Kapitel 1
Jetpack Compose

Über dieses Update-Kapitel

Die erste Fassung dieses Kapitels ist im Herbst 2020 im Buch »Kotlin – Das umfassende Handbuch« erschienen (Rheinwerk Verlag 2020). Jetpack Compose lag damals erst in einer frühen Alpha-Version vor.

Mittlerweile ist JetPack Compose 1.0 fertig. Als Update-Service für die Lesergemeinde meines Buchs habe ich das Kapitel auf dieser Basis vollständig aktualisiert. Der geänderte Text wurde allerdings nicht korrekturgelesen und wird daher mehr Tipp- und Rechtschreibfehler als üblich enthalten.

Mehr Informationen zu meinem Kotlin-Buch, weitere Updates sowie Downloads zu den hier präsentierten Beispielprojekten finden Sie hier:

https://kofler.info/buecher/kotlin
https://kofler.info/tag/kotlin-updates

1.1 Einführung

Jetpack Compose wurde erstmalig auf der Entwicklermesse *Google I/O* im Mai 2019 vorgestellt. Es handelt sich dabei um einen neuen Weg, um Benutzeroberflächen für Apps zusammenzustellen: Anstatt die Aktivitäten und Steuerelemente wie bisher in einem grafischen Editor einzurichten und intern im XML-Format zu speichern, wird dazu nun Kotlin-Code mit der Annotation `@Composable` verwendet.

Der riesige Vorteil von Jetpack Compose besteht darin, dass Sie sich nicht mehr länger mit dem sperrigen Layouteditor und unübersichtlichen Layoutregeln herumärgern müssen. Vielmehr können Sie das Erscheinungsbild Ihrer App durch verhältnismäßig kompakten Code ausdrücken und es direkt in Android Studio ansehen und testen. Das geht schneller, lässt sich besser dokumentieren und rascher ändern. Interessanterweise beschreitet Apple mit *SwiftUI* zugleich genau denselben Weg. Auch Xcode-Jünger haben die Nase voll von UI-Designs, deren Erstellung Tausende Maus- oder Trackpad-Klicks erfordert.

Ein weiterer Vertreter in der aktuell rasch wachsenden Familie von deklarativen UI-Frameworks ist *Flutter* (siehe *https://flutter.dev*). Es gibt also aktuell einen regelrechten Trend hin zu sogenannten *deklarativen* UI-Frameworks.

Sobald ich von den Compose-Plänen hörte, wollte ich den gesamten Android-Teil dieses Buchs entsprechend umstellen. Das hat sich leider als unmöglich herausgestellt: Im Sommer 2020 war Jetpack Compose immer noch im Preview-Stadium und ließ sich nur mit instabilen Canary-Versionen von Android Studio nutzen. (Canary-Versionen gehen den Beta-Versionen voraus und entsprechen in etwa dem, was bei anderen Produkten Alpha-Versionen sind.)

Außerdem gab es im Sommer 2020 keine brauchbare Dokumentation zu Jetpack Compose. Als Fertigstellungstermin der ersten stabilen Version von Jetpack Compose wurde vage das Jahr 2021 kommuniziert. Jetpack Compose ist zweifellos die Zukunft – aber diese war eben noch nicht da, als ich das Buch verfasste.

Mittlerweile hat sich viel geändert! Im Sommer 2021 hinterlässt Jetpack Compose 1.0 einen stabilen Eindruck. Damit ist jetzt der richtige Zeitpunkt, um sich produktiv mit Jetpack Compose auseinanderzusetzen. Die Testprogramme für dieses Kapitel habe ich mit Android Studio Arctic Fox (= Android Studio 2020.3) und Jetpack Compose 1.0 durchgeführt.

Neue Versionsnummern für Android Studio

Arctic Fox ist der Codename der aktuellen Android-Studio-Version 2020.3. Die Versionsnummer ergeben sich neuerdings aus der zugrundeliegendenden IntelliJ-Version. Beim aktuellen Release wirkt das etwas unglücklich: Obwohl Android Studio Arctic Fox im Juli 2021 erschienen ist, lautet die Versionsbezeichnung 2020.3 und vermittelt somit keine Aktualität. Es ist zu hoffen, dass künftige Android-Studio-Versionen mit neueren IntelliJ-Versionen synchronisiert werden.

Aktuelle Informationen zu Jetpack Compose finden Sie hier:

https://developer.android.com/jetpack/compose
https://developer.android.com/jetpack/compose/tutorial
https://developer.android.com/jetpack/androidx/releases/compose

Mehr als eine neue Programmiertechnik

Jetpack Compose ist nicht einfach ein neuer Weg, um Steuerelemente anzusprechen. Vielmehr will Google mit Jetpack Compose die App-Programmierung gleich in mehrerlei Hinsicht revolutionieren:

- Jetpack-Updates erfolgen außerhalb der Core-Android-Updates. Auch wenn ein Handy-Hersteller keine Lust hat, ein Update auf Android 11 oder 12 auszuliefern, können Sie in Ihren Apps die neuesten Jetpack-Features nutzen. Davon profitieren nicht nur Sie, sondern auch Ihre Kunden.
- Aufgrund des deklarativen Konzepts von Compose-Funktionen ändert sich der Datenfluss in Ihrer App grundlegend. Das erfordert anfangs ein komplettes Umdenken, führt aber längerfristig zu kompakterem besseren Code.

Video-Tipps

Investieren Sie eine halbe Stunde Zeit, und sehen Sie sich das folgende Video an!

https://youtu.be/VsStyq4Lzxo

Dieser bei der *Google I/O 2019* aufgenommene Vortrag zum eher nichtssagenden Thema »Declarative UI Patterns« erklärt, was sich Google bei der Konzeption von Jetpack Compose gedacht hat und warum Sie sich damit anfreunden sollten (und längerfristig müssen). Einige Details des Videos sind mittlerweile zwar überholt, etwa die darin erwähnte `Model`-Annotation, dennoch vermittelt die Präsentation einen guten Einblick in die neue Compose-Denkweise.

Aktuellere Videos, die sich weniger auf Grundlagen konzentrieren und mehr die praktische Anwendung zeigen, finden Sie hier:

https://www.youtube.com/watch?v=Ef1xKWjA9E8
https://www.youtube.com/watch?v=7Mf2175h3RQ

Licht und Schatten

Ich gebe es offen zu: Das Konzept von Jetpack Compose begeistert mich. Das liegt daran, dass ich lieber Code verfasse als umständlich per Maus oder Trackpad in versteckten Dialogen nach Layoutoptionen zu suchen. Die Arbeit per Tastatur ist nicht nur effizienter, sie lässt sich hinterher auch leichter nachvollziehen und besser dokumentieren. (Ich sehe das aus meiner Perspektive als Buchautor: Bei einem Algorithmus gibt ein Listing eine vollständige Referenz. Bei herkömmlichen Android-Apps muss ich hingegen immer hinzufügen: Außerdem müssen Sie im Dialog A die Option X einstellen und im Dialog B das Listenfeld Y auf den Wert Z stellen etc.)

Dessen ungeachtet ist das Compose-Konzept keineswegs frei von Problemen. Ein offensichtlicher Nachteil besteht im exzessiven Lambda-Einsatz, der zu einer Klammernhölle wie in JavaScript führt. Sobald Ihre App mehr als »Hello World!« am Bildschirm darstellen soll, lassen sich schier endlose Verschachtelungen von Compose-Funktionen kaum vermeiden.

Den Einstieg in Compose erschweren zudem unzählige Anleitungen in Blogs und auf Stack Overflow, die sich auf frühe Compose-Versionen beziehen und längst nicht mehr funktionieren.

Ebenfalls irritierend sind die relativ langen Build-Zeiten, die einem flotten interaktiven Layoutprozess im Wege stehen. Das hat damit zu tun, dass selbst relativ einfache Apps aus unzähligen Bibliotheken und Komponenten zusammengesetzt werden müssen. Jetpack Compose ändert an dieser Situation nichts.

Zu guter Letzt ist Jetpack Compose zwar als finales Release verfügbar, aber es trägt nun einmal erst die Versionsnummer 1.0. Nicht jeder Aspekt von Jetpack Compose wirkt bereits vollständig ausgereift (um es gelinde zu formulieren).

Kein Java

Jetpack Compose ist eine radikale Absage an Java: Jetpack Compose erfordert ein spezielles Kotlin-Compiler-Plugin und kann nicht durch Java-Code genutzt werden. Als Leserin oder Leser eines Kotlin-Buchs wird Ihnen das vermutlich egal sein. Die Frage ist nur, ob diese Entscheidung nicht den Erfolg von Compose behindern wird.

1.2 Hello Compose!

Um ein neues Projekt mit Jetpack Compose einzurichten, wählen Sie nach FILE • NEW PROJECT das Template EMPTY COMPOSE ACTIVITY aus. (Es ist zu erwarten, dass es in Zukunft mehr Templates für Compose-Projekte geben wird.)

Die Kotlin-Version ist in den beiden `build.gradle`-Dateien (*Project* und *Module*) mit 1.5.10 festgelegt. Neuere Versionsnummern wurden im Juli 2021 nicht unterstützt und führten zum Build-Fehler *Execution failed for task app:compileDebugKotlin*.)

Im neuen Projekt enthält die Datei `MainActivity.kt` in nur 20 Zeilen den gesamten Code für ein simples Muster-Projekt. Dabei richtet die Methode `setContent` die initiale Ansicht der App ein. Der Lambda-Ausdruck ruft die Funktion `HelloComposeTheme` auf, die in der Datei `ui/Theme.kt` definiert ist und Basiseinstellungen für das Layout der App enthält. (Der Name dieser Funktion variiert je nach Projekt. Er ergibt sich aus dem Projektnamen plus `Theme`.)

Die Funktion `Greeting` erzeugt ein Label mit dem Text `Hello Android!`. Grundsätzlich ist `Greeting` zwar nur eine ganz simple Funktion, die einen Parameter an ein Textfeld weitergibt. Aber Sie können `Greeting` auch als Muster für die Implementierung eigener Steuerelemente betrachten:

```
// Mustercode für eine 'Empty Compose Activity'
// Datei MainActivity.kt
class MainActivity : ComponentActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            HelloComposeTheme {
               Surface(color = MaterialTheme.colors.background) {
                  Greeting("Android")
               }
            }
        }
    }
}

@Composable
fun Greeting(name: String) {
    Text(text = "Hello $name!")
}

// nur für die Preview-Funktion in Android-Studio
@Preview(showBackground = true)
@Composable
fun DefaultPreview() {
    HelloComposeTheme {
        Greeting("Android")
    }
}
```

Die Funktion `DefaultPreview` ist für die App nicht relevant, ermöglicht es aber, das Ergebnis der `Greeting`-Funktion vorweg im Preview-Fenster anzusehen bzw. zu testen (siehe Abbildung 1.1).

Im Hello-World-Muster sind die Funktionen `Greeting` und `DefaultPreview` außerhalb der Aktivitätsklasse platziert. Das vermindert zwar die Verschachtelung, ist aber nur zweckmäßig, wenn Sie in den Funktionen nicht auf Eigenschaften der Klasse zugreifen müssen. Die Preview-Ansicht funktioniert auch für parameterfreie Methoden innerhalb der Klasse.

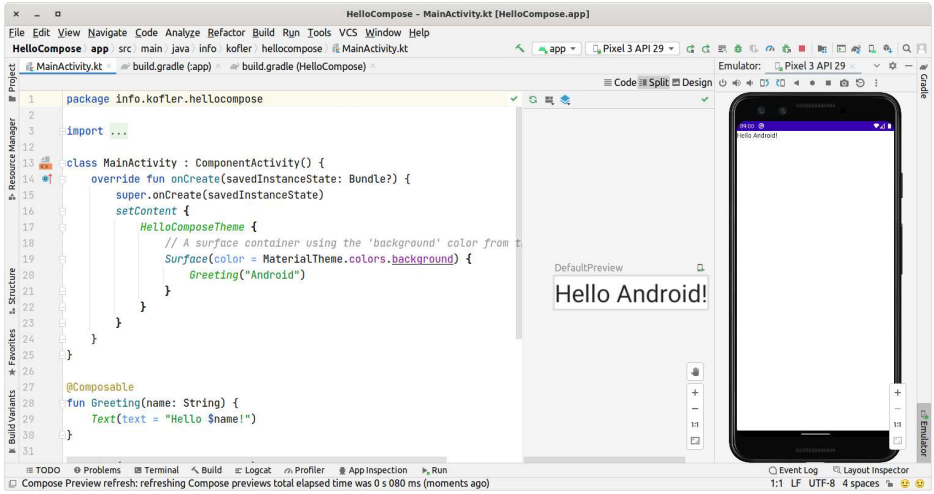

Abbildung 1.1 Links der Code, in der Mitte die Preview-Ansicht, rechts der Emulator

Groß- und Kleinschreibung

Normalerweise ist es in Kotlin üblich, dass die Namen von Funktionen und Methoden von Verben abgeleitet sind und mit einem kleinen Anfangsbuchstaben beginnen (`print`, `saveDocument` etc.).

Vollkommen anders sieht die Namensbildung von Composable-Funktionen aus: Namensgebend sind zumeist Substantive (`Column`, `Text`, `MyWidget` etc.), und der Anfangsbuchstabe ist groß.

Ich habe dazu keine offiziellen Richtlinien gefunden, aber die Konvention ist wohl darin begründet, dass die Funktionen Komponenten einer Oberfläche beschreiben, die in der Folge ähnlich wie Klassen eingesetzt werden.

Projektaufbau

Grundsätzlich folgt der Aufbau des Projekts denselben Regeln wie bei einem herkömmlichen Android-Projekt (siehe auch Abschnitt 21.5). Bei Aktivitäten bzw. Fragmenten, die Sie durch Compose-Funktionen zusammensetzen, entfällt aber die XML-Layout-Datei.

Dafür gibt es in Compose-Projekten standardmäßig vier zusätzliche Dateien `ui/Color.kt`, `ui/Shape.kt`, `ui/Theme.kt` und `ui/Type.kt` mit Voreinstellungen für Farben, Formen (z. B. abgerundete Rechtecke), Themen (Light/Dark Theme) und Textstilen. Diese Dateien geben ein Muster vor, wie Sie derartige Layouteinstellungen außerhalb

von Aktivitätsklassen definieren und so Ihren Code möglichst übersichtlich strukturieren (siehe auch Abschnitt 1.6, »Theming«).

Neu sind auch die beiden Dateien `res/values/themes/themes.xml` (eine »normale« Variante und eine mit spezifischen Einstellungen für den Dark Mode): Dort werden die in den `ui/*.kt`-Dateien durchgeführten Einstellungen zu Themen kombiniert (`<style name="Theme.<name>">`). Die Datei `AndroidManifest.xml` ordnet dann `Theme.<Projectname>.NoActionBar` der Hauptaktivität zu.

Vorsicht beim Import von Symbolen

Sobald Sie eigenen Code schreiben, müssen Sie alle möglichen Klassen, Funktionen usw. importieren. Eigentlich kein Problem, mit Alt+↵ kümmert sich Android Studio um den Rest. Passen Sie aber auf, dass Sie den richtigen Import wählen. Viele Symbolnamen (`Text`, `Button` usw.) sind sehr allgemeingültig und daher in mehreren Paketen enthalten. Wählen Sie nicht einfach das erste Paket in der Auswahlliste, sondern schauen Sie sich die zur Wahl stehenden Importe genau an. Richtig sind in der Regel die Importe, deren Paketname mit `androidx` beginnt.

Modifier

Alle Compose-Funktionen für Steuerelemente und Container sehen einen optionalen Parameter `modifier` vor. An diesen können Sie ein Objekt übergeben, das der Schnittstelle `Modifier` entspricht. `Modifier`-Objekte ermöglichen es, alle erdenklichen Formatierungs- und Layoutdetails von Steuerelementen und Containern zu steuern. Die im folgenden Beispiel verwendete Eigenschaft `.dp` wandelt eine ganze Zahl in *Density-independent Pixel* um (siehe auch Abschnitt 23.3, »Texte anzeigen und eingeben (TextView, EditText)«).

```
// rund um das Textfeld 4 Pixel frei lassen
// .dp = Display-independent Pixel
Text("Name:", modifier = Modifier.padding(4.dp))
```

Umfangreiche Modifier führen zu unübersichtlichem Code mit vielen Wiederholungen. Deswegen kann es zweckmäßig sein, den Modifier in einer Variable zwischenzuspeichern und diese Variable dann an verschiedenen Orten im Code zu verwenden:

```
val mymod = Modifier.requiredWidth(120.dp).padding(4.dp)
Text("Text 1", modifier = mymod)
Text("Text 2", modifier = mymod)
```

Um mehrere `Modifier` zu kombinieren, wenden Sie einfach mehrere Methoden hintereinander an. Lesbarer wird der Code, wenn Sie die Methoden zeilenweise untereinander anordnen:

```
val mymod = Modifier.requiredWidth(120.dp)
                    .padding(4.dp)
```

In der Compose-Bibliothek gibt es schier unendlich viele `Modifier`-Methoden. In diesem Kapitel kann ich nur die allerwichtigsten vorstellen. Eine umfassende Referenz finden Sie hier:

https://developer.android.com/reference/kotlin/androidx/compose/ui/Modifier

Wenn Sie eigene Compose-Funktionen entwickeln, kann es zweckmäßig sein, einen optionalen Modifier-Parameter vorzusehen. Damit können Sie beim Aufruf der Funktion weitere Einstellungen übergeben, diese dann aber innerhalb der Funktion weiter variieren. Viele vordefinierte Compose-Funktionen sind nach demselben Muster konzipiert:

```
@Composable
fun MyControl(modifier: Modifier = Modifier) {
    Text(..., modifier = modifier.padding(4.dp))
}
```

Datenverarbeitung (»MutableState« und »remember«)

Bei herkömmlichen Android-Apps erfolgt der Zugriff auf den Inhalt eines Steuerelements einfach über den Namen des Elements und dessen Eigenschaften, also z. B. `mylabel.text`. In Jetpack Compose ist diesbezüglich alles anders: Durch Compose-Funktionen ausgedrückte Steuerelemente haben keinen Namen – was einen Zugriff von außen unmöglich macht.

Generell versucht Jetpack Compose, die Trennung zwischen den Daten und deren Darstellung zu eliminieren. Google präsentiert dieses neue, deklarative Paradigma als großen Vorteil von Jetpack Compose. Tatsächlich erfordert diese Vorgehensweise aber eine vollkommen neue Konzeption des eigenen Codes.

Von zentraler Bedeutung sind dabei `MutableState`-Variablen. Der Zweck derartiger Variablen besteht darin, Änderungen zu verfolgen und dann eine Aktualisierung des Steuerelements (exakter: der zugeordneten Compose-Funktion) zu veranlassen. Dieser Vorgang wird auch *Recomposition* genannt (siehe auch den folgenden Kasten). Der Zugriff auf den Inhalt der Variable erfolgt über die Eigenschaft `value`.

Sie deklarieren `MutableState`-Variablen normalerweise mit der Compose-Funktion `remember`. Der Lambda-Ausdruck von `remember` wird einmalig beim Zusammensetzen des Compose-Ausdrucks, also bei der *Composition*, ausgewertet. In der Folge, also bei einer *Recomposition*, bleibt der bereits vorhandene Wert erhalten – daher die Bezeichnung *remember*.

```
// data1 hat den Typ MutableState<Boolean>,
// data2 den Typ MutableState<String>.
val data1 = remember { mutableStateOf(true) }
val data2 = remember { mutableStateOf("abc") }
println(data2.value)  // Ausgabe: abc
```

Composition versus Recomposition

Eine typische Oberfläche einer App setzt sich aus diversen ineinander verschachtelten Compose-Funktionen zusammen. Der erstmalige Aufbau der Oberfläche heißt *Composition*.

Sobald sich während der Laufzeit der App Daten ändern, muss die Oberfläche entsprechend angepasst werden. Der Inhalt, die Position und die Sichtbarkeit von Steuerelementen können sich dabei ändern. Aus Effizienzgründen versucht das Compose-Framework, möglichst zeit- und ressourcensparend vorzugehen. Es werden also nicht einfach sämtliche Compose-Funktionen komplett neu ausgeführt, vielmehr kommt es nur zu einem teilweisen Update. Dieser Vorgang wird *Recomposition* genannt.

Bemerkenswert am (Re-)Composition-Vorgang ist, dass der Oberflächen(neu)aufbau asynchron und nicht in einer nicht vorhersehbaren Reihenfolge vor sich geht. Sie dürfen sich also nicht darauf verlassen, dass mehrere Compose-Funktionen in der gleichen Abfolge wie in Ihrem Code ausgeführt werden! Mehr Hintergründe zum Recomposition-Konzept finden Sie hier:

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

`remember` kann auch ohne `MutableState` verwendet werden, um einen in der Folge unveränderlichen Wert einmalig zu initialisieren und zu speichern:

```
// rndValue hat den Typ Int.
val rndValue = remember { (1..100).random() }
```

Eine weitere Anwendungsform von `remember` erlaubt die Übergabe eines Parameters. Eine Neuauswertung des Lambda-Ausdrucks erfolgt in diesem Fall nur, wenn sich der Inhalt des Parameters im Vergleich zum letzten Aufruf verändert hat:

```
// Der Typ von mydata ergibt sich durch den Rückgabedatentyp
// von timeConsumingCalc().
val mydata = remember(someInput) {
    timeConsumingCalc(someInput)
}
```

In diesem Fall geht es nicht nur darum, dass sich Compose den Inhalt von `mydata` merken soll; gleichzeitig soll eine zeitaufwendige Verarbeitung eines Parameters wirklich nur dann durchgeführt werden, wenn diese notwendig ist.

Klick-Zähler

`MutableState`-Variablen können mit dem Inhalt von Compose-Steuerelementen verbunden und unabhängig davon verändert werden. Das folgende Beispiel zeigt eine ganz simple Anwendung: Jedes Mal, wenn Sie auf einen Button klicken, soll ein Zähler um 1 vergrößert werden. Der Inhalt des Zählers soll in einem Textfeld angezeigt werden (siehe Abbildung 1.2). Der erforderliche Code sieht so aus:

```
// Projekt HelloCompose, Datei MainActivity.kt
@Composable
fun ButtonCounter() {
    val cnt = remember { mutableStateOf(0) }
    Column {
        Button(content = { Text("Weiterzählen") },
               onClick = { cnt.value += 1 })
        Text("Der Button wurde ${cnt.value}-mal gedrückt.")
    }
}
```

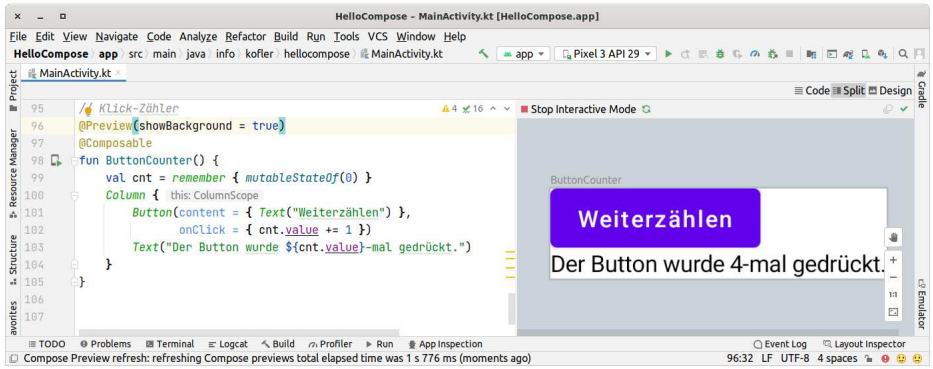

Abbildung 1.2 Klick-Zähler mit einer »MutableState«-Variablen

Bei der Deklaration von Variablen kann `remember` mit dem Schlüsselwort `by` kombiniert werden. Dabei wird der Zugriff auf den Inhalt delegiert (siehe auch Abschnitt 13.13, »Delegation«). Der resultierende Code wird damit noch eleganter, weil der Zugriff auf den Inhalt des `MutableState`-Objekts jetzt ohne `value` erfolgen kann. Beachten Sie aber, dass die Variable nun mit `var` statt mit `val` deklariert werden muss (was ohnedies logischer scheint). Außerdem müssen Sie die folgenden beiden Importe manuell hinzufügen.

```
import androidx.compose.runtime.getValue
import androidx.compose.runtime.setValue
```

```
// ButtonCounter-Variante mit Delegation
@Composable
fun ButtonCounter() {
    var cnt by remember { mutableStateOf(0) }
    Column {
        Button(content = { Text("Weiterzählen") },
               onClick = { cnt += 1 })
        Text("Der Button wurde ${cnt}-mal gedrückt.")
    }
}
```

Laut der Compose-Dokumentation führen beide Schreibweisen (also mit oder ohne `by`) intern zum gleichen Code. Es handelt sich also lediglich um »syntaktischen Zucker«:

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

Im Verlauf dieses Kapitels folgen noch einige weiterer Beispiele für den Einsatz von `MutableState`-Variablen.

> **Initialisierung nur in Composable-Funktionen**
>
> `remember` kann nur in Funktionen aufgerufen werden, die mit `@Composable` gekennzeichnet sind. Manchmal besteht die Notwendigkeit, solche Variablen in der Aktivitätsklasse zu definieren. In diesem Fall deklarieren Sie die Variable mit `lateinit var` und führen die Initialisierung in der `setContent`-Funktion innerhalb von `onCreate` durch. Ein konkretes Beispiel finden Sie in Abschnitt 1.8, »Beispiel: Fahrenheit-Umrechner«.

Status von Steuerelementen erhalten (»rememberSaveable«)

Wenn Sie in den Display-Einstellungen Ihres Smartphones den Dark Mode aktivieren bzw. wieder deaktivieren oder wenn Sie Ihr Gerät in das Querformat drehen, wird (wie bei gewöhnlichen Apps) die gerade aktuelle Aktivität neu erzeugt. Dabei verlieren sämtliche Steuerelemente mit Eingabemöglichkeit ihren Zustand. Ein `TextField` ist nach der Drehung also leer, der vorhin präsentierte Klick-Zähler springt zurück auf 0.

Auch `remember` hilft hier nichts, weil es nicht zu einer *Recomposition* kommt, sondern zu einem kompletten Neuaufbau der Oberfläche – wenn Sie so wollen also zu einem totalen Reset. Zum Glück gibt es eine einfache Lösung: Zum Speichern des Status verwenden Sie anstelle von `remember` die Funktion `rememberSaveable`:

```
// erhält den Status des Klick-Zählers auch bei einer
var cnt by rememberSaveable { mutableStateOf(0) }
```

So deklarierte Variablen behalten Ihren Wert auch dann, wenn eine Aktivität (letztlich ein Objekt einer von `ComponentActivity` abgeleitete Klasse) neu erzeugt wird. Noch mehr Optionen, um den Status von Apps zu verwalten, beschreibt die Compose-Dokumentation:

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

Der Statuserhalt gilt allerdings nur, während die App läuft. Wird sie neu gestartet, werden alle Werte bzw. Steuerelementinhalte weiterhin zurückgesetzt. Um Einstellungen und andere Daten dauerhaft zu speichern, können Sie z. B. auf die `SharedPreferences` zurückgreifen (siehe Abschnitt 23.8, »Preferences«).

Die Preview-Ansicht

Eine Besonderheit der Integration von Jetpack Compose mit Android Studio besteht darin, dass Sie im Code mehrere Compose-Funktion mit der Annotation `@Preview` kennzeichnen dürfen:

```
@Preview
@Composable
fun ButtonCounter() { ... }
```

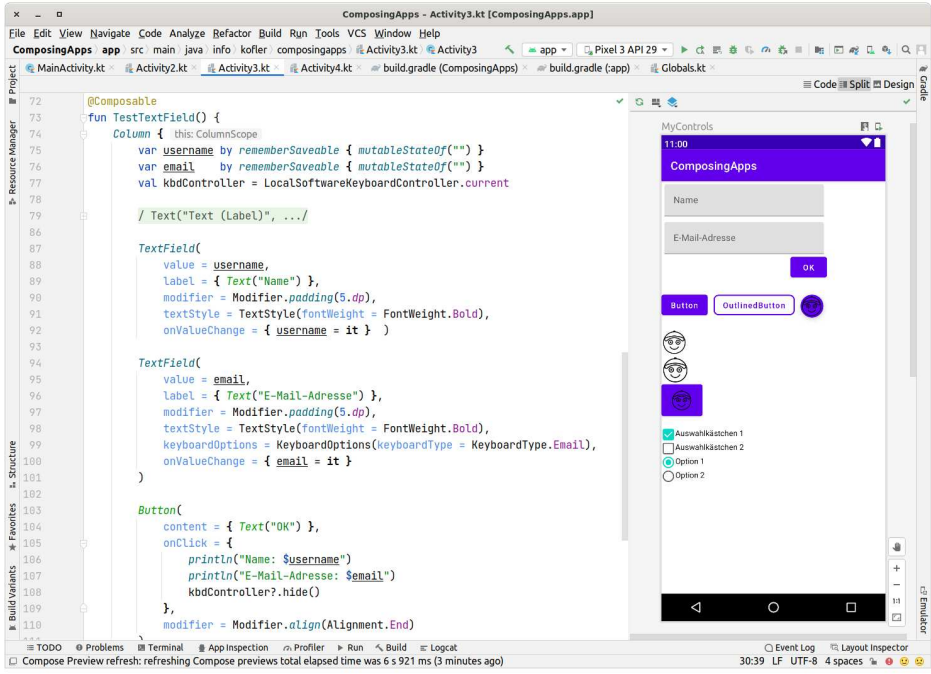

Abbildung 1.3 Die Vorschau einer Compose-Funktion zum Test diverser Steuerelemente

Damit erreichen Sie, dass die durch die Funktion ausgedrückte Komponente in der Preview-Ansicht angezeigt wird (siehe Abbildung 1.3). Die Voransicht enthält mehr Features, als auf den ersten Blick zu erkennen sind. So bewirkt ein Klick in die Preview-Ansicht, dass der Cursor im zugeordneten Code-Fenster an der Stelle platziert wird, wo die Compose-Funktion des Steuerelements aufgerufen wird. Damit hilft die Preview-Ansicht bei einer besonders effizienten Navigation im Code.

Mit dem Button INTERACTIVE können Sie eine Komponente auswählen und interaktiv benutzen. Wenn Sie dabei auf einen Button klicken, wird die entsprechende Aktion durchgeführt (siehe Abbildung 1.2). Beachten Sie, dass der interaktive Test einzelner Komponenten noch nicht ausgereift ist und bei meinen Tests mitunter Probleme verursachte. Das Feature muss in den Einstellungen im Dialogblatt EXPERIMENTAL explizit aktiviert werden.

Noch einen Schritt weiter geht DEPLOY VIEW: Es führt die ausgewählte Komponente (nicht die ganze App) auf dem angeschlossenen Smartphone oder im Emulator aus. Das ermöglicht noch realitätsnähere Tests.

Mit optionalen Parametern der `Preview`-Annotation können Sie das Erscheinungsbild der Vorschau steuern:

- `name` legt fest, wie Android Studio das Vorschauelement benennt. Standardmäßig verwendet Android Studio den Funktionsnamen.
- `heightDp` und `widthDp` legen die maximale Höhe und Breite der Vorschau in der Einheit *Density-independent Pixel* fest (siehe auch Abschnitt 23.3, »Texte anzeigen und eingeben (TextView, EditText)«).
- `showBackground = true` bewirkt, dass die Vorschau die Defaulthintergrundfarbe verwendet. Ohne diese Option heben sich die Vorschauelemente nur schlecht vom Preview-Fenster ab.
- `showSystemUi = true` bettet die Vorschau in den Rahmen eines Smartphones ein (siehe Abbildung 1.3). Das kostet zwar viel Platz, vermittelt aber eine viel bessere Vorstellung von der Größe und den Platzverhältnissen in einem richtigen Gerät.

Wenn Sie je eine Vorschauvariante im Light und im Dark Mode wünschen, verpacken Sie Ihre Compose-Funktion in eine zweite Funktion mit `XxxTheme(darkTheme = true)`. Beachten Sie, dass Sie im folgenden Listing `HelloComposeTheme` durch `<IhrProjektname>Theme` ersetzen müssen!

```
// normale Preview
@Preview(showSystemUi = true)
@Composable
fun MyControls() { ...}
```

```
// Preview im Dark Mode
@Preview(showSystemUi = true)
@Composable
fun DarkControls() {
    HelloComposeTheme(darkTheme = true) {
        Surface {
            MyControl("Very dark ...")
        }
    }
}
```

Noch mehr Optionen und Varianten finden Sie im folgenden Video bzw. in der Android-Developer-Dokumentation:

https://www.youtube.com/watch?v=exjL2kGPngI
https://developer.android.com/jetpack/compose/tooling

Keine Vorschau für Funktionen mit Parametern

Wenn Ihre Funktion Parameter ohne Defaultwerte hat, können Sie die `Preview`-Annotation nicht verwenden – Android Studio weiß ja nicht, welche Werte es an die Parameter übergeben soll. Wenn Sie dennoch eine Vorschau wünschen, dann bauen Sie in Ihren Code einfach eine weitere Funktion ohne Parameter ein, die die ursprüngliche Funktion mit sinnvollen Parametern aufruft. Die neue, parameterfreie Funktion können Sie mit `@Preview` ausstatten.

1.3 Steuerelemente

In diesem Abschnitt gebe ich Ihnen einen Überblick über die wichtigsten Compose-Funktionen für Steuerelemente und Container. Container helfen dabei, mehrere Steuerelemente zu gruppieren.

Textanzeige

Die Funktion `Text` haben Sie im Hello-World-Beispiel bereits kennengelernt. Ihre Anwendung ist denkbar einfach: Sie übergeben den anzuzeigenden Text an den Parameter `text`:

```
Text(text = "mein Text")
```

Da `text` der erste von einer ganzen Menge von Parametern ist, können Sie auf seine Nennung beim Aufruf der Funktion verzichten:

```
Text("mein Text")
```

Die restlichen Parameter (`modifier`, `color` usw.) steuern das Aussehen des Texts.

Die Schriftgröße sowie die Schriftattribute steuern Sie mit optionalen Parametern wie `fontSize`, `fontStyle` und `style`:

- `fontSize` erwartet die Textgröße in *Scale-independent Pixel* (Eigenschaft `sp`).
- `fontStyle` akzeptiert nur `FontStyle.Italic` und `FontStyle.Normal`.
- An `style` müssen Sie ein `TextStyle`-Objekt übergeben, in dem Sie weitere Parameter verändern. Im folgenden Beispielcode wird eine fette Schrift eingestellt. Alternativ können Sie auch auf die Stile zurückgreifen, die über die Klasse `MaterialTheme` vordefiniert sind, z. B. auf `MaterialTheme.typography.h3` (für eine Überschrift dritter Ordnung, siehe auch Abschnitt 1.6, »Theming«).
- Mit `fontFamily` können Sie die gewünschte Schriftart auswählen.

```
Text("lorem ipsum",
     fontSize = 18.sp,
     fontStyle = FontStyle.Italic,
     style = TextStyle(fontWeight = FontWeight.Bold))
```

Wenn Sie in den Text `\n`-Zeichen einbauen oder wenn der Text zu lang ist, wird er über mehrere Zeilen verteilt. Mit dem Parameter `maxLines` limitieren Sie die maximale Zeilenanzahl.

Texteingabe

Wenn Sie Ihren App-Nutzern eine Eingabemöglichkeit geben möchten, verwenden Sie anstelle von `Text` die Compose-Funktion `TextField`. Das resultierende Steuerelement entspricht in seiner Funktion `EditText` (siehe Abschnitt 23.3, »Texte anzeigen und eingeben (TextView, EditText)«).

Damit Sie den Inhalt des Textfelds steuern können, müssen Sie es mit einer `MutableState`-Variable verbinden. Bei jeder Änderung des Texts wird der Zustand der Variable (und damit auch der angezeigte Text!) durch den Lambda-Ausdruck des Parameters `onValueChange` verändert. Der Parameter `label` beschriftet das Textfeld. Der Parameter muss angegeben werden, aber es kann Sie niemand hindern, einfach einen leeren Lambda-Ausdruck zu übergeben (also `label = {}`):

```
var username by rememberSaveable { mutableStateOf("") }
TextField(value = username.value,
          label = { Text("Name") },
          onValueChange = { username = it }  )
```

Mit den optionalen Parametern `modifier`, `textStyle` und `keyboardType` können Sie das Aussehen des Textfelds und der Tastatur beeinflussen (siehe Abbildung 1.4).

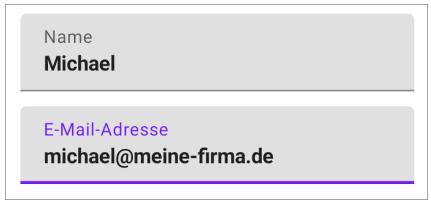

Abbildung 1.4 Zwei Texteingabefelder

```
// Projekt ComposingApps, Datei Activity3.kt
Column {
    var username by rememberSaveable { mutableStateOf("") }
    var email    by rememberSaveable { mutableStateOf("") }
    TextField(
        value = username,
        label = { Text("Name")},
        modifier = Modifier.padding(5.dp),
        textStyle = TextStyle(fontWeight = FontWeight.Bold),
        onValueChange = { username = it }  )
    TextField(
        value = email,
        label = { Text("E-Mail-Adresse")},
        modifier = Modifier.padding(5.dp),
        textStyle = TextStyle(fontWeight = FontWeight.Bold),
        keyboardOptions = KeyboardOptions(keyboardType =
    KeyboardType.Email),
        onValueChange = { email = it }  )
}
```

Ein ausführlicheres Beispiel zur Anwendung von `TextField` folgt in Abschnitt 1.8, »Beispiel: Fahrenheit-Umrechner«. Eine optische Variante zum `TextField` ist das `OutlinedTextField`: Es umgibt das Textfeld mit einem beschrifteten Rahmen.

Tastatur ausblenden

Bei Eingaben wird automatisch die Tastatur eingeblendet. Wie bei der herkömmlichen Android-Programmierung müssen Sie sich selbst darum kümmern, die Tastatur wieder auszublenden (siehe Abschnitt 23.3, »Texte anzeigen und eingeben (TextView, EditText)«). Jetpack Compose sieht zur Steuerung der Tastatur das Objekt `LocalSoftwareKeyboardController` vor. `current` kann allerdings auch `null` zurückgeben, wenn es im aktuellen Kontext keine Tastatur gibt. Mit `hide` können Sie nun die Tastatur ausblenden. Die Schnittstelle gilt allerdings noch als experimentell, dementsprechend muss der Code mit dem der Annotation `ExperimentalComposeUiApi` gekennzeichnet werden.

```
// Projekt ComposingApps, Datei Activity3.kt
@ExperimentalComposeUiApi
@Composable
fun TestTextField() {
    val kbdController = LocalSoftwareKeyboardController.current
    ...
    Button(content = { Text("OK") },
           onClick = { kbdController?.hide() } )
    ...
}
```

Bilder

Mit der Funktion `Image` stellen Sie Bilder dar. In der einfachsten Form übergeben Sie als Parameter die Funktion `painterResource` und verweisen auf eine Bitmap, die Sie zuvor in den Ressource Manager von Android Studio eingefügt haben.

Außerdem müssen Sie mit `contentDescription` eine Beschreibung des Bilds festlegen. Der Text ist als Hilfestellung für die *Accessibility Services* gedacht. Die Übergabe von `null` ist nur in Fällen vorgesehen, bei denen das Bild ausschließlich dekorative Zwecke erfüllt, also beispielsweise ein Hintergrundmuster enthält.

```
Image(painter = painterResource(id = R.drawable.<name>),
      contentDescription = "Beschreibung des Bilds")
```

Das Bild wird dann in der Originalgröße angezeigt. Um die Größe zu beeinflussen, übergeben Sie an den `modifier`-Parameter die gewünschte Größe und geben mit `contentScale` an, ob bzw. wie das Bild skaliert, angepasst oder abgeschnitten werden soll:

```
Image(painterResource(R.drawable.emo_cap),
      // 50x50 Display Independent Pixel
      modifier = Modifier.requiredSize(50.dp),
      contentScale = ContentScale.Fit,
      contentDescription = "Smiley-Icon")
```

Wenn Sie Klick-Ereignisse verarbeiten möchten, geben Sie einen zusätzlichen `clickable`-Modifier sowie einen Lambda-Ausdruck für das Ereignis an. Diese Vorgehensweise ist für sämtliche Compose-Elemente zulässig.

```
Image(...,
      modifier = Modifier
          .clickable { println("Klick") } )
```

Buttons

Die Compose-Funktion `Button` erzeugt einen anklickbaren Container. An den Parameter `content` übergeben Sie in der Regel einfach ein `Text`-Element mit der Beschriftung des Buttons. An dieser Stelle ist aber auch jede andere Compose-Funktion erlaubt – z. B. `Image`, um einen Button mit grafischen Elementen zu erzeugen.

```
// Text-Button
Button(content = { Text("OK") },
       onClick = {
           println("Name: $username")
           println("E-Mail-Adresse: $email")
       })
// Image-Button
Button(content = {
           Image(painterResource(R.drawable.emo_cap),
               modifier = Modifier.requiredSize(40.dp),
               contentScale = ContentScale.Fit,
               contentDescription = "Smiley") },
       onClick = { println("Klick") })
```

In vielen Beispielen im Internet scheint der `content`-Parameter zu fehlen. Stattdessen wird der Inhalt des Buttons in einem nachgestellten Lambda-Ausdruck formuliert. Ich habe versucht, das in meinen Beispielen zu vermeiden, weil die Lesbarkeit darunter stark leidet. Die wichtigste Information, nämlich der Inhalt des Buttons, folgt dann ganz zum Schluss:

```
Button(onClick = {
           println("Name: $username")
           println("E-Mail-Adresse: $email") })
    { Text("OK") }
```

Jetpack Compose stellt diverse Button-Varianten zur Wahl. Die Funktion `OutlinedButton` erzeugt einen umrahmten Button, der `FloatingActionButton` mit einem `Icon` als Content einen runden Action Button (siehe Abbildung 1.5). Die folgenden Zeilen zeigen Anwendungsbeispiele für beide Funktionen:

```
OutlinedButton(
    content = { Text("OutlinedButton") },
    onClick = { },
    border = BorderStroke(2.dp, MaterialTheme.colors.primary),
    shape = RoundedCornerShape(25))
```

```
FloatingActionButton(
    content = { Icon(painter = painterResource(
                                id = R.drawable.emo_cap),
                     modifier = Modifier.background(
                                MaterialTheme.colors.primary),
                     contentDescription = "Smiley") },
    onClick = { },
    modifier = Modifier.requiredSize(40.dp))
```

Abbildung 1.5 Ein normaler Button, ein »OutlinedButton« und ein »FloatingActionButton«

Optionsfelder und Auswahlkästchen (»RadioButton«, »Checkbox«)

Die Funktionen `Checkbox` und `RadioButton` dienen dazu, winzige Auswahlkästchen bzw. Options-Buttons darzustellen. Beide Steuerelemente sehen keine Möglichkeit vor, einen Beschriftungstext anzugeben. Den Text müssen Sie getrennt mit `Text` festlegen. Das wäre nicht weiter schlimm, hat aber die unangenehme Konsequenz, dass nur die Buttons selbst auf eine Berührung reagieren, nicht aber der dazugehörende Text. Das mindert die Bedienbarkeit stark. Abhilfe schafft ein zusätzlicher Modifier für den Text, um auch dort Klicks zu verarbeiten. Der resultierende Code wird dadurch umständlich und redundant.

Der aktuelle Zustand muss in `MutableState`-Variablen gespeichert und in den Lambda-Ausdrücken der Parameter `onCheckedChanged` bzw. `onSelect` verändert werden. Die folgenden Zeilen zeigen den erforderlichen Code für zwei voneinander unabhängige Auswahlkästchen bzw. für eine Gruppe aus zwei Radio-Buttons (siehe Abbildung 1.6):

```
// Projekt ComposingApps, Datei Activity3.kt
// Datentyp je MutableState<Boolean>
var check1 by rememberSaveable { mutableStateOf(true)  }
var check2 by rememberSaveable { mutableStateOf(false) }
var radio1 by rememberSaveable { mutableStateOf(true)  }
var radio2 by rememberSaveable { mutableStateOf(false) }

Column {
    Row {
        Checkbox(checked = check1,
                 onCheckedChange = { check1 = it })
        Text("Auswahlkästchen 1",
            Modifier.clickable { check1 = !check1 } )
    }
```

```
    Row {
        Checkbox(checked = check2,
                 onCheckedChange = { check2 = it })
        Text("Auswahlkästchen 2",
            Modifier.clickable { check2 = !check2 } )
    }
    Row {
        RadioButton(selected = radio1,
                    onClick = {
                        radio1 = true
                        radio2 = false
                    })
        Text("Option 1", Modifier.clickable {
                              radio1 = true
                              radio2 = false} )
    }
    Row {
        RadioButton(selected = radio2,
                    onClick = {
                        radio1 = false
                        radio2 = true
                    })
        Text("Option 2", Modifier.clickable {
                              radio1 = false
                              radio2 = true} )
    }
}
```

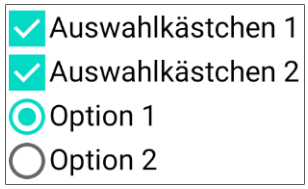

Abbildung 1.6 Auswahlkästchen und Radio-Buttons

1.4 Container

In den Beispielen des vorigen Abschnitts sind immer wieder die Compose-Funktionen `Column` und `Row` vorgekommen. Es handelt sich dabei um die beiden wichtigsten Funktionen, um mehrere Steuerelemente zu gruppieren und anzuordnen. In diesem Abschnitt zeige ich Ihnen verschiedene Anwendungsformen von `Column` und `Row` und stelle Ihnen einige weitere Container vor – unter anderem zur Darstellung von Listen.

Mehrere Steuerelemente platzieren (»Column« und »Row«)

Mit `Column` ordnen Sie mehrere Steuerelemente untereinander an:

```
Column {
    Control1(...)
    Control2(...)
}
```

Analog hilft `Row` dabei, mehrere Steuerelemente nebeneinander zu platzieren:

```
Row {
    for (txt in listOf("lorem", "ipsum", "dolores"))
        Text(txt)
}
```

Beide Container sehen sowohl nach außen als auch zwischen den Steuerelementen keine Abstände vor. Abhilfe schafft die Verwendung von `padding`-Modifiern:

```
// Abstand des Containers nach außen
Row(Modifier.padding(10.dp)) {
    for (txt in listOf("lorem", "ipsum", "dolores"))
        // Abstand der Textfelder zueinander
        Text(txt, modifier = Modifier.padding(10.dp))
}
```

Bei `Row` und `Column` ist `modifier` der erste Parameter. Deswegen ist es erlaubt, auf die Nennung des Parameters zu verzichten und direkt `Modifier.xxx` zu übergeben.

Rechteck (»Box«)

Zum Experimentieren mit `Row` und `Column` eignen sich `Box`-Objekte gut. Eine einfache Box können Sie so zusammensetzen:

```
Box(Modifier.requiredSize(40.dp, 60.dp)
            .background(Color.Red))
```

In den folgenden Beispielen verwende ich die Funktion `Quad`, die eine quadratische Box mit einer vorgegebenen Größe von 40 × 40 Pixel erzeugt (`requiredSize`), wobei die Farbe sowie zusätzliche Modifier als Parameter übergeben werden können:

```
// Projekt ComposingApps, Datei Activity2.kt
@Composable
fun Quad(color: Color = Color.Black,
         modifier: Modifier = Modifier)
{
    Box(modifier.requiredSize(40.dp).background(color))
}
```

Abbildung 1.7 Anordnung von Elementen in der Preview-Ansicht von Android Studio

Abstandhalter (»Spacer«) und Trennlinien (»Divider«)

Alternativ zum `padding`-Modifier können Sie zwischen den Elementen einer Zeile oder Spalte mit `Spacer` einen Platzhalter einbauen. Außerdem können Sie mit einem `Divider` eine dünne, horizontale Trennlinie zeichnen. (Beachten Sie, dass Sie den `Spacer` in Zeilen und Spalten einsetzen können, den `Divider` aber nur in Spalten!)

```
// Projekt ComposingApps, Datei Globals.kt
@Composable
fun DivideAndSpace() {
    Column {
        Quad(Color.Red)
        Spacer(Modifier.requiredHeight(10.dp))
        Divider(color = Color.Black, thickness = 2.dp)
        Spacer(Modifier.requiredHeight(10.dp))
```

```
        Row {
            Quad(Color.Blue)
            Spacer(Modifier.requiredHeight(10.dp))
            Quad(Color.Green)
        }
    }
}
```

Anordnung von Elementen in Zeilen und Spalten

Standardmäßig ist ein `Row`- oder `Column`-Container exakt so groß, wie die darin enthaltenen Elemente Platz beanspruchen. Die Steuerelemente werden linksbündig untereinander bzw. an ihrer oberen Kante nebeneinander platziert. Diese simplen Regeln sind selten ausreichend. Stattdessen wollen Sie die Elemente zentrieren, links- oder rechtsbündig ausrichten etc. Dazu gibt es diverse Möglichkeiten, von denen ich Ihnen hier einige vorstelle.

Das erste Ziel besteht darin, die gesamte Breite eines Smartphone-Bildschirms zu nutzen und je ein Element links- bzw. rechtsbündig anzuordnen. Der einfachste Weg besteht darin, zwischen den Elementen einen `Spacer` einzubauen. Entscheidend ist dabei der `weight`-Modifier: Der erste Parameter gibt an, wie viel Platz das Element relativ zu anderen Elementen mit einer `weight`-Angabe haben soll. Da bei den beiden Boxen keine Gewichtung angegeben ist, spielt der Wert `1F` keine Rolle, solange er nicht `0` ist. Implizit gilt aber für den optionalen zweiten Parameter `fill = true`. Damit beansprucht der `Spacer` so viel Platz wie möglich.

```
// Projekt ComposingApps, Datei Activity2.kt
fun TestWeight() {
    Row { Quad(Color.Blue)
          Spacer(Modifier.weight(1F))
          Quad(Color.Green) }
    ...
```

Im nächsten Beispiel sollen drei Elemente den zur Verfügung stehenden Platz anteilsmäßig nutzen.

```
    Row {
        Box(Modifier.background(Color.Blue)       // 16,7 %
              .requiredHeight(40.dp).weight(1F))
        Box(Modifier.background(Color.Red)        // 50 %
              .requiredHeight(40.dp).weight(2F))
        Box(Modifier.background(Color.Green)      // 33,3 %
              .requiredHeight(40.dp).weight(3F))
        }
    ...
}  // Ende fun TestWeight
```

Um mehrere Elemente gleichmäßig über den gesamten zur Verfügung stehenden Raum zu verteilen, verwenden Sie bei einer `Row` den Modifier `fillMaxWidth` und stellen den optionalen Parameter `horizontalArrangement` mit `Arrangement.SpaceBetween` bzw. `Arrangement.Evenly` ein. Im ersten Fall wird nur zwischen den Elementen ein Leerraum eingebaut, im zweiten Fall auch vor und nach dem letzten Element (siehe Abbildung 1.7). Analoge Einstellmöglichkeiten gibt es auch für `Column`, wobei der Parameter dort `verticalArrangement` lautet.

```
fun TestArrangement() {
    Row(Modifier.fillMaxWidth(),
        horizontalArrangement = Arrangement.SpaceBetween)
    {
        Quad(Color.Blue)
        Quad(Color.Gray)
        Quad(Color.Cyan)
    }
    Spacer(Modifier.preferredHeight(10.dp))
    Row(Modifier.fillMaxWidth(),
        horizontalArrangement = Arrangement.SpaceEvenly)
    {
        Quad(Color.Blue)
        Quad(Color.Gray)
        Quad(Color.Cyan)
    }
}
```

Um Elemente in einer Spalte zu zentrieren, stellen Sie den Parameter `horizontalGravity` auf `CenterHorizontally`:

```
fun TestGravity() {
    Column(horizontalGravity = Alignment.CenterHorizontally) {
        for (txt in listOf("Lorem", "ipsum",
                           "dolores", "est"))
            Text(txt)
    }
}
```

Steuerelemente übereinander zeichnen (»Box«)

Eine Alternative zu `Row` und `Column` ist das Element `Box`. Der wesentliche Unterschied besteht darin, dass `Box` eine Überlappung der Elemente erlaubt. Die Elemente werden in Reihenfolge übereinander gezeichnet, die im Lambda-Ausdruck vorgegeben ist (siehe den unteren Teil von Abbildung 1.7). Die Platzierung erfolgt durch `align`-Modifier.

```
// Projekt ComposingApps, Datei Activity2.kt
fun TestBox() {
    Box {
        Text("Text 1 wird zuerst ausgegeben.",
            Modifier.align(Alignment.TopCenter))
        Box(Modifier.align(Alignment.TopStart)
            .fillMaxHeight()
            .requiredWidth(50.dp)
            .background(Color.Yellow))
        Box(Modifier.align(Alignment.BottomEnd)
            .requiredSize(40.dp)
            .background(Color.Red))
        Text("Text 2 wird später ausgegeben.",
            Modifier.align(Alignment.Center))
    }
}
```

1.5 Listen

Um mehrere Elemente in einer verschiebbaren Liste unterzubringen, verwenden Sie am einfachsten die Compose-Funktionen `LazyColumn` bzw. `LazyRow`. Die Funktionsweise dieser Steuerelemente ist ähnlich wie beim `RecyclerView` (siehe in Abschnitt 23.7, »Listen und Tabellen (RecyclerView)«). `Lazy` bezieht sich darauf, dass nur tatsächlich sichtbare Listenelemente initialisiert und gezeichnet werden.

Die Listenelemente können Sie (müssen Sie aber nicht) mit der Funktion `Card` verpacken. Sie hilft bei der optischen Gestaltung von Listeneinträgen. Das folgende Beispiel verwendet dieselbe Klasse `Country`, die ich Ihnen bereits in Abschnitt 23.7, »Listen und Tabellen (RecyclerView)«, präsentiert habe. Sie speichert die Daten eines Bundeslands und initialisiert mit `initFromAssets` eine Liste aus der Asset-Datei `bundesländer.txt`.

Die Funktion `TestLazyColumn` durchläuft in einer Schleife alle Bundesländer und setzt eine Liste aus `CountryItems` zusammen. Die Compose-Funktion `CountryItem` stellt ein `Country`-Objekt dar, wobei die Flagge, der Name des Bundeslands und der Name der Hauptstadt angezeigt werden (siehe Abbildung 1.8). Der Zugriff auf die Ressourcen erfolgt über ein `Context`-Objekt (`LocalContext.current`).

```
// Projekt ComposingApps, Datei Activity4.kt
lateinit var countries : List<Country>

override fun onCreate(savedInstanceState: Bundle?) {
    countries = Country.initFromAssets(assets)
    ...
}
```

```
@Composable
fun TestLazyColumn() {
    LazyColumn {
        item {
            countries.forEach {
                CountryItem(it)
                // Divider()  // optionale Trennlinie zwischen
                              // den Listeneinträgen
            }
        }
    }
}

@Composable
fun CountryItem(country: Country) {
    val context = LocalContext.current
    Card(Modifier.fillMaxWidth()
            .padding(8.dp)
            .clickable
                { println("Klick auf ${country.name}") },
        shape = RoundedCornerShape(8.dp),
        backgroundColor = Color.LightGray)
    {
        Row {
            val rscname = context.packageName + ":drawable/" +
                country.name
                    .lowercase(Locale.GERMAN)
                    .replace("ü", "u")
                    .replace("-", "_")
            val id = context.resources
                .getIdentifier(rscname, null, null)
            Image(painter = painterResource(id),
                modifier = Modifier.requiredSize(60.dp, 40.dp)
                    .padding(4.dp),
                contentScale = ContentScale.Fit,
                contentDescription = "Country Flag"
            )
            Spacer(Modifier.requiredWidth(10.dp))
            Column {
                Text(country.name,
                    Modifier.padding(4.dp),
                    style = TextStyle(fontWeight =
                        FontWeight.Bold),
                    fontSize = 18.sp)
```

```
                Text(country.capital,
                    Modifier.padding(4.dp),
                    fontSize = 14.sp)
            }
        }
    }
}
```

Abbildung 1.8 Beispiel für die Darstellung von Listen

1.6 Theming

Bei der Entwicklung von Apps mit Jetpack Composing sollten Sie sich bemühen, möglichst keine fixen Farben und Größen einzustellen und stattdessen auf Eigenschaften des `MaterialTheme` zurückzugreifen. Das hat den Vorteil, dass Ihre Apps den Layoutvorgaben des Smartphones (Schriftgröße, Dark Mode etc.) folgen – und das mit minimalem Programmieraufwand.

```
// vermeiden Sie starre Einstellungen, ...
TextField(value = ..., label = ...,
          onValueChange = { ...},
          textStyle = TextStyle(fontSize = 26.sp),
          activeColor = Color.Black)

// ... verwenden Sie stattdessen das MaterialTheme
TextField(value = ..., label = ...,
          onValueChange = { ...},
          textStyle = MaterialTheme.typography.h4,
          activeColor = MaterialTheme.colors.onBackground)
```

Farben

Welche Farben, Schriften, Formen usw. Ihre App verwendet, definieren Sie in den Dateien `ui/*.kt`. Beispielsweise enthält der Template-Code für Compose-Apps in `ui/Color.kt` und `ui/Theme.kt` bereits einige Farbeinstellungen für den Standardmodus und den Dark Mode. Sie können dort Einstellungen ändern bzw. hinzufügen:

```
// Projekt Fahrenheit, Datei ui/Colors.kt
val Purple200 = Color(0xFFBB86FC)
val Purple500 = Color(0xFF6200EE)
val Purple700 = Color(0xFF3700B3)
val Teal200 = Color(0xFF03DAC5)

// Datei ui/Theme.kt
// Palette für den Dark Mode
private val DarkColorPalette = darkColors(
    primary = Purple200,
    primaryVariant = Purple700,
    secondary = Teal200,
    /* weitere Standardfarben ... */
)
private val LightColorPalette = lightColors(
    primary = Purple500,
    primaryVariant = Purple700,
    secondary = Teal200
    /* weitere Standardfarben
    background = Color.White,
    surface = Color.White,
    onPrimary = Color.White,
    onSecondary = Color.Black,
    onBackground = Color.Black,
    onSurface = Color.Black,
    */
)
```

`primary` ist die Hauptfarbe der App. `primaryVariant` und `secondary` definieren dazu zwei Varianten. `background` und `surface` definieren die Hintergrundfarben für große Flächen. `onBackground`, `onSurface` usw. legen fest, welche Farben Text oder andere grafische Elemente auf dem entsprechenden Hintergrund nutzen sollen. Eine detaillierte Erläuterung der Farbnomenklatur des *Material Designs* finden Sie hier:

https://material.io/design/color/the-color-system.html

Vergessen Sie nicht, dass Sie den Containern und Steuerelementen explizit Farben aus der Palette zuweisen müssen (also `MaterialTheme.colors.onBackground` usw.) – sonst kommen Defaultvorgaben zur Anwendung, die nicht immer zweckmäßig sind.

Schriften

Das Material Design sieht eine ganze Reihe von Standardschriften vor (Headline 1 bis 6, Subtitle 1 und 2, Body 1 und 2, Button, Caption, Overline):

https://material.io/design/typography/the-type-system.html

Wenn Sie App-weite Änderungen an diesen Standardschriften vornehmen möchten, ist die Datei `ui/Type.kt` der richtige Ort. Das Compose-Template enthält gleich ein Muster, wie Sie vorgehen müssen:

```
// Datei ui/Type.kt
val Typography = Typography(
    body1 = TextStyle(fontFamily = FontFamily.Default,
                      fontWeight = FontWeight.Normal,
                      fontSize = 16.sp)
)
```

Wenn Sie im Programm an einer Stelle eine Variante zu einer vorgegebenen Schrift benötigen, erstellen Sie eine modifizierte Kopie:

```
val redItalicBody = MaterialTheme.typography.body1.copy(
   color = Color.Red,
   fontStyle = FontStyle.Italic)
```

Anwendung des Themes

Ebenfalls in `ui/Theme.kt` ist das Theme Ihrer App definiert. Der Funktionsname lautet `<Appname>Theme`:

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

```
    MaterialTheme(colors = colors,
                  typography = Typography,
                  shapes = Shapes,
                  content = content)
}
```

Die Funktion `onCreate` ist dafür zuständig, dass dieses Theme tatsächlich zur Anwendung kommt. Anstatt in `setContent` direkt Container und Steuerelemente anzugeben, werden diese als Lambda-Ausdruck an die Funktion `<Appname>Theme` übergeben:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContent {
        // Schriften, Fonts etc. durch Themes steuern
        FahrenheitTheme {
            // Container + Steuerelemente der Oberfläche
        }
    }
}
```

Wenn Sie die Hintergrundfarbe der Oberfläche beeinflussen möchten, bauen Sie zusätzlich ein `Surface`-Element ein:

```
setContent {
    FahrenheitTheme {
        // Hintergrund der Benutzeroberfläche
        Surface(color = MaterialTheme.colors.background) {
            // Container + Steuerelemente der Oberfläche
        }
    }
}
```

Diese vorgegebene Vorgehensweise hat zwei Vorteile:

- Zum einen können Sie viele Farb- und Layouteinstellungen in zentralen Dateien vornehmen.
- Zum anderen kümmert sich Ihre App selbstständig um die Umstellung zwischen dem Standardmodus und dem Dark Mode.

1.7 Aktivitäten und UI-Gestaltung

Reale Apps, deren Aufgaben »Hello World!« übersteigen, bestehen aus mehreren Aktivitäten. Zum Zeitpunkt meiner Tests bot Android Studio die Möglichkeit, mittels File • New • Compose • Empty Activity eine neue Aktivitätsklasse hinzuzufügen. Es

ist zu erwarten, dass es in künftigen Versionen mehr Compose-Musterdateien geben wird, mit denen Sie ein Projekt um vorgefertigte Aktivitäten für häufig benötigte Aufgaben erweitern können.

Um von einer Aktivität zur nächsten zu wechseln, erzeugen Sie zuerst ein `Intent`-Objekt, das den aktuellen Kontext (Eigenschaft `applicationContext`) und die Zielaktivität enthält. Dieses Objekt übergeben Sie an die Methode `startActivity`:

```
val intent = Intent(applicationContext, Activity2::class.java)
startActivity(intent)
```

Über den Back-Button in der Navigation Bar bzw. über die entsprechende Geste gelangen Sie zurück in die vorige Aktivität. Per Code erreichen Sie eine Rückkehr in die vorige Aktivität ganz einfach, indem Sie die Methode `finish` ausführen. Falls Sie eine App Bar verwenden, müssen Sie sich aktuell selbst darum kümmern, einen entsprechenden Back-Button einzubauen und in dessen `onClick`-Parameter `finish` aufrufen. Ein entsprechendes Beispiel folgt gleich.

Komplexere Setups realisieren Sie am einfachsten mit der Compose-Funktion `Scaffold`. Diese gibt Ihrer App eine vorgefertigte Struktur. Mit relativ wenig eigenem Code können Sie Ihr Programm mit einer App Bar, einer Bottom Bar, einem Menü usw. ausstatten. In diesem Abschnitt konzentriere ich mich auf diese Vorgehensweise.

Alternativ können Sie Ihre App natürlich auch von Grund auf selbst zusammensetzen. In diesem Fall müssen Sie auch die gesamte Navigation zwischen den Aktivitäten selbst implementieren. Dazu greifen Sie – ähnlich wie bei traditionellen Android-Apps – auf einen Navigation Controller zurück. Dessen Funktionsweise ist hier dokumentiert:

https://developer.android.com/jetpack/compose/navigation

Keine Fragmente mehr

Jetpack Compose löst sich vollständig von Fragmenten. Diese werden nur eingeschränkt und primär aus Kompatibilitätsgründen zu vorhandenem herkömmlichen Code unterstützt. »Richtige« Compose-Apps setzen sich ausschließlich aus Aktivitäten zusammen.

UO-Gestaltung mit »Scaffold«

Bei der Gestaltung der Benutzeroberfläche Ihrer App greift Ihnen Jetpack Compose mit der `Scaffold`-Funktion (wörtlich »Gerüst, Arbeitsplattform«) unter die Arme. Mit der `Scaffold`-Funktion können Sie eine Aktivität um häufig verwendete UI-Elementen ergänzen, z. B. um eine App Bar, eine Bottom Bar, einen Floating Action Button, und einen Drawer (also ein Menü):

```
Scaffold(
    topBar = { TopAppBar(...) {...} },
    bottomAppBar = { BottomAppBar(...) {...} },
    floatingActionButton = { FloatingActionButton(...) {...} },
    floatingActionButtonPosition = ...,
    drawerContent = { MyDrawerControls() },
    content = { MyControls() },
    usw.
)
```

Sämtliche Parameter sind optional. Wenn Sie einzelne UI-Elemente nicht brauchen, lassen Sie den Parameter einfach weg. Das folgende Minibeispiel zeigt, wie Sie eine Aktivität mit Back Button realisieren. An die Funktion `TopAppBar` müssen Sie den gewünschten Titel und ein Icon übergeben. Es gibt weitere Parameter, z. B. `backgroundColor` oder `contentColor`, wenn Sie andere Farben wünschen, als in den Grundeinstellungen Ihrer App vorgesehen sind.

```
// Beispielprojekt HelloCompose, Datei MainActivity.kt
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            HelloComposeTheme {
                Surface {
                    MyScaffold()
                }
            }
        }
    }
    // Grundgerüst einer Aktivität mit App Bar
    @Composable
    fun MyScaffold() {
        Scaffold(
            topBar = {
                TopAppBar(
                    // Titel der Aktivität übernehmen
                    title = { Text(title.toString()) },
                    // Back Button: zurück zur vorigen Aktivität
                    navigationIcon = {
                       IconButton(
                           content = {
                               Icon(Icons.Filled.ArrowBack,
                               contentDescription = "back") },
                           onClick = { finish() })
                    }
```

```
                )   // Ende TopAppBar
            },      // Ende Lambda-Ausdrück für topBar = ...
            content = { MyControls() } )
    }
    // Inhalt der Aktivität
    @Composable
    fun MyControls() {
        Column {
            Text("test")
            // usw.
        }
    }
}
```

Im Beispielprojekt `ComposingApp` habe ich in der Datei `Globals.kt` die Funktion `AppBarWithBackButton` definiert. Sie erzeugt ein Gerüst für eine Aktivität mit einer App Bar und einem Back Button (zu sehen z. B. in Abbildung 1.8). Der Code sieht ganz ähnlich wie im vorhin abgedruckten Listing aus. Explizit hinweisen möchte ich aber auf den Parameter `content`, mit dem an die Funktion die Compose-Elemente übergeben werden, die Sie innerhalb der Aktivität darstellen wollen. Der Datentyp ist eine Funktion mit der Annotation `@Composable`, die als Parameter einen `PaddingValues`-Ausdruck erwartet und nichts zurückgibt (`-> Unit`).

```
// Projekt ComposingApps, Datei Globals.kt
@Composable
fun AppBarWithBackButton(
    activity: ComponentActivity,
    content: @Composable (PaddingValues) -> Unit)
{
    Scaffold(
        topBar = { TopAppBar( ... ) },  // wie im vorigen Listing
        content = content)
}
```

Diese Funktion rufe ich in den Dateien `Activity2.kt`, `Activity3.kt` usw. auf. Ein weiteres `Scaffold`-Beispiel finden Sie hier:

https://developer.android.com/reference/kotlin/androidx/compose/material/package-summary#scaffold

Popup-Dialoge

Die Compose-Funktion `AlertDialog` ermöglicht es, einfache Popup-Dialoge zu realisieren (siehe Abbildung 1.9). Etwas irritierend ist dabei die Vorgehensweise: Sie erzeugen den Dialog nicht dann, wenn Sie ihn tatsächlich brauchen, sondern schon vorweg bei der Initialisierung der Aktivität. Damit der Dialog nicht stört, machen Sie ihn aber

erst dann sichtbar, wenn eine `MutableState`-Variable (im Folgenden `popupVisible`) im Lambda-Ausdruck eines `onClick`-Parameters auf `true` gesetzt wird.

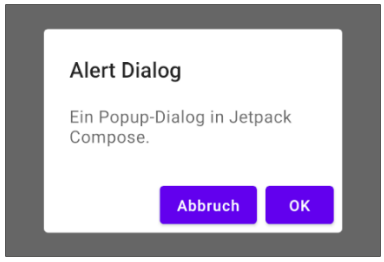

Abbildung 1.9 Ein Popup-Dialog

Zur Weitergabe des Ergebnisses habe ich mit `popupResult` noch eine `MutableState`-Variable verwendet. Der Zustand dieser Variable wird im Beispiel in einem Textfeld angezeigt.

An `AlertDialog` müssen eine Menge Parameter übergeben werden:

- `title` und `text` enthalten die Überschrift und den Text des Dialogs.
- `confirmButton` und `dismissButton` legen den Text und die Reaktion für die beiden Buttons fest. Der Parameter `dismissButton` ist optional. Wenn Sie ihn weglassen, hat der Dialog nur einen Button.
- `onDismissRequest` steuert die Reaktion, wenn der Benutzer außerhalb des Dialogs klickt oder versucht, den Dialog mit dem Back-Button zu verlassen. Wenn Sie hier nur ein geschwungenes Klammernpaar übergeben, kann der Dialog auf diese Weise nicht beendet werden.

```
// Projekt ComposingApps, Datei MainActivity.kt
class MainActivity : ComponentActivity() {
    lateinit var popupVisible : MutableState<Boolean>
    lateinit var popupResult : MutableState<Boolean>
    ...
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Column {
                MyAlertDialog()   // Dialog anfänglich unsichtbar
                Button( content = { Text("Popup-Dialog") },
                        onClick = { popupVisible.value = true } )
                Text("Popup-Ergebnis: ${popupResult.value}")

        }
    }
```

```
    @Composable
    fun MyAlertDialog() {
        if (popupVisible.value)
            AlertDialog(
                title = { Text("Alert Dialog") },
                text =  { Text("Ein Popup-Dialog in " +
                               "Jetpack Compose.") },
                confirmButton = {
                    Button(content = { Text("OK") },
                           onClick = {
                              popupVisible.value = false
                              popupResult.value = true } ) },
                dismissButton = {
                    Button(content = { Text("Abbruch") },
                           onClick = {
                              popupVisible.value = false
                              popupResult.value = false } ) },
                onDismissRequest = {
                    popupVisible.value = false
                    popupResult.value = false  }
            )
    }
}
```

1.8 Beispiel: Fahrenheit-Umrechner

Die Zielsetzung für das Abschlussbeispiel kennen Sie schon aus Kapitel 22: Eine kleine App soll beim Umrechnen zwischen Fahrenheit und Celsius helfen (siehe Abbildung 1.10). Die Farben der App passen sich bei Bedarf automatisch an den Dark Mode an. Die erforderlichen Farbeinstellungen befinden sich in der Datei `ui/Color.kt`.

Der restliche Code befindet sich in der Klasse `MainActivity`. Dort sind vier Variablen deklariert, die in der Methode `onCreate` initialisiert werden. Sie speichern den Inhalt der beiden Texteingabefelder, den Kontext der sowie eine Referenz auf den Tastatur-Controller. Aktivität:

```
// Projekt Fahrenheit, Datei MainActivity.kt
class MainActivity : ComponentActivity() {
   private lateinit var fahrenheit: MutableState<String>
   private lateinit var celsius: MutableState<String>
   private lateinit var context: Context
   private var kbdController: SoftwareKeyboardController? = null
```

```
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            fahrenheit    = rememberSaveable { mutableStateOf("") }
            celsius       = rememberSaveable { mutableStateOf("") }
            context       = LocalContext.current
            kbdController = LocalSoftwareKeyboardController.current

            FahrenheitTheme {
                Surface(color = MaterialTheme.colors.background) {
                    FahrenheitUI()
                }
            }
        }
    }
    ...
}
```

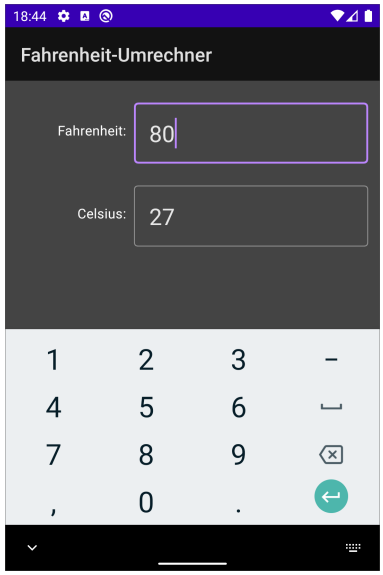

Abbildung 1.10 Der Fahrenheit-Umrechner als Jetpack-Compose-App im Dark Mode

Die Benutzeroberfläche der App wird in der Funktion `FahrenheitUI` zusammengesetzt. Bemerkenswert sind die Modifier für das zentrale `Column`-Element: Es soll die ganze Höhe des Bildschirms füllen (und wegen der enthaltenen Elemente auch die ganze Breite). Ein Klick auf den Hintergrund blendet die Tastatur aus.

```
@Composable
fun FahrenheitUI() {
    Column(Modifier.fillMaxHeight()
                   .clickable { kbdController?.hide() })
    {
        Spacer(Modifier.requiredHeight(15.dp))
        TemperatureBox("Fahrenheit:", txt = fahrenheit)
        Spacer(Modifier.requiredHeight(15.dp))
        TemperatureBox("Celsius:", txt = celsius)
    }
}
```

Temperatureingabe

Für die beiden beschrifteten Eingabefelder der App ist die Funktion `TemperaturBox` zuständig. An die Funktion werden zwei Parameter übergeben: die Zeichenkette zur Beschriftung des Eingabefelds sowie eine `MutableState`-Variable für dessen Inhalt.

Die Funktion beginnt mit der Initialisierung diverser Variablen. `type` merkt sich den ersten Buchstaben der Titelzeichenkette. Dieses Zeichen wird später an die Umrechnungsfunktion übergeben, damit dort klar ist, ob von Fahrenheit in Celsius oder in die umgekehrte Richtung gerechnet werden soll.

```
@Composable
fun TemperatureBox(title: String, txt: MutableState<String>) {
    val type = title.firstOrNull() ?: '-'

    Row(Modifier.fillMaxWidth(),
        verticalAlignment = Alignment.CenterVertically)
    {   // Beschriftung: 1/3 des horiz. Platzes, rechtsbündig
        Column(Modifier.padding(0.dp, 2.dp, 4.dp, 0.dp)
                       .weight(1F),
               horizontalAlignment= Alignment.End)
        {
            Text(title,
                 style = MaterialTheme.typography.body2,
                 color = MaterialTheme.colors.onBackground)
        }
        // Eingabefeld: 2/3 des horiz. Platzes
        OutlinedTextField(
            value = txt.value,
            label = {},
            modifier = Modifier.padding(4.dp, 0.dp, 12.dp, 0.dp)
                               .weight(2F),
            textStyle = MaterialTheme.typography.h5,
```

```
            keyboardOptions = KeyboardOptions(
                keyboardType = KeyboardType.Number),
            onValueChange = { convertTemp(type, it) })
    }
}
```

Eigentlich gibt es ja nur zwei Steuerelemente: einen `Text` und ein `OutlinedTextField`. Diese sind in eine `Row` verpackt, wobei der zur Verfügung stehende Platz aufgeteilt wird: Ein Drittel nimmt die Beschriftung ein (`weight(1F)`, zwei Drittel das Eingabefeld (`weight(2F)`).

Die `Column` für `Text` ist notwendig, um die Beschriftung rechtsbündig auszurichten. Die `OutlinedTextField`-Parameter `keyboardOptions` bewirkt, dass eine Tastatur zur Eingabe von Zahlen angezeigt wird. Mit `onValueChange` wird bei der Eingabe jedes Zeichens die Methode `convertTemp` aufgerufen.

Eingabeverarbeitung und Temperaturumrechnung

Die Methode `convertTemp` macht mehr, als nur die Temperatur zwischen den beiden Einheiten umzurechnen. Sie synchronisiert auch die `MutableState`-Variable des jeweils anderen Eingabefelds. Falls die Methode erkennt, dass der Benutzer ↵ gedrückt hat, wird das `\n`-Zeichen nicht gespeichert und stattdessen die Tastatur ausgeblendet.

```
fun convertTemp(type: Char, input: String) {
    if (input.contains("\n") || input.length > 5) {
        // Return und zu lange Eingaben ignorieren,
        // Tastatur ausblenden
        kbdController?.hide()
        return
    }
    if (type == 'C') {                // von Celsius zu Fahrenheit
        celsius.value = input
        val c = input.replace(",", ".").toDoubleOrNull()
            ?: return
        val f = round(c * 1.8 + 32).toInt()
        fahrenheit.value = f.toString()
    } else if (type == 'F') {         // von Fahrenheit zu Celsius
        fahrenheit.value = input
        val f = input.replace(",", ".").toDoubleOrNull()
            ?: return
        val c = round((f - 32) / 1.8).toInt()
        celsius.value = c.toString()
    }
}
```