Jetpack Compose State

Software Studio 2022

What to expect from this lab?

1. Deeper understanding of `@Composable` and how it interact with `State`
2. Learn how to use Navigation in Compose and handle complex UI logic

1. State in Jetpack Compose

1.1 State in Compose

> State determines what is shown in the UI at any particular time

```
@Composable
fun WaterCounter(modifier: Modifier = Modifier) {
    val count = 0
    Text(
        text = "You've had $count glasses.",
        modifier = modifier.padding(16.dp)
    )
}
```

1.2 Events in Compose

> State is. Events happen.

```kotlin
@Composable
fun WaterCounter(modifier: Modifier = Modifier) {
   Column(modifier = modifier.padding(16.dp)) {
       var count = 0
       Text("You've had $count glasses.")
       Button(onClick = { count++ }, Modifier.padding(top = 8.dp)) {
           Text("Add one")
       }
   }
}
```

1.3 Memory in a composable function

> **The Composition**: a description of the UI built by Jetpack Compose when it executes composables.
> **Initial Composition**: creation of a Composition by running composables the first time.
> **Recomposition**: re-running composables to update the Composition when data changes

1.3 Memory in a composable function (cont'd)

Compose has a special state tracking system in place that schedules recompositions for any composables that read a particular state

```kotlin
@Composable
fun WaterCounter(modifier: Modifier = Modifier) {
   Column(modifier = modifier.padding(16.dp)) {
      // Changes to count are now tracked by Compose
       val count: MutableState<Int> = mutableStateOf(0)

       Text("You've had ${count.value} glasses.")
        Button(onClick = { count.value++ }, Modifier.padding(top = 8.dp)) {
           Text("Add one")
       }
   }
}
```

1.3 Memory in a composable function (cont'd)

A value calculated by `remember` is stored in the Composition during the initial composition, and the stored value is kept across recompositions.

```kotlin
@Composable
fun WaterCounter(modifier: Modifier = Modifier) {
    Column(modifier = modifier.padding(16.dp)) {
        val count: MutableState<Int> = remember { mutableStateOf(0) }
        Text("You've had ${count.value} glasses.")
        Button(onClick = { count.value++ }, Modifier.padding(top = 8.dp)) {
            Text("Add one")
        }
    }
}
```

Q1. What is the difference between Composable Function and The Composition?

- What does `@Composable` mean?
- What is **The Composition**?

What does `@Composable` mean?

> This annotation more closely resembles a language keywork. A good analogy is Kotlin's suspend keywork

```kotlin
// function declaration
suspend fun MyFun() { … }

// function type
fun MyFun(myParam: suspend () -> Unit) { … }

//===========
// function declaration
@Composable fun MyFun() { … }

// function type
fun MyFun(myParam: @Composable () -> Unit) { … }
```

What does `@Composable` mean? (cont'd)

Like suspend functions require a calling context, meaning that you can only call suspend functions inside of another suspend function. Composable works the same way

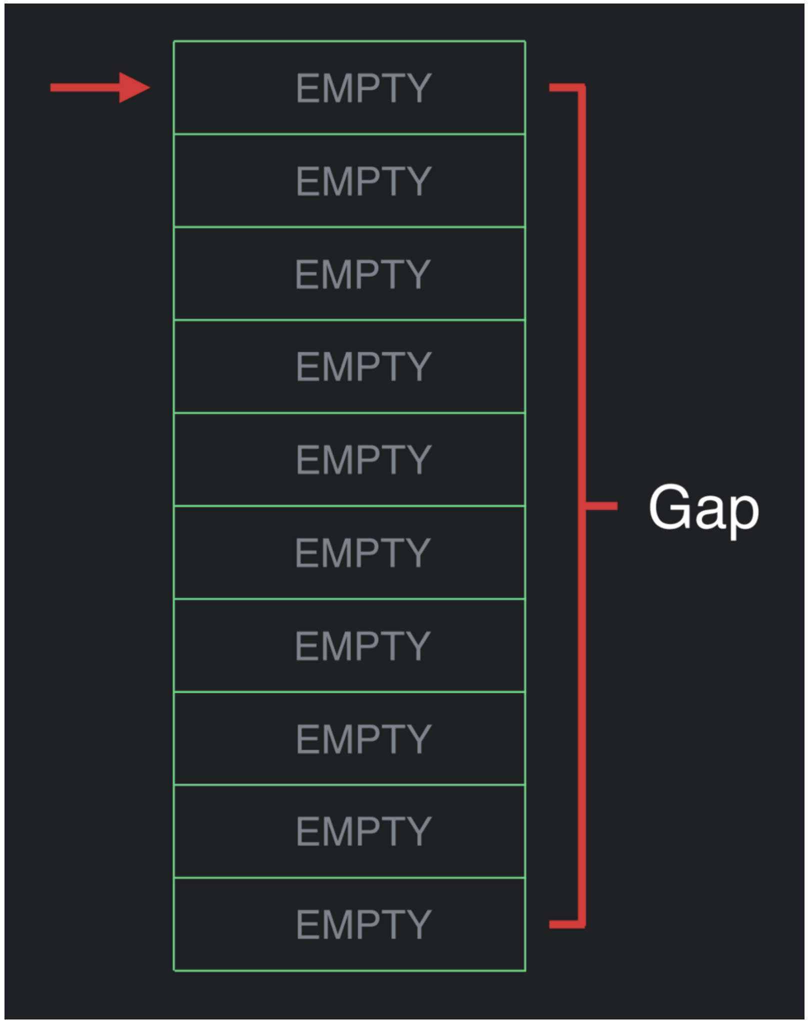
```kotlin
fun Example(a: () -> Unit, b: @Composable () -> Unit) {
   a() // allowed
   b() // NOT allowed
}
```

What does `@Composable` mean? (cont'd)

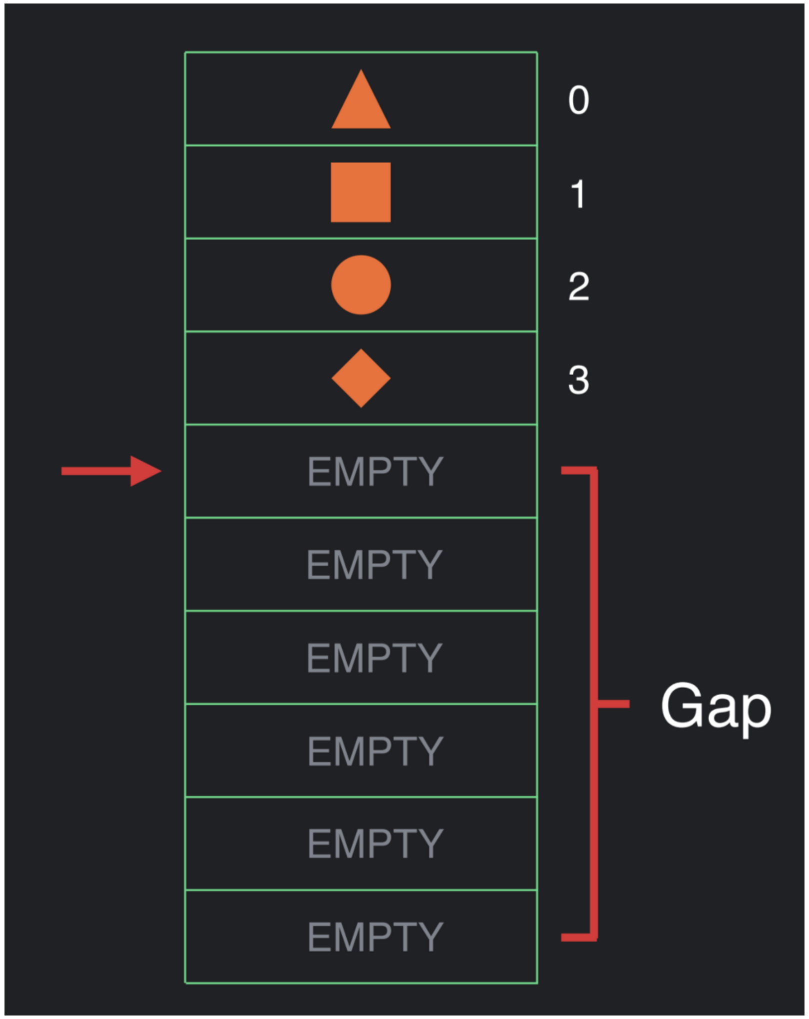
```
@Composable
fun Counter() {
 var count by remember { mutableStateOf(0) }
 Button(
   text="Count: $count",
   onPress={ count += 1 }
 )
}
```

```
// After compilers insert additional parameters and calls
fun Counter($composer: Composer) {
 $composer.start(123)
 var count by remember { mutableStateOf(0) }
 Button(
   text="Count: $count",
   onPress={ count += 1 }
 )
 $composer.end()
}
```

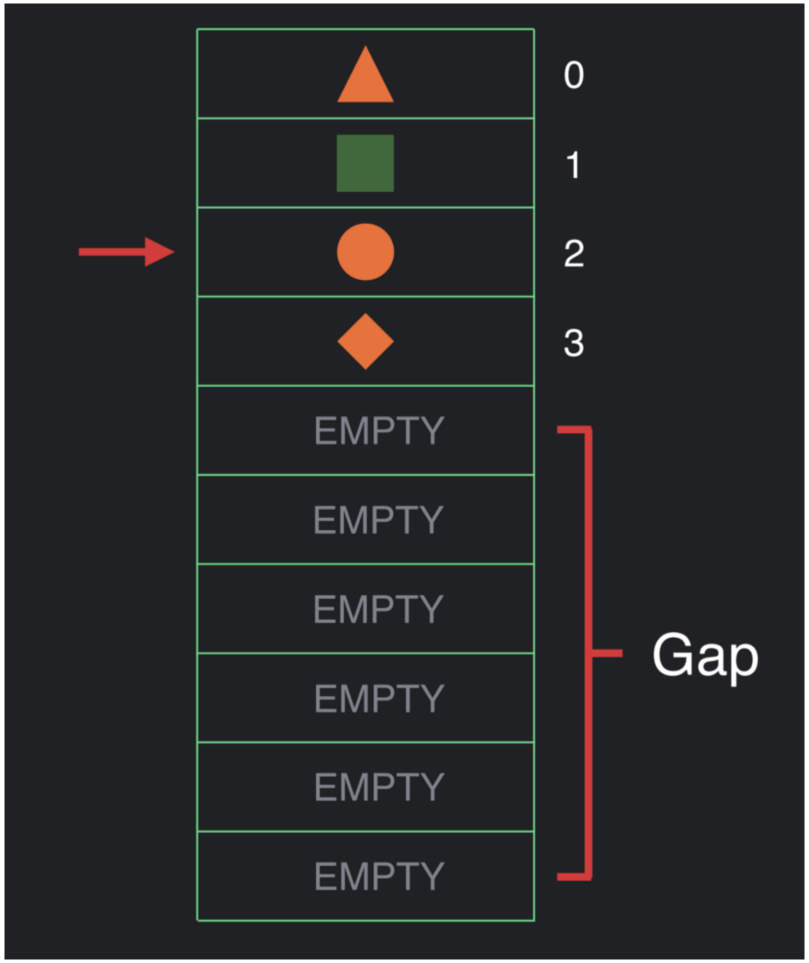
What does `@Composable` mean? (cont'd)



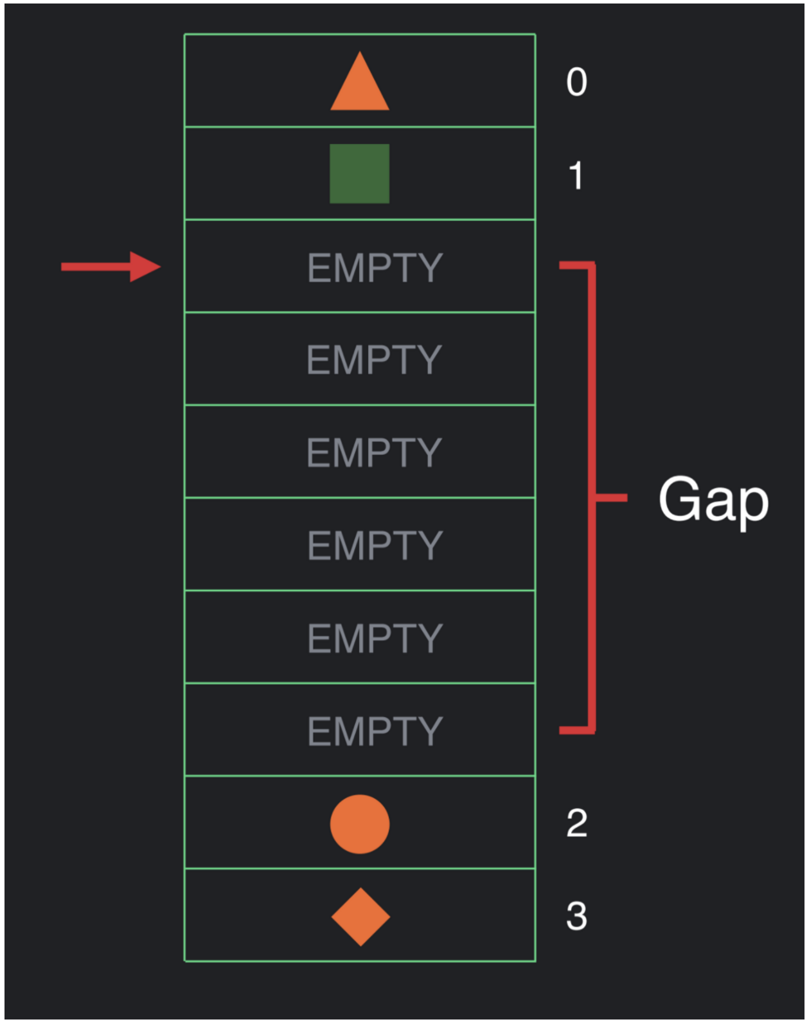
What does `@Composable` mean? (cont'd)



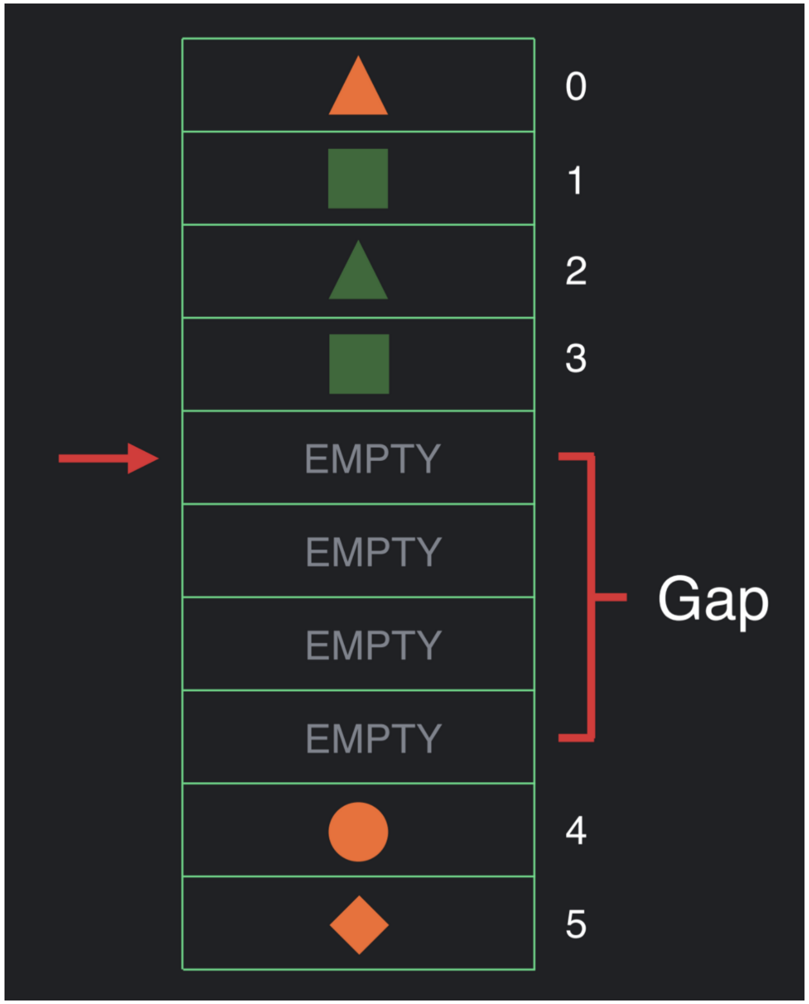
What does `@Composable` mean? (cont'd)



What does `@Composable` mean? (cont'd)

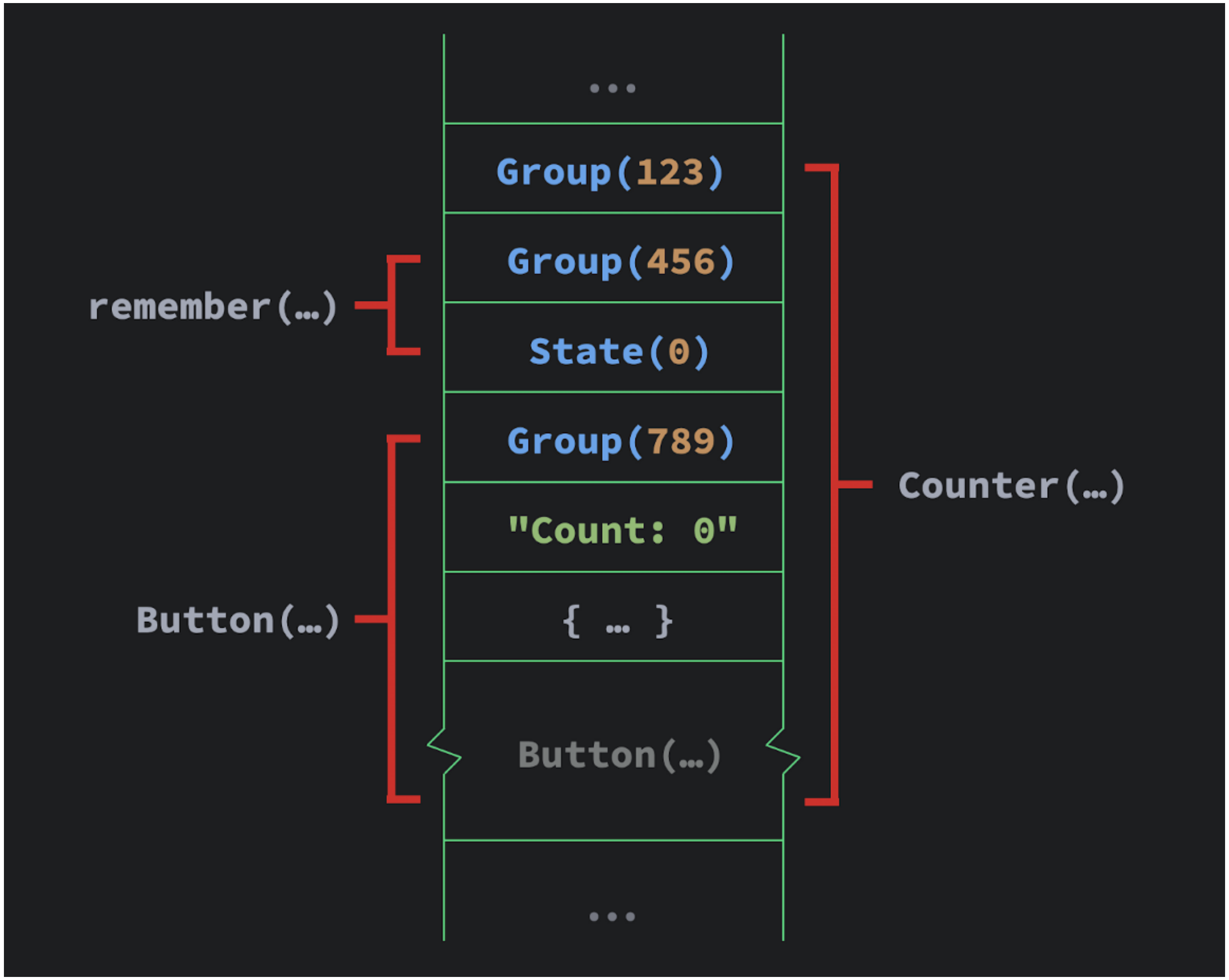


What does `@Composable` mean? (cont'd)



What does `@Composable` mean? (cont'd)

```
fun Counter($composer: Composer) {
 $composer.start(123)
 var count by remember($composer) { mutableStateOf(0) }
 Button(
   $composer,
   text="Count: $count",
   onPress={ count += 1 },
 )
 $composer.end()
}
```

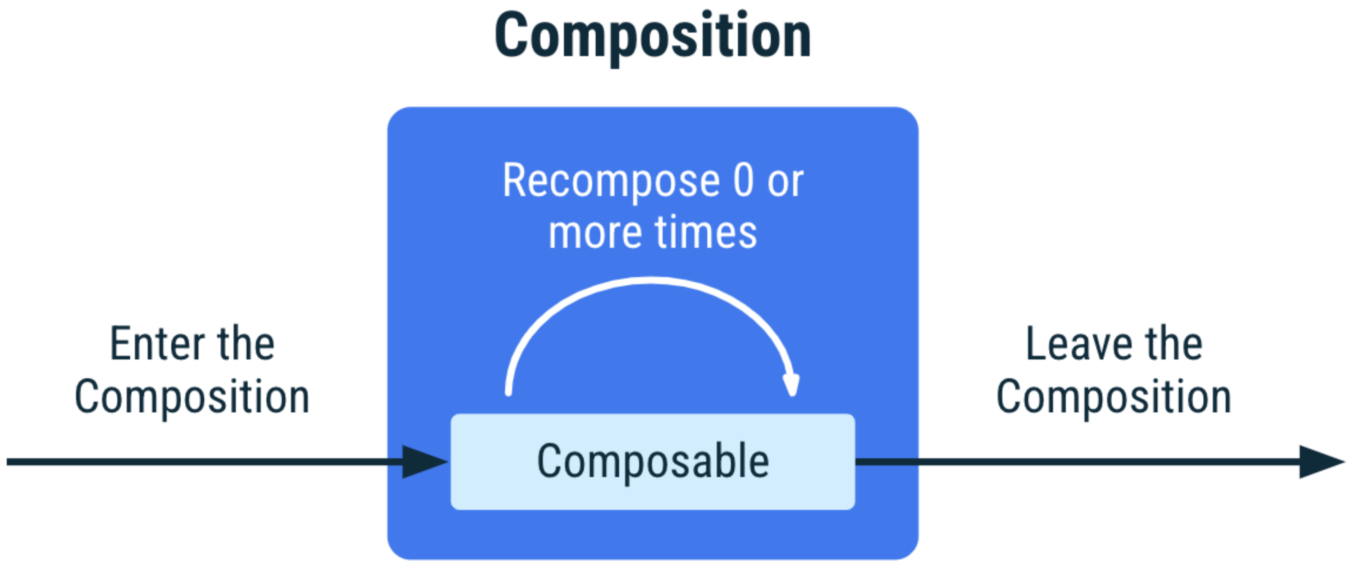

...
Group(123)
Group(456)
State(0)
Group(789)
"Count: 0"
{ … }
Button(…)
...
remember(…)
Button(…)
Counter(…)

Q1. What is the difference between Composable Function and The Composition?

- `@Composable`: a langauge keywork that tells compiler to inserts additional parameters and calls into the body of the function
- **The Composition**: a description of the UI built by Jetpack Compose when it executes composables.

1.4 State driven UI

- lifecycle of composables

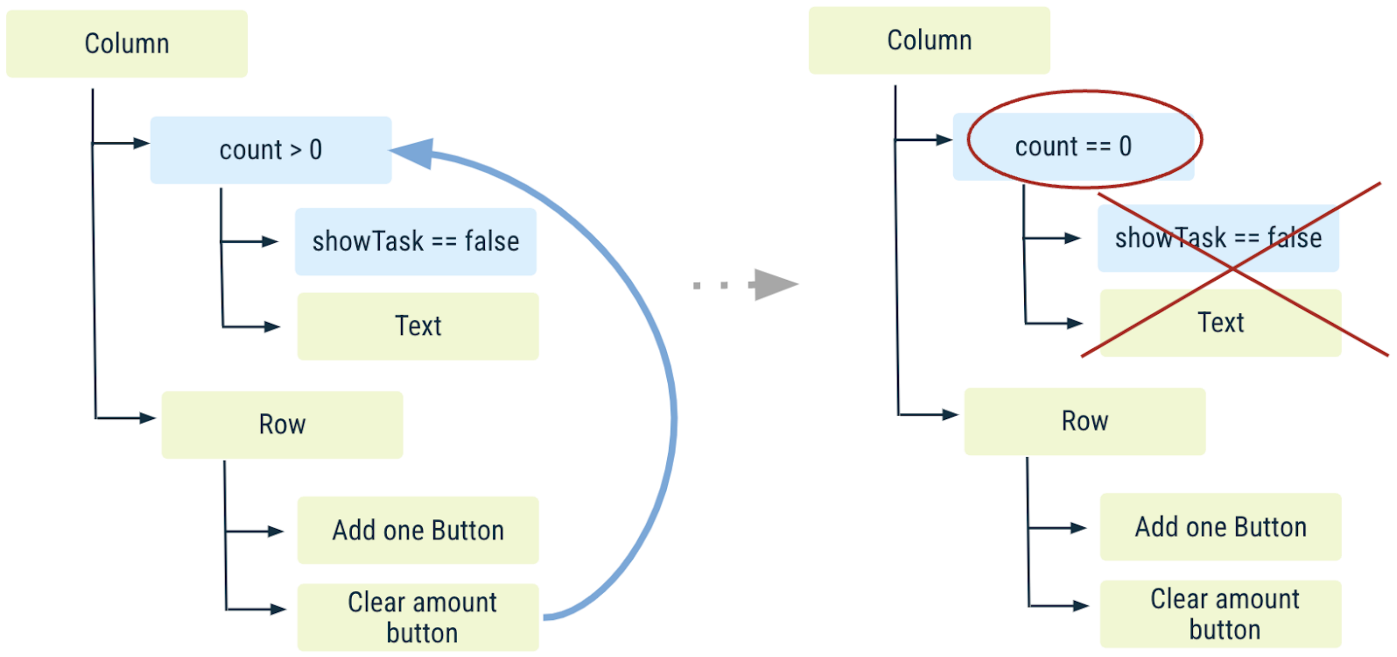


1.5 Remember in Composition

```
@Composable
fun WaterCounter(modifier: Modifier = Modifier) {
   Column(modifier = modifier.padding(16.dp)) {
       var count by remember { mutableStateOf(0) }
       if (count > 0) {
           var showTask by remember { mutableStateOf(true) }
           if (showTask) {
               WellnessTaskItem(
                   onClose = { showTask = false },
                   taskName = "Have you taken your 15 minute walk today?"
               )
           }
           Text("You've had $count glasses.")
       }

   }
}
```

1.5 Remember in Composition (cont'd)



1.6 Restore state in Compose

- remember is not retained across configuration changes
- `rememberSaveable`

1.7 State hoisting

- stateless composables
- state hoisting
- pass only the state that the composables need to avoid unnecessary recompositions

1.8 Work with lists

```kotlin
@Composable
fun WellnessTasksList(
    modifier: Modifier = Modifier,
    list: List<WellnessTask> = remember { getWellnessTasks() }
) {
    LazyColumn(
        modifier = modifier
    ) {
        items(list) { task ->
            WellnessTaskItem(taskName = task.label)
        }
    }
}

@Composable
fun WellnessTaskItem(taskName: String, modifier: Modifier = Modifier) {
   var checkedState by rememberSaveable { mutableStateOf(false) }

   WellnessTaskItem(
       taskName = taskName,
       checked = checkedState,
       onCheckedChange = { newValue -> checkedState = newValue },
       ...
   )
}
```

1.9 Observable MutableList

```
@Composable
fun WellnessScreen(modifier: Modifier = Modifier) {
   Column(modifier = modifier) {
       StatefulCounter()

       val list = remember { getWellnessTasks().toMutableStateList() }
       WellnessTasksList(list = list, onCloseTask = { task -> list.remove(task) })
   }
}

@Composable
fun WellnessTasksList(
   list: List<WellnessTask>,
   onCloseTask: (WellnessTask) -> Unit,
   modifier: Modifier = Modifier
) {
   LazyColumn(modifier = modifier) {
       items(
           items = list,
           key = { task -> task.id }
       ) { task ->
           WellnessTaskItem(taskName = task.label, onClose = { onCloseTask(task) })
       }
   }
}
```

```
@Composable
fun WellnessTaskItem(
   taskName: String, onClose: () -> Unit, modifier: Modifier = Modifier
) {
   var checkedState by rememberSaveable { mutableStateOf(false) }

   WellnessTaskItem(
     ...
   )
}
```

Q2. What will happen if we remove the code `key = { task -> task.id }`?

Q2. What will happen if we remove the code `key = { task -> task.id }`?

> By default, each item's state is keyed against the postion of the item in the list !!

1.10 State in ViewModel

```kotlin
data class WellnessTask(val id: Int, val label: String, var checked: Boolean = false)

class WellnessViewModel : ViewModel() {
  private val _tasks = getWellnessTasks().toMutableStateList()
  val tasks: List<WellnessTask>
    get() = _tasks

  fun remove(item: WellnessTask) {
    _tasks.remove(item)
  }

  fun changedTaskChecked(item: WellnessTask, checked: Boolean) {
    tasks.find { it.id == item.id }?.let { task ->
      task.checked = checked
    }
  }
}
```

- Compose is tracking for the `MutableList`, but not the state in our task. To fix it, either change the state to `MutableState` or make adjustments to the list

1.10 State in ViewModel (cont'd)

```kotlin
@Composable
fun WellnessScreen(
  modifier: Modifier = Modifier,
  wellnessViewModel: WellnessViewModel = viewModel()
) {
  Column(modifier = modifier) {
    StatefulCounter(modifier = modifier)

    WellnessTasksList(
      list = wellnessViewModel.tasks,
      onCheckedTask = { task, checked ->
        wellnessViewModel.changedTaskChecked(task, checked)
      },
      onCloseTask = { task -> wellnessViewModel.remove(task) })
  }
}
```

- `viewModel` returns an existing `ViewModel` or creates a new one in the given scope

2. Navigation & State Holder

Play around with Jetsnack and trace the code to learn how it gets implemented

1. Clone the project compose-samples from this link
2. Open the Jetsnack App and build it to play around

Trace the code and try to answer the following questions:

Q3. Where is the logic for deciding whether to show the bottom bar or not, how to implement it?

Q4. Where is the logic for alway navigate back to Home Screen after navigate up at any page?

Practice

1. Finish this codelab to learn how to use Navigation in Compose
2. Watch this great video (you can start from 06:40) to learn the best practice of state hoisting and handle complex UI state

Notes

1. You need Android Studio ChipMunk to run the project
2. If you encounter build error, you might need to change the version of "com.android.test" to "7.2.0" from "7.3.0-alpha09" in Project-Level build.gradle

```
plugins {
    ...
    id 'com.android.test' version '7.2.0' apply false
    ...
}
```

Reference

- 為什麼 remember 是 composable function? - @Composable 是什麼
- Under the hood of Jetpack Compose — part 2 of 2