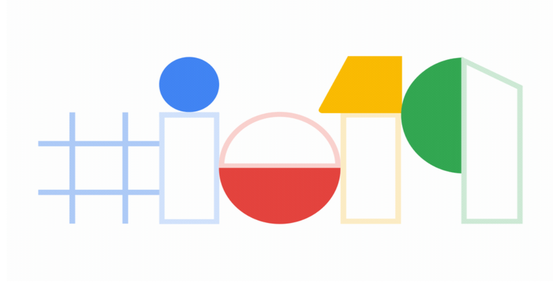


Extended Seoul

New Android Declarative UI patterns **Compose**

이승민

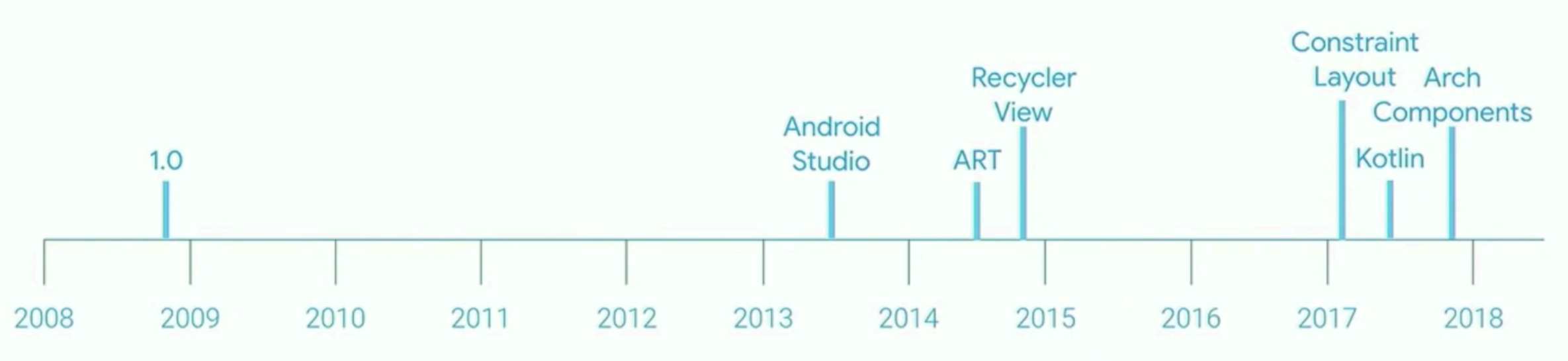
뱅크샐러드 안드로이드 개발자
GDE Android Korea

목차

1. 안드로이드 UI의 문제점
2. 구글이 원하는 새로운 UI
3. What is declarative?
4. Jetpack Compose
5. 정리

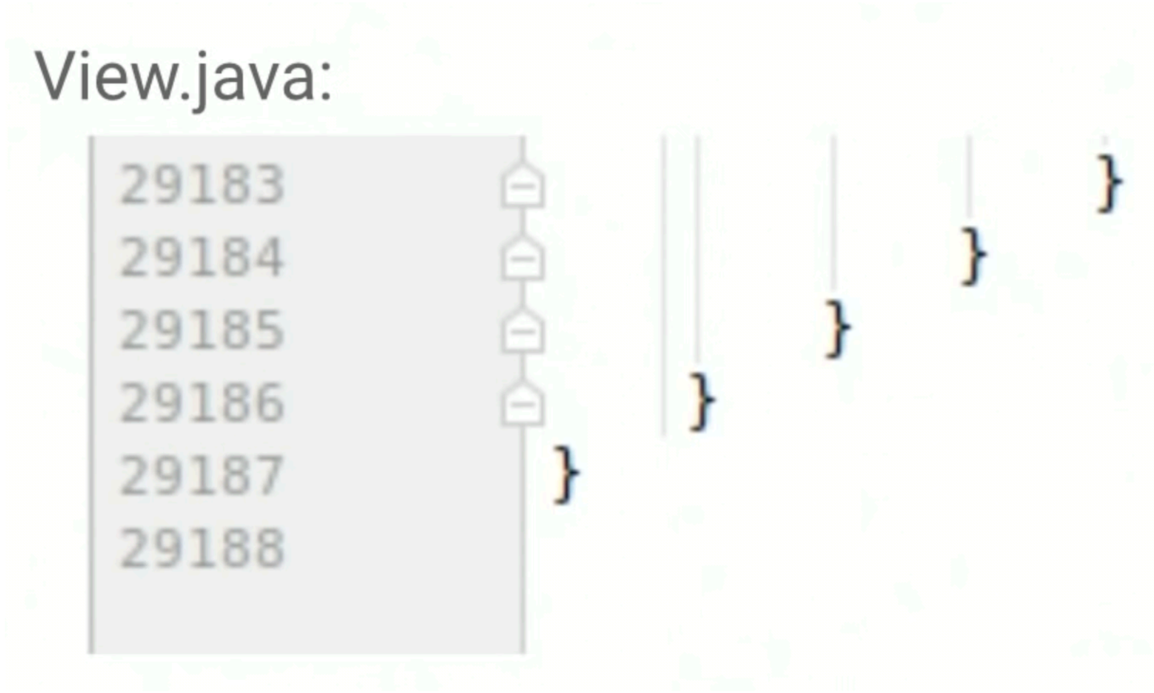
안드로이드 UI의 문제점

API Regrets



API Regrets

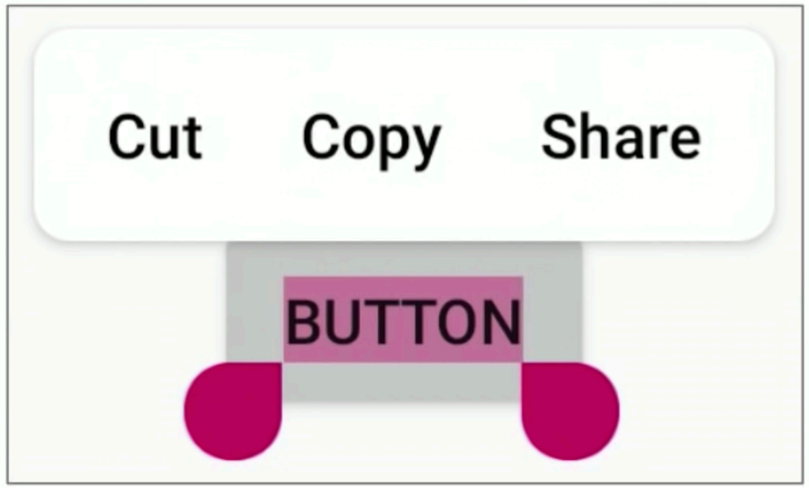
View.java:



약 30,000줄의 View 코드

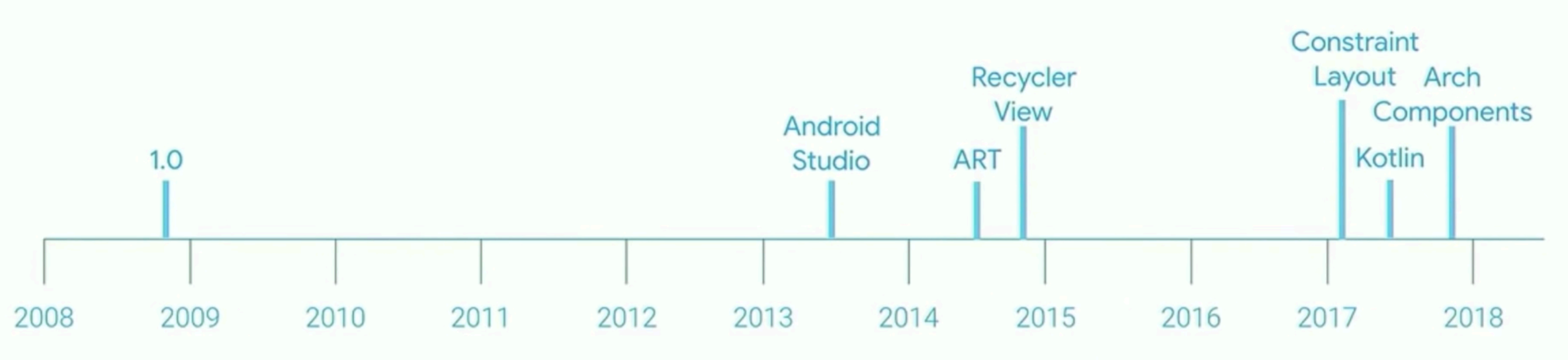
API Regrets

```
public class Button extends TextView {
```



속성이 어색한 상속

API Regrets - **Bundled**



SDK와 함께 UI가 변해옴

UI를 업데이트 하려면 SDK를 바꿔야 한다

Extended Seoul

복잡한 View 구조

Fragment

CustomView

복잡한 View 구조

```
class SquareImageView : AppCompatImageView {
    constructor(context: Context?) : super(context)
    constructor(context: Context?, attrs: AttributeSet?) : super(context, attrs)
    constructor(context: Context?, attrs: AttributeSet?, defStyleAttr: Int) :
super(context, attrs, defStyleAttr)

    override fun onMeasure(widthMeasureSpec: Int, heightMeasureSpec: Int) {
        super.onMeasure(widthMeasureSpec, widthMeasureSpec)
    }
}
```

알아야 할 부모의 것들이 많음

Too much code

Fragment

CustomView

Too much code

Fragment

CustomView

layout.xml

attrs.xml

styles.xml

SomeCustomView.kt

Multiple data flow

MVC

MVP

MVVM

MVI

Multiple data flow

MVC

MVP

MVVM

MVI

View State 분리하기 위한 노력

Multiple data flow

data flow가 View에도 있기 때문에

Architecture

What is data flow?

What is the source of truth - State

Who owns it

Who updates it

What is data flow?

What is the source of truth - State

Who owns it

Who updates it

3개 모두 View에도 존재

What is data flow?

Spinner example

onSelectedItemChanged 사용자가 값을 바꾸면 notify

그러나 값이 바뀐 '뒤에' notify

data flow가 분리

구글이 원하는 새로운 UI

Unbundled

from platform releases

SDK와 UI를 분리

UI 업데이트 마구마구!!!

간단한 View 구조

NO 상속 NO 보일러플레이트

작성한대로 출력되는 직관적인 UI 코드

Write less code

layout, attr, style

Write less code

layout, attr, style

하나의 UI Code

Single data flow

State는 하나의 Owner만 갖는다

하나의 Owner만 State를 변경한다

Owner가 이밴트를 감지하고 변경을 주도한다

=> View withOUT State

대안은?

Jetpack Compose

What is Jetpack Compose?

NEW Jetpack UI Widgets - NOT in SDK!

Declaritive UI Toolkits for Android

Kotlin Compile Plugin - Kotlin First!

기존 앱과 fully 호환

SUPER Experimental

What is declarative?

Declarative Programming

선언형 프로그래밍

함수형 프로그래밍

목표를 명시

SQL, HTML, 스칼라

Imperative Programming

명령형 프로그래밍

절차형, 객체지향 프로그래밍

알고리즘, State를 명시

Java, C, C++

명령형 sum

```
fun sumImperative(arr: List<Int>): Int {
    var result = 0
    arr.forEach {
        result += it
    }
    return result
}
```

선언형 sum

```
fun sumFunctional(arr: List<Int>): Int =
    arr.reduce { acc, i -> acc + arr[i] }
```

명령형 sum

```
fun sumImperative(arr: List<Int>): Int {
    var result = 0
    arr.forEach {
        result += it
    }
    return result
}
```

선언형 sum

```
fun sumFunctional(arr: List<Int>): Int =
    arr.reduce { acc, i -> acc + arr[i] }
```

Functional Programming

데이터(목표) 반환의 연속흐름

(함수) (스트림)

Extended Seoul

Imperative Android UI

XML에 모든 State를 명시

Kotlin에 모든 View 변경을 명시

Declarative Android UI

View를 반환하는 함수의 연속

Anko, Flutter

COMPOSE

Jetpack Compose

Basic Idea

- UI as a **function** (Declaritive)
- View 계층을 반환하는 함수

```
@Composable
fun Greeting(name: String) {
    Text("Hello $name")
}
```

Basic Idea

- UI as a **function** (Declaritive)
- View 계층을 반환하는 함수

```
@Composable
fun Greeting(name: String) {
    Text("Hello $name")
}
```

Initialize

```
class MainActivity : AppCompatActivity() {
    public override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Greeting(“World!")
            ...
        }
    }
}
```

View

```
@Composable
fun Greeting(name: String) {
    Text("Hello $name")
}
```

ListView

```
@Composable
fun Greeting(name: String) {
    Text("Hello $name")
}


@Composable
fun GreetingList(names: List<String>) {
    for (name in names) {
        Text("Hello $name")
    }
}
```

기존 List Adapter

```
class SuggestionsAdapter(
    private val inflater: LayoutInflater
) : ListAdapter<Suggestion, SuggestionsAdapter.ViewHolder>(SuggestionDiffCallback) {
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
        return ViewHolder(SuggestionItemBinding.inflate(inflater, parent, false))
    }
    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        val item = getItem(position)
        with(holder.binding) {
            text.text = item.name
            image.setImageBitmap(item.avatar)
        }
    }
    inner class ViewHolder(
        val binding: SuggestionItemBinding
    ) : RecyclerView.ViewHolder(binding.root)
}
```

Composable in Composable

```
@Composable
fun Greeting(name: String) {
    Text("Hello $name")
}
```

Composable in Composable

```
@Composable
fun Greeting(name: String) {
    Text("Hello $name")
}
```

```
/**
 * Simplified version of [Text] component with minimal set of customizations.
 */
@Composable
fun Text(
    text: String,
    style: TextStyle? = null,
    paragraphStyle: ParagraphStyle? = null,
    softWrap: Boolean = DefaultSoftWrap,
    overflow: TextOverflow = DefaultOverflow,
    textScaleFactor: Float = 1.0f,
    maxLines: Int? = DefaultMaxLines,
    selectionColor: Color = DefaultSelectionColor
)
```

Composable in Composable

```
@Composable
fun NewsFeed(stories: List<StoryData>) {
    for (story in stories)
        StoryWidget(story)
}

@Composable
fun StoryWidget(story: StoryData) {
    Padding(8.dp) {
        Column {
            Title(story.title)
            Image(story.image)
            Text(story.content)
        }
    }
}
```

Composable in Composable

```
@Composable
fun NewsFeed(stories: List<StoryData>) {
    for (story in stories)
        StoryWidget(story)
}

@Composable
fun StoryWidget(story: StoryData) {
    Padding(8.dp) {
        Column {
            Title(story.title)
            Image(story.image)
            Text(story.content)
        }
    }
}
```

Composable parameter

```
@Composable
fun NewsFeed(stories: List<StoryData>) {
    ScrollingList(stories) {
        StoryWidget(it)
    }
}

@Composable
fun <T> ScrollingList(
    dataList: List<T>,
    body: @Composable() (T) -> Unit
) {
    for (data in dataList)
        body(data)
}
```

Composable parameter

```
@Composable
fun NewsFeed(stories: List<StoryData>) {
    ScrollingList(stories) {
        StoryWidget(it)
    }
}

@Composable
fun <T> ScrollingList(
    dataList: List<T>,
    body: @Composable() (T) -> Unit
) {
    for (data in dataList)
        body(data)
}
```

Data Observe

```
@Composable
fun NewsFeed(stories: LiveData<List<StoryData>>) {
    stories.observe(owner) {
        ScrollingList(it) {
            StoryWidget(it)
        }
    }
}

data class StoryData(
    val title: LiveData<String>,
    val image: LiveData<String>,
    val content: LiveData<String>
)
```

Data Observe

```
@Composable
fun NewsFeed(stories: LiveData<List<StoryData>>) {
    stories.observe(owner) {
        ScrollingList(it) {
            StoryWidget(it)
        }
    }
}

data class StoryData(
    val title: LiveData<String>,
    val image: LiveData<String>,
    val content: LiveData<String>
)
```

Data Observe

```
@Composable
fun NewsFeed(stories: LiveData<List<StoryData>>) {
    stories.observe(owner) {
        ScrollingList(it) {
            StoryWidget(it)
        }
    }
}

data class StoryData(
    val title: LiveData<String>,
    val image: LiveData<String>,
    val content: LiveData<String>
)
```

Data Observe

```
@Composable
fun NewsFeed(stories: LiveData<List<StoryData>>) {
    stories.observe(owner) {
        ScrollingList(it) {
            StoryWidget(it)
        }
    }
}

data class StoryData(
    val title: LiveData<String>,
    val image: LiveData<String>,
    val content: LiveData<String>
)
```

Data Observe

```
@Model
data class StoryData(
    val title: String,
    val image: String,
    val content: String
)
```

Data Observe

```
@Model
data class StoryData(
    val title: String,
    val image: String,
    val content: String
)
```

Event

```
@Composable
fun NewsFeed(stories: List<StoryData>) {
    ScrollingList(stories) {
        StoryWidget(it) { onSelected(it) }
    }
}

@Composable
fun StoryWidget(story: StoryData, onClick: () -> Unit) {
    Clickable(onClick) {
        Padding(8.dp) {
            ...
        }
    }
}
```

Event

```
@Composable
fun NewsFeed(stories: List<StoryData>) {
    ScrollingList(stories) {
        StoryWidget(it) { onSelected(it) }
    }
}

@Composable
fun StoryWidget(story: StoryData, onClick: () -> Unit) {
    Clickable(onClick) {
        Padding(8.dp) {
            ...
        }
    }
}
```

Top-down data flow

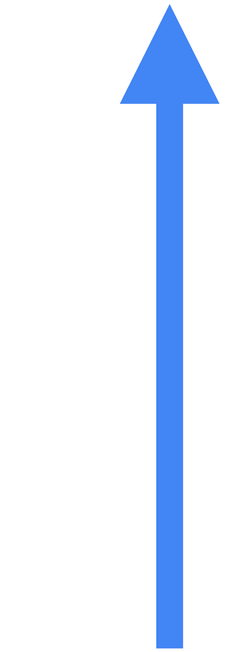
```
@Composable
fun NewsFeed(stories: List<StoryData>) {
    ScrollingList(stories) {
        StoryWidget(it) { onSelected(it) }
    }
}
@Composable
fun StoryWidget(story: StoryData) {
    Clickable(onClick) {
        Padding(8.dp) { ... }
    }
}
```

Data

Extended Seoul

Bottom-up event

```
@Composable
fun NewsFeed(stories: List<StoryData>) {
    ScrollingList(stories) {
        StoryWidget(it) { onSelected(it) }
    }
}
@Composable
fun StoryWidget(story: StoryData) {
    Clickable(onClick) {
        Padding(8.dp) { ... }
    }
}
```



Event

Extended Seoul

Single data flow

- 화면에서 관리하는 데이터를 뷰로 내린다
- 뷰에서 발생하는 이벤트를 화면에서 넘겨준 람다로 올린다

Single data flow

- 화면에서 관리하는 데이터를 뷰로 내린다
- 뷰에서 발생하는 이벤트를 화면에서 넘겨준 람다로 올린다

Data flow를 화면으로 통일!

(MVP에서는 Presenter / MVVM에서는 ViewModel)

Multiple data flow

Spinner example

onSelectedItemChanged 사용자가 값을 바꾸면 notify

그러나 값이 바뀐 '뒤에' notify

data flow가 분리

Single data flow Spinner

```
@Composable
fun FoodPreferences(data: UserPreferences) {
    val options = listOf(MILK, EGGS, BREAD)
    Spinner(
        options = options,
        selected = data.favoriteFood,
        onChanged = { selected -> data.favoriteFood = selected }
    )
}
```

Single data flow Spinner

```
@Composable
fun FoodPreferences(data: UserPreferences) {
    val options = listOf(MILK, EGGS, BREAD)
    Spinner(
        options = options,
        selected = data.favoriteFood,
        onChanged = { selected -> data.favoriteFood = selected }
    )
}
```

Notify 후 Single data source를 변경

정리

Extended

Unbundled

from platform releases

SDK와 UI를 분리

UI 업데이트 마구마구!!!

Single data flow

State는 하나의 Owner만 갖는다

하나의 Owner만 State를 변경한다

Owner가 이밴트를 감지하고 변경을 주도한다

=> View withOUT State

Basic Idea

- UI as a **function** (Declaritive)
- View 계층을 반환하는 함수

```
@Composable
fun Greeting(name: String) {
    Text("Hello $name")
}
```

Compose 사전작업

Kotlin

데이터를 View로 내려주세요 (Top-down data flow)

- DataBinding, LiveData, RxJava, etc...

마이그레이션

```
@Composable
@GenerateView
fun Greeting(name: String) { /* … */ }

<GreetingView android:id="@+id/greeting"
    app:name="@string/greeting_name" />

val greeting: GreetingView = findViewById(R.id.greeting)
greeting.setName("Jim")
```

Compose 사용하기

SUPER Experimental!

직접 빌드 해야함

Compose 사용하기

@Pluulove

repo

1. Installing Repo

먼저 Android Studio를 빌드하기 위해 build tool인 repo를 설치한다

https://source.android.com/setup/build/downloading#installing-repo

1. Download Repo tool

```
$ curl https://storage.googleapis.com/git-repo-downloads/repo > ~/bin/repo
$ chmod a+x ~/bin/repo
```

AndroidX

1. 소스 코드 체크아웃

```
repo init -u https://android.googlesource.com/platform/manifest -b androidx-master-dev
```

Extended Seoul

추천 공부

Flutter

Redux / React

[MVI]

GDG Seoul

DevFest

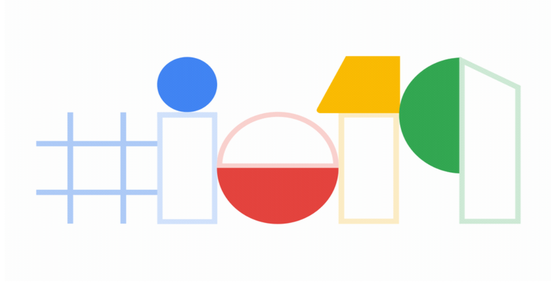
Data Uni-Directional
Architecture in Android

이승민, GDE Android Korea
@maryangmin

Extended
Seoul

Reference

- Google I/O 2019 영상
- 공식 가이드
- @tura님 블로그
- 정승욱님 발표자료
- @pluulove님 블로그



Extended Seoul

감사합니다!

이승민

뱅크샐러드 안드로이드 개발자
GDE Android Korea