

## JetPack Notes

1. The function is annotated with the `@Composable` annotation. All Composable functions must have this annotation; this annotation informs the Compose compiler that this function is intended to convert data into UI.
2. The function doesn't return anything. Compose functions that emit UI do not need to return anything, because they describe the desired screen state instead of constructing UI widgets.
3. Recomposition is the process of calling your composable functions again when inputs change. This happens when the function's inputs change. When Compose recomposes based on new inputs, it only calls the functions or lambdas that might have changed, and skips the rest. By skipping all functions or lambdas that don't have changed parameters, Compose can recompile efficiently.

Q. In views imperative approach if we have to set a new text then calling its function just write the new text in textview but in composables their is a recomposition means it redraws it fully again. so is it worthy to redrawn UI element every time their is change ? rather than just changing its single property?

Ans. **Selective Recomposition:** Compose doesn't re-render the entire UI tree from scratch when recomposition occurs. Only the composables that depend on the state that changed are re-executed. This means that if you have a complex UI with multiple components, only the parts that are directly affected by the state changes will be recomposed.

**Optimized Redrawing:** Compose leverages efficient rendering techniques. While recomposition might sound like redrawing the entire UI, the actual re-rendering on the screen is highly optimized to avoid unnecessary draw operations. So, recomposing doesn't always mean the screen is redrawn from scratch.

\*\*\*\*\* This is not satisfactory for me \*\*\*\*\*

Q. If the composable ui elements are immutable then changing its text property means a new UI element will be created , and if new one is created it will be redrawn from scratch ?

Ans. When the state changes, the framework re-invokes the composable function to reflect the new state, but this doesn't mean the entire UI is redrawn.

4. **Efficient Rendering:** When a composable like `Text` is recomposed, it doesn't mean it is completely redrawn on the screen from scratch. Compose optimizes this process:
- Compose detects changes at the composable level, not at the rendering level. If only the text changes, Compose might only re-render that text, not the surrounding layout.
  - The system can reuse the underlying rendering resources (like layouts, views, etc.) instead of recreating them. For example, changing the text doesn't mean the entire view hierarchy is recreated.

### **Immutability and Recomposition Efficiency:**

The immutability of composables simplifies UI updates by ensuring that every time the UI is recomposed, it's a fresh, clean description of the UI based on the current state. This avoids issues like UI inconsistencies, where some parts of the UI might not reflect the latest state changes (which can happen in imperative systems).

### **Conclusion:**

While composables are immutable and a "new" UI element is created on each state change, Compose is highly optimized to only recompose and re-render the parts of the UI that actually change. This makes the recomposition process efficient, even though it might initially seem like it involves a lot of unnecessary redrawing. In practice, the performance impact is minimal, and Compose's design allows for better maintainability, scalability, and correctness in UI updates.

So , recomposition means calling composable function again ... that doesn't means redrawing the ui. So if there is a change in the state (Basically data that was showed in ui) only those ui element will be redrawn that depend on that state change also it is optimised to the ui element property ... by reusing the already drawn ui element it just change what is needed not the whole ui element or not the whole layout .

Some important properties of composables:

1. Composable functions can execute in any order.
2. Composable functions can execute in parallel.
3. Recomposition skips as many composable functions and lambdas as possible.
4. Recomposition is optimistic and may be canceled.
5. A composable function might be run quite frequently, as often as every frame of an animation.

## Composable functions can execute in any order

If you look at the code for a composable function, you might assume that the code is run in the order it appears. But this isn't necessarily true. If a composable function contains calls to other composable functions, those functions might run in any order. Compose has the option of recognizing that some UI elements are higher priority than others, and drawing them first.

For example, suppose you have code like this to draw three screens in a tab layout:

```
@Composable
fun ButtonRow() {
    MyFancyNavigation {
        StartScreen()
        MiddleScreen()
        EndScreen()
    }
}
```

ThinkingInComposeSnippets.kt

The calls to `StartScreen`, `MiddleScreen`, and `EndScreen` might happen in any order. This means you can't, for example, have `StartScreen()` set some global variable (a side-effect) and have `MiddleScreen()` take advantage of that change. Instead, each of those functions needs to be self-contained.

## Composable functions can run in parallel

Compose can optimize recomposition by running composable functions in parallel. This lets Compose take advantage of multiple cores, and run composable functions not on the screen at a lower priority.

This optimization means a composable function might execute within a pool of background threads. If a composable function calls a function on a `ViewModel`, Compose might call that function from several threads at the same time.

To ensure your application behaves correctly, all composable functions should have no side-effects. Instead, trigger side-effects from callbacks such as `onClick` that always execute on the UI thread.

When a composable function is invoked, the invocation might occur on a different thread from the caller. That means code that modifies variables in a composable lambda should be avoided—both because such code is not thread-safe, and because it is an impermissible side-effect of the composable lambda.

3<sup>rd</sup> vala <https://developer.android.com/develop/ui/compose/mental-model#skips>

4<sup>th</sup> vala <https://developer.android.com/develop/ui/compose/mental-model#optimistic>

Recomposition starts whenever Compose thinks that the parameters of a composable might have changed. Recomposition is *optimistic*, which means Compose expects to finish recomposition before the parameters change again. If a parameter *does* change before recomposition finishes, Compose might cancel the recomposition and restart it with the new parameter.

When recomposition is canceled, Compose discards the UI tree from the recomposition. If you have any side-effects that depend on the UI being displayed, the side-effect will be applied even if composition is canceled. This can lead to inconsistent app state.

For overview of need of adaptive layout and version control of compose dependency (BOM) check the official site .

<https://developer.android.com/develop/ui/compose/building-adaptive-apps>

<https://developer.android.com/develop/ui/compose/bom>

These are just basics.... Not the implementation.