API Guidelines for Jetpack Compose

Last updated: March 10, 2021

Who this document is for

The Compose API guidelines outline the patterns, best practices and prescriptive style guidelines for writing idiomatic Jetpack Compose APIs. As Jetpack Compose code is built in layers, everyone writing code that uses Jetpack Compose is building their own API to consume themselves.

This document assumes a familiarity with Jetpack Compose's runtime APIs including `@Composable`, `remember {}` and `CompositionLocal`.

The requirement level of each of these guidelines is specified using the terms set forth in RFC2119 for each of the following developer audiences. If an audience is not specifically named with a requirement level for a guideline, it should be assumed that the guideline is OPTIONAL for that audience.

Jetpack Compose framework development

Contributions to the `androidx.compose` libraries and tools generally follow these guidelines to a strict degree in order to promote consistency, setting expectations and examples for consumer code at all layers.

Library development based on Jetpack Compose

It is expected and desired that an ecosystem of external libraries will come to exist that target Jetpack Compose, exposing a public API of `@Composable` functions and supporting types for consumption by apps and other libraries. While it is desirable for these libraries to follow these guidelines to the same degree as Jetpack Compose framework development would, organizational priorities and local consistency may make it appropriate for some purely stylistic guidelines to be relaxed.

App development based on Jetpack Compose

App development is often subject to strong organizational priorities and norms as well as requirements to integrate with existing app architecture. This may call for not only stylistic deviation from these guidelines but structural deviation as well. Where possible, alternative approaches for app development will be listed in this document that may be more appropriate in these situations.

Kotlin style

Baseline style guidelines

Jetpack Compose framework development MUST follow the Kotlin Coding Conventions outlined at https://kotlinlang.org/docs/reference/coding-conventions.html as a baseline with the additional adjustments described below.

Jetpack Compose Library and app development SHOULD also follow this same guideline.

Why

The Kotlin Coding Conventions establish a standard of consistency for the Kotlin ecosystem at large. The additional style guidelines that follow in this document for Jetpack Compose account for Jetpack Compose's language-level extensions, mental models, and intended data flows, establishing consistent conventions and expectations around Compose-specific patterns.

Singletons, constants, sealed class and enum class values

Jetpack Compose framework development MUST name deeply immutable constants following the permitted object declaration convention of `PascalCase` as documented here as a replacement for any usage of `CAPITALS_AND_UNDERSCORES`. Enum class values MUST also be named using `PascalCase` as documented in the same section.

Library development SHOULD follow this same convention when targeting or extending Jetpack Compose.

App Development MAY follow this convention.

Why

Jetpack Compose discourages the use and creation of singletons or companion object state that cannot be treated as *stable* over time and across threads, reducing the usefulness of a distinction between singleton objects and other forms of constants. This forms a consistent expectation of API shape for consuming code whether the implementation detail is a top-level `val`, a `companion object`, an `enum class`, or a `sealed class` with nested `object` subclasses. `myFunction(Foo)` and `myFunction(Foo.Bar)` carry the same meaning and intent for calling code regardless of specific implementation details.

Library and app code with a strong existing investment in `CAPITALS_AND_UNDERSCORES` in their codebase MAY opt for local consistency with that pattern instead.

Do

```
const val DefaultKeyName = "__defaultKey"

val StructurallyEqual: ComparisonPolicy = StructurallyEqualsImpl(...)

object ReferenceEqual : ComparisonPolicy {
    // ...
}

sealed class LoadResult<T> {
```

```
sealed class LoadResult<T> {
    object Loading : LoadResult<Nothing>()

    class Done(val result: T) : LoadResult<T>()
    class Error(val cause: Throwable) : LoadResult<Nothing>()
}

enum class Status {
    Idle,
    Busy
}
```

Don't

```
const val DEFAULT_KEY_NAME = "__defaultKey"

val STRUCTURALLY_EQUAL: ComparisonPolicy = StructurallyEqualsImpl(...)

object ReferenceEqual : ComparisonPolicy {
    // ...
}

sealed class LoadResult<T> {
    object Loading : LoadResult<Nothing>()
    class Done(val result: T) : LoadResult<T>()
    class Error(val cause: Throwable) : LoadResult<Nothing>()
}

enum class Status {
    IDLE,
    BUSY
}
```

Compose baseline

The Compose compiler plugin and runtime establish new language facilities for Kotlin and the means to interact with them. This layer adds a declarative programming model for constructing and managing mutable tree data structures over time. Compose UI is an example of one type of tree that the Compose runtime can manage, but it is not limited to that use.

This section outlines guidelines for `@Composable` functions and APIs that build on the Compose runtime capabilities. These guidelines apply to all Compose runtime-based APIs, regardless of the managed tree type.

Naming Unit @Composable functions as entities

Jetpack Compose framework development and Library development MUST name any function that returns `Unit` and bears the `@Composable` annotation using `PascalCase`, and the name MUST be

that of a noun, not a verb or verb phrase, nor a nouned preposition, adjective or adverb. Nouns MAY be prefixed by descriptive adjectives. This guideline applies whether the function emits UI elements or not.

App development SHOULD follow this same convention.

Why

Composable functions that return `Unit` are considered *declarative entities* that can be either *present* or *absent* in a composition and therefore follow the naming rules for classes. A composable's presence or absence resulting from the evaluation of its caller's control flow establishes both persistent identity across recompositions and a lifecycle for that persistent identity. This naming convention promotes and reinforces this declarative mental model.

Do

```
// This function is a descriptive PascalCased noun as a visual UI element
@Composable
fun FancyButton(text: String, onClick: () -> Unit) {
```

Do

```
// This function is a descriptive PascalCased noun as a non-visual element
// with presence in the composition
@Composable
fun BackButtonHandler(onBackPressed: () -> Unit) {
```

Don't

```
// This function is a noun but is not PascalCased!
@Composable
fun fancyButton(text: String, onClick: () -> Unit) {
```

Don't

```
// This function is PascalCased but is not a noun!
@Composable
fun RenderFancyButton(text: String, onClick: () -> Unit) {
```

Don't

```
// This function is neither PascalCased nor a noun!
@Composable
fun drawProfileImage(image: ImageAsset) {
```

Naming @Composable functions that return values

Jetpack Compose framework development and Library development MUST follow the standard Kotlin Coding Conventions for the naming of functions for any function annotated `@Composable` that returns a value other than `Unit`.

Jetpack Compose framework development and Library development MUST NOT use the factory function exemption in the Kotlin Coding Conventions for the naming of functions for naming any function annotated `@Composable` as a PascalCase type name matching the function's abstract return type.

Why

While useful and accepted outside of `@Composable` functions, this factory function convention has drawbacks that set inappropriate expectations for callers when used with `@Composable` functions.

Primary motivations for marking a factory function as `@Composable` include using composition to establish a managed lifecycle for the object or using `CompositionLocal`s as inputs to the object's construction. The former implies the use of Compose's `remember {}` API to cache and maintain the object instance across recompositions, which can break caller expectations around a factory operation that reads like a constructor call. (See the next section.) The latter motivation implies unseen inputs that should be expressed in the factory function name.

Additionally, the mental model of `Unit`-returning `@Composable` functions as declarative entities should not be confused with a, "virtual DOM" mental model. Returning values from `@Composable` functions named as `PascalCase` nouns promotes this confusion, and may promote an undesirable style of returning a stateful control surface for a present UI entity that would be better expressed and more useful as a hoisted state object.

More information about state hoisting patterns can be found in the design patterns section of this document.

Do

```
// Returns a style based on the current CompositionLocal settings
// This function qualifies where its value comes from
@Composable
fun defaultStyle(): Style {
```

Don't

```
// Returns a style based on the current CompositionLocal settings
// This function looks like it's constructing a context-free object!
@Composable
fun Style(): Style {
```

Naming @Composable functions that remember {} the objects they return

Jetpack Compose framework development and Library development MUST prefix any `@Composable` factory function that internally `remember {}`s and returns a mutable object with the prefix `remember`.

App development SHOULD follow this same convention.

Why

An object that can change over time and persists across recompositions carries observable side effects that should be clearly communicated to a caller. This also signals that a caller does not need to duplicate a `remember {}` of the object at the call site to attain this persistence.

Do

```
// Returns a CoroutineScope that will be cancelled when this call
// leaves the composition
// This function is prefixed with remember to describe its behavior
@Composable
fun rememberCoroutineScope(): CoroutineScope {
```

Don't

```
// Returns a CoroutineScope that will be cancelled when this call leaves
// the composition
// This function's name does not suggest automatic cancellation behavior!
@Composable
fun createCoroutineScope(): CoroutineScope {
```

Note that returning an object is not sufficient to consider a function to be a factory function; it must be the function's primary purpose. Consider a `@Composable` function such as `Flow<T>.collectAsState()`; this function's primary purpose is to establish a subscription to a `Flow`; that it `remember {}`s its returned `State<T>` object is incidental.

Naming CompositionLocals

A `CompositionLocal` is a key into a composition-scoped key-value table. `CompositionLocal`s may be used to provide global-like values to a specific subtree of composition.

Jetpack Compose framework development and Library development MUST NOT name `CompositionLocal` keys using "CompositionLocal" or "Local" as a noun suffix. `CompositionLocal` keys should bear a descriptive name based on their value.

Jetpack Compose framework development and Library development MAY use "Local" as a prefix for

a `CompositionLocal` key name if no other, more descriptive name is suitable.

Do

```
// "Local" is used here as an adjective, "Theme" is the noun.
val LocalTheme = staticCompositionLocalOf<Theme>()
```

Don't

```
// "Local" is used here as a noun!
val ThemeLocal = staticCompositionLocalOf<Theme>()
```

Stable types

The Compose runtime exposes two annotations that may be used to mark a type or function as *stable* - safe for optimization by the Compose compiler plugin such that the Compose runtime may skip calls to functions that accept only safe types because their results cannot change unless their inputs change.

The Compose compiler plugin may infer these properties of a type automatically, but interfaces and other types for which stability can not be inferred, only promised, may be explicitly annotated. Collectively these types are called, “stable types.”

`@Immutable` indicates a type where the value of any properties will **never** change after the object is constructed, and all methods are **referentially transparent**. All Kotlin types that may be used in a `const` expression (primitive types and Strings) are considered `@Immutable`.

`@Stable` when applied to a type indicates a type that is **mutable**, but the Compose runtime will be notified if and when any public properties or method behavior would yield different results from a previous invocation. (In practice this notification is backed by the `Snapshot` system via `@Stable` `MutableState` objects returned by `mutableStateOf()`.) Such a type may only back its properties using other `@Stable` or `@Immutable` types.

Jetpack Compose framework development, Library development and App development MUST ensure in custom implementations of `.equals()` for `@Stable` types that for any two references `a` and `b` of `@Stable` type `T`, `a.equals(b)` MUST **always** return the same value. This implies that any **future** changes to `a` must also be reflected in `b` and vice versa.

This constraint is always met implicitly if `a === b`; the default reference equality implementation of `.equals()` for objects is always a correct implementation of this contract.

Jetpack Compose framework development and Library development SHOULD correctly annotate `@Stable` and `@Immutable` types that they expose as part of their public API.

Jetpack Compose framework development and Library development MUST NOT remove the `@Stable` or `@Immutable` annotation from a type if it was declared with one of these annotations in a previous stable release.

Jetpack Compose framework development and Library development MUST NOT add the `@Stable` or `@Immutable` annotation to an existing non-final type that was available in a previous stable release without this annotation.

Why?

`@Stable` and `@Immutable` are behavioral contracts that impact the binary compatibility of code generated by the Compose compiler plugin. Libraries should not declare more restrictive contracts for preexisting non-final types that existing implementations in the wild may not correctly implement, and similarly they may not declare that a library type no longer obeys a previously declared contract that existing code may depend upon.

Implementing the stable contract incorrectly for a type annotated as `@Stable` or `@Immutable` will result in incorrect behavior for `@Composable` functions that accept that type as a parameter or receiver.

Emit XOR return a value

`@Composable` functions should either emit content into the composition or return a value, but not both. If a composable should offer additional control surfaces to its caller, those control surfaces or callbacks should be provided as parameters to the composable function by the caller.

Jetpack Compose framework development and Library development MUST NOT expose any single `@Composable` function that both emits tree nodes and returns a value.

Why

Emit operations must occur in the order the content is to appear in the composition. Using return values to communicate with the caller restricts the shape of calling code and prevents interactions with other declarative calls that come before it.

Do

```
// Emits a text input field element that will call into the inputState
// interface object to request changes
@Composable
fun InputField(inputState: InputState) {
// ...

// Communicating with the input field is not order-dependent
val inputState = remember { InputState() }

Button("Clear input", onClick = { inputState.clear() })

InputField(inputState)
```

Don't

```
// Emits a text input field element and returns an input value holder
@Composable
fun InputField(): UserInputState {
```

```
// ...


// Communicating with the InputField is made difficult
Button("Clear input", onClick = { TODO("???") })
val inputState = InputField()
```

Communicating with a composable by passing parameters forward affords aggregation of several such parameters into types used as parameters to their callers:

```
interface DetailCardState {
    val actionRailState: ActionRailState
    // ...
}

@Composable
fun DetailCard(state: DetailCardState) {
    Surface {
        // ...
        ActionRail(state.actionRailState)
    }
}

@Composable
fun ActionRail(state: ActionRailState) {
    // ...
}
```

For more information on this pattern, see the sections on hoisted state types in the Compose API design patterns section below.

Compose UI API structure

Compose UI is a UI toolkit built on the Compose runtime. This section outlines guidelines for APIs that use and extend the Compose UI toolkit.

Compose UI elements

A `@Composable` function that emits exactly one Compose UI tree node is called an *element*.

Example:

```
@Composable
fun SimpleLabel(
    text: String,
    modifier: Modifier = Modifier
) {
```

Jetpack Compose framework development and Library development MUST follow all guidelines in

Jetpack Compose framework development and Library development MUST follow all guidelines in this section.

Jetpack Compose app development SHOULD follow all guidelines in this section.

Elements return Unit

Elements MUST emit their root UI node either directly by calling emit() or by calling another Compose UI element function. They MUST NOT return a value. All behavior of the element not available from the state of the composition MUST be provided by parameters passed to the element function.

Why?

Elements are declarative entities in a Compose UI composition. Their presence or absence in the composition determines whether they appear in the resulting UI. Returning a value is not necessary; any means of controlling the emitted element should be provided as a parameter to the element function, not returned by calling the element function. See the, “hoisted state” section in the Compose API design patterns section of this document for more information.

Do

```
@Composable
fun FancyButton(
    text: String,
    onClick: () -> Unit,
    modifier: Modifier = Modifier
) {
```

Don't

```
interface ButtonState {
    val clicks: Flow<ClickEvent>
    val measuredSize: Size
}

@Composable
fun FancyButton(
    text: String,
    modifier: Modifier = Modifier
): ButtonState {
```

Elements accept and respect a Modifier parameter

Element functions MUST accept a parameter of type `Modifier`. This parameter MUST be named “`modifier`” and MUST appear as the first optional parameter in the element function's parameter list. Element functions MUST NOT accept multiple `Modifier` parameters.

If the element function’s content has a natural minimum size - that is, if it would ever measure with a non-zero size given constraints of minWidth and minHeight of zero - the default value of the `modifier`

parameter MUST be `Modifier` - the `Modifier` type's `companion object` that represents the empty `Modifier`. Element functions without a measurable content size (e.g. Canvas, which draws arbitrary user content in the size available) MAY require the `modifier` parameter and omit the default value.

Element functions MUST provide their modifier parameter to the Compose UI node they emit by passing it to the root element function they call. If the element function directly emits a Compose UI layout node, the modifier MUST be provided to the node.

Element functions MAY concatenate additional modifiers to the **end** of the received `modifier` parameter before passing the concatenated modifier chain to the Compose UI node they emit.

Element functions MUST NOT concatenate additional modifiers to the **beginning** of the received modifier parameter before passing the concatenated modifier chain to the Compose UI node they emit.

Why?

Modifiers are the standard means of adding external behavior to an element in Compose UI and allow common behavior to be factored out of individual or base element API surfaces. This allows element APIs to be smaller and more focused, as modifiers are used to decorate those elements with standard behavior.

An element function that does not accept a modifier in this standard way does not permit this decoration and motivates consuming code to wrap a call to the element function in an additional Compose UI layout such that the desired modifier can be applied to the wrapper layout instead. This does not prevent the developer behavior of modifying the element, and forces them to write more inefficient UI code with a deeper tree structure to achieve their desired result.

Modifiers occupy the first optional parameter slot to set a consistent expectation for developers that they can always provide a modifier as the final positional parameter to an element call for any given element's common case.

See the Compose UI modifiers section below for more details.

Do

```
@Composable
fun FancyButton(
    text: String,
    onClick: () -> Unit,
    modifier: Modifier = Modifier
) = Text(
    text = text,
    modifier = modifier.surface(elevation = 4.dp)
        .clickable(onClick)
        .padding(horizontal = 32.dp, vertical = 16.dp)
)
```

Compose UI layouts

A Compose UI element that accepts one or more `@Composable` function parameters is called a *layout*.

Example:

```
@Composable
fun SimpleRow(
    modifier: Modifier = Modifier,
    content: @Composable () -> Unit
) {
```

Jetpack Compose framework development and Library development MUST follow all guidelines in this section.

Jetpack Compose app development SHOULD follow all guidelines in this section.

Layout functions SHOULD use the name "`content`" for a `@Composable` function parameter if they accept only one `@Composable` function parameter.

Layout functions SHOULD use the name "`content`" for their primary or most common `@Composable` function parameter if they accept more than one `@Composable` function parameter.

Layout functions SHOULD place their primary or most common `@Composable` function parameter in the last parameter position to permit the use of Kotlin's trailing lambda syntax for that parameter.

Compose UI modifiers

A `Modifier` is an immutable, ordered collection of objects that implement the `Modifier.Element` interface. Modifiers are universal decorators for Compose UI elements that may be used to implement and add cross-cutting behavior to elements in an opaque and encapsulated manner. Examples of modifiers include altering element sizing and padding, drawing content beneath or overlapping the element, or listening to touch events within the UI element's bounding box.

Jetpack Compose framework development and Library development MUST follow all guidelines in this section.

Modifier factory functions

Modifier chains are constructed using a fluent builder syntax expressed as Kotlin extension functions that act as factories.

Example:

```
Modifier.preferredSize(50.dp)
    .backgroundColor(Color.Blue)
    .padding(10.dp)
```

Modifier APIs MUST NOT expose their Modifier.Element interface implementation types.

Modifier APIs MUST be exposed as factory functions following this style:

```
fun Modifier.myModifier(
    param1: ...,
```

```
    paramN: ...
): Modifier = then(MyModifierImpl(param1, ... paramN))
```

Composed modifiers

Modifiers that must take part in composition (for example, to read `CompositionLocal` values, maintain element-specific instance state or manage object lifetimes) can use the `Modifier.composed {}` API to create a modifier that is a modifier instance factory:

```
fun Modifier.myModifier(): Modifier = composed {
    val color = LocalTheme.current.specialColor
    backgroundColor(color)
}
```

Composed modifiers are composed at each point of application to an element; the same composed modifier may be provided to multiple elements and each will have its own composition state:

```
fun Modifier.modifierWithState(): Modifier = composed {
    val elementSpecificState = remember { MyModifierState() }
    MyModifier(elementSpecificState)
}

// ...
val myModifier = someModifiers.modifierWithState()

Text("Hello", modifier = myModifier)
Text("World", modifier = myModifier)
```

As a result, **Jetpack Compose framework development and Library development** SHOULD use `Modifier.composed {}` to implement composition-aware modifiers, and SHOULD NOT declare modifier extension factory functions as `@Composable` functions themselves.

Why

Composed modifiers may be created outside of composition, shared across elements, and declared as top-level constants, making them more flexible than modifiers that can only be created via a `@Composable` function call, and easier to avoid accidentally sharing state across elements.

Layout-scoped modifiers

Android's View system has the concept of LayoutParams - a type of object stored opaquely with a ViewGroup's child view that provides layout instructions specific to the ViewGroup that will measure and position it.

Compose UI modifiers afford a related pattern using `ParentDataModifier` and receiver scope objects for layout content functions:

Example

```
@Stable
interface WeightScope {
    fun Modifier.weight(weight: Float): Modifier
}

@Composable
fun WeightedRow(
    modifier: Modifier = Modifier,
    content: @Composable WeightScope.() -> Unit
) {
// ...

// Usage:
WeightedRow {
    Text("Hello", Modifier.weight(1f))
    Text("World", Modifier.weight(2f))
}
```

Jetpack Compose framework development and library development SHOULD use scoped modifier factory functions to provide parent data modifiers specific to a parent layout composable.

Compose API design patterns

This section outlines patterns for addressing common use cases when designing a Jetpack Compose API.

Prefer stateless and controlled @Composable functions

In this context, “stateless” refers to `@Composable` functions that retain no state of their own, but instead accept external state parameters that are owned and provided by the caller. “Controlled” refers to the idea that the caller has full control over the state provided to the composable.

Do

```
@Composable
fun Checkbox(
    isChecked: Boolean,
    onToggle: () -> Unit
) {
// ...

// Usage: (caller mutates optIn and owns the source of truth)
Checkbox(
    myState.optIn,
    onToggle = { myState.optIn = !myState.optIn }
)
```

Don't

```
@Composable
fun Checkbox(
    initialValue: Boolean,
    onChecked: (Boolean) -> Unit
) {
    var checkedState by remember { mutableStateOf(initialValue) }
// ...

// Usage: (Checkbox owns the checked state, caller notified of changes)
// Caller cannot easily implement a validation policy.
Checkbox(false, onToggled = { callerCheckedState = it })
```

Separate state and events

Compose's `mutableStateOf()` value holders are observable through the `Snapshot` system and can notify observers of changes. This is the primary mechanism for requesting recomposition, relayout, or redraw of a Compose UI. Working effectively with observable state requires acknowledging the distinction between *state* and *events*.

An observable *event* happens at a point in time and is discarded. All registered observers at the time the event occurred are notified. All individual events in a stream are assumed to be relevant and may build on one another; repeated equal events have meaning and therefore a registered observer must observe all events without skipping.

Observable *state* raises change *events* when the state changes from one value to a new, unequal value. State change events are *conflated;* only the most recent state matters. Observers of state changes must therefore be *idempotent;* given the same state value the observer should produce the same result. It is valid for a state observer to both skip intermediate states as well as run multiple times for the same state and the result should be the same.

Compose operates on *state* as input, not *events*. Composable functions are *state observers* where both the function parameters and any `mutableStateOf()` value holders that are read during execution are inputs.

Hoisted state types

A pattern of stateless parameters and multiple event callback parameters will eventually reach a point of scale where it becomes unwieldy. As a composable function's parameter list grows it may become appropriate to factor a collection of state and callbacks into an interface, allowing a caller to provide a cohesive policy object as a unit.

Before

```
@Composable
fun VerticalScroller(
```

```
    scrollPosition: Int,

    scrollRange: Int,
    onScrollPositionChange: (Int) -> Unit,
    onScrollRangeChange: (Int) -> Unit
) {
```

After

```
@Stable
interface VerticalScrollerState {
    var scrollPosition: Int
    var scrollRange: Int
}

@Composable
fun VerticalScroller(
    verticalScrollerState: VerticalScrollerState
) {
```

In the example above, an implementation of `VerticalScrollerState` is able to use custom get/set behaviors of the related `var` properties to apply policy or delegate storage of the state itself elsewhere.

Jetpack Compose framework and Library development SHOULD declare hoisted state types for collecting and grouping interrelated policy. The VerticalScrollerState example above illustrates such a dependency between the scrollPosition and scrollRange properties; to maintain internal consistency such a state object should clamp scrollPosition into the valid range during set attempts. (Or otherwise report an error.) These properties should be grouped as handling their consistency involves handling all of them together.

Jetpack Compose framework and Library development SHOULD declare hoisted state types as `@Stable` and correctly implement the `@Stable` contract.

Jetpack Compose framework and Library development SHOULD name hoisted state types that are specific to a given composable function as the composable function's name suffixed by, "`State`".

Default policies through hoisted state objects

Custom implementations or even external ownership of these policy objects are often not required. By using Kotlin's default arguments, Compose's `remember {}` API, and the Kotlin "extension constructor" pattern, an API can provide a default state handling policy for simple usage while permitting more sophisticated usage when desired.

Example:

```
fun VerticalScrollerState(): VerticalScrollerState =
    VerticalScrollerStateImpl()

private class VerticalScrollerStateImpl(
```

```
    scrollPosition: Int = 0,
    scrollRange: Int = 0
) : VerticalScrollerState {
    private var _scrollPosition by
        mutableStateOf(scrollPosition, structuralEqualityPolicy())

    override var scrollPosition: Int
        get() = _scrollPosition
        set(value) {
            _scrollPosition = value.coerceIn(0, scrollRange)
        }

    private var _scrollRange by
        mutableStateOf(scrollRange, structuralEqualityPolicy())

    override var scrollRange: Int
        get() = _scrollRange
        set(value) {
            require(value >= 0) { "$value must be > 0" }
            _scrollRange = value
            scrollPosition = scrollPosition
        }
}

@Composable
fun VerticalScroller(
    verticalScrollerState: VerticalScrollerState =
        remember { VerticalScrollerState() }
) {
```

Jetpack Compose framework and Library development SHOULD declare hoisted state types as interfaces instead of abstract or open classes if they are not declared as final classes.

When designing an open or abstract class to be properly extensible for these use cases it is easy to create hidden requirements of state synchronization for internal consistency that are difficult (or impossible) for an extending developer to preserve. Using an interface that can be freely implemented strongly discourages private contracts between composable functions and hoisted state objects by way of Kotlin internal-scoped properties or functionality.

Jetpack Compose framework and Library development SHOULD provide default state implementations remembered as default arguments. State objects MAY be required parameters if the composable cannot function if the state object is not configured by the caller.

Jetpack Compose framework and Library development MUST NOT use `null` as a sentinel indicating that the composable function should internally `remember {}` its own state. This can create accidental inconsistent or unexpected behavior if `null` has a meaningful interpretation for the caller and is provided to the composable function by mistake.

Do

```
@Composable
```

```
fun VerticalScroller(
    verticalScrollerState: VerticalScrollerState =
        remember { VerticalScrollerState() }
) {
```

Don't

```
// Null as a default can cause unexpected behavior if the input parameter
// changes between null and non-null.
@Composable
fun VerticalScroller(
    verticalScrollerState: VerticalScrollerState? = null
) {
    val realState = verticalScrollerState ?:
        remember { VerticalScrollerState() }
```

Default hoisted state for modifiers

The `Modifier.composed {}` API permits construction of a Modifier factory that will be invoked later. This permits the associated Modifier factory function to be a "regular" (non-`@Composable`) function that can be called outside of composition while still permitting the use of composition to construct a modifier implementation for each element it is applied to. This does not permit using `remember {}` as a default argument expression as the factory function itself is not `@Composable`.

Jetpack Compose framework and library development SHOULD provide an overload of Modifier factory functions that accept hoisted state parameters that omits the hoisted state object as a means of requesting default behavior, SHOULD NOT use null as a default sentinel to request the implementation to `remember {}` an element-instanced default, and SHOULD NOT declare the Modifier factory function as `@Composable` in order to use `remember {}` in a default argument expression.

Do

```
fun Modifier.foo() = composed {
    FooModifierImpl(remember { FooState() }, LocalBar.current)
}

fun Modifier.foo(fooState: FooState) = composed {
    FooModifierImpl(fooState, LocalBar.current)
}
```

Don't

```
// Null as a default can cause unexpected behavior if the input parameter
// changes between null and non-null.
fun Modifier.foo(
    fooState: FooState? = null
) = composed {
```

```
) = composed {
    FooModifierImpl(
        fooState ?: remember { FooState() },
        LocalBar.current
    )
}
```

Don't

```
// @Composable modifier factory functions cannot be used
// outside of composition.
@Composable
fun Modifier.foo(
    fooState: FooState = remember { FooState() }
) = composed {
    FooModifierImpl(fooState, LocalBar.current)
}
```

Extensibility of hoisted state types

Hoisted state types often implement policy and validation that impact behavior for a composable function that accepts it. Concrete and especially final hoisted state types imply containment and ownership of the source of truth that the state object appeals to.

In extreme cases this can defeat the benefits of reactive UI API designs by creating multiple sources of truth, necessitating app code to synchronize data across multiple objects. Consider the following:

```
// Defined by another team or library
data class PersonData(val name: String, val avatarUrl: String)

class FooState {
    val currentPersonData: PersonData

    fun setPersonName(name: String)
    fun setPersonAvatarUrl(url: String)
}

// Defined by the UI layer, by yet another team
class BarState {
    var name: String
    var avatarUrl: String
}

@Composable
fun Bar(barState: BarState) {
```

These APIs are difficult to use together because both the FooState and BarState classes want to be the source of truth for the data they represent. It is often the case that different teams, libraries, or modules do not have the option of agreeing on a single unified type for data that must be shared across systems

do not have the option of agreeing on a single unified type for data that must be shared across systems.

These designs combine to form a requirement for potentially error-prone data syncing on the part of the app developer.

A more flexible approach defines both of these hoisted state types as interfaces, permitting the integrating developer to define one in terms of the other, or both in terms of a third type, preserving single source of truth in their system's state management:

```
@Stable
interface FooState {
    val currentPersonData: PersonData

    fun setPersonName(name: String)
    fun setPersonAvatarUrl(url: String)
}

@Stable
interface BarState {
    var name: String
    var avatarUrl: String
}

class MyState(
    name: String,
    avatarUrl: String
) : FooState, BarState {
    override var name by mutableStateOf(name)
    override var avatarUrl by mutableStateOf(avatarUrl)

    override val currentPersonData: PersonData =
        PersonData(name, avatarUrl)

    override fun setPersonName(name: String) {
        this.name = name
    }

    override fun setPersonAvatarUrl(url: String) {
        this.avatarUrl = url
    }
}
```

Jetpack Compose framework and Library development SHOULD declare hoisted state types as interfaces to permit custom implementations. If additional standard policy enforcement is necessary, consider an abstract class.

Jetpack Compose framework and Library development SHOULD offer a factory function for a default implementation of hoisted state types sharing the same name as the type. This preserves the same simple API for consumers as a concrete type. Example:

```
@Stable
```

```
interface FooState {
    // ...
}

fun FooState(): FooState = FooStateImpl(...)

private class FooStateImpl(...) : FooState {
    // ...
}

// Usage
val state = remember { FooState() }
```

App development SHOULD prefer simpler concrete types until the abstraction provided by an interface proves necessary. When it does, adding a factory function for a default implementation as outlined above is a source-compatible change that does not require refactoring of usage sites.