Lifecycle of composables

In this page, you'll learn about the lifecycle of a composable and how Compose decides whether a composable needs recomposition.

Lifecycle overview

As mentioned in the Managing state documentation (/jetpack/compose/state), a Composition describes the UI of your app and is produced by running composables. A Composition is a tree-structure of the composables that describe your UI.

When Jetpack Compose runs your composables for the first time, during *initial composition*, it will keep track of the composables that you call to describe your UI in a Composition. Then, when the state of your app changes, Jetpack Compose schedules a *recomposition*. Recomposition is when Jetpack Compose re-executes the composables that may have changed in response to state changes, and then updates the Composition to reflect any changes.

A Composition can only be produced by an initial composition and updated by recomposition. The only way to modify a Composition is through recomposition.

Key Point: The lifecycle of a composable is defined by the following events: entering the Composition, getting recomposed 0 or more times, and leaving the Composition.

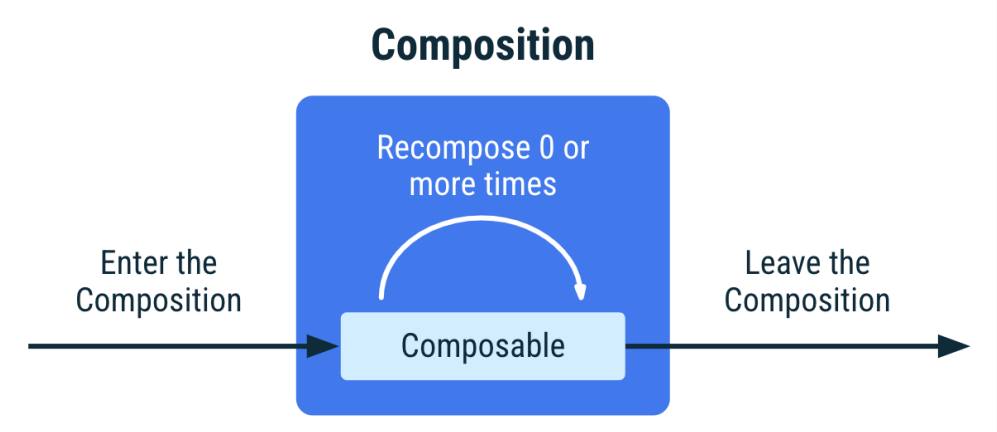

Figure 1. Lifecycle of a composable in the Composition. It enters the Composition, gets recomposed 0 or more times, and leaves the Composition.

Recomposition is typically triggered by a change to a `State<T>` (/reference/kotlin/androidx/compose/runtime/State) object. Compose tracks these and runs all composables in the Composition that read that particular `State<T>`, and any composables that they call that cannot be skipped (#skipping).

Note: A Composable's' lifecycle is simpler than the lifecycle of views, activities, and fragments. When a composable needs to manage or interact with external resources that *do* have a more complex lifecycle, you should use effects (#state-effect-use-cases).

If a composable is called multiple times, multiple instances are placed in the Composition. Each call has its own lifecycle in the Composition.

```
@Composable
fun MyComposable() {
    Column {
        Text("Hello")
        Text("World")
    }
}
ose/snippets/src/main/java/com/example/compose/snippets/lifecycle/LifecycleSnippets.kt#L33-L39)
```

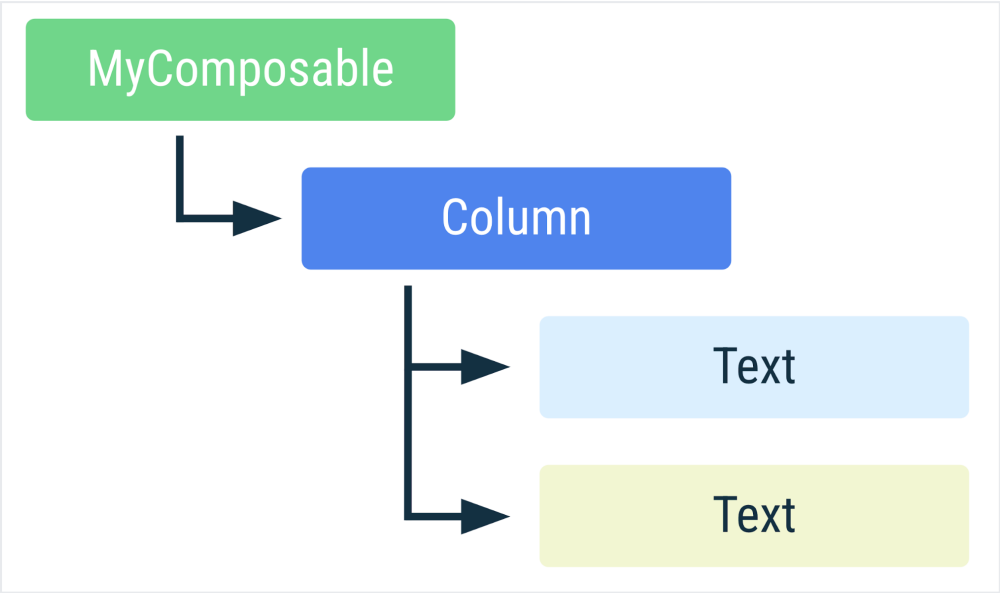

Figure 2. Representation of `MyComposable` in the Composition. If a composable is called multiple times, multiple instances are placed in the Composition. An element having a different color is indicative of it being a separate instance.

Anatomy of a composable in Composition

The instance of a composable in Composition is identified by its **call site**. The Compose compiler considers each call site as distinct. Calling composables from multiple call sites will create multiple instances of the composable in Composition.

Key Term: The **call site** is the *source code location* in which a composable is called. This influences its place in Composition, and therefore, the UI tree.

If during a recomposition a composable calls different composables than it did during the previous composition, Compose will **identify which composables were called or not called** and for the composables that were called in both compositions, Compose will **avoid recomposing them if their inputs haven't changed**.

Preserving identity is crucial to associate side effects with their composable, so that they can complete successfully rather than restart for every recomposition.

Consider the following example:

```
@Composable
fun LoginScreen(showError: Boolean) {
    if (showError) {
        LoginError()
    }
    LoginInput() // This call site affects where LoginInput is placed in Comp
}

@Composable
fun LoginInput() { /* ... */ }

@Composable
fun LoginError() { /* ... */ }
ose/snippets/src/main/java/com/example/compose/snippets/lifecycle/LifecycleSnippets.kt#L43-L55)
```

In the code snippet above, `LoginScreen` will conditionally call the `LoginError` composable and will always call the `LoginInput` composable. Each call has a unique call site and source position, which the compiler will use to uniquely identify it.

Figure 3. Representation of `LoginScreen` in the Composition when the state changes and a recomposition occurs. Same color means it hasn't been recomposed.

Even though `LoginInput` went from being called first to being called second, the `LoginInput` instance will be preserved across recompositions. Additionally, because `LoginInput` doesn't have any parameters that have changed across recomposition, the call to `LoginInput` will be skipped by Compose.

Add extra information to help smart recompositions

Calling a composable multiple times will add it to Composition multiple times as well. When calling a composable multiple times from the same call site, Compose doesn't have any information to uniquely identify each call to that composable, so the execution order is used in addition to the call site in order to keep the instances distinct. This behavior is sometimes all that is needed, but in some cases it can cause unwanted behavior.

```
@Composable
fun MoviesScreen(movies: List<Movie>) {
    Column {
        for (movie in movies) {
            // MovieOverview composables are placed in Composition given its
            // index position in the for loop
            MovieOverview(movie)
        }
    }
}
ose/snippets/src/main/java/com/example/compose/snippets/lifecycle/LifecycleSnippets.kt#L59-L68)
```

In the example above, Compose uses the execution order in addition to the call site to keep the instance distinct in the Composition. If a new `movie` is added to the *bottom* of the list, Compose can reuse the instances already in the Composition since their location in the list haven't changed and therefore, the `movie` input is the same for those instances.

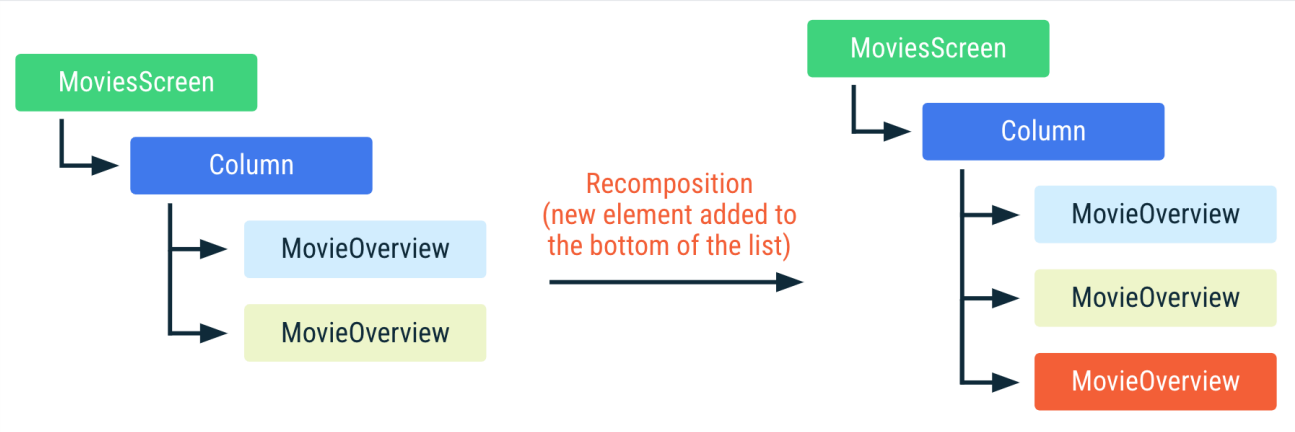

Figure 4. Representation of `MoviesScreen` in the Composition when a new element is added to the bottom of the list. `MovieOverview` composables in the Composition can be reused. Same color in `MovieOverview` means the composable hasn't been recomposed.

However, if the `movies` list changes by either adding to the *top* or the *middle* of the list, removing or reordering items, it'll cause a recomposition in all `MovieOverview` calls whose input parameter has changed position in the list. That's extremely important if, for example, `MovieOverview` fetches a movie image using a side effect. If recomposition happens while the effect is in progress, it will be cancelled and will start again.

```
@Composable
fun MovieOverview(movie: Movie) {
    Column {
        // Side effect explained later in the docs. If MovieOverview
        // recomposes, while fetching the image is in progress,
        // it is cancelled and restarted.
        val image = loadNetworkImage(movie.url)
        MovieHeader(image)

        /* ... */
    }
}
ose/snippets/src/main/java/com/example/compose/snippets/lifecycle/LifecycleSnippets.kt#L72-L88)
```

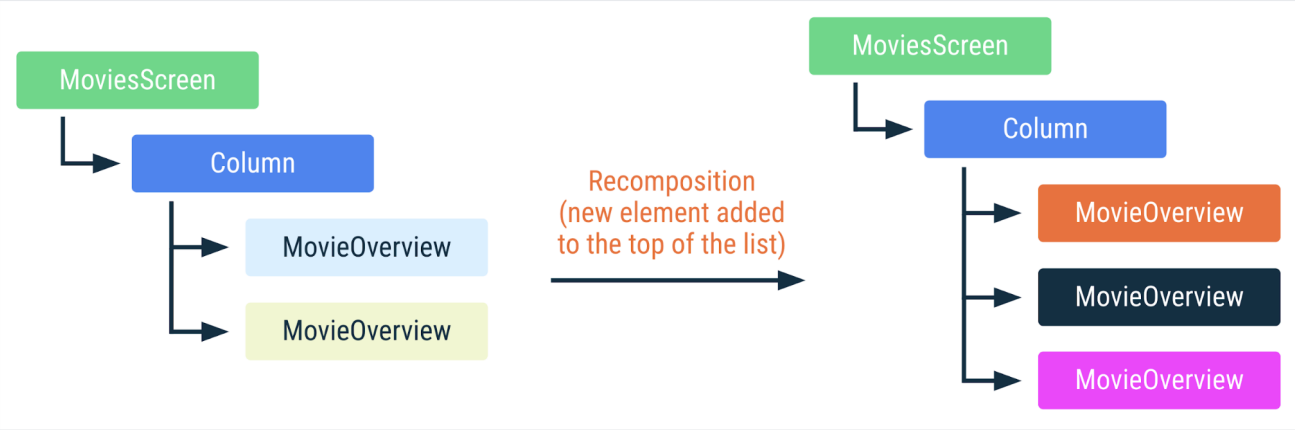

Figure 5. Representation of `MoviesScreen` in the Composition when a new element is added to the list. `MovieOverview` composables cannot be reused and all side effects will restart. A different color in `MovieOverview` means the composable has been recomposed.

Ideally, we want to think of the identity of the `MovieOverview` instance as linked to the identity of the `movie` that is passed to it. If we reorder the list of movies, ideally we would similarly reorder the instances in the Composition tree instead of recomposing each `MovieOverview` composable with a different movie instance. Compose provides a way for you to tell the runtime what values you want to use to identify a given part of the tree: the `key` (/reference/kotlin/androidx/compose/runtime/package-summary#key(kotlin.Array,kotlin.Function0)) composable.

By wrapping a block of code with a call to the key composable with one or more values passed in, those values will be combined to be used to identify that instance in the composition. The value for a `key` does not need to be *globally* unique, it needs to only be unique amongst the invocations of composables at the call site. So in this example, each `movie` needs to have a `key` that's unique among the `movies`; it's fine if it shares that `key` with some other composable elsewhere in the app.

```
@Composable
fun MoviesScreenWithKey(movies: List<Movie>) {
    Column {
        for (movie in movies) {
            key(movie.id) { // Unique ID for this movie
                MovieOverview(movie)
            }
        }
    }
}
```

se/snippets/src/main/java/com/example/compose/snippets/lifecycle/LifecycleSnippets.kt#L92-L101)

With the above, even if the elements on the list change, Compose recognizes individual calls to `MovieOverview` and can reuse them.

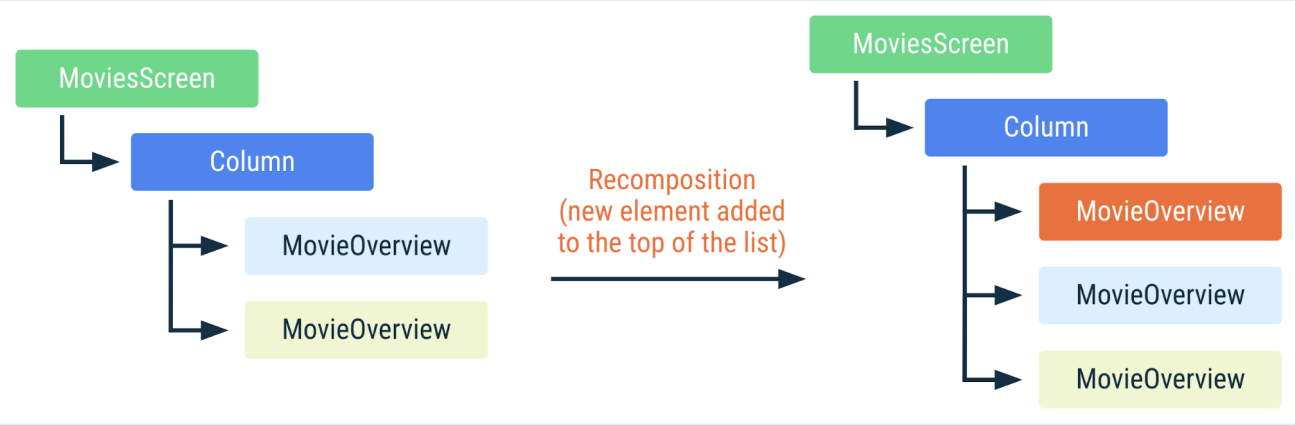

Figure 6. Representation of `MoviesScreen` in the Composition when a new element is added to the list. Since the `MovieOverview` composables have unique keys, Compose recognizes which `MovieOverview` instances haven't changed, and can reuse them; their side effects will continue executing.

Key Point: Use the `key` composable to help Compose identify composable instances in Composition. It's important when multiple composables are called from the same call site and contain side-effects or internal state.

Some composables have built-in support for the `key` composable. For example, `LazyColumn` accepts specifying a custom `key` in the `items` DSL.

```
@Composable
fun MoviesScreenLazy(movies: List<Movie>) {
    LazyColumn {
        items(movies, key = { movie -> movie.id }) { movie ->
            MovieOverview(movie)
        }
    }
}
e/snippets/src/main/java/com/example/compose/snippets/lifecycle/LifecycleSnippets.kt#L105-L112)
```

Skipping if the inputs haven't changed

If a composable is already in the Composition, it can skip recomposition if all the inputs are stable and haven't changed.

A stable type must comply with the following contract:

- The result of `equals` for two instances will *forever* be the same for the same two instances.
- If a public property of the type changes, Composition will be notified.
- All public property types are also stable.

There are some important common types that fall into this contract that the compose compiler will treat as stable, even though they are not explicitly marked as stable by using the `@Stable` annotation:

- All primitive value types: `Boolean`, `Int`, `Long`, `Float`, `Char`, etc.
- Strings
- All Function types (lambdas)

All of these types are able to follow the contract of stable because they are immutable. Since immutable types never change, they never have to notify Composition of the change, so it is much easier to follow this contract.

Note: All deeply immutable types can safely be considered stable types.

One notable type that is stable but *is* mutable is Compose's `MutableState` type. If a value is held in a `MutableState`, the state object overall is considered to be stable as Compose will be notified of any changes to the `.value` property of `State`.

When all types passed as parameters to a composable are stable, the parameter values are compared for equality based on the composable position in the UI tree. Recomposition is skipped if all the values are unchanged since the previous call.

Key Point: Compose skips the recomposition of a composable if all the inputs are stable and haven't changed. The comparison uses the **`equals`** method.

Compose considers a type stable only if it can prove it. For example, an interface is generally treated as not stable, and types with mutable public properties whose implementation could be immutable are not stable either.

If Compose is not able to infer that a type is stable, but you want to force Compose to treat it as stable, mark it with the `@Stable` (/reference/kotlin/androidx/compose/runtime/Stable) annotation.

```
// 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
}
e/snippets/src/main/java/com/example/compose/snippets/lifecycle/LifecycleSnippets.kt#L116-L124)
```

In the code snippet above, since `UiState` is an interface, Compose could ordinarily consider this type to be not stable. By adding the `@Stable` annotation, you tell Compose that this type is stable, allowing Compose to favor smart recompositions. This also means that Compose will treat all its implementations as stable if the interface is used as the parameter type.

Key Point: If Compose is not able to infer the stability of a type, annotate the type with `@Stable` to allow Compose to favor smart recompositions.

Last updated 2023-07-05 UTC.