WaffleStudio 2023
Rookies 21.5 Seminar

Android Seminar 5
2023.12.01 (금)

Instructor : 양주현 (@JuTaK97) / TA : 송동엽 (@eastshine2741)

오늘 배울 것

- Jetpack Compose (2)
- 앞으로의 방향성 (안드로이드를 계속 공부 하려면…)

Jetpack Compose 지난주 복습

- Composable 함수들로 UI를 구성한다.

- Composable 함수들은 State를 subscribe해서, state가 바뀌면 recomposition을 통해 UI를 다시 그린다

- 상태 hoisting을 통해 재사용 가능한 Composable 함수를 만들어야 한다

- remember

- Modifier

- LaunchedEffect, rememberCoroutineScope

Jetpack Compose (10)

Recomposition 조금 더 자세히 알아보기

- 복습

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

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

showError 상태가 바뀌면 LoginInput은 다시 그려질까?

Jetpack Compose (11)

언제 재구성을 건너뛸 수 있고, 언제 재구성을 건너뛸 수 없는가??

"컴포지션에 이미 컴포저블이 있는 경우, 모든 입력이 안정적이고 변경되지 않았으면 리컴포지션을 건너뛸 수 있다."

타입이 "Stable"하다는 것은?

- equals 결과가 동일한 두 인스턴스는 영원토록 동일해야 한다.
- public property가 변경되면 recomposition이 일어난다. (예: MutableState)
- 모든 public property도 Stable해야 한다.

예시로 이해해 보자!

Jetpack Compose (12)

우선, Primitive type, 문자열, 람다는 태생부터 변경될 수 없다. 따라서 Stable

- (리마인드) Stable이면 뭐가 달라지죠?

  이미 컴포지션에 컴포저블이 있고, 모든 값이 변경되지 않았다면 재구성을 건너뛸 수 있다

  예제: For-loop에 Composable 함수를 넣어 보기

```
@Composable
fun ForLoopPrimitive(i: Int) {
    for (ii in 0..i) {
        Composable1(i = ii)
    }
}
```

```
@Composable
private fun Composable1(i: Int) {
    SideEffect {
        Log.d("aaaa", "Composable1 $i")
    }
    Text(text = i)
}
```

Jetpack Compose (13)

일반적인 클래스가 Stable이 되려면?

- 모든 public property가 Stable이어야 한다.

- 컴파일러가 stable이라고 알아서 판단할 수 없는 타입이라도, 내가 Stable이라고 보장한다라고 컴파일러를 설득하기 : @Stable

```
@Composable
fun ForLoopClass(i: Int) {
    for (ii in 0..i) {
        Composable1(i =A(ii))
    }
}
```

- 상황 1. 클래스 A의 인스턴스끼리 절대 equal이 아니면?
- 상황 2. 클래스 A의 멤버 ii가 stable이 아니면?
- 상황 3. @Stable을 붙이면?

Jetpack Compose (14)

[상황 0]

```
class A(val aa: Int) {
    override fun equals(other: Any?): Boolean {
        return (other as? A)?.aa == this.aa
    }
}
```

- 클래스 A는 아주 전형적인 Stable 타입이다. equals가 true이면, 항상 진짜로 동일한 A 인스턴스이고, 재구성을 건너뛰는 게 이치에 맞다.

- 컴파일러도 이를 Stable로 인식하고, 재구성을 건너뛰게 된다.

Jetpack Compose (15)

[상황 1]

```
class A(val aa: Int) {
    override fun equals(other: Any?): Boolean {
        // return (other as? A)?.aa == this.aa
        return false
    }
}
```

A의 인스턴스끼리 언제나 다르기 때문에, 상태 i가 변했을 때 각 for문 루프의 컴포저블은 건너뛰지 못하고 재구성된다.

Jetpack Compose (16)

[상황 2]

```
class A(var aa: Int) {
    override fun equals(other: Any?): Boolean {
        return (other as? A)?.aa == this.aa
    }
}
```

- A의 공개 멤버 변수 aa가 var이면, A의 인스턴스끼리 equals가 true라 하여도 언제나 진짜 같은 인스턴스라고 보장할 수 없다. 즉 Stable 타입이 되기 위한 규약 위반

- 따라서 역시 재구성을 건너뛰지 못한다.

Jetpack Compose (17)

[상황 3]

```
@Stable
class A(var aa: Int) {
    override fun equals(other: Any?): Boolean {
        return (other as? A)?.aa == this.aa
    }
}
```

- “aa 값이 바뀌더라도 동일한 컴포저블이므로 재구성을 건너뛰어도 괜찮다”, 혹은 “aa 값이 var이긴 하지만 나는 안 바꿀 거다”, “aa가 var이고 나는 막 바꿀 거지만 aa 값에 따라서 UI는 안 바꿔도 된다” 라고 확신하는 상황이라면 @Stable을 붙여서, 컴파일러가 이를 Stable로 인식하게 한다.

- aa 값이 바뀌어도 Text에 다른 값이 표시되지 않는다. (재구성을 건너뛰었기 때문!)

Jetpack Compose (18)

[상황 3 심화]

```
@Stable
class A(var aa: Int) {
    override fun equals(other: Any?): Boolean {
        return true
    }
}
```

- 보통은 aa 값 변화를 UI에 반영할 것이므로, Stable을 잘못 사용하면 UI가 바뀌어야 할 때 바뀌지 않는 참사가 날 수 있다.

```
@Composable
fun ForLoopClass() {
    var i by remember {
        mutableStateOf(0)
    }
    var n by remember {
        mutableStateOf(0)
    }
    Column {
        Box(modifier = Modifier
            .background(Color.Black)
            .size(100.dp)
            .clickable {
                i++
                n++
            })
        for (ii in 0..i) {
            Composable1(a = A(ii + n))
        }
    }
}
```

Jetpack Compose (19)

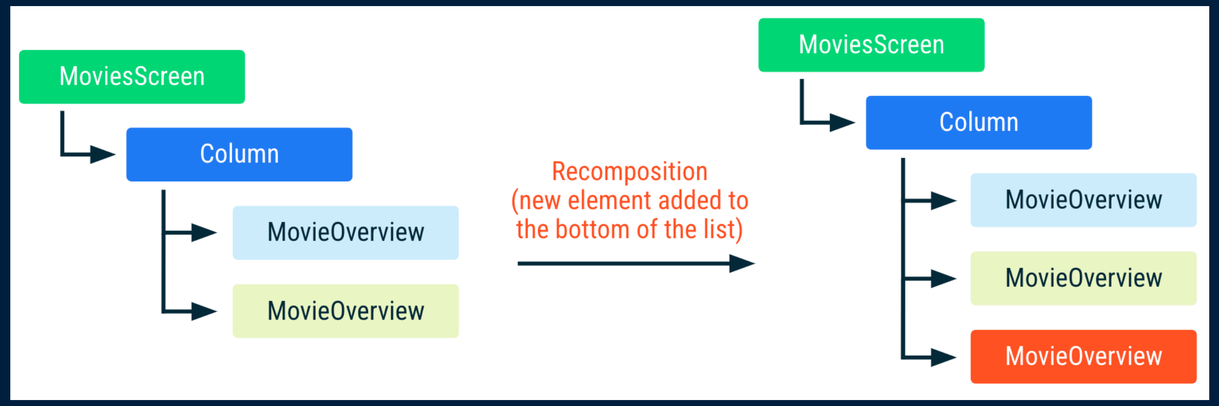
var 말고도, List 등의 공개 속성을 가지면 컴파일러가 unstable하다고 판단한다.

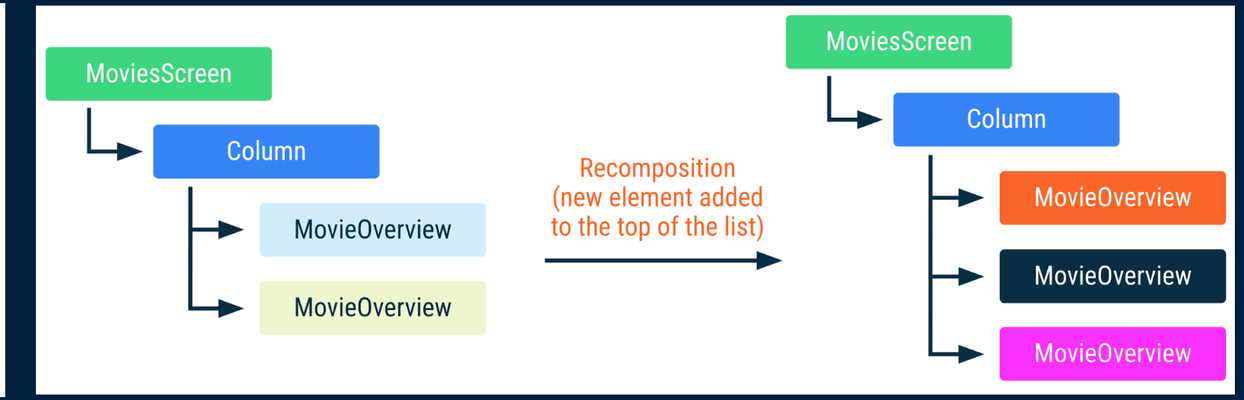
```
@Stable // 여기를 바꿔 보면?!
class NotStable(val aa: List<Int>) {
    override fun equals(other: Any?): Boolean {
        if (other !is NotStable) return false
        return (aa.size == other.aa.size) && aa.zip(other.aa).all { (p, q) -> (p == q) }
    }
}
```

Jetpack Compose (20)

recomposition과 key

- 컴포저블을 고유하게 식별할 수 있는 정보를 제공하면, 더 효율적인 스마트 재구성에 도움이 되지 않을까?





Column에서 여러 컴포저블을 표시할 때 마지막에 원소가 추가되는 경우 기존 컴포저블들에는 재구성이 일어나지 않지만, 중간에 추가되거나 shuffle되면 모두 재구성된다.

이를 해결하기 위한 개념이 key

Jetpack Compose (21)

- key 컴포저블으로 감싼다.

- key로 설정한 a가 같은 경우, 내부 블록의 컴포저블을 같은 것으로 인식하고, 재구성을 건너뛴다.

- LazyColumn은 이를 기본으로 내장하고 있다.

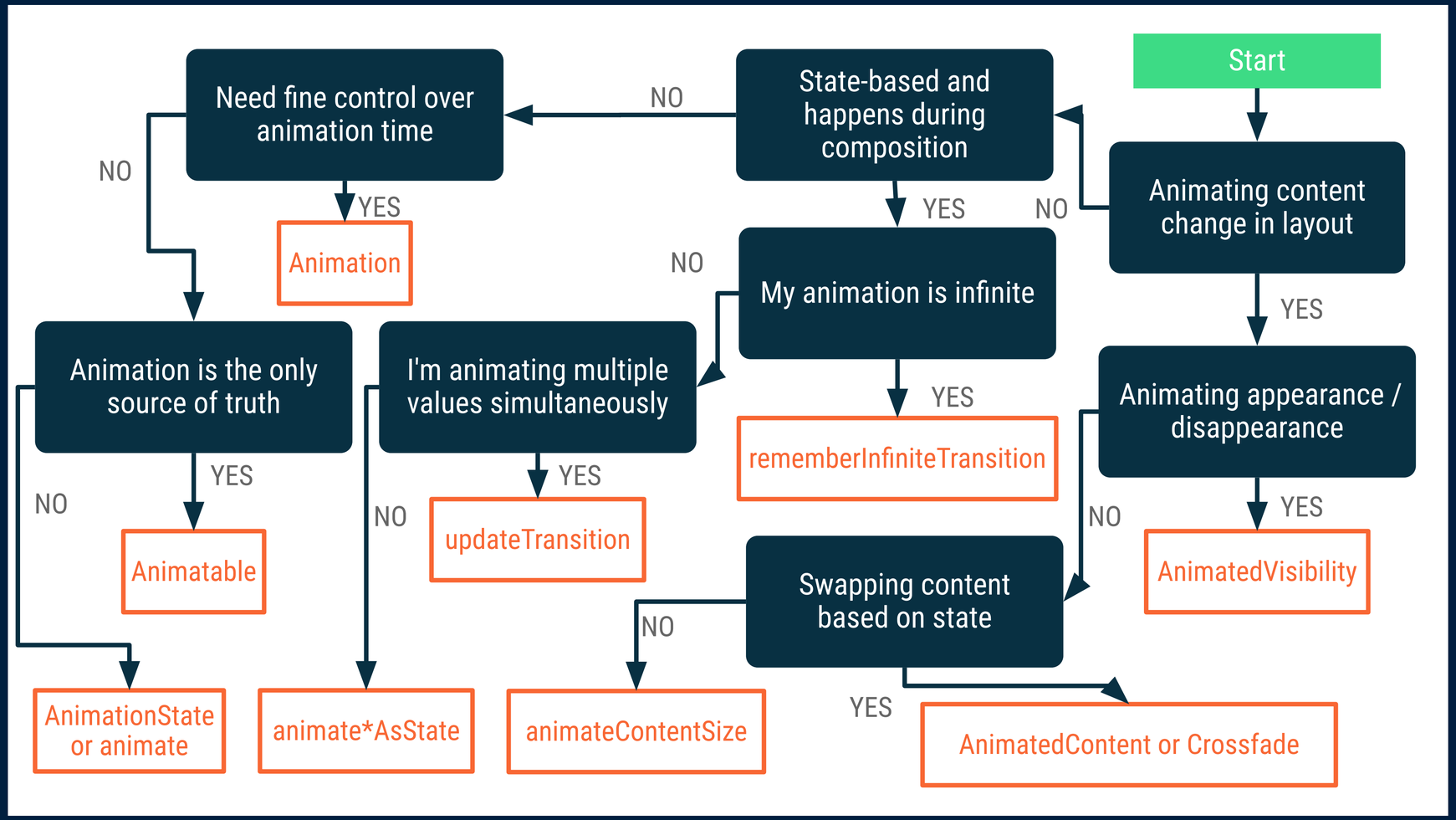
```
LazyColumn {
    items(list, key = { it.a }) {
        Composable2(item = it)
    }
}
```

```
@Composable
fun ForLoopWithKey() {
    var num by remember { mutableStateOf(1) }
    var list by remember {
        mutableStateOf(listOf<StableClass>())
    }
    Column {
        Box(modifier = Modifier
            .background(Color.Black)
            .size(100.dp)
            .clickable {
                num++
                list = list.toMutableList().apply { add(0, StableClass(num)) }.toList()
            })
        Column {
            list.forEach {
//                key(it.a) { // Key가 있고 없고가 어떻게 다를까?
                    Composable1(item = it)
//                }
            }
        }
    }
}
```

Jetpack Compose (22)

애니메이션 맛보기

- AnimatedVisibility
- Animate*AsState



이제 무엇을 공부하죠?

1. 내부 동작 공부
   - 컴포즈 내부 동작
   - 뷰 시스템 내부 동작
   - 안드로이드 플랫폼 내부 동작 등등…
2. 다양한 구현 공부
   - 우리가 일반적으로 쓰는 모든 앱에 존재하지만, 아직 구현해 보지 못한 것들

     웹뷰, 보안/암호화, 딥 링크, 푸시 알림, 미디어 재생, 카메라, 생체 인증, 위젯, 파일 접근, 블루투스, NFC, …
   - 다양한 UI들 구현해보기

     다중 스크롤, 바텀시트, 다양한 애니메이션, …

이제 무엇을 공부하죠?

3. 하위 기술 스택 공부

- 리액티브 프로그래밍 : RxJava, Kotlin Flow
- 비동기 프로그래밍 : Kotlin Coroutine, Java Thread
- 실시간 통신 : WebRTC, WebSocket

4. 아키텍처, 설계 공부

- 클린 아키텍처, 클린 코드 (이론)
- 테스트 코드
- 멀티 모듈 프로젝트 (feat. Gradle)

이제 무엇을 공부하죠?

5. 빌드, 배포, CI/CD

- Github Action으로 CI/CD 구축하기
- Gradle script로 다양한 빌드 설정하기
- Firebase 연동, Google Play Store에 앱 업로드해 보기

6. 크로스 플랫폼

- Flutter, React Native, Kotlin Multiplatform 등등

수고하셨습니다!

이제, 배운 지식을 활용해서 멋진 토이프로젝트를 완성하세요