20 THINGS YOU SHOULD NEVER DO

IN JETPACK COMPOSE

This list is for all mobile developers who use the modern UI framework Jetpack Compose for building their app's UI.
You can either use this to just learn about the mistakes to avoid them in future or to take this as a checklist and make sure your existing projects don't contain any of these mistakes.

Please note: For the sake of simplicity, the code snippets shown here only deal with the mistake shown in each case and do not claim to avoid all other mistakes as well.

Philipp Lackner
DROP TABLE

1 CALLING NON-COMPOSE CODE IN COMPOSABLE FUNCTIONS

This is a classic, but huge mistake of many Compose beginners. Whenever a composable gets recomposed (that means a state value this composable uses is changed), the composable functions gets called again. In general, composable functions can be called at any time and in any order.
If you now put non-Compose code in a composable function - that means calling a function without **`@Compose`** annotation in a function with **`@Compose`** annotation - this function could be executed a lot of times.

Take a look at the following example:

```
@Composable
fun BookingList() {
    val scope = rememberCoroutineScope()
    var bookings by remember {
        mutableStateOf<List<Booking>>(emptyList())
    }

    scope.launch {
        bookings = loadBookings()
    }

    LazyColumn {
        items(bookings) {
            // ...
        }
    }
}
```

BAD

Whenever this **`BookingList`** composable is recomposed, it will launch a new coroutine and load bookings with a long running network call. That's terrible of course.

```
@Composable
fun BookingList() {
    var bookings by remember {
        mutableStateOf<List<Booking>>(emptyList())
    }
    LaunchedEffect(true) {
        bookings = loadBookings()
    }

    LazyColumn {
        items(bookings) {
            // ...
        }
    }
}
```

GOOD

Make use of the effect handlers of Jetpack Compose such as **`LaunchedEffect`**, **`DisposableEffect`**, etc. There's a dedicated video on Philipp's YouTube channel about this topic.

2 USING MUTABLELIST AS A STATE

Misunderstanding how Compose state works, can lead to lots of bugs and unexpected behavior. Take a look at this example:

```
@Composable
fun NamesList() {
    val names by remember {
        mutableStateOf(mutableListOf<String>())
    }

    LazyColumn {
        item {
            Button(onClick = { names.add("Hans") }) {
                Text(text = "Add name")
            }
        }
        items(names) { name ->
            Text(name)
        }
    }
}
```

BAD

After a click on the button, a name will be added to the list. However, the actual list will not be recomposed, since Compose can't detect changes in mutable data types such as **`MutableList`**.

```
@Composable
fun NamesList() {
    var names by remember {
        mutableStateOf(listOf<String>())
    }

    LazyColumn {
        item {
            Button(onClick = {
                names = names + "Hans"
            }) {
                Text(text = "Add name")
            }
        }
        items(names) { name ->
            Text(name)
        }
    }
}
```

GOOD

Instead, make the state an immutable list. Whenever a state is completely changed and replaced with a new value, Compose will detect the change and recompose any composables that use that state.
You can manipulate immutable lists the same way as mutable lists, just that with every change a new instance of the list is created.

3 CREATING STATE WITH REMEMBER

A very common way of creating state in Compose is by using **`remember`**. That itself is correct, since it won't recreate the state on every recomposition due to remember. But, in real apps you should always think twice if this can't backfire, since remember will only cache the value across recompositions as long as there is no configuration change or process death involved. Take a look at this:

```
@Composable
fun LoginScreen() {
    var emailText by remember {
        mutableStateOf("")
    }

    TextField(
        value = emailText,
        onValueChange = { emailText = it }
    )
}
```

BAD

As soon as there is a configuration change here (e.g. the user rotates their screen), the **`emailText`** state will be reset to an empty string again and the text field will be empty. Quite frustrating for the user, huh?

GOOD

Instead, we recommend to save your state in **`ViewModels`** or if you really want to have the state inside your composables to use **`rememberSaveable`** which will survive configuration changes.

Inside your **`ViewModel`**, you then have the option to make use of **`SavedStateHandle`** to restore the state after process death.

```
class LoginViewModel(
    private val savedStateHandle: SavedStateHandle
): ViewModel() {

    val emailText by savedStateHandle.saveable("emailText") {
        mutableStateOf("")
    }

    fun onEmailTextChange(value: String) {
        savedStateHandle["emailText"] = value
    }
}
```

Then simply instantiate your **`ViewModel`** in the **`NavHost`** and pass down the state you need for the screen.

```
@Composable
fun AppRoot() {
    val navController = rememberNavController()
    NavHost(
        navController = navController,
        startDestination = "login"
    ) {
        composable("login") {
            val viewModel = viewModel<LoginViewModel>()
            LoginScreen(
                emailText = viewModel.emailText,
                onEmailTextChange = viewModel::onEmailTextChange
            )
        }
    }
}
```

This is how the updated **`LoginScreen`** would look like when using a **`ViewModel`**.

```
@Composable
fun LoginScreen(
    emailText: String,
    onEmailTextChange: (String) -> Unit
) {
    TextField(
        value = emailText,
        onValueChange = onEmailTextChange
    )
    // ...
}
```

4 NOT USING KEYS INSIDE A LAZY COLUMN

Whenever a list used for a **`LazyColumn`** is updated, the **`LazyColumn`** doesn't know which items changed, so it will just update and recompose all visible ones.

```
@Composable
fun NoteList(notes: List<Note>) {
    LazyColumn {
        items(notes) { note ->
            // ...
        }
    }
}
```

BAD

When the notes change, every visible composable in the **`LazyColumn`** will be recomposed.

```
@Composable
fun NoteList(notes: List<Note>) {
    LazyColumn {
        items(
            items = notes,
            key = { note ->
                note.id
            }
        ) { note ->
            // ...
        }
    }
}
```

GOOD

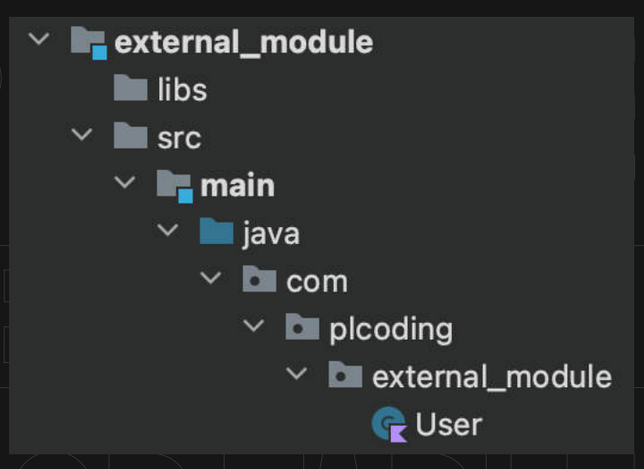
Instead, make use of the keys lambda to tell the **`LazyColumn`** how to uniquely identify each item, for example by each note's ID. That way the **`LazyColumn`** will only recompose the items that actually changed.
Bonus: You can also easily animate list changes that way with the **`animateItemPlacement()`** modifier.

5 USING UNSTABLE CLASSES FROM EXTERNAL MODULES

Compose has the concept of stability and unstability. In short, a class is marked as stable by the Compose compiler if all these conditions are true for this class:

1. The result of **`equals()`** will always return the same result for two instances
2. When a public property of the type changes, composition will be notified.
3. All public property types are stable.

However, what many people don't know: If you use classes from external modules or libraries that don't use Compose, these classes are unstable by default.



BAD

This app uses an external non-Compose module. This could also just be a normal non-Compose library added as a dependency.

```
data class User(
    val id: String,
    val name: String,
    val isAdmin: Boolean,
    val profilePictureUrl: String
)
```

Let's now assume, this module includes this **`User`** model.

```
@Composable
fun UserProfile(user: User) {
    Column {
        ProfilePicture(user.profilePictureUrl)
        Text(text = user.name)

        if(user.isAdmin) {
            Text(text = "ADMIN")
        }
    }
}
```

This **`User`** model is now used inside a composable. Normally the composable would only recompose if the user model or a field inside actually changed.
However, since classes from external non-Compose modules are unstable by default, all composables that use this user instance will be recomposed with every change of any field of the user model.

```
fun User.toComposeUser(): ComposeUser {
    return ComposeUser(id, name, isAdmin, profilePictureUrl)
}

fun ComposeUser.toUser(): User {
    return User(id, name, isAdmin, profilePictureUrl)
}
```

GOOD

Create a mapper that maps the library's user model to a user model inside your Compose module, since that will be considered stable if above's conditions are met (because the module uses Compose)

```
@Composable
fun UserProfile(user: ComposeUser) { ... }
```

Then use this Compose user model inside your composable to only let it recompose when things actually changed.

6 CONSUMING FLOWS WITH COLLECTASSTATE()

In Compose, we can transform a **`Flow`** to a Compose state with the **`collectAsState()`** function.
However, better avoid this function in Android projects, since it doesn't know anything about the lifecycle of your **`Activity`**. That means when your **`Activity`** goes in the background, the underlying **`Flow`** will still be executed, even though the user doesn't see the changes on the UI.
Note, that this does not affect **`StateFlows`** created with **`asStateFlow()`**, but rather only those created with **`stateIn()`**.

```
class CounterViewModel: ViewModel() {

    private val _counter = MutableStateFlow(0)
    val counter = _counter
        .onEach {
            saveCounterToDb(it)
        }
        .stateIn(viewModelScope, SharingStarted.WhileSubscribed(5000), 0)
}

@Composable
fun Counter() {
    val viewModel = viewModel<CounterViewModel>()
    val counter by viewModel.counter.collectAsState()

    Text(text = counter.toString())
}
```

BAD

Even, if your app is in the background, the counter **`Flow`** will be executed and will run its DB operations, which is often not what you want, if a state is only about updating the UI with some side-effects (such as the DB call).

```
// Add this dependency to build.gradle
implementation("androidx.lifecycle:lifecycle-runtime-compose:2.6.1")

@Composable
fun Counter() {
    val viewModel = viewModel<CounterViewModel>()
    val counter by viewModel.counter.collectAsStateWithLifecycle()

    Text(text = counter.toString())
}
```

GOOD

You can use **`collectAsStateWithLifecycle()`** to get a lifecycle-aware **`Flow`** collector which will not run when the app goes in the background.

7 ANIMATING TRANSFORM OUTSIDE OF GRAPHICSLAYER

When it comes to transform animations (rotation, scale, position) in Compose, there are multiple ways to do that. You can either use the transform modifiers directly or the **`graphicsLayer`** modifier. Let's see why using the modifiers directly is an issue:

```
@Composable
fun RotatingBox() {
    val transition = rememberInfiniteTransition()
    val rotationRatio by transition.animateFloat(
        initialValue = 0f,
        targetValue = 1f,
        animationSpec = infiniteRepeatable(
            animation = tween(5000),
        )
    )
    Box(modifier = Modifier
            .rotate(rotationRatio * 360f)
            .size(100.dp)
            .background(Color.Red))
}
```

BAD

Whenever **`rotationRatio`** changes (which is many times a second), the **`Box`** composable will be recomposed since it uses a state that changed (which is the rotation ratio).

```
@Composable
fun RotatingBox() {
    val transition = rememberInfiniteTransition()
    val rotationRatio by transition.animateFloat(
        initialValue = 0f,
        targetValue = 1f,
        animationSpec = infiniteRepeatable(
            animation = tween(5000),
        )
    )
    Box(modifier = Modifier
        // This is better
            .graphicsLayer {
                rotationZ = rotationRatio * 360f
            }
            .size(100.dp)
            .background(Color.Red))
}
```

GOOD

Composables shouldn't be recomposed if their appearance doesn't change. The composable will look exactly the same after a rotation, it's just a bit rotated.
Therefore, you can make use of the **`graphicsLayer`** modifier which will have the same effect, but won't cause these hundreds of recompositions.
Therefore: If clipping, transform or alpha changes -> use **`graphicsLayer`**

8 CREATING VIEWMODELS ON SCREEN LEVEL WITH HILT

`ViewModels` are the typical way to store and manage state on Android. Yet, you'll see many examples online where a **`ViewModel`** is instantiated and used like this:

```
@Composable
fun LoginScreen() {
    val viewModel: LoginViewModel = hiltViewModel()
    val state = viewModel.state
    Column {
        if(state.isLoading) {
            CircularProgressIndicator()
        }
        Button(onClick = { viewModel.login() }) {
            Text(text = "Login")
        }
    }
}
```

BAD

As soon as your **`ViewModel`** has injected dependencies in its constructor, this will break the preview and isolated UI tests, since this screen can't be used in isolation. The reason is that in order to use this screen, it always needs context to the rest of the application and the Dagger-Hilt modules to create an instance of the **`ViewModel`**.

```
sealed interface LoginEvent {
    object Login: LoginEvent
    // More events...
}

class LoginViewModel: ViewModel() {

    var state by mutableStateOf(LoginState())
        private set

    fun onEvent(event: LoginEvent) {
        when(event) {
            is LoginEvent.Login -> { /* Handle login */ }
            // ...
        }
    }
}
```

GOOD

We recommend to structure your **`ViewModels`** like on the left for Compose projects.
Each **`ViewModel`** holds a screen **`state`** instance and exposes an **`onEvent`** function which receives the different UI actions a user could perform on the screen.

```
@Composable
fun LoginScreen(
    state: LoginState,
    onEvent: (LoginEvent) -> Unit
) {
    Column {
        if(state.isLoading) {
            CircularProgressIndicator()
        }
        Button(onClick = { onEvent(LoginEvent.Login) }) {
            Text(text = "Login")
        }
        // ...
    }
}
```

Then, instead of passing the **`ViewModel`** instance to the screen, you just pass down the **`state`** and an **`onEvent`** lambda, so you can easily instantiate this screen for your preview and UI tests without having a **`ViewModel`** instance.

```
@Composable
fun LoginScreenRoot() {
    val viewModel = hiltViewModel<LoginViewModel>()
    LoginScreen(
        state = viewModel.state,
        onEvent = viewModel::onEvent
    )
}
```

After that, you just need to create an extra composable which wraps around each screen where you can safely instantiate a **`ViewModel`**, since this screen doesn't need a preview or isolated UI tests.

9 SETTING EXPANDING SIZES IN SUB-COMPOSABLES

`Modifiers` in Compose are powerful, yet they are also often misused. One major goal of Compose should be to make your components as reusable as possible. However, you often see something like this:

```
@Composable
fun MyButton(
    onClick: () -> Unit,
    modifier: Modifier = Modifier
) {
    Button(
        onClick = onClick,
        modifier = modifier
            .clip(RoundedCornerShape(100))
            .fillMaxWidth()
    ) {
        Text(text = "Cool button")
    }
}
```

BAD

We recommend to avoid using expanding size modifiers at the outermost composable for reusable composables.
In the example at the left, you can see the **`fillMaxWidth()`** modifier being used on a styled button which might be reused across the app.
However, the **`fillMaxWidth()`** modifier forces every single button to always fill the whole width of the parent composable which prevents you from placing two buttons next to each other without workarounds.

```
@Composable
fun MyButton(
    onClick: () -> Unit,
    modifier: Modifier = Modifier
) {
    Button(
        onClick = onClick,
        modifier = modifier
            .clip(RoundedCornerShape(100))
    ) {
        Text(text = "Cool button")
    }
}
```

GOOD

Better don't hardcode such sizes in the root modifiers of a reusable composable, but rather pass them from the outside to keep it flexible.
You can set fixed sizes to such composables, if the size is a core characteristic of this composable and every instance of it will always look the same.

```
MyButton(
    onClick = { /*TODO*/ },
    modifier = Modifier
        .fillMaxWidth() // <- better
)
```

The modifier can now easily be applied for each button individually.

10 NOT USING REMEMBER FOR HEAVY COMPUTATIONS

The **`remember`** function can be used to cache a computed value across recompositions. Pretty useful, right? Yet, many people don't use it for heavy computations which causes a low performance of your app and UI:

```kotlin
@Composable
fun EncryptedImage(
    encryptedBytes: ByteArray,
    modifier: Modifier = Modifier
) {
    val decryptedBytes = CryptoManager.decrypt(encryptedBytes)
    val bitmap = BitmapFactory.decodeByteArray(decryptedBytes, 0, decryptedBytes.size)
    Image(
        bitmap = bitmap.asImageBitmap(),
        contentDescription = null,
        modifier = modifier
    )
}
```

BAD

Here, the bytes will be decrypted on every single recomposition which is a lot of computation effort.

```kotlin
@Composable
fun EncryptedImage(
    encryptedBytes: ByteArray,
    modifier: Modifier = Modifier
) {
    val bitmap = remember(encryptedBytes) {
        val decryptedBytes = CryptoManager.decrypt(encryptedBytes)
        BitmapFactory.decodeByteArray(decryptedBytes, 0, decryptedBytes.size)
    }
    Image(
        bitmap = bitmap.asImageBitmap(),
        contentDescription = null,
        modifier = modifier
    )
}
```

GOOD

Better use **`remember`** with a key, so that it gets recomputed when the key changes.
Note, that this is just an example to illustrate this and something like decrypting should happen in a separate class and be called from your **`ViewModel`**. This can be applied to heavy UI related computations as well though (e.g. evaluating a complex condition)

11 OVERUSING HARDCODED DP UNITS

Sometimes, we have composable that just have a fixed size. But very often we also don't. In these cases, avoid using hardcoded DP values for a composable's dimensions and switch to relative sizes using modifiers such as **`fillMaxSize()`**, **`weight()`** or **`widthIn()`**.

```kotlin
@Composable
fun LoginScreen(state: LoginState) {
    Column {
        TextField(
            value = state.email,
            onValueChange = { /* ... */ },
            modifier = Modifier.width(300.dp)
        )
    }
}
```

BAD

This text field will have a width of **`300dp`** on every screen size. Just because it looks fine on the one device you tested it on doesn't mean it'll look find on smaller, larger or tablet devices.

```kotlin
@Composable
fun LoginScreen(state: LoginState) {
    Column {
        TextField(
            value = state.email,
            onValueChange = { /* ... */ },
            modifier = Modifier
                .widthIn(max = 400.dp)
                .fillMaxWidth()
        )
    }
}
```

GOOD

Instead, make use of relative sizes (which you can also combine with a max fixed size like on the left).
This text field will now fill the whole width on phones while not stretching over the whole screen on tablets which would look weird.
On tablets it'll have a fixed size of **`400dp`**.

12 FORGETTING ABOUT TOUCH TARGET SIZE

When creating clickable composables we have to take care of the touch target size. If your composable is a small one, it might be hard for the user to click on it since it doesn't have a big area. Consider this when creating such composables.

```
@Composable
fun OptionsButton(
    onClick: () -> Unit,
    modifier: Modifier = Modifier
) {
    Icon(
        imageVector = Icons.Default.Menu,
        contentDescription = "Options",
        modifier = modifier
            .clickable { onClick() }
    )
}
```

BAD

An icon is usually smaller than someone's finger which is why clicking on it can be frustrating due to its small surface.

```
@Composable
fun OptionsButton(
    onClick: () -> Unit,
    modifier: Modifier = Modifier
) {
    IconButton(
        onClick = onClick,
        modifier = modifier
    ) {
        Icon(
            imageVector = Icons.Default.Menu,
            contentDescription = "Options",
        )
    }
}
```

GOOD

Instead, make use of **`IconButtons`**, since they already take care of this large enough touch target size.
Of course, this doesn't only count for clickable icons, but for any small clickable composable.

13 NOT CHECKING VIEW DECOMPOSITION STRATEGY IN FRAGMENTS

Compose offers great interoperability with XML. However, when using a **`ComposeView`** inside a **`Fragment`**, make sure to set the correct view decomposition strategy to make sure the composition is properly disposed. By default, the composition is disposed when the underlying **`ComposeView`** is detached from the window which is desired for pure Jetpack Compose apps. However, when adding Compose incrementally, this might not be what you want.

```
class LoginFragment : Fragment() {

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        return ComposeView(requireContext()).apply {
            setContent {
                // Composable screen
            }
        }
    }
}
```

BAD

Just using a **`ComposeView`** like this won't make sure that the composition is fully bound to the **`Fragment`**'s lifecycle which can lead to state loss under certain circumstances.

```
return ComposeView(requireContext()).apply {
    setViewCompositionStrategy(
      ViewCompositionStrategy.DisposeOnViewTreeLifecycleDestroyed
    )
    setContent {
        // Composable screen
    }
}
```

GOOD

Instead, set the view decomposition strategy to the one from the left to tie the composition to the **`Fragment`**'s lifecycle. Note, that this is only required in **`Fragments`**.

14 MIXING STATE NAMING

In UI development, there are 2 ways how you can create and name your state:

1. You name it based on what kind of impact it will have on the UI
2. You name it based on what logical behavior it represents

Which approach you choose is up to you, but it's important to stay consistent with one approach. Take a lookg at these examples:

```
data class LoginState(
    val emailText: String = "",
    val isProgressBarVisible: Boolean = false,
    val isLoginFailed: Boolean = false
)
```

BAD

`isProgressBarVisible` is named based on (1), since it describes exactly the impact it will have on the UI (that it will show or hide a progress bar).
`isLoginFailed` on the other hand describes a logical behavior, which doesn't directly reveals what kind of impact it will have on the UI.
The example on the left uses inconsistent naming, since it uses both approaches for state naming.

```
data class LoginState(
    val emailText: String = "",
    val isLoggingIn: Boolean = false,
    val isLoginFailed: Boolean = false
)
```

GOOD

Better stay consistent. In this example, rename the **`isProgressBarVisible`** state to **`isLoggingIn`**, so it also reflects behavior.
That way, each composable can decide for itself what it wants to do with this state.

15 FORGETTING ABOUT MAKING COLUMNS SCROLLABLE

Sometimes, your screen consists of many composable put in a normal **`Column`**. Even though, it's a static number of composables with a fix size, you should consider making this screen scrollable. Just because everything fits on the screen on your test device doesn't mean everything will fit on smaller devices. Therefore, make sure to keep your screens scrollable as soon as they have a reasonable size.

```
@Composable
fun AgendaScreen() {
    Column {
        // Lots of composables
    }
}
```

BAD

If you have lots of composables on a screen, they might not fit on a smaller screen.

```
@Composable
fun AgendaScreen() {
    Column(
        modifier = Modifier
            .verticalScroll(rememberScrollState())
    ) {
        // Lots of composables
    }
}
```

GOOD

You don't do anything wrong with making content scrollable. Therefore, consider adding the **`verticalScroll()`** modifier to your outer **`Columns`** of a screen.

16 NOT NAMING LAMBDAS IN COMPOSABLES

Lambdas play a major role in Jetpack Compose to react to user actions and hoist state. However, Kotlin's feature to inline the last lambda function of a function's parameter list can make your code unreadable.

```
@Composable
fun LoginScreen() {
    VerticalScrollContainer(
        content = {
            // ...
        }
    ) {

    }
}
```

BAD

Can you tell when the last lambda function is called and what it is used for?
Unlikely, therefore make sure to name your lambdas if it's not very obvious.

```
@Composable
fun LoginScreen() {
    VerticalScrollContainer(
        content = {
            // ...
        },
        onScroll = { scrollOffset ->

        }
    )
}
```

GOOD

Now it should be all clear what the trailing lambda was used for :)
If a lambda is used to put composable content it's often clear when it's a trailing lambda. If it's used as a callback, better use named parameters.

17 MISUSING REMEMBERCOROUTINESCOPE

With **`rememberCoroutineScope()`**, we can get a coroutine scope that is aware of the current composition. Yet, many people use it the wrong way:

```
@Composable
fun LoginScreen(
    viewModel: LoginViewModel,
    navController: NavController
) {
    val scope = rememberCoroutineScope()
    Button(onClick = {
        scope.launch {
            val result = viewModel.login() // Suspending function
            if(result == Result.SUCCESS) {
                navController.navigate("home")
            }
        }
    }) {
        Text(text = "Login")
    }
}
```

BAD

Don't use a composables coroutine scope to execute suspend functions that aren't UI related. Your **`ViewModels`** shouldn't expose suspend functions to the UI, since this coroutine scope will be cancelled after a configuration change such as a screen rotation. In that case, the login call would be cancelled as well.

```
@Composable
fun LoginScreen(
    viewModel: LoginViewModel,
    navController: NavController
) {
    LaunchedEffect(key1 = viewModel.isLoggedIn) {
        if(viewModel.isLoggedIn) {
            navController.navigate("home") {
                popUpTo("home") {
                    inclusive = false
                }
            }
        }
    }
    Button(onClick = {
        viewModel.login() // Not suspending
    }) {
        Text(text = "Login")
    }
}
```

GOOD

Instead, launch such functions inside of a **`viewModelScope`** coroutine and update a state when the suspending call was finished, such as on the left.
Only use the composable coroutine scope for executing UI-related suspend functions such as triggering an animation or showing a snackbar.

18 FREQUENTLY CHANGING STATE IN SUB-COMPOSABLE AND GRAPHICSLAYER

As we already learnt in (7), state that affects a change in composable's transform, clipping or alpha should not need a recomposition, since we can use the **`graphicsLayer`** modifier for that. However, if such frequently changing state is passed down to a sub-composable it can still cause many unwanted recompositions.

BAD

Whenever the user scrolls here, **`scrollState`** will be changed which will then trigger a recomposition of **`ListItem`**, since the state changed.

```
@Composable
fun ListScreen() {
    val scrollState = rememberScrollState()
    Column(
        modifier = Modifier
            .fillMaxSize()
            .verticalScroll(scrollState)
    ) {
        for(i in 1..50) {
            ListItem(
                alpha = scrollState.value / 50f,
                modifier = Modifier.fillMaxWidth()
            )
        }
    }
}
```

```
@Composable
fun ListItem(
    alpha: Float,
    modifier: Modifier = Modifier
) {
    Text(
        text = "List item",
        modifier = modifier
            .padding(32.dp)
            .graphicsLayer {
                this.alpha = alpha
            }
    )
}
```

GOOD

If your state is passed as the result of a lambda instead, you can make the **`ListItem`** composable skip the composition phase and move on right to the layout phase, since the lambda itself doesn't change and therefore won't trigger a recomposition.

This is only relevant when passing state down to another composable function. If you'd inline the content of **`ListItem`** in the for-loop, you wouldn't need to make this a lambda function.

```
@Composable
fun ListScreen() {
    val scrollState = rememberScrollState()
    Column(
        modifier = Modifier
            .fillMaxSize()
            .verticalScroll(scrollState)
    ) {
        for(i in 1..50) {
            ListItem(
                alpha = { scrollState.value / 50f },
                modifier = Modifier.fillMaxWidth()
            )
        }
    }
}
```

```
@Composable
fun ListItem(
    alpha: () -> Float,
    modifier: Modifier = Modifier
) {
    Text(
        text = "List item",
        modifier = modifier
            .padding(32.dp)
            .graphicsLayer {
                this.alpha = alpha()
            }
    )
}
```

19 NOT USING CONTENT PADDING OF SCAFFOLD

A **`Scaffold`** is a layout that helps you to place common Android UI components correctly on the screen, such as navigation drawers, snackbars or toolbars. Yet, a common mistake is to ignore the provided **`contentPadding`** parameter from the **`Scaffold`**:

```
@Composable
fun LoginScreen() {
    Scaffold(
        modifier = Modifier.fillMaxSize()
    ) {
        Box(modifier = Modifier.fillMaxSize()) {
            // Screen content
        }
    }
}
```

BAD

When the **`Scaffold`** contains typical **`Scaffold`** elements such as a **`BottomAppBar`**, this will reduce the size of the remaining space.
To consider this remaining space in the **`Scaffold`**'s content, make sure to use its **`contentPadding`** parameter, as shown below.

```
@Composable
fun LoginScreen() {
    Scaffold(
        modifier = Modifier.fillMaxSize()
    ) { paddingValues ->
        Box(
            modifier = Modifier
                .fillMaxSize()
                .padding(paddingValues)
        ) {
            // Screen content
        }
    }
}
```

GOOD

By applying the **`paddingValues`** to the **`Box`** composable, you make sure that none of the **`Box`** content gets hidden behind **`Scaffold`** composables.

20 RETURNING IN COMPOSABLE FUNCTIONS

Avoid using the return keyword in composable functions, since this can lead to undefined behavior when the composition phase is skipped.

```
@Composable
fun LoginScreen(state: LoginState) {
    Column(
        modifier = Modifier.fillMaxSize()
    ) {
        EmailTextField(/* ... */)
        PasswordTextField(/* ... */)
        Text(
            text = state.loginError ?: return@Column,
        )
    }
}
```

BAD

Don't use return in any form inside a **@Composable** annotated function.

```
@Composable
fun LoginScreen(state: LoginState) {
    Column(
        modifier = Modifier.fillMaxSize()
    ) {
        EmailTextField(/* ... */)
        PasswordTextField(/* ... */)
        state.loginError?.let { error ->
            Text(
                text = error
            )
        }
    }
}
```

GOOD

Better use **let** or if-statements for null checks.

WHERE TO GO FROM HERE?

While this shows how incredibly complex Jetpack Compose can be, this framework can also speed up your work by a lot compared to XML. We from PL Coding definitely see Jetpack Compose not only as the future of native Android development, but as the future for building UI with Kotlin.

THE INDUSTRY-READY DEVELOPER BUNDLE

If you want to learn how you can facilitate the power of Jetpack Compose to build scalable and industry-ready apps with it, take a look at this discounted course bundle. In 3 different courses, you will learn everything you need as an industry-ready Android developer. This includes:

- Scalable, testable and understandable multi-module architecture and Compose UI
- How you can build native multiplatform apps with KMM and Compose UI for Android
- Building fully automated test pipelines for your apps to always make sure they work as expected

Get the bundle and more information about each course here:
https://pl-coding.com/premium-courses

THANKS FOR READING!

Philipp Lackner
Native Android Developer & Consultant

Philipp Lackner
DROP TABLE