

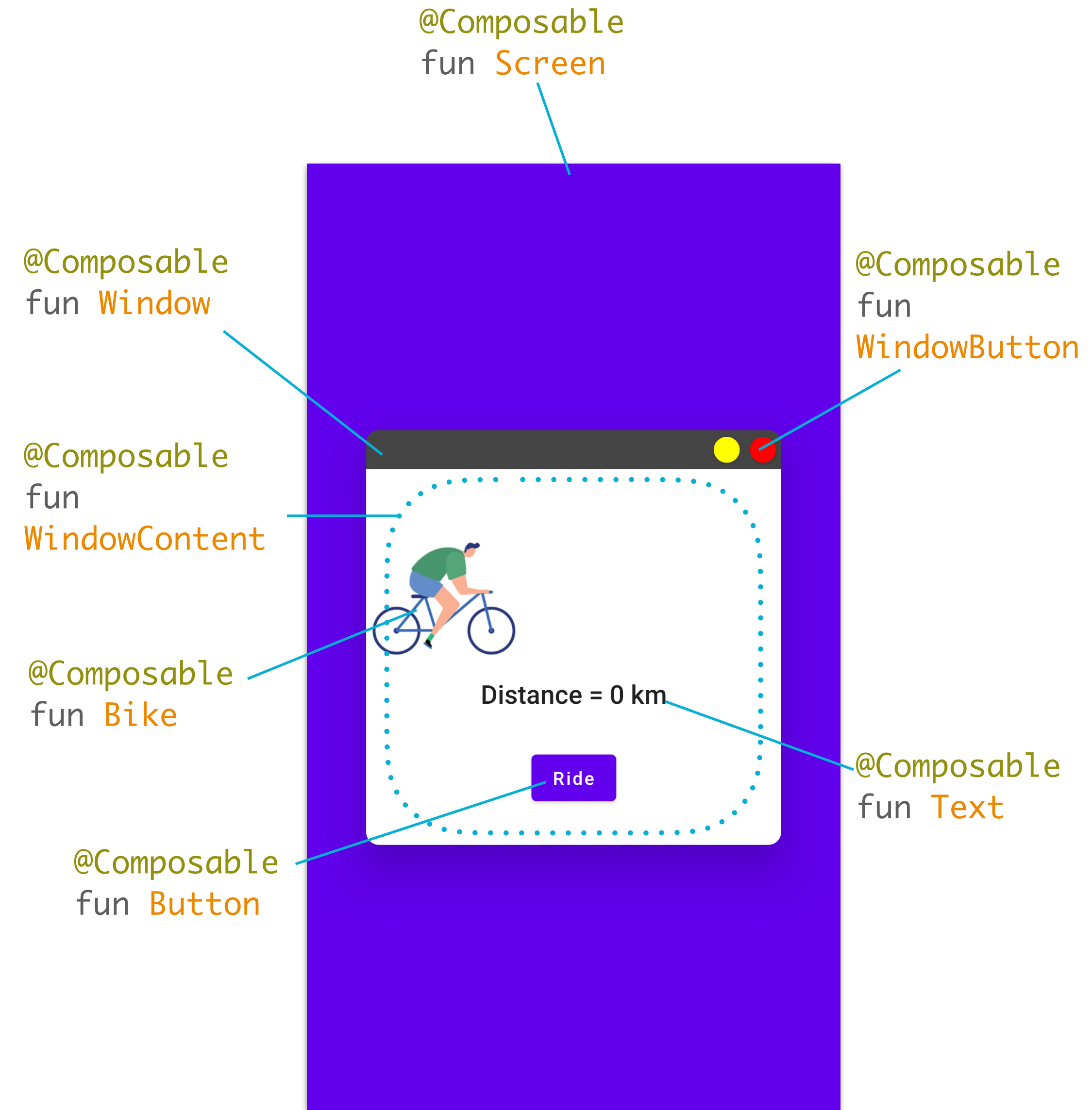
Side-Effect in Compose

Prerequisites : Terminologies of Compose

Composable

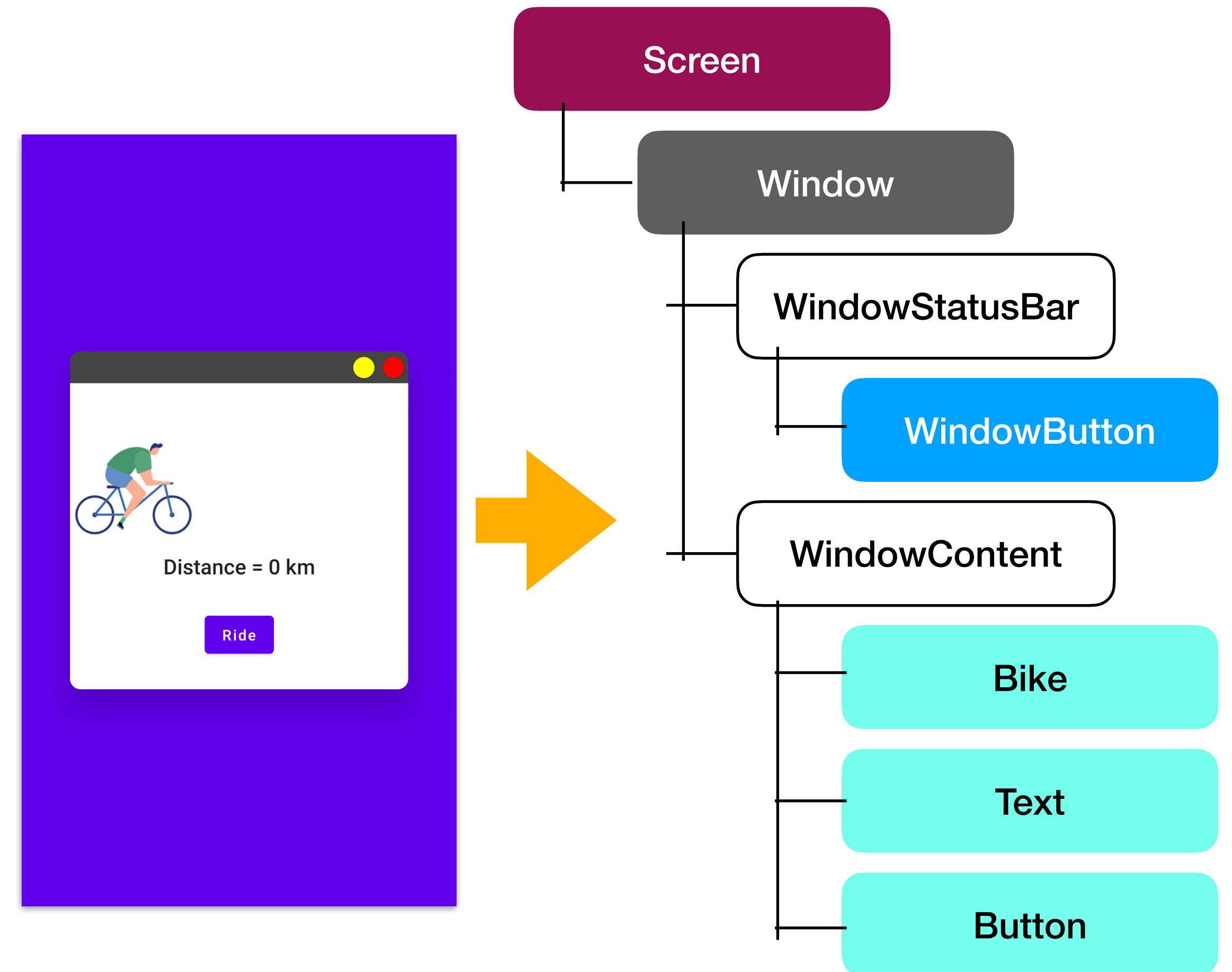
- Compose로 개발된 앱의 구성 단위 function
- Composable = View 가 아님에 유의

※ *Composable* 수행 결과로 생성된 *View*가 우리 눈에 보이는 것
*View*를 생성하지 않는 *Composable*도 존재할 수 있음



Composition

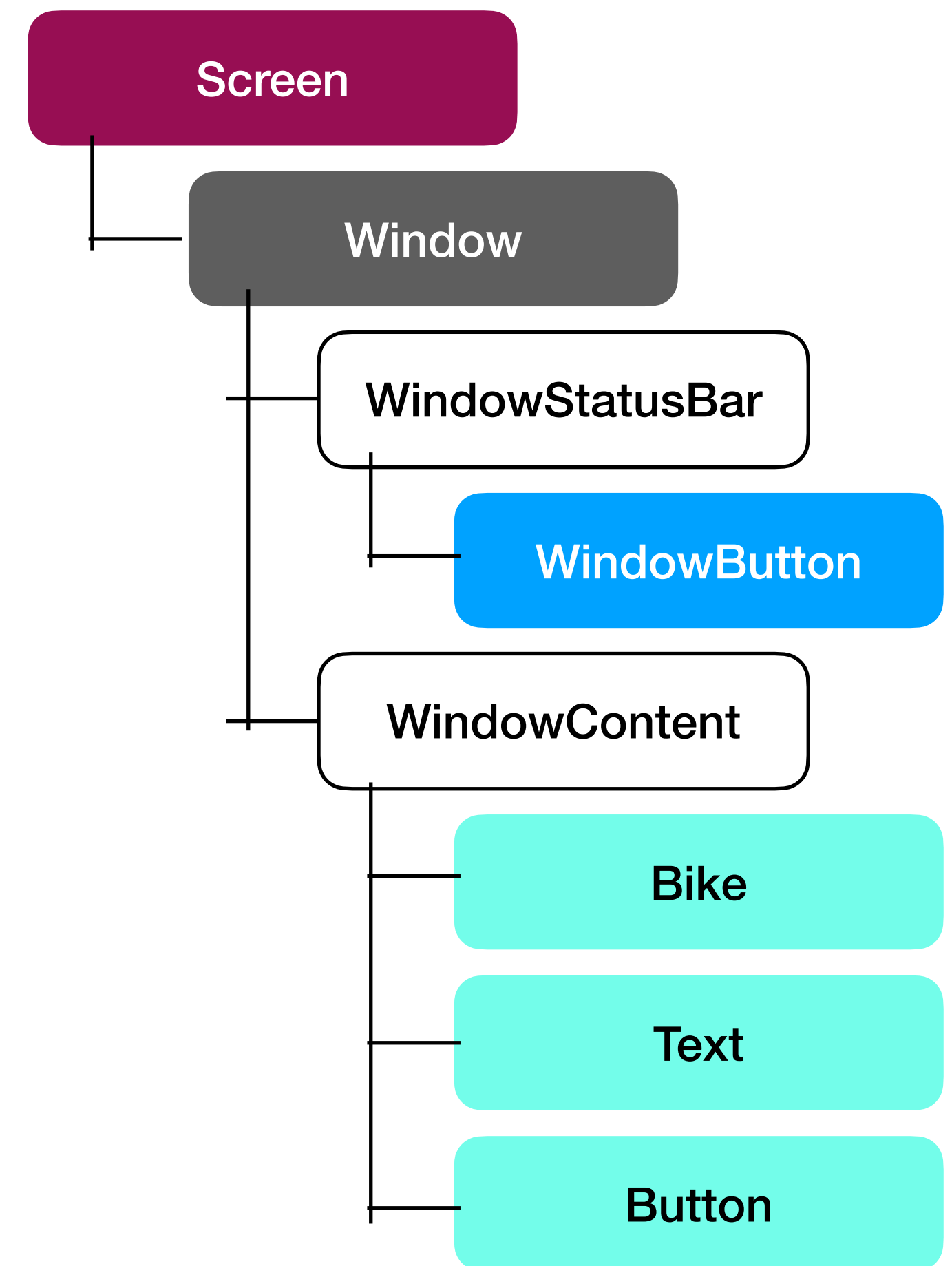
1. Composable로 구성된 tree structure



Composition

2. Composition을 구성(compose)하는 행위

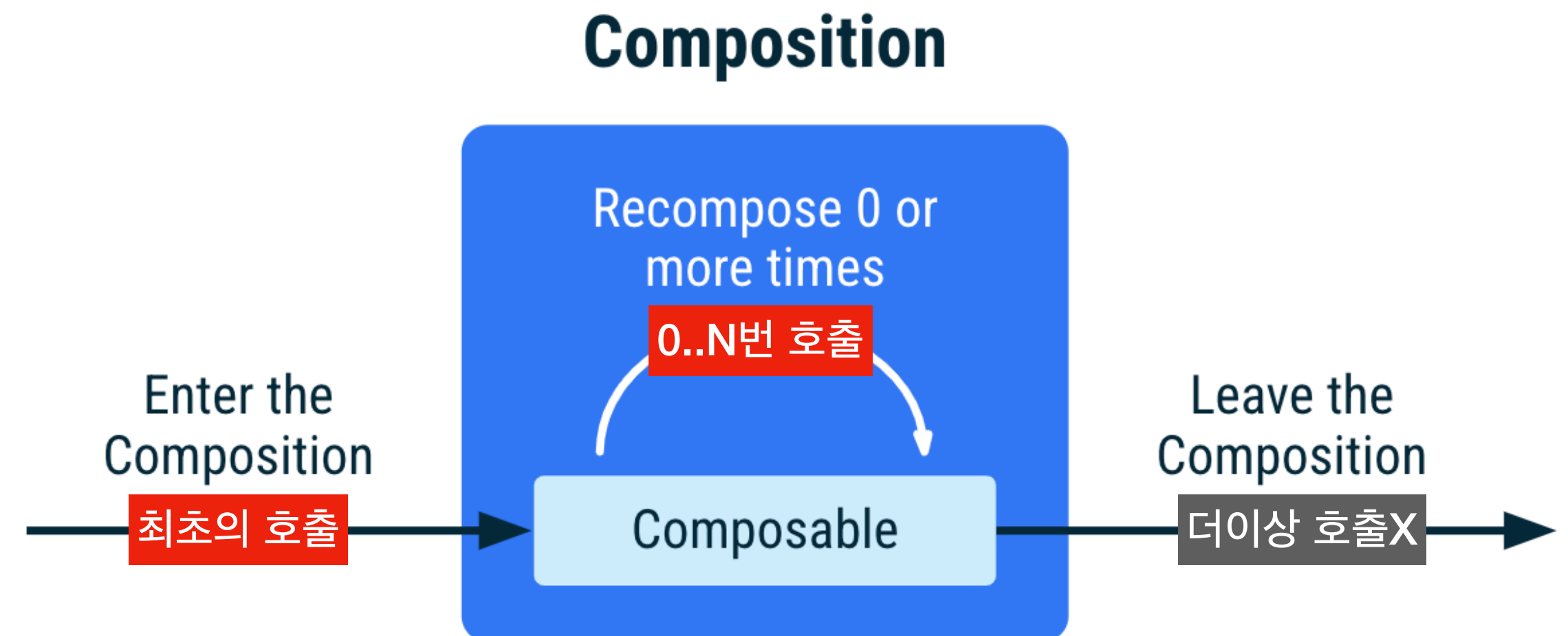
- initial composition
 - 최초의 Composition(tree)을 생성하는 행위
- recomposition
 - 최초의 Composition(tree)에 변경을 가하는 행위



Lifecycle of Composables

- Composition(tree)으로의 enter
- recompose
- Composition(tree)에서의 leave

Composable은 몇번이고 호출될 수 있다



Composer, Recomposer

- Composer
 - Composition(tree) 관리 주체 (tree nodes 사이의 관계/구성을 꿰고 있는 주체)
- Recomposer
 - composition을 수행하고 그 결과를 Composition(tree)에 반영하는 scheduler

State, Snapshot

- State
 - Composable이 의존하는 value의 holder Type (e.g. *State<Boolean>*)
 - State 객체의 value가 변경되면 recomposition이 일어난다
- Snapshot ÷ 특정 시점에 State의 value

Kotlin Flow

- 여러 값을 순차적으로 발산하는 Type
- Flow 내부에서는 값을 emit하고 (=생산)
Flow 밖에서는 값들을 collect한다 (=소비)
- Flow is cold (소비자가 있어야 생산을 시작한다)
- coroutine이 필요하다

※ `collect()` 함수가 suspend function 이다

```
fun myFlow(): Flow<Int> = flow {  
    for (i in 1..3) {  
        delay(100)  
        emit(i)  
    }  
}  
  
fun main() = runBlocking {  
    myFlow().collect {  
        println(it)  
    }  
}
```

Side-Effect

Compose에서의 의미

- Composable이 자기 scope 밖의 상태를 바꾸는 행위 또는 그 결과 상태
- Composable은 몇번이고 호출될 수 있기 때문에

side-effect 생성 코드를 Composable 내에 만들지 말아야 한다

끝

감사합니다.

Side-Effect

Compose에서의 의미

- Composable이 자기 scope 밖의 상태를 바꾸는 행위 또는 그 결과 상태
- Composable은 몇번이고 수행될 수 있기 때문에
side-effect 생성 코드를 Composable 내에 만들지 말아야 한다
- 그럼에도 불구하고 하겠다면, 철저히 제어된 환경에서만 side-effect를 이용해야 한다
 - 너의 코드가 언제 수행/취소/완료가 되는지, 그 결과 상태가 의도한 바인지 숙지해라
 - 이를 도와주는 API를 이용해라

Side-Effect API

LaunchedEffect

Composable에서 Coroutine을 launch한다 * *Recomposer*가 *CoroutineContext*를 가지고 있기에 가능한 일

-  는 언제 실행?
 - `@Composable`
`fun LaunchedEffect()` 가
composition에 enter할 때
-  는 언제 취소?
 - key가 바뀔 때
 - `@Composable`
`fun LaunchedEffect()` 가
composition에서 leave할 때

```
@Composable
fun MyComposable() {
    LaunchedEffect(key1 = Unit) {
        asynchronousRoutine()
    }
}

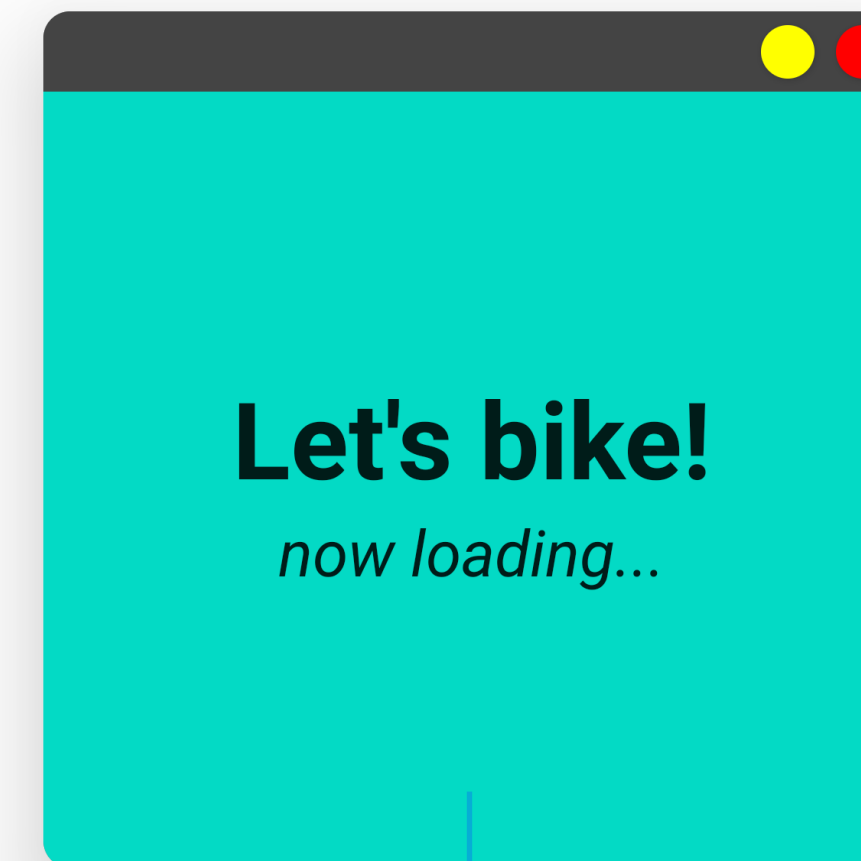
suspend fun asynchronousRoutine() {
    delay(1_000_000)
}
```

LaunchedEffect

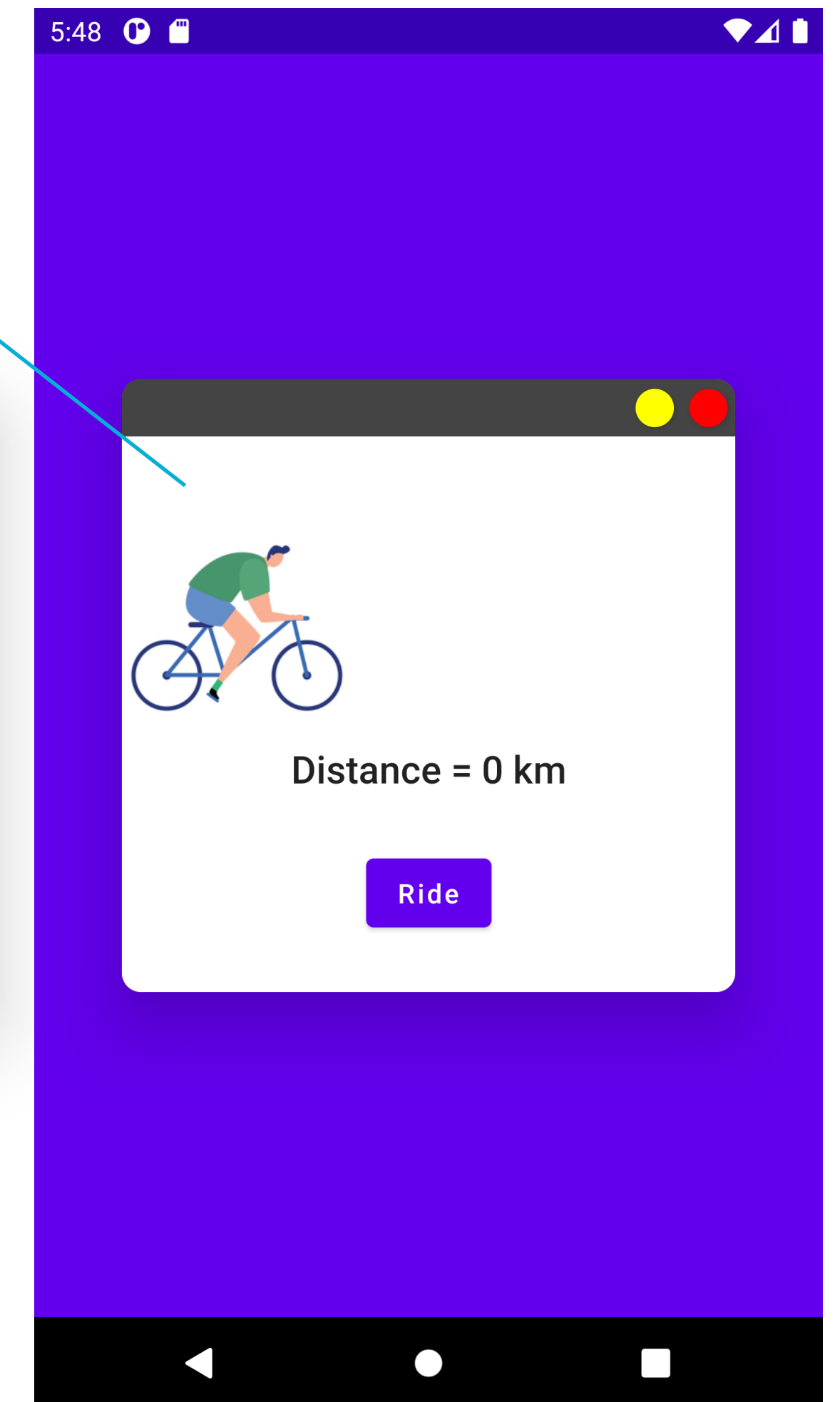
예제

```
@Composable
fun SplashScreenContent(onReady: () -> Unit) {
    //FIXME loadingResources() 수행이 끝나면
    //FIXME onReady()를 호출하고 싶다.
    //FIXME 어떡하지?
}
```

@Composable
fun BikeWindowContent



@Composable
fun SplashScreenContent



rememberUpdatedState

recomposition에서 최신 값을 지킨다

- 이 코드에는 문제가 있다. 무엇일까?

```
@Composable
fun SplashScreenContent(onReady: () -> Unit) {

    // 생략

    LaunchedEffect(Unit) {
        loadingResourcesForALongTime()
        onReady()
    }
}
```


rememberUpdatedState

recomposition에서 최신 값을 지킨다

- 수정된 코드

```
@Composable
fun SplashScreenContent(onReady: () -> Unit) {

    // 생략

    val currentOnReady by rememberUpdatedState(onReady)
    LaunchedEffect(Unit) {
        loadingResourcesForALongTime()
        currentOnReady()
    }
}
```

=

```
val currentOnReady by remember {
    mutableStateOf(onReady)
}.apply {
    value = onReady
}
```

rememberCoroutineScope

Composable이 기억하는 CoroutineScope를 생성한다

- scope는 언제 실행?
 - 당신이 원할 때, 원하는 곳에서
- scope는 언제 취소?
 - `@Composable`
`fun rememberCoroutineScope()` 가
composition에서 leave할 때

```
@Composable
fun MyComposable(name: String) {
    val scope = rememberCoroutineScope()

    scope.launch {
        asynchronousRoutine()
    }

    YourComposable(scope)

    TheirComposable(function = {
        scope.launch {
            asynchronousRoutine()
        }
    })
}
```

LaunchedEffect vs. rememberCoroutineScope

둘은 비슷해보이지만 다르다

- 둘의 차이는 무엇일까?

```
@Composable
fun SplashScreenContent(onReady: () -> Unit) {

    // 생략

    LaunchedEffect(Unit) {
        loadingResourcesForALongTime()
        onReady()
    }
}
```



```
@Composable
fun SplashScreenContent(onReady: () -> Unit) {

    // 생략

    val scope = rememberCoroutineScope()
    scope.launch {
        loadingResourcesForALongTime()
        onReady()
    }
}
```

snapshotFlow

State의 Snapshot을 추적하여 Flow로 변환

-  는 언제 실행?
 - Flow에 소비자가 생긴 순간부터
 - Flow가 참조하는 State의 value가 바뀔 때마다
※ like distinctUntilChanged()
-  는 언제 취소?
 - 직접 취소해야 함

```
@Composable
fun MyButton(isEnabled: Boolean) {
    val currentEnabled by rememberUpdatedState(isEnabled)

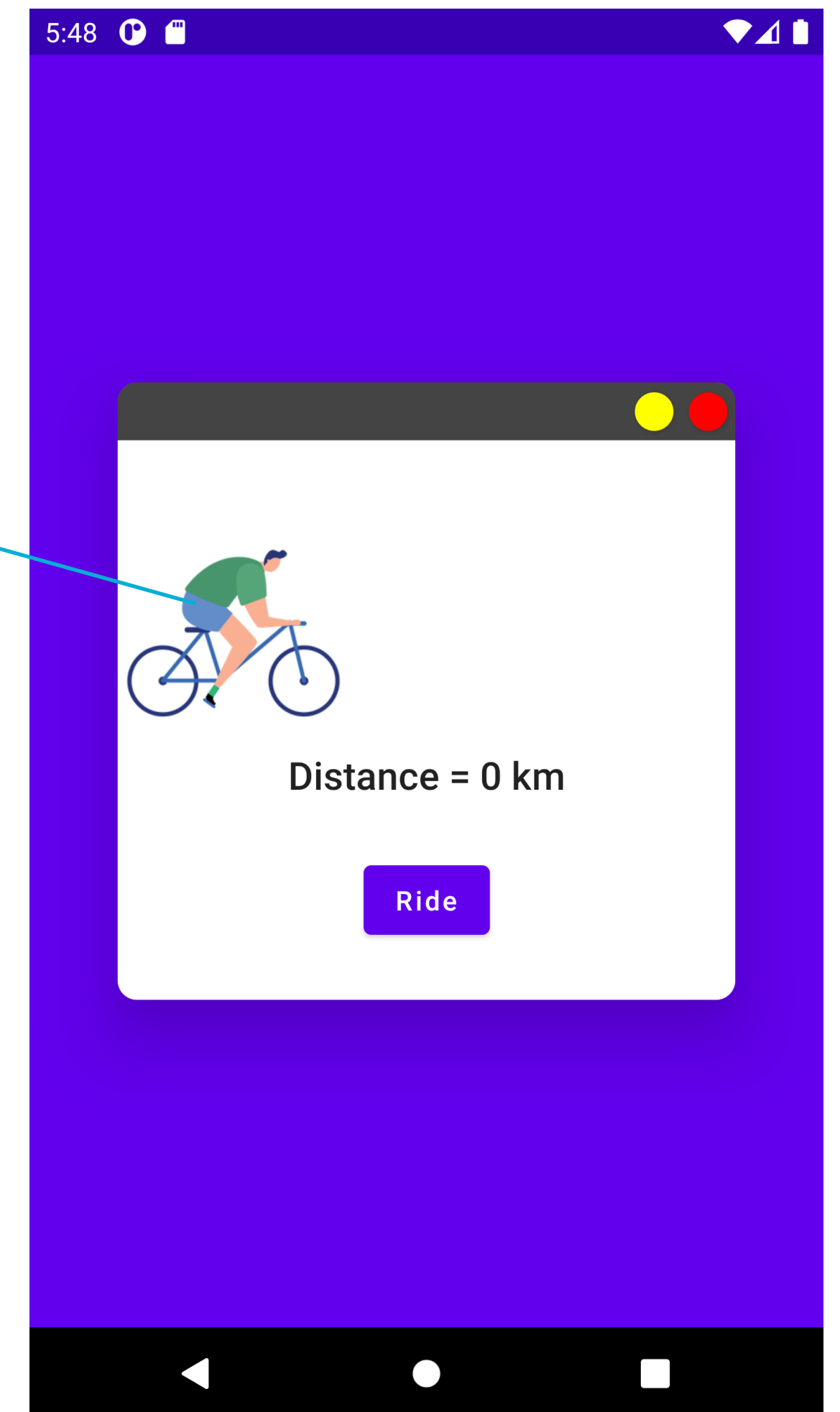
    LaunchedEffect(true) {
        snapshotFlow { currentEnabled }
            .onEach {
                Log.d("LOG_TAG", "current enable: $it")
            }
            .filter { it == true }
            .distinctUntilChanged()
            .collect { onBecomeEnabled() }
    }
}
```

snapshotFlow

예제

```
@Composable
fun Bike(offsetState: State<Int>) {
    Image(
        painter = painterResource(id = R.drawable.bike),
        modifier = Modifier
            .padding(top = 40.dp)
            .size(SIZE_BIKE.dp)
            .absoluteOffset(x = offset.value.dp),
        contentDescription = "robot",
    )

    //FIXME offset 이 끝(=200)에 도달하는 순간마다
    //FIXME 이를 서버에 보고하고 싶다
    //FIXME reportBikeOffsetToServer() 함수는 suspend 함수이다.
    //FIXME 어떡하지?
}
```



DisposableEffect

아름다운 Composable은 머문 자리도 아름답습니다

- 는 언제 실행?

- `@Composable`
`fun DisposableEffect()`가
composition에 enter할 때
- key가 바뀔 때

- 는 언제 실행?


- `@Composable`
`fun DisposableEffect()`가
composition에서 leave할 때

- key가 바뀌어서 가 재실행되기 직전

```
@Composable
fun Screen(lifecycleOwner: LifecycleOwner
           = LocalLifecycleOwner.current) {
    DisposableEffect(lifecycleOwner) {
        val observer = LifecycleEventObserver { _, event ->
            Log.d(TAG, "Screen() got an lifecycle event: $event")
        }
        lifecycleOwner.lifecycle.addObserver(observer)
    }
    onDispose {
        Log.d(TAG, "Screen() leaves composition or owner changed")
        lifecycleOwner.lifecycle.removeObserver(observer)
    }
}
```

SideEffect

composition마다 뒤따라 일을 한다

-  는 언제 실행?
- `@Composable`
`fun SideEffect()` 가 속한 Composable의
(re)composition이 완료될 때마다
그 직후

```
@Composable
fun Counter() {
    var number by remember { mutableStateOf(1) }

    println("A) ${number++}")


    SideEffect {
        println("B) ${number++}")
    }

    println("C) ${number++}")
}
```

위 코드를 수행하면 출력 결과는 무엇일까?

produceState

‘initial value’ → ‘Coroutine으로 계산된 value’를 갖는 State를 만든다

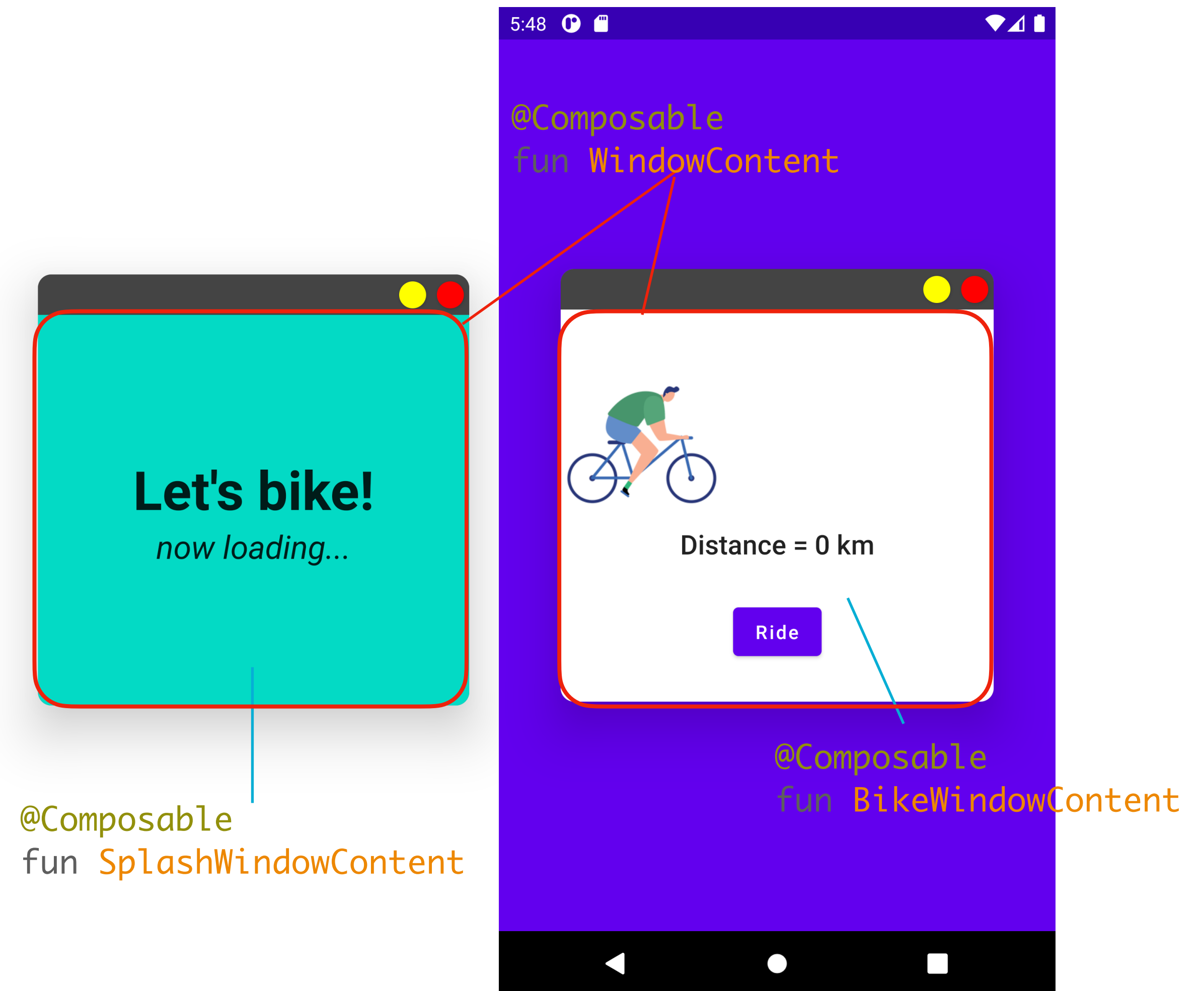
- 는 언제 실행? 언제 취소?
- LaunchedEffect와 동일
 - ※ *produceState* 내부는 *remember* & *LaunchedEffect* 로 구성됨

```
@Composable
fun getLatestMyState(seed: Int): MyState {
    val myState by produceState(
        initialValue = MyState.NONE,
        key1 = seed
    ) {
        anAsynchronousRoutine(seed)
        value = MyState.DETERMINED
    }
    return myState
}
```


produceState


예제

- *LaunchedEffect*와 동일한 예제에서 리소스 로딩과 State 변경의 책임을 *SplashWindowContent* 대신 상위 Composable인 *WindowContent*로 끌어올려보자



derivedStateOf

State(s)로부터 새로운 State를 생성한다

-  는 언제 실행?
 - State.value를 최초로 읽을 때
 - State가 의존하는 State(s).value가 바뀔때

```
@Composable
fun GooGooDan(
    number1: State<Int>,
    number2: State<Int>
) {
    val resultNumber by remember {
        derivedStateOf {
            number1.value * number2.value
        }
    }
    Text(text = "Result is $resultNumber")
}
```

derivedStateOf

예제

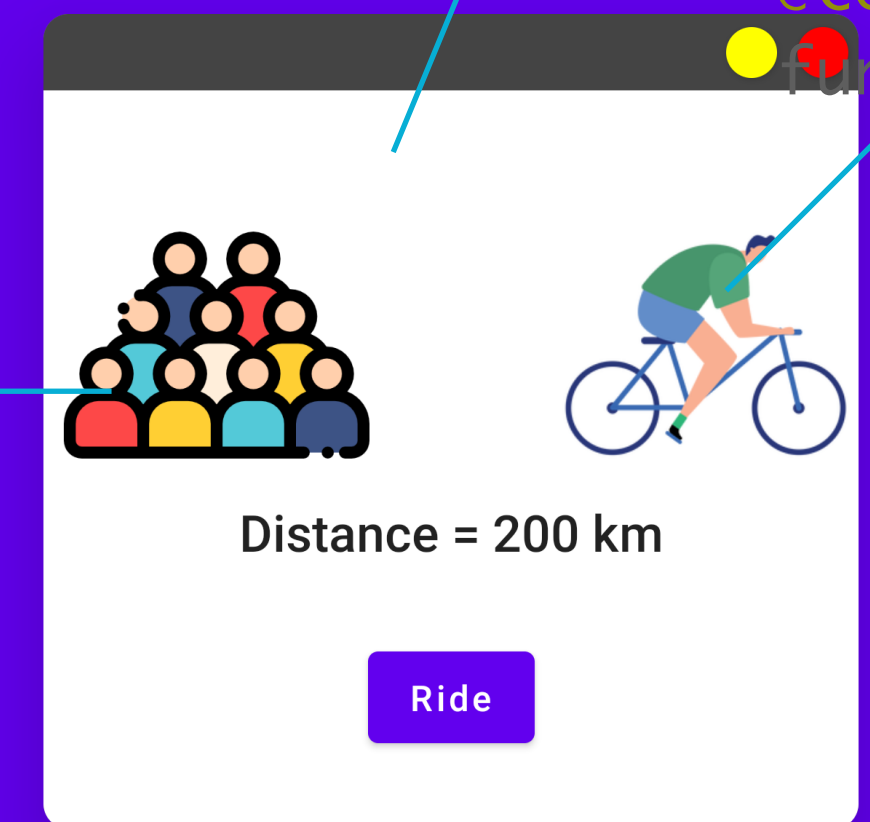
```
@Composable
fun BikeWindowContent() {
    val bikeOffsetState = animateDpAsState(..)

    //FIXME bikeOffset이 100을 넘은 순간부터
    //FIXME 아래 조건식이 참이 되도록 하여
    //FIXME Crowd()를 그리고 싶다
    if (showCrowd) {
        Crowd()
    }
    Bike(bikeOffsetState)
}
```

@Composable
fun Crowd

@Composable
fun BikeWindowContent

@Composable
fun Bike



Conclusion

- Compose에서의 side-effect는
- Compose & 코드작성자 모두에 의해
철저히 제어된 환경에서 이용되기만 한다면의도된 부(副)작용일 것이다.
- 그렇지 않으면 부정(不正)작용이 될테니 주의하자

번외.

왜 우리는 Compose를 알아야 할까?



VS.



왜 우리는 Compose를 알아야 할까?

“월레스와 그로밋은 더이상 제작되지 않기 때문”

라고 거창하게 한번 생각해봤어요..

끝

감사합니다.