DEVIEW 2021

# Android Jetpack Compose 실제 서비스 적용 후기

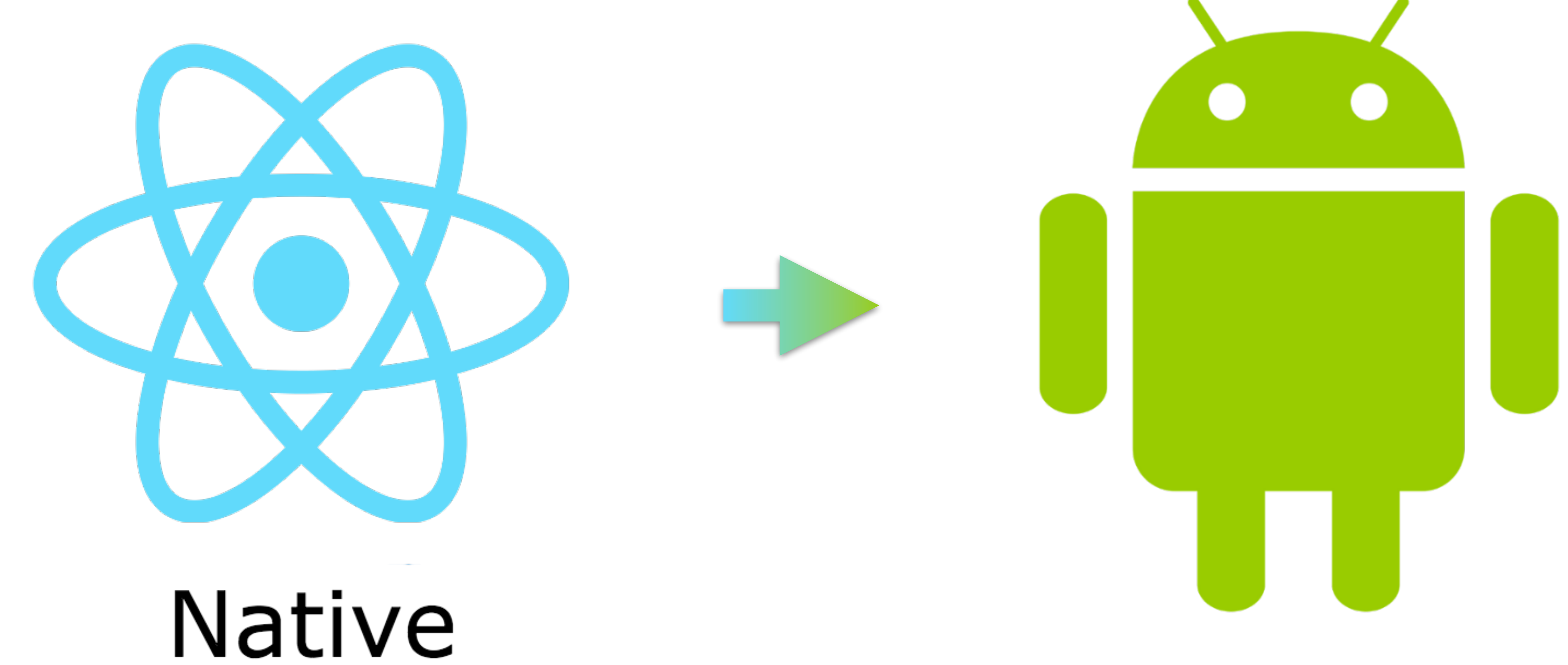
남상혁 NAVER / ABC Studio

# CONTENTS

1. 적용 배경
2. 중요 개념
3. 장점
4. 단점
5. 주의할 점
6. State에 맞게 추가한 개념

# 1. 적용 배경

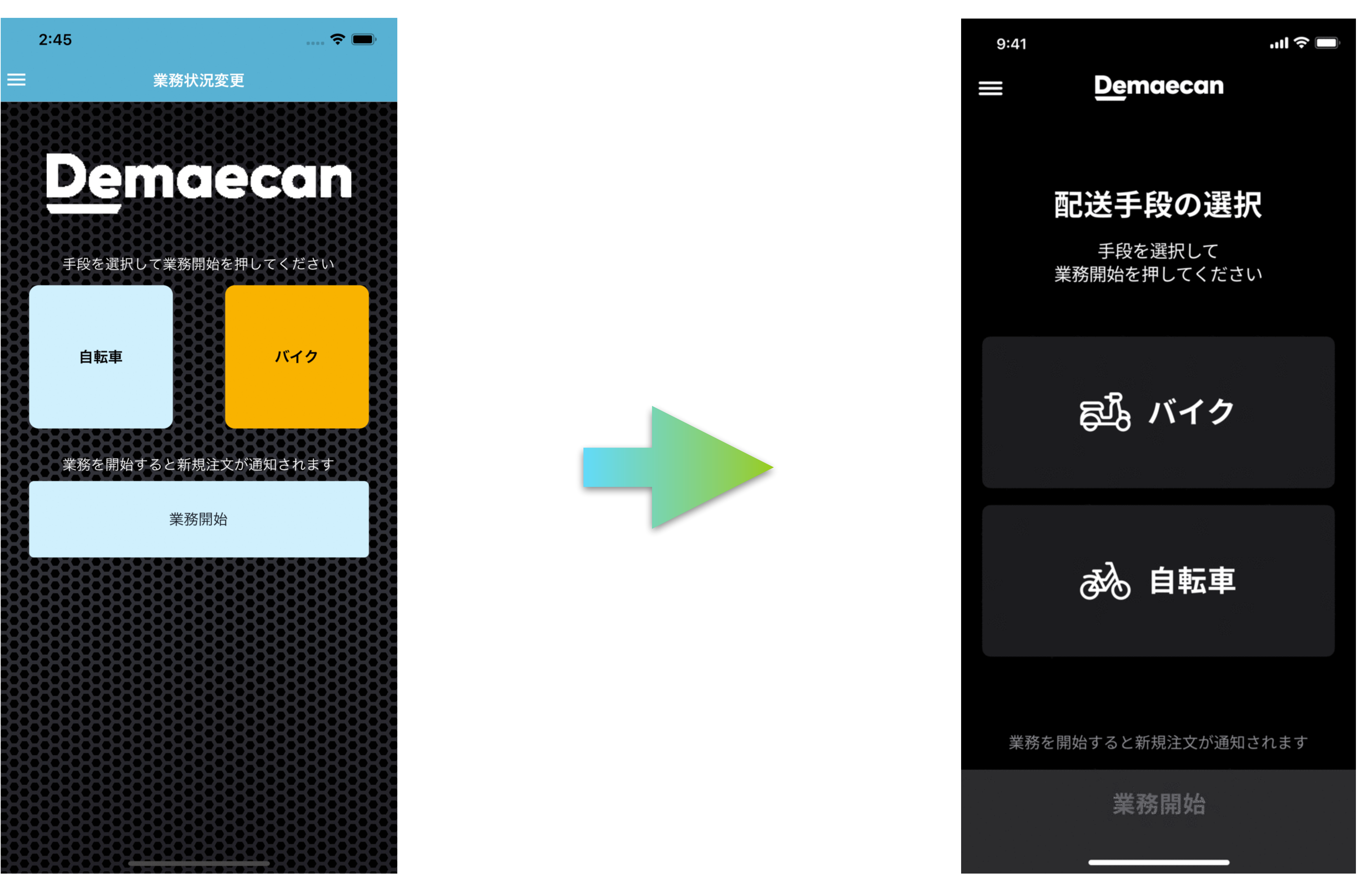
# 1. 적용 배경




React Native → Android 리뉴얼

- 일본 배달앱 'Demae-can'의 Driver 앱

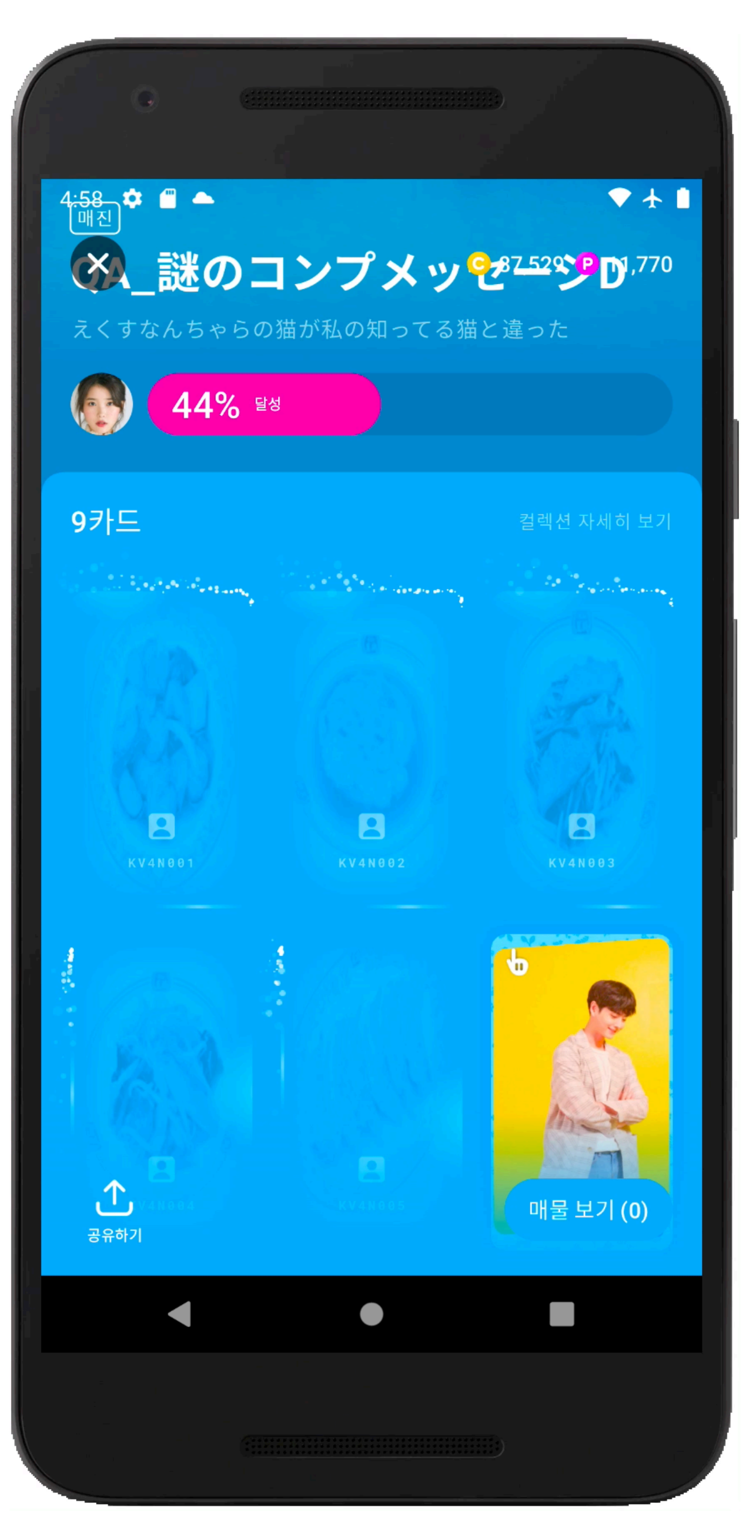
# 1. 적용 배경




## 비교적 적은 Spec과 화면

- Fragment 단위로 14개 정도의 화면
- 이정도면 Trouble Shooting하면서 적용해볼 수 있지 않을까?

# 1. 적용 배경




## View의 onDraw를 이용하는 기존 Project

- Jetpack Compose 개념 이해 수월

# 2. 중요 개념

# 2.1 기존 Native UI 개념

<Page1>

ConstraintLayout
RecyclerView
ImageView
...

Activity

XML

Inflate

<Page2>

Inflate

XML

Fragment

LinearLayout
TextView
Button
...

<Page3>

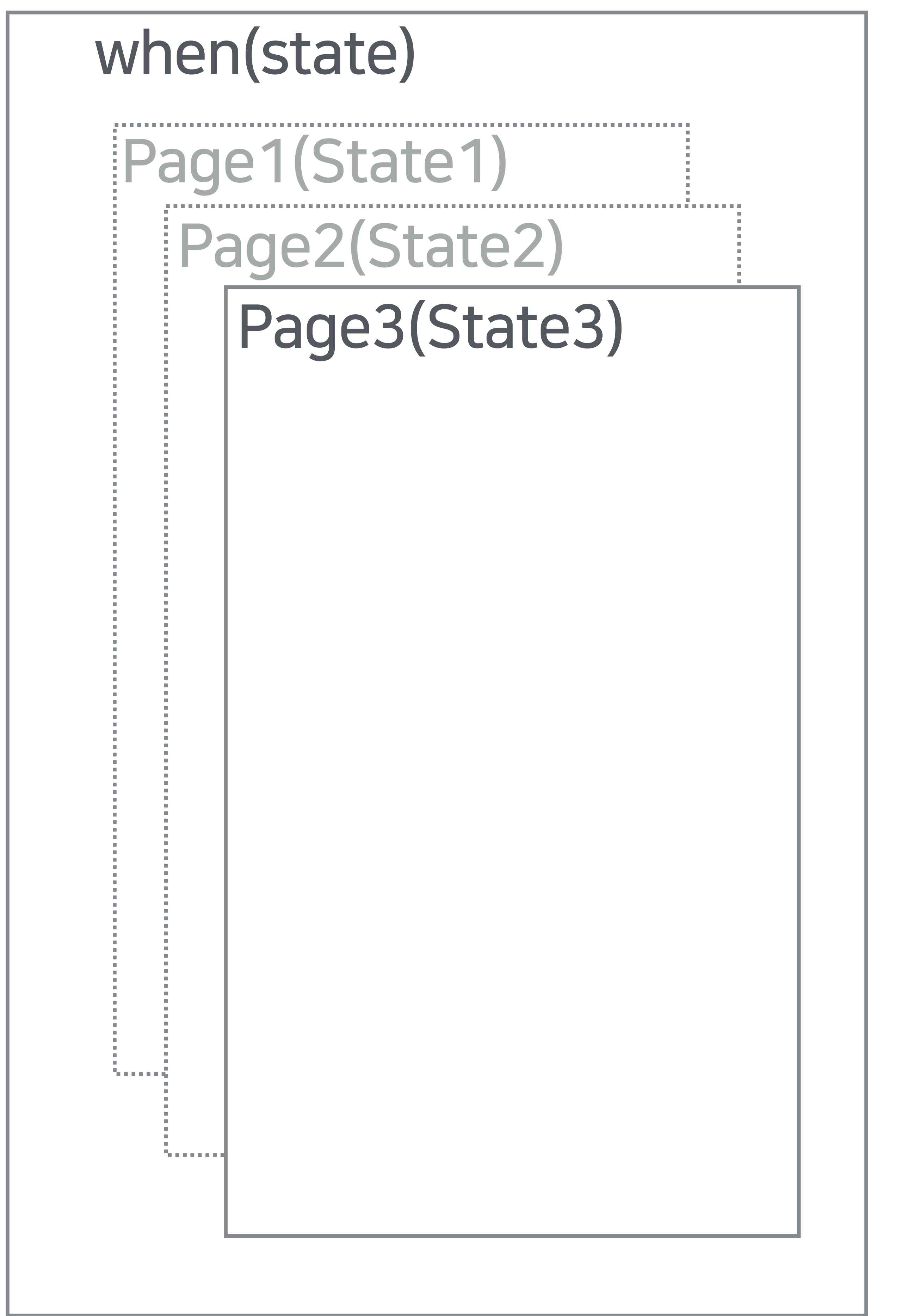
Inflate

XML

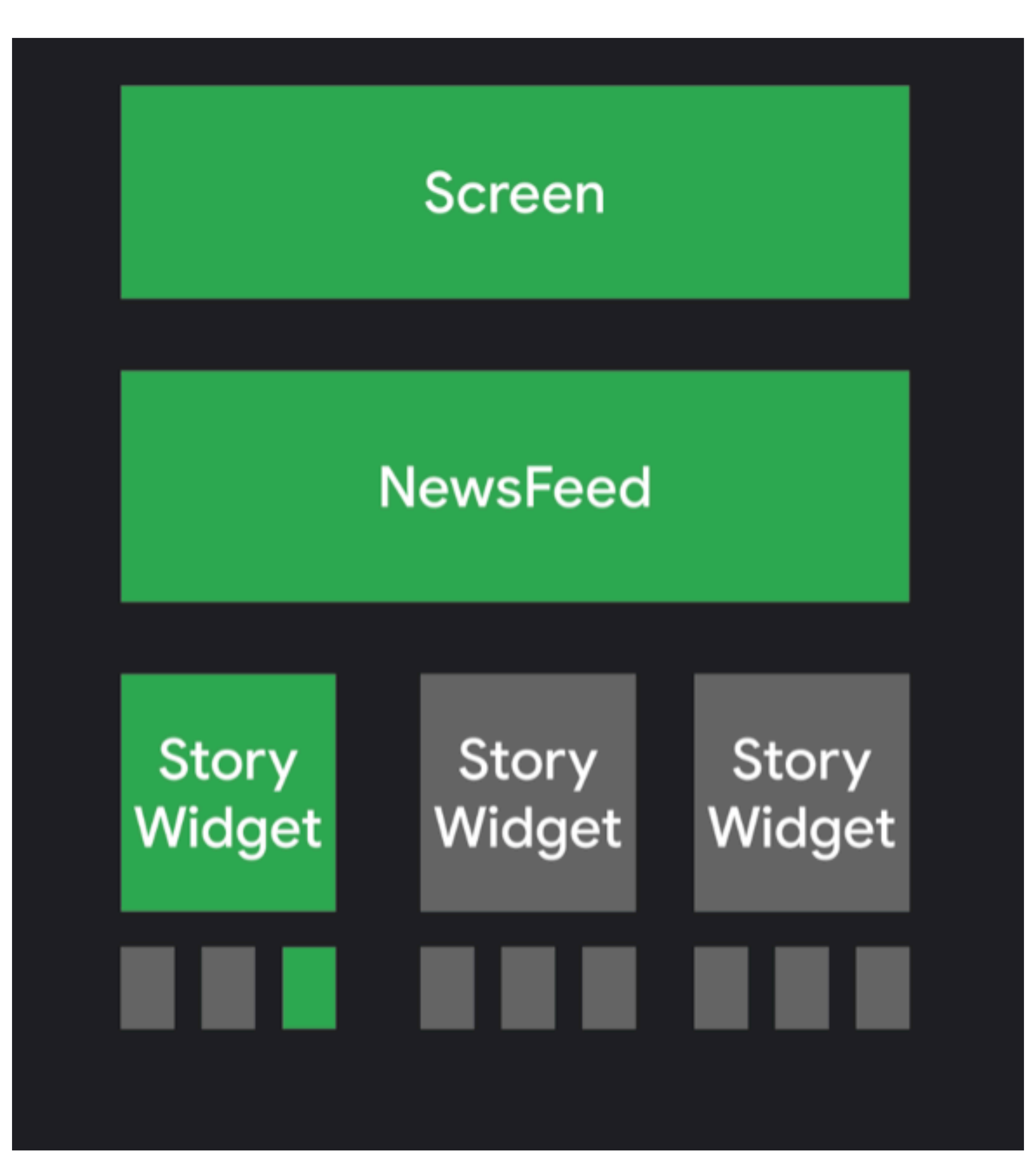
Fragment

LinearLayout
TextView
Button
...

# 2.2 Jetpack Compose 개념



# 2.2 Jetpack Compose 개념




https://developer.android.com/jetpack/compose/mental-model

# 2.3 기존 Native UI 구성

- View의 형태를 XML로 미리 구성
- XML 안의 View를 읽어와서(findViewById or binding)
- View를 어떻게 만들지 명령

```
<TextView
    android:id="@+id/sample_text"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:padding="20dp"
    android:gravity="center"
    android:textSize="30sp"
    android:text="Hello"
    android:visibility="visible"/>

<ImageView
    android:id="@+id/sample_image"
    android:layout_width="60dp"
    android:layout_height="60dp"
    android:layout_margin="20dp"
    android:layout_gravity="center"
    android:contentDescription="image"
    android:src="@drawable/android"
    android:visibility="gone"/>

<androidx.appcompat.widget.AppCompatButton
    android:id="@+id/sample_btn"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:padding="20dp"
    android:textSize="30sp"
    android:text="CLICK"/>
```

# 2.3 기존 Native UI 구성

- View의 구성을 XML로 미리 구성
- XML 안의 View를 읽어와서(findViewById or binding)
- View를 어떻게 만들지 명령

```xml
<TextView
    android:id="@+id/sample_text"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:padding="20dp"
    android:gravity="center"
    android:textSize="30sp"
    android:text="Hello"
    android:visibility="visible"/>

<ImageView
    android:id="@+id/sample_image"
    android:layout_width="60dp"
    android:layout_height="60dp"
    android:layout_margin="20dp"
    android:layout_gravity="center"
    android:contentDescription="image"
    android:src="@drawable/android"
    android:visibility="gone"/>

<androidx.appcompat.widget.AppCompatButton
    android:id="@+id/sample_btn"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:padding="20dp"
    android:textSize="30sp"
    android:text="CLICK"/>
```

```kotlin
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)
    binding.sampleBtn.setOnClickListener {
        binding.sampleText.visibility = if(binding.sampleText.visibility == View.VISIBLE) {
             View.GONE
        } else {
            View.VISIBLE
        }

        binding.sampleImage.visibility = if(binding.sampleText.visibility == View.VISIBLE) {
            View.GONE
        } else {
            View.VISIBLE
        }
    }
}
```

# 2.3 기존 Native UI 구성

- View의 구성을 XML로 미리 구성
- XML 안의 View를 읽어와서(findViewById or binding)
- View를 어떻게 만들지 명령

```xml
<TextView
    android:id="@+id/sample_text"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:padding="20dp"
    android:gravity="center"
    android:textSize="30sp"
    android:text="Hello"
    android:visibility="visible"/>

<ImageView
    android:id="@+id/sample_image"
    android:layout_width="60dp"
    android:layout_height="60dp"
    android:layout_margin="20dp"
    android:layout_gravity="center"
    android:contentDescription="image"
    android:src="@drawable/android"
    android:visibility="gone"/>

<androidx.appcompat.widget.AppCompatButton
    android:id="@+id/sample_btn"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:padding="20dp"
    android:textSize="30sp"
    android:text="CLICK"/>
```

```kotlin
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)
    binding.sampleBtn.setOnClickListener {
        binding.sampleText.visibility = if(binding.sampleText.visibility == View.VISIBLE) {
             View.GONE
        } else {
            View.VISIBLE
        }

        binding.sampleImage.visibility = if(binding.sampleText.visibility == View.VISIBLE) {
            View.GONE
        } else {
            View.VISIBLE
        }
    }
}
```

11:55
My Application
Hello
CLICK

# 2.4 선언형 Jetpack Compose 구성

- 그리고 싶은 View자체를 선언
- State에 맞게 View를 선택적으로 그림

```
@Composable
fun MyView() {
    var state by remember { mutableStateOf(false) }
    Column(modifier = Modifier.fillMaxWidth()) {
        if(!state) {
            SampleText("Hello")
        } else {
            Image(
                painter = painterResource(R.drawable.compose),
                contentDescription = "Compose Image",
                modifier = Modifier.padding(20.dp).size(60.dp).align(Alignment.CenterHorizontally)
            )
        }
        Button(
            onClick = { state = !state },
            modifier = Modifier.padding(20.dp).fillMaxWidth()
        ) {
            SampleText("CLICK")
        }
    }
}
```

Hello

CLICK

$$UI = f(State)$$

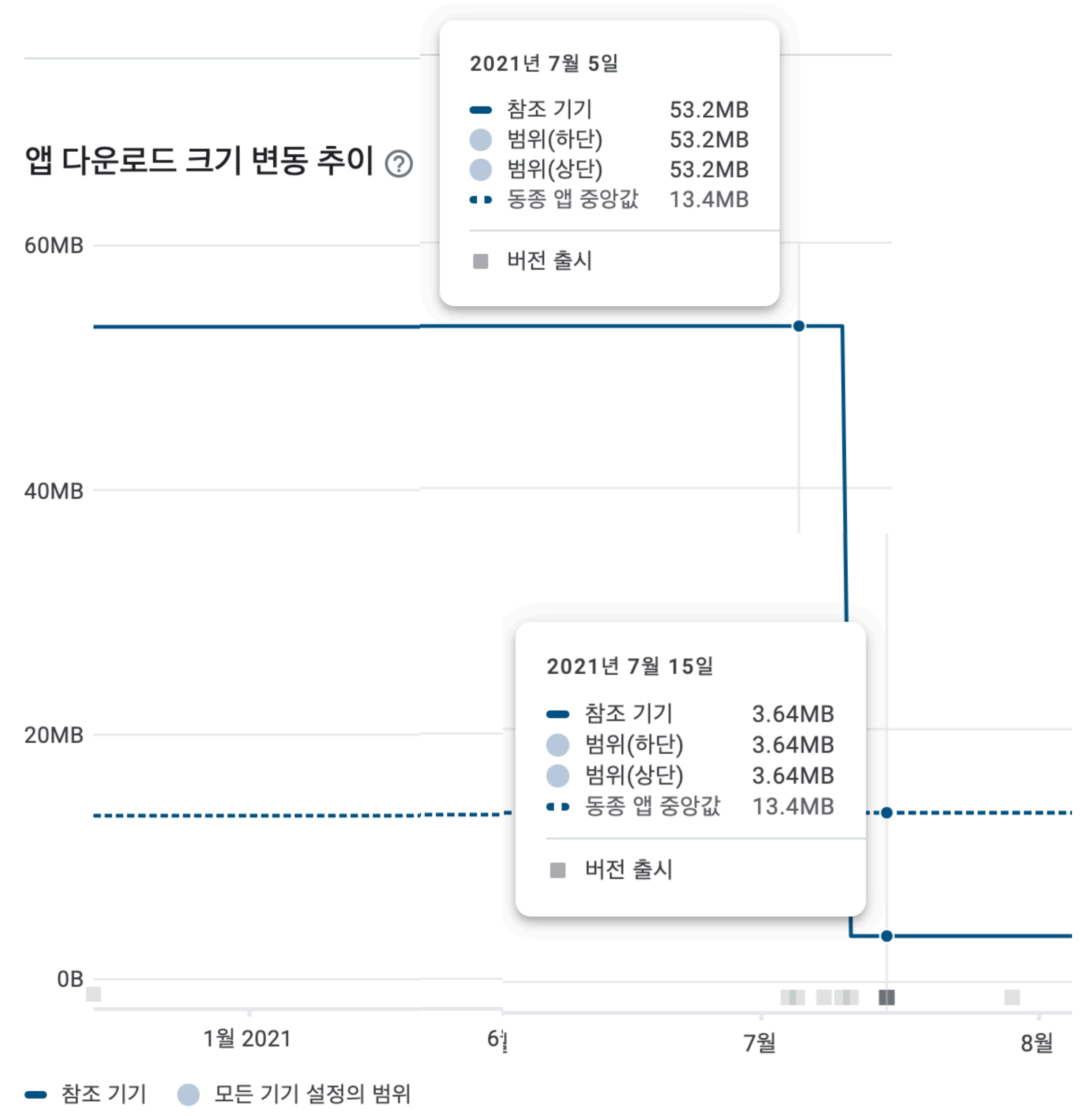
# 3. 장점

# 3.1 Build 속도

## Gradle 제외한 Code 수정 후의 ReBuild 속도

- 27sec → 3sec

# 3.2 APK Size




## 앱 다운로드 크기 감소

- 53.2MB→ 3.64MB
- App Bundle을 사용하지 않고도 충분히 작은 Size

# 3.3 Native 기능 및 Context 사용

DEVIEW 2021

**Native를 사용하기도 Context를 사용하기도 불편함이 없음**

- LocalContext
- LocalFocusManager
- LocalClipboardManager
- LocalDensity

...

```
@Composable
fun MyView() {
  val context = LocalContext.current
  val webViewClient = WebViewClient()
  AndroidView(
    factory = {
      WebView(context).apply {
        this.webViewClient = webViewClient
        this.loadUrl("file:///android_asset/Test.html")
      }
    }
)
```

# 3.4 XML을 벗어난 UI 개발

XML을 사용하지 않고
Kotlin Code만으로 대부분의 UI 개발 가능

- none XML layout

```
res
  drawable
  drawable-v24
  mipmap-anydpi-v26
  mipmap-hdpi
  mipmap-mdpi
  mipmap-xhdpi
  mipmap-xxhdpi
  mipmap-xxxhdpi
  values
  values-night
AndroidManifest.xml
```

List를 만들기 위해서
복잡한 RecyclerView 및 Adapter를 더 이상 사용하지 않아도 됨

- Column, Row

```
@Composable
fun MyView() {
    LazyColumn(modifier = Modifier.wrapContentSize()) {
        items(list) { item ->
            ItemComponent(data = item)
        }
    }
}
```

# 3.6 Live Edit of literals

## Preview 뿐만아니라 Emulator에서도 가능

My Application – ComposeMainActivity.kt [My_Application.app]

ComposeMainActivity.kt

Live Edit of literals: ON | Code | Split | Design

```kotlin
@Composable
fun MyView() {
    var state by remember { mutableStateOf( value: false) }
    Column(modifier = Modifier.fillMaxWidth()) {  this: ColumnScope
        if(!state) {
            SampleText( text: "Hello")
        } else {
            Image(
                painter = painterResource(R.drawable.compose),
                contentDescription = "Compose Image",
                modifier = Modifier.padding(20.dp).size(60.dp).align(
            )
        }
        Button(
            onClick = { state = !state },
            modifier = Modifier.padding(20.dp).fillMaxWidth()
        ) {  this: RowScope
            SampleText( text: "CLICK")
        }
    }
```

Launch succeeded (3 minutes ago)

Hello

CLICK

# 3.7 가볍다

## Tree를 전부 탐색하는 것이 아니라 State가 변경된 지점만 탐색

```kotlin
@Composable
fun LoginScreen(isError: Boolean) {
    if (isError) {
        LoginError()
    }
    LoginInput() // This call site affects where LoginInput is placed in Composition
}

@Composable
fun LoginInput() { /* ... */ }
```

LoginScreen

LoginInput

Recomposition
(isError = true)

LoginScreen

LoginError

LoginInput

https://developer.android.com/jetpack/compose/lifecycle

# 4. 단점

# 4.1 LifeCycle 대응

## App이 pause resume될 때 동작을 추가하고 싶은데..

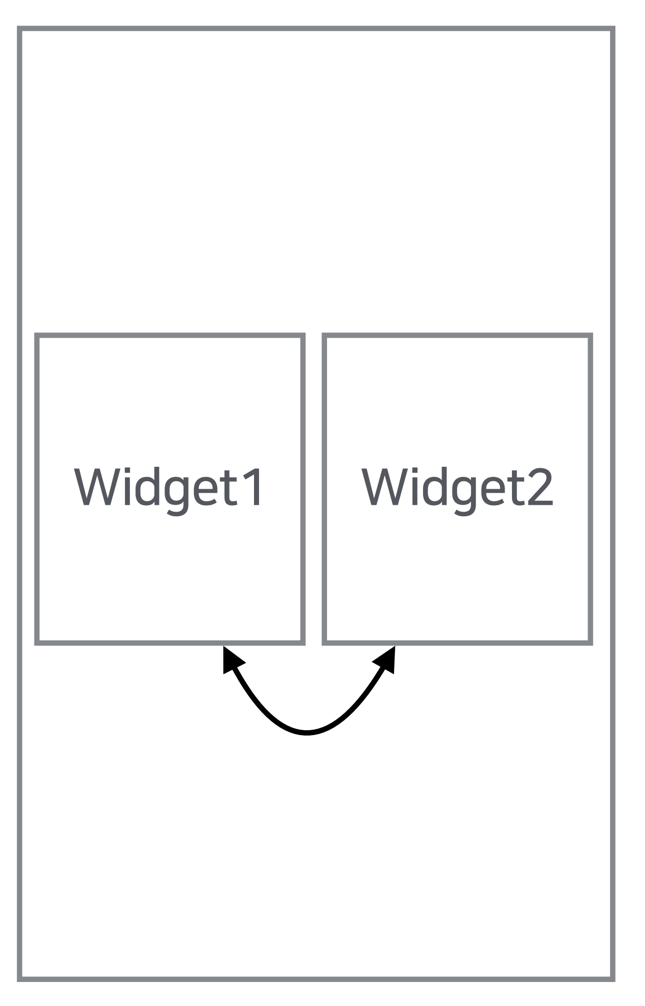
- Composable 안에서 LifeCycle을 Trigger할 수 없다.
- Activity에서 전개

```
override fun onResume() {
    lifecycleStateModel.notifyOnActivityLifeCycleUpdated(
        ActivityLifeCycle.RESUME,
        this@MainActivity
    )

    super.onResume()
}

override fun onPause() {
    lifecycleStateModel.notifyOnActivityLifeCycleUpdated(
        ActivityLifeCycle.PAUSE,
        this@MainActivity
    )
    super.onPause()
}
```

# 4.2 Composable간 공용 변수 사용

Widget 각각이 fun(독립적)이기때문에
같은 화면 안에서도 같은 변수를 설정 해놓고 사용할 수 없다.

- DI or 전역관리 or 변수를 계속 넘겨다녀야 함

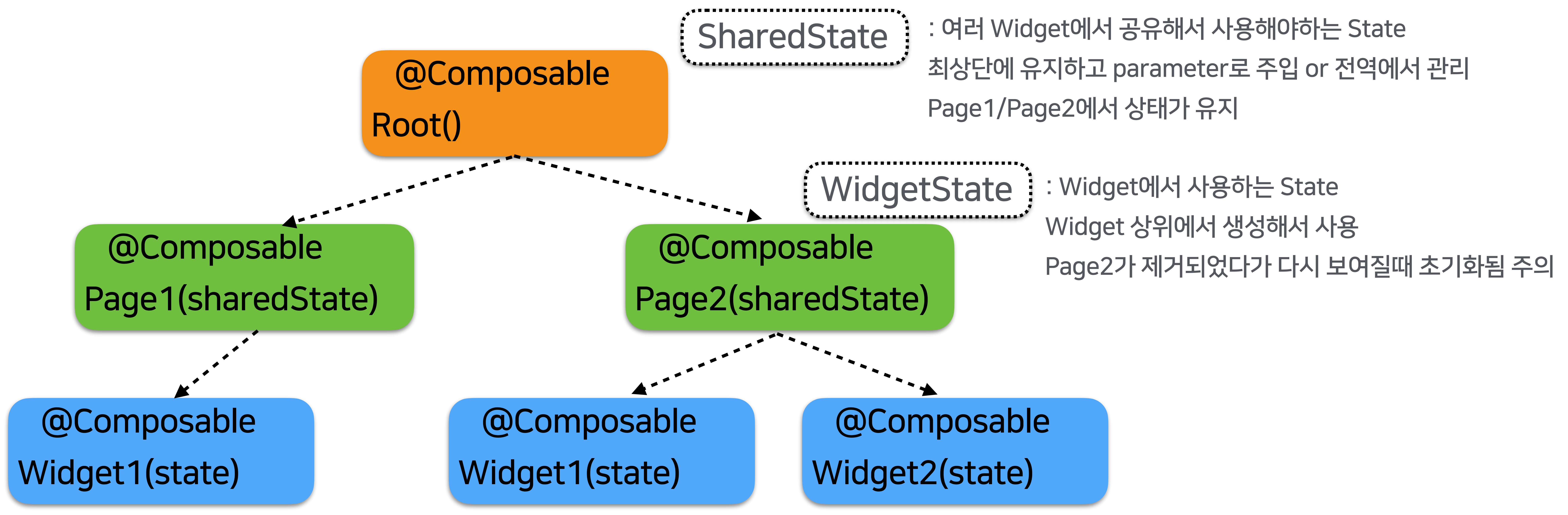



```
var sharedVar = 0
@Composable
fun Widget1() {
    Text(text = "sharedVar = $sharedVar")
}
@Composable
fun Widget2() {
    Text(text = "sharedVar = $sharedVar")
}
```

# 4.3 State 관리

## 전역에서 유지해야하는 State, 페이지안에서만 쓸 State

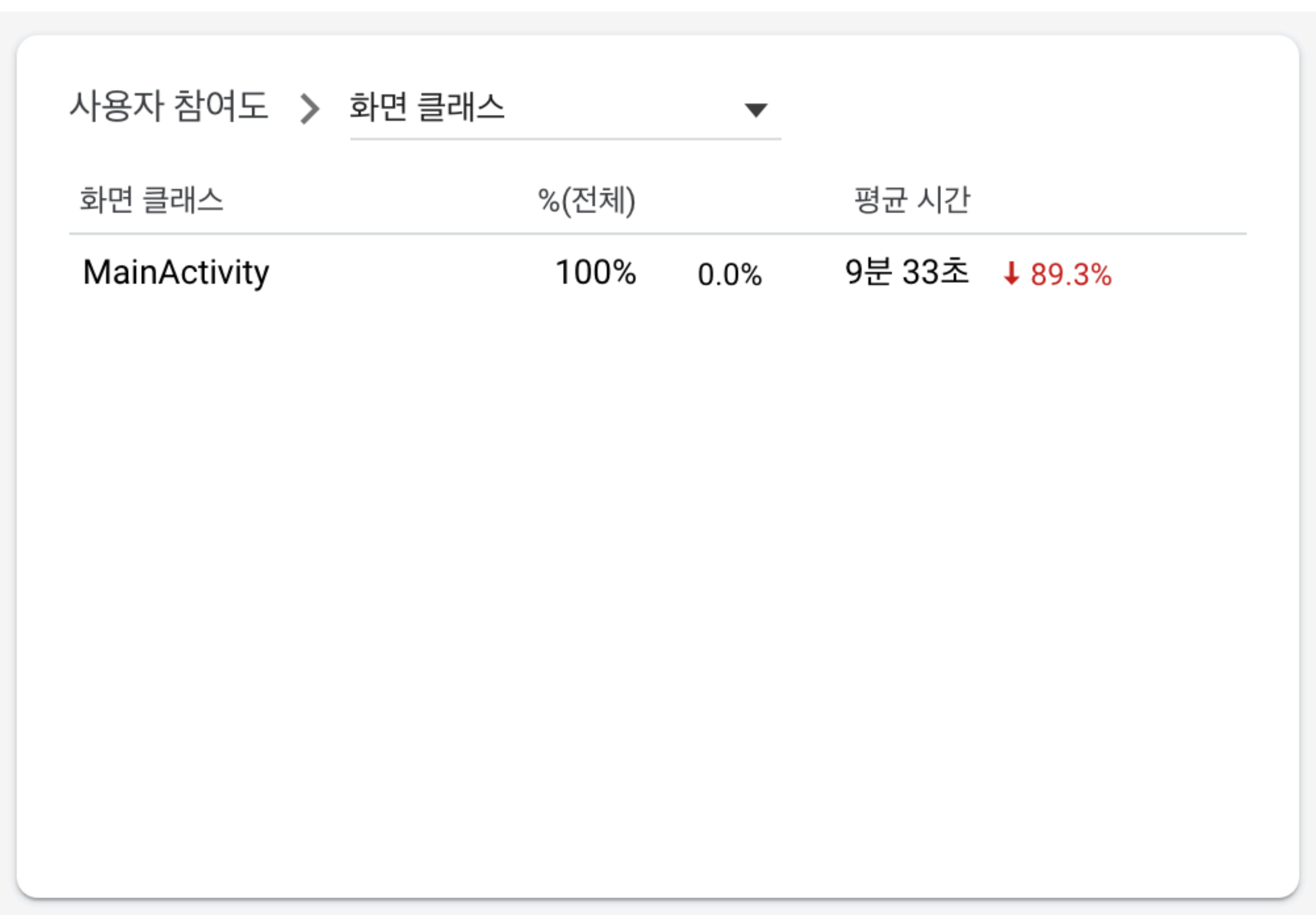
- remember, rememberSaveable 적절히 활용 필요



# 4.4 Firebase Tracking

## Activity Base가 아니기 때문에 화면별 Tracking이 쉽지 않음

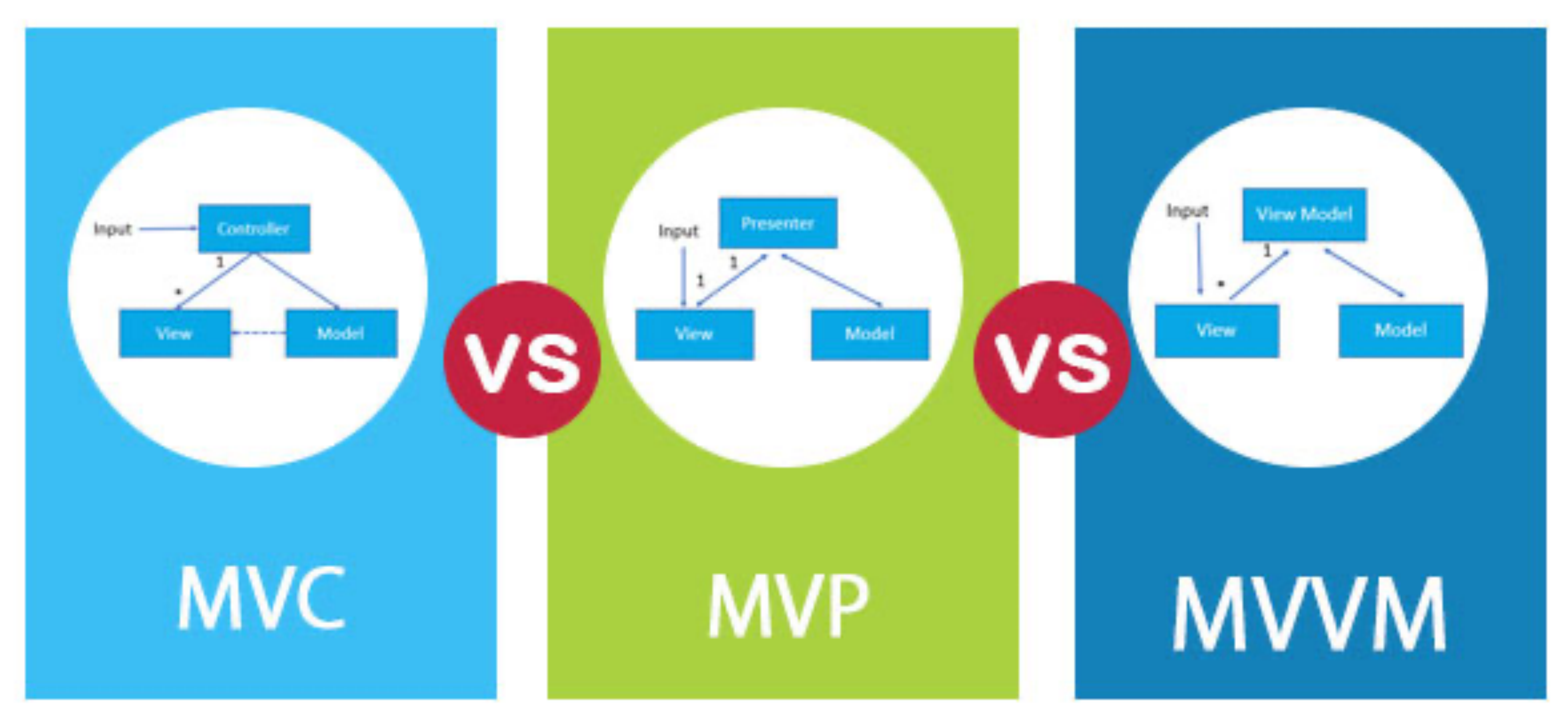
- 화면 진입시점을 체크해서 Logging을 추가해줘야 함



# 4.5 선호되는 Architecture의 부재

## MVP? MVVM? Clean Architecture?

- Jetpack Compose는 이렇게 쓰는게 좋더라. 하는 자료들이 아직 별로 없음



https://laptrinhx.com/mvc-vs-mvp-vs-mvvm-1934235033/
https://www.raywenderlich.com/3595916-clean-architecture-tutorial-for-android-getting-started

# 4.6 실수할 여지가 많다.

## 모든 페이지가 State로 연결되어있고 눈으로 확인할 수 없기때문에..

- 명령형은 내가 명령을 내리기 때문에 내가 하지 않은 일이 발생하지 않음
- 선언형은 구성을 잘못해놓으면 내가 하지 않은 일도 **State**관리의 오류로 발생 가능

- But 반대로 실수할 여지가 적을 수도..

- **data(state)**에 의존하기때문에 처음 구성을 잘 해놓으면 논리적으로 오류가 발생할 일이 없음

```
@Composable
fun LoginScreen(isError: Boolean) {
    if (isError) {
        LoginError()
    }
}
```

※ isError 데이터가 바뀌지 않는 이상 LoginError는 발생하지 않음

<결론>

Spec의 범위가 크지 않다면, 한 번쯤 적용해
보는 것을 추천!

# 5. 주의할 점

# 5.1 State를 변경했는데 재구성이 안되요

## Recomposition은 해당 State 값을 사용하는 Composable만

```
@Composable
fun NamePicker(
    header: String,
    names: List<String>,
    onNameClicked: (String) -> Unit
) {
    Column {
        // this will recompose when [header] changes, but not when [names] changes
        Text(header, style = MaterialTheme.typography.h5)
        Divider()

        // LazyColumn is the Compose version of a RecyclerView.
        // The lambda passed to items() is similar to a RecyclerView.ViewHolder.
        LazyColumn {
            items(names) { name ->
                // When an item's [name] updates, the adapter for that item
                // will recompose. This will not recompose when [header] changes
                NamePickerItem(name, onNameClicked)
            }
        }
    }
}
```

# 5.2 State를 변경했는데 재구성이 안되요2

## Model 쓸때 조심할 것

- Model이 State라도 Model안의 parameter만 변경될 경우 갱신X

```
data class TestModel(var num: Int)

@Composable
fun OnClickTest() {
    val model by remember { mutableStateOf(TestModel(0)) }

    Column(modifier = Modifier.fillMaxWidth()) {
        Button(
            onClick = { model.num++ },
        ) {
            Text("CLICK")
        }

        Text("data ${model.num}")
    }
}
```

# 5.2 State를 변경했는데 재구성이 안되요2

## Model 쓸때 조심할 것

- State인 Model자체가 변경되어야 Recomposition

```
data class TestModel(var num: Int)

@Composable
fun OnClickTest() {
    val model by remember { mutableStateOf(TestModel(0)) }

    Column(modifier = Modifier.fillMaxWidth()) {
        Button(
            onClick = { model = TestModel(model.num.plus(1)) },
        ) {
            Text("CLICK")
        }

        Text("data ${model.num}")
    }
}
```

# 5.3 Composable 함수가 너무 자주 불려요

Composable은 여러번 호출 될 수 있기때문에
Composable 안에서 다른 함수를 호출할 때는 주의!

- LaunchedEffect 사용

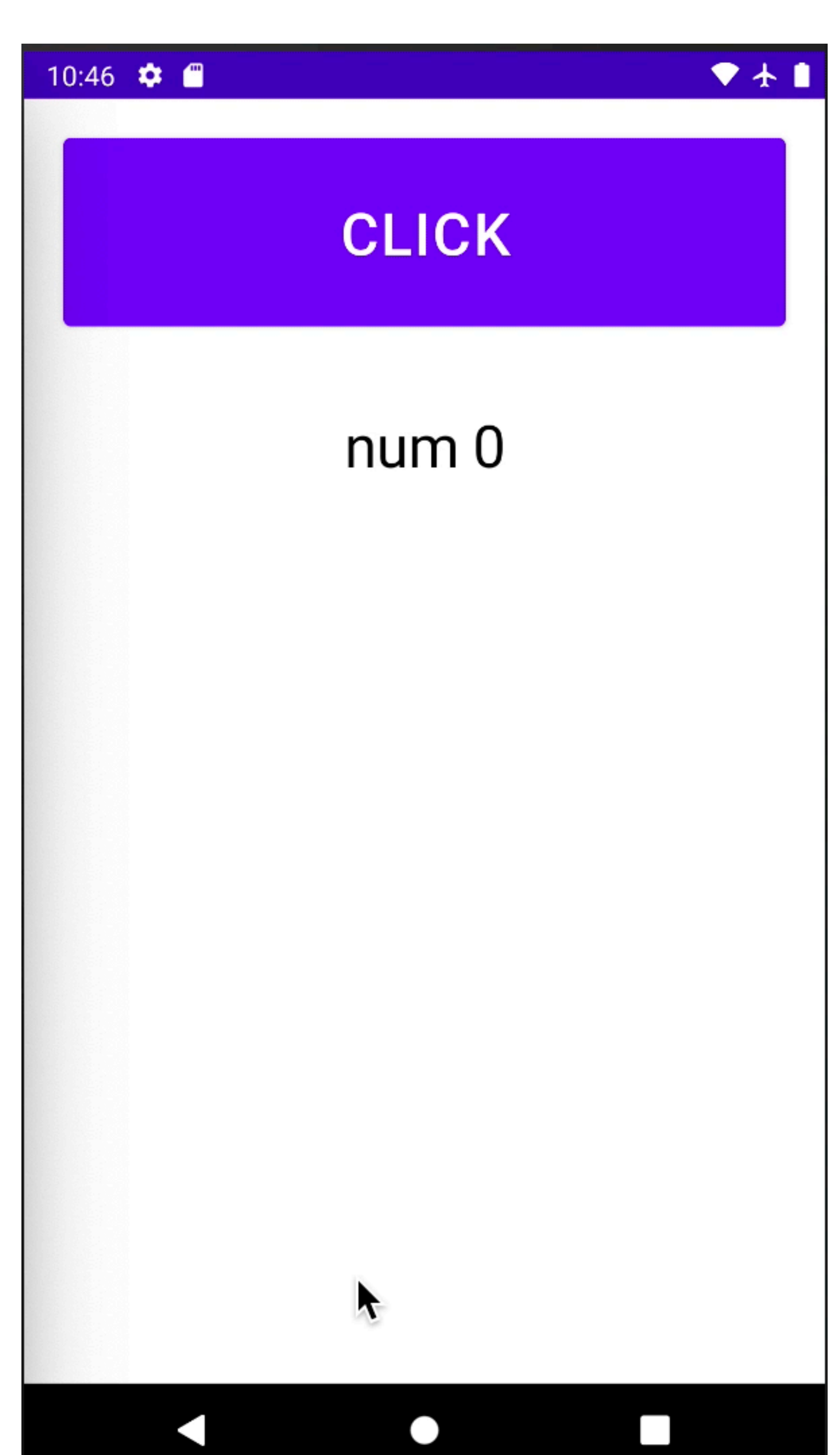
```
@Composable
fun MyView() {
    refreshModel() // Recomposition마다 호출

    Text("model")

    LaunchedEffect(Unit) { // initialComposition에만 호출
        refreshModel()
    }
}
```

# 5.4 변수 생성 주의

## Composable 내부에서 변수를 생성할 경우, 꼭 State를 사용

- State가 아닌 변수는 값이 변경되어도 Recomposition 하지 않음

```
@Composable
fun OnClickTest() {
    var num = 0

    Column(modifier = Modifier.fillMaxWidth()) {
        Button(
            onClick = { num++ },
        ) {
            Text("CLICK")
        }

        Text("num $num")
    }
}
```

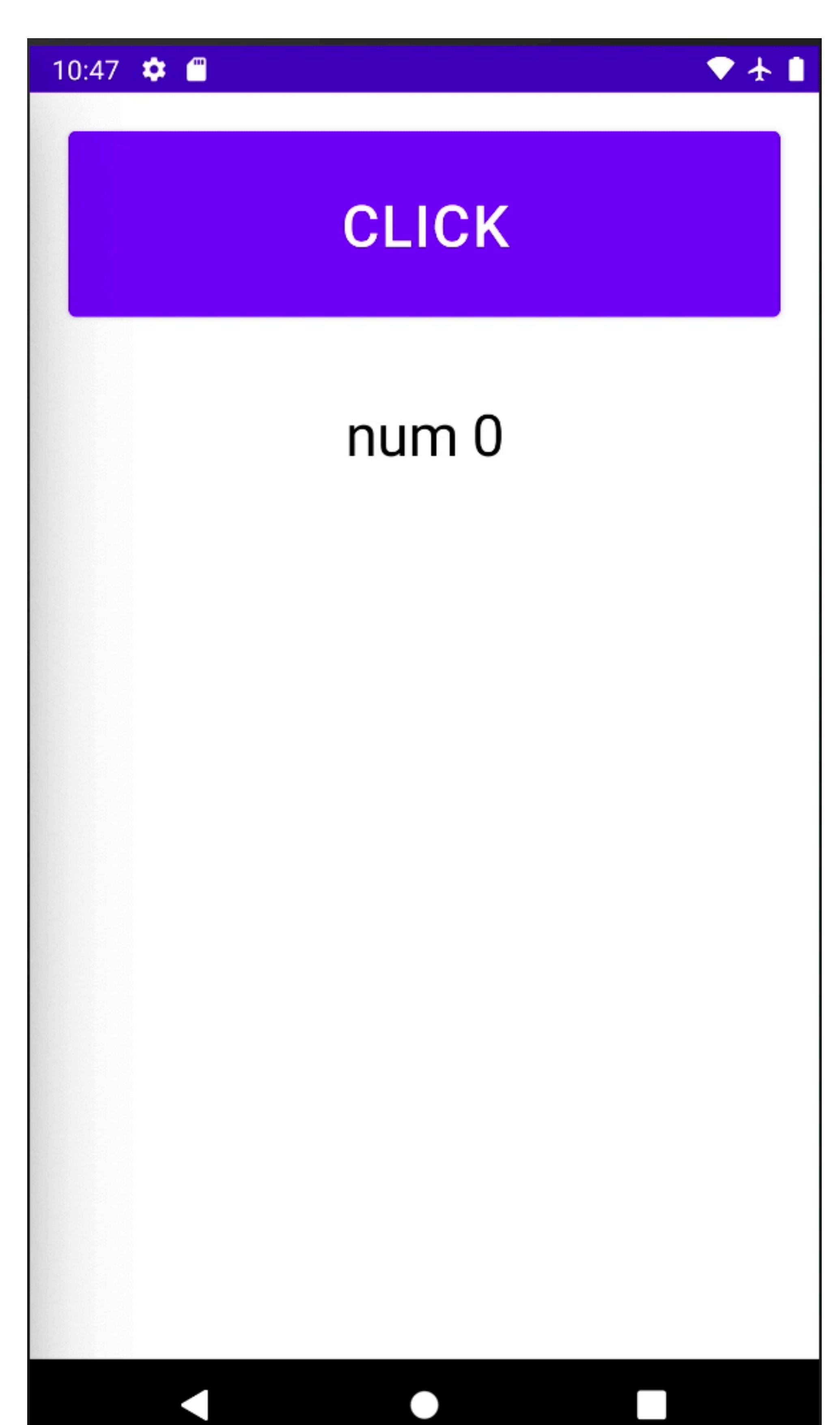


# 5.4 변수 생성 주의

## Composable 내부에서 변수를 생성할 경우, 꼭 State를 사용

- State가 변경될때마다 State를 사용하는 Composable은 Recomposition

```kotlin
@Composable
fun OnClickTest() {
    var num by remember { mutableStateOf(0) }

    Column(modifier = Modifier.fillMaxWidth()) {
        Button(
            onClick = { num++ },
        ) {
            Text("CLICK")
        }

        Text("num $num")
    }
}
```

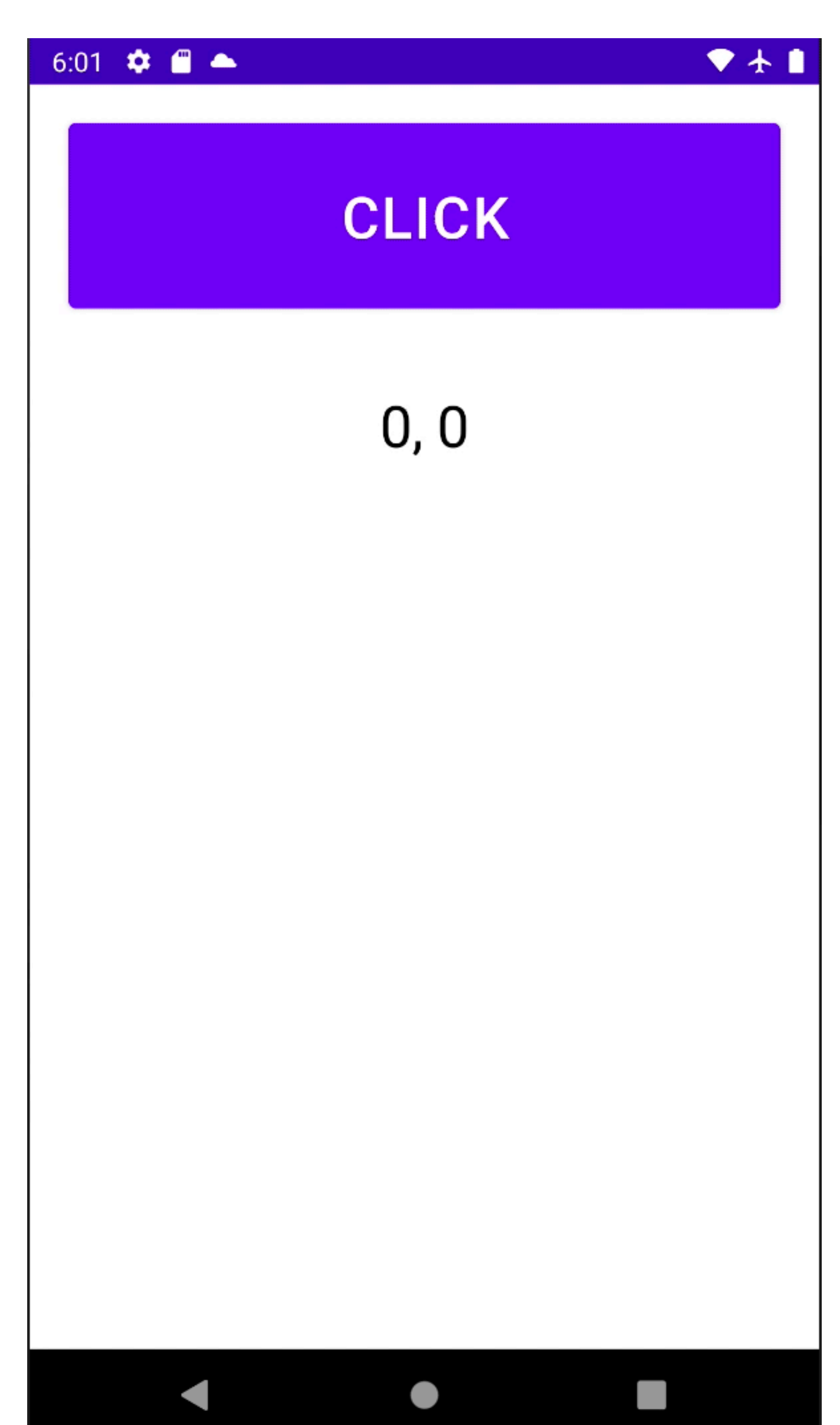


# 5.5 변수 생성 주의2

## Composable 내부에서 변수를 생성할 경우, remember를 꼭 사용

- Recomposition 과정에서 변수가 재생성 됨

```
@Composable
fun OnClickTest() {
    var var1 by remember { mutableStateOf(0) }
    var var2 = 0

    Column(modifier = Modifier.fillMaxWidth()) {
        Button(
            onClick = { var1++; var2++ },
        ) {
            Text("CLICK")
        }
        Text("$var1, $var2")
    }
}
```

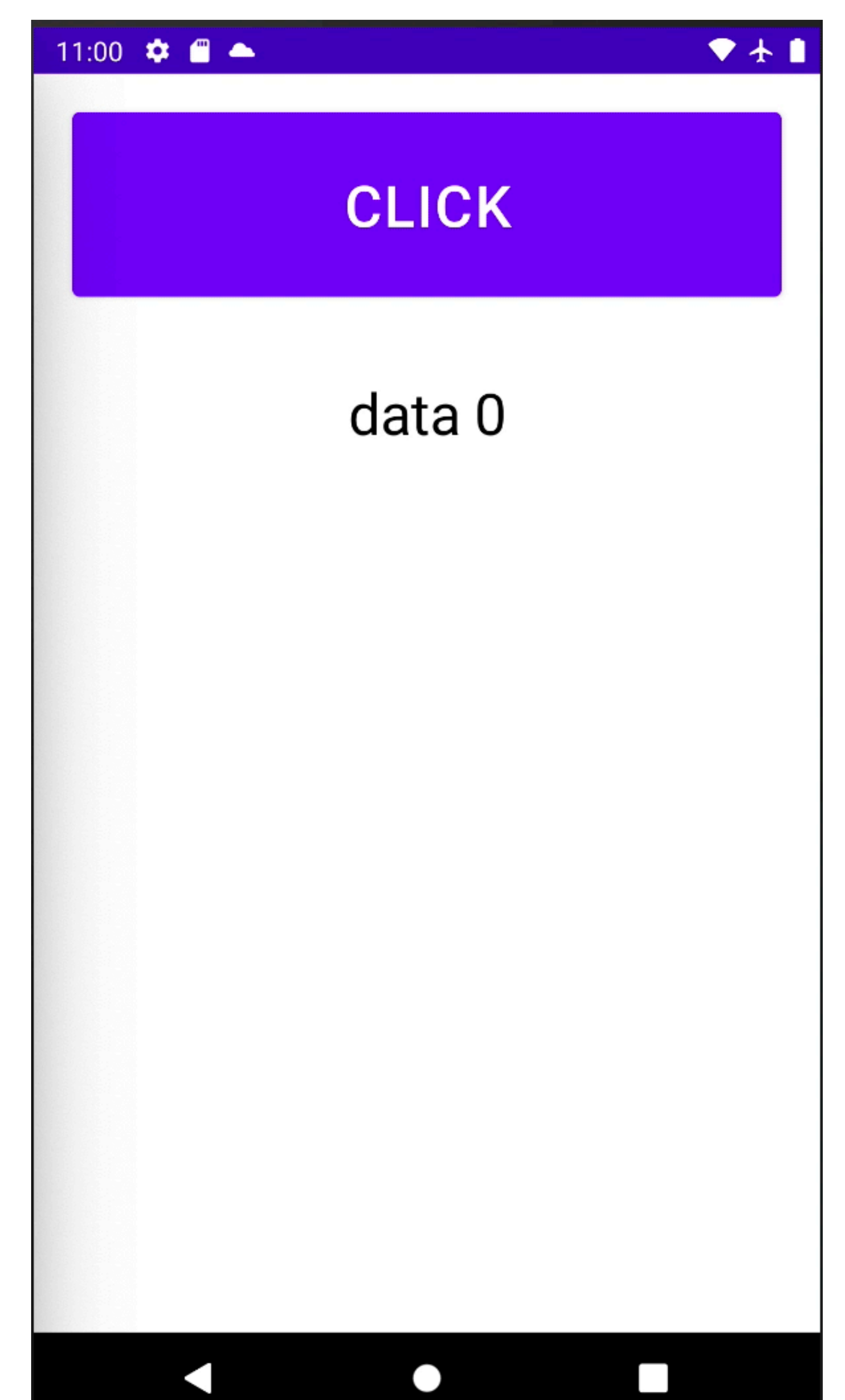


# 5.6 객체 생성 주의

## Composable 내부에서 객체를 생성할 경우, remember를 꼭 사용

- Recomposition 과정에서 객체가 재생성 됨

```
data class TestData(val num: MutableLiveData<Int> = MutableLiveData(0))

@Composable
fun OnClickTest() {
    val data = TestData()
    val num = data.num.observeAsState()

    Column(modifier = Modifier.fillMaxWidth()) {
        Button(
            onClick = { data.num.value = num.value?.plus(1) },
        ) {
            Text("CLICK")
        }
        Text("data ${num.value}")
    }
}
```

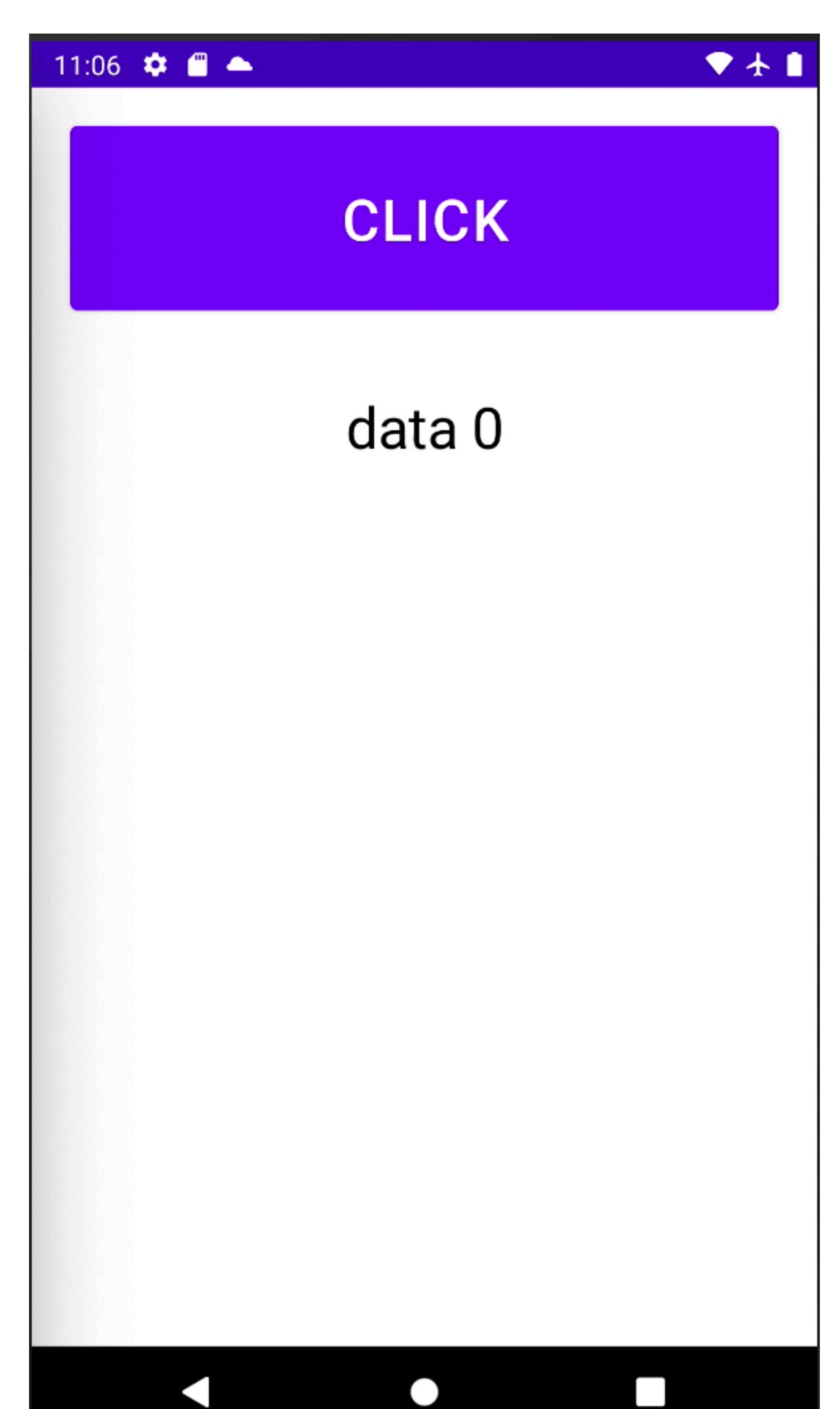


# 5.6 객체 생성 주의

## Composable 내부에서 객체를 생성할 경우, remember를 꼭 사용

- remember를 사용하여 재생성 방지 or 상위에서 주입받아 사용

```
data class TestData(val num: MutableLiveData<Int> = MutableLiveData(0))

@Composable
fun OnClickTest() {
    val data by remember { mutableStateOf(TestData()) }
    val num = data.num.observeAsState()

    Column(modifier = Modifier.fillMaxWidth()) {
        Button(
            onClick = { data.num.value = num.value?.plus(1) },
        ) {
            Text("CLICK")
        }
        Text("data ${num.value}")
    }
}
```

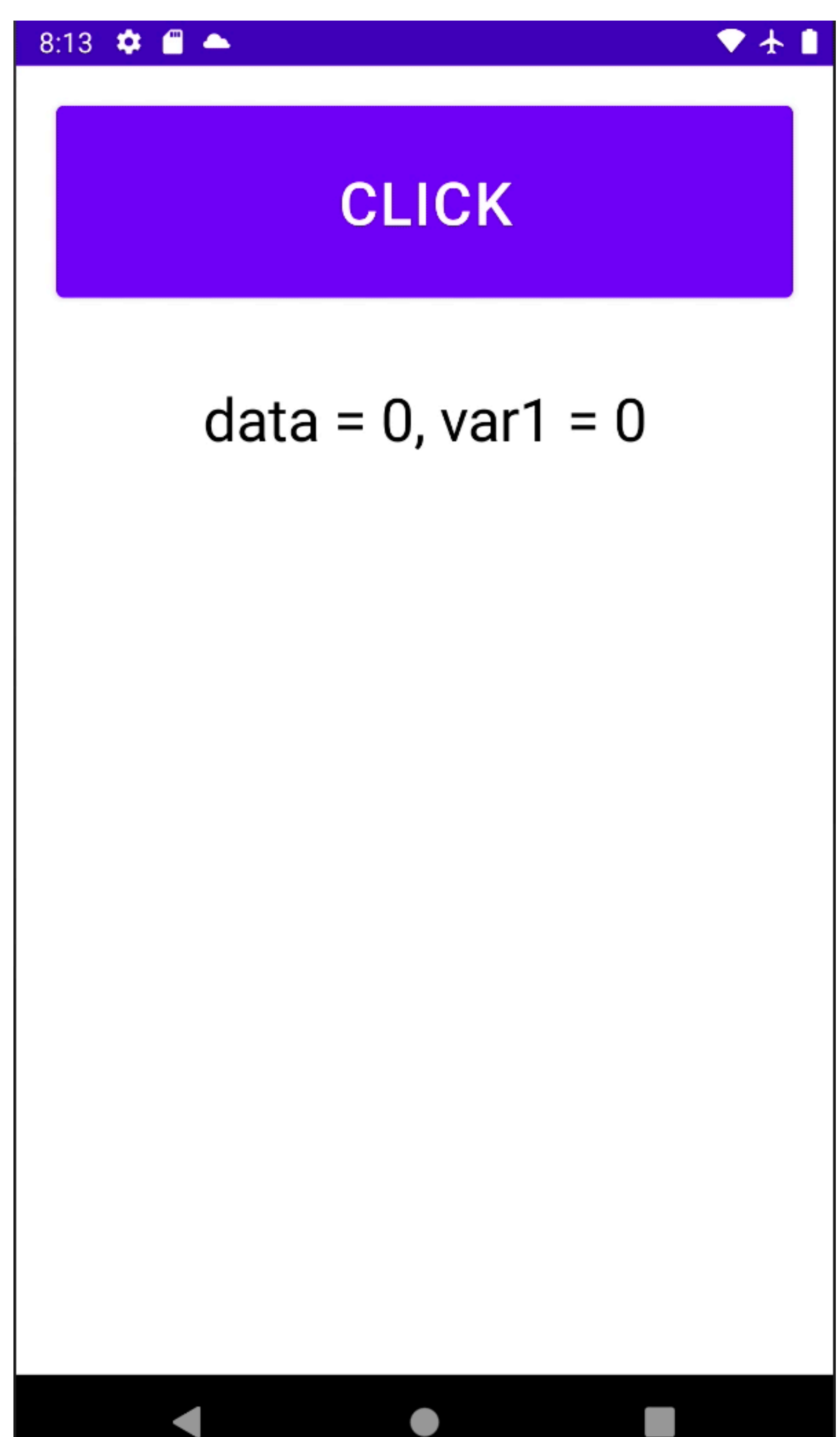


# 5.7 Parcelable

## Parcelable, Serializable한 Object를 사용해서 State를 구성

- Object Reference가 바뀌어도 내부 값의 변경을 체크해서 Recomposition

```
@Parcelize
data class TestData(val num: Int): Parcelable

var var1 = 0
@Composable
fun Root() {
    var data by remember { mutableStateOf(TestData(0)) }

    Column(modifier = Modifier.fillMaxWidth()) {
        Button(
            onClick = { data = TestData(data.num); var1++ },
        ) {
            Text("CLICK")
        }

        Text("data = ${data.num}, var1 = $var1")
    }
}
```

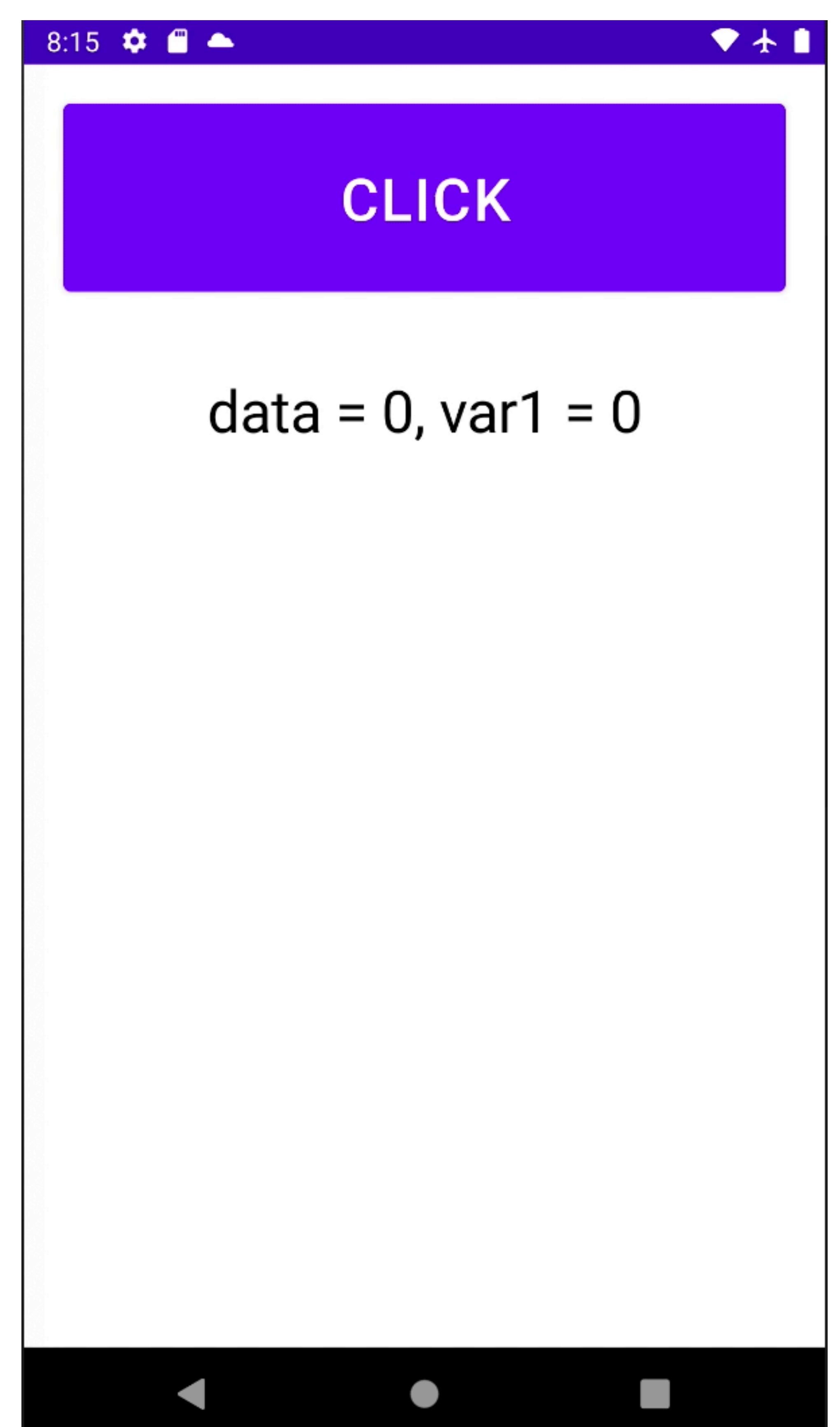


# 5.7 Parcelable

## Parcelable, Serializable한 Object를 사용해서 State를 구성

- Object Reference가 바뀌어도 내부 값의 변경을 체크해서 Recomposition

```
@Parcelize
data class TestData(val num: Int): Parcelable

var var1 = 0
@Composable
fun Root() {
    var data by remember { mutableStateOf(TestData(0)) }

    Column(modifier = Modifier.fillMaxWidth()) {
        Button(
            onClick = { data = TestData(data.num + 1); var1++ },
        ) {
            Text("CLICK")
        }

        Text("data = ${data.num}, var1 = $var1")
    }
}
```



# 6. State에 맞게 추가한 개념

# 6.1 MVSM

Model - View - StateModel

**UI = *f*(State)**

- StateModel의 State의 변화에 따라 UI가 Seamless하게 변환

# 6.1 MVSM

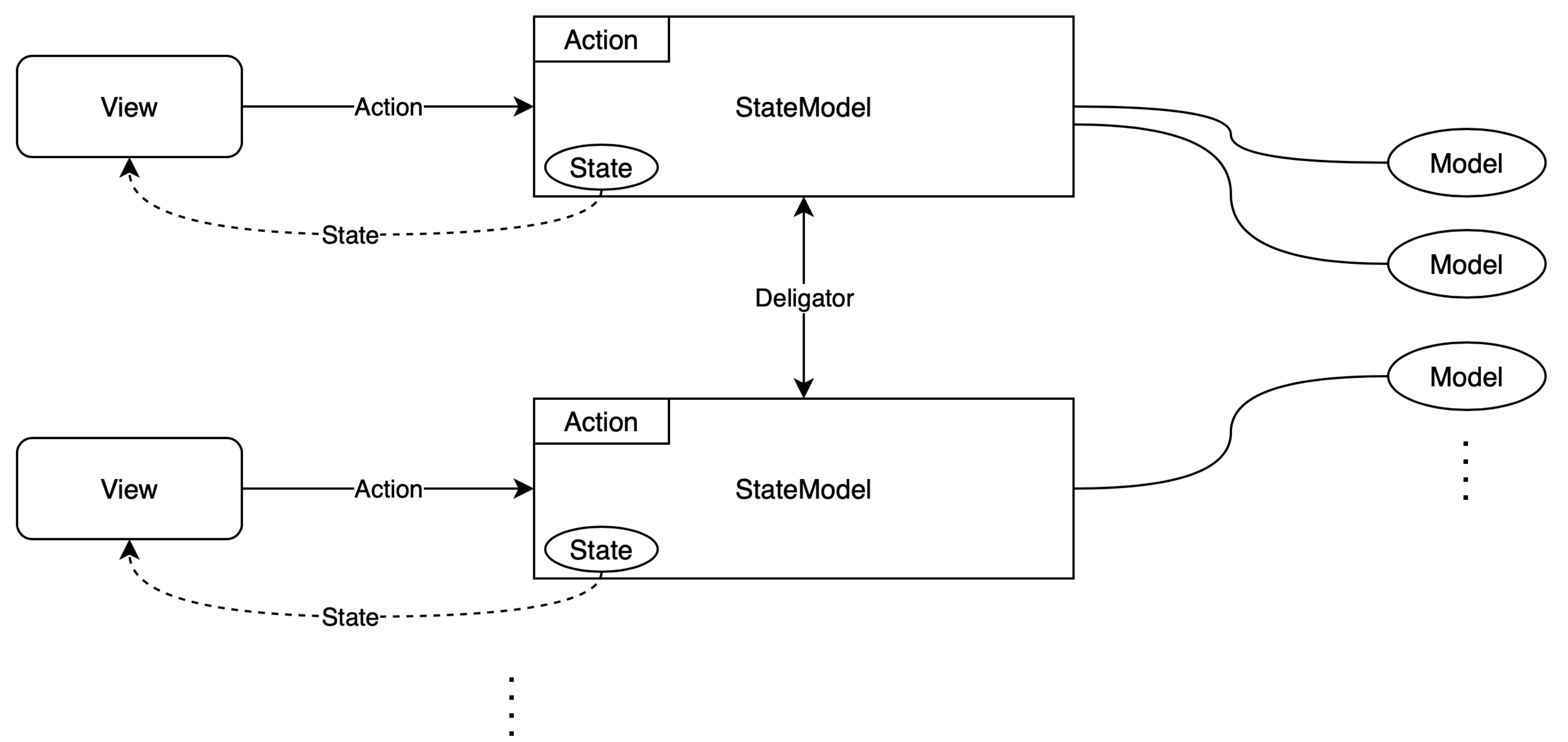
View
Action
StateModel
State

Deligator

View
Action
StateModel
State

Model
Model
Model

# 6.1 MVSM




- Action : View에서 StateModel로 Action을 넘겨 로직을 수행
  → State 변경
  → View 자동 갱신 by State

- Delegator : Action을 전달할 때 다음 동작을 정의

**View(StateModel(Action))**
**= State**

View는 StateModel에 Action을 적용해서 나온 State에 대한 결과물

# Thank You