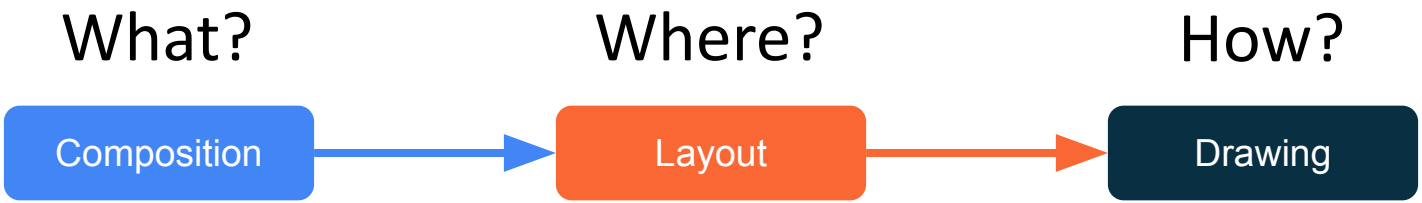
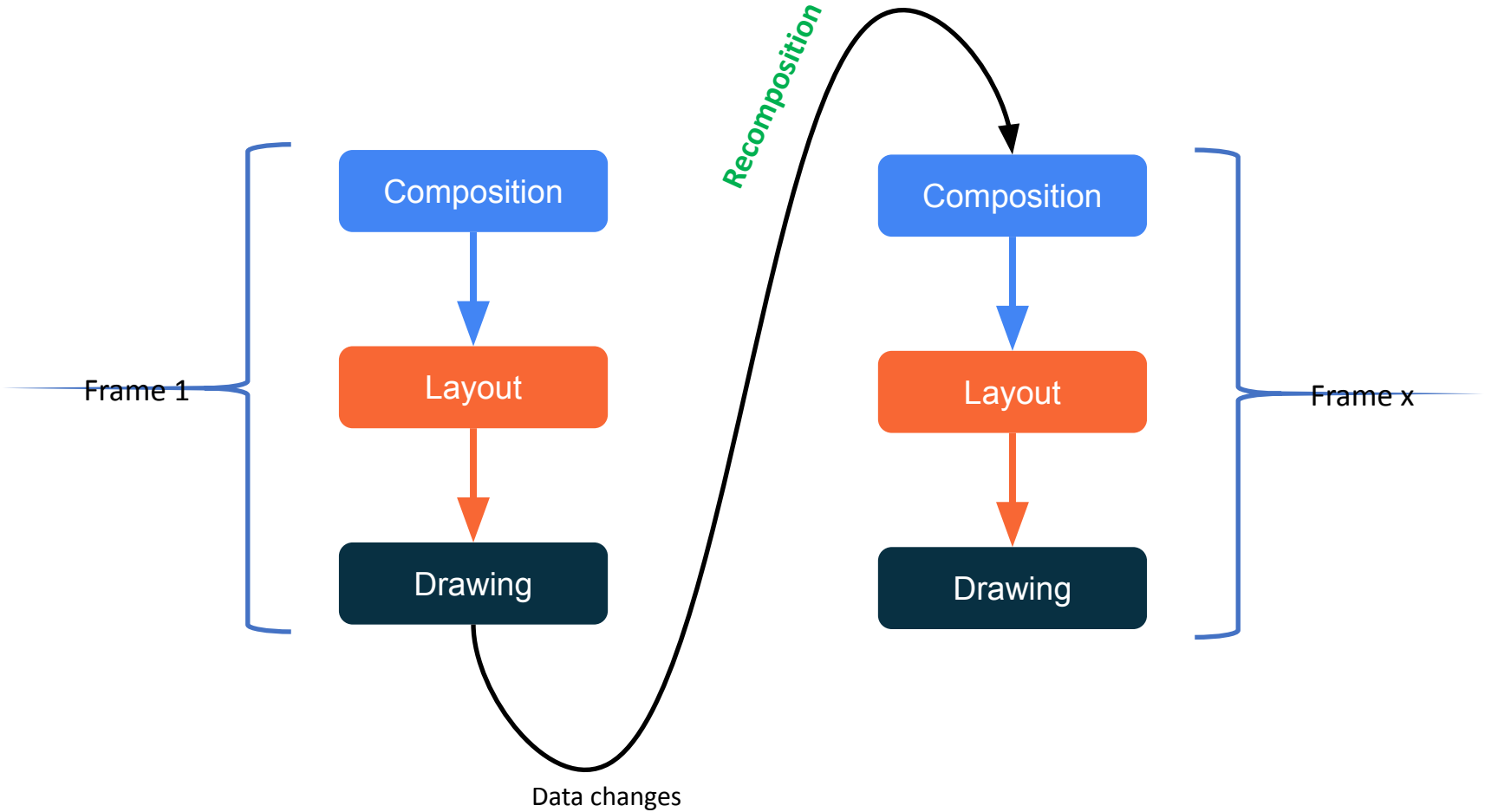
Compose Performance

Hunting for unnecessary recompositions

Philipp Nowak

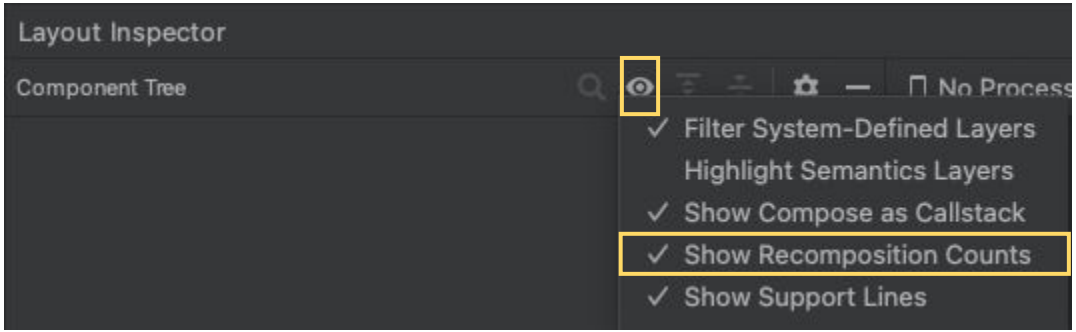
@philnowak96


What?
Where?
How?
Composition
Layout
Drawing


Recomposition
Composition
Layout
Drawing
Composition
Layout
Drawing
Frame 1
Frame x
Data changes

How to check for recompositions?

1) Add some logs to your Composables
2) Open Layout Inspector (Tools → Layout Inspector)
   - Activate “Show Recomposition Counts“ (needs Android Studio Dolphin or higher)



Read of frequently changing state

```
@Composable
fun LongListScreen(viewModel: LongListViewModel) {

}
```

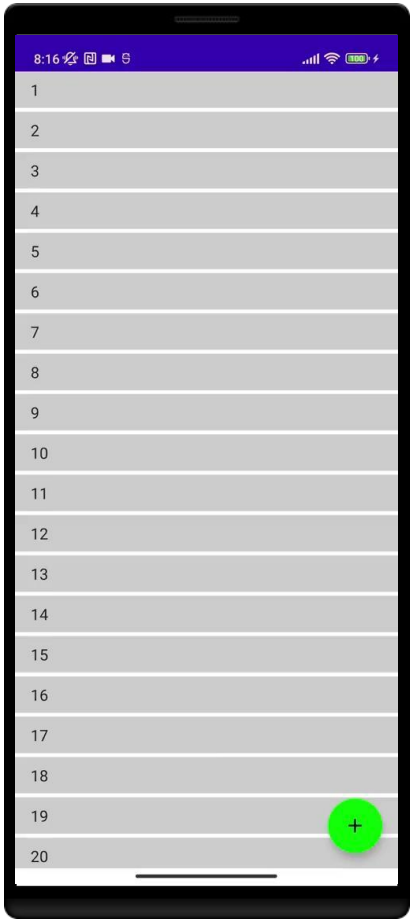
```
@Composable
fun LongListScreen(viewModel: LongListViewModel) {
    val uiState by viewModel.uiState.collectAsStateWithLifecycle()
    val listState = rememberLazyListState()
}
```

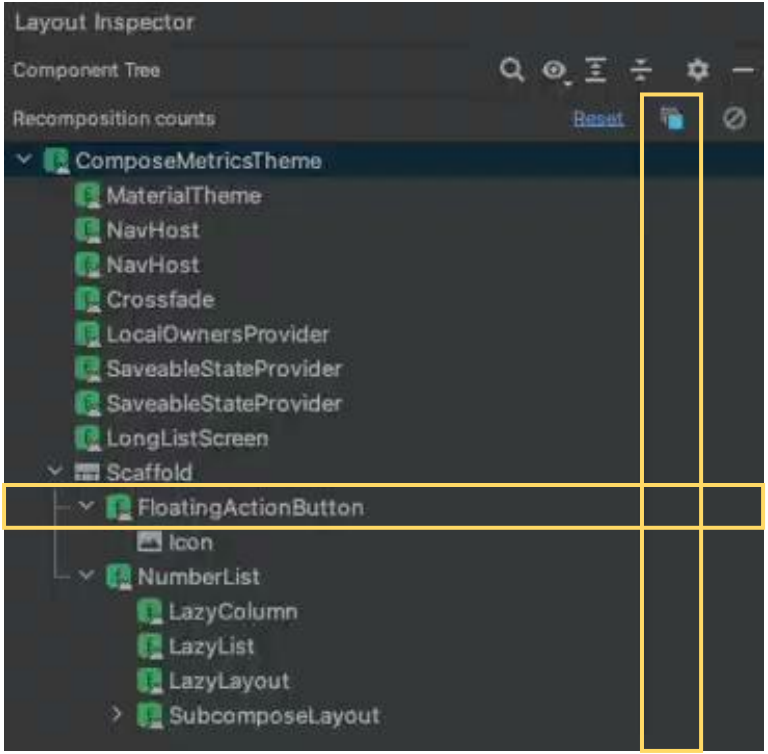
```
@Composable
fun LongListScreen(viewModel: LongListViewModel) {
    val uiState by viewModel.uiState.collectAsStateWithLifecycle()
    val listState = rememberLazyListState()

    NumberList(listState = listState, numbers = uiState.numbers, …)
}
```

```
@Composable
fun LongListScreen(viewModel: LongListViewModel) {
    val uiState by viewModel.uiState.collectAsStateWithLifecycle()
    val listState = rememberLazyListState()

    Scaffold(

    ) {
        NumberList(listState = listState, numbers = uiState.numbers, …)
    }
}
```

```
@Composable
fun LongListScreen(viewModel: LongListViewModel) {
    val uiState by viewModel.uiState.collectAsStateWithLifecycle()
    val listState = rememberLazyListState()

    Scaffold(
        floatingActionButton = {
            FloatingActionButton(backgroundColor = fabColor, onClick = {}) {
                Icon(imageVector = Icons.Default.Add, contentDescription = null)
            }
        }
    ) {
        NumberList(listState = listState, numbers = uiState.numbers, …)
    }
}
```

```
@Composable
fun LongListScreen(viewModel: LongListViewModel) {
    val uiState by viewModel.uiState.collectAsStateWithLifecycle()
    val listState = rememberLazyListState()
    val fabColor = if (listState.isScrolledToTop()) Color.Green else Color.Cyan

    Scaffold(
        floatingActionButton = {
            FloatingActionButton(backgroundColor = fabColor, onClick = {}) {
                Icon(imageVector = Icons.Default.Add, contentDescription = null)
            }
        }
    ) {
        NumberList(listState = listState, numbers = uiState.numbers, …)
    }
}
```


8:16
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
+

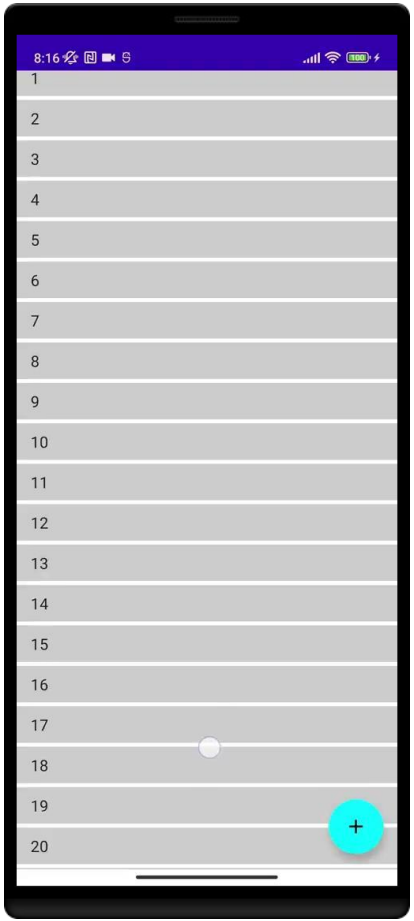

Layout Inspector
Component Tree
Recomposition counts
Reset
ComposeMetricsTheme
MaterialTheme
NavHost
NavHost
Crossfade
LocalOwnersProvider
SaveableStateProvider
SaveableStateProvider
LongListScreen
Scaffold
FloatingActionButton
Icon
NumberList
LazyColumn
LazyList
LazyLayout
SubcomposeLayout

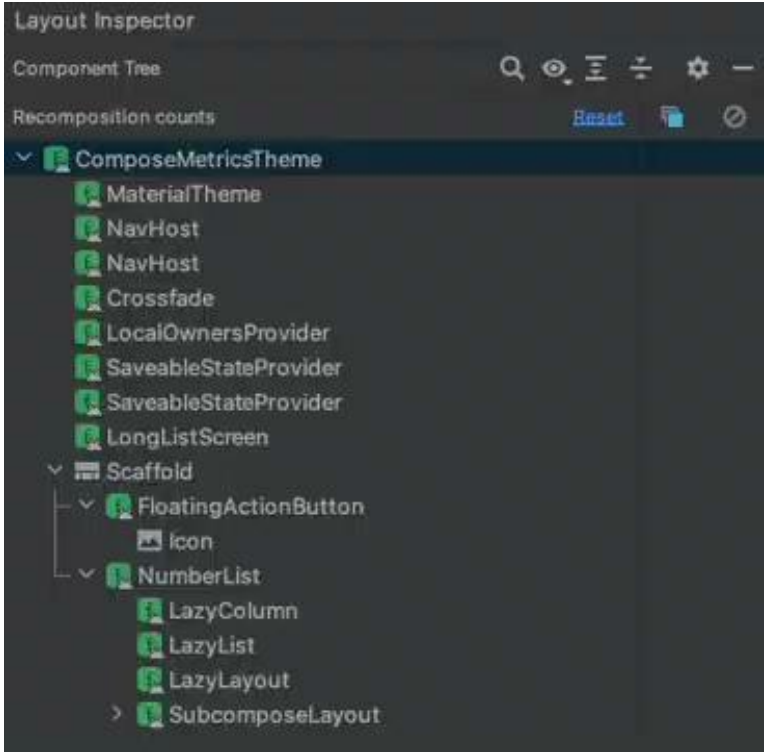
```
@Composable
fun LongListScreen(viewModel: LongListViewModel) {
    val uiState by viewModel.uiState.collectAsStateWithLifecycle()
    val listState = rememberLazyListState()
    val fabColor = if (listState.isScrolledToTop()) Color.Green else Color.Cyan

    Scaffold(
        floatingActionButton = {
            FloatingActionButton(backgroundColor = fabColor, onClick = {}) {
                Icon(imageVector = Icons.Default.Add, contentDescription = null)
            }
        }
    ) {
        NumberList(listState = listState, numbers = uiState.numbers, …)
    }
}

private fun LazyListState.isScrolledToTop() =
    firstVisibleItemScrollOffset == 0 && firstVisibleItemIndex == 0
```

```
@Composable
fun LongListScreen(viewModel: LongListViewModel) {
    val uiState by viewModel.uiState.collectAsStateWithLifecycle()
    val listState = rememberLazyListState()
    val fabColor = if (listState.isScrolledToTop()) Color.Green else Color.Cyan

    Scaffold(
        floatingActionButton = {
            FloatingActionButton(backgroundColor = fabColor, onClick = {}) {
                Icon(imageVector = Icons.Default.Add, contentDescription = null)
            }
        }
    ) {
        NumberList(listState = listState, numbers = uiState.numbers, …)
    }
}

private fun LazyListState.isScrolledToTop() =
    firstVisibleItemScrollOffset == 0 && firstVisibleItemIndex == 0
```

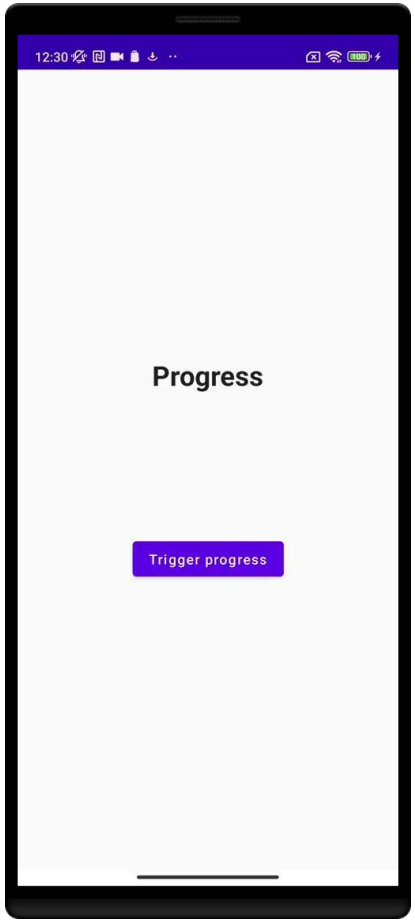
```
@Composable
fun LongListScreen(viewModel: LongListViewModel) {
    val uiState by viewModel.uiState.collectAsStateWithLifecycle()
    val listState = rememberLazyListState()
    val fabColor by remember {
        derivedStateOf { if (listState.isScrolledToTop()) Color.Green else Color.Cyan }
    }

    Scaffold(
        floatingActionButton = {
            FloatingActionButton(backgroundColor = fabColor, onClick = {}) {
                Icon(imageVector = Icons.Default.Add, contentDescription = null)
            }
        }
    ) {
        NumberList(listState = listState, numbers = uiState.numbers, …)
    }
}


private fun LazyListState.isScrolledToTop() =
    firstVisibleItemScrollOffset == 0 && firstVisibleItemIndex == 0
```

```
@Composable
fun LongListScreen(viewModel: LongListViewModel) {
    val uiState by viewModel.uiState.collectAsStateWithLifecycle()
    val listState = rememberLazyListState()
    val fabColor by remember {
        derivedStateOf { if (listState.isScrolledToTop()) Color.Green else Color.Cyan }
    }

    Scaffold(
        floatingActionButton = {
            FloatingActionButton(backgroundColor = fabColor, onClick = {}) {
                Icon(imageVector = Icons.Default.Add, contentDescription = null)
            }
        }
    ) {
        NumberList(listState = listState, numbers = uiState.numbers, …)
    }
}

private fun LazyListState.isScrolledToTop() =
    firstVisibleItemScrollOffset == 0 && firstVisibleItemIndex == 0
```


8:16
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
+


Layout Inspector
Component Tree
Recomposition counts
Reset
ComposeMetricsTheme
MaterialTheme
NavHost
NavHost
Crossfade
LocalOwnersProvider
SaveableStateProvider
SaveableStateProvider
LongListScreen
Scaffold
FloatingActionButton
Icon
NumberList
LazyColumn
LazyList
LazyLayout
SubcomposeLayout

Usage of frequently changing state in Modifiers

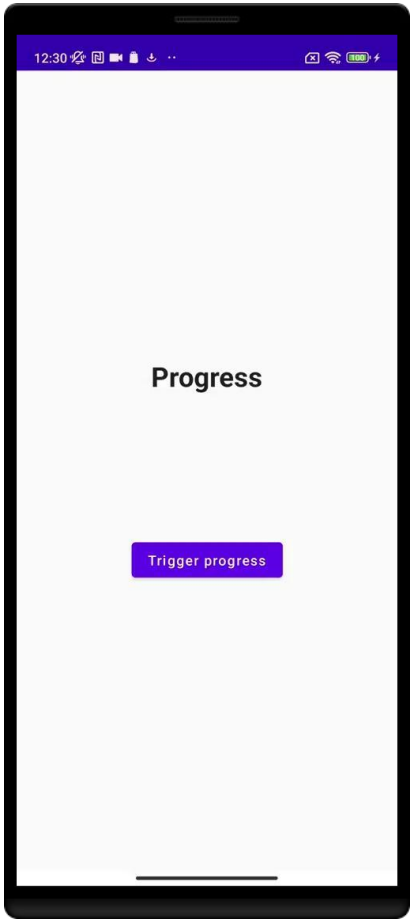

12:30
Progress
Trigger progress

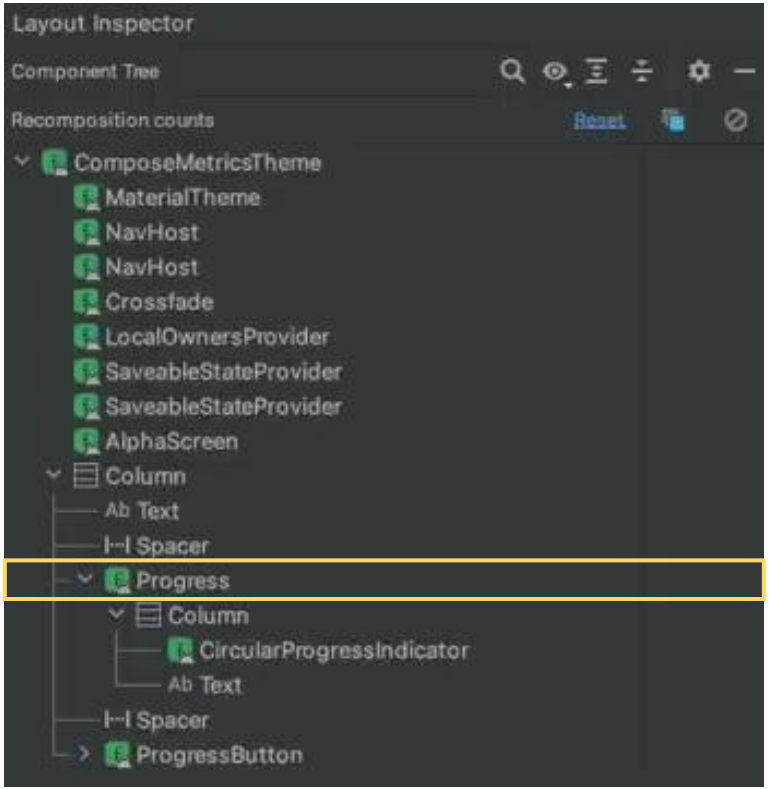
```
@Composable
fun AlphaScreen(viewModel: AlphaViewModel) {
    val uiState by viewModel.uiState.collectAsStateWithLifecycle()

    Column(…) {

    }
}
```

```
@Composable
fun AlphaScreen(viewModel: AlphaViewModel) {
    val uiState by viewModel.uiState.collectAsStateWithLifecycle()

    Column(…) {
        Text(text = "Progress“, …)
        Progress()
        ProgressButton(onTriggerProgress = …)
    }
}
```

```
@Composable
fun AlphaScreen(viewModel: AlphaViewModel) {
    val uiState by viewModel.uiState.collectAsStateWithLifecycle()

    Column(…) {
        Text(text = "Progress“, …)
        Progress()
        ProgressButton(onTriggerProgress = …)
    }
}


@Composable
private fun Progress() {
    Column(…) {
        CircularProgressIndicator()
        Text(text = "Loading...")
    }
}
```

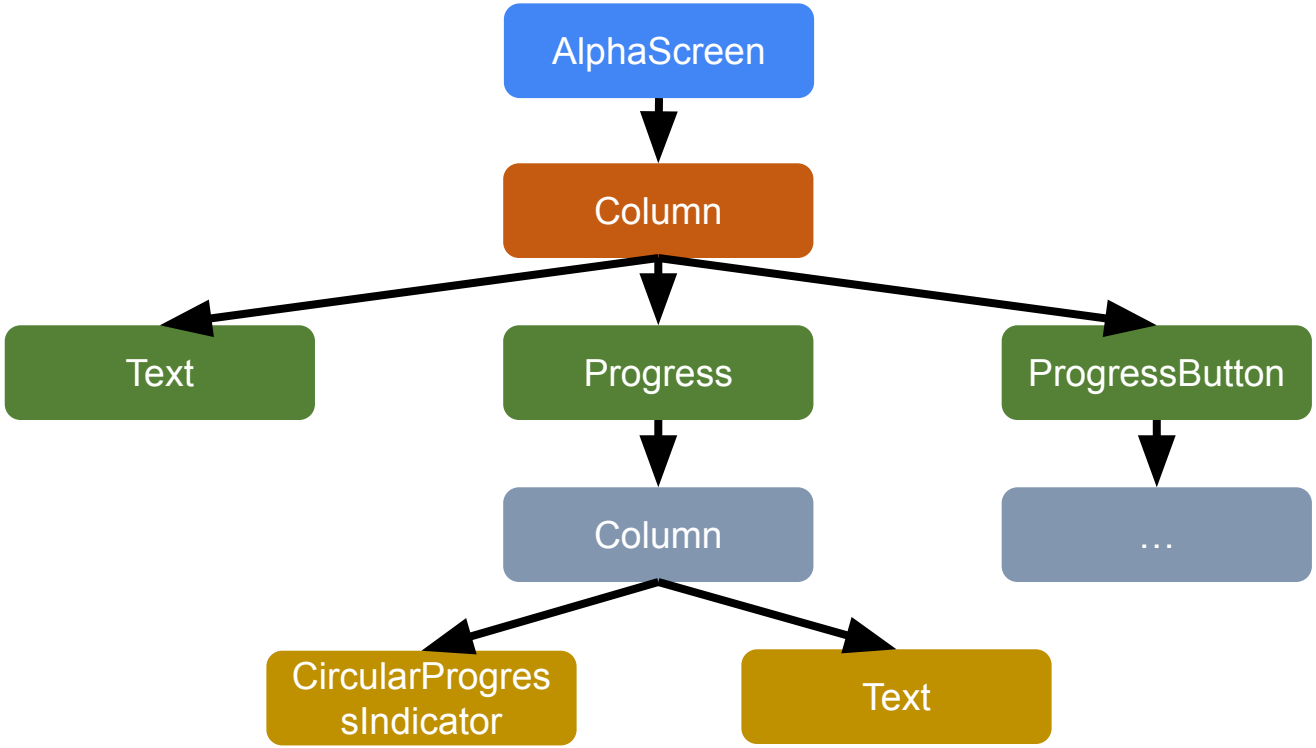
```
@Composable
fun AlphaScreen(viewModel: AlphaViewModel) {
    val uiState by viewModel.uiState.collectAsStateWithLifecycle()

    Column(…) {
        Text(text = "Progress", …)
        Progress(isProgressVisible = uiState.isProgressVisible)
        ProgressButton(onTriggerProgress = …)
    }
}


@Composable
private fun Progress(isProgressVisible: Boolean) {
    val loadingIndicatorVisibility by animateFloatAsState(
        targetValue = if (isProgressVisible) 1f else 0f, animationSpec = tween()
    )

    Column(…,
        modifier = Modifier.alpha(loadingIndicatorVisibility)
    ) {
        CircularProgressIndicator()
        Text(text = "Loading...")
    }
}
```

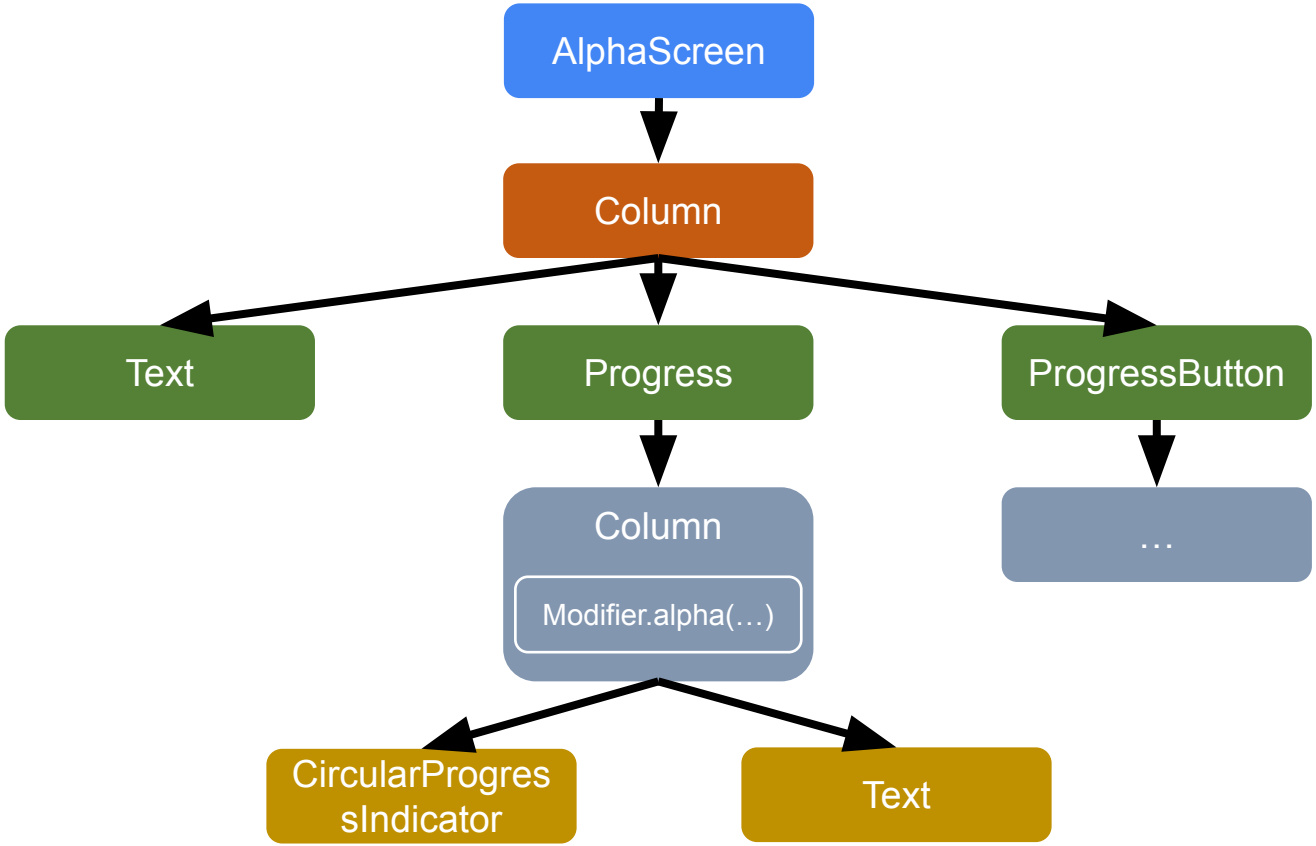

12:30
Progress
Trigger progress


Layout Inspector
Component Tree
Recomposition counts
Reset
ComposeMetricsTheme
MaterialTheme
NavHost
NavHost
Crossfade
LocalOwnersProvider
SaveableStateProvider
SaveableStateProvider
AlphaScreen
Column
Text
Spacer
Progress
Column
CircularProgressIndicator
Text
Spacer
ProgressButton

```
@Composable
fun AlphaScreen(viewModel: AlphaViewModel) {
    val uiState by viewModel.uiState.collectAsStateWithLifecycle()

    Column(…) {
        Text(text = "Progress", …)
        Progress(isProgressVisible = uiState.isProgressVisible)
        ProgressButton(onTriggerProgress = …)
    }
}


@Composable
private fun Progress(isProgressVisible: Boolean) {
    val loadingIndicatorVisibility by animateFloatAsState(
        targetValue = if (isProgressVisible) 1f else 0f, animationSpec = tween()
    )

    Column(…,
        modifier = Modifier.alpha(loadingIndicatorVisibility)
    ) {
        CircularProgressIndicator()
        Text(text = "Loading...")
    }
}
```

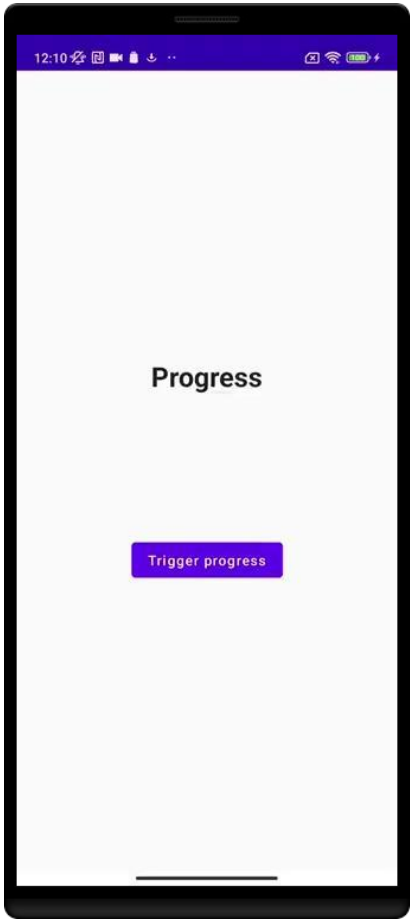
```kotlin
@Composable
fun AlphaScreen(viewModel: AlphaViewModel) {
    val uiState by viewModel.uiState.collectAsStateWithLifecycle()

    Column(…) {
        Text(text = "Progress", …)
        Progress(isProgressVisible = uiState.isProgressVisible)
        ProgressButton(onTriggerProgress = …)
    }
}


@Composable
private fun Progress(isProgressVisible: Boolean) {
    val loadingIndicatorVisibility by animateFloatAsState(
        targetValue = if (isProgressVisible) 1f else 0f, animationSpec = tween()
    )

    Column(…,
        modifier = Modifier.alpha(loadingIndicatorVisibility)
    ) {
        CircularProgressIndicator()
        Text(text = "Loading...")
    }
}
```

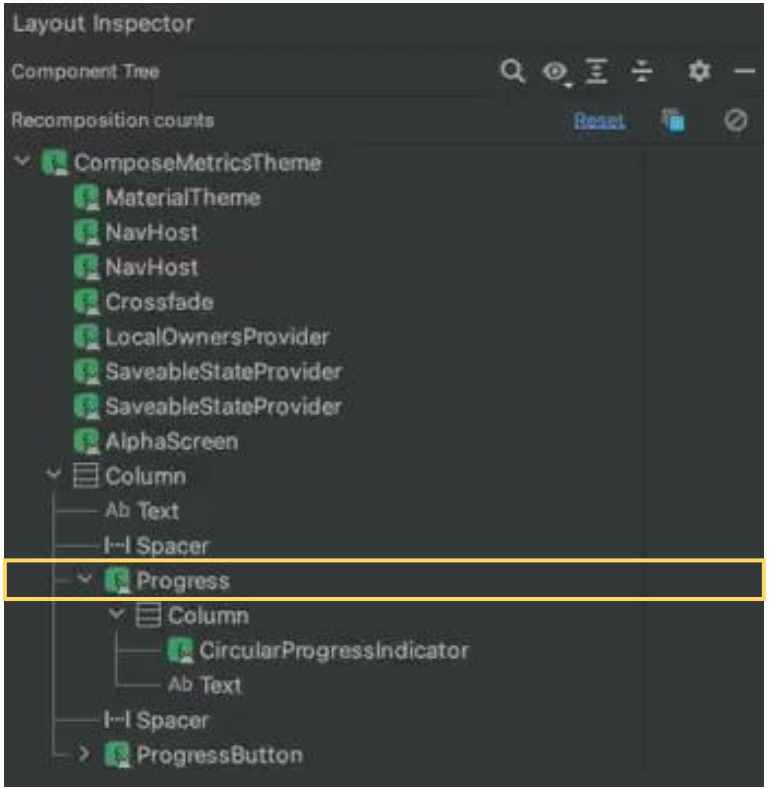

AlphaScreen
Column
Text
Progress
ProgressButton
Column
…
CircularProgres
sIndicator
Text


AlphaScreen
Column
Text
Progress
ProgressButton
Column
Modifier.alpha(…)
…
CircularProgres
sIndicator
Text

```kotlin
@Composable
fun AlphaScreen(viewModel: AlphaViewModel) {
    val uiState by viewModel.uiState.collectAsStateWithLifecycle()

    Column(…) {
        Text(text = "Progress", …)
        Progress(isProgressVisible = uiState.isProgressVisible)
        ProgressButton(onTriggerProgress = …)
    }
}


@Composable
private fun Progress(isProgressVisible: Boolean) {
    val loadingIndicatorVisibility by animateFloatAsState(
        targetValue = if (isProgressVisible) 1f else 0f, animationSpec = tween()
    )

    Column(…,
        modifier = Modifier.alpha(loadingIndicatorVisibility)
    ) {
        CircularProgressIndicator()
        Text(text = "Loading...")
    }
}
```

```
@Composable
fun AlphaScreen(viewModel: AlphaViewModel) {
    val uiState by viewModel.uiState.collectAsStateWithLifecycle()

    Column(…) {
        Text(text = "Progress", …)
        Progress(isProgressVisible = uiState.isProgressVisible)
        ProgressButton(onTriggerProgress = …)
    }
}


@Composable
private fun Progress(isProgressVisible: Boolean) {
    val loadingIndicatorVisibility by animateFloatAsState(
        targetValue = if (isProgressVisible) 1f else 0f, animationSpec = tween()
    )

    Column(…,
        modifier = Modifier.graphicsLayer { alpha = loadingIndicatorVisibility }
    ) {
        CircularProgressIndicator()
        Text(text = "Loading...")
    }
}
```


12:10
Progress
Trigger progress


Layout Inspector
Component Tree
Recomposition counts
Reset
ComposeMetricsTheme
MaterialTheme
NavHost
NavHost
Crossfade
LocalOwnersProvider
SaveableStateProvider
SaveableStateProvider
AlphaScreen
Column
Text
Spacer
Progress
Column
CircularProgressIndicator
Text
Spacer
ProgressButton

Restartable

Skippable

Restartable

- applicable for Composable functions
- scope to start recomposition
- most of the Composable functions are restartable

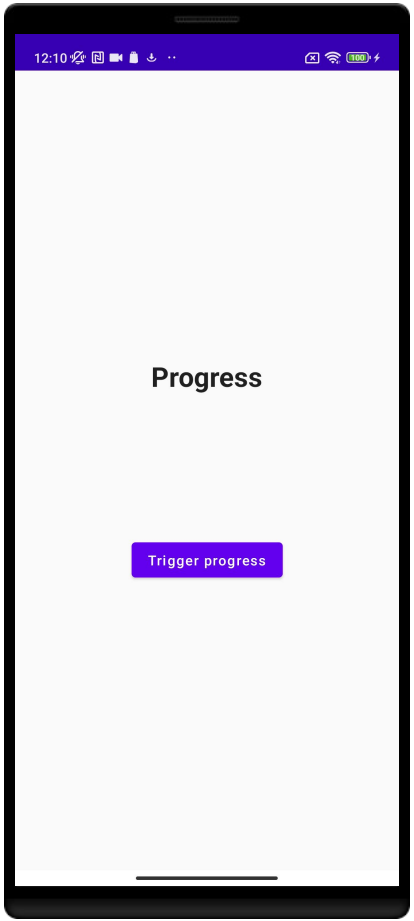
Skippable

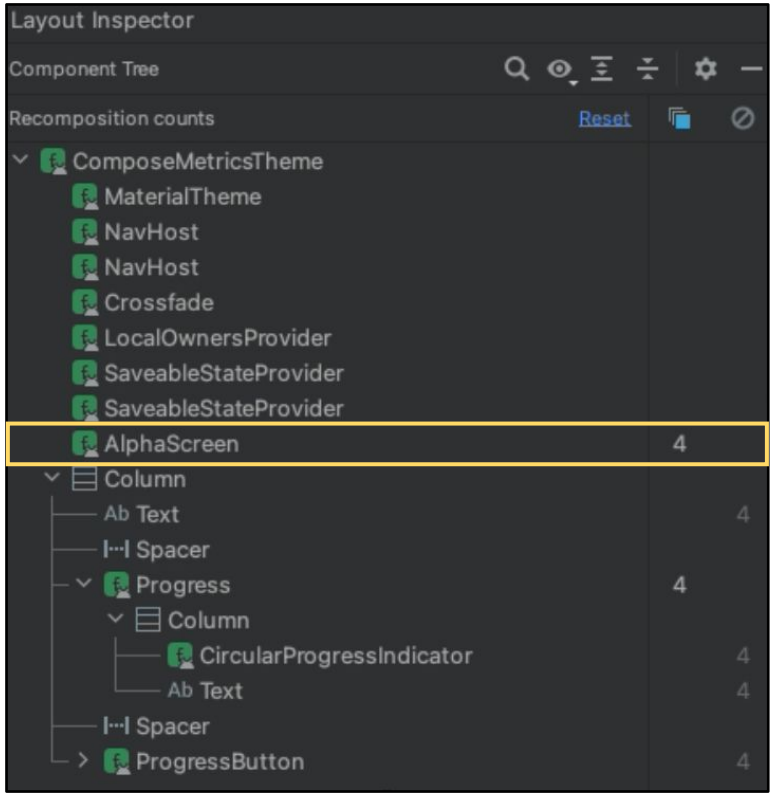
Restartable

- applicable for Composable functions
- scope to start recomposition
- most of the Composable functions are restartable

Skippable

- applicable for Composable functions
- skip re-execution during recomposition if not necessary
- not every Composable function has to be skippable


12:10
Progress
Trigger progress


Layout Inspector
Component Tree
Recomposition counts
Reset
ComposeMetricsTheme
MaterialTheme
NavHost
NavHost
Crossfade
LocalOwnersProvider
SaveableStateProvider
SaveableStateProvider
AlphaScreen 4
Column
Text 4
Spacer
Progress 4
Column
CircularProgressIndicator 4
Text 4
Spacer
ProgressButton 4

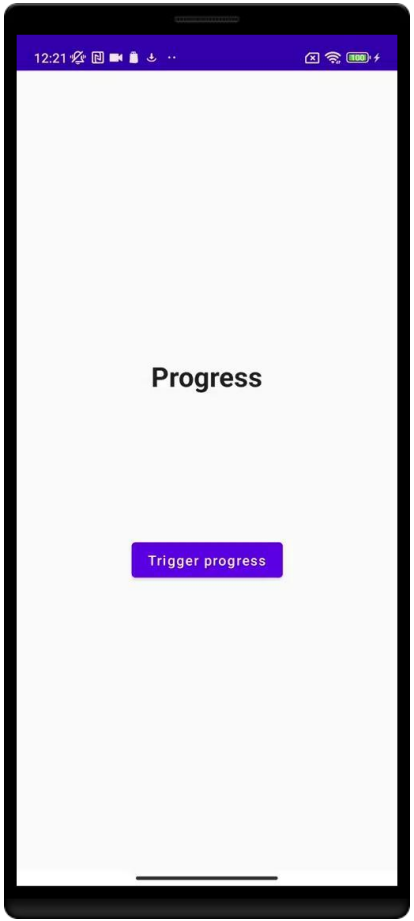
```
@Composable
fun AlphaScreen(viewModel: AlphaViewModel) {
    val uiState by viewModel.uiState.collectAsStateWithLifecycle()

    Column(…) {
        Text(text = "Progress", …)
        Progress(isProgressVisible = uiState.isProgressVisible)
        ProgressButton(onTriggerProgress = …)
    }
}


@Composable
private fun Progress(isProgressVisible: Boolean) {
    val loadingIndicatorVisibility by animateFloatAsState(
        targetValue = if (isProgressVisible) 1f else 0f, animationSpec = tween()
    )

    Column(…,
        modifier = Modifier.graphicsLayer { alpha = loadingIndicatorVisibility }
    ) {
        CircularProgressIndicator()
        Text(text = "Loading...")
    }
}
```

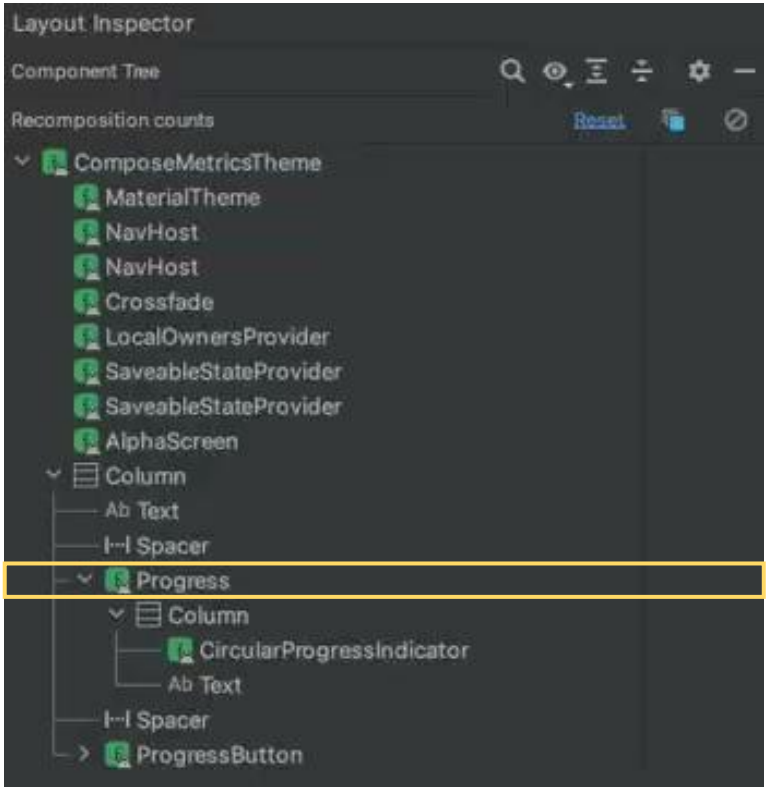
```
@Composable
fun AlphaScreen(viewModel: AlphaViewModel) {
    val uiState by viewModel.uiState.collectAsStateWithLifecycle()

    Column(…) {
        Text(text = "Progress", …)
        Progress(isProgressVisible = uiState.isProgressVisible)
        ProgressButton(onTriggerProgress = …)
    }
}


@Composable
private fun Progress(isProgressVisible: Boolean) {
    val loadingIndicatorVisibility by animateFloatAsState(
        targetValue = if (isProgressVisible) 1f else 0f, animationSpec = tween()
    )

    Column(…,
        modifier = Modifier.graphicsLayer { alpha = loadingIndicatorVisibility }
    ) {
        CircularProgressIndicator()
        Text(text = "Loading...")
    }
}
```

```
@Composable
inline fun Column(
    modifier: Modifier = Modifier,
    verticalArrangement: Arrangement.Vertical = Arrangement.Top,
    horizontalAlignment: Alignment.Horizontal = Alignment.Start,
    content: @Composable ColumnScope.() → Unit
) {
    …
}
```

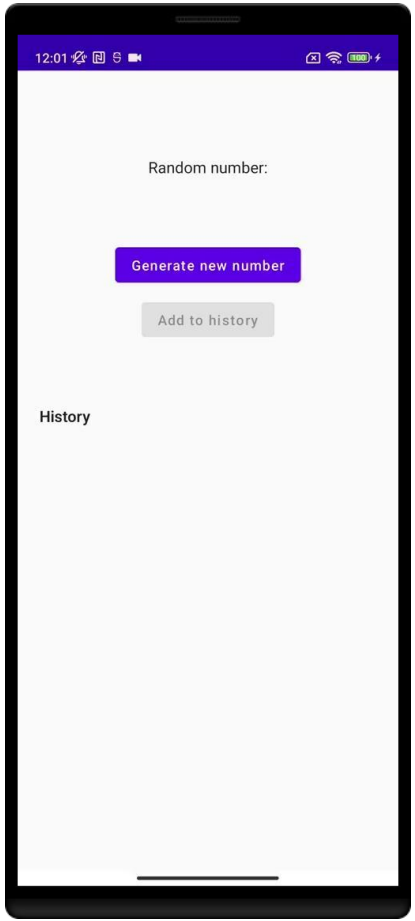
```
@Composable
fun AlphaScreen(viewModel: AlphaViewModel) {
    val uiState by viewModel.uiState.collectAsStateWithLifecycle()

    Column(…) {
        Text(text = "Progress", …)
        Progress(isProgressVisible = uiState.isProgressVisible)
        ProgressButton(onTriggerProgress = …)
    }
}


@Composable
private fun Progress(isProgressVisible: Boolean) {
    val loadingIndicatorVisibility by animateFloatAsState(
        targetValue = if (isProgressVisible) 1f else 0f, animationSpec = tween()
    )

    Column(…,
        modifier = Modifier.graphicsLayer { alpha = loadingIndicatorVisibility }
    ) {
        CircularProgressIndicator()
        Text(text = "Loading...")
    }
}
```

```
@Composable
fun AlphaScreen(viewModel: AlphaViewModel) {
    val uiState by viewModel.uiState.collectAsStateWithLifecycle()

    Column(…) {
        Text(text = "Progress", …)
        Progress(isProgressVisible = { uiState.isProgressVisible })
        ProgressButton(onTriggerProgress = …)
    }
}


@Composable
private fun Progress(isProgressVisible: () → Boolean) {
    val loadingIndicatorVisibility by animateFloatAsState(
        targetValue = if (isProgressVisible()) 1f else 0f, animationSpec = tween()
    )

    Column(…,
        modifier = Modifier.graphicsLayer { alpha = loadingIndicatorVisibility }
    ) {
        CircularProgressIndicator()
        Text(text = "Loading...")
    }
}
```


12:21
Progress
Trigger progress


Layout Inspector
Component Tree
Recomposition counts
Reset
ComposeMetricsTheme
MaterialTheme
NavHost
NavHost
Crossfade
LocalOwnersProvider
SaveableStateProvider
SaveableStateProvider
AlphaScreen
Column
Text
Spacer
Progress
Column
CircularProgressIndicator
Text
Spacer
ProgressButton

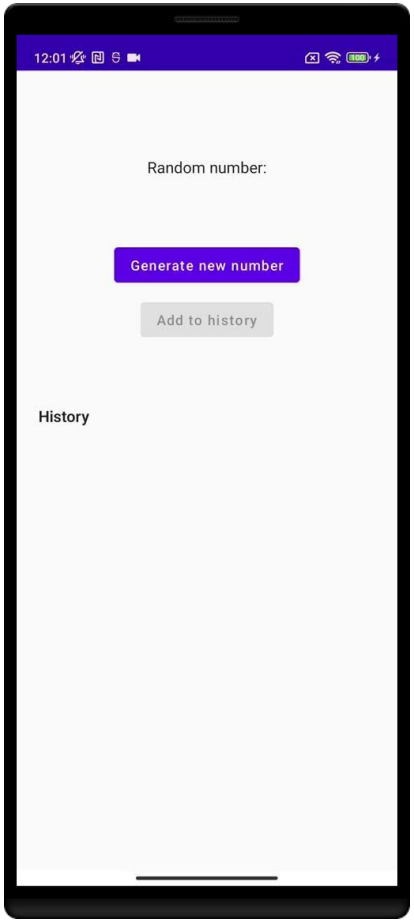
Stabilize the unstable #1

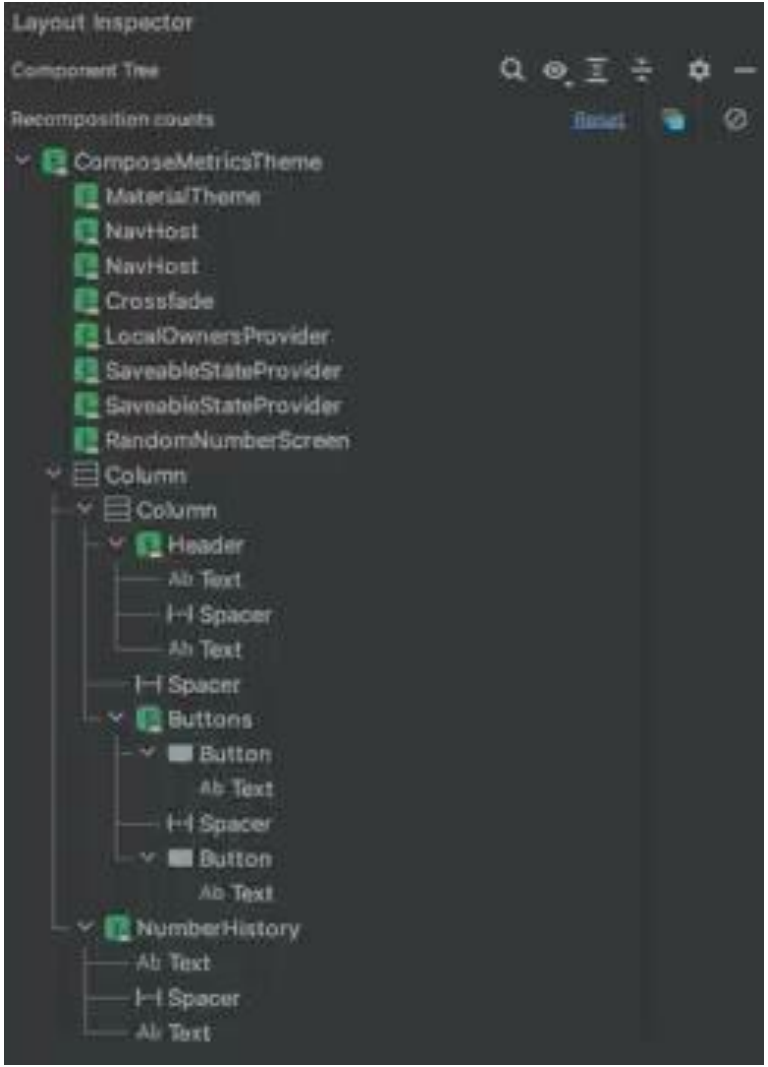

12:01
Random number:
Generate new number
Add to history
History

```
@Composable
fun RandomNumberScreen(viewModel: RandomNumberViewModel) {
    val uiState by viewModel.uiState.collectAsStateWithLifecycle()

    Column(…) {
        Header(randomData = uiState)
        Buttons(
            currentNumber = uiState.currentNumber,
            onGenerateNumber = { viewModel.generateNewNumber() },
            onAddToHistory = { viewModel.addToHistory() },
        )
        NumberHistory(numbers = uiState.history)
    }
}
```

```
@Composable
fun RandomNumberScreen(viewModel: RandomNumberViewModel) {
    val uiState by viewModel.uiState.collectAsStateWithLifecycle()

    Column(…) {
        Header(randomData = uiState)
        Buttons(
            currentNumber = uiState.currentNumber,
            onGenerateNumber = { viewModel.generateNewNumber() },
            onAddToHistory = { viewModel.addToHistory() },
        )
        NumberHistory(numbers = uiState.history)
    }
}


@Composable
private fun Header(randomData: RandomNumberUiState) {
    val numberText = remember(numbers) { numbers.joinToString(", ") }
    Text(text = "Random number:")
    Text(text = randomData.currentNumber?.toString() ?: "", …)
    Text(text = numberText)
    Column(…) {
        Button(onClick = onGenerateNumber) {…}
        Button(enabled = currentNumber ≠ null, onClick = onAddToHistory) {…}
    }
}
```

```
@Composable
private fun Buttons(
    currentNumber: Int?,
    onGenerateNumber: () -> Unit,
    onAddToHistory: () -> Unit,
) { }
}
}
```


12:01
Random number:
Generate new number
Add to history
History


Layout Inspector
Component Tree
Recomposition counts
Reset
ComposeMetricsTheme
MaterialTheme
NavHost
NavHost
Crossfade
LocalOwnersProvider
SaveableStateProvider
SaveableStateProvider
RandomNumberScreen
Column
Column
Header
Text
Spacer
Text
Spacer
Buttons
Button
Text
Spacer
Button
Text
NumberHistory
Text
Spacer
Text

Compose Compiler Metrics

```
android {
    …
    kotlinOptions {
        if (project.findProperty("enableComposeCompilerReports") == "true") {
            freeCompilerArgs += [
                    "-P",
                    "plugin:androidx.compose.compiler.plugins.kotlin:reportsDestination=" +
                            project.buildDir.absolutePath + "/compose_metrics"
            ]
            freeCompilerArgs += [
                    "-P",
                    "plugin:androidx.compose.compiler.plugins.kotlin:metricsDestination=" +
                            project.buildDir.absolutePath + "/compose_metrics"
            ]
        }
    }
}
```

```
./gradlew assembleRelease -PenableComposeCompilerReports=true
```

Following files will be generated:

- <moduleName>-module.json
- <moduleName>-composables.txt
- <moduleName>-composables.csv
- <moduleName>-classes.txt

```
"skippableComposables": 22,
restartable skippable scheme("[androidx.compose.ui.UiComposable]") fun Buttons(
  stable currentNumber: Int?
  stable onGenerateNumber: Function0<Unit>
  stable onAddToHistory: Function0<Unit>
)
restartable scheme("[androidx.compose.ui.UiComposable]") fun NumberHistory(
  unstable numbers: List<Int>
)
"knownUnstableArguments": 9,
```

Immutable

Stable

Immutable

- applicable for types
- value of any property never changes after object has been instantiated
- e.g. all primitive types (Boolean, Float, Int, …), Strings, all function types (lambdas)

Stable

Immutable

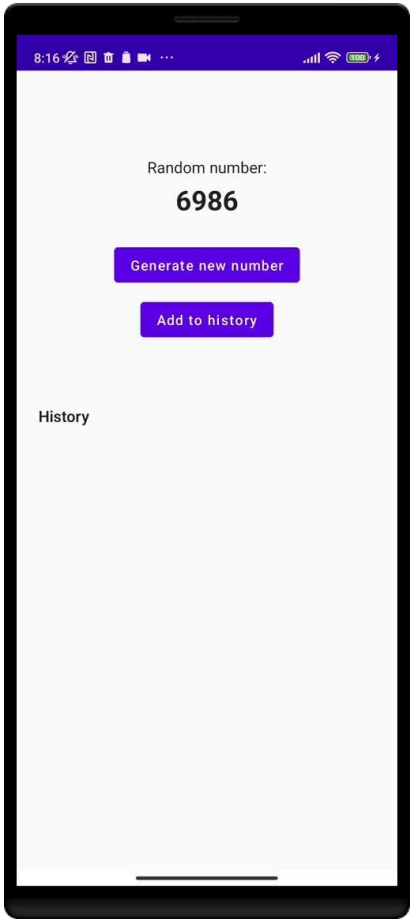
- applicable for types
- value of any property never changes after object has been instantiated
- e.g. all primitive types (Boolean, Float, Int, …), Strings, all function types (lambdas)

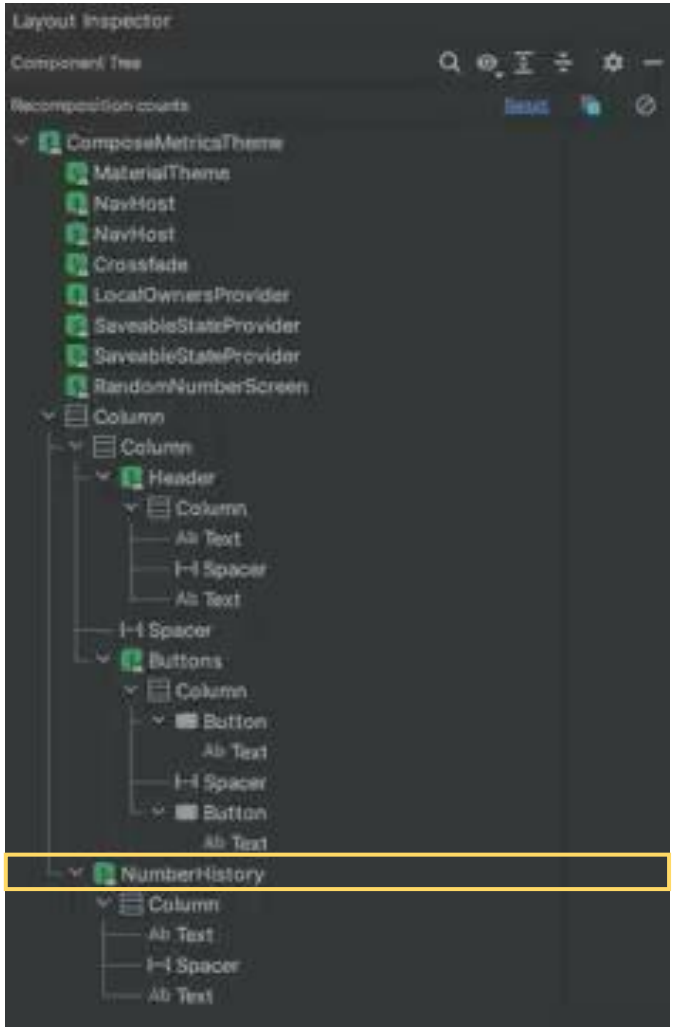
Stable

- applicable for types
- type is mutable but Compose runtime will be notified whenever anything changes
- use state, e.g. `mutableStateOf()`

Requirements for a type beeing considered stable:

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


8:16
Random number:
6986
Generate new number
Add to history
History


Layout Inspector
Component Tree
Recomposition counts
Reset
ComposeMetricsTheme
MaterialTheme
NavHost
NavHost
Crossfade
LocalOwnersProvider
SaveableStateProvider
SaveableStateProvider
RandomNumberScreen
Column
Column
Header
Column
Text
Spacer
Text
Spacer
Buttons
Column
Button
Text
Spacer
Button
Text
NumberHistory
Column
Text
Spacer
Text

```
@Composable
private fun Header(randomData: RandomData) {
    Column(…) {
        Text(text = "Random number:")
        Text(text = randomData.currentNumber?.toString() ?: "", …)
    }
}

restartable scheme("[androidx.compose.ui.UiComposable]") fun Header(
  unstable randomData: RandomData
)
```

```
data class RandomData(var currentNumber: Int?, val history: List<Int>)

unstable class RandomData {
  stable var currentNumber: Int?
  unstable val history: List<Int>
  <runtime stability> = Unstable
}
```

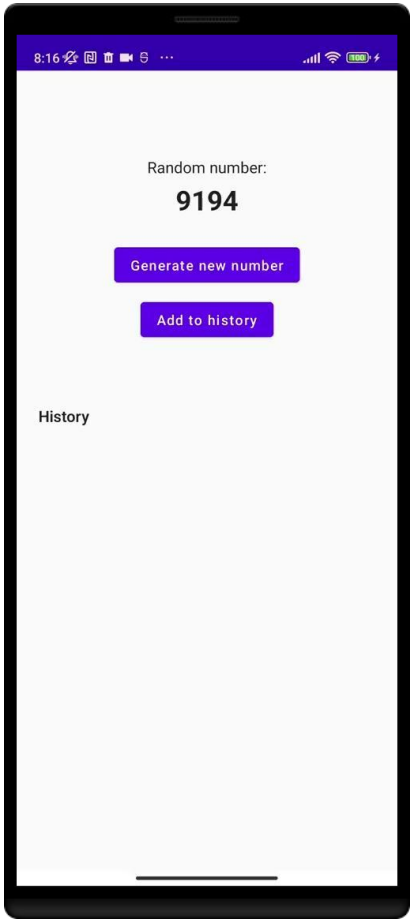
```kotlin
@Composable
private fun Header(randomData: RandomData) {
    Column(…) {
        Text(text = "Random number:")
        Text(text = randomData.currentNumber?.toString() ?: "", …)
    }
}
```

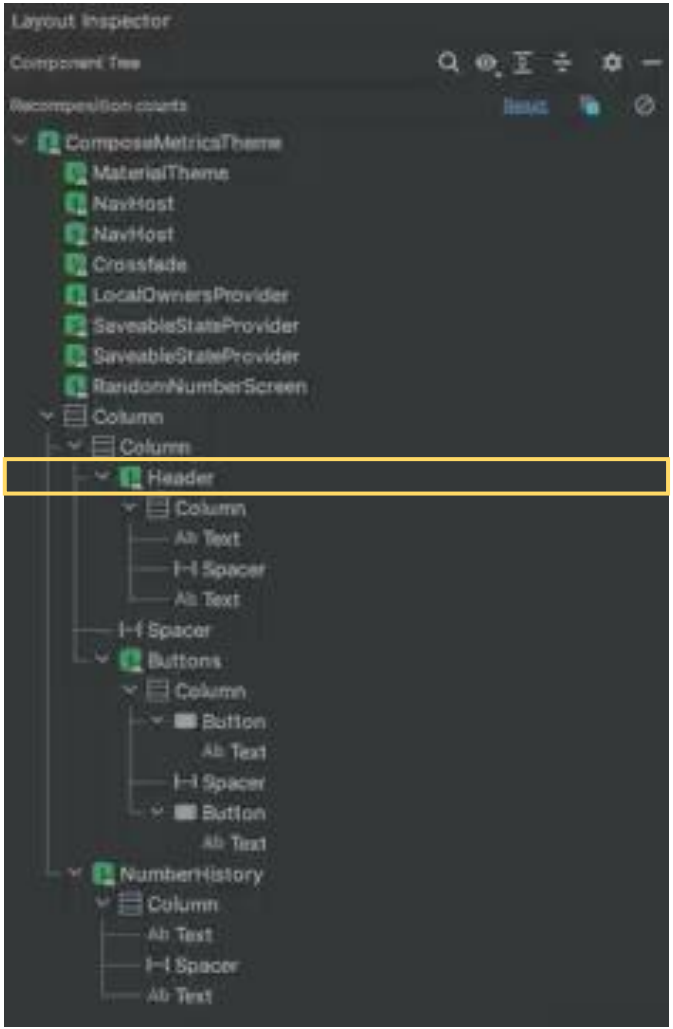
```
restartable scheme("[androidx.compose.ui.UiComposable]") fun Header(
  unstable randomData: RandomData
)
```

```kotlin
data class RandomData(var currentNumber: Int?, val history: List<Int>)
```

```
unstable class RandomData {
  stable var currentNumber: Int?
  unstable val history: List<Int>
  <runtime stability> = Unstable
}
```

```
@Composable
private fun Header(currentNumber: Int?) {
    Column(…) {
        Text(text = "Random number:")
        Text(text = currentNumber?.toString() ?: "", …)
    }
}

restartable skippable scheme("[androidx.compose.ui.UiComposable]") fun Header(
  stable currentNumber: Int?
)
```


8:16
Random number:
9194
Generate new number
Add to history
History


Layout Inspector
Component Tree
Recomposition counts
Reset
ComposeMetricsTheme
MaterialTheme
NavHost
NavHost
Crossfade
LocalOwnersProvider
SaveableStateProvider
SaveableStateProvider
RandomNumberScreen
Column
Column
Header
Column
Text
Spacer
Text
Spacer
Buttons
Column
Button
Text
Spacer
Button
Text
NumberHistory
Column
Text
Spacer
Text

```
@Composable
private fun Buttons(
    currentNumber: Int?,
    onGenerateNumber: () → Unit,
    onAddToHistory: () → Unit
) {
    Column(…) {
        Button(onClick = onGenerateNumber) {…}
        Button(enabled = currentNumber ≠ null, onClick = onAddToHistory) {…}
    }
}

restartable skippable scheme("[androidx.compose.ui.UiComposable]") fun Buttons(
  stable currentNumber: Int?
  stable onGenerateNumber: Function0<Unit>
  stable onAddToHistory: Function0<Unit>
)
```

```
Buttons(
    currentNumber = uiState.currentNumber,
    onGenerateNumber = { viewModel.generateNewNumber() },
    onAddToHistory = { viewModel.addToHistory() },
)

unstable class RandomNumberViewModel {
  …
  <runtime stability> = Unstable
}

class GenerateNumberLambda(private val viewModel: RandomNumberViewModel) {
    operator fun invoke() {
        viewModel.generateNewNumber()
    }
}
```

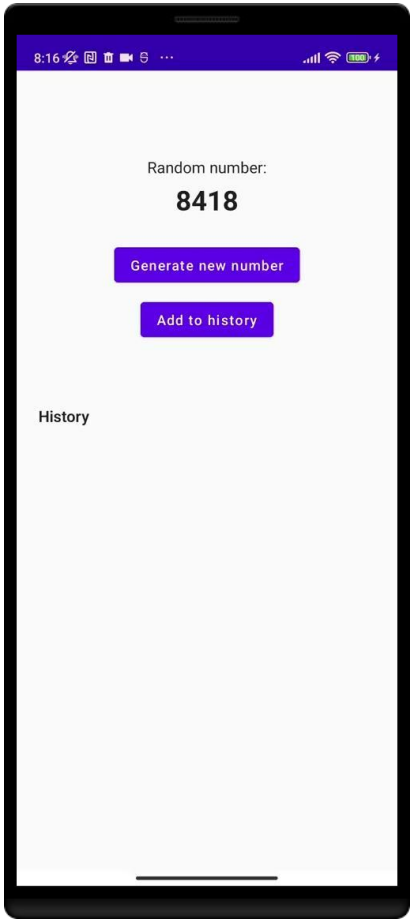
```
Buttons(
    currentNumber = uiState.currentNumber,
    onGenerateNumber = viewModel::generateNewNumber,
    onAddToHistory = viewModel::addToHistory,
)


val onGenerateNumber = remember(viewModel) { { viewModel.generateNewNumber() } }
val onAddToHistory = remember(viewModel) { { viewModel.addToHistory() } }
Buttons(
    currentNumber = uiState.currentNumber,
    onGenerateNumber = onGenerateNumber,
    onAddToHistory = onAddToHistory,
)
```

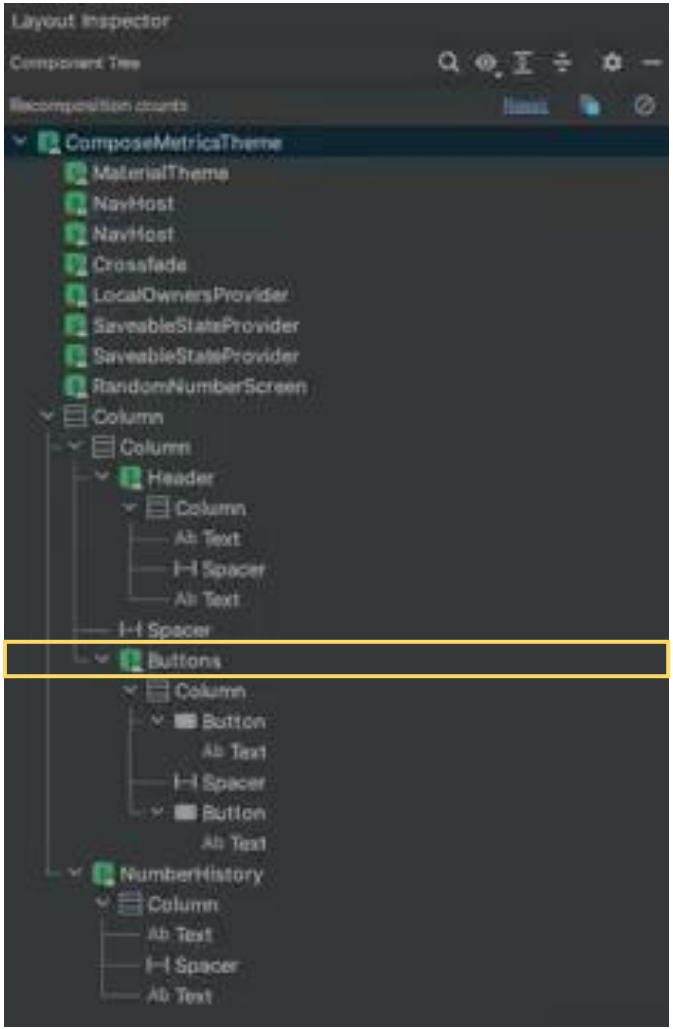
```kotlin
Buttons(
    currentNumber = uiState.currentNumber,
    onGenerateNumber = viewModel::generateNewNumber,
    onAddToHistory = viewModel::addToHistory,
)
```

```kotlin
val onGenerateNumber = remember(viewModel) { { viewModel.generateNewNumber() } }
val onAddToHistory = remember(viewModel) { { viewModel.addToHistory() } }

Buttons(
    currentNumber = uiState.currentNumber,
    onGenerateNumber = onGenerateNumber,
    onAddToHistory = onAddToHistory,
)
```

```
@Composable
private fun Buttons(
    currentNumber: Int?,
    onGenerateNumber: () → Unit,
    onAddToHistory: () → Unit
) {
    Column(…) {
        Button(onClick = onGenerateNumber) {…}
        Button(enabled = currentNumber ≠ null, onClick = onAddToHistory) {…}
    }
}
```

```kotlin
@Composable
private fun Buttons(
    historyButtonEnabled: Boolean,
    onGenerateNumber: () -> Unit,
    onAddToHistory: () -> Unit
) {
    Column(…) {
        Button(onClick = onGenerateNumber) {…}
        Button(enabled = historyButtonEnabled, onClick = onAddToHistory) {…}
    }
}

Buttons(
    historyButtonEnabled = uiState.currentNumber != null,
    onGenerateNumber = viewModel::generateNewNumber,
    onAddToHistory = viewModel::addToHistory,
)
```


8:16
Random number:
8418
Generate new number
Add to history
History


Layout Inspector
Component Tree
Recomposition counts
ComposeMetricsTheme
MaterialTheme
NavHost
NavHost
Crossfade
LocalOwnersProvider
SaveableStateProvider
SaveableStateProvider
RandomNumberScreen
Column
Column
Header
Column
Text
Spacer
Text
Spacer
Buttons
Column
Button
Text
Spacer
Button
Text
NumberHistory
Column
Text
Spacer
Text

```
@Composable
private fun NumberHistory(numbers: List<Int>) {
    val numberText = remember(numbers) { numbers.joinToString(", ") }
    Column {
        Text(text = "History“, …)
        Text(text = numberText)
    }
}
```

```
@Composable
private fun NumberHistory(numbers: List<Int>) {
    val numberText = remember(numbers) { numbers.joinToString(", ") }
    Column {
        Text(text = "History“, …)
        Text(text = numberText)
    }
}
```

```
restartable scheme("[androidx.compose.ui.UiComposable]") fun NumberHistory(
  unstable numbers: List<Int>
)

val numberList: List<Int> = mutableListOf(1, 2, 3)
```

1) Wrap the `List` and annotate the wrapper with `@Stable`

2) Use Kotlinx Immutable Collections*

- `ImmutableList, PersistentList`
- `ImmutableSet, PersistentSet`
- `...`

* https://github.com/Kotlin/kotlinx.collections.immutable

```
@Composable
private fun NumberHistory(numbers: List<Int>) {
    val numberText = remember(numbers) { numbers.joinToString(", ") }
    Column {
        Text(text = "History“, …)
        Text(text = numberText)
    }
}
```

```
@Composable
private fun NumberHistory(numbers: ImmutableList<Int>) {
    val numberText = remember(numbers) { numbers.joinToString(", ") }
    Column {
        Text(text = "History“, …)
        Text(text = numberText)
    }
}
```

```
restartable skippable scheme("[androidx.compose.ui.UiComposable]") fun NumberHistory(
  stable numbers: ImmutableList<Int>
)
```

```kotlin
data class RandomData(var currentNumber: Int?, val history: List<Int>)
```

```kotlin
data class RandomData(var currentNumber: Int?, val history: PersistentList<Int>)
```

```
unstable class RandomData {
  stable var currentNumber: Int?
  stable val history: PersistentList<Int>
  <runtime stability> = Unstable
}
```

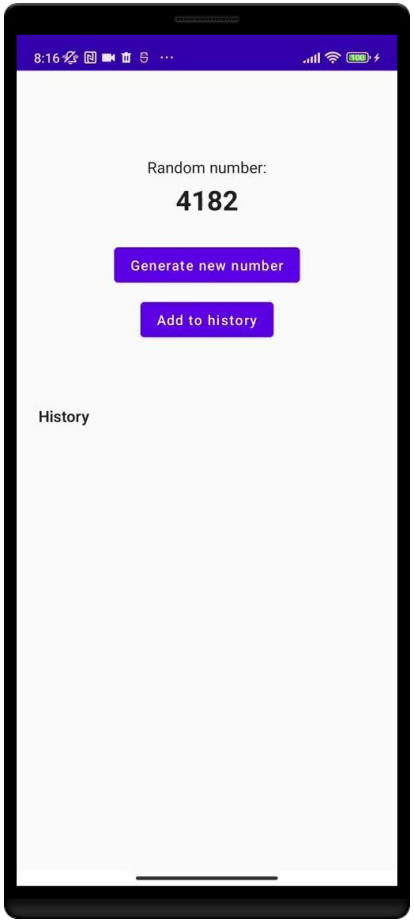
* PersistentList is considered stable since Compose Compiler version 1.4.0

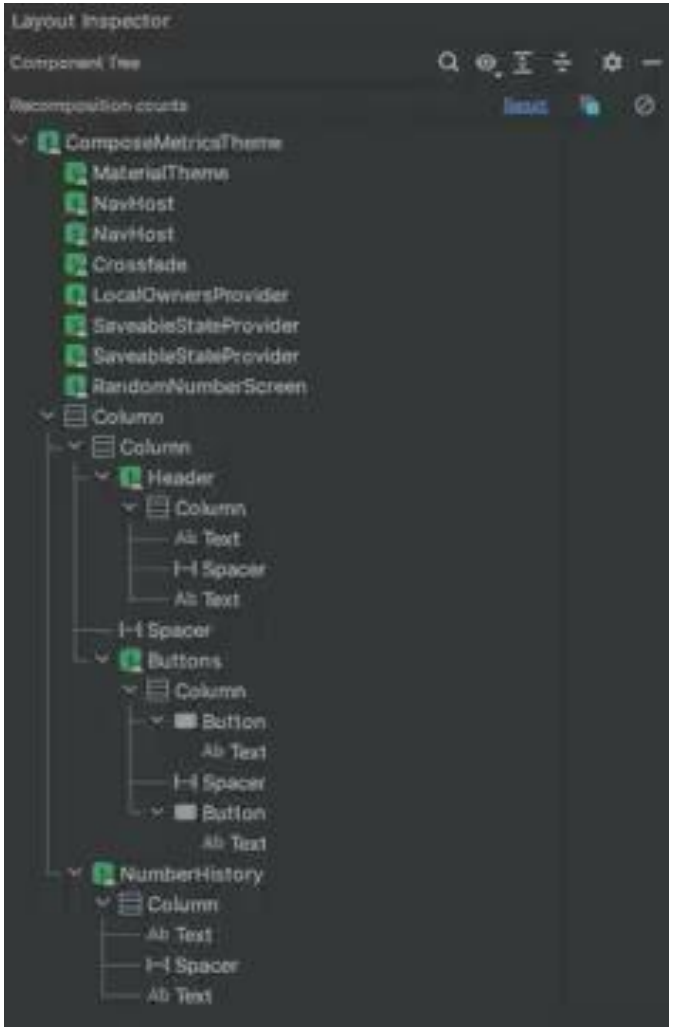
```
data class RandomData(val currentNumber: Int?, val history: PersistentList<Int>)

unstable class RandomData {
  stable var currentNumber: Int?
  stable val history: PersistentList<Int>
  <runtime stability> = Unstable
}
```

* PersistentList is considered stable since Compose Compiler version 1.4.0

```
data class RandomData(val currentNumber: Int?, val history: PersistentList<Int>)

stable class RandomData {
  stable val currentNumber: Int?
  stable val history: PersistentList<Int>
  <runtime stability> =
}
```

* PersistentList is considered stable since Compose Compiler version 1.4.0


8:16
Random number:
4182
Generate new number
Add to history
History


Layout Inspector
Component Tree
Recomposition counts
Reset
ComposeMetricsTheme
MaterialTheme
NavHost
NavHost
Crossfade
LocalOwnersProvider
SaveableStateProvider
SaveableStateProvider
RandomNumberScreen
Column
Column
Header
Column
Text
Spacer
Text
Spacer
Buttons
Column
Button
Text
Spacer
Button
Text
NumberHistory
Column
Text
Spacer
Text

Stabilize the unstable #2

```kotlin
@Composable
private fun NumberHistory(history: HistoryData) {
    val numberText = remember(history) { history.history.joinToString(", ") }
    Column {
        Text(text = "History", …)
        Text(text = numberText)
    }
}


data class RandomData2(val currentNumber: Int?, val history: HistoryData)

data class HistoryData(val history: PersistentList<Int>)


restartable scheme("[androidx.compose.ui.UiComposable]") fun NumberHistory(
  unstable history: HistoryData
)
```

ComposeMetrics 〉 lib 〉 src 〉 main 〉 java 〉 com 〉 nowdev 〉 composemetrics 〉 external 〉 RandomData2.kt

1) Create separate UI models in the app module + `map()` functions

2) Don‘t pass class arguments if you only need primitive types

3) Add androidx.compose.runtime:runtime dependency to the lib module and mark class with `@Stable` if applicable

```
@Composable
private fun NumberHistory(history: HistoryData) {
    val numberText = remember(history) { history.history.joinToString(", ") }
    Column {
        Text(text = "History", …)
        Text(text = numberText)
    }
}

data class HistoryData(val history: PersistentList<Int>)
```

```
@Composable
private fun NumberHistory(history: HistoryData) {
    val numberText = remember(history) { history.history.joinToString(", ") }
    Column {
        Text(text = "History", …)
        Text(text = numberText)
    }
}


@Stable
data class HistoryData(val history: PersistentList<Int>)


restartable skippable scheme("[androidx.compose.ui.UiComposable]") fun NumberHistory(
  stable history: HistoryData
)
```

ℹ️ Attention

- Don‘t try to make your entire UI skippable / all class types stable
- Watch out for performance issues and investigate if changes are worth
- Compose is trying to care for as many things as possible automatically

- Ben Trengrove: “Jetpack Compose Stability Explained”
  https://medium.com/androiddevelopers/jetpack-compose-stability-explained-79c10db270c8

- Jaewoong Eum, Marin: “6 Jetpack Compose Guidelines to Optimize Your App Performance”
  https://getstream.io/blog/jetpack-compose-guidelines/

- Chris Banes: “Composable metrics”
  https://chrisbanes.me/posts/composable-metrics/

- Ben Trengrove: “Jetpack Compose: Debugging Recomposition”
  https://medium.com/androiddevelopers/jetpack-compose-debugging-recomposition-bfcf4a6f8d37

- IceRock Development: “Optimize or Die. Profiling and Optimization in Jetpack Compose”
  https://medium.com/icerock/optimize-or-die-profiling-and-optimization-in-jetpack-compose-a165c8897b3f

- Android Developers: “Lifecycle of composables”
  https://developer.android.com/jetpack/compose/lifecycle

Thank you!

Philipp Nowak

@philnowak96