also works as an iterator of composition groups, so it can provide information about them to the tools so those are able to inspect and present details of the Composition.

Now it's about time to learn about the change list.

For more details about the slot table, I recommend reading [this post by Leland Richardson<sup>a</sup>](#) from the Jetpack Compose team.

<sup>a</sup><https://medium.com/androiddevelopers/under-the-hood-of-jetpack-compose-part-2-of-2-37b2c20c6cd>

## The list of changes

We have learned about the slot table, how it allows the runtime to keep track of the current state of the Composition. Right, but what is the exact role of the list of changes then? when is it produced? what does it model? when are those changes applied, and for what reason?. We still have quite a few things to clarify. This section will be adding another piece to the puzzle. Let's try to put things in order.

Every time a composition (or recomposition) takes place, the Composable functions from our sources are executed and **emit**. “Emitting”, we have used that word many times already. Emitting means creating **deferred changes** to update the slot table, and ultimately also the materialized tree. Those changes are stored as a list. Generating this fresh list of changes is based on what is already stored in the slot table. Remember: Any changes to the tree must depend on the current state of the Composition.

An example of this can be moving a node. Imagine reordering the Composables of a list. We need to check where that node was placed before in the table, remove all those slots, and write them again but starting from a new position.

In other words, every time a Composable emits it is looking at the slot table, creating a deferred change according to the needs and the current information available, and adding it to a list with all the changes. Later, when Composition is finished, it will be time for materialization, and those **recorded changes** will effectively execute. That is when they effectively update the slot table with the most fresh available information of the Composition. This process is what makes the emitting process very fast: It simply creates a deferred action that will be waiting to be run.

Following this, we can see how change list is what ultimately makes the changes to the table. Right after that, it will notify the Applier to update the materialized node tree.

As we said above, the Recomposer orchestrates this process and decides what thread to compose or recompose on, and what thread to use for applying the changes from the list. The latter will also be the default context used by `LaunchedEffect` to run effects.

And with this, we have a clearer view on how changes are recorded, deferred, and ultimately executed, and also how all the state is stored in the slot table. Now it is a good time to learn about the Composer.

## The Composer

The injected `$composer` is what connects the Composable functions we write to the Compose runtime.

## Feeding the Composer

Let's explore how nodes are added to the in memory representation of the tree. We can use the `Layout Composable` to drive the example. `Layout` is the plumbings of all the UI components provided by Compose UI. This is how it looks in code:

Layout.kt

---

```
1 @Suppress("ComposableLambdaParameterPosition")
2 @Composable inline fun Layout(
3     content: @Composable () -> Unit,
4     modifier: Modifier = Modifier,
5     measurePolicy: MeasurePolicy
6 ) {
7     val density = LocalDensity.current
8     val layoutDirection = LocalLayoutDirection.current
9     ReusableComposeNode<ComposeUiNode, Applier<Any>>(
10         factory = ComposeUiNode.Constructor,
11         update = {
12             set(measurePolicy, ComposeUiNode.SetMeasurePolicy)
13             set(density, ComposeUiNode.SetDensity)
14             set(layoutDirection, ComposeUiNode.SetLayoutDirection)
15         },
16         skippableUpdate = materializerOf(modifier),
17         content = content
18     )
19 }
```

---

Layout uses `ReusableComposeNode` to emit a `LayoutNode` into the composition. But even if this might sound like creating and adding the node right away, what it really does is **teaching the runtime** how to create, initialize and insert the node at the current location in the Composition **when the time comes**. Here is the code:

#### ReusableComposeNode.kt

---

```

1  @Composable
2  inline fun <T, reified E : Applier<*>> ReusableComposeNode(
3      noinline factory: () -> T,
4      update: @DisallowComposableCalls Updater<T>.() -> Unit,
5      noinline skippableUpdate: @Composable SkippableUpdater<T>.() -> Unit,
6      content: @Composable () -> Unit
7  ) {
8      // ...
9      currentComposer.startReusableNode()
10     // ...
11     currentComposer.createNode(factory)
12     // ...
13     Updater<T>(currentComposer).update() // initialization
14     // ...
15     currentComposer.startReplaceableGroup(0x7ab4aae9)
16     content()
17     currentComposer.endReplaceableGroup()
18     currentComposer.endNode()
19 }
```

---

I'm omitting some not (yet) relevant parts, but note how it delegates everything to the `currentComposer` instance. We can also see how it uses the chance to start a replaceable group to wrap the content of this Composable when storing it. Any children emitted within the `content` lambda will effectively be stored as children of this group (and therefore also the Composable) in the Composition.

The same emitting operation is done for any other Composable functions. See `remember` for instance:

#### Composables.kt

---

```

1  @Composable
2  inline fun <T> remember(calculation: @DisallowComposableCalls () -> T): T =
3      currentComposer.cache(invalid = false, calculation)
```

---

The `remember` Composable function uses the `currentComposer` to cache (`remember`) the value returned by the provided lambda into the composition. The `invalid` parameter forces an update for the value regardless of it being previously stored. The `cache` function is coded like this:

**Composer.kt**


---

```

1 @ComposeCompilerApi
2 inline fun <T> Composer.cache(invalid: Boolean, block: () -> T): T {
3     return rememberedValue().let {
4         if (invalid || it === Composer.Empty) {
5             val value = block()
6             updateRememberedValue(value)
7             value
8         } else it
9     } as T
10 }
```

---

First, it searches for the value in the Composition (slot table). If it is not found, it will emit changes to schedule an update for the value (or in other words, record). Otherwise, it will return the value as is.

## Modeling the Changes

As explained in the previous section, all the emitting operations delegated to the currentComposer are internally modeled as Changes that are added to a list. A Change is a deferred function with access to the current Applier and SlotWriter (remember there is a single active writer at a time). Let's have a look at it in code:

**Composer.kt**


---

```

1 internal typealias Change = (
2     applier: Applier<*>,
3     slots: SlotWriter,
4     rememberManager: RememberManager
5 ) -> Unit
```

---

These changes are added to the list (recorded). The action of “emitting” essentially means creating these Changes, which are deferred lambdas to potentially add, remove, replace, or move nodes from the slot table, and consequently notify the Applier (so those changes can be materialized).

For this reason, whenever we talk about “emitting changes” we might also use the words “recording changes” or “scheduling changes”. It's all referring to the same thing.

After composition, once all the Composable function calls complete and all the changes are recorded, **all of them will be applied in a batch by the applier**.

The composition itself is modeled with the `Composition` class. We are keeping that aside for now, since we will learn about the composition process in detail in the sections to come later in this chapter. Let's learn a few more details about the Composer first.

## Optimizing when to write

As we have learned above, inserting new nodes is delegated to the Composer. That means it always knows when it is already immersed in the process of inserting new nodes into the composition. When that is the case, the Composer can shortcut the process and start writing to the slot table right away when changes are emitted, instead of recording them (adding them to the list to be interpreted later). In other case, those changes are recorded and deferred, since it's not the time to make them yet.

## Writing and reading groups

Once the Composition is done, `composition.applyChanges()` is finally called to materialize the tree, and changes are written to the slot table. The Composer can write different types of information: data, nodes, or groups. That said, all of them are ultimately stored in the form of groups for simplicity. They just happen to have different group fields for differentiation.

The Composer can “start” and “end” any group. That has different meanings depending on the actions being taken. If it is writing, it will stand for “group created” and “group removed” from the slot table. If it is reading, the `SlotReader` will be asked to move its read pointers in and out of the group to start or end reading from it.

Nodes on the Composable tree (ultimately groups in the table) are not only inserted, but can also be removed, or moved. Removing a group means removing it and all its corresponding slots from the table. To do this, the Composer asks to reposition the `SlotReader` accordingly and make it skip the group (since it's not there anymore), and record operations to remove all its nodes from the applier. Any modification actions need to be scheduled (recorded) and applied later as a batch as explained above, mostly to ensure they all make sense together. The Composer will also discard any pending invalidations for the removed group, since they will not happen ever.

Not all groups are restartable, replaceable, movable, or reusable. Among other things that are also stored as groups, we can find the defaults wrapper block. This block surrounds remembered values for Composable calls necessary to produce default parameters: e.g: `model: Model = remember { DefaultModel() }`. This is also stored as a very specific group.

When the Composer wants to start a group, the following things happen:

- If the composer is in the process of inserting values, it will go ahead and write to the slot table while it is at it, since there is no reason to wait.
- In other case, if there are pending operations, it will record those changes to the applied when applying changes. Here, the Composer will try to reuse the group in case it already exists (in the table).
- When the group is already stored **but in a different position** (it has been moved), an operation to move all the slots for the group is recorded.
- In case the group is new (not found in the table), it will move into inserting mode, which will write the group and all its children to an intermediate insertTable (another SlotTable) until the group is complete. That will schedule the groups to be inserted into the final table.
- Finally, if the Composer is not inserting and there are no pending write operations, it will try to start reading the group.

Reusing groups is common. Sometimes it is not needed to create a new node, but we can reuse it in case it already exists. (See `ReusableComposeNode` above). That will emit (record) the operation to navigate to the node by the `Applier`, but will skip the operations to create and initialize it.

When a property of a node needs an update, that action is also recorded as a `Change`.

## Remembering values

We learned how the Composer has the ability to remember values into the Composition (write them to the slot table), and it can also update those values later on. The comparison to check if it changed from last composition is done right when `remember` is called, but the update action is recorded as a `Change` unless the Composer is already inserting.

When the value to update is a `RememberObserver`, then the Composer will also record an implicit `Change` to track the remembering action in the Composition. That will be needed later when all those remembered values need to be forgotten.

## Recompose scopes

Something else that also happens via the Composer are the recompose scopes, which enable smart recomposition. Those are directly linked to restart groups. Every time a restart group is created, the Composer creates a `RecomposeScope` for it, and sets it as the `currentRecomposeScope` for the Composition.

A RecomposeScope models a region of the Composition that can be recomposed independently of the rest of the Composition. It can be used to manually invalidate and trigger recomposition of a Composable. An invalidation is requested via the composer, like: `composer.currentRecomposeScope().invalidate()`. For recomposing, the Composer will position the slot table to the starting location of this group, and then call the recompose block passed to the lambda. That will effectively invoke the Composable function again, which will emit one more time, and therefore ask the Composer to override its existing data in the table.

The composer maintains a Stack of all the recompose scopes that have been invalidated. Meaning they are pending to be recomposed, or in other words, need to be triggered in next recomposition. The `currentRecomposeScope` is actually obtained by peeking into this Stack.

That said, RecomposeScopes are not always enabled. That only happens when Compose finds read operations from State snapshots within the Composable. In that case, the Composer marks the RecomposeScope as used, which makes the inserted “end” call at the end of the Composable **not return null** anymore, and therefore activate the recomposition lambda that follows (see below, after the ? character).

#### RecomposeScope.kt

---

```

1 // After compiler inserts boilerplate
2 @Composable
3 fun A(x: Int, $composer: Composer<*>, $changed: Int) {
4     $composer.startRestartGroup()
5     // ...
6     f(x)
7     $composer.endRestartGroup()?.updateScope { next ->
8         A(x, next, $changed or 0b1)
9     }
10 }
```

---

The Composer can recompose all invalidated child groups of the current parent group when recomposition is required, or simply make the reader skip the group to the end when it is not. (see: comparison propagation section in chapter 2).

## SideEffects in the Composer

The Composer is also able to record SideEffects. A SideEffect always runs **after composition**. They are recorded as a function to call when changes to the corresponding tree are **already applied**. They represent effects that happen on the side, so this type of effect is completely agnostic of the Composable lifecycle. We’ll not get things done like automatic cancellation when leaving the Composition, neither retrying effects on recomposition. That is because this type of effect is **not stored in the slot table**, and therefore simply discarded if composition fails. We’ll learn about this and its purpose in the chapter about effect handlers. Still it is interesting to notice how they are recorded via the Composer.

## Storing CompositionLocals

The Composer also provides means to register `CompositionLocals` and obtain its values given a key. `CompositionLocal.current` calls rely on it. A Provider and its values are also stored as a group in the slot table, all together.

## Storing source information

The Composer also stores source information in the form of `CompositionData` gathered during Composition to be leveraged by Compose tools.

## Linking Compositions via `CompositionContext`

There is not a single Composition but a tree of compositions and subcompositions. Subcompositions are Compositions created inline with the only intention to construct a separate composition in the context of the current one to support independent invalidation.

A Subcomposition is connected to its parent Composition with a parent `CompositionContext` reference. This context exists to link Composition and subcompositions together as a tree. It ensures that `CompositionLocals` and invalidations are transparently resolved / propagated down the tree as if they belonged to a single Composition. `CompositionContext` itself is also written to the slot table as a group.

Creating Subcompositions is usually done via `rememberCompositionContext`:

`Composables.kt`

---

```
1 @Composable fun rememberCompositionContext(): CompositionContext {
2     return currentComposer.buildContext()
3 }
```

---

This function remembers a new Composition at the current position in the slot table, or returns it in case it's already remembered. It is used to create a Subcomposition from places where separate Composition is required, like the `VectorPainter` (see `VectorPainter.kt` snippet earlier in this chapter), a Dialog, the `SubcomposeLayout`, a Popup, or the actual `AndroidView`, which is a wrapper to integrate Android Views into Composable trees.

## Accessing the current State snapshot

The Composer has a reference to the current snapshot, as in a snapshot of the values return by mutable states and other state objects for the current thread. All state object will have the same value in the snapshot as they had when the snapshot was created unless they are explicitly changed in the snapshot. this will be expanded in the chapter about state management.

## Navigating the nodes

Navigation of the node tree is performed by the applier, but not directly. It is done by recording all the locations of the nodes as they are traversed by the reader and recording them in a `downNodes` array. When the node navigation is realized all the downs in the `down nodes` is played to the applier. If an up is recorded before the corresponding down is realized then it is simply removed from the `downNodes` stack, as a shortcut.

## Keeping reader and writer in sync

This is a bit low level, but given groups can be inserted, deleted, or moved, the location of a group in the writer might differ than its location in the reader for a while (until changes are applied). That makes it needed to maintain a delta to track the difference. That delta is updated with inserts, deletes, and moves, and reflects the “unrealized distance the writer must move to match the current slot in the reader” per what the docs say.

## Applying the changes

As we have mentioned many times in this chapter, the `Applier` is in charge to do this. The current `Composer` delegates on this abstraction to apply all the recorded changes after the `Composition`. This is what we know as “materializing”. This process executes the list of `Changes` and, as a result, it updates the slot table and interprets the `Composition` data stored on it to effectively yield a result.

**The runtime is agnostic of how the `Applier` is implemented.** It relies on a public contract that client libraries are expected to implement. That is because the `Applier` is an integration point with the platform, so it will vary depending on the use case. This contract looks like this:

`Applier.kt`

---

```

1 interface Applier<N> {
2     val current: N
3     fun onBeginChanges() {}
4     fun onEndChanges() {}
5     fun down(node: N)
6     fun up()
7     fun insertTopDown(index: Int, instance: N)
8     fun insertBottomUp(index: Int, instance: N)
9     fun remove(index: Int, count: Int)
10    fun move(from: Int, to: Int, count: Int)
11    fun clear()
12 }
```

---

The first thing we see is the `N` type parameter in the contract declaration. That is the type for the nodes we are applying. This is why compose can work with generic call graphs or node trees. It is always agnostic of the type of nodes used. The `Applier` provides operations to traverse the tree, insert, remove, or move nodes around, but it doesn't care about the type of those nodes or how they are ultimately inserted. Spoiler: **That will be delegated to the nodes themselves.**

The contract also defines how to remove all children in a given range from the current node, or move children from the current node to change their positions. The `clear` operation defines how to point to the root and remove all nodes from the tree, preparing both the `Applier` and its root to be used as the target of a new composition in the future.

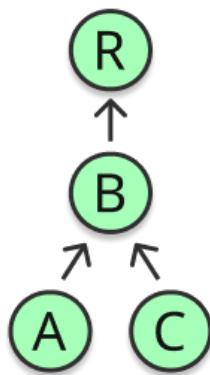
The `Applier` traverses the complete tree visiting and applying all nodes. The tree can be traversed from top to bottom, or from bottom to top. It always keeps a reference of the current node it is visiting and applying changes to. It has calls to begin and end applying changes that the `Composer` will call before and after, and it provides means to insert top-down, or bottom-up, and to navigate top-down (navigate to the child node of the current one), or bottom-up (navigate to the parent of the current node).

## Performance when building the node tree

There is an important difference between building the tree top-down or doing it bottom-up. I'll extract this specific example from the official docs, since it is already pretty meaningful.

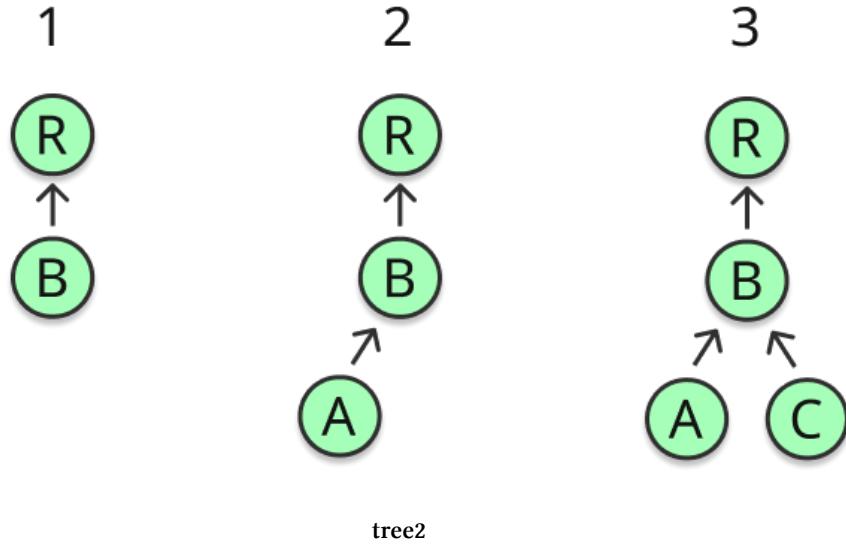
### Inserting top-down

Consider the following tree:



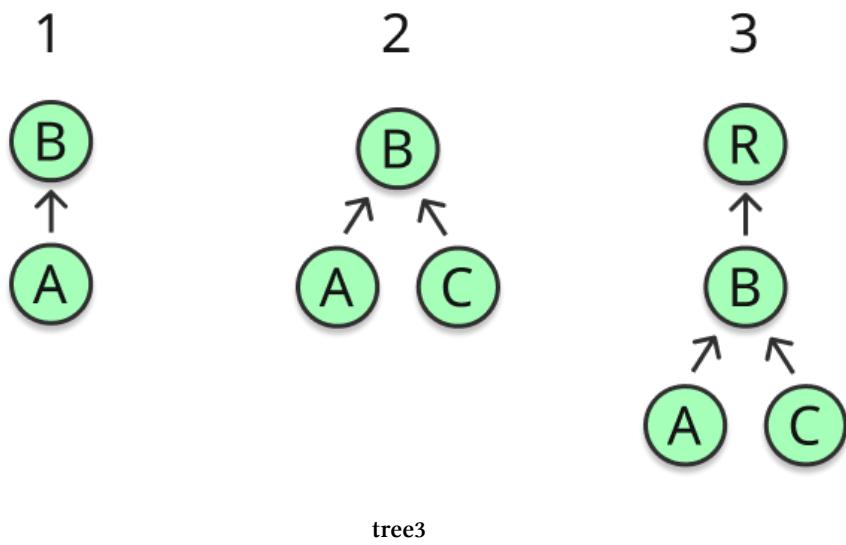
tree1

If we wanted to build this tree top-down, we would first insert `B` into `R`, then insert `A` into `B`, and finally insert `C` into `B`. I.e:



## Inserting bottom-up

A bottom-up building of the tree starts by inserting A and C into B, then inserting the B tree into R.



Performance for building a tree top-down versus bottom-up can vary considerably. That decision is up to the `Applier` implementation used, and it usually relies on the number of nodes that need to be notified every time a new child is inserted. Imagine that the graph we want to represent with Compose requires notifying all ancestors of a node whenever it is inserted. In top to bottom, each insertion could notify multiple nodes (parent, parent of its parent... etc). That count will grow exponentially with each new level inserted. If it was bottom up instead, you'd always only notify the direct parent, since the parent is still not attached to the tree. But this can be the other way around if

our strategy is notifying all children instead. So, always depends on the tree we are representing and how changes need to be notified top or down the tree. The only key point here is pick one strategy or the other for insertion, but never both.

## How changes are applied

As we have described above, client libraries provide implementations for the `Applier` interface, one example of this being the `UiApplier`, for Android UI. We can use that one as a perfect example on what “applying a node” means and how that yields components we can see on screen for this specific use case.

If we look at the implementation, it is very narrow:

`UiApplier.kt`

---

```
1 internal class UiApplier(
2     root: LayoutNode
3 ) : AbstractApplier<LayoutNode>(root) {
4
5     override fun insertTopDown(index: Int, instance: LayoutNode) {
6         // Ignored.
7     }
8
9     override fun insertBottomUp(index: Int, instance: LayoutNode) {
10        current.insertAt(index, instance)
11    }
12
13    override fun remove(index: Int, count: Int) {
14        current.removeAt(index, count)
15    }
16
17    override fun move(from: Int, to: Int, count: Int) {
18        current.move(from, to, count)
19    }
20
21    override fun onClear() {
22        root.removeAll()
23    }
24
25    override fun onEndChanges() {
26        super.onEndChanges()
27        (root.owner as? AndroidComposeView)?.clearInvalidObservations()
28    }
29 }
```

---

The first thing we see is that the generic type `N` has been fixed to be `LayoutNode`. That is the type of node that Compose UI has picked to represent the UI nodes that will be rendered.

Next thing we notice is how it extends `AbstractApplier`. That is a default implementation that stores the visited nodes in a `Stack`. Every time a new node is visited down the tree, it will add it to the stack, and every time the visitor moves up, it'll remove the last node visited from the top of the stack. This is usually common across appliciers, so it is likely a good idea to have it in a common parent class.

We also see how `insertTopDown` is ignored in the `UiApplier`, since insertions will be performed bottom up in the case of Android. As we said above, it is important to pick one strategy or the other, not both. In this case bottom-up will be more appropriate to avoid duplicate notifications when a new child is inserted. This difference in terms of performance was explained earlier.

Methods to insert, remove, or move a node are all **delegated to the node itself**. `LayoutNode` is how Compose UI models a UI node, hence it knows everything about the parent node and its children. Inserting a node means attaching it to its new parent in a given position (it can have multiple children). Moving it is essentially reordering the list of children for its parent. Finally, removing it simply means removing it from the list.

Whenever it is done applying the changes, it can call `onEndChanges()` that will delegate on the root node owner for a final required action. `-onBeginChanges()` is always assumed to be called before applying changes, so `onEndChanges()` needs to be called in the end. At this point, any pending invalid observations are cleared. These are snapshot observations meant to automatically re-invoke layout or draw when the values they read from and depend on have changed. Imagine nodes being added, inserted, replaced, or moved, and how that can affect things like measuring or layout.

## Attaching and drawing the nodes

Once we got here we can finally answer the real question: How inserting a node on the tree (attaching it to its parent) means we ultimately see it on screen? The answer is: **The node knows how to attach and draw itself**.

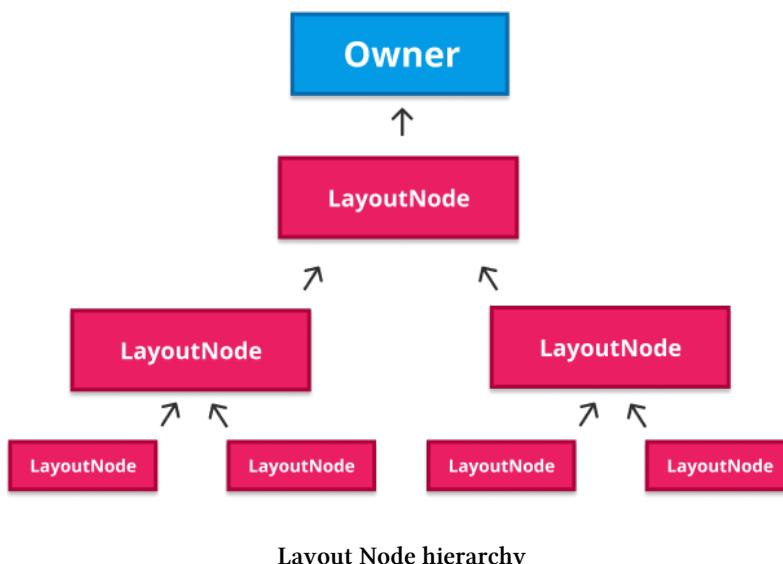
We will only summarize the process very briefly for now, since there is a deep dive of this in Chapter 4 (Compose UI). This will be enough for closing the circle and achieving a complete mental model at this point.

`LayoutNode` is the node type picked for this specific use case (Android UI). When the `UiApplier` implementation delegates the insertion to it, things happen in the following order:

- Check that the conditions for inserting the node are fulfilled –e.g: it doesn't have a parent already–.
- Invalidates the list of sorted children in the Z index. This is a parallel list that maintains all the children sorted by their Z index, so they can be drawn in order (lower Z index first). Invalidating the list makes it get recreated (and ordered) whenever it is needed again.

- Attach the new node to its parent and to its Owner. (Explained below).
- Invalidate!

The owner lives at the root of the tree, and implements the **connection of our Composable tree with the underlying View system**. We can think of it as a thin integration layer with Android. Actually, it is implemented by `AndroidComposeView` (a standard `View`). All layout, draw, input, and accessibility is hooked through the owner. A `LayoutNode` must be attached to an Owner in order to show up on screen, and its owner **must be the same than the owner of its parent**. The owner is also part of Compose UI. After we attach a node, we can call `invalidate` via the Owner, so the Composable tree can be rendered.



Finally, the ultimate integration point takes place when the `Owner` is set. That happens whenever we call `setContent` from an `Activity`, `Fragment`, or a `ComposeView`. At that point, an `AndroidComposeView` is created, attached to the `View` hierarchy, and set as the `Owner`, so it can perform invalidation on demand.

Profit! We finally know how Compose UI materializes a node tree for Android. We will expand much more on all this in the following chapter.

So we have closed the cycle, but maybe a bit too early. So far we have gathered tons of interesting details and we have a better picture of how things work around Composition, but what about the Composition process itself?

Let's go for it.

## Composition

We've learned lots of interesting details about the Composer in the previous section. We know how it records changes to write to or read from the slot table, how those changes are emitted when

Composable functions execute during the Composition, and how those recorded changes are applied in the end. But truth is we didn't give a word (yet) about who is in charge of creating a Composition, how, when does it take place, or what steps are involved. Composition is our missing piece so far.

We've said that Composer has a reference to the Composition, but that could make us think that the Composition is created and owned by the Composer, when it is actually the other way around. When a Composition is created, it builds a Composer by itself. The Composer becomes accessible via the currentComposer machinery, and it will be used to create and update the tree managed by the Composition.

The entry point to the Jetpack Compose runtime by client libraries is split in **two different parts**:

- Writing Composable functions: That will make them emit all the relevant information, and therefore connect our use case with the runtime.
- Composable functions are great but they'll never execute without a Composition process. That's why another entry point is required: setContent. This is the integration layer with the target platform, and a Composition is created and initiated here.

## Creating a Composition

For Android for example, that can be a ViewGroup.setContent call, which returns a new Composition:

Wrapper.android.kt

---

```
1 internal fun ViewGroup.setContent(
2     parent: CompositionContext,
3     content: @Composable () -> Unit
4 ): Composition {
5     // ...
6     val composeView = ...
7     return doSetContent(composeView, parent, content)
8 }
9
10 private fun doSetContent(
11     owner: AndroidComposeView,
12     parent: CompositionContext,
13     content: @Composable () -> Unit
14 ): Composition {
15     // ...
16     val original = Composition(UiApplier(owner.root), parent) // Here!
17     val wrapped = owner.view.getTag(R.id.wrapped_composition_tag)
18     as? WrappedComposition ?: WrappedComposition(owner, original).also {
19         owner.view.setTag(R.id.wrapped_composition_tag, it)
```

```

20     }
21     wrapped.setContent(content)
22     return wrapped
23 }
```

---

A `WrappedComposition` is a decorator that knows how to link a `Composition` to an `AndroidComposeView` so it connects it directly to the Android View system. It starts controlled effects to keep track of things like keyboard visibility changes or accessibility, and pipes information about the Android Context that will be exposed to the `Composition` as `CompositionLocals`. (i.e: the context itself, configuration, the current `LifecycleOwner`, the current `savedStateRegistryOwner`, or the owner's view, among others). This is how all those things become implicitly available for all our Composable functions.

Note how an instance of a `UiApplier` that starts pointing to the root `LayoutNode` of the tree is passed to the `Composition`. (The `Applier` is a visitor for nodes, so it starts pointing to the root one). This is the first time we explicitly see how it is the client library the one in charge to pick the implementation for the `Applier`.

We can also see how `composition.setContent(content)` is called in the end. `Composition#setContent` is what sets the content of the `Composition`. (Updates the `Composition` with all the information provided by `content`).

Another very good example of creating a `Composition` can be the `VectorPainter`, also part of Compose UI and used to paint vectors on screen. Vector painters create and maintain their own `Composition`:

#### VectorPainter.kt

```

1  @Composable
2  internal fun RenderVector(
3      name: String,
4      viewportWidth: Float,
5      viewportHeight: Float,
6      content: @Composable (viewportWidth: Float, viewportHeight: Float) -> Unit
7  ) {
8      // ...
9      val composition = composeVector(rememberCompositionContext(), content)
10
11     DisposableEffect(composition) {
12         onDispose {
13             composition.dispose() // composition needs to be disposed in the end!
14         }
15     }
16 }
17 }
```

```

18 private fun composeVector(
19     parent: CompositionContext,
20     composable: @Composable (viewportWidth: Float, viewportHeight: Float) -> Unit
21 ): Composition {
22     val existing = composition
23     val next = if (existing == null || existing.isDisposed) {
24         Composition(VectorApplier(vector.root), parent) // Here!
25     } else {
26         existing
27     }
28     composition = next
29     next.setContent {
30         composable(vector.viewportWidth, vector.viewportHeight)
31     }
32     return next
33 }
```

---

We will explore this further in an upcoming chapter about advanced Jetpack Compose use cases, but we can note here how a different `Applier` strategy is picked: a `VectorApplier` that starts pointing to the root node in the vector tree, which in this case will be a `VNode`.

Finally, another example of this that we could also find in Compose UI is the `SubcomposeLayout`, which is a `Layout` that maintains its own `Composition` so it is able to subcompose its content during the measuring phase. This can be useful when we need the measure of a parent for the composition of its children.

Regardless of the use case, whenever a `Composition` is created, a parent `CompositionContext` can be passed (see above). But note that it can be `null`. The parent context (if available) will be used to link the new composition logically to an existing one, so that invalidations and `CompositionLocals` can resolve across compositions as if they were the same one.

When creating a `Composition` it is also possible to pass a recompose context, which will be the `CoroutineContext` used by the `Applier` for applying the changes and ultimately materialize the tree. If not provided, it defaults to the one provided by the `Recomposer`, which is `EmptyCoroutineContext`. That means Android will likely recompose on `AndroidUiDispatcher.Main`.

The same way a `Composition` is created, it must be disposed –i.e: `composition.dispose()` when it is not needed anymore. That is when the UI (or alternative use cases) for it are disposed. We could say that a `Composition` is scoped to its owner. Sometimes disposal might be a bit hidden, like in the case of `ViewGroup.setContent` (behind a lifecycle observer), but it is always there.

## The initial Composition process

Whenever a new `Composition` is created, a call to `composition.setContent(content)` always follows (see the previous 2 snippets). That is in fact where the `Composition` is initially populated

(the slot table is filled up with relevant data).

This call is delegated to the parent Composition to trigger the initial Composition process (Remember how Compositions and Subcompositions are linked via a parent `CompositionContext`):

#### Composition.kt

---

```

1 override fun setContent(content: @Composable () -> Unit) {
2     // ...
3     this.composable = content
4     parent.composeInitial(this, composable) // `this` is the current Composition
5 }
```

---

For Subcompositions, the parent will be another Composition. For the root Composition, the parent will be the Recomposer. But regardless of that, logics for performing the initial Composition will always rely on the Recomposer in any case, since for Subcompositions, the `composeInitial` call delegates to the parent over and over until it reaches the root Composition.

So the call to `parent.composeInitial(composition, content)` can be translated to `recomposer.composeInitial(composition, content)`, and it does a few important things here to populate the initial Composition:

- Takes a **snapshot** of the current value of all the State objects. Those values will be isolated from potential changes from other snapshots. This snapshot is **mutable**, but at the same time it is concurrent safe. It can be modified safely without affecting any other existing State snapshots, since any changes to any of its State objects will happen only for it, and it will atomically sync all those changes with the global shared state in a later step.
- The State values from this mutable snapshot can only be modified from the block passed when calling `snapshot.enter(block: () -> T)`.
- When taking the snapshot, the Recomposer also passes observers for any reads or writes to the mentioned State objects, so the Composition can be notified accordingly when those take place. That allows the Composition to flag the affected recomposition scopes as used, which will make them recompose when the time comes.
- Enters the snapshot –I.e: `snapshot.enter(block)`– by passing the following block: `composition.composeContent(content)`. That is **where the Composition actually takes place**. The action of entering is what lets the Recomposer know that any State objects read or written during Composition will be tracked (notified to the Composition).
- The Composition process is delegated to the Composer. More on this step below this list.
- Once the Composition is done, any changes to State objects are made to the current State snapshot only, so its time to propagate those changes to the global state. That happens via `snapshot.apply()`.

That is the rough order of things around the initial Composition. Everything regarding the State snapshot system will be expanded with much more detail in the upcoming chapter about this topic.

Now, let's elaborate the actual Composition process itself, delegated to the Composer. This is how things happen in rough terms.

- Composition cannot be started if it's already running. In that case an exception is thrown and the new Composition is discarded. Reentrant Composition is not supported.
- If there are any pending invalidations, it will copy those to the invalidations list maintained by the Composer for the RecomposeScopes pending to invalidate.
- Moves the flag `isComposing` to be true since Composition is about to start.
- Calls `startRoot()` to start the Composition, that will start the root group for the Composition in the slot table and initialize other required fields and structures.
- Call `startGroup` to start a group for the content in the slot table.
- Invokes the content lambda so it emits all its changes.
- Calls `endGroup` to end the group in the slot table.
- Calls `endRoot()` to end the Composition.
- Moves the flag `isComposing` to be `false`, since Composition is done.
- Clears other structures maintaining temporary data.

## Applying changes after initial Composition

After the initial Composition, the `Applier` is notified to apply all the changes recorded during the process: `composition.applyChanges()`. This is done via the `Composition` also, which calls `applier.onBeginChanges()`, goes over the list of changes executing all of them and passing the required `Applier` and `SlotWriter` instances to each change. Finally, after all changes are applied, it calls `applier.onEndChanges()`. This is the natural process.

After this, dispatches all registered `RememberedObservers`, so any classes implementing the `RememberObserver` contract can be notified when entering or leaving the Composition. Things like `LaunchedEffect` or `DisposableEffect` implement it, so they can constrain the effect to the Composable lifecycle within the Composition.

Right after, all `SideEffects` are triggered in the same order they were recorded.

## Additional information about the Composition

A Composition is aware of its pending invalidations for recomposition. It also knows if it is currently composing. This knowledge can be used to apply invalidations instantly (when it is), or defer them otherwise. It can also be used by the Recomposer to discard recompositions when it is true.

The runtime relies on a variant of the Composition called `ControlledComposition` that adds a few extra functions so it can be controlled from the outside. That way, the Recomposer can orchestrate invalidations and further recomposition. Functions like `composeContent` or `recompose` are good examples of this. The Recomposer can trigger those actions in the composition when needed.

The Composition provides means to detect if a set of objects are being observed by itself so to enforce recomposition when those vary. For instance, this is used by the Recomposer to enforce recomposition in a child composition when a `CompositionLocal` varies in a parent composition. Remember Compositions are connected via parent `CompositionContext` for this matter.

Sometimes an error is found during composition, in that case it can be aborted, which is pretty much like resetting the Composer and all its references / stacks and everything.

The composer assumes it is skipping recomposition when it is not inserting nor reusing, there are no invalid providers (since that would require recomposition) and the `currentRecomposeScope` doesn't require recomposition. A chapter on smart recomposition is also coming up.

## The Recomposer

We already know how the initial Composition takes place, and also learned a few things about `RecomposeScopes` and invalidation. But we still know close to nothing regarding how the Recomposer actually works. How is it created and when does it start running? How does it start listening for invalidations to automatically trigger recomposition? Some questions likely arise.

The Recomposer controls the `ControlledComposition`, and it triggers recompositions when needed to ultimately apply updates to it. It also determines what thread to compose or recompose on, and what thread to use for applying the changes.

Let's learn how to create a Recomposer and make it start awaiting for invalidations.

## Spawning the Recomposer

The entry point to Jetpack Compose by client libraries is creating a Composition and calling `setContent` over it –see section above: [Creating a Composition](#)–. When creating the Composition it is required to provide a parent for it. Given the parent of a root Composition is a Recomposer, this is also the moment to create it.

This entry point is the connection between the platform and the Compose runtime, and it is code provided by the client. In the case of Android, that is Compose UI. This library creates a Composition (which internally creates its own Composer), and a Recomposer to use as its parent.

Note that each potential use case for each platform is prone to create its own Composition as we have learned before, and the same way, it will also likely create its own Recomposer.

When we want to use Compose on Android `ViewGroups`, we call `ViewGroup.setContent` which ultimately, and after some indirections, delegates creating the parent context to a Recomposer factory:

```
1 fun interface WindowRecomposerFactory {  
2  
3     fun createRecomposer(windowRootView: View): Recomposer  
4  
5     companion object {  
6         val LifecycleAware: WindowRecomposerFactory = WindowRecomposerFactory { rootView\  
7             ->  
8                 rootView.createLifecycleAwareViewTreeRecomposer()  
9             }  
10        }  
11    }
```

This factory creates a Recomposer for the current window. I find the creation process very interesting to explore, since it provides many clues about how Android resolves the integration with Compose.

Passing a reference to the root view is needed for calling `createRecomposer`, since the created Recomposer will be **lifecycle-aware**, meaning that it will be linked to the `ViewTreeLifecycleOwner` at the root of the View hierarchy. This will allow to cancel (shutdown) the Recomposer when the view tree is unattached, for instance, which is important to avoid leaking the recomposition process.  
–This process is modeled as a suspended function that will otherwise leak.–

Infix for what it is coming below: In Compose UI, all the things happening on UI are coordinated / dispatched using the `AndroidUiDispatcher`, which for that reason is associated with a `Choreographer` instance and a handler for the main `Looper`. This dispatcher performs event dispatch during the handler callback or choreographer's animation frame stage, **whichever comes first**. It also has a `MonotonicFrameClock` associated that uses `suspend` to coordinate frame rendering. This is what drives the whole UX in Compose, and things like animations depend a lot on it for achieving a smooth experience in sync with the system frames.

First thing the factory function does is creating a `PausableMonotonicFrameClock`. This is a wrapper over the `AndroidUiDispatcher` monotonic clock that adds support for manually pausing the dispatch of `withFrameNanos` events until it is resumed. That makes it useful for cases where frames should **not be produced** during specific periods of time, like when a Window hosting a UI is no longer visible.

Any `MonotonicFrameClock` is also a `CoroutineContext.Element`, which means it can be combined with other `CoroutineContexts`.

When instantiating the Recomposer, we must provide a `CoroutineContext` to it. This context is created using a combination of the current thread context from the `AndroidUiDispatcher` and the pausable frame clock just created.

**WindowRecomposer.android**

```
1 val contextWithClock = currentThreadContext + (pausableClock ?: EmptyCoroutineContex\
2 t)
3 val recomposer = Recomposer(effectCoroutineContext = contextWithClock)
```

This combined context will be used by the Recomposer to create an internal Job to ensure that all composition or recombination effects can be cancelled when shutting down the Recomposer. This will be needed when an Android window is getting destroyed or unattached, for example. **This context will be the one used for applying changes** after composition / recombination, and will also be the default context used by LaunchedEffect to run effects. –That makes effects start in the same thread we use to apply changes, which in Android is usually the main thread. Of course we can always jump off the main thread at will within our effects.–

LaunchedEffect is an effect handler that will be explained in detail in the chapter about this topic. All the effect handlers are Composable functions and therefore emit changes that are recorded. LaunchedEffect is indeed recorded and written to the slot table when the time comes, so it is Composition lifecycle aware, not like SideEffect.

Finally, a coroutine scope is created using the same combined context: i.e: val runRecomposeScope = CoroutineScope(contextWithClock). This scope will be used to launch the recombination job (a suspend function), which will await for invalidations and trigger recompositions accordingly. Let's peek into the code and discuss some ideas about it.

**WindowRecomposer.android.kt**

```
1 viewTypeLifecycleOwner.lifecycle.addObserver(
2     object : LifecycleEventObserver {
3         override fun onStateChanged(lifecycleOwner: LifecycleOwner, event: Lifecycle.Eve\
4 nt) {
5             val self = this
6
7             when (event) {
8                 Lifecycle.Event.ON_CREATE ->
9                     runRecomposeScope.launch(start = CoroutineStart.UNDISPATCHED) {
10                     try {
11                         recomposer.runRecomposeAndApplyChanges()
12                     } finally {
13                         // After completion or cancellation
14                         lifecycleOwner.lifecycle.removeObserver(self)
15                     }
16                 }
17             }
18         }
19     }
20 )
```

```

17     Lifecycle.Event.ON_START -> pausableClock?.resume()
18     Lifecycle.Event.ON_STOP -> pausableClock?.pause()
19     Lifecycle.Event.ON_DESTROY -> {
20         recomposer.cancel()
21     }
22 }
23 }
24 }
25 )

```

---

Here is where the things happen. An observer is attached to the view tree lifecycle, and it will use the pausable clock to resume and pause event dispatch when view tree is started and stopped, respectively. It will also shutdown (cancel) the Recomposer on destroy, and launch the recompilation job on create.

The recompilation job is started by `recomposer.runRecomposeAndApplyChanges()`, which is the suspend function mentioned above that will await for the invalidation of any associated Composers (and their `RecomposeScopes`), recompile them, and ultimately apply the new changes to their associated Composition.

This factory is how Compose UI spawns a Recomposer connected to the Android lifecycle. It works nicely as an example of how the Recomposer is created at the integration point with the platform, along with the Composition. As a refresher, here we can see again how the composition was created when setting the content for `ViewGroups`:

#### Wrapper.android.kt

```

1 internal fun ViewGroup.setContent(
2     parent: CompositionContext, // Recomposer is passed here!
3     content: @Composable () -> Unit
4 ): Composition {
5     // ...
6     val composeView = ...
7     return doSetContent(composeView, parent, content)
8 }
9
10 private fun doSetContent(
11     owner: AndroidComposeView,
12     parent: CompositionContext,
13     content: @Composable () -> Unit
14 ): Composition {
15     // ...
16     val original = Composition(UiApplier(owner.root), parent) // Here!
17     val wrapped = owner.view.getTag(R.id.wrapped_composition_tag)
18     as? WrappedComposition ?: WrappedComposition(owner, original).also {

```

```

19     owner.view.setTag(R.id.wrapped_composition_tag, it)
20 }
21 wrapped.setContent(content)
22 return wrapped
23 }
```

---

That parent there will be a Recomposer, and will be provided by the caller of `setContent`, that for this use case it is the `AbstractComposeView`.

## Recomposition process

The `recomposer.runRecomposeAndApplyChanges()` function is called to start awaiting for invalidations and automatically recompose when those take place. Let's learn the different steps involved.

On a previous section we learned how snapshot State is modified within its own snapshot, but later those changes need to be propagated to the global state via `snapshot.apply()` for sync. When calling `recomposer.runRecomposeAndApplyChanges()`, the first thing it does is registering an observer for that change propagation. When that happens, this observer awakes and adds all those changes to a list of snapshot invalidations that are propagated to all known composers so they can record what parts of the composition need to be recomposed. In simple terms, this observer is a stepping stone for triggering automatic recomposition when State changes.

After registering the snapshot apply observer, the Recomposer invalidates all Compositions to assume everything has changed as a starting point. Any changes happening before this moment have not been tracked, so this is a way to start from scratch. Then it suspends until there is work available for recomposition. “Having work available” means having any pending State snapshot invalidations, or any composition invalidations coming from `RecomposeScopes`.

The next thing the Recomposer does is using the monotonic clock provided when creating it, and call `parentFrameClock.withFrameNanos {}` to await for the next frame. The rest of the work from here will be performed at that time and not before. The intention is to coalesce changes to the frame.

Inside this block, the Recomposer dispatches the monotonic clock frames first for any potential awaiters (like animations). That might yield new invalidations as a result that also need to be tracked (e.g: toggling a conditional Composable when an animation ends).

And now it's time for the real action. The Recomposer takes all the pending snapshot invalidations, or in other words, all the State values modified since last call to recompose, and records all those changes in the composer as pending recompositions.

There could also be invalidated Compositions –via `composition.invalidate()`–, for example when a State is written in a Composable lambda–. For each one of those, the Recomposer performs recomposition (a section on this below) and adds it to the list of Compositions with changes pending to apply.

Recomposing means recalculating all the Changes necessary for the Composition state (slot table) and the materialized tree (Applier), as we have learned. We have seen how that is done already – see section: “[The initial Composition process](#)” –. Recomposition reuses all that code, so no point on repeating all the steps that the process follows here.

Later, it finds potential trailing recompositions that need to be composed because of a value change by a composition, and schedules them for recomposition also. This can happen for example if a `CompositionLocal` changes in a parent and was read in a child composition that was otherwise valid.

Finally, it goes over all the Compositions with changes to apply and calls `composition.applyChanges()` on them. After that, it updates the Recomposer state.

## Concurrent recomposition

The Recomposer has the ability to perform recompositions concurrently, even if Compose UI does not make use of this feature. Any other client libraries could rely on it though, based on their needs.

The Recomposer provides a concurrent counterpart to the `runRecomposeAndApplyChanges` function that is called `runRecomposeConcurrentlyAndApplyChanges`. This is another suspend function for awaiting for State snapshot invalidations and triggering automatic recompositions like the former, but with the only difference being that the latter will perform recomposition of invalidated Compositions in a `CoroutineContext` provided from the outside:

Recomposer.kt

---

```
1 suspend fun runRecomposeConcurrentlyAndApplyChanges(
2     recomposeCoroutineContext: CoroutineContext
3 ) { /* ... */ }
```

---

This suspend function creates its own `CoroutineScope` using the passed context and uses it to spawn and coordinate all the child jobs created for all the concurrent recompositions required.

## Recomposer states

The Recomposer switches over a series of states during its lifespan:

**Recomposer.kt**

---

```
1 enum class State {
2     ShutDown,
3     ShuttingDown,
4     Inactive,
5     InactivePendingWork,
6     Idle,
7     PendingWork
8 }
```

---

This has been extracted directly from the kdocs, and there is no point on rewording it. Here you have what each one of those states means:

- **ShutDown**: Recomposer was cancelled and cleanup work completed. Cannot be used anymore.
- **ShuttingDown**: Recomposer was cancelled but it still in the middle of the cleanup process. Cannot be used anymore.
- **Inactive**: Recomposer will ignore invalidations from Composers and will not trigger recomposition accordingly. `runRecomposeAndApplyChanges` has to be called to start listening. This is the initial state of a Recomposer after creation.
- **InactivePendingWork**: There is the chance that the Recomposer is inactive but already has some pending effects awaiting a frame. The frame will be produced as soon as the recomposer starts running.
- **Idle**: Recomposer is tracking composition and snapshot invalidations, but there is currently no work to do.
- **PendingWork**: Recomposer has been notified of pending work and is already performing it or awaiting the opportunity to do it. (We already described what “pending work” means for the Recomposer).

# 4. Compose UI

When we talk about Jetpack Compose we are usually referring to all its parts as a whole: The compiler, the runtime, and maybe Compose UI. In previous chapters, we learned about the Compiler, and how it enables optimizations and features in the runtime. After that, we studied the runtime itself, and discovered how it contains the real machinery behind Compose, where all its power and features reside. Finally, it's the turn of Compose UI, a client library for the runtime.

A short disclaimer before we start. I picked Compose UI as an example of a client library for the Compose runtime in this book, but note that there are other client libraries like [Compose for Web<sup>a</sup>](#), built and maintained by JetBrains, or [Mosaic<sup>b</sup>](#), a command line UI library by Jake Wharton. These libraries could suit as nicely as an example. Also note that the last chapter of this book is actually an in-depth look on how to write client libraries for Jetpack Compose.

<sup>a</sup><https://compose-web.ui.pages.jetbrains.team/>

<sup>b</sup><https://github.com/JakeWharton/mosaic>

## Integrating UI with the Compose runtime

Compose UI is a **Kotlin multiplatform** framework. It provides building blocks and machinery to emit UI via Composable functions. On top of that, this library also [includes Android and Desktop sourcesets<sup>7</sup>](#) that provide integration layers for Android and Desktop.

JetBrains actively maintains the Desktop sourceset, while Google maintains the Android and common ones. Both, Android and Desktop sourcesets rely on the common sourceset. Compose for Web stays out of Compose UI so far, since it is built using the DOM.

When integrating UI with the Compose runtime, the aim is to build a layout tree that the user can experience on screen. This tree is created and later updated by executing Composable functions that emit UI. The node type used for the tree is only known by Compose UI, so the runtime can stay agnostic of it. Even if Compose UI is already a Kotlin multiplatform framework per se, its node types are only supported by Android and Desktop so far. Other libraries like Compose for Web use different node types. For this reason, the node types emitted by a client library must only be known

<sup>7</sup><https://cs.android.com/androidx/platform/frameworks/support/+/androidx-main:compose/ui/ui/src/>

by the client library, and the runtime delegates on it the acts of inserting, removing, moving, or replacing nodes from the tree. We will get back to this later in this chapter.

The processes of initial composition and later recompositions take part in the process of building and updating the layout tree. These processes execute our Composable functions, which makes them schedule changes to insert, remove, move, or replace nodes from the tree. This creates a list of changes that is traversed later on using the `Applier` to detect the changes that affect the structure of the tree, and map those to actual changes to the tree, so the end user can experience them. If we are in the initial process of composition, these changes will insert all nodes and therefore build up our layout tree. If we are in a recomposition, they will update it. Recomposition will be triggered when the input data for our Composable functions changes (i.e: their parameters or mutable state read from them).

In previous chapters went over all this briefly. This is the perfect chapter to expand on the topic.

## Mapping scheduled changes to actual changes to the tree

When Composable functions execute during the processes of composition or recomposition, they emit their changes. On top of this, a side table called `Composition` is used (we will capitalize this one from now on, to differentiate it from the process of composition). This table contains relevant data to map from Composable function execution (scheduled changes) to actual changes to the node tree.

In an application using Compose UI we can have as many Compositions as the number of node trees we need to represent. This might sound a bit surprising at this point, since so far on this book we have never mentioned that Compositions can be multiple. But they actually are! Let's learn more about this, and while we do it, we will rapidly understand how the layout tree is built, and what node types are used.

## Composition from the point of view of Compose UI

If we take the Android integration as an example, the more frequent entry point from the Compose UI library into the runtime takes place when we call `setContent`, maybe for one of our screens.

**MainActivity.kt**

---

```
1 class MainActivity : ComponentActivity() {
2     override fun onCreate(savedInstanceState: Bundle?) {
3         super.onCreate(savedInstanceState)
4         setContent {
5             MaterialTheme {
6                 Text("Hello Compose!")
7             }
8         }
9     }
10 }
```

---

But a screen (E.g: Activity/Fragment in Android) is not the only place where we can find a `setContent` call. It can also happen in the middle of our `View` hierarchy, for example, via `ComposeView` (e.g: in a hybrid Android app):

**ComposeView.setContent**

---

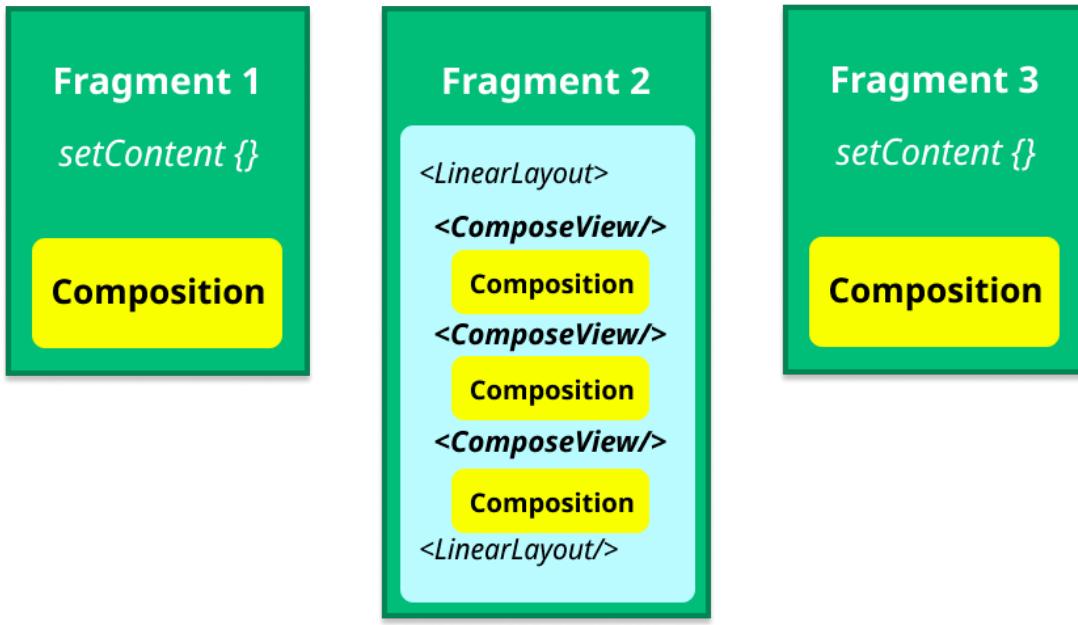
```
1 ComposeView(requireContext()).apply {
2     setContent {
3         MaterialTheme {
4             Text("Hello Compose!")
5         }
6     }
7 }
```

---

In this example, we are creating the view programmatically, but it could also be part of any layout hierarchy defined via XML within our app.

The `setContent` function creates a new root **Composition**, and then it reuses it as possible. I'm calling these "root" Compositions, since each one hosts an independent Composable tree. Those compositions are not connected to each other in any way. Each Composition will be as simple or as complex as the UI it represents.

With this mindset, we could imagine multiple node trees in our app, each one linked to a different Composition. Let's imagine an Android app with 3 Fragments (image below). Fragments 1 and 3 call `setContent` to hook their Composable trees, while Fragment 2 declares multiple `ComposeViews` on its layout, (and calls `setContent` on them). In this scenario, our app would have 5 root Compositions, all of them completely independent.



Multiple root Compositions

To create any of those layout hierarchies, the associated Composer will run the composition process. When doing so, all the Composable functions within the corresponding `setContent` call will execute and emit their changes. In the case of Compose UI, those will be changes to insert, move, or replace UI nodes, and they will be emitted by the usual UI building blocks. I.e: Box, Column, LazyColumn...etc. Even if those Composables usually belong to different libraries (foundation, material), all of them are ultimately defined as Layouts (compose-ui), which means they emit the same node type: A `LayoutNode`.

The `LayoutNode` was already introduced in the previous chapter. It is the representation of a UI block, and for that reason, it is the node type most frequently used for a root Composition in Compose UI.

Any Layout Composable emits a `LayoutNode` node into the Composition, and it does it via a `ReusableComposeNode` (Note that `ComposeUiNode` is a contract implemented by `LayoutNode`):

### Layout.kt

```

1  @Composable inline fun Layout(
2      content: @Composable () -> Unit,
3      modifier: Modifier = Modifier,
4      measurePolicy: MeasurePolicy
5  ) {
6      val density = LocalDensity.current
7      val layoutDirection = LocalLayoutDirection.current
8      val viewConfiguration = LocalViewConfiguration.current
9

```

```
10 // Emits a LayoutNode!
11 ReusableComposeNode<ComposeUiNode, Applier<Any>>(
12     factory = { LayoutNode() },
13     update = {
14         set(measurePolicy, { this.measurePolicy = it })
15         set(density, { this.density = it })
16         set(layoutDirection, { this.layoutDirection = it })
17         set(viewConfiguration, { this.viewConfiguration = it })
18     },
19     skippableUpdate = materializerOf(modifier),
20     content = content
21 )
22 }
```

---

This will emit a change to insert or update a reusable node into the composition. And this will happen for any UI building blocks we use.

Reusable nodes are an optimization in the Compose runtime. When the key for a node changes, reusable nodes allow the Composer to recompose the node content (update it in place during recomposition) instead of discarding it and creating a new one. To allow this, the Composition acts like it is creating new content but the slot table is traversed like if it is recomposing. This optimization is only possible for nodes that can be completely described by the `set` and `update` operations in the `emit` call, or in other words, nodes that contain no hidden internal state. This is true for `LayoutNode`, but not for `AndroidView` for example. For that reason, `AndroidView` uses a standard `ComposeNode` instead of a reusable one.

`ReusableComposeNode` will create the node (via the `factory` function), initialize it (`update` lambda), and create a replaceable group to wrap all its content. A unique key will be assigned to this group so it can be identified later. Any nodes emitted as a result of calling the `content` lambda within the replaceable group will effectively become children of this node.

The `set` calls within the `update` block schedule their trailing lambdas to be executed only when the node is first created or the value for the corresponding property has changed since last time it was remembered.

That is how `LayoutNodes` are fed to each one of the multiple Compositions that our application might have. And this might lead us to think that any Composition contains `LayoutNodes` only. But that is wrong! There are other types of Compositions and other types of nodes we need to think of when learning about how to feed the runtime in Compose UI.

## Subcomposition from the point of view of Compose UI

Compositions not only exist at a root level. A Composition can also be created in a deeper level of our Composable tree, and get linked to its parent Composition. This is what Compose refers to as **Subcomposition**. Something we learned in the previous lesson is that Compositions can be connected as a tree. That is, each Composition has a reference to its parent `CompositionContext`, which stands for its parent Composition (except for the root one, whose parent is the `Recomposer` itself). This is how the runtime ensures that `CompositionLocals` and invalidations can be resolved / propagated down the tree as if we had a single Composition.

In Compose UI, there are two main reasons to create a Subcomposition:

- Deferring the initial composition process until some information is known.
- Changing the type of the node produced by a subtree.

Let's address both of them.

### Deferring initial composition process

We have an example of this in `SubcomposeLayout`, which is an analogue of `Layout` that creates and runs an independent composition during the layout phase. This allows child Composables to depend on any values calculated in it. `SubcomposeLayout` is used by `BoxWithConstraints` for example, which exposes the incoming parent constraints in the block so it can adapt its content based on them. In the following example extracted from the official docs, `BoxWithConstraints` is used to decide between two different composables depending on the available `maxHeight`.

#### BoxWithConstraints Sample

---

```

1 BoxWithConstraints {
2     val rectangleHeight = 100.dp
3     if (maxHeight < rectangleHeight * 2) {
4         Box(Modifier.size(50.dp, rectangleHeight).background(Color.Blue))
5     } else {
6         Column {
7             Box(Modifier.size(50.dp, rectangleHeight).background(Color.Blue))
8             Box(Modifier.size(50.dp, rectangleHeight).background(Color.Gray))
9         }
10    }
11 }
```

---

The creator of a Subcomposition can control when the initial composition process happens, and `SubcomposeLayout` decides to do it during the layout phase, as opposed to when the root is composed.

Subcomposition allows to recompose independently of the parent Composition. In SubcomposeLayout for example, whenever a layout occurs, the parameters passed to its lambda might vary, and in that case it will trigger a recomposition. In the other hand, if a state that is read from a subcomposition changes, a recomposition will be scheduled for the parent Composition once the initial composition is performed.

In terms of nodes emitted, SubcomposeLayout also emits a LayoutNode, so the node type used for the subtree will be the same than the one used by the parent Composition. This leads to the following question: Is it possible to support different node types within a single Composition?

Well, it is technically possible, as long as the corresponding Applier allows it. It boils down to what is meant by the node type. If the node type used is a common parent for multiple subtypes, then different node types could be supported. Even though, that could make the Applier logic more cumbersome. The truth is that the Applier implementations available in Compose UI are fixed to a single node type.

That said, Subcomposition can actually enable **support for a completely different node type in a subtree**. Which is the second use case for Subcomposition that we listed above.

## Changing the node type in a subtree

Compose UI has a good example for this one: Composables that create and display vector graphics (e.g: rememberVectorPainter).

Vector Composables are a nice case study, since they also create their own Subcomposition to model the vector graphic as a tree. When composed, a Vector Composable emits a different node type to feed its Subcomposition: VNode. This is a recursive type that models standalone Paths or groups of Paths.

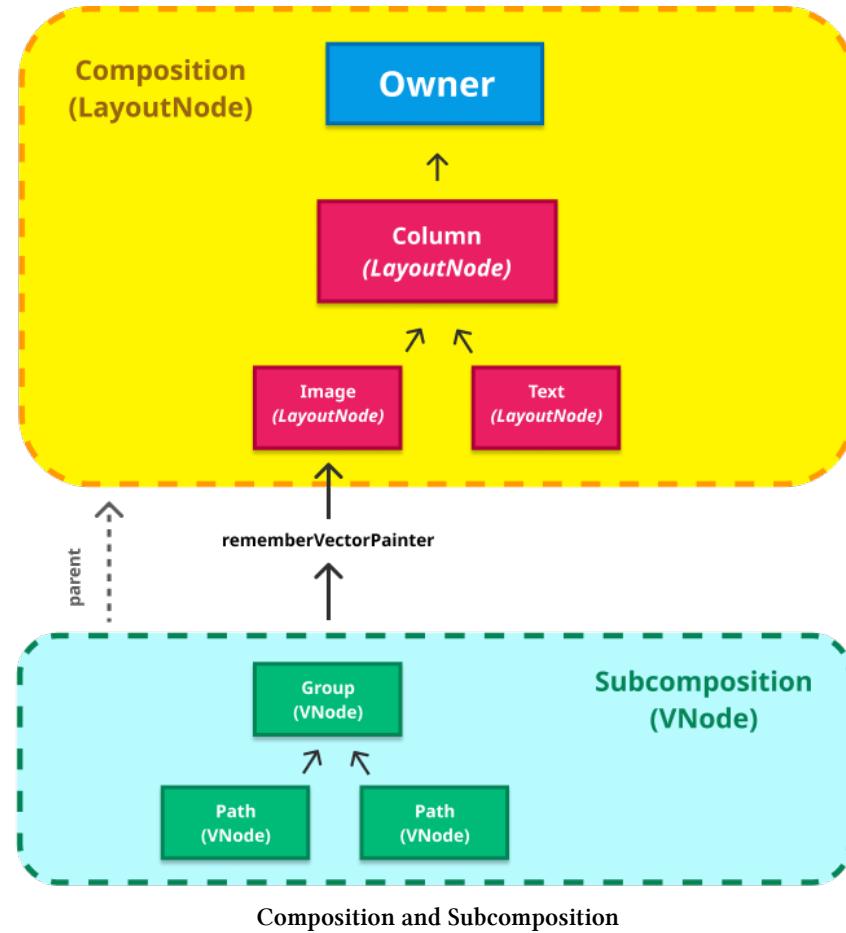
### Vector painter example

---

```
1 @Composable
2 fun MenuButton(onMenuClick: () -> Unit) {
3     Icon(
4         painter = rememberVectorPainter(image = Icons.Rounded.Menu),
5         contentDescription = "Menu button",
6         modifier = Modifier.clickable { onMenuClick() }
7     )
8 }
```

---

Something interesting to reflect on here is that we normally paint these vectors using a VectorPainter within an Image, an Icon, or some similar Composable to display it, as we see in the snippet above. That means the enclosing Composable is a Layout, hence it emits a LayoutNode into its associated Composition. But at the same time, the VectorPainter creates its own Subcomposition for the vector, and links it to the former Composition, which will become its parent. Here is a graphical example:



This configuration enables the vector subtree (Subcomposition) to use a different node type: VNode.

Vectors are modeled via Subcomposition because it is usually convenient to access some of the `CompositionLocals` available in the parent Composition from within the vector Composable call (e.g: `rememberVectorPainter`). Things like theme colors or density can be good examples.

Subcompositions created for vectors are disposed whenever their corresponding `VectorPainter` leaves the parent Composition, which will happen whenever its enclosing Composable also does. We will learn more about the Composable lifecycle in an upcoming chapter, but keep in mind that any Composable enters and leaves the Composition at some point.

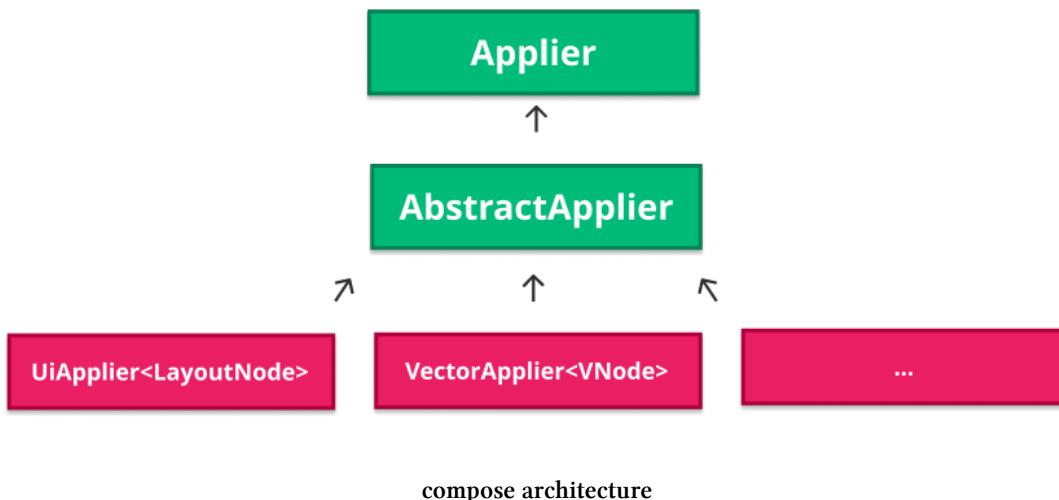
And with this, we have a more complete picture of how the tree looks like in an average Compose UI app (Android or Desktop), where we usually have root Compositions and Subcompositions. Time to understand the other side of the integration with the platform: Materializing the changes to experience them on screen.

## Reflecting changes in the UI

We have learned how UI nodes are emitted and fed to the runtime by running the processes of initial composition and later recompositions. At that point, the runtime takes it over and performs its work, as we learned in chapter 3. But this is only one side of the story, isn't it? There also needs to exist some integration to reflect all those emitted changes in the actual UI, so the user can experience them. That process is what is often called "materialization" of the node tree, and it is also a responsibility of the client library, in this case Compose UI. We also introduced this concept in the previous chapter, so let's use this chance to explore it in more detail.

## Different types of Appliers

Earlier in this book we described an **Applier** as an abstraction that the runtime relies on for ultimately materializing any changes from the tree. This inverts the dependencies to allow the runtime to stay completely agnostic of what platform the library is consumed from. This layer of abstraction allows client libraries like Compose UI to hook their own **Applier** implementations, and with them, **pick their own node type/s**, that will be used for integration with the platform. Here we have a simple diagram showcasing this:



*The top two boxes (Applier and AbstractApplier) are part of the Compose runtime. The bottom ones list some Applier implementations and are provided by Compose UI.*

The **AbstractApplier** is a base implementation provided by the Compose runtime for sharing logic between the different appliers. It stores the **visited** nodes in a **stack**, and it maintains a reference to the current node visited, so it knows what node it should be performing operations on. Whenever a new node is visited down the tree, the Composer notifies the Applier by calling `applier#down(node: N)`. That pushes the node to the stack, and the Applier can run any required operations on it.

Whenever the visitor needs to move back to the parent, the `Composer` calls `applier#up()`, and that pops the last node visited from the stack.

Let's understand this with a fairly simple example. Let's imagine we have the following Composable tree to materialize:

#### AbstractApplier example

---

```
1 Column {  
2     Row {  
3         Text("Some text")  
4         if (condition) {  
5             Text("Some conditional text")  
6         }  
7     }  
8     if (condition) {  
9         Text("Some more conditional text")  
10    }  
11 }
```

---

Whenever condition changes, the `Applier` will:

- Receive a down call for `Column`.
- Then another down call to get into the `Row`.
- Then a delete (or insert, depending on `condition`) for the optional child `Text`.
- After that, an `up` call will follow to get back to the parent (`Column`).
- Finally, a delete (or insert) for the second conditional `Text`.

The stack and the `down` and `up` operations are pulled into the `AbstractApplier` so child appliciers can share the same navigation logic regardless of the node types they work with. This provides a parent to child relation between the nodes, so it technically removes the need for specific node types to maintain this relation for navigating the tree. Even though, specific node types are still free to implement their own parent to child relation if they happen to need it for more specific reasons particular to the client library.

This is actually the case of `LayoutNode`, since not all its operations are performed during the composition. For example, if a node needs to be redrawn for any reason, Compose UI iterates through the parents to find the node that created the layer that the node draws to, in order to call `invalidate` on it. All that happens out of the composition, so Compose UI requires a means to traverse the tree up and down at will.

This is an interesting moment for a flashback to Chapter 3. In that chapter we described how the `Applier` can build the node tree **top-down** or **bottom-up**. We also described how there are performance implications for each approach, and how those implications depend on the number of nodes that need to be notified every time a new node is inserted. If you feel a bit lost now, don't worry. I totally recommend you to go back to the “**Performance when building the node tree**” section from Chapter 2 for a minute, and give it a quick re-read. I wanted to recap on this because there are actual examples of both strategies to build up the node tree in Compose UI. Those are put into practice by the two types of `Appliers` used by the library.

Compose UI provides two implementations of the `AbstractApplier` to integrate the Android platform with the Jetpack Compose runtime:

- `UiApplier`: For rendering most of the Android UI. This one fixes the node type to `LayoutNode`, so it will materialize all the `Layouts` in our tree.
- `VectorApplier`: For rendering vector graphics. It fixes the node type to `VNode` in order to represent and materialize vector graphics.

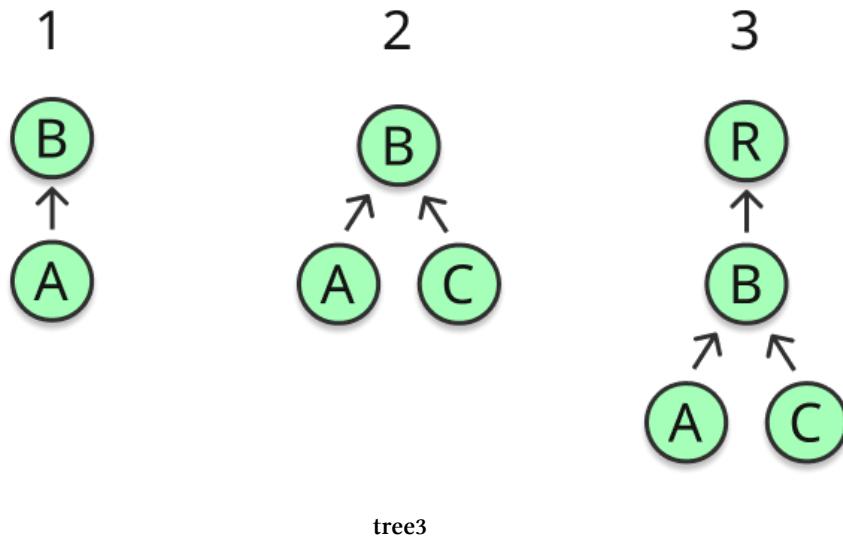
As you can see, those are the two node types we have introduced earlier in this chapter.

These are the only two implementations provided for Android so far, but the number of implementations available for a platform is not necessarily fixed. More could be added to Compose UI in the future, if there is a need to represent different node trees than the ones existing today.

Depending on the type of the node visited, a different `Applier` implementation will be used. E.g: If we have a root `Composition` fed with `LayoutNodes`, and a `Subcomposition` fed with `VNodes`, both `appliers` will be used to materialize the complete UI tree.

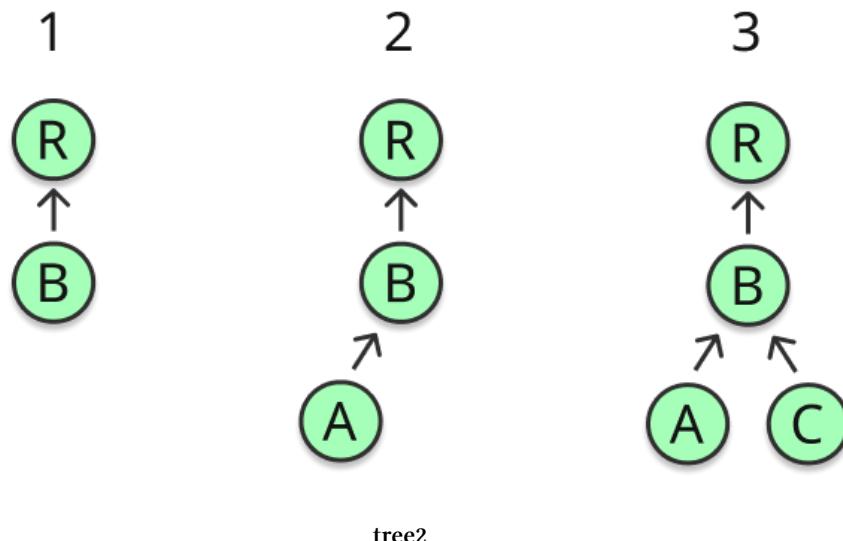
Let's give a rapid look to the strategies used by both `appliers` to build up the tree.

The `UiApplier` inserts nodes bottom-up. This is to avoid duplicate notifications when a new node enters the tree. Let me rescue the bottom-up insertion strategy diagram from Chapter 2 to make it clear:



A bottom-up building of the tree starts by inserting A and C into B, then inserting the B tree into R to complete the tree. That means it **only notifies the direct parent every time a new node is inserted**. This is particularly interesting for Android UIs (and therefore the `UiApplier`) where we normally have a lot of nesting (especially in Compose UIs where overdraw is not a problem) and therefore a lot of ancestors to notify.

In the other hand, the `VectorApplier` is an example of building the tree top-down. If we wanted to build the sample tree from above using a top-down strategy, we would first insert B into R, then insert A into B, and finally insert C into B. I.e:



In this strategy, each time we insert a new node we need to notify all its ancestors. But in the context of vector graphics, there is no need to propagate notifications to any nodes at all, so either strategy would be equally performant and therefore completely valid. There is not a strong reason to pick

top-down over bottom-up. Whenever a new child gets inserted into a `VNode`, the listener for that node is notified, but the children or parents are not.

Since we already have a good sense of the two different implementations of `Applier` that the Compose UI library uses, it is time to understand how both of them ultimately materialize a change in the UI.

## Materializing a new LayoutNode

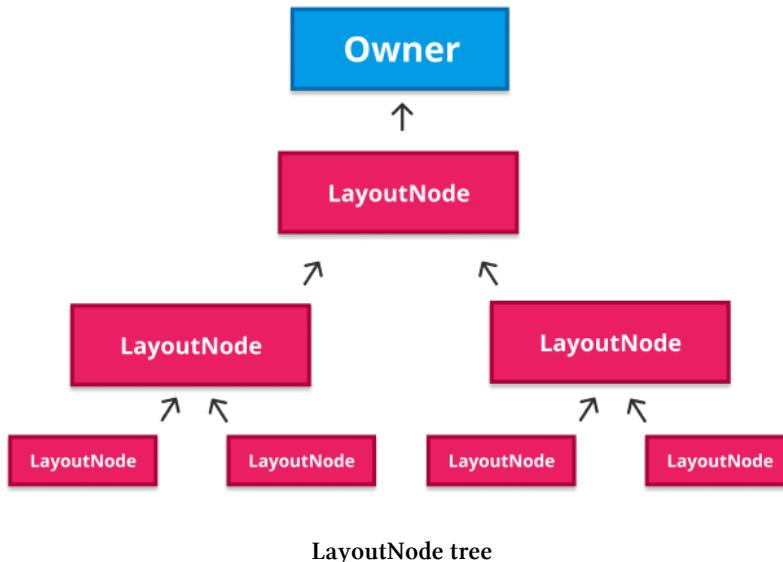
Here we have a simplification of the `UiApplier` used by Compose UI:

`UiApplier.android.kt`

```
1 internal class UiApplier(
2     root: LayoutNode
3 ) : AbstractApplier<LayoutNode>(root) {
4
5     override fun insertTopDown(index: Int, instance: LayoutNode) {
6         // Ignored. (The tree is built bottom-up with this one).
7     }
8
9     override fun insertBottomUp(index: Int, instance: LayoutNode) {
10        current.insertAt(index, instance)
11    }
12
13    override fun remove(index: Int, count: Int) {
14        current.removeAt(index, count)
15    }
16
17    override fun move(from: Int, to: Int, count: Int) {
18        current.move(from, to, count)
19    }
20
21    override fun onClear() {
22        root.removeAll()
23    }
24    ...
25 }
```

In this implementation we clearly see how the node type is fixed to be `LayoutNode`, and how all operations to `insert`, `remove`, or `move` nodes from the tree are delegated to the current node visited. That makes sense, since the `LayoutNode` knows how to materialize itself so the runtime can stay agnostic of that.

The `LayoutNode` is a pure Kotlin class with no Android dependencies, since it only models a UI node, and it is meant to be used by multiple platforms (Android, Desktop). It maintains a list of its children, and provides operations to insert, remove, or move (reorder) child nodes. `LayoutNodes` are connected as a tree, so each `LayoutNode` has a reference to its parent, and all of them are connected to the same `Owner`. This requirement is imposed to every new node attached. This is a diagram we already shared earlier in the book that shows the hierarchy.



The `Owner` is an abstraction, so each platform can implement it differently. It is expected to be the integration point with the platform. In Android, it is a `View` (`AndroidComposeView`), so it is the underlying connection between our Composable tree (`LayoutNodes`) and the Android `View` system. Whenever a node is attached, detached, reordered, remeasured, or updated anyhow, an invalidation can be triggered via the `Owner` using the good old Android `View` APIs, so the latest changes will be reflected on screen in the next drawing pass. That is basically how the magic happens.

Let's dive into a simplification of the `LayoutNode#insertAt` operation in order to understand how a new node gets inserted and therefore materialized. –Note that this is intentionally omitting some details, since implementation details might vary over time.–

#### `LayoutNode#insertAt.kt`

---

```

1 internal fun insertAt(index: Int, instance: LayoutNode) {
2     check(instance._foldedParent == null) {
3         "Cannot insert, it already has a parent!"
4     }
5     check(instance.owner == null) {
6         "Cannot insert, it already has an owner!"
7     }
8
9     instance._foldedParent = this

```

```
10    _foldedChildren.add(index, instance)
11    onZSortedChildrenInvalidated()
12
13    instance.outerLayoutNodeWrapper.wrappedBy = innerLayoutNodeWrapper
14
15    val owner = this.owner
16    if (owner != null) {
17        instance.attach(owner)
18    }
19 }
```

---

After a couple of safety checks to ensure that the node is not already in the tree neither attached, the current node is set as the parent of the new one getting inserted. Then, the new node is added to the list of children maintained by its new parent. On top of this, the list of sorted children in the Z index gets invalidated. This is a parallel list that maintains all the children sorted by their Z index, so they can be drawn in order (lower Z index first). Invalidating the list makes it get sorted again. This becomes necessary after inserting a new node, since Z index is not only determined by the order of `placeable.place()` calls in the layout (the order on what they are placed), but can also be set to an arbitrary value via a modifier. I.e: `Modifier.zIndex()`. (This is equivalent to how legacy `Views` are displayed on top of other `Views` when they are placed **after** those in a layout, and how we can also set their Z index to an arbitrary value).

The next thing we see is an assignment involving some sort of “outer” and “inner” `LayoutNodeWrappers`.

```
1 instance.outerLayoutNodeWrapper.wrappedBy = innerLayoutNodeWrapper
```

This is related to how a node, its modifiers, and its children are measured and drawn, and it will be described in detail in a standalone section about measuring later in this chapter, since it involves some complexity. We will get back to it soon!

Finally, it is time to attach the node, which means assigning it the same `Owner` than its new parent. Here is a simplification of the `attach` call, which is called over the node getting inserted (`instance.attach(owner)`):

## LayoutNode#attach.kt

---

```

1 internal fun attach(owner: Owner) {
2     check(_foldedParent == null || _foldedParent?.owner == owner) {
3         "Attaching to a different owner than the parent's owner"
4     }
5     val parent = this.parent // [this] is the node being attached
6
7     this.owner = owner
8
9     if (outerSemantics != null) {
10        owner.onSemanticsChange()
11    }
12    owner.onAttach(this)
13    _foldedChildren.forEach { child ->
14        child.attach(owner)
15    }
16
17    requestRemeasure()
18    parent?.requestRemeasure()
19 }
```

---

Here we find the guard that enforces all child nodes to get assigned the same `Owner` than their parent. Since the `attach` function is called recursively on the children, the complete subtree hanging from this node will ultimately get attached to the same `Owner`. That is to enforce all the invalidations required by any nodes from the same Composable tree to be piped through the same `View`, so it can handle all the coordination. After the guards, the `owner` is assigned.

At this point, if the node getting attached includes any semantics metadata, the `Owner` is notified. That can happen when a node is inserted, or when a node is removed. It can also happen if a value from the node gets updated, or if a semantic modifier is added or removed without the actual node being added or removed. In Android, the `Owner` has a delegate for accessibility that will forward the notifications to, so it can take care of the change, update the semantic trees, and do the wiring with the Android SDK accessibility APIs. We will explore the guts of Compose semantics in a later section.

The Semantic Tree is a parallel tree that describes the UI in a way that is understood by accessibility services and the testing framework so they can inspect it. There are two semantic trees maintained in parallel, but we will describe those in detail in an upcoming section.

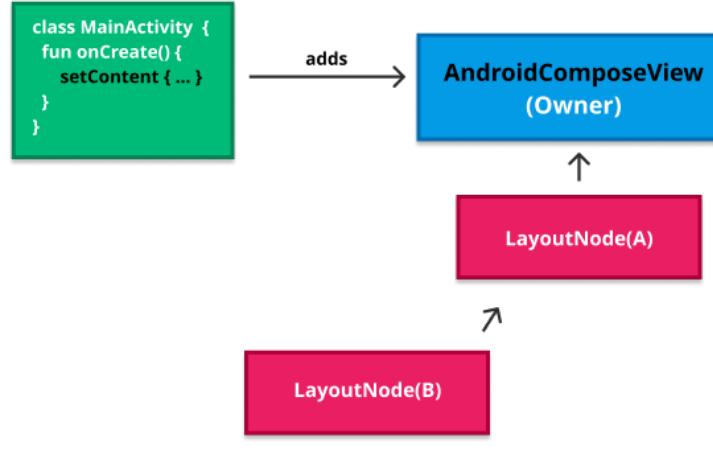
After this, remeasuring is proactively requested for the new node and its parent. This is a very important step in the process as I explained earlier, since it effectively materializes the node: Any

remeasuring request is piped via the `Owner`, so it uses the `View` primitives to call `invalidate` or even `requestLayout` when needed. This will effectively make the node finally show up on screen.

## Closing the circle

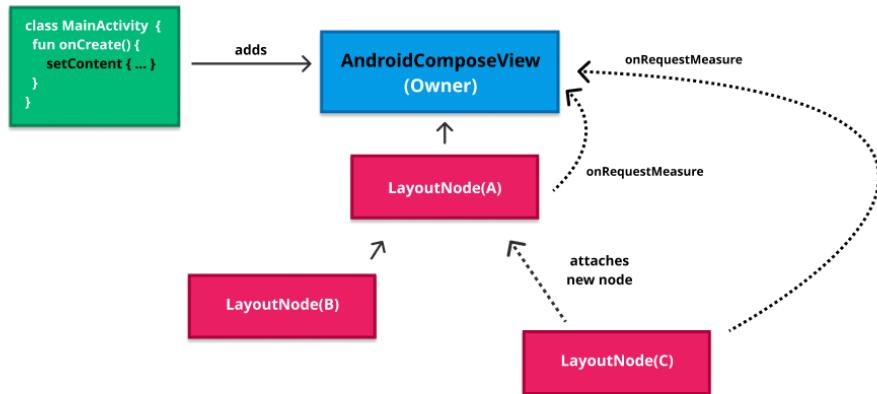
To close the full circle, the `Owner` gets attached to the `View` hierarchy as soon as the `setContent` call takes place in an `Activity`, `Fragment`, or `ComposeView`. That was the only part missing so far.

Let's recap on all this process with a simple example. Imagine we have the following tree, where `Activity#setContent` is called, so an `AndroidComposeView` is created and attached to the `View` hierarchy. And where we already have a couple `LayoutNodes` in place: A root node (`LayoutNode(A)`), and a child (`LayoutNode(B)`).



Materialization 1

Now, let's imagine that the `Applier` calls `current.insertAt(index, instance)` to insert (materialize) a new node `LayoutNode(C)`. This will attach the new node, which will request a remeasure for itself and its new parent, via the `Owner`.



### Materialization 2

When this happens, `AndroidComposeView#invalidate` will get called in most cases. It does not matter if both nodes (current, and parent) invalidate the same `View` (Owner) at the same time, since invalidation is like flagging the `View` as dirty. You can do it multiple times between two frames, but the `View` will get redrawn only once in any case (during the next drawing phase). At that moment, `AndroidComposeView#dispatchDraw` will get called, and that is where Compose UI does the actual remeasure and layout of all the requested nodes. If during such remeasure the root node's size changes, `AndroidComposeView#requestLayout()` is called in order to re-trigger `onMeasure` and be able to affect the sizes of all the `Views` siblings.

After measuring and layout, the `dispatchDraw` call will end up calling the draw function in the root `LayoutNode`, which knows how to draw itself to the Canvas and also triggers draw for all its children.

If a request for remeasuring a node takes place while the node is already measuring, it will be ignored. The same will happen if the node has a remeasure scheduled already.

Nodes are always measured first, then layout, and finally drawn. That is the order of things.

This is how a new node is inserted / materialized so the user can experience it on screen.

## Materializing a change to remove nodes

Removing one or multiple child nodes looks very similar. The `UiApplier` calls `current.removeAt(index, count)` to delegate removing any amount of children to the current node itself. Then, the current node (parent) iterates over all the children to remove, starting from the last one. For each one, it removes the child from the current node's list of children, triggers reordering of the Z index children list, and detaches the child and all its children subsequently from

the tree. To do that, it resets their `owner` reference to `null` and requests a remeasure for the parent, since it will be affected by the removal.

The same way it did when attaching a new node, the `Owner` is notified in case semantics change as a result of removing the node/s.

## Materializing a change to move nodes

Or in other words, reordering the children. When the `UiApplier` calls `current.move(from, to, count)` to move one or multiple children, it also iterates over them and calls `removeAt` (explained above) for each node to be moved. Then it adds the node again but in its new position. Finally, it requests a remeasuring for the current node (the parent).

## Materializing a change to clear all the nodes

This is the same than removing multiple nodes. It iterates over all the children (starting from the last one) and detaches every child, which requests a remeasure from the parent.

## Measuring in Compose UI

We already know how and when a remeasure is requested. Time to understand how measuring actually works.

Any `LayoutNodes` can request a remeasure via the `Owner`, for example when a child is attached, detached, or moved. At that point, the view (`Owner`) is flagged as “dirty” (`invalidate`), and the node gets added to a **list of nodes to remeasure and relayout**. In the next drawing pass, `AndroidComposeView#dispatchDraw` will get called (as it would happen for any invalidated `ViewGroup`), and the `AndroidComposeView` will iterate over the list and use a delegate to perform those actions.

For each node scheduled for a remeasure and relayout, 3 steps are taken (in this order):

1. Checks if the node needs a remeasure and performs it in that case.
2. After measuring, it checks if the node needs a relayout and performs it in that case.
3. Finally, it checks if there are any postponed measure requests for any nodes, and schedules a remeasure for those. I.e: adds those to the list of nodes to remeasure and relayout in the next pass, which takes us back to step 1. Remeasure requests get postponed when they take place while measuring is already happening.

For measuring each node (1), it will delegate it to the outer `LayoutNodeWrapper`. Remember? We said we’d cover the inner and outer wrappers of a `LayoutNode` in detail, so here we are. But before we do

that, a little infix: If the size of the node changes as a result of measuring, and the node has a parent, it will request a remeasure or a relayout for the parent, as required.

Let's go for the wrappers now, so we learn how a measuring pass takes place.

Going back to the `LayoutNode#insertAt` function for a second (the one called by the `UiApplier` for inserting a new node), we detected an assignment related to the outer and inner `LayoutNodeWrappers`:

`LayoutNode#insertAt.kt`

---

```
1 internal fun insertAt(index: Int, instance: LayoutNode) {
2     ...
3     instance.outerLayoutNodeWrapper.wrappedBy = innerLayoutNodeWrapper
4     ...
5 }
```

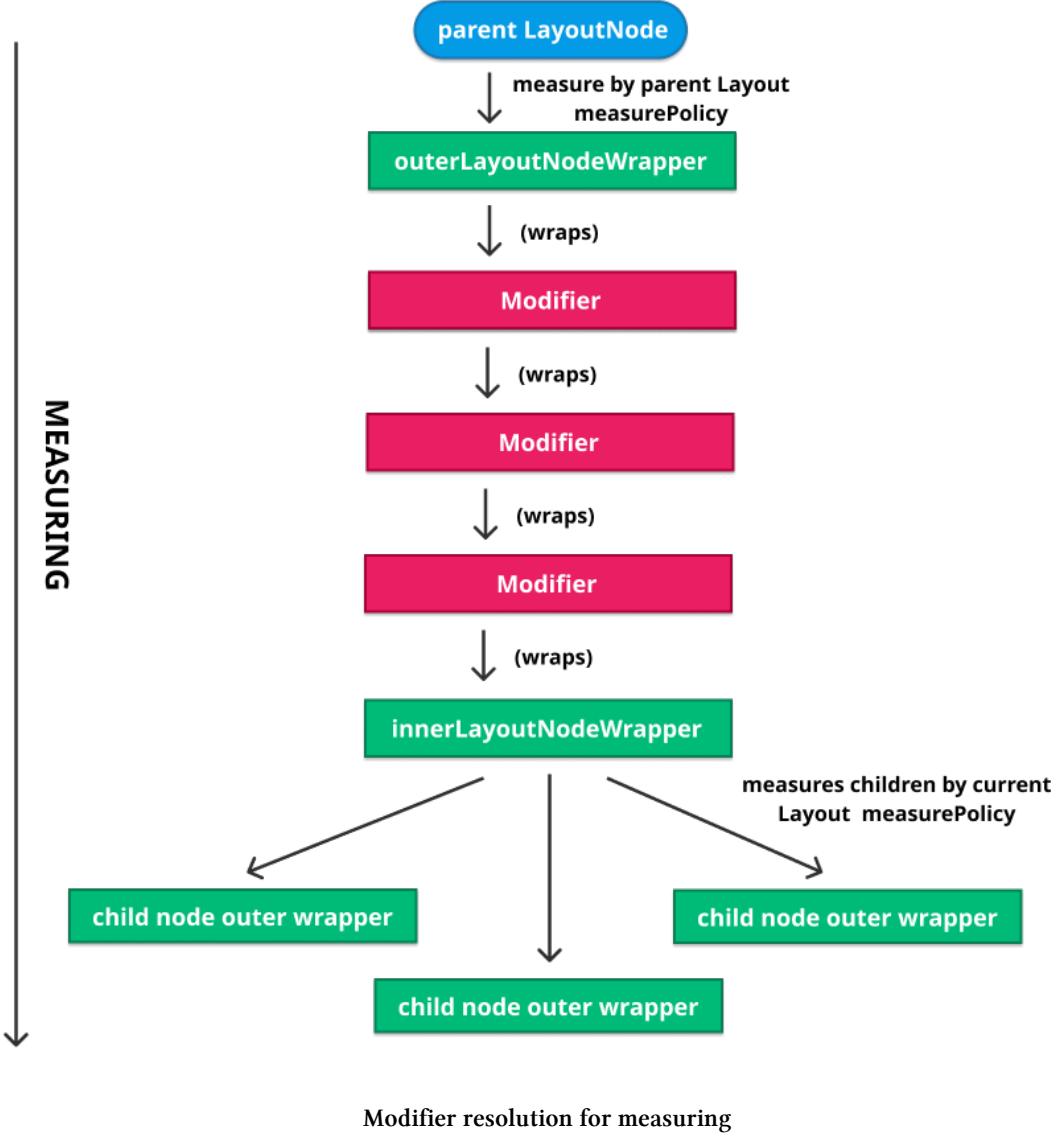
---

Each `LayoutNode` has an outer and an inner `LayoutNodeWrapper`. The outer one takes care of measuring and drawing the current node, and the inner one takes care of doing the same for its children.

This is great, but sadly also incomplete. Truth is, a node can have modifiers applied, and modifiers can also affect **measuring**, so they also need to be accounted for when measuring the node. For example: `Modifier.padding` directly affects the measure of the node's children. On top of this, modifiers can even affect the size of other modifiers chained after them. E.g: `Modifier.padding(8.dp).background(Color.Red)`, where only the space left after applying the padding will be colored. All this means that there is a need to keep the measured size of the modifier somewhere. **But Modifier is a stateless thing**, hence a wrapper is needed to keep its state. For this reason, `LayoutNode` does not only have outer and inner wrappers, but it also has a wrapper for each one of the modifiers applied to it. And all the wrappers (outer, modifiers, and inner) are chained so they are always resolved and applied in order.

The wrapper for a modifier includes its measured size, but also other hooks that can also be affected by measuring, like the ones to perform drawing (for modifiers like `Modifier.drawBehind()`), or the ones related to hit testing touches.

Here is how all the wrappers are linked together:



1. A parent LayoutNode uses its `measurePolicy` (defined in the Layout composable, more on this later) to measure the outer wrapper of all its children.
2. The outer wrapper of each children wraps the first modifier in the chain.
3. That one wraps the second.
4. Which wraps the third.
5. Which wraps the inner wrapper (assuming we have 3 modifiers in this node).
6. Which gets us back to step 1: The inner wrapper uses the current node's `measurePolicy` to measure the outer wrapper of each children.

This ensures that measuring is done in order and modifiers are also accounted for. Note that this wrapping is only done for `LayoutModifiers` starting from Compose 1.2, since other types of modifiers

are wrapped into easier abstractions. But regardless of the abstraction used, the approach remains the same.

When it comes to drawing, it works in the same way, but in the last step, the inner wrapper simply iterates over the list of children ordered by Z index, and calls draw on each one of them.

Now, let's get back to the `LayoutNodeWrapper` assignment from the `insertAt` function:

```
1 instance.outerLayoutNodeWrapper.wrappedBy = innerLayoutNodeWrapper
```

See how the outer wrapper of the node getting inserted is wrapped by the inner wrapper of the current node (its new parent). That is represented by both the first and last steps in the diagram we shared above.

When attaching a new node, all `LayoutNodeWrappers` are notified. Since they are stateful, they have a lifecycle, so they are notified about any attach and detach just in case they need to initialize and dispose anything. One example of this is focus modifiers, which send focus events when attaching them.

Whenever remeasure is requested for a node, the action is delegated to its `outerLayoutNodeWrapper`, which makes use of the parent's measure policy for measuring it. After this, it follows the chain to remeasure each one of its modifiers, to finally end up in the inner `LayoutNodeWrapper` for remeasuring the children by using the current node's measure policy.

While measuring a node, any mutable state reads within the measure lambda (measure policy) are recorded. That will make the lambda reexecute itself whenever the mentioned state varies. After all, the measuring policy is passed from the outside, and it is free to rely on Compose State. Automatically reading snapshot state is something we will expand a lot in the next chapter.

After measuring, the previous measured size is compared to the current one, in order to request a remeasure for the parent in case it changed.

Time to learn something about measuring policies.

## Measuring policies

When a node needs to be measured, the corresponding `LayoutNodeWrapper` relies on the measuring policy provided when emitting the node. That is:

**Layout.kt**

---

```
1 @Composable inline fun Layout(
2     content: @Composable () -> Unit,
3     modifier: Modifier = Modifier,
4     measurePolicy: MeasurePolicy
5 ) {
6     ...
7     ReusableComposeNode<ComposeUiNode, Applier<Any>>(
8         factory = { LayoutNode() },
9         update = {
10             set(measurePolicy, { this.measurePolicy = it })
11             ...
12         },
13         skippableUpdate = materializerOf(modifier),
14         content = content
15     )
16 }
```

---

But note how the policy is actually passed from the outside. The `LayoutNode` always remains agnostic of the measuring policies used to measure itself and its children. Compose UI expects any implementations of `Layout` to provide their own measuring policy, since it varies for each use case.

Each time the measuring policy for a `LayoutNode` changes, a remeasuring is requested.

If you have created any custom layouts in Jetpack Compose before, this might be starting to sound a bit familiar. Measuring policies are the lambda we pass to a custom `Layout` when we create it. If you don't have experience with it, I definitely recommend giving a read to [the official Jetpack Compose docs about this topic<sup>8</sup>](#).

One of the most simple policies is probably the one set by the `Spacer` Composable:

**Spacer.kt**

---

```
1 @Composable
2 fun Spacer(modifier: Modifier) {
3     Layout({}, modifier) { _, constraints ->
4         with(constraints) {
5             val width = if (hasFixedWidth) maxWidth else 0
6             val height = if (hasFixedHeight) maxHeight else 0
7             layout(width, height) {}
8         }
9     }
10 }
```

---

<sup>8</sup><https://developer.android.com/jetpack/compose/layouts/custom>

The trailing lambda defines the policy. It is the actual implementation of the `MeasurePolicy#measure` function, which works with a list of measurables for the layout children (none in this case, the parameter is ignored), and the constraints that each child should respect. The constraints are used to determine the width and height of the layout. When they are fixed (they are set to an exact width or height), the `Spacer` basically sets them (`maxWidth == minWidth` and `maxHeight == minHeight` in this case). Otherwise it defaults both dimensions to 0. This effectively means that the `Spacer` will always need to get some size constraints imposed, via the parent or a modifier. This makes sense, since a `Spacer` does not contain any children, so it cannot adapt to its wrapped content.

Another (likely more complete) example of a measuring policy can be found in the `Box` Composable:

#### Box.kt

---

```
1 @Composable
2 inline fun Box(
3     modifier: Modifier = Modifier,
4     contentAlignment: Alignment = Alignment.TopStart,
5     propagateMinConstraints: Boolean = false,
6     content: @Composable BoxScope.() -> Unit
7 ) {
8     val measurePolicy = rememberBoxMeasurePolicy(contentAlignment, propagateMinConstra\
9 ints)
10    Layout(
11        content = { BoxScopeInstance.content() },
12        measurePolicy = measurePolicy,
13        modifier = modifier
14    )
15 }
```

---

This policy depends on the alignment set for the box (defaults to `TopStart`, so it will align children from top to bottom and from left to right), and if the parent min constraints need to be imposed to the content. There are lots of components defined on top of `Box`, or including it in their layouts, like `Crossfade`, `Switch`, `Surface`, `FloatingActionButton`, `IconButton`, `AlertDialog`... and many more. We can't mention all of them, but for this reason the option to propagate the minimum constraints stays open so each component can decide. Let's dive into the measuring policy.

Bear in mind that this is all implementation details, so might change over time. Even though, it can work as a didactic example. The policy starts like this:

**Box.kt**


---

```

1 MeasurePolicy { measurables, constraints ->
2     if (measurables.isEmpty()) {
3         return@MeasurePolicy layout(
4             constraints.minWidth,
5             constraints.minHeight
6         ) {}
7     }

```

---

Which means that, in case the Box has no children at all, it will adapt to the minimum width and height imposed by the parent (or a modifier). After this, it takes the imposed constraints as they are in case the minimum constraints are set to be propagated to the children. Otherwise it removes the minimum width and height limitations so children can decide about that:

**Box.kt**


---

```

1 val contentConstraints = if (propagateMinConstraints) {
2     constraints
3 } else {
4     constraints.copy(minWidth = 0, minHeight = 0)
5 }

```

---

After getting the constraints ready, it checks if there is only one children, and if that is the case it proceeds to measuring and placing it:

**Box.kt**


---

```

1 if (measurables.size == 1) {
2     val measurable = measurables[0]
3     ...
4     if (!measurable.matchesParentSize) {
5         placeable = measurable.measure(contentConstraints)
6         boxWidth = max(constraints.minWidth, placeable.width)
7         boxHeight = max(constraints.minHeight, placeable.height)
8     } else {
9         boxWidth = constraints.minWidth
10        boxHeight = constraints.minHeight
11        placeable = measurable.measure(
12            Constraints.fixed(constraints.minWidth, constraints.minHeight)
13        )
14    }
15    return@MeasurePolicy layout(boxWidth, boxHeight) {
16        placeInBox(placeable, measurable, layoutDirection, boxWidth, boxHeight, alignmen\

```

```
17 t)  
18 }  
19 }
```

---

We find an interesting distinction here. We have two scenarios: sizing the Box to wrap its content or sizing it to match its parent.

When the (only) children is not set to match the parent size (e.g: `Modifier.fillMaxSize()`), the Box will adapt its size to its wrapped content. To do it, it measures the child first, using the imposed constraints. That returns the size that the child would take if those constraints were imposed. Then the Box uses that size to determine its own size. The box width will be the maximum value from comparing the `minWidth` from the constraints, and the child width. And the same happens for the height. This effectively means that the box will never get smaller than its single child.

In the other hand, when the Box is set to match the parent size, it will set its width and height to exactly the same value than the minimum width and height imposed by the parent constraints.

But this only covers the case when there is a single child. What happens when there are more? The policy also defines that, of course. It starts by measuring all the children that are set to **not** match parent size in order to get the size of the Box:

#### Box.kt

```
1 val placeables = arrayOfNulls<Placeable>(measurables.size)  
2  
3 var hasMatchParentSizeChildren = false  
4 var boxWidth = constraints.minWidth  
5 var boxHeight = constraints.minHeight  
6 measurables.fastForEachIndexed { index, measurable ->  
7     if (!measurable.matchesParentSize) {  
8         val placeable = measurable.measure(contentConstraints)  
9         placeables[index] = placeable  
10        boxWidth = max(boxWidth, placeable.width)  
11        boxHeight = max(boxHeight, placeable.height)  
12    } else {  
13        hasMatchParentSizeChildren = true  
14    }  
15 }
```

---

The first line in this snippet initializes a collection of `placeables` to keep track of all of the measured children, since it will need to place them eventually.

After this, it iterates over all the children in order to calculate the maximum width and height possible between min constraints imposed by the parent, and the width and height that each child would take for those constraints. This effectively makes the Box adapt to its children when they

exceed the imposed minimum constraints, or otherwise set the minimum constraints as its size. Since this process requires measuring each child, it will also use the chance for adding the resulting placeables to the list. Note that any children than are set to **match the parent size** are ignored during this process, since they will be accounted for measuring the Box in the next step.

When measuring children that are set to match the parent size, there is something interesting to highlight: If we have unbounded constraints for the Box calculated so far, it will imply setting the minimum constraints (for measuring each child) to 0. That means that each child will decide how narrow it wants to be. This scenario is only possible when the `boxWidth` or `boxHeight` calculated so far are equal to the infinity, which can happen only if the minimum dimensions imposed by the parent in the constraints were unbounded. In case they are not, the already calculated `boxWidth` and `boxHeight` will be used:

#### Box.kt

---

```
1 if (hasMatchParentSizeChildren) {  
2     // The infinity check is needed for default intrinsic measurements.  
3     val matchParentSizeConstraints = Constraints(  
4         minWidth = if (boxWidth != Constraints.Infinity) boxWidth else 0,  
5         minHeight = if (boxHeight != Constraints.Infinity) boxHeight else 0,  
6         maxWidth = boxWidth,  
7         maxHeight = boxHeight  
8     )  
9     measurables.fastForEachIndexed { index, measurable ->  
10        if (measurable.matchesParentSize) {  
11            placeables[index] = measurable.measure(matchParentSizeConstraints)  
12        }  
13    }  
14 }
```

---

At the bottom most of the previous snippet we can see how all the children that match parent size are measured using the calculated constraints, and added to the list of `placeables`.

The final action in this measure policy is essentially to create the layout using the calculated width and height, and place all the children inside.

**Box.kt**

---

```
1 layout(boxWidth, boxHeight) {
2     placeables.forEachIndexed { index, placeable ->
3         placeable as Placeable
4         val measurable = measurables[index]
5         placeInBox(placeable, measurable, layoutDirection, boxWidth, boxHeight, alignmen\
6 t)
7     }
8 }
```

---

Cheers! we have reached the end of it. We got a proper measuring policy in place.

There are many more examples of measuring policies in Compose UI. Even if we can't list and describe all of them, going over this one should have helped to understand how they work in more detail.

## Intrinsic measurements

A `MeasurePolicy` includes some methods to calculate the intrinsic size of a layout. That is, the estimated size of a layout when we don't have constraints available.

Intrinsic measures come handy whenever we need to know the estimated dimensions of a child before we can actually measure it. One example can be if we want to match the height of a child with the height of its tallest sibling. How could we do this during the measuring phase, if siblings have not been measured yet?

Well, one way to do this could be subcomposition, but sometimes we don't need to go that far. We might also think about measuring twice, but that is not possible: Compose enforces a single time measure for Composables (for performance reasons). Trying to measure twice will throw.

Intrinsics can be a good compromise solution. Any `LayoutNode` has a measure policy assigned as we've already learned, but it also has an intrinsics measure policy that depends on the former. This dependency makes any ancestors depending on the intrinsic measures of the node get their layout recalculated whenever the node's measure policy changes.

This intrinsics policy assigned to the `LayoutNode` provides methods to calculate:

- `minIntrinsicWidth` for a given height.
- `minIntrinsicHeight` for a given width.
- `maxIntrinsicWidth` for a given height.
- `maxIntrinsicHeight` for a given width.

As you can see, we always need to provide the opposite dimension in order to calculate the one we need. That is because we don't have any constraints, so the only clue we can give to the library to calculate an appropriate size **to paint the layout content correctly**, is one of the dimensions (so it can calculate the other).

We see this more clearly by reading the official kdocs of each one of those functions:

- The `minIntrinsicWidth` function provides the minimum width a layout can take, given a specific height, such that the content of the layout can be painted correctly.
- The `minIntrinsicHeight` function provides the minimum height this layout can take, given a specific width, such that the content of the layout will be painted correctly.
- The `maxIntrinsicWidth` function provides the minimum width such that increasing it further will not decrease the minimum intrinsic height.
- The `maxIntrinsicHeight` function provides the minimum height such that increasing it further will not decrease the minimum intrinsic width.

We can use `Modifier.width(intrinsicSize: IntrinsicSize)` (or its `height` counterpart) as an example to understand intrinsics. Note that this is a different variant than the usual `Modifier.width(width: Dp)`. While the latter is used to declare an **exact** preferred width for the node, the former is used to declare a preferred width for the node that matches its own min or max intrinsic width. Here is how it is implemented:

#### Intrinsic.kt

---

```
1 @Stable
2 fun Modifier.width(intrinsicSize: IntrinsicSize) = when (intrinsicSize) {
3   IntrinsicSize.Min -> this.then(MinIntrinsicWidthModifier)
4   IntrinsicSize.Max -> this.then(MaxIntrinsicWidthModifier)
5 }
```

---

We can call it like `Modifier.width(IntrinsicSize.Max)`. In that case, it will pick the `MaxIntrinsicWidthModifier`, which overrides the minimum intrinsic width to match the maximum one (i.e: make it exact), and also fixes the content constraints to match the maximum intrinsic width possible for the incoming `maxHeight` constraint. Content constraints are used when measuring with an intrinsics modifier.

**Intrinsic.kt**

---

```
1 private object MaxIntrinsicWidthModifier : IntrinsicSizeModifier {
2     override fun MeasureScope.calculateContentConstraints(
3         measurable: Measurable,
4         constraints: Constraints
5     ): Constraints {
6         val width = measurable.maxIntrinsicWidth(constraints.maxHeight)
7         return Constraints.fixedWidth(width)
8     }
9
10    override fun IntrinsicMeasureScope.minIntrinsicWidth(
11        measurable: IntrinsicMeasurable,
12        height: Int
13    ) = measurable.maxIntrinsicWidth(height)
14 }
```

---

This is nice, but what we likely need here is to understand its ultimate effect from the end user perspective, for a good mental mapping. How does UI look like, or what is its behavior when we use this modifier? We can have a look to the `DropdownMenu` Composable for this, which relies on `DropdownMenuContent` in order to display the menu items within a `Column`:

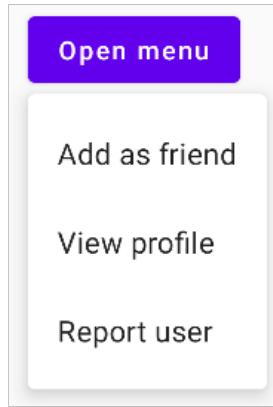
**DropDownMenuContent composable**

---

```
1 @Composable
2 fun DropdownMenuContent(...) {
3     ...
4     Column(
5         modifier = modifier
6             .padding(vertical = DropdownMenuVerticalPadding)
7             .width(IntrinsicSize.Max)
8             .verticalScroll(rememberScrollState()),
9         content = content
10    )
11    ...
12 }
```

---

It sets the preferred `Column` width to match the max intrinsic width of all its children (menu items). By doing this, it essentially enforces the dropdown menu to **match the width of its widest child**.



### Dropdown Menu

Feel free to give a read to the [official intrinsic measurement documentation](#)<sup>9</sup> for more examples of how intrinsics are used for real use cases in Android UI.

Let's do a dive into the layout Constraints now, since they became very relevant so far.

## Layout Constraints

Layout Constraints can come from a parent `LayoutNode` or a modifier. Layouts or layout modifiers use constraints to measure their children layouts. To choose constraints, a range in pixels is used by providing minimum and maximum values for width and height. The measured layouts (children) must fit within those constraints.

- `minWidth <= chosenWidth <= maxWidth`
- `minHeight <= chosenHeight <= maxHeight`

Most existing layouts either forward down unmodified constraints to their children, or with lose min constraints (set to 0). One example of the latter is `Box`, as we have learned when reading about measuring policies. We can showcase this with another scenario:

Sometimes a parent node or a modifier wants to ask its children for their preferred size. When that is the case, it can pass down infinite `maxWidth` or `maxHeight` constraints (i.e: `Constraints.Infinity`). When a child is defined to fill all the available space, passing unbounded constraints for that dimension is like a signal to let him decide what size to take. Imagine we have the following `Box`:

---

<sup>9</sup><https://developer.android.com/jetpack/compose/layouts/intrinsic-measurements>

**Box filling max height.**

---

```
1 Box(Modifier.fillMaxHeight()) {  
2     Text("test")  
3 }
```

---

By default, this Box will fill all the available height. But if we put it inside a `LazyColumn` (which measures children with infinite height constraints, since it is scrollable), the Box will instead wrap its content and take the height of the Text. That is because filling infinity does not make sense. Among the core layout components, it is very common for children to size themselves to wrap their content under these conditions. However, that ultimately depends on how the layout is defined.

`LazyColumn` can also work as a nice case study for learning how to use unbounded Constraints. It relies on a more generic layout called `LazyList` which uses subcomposition (`SubcomposeLayout`) for measuring children. Subcomposition is useful when we need to compose items lazily based on the available size. In this case, the screen size, since `LazyList` only composes the items visible on screen. Here is how the constraints for measuring its are created:

**LazyMeasuredItemProvider.kt**

---

```
1 // The main axis is not restricted  
2 val childConstraints = Constraints(  
3     maxWidth = if (isVertical) constraints.maxWidth else Constraints.Infinity,  
4     maxHeight = if (!isVertical) constraints.maxHeight else Constraints.Infinity  
5 )
```

---

Each time a child needs to be measured, these constraints are used. First, it will subcompose the item's content. This takes place during the `LazyList` measuring pass, which is possible thanks to `SubcomposeLayout`. Subcomposing the content results in a list of measurables for the visible children, which are then measured using the created `childConstraints`. At that point, children can basically pick their own height, since the height constraint is unbounded.

In the other hand, sometimes what we want is to set an exact size for all the children. When a parent or a modifier wants to impose an exact size, it enforces `minWidth == maxWidth`, and `minHeight == maxHeight`. That will basically enforce the child to fit in that exact space.

An example of this can be found in the `LazyVerticalGrid` Composable, which efficiently displays a vertical grid with a dynamic number of items. This Composable is very similar to `LazyColumn` or `LazyRow`, since it also composes its content lazily, and only the Composables visible on screen. When the number of cells in the grid is fixed, it actually uses a `LazyColumn`, and for each column it renders an `ItemRow` containing several items (as many as the span count establishes). The `ItemRow` layout measures each one of its children (columns) with fixed width, which is determined by the span count (number of columns), the column size, and the spacing in between the items.

**LazyGrid.kt**

---

```
1 val width = span * columnSize + remainderUsed + spacing * (span - 1)
2 measurable.measure(Constraints.fixedWidth(width))
```

---

This creates the following constraints, so `width` is fixed, and height ranges between `0` and `Infinity` (unbounded):

**Constraints.kt**

---

```
1 minWidth = width,
2 maxWidth = width,
3 minHeight = 0,
4 maxHeight = Infinity
```

---

If you want to explore more examples of diverse uses of `Constraints` I absolutely recommend inspecting the internals of the common layout modifiers for size, or any `Layouts` you feel familiar with from the Compose UI sources. I promise, it is a very didactic exercise to understand measuring.

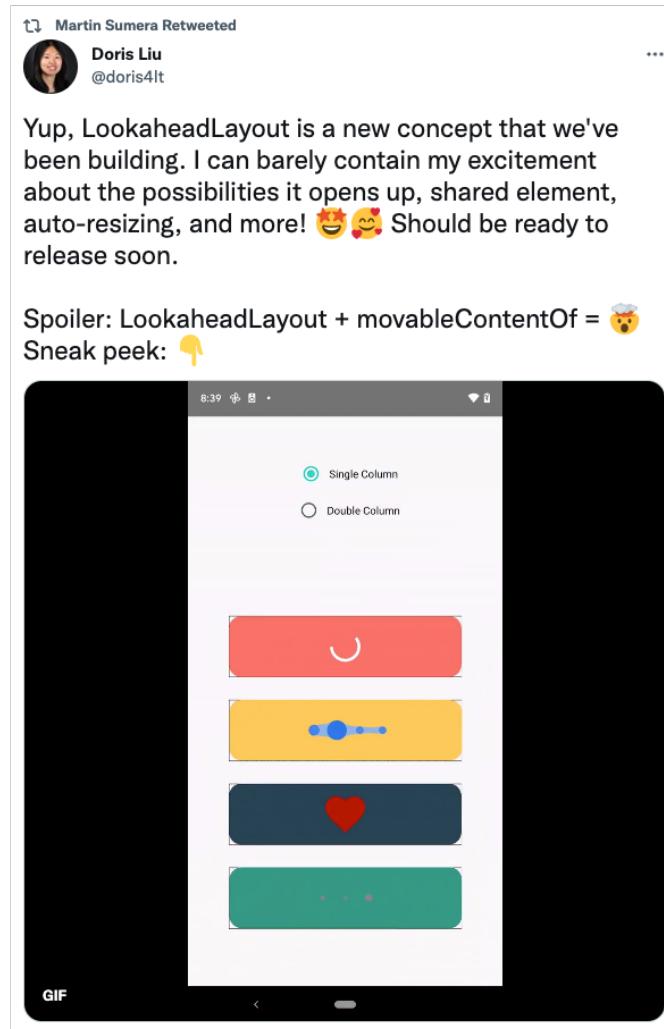
`Constraints` are modeled as an `inline` class that represents the 4 available constraints (i.e: `minWidth`, `minHeight`, `maxWidth`, and `maxHeight`) using a single `Long` value and a few bitmask for reading the different constraints from it.

## LookaheadLayout

It feels very natural to introduce `LookaheadLayout` here, since it is all about measuring and layout in Compose. You might remember [this tweet from Doris Liu<sup>10</sup>](#) where she introduced it and shared a cool animation showing some examples of shared element transitions:

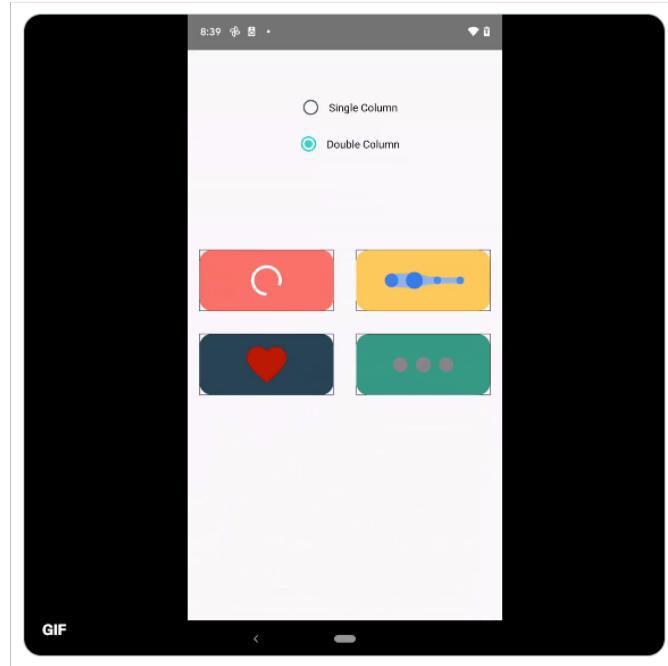
---

<sup>10</sup><https://twitter.com/doris4lt/status/1531364543305175041>



### LookaheadLayout

Sadly we cannot experience the animations in a static book, but those elements were animated when switching between two different screen states using the radio buttons from above. Here is how they look after transitioning to the double column style. The black rectangles represent the animation targets for size and position for each one of the shared elements.



### LookaheadLayout

Let's go over an example with actual code. Imagine a `SmartBox` Composable that is able to switch between `Row` and `Column` layouts based on a mutable state that toggles on click:

```
1  @Composable
2  fun SmartBox() {
3      var vertical by remember { mutableStateOf(false) }
4
5      Box(Modifier.clickable { vertical = !vertical }) {
6          if (vertical) {
7              Column {
8                  Text("Text 1")
9                  Text("Text 2")
10             }
11         } else {
12             Row {
13                 Text("Text 1")
14                 Text("Text 2")
15             }
16         }
17     }
18 }
```

Ideally, both texts would be **shared elements** between the two states, since they are exactly the same. It would also be great to have an easy way to animate them (they change instantly now). We might also want to use `movableContentOf11` to allow them to be reused and not lose their state during/after the animation.

A more ambitious version of the example from above would be to navigate between two Composable screens in an app based on a mutable state. The following example is a bit naive, since we would probably have more than just two screens in our app, but it will work for what is worth here:

```
1 @Composable
2 fun TvShowApp() {
3     var listScreen by remember { mutableStateOf(true) }
4
5     Box(Modifier.clickable { listScreen = !listScreen }) {
6         if (listScreen) {
7             CharacterList()
8         } else {
9             Detail()
10        }
11    }
12 }
```

These two screens could also contain some shared elements, so those are animated when we clicked on a character row within the `CharacterList`. Maybe the character image, or the whole character row. In this scenario, each sub-tree would likely be more complex than a couple texts, since it represents a complete screen. If we wanted to animate the transition, we would need to set some magic numbers as animation targets, only because we happen to know the final position and size of all the shared elements in the destination screen. But this is not great. Ideally, Compose UI should be able to pre-calculate and give us that information beforehand, so we could use it to set our animations targets. Here is where `LookaheadLayout` comes into play.

`LookaheadLayout` is able **pre-calculate the new measure and placement of its direct or indirect children** when they change. This gives each child the ability to observe the pre-calculated values from its measure/layout pass, and use those to reize/reposition itself in order to gradually change over time (which creates the animation effect). In the shared element transition from the example above, each shared element would observe what would be its final size and position in the screen it is transitioning to, and use those values to animate itself. Another example can be a morph animation.

## Yet another way of pre-calculating layouts

If we step back for a second, we could think of `LookaheadLayout` as yet another way of pre-calculating layouts in Jetpack Compose, along with `SubcomposeLayout` (subcomposition) and `intrinsics`. That said, there are important differences between the three that would be good to clear up front.

---

<sup>11</sup>[https://developer.android.com/reference/kotlin/androidx/compose/runtime/package-summary#movableContentOf\(kotlin.Function0\)](https://developer.android.com/reference/kotlin/androidx/compose/runtime/package-summary#movableContentOf(kotlin.Function0))

- **SubcomposeLayout**: Delays composition until measure time, so we can use the available space to determine what nodes/subtrees to build. It is **more about conditional composition than pre-layout**. It is also quite expensive, so I would not recommend using it for layout pre-calculation in any other scenario, since it is much more than a measure/layout pass.
- **Intrinsics**: They are quite more efficient than subcomposition, and work very similarly to LookaheadLayout internally. Both approaches invoke the measure lambdas provided by the user in LayoutModifiers or MeasurePolicies with different constraints in the same frame. But in the case of intrinsics, they are more of a **tentative calculation** in order to perform real measuring using the obtained values. Imagine a row with 3 children. In order to make its height match the height of the tallest child, it would need to get the intrinsic measures of all its children, and finally measure itself using the maximum one.
- **LookaheadLayout**: Used for precise pre-calculation of size and position of any (direct or indirect) child in order to power automatic animations. On top of measuring, LookaheadLayout also does a placement calculation based on the lookahead sizes. LookaheadLayout also does a more aggressive caching than intrinsics to **avoid looking ahead unless the tree has changed** (e.g: new layout nodes, modifier change, etc). To achieve this, LookaheadDelegate caches the lookahead measurement and placement in order to skip unnecessary re-calculations. Another difference with intrinsics is that there is an implicit guarantee in LookaheadLayout that the layout will finally arrive at the state calculated by the lookahead, so it does not permit users to manipulate the lookahead constraints.

## How it works

In practice, LookaheadLayout performs a lookahead pass of measure and layout before the “normal” measure/layout pass, so the latter can make use of the values pre-calculated during the former in order to update the node on every frame. This lookahead pass happens only when the tree changes or when layout changes as a result of a state change.

When the lookahead pass takes place, **layout animations are bypassed**, so measure and layout are performed as if animations would be finished already. In the future, all the layout animation APIs will get updated to be automatically skipped during lookahead passes, by design. This should work for any direct or indirect child of the LookaheadLayout.

To expose the pre-calculated data, LookaheadLayout executes its content lambda in a LookaheadLayoutScope, which gives access to a couple modifiers that its children can use:

- `Modifier.intermediateLayout`: Called whenever the layout with the modifier is remeasured. Gives access to the new lookahead (pre-calculated) size of the layout, and it is expected to produce an intermediate layout based on that (target) size.
- `Modifier.onPlaced`: Called on any re-layout of the layout with the modifier. Allows the child to adjust its own placement based on its parent. It provides access to lookahead coordinates for both the LookaheadLayout and the modifier itself (child of LookaheadLayout), which allows to calculate both the lookahead position and current position relative to the parent. These can be

saved and used from `intermediateLayout` when generating the intermediate layout in order to animate the layout position.

These two modifiers run during the normal measure/layout pass, so they can use the pre-calculated info provided in their lambdas to resize/reposition the layout towards the target values. The expectation is that users build custom modifiers on top of these two in order to create custom animations. Here is an example of a custom modifier to animate the constraints used to measure its children based on the lookahead size (extracted from the official sources):

```
1 fun Modifier.animateConstraints(lookaheadScope: LookaheadLayoutScope) =
2     composed {
3         var sizeAnimation: Animatable<IntSize, AnimationVector2D>? by remember {
4             mutableStateOf(null)
5         }
6         var targetSize: IntSize? by remember { mutableStateOf(null) }
7         LaunchedEffect(Unit) {
8             snapshotFlow { targetSize }.collect { target ->
9                 if (target != null && target != sizeAnimation?.targetValue) {
10                     sizeAnimation?.run {
11                         launch { animateTo(target) }
12                     } ?: Animatable(target, IntSize.VectorConverter).let {
13                         sizeAnimation = it
14                     }
15                 }
16             }
17         }
18     }
19     with(lookaheadScope) {
20         this@composed.intermediateLayout { measurable, _, lookaheadSize ->
21             targetSize = lookaheadSize
22             val (width, height) = sizeAnimation?.value ?: lookaheadSize
23             val animatedConstraints = Constraints.fixed(width, height)
24
25             val placeable = measurable.measure(animatedConstraints)
26             layout(placeable.width, placeable.height) {
27                 placeable.place(0, 0)
28             }
29         }
30     }
31 }
```

- A size animation is created to animate the size values (width/height). The animation will be restarted every time the layout changes (see `snapshotFlow`).

- This animation runs within a `LaunchedEffect` in order to avoid running uncontrolled side effects in the measure/layout phase (when this modifier runs).
- Pre-calculated `lookaheadSize` is available in the `intermediateLayout` lambda and set as the `targetSize` in order to trigger the animation when it changes.
- `lookaheadSize` is used to measure children so they gradually change their size. This is done by creating new fixed constraints that follow the size animation value **on every frame**, which creates the ultimate animation effect over time.

The lambda from `intermediateLayout` is skipped during the lookahead pass, since this modifier is used to produce intermediate states towards the lookahead one.

Once we have the custom modifier to animate constraints, we can use it from any `LookaheadLayout`. Here is an example, also extracted from the official sources:

```
1 LookaheadLayout(  
2     content = {  
3         var fullWidth by remember { mutableStateOf(false) }  
4         Row(  
5             (if (fullWidth) Modifier.fillMaxWidth() else Modifier.width(100.dp))  
6                 .height(200.dp)  
7                 .animateConstraints(this@LookaheadLayout)  
8                 .clickable { fullWidth = !fullWidth } ) {  
9                     Box(  
10                         Modifier  
11                             .weight(1f)  
12                             .fillMaxHeight()  
13                             .background(Color.Red)  
14                     )  
15                     Box(  
16                         Modifier  
17                             .weight(2f)  
18                             .fillMaxHeight()  
19                             .background(Color.Yellow)  
20                     )  
21                 }  
22             }  
23     ) { measurables, constraints ->  
24         val placeables = measurables.map { it.measure(constraints) }  
25         val maxWidth: Int = placeables.maxOf { it.width }  
26         val maxHeight = placeables.maxOf { it.height }
```

```

27
28     layout(maxWidth, maxHeight) {
29         placeables.forEach {
30             it.place(0, 0)
31         }
32     }
33 }
```

- Mutable state is used to toggle the Row between “full width” and “short width”.
- The Row layout changes are animated using the `animateConstraints` custom modifier from the previous snippet. Each time the mutable state is toggled, the Row width will change and that will trigger a new lookahead pre-calculation (and therefore also the animation).
- Max width and max height among all the children are used to measure the `LookaheadLayout` so all of them can fit inside.
- All the children are placed in 0,0.

And this produces a nice automatic animation any time the layout changes. This has been a good example of a resize animation based on the pre-calculated lookahead size. Now, what about the lookahead position?

As we mentioned earlier, there is also a `Modifier.onPlaced` available in the `LookaheadLayoutScope`. The `onPlaced` modifier exists for adjusting the placement (position) of the child based on its parent. It provides enough data to calculate both the lookahead position and current position of the layout relative to the parent. Then we can save those in variables and use them on subsequent calls to `intermediateLayout`, so we can also re-position the layout accordingly towards the lookahead values.

Here is an example of it, also extracted from the official sources. It animates the local position of the modified layout whenever the layout changes:

```

1 fun Modifier.animatePlacementInScope(lookaheadScope: LookaheadLayoutScope) =
2     composed {
3         var offsetAnimation: Animatable<IntOffset, AnimationVector2D>? by remember {
4             mutableStateOf(null)
5         }
6
7         var placementOffset: IntOffset by remember { mutableStateOf(IntOffset.Zero) }
8         var targetOffset: IntOffset? by remember { mutableStateOf(null) }
9         LaunchedEffect(Unit) {
10             snapshotFlow {
11                 targetOffset
12             }.collect { target ->
13                 if (target != null && target != offsetAnimation?.targetValue) {
```

```

14     offsetAnimation?.run {
15         launch { animateTo(target) }
16     } ?: Animatable(target, IntOffset.VectorConverter).let {
17         offsetAnimation = it
18     }
19 }
20 }
21 }
22
23 with(lookaheadScope) {
24     this@composed.onPlaced { lookaheadScopeCoordinates, layoutCoordinates ->
25         // the *target* position of this modifier in local coordinates.
26         targetOffset = lookaheadScopeCoordinates.localLookaheadPositionOf(
27             layoutCoordinates
28         ).round()
29
30         // the *current* position of this modifier in local coordinates.
31         placementOffset = lookaheadScopeCoordinates.localPositionOf(
32             layoutCoordinates, Offset.Zero
33         ).round()
34     }
35     .intermediateLayout { measurable, constraints, _ ->
36         val placeable = measurable.measure(constraints)
37         layout(placeable.width, placeable.height) {
38             val (x, y) = offsetAnimation?.run { value - placementOffset }
39             ?: (targetOffset!! - placementOffset)
40             placeable.place(x, y)
41         }
42     }
43 }
44 }
```

- An offset animation is used to animate the position. It will animate the offset relative to the parent.
- This animation runs within a `LaunchedEffect` once again to avoid running uncontrolled side effects in the measure/layout phase (when this modifier runs).
- `lookaheadScopeCoordinates` and `layoutCoordinates` are available in the `onPlaced` lambda. The former corresponds to the lookahead coordinates of the `LookaheadLayout` itself, and the latter provides the lookahead coordinates of this child layout. Those values are used to calculate the current and target offsets in local coordinates.
- The calculated offsets are ultimately used from `intermediateLayout` to update the layout position accordingly (see `placeable.place(x, y)`).

Once we have the custom modifier ready, we can use it for any (direct or indirect) children of a `LookaheadLayout`, exactly the same way as we did in the previous examples.

## Internals of `LookaheadLayout`

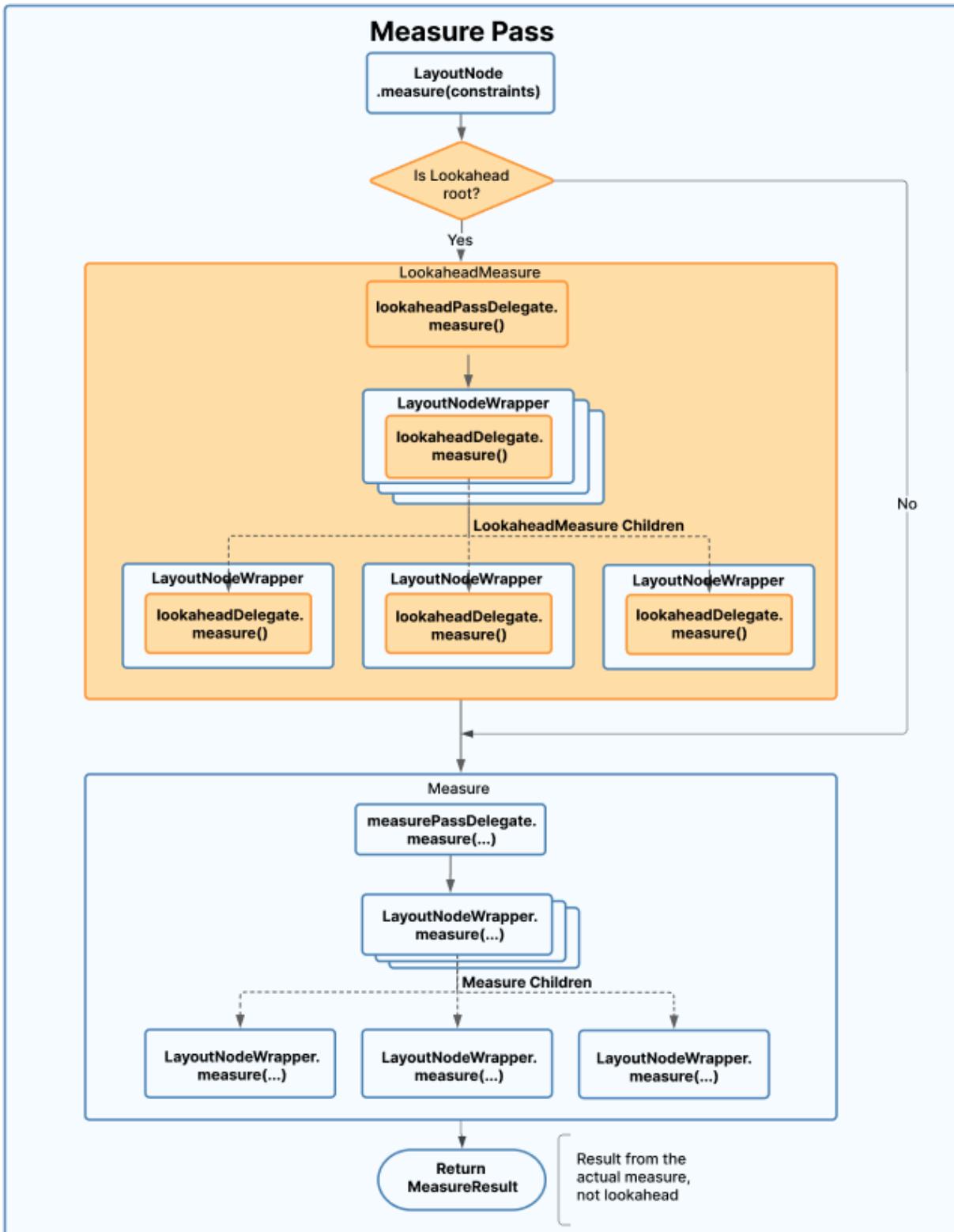
Now that we have a sense on what `LookaheadLayout` is and how it works, we are ready to learn about its internals. And the best way to do it is with some diagrams.

The diagrams we are going to show in this section have been facilitated by [Doris Liu<sup>12</sup>](#) from the Google Jetpack Compose team. They show what happens during the initial measure and layout passes, including lookahead measure and lookahead layout (in orange).

Let's start with the measure pass.

---

<sup>12</sup><https://twitter.com/doris4lt>



Lookahead measure pass

When a `LayoutNode` needs to be measured for the first time (e.g: it has just been attached) it will check if it is placed at the root of a `LookaheadLayout`, in order to start the lookahead measure pass first (the lookahead pass only runs for the `LookaheadLayout` subtree).

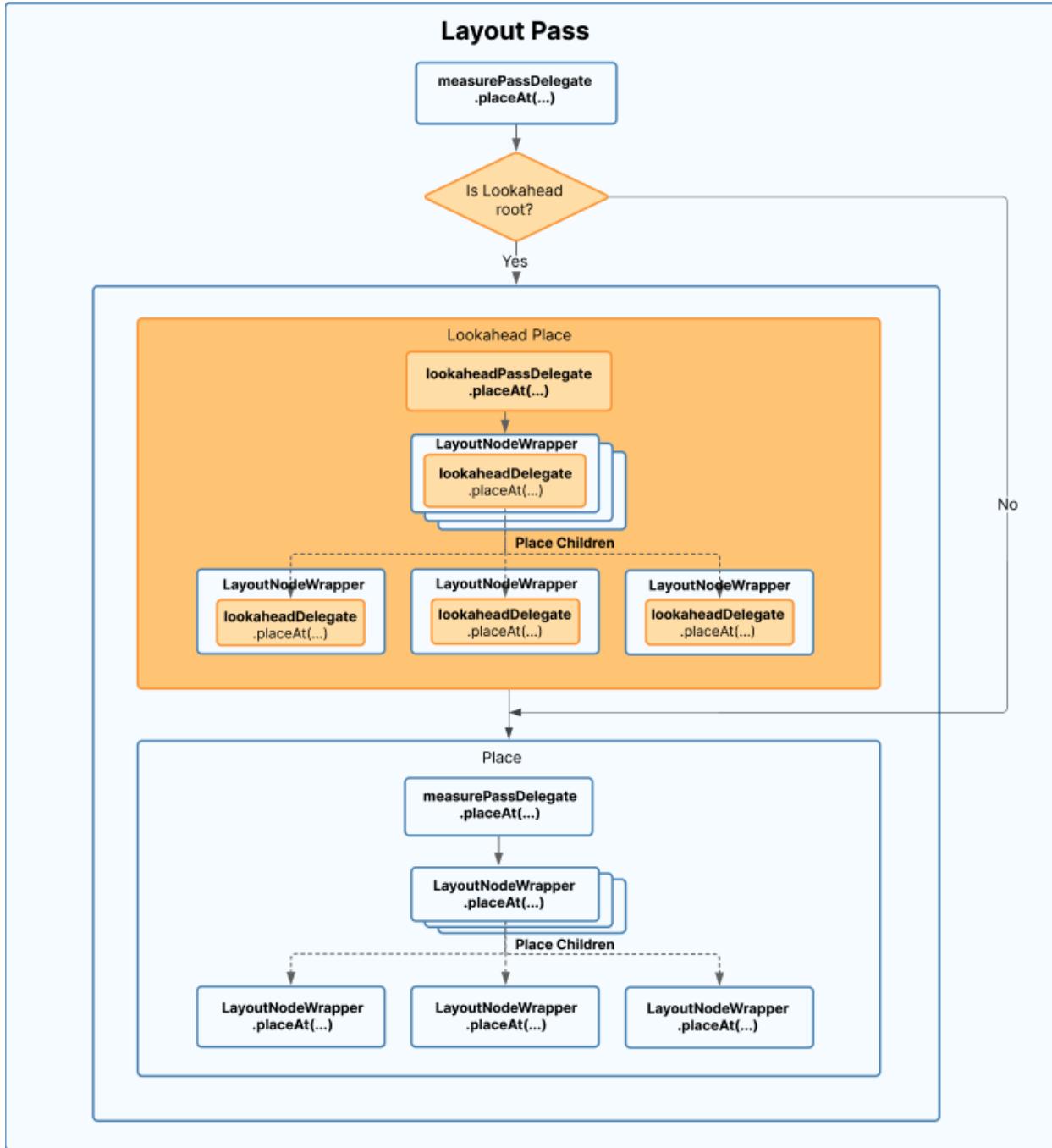
The `LayoutNode` calls `LookaheadPassDelegate#measure()` to start the lookahead measure pass. This delegate takes care of all the incoming lookahead measure/layout requests for the node. This call uses the outer `LayoutNodeWrapper` (normally used to measure the node) to run its lookahead measuring, via its `LookaheadDelegate`.

Looking back to the graphics about measuring in Compose, where we learned about the chain of `LayoutNodeWrappers`, we will remember that modifiers are also wrapped and chained, so the outer wrapper measures the current node, and wraps the first modifier wrapper, which wraps the second modifier wrapper, and so on. The last modifier in the chain wraps the inner `LayoutNodeWrapper`, which is used to measure children (via their outer wrapper). This is also what happens here, but **lookahead measure** is executed on all the steps via their `LookaheadDelegate`, instead of performing a “normal” measure.

Once the lookahead measure is done for the root node and all its direct or indirect children, or if the node is not at the root of the `LookaheadLayout`, the “normal” measure pass runs.

The process of measuring basically follows what we have described a second ago. The only difference is that now, the `MeasurePassDelegate` is used in all the steps instead of the lookahead one, since it is time to perform **real measuring**.

Let's give a look to the layout pass now.



Lookahead layout pass

The layout pass is not different at all. It is exactly the same we have already described for the measure pass, and the same delegates are used. The only difference is that in this case, `placeAt(...)` is called, in order to place the node and its children (for normal layout pass), or in order to calculate its lookahead position (for lookahead layout pass, in orange).

So far we have focused on nodes that are measured/laid out for the first time. In the other hand,

when a `LayoutNode` needs a remeasure/layout (e.g: its content has changed), the timing is a bit different. The invalidation for measure/layout and lookahead is optimized to reduce invalidation scope as much as possible. That way, `LayoutNodes` that aren't affected by a change in the tree will not be invalidated. This makes it entirely possible that only a (small) portion of the `LookaheadLayout` subtree gets invalidated due to changes to the tree.

## Some extra bits

When a new `LayoutNode` is attached, it inherits the out-most existing `LookaheadScope` from the tree, so all the direct or indirect children of a `LookaheadLayout` can share the same scope. This is because nested `LookaheadLayouts` are supported. Compose UI also ensures that a single lookahead pass for all of them.

`LookaheadLayout` can be combined with `movableContentOf` and `movableContentWithReceiverOf` in order to keep the state of the `Composables` while they are animated.

`LookAheadLayout` will be released in Compose 1.3.

## Modeling modifier chains

We have covered modifiers a little bit during this chapter, but they are obviously more than what we have learned. I couldn't miss the chance to give them a few sections to cover their internals a little bit, since they are a really important piece of Compose UI.

The `Modifier` interface models a collection of immutable elements that decorate or add behavior to UI `Composables`. `Modifier` is an abstraction that provides combination capabilities to chain any types of modifiers (`then`), folding capabilities for accumulating a value while traversing the chain (`foldIn` and `foldOut`), and some operations to check if `any` or `all` of the modifiers in the chain comply with a given predicate.

Whenever we find a chain of modifiers in a codebase like the following one, what we get is a “linked list” in return, referenced by the head of it: The `Modifier` type itself.

### Chaining modifiers

---

```
1 Box(  
2     modifier.then(indentMod) // Returns Modifier  
3         .fillMaxWidth() // Returns Modifier  
4         .height(targetThickness) // Returns Modifier  
5         .background(color = color) // Returns Modifier  
6     )
```

---

Note that chaining modifiers can be explicit or implicit, like in the snippet above, where we have a combination of the two. When `then` is not specified, it is because we are chaining those modifiers

via extension functions, which are actually calling `then` internally for us. Both approaches are equivalent. In real projects, extension functions are quite more frequent. Here is an example of a modifier declared via an extension function:

#### Padding.kt

---

```
1 @Stable
2 fun Modifier.padding(
3     horizontal: Dp = 0.dp,
4     vertical: Dp = 0.dp
5 ) = this.then(
6     PaddingModifier(
7         start = horizontal,
8         top = vertical,
9         end = horizontal,
10        bottom = vertical,
11        ...
12    )
13 )
```

---

When `then` is called to chain two modifiers, it results in a `CombinedModifier`, which is how the chaining is modeled. The `CombinedModifier` has a reference to the current modifier (outer), and a pointer to the next modifier in the chain (inner), which can also be a `CombinedModifier`.

#### Modifier.kt

---

```
1 class CombinedModifier(
2     private val outer: Modifier,
3     private val inner: Modifier
4 ): Modifier
```

---

The nodes are called `outer` and `inner` because the current node wraps the next one in a chain of `CombinedModifiers`:

#### Chaining modifiers

---

```
1 CombinedModifier(a, CombinedModifier(b, CombinedModifier(c, d)))
```

---

That is how modifier chains are modeled. But how are modifiers actually set to the `LayoutNode`? We are about to learn it.

## Setting modifiers to the LayoutNode

Any `LayoutNode` has a `Modifier` (or a chain of them) assigned. When we declare a `Layout`, one of the parameters passed to it by Compose UI is the update lambda:

## Layout.kt

---

```

1 @Composable inline fun Layout(
2     modifier: Modifier = Modifier,
3     measurePolicy: MeasurePolicy
4 ) {
5     val density = LocalDensity.current
6     val layoutDirection = LocalLayoutDirection.current
7     val viewConfiguration = LocalViewConfiguration.current
8
9     val materialized = currentComposer.materialize(modifier)
10
11    ReusableComposeNode<ComposeUiNode, Applier<Any>>(
12        factory = { LayoutNode() },
13        update = {
14            set(measurePolicy, { this.measurePolicy = it })
15            set(density, { this.density = it })
16            set(layoutDirection, { this.layoutDirection = it })
17            set(viewConfiguration, { this.viewConfiguration = it })
18            set(materialized, { this.modifier = it })
19        },
20    )
21 }

```

---

The update lambda is called as soon as the node is created via the factory one (right above it), and it initializes or updates the state of the LayoutNode. Here, things like measuring policy, density, layout direction, or view configuration are set. And of course also modifiers. In its final line, we see how a “materialized” modifier chain is set to the LayoutNode:

## Layout.kt

---

```

1 set(materialized, { this.modifier = it })

```

---

So the question now is: what is a “materialized” modifier? A few lines above we can find the place where the modifier gets “materialized” (right before emitting the node):

## Layout.kt

---

```

1 val materialized = currentComposer.materialize(modifier)

```

---

The `modifier` parameter here could be a single modifier or chain of them. If all the modifiers passed to the Layout are “standard”, this function simply returns them unmodified, so they can be set to the LayoutNode without any additional processing. So that is how “standard” or “normal” modifiers

are set. But wait a minute. “Standard” modifiers? Are there any other existing types of modifiers in Compose UI?

Indeed, there is a second type: The composed modifiers. If we look back to some sections ago, we said that modifiers are stateless, and therefore they need to get wrapped in order to hold their state. Indeed, that is true, **for “standard” modifiers**. But turns out that composed modifiers are actually a special type of modifiers that are **stateful**. This makes sense when a Composition is needed for implementing them. An example of this is when we need to remember something from the modifier logic. Or when the modifier needs to read from a `CompositionLocal`. Those things would never be possible without running the modifier lambda in the context of a Composition.

Composed modifiers are **composed for each element they modify**. If a Layout gets one or multiple composed modifiers passed as part of their modifier chain, they will need to be composed via their Composable factory function first, before getting assigned to the `LayoutNode`:

#### ComposedModifier.kt

```
1 private open class ComposedModifier(
2     inspectorInfo: InspectorInfo.() -> Unit,
3     val factory: @Composable Modifier.() -> Modifier
4 ) : Modifier.Element, InspectorValueInfo(inspectorInfo)
```

Running the `factory` Composable lambda first is required because `LayoutNodes` don't know how to work with composed modifiers, so they need to be converted into regular ones before assigning them. The `factory` lambda will get executed in the context of a Composition every time the node is used, and that will unlock access to any Composable functions within the block. The `factory` lambda is what we use to write the composed modifier.

Let's learn this with an example: The `clickable` modifier. Also note how the `Modifier.composed()` extension function is used to create stateful (composed) modifiers:

#### Clickable.kt

```
1 fun Modifier.clickable(
2     interactionSource: MutableInteractionSource,
3     indication: Indication?,
4     enabled: Boolean = true,
5     onClick: () -> Unit
6     ...
7 ) = composed(
8     factory = {
9         val onClickState = rememberUpdatedState(onClick)
10        val pressedInteraction = remember { mutableStateOf<PressInteraction.Press?>(null\)
11    } }
12     ...
13     val isClickableInScrollableContainer = remember { mutableStateOf(true) }
```

```
14     val delayPressInteraction = rememberUpdatedState {
15         isClickableInScrollableContainer.value || isRootInScrollableContainer()
16     }
17     val gesture = Modifier.pointerInput(interactionSource, enabled) {
18         ...
19     }
20     Modifier
21         .then(...adds more extra modifiers)
22         .genericClickableWithoutGesture(
23             gestureModifiers = gesture,
24             ...
25         )
26     },
27     inspectorInfo = ...append info about the modifier for dev tooling
28 )
```

---

Since a Composition is required to be able to remember state within the block, a composed modifier is used.

Apart from the `clickable` modifier, there are many other examples of composed modifiers. Modifiers like `focusable`, `scroll`, `swipeable`, `border`, `selectable`, `pointerInput`, `draggable`, `toggleable`, and many more. For more examples of this, I recommend inspecting the sources and searching for usages of the `Modifier.composed()` extension function.

Sometimes Compose UI needs to create an ad-hoc `LayoutNode` for some purpose, and it also uses the chance to set modifiers to it. An example of this is the root `LayoutNode` for a given `Owner`, like `AndroidComposeView`:

#### AndroidComposeView.android.kt

---

```
1 override val root = LayoutNode().also {
2     it.measurePolicy = RootMeasurePolicy
3     it.modifier = Modifier
4         .then(semanticsModifier)
5         .then(_focusManager.modifier)
6         .then(keyInputModifier)
7     it.density = density
8 }
```

---

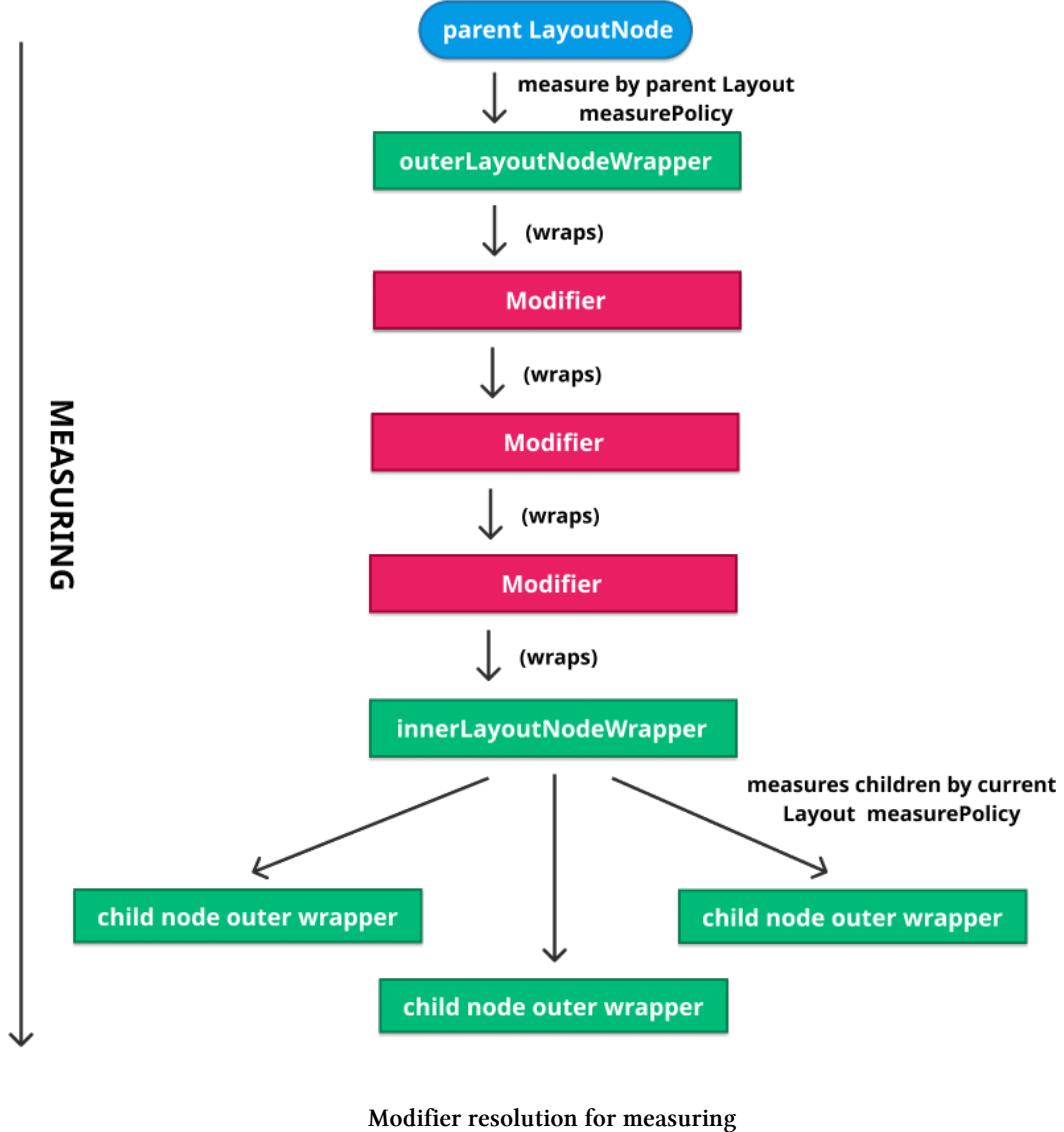
When `AndroidComposeView` hooks its root `LayoutNode`, it uses the chance to set the measure policy for it, its density, and a few modifiers for hooking accessibility through the `View` system. Those set the default configuration for the semantic tree, a manager for handling focus for accessibility, and key input handling, which is also needed for it.

## How LayoutNode ingests new modifiers

Whenever a modifier (or a chain of modifiers) is set to a `LayoutNode`, all the modifiers already set for the current node are saved into a cache. This cache is used to find modifiers that can be reused for the new modifier chain being set. Modifiers will be reused in case they match the modifier instance being set (more about this later).

All modifiers in the cache are initialized as not reusable. Then, the node folds over the new chain of modifiers, starting from its head. This is done via the `Modifier#foldIn` function, one of the `Modifier` folding capabilities. For each modifier, it will check if the cache contains an equivalent one, and if so, it will mark it as reusable. Then, if the cached modifier has a parent, it will crawl up the hierarchy for flagging it and all its ancestors as reusable also. This will complete an initial pre-processing of the cached modifiers, so they are ready for the next step.

After this, the `LayoutNode` proceeds to rebuild its outer `LayoutNodeWrapper`. The outer wrapper wraps the first modifier in the chain, the first modifier wraps the second, and so on. Since we are setting a new chain of modifiers, the outer wrapper needs to be rebuilt.



To rebuild the outer wrapper, it folds over the new chain of modifiers again, but in the opposite direction: From the tail to its head. That is done via the `Modifier#foldOut` function. The goal now is to build up (aggregate) the new chain of `LayoutNodeWrappers` for the new modifiers.

Since we want to start from the tail, the initial value for folding will be the `innerLayoutNodeWrapper` (bottom most of the diagram). Then we can crawl up to go through all the modifiers set in this `LayoutNode`.

For each modifier, it will first check if it can reuse it by checking the cache, and it will do so if that's the case. If it reuses it, it will chain it and remove it from the cache. In case it cannot reuse it, it will wrap the modifier using an appropriate `LayoutNodeWrapper` type for the specific modifier type, and then chain it. There are multiple wrapper types, and each one has its own capabilities depending on the nature of the modifier.

When the folding reaches the head (i.e: the `outerLayoutNodeWrapper`), the outer wrapper is assigned as the inner wrapper for the parent node, as we can see at the top-most of the diagram. And this completes the process of building up the new wrapper chain.

The next step is to detach all the remaining modifiers from the cache, since they will not be reused, and follow up by calling `attach()` on all the new wrappers. Finally, the cache can be cleared, and all the modifiers can be redrawn. To do it, they get invalidated.

The last step is to request a remeasure of the parent in case it is required.

## Drawing the node tree

We have talked a lot about measuring and layout. But only a little bit about drawing. For drawing, the `LayoutNodeWrapper` chain is also followed, so that the current node gets drawn first, then its modifiers (in order), and finally its children. This process repeats itself for every child until the complete node hierarchy is drawn. But let's step back to go in more detail about it.

In the Android system, the draw pass takes place after measuring and layout. In every drawing tick, the system calls `draw` on our `View` or our `ViewGroup`, and it performs the drawing. In Compose that is not different. When a `LayoutNode` requests a remeasure, it gets flagged as dirty, and in the next drawing pass, the `Owner`, (a `ViewGroup` like `AndroidComposeView`) will remeasure and relayout all the dirty nodes, and then proceed to do the drawing. Drawing takes place in the `dispatchDraw` function, which is where `Android ViewGroups` draw their children.

Let's stick with `AndroidComposeView` as an example of an `Owner`. Right before drawing, it invalidates the drawing layers of all the `LayoutNodes` in the hierarchy. Which means they are flagged as needing to be redrawn. That also includes all the modifier wrappers, that can also be used for drawing. Whenever a drawing layer for a node needs to get invalidated, it will first check that it is not drawing already.

The drawing process is kicked off starting from the root `LayoutNode`. This is done via `root.draw(canvas)`, which will be delegated to the node's `outerLayoutNodeWrapper`.

The `Canvas` used by Compose UI is an abstraction, since **Compose UI is a multiplatform library**. For Android, it delegates all the drawing to the native `Canvas`. On top of this, the Compose `Canvas` offers a more ergonomic api surface than the native one. One key differences between both is that the Compose `Canvas` functions do not accept a `Paint` object anymore, since allocating `Paint` instances is quite expensive in Android, and particularly not recommended during `draw` calls. Instead of doing that, the team decided to rework the api surface so functions create and reuse the same `Paint` implicitly.

Each `LayoutNodeWrapper` has its own drawing layer, and the outer wrapper is not an exception. When drawing is dispatched to the root node via `root.draw(canvas)` the drawing layer for the outer wrapper does the job. There are different scenarios to contemplate here:

1. If there is a drawing layer associated to the wrapper, it will dispatch drawing on it. That will effectively draw the node, which can be a `LayoutNode`, or a modifier.
2. If there is not a drawing layer associated to this wrapper, it checks if there are any draw modifiers associated. If so, it draws all of them. (Draw modifiers are attached to the wrapper as a linked list. They are not wrapped by their own separate wrappers as other modifiers do).
3. If there is not a drawing layer associated to the wrapper neither any draw modifiers, it proceeds to call `draw` over the next wrapper in the chain, to continue with the process of drawing the complete `LayoutNode` hierarchy.

Step 1 is where drawing the node takes place. For this matter, Compose UI provides two types of drawing layers for a `LayoutNode`:

- `RenderNodeLayer`: For drawing `RenderNodes`. This is the default for rendering Jetpack Compose UI nodes. `RenderNodes` are a tool for more efficient hardware driven drawing. They allow to draw once, then redraw multiple times very efficiently.
- `ViewLayer`: This one is a `View` based implementation. It is only used as a fallback when `RenderNodes` can't be used directly. This implementation is more “hacky” than the `RenderNodeLayer`, since it treats `Views` as `RenderNodes`, and to do that it requires some piping. But `Views` are much more than `RenderNodes`, indeed.

Both implementations are hardware accelerated, since both end up relying in `RenderNodes`, directly or indirectly. Both could be used as valid implementations for `Modifier.graphicsLayer`, for example.

Drawing layers are created whenever a new node is attached, and the decision on what type of layer to use relies on the `Owner`. In the case of `AndroidComposeView`, it will always prioritize `RenderNodeLayers` if the platform version supports it, and fallback to `ViewLayers` otherwise.

Different Android versions provide different APIs for `RenderNodes`, so the `RenderNodeLayer` relies on an abstraction provided by Compose UI that delegates drawing to the corresponding `RenderNode` system implementation at the time of executing.

In the other hand, `ViewLayer` relies on `Views` to do the drawing. Any `ViewLayer` has a container `View` associated that will use to trigger the drawing. This container is usually the `Owner`, a `ViewGroup`. When a `ViewLayer` is created, it assigns itself as a **child of its container**. This is only possible because `ViewLayer` is actually implemented as a `View`, which is the actual trick here. When the layer needs to draw, it will call its container `ViewGroup#drawChild` function, so drawing the child (layer) will just reuse the `View` drawing machinery. As we stated earlier, this is more hacky than simply using `RenderNodes` directly, since it involves `Views` to make it work.

Before drawing, a `ViewLayer` has the chance to enable “elevation mode”, in case it has some elevation assigned. “Elevation mode” is supported by the Android `Canvas`, and it allows rendering shadows, and rearranging layers with different elevations based on those. Once “elevation mode” is enabled, the layer can proceed to draw, and after that, it will disable it.

Once all the wrappers in the chain are drawn, `AndroidComposeView#dispatchDraw` notifies all the layers flagged as dirty to update their display lists, which updates the drawing on the current canvas.

Layers are flagged as dirty when they get invalidated. Since we invalidated all layers at the beginning of the `dispatchDraw` function, this will enforce all layers for all the nodes in the hierarchy to update.

The previous operations to update the display list can result in the invalidation of the same layers. In those cases, those layers will be flagged as dirty at this point, so they can be updated in the next pass.

Sometimes we read snapshot state when drawing a `LayoutNode` into a layer. Whenever this state changes, we'd likely need to redraw the node for coherence. The root has a layer modifier that observes state changes in all the children nodes in order to invalidate their drawing layers accordingly. That is part of the measure policy set for the root node:

`AndroidComposeView.android.kt`

```
1 override val root = LayoutNode().also {  
2     it.measurePolicy = RootMeasurePolicy  
3     it.modifier = Modifier  
4         .then(semanticsModifier)  
5         .then(_focusManager.modifier)  
6         .then(keyInputModifier)  
7     it.density = density  
8 }
```

This `RootMeasurePolicy` measures and places children attached to the root node. For doing so, it calls `placeable.placeRelativeWithLayer(0, 0)`, which places the measured child in coordinates  $(0, 0)$ , and **introduces a graphic layer**. This graphic layer is configured to automatically observe and react to any snapshot state changes.

## Semantics in Jetpack Compose

Probably the best way to start this chapter is with a brief summary about semantics. In Jetpack Compose, Composition is a tree that describes our UI. Paralelly, there is another tree that describes UI in an alternative way that the accessibility services and the testing framework can understand. The nodes on this tree provide relevant metadata about their semantic meaning.

In previous sections we learned that the `Owner` of a `LayoutNode` hierarchy has an accessibility delegate to pipe semantics through the Android SDK. Then, every time a node is attached, detached, or its semantics are updated, the semantic trees are notified via the `Owner`.

When the `Owner` is created (for `AndroidComposeView` that is whenever `setContent` is called), it creates an ad-hoc root `LayoutNode` with some default modifiers set:

**AndroidComposeView.android.kt**

---

```
1 override val root = LayoutNode().also {
2     it.measurePolicy = RootMeasurePolicy
3     it.modifier = Modifier
4         .then(semanticsModifier)
5         .then(_focusManager.modifier)
6         .then(keyInputModifier)
7     it.density = density
8 }
```

---

These modifiers are relevant for accessibility and semantics. The semantics modifier adds the core semantics with default configuration to the root node in order to start building up the semantics tree. On top of it, the focus manager modifier sets the root modifier for handling focus across the Composable hierarchy. This manager takes care of setting and moving focus across our Composables as required. Finally, the key input modifier handles keyboard input and pipes any KeyDown events to the focus manager, so focus can also be handled using a keyboard. The latter two modifiers are very important for accessibility also.

For adding semantics to other nodes different than the root, we do it via the `Modifier.semantics` function. Every semantics modifier we set to a `Layout` contains an `id`, and some semantics configuration:

**SemanticsModifier.kt**

---

```
1 fun Modifier.semantics(
2     mergeDescendants: Boolean = false,
3     properties: (SemanticsPropertyReceiver.() -> Unit)
4 ): Modifier = composed(
5     inspectorInfo = debugInspectorInfo {
6         name = "semantics"
7         this.properties["mergeDescendants"] = mergeDescendants
8         this.properties["properties"] = properties
9     }
10 ) {
11     val id = remember { SemanticsModifierCore.generateSemanticsId() }
12     SemanticsModifierCore(id, mergeDescendants, clearAndSetSemantics = false, properties)
13 }
```

---

Here, we can see how some debug inspector information is implicitly created for providing some details about this node to the Compose tools, so that they can show it while inspecting the Composable tree.

An id is automatically generated and remembered. These ids are unique across LayoutNode hierarchies (shared statically), and generated sequentially. Each new generated id is bigger than the previous one. That means that a new Composition might start generating ids from the very beginning, or sequentially after the latest id generated by any other Composition previously created.

Finally, the modifier is created using those elements and the provided configuration parameters.

Right now, the semantics modifier is implemented as a composed modifier, since it needs access to the Composition context in order to use remember for the generated id in the block. That said, this implementation will change soon, since there is an ongoing refactor that will move the id to be generated when creating the corresponding LayoutNode instead, so remember will be needed anymore, and therefore the need for a Composition context will be removed.

This implementation of the semantics modifier is getting a refactor

When AndroidComposeView assigns the default semantics modifier to the root node, it creates it the same way:

AndroidComposeView.android.kt

```
1 private val semanticsModifier = SemanticsModifierCore(  
2     id = SemanticsModifierCore.generateSemanticsId(),  
3     mergeDescendants = false,  
4     clearAndSetSemantics = false,  
5     properties = {}  
6 )
```

When we use Composables provided by the material or foundation libraries, it is very likely that those already hook their semantics implicitly. This is nice and welcome, but it will not happen when we work in our own custom layouts. For this reason, it is important that every time we write a new Layout in Jetpack Compose, we also provide its semantics. Accessibility and testing **must be a priority**.

## Notifying about semantic changes

Something we have said a couple times already is how any semantic changes are notified to the Android SDK via the Owner. Let's learn a bit more about this.

The AndroidX Core library added an AccessibilityDelegateCompat for normalizing working with accessibility services across system versions. The Owner of a LayoutNode hierarchy uses an implementation of this delegate to handle accessibility changes. This implementation makes use of the system accessibility services that it can retrieve via the Android Context obtained from the Owner.

When a semantics change is notified through the `Owner`, an action to check for semantic changes on the semantic tree is posted to the main looper via a native `Handler`. This action does the following things in this order:

1. Compares the old and new semantic trees looking for structural changes. That is: children added or removed. When structural changes are detected, the delegate uses a conflated `Channel` to notify about those. That is because the code that notifies the system accessibility services is a suspend function **running in a coroutine**. This task is looping over the entire lifetime of the Compose instance, consuming recent layout changes and sending events to the accessibility framework in batches (every 100ms). The appropriate way to produce events for this task to consume is a `Channel`.
2. Compares the old and new semantic trees looking for changes in the semantic properties. When semantic properties change, it notifies the accessibility services using the native machinery (`ViewParent#requestSendAccessibilityEvent`).
3. Updates the list of previous semantic nodes with all the current semantic nodes.

## Merged and unmerged semantic trees

Jetpack Compose provides two semantic trees: **Merged** and **unmerged**. Sometimes we need to merge a set of Composables semantically, since it has better semantic meaning for an end user to experience them as a group than separately. Imagine that accessibility tools like TalkBack read each single minor Composable within each single row in a very big list of items. It would yield a highly exhausting and very useless user experience. We would likely prefer it to read each row altogether.

Merging is done via the `mergeDescendants` property we can read in the snippet from above. This property allows us to decide when a semantics node (i.e: for a Composable or a modifier) needs to merge its descendants into it. This is actually done for many foundation and material Composables and modifiers out of the box. The merged tree will perform the merging based on the `mergeDescendants` we can set the nodes. The unmerged one, will not do it. It will keep those nodes separate. Tools decide what tree they want to consume.

Here is how merging takes place internally:

Semantic properties have a merge policy assigned. Let's use `contentDescription` as an example:

### MyButton.kt

---

```
1 MyButton(  
2     modifier = Modifier.semantics { contentDescription = "Add to favorites" }  
3 )
```

---

Semantic properties are defined via extension properties over the `SemanticsPropertyReceiver`, which is the scope used for `semantics` blocks that allows us to set any properties within the block.

**SemanticProperties.kt**

---

```
1 interface SemanticsPropertyReceiver {
2     operator fun <T> set(key: SemanticsPropertyKey<T>, value: T)
3 }
```

---

For setting a semantic property, you need two things: A semantic key, and a value. Semantic keys are defined in a type safe way, and require a property name, and a merge policy to be created:

**SemanticProperties.kt**

---

```
1 val ContentDescription = SemanticsPropertyKey<List<String>>(
2     name = "ContentDescription",
3     mergePolicy = { parentValue, childValue ->
4         parentValue?.toMutableList()?.also { it.addAll(childValue) } ?: childValue
5     }
6 )
```

---

The merge policy lambda is where the property decides how to merge its descendants. In this case, the ContentDescription adds all descendant ContentDescription values to a list, but could do whatever it wanted. The default merge policy for semantic properties in Compose is actually no merging at all. It keeps the parent value if available for a node, and discards the child.

Note that this book is focused on internals only, so its purpose is not to list all the available configuration options or APIs for semantics. For more details about Jetpack Compose semantics from the API surface and end user perspective, I definitely recommend reading the [really detailed official documentation<sup>13</sup>](#).

---

<sup>13</sup><https://developer.android.com/jetpack/compose/semantics>

# 5. State snapshot system

Jetpack Compose has a particular way to represent state and propagate state changes which drives the ultimate reactive experience: The state snapshot system. This reactive model enables our code to be more powerful and concise, since it allows components to recompose automatically based on their inputs and only when required, avoiding all the boilerplate we'd need if we had to notify those changes manually (as we have been doing with the Android View system in the past).

Let's start this chapter by introducing the term "snapshot state".

## What snapshot state is

Snapshot state refers to **isolated state that can be remembered and observed for changes**. Snapshot state is what we get when calling functions like `mutableStateOf`, `mutableStateListOf`, `mutableStateMapOf`, `derivedStateOf`, `produceState`, `collectAsState`, or any of the like. All those calls return some type of State, and devs frequently refer to it as snapshot state.

Snapshots state is named like that since it is part of the state snapshot system defined by the Jetpack Compose runtime. This system models and coordinates state changes and change propagation. It is written in a decoupled way, so it could theoretically be used by other libraries that want to rely on observable state.

Regarding change propagation, one of the things we learned in chapter 2 was that all Composable declarations and expressions are wrapped by the Jetpack Compose compiler to **automatically track any snapshot state reads within their bodies**. That is how snapshot state is (automatically) observed. The goal is that every time the state the Composable reads varies, the runtime can invalidate the Composable's `RecomposeScope`, so it is executed again in next recomposition (recomposed).

This is infrastructure code provided by Compose that is therefore not needed in any client codebases. Clients of the runtime like Compose UI can be completely agnostic of how invalidation and state propagation is done, or how recomposition is triggered, and only focus on providing the building blocks that work with that state: The Composable functions.

But snapshot state is not only about automatically notifying changes to trigger recomposition. The word snapshot is part of the name for a very important reason: **state isolation**. That stands for the level of isolation we apply in the context of concurrency.

Imagine handling mutable state across threads. It can rapidly become a mess. Strict coordination and synchronization is required to ensure the state integrity, since it can be read and/or modified from different threads at the same time. This opens the door to collisions, hard to detect bugs, and race conditions.

Traditionally, programming languages have dealt with this in different ways, one of them being immutability. Immutable data can never be modified after created, which makes it completely safe in concurrent scenarios. Another valid approach can be the actor system. This system focuses on **state isolation** across threads. Actors keep their own copy of the state, and communication / coordination is achieved via messages. There needs to exist some coordination to keep the global program state coherent in case this state is mutable. The Compose snapshot system is not based on the actor system, but it is actually closer to this approach.

Jetpack Compose leverages mutable state, so Composable functions can automatically react to state updates. The library wouldn't make sense with immutable state only. This means that it needs to solve the problem of shared state in concurrent scenarios, since composition can be possible in multiple threads (remember chapter 1). The Compose approach to this is the state snapshot system, and it is based on state isolation and later change propagation to allow **working with mutable state safely across threads**.

The snapshot state system is modeled using a [concurrency control system<sup>14</sup>](#), since it needs to **coordinate state across threads** in a safe manner. Shared mutable state in concurrent environments is not an easy deal, and it is a generic problem agnostic of the actual use case for the library. We are diving into concurrency control systems in detail and how Compose makes use of them in the following section.

Before completing this introduction, it can be useful to peek into the `State` interface, which any snapshot state object implements. Here is how it looks in code:

#### SnapshotState.kt

---

```
1 @Stable
2 interface State<out T> {
3     val value: T
4 }
```

---

This contract is flagged as `@Stable`, since Jetpack Compose provides and uses stable implementations only (by design). Recapping a bit, this means that any implementation of this interface **must** ensure that:

- The result of `equals` between two `States` is coherent: it always returns the same result when comparing the same two instances.
- When a public property of the type changes (`value`), composition is notified.
- All its public property types are also stable (`value`).

These properties really represent what snapshot state is. In the following sections we will learn how every time a snapshot state object is written (modified), Composition is notified, as one of the mentioned rules require.

---

<sup>14</sup>[https://en.wikipedia.org/wiki/Concurrency\\_control](https://en.wikipedia.org/wiki/Concurrency_control)

Make sure to give a read to this post by Zach Klipp<sup>15</sup> introducing some of these ideas. I highly recommend that post.

Let's learn a bit about concurrency control systems now. It will help us a lot to easily understand why the Jetpack Compose state snapshot system is modeled the way it is.

## Concurrency control systems

The state snapshot system is implemented following a concurrency control system, so let's introduce this concept first.

In computer science, “concurrency control” is about ensuring correct results for concurrent operations, which means coordination and synchronization. Concurrency control is represented by a series of rules that ensure the correctness of the system as a whole. But this coordination always comes with a cost. Coordination usually impacts performance, so the key challenge is to design an approach that is as efficient as possible without significant drops in performance.

One example of concurrency control is the transaction system present in most of the database management systems (DBMS) today. Concurrency control in this context ensures that any database transactions performed in concurrent environments are done in a safe manner without violating the data integrity of the database. The aim is to maintain correctness. The term “safety” here covers things like ensuring that transactions are atomic, that they can be reverted safely, that no effect of a committed transaction is ever lost, and that no effect of an aborted transaction remains in the database. It can be a complex problem to solve.

Concurrency control is not only frequent in DBMS but also in other scenarios like programming languages, where it is used to implement transactional memory, for instance. That is actually the use case for the state snapshot system. Transactional memory attempts to simplify concurrent programming by allowing a group of load and store operations to execute in an atomic way. Actually, in the Compose state snapshot system, state writes are applied as a single atomic operation when state changes from a snapshot are propagated to other snapshots. Grouping operations like this simplifies coordination between concurrent reads and writes of shared data in parallel systems / processes. On top of this, atomic changes can be easily aborted, reverted, or reproduced. –I.e: having history of reproducible changes to potentially reproduce any version of the program state.–

There are different categories of concurrency control systems:

- Optimistic: Do not block any reads or writes and be optimistic about those being safe, then abort a transaction to prevent the violation if it will break the required rules when committed. An aborted transaction is immediately re-executed which implies an overhead. This one can be a good strategy when the average amount of aborted transactions is not too high.
- Pessimistic: Block an operation from a transaction if it violates the rules, until the possibility of violation disappears.

---

<sup>15</sup><https://dev.to/zachklipp/a-historical-introduction-to-the-compose-reactive-state-model-19j8>

- Semi-optimistic: This is a mix of the other two, a hybrid solution. Block operations only in some situations and be optimistic (then abort on commit) for others.

Performance for each category can differ based on factors like the average transaction completion rates (throughput), level of parallelism required, and other factors like the possibility of deadlocks. Non-optimistic categories are considerably more prone to deadlocks, which are often resolved by aborting a stalled transaction (hence release the others) and restarting it as soon as possible.

Jetpack Compose is **optimistic**. State update collisions are only reported when propagating the changes (in the end), and then they are tried to be merged automatically or discarded (changes aborted) otherwise. More on this later.

The Jetpack Compose approach to concurrency control systems is simpler to the ones we can find on DBMS for example. It is only used to maintain correctness. Other features that can be found in database transactions like being recoverable, durable, distributed or replicated are not true for the Compose state snapshot system. (They don't have the "D" part of "[ACID<sup>16</sup>](#)"). Even though, Compose snapshots are in-memory, in-process only. They are atomic, consistent, and isolated.

In conjunction with the different categories of concurrency control listed (optimistic, pessimistic, semi-optimistic), there are some types that can also be used, one of them being the **Multiversion concurrency control (MVCC)**: That is the one Jetpack Compose uses to implement the state snapshot system. This system increases concurrency and performance by **generating a new version of a database object each time it is written. It also allows reading the several last relevant versions of the object**.

Let's describe this in depth and also explain its purpose.

## Multiversion concurrency control (MCC or MVCC)

The Compose global state is shared across Compositions, which also means **threads**. Composable functions should be able to run concurrently (the door for parallel recomposition is always open). If they execute in parallel, they can read or modify snapshot state concurrently, so state isolation is going to be needed.

One of the main properties of concurrency control is actually **isolation**. This property ensures correctness in scenarios of concurrent access to data. The simplest way to achieve isolation is to block all readers until writers are done, but that can be awful in terms of performance. MVCC (and therefore Compose) does better than that.

To achieve isolation, MVCC keeps **multiple copies** of the data (snapshots), so each thread can work with an isolated snapshot of the state at a given instant. We can understand those as different **versions** of the state ("multiversion"). Modifications done by a thread remain invisible to other threads until all the local changes are completed and propagated.

---

<sup>16</sup><https://en.wikipedia.org/wiki/ACID>

In a concurrency control system this technique is called “snapshot isolation”, and it is defined as the isolation level used to determine which version each “transaction” (snapshot in this use case) sees.

MVCC also leverages immutability, so whenever data is written a new copy of the data is created, instead of modifying the original one. This leads to having **multiple versions of the same data stored in memory**, like a history of all the changes over the object. In Compose these are called “state records”, and we are going over those in detail in a few sections.

Another particularity of MVCC is that it creates **point-in-time consistent views** of the state. This is usually a property of backup files, and means that all references to objects on a given backup stay coherent. In MVCC, this is often ensured via a transaction ID, so any read can reference the corresponding ID to determine what version of the state to use. That is actually how it works in Jetpack Compose. **Each snapshot is assigned its own ID**. Snapshot ids are monotonically increasing value, so it makes snapshots naturally ordered. Since snapshots are differentiated by their IDs, reads and writes are isolated from each other without the need for locking.

Now that we have an idea of why a concurrency control system is needed, and how Multiversion concurrency control works, it is a great time to dive into the internals of the state snapshot system.

If you want to dive deeper into concurrency control systems or MVCC, I highly recommend reading more about [Concurrency control<sup>a</sup>](#) and [Multiversion concurrency control<sup>b</sup>](#).

<sup>a</sup>[https://en.wikipedia.org/wiki/Concurrency\\_control](https://en.wikipedia.org/wiki/Concurrency_control)

<sup>b</sup>[https://en.wikipedia.org/wiki/Multiversion\\_concurrency\\_control](https://en.wikipedia.org/wiki/Multiversion_concurrency_control)

## The Snapshot

A snapshot can be taken at any point in time. It reflects the current state of the program (all the snapshot state objects) at a given instant (when the snapshot is taken). Multiple snapshots can be taken, and all of them will receive **their own isolated copy of the program state**. That is, a copy of the current state of all the snapshot state objects at that point in time. (Objects implementing the State interface).

This approach makes state safe for modification, since updating a state object in one of the snapshots will not affect another copies of the same state object in others. Snapshots are isolated from each other. In a concurrent scenario with multiple threads, each thread would point to a different snapshot and therefore a different copy of the state.

The Jetpack Compose runtime provides the `Snapshot` class to model the **current** state of the program. Any code that wants to take a Snapshot just needs to call the static method for it: `val snapshot =`

`Snapshot.takeSnapshot()`. This will take a snapshot of the current value of all the state objects, and those values will be preserved until `snapshot.dispose()` is called. That will determine the lifespan of the snapshot.

Snapshots have a lifecycle. Whenever we are done using a snapshot, it needs to be disposed. If we don't call `snapshot.dispose()` we will be leaking all the resources associated with the snapshot, along with its retained state. A snapshot is considered **active** between the created and disposed states.

When a snapshot is taken it is given an ID so all the state on it can be easily differentiated from other potential versions of the same state retained by other snapshots. That allows to **version** the program state, or in other words, keep the program state **coherent according to a version** (multiversion concurrency control).

The best way to understand how Snapshots work is by code. I'm going to extract a snippet directly from [this really didactic and detailed post by Zach Klipp<sup>17</sup>](#) for this matter:

#### SnapshotSample.kt

---

```

1 fun main() {
2     val dog = Dog()
3     dog.name.value = "Spot"
4     val snapshot = Snapshot.takeSnapshot()
5     dog.name.value = "Fido"
6
7     println(dog.name.value)
8     snapshot.enter { println(dog.name.value) }
9     println(dog.name.value)
10 }
11
12 // Output:
13 Fido
14 Spot
15 Fido

```

---

The `enter` function, also commonly referred to as “entering the snapshot”, **runs a lambda in the context of the snapshot**, so the snapshot becomes its source of truth for any state: All the state read from the lambda will get its values from the snapshot. This mechanism allows Compose and any other client libraries to run any piece of logic that works with state in the context of a given snapshot. This happens locally in the thread, and until the call to `enter` returns. Any other threads remain completely unaffected.

In the example above we can see how the dog name is “Fido” after updating it, but if we read it from the context of the snapshot (`enter` call), it returns “Spot”, which is **the value it had when the snapshot was taken**.

---

<sup>17</sup><https://dev.to/zachklipp/introduction-to-the-compose-snapshot-system-19cn>

Note that inside `enter` it is possible to read and write state, depending on the type of the snapshot we are using (read-only vs mutable). We will go over mutable snapshots later.

The snapshot you create via `Snapshot.takeSnapshot()` is a read-only one. Any state it holds cannot be modified. If we try to write to any state object in the snapshot, an exception will be thrown.

But not everything will be reading state, we might also need to update it (write). Compose provides a specific implementation of the `Snapshot` contract that allows mutating the state it holds: `MutableSnapshot`. On top of that, there are also other additional implementations available. Here we have a collapsed view of all the different types:

#### Snapshot.kt

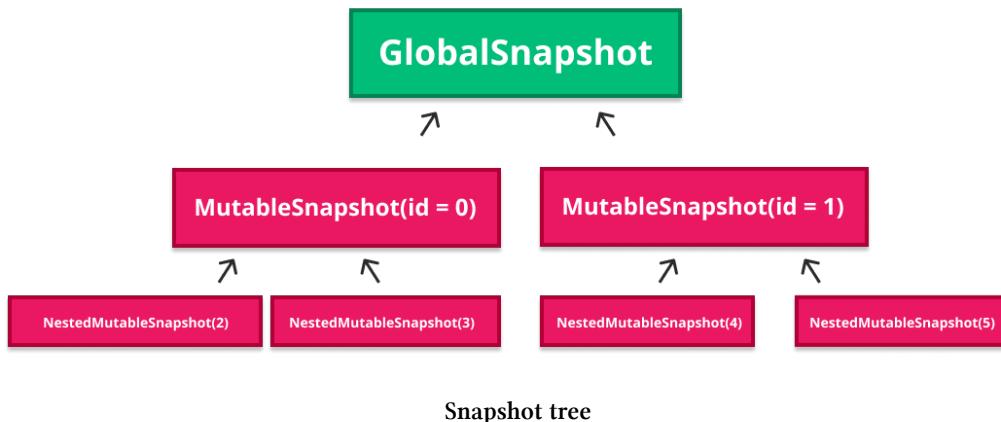
```
1 sealed class Snapshot(...) {  
2     class ReadonlySnapshot(...) : Snapshot() {...}  
3     class NestedReadonlySnapshot(...) : Snapshot() {...}  
4     open class MutableSnapshot(...) : Snapshot() {...}  
5     class NestedMutableSnapshot(...) : MutableSnapshot() {...}  
6     class GlobalSnapshot(...) : MutableSnapshot() {...}  
7     class TransparentObserverMutableSnapshot(...) : MutableSnapshot() {...}  
8 }
```

Let's go over the different types very briefly:

- `ReadonlySnapshot`: Snapshot state objects held by it cannot be modified, only read.
- `MutableSnapshot`: Snapshot state objects held by it can be read and modified.
- `NestedReadonlySnapshot` and `NestedMutableSnapshot`: Child read-only and mutable snapshots, since snapshots form a tree. A snapshot can have any number of nested snapshots. More on this later.
- `GlobalSnapshot`: Mutable snapshot that holds the global (shared) program state. It is effectively the ultimate root of all snapshots.
- `TransparentObserverMutableSnapshot`: This one is a special case. It does not apply any state isolation, and exists only to notify read and write observers whenever a state object is read / written. All state records on it are automatically flagged as invalid, so they are not be visible/readable by any other snapshot. The ID of this type of snapshot is always the one of its parent, so any records created for it are actually associated with the parent instead. It is “transparent” in the sense that all operations performed on it are as if they were performed in the parent snapshot.

## The snapshot tree

As we explained above, **snapshots form a tree**. Among the different snapshot types we can find `NestedReadOnlySnapshot` and `NestedMutableSnapshot` for this reason. Any snapshot can contain any number of nested snapshots. The root of the tree is the `GlobalSnapshot`, holding the global state.



Nested snapshots are like independent copies of the snapshot that can be disposed independently. That allows to dispose it while keeping the parent snapshot active. They are frequent in Compose when we are dealing with **subcomposition**, for instance.

Short flashback to chapter 2. we described that subcompositions are compositions created inline (within the parent composition) with the only intention to support **independent invalidation**. Compositions and subcompositions are also connected as a tree.

Some examples of subcomposition where a nested snapshot is created are when a lazy list item or a `BoxWithConstraints` are composed. We can also find subcomposition in `SubcomposeLayout`, or the `VectorPainter` for example (see examples from chapter 2).

When subcomposition is needed, a nested snapshot is created to store and isolate its state, so the snapshot can be disposed when subcomposition is gone, while keeping the parent composition and parent snapshot alive. If any changes take place to the nested snapshot, those are propagated to the parent.

All the snapshot types provide a function to take a nested snapshot and attach it to the parent. I.e: `Snapshot#takeNestedSnapshot()`, or `MutableSnapshot#takeNestedMutableSnapshot()`.

A child read-only snapshot can be produced from any snapshot type. A mutable snapshot can only be produced from another mutable snapshot (or from the global snapshot which can be thought of as a mutable snapshot).

## Snapshots and threading

It is important to think of snapshots as separate structures that live outside of the scope of any thread. A thread can indeed have a current snapshot, but snapshots are **not necessarily bound to a thread**. A thread can enter and leave a snapshot arbitrarily, and a child snapshot can be entered by a separate thread. Actually, parallel work is one of the intended use cases for snapshots. Several child threads can be spawned, each with their own snapshot.

Once we define mutable snapshots, we'll also learn how child snapshots must notify their changes to the parent to keep coherence. Changes on all threads will be isolated from each other, and colliding updates by different threads will be detected and addressed (more on this later). Nested snapshots allow such a break-down of work to be recursive. All this potentially unlocks features like parallel composition.

It is always possible to retrieve the current snapshot for a thread via `Snapshot.current`. That will return the current thread snapshot if there is one, or the global snapshot (holding the global state) in other case.

## Observing reads and writes

The Compose runtime has the ability to trigger recomposition when state that is observed is written. It would be nice to understand how that machinery, that we already described in previous chapters, is connected to the state snapshot system. Let's go for it, but let's start by learning how to observe reads first.

Whenever take a snapshot (i.e: `Snapshot.takeSnapshot()`), what we get in return is a `ReadOnlySnapshot`. Since the state objects from this snapshot cannot be modified, only read, all the state in the snapshot will be preserved until it gets disposed. The `takeSnapshot` function allows us to pass a `readObserver` (as an optional parameter). This observer will be notified every time any state object is read from the snapshot **within the `enter` call**:

---

### ReadOnlySnapshot.kt

---

```

1 // simple observer to track the total number of reads
2 val snapshot = Snapshot.takeSnapshot { reads++ }
3 // ...
4 snapshot.enter { /* some state reads */ }
5 // ...

```

---

One example of this can be the `snapshotFlow` function: `fun <T> snapshotFlow(block: () -> T): Flow<T>`. This function converts `State<T>` objects into a `Flow`. When collected, it runs its block and emits the result of the `State` objects read in it. When one of the `State` objects read mutates, the `Flow` emits the new value to its collector. To achieve this behavior, it needs to record all the state reads so it can reexecute the block whenever any of those state objects change. To keep track of the reads, it takes a read-only snapshot and passes a read observer so it can store them in a `Set`:

**SnapshotFlow.kt**


---

```

1 fun <T> snapshotFlow(block: () -> T): Flow<T> {
2     // ...
3     snapshot.takeSnapshot { readSet.add(it) }
4     // ...
5     // Do something with the Set
6 }
```

---

Read-only snapshots not only notify their read `readObserver` when some state is read, but also their parent's `readObserver`. A read on a nested snapshot must be visible to all the parents and their observers, so all the observers on the snapshot tree are notified accordingly.

Let's go for observing writes now.

Observers are also possible for writes (state updates), so `writeObserver` can only be passed when creating a `mutable` snapshot. A mutable snapshot is a snapshot that allows to modify the state it holds. We can take one by calling `Snapshot.takeMutableSnapshot()`. Here, we are allowed to pass optional read and write observers to get notified about any reads and/or writes.

A good example of observing reads and writes can be the `Recomposer`, which is able to track any reads and writes into the Composition, to automatically trigger recomposition when required. Here it is:

**Recomposer.kt**


---

```

1 private fun readObserverOf(composition: ControlledComposition): (Any) -> Unit {
2     return { value -> composition.recordReadOf(value) } // recording reads
3 }
4
5 private fun writeObserverOf(
6     composition: ControlledComposition,
7     modifiedValues: IdentityArraySet<Any>?
8 ): (Any) -> Unit {
9     return { value ->
10         composition.recordWriteOf(value) // recording writes
11         modifiedValues?.add(value)
12     }
13 }
14
15 private inline fun <T> composing(
16     composition: ControlledComposition,
17     modifiedValues: IdentityArraySet<Any>?,
18     block: () -> T
19 ): T {
20     val snapshot = Snapshot.takeMutableSnapshot(
```

```

21     readObserverOf(composition),
22     writeObserverOf(composition, modifiedValues)
23 )
24 try {
25     return snapshot.enter(block)
26 } finally {
27     applyAndCheck(snapshot)
28 }
29 }
```

---

The `composing` function is called both when creating the initial `Composition` and for every recomposition. This logic relies on a `MutableSnapshot` that allows state to be not only read but also written, and any reads or writes in the `block` are tracked by (notified to) the `Composition`. (See the `enter` call).

The `block` passed to it will essentially be the code that runs the composition or recomposition itself, and therefore executes all `Composable` functions on the tree to calculate the list of changes. Since that happens inside the `enter` function, that will make any reads or writes automatically tracked.

Every time a `snapshot` state write is tracked into the composition, the corresponding `RecomposeScopes` reading the very same `snapshot` state will be invalidated and recomposition will trigger.

The `applyAndCheck(snapshot)` call in the end propagates any changes happening during the composition to other snapshots and the global state.

This is how observers look in code, they are simple functions:

#### `ReadWriteObservers.kt`

---

```

1 readObserver: ((Any) -> Unit)?
2 writeObserver: ((Any) -> Unit)?
```

---

There is some utility function to start observing reads and writes in the current thread. That is `Snapshot.observe(readObserver, writeObserver, block)`. This function is used by `derivedStateOf` to react to all object reads from the provided block, for instance. This is the only place where the `TransparentObserverMutableSnapshot` is used (one of the `Snapshot` types available). A parent (root) snapshot of this type is created with the only purpose of notifying reads to observers, as explained in previous sections. This type was added by the team to avoid having to have a list of callbacks in the snapshot for a special case.

## MutableSnapshots

We have talked much about state updates (writes), but we didn't really go in detail about mutable snapshots yet. Let's do it now without further ado.

`MutableSnapshot` is the snapshot type used when working with mutable snapshot state where we need to track writes to automatically trigger recomposition.

In a mutable snapshot, any state object will have the same value as it had when the snapshot was taken, **unless it is locally changed in the snapshot**. All changes made in a `MutableSnapshot` are **isolated** from the changes done by other snapshots. Changes propagate from bottom to top on the tree. A child nested mutable snapshot needs to apply its changes first, and then propagate those to the parent or to the global snapshot in case it is the root of the tree. That is done by calling `NestedMutableSnapshot#apply` (or `MutableSnapshot#apply` if it is not nested).

Propagating from bottom to top ensures that changes will reach the global state only when the root snapshot is applied, which can only happen after all the nested snapshots have been already applied.

The following paragraph is extracted directly from the Jetpack Compose runtime kdocs:

*Composition uses mutable snapshots to allow changes made in Composable functions to be temporarily isolated from the global state and is later applied to the global state when the composition is applied. If `MutableSnapshot.apply` fails applying this snapshot, the snapshot and the changes calculated during composition are disposed and a new composition is scheduled to be calculated again.*

So, when applying the Composition (rapid flashback: we apply changes via the `Applier` as the last step in the composition), any changes in mutable snapshots are applied and notified to their parents, or ultimately the global snapshot (program state). If there is a failure when applying these changes, a new composition is scheduled.

A mutable snapshot also has a lifecycle. It always ends by calling `apply` and/or `dispose`. That is required both to propagate state modifications to other snapshots, and to avoid leaks.

Changes propagated via `apply` are applied **atomically**, meaning that the global state or the parent snapshot (in case its nested) will see all those changes as a **single atomic change**. That will clean the history of state changes a bit so it is easier to identify, reproduce, abort, or revert. Remember this is what Transactional memory is about, as we described when learning about Concurrency control systems.

If a mutable snapshot is disposed but never applied, all its pending state changes are discarded.

Here is a practical example of how `apply` works in client code:

#### ApplyMutableSnapshotSample.kt

```
1 class Address {
2     var streetname: MutableState<String> = mutableStateOf("")
3 }
4
5 fun main() {
6     val address = Address()
7     address.streetname.value = "Some street"
8 }
```

```
9  val snapshot = Snapshot.takeMutableSnapshot()
10 println(address.streetname.value)
11 snapshot.enter {
12     address.streetname.value = "Another street"
13     println(address.streetname.value)
14 }
15 println(address.streetname.value)
16 snapshot.apply()
17 println(address.streetname.value)
18 }
19
20 // This prints the following:
21
22 // Some street
23 // Another street
24 // Some street
25 // Another street
```

---

When we print from within the `enter` call, the value is “Another street”, so the modification is visible. That is because we are running in the context of the snapshot. But if we print right after the `enter` call (outside), the value seems reverted to the original one. That is because changes in a `MutableSnapshot` are isolated from any other snapshots. After calling `apply`, changes are propagated, and then we can see how printing the `streetname` again finally prints the modified value.

Note that only state updates done within the `enter` call will be tracked and propagated.

There is also the alternative syntax: `Snapshot.withMutableSnapshot` to shortcut this pattern. It will ensure that `apply` is called in the end.

```
1 fun main() {
2     val address = Address()
3     address.streetname.value = "Some street"
4
5     Snapshot.withMutableSnapshot {
6         println(address.streetname.value)
7         address.streetname.value = "Another street"
8         println(address.streetname.value)
9     }
10    println(address.streetname.value)
11 }
```

The way `apply` is called in the end might remind us of how a list of changes is also recorded and applied later by the Composer –see chapter 3–. It is the same concept once again. Whenever we need to make sense of a list of changes on a tree all together, there is a need to record/defer those, so we can apply (trigger) them in the correct order and impose coherence at that moment. That is the only time when the program knows about all the changes, or in other words, when it has the big picture.

It is also possible to register `apply` observers to observe the ultimate modification changes. That is done via `Snapshot.registerApplyObserver`.

## GlobalSnapshot and nested snapshots

The `GlobalSnapshot` is a type of mutable snapshot that happens to hold the global state. It will get updates coming from other snapshots following the bottom to top order described above.

A `GlobalSnapshot` cannot be nested. There is only one `GlobalSnapshot` and it is effectively the ultimate root of all snapshots. It holds the current global (shared) state. For this reason, a global snapshot can't be applied (it has no `apply` call).

To apply changes in the global snapshot, it must be “advanced”. That is done by calling `Snapshot.advanceGlobalSnapshot()`, which clears the previous global snapshot and creates a new one, that accepts all the valid state from the previous one. `Apply` observers are also notified in this case, since those changes are effectively “applied”, even if the mechanism is different. The same way, it is also not possible to call `dispose()` on it. Disposing a global snapshot is also done by advancing it.

In Jetpack Compose, the global snapshot is created during initialization of the snapshot system. In JVM this happens when `SnapshotKt.class` is initialized by Java or the Android runtime.

After this, the global snapshot manager is started when creating the Composer, and then each composition (including initial composition and any further recompositions) creates its own nested mutable snapshot and attaches it to the tree, so it can store and isolate all the state for the composition. Compositions will also use this chance to register read and write observers to track reads and writes into the Composition. Remember the `composing` function:

### Recomposer.kt

---

```

1 // Called for the initial composition and also for every recomposition.
2 private inline fun <T> composing(
3     composition: ControlledComposition,
4     modifiedValues: IdentityArraySet<Any>?,
5     block: () -> T
6 ): T {
7     val snapshot = Snapshot.takeMutableSnapshot(
8         readObserverOf(composition),
9         writeObserverOf(composition, modifiedValues)
10    )
11    try {
12        return snapshot.enter(block)
13    } finally {
14        applyAndCheck(snapshot)
15    }
16 }
```

---

Finally, any subcomposition can create its own nested snapshot and attach it to the tree, to support invalidation while keeping the parent active. That would give us a complete picture of how a snapshot tree can look.

Another interesting detail to share is that, when the Composer is created, right when creating the Composition, a call to `GlobalSnapshotManager.ensureStarted()` is done. That is part of the integration with the platform (Compose UI), and it will start observing all writes to the global state and schedule periodic dispatch of the snapshot apply notifications in the `AndroidUiDispatcher.Main` context.

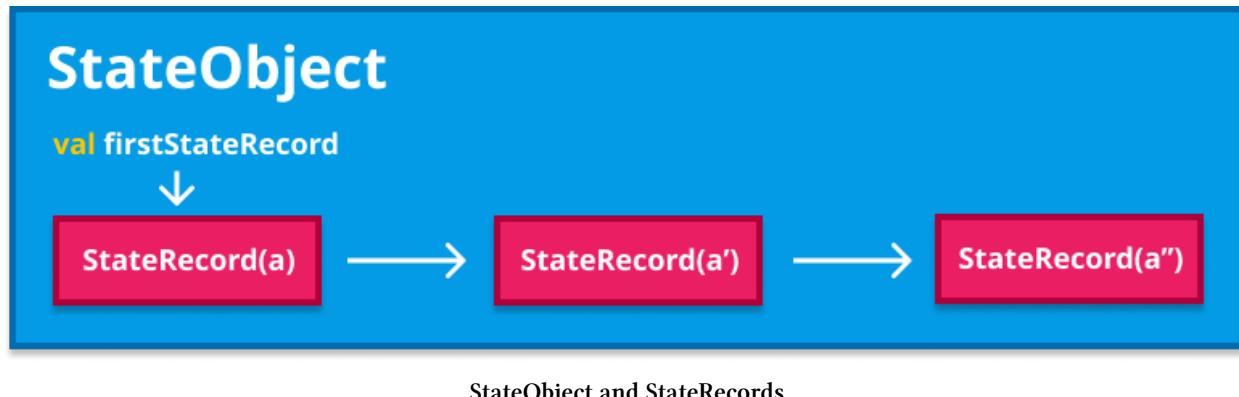
## StateObjects and StateRecords

Multiversion concurrency control ensures that every time state is written, a new version of it is created (copy-on-write). The Jetpack Compose state snapshot system complies with this, so it is possible to end up with multiple versions of the same snapshot state object stored.

This design is important for performance to three ways. First, the cost of creating a snapshot is  $O(1)$ , not  $O(N)$  (where  $N$  would be the number of state objects). Second, the cost of committing a snapshot is  $O(N)$ , where  $N$  is the number of objects mutated in the snapshot. Third, snapshots do not have a list of snapshot data anywhere (only a transitory list of modified objects) so state objects can be collected by the garbage collector freely without the Snapshot system having to be notified.

Internally, a snapshot state object is modeled as a `StateObject`, and each one of the multiple versions stored for that object is a `StateRecord`. Every record holds the data for a single version of the

state. The version (record) that each snapshot sees corresponds to the most up to date valid version available **when the snapshot was taken**. (The valid one with the highest snapshot ID).



But what makes a state record valid?

Well, “valid” is always relative to a particular snapshot. Records are associated with the ID of the snapshot in which the record was created. A state record is considered valid **for a snapshot** if its recorded ID is less than or equal to the snapshot id (that is, created in the current or a previous snapshot), and not part of the snapshot’s `invalid` set, or specifically flagged as invalid. Any valid records from a previous snapshot are automatically copied to the new one.

Which leads to the question: what makes a record be part of the mentioned `invalid` set or explicitly flagged as invalid?

- Records created **after** the current snapshot are considered invalid, since the snapshot they were created for was taken **after** this snapshot.
- Records created for a snapshot that was already open at the time this snapshot was created are added to the `invalid` set, so they are also considered invalid.
- Records created in a snapshot that was disposed before it was applied are explicitly flagged as invalid, also.

An invalid record is a record that is not visible by any snapshot, thus it cannot be read. When a snapshot state is read from a Composable function, that record will not be taken into account to return its most up to date valid state.

Back to the state objects. Here is a brief example of how they are modeled in the state snapshot system:

## Snapshot.kt

---

```

1 interface StateObject {
2     val firstStateRecord: StateRecord
3
4     fun prependStateRecord(value: StateRecord)
5
6     fun mergeRecords(
7         previous: StateRecord,
8         current: StateRecord,
9         applied: StateRecord
10    ): StateRecord? = null
11 }
```

---

Any mutable snapshot state object created by any means will implement this interface. Some examples are the state returned by the `mutableStateOf`, `mutableStateListOf`, or `derivedStateOf` runtime functions, among others.

Let's dive into the `mutableStateOf(value)` function as an exercise.

## SnapshotState.kt

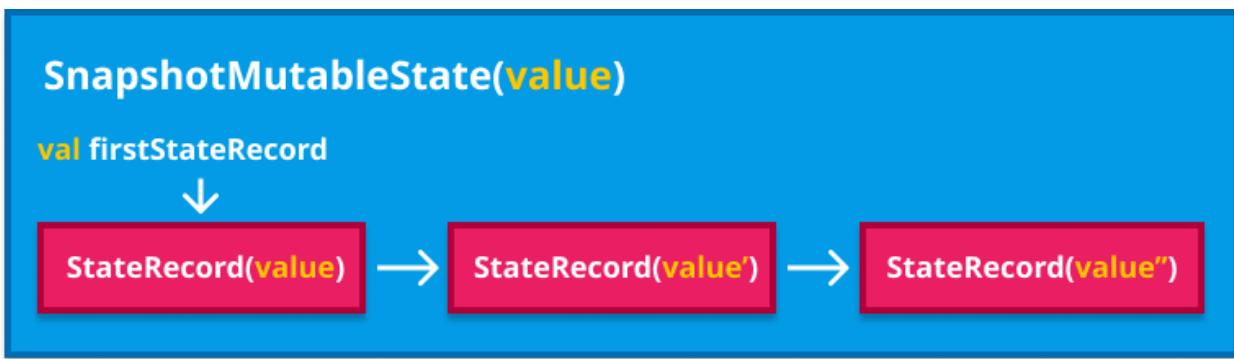
---

```

1 fun <T> mutableStateOf(
2     value: T,
3     policy: SnapshotMutationPolicy<T> = structuralEqualityPolicy()
4 ): MutableState<T> = createSnapshotMutableState(value, policy)
```

---

This call returns an instance of `SnapshotMutableState`, which is essentially an observable mutable state, or in other words, a state that can be updated and will automatically notify observers about it. This class is a `StateObject`, and for that reason it maintains a linked list of records storing different versions of the state (`value` in this case). Each time the state is read, the list of records is traversed to find and return **the most recent valid one**.



If we look back at the `StateObject` definition, we can see how it has a pointer to the first element of the linked list of records, and each record points to the next one. It also allows to prepend a new record to the list (making it become the new `firstStateRecord`).

Another function part of the `StateObject` definition is the `mergeRecords` one. If you remember, we previously mentioned that the system can merge conflicts automatically when possible. That is what this function is for. The merging strategy is simple and will be covered in detail later.

Let's inspect `StateRecords` a bit now.

#### Snapshot.kt

---

```
1 abstract class StateRecord {
2     internal var snapshotId: Int = currentSnapshot().id // associated with
3
4     internal var next: StateRecord? = null // points to the next one
5
6     abstract fun assign(value: StateRecord)
7
8     abstract fun create(): StateRecord
9 }
```

---

Here we can see how each record is associated a snapshot ID. That will be the ID of the snapshot in which the record was created. That is what will determine if the record is valid for a given snapshot following the requirements described above.

We said that whenever an object is read, the list of `StateRecords` for a given snapshot state (`StateObject`) is traversed, looking for the most recent valid one (with the highest snapshot ID). The same way, when a snapshot is taken, the most recent valid state of every snapshot state object is captured, and that will be the state used for all the lifespan of the new snapshot. (Unless it is a mutable snapshot and the state is mutated locally).

The `StateRecord` also has functions to assign it a value from another record, and to initially create it.

`StateRecord` is also a contract (interface). The different implementations available are defined by each existing type of `StateObject`. That is because records store relevant information for the state object, which differs for each type (per use case).

Following the example of the `mutableStateOf`, we learned that it returns a `SnapshotMutableState`, which is a `StateObject`. It will maintain a linked list of records of a very specific type: `StateStateRecord`. That record is just a wrapper over a value of type `T`, since that is all the information we need to store per record in this case.

## SnapshotMutableState(value: T)

```
val firstStateRecord
```



```
StateStateRecord(value)
```



```
StateStateRecord(value')
```



```
StateStateRecord(value'')
```

mutableStateOf

Another good example can be the case of `mutableStateListOf`. It creates a `SnapshotStateList`, which is another implementation of `StateObject`. This state models a mutable list that can be observed (implements the `MutableList` Kotlin collection contract), so its records will have the type `StateListStateRecord`, defined by itself. This record uses a `PersistentList` (see [Kotlin immutable collections<sup>18</sup>](#)) to hold a version of the state list.

## SnapshotStateList(persistentListOf())

```
val firstStateRecord
```



```
StateListStateRecord(list)
```



```
StateListStateRecord(list')
```



```
StateListStateRecord(list'')
```

mutableStateOf

## Reading and writing state

Or in other words, reading and writing state records.

“When an object is read, the list of `StateRecords` for a given snapshot state (`StateObject`) is traversed, looking for the most recent valid one (with the highest snapshot ID).” Let’s see how that looks in code.

---

<sup>18</sup><https://github.com/Kotlin/kotlinx.collections.immutable>

**TextField.kt**


---

```

1 @Composable
2 fun TextField(...) {
3     // ...
4     var textFieldValueState by remember { mutableStateOf(TextFieldValue(text = value))\
5 }
6     // ...
7 }
```

---

This is the `TextField` composable from the `compose.material` library. It remembers a mutable state for the text value, so every time the value is updated the composable recomposes to show the new character on screen.

Let's keep the call to `remember` aside, since it is not relevant for what is worth for this explanation. Here is the `mutableStateOf` function used to create the snapshot state:

**SnapshotState.kt**


---

```

1 fun <T> mutableStateOf(
2     value: T,
3     policy: SnapshotMutationPolicy<T> = structuralEqualityPolicy()
4 ): MutableState<T> = createSnapshotMutableState(value, policy)
```

---

This ultimately creates a `SnapshotMutableState` state object that gets the `value: T` and a `SnapshotMutationPolicy<T>` as arguments. It will wrap (store in memory) the value and use the mutation policy whenever it needs to be updated, to check if the new value passed is different than the current one or not. Here is how that `value` property is defined in the class:

**SnapshotState.kt**


---

```

1 internal open class SnapshotMutableStateImpl<T>(
2     value: T,
3     override val policy: SnapshotMutationPolicy<T>
4 ) : StateObject, SnapshotMutableState<T> {
5
6     override var value: T
7         get() = next.readable(this).value
8         set(value) = next.withCurrent {
9             if (!policy.equivalent(it.value, value)) {
10                 next.overwritable(this, it) { this.value = value }
11             }
12         }
13
14     private var next: StateStateRecord<T> = StateStateRecord(value)
```

```
15
16    // ...
17 }
```

---

Whenever we use the getter to access the inner value from our `TextField Composable` (i.e: `textFieldValueState.value`), it will take the reference to the next state record (first one on the linked list) to start the iteration by calling `readable`. The `readable` function does the iteration to find the current (most fresh) valid readable state for the current snapshot. It also notifies any registered read observers. For every new item iterated it will check if it's valid, following the conditions for valid defined in the previous section. The current snapshot will be the current thread snapshot or the global snapshot if the current thread is not associated to any.

So that is how snapshot state is read for `mutableStateOf`. It will be similar for other mutable snapshot state implementations available like the one returned by `mutableStateListOf`.

For writing the state, we can look at the property setter. Let's add it here again:

#### SnapshotState.kt

---

```
1 set(value) = next.withCurrent {
2     if (!policy.equivalent(it.value, value)) {
3         next.overwritable(this, it) { this.value = value }
4     }
5 }
```

---

The `withCurrent` function calls `readable` under the hood so it can run the provided block passing the current most fresh valid readable state record to it as a parameter.

After that, it checks if the new value is equivalent to the current one or not using the provided `SnapshotMutationPolicy`, and if they are not, it starts the writing process. The function `overwritable` does this job.

I am intentionally not diving into deeper implementation details since those might easily vary in the future, but I'll explain it briefly: It runs the block using a `writable` state record, and proposes a candidate record that in this case will be the current most fresh valid one. If it is valid for the current snapshot it will use it to do the writing, otherwise it will create a new record and prepend it to the list to make it be the new initial one. The block does the actual modification over it.

Finally, it notifies any registered write observers.

## Removing or reusing obsolete records

Multiversion concurrency control introduces an interesting challenge, due to the fact that we can have multiple versions of the same state stored (records): **Removing versions that become obsolete**

and will never be read. We'll explain how Compose solves this problem in a second, but let's introduce the concept of "open snapshots" first. It will be handy.

Any new snapshot taken is added to a set of open snapshots, and will be there until it is proactively closed. While a snapshot stays open, all its state records are considered invalid for other snapshots (not readable). Closing a snapshot means all its records become automatically valid (readable) to any new snapshots created.

Once we know this, let's learn how Compose recycles obsolete records:

1. It tracks the lowest open snapshot. Compose keeps track of a set of open snapshot ids. Those ids are monotonically generated and constantly increasing.
2. If a record is valid but not visible in the lowest open snapshot, then it can be safely reused as it will never be selected by any other snapshot.

Reusing obscured records leads to there typically being only 1 or 2 records in a mutable state object, which improves performance quite a bit. As snapshots are applied the record that is obscured will be reused for the next snapshot. If a snapshot is disposed before apply then all the records are flagged as invalid (discarded) meaning they can be reused immediately.

## Change propagation

Before explaining how changes in mutable snapshots are propagated, it might be useful to recap on what "closing" and "advancing" a snapshot means, so we can grasp both terms.

Closing a snapshot effectively removes its ID from the set of open snapshot IDs, and the consequence of this is that all its state records (records associated with its ID) become visible/readable by any new snapshots created. That makes closing a snapshot an effective way to propagate state changes.

When closing a snapshot, many times we want to replace it by a new one created right away. That is called "advancing" a snapshot. The new snapshot created gets a new ID generated by incrementing the previous one. This ID is then added to the set of open snapshot IDs.

As we have learned, the global snapshot is never applied but always advanced, making all its changes visible to the new global snapshot created. Mutable snapshots can also be advanced when their nested snapshots apply their changes.

Now that we understand this well, we are ready to learn how changes in mutable snapshots are propagated.

When calling `snapshot.apply()` on a **mutable** snapshot, all the local changes made to state objects within its scope are propagated to the parent (in case it is a nested mutable snapshot), or to the global state.

Calling `apply` and/or `dispose` delimits the lifespan of the snapshot. A mutable snapshot that is applied can also be disposed after. However, calling `apply` after `dispose` will throw, since those changes are already discarded.

Per what we have described, to get all the local changes propagated (visible to new snapshots taken), it should be enough to simply **remove the snapshot from the active snapshot set**. Whenever a snapshot is created, a copy of the current open snapshots are passed in as the set of invalid snapshots (that is, no snapshot that has not already been applied should be visible to the new snapshot). Simply removing the snapshot id from the set of open snapshots is enough for every new snapshot to treat the records created during this snapshot as valid, and therefore they can be returned when their corresponding state object is read.

But this should only be done after determining that there are no state collisions (colliding writes), since those would need to be addressed first.

When snapshots are applied, changes made by the applying snapshot are **added together with changes of other snapshots**. A state object has a single linked list of records where all the changes are aggregated. This opens the door to write collisions, since multiple snapshots might try to apply changes over the same state objects. When a mutable snapshot wants to apply (notify / propagate) its local changes, it tries to detect potential write collisions and **merge those as possible**. Merging is covered in detail in the next section.

We have two scenarios to cover here:

### No pending local changes

If there are no pending local changes in the snapshot:

- The mutable snapshot gets proactively closed (removes it from the set of open snapshot ids, making all its state records automatically visible/readable by new snapshots taken).
- The global snapshot is “advanced” (same as closed but also replaced by a new global snapshot created).
- Uses the chance to check if there were any state changes in the global snapshot also, so the mutable snapshot can notify any potential apply observers about those changes in that case.

### With pending local changes:

When there are pending changes:

- Detects collisions and calculates the merged records using an optimistic approach (remember concurrency control categories). Collisions are tried to be merged automatically or discarded otherwise.
- For every pending local change it checks if it is different than the current value. If it is not, it ignores the change and keeps the current value.

- If it's an actual change (different), it checks the already calculated optimistic merges to decide whether to keep the previous, the current, or the applied record. It can actually create a merge of all of them.
- In case it had to perform a merge of the records, it'll create a new record (immutability) and assign the snapshot id to it (associate it with the mutable snapshot), then prepend it to the linked list of records, making it effectively be the first one on the list.

In case there's any failure when applying the changes, it will fallback to the same process done when there are no pending local changes. That is closing the mutable snapshot to make its records visible to any new ones, advancing the global snapshot (close and replace it by a new one), so it includes all the changes in the mutable snapshot just closed, and notifies any apply observers about any global state changes detected.

For nested mutable snapshots the process varies a bit, since those do not propagate their changes to the global snapshot but to their parent. For that reason, they add all its modified state objects to the modified set of the parent. Since all those changes need to be visible by the parent, the nested mutable snapshot removes its own id from the parent set of invalid snapshots.

## Merging write conflicts

To do the merges, the mutable snapshot iterates over its list of modified states (local changes), and for every change it does the following:

- Obtains the current value (state record) in the parent snapshot or the global state.
- Obtains the previous value before applying the change.
- Obtains the state the object would have after applying the change.
- Tries to automatically merge the three of them. This is delegated into the state object, which relies on a provided **merging policy** (see `StateObject` definition some sections ago).

Truth is none of the available policies in the runtime support proper merging at the moment, so colliding updates will result in a runtime exception notifying the user about the problem. To avoid falling into this scenario, Compose guarantees that collisions are not possible by using unique keys to access state objects (state object remembered in a composable function often have unique access property). Given `mutableStateOf` uses a `StructuralEqualityPolicy` for merges, it compares two versions of the object via a deep equals (`==`), so all properties are compared, including the unique object key, making it impossible for two objects to collide.

Auto merge of conflicting changes was added as a potential optimization that Compose does not make use of yet, but other libraries could.

A custom `SnapshotMutationPolicy` can be provided by implementing this interface. An example (extracted from the Compose docs) can be a policy that treats `MutableState<Int>` as a counter. This policy assumes that changing the state value to the same is not considered a change, hence any changes to a mutable state with a `counterPolicy` can never cause an apply conflict.

**CounterPolicy.kt**


---

```

1 fun counterPolicy(): SnapshotMutationPolicy<Int> = object : SnapshotMutationPolicy<I\
2 nt> {
3     override fun equivalent(a: Int, b: Int): Boolean = a == b
4     override fun merge(previous: Int, current: Int, applied: Int) =
5         current + (applied - previous)
6 }
```

---

Two values are considered equivalent when they are the same, and therefore the current value will be kept. Note how merging is obtained adding the difference between the new applied value and the previous one, so the current value always reflects the total amount stored.

This paragraph is extracted from the official docs also, as it is very explanatory: *As the name of the policy implies, it can be useful when counting things, such as tracking the amount of a resource consumed or produced while in a snapshot. For example, if snapshot A produces 10 things and snapshot B produces 20 things, the result of applying both A and B should be that 30 things were produced.*

**CounterPolicy2.kt**


---

```

1 val state = mutableStateOf(0, counterPolicy())
2 val snapshot1 = Snapshot.takeMutableSnapshot()
3 val snapshot2 = Snapshot.takeMutableSnapshot()
4 try {
5     snapshot1.enter { state.value += 10 }
6     snapshot2.enter { state.value += 20 }
7     snapshot1.apply().check()
8     snapshot2.apply().check()
9 } finally {
10    snapshot1.dispose()
11    snapshot2.dispose()
12 }
13
14 // State is now 30 as the changes made in the snapshots are added together.
```

---

We have a single mutable state using the counter policy for comparison, and a couple of snapshots that try to modify it and apply the changes. This would be the perfect scenario for collisions, but given our counter policy, any collisions are completely avoided.

This is only a simple example of how to provide a custom `SnapshotMutationPolicy` that avoid conflicts, so we can get the point. Another implementation where collisions wouldn't be possible could be one for sets that can only add elements, not remove. Other useful types (such as ropes) can similarly be turned into conflict-free data-types given certain constraints on how they work and what the expected result is.

We could also provide custom policies that accept collisions but resolve them by merging the data using the `merge` function.

# 6. Effects and effect handlers

Before jumping into effect handlers it is probably welcome to recap a bit about what to consider a side effect. That will give us some context about why it is key to keep side effects under control in our Composable trees.

## Introducing side effects

Side effects were covered in chapter one when learning about the properties of Composable functions. We learned that side effects make functions non-deterministic, and therefore they make it hard for developers to reason about code.

In essence, a side effect is anything that escapes the control and scope of a function. Imagine a function that is expected to add two numbers:

Add.kt

---

```
1 fun add(a: Int, b: Int) = a + b
```

---

This is also frequently referred to as a “pure” function, since it only uses its inputs to calculate a result. That result will never vary for the same input values, since the only thing the function does is adding them. Therefore we can say this function is **deterministic**, and we can easily reason about it.

Now, let’s consider adding some collateral actions to it:

AddWithSideEffect.kt

---

```
1 fun add(a: Int, b: Int) =  
2     calculationsCache.get(a, b) ?:  
3         (a + b).also { calculationsCache.store(a, b, it) }  
4 }
```

---

We are introducing a calculations cache to save computation time if the result was already computed before. This cache escapes the control of the function, so nothing tells us whether the value read from it has not been modified since last execution, for example. Imagine that this cache is getting updated concurrently from a different thread, and suddenly two sequential calls to `get(a, b)` for the same inputs return two different values:

**AddWithSideEffect2.kt**

---

```
1 fun main() {  
2     add(1, 2) // 3  
3     // Another thread calls: cache.store(1, 2, res = 4)  
4     add(1, 2) // 4  
5 }
```

---

The add function returns a different value for the same inputs, hence it is not deterministic anymore. The same way, imagine that this cache was not in-memory but relied on a database. We could get exceptions thrown by get and store calls depending on something like currently missing a connection to the database. Our calls to add could also fail under unexpected scenarios.

As a recap we can say that side effects are unexpected actions happening on the side, out of what callers would expect from the function, and that can alter its behavior. Side effects make it hard for developers to reason about code, and also remove testability, opening the door to flakiness.

Different examples of side effects can be writing to or reading from a global variable, accessing a memory cache, a database, performing a network query, displaying something on screen, reading from a file... etc.

## Side effects in Compose

We learned how we fall into the same issues when side effects are executed within Composable functions, since that effectively makes the effect escape the control and constraints imposed by the Composable lifecycle.

Something we have also learned previously is how any Composable can suffer multiple recompositions. For that reason, running effects directly within a Composable is not a great idea. This is something we already mentioned in chapter 1 when listing the properties of Composable functions, one of them being that Composable functions are restartable.

Running effects inside a Composable is too risky since it can potentially compromise the integrity of our code and our application state. Let me bring back an example we used in chapter 1: A Composable function that loads its state from network:

**SideEffect.kt**

---

```
1 @Composable
2 fun EventsFeed(networkService: EventsNetworkService) {
3     val events = networkService.loadAllEvents() // side effect
4
5     LazyColumn {
6         items(events) { event ->
7             Text(text = event.name)
8         }
9     }
10 }
```

---

The effect here will run on every recomposition, which is likely not what we are looking for. The runtime might require to recompose this Composable many times in a very short period of time. The result would be lots of concurrent effects without any coordination between them. What we probably wanted was to run the effect only once on first composition instead, and keep that state for the complete Composable lifecycle.

Now, let's imagine that our use case is Android UI, so we are using `compose-ui` to build a Composable tree. Any Android applications contain side effects. Here is an example of what could be a side effect to keep an external state updated.

**SideEffect2.kt**

---

```
1 @Composable
2 fun MyScreen(drawerTouchHandler: TouchHandler) {
3     val drawerState = rememberDrawerState(DrawerValue.Closed)
4
5     drawerTouchHandler.enabled = drawerState.isOpen
6
7     // ...
8 }
```

---

This composable describes a screen with a drawer with touch handling support. The drawer state is initialized as `Closed`, but might change to `Open` over time. For every composition and recomposition, the composable notifies the `TouchHandler` about the current drawer state to enable touch handling support only when it's `Open`.

Line `drawerTouchHandler.enabled = drawerState.isOpen` is a side effect. We're assigning a callback reference on an external object as a **side effect of the composition**.

As we have described already, the problem on doing it right in the Composable function body is that we don't have any control on when this effect runs, so it'll run on every composition / recomposition, and will **never get disposed**, opening the door to potential leaks.

Getting back to the example of a network request, what would happen if, a composable that triggered a network request as a side effect, leaves the composition before it completes?. We might prefer cancelling the job at that point, right?

Since side effects are required to write stateful programs, Jetpack Compose offers mechanisms to run side effects on a lifecycle-aware manner, so one can span a job across recompositions, or get it automatically cancelled when the Composable leaves the composition. These mechanisms are called **effect handlers**.

## What we need

Compositions can be **offloaded to different threads**, executed in parallel, or in different order, among other runtime execution strategies. That's a door for diverse potential optimizations the Compose team wants to keep open, and that is also why we would never want to run our effects right away during the composition without any sort of control.

Overall, we need mechanisms for making sure that:

- Effects run on the correct composable lifecycle step. Not too early, not too late. Just when the composable is ready for it.
- Suspended effects run on a conveniently configured runtime (Coroutine and convenient `CoroutineContext`).
- Effects that capture references have their chance to dispose those when leaving composition.
- Ongoing suspended effects are cancelled when leaving composition.
- Effects that depend on an input that varies over time are automatically disposed / cancelled and relaunched every time it varies.

These mechanisms are provided by Jetpack Compose and called **Effect handlers** ☒

All the effect handlers shared on this post are available in the latest `1.0.0-beta02`. Remember Jetpack Compose froze public API surface when entering beta so they will not change anymore before the `1.0.0` release.

## Effect Handlers

Before describing them let me give you a sneak peek on the `@Composable` lifecycle, since that'll be relevant from this point onwards.

Any composable enters the composition when materialized on screen, and finally leaves the composition when removed from the UI tree. Between both events, effects might run. Some effects can outlive the composable lifecycle, so you can span an effect across compositions.

This is all we need to know for now, let's keep moving.

We could divide effect handlers in two categories:

- **Non suspended effects:** E.g: Run a side effect to initialize a callback when the Composable enters the composition, dispose it when it leaves.
- **Suspended effects:** E.g: Load data from network to feed some UI state.

## Non suspended effects

### DisposableEffect

It represents a side effect of the composition lifecycle.

- Used for non suspended effects that **require being disposed**.
- Fired the first time (when composable enters composition) and then every time its keys change.
- Requires `onDispose` callback at the end. It is disposed when the composable leaves the composition, and also on every recomposition when its keys have changed. In that case, the effect is disposed and relaunched.

#### DisposableEffect.kt

---

```

1  @Composable
2  fun backPressHandler(onBackPressed: () -> Unit, enabled: Boolean = true) {
3      val dispatcher = LocalOnBackPressedDispatcherOwner.current.onBackPressedDispatcher
4
5      val backCallback = remember {
6          object : OnBackPressedCallback(enabled) {
7              override fun handleOnBackPressed() {
8                  onBackPressed()
9              }
10         }
11     }
12
13     DisposableEffect(dispatcher) { // dispose/relaunch if dispatcher changes
14         dispatcher.addCallback(backCallback)
15         onDispose {
16             backCallback.remove() // avoid leaks!
17         }
18     }
19 }
```

---

Here we have a back press handler that attaches a callback to a dispatcher obtained from a `CompositonLocal` (old Ambients). We want to attach the callback when the composable enters the

composition, and also when the dispatcher varies. To achieve that, we can **pass the dispatcher as the effect handler key**. That'll make sure the effect is disposed and relaunched in that case.

Callback is also disposed when the composable finally leaves the composition.

If you'd want to only run the effect once when entering the composition and dispose it when leaving you could **pass a constant as the key**: `DisposableEffect(true)` or `DisposableEffect(Unit)`.

Note that `DisposableEffect` always requires at least one key.

## SideEffect

Another side effect of the composition. This one is a bit special since it's like a "fire on this composition or forget". If the composition fails for any reason, it is **discarded**.

If you are a bit familiar with the internals of the Compose runtime, note that it's an effect **not stored in the slot table**, meaning it does not outlive the composition, and it will not get retried in future across compositions or anything like that.

- Used for effects that **do not require disposing**.
- Runs after every single composition / recomposition.
- Useful to **publishing updates to external states**.

### SideEffect.kt

---

```
1 @Composable
2 fun MyScreen(drawerTouchHandler: TouchHandler) {
3     val drawerState = rememberDrawerState(DrawerValue.Closed)
4
5     SideEffect {
6         drawerTouchHandler.enabled = drawerState.isOpen
7     }
8
9     // ...
10 }
```

---

This is the same snippet we used in the beginning. Here we care about the current state of the drawer, which might vary at any point in time. In that sense, we need to notify it for every single composition or recomposition. Also, if the `TouchHandler` was a singleton living during the complete application execution because this was our main screen (always visible), we might not want to dispose the reference at all.

We can understand `SideEffect` as an effect handler meant to **publish updates** to some external state not managed by the compose State system to keep it always on sync.

## currentRecomposeScope

This is more an effect itself than an effect handler, but it's interesting to cover.

As an Android dev you might be familiar with the `View` system `invalidate` counterpart, which essentially enforces a new measuring, layout and drawing passes on your view. It was heavily used to create frame based animations using the `Canvas`, for example. So on every drawing tick you'd invalidate the view and therefore draw again based on some elapsed time.

The `currentRecomposeScope` is an interface with a single purpose:

`RecomposeScope.kt`

---

```

1 interface RecomposeScope {
2     /**
3         * Invalidate the corresponding scope, requesting the composer recompose this sc\
4     ope.
5     */
6     fun invalidate()
7 }
```

---

So by calling `currentRecomposeScope.invalidate()` it will invalidate composition locally  $\otimes$  **en-forces recomposition**.

It can be useful when using a source of truth that is **not** a Compose State snapshot.

`MyComposable.kt`

---

```

1 interface Presenter {
2     fun loadUser(after: @Composable () -> Unit): User
3 }
4
5 @Composable
6 fun MyComposable(presenter: Presenter) {
7     val user = presenter.loadUser { currentRecomposeScope.invalidate() } // not a Stat\
8     e!
9
10    Text("The loaded user: ${user.name}")
11 }
```

---

Here we have a presenter and we manually invalidate to enforce recomposition when there's a result, since we're not using State in any way. This is obviously a very edgy situation, so you'll likely prefer leveraging State and smart recomposition the big majority of the time.

So overall,  $\otimes$  Use sparingly!  $\otimes$  Use State for smart recomposition when it varies as possible, since that'll make sure to get the most out of the Compose runtime.

For frame based animations Compose provides APIs to suspend and await until the next rendering frame on the choreographer. Then execution resumes and you can update some state with the elapsed time or whatever leveraging smart recomposition one more time. I suggest reading [the official animation docs<sup>19</sup>](#) for a better understanding.

## Suspended effects

### `rememberCoroutineScope`

This call creates a `CoroutineScope` used to create jobs that can be thought as children of the composition.

- Used to run **suspended effects bound to the composition lifecycle**.
- Creates `CoroutineScope` bound to this composition lifecycle.
- The scope is **cancelled when leaving the composition**.
- Same scope is returned across compositions, so we can keep submitting more tasks to it and all ongoing ones will be cancelled when finally leaving.
- Useful to launch jobs **in response to user interactions**.
- Runs the effect on the applier dispatcher (Usually `AndroidUiDispatcher.Main20`) when entering.

`rememberCoroutineScope.kt`

---

```

1  @Composable
2  fun SearchScreen() {
3      val scope = rememberCoroutineScope()
4      var currentJob by remember { mutableStateOf<Job?>(null) }
5      var items by remember { mutableStateOf<List<Item>>(emptyList()) }
6
7      Column {
8          Row {
9              TextField("Start typing to search",
10                  onValueChange = { text ->
11                      currentJob?.cancel()
12                      currentJob = scope.async {
13                          delay(threshold)
14                          items = viewModel.search(query = text)
15                      }
16                  }
17          )
}

```

<sup>19</sup><https://developer.android.com/jetpack/compose/animation#targetbasedanimation>

<sup>20</sup><https://cs.android.com/androidx/platform/frameworks/support/+/androidx-main:compose/ui/ui/src/androidMain/kotlin/androidx/compose/ui/platform/AndroidUiDispatcher.android.kt>

```

18     }
19     Row { ItemsVerticalList(items) }
20   }
21 }
```

---

This is a throttling on the UI side. You might have done this in the past using `postDelayed` or a `Handler` with the `View` system. Every time a text input changes we want to cancel any previous ongoing jobs, and post a new one with a delay, so we always enforce a minimum delay between potential network requests, for example.

The difference with `LaunchedEffect` is that `LaunchedEffect` is used for scoping jobs initiated by the composition, while `rememberCoroutineScope` is thought for scoping jobs initiated by a user interaction.

## LaunchedEffect

This is the suspending variant for loading the initial state of a Composable, as soon as it enters the composition.

- Runs the effect when entering the composition.
- Cancels the effect when leaving the composition.
- Cancels and relaunches the effect when key/s change/s.
- Useful to span a job across recompositions.
- Runs the effect on the applier dispatcher (Usually `AndroidUiDispatcher.Main`<sup>21</sup>) when entering.

### LaunchedEffect.kt

```

1 @Composable
2 fun SpeakerList(eventId: String) {
3     var speakers by remember { mutableStateOf<List<Speaker>>(emptyList()) }
4     LaunchedEffect(eventId) { // cancelled / relaunched when eventId varies
5         speakers = viewModel.loadSpeakers(eventId) // suspended effect
6     }
7
8     ItemsVerticalList(speakers)
9 }
```

---

Not much to say. The effect runs once when entering then once again every time the key varies, since our effect depends on its value. It'll get cancelled when leaving the composition.

Remember that it's also cancelled every time it needs to be relaunched. `LaunchedEffect` requires at least one key.

<sup>21</sup><https://cs.android.com/androidx/platform/frameworks/support/+/androidx-main:compose/ui/ui/src/androidMain/kotlin/androidx/compose/ui/platform/AndroidUiDispatcher.android.kt>

## produceState

This is actually syntax sugar built on top of LaunchedEffect.

- Used when your LaunchedEffect ends up feeding a State (which is most of the time).
- Relies on LaunchedEffect.

### produceState.kt

---

```

1 @Composable
2 fun SearchScreen(eventId: String) {
3     val uiState = produceState(initialValue = emptyList<Speaker>(), eventId) {
4         viewModel.loadSpeakers(eventId) // suspended effect
5     }
6
7     ItemsVerticalList(uiState.value)
8 }
```

---

You can provide a default value for the state, and also **one or multiple keys**.

The only gotcha is that produceState allows to not pass any key, and in that case it will call LaunchedEffect with Unit as the key, making it **span across compositions**. Keep that in mind since the API surface does not make it explicit.

## Third party library adapters

We frequently need to consume other data types from third party libraries like Observable, Flow, or LiveData. Jetpack Compose provides adapters for the most frequent third party types, so depending on the library you'll need to fetch a different dependency:

### Dependencies.kt

---

```

1 implementation "androidx.compose.runtime:runtime:$compose_version" // includes Flow \
2 adapter
3 implementation "androidx.compose.runtime:runtime-livedata:$compose_version"
4 implementation "androidx.compose.runtime:runtime-rxjava2:$compose_version"
```

---

**All those adapters end up delegating on the effect handlers.** All of them attach an observer using the third party library apis, and end up mapping every emitted element to an ad hoc MutableState that is exposed by the adapter function as an immutable State.

Some examples for the different libraries below ☰

## LiveData

**LiveData.kt**


---

```

1 class MyComposableVM : ViewModel() {
2     private val _user = MutableLiveData(User("John"))
3     val user: LiveData<User> = _user
4     //...
5 }
6
7 @Composable
8 fun MyComposable() {
9     val viewModel = viewModel<MyComposableVM>()
10
11     val user by viewModel.user.observeAsState()
12
13     Text("Username: ${user?.name}")
14 }
```

---

Here<sup>22</sup> is the actual implementation of `observeAsState` which relies on `DisposableEffect` handler.

**RxJava2****RxJava2.kt**


---

```

1 class MyComposableVM : ViewModel() {
2     val user: Observable<ViewState> = Observable.just(ViewState.Loading)
3     //...
4 }
5
6 @Composable
7 fun MyComposable() {
8     val viewModel = viewModel<MyComposableVM>()
9
10    val uiState by viewModel.user.subscribeAsState(ViewState.Loading)
11
12    when (uiState) {
13        ViewState.Loading -> TODO("Show loading")
14        ViewState.Error -> TODO("Show Snackbar")
15        is ViewState.Content -> TODO("Show content")
16    }
17 }
```

---

<sup>22</sup><https://cs.android.com/androidx/platform/tools/dokka-devsite-plugin/+master/testData/compose/source/androidx/compose/runtime/livedata/LiveDataAdapter.kt>

Here<sup>23</sup> is the implementation for `subscribeAsState()`. Same story ☺The same extension is also available for `Flowable`.

## KotlinX Coroutines Flow

Flow.kt

---

```

1  class MyComposableVM : ViewModel() {
2      val user: Flow<ViewState> = flowOf(ViewState.Loading)
3      //...
4  }
5
6  @Composable
7  fun MyComposable() {
8      val viewModel = viewModel<MyComposableVM>()
9
10     val uiState by viewModel.user.collectAsState(ViewState.Loading)
11
12     when (uiState) {
13         ViewState.Loading -> TODO("Show loading")
14         ViewState.Error -> TODO("Show Snackbar")
15         is ViewState.Content -> TODO("Show content")
16     }
17 }
```

---

Here<sup>24</sup> is the implementation for `collectAsState`. This one is a bit different since `Flow` needs to be consumed from a suspended context. That is why it relies on `produceState` instead which delegates on `LaunchedEffect`.

So, as you can see all these adapters rely on the effect handlers explained in this post, and you could easily write your own following the same pattern, if you have a library to integrate.

---

<sup>23</sup><https://cs.android.com/androidx/platform/tools/dokka-devsite-plugin/+/master:testData/compose/source/androidx/compose/runtime/rxjava2/RxJava2Adapter.kt>

<sup>24</sup><https://cs.android.com/androidx/platform/tools/dokka-devsite-plugin/+/master:testData/compose/source/androidx/compose/runtime/SnapshotState.kt>

# 7. Advanced Compose Runtime use cases

So far, the book was discussing Compose in the context of Android since it is the angle most people are coming from. The applications of Compose, however, expand far beyond Android or user interfaces. This chapter will go through some of those advanced usages with practical examples.

## Compose runtime vs Compose UI

Before jumping back into internals, it is important to set a distinction between [Compose UI](#) and [Compose runtime](#)<sup>25</sup>. The **Compose UI** is the new UI toolkit for Android, with the tree of `LayoutNodes` which later draw their content on the canvas. The **Compose runtime** provides underlying machinery and many state/composition-related primitives.

With Compose compiler receiving support for the complete spectrum of Kotlin platforms, it is now possible to use the runtime for managing UI or any other tree hierarchies almost everywhere (as long as it runs Kotlin). Note the “other tree hierarchies” part: almost nothing in Compose runtime mentions UI (or Android) directly. While the runtime was surely created and optimized to support that use case, it is still generic enough to build tree structures of any kind. In fact, it is very similar in this matter to React JS, which primary use was to create UI on the web, but it has found much broader use in things like [synthesizers](#) or [3D renderers](#)<sup>26</sup>. Most of the custom renderers reuse core functionality from React runtime but provide their own building blocks in place of browser DOM.

It is no secret that Compose devs were inspired by React while making the library. Even the first prototypes - [XML directly in Kotlin](#)<sup>a</sup> is reminiscent of the HTML-in-JS approach React has. Unsurprisingly, most of the third-party things made with React over the years can be replicated with Compose and run with Kotlin multiplatform.

<sup>a</sup><https://twitter.com/AndroidDev/status/1363207926580711430>

<sup>25</sup><https://jakewharton.com/a-jetpack-compose-by-any-other-name/>

<sup>26</sup><https://github.com/chentsulin/awesome-react-renderer>

```
class ContactsView: Component() {

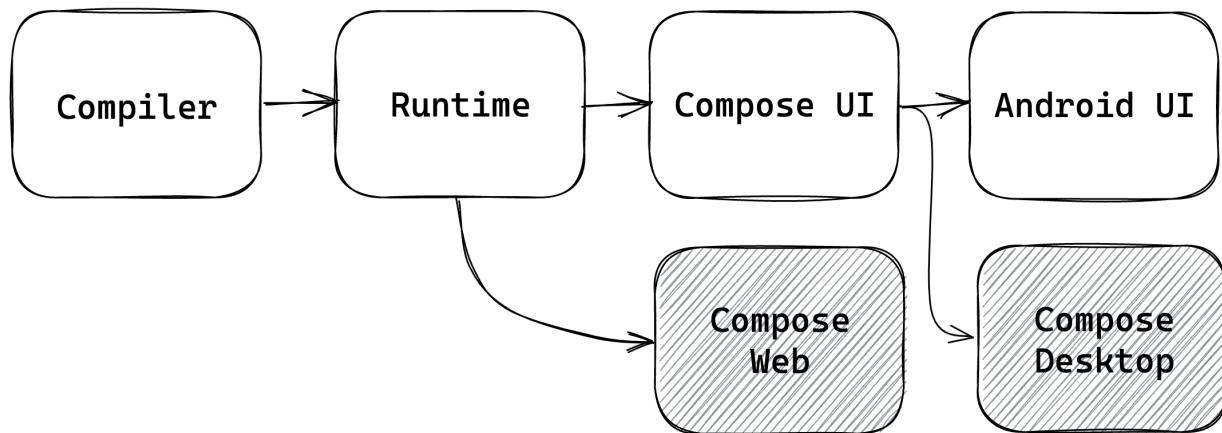
    var contacts: List<Contact>

    override fun compose() {
        <LinearLayout orientation="vertical">
            for (contact in contacts) {
                <LinearLayout>
                    <ImageView image={contact.photo} />
                    <TextView text={contact.name} />
                </LinearLayout>
            }
        </LinearLayout>
    }
}
```

Early prototype of Compose

Even before the Android version of Compose was out of beta, JetBrains already started adopting Compose for Kotlin multiplatform: at the time of writing, they are working on a JVM version for desktop and JS version for browsers, with iOS rumoured to be the next. All of these examples are reusing different parts of Compose:

- Compose for Desktop managed to stay very close to the Android implementation, reusing the whole rendering layer of Compose UI, thanks to ported Skia wrappers. The event system was also extended for better support of mouse/keyboard.
- Compose for iOS (still in development) uses Skia as rendering layer as well. It also reuses a big chunk of existing logic that was portable to Kotlin/Native from JVM.
- Compose for Web went down a path of relying on browser DOM for displaying elements, reusing only compiler and runtime. The available components are defined on top of HTML/CSS, resulting in a very different system from Compose UI. The runtime and compiler, however, are used almost the same way, even though the underlying platform is completely different. With Kotlin WASM support around the corner, Skia based version of Compose is also gaining traction on web, potentially tying all three multiplatform versions together.



– Multiplatform modules by JetBrains

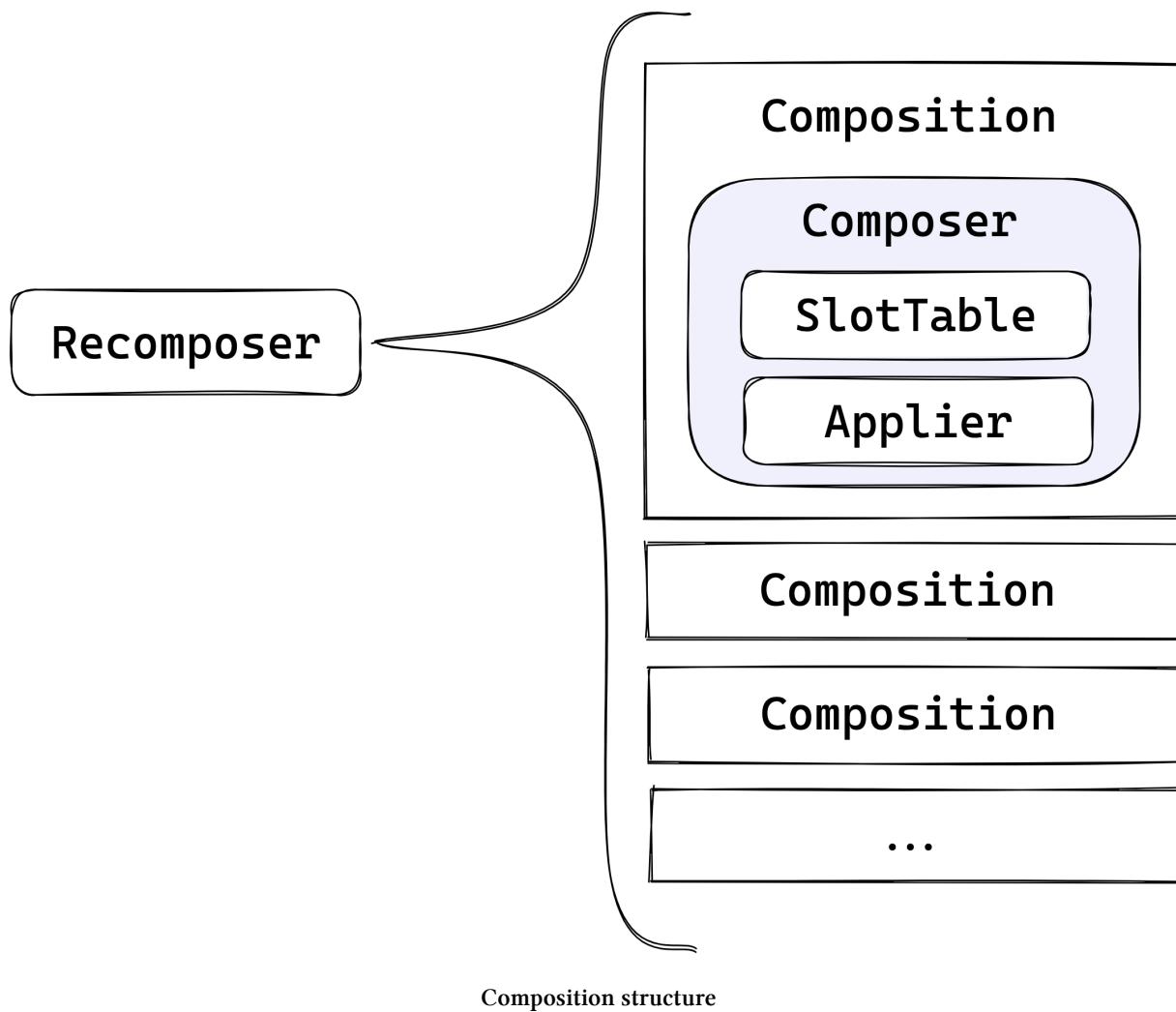
Module structure of Compose with multiplatform

And now... back to the code!

## (Re-) Introducing composition

Composition provides the context for all composable functions. It provides the “cache” backed by the SlotTable and the interface to create custom trees through Applier. Recomposer drives Composition, initiating recomposition whenever something (e.g. state) relevant to it has changed. As documentation mentions<sup>27</sup>, the Composition is usually constructed for you by the framework itself, but guess what? This is a chapter about *unusual* use cases, and later we will manage it by ourselves a few times.

<sup>27</sup><https://cs.android.com/androidx/platform/frameworks/support/+/56f60341d82bf284b8250cf8054b08ae2a91a787:compose/runtime/runtime/src/commonMain/kotlin/androidx/compose/runtime/Composition.kt>



To construct `Composition`, you can use the provided factory method:

#### Composition.kt

---

```

1 fun Composition(
2     applier: Applier<*>,
3     parent: CompositionContext
4 ): Composition = ...

```

---

- Parent context is usually available within any composable function through `rememberCompositionContext()`. Alternatively, `Recomposer` implements `CompositionContext` as well, and it is [obtainable<sup>28</sup>](#) on Android or can be created separately for your own needs.

<sup>28</sup><https://cs.android.com/androidx/platform/frameworks/support/+/56f60341d82bf284b8250cf8054b08ae2a91a787:compose/ui/ui/src/androidMain/kotlin/androidx/compose/ui/platform/WindowRecomposer.android.kt>

- The second parameter is the `Applier`, dictating how to create and connect the tree produced by the `Composition`. The previous chapters already discussed it in detail before and we will go through a few good examples on how to implement it later in this chapter.

Fun fact! You can provide an `Applier` instance that does absolutely nothing, if you are here for other properties of composable functions. Even without nodes, `@Composable` annotation can power data stream conversions or event handlers which react to state changes as all composables do (see Cash App's `Molecule`<sup>3</sup> for example). Just make an `Applier<Nothing>` and don't use `ComposeNode` there!

<sup>3</sup><https://github.com/cashapp/molecule>

Now into the ocean (of code) we go! The rest of this chapter focuses on using **Compose runtime** without **Compose UI**: The first example of such is from Compose UI library, where custom tree is used to render vector graphics (we briefly covered it in earlier chapters). After that, we will switch to Kotlin/JS and create a toy version of the browser DOM management library with Compose.

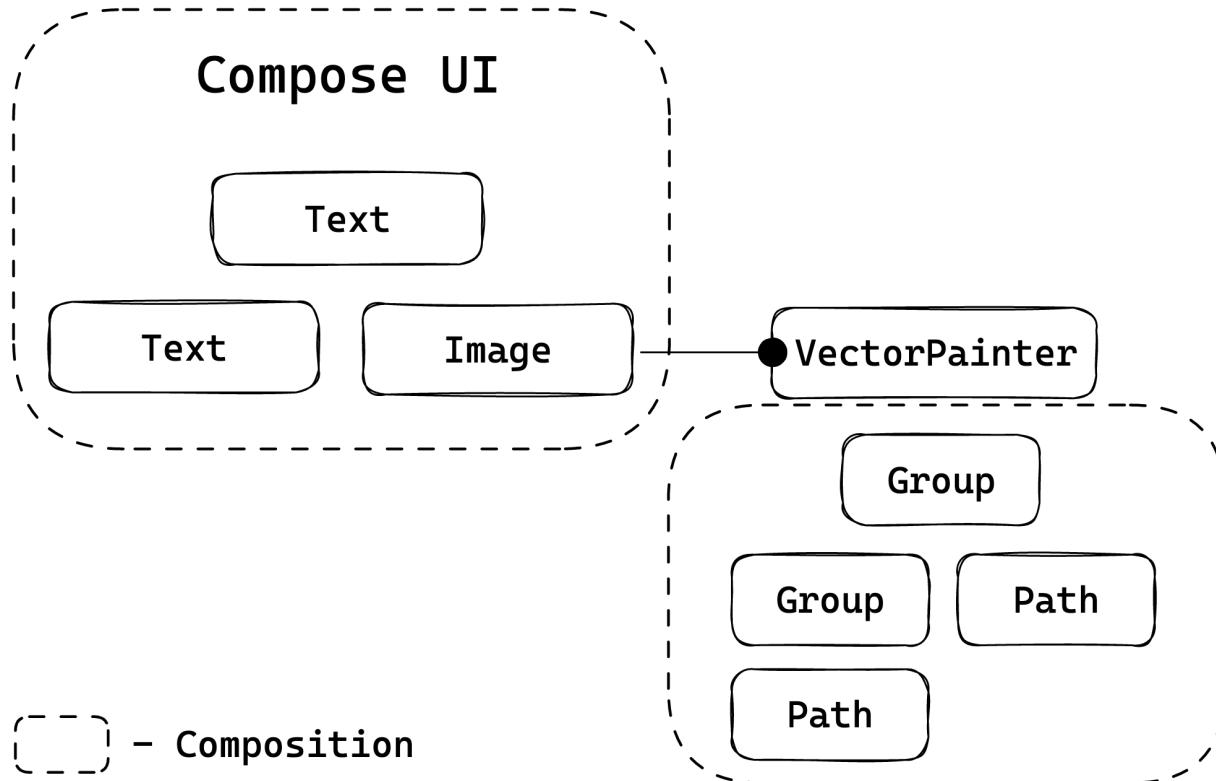
## Composition of vector graphics

Vector rendering in Compose is implemented through the `Painter` abstraction, similar to the `Drawable` in classic Android system:

`VectorExample.kt`

```
1 Image(  
2     painter = rememberVectorPainter { width, height ->  
3         Group(  
4             scaleX = 0.75f,  
5             scaleY = 0.75f  
6         ) {  
7             val pathData = PathData { ... }  
8             Path(pathData = pathData)  
9         }  
10    }  
11 )
```

The functions inside `rememberVectorPainter` block (`Group` and `Path` in particular) are composable and well, but a different kind. Instead of creating `LayoutNodes` as the other composable functions in Compose UI, they create elements specific to the vector. Combining them results in a vector tree, which is later drawn into the canvas.



Compose UI and `VectorPainter` composition.

The `Group` and `Path` exist in a different **composition** from the rest of the UI. That composition is contained within `VectorPainter` and only allows usage of elements describing a vector image, while usual UI composables are forbidden.

The check for vector composables is done during runtime at the moment of writing, so the compiler will happily skip over if you use `Image` or `Box` inside the `VectorPainter` block. This makes writing such painters potentially unsafe, but there were rumours of Compose compiler team improving compile-time safety for cases like this in the future.

Most of the rules about states, effects, and everything about **runtime** discussed in the previous chapters carry over from the UI composition to the vector one. For example, transition API can be used to animate changes of the vector image alongside the UI. Check Compose demos for more details: [VectorGraphicsDemo.kt<sup>29</sup>](#) and [AnimatedVectorGraphicsDemo.kt<sup>30</sup>](#).

<sup>29</sup><https://cs.android.com/androidx/platform/frameworks/support/+/56f60341d82bf284b8250cf8054b08ae2a91a787:compose/ui/ui/integration-tests/ui-demos/src/main/java/androidx/compose/ui/demos/VectorGraphicsDemo.kt>

<sup>30</sup><https://cs.android.com/androidx/platform/frameworks/support/+/56f60341d82bf284b8250cf8054b08ae2a91a787:compose/animation/animation/integration-tests/animation-demos/src/main/java/androidx/compose/animation/demos/vectorgraphics/AnimatedVectorGraphicsDemo.kt>

## Building vector image tree

The vector image is created from elements simpler than LayoutNode to better tailor to the requirements of vector graphics:

VNode.kt

```
1 sealed class VNode {
2     abstract fun DrawScope.draw()
3 }
4
5 // the root node
6 internal class VectorComponent : VNode() {
7     val root = GroupComponent()
8
9     override fun DrawScope.draw() {
10         // set up viewport size and cache drawing
11     }
12 }
13
14 internal classPathComponent : VNode() {
15     var pathData: List<PathNode>
16     // more properties
17
18     override fun DrawScope.draw() {
19         // draw path
20     }
21 }
22
23 internal class GroupComponent : VNode() {
24     private val children = mutableListOf<VNode>()
25     // more properties
26
27     override fun DrawScope.draw() {
28         // draw children with transform
29     }
30 }
```

The nodes above define a tree structure similar to the one used in classic vector drawable XMLs. The tree itself is built from two main types of nodes:

- GroupComponent, which combines children and applies a shared transform to them;
- PathComponent, a leaf node (without children) that draws the pathData.

`fun DrawScope.draw()` provides a way to draw the content of the nodes and their children. The signature of this function is the same as in `Painter` interface which is integrated with the root of this tree later.

The same `VectorPainter` is used to show the XML vector drawable resources from the classic Android system. The XML parser creates a similar structure which is converted to a chain of `Composable` calls, resulting in the same implementation for seemingly different kinds of resources.

The tree nodes above are declared as internal, and the only way to create them is through corresponding `@Composable` declarations. Those functions are the ones used in the example with `rememberVectorPainter` at the start of this section.

### VectorComposables.kt

```
1 @Composable
2 fun Group(
3     scaleX: Float = DefaultScaleX,
4     scaleY: Float = DefaultScaleY,
5     ...
6     content: @Composable () -> Unit
7 ) {
8     ComposeNode<GroupComponent, VectorApplier>(
9         factory = { GroupComponent() },
10        update = {
11            set(scaleX) { this.scaleX = it }
12            set(scaleY) { this.scaleY = it }
13            ...
14        },
15        content = content
16    )
17 }
18
19 @Composable
20 fun Path(
21     pathData: List<PathComponent>,
22     ...
23 ) {
24     ComposeNode<PathComponent, VectorApplier>(
25         factory = { PathComponent() },
26         update = {
27             set(pathData) { this.pathData = it }
28             ...
29         }
30     )
31 }
```

```

29         }
30     )
31 }
```

---

ComposeNode calls emit the node into composition, creating tree elements. Outside of that, @Composable functions don't need interact with the tree at all. After the initial insertion (when the node element is created), Compose tracks updates for the defined parameters and incrementally updates related properties.

- factory parameter defines how the tree node gets created. Here, it is only calling constructors for corresponding Path or Group components.
- update provides a way to update properties of already created instance incrementally. Inside the lambda, Compose memoizes the data with helpers

(such as fun <T> Updater.set(value: T) or fun <T> Updater.update(value: T)) which refresh the tree node properties only when provided value changes to avoid unnecessary invalidations.

- content is the way to add child nodes to their parent. This composable parameter is executed after the update of the node is finished, and all the nodes that are emitted are then parented to the current node. ComposeNode also has an overload without the content parameter, which can be used for leaf nodes, e.g. for Path.

To connect child nodes to the parent, Compose uses Applier, briefly discussed above. VNodes are combined through the VectorApplier:

#### VectorApplier.kt

---

```

1 class VectorApplier(root: VNode) : AbstractApplier<VNode>(root) {
2     override fun insertTopDown(index: Int, instance: VNode) {
3         current.asGroup().insertAt(index, instance)
4     }
5
6     override fun insertBottomUp(index: Int, instance: VNode) {
7         // Ignored as the tree is built top-down.
8     }
9
10    override fun remove(index: Int, count: Int) {
11        current.asGroup().remove(index, count)
12    }
13
14    override fun move(from: Int, to: Int, count: Int) {
15        current.asGroup().move(from, to, count)
16    }
```

```
17
18     override fun onClear() {
19         root.asGroup().let { it.remove(0, it.numChildren) }
20     }
21
22     // VectorApplier only works with [GroupComponent], as it cannot add
23     // children to [PathComponent] by design
24     private fun VNode.asGroup(): GroupComponent {
25         return when (this) {
26             is GroupComponent -> this
27             else -> error("Cannot only insert VNode into Group")
28         }
29     }
30 }
```

Most of the methods in `Applier` interface frequently result in list operations (`insert/move/remove`). To avoid reimplementing them over and over again, `AbstractApplier` even provides convenience extensions for `MutableList`. In the case of `VectorApplier`, these list operations are implemented directly in a `GroupComponent`.

`Applier` provides two methods of insertion: `topDown` and `bottomUp`, with different order of assembling the tree:

- `topDown` first adds a node to the tree and then adds its children, inserting them one by one;
- `bottomUp` creates the node, adds all children, and only then inserts it into the tree.

The underlying reason is performance: some environments have the associated cost of adding children to the tree (think re-layout when adding a View in the classic Android system). For the vector use-case, there's no such performance cost, so the nodes are inserted top-down. See the [Applier documentation<sup>a</sup>](#) for more information.

<sup>a</sup><https://cs.android.com/androidx/platform/frameworks/support/+/androidx-main:compose/runtime/runtime/src/commonMain/kotlin/androidx/compose/runtime/Applier.kt;l=67>

## Integrating vector composition into Compose UI

With the `Applier` in place, the vector composition is almost ready for use. The last part is the `Painter` integration.

**VectorPainter.kt**

---

```
1 class VectorPainter internal constructor() : Painter() {
2     ...
3
4     // 1. Called in the context of UI composition
5     @Composable
6     internal fun RenderVector(
7         content: @Composable (...) -> Unit
8     ) {
9         // 2. The parent context is captured with [rememberCompositionContext]
10        // to propagate its values, e.g. CompositionLocals.
11        val composition = composeVector(
12            rememberCompositionContext(),
13            content
14        )
15
16        // 3. Whenever the UI "forgets" the VectorPainter,
17        // the vector composition is disposed with [DisposableEffect] below.
18        DisposableEffect(composition) {
19            onDispose {
20                composition.dispose()
21            }
22        }
23    }
24
25    private fun composeVector(
26        parent: CompositionContext,
27        composable: @Composable (...) -> Unit
28    ): Composition {
29        ...
30        // See implementation below
31    }
32 }
```

---

The first part of integration is connecting Compose UI composition and the vector image composition:

1. RenderVector accepts content with composable description of the vector image. The Painter instance is usually kept the same between recompositions (with remember), but RenderVector is called on each composition if content has changed.
2. Creating composition always requires a parent context, and here it is taken from the UI composition with rememberCompositionContext. It ensures that both are connected to the same

Recomposer and all internal values (e.g. CompositionLocals for density) are propagated to the vector composition as well.

3. The composition is preserved through updates but should be disposed whenever RenderVector leaves the scope. DisposableEffect manages this cleanup similarly to other kinds of subscriptions in Compose.

Finally, the last step is to populate the composition with image content to create a tree of vector nodes, which is later used to draw vector image on canvas:

#### VectorPainter.kt

---

```
1 class VectorPainter : Painter() {
2     // The root component for the vector tree
3     private val vector = VectorComponent()
4     // 1. Composition with vector elements.
5     private var composition: Composition? = null
6
7     @Composable
8     internal fun RenderVector(
9         content: @Composable (...) -> Unit
10    ) {
11        ...
12        // See full implementation above
13    }
14
15    private fun composeVector(
16        parent: CompositionContext,
17        composable: @Composable (...) -> Unit
18    ): Composition {
19        // 2. Creates composition or reuses an existing one
20        val composition =
21            if (this.composition == null || this.composition.isDisposed) {
22                Composition(
23                    VectorApplier(vector.root),
24                    parent
25                )
26            } else {
27                this.composition
28            }
29            this.composition = composition
30
31        // 3. Sets the vector content to the updated composable value
32        composition.setContent {
33            // Vector composables can be called inside this block only
34        }
35    }
36}
```

```
34     composable(vector.viewportWidth, vector.viewportHeight)
35 }
36
37     return composition
38 }
39
40 // Painter interface integration, is called every time the system
41 // needs to draw the vector image on screen
42 override fun DrawScope.onDraw() {
43     with(vector) {
44         draw()
45     }
46 }
47 }
```

---

1. The painter maintains its own composition, because ComposeNode requires the applier to match whatever is passed to the composition and UI context uses applier incompatible with vector nodes.
2. This composition is refreshed if the painter was not initialized or its composition went out of scope.
3. After creating the composition, it is populated through setContent, similar to the one used inside the ComposeView. Whenever RenderVector is called with different content, setContent is executed again to refresh vector structure. The content adds children to the root node that is later used for drawing contents of Painter.

With that, the integration is finished, and the VectorPainter can now draw @Composable contents on the screen. The composables inside the painter also have access to the state and composition locals from the UI composition to drive their own updates.

With that, you know how to create a custom tree and embed it into the already existing composition. In the next part, we will go through creating a standalone Compose system based on the same principles... in Kotlin/JS.

## Managing DOM with Compose

Multiplatform support is still a new thing for Compose with only runtime and compiler available outside of the JVM ecosystem. These two modules, however, is all we need to create a composition and run something in it, which leads to more experiments!

Compose compiler from Google dependencies supports all Kotlin platforms, but runtime is distributed for Android only. Jetbrains, however, publish their own (mostly unchanged) version of Compose<sup>a</sup> with multiplatform artifacts for JS as well.

<sup>a</sup><https://github.com/JetBrains/compose-jb/releases>

The first step to make Compose magic happen is to figure out the tree it should operate on. Thankfully, browsers already have the “view” system in place based on HTML/CSS. We can manipulate these elements from JS through DOM ([Document Object Model<sup>31</sup>](#)) API, which is also provided by Kotlin/JS standard library.

Before starting with JS, let’s look at HTML representation inside the browser.

sample.html

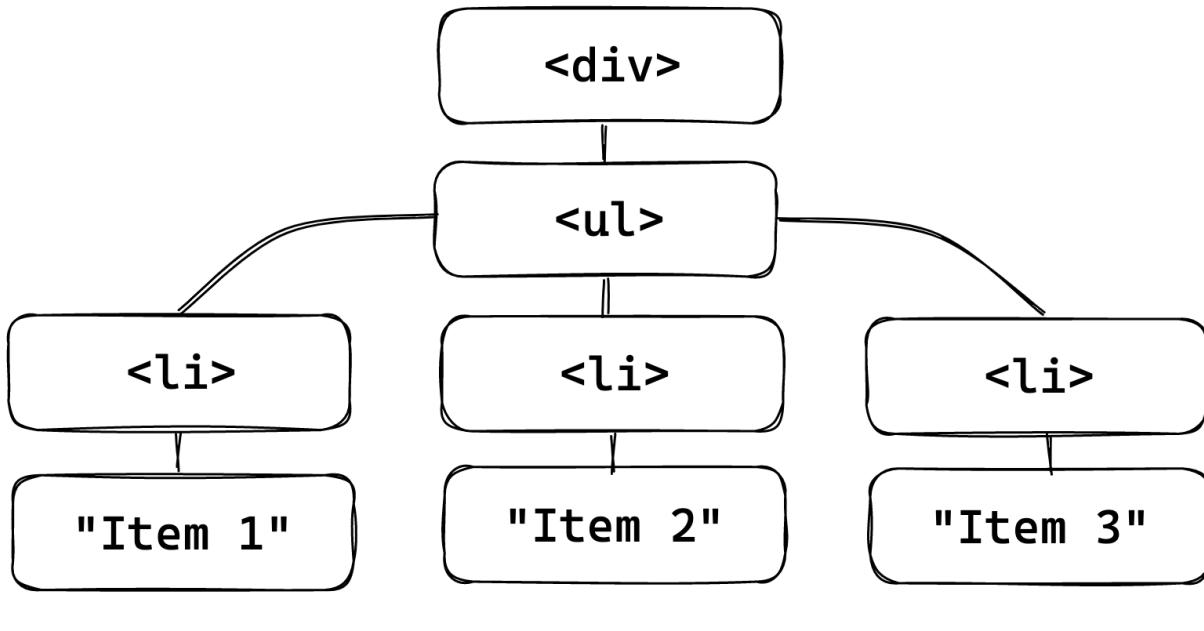
---

```
1 <div>
2   <ul>
3     <li>Item 1</li>
4     <li>Item 2</li>
5     <li>Item 3</li>
6   </ul>
7 </div>
```

---

The HTML above displays an unordered (bulleted) list with three items. From the perspective of the browser, this structure looks like this:

<sup>31</sup>[https://developer.mozilla.org/en-US/docs/Web/API/Document\\_Object\\_Model/Introduction](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction)

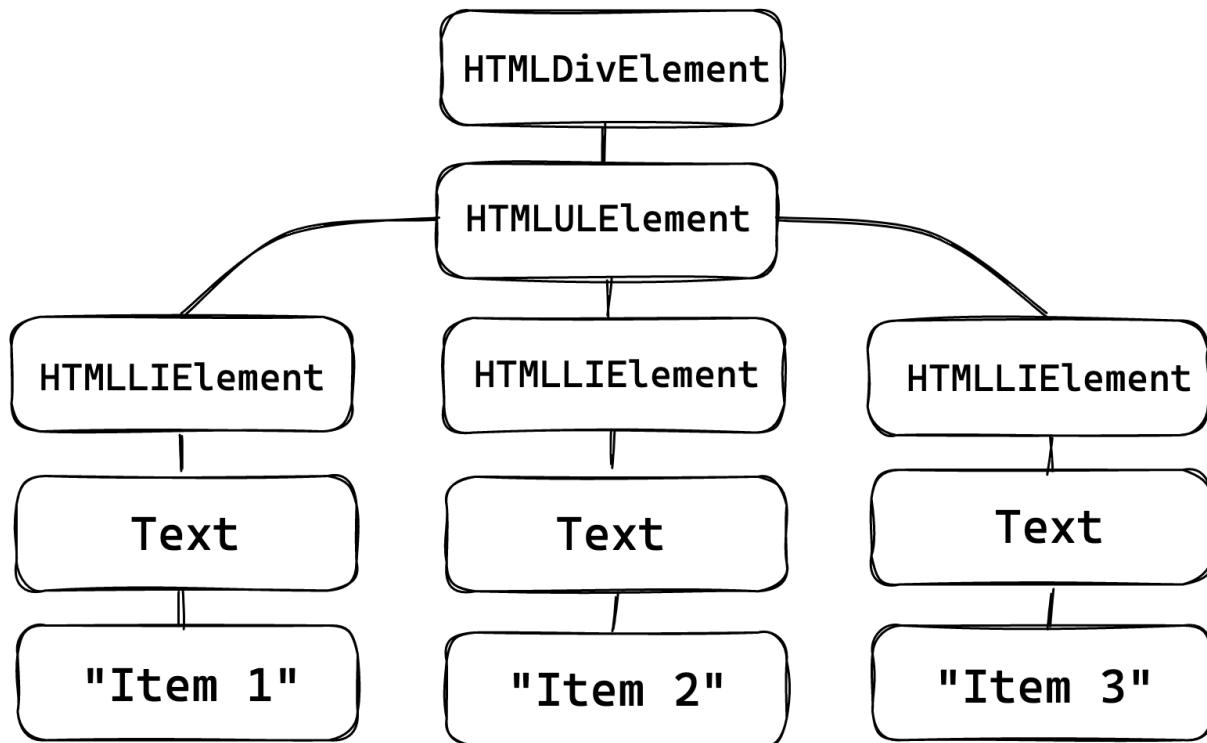


HTML tree representation in the browser

The DOM is a tree-like structure built from elements which are exposed in Kotlin/JS as `org.w3c.dom.Node`. The relevant elements for us are:

- HTML elements (subclasses of `org.w3c.dom HTMLElement`) are representing the tags (e.g. `li` or `div`). They can be created with `document.createElement(<tagName>)` and browser will automatically find correct implementation for a tag,
- Text between the tags (e.g. "Test" in the examples above) represented as a `org.w3c.dom.Text`. Instances of this element can be created with `document.createTextNode(<value>)`

Using these DOM elements, JS sees this tree the following way:



HTML tree representation for JS

These elements will provide the basis for the Compose-managed tree, similarly to how VNodes are used for vector image composition in the previous part.

#### `HtmlTags.kt`

---

```

1  @Composable
2  fun Tag(tag: String, content: @Composable () -> Unit) {
3      ComposeNode<HTMLElement, DomApplier>(
4          factory = { document.createElement(tag) as HTMLElement },
5          update = {},
6          content = content
7      )
8  }
9
10 @Composable
11 fun Text(value: String) {
12     ReusableComposeNode<Text, DomApplier>(
13         factory = { document.createTextNode("") },
14         update = {
15             set(value) { this.data = it }
16         }
17     )
  
```

18 }

Tags cannot be changed in place, as the `<audio>` has a completely different browser representation from `<div>`, so if the tag name has changed, it should be recreated. Compose does not handle this automatically, so it is important to avoid passing different values for tag names into the same composable.

The simplest way to achieve recreation of the nodes is to wrap each node in a separate composable (e.g. `Div` and `UI` for corresponding elements). By doing so, you create different compile-time groups for each of them, hinting to Compose that those elements should be replaced completely instead of just updating their properties.

Text elements, however, are structurally the same, and we indicate it with `ReusableComposeNode`. This way, even when Compose finds these nodes inside different groups, it will reuse the instance. To ensure correctness, the text node is created without content, and the value is set with `update` parameter.

To combine elements into a tree, Compose requires an `Applier` instance operating on DOM elements. The logic for it is very similar to the `VectorApplier` above, except the DOM node methods for adding/removing children are slightly different. Most of the code there is completely mechanical (moving elements to correct indices), so I will omit it here. If you are looking for a reference, I recommend checking [Applier used in Compose for Web<sup>32</sup>](#).

## Standalone composition in the browser

To start combining our new composables into UI, Compose requires an active composition. In Compose UI, all the initialization is already done in the `ComposeView`, but for the browser environment it needs to be created from scratch.

The same principles can be applied for the different platforms as well, as all the components described below exist in the “common” Kotlin code.

<sup>32</sup><https://github.com/JetBrains/compose-jb/blob/6d97c6d0555f056d2616f417c4d130e0c2147e32/web/core/src/jsMain/kotlin/org/jetbrains/compose/web/DomApplier.kt#L63-L91>

**renderComposable.kt**

```
1 fun renderComposable(root: HTMLElement, content: @Composable () -> Unit) {
2     GlobalSnapshotManager.ensureStarted()
3
4     val recomposerContext = DefaultMonotonicFrameClock + Dispatchers.Main
5     val recomposer = Recomposer(recomposerContext)
6
7     val composition = ControlledComposition(
8         applier = DomApplier(root),
9         parent = recomposer
10    )
11
12     composition.setContent(content)
13
14     CoroutineScope(recomposerContext).launch(start = UNDISPATCHED) {
15         recomposer.runRecomposeAndApplyChanges()
16     }
17 }
```

`renderComposable` hides all the implementation details of composition start, providing a way to render composable elements into a DOM element. Most of the setup inside is connected to initializing `Recomposer` with correct clock and coroutine context:

- First, the snapshot system (responsible for state updates) is initialized. `GlobalSnapshotManager` is intentionally left out of runtime, and you can copy it from [Android source<sup>33</sup>](#) if the target platform doesn't have one provided. It is the only part that is not provided by the runtime at the moment.
- Next, the coroutine context for `Recomposer` is created with JS defaults. The default `MonotonicClock` for browsers is controlled with `requestAnimationFrame` (if you are using JetBrains implementation), and `Dispatchers.Main` references the only thread JS operates on. This context is used to run recompositions later.
- Now we are ready to create a composition. It is created the same way as in the vector example above, but now the recomposer is used as a composition parent (`recomposer` always has to be a parent of the top-most composition).
- Afterwards, composition content is set. All the updates to this composition should happen inside provided composable, as new invocations of `renderComposable` will recreate everything from scratch.
- The last part is to start the process of recompositions by launching a coroutine with `Recomposer.runRecomposeAndApplyChanges`. On Android, this process is usually tied to the activity/view lifecycle, with calling `recomposer.cancel()` to stop the recomposition process.

<sup>33</sup><https://cs.android.com/androidx/platform/frameworks/support/+androidx-main:compose/ui/ui/src/androidMain/kotlin/androidx/compose/ui/platform/GlobalSnapshotManager.android.kt>

Here, the composition lifecycle is tied to the lifetime of the page, so no cancellations are needed.

Primitives above can now be combined together to render content of a HTML page:

#### HtmlSample1.kt

```
1 fun main() {
2     renderComposable(document.body!!) {
3         // equivalent of <button>Click me!</button>
4         Tag("button") {
5             Text("Click me!")
6         }
7     }
8 }
```

Creating static content, however, can be achieved by much easier means, and Compose was required in the first place to achieve interactivity. In most cases, we expect something to happen when the button is clicked, and in DOM it can be achieved with, similar to Android views, click listeners.

In Compose UI, many listeners are defined through `Modifier` extensions, but their implementation is specific to `LayoutNode`, thus, not usable for this toy web library. It is possible to copy `Modifier` behavior from Compose UI and adjust nodes used here to provide better integration with event listeners through modifiers, but it is left as an exercise to the reader.

#### HtmlTags.kt

```
1 @Composable
2 fun Tag(
3     tag: String,
4     // this callback is invoked on click events
5     onClick: () -> Unit = {},
6     content: @Composable () -> Unit
7 ) {
8     ComposeNode<HTMLElement, DomApplier>(
9         factory = { createElement(tag) },
10        update = {
11            // when listener changes, the listener on the DOM node is re-set
12            set(onClick) {
13                this.onclick = { _ -> onClick() }
14            }
15        },
16    ),
17 }
```

```
16     content = content
17   )
18 }
```

---

Each tag can now define a click listener as a lambda parameter which is propagated to a DOM node with handy `onClick` property defined for all `HTMLElements`. With that addition, clicks can now be handled by passing `onClick` parameter to the `Tag` composable.

#### HtmlSampleCounter.kt

```
1 fun main() {
2   renderComposable(document.body!!) {
3     // Counter state is updated on click
4     var counterState by remember { mutableStateOf(0) }
5
6     Tag("h1") {
7       Text("Counter value: $counterState")
8     }
9
10    Tag("button", onClick = { counterState++ }) {
11      Text("Increment!")
12    }
13  }
14}
```

---

From here, there are multiple ways to expand this toy library, adding support for CSS, more events, and elements. JetBrains team is currently experimenting on a more advanced version of Compose for Web. It is built on the same principles as the toy version we explored in this chapter but is more advanced in many ways to support a variety of things you can build on the web. You can try [the tech demo<sup>34</sup>](#) yourself with Kotlin/JS projects to learn more.

## Conclusion

In this chapter, we explored how core Compose concepts can be used to built systems outside of Compose UI. Custom compositions are harder to meet in the wild, but they are a great tool to have in your belt if you are already working in Kotlin/Compose environment.

The vector graphics composition is a good example of integrating custom composable trees into Compose UI. The same principles can be used to create other custom elements which can easily interact with states/animations/composition locals from UI composition.

---

<sup>34</sup><https://compose-web.ui.pages.jetbrains.team/>

It is also possible to create standalone compositions on all Kotlin platforms! We explored that by making a toy version of the DOM management library based on Compose runtime in a browser through the power of Kotlin/JS. In a similar fashion, Compose runtime is already used to manipulate UI trees in some projects outside of Android (see [Mosaic<sup>35</sup>](#), Jake Wharton's take on CLI).

I encourage you to experiment on your own ideas with Compose, and provide feedback to Compose team in #compose Kotlin slack channel! Their primary goal is still defined by Compose UI, but they are very excited to learn more about other things Compose is used for.

---

<sup>35</sup><https://github.com/JakeWharton/mosaic>