Introduction to Jetpack Compose

Declarative UI toolkit for Android

About me

3 yrs.

나윤호

velog.io/@tura

@turastory

Riiid!

19. 04~

13. 05 17. 04 17. 05 19. 05 19. 06

Declarative

React makes it painless to create interactive UIs. Design simple views for each state in your application, and React will efficiently update and render just the right components when your data changes.

Litho circleci passing bintray v0.26.1 chat on gitter

Litho is a declarative framework for building efficient UIs on Android.

- **Declarative:** Litho uses a declarative API to define UI componen layout for your UI based on a set of immutable inputs and the fra rest.
- **Asynchronous layout:** Litho can measure and layout your UI ahe the UI thread.

Introduction to declarative UI

This introduction describes the conceptual difference between the declaractive style used by Flutter, and the imperative style used by many other UI frameworks.

Declarative Syntax

SwiftUI uses a declarative syntax so you can simply state what your user interfac should do. For example, you can write that you want a list of items consisting of t fields, then describe alignment, font, and color for each field. Your code is simple and easier to read than ever before, saving you time and maintenance.

```
import SwiftUI

struct Content : View {

    @State var model = Themes.listModel

    var body: some View {
        List(model.items, action: model.selectItem) { item in
```

Jetpack Compose

A declarative toolkit for building UI

Jetpack Compose is an unbundled toolki programming model with the concisenes

#extended

Declarative

13. 05 | 17. 04 | 17. 05 | 19. 05 | 19. 06





Declarative

React makes it painless to create inter ctive U
Design simple views for each state in ur
application, and React will efficiently update and
render just the right components when your data
changes.

Litho

circleci passing | bintray v0.26.1 | chat on gitter

Litho is a declarative framework for building efficient UIs on Android.

- **Declarative:** Litho uses a declarative API to define UI componen
layout for your UI based on a set of immutable inputs and the fra
rest.
- **Asynchronous layout:** Litho can measure and layout your UI ahe
the UI thread.

Introduc on to declarat ve UI

This c ddu d describe the ep diff nce be en t ecla sty used b er, an im tive style used by many other UI frameworks.

Declarative Syntax

SwiftUI uses a declarative syntax so you can simply state what your user interfac
should do. For example, you can write that you want a list of items consisting of t
fields, then describe alignment, font, and color for each field. Your code is simple
and easier to read than ever before, saving you time and maintenance.

```
import SwiftUI

struct Content : View {

    @State var model = Themes.listModel

    var body: some View {
        List(model.items, action: model.selectItem) { item in
```

Jetpack Compose

A declarative toolkit for building UI

Jetpack Compose is an unbundled toolki
programming model with the concisenes

extended

Declarative UI

What is Declarative UI Programming?

Imperative

```
with (binding) {
    background.setBackgroundColor(Color.RED)
    icon.setImageDrawable(completedIcon)
    container.removeAllViews()
    container.addView(createCompletedView())
}
```

Specify HOW
Eager Evaluation

Imperative

```
with (binding) {
    background.setBackgroundColor(Color.RED)
    icon.setImageDrawable(completedIcon)
    container.removeAllViews()
    container.addView(createCompletedView())
}
```

Specify HOW
Eager Evaluation

Declarative

```
val state = State(
    backgroundColor = Color.RED,
    icon = completedIcon,
    children = [
        createCompletedView()
    ]
)

view.render(state)
```

Specify WHAT
Lazy Evaluation

Why Declarative?

More and more complex UI

Animations, Transitions

Focus on What to do, rather than how

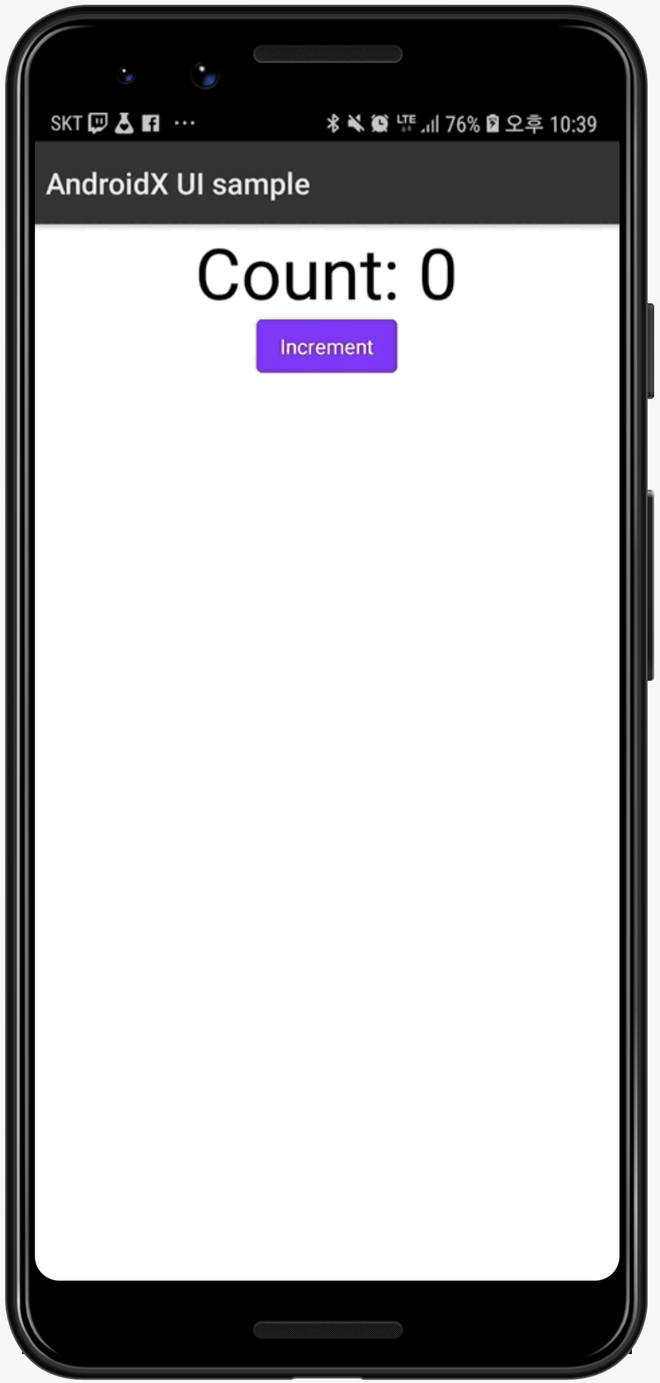
A Quick Look

Say hello to Jetpack Compose

#io extended

Counter App

Click -> +1



Usual Way

```
val binding = DataBindingUtil
    .setContentView<ActivityCounterBinding>(
        this,
        R.layout.activity_counter
    )

var count = 0

binding.incrementButton.setOnClickListener {
    binding.counterText.text =
        getString(R.string.counter, ++count)
}
```

Rx Way

```
val binding = DataBindingUtil
    .setContentView<ActivityCounterBinding>(
        this,
        R.layout.activity_counter
    )

var count = 0

binding.incrementButton
    .clicks()
    .throttleFirst(300, TimeUnit.MILLISECONDS)
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe({
        binding.counterText.text =
            getString(R.string.counter, count)
    }, Throwable::printStackTrace)
    .disposedBy()
```

Compose Way

```
setContent {
    Center {
        Column {
            val count = +state { 0 }

            Text(
                text = "Count: ${count.value}",
                style = +themeTextStyle { this.h3 }
            )
            Button(
                text = "Increment",
                onClick = { count.value += 1 }
            )
        }
    }
}
```

Backgrounds

What brings us to Jetpack Compose?

Problems

1. Tightly coupled with framework
2. Distributed codes
3. Inconsistent data flow

Jetpack Compose

Let’s dive into Compose!

#extended

1. UI as a function

1. UI as a function

Integer →

```
fun square(x: Int) =
    x * x
```

→ Integer

1. UI as a function

String →

```
fun Greeting(name: String) =
    println("Hello, $name")
```

→ stdout

1. UI as a function

Data →

```
fun Greeting(name: String) =
    Text("Hello, $name")
```

→ UI

1. UI as a function

Key Concepts

Mark as a composable widget

```
@Composable
fun Greeting(name: String) =
    Text("Hello, $name")
```

1. UI as a function

Key Concepts

```
@Composable
fun Greeting(name: String) =
    Text("Hello, $name")
```

UI Hierarchy

1. UI as a function

Kotlin Super Power

```
@Composable
fun Loading() = CircularProgressIndicator()

@Composable
fun FriendWidget(friend: Friend) {
    val image = asyncLoad(placeHolder) { loadImage(friend.url) }
    Image(image)
    Text(
        text = friend.name,
        style = +themeTextStyle { h4 }
    )
}

@Composable
fun friendsList(state: FriendsListState) {
    val friends = state.friends
    if (friends.isEmpty()) {
        Loading()
    } else {
        friends.forEach(this::FriendWidget)
    }
}
```

1. UI as a function

Kotlin Super Power

```
@Composable
fun Loading() = CircularProgressIndicator()

@Composable
fun FriendWidget(friend: Friend) {
    val image = asyncLoad(placeHolder) { loadImage(friend.url) }
    Image(image)
    Text(
        text = friend.name,
        style = +themeTextStyle { h4 }
    )
}

@Composable
fun friendsList(state: FriendsListState) {
    val friends = state.friends
    if (friends.isEmpty()) {
        Loading()
    } else {
        friends.forEach(this::FriendWidget)
    }
}
```

Child Component
Composables are composable

1. UI as a function

Kotlin Super Power

```
@Composable
fun Loading() = CircularProgressIndicator()

@Composable
fun FriendWidget(friend: Friend) {
    val image = asyncLoad(placeHolder) { loadImage(friend.url) }
    Image(image)
    Text(
        text = friend.name,
        style = +themeTextStyle { h4 }
    )
}

@Composable
fun friendsList(state: FriendsListState) {
    val friends = state.friends
    if (friends.isEmpty()) {
        Loading()
    } else {
        friends.forEach(this::FriendWidget)
    }
}
```

Conditions / Loop

1. UI as a function

Kotlin Super Power

```
@Composable
fun Loading() = CircularProgressIndicator()

@Composable
fun FriendWidget(friend: Friend) {
    val image = asyncLoad(placeHolder) { loadImage(friend.url) }
    Image(image)
    Text(
        text = friend.name,
        style = +themeTextStyle { h4 }
    )
}

@Composable
fun friendsList(state: FriendsListState) {
    val friends = state.friends
    if (friends.isEmpty()) {
        Loading()
    } else {
        friends.forEach(this::FriendWidget)
    }
}
```

Coroutines (Async Loading)

1. UI as a function

Update UI

```
@Model
data class Friend(
    var name: String
)


@Composable
fun App() {
    val friend by +state { Friend("tura") }
    FriendWidget(friend)
    Button(
        text = "Click me!",
        onClick = {
            friend.name = randomName()
        }
    )
}
```

1. UI as a function

Update UI

```
@Model
data class Friend(
    var name: String
)
```

Use @Model to make class observable

```
@Composable
fun App() {
    val friend by +state { Friend("tura") }
    FriendWidget(friend)
    Button(
        text = "Click me!",
        onClick = {
            friend.name = randomName()
        }
    )
}
```

1. UI as a function

Update UI

```
@Model
data class Friend(
    var name: String
)


@Composable
fun App() {
    val friend by +state { Friend("tura") }
    FriendWidget(friend)
    Button(
        text = "Click me!",
        onClick = {
            friend.name = randomName()
        }
    )
}
```

Define local state

1. UI as a function

Update UI

```
@Model
data class Friend(
    var name: String
)


@Composable
fun App() {
    val friend by +state { Friend("tura") }
    FriendWidget(friend)
    Button(
        text = "Click me!",
        onClick = {
            friend.name = randomName()
        }
    )
}
```

Changing value causes Recomposition

1. UI as a function


```
@Composable
fun PhoneInputWidget(data: InputData) {
    EditableText(
        text = data.phoneNumber,
        onChange = { newValue →
            newValue
                .let(this::filterInvalidPhoneCharacters)
                .let(this::formatPhone)
                .let {
                    data.phoneNumber = it
                }
        }
    )
}
```

1. UI as a function


```
@Composable
fun PhoneInputWidget(data: InputData) {
    EditableText(
        text = data.phoneNumber,
        onChange = { newValue →
            newValue
                .let(this::filterInvalidPhoneCharacters)
                .let(this::formatPhone)
                .let {
                    data.phoneNumber = it
                }
        }
    )
}
```

No internal state
Always receive data from outer world

1. UI as a function


```
@Composable
fun PhoneInputWidget(data: InputData) {
    EditableText(
        text = data.phoneNumber,
        onChange = { newValue →
            newValue
                .let(this::filterInvalidPhoneCharacters)
                .let(this::formatPhone)
                .let {
                    data.phoneNumber = it
                }
        }
    )
}
```

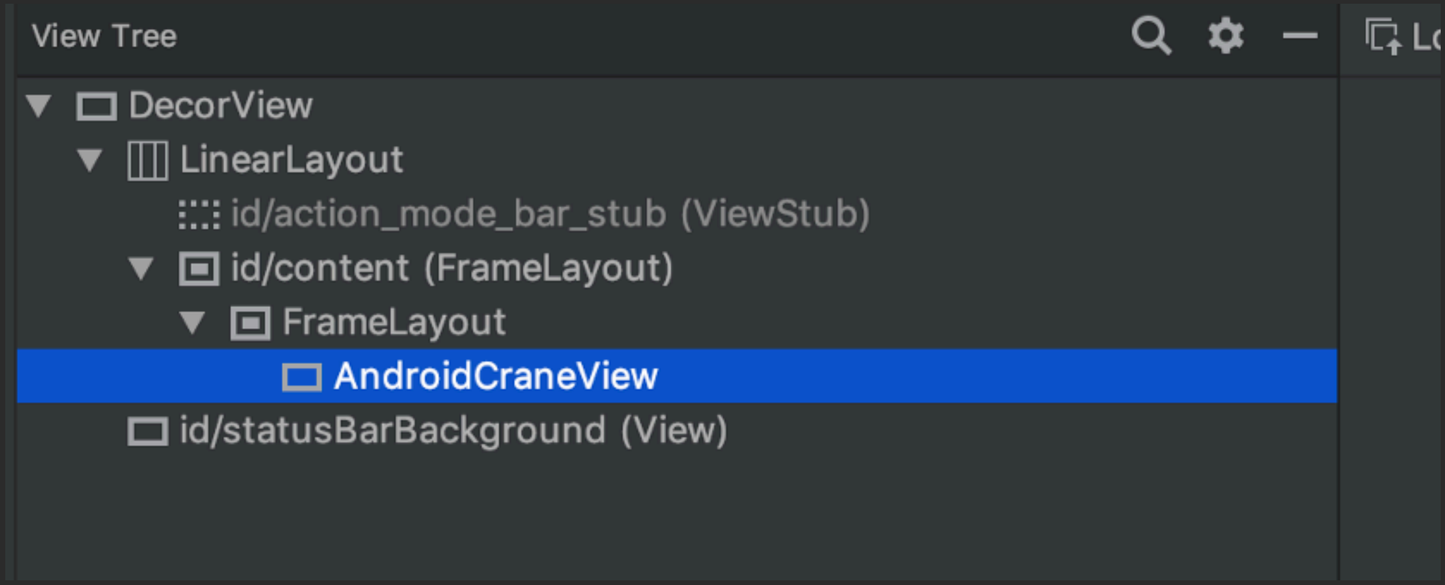
No internal state
Always receive data from outer world

... And make validation BEFORE update UI

2. No more view

2. No more view

Directly draw onto Canvas



Screenshot from Thijs Suijten - Diving into Jetpack Compose

2. No more view

Attributes are just normal classes

```
@Composable
fun Text(
    text: String,
    style: TextStyle? = null,
    textAlign: TextAlign = DefaultTextAlign,
```

```
data class TextStyle(
    val color: Color? = null,
    val fontSize: Float? = null,
    val fontSizeScale: Float? = null,
    val fontWeight: FontWeight? = null,
    val fontStyle: FontStyle? = null,
    val fontSynthesis: FontSynthesis? = null,
    val fontFeatureSettings: String? = null,
    val letterSpacing: Float? = null,
    val wordSpacing: Float? = null,
```

3. Exploring APIs

3. Exploring APIs

Annotations

@Composable

Mark function as a Composable widget

@Model

Mark class as an Observable Data for Composable Widgets

3. Exploring APIs

Effect

Codes that should be executed during executing composition

Composition Phase

Execution Phase

```
@Composable
fun CounterWidget() {
    Log.d("Counter", "CounterWidget start")
    val count = +state { 0 }
    Text(text = "Count: ${count.value}")
    Button(onClick = { count.value++ })
}
```

3. Exploring APIs

Effect

Codes that should be executed during executing composition

Composition Phase	Execution Phase

```
@Composable
fun CounterWidget() {
    Log.d("Counter", "CounterWidget start")
    val count = +state { 0 }
    Text(text = "Count: ${count.value}")
    Button(onClick = { count.value++ })
}
```

3. Exploring APIs

Effect

Codes that should be executed during executing composition

Composition Phase

Execution Phase

```
@Composable
fun CounterWidget() {
    Log.d("Counter", "CounterWidget start")
    val count = +state { 0 }
    Text(text = "Count: ${count.value}")
    Button(onClick = { count.value++ })
}
```

Use +state to define local state

3. Exploring APIs

Effects

memo

Used when some value is needed

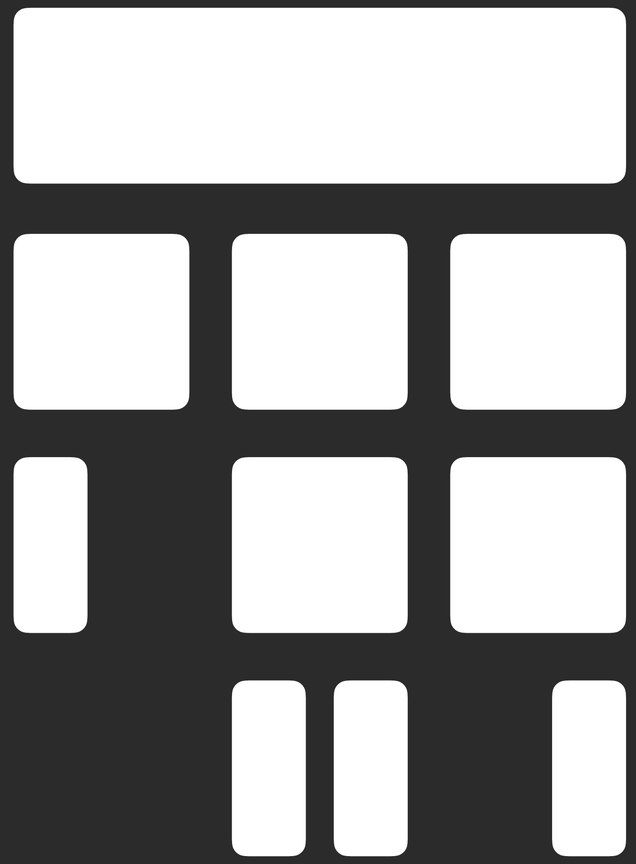
state

```
memo { State(init()) }
```

Used when local mutable state is needed

ambient

Used when data flow is too complex



3. Exploring APIs

Effects

memo

Used when some value is needed

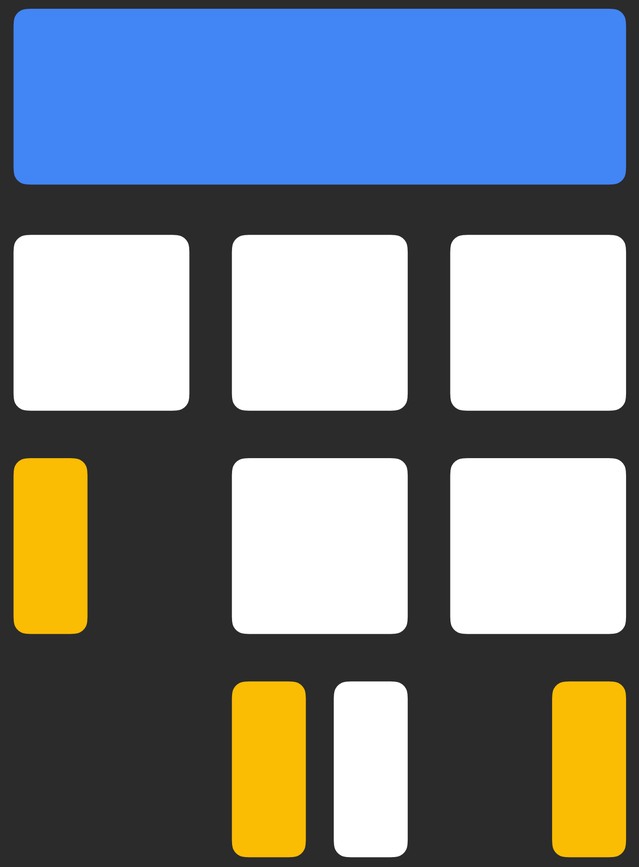
state

```
memo { State(init()) }
```

Used when local mutable state is needed

ambient

Used when data flow is too complex



3. Exploring APIs

Effects

memo

Used when some value is needed

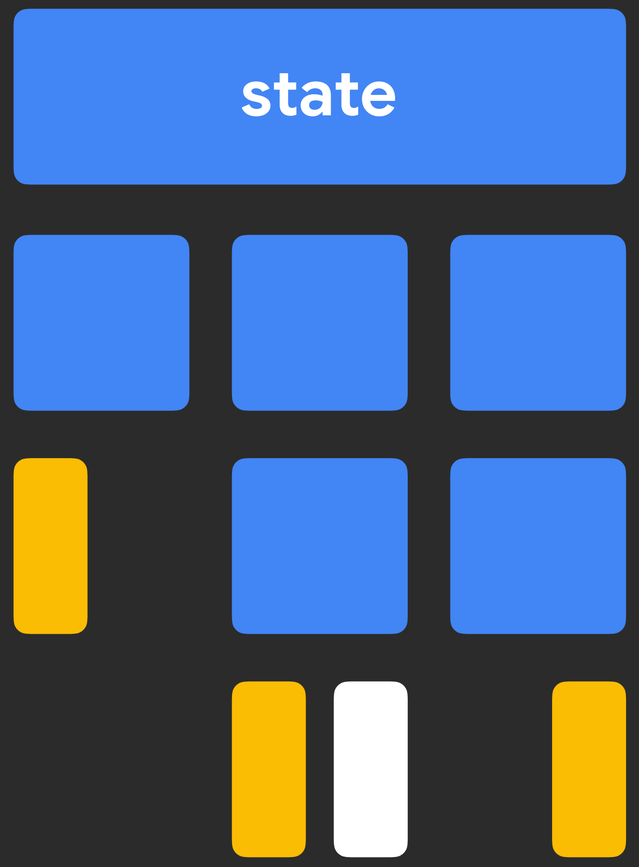
state

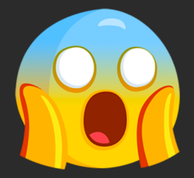
```
memo { State(init()) }
```

Used when local mutable state is needed

ambient

Used when data flow is too complex





3. Exploring APIs

memo

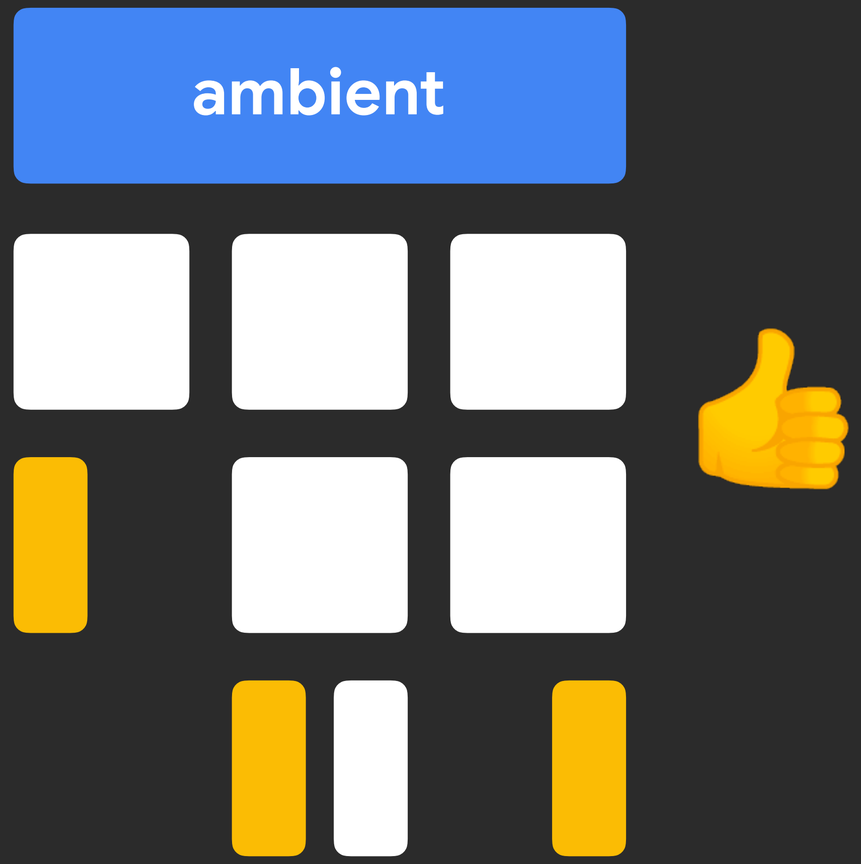
Used when some value is needed

state `memo { State(init()) }`

Used when local mutable state is needed

ambient

Used when data flow is too complex



3. Exploring APIs

onActive

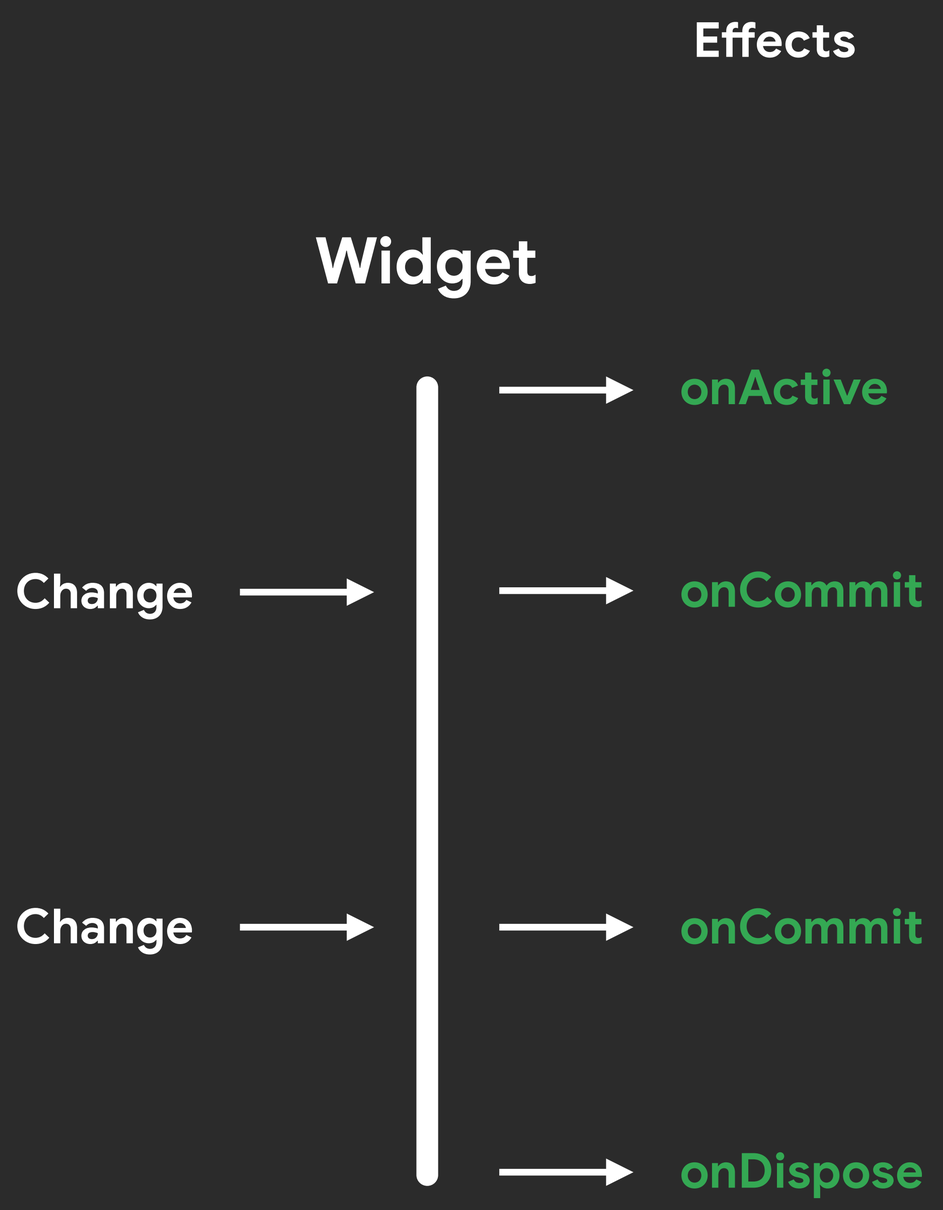
Run code when widget starts showing

onDispose

Run code when widget is not showing anymore

onCommit

Run code when composition happens



3. Exploring APIs

IDE Support

KTX Tags

```
142    FriendWidget(friend)
       Button(
```

Stateless Functional Components (SFCs) should not be invoked, use <FriendWidget /> syntax instead

```
Center {
    Column {
        val count : State<Int> = +state { 0 }
        Text(style = +themeTextStyle { h3 })

        Text(
            text = "Count: ${count.value}",
            style = +themeTextStyle { this.h3 }
        )
        Button(
            text = "Increment",
            onClick = { count.value += 1 }
        )
    }
}
```

```
<AndroidCraneView ref=rootRef>
    var reference: CompositionReference? = null
    var cc: CompositionContext? = null
```

SFC?

Stateless Functional Component

React?

Try Jetpack Compose

Try it out by yourself!

1. Setup repo
2. Get source
3. Launch Android Studio
4. Browse samples

1. Setup repo

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

λ mv repo ~/bin
λ echo ‘PATH=$HOME/bin:$PATH’ >> .zshrc

# For bash
λ echo ‘PATH=$HOME/bin:$PATH’ >> .bashrc
```

2. Get source

3. Launch Android Studio

4. Browse samples

1. Setup repo

2. Get source

```
λ mkdir androidx-master-dev
λ cd android-master-dev

# Get source for branch ‘androidx-master-dev’
λ repo init -u https://android.googlesource.com/platform/manifest \
      -b androidx-master-dev

# Sync with remote
λ repo sync -j8 -c
```

3. Launch Android Studio

4. Browse samples

1. Setup repo

2. Get source

3. Launch Android Studio

```
# Launch specific version of Android Studio
λ cd androidx-master-dev/frameworks/support/ui
λ ./studiow
```

When version is different…

Jetpack Compose: Could not set unknown property 'useIR'

2

I'm trying to compile AndroidX's Jetpack Compose following the instructions available at the README.md file with Android Studio 3.5 Beta 1 and I'm getting the following error from Gradle:

ERROR: Could not set unknown property 'useIR' for task ':ui-android-view:compileDebugKotlin' of type org.jetbrains.kotlin.gradle.tasks.KotlinCompile.

4. Browse samples

1. Setup repo
2. Get source
3. Launch Android Studio
4. **Browse samples**

:ui-demos :ui-material-studies

How can I use Compose

:compose-runtime

How Compose works

:compose-plugin-cli :compose-plugin-ide

Behind the scenes - Resolver, Quick Fix

Recap

Recap

New Declarative UI Toolkit for Android

No more XML, Views - **Just Kotlin**

Pre-alpha

We are hiring!!

Riid!

Riid!

/ get rid of sth /

https://career.riiid.app/android-developer

100% Kotlin

RxJava

Gradle Kotlin DSL

Fun

Riiid!

Q & A