

# Mobile Programming

## Android 8: Jetpack Compose



Ruedi Arnold



# Inhalt

- Kotlin: DSLs als Type Safe Builders
  - Lambda-Ausdrücke & Funktionen höherer Ordnung
  - Function Types with Receiver
  - Scope Functions
- Jetpack Compose
  - Deklarative vs. imperative UI-Programmierung
  - Composable Functions & State Hoisting



<http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg>

**Compose  
baut auf  
Kotlin**

# Einleitung: Kotlin & Jetpack Compose

- Jetpack Compose baut stark auf Kotlin
  - <https://developer.android.com/jetpack/compose/kotlin>  
Inhaltsverzeichnis davon rechts im Bild ->
- Einige Dinge davon haben wir schon angeschaut, andere weniger...
- Bevor wir Jetpack Compose anschauen, folgen zuerst Infos zu Type Safe Builders
  - Damit wird hoffentlich klar(er), wie Jetpack Compose konzeptionell (auf Sprach-Ebene) funktioniert

## Table of contents

Default arguments

Higher-order  
functions and  
lambda  
expressions

Trailing lambdas

Scopes and  
receivers

Delegated  
properties

Destructuring data  
classes

Singleton objects

Type-safe builders  
and DSLs

Kotlin Coroutines

Demo-Code: <https://gitlab.enterpriselab.ch/kotlin/course> 😎

# Kotlin: DSL-Beispiele

- DSL = domain-specific language
  - [https://en.wikipedia.org/wiki/Domain-specific\\_language](https://en.wikipedia.org/wiki/Domain-specific_language)
- Wir schauen DSL-Code-Bsp. aus den folgenden drei Domänen an:
  - Familien-Beziehungen (siehe folgende Code-Demo)
  - HTML (aus Kotlin Doku, siehe <https://kotlinlang.org/docs/type-safe-builders.html>)
  - Jetpack Compose (siehe später bzw. Übung)

...stay tuned! 





<http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg>

## Kotlin: Lambda- Ausdrücke & Funktionen höherer Ordnung

# Typ von Lambdas: Java vs. Kotlin

= Single Abstract Method

- Erinnerung Java: Functional Interface (= SAM-Type)...
  - Grund: Rückwärtskompatibilität
- Kotlin: Function Type
  - `val noArgNoReturn = {print("Just printing")}`
  - `val n : () -> Unit = {print("Just printing")}`
  - **Konsistente & sinnvolle Syntax** (Kein "Handstand" wie mit `@FunctionalInterface` in Java...) 😊
  - `Unit` entspricht `Void` von Java

<https://kotlinlang.org/docs/lambdas.html#function-types>

# Lambda-Ausdrücke: Syntax (an Beispielen)

```
val double1: (Int) -> Int = { x: Int -> 2 * x }
```

Kompakter durch Typinferenz:

```
val double2 = { x: Int -> 2 * x }
```

Kompakter durch impliziten Parametername **it** :

```
val double3: (Int) -> Int = { 2 * it }
```

Kompakter möglich mit mehr Kontext, siehe später...



# Funktionen höherer Ordnung & Lambdas

- Typischer Lambda-Einsatz: Argument für Funktionen höherer Ordnung wie `filter(...)`, `map(...)` und `fold(...)` z.B. auf `List<out E>`

...hm, out?

Kotlin kennt bei Generics  
"Declaration-site Variance"  
anstelle von Wildcards@Java  
(Syntax: `extends` & `super`)

```
val result = listOf<Int>(42, 7, -1, 0)
    .filter { it > 0 }
    .map { it * it }
    .fold(0, { a, b -> a + b } )
```

- Hinweis: Anstelle von `fold(...)` wäre hier `sum()` einfacher

# "Exotische" Syntax mit Trailing Lambdas...

- Was macht der folgende Code?

- Unterschied zum Beispiel vorhin?

```
val result = listOf<Int>(42, 7, -1, 0)
    .filter { it > 0 }
    .map { it * it }
    .fold(0) { a, b -> a + b }
```

- fold(...) hat zwei Argumente:

```
inline fun <T, R> Iterable<T>.fold(
    initial: R,
    operation: (acc: R, T) -> R
): R
```

- **Trailing Lambda:** möglich wenn letztes Argument einer Funktion ein Funktionstyp. (M.E. rel. gewöhnungsbedürftig aber sinnvoll 😊)

# Trailing Lambdas inkl. Demo

- Q: Was tut der folgende Code:

```
myTrailingLambda {  
    print("yes we can")  
}
```

- A: Die folgende Funktion mit einem Argument aufrufen, in der **Trailing-Lambda**-Syntax

```
fun myTrailingLambda(lambda: () -> Unit) {
```

- Ginge auch "herkömmlich" mit (...):

```
myTrailingLambda({ print("yes we can") })
```

# Funktionstypen: Verschiedene Optionen

## Äquivalente Instanzen von "Function Types":

- Aufruf: `sumX(1, 41)`

### 1. Mit Lambda-Ausdruck:

```
val sum1 = { x: Int, y: Int -> x + y }
```

### 2. Mit anonymer Funktion:

```
val sum2 = fun(x: Int, y: Int) : Int = x + y
```

### 3. Mit Callable Reference (analog zu Java's Method Refs)

```
val sum3 = ::sum
```

- Notwendige Funktions-Signatur: `fun sum(x: Int, y: Int) : Int`



<http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg>

## Functions with Receiver

# Empfänger von Funktionen...

- Sinnlos & nicht-kompilierend: `{ toLong() }`
  - Annahme: Es gibt keine Funktion `toLong()` im Scope
  - Auf welchem Objekt soll `toLong()` aufgerufen werden? - Da fehlt ein Empfänger...
  - Z.B. `Int` hätte eine Funktion `toLong()`...

- Lösung: Function Type with receiver

```
val intToLong: Int.() -> Long = { toLong() }  
42.intToLong()
```

- Bsp. von <https://stackoverflow.com/questions/45875491/what-is-a-receiver-in-kotlin>



# Function Types with Receivers

- Weitere Beispiele:

```
val a: Int.() -> Long           // taking an integer as receiver producing a long
val b: String.(Long) -> String  // taking a string as receiver and long as parameter producing a string
val c: GUI.() -> Unit           // taking an GUI and producing nothing
-
```

Siehe `ch.wherever.kotlin.lambda.Receiver.kt`

- Typische Anwendung: Erstellung eigener DSLs (Domain-Specific Languages) mittels Type Safe Builders



<http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg>

## Scope Functions

# Scope Functions: Doku Text

"The Kotlin standard library contains several functions whose sole purpose is to execute a block of code within the context of an object. When you call such a function on an object with a lambda expression provided, it forms a temporary scope. In this scope, you can access the object without its name. Such functions are called **scope functions**. **There are five of them: let, run, with, apply, and also**. Basically, these functions do the same: execute a block of code on an object. What's different is how this object becomes available inside the block and what is the result of the whole expression."

<https://kotlinlang.org/docs/scope-functions.html#scope-functions>

# Scope Functions: Typische Anwendung

```
Person("Alice", 20, "Amsterdam").let {  
    println(it)  
    it.moveTo("London")  
    it.incrementAge()  
    println(it)  
}
```

- Bsp. von <https://kotlinlang.org/docs/scope-functions.html#scope-functions>
  - Selber durchspielen! 😊

# Scope Functions: Scope & Kontext

- Die 5 Scope Functions unterscheiden sich in:
  - Zugriff auf Kontext-Objekt: **this** oder **it**
  - Rückgabewert: **Kontext-Objekt** oder **Lambda-Resultat**

| Function           | Object reference  | Return value   | Is extension function                        |
|--------------------|-------------------|----------------|--|
| <code>let</code>   | <code>it</code>   | Lambda result  | Yes  |
| <code>run</code>   | <code>this</code> | Lambda result  | Yes  |
| <code>run</code>   | -                 | Lambda result  | No: called without the context object        |
| <code>with</code>  | <code>this</code> | Lambda result  | No: takes the context object as an argument. |
| <code>apply</code> | <code>this</code> | Context object | Yes  |
| <code>also</code>  | <code>it</code>   | Context object | Yes  |

<https://kotlinlang.org/docs/scope-functions.html#scope-functions>

# Spezialfall: run ohne Kontext

- Zweck: Block kann ausgeführt werden, wo bloss ein Ausdruck erwartet ist

- Doku-Bsp.:  
Variablen-Init.

(<https://kotlinlang.org/docs/scope-functions.html#run>)

```
val hexNumberRegex = run {  
    val digits = "0-9"  
    val hexDigits = "A-Fa-f"  
    val sign = "+-"  
  
    Regex(pattern: "[$sign]?[$digits$hexDigits]+") ^run  
}
```

- Impl von run(...):

```
@kotlin.internal.InlineOnly  
public inline fun <R> run(block: () -> R): R {  
    contract { this: ContractBuilder  
        | callsInPlace(block, InvocationKind.EXACTLY_ONCE)  
    }  
    return block()  
}
```



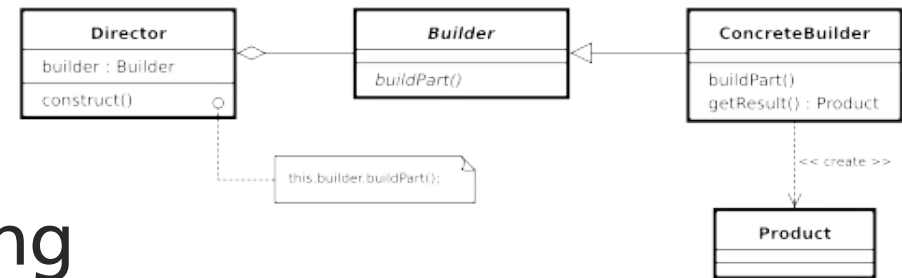


<http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg>

## Type Safe Builders

# Recap: Builder Pattern

UML von [https://en.wikipedia.org/wiki/Builder\\_pattern](https://en.wikipedia.org/wiki/Builder_pattern)



- Separiert Code zur Erzeugung und zur Representation von Objekten
- Löst typischerweise diese beiden Probleme bzw. Unschönheiten:
  - Konstruktoren/Factory-Methoden mit vielen Argumenten
  - Neue Objekte mit unvollständigem oder inkonsistentem Zustand
- Einfaches & bekanntes Beispiel aus Java: Klasse `java.lang.StringBuilder`

# Eigene DSLs mit Type Safe Builders

- Builder Pattern, Kotlin style...
  - Typ-sicher, d.h. vom Compiler überprüft
  - Ermöglichen elegante Builder-Implementierungen
- Type Safe Builders nutzen bei Kotlin i.A. extensiv:
  - Trailing Lambdas
  - Function literals with receiver
  - Ggf. Scope Functions

→ Syntax erscheint ggf. (initial) gewöhnungsbedürftig...

# Type Safe Builder: Anwendungs-Bsp.

- Wir wollen (traditionelle) Familien-Strukturen erzeugen mit zwei Eltern-Teilen und beliebig vielen Kindern

- Aufruf in Kotlin wie folgt:

- Wie funktioniert das?

Wie sieht die Signatur dieser Funktion wohl aus?..

Was passiert hier?  
(Block in Block...)

Siehe `ch.wherever.kotlin.typeSafeBuilder.family`

```
val familySchmitter = family { this: Family
    name = "Schmitter"
    residence = "Luzern"
    parent { this: Family.Parents
        name = "Claudia"
        gender = FEMALE
    }
    child { this: Family.Child
        name = "Max"
        gender = MALE
    }
}
```

# Builder-Code im Detail: Receiver erzeugen...

```
val familySchmitter = family { this: Family
```

Trailing-Lambda-Aufruf der Funktion family



init = Function with Receiver

```
fun family(init: Family.() -> Unit): Family {  
    val family = Family()  
    family.init()  
    return family  
}
```

Erzeugung des Empfängers

Empfänger der Funktion übergeben  
(d.h. Aufruf der Funktion auf  
dem Empfänger)

- Siehe Package `ch.whenever.kotlin.typeSafeBuilder.family`

# Code-Detail: parent inkl. init-Block

```
val familySchmitter = family { this: Family
    name = "Schmitter"
    residence = "Luzern"
    parent { this: Family.Parent
        name = "Claudia"
        gender = FEMALE
    }
}
```

Trailing-Lambda-Aufruf der Funktion parent

Erzeugung vom Empfänger

```
fun parent(init: Parent.() -> Unit) = Parent().also { it: Parent
    it.init()
    familyMembers.add(it)
}
```

Empfänger der Funktion übergeben (d.h. Aufruf der Funktion auf dem Empfänger)

- Siehe Package `ch.whenever.kotlin.typeSafeBuilder.family`



# Hinweis: Scope Functions im Einsatz

- `family` und `parent` tun (praktisch) dasselbe:
  - Empfänger erzeugen
  - Lambda ausführen
  - Empfänger zurück liefern
- `parent` und `child` tun genau dasselbe, zwei verschiedene Implementierungen (einmal mit, einmal ohne scope Function):

```
fun parent(init: Parent.() -> Unit) = Parent().also { it: Parent  
    it.init()  
    familyMembers.add(it)  
}
```

Scope Function!

```
fun child(init: Child.() -> Unit): Child {  
    val child = Child()  
    child.init()  
    familyMembers.add(child)  
    return child  
}
```

# Analog: HTML-Beispiel Kotlin-Doku

- Selber damit rumspielen! 😊

```
fun result() =  
    html {  
        head {  
            title {+"XML encoding"  
        }  
        body {  
            h1 {+"XML encoding wit
```

<https://kotlinlang.org/docs/reference/type-safe-builders.html>

Analoge Beobachtung wie  
bei parent und child im  
Family-Beispiel...

```
fun head(init: Head.() -> Unit) : Head {  
    val head = Head()  
    head.init()  
    children.add(head)  
    return head  
}  
  
fun body(init: Body.() -> Unit) : Body {  
    val body = Body()  
    body.init()  
    children.add(body)  
    return body  
}
```

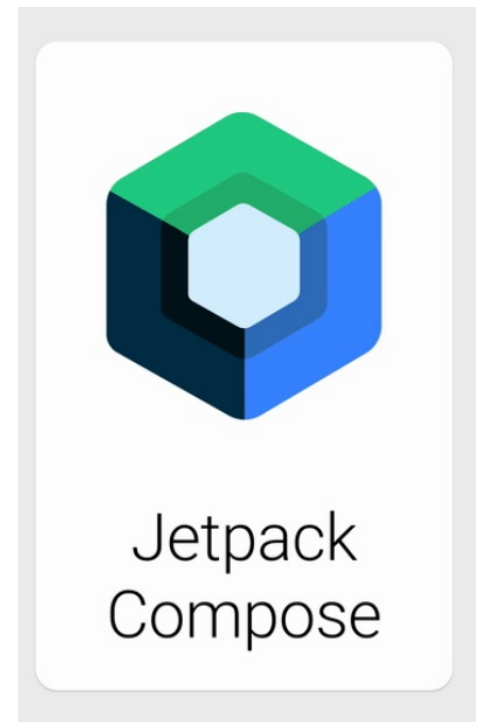
Actually these two functions do just the same thing,  
so we can have a generic version, `initTag` :

```
protected fun <T : Element> initTag(tag: T,  
    tag.init()  
    children.add(tag)  
    return tag  
}
```

# Jetpack Compose & Type Safe Builder

- Jetpack Compose verwendet DSLs für Android-App-GUIs
- D.h. wir können neu das GUI (und Funktionalität) von Android-Apps durch Kotlin-Code deklarieren
  - D.h. layout.xml braucht's nicht mehr, Layout & Logik wird neu durch Code beschrieben!

Let's dive in!..





<http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg>

## HSLU I Modulevaluation

# JETZT: Offizielle Modulevaluation HSLU I & Pause...

- Bitte jetzt ausfüllen!
- Danke schön für konstruktive Rückmeldungen 😊

From: Brühlmann Karin HSLU I <karin.bruehlmann@hslu.ch>  
Subject: Modulevaluation Frühlingssemester 2022  
Date: 4 April 2022 at 13:40:35 CEST  
Cc: ...

Liebe Dozierende

Gerne informieren wir euch, welche Module im Frühlingssemester 2022 evaluiert werden. Ihr findet diese, sortiert nach Modulanlassnummer, im Anhang.

Die elektronischen Evaluationen beginnen nächste Woche am Montag, 11. April 2022 um 10:00 Uhr und werden am Sonntagabend, 24. April 2022 geschlossen. Die Studierenden erhalten am Mittwoch, 20. April 2022 einen Reminder.

Bitte gebt den Studierenden rund zehn Minuten Zeit, um die Evaluation während dem entsprechenden Unterricht auszufüllen.

Vielen Dank für eure Unterstützung und herzliche Grüsse  
Karin





<http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg>

## Jetpack Compose



# Thinking in Compose

"Jetpack Compose is a modern declarative UI Toolkit for Android. Compose makes it easier to write and maintain your app UI by providing a declarative API that allows you to render your app UI without imperatively mutating frontend views. This terminology needs some explanation, but the implications are important for your app design."

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

| Latest Update | Stable Release | Release Candidate | Beta Release | Alpha Release |
|---------------|----------------|-------------------|--------------|---------------|
| April 6, 2022 | 1.1.1          | -                 | -            | 1.2.0-alpha07 |

# The declarative programming paradigm

- Jetzt Abschnitt selber lesen:

<https://developer.android.com/jetpack/compose/mental-model#paradigm>

- Im Modul "Programming Concepts & Paradigms" (PCP) geht's zentral um deklarative vs. imperative Programmierung (Prolog, Scheme, modern Java)

- ...besuchen! (falls noch nicht passiert)

- Wer ist/war im PCP?..

...quasi "Werbung in eigener (Modul-)Sache"! 🤪

- Spontane Rückmeldungen zum Modul? 🧐

The declarative programming paradigm

Historically, an Android view hierarchy has been representable as a tree. When the app changes because of things like user interactions, the UI must update to display the current data. The most common way of updating the UI is to call `findViewById()` to find a view, and then change it, like `textView.setText(newText)` or `imageView.setImageResource(R.drawable.new_image)`.

# Jetpack Compose: Grundidee

- Bisher: Zustand von UI-Widgets haben Zustand, dieser wird programmatisch geändert
  - **Imperatives Paradigma**
    - lat.: imperare = befehlen
- Compose: UI-Widget haben (möglichst) keinen Zustand, ganze App wird beschrieben
  - **Deklaratives Paradigma**
    - lat.: declaratio = Erklärung

# Eine einfache "Composable Function"



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

"this annotation informs the Compose compiler that this function is intended to convert data into UI."

"Composable functions emit UI hierarchy by calling other composable functions"

<https://developer.android.com/jetpack/compose/mental-model#simple-example>

# Eigenschaften von Composable Functions

- Nehmen Daten entgegen (aktueller UI-Zustand)
- Keine Rückgabe: "Compose functions that **emit UI** do not need to return anything, because they describe the desired screen state instead of constructing UI widgets"
- Schnell: werden (potentiell) oft neu gezeichnet
- Idempotent: verhält sich gleich, wenn mehrmals mit denselben Argumenten aufgerufen
- Keine Seiteneffekte: verändert keinen globalen Zustand o.ä.

...sind also "pure functions" 😊

<https://developer.android.com/jetpack/compose/mental-model#simple-example>

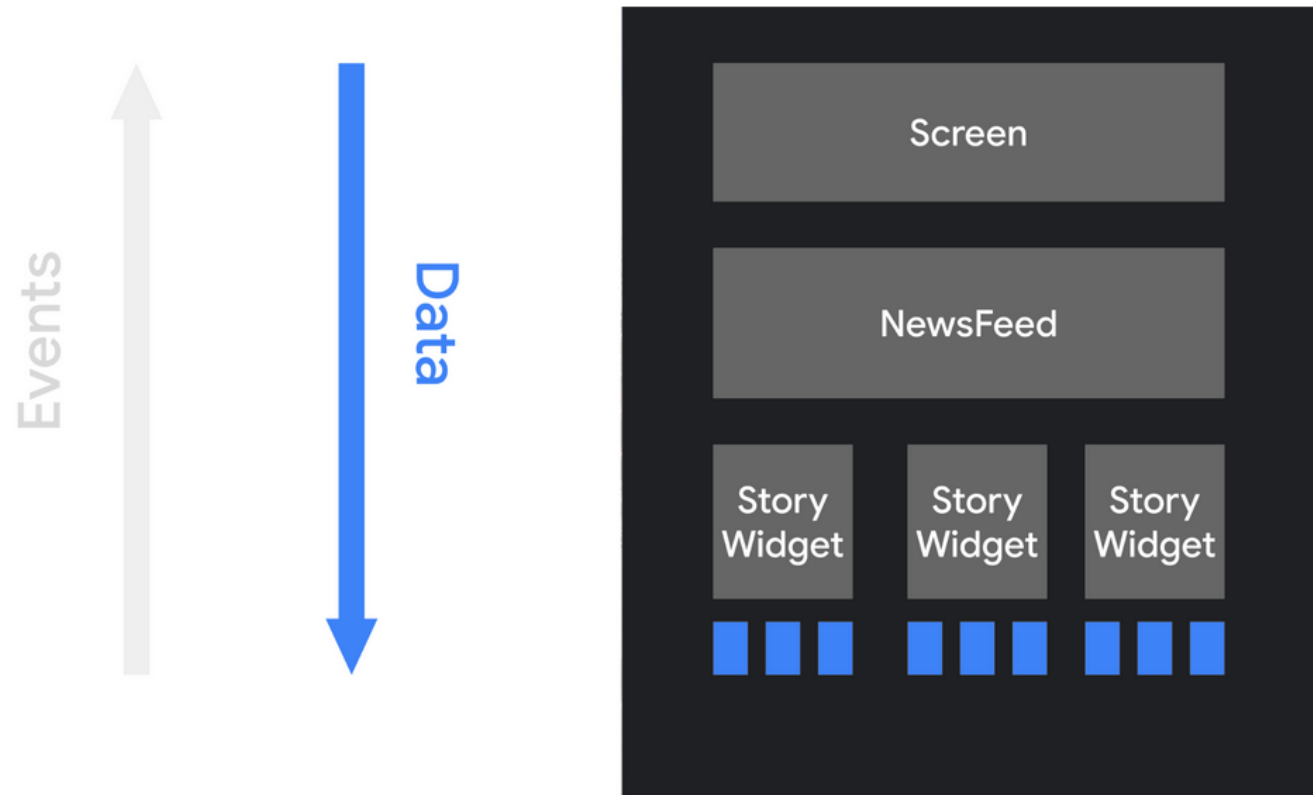
# The declarative paradigm shift

"With many imperative object-oriented UI toolkits, you initialize the UI by instantiating a tree of widgets. You often do this by inflating an XML layout file. Each widget maintains its own internal state, and exposes getter and setter methods that allow the app logic to interact with the widget.

In Compose's declarative approach, widgets are relatively stateless and do not expose setter or getter functions. In fact, widgets are not exposed as objects. You update the UI by calling the same composable function with different arguments. This makes it easy to provide state to architectural patterns such as a ViewModel, as described in the Guide to app architecture. Then, your composables are responsible for transforming the current application state into a UI every time the observable data updates."

<https://developer.android.com/jetpack/compose/mental-model#paradigm>

# Daten: von der App-Logik zum UI

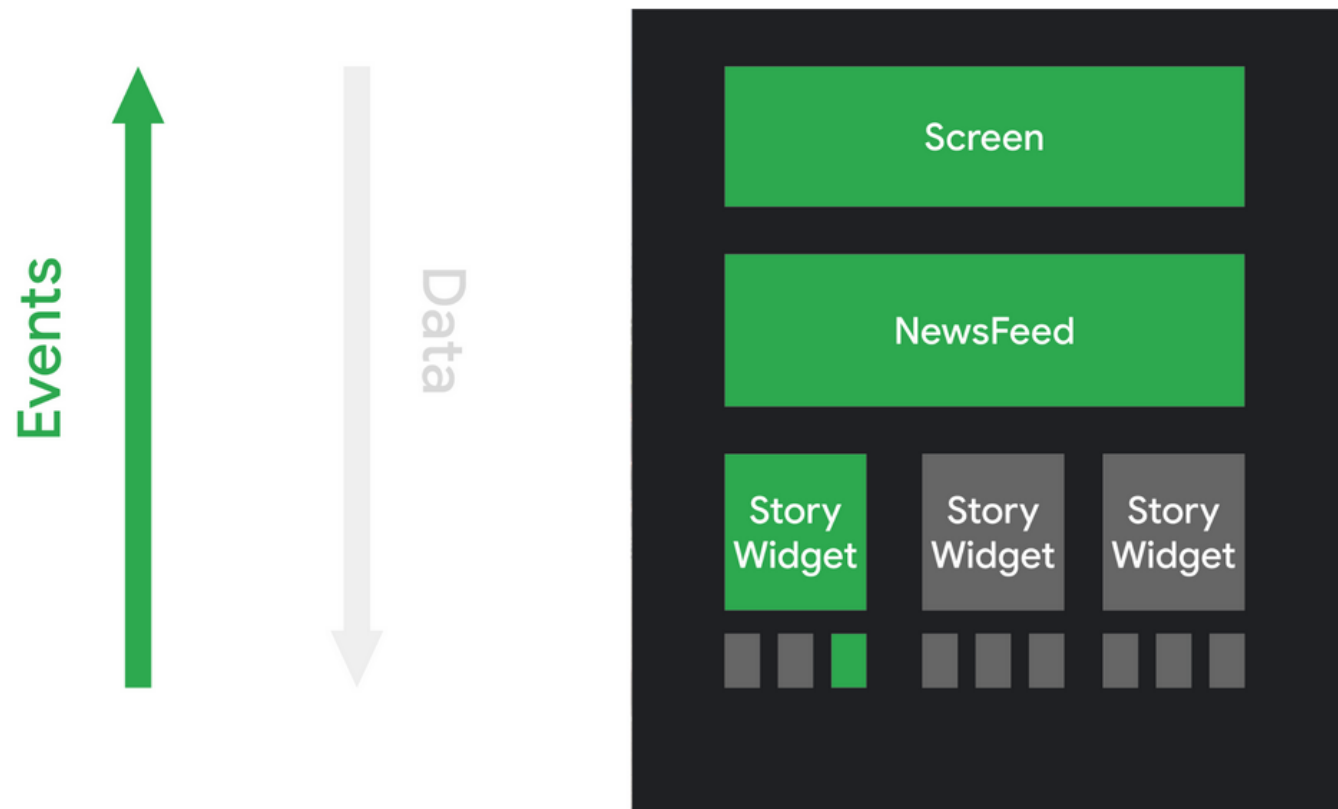


**Figure 2.** The app logic provides data to the top-level composable function. That function uses the data to describe the UI by calling other composables, and passes the appropriate data to those composables, and on down the hierarchy.

<https://developer.android.com/jetpack/compose/mental-model#paradigm>



# Events: vom UI zur App-Logik



**Figure 3.** The user interacted with a UI element, causing an event to be triggered. The app logic responds to the event, then the composable functions are automatically called again with new parameters, if necessary.

<https://developer.android.com/jetpack/compose/mental-model#paradigm>



# Jetpack Compose: (top) Doku

- <https://developer.android.com/jetpack/compose/>
- Core Concepts
  - <https://developer.android.com/jetpack/compose/mental-model>
- Kurs "Pathway to Compose"
  - <https://developer.android.com/courses/pathways/compose>
- Code-Labs, Guides, ...

## Foundation

[Thinking in Compose](#)

Managing state

Lifecycle and navigation

## Development

Android Studio

Tooling

Kotlin for Compose

## Design

Layout



<http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg>

## Übung 8:

**Composables,  
State hoisting &  
"Compose-DSL"**

# Übung 8: 3 Aufgaben & Demo (Siehe Aufg.-Blatt)

## 1. Code-Lab "Jetpack Compose basics" komplett durchgehen & ausführen

🕒 65 mins remaining

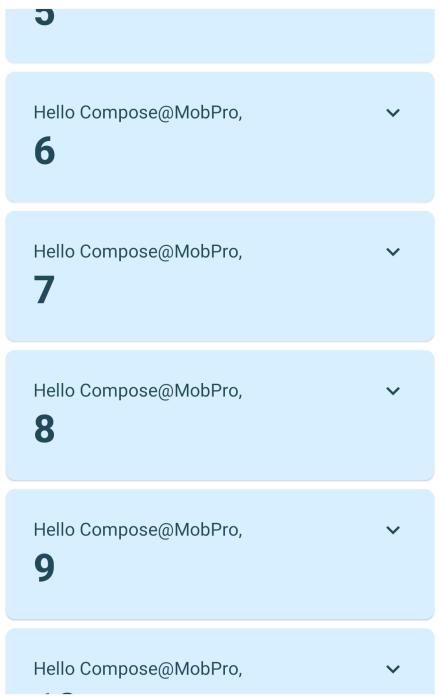
- <https://developer.android.com/codelabs/jetpack-compose-basics>
- Dazu in Android Studio ein **eigenes neues Compose-Projekt** mit dem Namen **"FirstCompose"** erzeugen

## 2. Erweiterung: "Dynamischer Titel"

## 3. Erweiterung: "Zähler- & Reset-Knopf"

Last clicked #42

I've been clicked 6 times reset counter

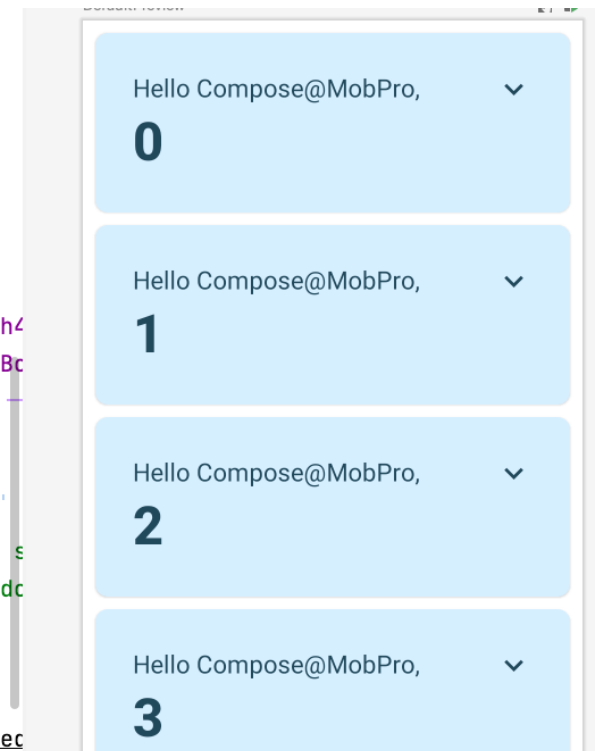


# Code-Lab Basics: Demo & Hinweise

- Code-Lab schrittweise durchspielen & Code-Bsp. möglichst im Detail nachvollziehen...
  - Projektname: "FirstCompose"
  - Package: `ch.hslu.mobpro.firstCompose`

```
Column(  
    modifier = Modifier  
        .weight( weight: 1f)  
        .padding(12.dp)  
) {  
    this: ColumnScope  
    Text(text = "Hello Compose@MobPro, ")  
    Text(  
        text = name,  
        style = MaterialTheme.typography.h4  
        fontWeight = FontWeight.ExtraB  
    )  
    if (expanded) {  
        Text(  
            text = ("Compose ipsum color s  
                "padding theme elit, sed do  
        )  
    }  
}
```

MobPro - `IconButton(onClick = { expanded = !expanded`



# Composables at Work: "alles" ist Composable...

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState:
        super.onCreate(savedInstanceState)
        setContent {
            BasicsCodeLabTheme {
                MyApp()
            }
        }
}

@Composable
fun BasicsCodeLabTheme(
    darkTheme: Boolean = isSystemInDarkTheme(),
    content: @Composable () -> Unit
) {
    val colors = if (darkTheme) {
        DarkColorPalette
    } else {
        LightColorPalette
    }

    @Composable
    private fun MyApp() {
        var shouldShowOnboarding by rememberSaveable { mutableStateOf(true) }

        if (shouldShowOnboarding) {
            OnboardingScreen(onContinueClicked = { shouldShowOnboarding = false })
        } else {
            Greetings()
        }
    }
}
```

# ComponentActivity.setContent(...)

Composes the given composable into the given activity. The `content` will become the root view of the given activity.

This is roughly equivalent to calling `ComponentActivity.setContentView` with a `ComposeView` i.e.:

```
setContentView(  
    ComposeView(this).apply {  
        setContent {  
            MyComposableContent()  
        }  
    }  
)
```

Params: parent - The parent composition reference to coordinate scheduling of composition updates  
content - A `@Composable` function declaring the UI contents

```
public fun ComponentActivity.setContent(  
    parent: CompositionContext? = null,  
    content: @Composable () → Unit  
) {
```

Composable-Lambda = letztes Argument:  
d.h. trailing Lambda-Syntax möglich! 😊



# LazyColumn als Type Safe Builder

## Type-safe builders and DSLs

---

Kotlin allows creating [domain-specific languages \(DSLs\)](#) with type-safe builders. DSLs allow building complex hierarchical data structures in a more maintainable and readable way.

Jetpack Compose uses DSLs for some APIs such as `LazyRow` and `LazyColumn`.



```
@Composable
fun MessageList(messages: List<Message>) {
    LazyColumn {
        // Add a single item as a header
        item {
            Text("Message List")
        }

        // Add list of messages
        items(messages) { message ->
            Message(message)
        }
    }
}
```

<https://developer.android.com/jetpack/compose/kotlin#dsl>

# Compose-DSL-Code am Bsp. LazyColumn

The vertically scrolling list that only composes and lays out the currently visible items. The `content` block defines a DSL which allows you to emit items of different types. For example you can use `LazyListScope.item` to add a single item and `LazyListScope.items` to add a list of items.

`@Composable`

```
private fun Greetings(names: List<String>) {  
    LazyColumn(modifier = Modifier.padding(16.dp)) {  
        items(items = names) { name  
            Greeting(name = name)  
        }  
    }  
}
```

(Analoger Mechanismus zum Aufruf von parent im Family-Bsp. von vorne ☺)

Aufruf der DSL-Funktion items vom Interface LazyListScope

`@Composable`

```
fun LazyColumn(  
    modifier: Modifier = Modifier,  
    state: LazyListState = rememberLazyListState(),  
    contentPadding: PaddingValues = PaddingValues(0.dp),  
    reverseLayout: Boolean = false,  
    verticalArrangement: Arrangement.Vertical =  
        if (!reverseLayout) Arrangement.Top else Arrangement.Bottom,  
    horizontalAlignment: Alignment.Horizontal = Alignment.Start,  
    flingBehavior: FlingBehavior = scrollableDefaults.flingBehavior(),  
    content: LazyListScope.() → Unit
```

Function Type with Receiver!

# State hoisting (Onboarding anzeigen oder nicht?)

- Zustand speichern mit **rememberSaveable**
  - Analog legen wir in Übung 8 eigene Variablen an (für Zähler & angeklickte Zeile)
- Syntax mit **by** heisst "Delegated property"
  - <https://kotlinlang.org/docs/delegated-properties.html>
- Dieser Zustand-Mechanismus heisst "State hoisting" (siehe später...)

```
@Composable
private fun MyApp() {
    var shouldShowOnboarding by rememberSaveable { mutableStateOf( value: true) }

    if (shouldShowOnboarding) {
        OnboardingScreen(onContinueClicked = { shouldShowOnboarding = false })
    } else {
        Greetings()
    }
}
```

rememberSaveable:  
praktische Funktion! 😊

# Übung 8: Code Erweiterungen 🚀

- Listen-Zustand merken & im Titel anzeigen
  - Wo Variable deklarieren?
- Eigene Composables  
CounterButton,  
ResetButton und  
CounterButtonRow
- D.h.: Übung & Anwendung von  
State-Hoisting 😊

Last clicked #42

I've been clicked 6 times reset counter

5

Hello Compose@MobPro,

6

Hello Compose@MobPro,

7

Hello Compose@MobPro,

8

Hello Compose@MobPro,

9

Hello Compose@MobPro,

# Übung 8: Anpassungen Greetings

Last clicked #42

I've been clicked 6 times

reset counter

5

Hello Compose@MobPro,

6

## ■ Vorher (Code-Lab):

`@Composable`

```
private fun Greetings(names: List<String> = List(size: 1000)
    LazyColumn(modifier = Modifier.padding(vertical = 4.dp)) {
        items(items = names) { name ->
            Greeting(name = name)
```

LazyColumn

## ■ Nachher (Ü8) anstelle der LazyColumn eine Column und die LazyColumn als drittes Element darin:

```
Column(modifier = Modifier.fillMaxHeight()) { this: ColumnScope
    MainTitle(text = lastText)
    CounterButtonRow()
    LazyColumn(modifier = Modifier.padding(vertical = 4.dp)) {
        items(items = names) { name ->
            Greeting(name = name,
```

Neu: Column

Bisherige  
LazyColumn ...

# Stateless Composeables & State Hoisting

to hoist = heben, hochziehen

"State hoisting in Compose is a pattern of moving state to a composable's caller to make a composable stateless. The general pattern for state hoisting in Jetpack Compose is to replace the state variable with two parameters:"

- `value: T`: the current value to display
- `onValueChange: (T) -> Unit`: an event that requests the value to change, where `T` is the proposed new value

<https://developer.android.com/jetpack/compose/state>



**Key Point:** When hoisting state, there are three rules to help you figure out where state should go:

1. State should be hoisted to at least the **lowest common parent** of all composables that use the state (read).
2. State should be hoisted to at least the **highest level it may be changed** (write).
3. If **two states change in response to the same events** they should be **hoisted together**.

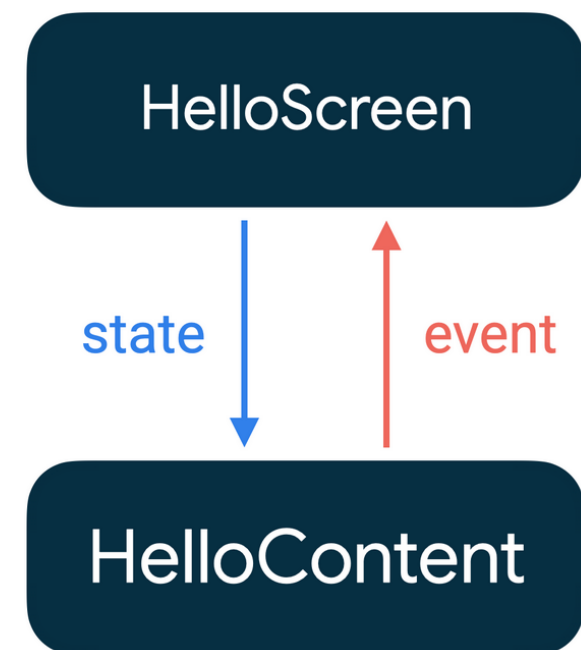
You can hoist state higher than these rules require, but underhoisting state will make it difficult or impossible to follow unidirectional data flow.

# State Hoisting: Motivation...

Ziel: Entkopplung von Zustand & Darstellung

"By hoisting the state out of HelloContent, it's easier to reason about the composable, reuse it in different situations, and test. HelloContent is decoupled from how its state is stored."

<https://developer.android.com/jetpack/compose/state>





# State hoisting @ Code-Lab-Code

- Nochmals Code-Beispiel von Folie 47...

```
@Composable
private fun MyApp() {
    var shouldShowOnboarding by rememberSaveable { mutableStateOf( value: true) }

    if (shouldShowOnboarding) {
        OnboardingScreen(onContinueClicked = { shouldShowOnboarding = false })
    } else {
        Greetings()
    }
}
```

- Q: Wie kommt Zustandsänderung in den OnboardingScreen (bzw. von dort zurück)?
- A: Mit einem Callback! 😊

```
@Composable
```

```
fun OnboardingScreen(onContinueClicked: () -> Unit) {
```

# Ü8: State Hoisting - Code & Demo "Counter"

I've been clicked 6 times

- Zähler-Knopf: Anzahl Klicks
  - Braucht also Zustand...
- Widget ist zustandlos, d.h. Daten müssen von aussen (d.h. oben in der Composable-Hierarchie) kommen...

```
@Composable
fun CounterButtonRow() {
    var counterState by rememberSaveable { mutableStateOf(0) }

    Row(
        modifier = Modifier
            .padding(12.dp)
    ) {
        this: RowScope
        CounterButton(
            count = counterState,
            updateCount = { newCount ->
                counterState = newCount
            }
        )
    }
}
```

"Source of truth" für den Zähler-Zustand

Zähler +1 wenn Knopf gedrückt wird

Wichtig: count UND updateCount wird mitgegeben!

```
@Composable
fun CounterButton(count: Int, updateCount: (Int) -> Unit) {
    Button(
        onClick = { updateCount(count + 1) },
        k Compose
```

## Übung 8: Knopffarbe ändern

- Je nach aktuellem Wert vom Zähler soll CountButton andere Farbe haben...

@Composable

```
fun CounterButton(count: Int, updateCount: (Int) -> Unit) {  
    Button(  
        onClick = { updateCount(count + 1) },  
        colors = ButtonDefaults.buttonColors(  
            backgroundColor = if (count > 5) Color.Green else
```

I've been clicked 5 times

I've been clicked 6 times

# Übung 8: Dynamischer Titel

Last clicked #42

I've been clicked 6 times

reset counter

Hello Compose@MobPro,



## ■ Click auf Row ändert Titel...

Woher kommt updateLastText? Wo soll dafür die Single Source of Truth sein?..

@Composable

```
private fun CardContent(name: String, updateLastText: (String) -> U
    var expanded by remember { mutableStateOf(value: false) }
```

Row(

modifier = Modifier

.padding(12.dp)

.animateContentSize(

animationSpec = spring(

dampingRatio = Spring.DampingRatioMediumBouncy,

stiffness = Spring.StiffnessLow

)

)

clickable-Modifier aufrufen!

.clickable { updateLastText("Last clicked #\${name}") }

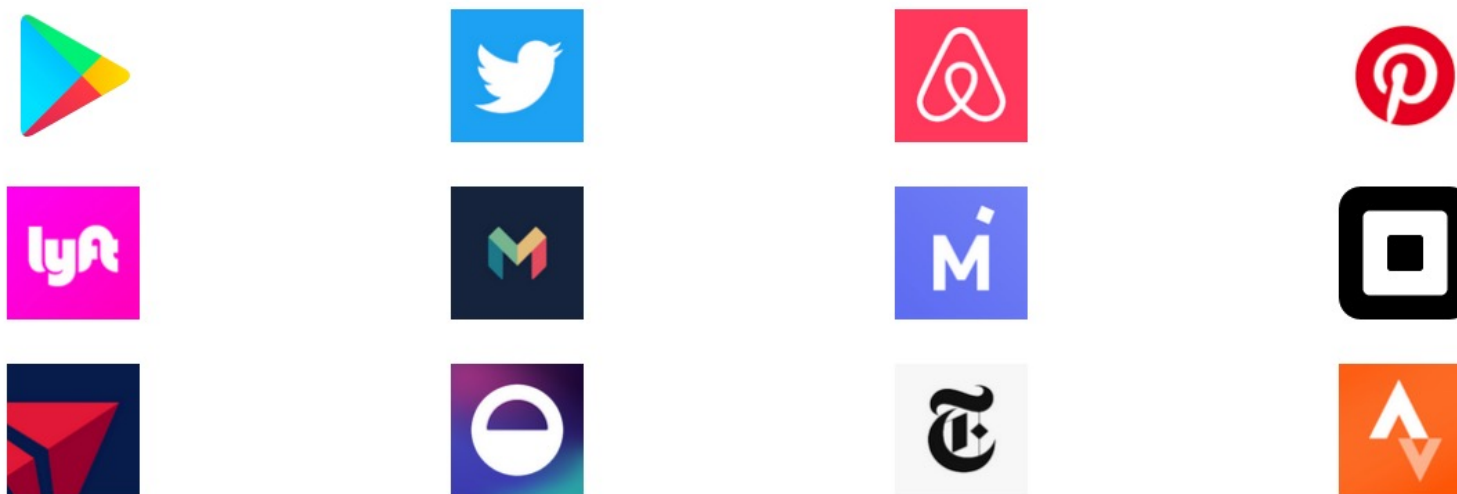


<http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg>

## Daklarative GUI: Fazit & Ausblick

# Einsatz & Verbreitung

- Android setzt stark auf Compose
  - "Apps built with Compose"



<https://developer.android.com/jetpack/compose>



# Fazit & Ausblick





- Die GUI-Welt wird deklarativer!
  - Hybrid: Flutter (<https://flutter.dev/>, seit 2017)
  - iOS: SwiftUI (<https://developer.apple.com/xcode/swiftui/>, seit 2019)
  - **Android neu: Jetpack Compose** (aktuell alpha) ...(b)leading edge technology! 🙌
- Deklaratives Paradigma bietet Vorteile (vs. imperativ)
  - Zustandslos, einfacher testbar
  - Weniger Code & alles Code (vs. xml & Code)
  - Wiederverwendbar (Komposition!)
  - ...
- Meine Prognose: Gekommen um zu bleiben! 😎



<http://en.wikipedia.org/wiki/File:Somethingdifferent.jpg>

## Vor-Infos Team-Projekt

# Grobüberblick 2/2

| SW | Datum | Inhalt  |
|----|-------|---|
| 08 | 11.4. | Android 8: Jetpack Compose  |
| 09 | 18.4. |  OSTERN (frei)  |
| 10 | 25.4. | Android 7: Testing & Build System   |
| 11 | 2.5.  | <b>Team-App-Projekt</b>   |
| 12 | 9.5.  |   |
| 13 | 16.5. |   |
| 14 | 23.5. |   |
| 15 | 30.5. |   |
| 13 | 16.5. | ggf. inkl. Gastvorträge (Ubique, Xamarin, Flutter, ???, ...)  |
| 15 | 30.5. | App-Präsentationen & Demos, Abschluss   |

# Grob-Infos zum Team-Projekt (mehr am 25.4.!)

- Selbständige Arbeit
- 2er-Teams
  - Max. 1 3er- oder 1er-Team 😊
- Dozierende = Coaches
- App-Inhalt frei pro Team wählbar!
  - Jedoch: Technische (nicht-funktionale) Anforderungen
- Es wird vorgegeben Termine & konkret erwartete Artefakte geben
  - Z.B. Papier-Prototyp, d.h. Skizzen/Zeichnungen von **jedem relevanten Bildschirm** der gewünschten App

# Team-Projekt: Kickoff am 25.4!

- D.h. 5 Wochen Zeit für eigene App 😊
  - Design, Implementierung & Präsentation

...z.B. falls jemandem  
über Ostern langweilig  
sein sollte 🐰🐰

- Mit was ihr ggf. bereits beginnen könnt:
  - 2er-Team bilden
  - App-Ideen sammeln & Skizzen davon erstellen
  - Techn. Prototypen implementieren
  - ...