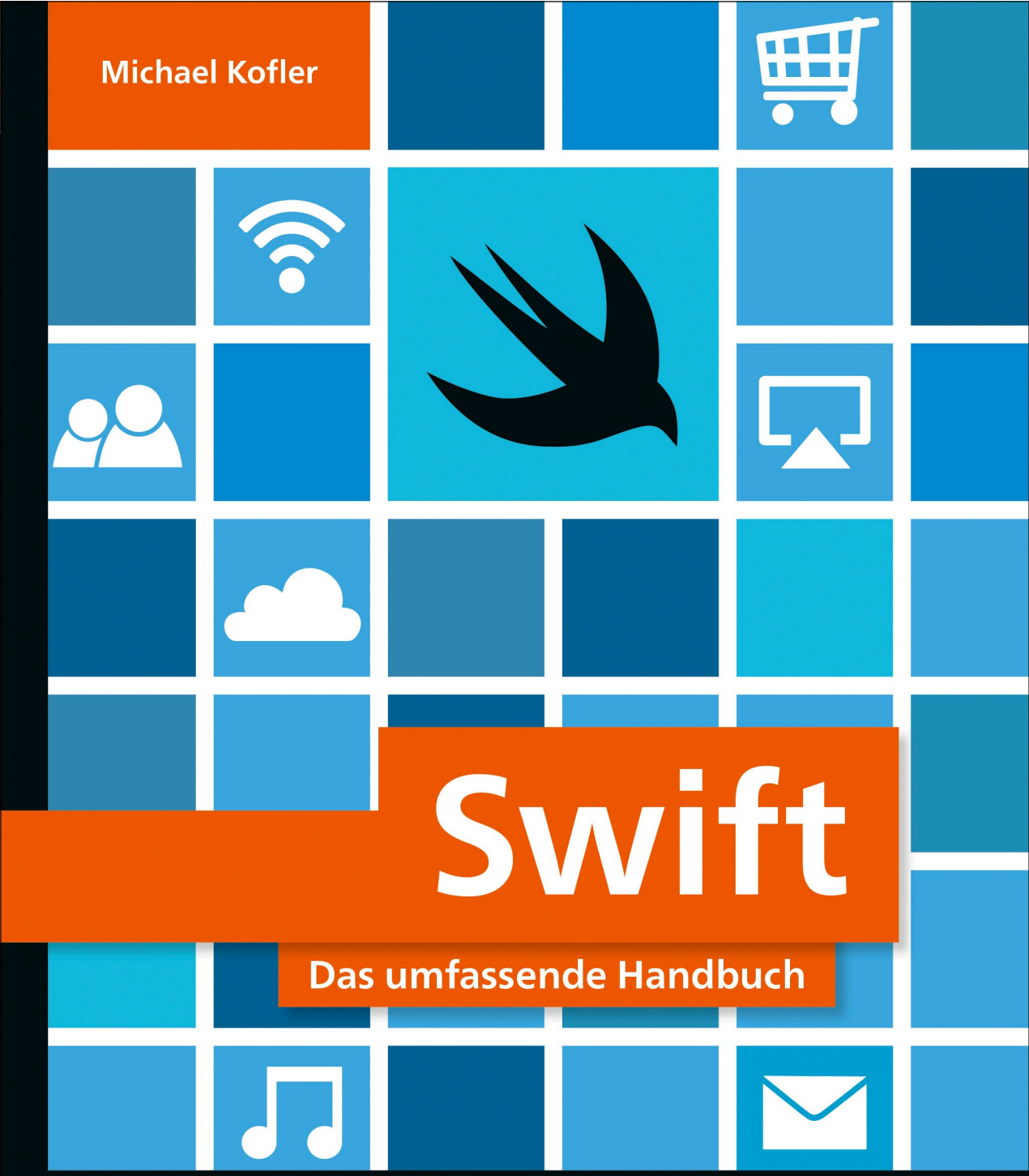
Michael Kofler



Swift

Das umfassende Handbuch

- Von den Syntax-Grundlagen bis zur App-Entwicklung mit SwiftUI
- Optionals, Closures, Data Binding, SwiftData
- Vollständige Beispiel-Apps, inkl. App-Store-Integration

5., aktualisierte Auflage

Rheinwerk Computing

Vorwort

Als Apple 2014 die Programmiersprache Swift vorstellte, war das gleichsam ein Befreiungsschlag: Objective-C wird zwar das Fundament von macOS, iOS etc. bleiben – aber das ändert nichts daran, dass diese Sprache in den 1980er-Jahren entworfen wurde und in keinerlei Hinsicht mit modernen Programmiersprachen mithalten kann.

Swift ist dagegen ein sauberer Neuanfang. Bei der Vorstellung wurde Swift auch *Objective-C without the C* genannt. Natürlich ist Swift von Objective-C beeinflusst – schließlich muss Swift kompatibel mit den unzähligen Apple-Bibliotheken sein. Gleichzeitig realisiert Swift viele neue Ideen und greift Konzepte von C#, Haskell, Java, Python etc. auf. Daraus ergeben sich mehrere Vorteile:

- Swift zählt zu den modernsten Programmiersprachen, die es momentan gibt.
- Code lässt sich in Swift syntaktisch sehr elegant und kompakt formulieren.
- Der resultierende Code ist besser lesbar und wartbar.
- Swift ist ein Open-Source-Produkt und steht auch für Linux zur Verfügung. Der Entwicklungsprozess erfolgt offen und transparent.

Swift 6, SwiftUI und SwiftData

Während die ersten Swift-Versionen in relativ rascher Abfolge erschienen, begann mit Swift 5 eine längere Phase ohne ein neues Major Release. Apple hat diese Zeit einerseits genutzt, um vorhandene Sprachfeatures zu stabilisieren, andererseits dazu, Schritt für Schritt ein komplett neues Framework für die asynchrone Programmierung zu schaffen. Damit setzt die im Herbst 2024 vorgestellte Version 6 einen Meilenstein.

Noch mehr hat sich abseits der eigentlichen Programmiersprache getan: Swift ist nur so gut wie die Bibliotheken, auf die es zurückgreifen kann. Die wichtigste Bibliothek für die App-Entwicklung war fast zwei Jahrzehnte lang das für das erste iPhone entwickelte UIKit. Mit UIKit ist es so ähnlich wie mit Objective-C: Die Bibliothek funktioniert, aber die Programmierung ist enorm umständlich.

Seit 2019 gibt es mit SwiftUI eine moderne Alternative dazu. SwiftUI ist eine grundlegend neue Bibliothek zur Gestaltung grafischer Benutzeroberflächen. Anders als UIKit ist SwiftUI für *alle* Apple-Plattformen geeignet, von macOS über iOS, tvOS, watchOS bis hin zu visionOS.

Der Einstieg in SwiftUI ist allerdings nicht ganz einfach: Die deklarative Syntax von SwiftUI, der gewöhnungsbedürftige Umgang mit `@State`-Variablen – selbst wenn Sie schon Erfahrung mit der App-Entwicklung haben, ist die erste App mit SwiftUI ein mitunter frustrierender Neuanfang. Das gilt heute noch, aber es galt noch viel mehr für alle, die schon 2019 oder 2020 erste Experimente mit SwiftUI machten. Apple hat die Bibliothek leider zu früh und zu unausgereift ausgeliefert. Unter dem schlechten Ruf, den sich SwiftUI damals erworben hat, leidet SwiftUI bis heute. Zu Unrecht! Mittlerweile sind die schlimmsten Kinderkrankheiten ausgestanden, das Internet ist voller WWDC-Videos, Blogs und Tutorials, die bei den ersten Schritten helfen.

Natürlich existiert das UIKit parallel dazu weiterhin. Millionen Zeilen Code, die damit erstellt wurden, müssen weiter gewartet werden. Aber neue Projekte sollten – trotz mancher Einschränkungen, die bis heute für SwiftUI gelten – unbedingt mit SwiftUI gestartet werden. Ich behandle deswegen in diesem Buch ausschließlich SwiftUI. Bücher, die UIKit beschreiben, gibt es schon genug – inklusive meiner eigenen Bücher zu Swift 2 bis 5.

SwiftUI war für Apple nur der Anfang für die Modernisierung grundlegender Bibliotheken. 2023 folgte mit SwiftData ein weiterer Schritt. SwiftData ist eine Bibliothek zur persistenten Speicherung eigener Daten, wahlweise lokal oder in der iCloud. SwiftData gilt als logischer Nachfolger von Core Data.

Über dieses Buch

Dieses Buch vermittelt einen kompakten Einstieg in die Programmiersprache Swift in der Version 6. Das Buch ist in fünf Teile gegliedert:

- **Teil I** stellt Ihnen Swift und Xcode vor. Damit Sie eine Idee davon bekommen, wie App-Programmierung in der Praxis aussieht, präsentiert bereits Kapitel 2, »Learning by Doing: Die erste App«, ein erstes konkretes Beispiel.
- **Teil II** richtet sich an alle, die Swift systematisch lernen wollen. Die Themenpalette reicht vom Umgang mit Variablen und elementaren Datentypen bis hin zur Syntax der objekt- und protokollorientierten Programmierung.
- In **Teil III** geht es um SwiftUI. In mehreren Kapiteln lernen Sie anhand zahlreicher Beispiele wichtige Bestandteile und Arbeitsweisen rund um SwiftUI kennen. Ein eigenes Kapitel ist dabei der richtigen Verbindung zwischen Benutzeroberfläche und Daten gewidmet. Hier lernen Sie State- und Bindung-Variablen kennen, Observable-Klassen und das MVVM-Muster (*Model, View-Model, View*).
- **Teil IV** fasst wichtige Programmiertechniken in Bausteinform zusammen. Egal, ob Sie asynchronen Code entwickeln, Dateien lesen oder schreiben, Netzwerkfunktionen oder REST-APIs nutzen möchten – hier finden Sie geeignete Anleitungen. Zwei eigene Kapitel behandeln die iCloud-Programmierung und die schon erwähnte SwiftData-Bibliothek.

- In **Teil V** heißt es wieder *Learning by Doing*. Anhand von zwei umfangreichen Beispielen zeige ich Ihnen, wie Sie das im Buch vermittelte Wissen zu »richtigen« Apps kombinieren. Mit dabei ist die Lokalisierung von Apps, also die Unterstützung mehrerer Sprachen. Außerdem gebe ich Ihnen eine Menge Tipps, wie Sie Ihre eigene App in den App Store bringen.

Es ist Marketing-Überlegungen geschuldet, dass dieses Buch als »5. Auflage« verkauft wird. Aus meiner Sicht handelt es sich um ein vollkommen neues Buch. Größere Textpassagen habe ich nur bei Teil II aus der Vorauflage übernommen.

Neu ist das Buch aber auch aus einer anderen Perspektive: KI-Tools ändern gerade, wie Sie und ich programmieren. Meine eigenen Erfahrungen damit haben dieses Buch stark beeinflusst. Mehr denn je erscheint es mir wichtig, Grundlagenwissen zu vermitteln. Die Lektüre dieses Buchs soll bei Ihnen das Fundament schaffen, mit dem Sie in der Folge KI-Tools effizient anwenden können.

Um von diesem Buch maximal zu profitieren, benötigen Sie weder Vorkenntnisse in Xcode noch in der App-Entwicklung. Ich setze aber voraus, dass Sie bereits Erfahrungen mit einer beliebigen Programmiersprache gesammelt haben. Ich erkläre Ihnen in diesem Buch also, wie Sie in Swift mit Variablen umgehen, Schleifen programmieren und Klassen entwickeln, aber nicht, was Variablen sind, wozu Schleifen dienen und warum Klassen das Fundament der objektorientierten Programmierung sind. So kann ich Swift kompakt und ohne viel Overhead beschreiben und den Schwerpunkt auf die konkrete Anwendung legen.

Viel Spaß!

Eine neue Programmiersprache zu erlernen ist immer eine Herausforderung. Noch schwieriger ist es, einen Überblick über die schier unüberschaubare Fülle von Bibliotheken zu gewinnen, die Sie zur App-Entwicklung brauchen. Dieses Buch soll Ihnen bei beiden Aspekten helfen.

Wenn Sie in die App-Entwicklung mit Swift einsteigen, haben Sie das Privileg, mit einer der modernsten aktuell verfügbaren Programmiersprachen zu arbeiten. Sobald Sie die ersten Schritte einmal erfolgreich absolviert haben, wird die Faszination für diese Sprache auch Sie erfassen. Bei Ihrer Reise durch die neue Welt von Swift wünsche ich Ihnen viel Spaß und Erfolg!

Michael Kofler (*https://kofler.info*)

Beispieldateien

Die Beispieldateien zu diesem Buch können Sie hier herunterladen:

https://www.rheinwerk-verlag.de/6081

Kapitel 13
Views

Nach den zugegebenermaßen recht theoretischen Kapiteln zur Syntax der Sprache Swift folgt nun im dritten Teil dieses Buchs eine Einführung zu SwiftUI. Sie werden sehen, dass die Lektüre dieser Kapitel deutlich abwechslungsreicher ist: Der praktische Nutzen der einzelnen Komponenten (»Views«) erschließt sich sofort, zahlreiche Abbildungen lockern den Text auf.

SwiftUI steht für *Swift User Interface*. Die relativ neue Bibliothek eignet sich zur Gestaltung von grafischen Oberflächen für Apps auf allen erdenklichen Geräten – von der Apple Watch bis zur Brille Apple Vision Pro. Ich konzentriere mich in diesem Buch allerdings auf die Betriebssysteme iOS für iPhones und iPads sowie macOS für MacBooks und andere »herkömmliche« Computer von Apple.

Apple hat SwiftUI 2019 vorgestellt. Damals war SwiftUI aber alles andere als praxistauglich. Wer sich darauf verlassen hat, dass Apple normalerweise fertige, ausgereifte Produkte vorstellt, wurde 2019 enttäuscht. Die Bibliothek leidet bis heute darunter, dass viele Entwicklerinnen und Entwickler SwiftUI ausprobierten, nach einigen Monaten frustriert zum UIKit zurückkehrten und SwiftUI danach keine zweite Chance mehr gaben. Das ist schade, weil SwiftUI dem UIKit mittlerweile in beinahe jeder Hinsicht weit überlegen ist. (Es gibt aber noch immer vereinzelte UIKit-Komponenten ohne SwiftUI-Entsprechung. Zur Not müssen Sie UIKit-Code mit SwiftUI kombinieren – siehe Abschnitt 17.5, »UIKit-Views in SwiftUI verwenden«.)

Im Mittelpunkt dieses Kapitels stehen »Views« (Ansichten), also die Grundbausteine jeder App. Syntaktisch gesehen ist eine `View` ein Protokoll. Es schreibt vor, dass jede Struktur, die dieses Protokoll implementiert, die Eigenschaft `body` zur Verfügung stellt. Der Code für `body` legt dann fest, wie die View aussieht und funktioniert.

In der SwiftUI-Bibliothek gibt es zahlreiche vordefinierte Strukturen, die dem `View`-Protokoll entsprechen: Buttons, Textfelder, Listen, Stacks (Container) usw. Sie setzen Ihre eigenen Apps aus solchen vordefinierten Komponenten zusammen oder schreiben Code für eigene Views, also gewissermaßen für eigene Steuerelemente. Das gelingt verblüffend einfach!

In diesem Kapitel stelle ich Ihnen grundlegende Views wie `Button`, `Text`, `HStack` oder `VStack` vor und erkläre Ihnen, wie Sie daraus Oberflächen kombinieren und deren

Aussehen modifizieren. Zur optischen Gestaltung sowie zur Veränderung anderer Eigenschaften setzen Sie sogenannte Modifier ein. Auf technischer Ebene handelt es sich dabei um nachgestellte Methoden.

Um auf Ereignisse wie Berührungen oder Fingergesten zu reagieren, statten Sie Ihren Code mit Action-Closures aus. Der dort formulierte Code wird ausgeführt, sobald ein Ereignis auftritt.

Auch wenn Sie SwiftUI-Oberflächen ausschließlich durch Code steuern, hilft Ihnen Xcode bei der Gestaltung und beim Test. Daher gehe ich in diesem Kapitel auch darauf ein, mit welchen Funktionen Xcode den App-Entwurf unterstützt.

In den weiteren Kapiteln geht es dann um den Datenaustausch zwischen Views und Ihren eigenen Daten (Bindings), um den Umgang mit Listen, um die Gestaltung von Apps mit mehreren Ansichten (Navigation) sowie um die Programmierung von optisch ansprechenden Übergängen zwischen verschiedenen Ansichten (Animationen).

Deklarativ versus imperativ

SwiftUI ist ein deklaratives Framework zur Gestaltung von Oberflächen. »Deklarativ« bedeutet, dass Sie per Code das Aussehen und die Funktionsweise festlegen, ohne aber jedes Detail der dahinterliegenden Logik auszuformulieren. Das Framework weiß selbst, wie es sich beim Berühren eines Buttons verhalten soll, welche Daten/Variablen es mit der Oberfläche synchronisiert und wie es Komponenten der Oberfläche je nach Zielplattform (iOS, macOS etc.), Modus (Light/Dark Mode), Drehung (Portrait/Landscape) und Schriftgröße optisch und funktionell gestaltet.

Im Gegensatz dazu ist UIKit ein imperatives Framework. Hier legen Sie selbst fest, wie Ihr Programm reagiert, wenn ein Button berührt bzw. angeklickt wird. Sie formulieren selbst Code für Aktionen, verändern selbst Daten/Variablen in Ihrem Programm bzw. den Inhalt der Steuerelemente. Solange es nur einen Button gibt, ist die Programmierung imperativer Frameworks trivial einfach. Mit mehreren sich gegenseitig beeinflussenden Optionen, Programmzuständen und Ereignissen explodiert aber die Anzahl der Varianten, in welcher Reihenfolge imperativ formulierter Code ausgeführt wird. Oft gibt es zum Schluss Sonderfälle, bei denen sich Ihre App falsch verhält. Das Debugging ist dann sehr aufwendig, weil es kaum gelingt, die zum Fehler führende Ereignisabfolge zu rekonstruieren.

Deklarative Frameworks gelten aktuell als der beste Weg, komplexe Oberflächen zu gestalten. Deklarative Frameworks sind auch für viele andere Sprachen modern geworden, z. B. Jetpack Compose (Kotlin), Flutter (Dart) oder React (JavaScript/TypeScript).

SwiftUI versus UIKit

Trotz der unumstrittenen Vorteile des deklarativen Konzepts hatte SwiftUI einen schweren Start. Ein Grund bestand darin, dass die ersten SwiftUI-Versionen tatsächlich komplett unausgereift waren. Dazu kam die anfangs äußerst schwache Xcode-Unterstützung: Während UIKit-Oberflächen relativ intuitiv mit ein paar Mausklicks zusammengestellt und in XML-Dateien gespeichert werden, setzt SwiftUI rein auf Swift-Code. Persönlich sehe ich darin einen großen Vorteil, aber natürlich ist die erstmalige Umstellung unbequem. Mittlerweile hat Apple die Xcode-Integration stark verbessert: Xcode unterstützt Sie nun mit Dialogen bei der Einstellung grundlegender Eigenschaften von Views, beim Einfügen neuer Views sowie bei der Restrukturierung des Codes. Vor allem wenn Sie den Umgang mit SwiftUI neu lernen, sind diese Xcode-Features hilfreich. (Ein paar Wochen später werden Sie den Großteil des Codes über die Tastatur eingeben – das ist bei Weitem der schnellste Weg.)

Während diese behobenen Probleme wie ausgestandene Kinderkrankheiten erscheinen, kritisieren UIKit-Fans, dass SwiftUI das etablierte Framework bis heute nicht vollständig ersetzen kann. Beispielsweise fehlen Möglichkeiten zur Darstellung von Webseiten (`UIWebView`), zur Darstellung und Bearbeitung von formatiertem Text (`UITextView`), zum Abspielen von Mediendateien und zum Zugriff auf die Kamera, zur Ausführung von Low-Level-Grafik-Code etc. Der einfachste Ausweg besteht darin, die entsprechenden UIKit-Komponenten in die eigene SwiftUI-App zu integrieren. Das gelingt am einfachsten über eine Kompatibilitätsschicht (`UIViewRepresentable` und `UIViewControllerRepresentable`).

Trotz dieser Einschränkungen habe ich mich dazu entschlossen, in diesem Buch voll auf SwiftUI zu setzen. Natürlich ist das UIKit als Technologie nicht tot: Alleine zur Wartung alten Codes wird es das Framework sicher noch viele Jahre geben. Die Zukunft gehört aber ganz eindeutig SwiftUI. Das neue Framework bietet gegenüber dem UIKit unzählige Vorteile: einfacherer, kürzerer und besser wartbarer Code, effizientere Ausführung, höhere Entwicklungsgeschwindigkeit, höhere Kompatibilität über mehrere Plattformen usw. Wenn Sie also eine neue App starten, sollte SwiftUI unbedingt die erste Wahl sein!

13.1 Grundlagen

In Kapitel 1, »Hello, World!«, habe ich Ihnen bereits den Aufbau des Hello-World-Codes erläutert, mit dem jedes neue SwiftUI-Projekt startet. Ich gehe auch in diesem Kapitel davon aus, dass Sie mit einem neuen Testprojekt vom Typ Multiplatform • App experimentieren. Nachdem Sie das Projekt geöffnet haben, ersetzen Sie in der Datei `ContentView.swift` den vorgegebenen Code für die `body`-Eigenschaft durch Ihren eigenen Code. Im Unterschied zu Kapitel 1 gehe ich jetzt aber davon aus, dass Ihnen

Begriffe wie »Closure«, »Konstruktor«, »Protokoll« oder »Read-only Property« vertraut sind.

Ich empfehle Ihnen, im Preview-Fenster ein iPhone als Vorschaugerät zu verwenden. Die Ausführung des dafür notwendigen Simulators kostet mehr Rechenzeit und Arbeitsspeicher als die Defaultansicht im macOS-Modus. Wenn Sie aber primär iOS-Apps entwickeln möchten, sollten Sie diese Nachteile in Kauf nehmen. Im Gegenzug erhalten Sie eine iOS-typische Arbeitsumgebung, was beim Erlernen von SwiftUI im Hinblick auf die App-Entwicklung durchaus hilfreich ist.

Was ist die »ContentView«?

Der Mustercode neuer Apps erzeugt in `<projektname>App.swift` eine Instanz der `ContentView`-Struktur und zeigt diese an. Somit bestimmt die `ContentView` das anfängliche Erscheinungsbild Ihrer App.

`ContentView` ist aber kein vorgegebener Name in SwiftUI. Sie können die Startansicht Ihrer App ebenso gut in einer eigenen Struktur realisieren. Diese Struktur muss wie `ContentView` das `View`-Protokoll implementieren. Außerdem müssen Sie den Main-Code in `<projektname>App.swift` anpassen und dort anstelle von `ContentView` den Konstruktor Ihrer eigenen Struktur aufrufen.

View-Gestaltung mit Modifiern

Views sind Komponenten einer App. Es gibt Views zur Darstellung von Texten, Bildern oder Listen, zur Realisierung von Buttons und Optionsfeldern, zur Auswahl von Datum und Uhrzeit usw. Wichtig ist, dass an vielen Stellen im SwiftUI-Code exakt *ein* `View`-Element erwartet wird. Sie müssen dort also Code eingeben, der genau eine View (z. B. einen Text oder einen Button) erzeugt und zurückgibt. Alles andere, also z. B. der Aufruf von `print` oder das Erzeugen mehrerer Views, führt zu einem Syntaxfehler. Falls Sie mehrere Views neben- oder übereinander platzieren möchten, müssen Sie diese in Container-Views wie `HStack` oder `VStack` verpacken. Darauf gehe ich gleich ein.

Views verwenden bei der Darstellung ihres Inhalts Defaulteinstellungen für Farben und Schriftgrößen, die von der Zielplattform abhängig sind. tvOS verwendet standardmäßig viel größere Schriften als iOS. Gleichzeitig berücksichtigt SwiftUI aber auch die gerade aktiven Einstellungen des Geräts (Light oder Dark Mode, vom Benutzer eingestellte größere oder kleinere Schriften etc.

Wenn Sie eine von den Defaulteinstellungen abweichende Darstellung wünschen, fügen Sie dem View-Konstruktor sogenannte Modifier hinzu. Modifier sind spezielle

Methoden, die die View verändern. Je nach View-Typ können Sie damit Farben, Größen, Abstände sowie die Ausrichtung des Inhalts beeinflussen (siehe Abbildung 13.1).

Als Ausgangspunkt zum Experimentieren mit Modifiern verwenden Sie am besten eine `Text`-View mit einem etwas längeren Text:

```
// Projekt hello-swiftui, Datei ContentView.swift
import SwiftUI  // importiert das SwiftUI-Framework, in
                // jeder Code-Datei erforderlich!

struct ContentView: View {
    var body: some View {
        Text("Lorem ipsum dolor sit amet, consetetur sadipscing
              elitr, sed diam nonumy eirmod tempor invidunt ut
              labore et dolore magna aliquyam erat, sed diam
              voluptua.")
            .font(.body)                        // Schriftgröße
            .fontWeight(.bold)                  // fette Schrift
            .foregroundColor(.blue)             // blaue Schrift
            .background(.yellow)                // gelber Hintergr.
            .multilineTextAlignment(.center) // zentriert
            .lineLimit(3)                       // max. 3 Zeilen
    }
}
```

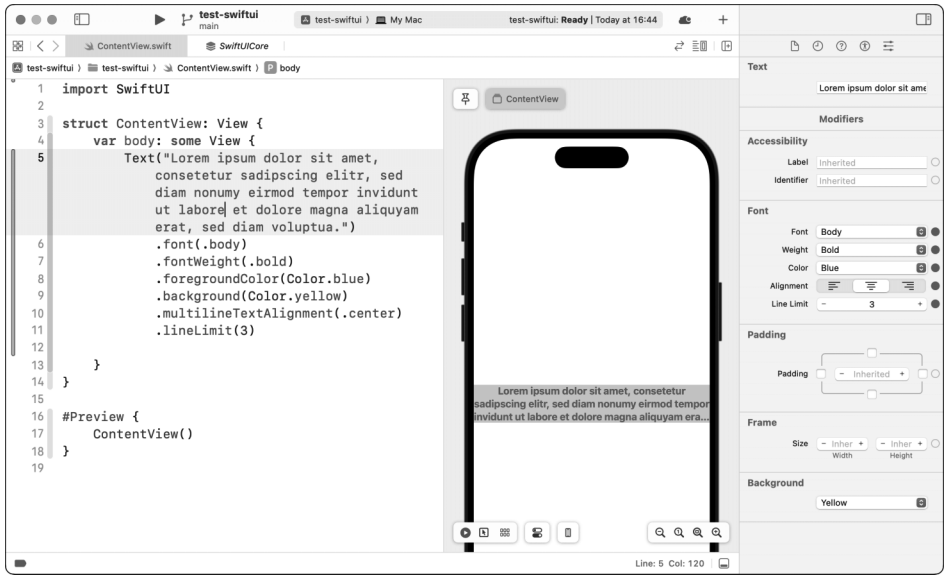


Abbildung 13.1 Die Darstellung des Texts wurde durch etliche Modifier beeinflusst. Rechts ist der Attributinspektor zu sehen.

Modifier sind Methoden, die als Ergebnis eine neue View zurückgeben. (Die Bezeichnung »Modifier« ist also irreführend! Modifier-Methoden verändern eine View nicht, sondern erzeugen eine neue View, bei der *eine* Eigenschaft anders ist als bisher.)

Im obigen Beispiel erzeugt `Text("Lorem ...")` die originäre Text-View. Die nachfolgenden Modifier liefern immer wieder neue Views. Tatsächlich angezeigt wird nur das vom letzten Modifier gelieferte Ergebnis.

Es ist üblich, die Modifier zeilenweise anzuwenden. Das liefert den am besten lesbaren Code. Syntaktisch ist es aber natürlich auch korrekt, mehrere Modifier in einer Zeile zu kombinieren, also beispielsweise:

```
Text("Lorem ...").font(.body).fontWeight(.bold)
    .foregroundColor(.blue).background(.yellow)
    .multilineTextAlignment(.center).lineLimit(3)
```

Die Modifier-Reihenfolge ist wichtig

Bei diesem Beispiel ist es egal, in welcher Reihenfolge Sie die Modifier anwenden. Sämtliche beeinflussten Eigenschaften sind voneinander unabhängig. Das ist aber nicht immer der Fall! Gerade bei Modifiern, die die Darstellung von Bildern beeinflussen, sieht das Endergebnis je nach Reihenfolge unterschiedlich aus!

Welche Modifier gibt es? Eine ganze Menge! Im Verlauf der nächsten Kapitel werde ich Ihnen eine Menge näher vorstellen. Bei der Code-Eingabe hilft die automatische Vervollständigung. Alternativ können Sie das betreffende View-Element im Code mit der rechten Maustaste anklicken und Show Swift UI Inspector anklicken. Damit gelangen Sie in einen Dialog, in dem Sie per Mausklick die wichtigsten Modifier aktivieren bzw. einstellen können. Dieselben Optionen zeigt auch der Attributinspektor in der rechten Xcode-Seitenleiste (siehe Abbildung 13.1).

Container-Views

Die `body`-Eigenschaft des `View`-Protokolls erwartet genau eine `View`-Instanz. Die Oberflächen von Apps bestehen aber aus vielen Views. Die Lösung dieses Dilemmas sind Container-Views. In ihnen können mehrere andere Views platziert werden. Nach außen hin gelten Container aber als *eine* View. Die beiden wichtigsten Container sind der `HStack` und der `VStack`. In ihnen werden mehrere Views nebeneinander (horizontal) oder übereinander (vertikal) angeordnet (siehe Abbildung 13.2).

Das folgende Listing zeigt die Anwendung von `VStack` und `HStack` und stellt gleich den nächsten Modifier vor: `padding` umgibt die View mit einem außen liegenden Rand (standardmäßig 16 Punkte). `padding` wird oft dazu verwendet, um den Abstand zwischen Views zu vergrößern. Der `padding`-Abstand gilt normalerweise für alle vier

Ränder. Optional können Sie den Abstand aber auch gezielt nur an einzelnen Seiten anwenden, z. B. mit `.padding(.leading, 20)` oder '`.padding(.vertical, 6)`.

```
// Projekt hello-swiftui, Datei Container.swift
struct ContainerTest1: View {
    var body: some View {
        VStack {
            Text("Text 1")
                .background(.yellow)
            Text("Text 2")
                .background(.gray)
            HStack {
                Text("Text 3")
                    .background(.yellow)
                Text("Text 4")
                    .background(.gray)
            }
            .padding(8)
            .background(.blue)

            Text("Text 5")
                .padding() // 16 by default
                .background(.white)
        }
        .padding()          // 16 by default
        .background(.green)
    }
}
```

Abbildung 13.2 VStack- und HStack-Container

`padding` und `background` sind ein gutes Beispiel dafür, wie wichtig die Reihenfolge der Modifier ist. Im folgenden Listing wird »Text A« zuerst von einem 16 Punkt breiten Rand umgeben. Der Text samt Rand wird rot eingefärbt.

```
// um 20 Point vergrößerte Textbox mit rotem Hintergrund
Text("Text A")
    .padding(20)
    .background(.red)
```

Bei »Text B« ist dagegen nur der Texthintergrund blau. `padding` wird danach angewendet. Der Rand hat die Hintergrundfarbe des Containers, wirkt also durchsichtig.

```
// Textbox mit blauem Hintergrund, rundherum ein 20 Punkt
// breiter Rand in der Hintergrundfarbe des Containers
Text("Text B")
    .background(.blue)
    .padding(20)
```

Punkt versus Pixel

Absolute Maße werden grundsätzlich in Punkt ausgedrückt. Auf iPhones und iPads der ersten Generation entspricht ein Punkt einem Pixel. Bei Retina-Geräten entspricht ein Punkt pro Dimension zwei Pixeln, d. h., es werden $2 \times 2 = 4$ Pixel gezeichnet. Aktuelle Geräte haben eine noch höhere Auflösung, ein Punkt entspricht also noch mehr Pixeln. Punkt ist damit eine von der tatsächlichen Auflösung des Geräts unabhängige Einheit.

Mehrere Views mit ForEach erzeugen

In Containern und Listen wollen Sie oft in einer Schleife mehrere Views erzeugen (siehe Abbildung 13.3). Dabei hilft `ForEach`. Das ist kein grundlegendes Swift-Sprachelement, sondern vielmehr eine in der SwiftUI-Bibliothek definierte Struktur, die Elemente eines bestimmten Typs (im folgenden Code: `Text`-Views) produziert und zurückgibt. Sie übergeben an `ForEach` eine Aufzählung und geben in der nachfolgenden Closure an, wie aus der Schleifenvariable `$0` eine View erzeugt wird.

```
// Projekt hello-swiftui, Datei Layout.swift
// erzeugt zehn Texte mit bunten Hintergrundfarben
VStack {
    ForEach(0..<10) {
        Text("Text \($0)")
            .padding(8)
            .background(Color.random())
    }
}
```

```
// Erweiterung zu 'Color' zum Erzeugen einer zufälligen Farbe
extension Color {
    static func random() -> Color {
        Color(
            red: Double.random(in: 0...1),
            green: Double.random(in: 0...1),
            blue: Double.random(in: 0...1)
        )
    }
}
```

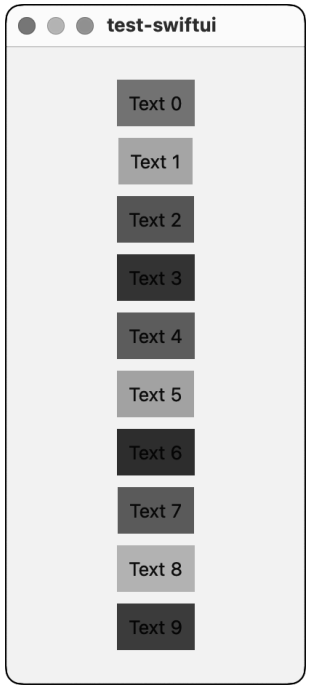


Abbildung 13.3 Zehn Texte mit zufälligen Hintergrundfarben in einem VStack, hier in einer macOS-App dargestellt

ForEach mit id-Parameter

Im obigen Beispiel produziert `VStack { ForEach(0..<10){ ... } }` zehn Textfelder. Wenn Sie den Wertebereich in `ForEach` auf `0...9` umstellen, funktioniert das Beispiel aber plötzlich nicht mehr! Die angezeigte Fehlermeldung wird Ihnen nicht weiterhelfen:

```
Cannot convert value of type 'ClosedRange<Int>' to expected
argument type 'Range<Int>'
```

Das Problem besteht darin, dass das SwiftUI-Framework jede View eindeutig identifizieren und zuordnen können muss. Der Zahlenbereich `0..<10` ist intern ein `Range<Int>`. Dieses entspricht dem Protokoll `RandomAccessCollection`, welches wiederum `Identifiable` implementiert. Mit anderen Worten: Jede von `0..<10` produzierte Zahl erfüllt den vom SwiftUI geforderten Eindeutigkeitsanspruch und ist für die Zuordnung ausreichend.

Wenn Sie dagegen den scheinbar gleichwertigen Zahlenbereich `0...9` verwenden, haben Sie intern eine Struktur vom Typ `ClosedRange<Int>`, der *nicht* dem `Identifiable`-Protokoll entspricht! (Warum sich Apple entschlossen hat, die beiden Range-Operatoren intern unterschiedlich zu implementieren, dürfen Sie mich nicht fragen.)

Es gibt mehrere Lösungen zu diesem Problem. In unserem Fall ist es am einfachsten, eben den Operator `..<` und nicht `...` einzusetzen. Aber oft wollen Sie mit `ForEach` gar keinen Zahlenbereich abbilden, sondern alle Elemente einer Aufzählung durchlaufen, also z. B. `ForEach(users){ user in ... }`. Der Idealfall besteht darin, dass die zugrunde liegende `User`-Klasse das Protokoll `Identifiable` implementiert, also selbst eine eindeutige `id`-Eigenschaft zur Verfügung stellt. Wenn das nicht der Fall ist, müssen Sie in `ForEach` den optionalen `id`-Parameter verwenden und – zumeist in der KeyPath-Syntax – einen Parameter angeben, der für alle Elemente eindeutig ist.

```
// setzt voraus, dass die User-Klasse eine email-Eigenschaft hat
// UND alle Benutzer eindeutige E-Mail-Adressen haben
ForEach(users, id: \.email) { ... }
```

Den `id`-Parameter können Sie natürlich auch verwenden, wenn Sie an `ForEach` einen Bereich mit dem Operator `...` übergeben. `\.self` verweist hier auf das gesamte gerade von `ForEach` produzierte Objekt, hier also eine Zahl zwischen 1 und 10. Wichtig ist nur, dass die Werte eindeutig sind!

```
// verwendet die Zahlenwerte zur Identifikation
ForEach(0..<10, id: \.self) {
    Text("Text \($0)")
}
```

Layout

Vielleicht ist es Ihnen aufgefallen: Die bisherigen Beispiele enthielten, abgesehen vom `padding`-Modifier, keine konkreten Layout-Anweisungen. Tatsächlich ist der Ansatz von SwiftUI diesbezüglich ganz anders als bei UIKit, wo es mit stundenlanger Arbeit verbunden war, passende Layoutregeln zu entwerfen. Bei SwiftUI überlassen Sie die Layoutdetails weitgehend dem Framework: Sie geben nur vor, welche Elemente wie relativ zueinander positioniert werden. Soweit möglich, sind Views automatisch so groß, um ihren Inhalt aufzunehmen. Die Größe einer `Text`-View hängt also vom Text und der Schriftgröße ab.

Ganz ohne manuelle Eingriffe gelingt aber auch in SwiftUI kein optimales Layout: Die Container `HStack` und `VStack` zur horizontalen und vertikalen Anordnung sowie den `padding`-Modifier zur Einstellung der Abstände zwischen den Views habe ich Ihnen bereits vorgestellt.

Offen ist bisher geblieben, wie Sie die Ausrichtung von Views steuern können. Nehmen wir an, Sie wollen, dass die Views in einem `VStack` nicht mittig ausgerichtet werden (das ist das Standardverhalten), sondern links- oder rechtsbündig. Eigentlich wäre zu erwarten, dass auch dieses Verhalten durch einen Modifier gesteuert werden kann – doch die Suche bleibt vergeblich.

Manche (zum Glück wenige) View-Eigenschaften können nur durch Parameter beeinflusst werden, die an die Init-Funktion der jeweiligen View übergeben werden. Das erscheint inkonsequent, hat aber technische Gründe. Init-Parameter beeinflussen fundamentale Eigenschaften von Views, die bereits bei der Erzeugung festgelegt und nicht später dynamisch verändert werden können. Die Ausrichtung von Views innerhalb eines Containers zählt da offensichtlich dazu.

```
// VStack mit drei Texten, die linksbündig ausgerichtet sind
VStack(alignment: .leading) {
    Text("lorem ipsum")
    Text("dolor")
    Text("sit")
}
```

Die drei Texte im `VStack` werden nun linksbündig angezeigt, der gesamte `VStack` aber weiter mittig im iPhone-Screen. Nehmen wir an, Sie wollen den ganzen `VStack` in der rechten unteren Ecke platzieren (siehe Abbildung 13.4). Das gelingt am einfachsten, indem Sie die Komponente neuerlich in einen `HStack` und einen `VStack` verpacken und als zusätzliches Element einen sogenannten `Spacer` hinzufügen. Das ist eine View, die einfach so viel Platz wie möglich einnimmt.

```
// Projekt hello-swiftui, Datei Layout.swift
HStack {
    Spacer()      // horizontal Platz ausfüllen
    VStack {
        Spacer()  // vertikal Platz ausfüllen
        VStack(alignment: .leading) {
            Text("lorem ipsum")
            Text("dolor")
            Text("sit")
        }
        .background(.yellow)
    }
}
```

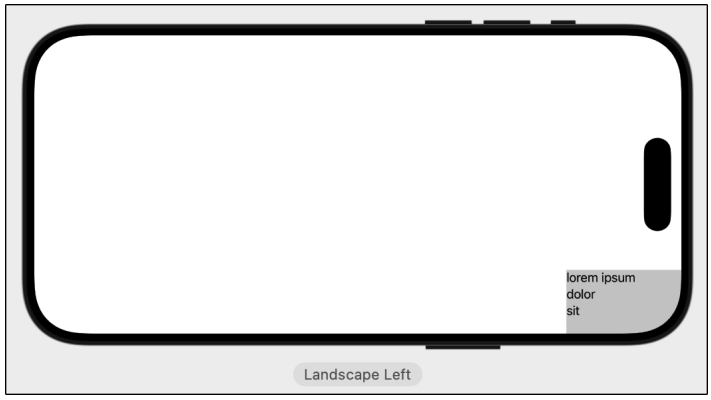


Abbildung 13.4 Linksbündiger Text in der rechten unteren Ecke einer iPhone-Vorschau im Querformat

In Ausnahmefällen können Sie die Größe einer View durch den `frame`-Modifier limitieren. Am häufigsten ist das bei der `Image`-View notwendig. Entsprechende Beispiele folgen in Abschnitt 13.4, »Image-Views«.

SwiftUI-Syntax

SwiftUI-Code sieht ein wenig anders aus als »gewöhnlicher« Swift-Code, wie Sie ihn in den bisherigen Kapiteln gesehen haben. Schon bei einfachen Beispielen entstehen unzählige Klammerebenen, wobei auf ersten Blick nicht immer klar ist, welchen syntaktischen Zweck die Klammern erfüllen. Relativ oft schließen Klammernpaare den Code zur Rückgabe eines Read-only-Properties ein (z. B. `body`). In anderen Fällen handelt es sich um Closures, deren Ergebnis an den (versteckten) letzten Parameter der Init-Funktion einer View übergeben wird. SwiftUI-Code nützt diverse Eigenheiten und Kurzschreibweise der Swift-Syntax aus, aber es handelt sich immer um *echten* Swift-Code.

```
struct ContentView: View {      // Code-Block für struct
   @State private var rotation = 0.0
   var body: some View {        // Read-only Computed Property
      VStack {                  // Closure für VStack-Init()
          ForEach(0..<10) {     // Closure für ForEach
              Text("Text \($0)")
          }
          Button("Rotieren") { // Closure für Button-Init()
              rotation += 45
          }
          .rotationEffect(.degrees(rotation))
      }
   }
}
```

Der obige Code enthält einige Elemente, die in diesem Kapitel noch nicht behandelt wurden: Auf Buttons gehe ich im Abschnitt 13.3, »Buttons und Optionen«, näher ein, auf State-Variablen in Kapitel 14, »State, Binding und Observable«.

Wenn die Anzahl der Klammerebenen zu groß wird (persönlich versuche ich, sechs Ebenen möglichst nicht zu überschreiten), sollten Sie einen Teil des Codes in eine eigene View auslagern. Xcode ist Ihnen dabei behilflich. Entsprechende Beispiele folgen in Abschnitt 14.6, »Code-Organisation und eigene Views«.

13.2 Text

In diesem und den folgenden Abschnitten versuche ich, einen ersten Überblick über die wichtigsten in der SwiftUI-Bibliothek vorgegebenen Views zu geben. Weitere Views, z. B. zur Darstellung von Listen oder zur Navigation zwischen verschiedenen App-Ansichten, folgen in den weiteren Kapiteln. In diesem Abschnitt beginne ich mit Views zur Darstellung und Eingabe von Text (siehe Abbildung 13.5).

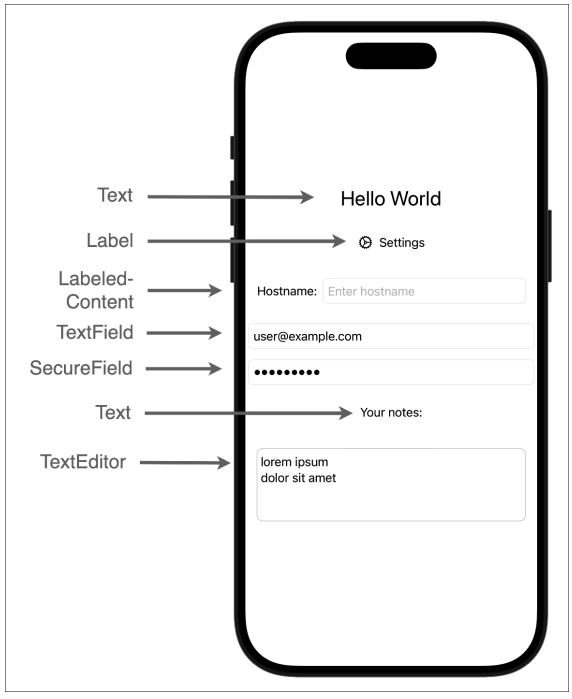


Abbildung 13.5 Grundlegende Text-Views

Das folgende Listing zeigt die Anwendung elementarer Text-Views. Die im Listing genannten Variablen müssen mit `@State` deklariert werden (z. B. `@State var email = ""`). Auf die Synchronisation zwischen Views und Variablen gehe ich in Kapitel 14,

»State, Binding und Observable«, ausführlich ein. Aus Platzgründen habe ich aus dem Listing diverse `padding`-Anweisungen entfernt.

```
// Projekt hello-swiftui, Datei Text.swift
VStack {
    // Text anzeigen
    Text("Hello World")
        .font(.title)
    // Text plus Icon
    Label("Settings", systemImage: "gear")
    // beschriftet die View in der nachfolgenden Closure
    LabeledContent("Hostname:") {
        TextField("Enter hostname", text: $hostname)
            .textFieldStyle(.roundedBorder)
    }
    // Texteingabe
    TextField("E-Mail", text: $email)
        .textFieldStyle(.roundedBorder)
        .keyboardType(.emailAddress)
    // Passworteingabe
    SecureField("Password", text: $password)
        .textFieldStyle(.roundedBorder)
    Text("Your notes:")
    // Text editor for multiline input
    TextEditor(text: $notes)
        .frame(height: 100)
        .overlay(
            RoundedRectangle(cornerRadius: 8)
                .stroke(Color.gray.opacity(0.4), lineWidth: 1)
        )
}
```

Text anzeigen

`Text` ist die populärste View zur Darstellung von Text. Der Text kann nicht verändert werden. Die Größe der View ergibt sich aus ihrem Inhalt, es sei denn, sie wird durch `frame`- oder `lineLimit`-Modifier beschränkt. In diesem Kapitel gab es bereits eine Menge Beispiele für die Anwendung dieser View.

Wenn Sie den Text mit einem Icon kombinieren möchten, bietet sich `Label` an. Über den Parameter `systemImage` können Sie wie im obigen Listing aus unzähligen vordefinierten Icons wählen. Alternativ können Sie mit dem Parameter `image` auf ein eigenes Bild verweisen, das sich in den Assets Ihrer App befindet (siehe auch Abschnitt 13.4, »Image-Views«). Eine dritte Option besteht darin, den Text und das darzustellende Bild selbst zu erzeugen. Das gibt Ihnen die größtmögliche Kontrolle über das Erschei-

nungsbild, ist syntaktisch aber umständlich. (Details zum Umgang mit der `Image`-View folgen in Abschnitt 13.4).

```
Label {
    Text("Login")
}
icon: {
    Image("my-custom-icon")
        .resizable()
        .scaledToFit()
        .frame(width: 20, height: 20)
}
```

Häufig verwenden Sie `Text` oder `Label` nur, um eine nachfolgende View zu beschriften. Für diesen Zweck können Sie auch `LabeledContent` verwenden. Der Vorteil: Das Erscheinungsbild der Beschriftung passt sich an den Stil des Kontexts an (z. B. einer Symbolleiste).

Markdown-Formatierung

Die `Text`-View hat ein verborgenes Feature: Sie berücksichtigt in der Zeichenkette enthaltene Markdown-Formatierungen und zeigt diese direkt an:

```
// formatiert 'bold' fett und 'italic' kursiv
Text("Text with **bold** and *italic* words.")
```

Swift betrachtet den Parameter der Init-Funktion von `Text` nicht als normale Zeichenkette, sondern als `LocalizedStringKey`, also als Zeichenkette, die bei mehrsprachigen Apps durch Texte in anderen Sprachen ausgetauscht wird (siehe auch Abschnitt 24.2, »Mehrsprachige Apps«). Die Verarbeitung grundlegender Markdown-Formatierungen gehört zu den `LocalizedStringKey`-Features.

Wenn Sie die automatische Formatierung nicht wünschen, stellen Sie den Markdown-Sonderzeichen \ voran, verwenden den `verbatim`-Parameter oder geben explizit an, dass die Zeichenkette als `String` betrachtet werden soll.

```
let txt = "**not bold**"  // txt ist ein String, kein
Text(txt)                 // LocalizedStringKey
Text(verbatim: "**not bold**")
Text("**not bold**" as String)
```

Damit die Markdown-Formatierung auch bei Texten funktioniert, die außerhalb von `Text` deklariert wurden, verwenden Sie explizit den Datentyp `LocalizedStringKey` oder verarbeiten die Zeichenkette mit `LocalizedStringKey(s)`. Die folgenden Zeilen mit einer mehrzeiligen Raw-Zeichenketten zeigen die Grenzen der Markdown-Logik (siehe Abbildung 13.6).

```
let msg : LocalizedStringKey = #"""
    Regular text.

      * **Bold** text.
      * *Italic* text.
      * ***Bold, italic*** text.

    `Code works also: <html>\a\b\c`

    ```
    but multi-line
    listings fail
    ```

    My website: kofler.info
    """#

Text(msg)
```

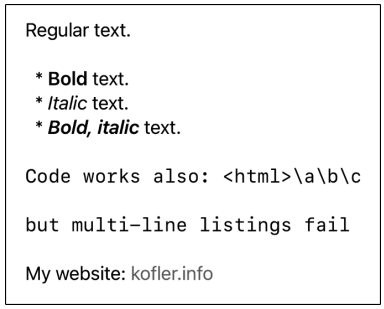


Abbildung 13.6 Markdown-Formatierung von Text

Texteingabe (TextField und SecureField)

Zur Eingabe einzeiliger Texte verwenden Sie normalerweise ein `TextField`. Der im ersten Parameter übergebene Text wird in grauer Schrift im Textfeld angezeigt, bis Sie mit der Eingabe beginnen. Der Text ist also ein Indikator für den Benutzer, welche Eingabe in dem Feld erwartet wird. Bei einfachen Formularen können Sie damit auf die explizite Beschriftung jedes Textfelds verzichten.

`TextField` nutzt standardmäßig die gesamte horizontale Breite, was bei einem iPhone im Portrait-Modus zweckmäßig ist. Wenn Sie ein `TextField` in einer iPad-App oder unter macOS verwenden, müssen Sie die Breite limitieren – entweder indirekt über einen Container oder direkt mit dem `frame`-Modifier (z. B. `frame(maxWidth: 400)`). Anders als bei `Text` ist es nicht möglich, die gewünschte Zeilenanzahl direkt festzulegen.

Standardmäßig ist ein `TextField` von einem dünnen rechteckigen Rahmen umgeben. Wenn Sie abgerundete Formen vorziehen, verwenden Sie den Modifier `textFieldStyle(.roundedBorder)`.

Zur Eingabe von Passwörtern verwenden Sie ein `SecureField`. Anstelle des eingegebenen Texts werden nur Punkte angezeigt.

Eingabe mehrzeiliger Text (TextEditor)

Für mehrzeilige Eingaben benötigen Sie einen `TextEditor`. Diese View nimmt automatisch den gesamten zur Verfügung stehenden Raum ein. Deswegen ist es oft notwendig, die Größe mit dem Modifier `frame` zu beschränken.

Wie `TextField` fehlt auch dem `TextEditor` eine optische Umrandung. Zudem bleibt der Modifier `textFieldStyle` wirkungslos. Um die View mit einem Rahmen zu umgeben, verwenden Sie am besten den `overlay`-Modifier und zeichnen damit ein abgerundetes Rechteck über die View (siehe das vorherige Listing).

Tastatur

Solange Sie Ihre Apps in der Xcode-Vorschau oder im Simulator ausführen, können Sie Texteingaben mit Ihrer Mac-Tastatur durchführen. Die Software-Tastatur von iOS erscheint gar nicht. Um realistischere Tests durchzuführen, können Sie diese Funktion im Menü des Simulators unter I/O • KEYBOARD deaktivieren. Alternativ führen Sie die App auf einem echten iPhone oder iPad aus.

iOS kennt verschiedene Tastaturvarianten zur Eingabe von beliebigem Text, E-Mail-Adressen, Zahlen bzw. Telefonnummern etc. Die gewünschte Variante steuern Sie über den Modifier `keyboardType`. Zulässige Einstellungen sind unter anderem `.emailAddress`, `.numberPad`, `.decimalPad`, `.phonePad` und `.URL`. Falls Sie Multi-Plattform-Apps entwickeln, dürfen Sie `keyboardType` nur für den iOS-Build verwenden!

```
// Email keyboard (nur iOS!)
TextField("Email", text: $email)
    #if os(iOS)
    .keyboardType(.emailAddress)
    #endif
```

Sofern es keine Bluetooth-Tastatur gibt, wird die iOS-Software-Tastatur bei Bedarf automatisch eingeblendet. Problematischer ist das Ausblenden: Nicht immer erkennt iOS, dass Ihre Eingabe abgeschlossen ist und Sie eigentlich gerne den von der Tastatur verborgenen Teil der App wieder sehen würden. In solchen Fällen müssen Sie die Tastatur explizit ausblenden.

Die Vorgehensweise ist leider umständlich: Sie müssen innerhalb Ihrer `ContentView` oder einer anderen App-Ansicht eine Variable mit dem Property Wrapper `@FocusState`

deklarieren. Sie steuert, ob ein Element Ihrer App den Tastaturfokus hat. Diese Variable synchronisieren Sie bei allen Eingabefeldern mit dem Modifier `focused`. Über einen Button oder über eine Berührung des Hintergrunds der App haben Sie nun die Möglichkeit, diese Variable auf `false` zu stellen und die Tastatur so auszublenden.

```
// Projekt hello-swiftui, Datei Text.swift
struct KeyboardHandling: View {
    @State private var password = ""
    @State private var email = ""
    @FocusState private var isFocused: Bool

    var body: some View {
        ZStack {
            // gesamten Hintergrund berührbar machen
            Color.clear
                .contentShape(Rectangle())
                .onTapGesture {
                    // Tastatur ausblenden
                    isFocused = false
                }
            // über dem Hintergrund: VStack mit den TextFields
            VStack {
                TextField("Email", text: $email)
                    .keyboardType(.emailAddress)
                    .focused($isFocused)
                SecureField("Password", text: $password)
                    .focused($isFocused)
                // ...
            }
        }
    }
}
```

Ein letztes Problem besteht darin, den ganzen Hintergrund der App berührbar zu machen. Am zuverlässigsten ist die im vorigen Listing skizzierte Variante, bei der in einem `ZStack` unter die eigentliche Oberfläche eine weitere Ebene platziert wird, die nur aus einer unsichtbaren Farbe besteht. `Color` erfüllt das `View`-Protokoll und gilt daher als gültiges Element im `ZStack`-Container. Die Ebene wird mit einem Rechteck ausgefüllt. Bei einer Berührung (`onTapGesture`) wird der Code in der nachfolgenden Closure ausgeführt.

13.3 Buttons und Optionen

In seiner Grundform ist die `Button`-View denkbar einfach anzuwenden: Sie übergeben als Parameter die Beschriftung und stellen den beim Anklicken auszuführenden Code als Closure hintan:

```
Button("Click me") {
  doSomething()
}
```

In der Praxis werden Sie in Buttons oft Aktionen auslösen, z. B. das Schließen eines Dialogs oder den Wechsel in eine andere Ansicht der App. Beispiele dazu folgen in den weiteren Kapiteln. Um einen Button einfach nur auszuprobieren, möchten Sie vermutlich irgendein optisches Feedback. Ich zeige Ihnen hier, wie Sie den Button mit jedem Klick um 45 Grad drehen. Dazu speichern Sie den Rotationszustand des Buttons in der Variablen und verwenden diese Variable, um den Modifier `rotationEffect` einzustellen. Auch wenn das Beispiel schon wieder in das Thema des nächsten Kapitels reicht, in dem ich Ihnen den Property Wrapper `@State` genau erkläre (siehe Kapitel 14, »State, Binding und Observable«), sollte der folgende Code leicht verständlich sein.

```
// Projekt hello-swiftui, Datei Button.swift
struct RotatingButton: View {
    @State private var rotation = 0.0

    var body: some View  {
        Button("Click me") {
            rotation += 45
        }
        .rotationEffect(.degrees(rotation))
    }
}
```

Button-Gestaltung

Zur optischen Gestaltung des Buttons stehen diverse Modifier zur Wahl. Das folgende Listing zeigt die Anwendung der drei wichtigsten:

```
Button("Click me") { doSomething() }
    .buttonStyle(.borderedProminent)  // ausgefüllt
    .controlSize(.large)              // größer
    .tint(.red)                       // rot
```

Alternativ können Sie mit dem `role`-Parameter die Funktion des Buttons beschreiben. Die beiden zulässigen Einstellungen sind `.destructive` und `.cancel`:

```
Button("Delete", role: .destructive) {
    deleteAllData()
}
```

Buttons müssen nicht unbedingt aus Text bestehen. Sie können jede beliebige View in einen Button verpacken. Dazu formulieren Sie nach `Button` zwei Closures, eine für die Reaktion und eine für die Gestaltung. Die Reihenfolge dieser Closures ist ein wenig verwirrend: Die Action-Closure muss zuerst angegeben werden! (Sie werden im Verlauf der folgenden Kapitel feststellen, dass das ein übliches SwiftUI-Muster ist: Wenn es zwei Closures gibt, ist die erste für die Aktion zuständig, die zweite für das Aussehen.)

```
Button {
    deleteData()
} label: {
    // rot ausgefüllter Kreis mit Mülleimer-Icon
    Image(systemName: "trash")
        .padding()
        .background(Color.red)
        .foregroundColor(.white)
        .clipShape(Circle())
}
```

Closure-Interna

Wie kommt die anfänglich irritierende Closure-Reihenfolge zustande? Hier lohnt es sich, einmal genauer auf die Deklaration der `Button`-Struktur zu sehen. In Xcode klicken Sie dazu mit cmd auf das Schlüsselwort `Button`. Ich habe hier nur die Grunddeklaration und eine Extension wiedergegeben, auf alle weiteren Extensions und Sonderfälle verzichtet.

```
// interne Deklaration der Button-Struktur
struct Button<Label> : View where Label : View {
    // für Button { myAction() } label: { returnView() }
    init(action: @escaping () -> Void,
         @ViewBuilder label: () -> Label)
}
// Convenience Init, akzeptiert Text als ersten Parameter
extension Button where Label == Text {
    // erlaubt Button("text") { myAction() }
    init(_ titleKey: String, action: @escaping () -> Void)
}
```

In `struct Button<Label>` wird also die allgemeingültige Init-Funktion für den Button definiert, zuerst mit dem `action`-Parameter, dann `label`. Die übliche Formulierung

```
Button {
    myAction()  // Reaktion
} label: {
    Image(...)  // Aussehen
}
```

ist eigentlich eine Kurzschreibweise mit nachgestellten Closures (siehe Abschnitt 7.5, »Closures«). Ohne Nachstellen müssen beide Parameter benannt werden, und der Code würde so aussehen:

```
Button(action: { myAction() },
       label:  { Image(...) })
```

Zuletzt noch eine Auffrischung zum `@escaping`-Attribut (siehe Abschnitt 12.2, »Attribute, Property Wrapper und Makros«): Das Attribut ist notwendig, weil die `action`-Closure nicht sofort ausgeführt werden soll, sondern erst später, wenn der Button tatsächlich berührt wird.

Button-Varianten: Toggle, Picker, Stepper und Slider

Zum klassischen Button gibt es einige mehr oder weniger offensichtliche Varianten, um Optionen bzw. Werte in einem vorgegebenen Bereich auszuwählen (siehe Abbildung 13.7). Im folgenden Listing präsentiere ich Ihnen die grundsätzliche Anwendung von `Toogle`, `Picker`, `Stepper` und `Slider`. Die Views sind in ein Formular mit mehreren Abschnitten verpackt. `Form` ist ein Container, der speziell zur Gestaltung von Dialogen oder Formularen gedacht ist. `Section` gliedert den Dialog in mehrere Bereiche und beschriftet diese gleich.

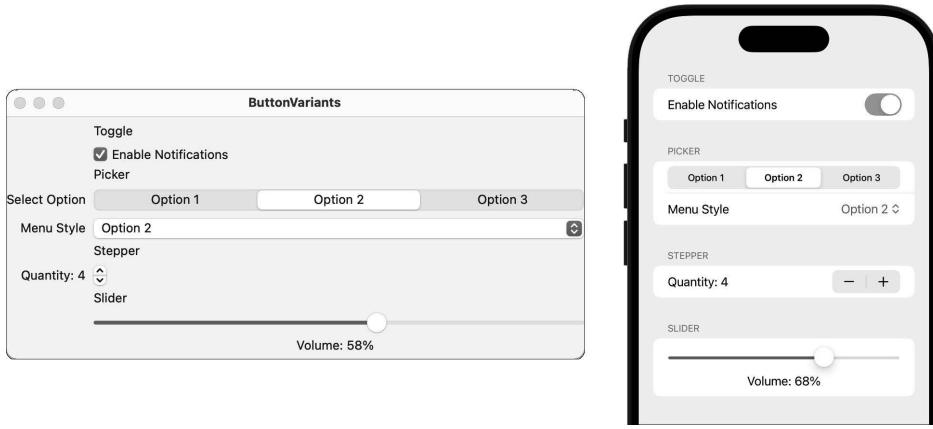


Abbildung 13.7 Derselbe Code, aber ganz unterschiedliches Erscheinungsbild unter macOS und iOS

```
// Projekt hello-swiftui, Datei Button.swift
struct ButtonVariants: View {
    @State private var isNotificationsEnabled = false
    // analog für weitere Variablen

    var body: some View {
        Form {
            // Toggle unter iOS = Checkbox unter macOS
            Section(header: Text("Toggle")) {
                Toggle("Enable Notifications",
                       isOn: $isNotificationsEnabled)
            }
            Section(header: Text("Picker")) {
                // Picker als Multi-Button
                Picker("Select Option",
                       selection: $selectedOption)
                {
                    Text("Option 1").tag(0)
                    Text("Option 2").tag(1)
                    Text("Option 3").tag(2)
                }
                .pickerStyle(.segmented)
                // synchroner Picker als Menü
                Picker("Menu Style", selection: $selectedOption)
                {
                    Text("Option 1").tag(0)
                    Text("Option 2").tag(1)
                    Text("Option 3").tag(2)
                }
                .pickerStyle(.menu)
            }
            // Stepper und Slider
            Section(header: Text("Stepper")) {
                Stepper("Quantity: \(quantity)",
                        value: $quantity, in: 1...10)
            }
            Section(header: Text("Slider")) {
                VStack {
                    Slider(value: $volume)
                    Text("Volume: \(Int(volume * 100))%")
                }
            }
        }
    }
}
```

Wenn Sie bei der Änderung eines Zustands Code ausführen möchten, ergänzen Sie die View um einen onChange-Modifier. Die nachfolgende Closure wird bei jeder Änderung des Status ausgeführt. onChange-Code sollte aber nur in Ausnahmefällen notwendig sein. Im Regelfall reicht es aus, den Zustand des Auswahlelements mit einer @State-Variablen zu verbinden. Einmal mehr: Auf die Hintergründe solcher Variablen gehe ich in Kapitel 14 näher ein.

```
@State private var selectedOption = 0

Picker("Select Option", selection: $selectedOption) {
    Text("Option 1").tag(0)
    ...
}
// diesen Code bei jeder Zustandsänderung ausführen
.onChange(of: selectedOption) {
    print("Picker-Auswahl: \(selectedOption)")
}
```

Checkboxes für macOS

Es gibt in SwiftUI keine Checkbox-View. In den iOS-Richtlinien für die Gestaltung von Oberflächen gibt es dieses Eingabeelement tatsächlich nicht mehr. Unter macOS übernimmt Toggle diese Rolle, wobei das Element automatisch wie eine herkömmliche Checkbox dargestellt wird (siehe Abbildung 13.7).

13.4 Bitmaps und Icons (Image)

Häufig wollen Sie in Ihrer App Bitmaps oder Icons darstellen. Genau diese Aufgabe erfüllt die Image-View. Im einfachsten Fall sieht die Verwendung so aus:

```
// Projekt hello-swiftui, Datei ImageAndCanvas.swift
VStack {
    // System-Icon in Defaultgröße
    Image(systemName: "gear")
    // Bitmap aus den Assets, individuelle Größe
    Image("camera-tripod")
        .resizable()
        .frame(width: 60, height: 60)
}
```

System-Icons

`Image(systemname: "xxx")` lädt ein Icon aus einer riesigen, in iOS bzw. macOS integrierten Sammlung von über 6000 vordefinierten Icons. Diese Symbole sind Teil der SF-Symbols-Bibliothek, die hier dokumentiert ist:

https://developer.apple.com/design/human-interface-guidelines/sf-symbols

Bei der Suche nach einem Icon und dem zugeordneten Namen hilft die App »SF Symbols«. Merkwürdigerweise ist das Programm nicht im App Store zu finden. Stattdessen muss es von der Apple-Website heruntergeladen werden:

https://developer.apple.com/sf-symbols

Die Symbole sind intern Schriftzeichen, weswegen ihr Erscheinen unter anderem mit dem `font`-Modifier verändert werden kann. Einige Beispiele:

```
Image(systemName: "square.and.arrow.down.badge.clock")
    .foregroundColor(.red)             // rot
    .font(.system(size: 48,            // 48 Punkt
                  weight: .semibold))  // halbfett
    .symbolVariant(.fill)              // ausgefüllt
    .symbolEffect(.bounce)             // animiert
```

`symbolEffect` kann diverse Animationseffekte einschalten, z. B. `.bounce`, `.breath`, `.pulse` etc. Da eine dauerhafte Animation schnell nervt, sollten Sie die Animation über eine `@State`-Variable zeitlich beschränken. `Task` und `await` sind allerdings ein Vorgriff auf die asynchrone Programmierung (siehe Kapitel 18, »Asynchrone Programmierung«).

```
@State private var isAnimating = false

Image(systemName: "wifi")
    .font(.largeTitle)
    .symbolEffect(.bounce, isActive: isAnimating)
    .onAppear {
        Task {
            // Animation starten
            isAnimating = true
            // und nach einer Sekunde wieder stoppen
            try? await Task.sleep(for: .milliseconds(1000))
            isAnimating = false
        }
    }
```

Assets

Um eigene Bilddateien mit einer App mitzuliefern, fügen Sie diese am besten per Drag-and-drop in eine Asset-Datei (Kennung `*.xcassets`) Ihres Xcode-Projekts ein. Das hat zwei Vorteile: Erstens können Sie so mehrere Bitmaps an einem Ort verwalten, ohne den Projektbaum unübersichtlich zu machen. Zweitens können die Bilder in mehreren Auflösungen gespeichert werden. Ihre App wird immer die Bitmap in der richtigen Auflösung verwenden. (Dieses Argument wird übrigens zunehmend hinfällig, weil es in der iOS-Welt keine Nicht-Retina-Geräte mehr gibt. Anders sieht es zugegebenermaßen bei macOS-Apps aus. Wie dem auch sei: Wenn Sie eine Bitmap nur in einer Auflösung haben, funktioniert das auf jeden Fall auch.)

Unterstützte Bildformate sind PNG, JPG und HEIC. Beim Zugriff auf Ihre Assets müssen Sie darauf achten, an `Image` den Namen *ohne* die Dateikennung zu übergeben – also `Image("mypicture")`, auch wenn der Dateiname `mypicture.jpg` lautet.

Bildgröße, Overlays und Clipping

Die `Image`-View versucht, die »richtige« Bildgröße zu erkennen, und macht die View genau so groß. Das klappt aber nur in seltenen Fällen so, wie Sie sich die Darstellung vorstellen. Die gewünschte Bildgröße erreichen Sie oft durch die Kombination der folgenden drei Modifier:

```
// Bildgröße vorgeben, gesamtes Bild unverzerrt darstellen
Image("assetname")
    .resizable()                      // veränderliche Größe
    .aspectRatio(contentMode: .fit)   // Proportionen erhalten
    .frame(height: 60)                // Höhe vorgeben
```

Mit `frame` geben Sie wahlweise die gewünschte Höhe oder Breite an. Wenn Sie umgekehrt den gesamten, maximal zur Verfügung stehenden Raum ausfüllen möchten, verwenden Sie `aspectRatio(contentMode: .fill)` in Kombination mit `clipped`:

```
// Bild in max. Größe anzeigen, an den Rändern beschneiden
Image("assetname")
    .resizable()
    .aspectRatio(contentMode: .fill)  // max. Raum nutzen
    .clipped()                        // Ränder beschneiden
```

Vielleicht wollen Sie das Bild mit Effekten darstellen. Im folgenden Beispiel wird das Bild zuerst mittig auf ein Quadrat mit 60 × 60 Punkten beschnitten, dann auf einen Kreis, der genau in dieses Quadrat passt. Mit dem `overlay`-Modifier wird der Kreis nun von einer 2 Punkt breiten, grauen Linie umgeben. Schließlich wird der gesamte Kreis von einem Schatten umgeben (siehe Abbildung 13.8).

```
// Bild kreisförmig ausschneiden und mit Rahmen und
// Schatten versehen
Image("assetname")
    .resizable()
    .scaledToFill()
    .frame(width: 60, height: 60)
    .clipShape(Circle())
    .overlay(Circle().strokeBorder(.gray, lineWidth: 2))
    .shadow(color: .black.opacity(0.5), radius: 10, x: 0, y: 5)
```

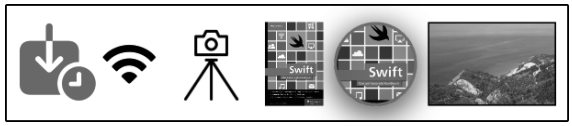


Abbildung 13.8 Einige Icons und Bilder

AsyncImage

Bis jetzt bin ich davon ausgegangen, dass die darzustellende Bilddatei schon vorliegt. Oft müssen Bilder aber zuerst aus dem Internet geladen werden. Dabei haben Sie zwei Optionen: Entweder kümmern Sie sich selbst um den Download-Prozess, oder Sie überlassen diese Arbeit inklusive Caching der `AsyncImage`-View. Das macht beim Programmieren weniger Arbeit, gibt aber nur eingeschränkte Möglichkeiten, den Prozess zu beeinflussen bzw. auf Download-Probleme zu reagieren.

`AsyncImage` erwartet drei Parameter: Eine `URL`-Instanz mit der Download-Adresse, eine Closure zur Verarbeitung des heruntergeladenen Images sowie eine weitere Closure, die während des Downloads einen Platzhalter anzeigt. Hier wird als Platzhalter die `ProgressView` angezeigt, ein rotierendes Symbol. Das heruntergeladene Bild wird schließlich von einem schwarzen Rahmen umgeben.

Beachten Sie, dass der Modifier `resizeable` sofort in der `content`-Closure aufgerufen werden muss. Eine spätere Anwendung bleibt wirkungslos. Die anderen hier eingesetzten Modifier können Sie wahlweise direkt auf die `image`-Variable oder wie im folgenden Listing auf das `AsyncImage` anwenden.

```
// Bild herunterladen, auf 60 Punkt Höhe skalieren und
// mit einem schwarzen Rand umgeben
AsyncImage(
    url: URL(string: "https://kofler.info/uploads/preber.jpg"),
    content: { image in
        image.resizable()  // resizable() unbedingt hier!
    },
    placeholder: { ProgressView() }
)
```

```
.aspectRatio(contentMode: .fit)
.frame(height: 60)
.border(.black, width: 1)
```

Wenn Sie dieses Beispiel mit dem vorigen vergleichen, fragen Sie sich vielleicht, warum ich zuvor den `overlay`-Modifier verwendet habe, hier aber `border` eingesetzt habe. Die Antwort ist einfach: `border` zeichnet einen Rahmen um die ganze View. Bei rechteckigen Formen passt das. Bei abgerundeten, kreisförmigen oder sonst wie besonders gestaltete Formen müssen Sie dagegen auf das etwas umständlichere `overlay`-Verfahren zurückgreifen.

13.5 Grafik (Canvas, Path und Shape)

Es gibt verschiedene Wege, um in SwiftUI-Oberflächen freie Grafikkommandos auszuführen. Ich stelle Ihnen hier ganz kurz drei Verfahren vor, ohne aber auf die Details der Grafikprogrammierung einzugehen.

Zum Zeichnen komplexer Grafiken ist die `Canvas`-View vorgesehen. Ihren Grafik-Code formulieren Sie in einer Closure, deren beide Parameter Ihnen die Größe des View-Rechtecks sowie den Grafikkontext zur Verfügung stellen. Im folgenden Code beschreibt das `Path`-Objekt den Umriss des zu zeichnenden Objekts – ein Dreieck (siehe Abbildung 13.9). Den Grafikkontext brauchen Sie in der Folge, um darauf Zeichenmethoden wie `fill` oder `stroke` für dieses `Path`-Objekt anzuwenden.

```
// Projekt hello-swiftui, Datei ImageAndCanvas.swift
// zeichnet ein gelbes Dreieck mit rotem Rand in einem Canvas mit
// einer Größe von 300x200 Punkt; der Canvas ist grau umrandet
Canvas { context, size in
    var path = Path()  // Dreiecks-Pfad zusammenstellen
    path.move(to: CGPoint(x: 0, y: size.height/2))
    path.addLine(to: CGPoint(x: size.width/2, y: 0))
    path.addLine(to: CGPoint(x: size.width, y: size.height))
    path.closeSubpath()
    // Pfad gelb füllen, rot umranden
    context.fill(path, with: .color(.yellow))
    context.stroke(path, with: .color(.red), lineWidth: 2)
}
.frame(width: 300, height: 200)
.border(.black, width: 1)
```

Wenn es Ihnen darum geht, wiederverwendbare Formen zu schaffen, bietet sich `Shape` an. Die folgenden Zeilen definieren zuerst die von `Shape` abgeleitete Struktur `Triangle` und wenden diese dann an. Es gibt übrigens einige vordefinierte `Shape`s, z. B. `Rectangle`, `Circle`, `RoundedRectangle` sowie `Capsule` (pillenförmig).

```
// bildet eine dreieckige Form innerhalb eines Rechtecks
struct Triangle: Shape {
    func path(in rect: CGRect) -> Path {
        Path { path in
            path.move(to: CGPoint(x: 0, y: rect.midY))
            path.addLine(to: CGPoint(x: rect.midX, y: 0))
            path.addLine(to: CGPoint(x: rect.maxX, y: rect.maxY))
            path.closeSubpath()
        }
    }
}
...
// nutzt die Triangle-Form
Triangle()
    .fill(.yellow)
    .stroke(.red, lineWidth: 2)
    .frame(width: 100, height: 100)
    .border(.gray, width: 1)
```

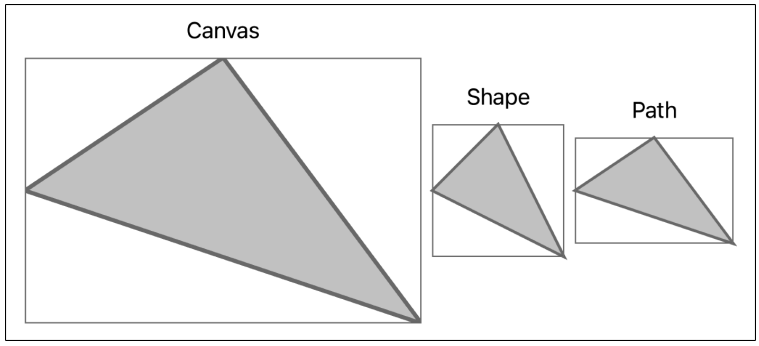


Abbildung 13.9 Rein optisch sehen alle drei Varianten (Canvas, Shape und Ad-hoc-Path) gleich aus.

Ein dritter Weg besteht darin, `Path` direkt zu verwenden, also ohne den Umweg über `Canvas` oder `Shape`. Das kann in einfachen Fällen den Code minimieren. Dabei ist zu beachten, dass `Path` an sich nicht dem `View`-Protokoll entspricht. Ein `Path`-Objekt wird aber `View`-kompatibel, sobald ein Modifier darauf angewendet wird.

```
// Ad-hoc-Path direkt zeichnen
Path { path in
    path.move(to: CGPoint(x: 60, y: 0))
    path.addLine(to: CGPoint(x: 0, y: 40))
    path.addLine(to: CGPoint(x: 120, y: 80))
    path.closeSubpath()
}
```

```
.fill(.yellow)  // Path -> View
.stroke(.red, lineWidth: 2)
.frame(width: 120, height: 80)
.border(.gray, width: 1)
```

13.6 Container (Stack, ScrollView, Grid)

Container brauchen Sie immer dann, wenn SwiftUI an einer Stelle im Code *eine* View erwartet, Sie aber mehrere Views darstellen möchten. Die zwei wichtigsten Container habe ich Ihnen in Abschnitt 13.1, »Grundlagen«, bereits vorgestellt: `VStack` platziert mehrere Views untereinander, `HStack` nebeneinander.

Standardmäßig werden die Views mittig zueinander angeordnet. Das können Sie mit dem optionalen Parameter `alignment` ändern. Die zulässigen Einstellungen sehen so aus:

- `VStack(alignment: xxx)`: `.center` (Defaultwert), `.leading`, `.trailing`
- `HStack(alignment: xxx)`: `.center` (Defaultwert), `.top`, `.bottom`, `.firstTextBaseline` und `.lastTextBaseline`

Die `TextBaseline`-Enumerationswerte helfen dabei, Texte mit unterschiedlichen Schriftgrößen optisch ansprechend auszurichten (siehe Abbildung 13.10).

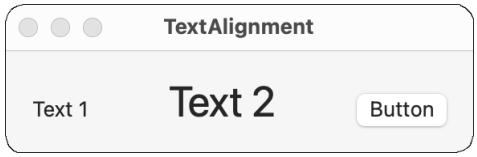


Abbildung 13.10 Die Texte sind an der Grundlinie ausgerichtet.

```
// Projekt hello-swiftui, Datei Container.swift
struct TextAlignment: View {
    var body: some View {
        HStack(alignment: .firstTextBaseline, spacing: 50) {
            Text("Text 1")
            Text("Text 2")
                .font(.largeTitle)
            Button("Button") { }
        }
        .padding()
    }
}
```

Abstandsteuerung in HStack und VStack (spacing, padding, Spacer)

Es gibt drei Mechanismen, um die Abstände zwischen den Elementen eines `HStack`s oder `VStack`s zu beeinflussen:

- Standardmäßig fügen beide Container einen kleinen Abstand zwischen den enthaltenen Views ein. Mit dem optionalen `spacing`-Parameter verkleinern oder vergrößern sie diesen Abstand bei Bedarf (siehe das obige Listing). Der Abstand gilt nur zwischen den Views, nicht am Anfang und am Ende.
- Mit dem Modifier `padding`, den Sie auf die im Container enthaltenen Views anwenden, können Sie bei jedem Element individuell Abstände in alle Richtungen vorgeben.
- Schließlich können Sie zwischen den Elementen `Spacer` einfügen. Das sind unsichtbare Views, die sich so viel Platz nehmen wie sie bekommen. Die folgende Kombination führt dazu, dass der `VStack` den größtmöglichen Bereich ausfüllt und der Text darin vertikal mittig dargestellt wird:

```
VStack {
    Spacer()
    Text("Text").border(.red)
    Spacer()
}
.border(.blue)
```

Views übereinanderlegen (ZStack)

Mit dem `ZStack` können Sie mehrere Views übereinanderlegen. *z* bezieht sich hier auf die Z-Achse, die in die Tiefe geht. Die zuerst im Container angegebenen Views werden ganz hinten angeordnet. Eine typische Anwendung für den `ZStack` ist die Gestaltung eines Hintergrunds, auf dem weitere Views platziert werden sollen.

```
ZStack {
    Color.yellow             // zuerst ein gelber Hintergrund
        .ignoresSafeArea()   // über die üblichen Grenzen hinaus
    Image(systemName: "star.fill")       // darüber ein
        .resizable()                     // weißer Stern
        .frame(width: 100, height: 100)
        .foregroundColor(.white)
    Text("Lorem ipsum\ndolor sit amet")  // und schließlich
        .font(.largeTitle)               // ein Text
}
```

ScrollView

Wenn die in einem Container angezeigten Views die Screen- oder Fenstergröße (macOS) überschreiten, werden sie einfach abgeschnitten. Es ist dann oft unmöglich, die Elemente zu bedienen. Abhilfe schafft die `ScrollView`, die virtuell beliebig viel Platz zur Verfügung stellt (siehe Abbildung 13.11). Die `ScrollView` arbeitet normalerweise vertikal, kann aber auch horizontal oder in beiden Richtungen eingesetzt werden.

```
ScrollView { ... }                              // vertikal
ScrollView(.horizontal) { ... }                 // horizontal
ScrollView([.horizontal, .vertical]) { ... }  // beides
```

In der `ScrollView`-Closure können Sie beliebig viele Views anordnen. Die `ScrollView` kümmert sich aber nicht um das Layout. Deswegen werden mehrere Elemente meist in einen `VStack` oder `HStack` verpackt (oder in deren Varianten `LazyV/HStack`, die ich gleich behandle).

Der oder die Scrollbalken werden von macOS/iOS nur während einer Scroll-Bewegung eingeblendet. Diese Anzeige können Sie mit `ScrollView(showsIndicators: false)` ganz unterbinden. Umgekehrt ist es aber nicht möglich, eine ständige Darstellung der Balken zu erzwingen.

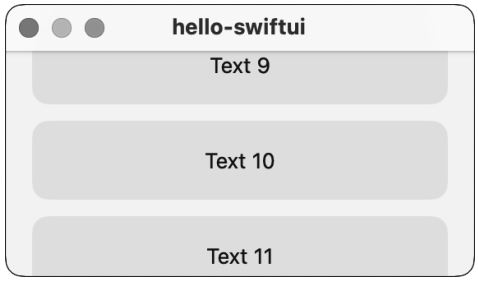


Abbildung 13.11 Darstellung von 20 Textfeldern

```
// Projekt hello-swiftui, Datei Container.swift
ScrollView {
    VStack(spacing: 10) {
        ForEach(1..<20) { number in
            Text("Text \(number)")
                .frame(maxWidth: .infinity)
                .padding()  // für die Texte
                .background(Color.blue.opacity(0.1))
                .cornerRadius(10)
        }
    }
    .padding() // für die ScrollView
}
```

Der Beispielcode zeigt eine einfache Anwendung der `ScrollView`. Der Modifier: `frame(maxWidth: .infinity)` bewirkt, dass jedes Textfeld die maximale Breite seines Containers nutzt, also z. B. die Breite des iPhone-Displays oder eines macOS-Fensters. Die Kombination aus `padding`, `background` und `cornerRadius` mit einer transparenten Hintergrundfarbe ergibt eine ansprechende optische Gestaltung.

LazyHStack und LazyVStack

`HStack` und `VStack` erzeugen sofort *alle* darin dargestellten Views und kümmern sich um deren Layout. Solange relativ wenige Views im Spiel sind, ist das der beste Ansatz. Aber nehmen Sie an, Sie wollen nicht wie im vorigen Beispiel 20 Texte anzeigen, sondern 2000! Es würde eine Menge Zeit und Speicherplatz brauchen, zuerst alle `Text`-Views zu erzeugen, nur um dann (vorerst) gerade einmal die ersten fünf oder zehn Elemente anzuzeigen.

Für solche Fälle sind der `LazyHStack` bzw. dessen Variante `LazyVStack` gedacht. Diese Container erzeugen immer nur die Elemente, die gerade sichtbar sind. Das funktioniert sogar dann, wenn die Größe der Elemente variiert! Die Anwendung der `Lazy`-Varianten von `HStack` und `VStack` ist nur innerhalb einer `ScrollView` sinnvoll.

```
// Projekt hello-swiftui, Datei Container.swift
ScrollView {
    LazyVStack(spacing: 10) {
        ForEach(1..<2001) { number in
            // erzeugt Textfelder in unterschiedlicher Höhe
            let height = CGFloat(20  + 10 * (number % 5))
            Text("Text \(number)")
                .frame(maxWidth: .infinity)
                .frame(height: height)
        }
    }
}
```

Darstellung von Listen

Bevor Sie einen `LazyVStack` einsetzen, sollten Sie sich überlegen, ob Sie Ihr Ziel nicht besser mit einer für die Darstellung von Listen optimierten View erreichen. Auf dieses in der Praxis sehr wichtige Thema gehe ich in Kapitel 15, »Listen«, ausführlich ein.

Form, Grid und Table

SwiftUI kennt noch mehr Container, auf die ich hier aber nur in aller Kürze eingehe. Da ist einmal `Form` zur Gestaltung von umfangreichen Dialogen. Im Prinzip leiten Sie den Dialog mit `Form` ein und gliedern und beschriften dann die diversen Optionen oder Eingabefelder mit `Section`. Ein Anwendungsbeispiel habe ich Ihnen bereits in Abschnitt 13.3, »Buttons und Optionen«, gezeigt (siehe Abbildung 13.7).

Mit `Grid` können Sie andere Views in einem Raster anordnen (siehe Abbildung 13.12). Der folgende Beispiel-Code illustriert nur die Basissyntax von `Grid` und `GridRow`. Darüber hinaus gibt es unzählige Gestaltungsmöglichkeiten. Beispielsweise können Sie bei Views innerhalb des Grids mit Modifiern wie `gridColumnAlignment` die Ausrichtung innerhalb der Zelle beeinflussen. `Divider` baut horizontale Trennlinien ein etc.

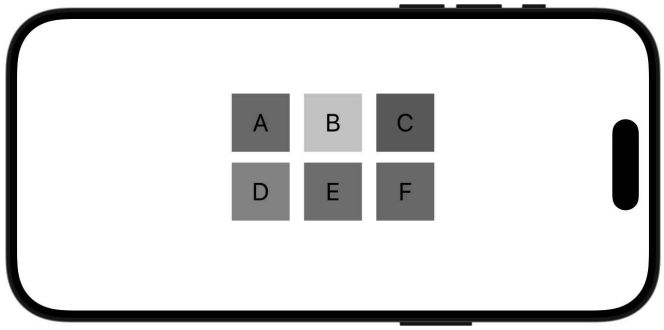


Abbildung 13.12 »Grid« platziert Views rasterförmig.

Um eine Menge Wiederholungen im Code der Textzellen zu vermeiden, habe ich hier mit `TextSquare` eine eigene View definiert. Der Code sollte ohne weitere Erklärungen verständlich sein. Hintergründe zu eigenen Views folgen dann in Abschnitt 14.6, »Code-Organisation und eigene Views«.

```
// Projekt hello-swiftui, Datei Container.swift
struct GridTest: View {
    var body: some View {
        Grid(alignment: .left,
             horizontalSpacing: 20,
             verticalSpacing: 15)
         {
            GridRow {
                TextSquare(txt: "A", color: .red)
                TextSquare(txt: "B", color: .yellow)
                TextSquare(txt: "C", color: .blue)
            }
```

```
            GridRow {
                TextSquare(txt: "D", color: .cyan)
                TextSquare(txt: "E", color: .gray)
                TextSquare(txt: "F", color: .red)
            }
        }
    }
}
struct TextSquare: View {
    let txt: String
    let color: Color
    var body: some View {
        Text(txt)
            .font(.largeTitle)
            .frame(width: 80, height: 80)
            .background(color)
    }
}
```

Zu `Grid` gibt es die Variante `LazyVGrid`, die vor allem zur vertikalen Darstellung großer, mehrspaltiger Foto-Galerien gedacht ist. Der folgende Code erzeugt zuerst ein Array mit 1000 Farben (siehe Abbildung 13.13).

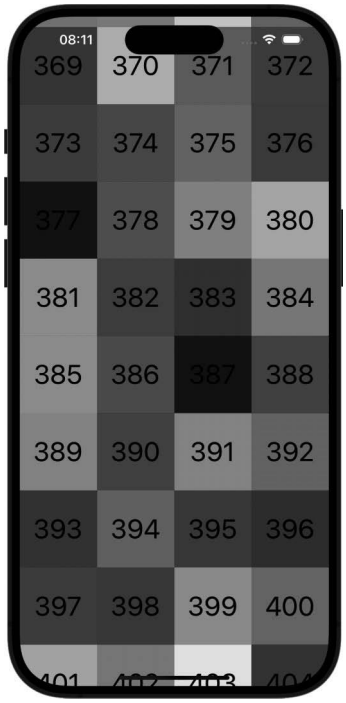


Abbildung 13.13 Farbenfrohe Demonstration des LazyVGrid-Containers

```
// Projekt hello-swiftui, Datei Container.swift
struct LazyVGridTest: View {
    // 1000 zufällige Farben speichern
    let colors: [Color] = (0..<1000).map { _ in
        Color.random()
    }
    // Grid-Layout mit vier Spalten ohne Rand
    let columns = [
            GridItem(.flexible(minimum: 0), spacing: 0),
            GridItem(.flexible(minimum: 0), spacing: 0),
            GridItem(.flexible(minimum: 0), spacing: 0),
            GridItem(.flexible(minimum: 0), spacing: 0)
        ]
    // LazyVGrid in ScrollView
    var body: some View {
        ScrollView {
            LazyVGrid(columns: columns, spacing: 0) {
                ForEach(1..<1001) { index in
                    Rectangle()
                        .fill(colors[index])
                        // quadratisch und die Spalte ausfüllend
                        .aspectRatio(1, contentMode: .fit)
                        .overlay(  // Beschriftung
                            Text("\(index)").font(.largeTitle)
                        )
                }
            }
        }
    }
}
```

Bei diesem Beispiel müssen die Farben vorweg dauerhaft in einem Array gespeichert werden. Eine dynamische Farbzuordnung würde dazu führen, dass sich die Zuordnung der Farben zu den Feldern während des Scrollens verändert. Das `columns`-Array mit vier `GridItem`-Elementen beschreibt das Aussehen der Spalten. Das `LazyVGrid` muss in einer `ScrollView` platziert werden.

Group

Die `Group`-View ist zwar kein Container, hilft aber dabei, mehrere Views gemeinsam zu verarbeiten. `Group` wird oft dazu eingesetzt, die gleichen Modifier auf mehrere Views anzuwenden und so redundanten Code zu vermeiden.

```
Group {
    Text("lorem")
    Text("ipsum")
    Text("dolor")
    Text("sit")
}
.padding(8)
.font(.caption)
.foregroundColor(.red)
```

Es gibt zwei weitere Anwendungsfälle für `Group`: Manchmal wollen Sie mehrere Views übergeben, wo nur eine View erwartet wird. `Group` löst dieses Problem zumindest auf syntaktischer Ebene. Aber auch an Stellen, an denen mehrere Views erlaubt sind (z. B. in einem `VStack`) kann `Group` weiterhelfen: Aus Performance-Gründen akzeptiert Swift maximal zehn im Code angegeben Views. Mit `Group` können Sie dieses Limit überschreiten.

13.7 Farbe, Datum und Uhrzeit auswählen (Date- und ColorPicker)

In iOS/macOS gibt es vorgefertigte Dialoge für Ja/Nein-Entscheidungen, zur Auswahl von Farben sowie zur Einstellung von Datum und Uhrzeit. Um die Nutzung dieser Dialoge bzw. entsprechender Views geht es in diesem Abschnitt.

Standarddialoge für Ja/Nein-Entscheidungen

Die Anzeige der Standarddialoge für Warnungen (Alert) und Bestätigung (Confirmation, siehe Abbildung 13.14) erfolgt über Modifier, die Sie der Grund-View hinzufügen – im folgenden Listing einem `VStack` für die Buttons der App. Die Sichtbarkeit steuern Sie durch `@State`-Variablen. Im folgenden Code werden diese in den Button-Action-Closures gesetzt. Die Integration der Dialoge in Modifiern hat wohl technische Gründe. Der resultierende Code ist leider alles andere als übersichtlich – selbst bei dem hier präsentierten minimalistischen Beispiel.

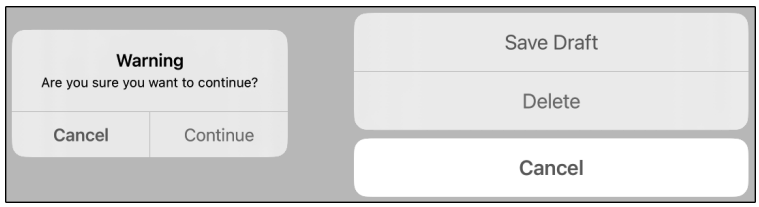


Abbildung 13.14 Links die Alert-Rückfrage in iOS-Optik, rechts der Bestätigungsdialog

```
// Projekt hello-swiftui, Datei Dialogs.swift
@State private var showingAlert = false
@State private var showingConfirm = false
...
VStack(spacing: 20) {
    // Buttons, um die Dialoge anzuzeigen
    Button("Show Alert") {
        showingAlert = true
    }
    Button("Show Confirmation Dialog") {
        showingConfirm = true
    }
}
// Dialog-Modifier für die gesamte View (hier VStack)
.alert("Warning", isPresented: $showingAlert) {
    Button("Cancel", role: .cancel) { }
    Button("Continue", role: .destructive) {
        // Code hier ausführen, wenn 'Continue'
    }
} message: {
    Text("Are you sure you want to continue?")
}
.confirmationDialog("Choose an action",
                    isPresented: $showingConfirm)
{
    Button("Save Draft") {
        // Code hier ausführen, wenn 'Save Draft'
    }
    Button("Delete", role: .destructive) {
        // Code hier ausführen, wenn 'Delete'
    }
    Button("Cancel", role: .cancel) { }
}
```

Farbauswahl

Mit dem `ColorPicker` bauen Sie eine Art Button in Ihren Dialog ein. Per Berührung oder Mausklick kann der Benutzer dann eine Farbe auswählen. Im folgenden Beispiel-Code wird die Farbe auf den Hintergrund des Screens angewendet. Der Modifier `fixedSize` bewirkt hier, dass die Beschriftung des Color Pickers und der dazugehörende Farb-Button kompakt dargestellt werden.

```
// Projekt hello-swiftui, Datei Dialogs.swift
@State private var backgroundColor = Color.white
...
ZStack {
    // Hintergrund für den ganzen Bildschirm
    backgroundColor.ignoresSafeArea()

    // Hintergrundfarbe ändern
    ColorPicker("Choose background color",
                selection: $backgroundColor)
    .fixedSize()
}
```

Datum und Uhrzeit

Die Anwendung des `DatePickers` zur Einstellung von Datum und Uhrzeit (siehe Abbildung 13.15) ist denkbar unkompliziert.

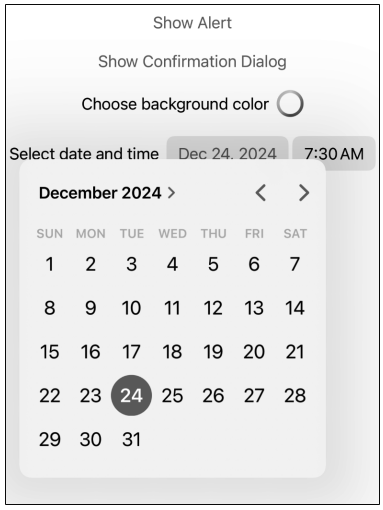


Abbildung 13.15 Der »Date-Picker« zur Auswahl eines Datums, hier auf einem iPhone mit US-Regionaleinstellungen

```
// Projekt hello-swiftui, Datei Dialogs.swift
@State private var selectedDateTime = Date()
...
DatePicker(
    "Select date and time",
    selection: $selectedDateTime,
    displayedComponents: [.date, .hourAndMinute]
)
.fixedSize()
```

Im Gegensatz zum `UIDatePicker` aus UIKit besteht beim `DatePicker` von SwiftUI aktuell aber keine Möglichkeit, bei den Minuten ein Intervall vorzugeben (z. B. `minuteInterval = 15`, wenn die Zeit nur auf 15 Minuten genau eingestellt werden soll). Diese Einschränkung können Sie umgehen, indem Sie in Ihren `View`-Code eine Init-Funktion einbauen, die die UIKit-Eigenschaft verändert. Wichtig ist, dass diese Einstellung gesetzt wird, bevor Sie den SwiftUI-`DatePicker` nutzen. Beachten Sie, dass diese Einstellung iOS-spezifisch ist und unter macOS nicht verwendet werden kann.

```
// Quelle: https://stackoverflow.com/questions/58976654
init() {
    UIDatePicker.appearance().minuteInterval = 15
}
```

Anstelle über einen Dialog können Sie Datum und Uhrzeit auch über virtuelle Räder einstellen (`datePickerStyle(.wheel)`), die iOS einblendet. Wenn Sie diese Variante vorziehen, ist die Integration in den Code etwas umständlicher. Im folgenden Code aktiviert ein Button die Sichtbarkeit des Elements, das mit einer `NavigationView` in einen eigenen Dialog verpackt ist. Die Zeiteinstellung schließen Sie mit dem Button Done ab.

```
// Projekt hello-swiftui, Datei Dialogs.swift
@State private var selectedDateTime = Date()
@State private var showingPicker = false
...
Button("Select Date & Time") {
    showingPicker = true
}
.sheet(isPresented: $showingPicker) {
    NavigationView {
        DatePicker("Select", selection: $selectedDateTime)
            .datePickerStyle(.wheel))
            .navigationBarItems(trailing: Button("Done") {
                showingPicker = false
            })
    }
}
```

13.8 Ereignisse (Gestures)

Spätestens mit der Verwendung des ersten Buttons in einer App kennen Sie das Konzept von SwiftUI, dass Sie zur Reaktion auf bestimmte Ereignisse den Code in einer Closure formulieren. Je nach Steuerelement und Ereignis können Sie den Code direkt an die Init-Funktion übergeben (`Button`) oder müssen einen Ereignis-Modifier wie `onTapGesture` oder `onLongPressGesture` verwenden.

```
// Projekt hello-swiftui, Datei Gestures.swift
Button("Touch me") { // Gesture-Closure
    // Code zur Reaktion auf Click/Tap
}
Text("Touch me")
    .onTapGesture {
        // Code zur Reaktion auf Click/Tap
    }
Text("Touch me at least 2 seconds")
    .onLongPressGesture(minimumDuration: 2.0) {
        // Code zur Reaktion auf lange Berührung
    }
```

iOS und macOS kennen unzählige weitere Gesten, die je nach Betriebssystem auf dem Display oder auf dem Touchpad auszuführen sind: Drag/Swipe-Bewegungen mit einem oder mehreren Fingern, Drehen, Verkleinern (zwei Finger zusammenziehen), Vergrößern (Finger auseinanderziehen) usw. Anstatt für jede Gesture einen eigenen Modifier zu schaffen, gibt es zur Verarbeitung derartiger Ereignisse einen universellen `gesture`-Modifier. Dort bauen Sie dann die gewünschte `Gesture` ein.

```
// Projekt hello-swiftui, Datei Gestures.swift
@State private var scale = 1.0
...
var body: some View {
    Rectangle()
        .fill(.green)
        .frame(width: 200, height: 200)
        .scaleEffect(scale)
        .gesture(
            MagnifyGesture()
                .onChanged { value in
                    // Skalierungsfaktor des Rechtecks ändern
                    scale = value.magnification
                }
            )
        .overlay(Text("Shrink / Expand"))
}
```

Das obige Listing zeigt im Zentrum des Bildschirms ein grünes Rechteck an. Dessen Ausmaße können mit der `MagnifyGesture` verkleinert oder vergrößert werden. Der Skalierungsfaktor wird in der `@State`-Variablen `scale` gespeichert (siehe auch Kapitel 14, »State, Binding und Observable«).

Klick, Doppelklick und Drag in vier Richtungen

Wenn Sie in einer App verschiedene Arten von Gestures erkennen und verarbeiten müssen, wird der Code rasch unübersichtlich. Das Ziel des folgenden Beispiels besteht darin, im gesamten Bildschirm einfache Berührungen (Taps), Doppel-Taps sowie Drag-Bewegungen in vier Richtungen zu erkennen.

Der Code beginnt mit der Definition einer Enumeration für die vier Richtungen. Die Enumeration enthält auch eine Methode, die den Bewegungsvektor auswertet und der dominierenden Richtung zuordnet. Die State-Variable `lastGesture` merkt sich den Namen des zuletzt erkannten Ereignisses. Dieser Name wird in einem Overlay-Text in der Bildschirmmitte angezeigt.

Die Ereignisse werden im gesamten Hintergrundbereich der App verarbeitet. `Color.clear` erzeugt einfach einen unsichtbaren Hintergrund, auf den die Gesture-Modifier angewendet werden. Damit das Programm auch unter macOS korrekt funktioniert, muss der `onTapGesture`-Modifier für Doppelklicks *vor* dem für einzelne Klicks angewendet werden. (Für iOS spielt die Reihenfolge dagegen keine Rolle.)

```
// Projekt hello-swiftui, Datei Gestures.swift
// Enumeration für Swipe-Richtungen
enum SwipeDirection {
    case left, right, up, down
    var description: String {
        switch self {
            case .left:  return "left"
            case .right: return "right"
            case .up:    return "up"
            case .down:  return "down"
        }
    }
    // Methode, die dem Bewegungsvektor die dominante
    // Richtung zuordnet
    static func from(translation: CGSize) -> SwipeDirection {
        if abs(translation.width) > abs(translation.height) {
            return translation.width  < 0 ? .left : .right
        } else {
            return translation.height < 0 ? .up : .down
        }
    }
}
...
@State private var lastGesture = "No gesture detected"
...
```

```
// gesamter Hintergrund empfängt Gesture-Ereignisse
Color.clear
    .overlay(Text(lastGesture))  // Ereignis anzeigen
    .contentShape(Rectangle())
    .onTapGesture(count: 2) {    // vor onTapGesture !
        lastGesture = "Double tap detected"
    }
    .onTapGesture {
        lastGesture = "Single tap detected"
    }
    .gesture(
        DragGesture(minimumDistance: 50)
            .onEnded { value in
               let direction =
                   SwipeDirection.from(
                      translation: value.translation)
               lastGesture = "Swipe \(direction.description)"
            }
    )
    .ignoresSafeArea()
```

13.9 Vorschau (Preview-Optionen)

Während der Entwicklung werden Sie Ihre neue App natürlich gelegentlich in einem Simulator oder auch auf einem iPhone oder iPad ausführen (siehe Abschnitt 13.10). In vielen Fällen reicht es aber aus, eine Ansicht Ihrer App im Preview-Fenster von Xcode anzusehen bzw. auszuprobieren. Der größte Vorteil dieser Arbeitsweise ist die Geschwindigkeit: Solange die Preview-Funktion aktiv ist, wird die Vorschau nahezu verzögerungslos mit jeder Code-Änderung aktualisiert. Das ist gerade bei Layout-Optimierungen sehr hilfreich.

Neue Xcode-Projekte enthalten bereits Preview-Code für die `ContentView`. Um die Vorschau für andere Views zu aktivieren, bauen Sie Zeilen nach dem folgenden Muster in Ihre Code-Datei ein:

```
#Preview {
    MyView()  // Name der anzuzeigenden View
}
```

Entsprechende Zeilen dürfen sich in *jeder* SwiftUI-Datei befinden. Sie können dabei auf selbst definierte View-Strukturen verweisen, auch wenn diese in anderen Dateien deklariert sind. Ebenso ist es zulässig, in einer Code-Datei mehrere Vorschauen einzubauen. In diesem Fall ist es zweckmäßig, die Preview-Blöcke zu beschriften, indem Sie eine Zeichenkette als Parameter übergeben; Xcode zeigt diese Beschriftung dann

auch im Preview-Fenster an und ermöglicht es Ihnen, blitzschnell zwischen der gerade aktiven Vorschau zu wechseln. Das Beispielprojekt zu diesem Kapitel macht von dieser Möglichkeit intensiv Gebrauch (siehe Abbildung 13.16).

```
// Projekt hello-swiftui, Datei Gestures.swift
#Preview("Pre-defined actions") {
    GestureTest()
}
#Preview("Tap and Swipe") {
    TapAndSwipe()
}
```

Xcode-Preview-Funktionen

Xcode kann mehr, als eine Vorschau Ihrer gesamten App bzw. einer speziellen Ansicht auszuführen! So können Sie unterschiedliche Geräte oder Modelle auswählen (macOS-Fenster, iPhone, iPad usw.), Ihre App im Querformat, im Dark Mode oder mit vergrößerter Schrift ausführen. Wenn Sie möchten, können Sie sogar mehrere Varianten parallel nebeneinander darstellen, also z. B. das Aussehen Ihrer App im Light und im Dark Mode *gleichzeitig* testen (siehe Abbildung 13.16). Probieren Sie es einfach aus!

Abbildung 13.16 Gleichzeitige Anzeige mehrerer Preview-Ansichten

Die Vorschau einer besonders wichtigen Datei können Sie anpinnen; damit bleibt die Preview-Ansicht selbst dann zugänglich, wenn Sie in eine andere Code-Datei wechseln.

Sollten Sie die Orientierung in Ihrem Code und den dort erzeugten Views verloren haben, aktivieren Sie den Modus SELECTABLE. Zwar können Sie Ihr Programm nun nicht mehr steuern bzw. bedienen, aber dafür führt ein Klick auf ein sichtbares Element direkt zum zugrunde liegenden Code.

Preview-Funktionen per Code steuern

Einige Funktionen können Sie auch mit Modifiern steuern, die Sie im `Preview`-Code auf die betreffende View anwenden. Das folgende Listing gibt dafür einige Beispiele:

```
// Projekt hello-swiftui, Datei Preview.swift
// Farbmodus
#Preview("Dark Mode") {
    MyView()
        .preferredColorScheme(.dark)
}
// Fenstergröße (nur für macOS!)
#Preview("Custom Window Size") {
    MyView()
        .frame(width: 600, height: 300)
}
// Sprache/Lokalisierung
#Preview("Localization") {
    MyView()  // Preview mit französischen Spracheinstellungen
        .environment(\.locale, .init(identifier: "fr"))
}
// Preview von mehreren Views mit unterschiedlichen Parametern
#Preview("Different States") {
    VStack(spacing: 20) {
        MyView(initialText: "Empty")
        MyView(initialText: "With some content")
        MyView(initialText: "A really long piece of
                             text that might overflow")
    }
}
```

Das letzte Beispiel zeigt, dass Sie innerhalb des `Preview`-Codes mittels eines Containers mehrere Views oder die gleiche View mit unterschiedlichen Parametern darstellen können.

Spracheinstellungen im Build-Schema

`.environment(\.locale, .init(identifier: "xx_XX"))` ändert zwar die Namen von Wochentagen und Monaten, nicht aber die restliche Formatierung von Daten (also z. B. »13:30« versus »1:30 PM» oder »31. Oct.« versus »Oct. 31st«). Damit Sie eine App vollständig in einer bestimmten Sprache ausprobieren können, müssen Sie die Xcode-Build-Einstellungen ändern: Product • Scheme • Edit Scheme, dann das Schema Run/Debug auswählen, Dialogblatt Options, dort die Punkte App Language und App Region verändern.

Daten initialisieren und verändern

Soweit Sie die Vorschau nicht für die zentrale `ContentView` Ihrer App verwenden, müssen Sie oft vorweg Daten initialisieren und an Ihre eigene View übergeben. Im einfachsten Fall deklarieren Sie einfach eine entsprechende Variable und übergeben diese als Parameter.

```
#Preview {
    let myUser = User(name: "John Doe",
                      email: "john@example.com", ...)
    UserDetailView(user: myUser)
}
```

Falls die Daten in der View verändert werden sollen, müssen Sie der Variablen üblicherweise `@State` voranstellen. Speziell für `Preview`-Code kommt noch das Attribut `@Previewable` hinzu. Auf diesen und andere Sonderfälle gehe ich in Abschnitt 14.1, »Variablen synchronisieren (@State und @Binding)« näher ein.

```
#Preview {
    @PreviewState var myUser =
        User(name: "John Doe", email: "john@example.com", ...)
    EditUserView(user: myUser)
}
```

Einschränkungen und Tastenkürzel

In der Praxis funktioniert die Vorschau leider nicht immer so gut wie in den WWDC-Videos. Ein Problem besteht darin, dass die Vorschau häufig pausiert wird – etwa bei größeren Code-Änderungen, bei (auch nur vorübergehenden) Syntaxfehlern im Code oder bei der Ausführung Ihrer App in einem Simulator oder auf echter Hardware. Abhilfe schaffen in den meisten Fällen der Refresh-Button bzw. die Tastenkombination alt+cmd+P. Es wird vermutlich nicht lange dauern, bis Sie diese Tastenkombination verinnerlichen.

Tastenkürzel	Funktion
alt+cmd+P	Preview neu starten (Refresh)
alt+cmd+↵	Preview-Seitenleiste ein-/ausblenden

Tabelle 13.1 Preview-Tastenkürzel

Eine grundlegende Einschränkung der Preview-Funktion besteht darin, dass Ihre App syntaktisch absolut fehlerfrei sein muss. Eine einzige falsche Klammer, womöglich in einer Code-Datei, die im aktuellen Kontext gar nicht relevant ist, stoppt die Preview-Anzeige. Das ist gerade bei größeren Code-Umbauten problematisch, bei denen die

Preview ja ein wichtiges Werkzeug zur Fehlersuche ist. Einen echten Ausweg gibt es in diesem Fall nicht. Ich kann Ihnen nur empfehlen, Ihren Code generell in möglichst kleinen Schritten weiterzuentwickeln bzw. zur Not die zuletzt durchgeführten Änderungen mit cmd+Z rückgängig zu machen.

13.10 Apps auf dem eigenen iPhone ausführen

Für anfängliche Tests reicht es aus, den eigenen Code im Preview-Fenster oder im Simulator auszuführen. Früher oder später wollen Sie Ihre Apps auf eigenen Geräten (iPhone, iPad etc.) ausprobieren. Dazu sind vier Vorbereitungsschritte notwendig, für die Sie beim ersten Mal eine Viertelstunde Zeit einrechnen müssen. In Zukunft gelten die neuen Einstellungen dann per Default.

- **iOS-Entwicklermodus:** Zuerst aktivieren Sie auf dem betreffende Gerät den Entwicklermodus. Unter iOS finden Sie die Optionen in der App *Einstellungen* ganz unten im Dialogblatt Datenschutz & Sicherheit. Zur Aktivierung muss das Gerät neu gestartet werden.

  Sofern sich Ihr Mac mit Xcode und das iPhone oder iPad im gleichen Netzwerk befinden, sollte Xcode das Gerät selbstständig erkennen (siehe Abbildung 13.17). Beim erstmaligen Koppeln werden allerdings diverse Cache-Daten vom iPhone heruntergeladen, weswegen der Prozess einige Minuten dauert. Xcode zeigt im Dialog Windows • Devices and Simulators eine Fortschrittanzeige an.

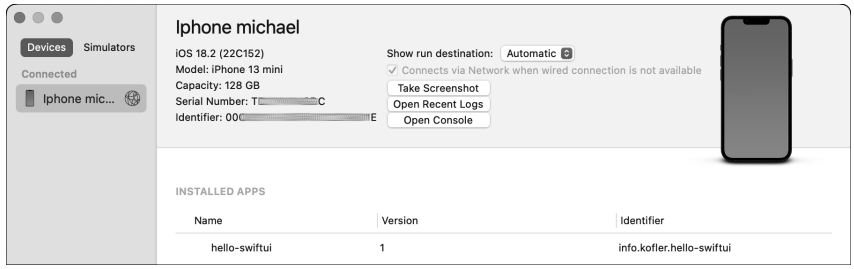


Abbildung 13.17 Xcode kann mit dem iPhone des Autors kommunizieren

- **Xcode-Account-Einstellungen:** Die Wartezeit können Sie nutzen und inzwischen im Dialogblatt Accounts der Xcode-Einstellungen entweder Ihren Developer Account oder Ihre Apple ID als Account hinzufügen (siehe Abbildung 13.18).

  Zur Ausführung von Apps auf Ihren Geräten ist also kein kostenpflichtiger Developer Account erforderlich, allerdings ist die Verwendung einer »gewöhnlichen« Apple ID mit Nachteilen verbunden. Insbesondere können Sie so signierte Apps nur sieben Tage lang ausführen/testen. Danach müssen Sie die App neu kompilieren und auf Ihr iPhone oder iPad übertragen.

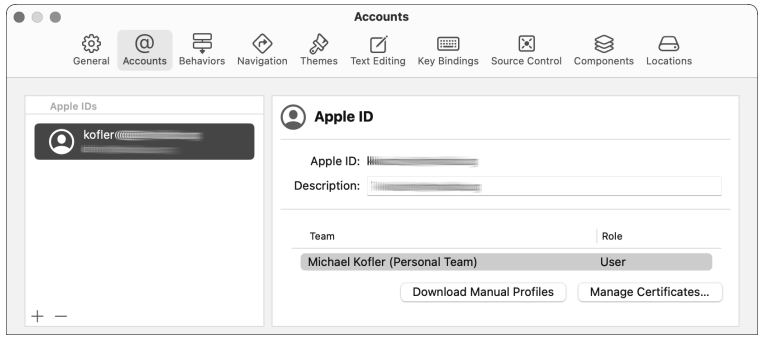


Abbildung 13.18 Die eigene Apple ID oder die des Developer Accounts muss in den Xcode-Einstellungen eingetragen werden.

- **Xcode-Signing:** In den Target-Einstellungen Ihrer App stellen Sie unter GENERAL • SIGNING Ihren Account als TEAM ein (siehe Abbildung 13.19).

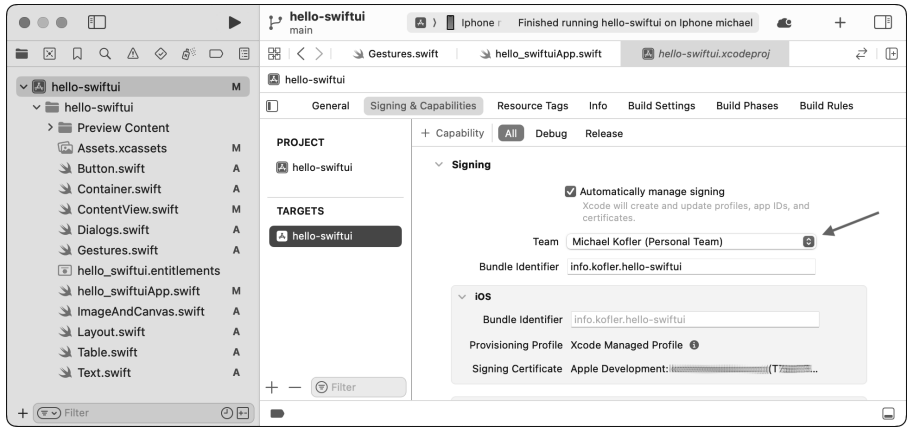


Abbildung 13.19 Die App muss mit dem Schlüssel Ihrer Apple ID oder Ihres Developer Accounts signiert werden.

- **Zertifikat vertrauen:** Wenn Sie in der Titelleiste von Xcode Ihr iPhone/iPad als Zielgerät auswählen und die App auszuführen versuchen, erscheint die Fehlermeldung, dass Ihr iPhone/iPad dem Entwicklungsteam nicht vertraut. Um auch diese Hürde zu umgehen, öffnen Sie auf Ihrem iPhone/iPad nochmals die Einstellungen und suchen nach ALLGEMEIN • VPN UND GERÄTEVERWALTUNG. Dort wird in der Rubrik ENTWICKLER-APP bereits Ihre Apple-ID angezeigt, allerdings als *nicht vertrauenswürdig*. Klicken Sie auf den Eintrag, und vertrauen Sie Ihrem eigenen Konto!

Kapitel 23
Währungskalkulator

Dieses Kapitel stellt einen Währungsrechner für das iPhone vor. Die App bezieht die Kurse von ca. 30 Währungen von der Website der Europäischen Zentralbank. Der App-Benutzer kann zwei dieser Währungen auswählen und dann unkompliziert Beträge zwischen diesen Währungen umrechnen.

Die Benutzeroberfläche der App steht in zwei Sprachen zur Verfügung: Deutsch und Englisch. Die App kann zudem kostenlos aus dem App Store heruntergeladen werden:

https://itunes.apple.com/us/app/wahrungs-rechner/id985004449

Hintergrundinformationen zur Lokalisierung sowie zu den Arbeiten, um das Programm App-Store-tauglich zu machen, finden Sie in Kapitel 24, »App Store & Co.«. In diesem Kapitel geht es nur um die eigentliche Funktionsweise des Programms.

Sowohl der Code der App als auch dessen Beschreibung in diesem Kapitel folgen dem MVVM-Pattern (siehe Abschnitt 14.5, »Model-View-ViewModel (MVVM)«). Die folgenden Abschnitte geben zuerst einen Überblick über die Funktionen der App und beschreiben dann das Datenmodell, das View Model (also die UI-Logik zur Datenverwaltung) und zuletzt die eigentliche Benutzeroberfläche. Es ist erschreckend, wie viel Code für die auf den ersten Blick so simple App notwendig ist!

23.1 App-Überblick

Die Benutzeroberfläche der App besteht aus zwei Tabs (siehe Abbildung 23.1): In der Hauptansicht können Sie in einem der beiden Textfelder eine Zahl eingeben. Der Betrag wird unmittelbar von der einen Währung in die andere umgerechnet. Das funktioniert unabhängig davon, welches der beiden Eingabefelder Sie verwenden. Die Tastatur wird ausgeblendet, sobald Sie die Oberfläche irgendwo außerhalb der Texteingabefelder berühren.

Im zweiten Tab mit den Einstellungen können Sie zwei Währungen aus einem Pool von circa 30 Währungen auswählen. Die Auswahl wird dauerhaft gespeichert. Mit dem Doppelpfeil-Button können Sie die erste und die zweite Währung vertauschen.

Die App lädt beim Start automatisch die aktuellen Kurse von der Europäischen Zentralbank und speichert diese in einem Cache-Verzeichnis. Falls das iPhone gerade keine Internetverbindung hat, kann es auf die zuletzt gespeicherten Kurse zurückgreifen. Der Einstellungs-Tab zeigt das Datum der Umrechnungskurse an und bietet die Möglichkeit, die Daten zu aktualisieren.

Abbildung 23.1 Die beiden Tabs des Währungsumrechners

Was heißt »aktuell«?

Die EZB legt die Wechselkurse aber nur einmal täglich fest, und das auch nur an Arbeitstagen. Die Kurse werden dann etwa um 16:00 Uhr auf der folgenden Seite veröffentlicht:

www.ecb.europa.eu/stats/exchange/eurofxref/html/index.en.html

Es handelt sich also nicht um Aktienkurse, die sich sekündlich ändern.

Flaggen

Die in der App benötigten Flaggen befinden sich in der Assets-Datei `Flags.xcassets`. Die Bitmaps können von der folgenden Website kostenlos heruntergeladen werden:

https://flagpedia.net

Zusammen mit der App werden fast 200 Flaggen ausgeliefert, obwohl die EZB aktuell nur 31 Währungskurse zur Verfügung stellt. Wenn Sie die App schlanker machen möchten, können Sie alle überflüssigen Flaggen löschen. Ich habe darauf verzichtet, auch weil die Möglichkeit besteht, dass die EZB in Zukunft weitere Umrechnungskurse zur Verfügung stellt. Zusätzlich zur Flaggensammlung von *https://flagpedia.net* habe ich die Bitmap `eu.png` mit der EU-Flagge eingefügt (Quelle: Wikipedia).

Vielleicht fragen Sie sich, wie Sie Ihrem Projekt eine zusätzliche Asset-Datei hinzufügen können. Das gelingt im Projekt-Explorer per Kontextmenü mit New File From Template • Resource • Asset Catalog.

Verbesserungsideen

Wenn Sie Gefallen an der App finden oder einfach zu Übungszwecken eigenen Code hinzufügen möchten, habe ich ein paar Verbesserungsideen:

- Die App unterstützt als einzige Zielplattform das iPhone. Sie können unter den Target-Einstellungen (General • Supported Destinations) iPads als zweite Plattform hinzufügen. Die App funktioniert prinzipiell auf Anhieb, das Layout sollte aber für den wesentlich größeren Bildschirm optimiert werden.
- Ein eigenes Tab-Blatt könnte eine numerische Auflistung aller Kurse anzeigen, also z. B. 1 EUR = 1,09 USD.
- Ein weiteres Tab-Blatt könnte eine Kurve mit der Kursentwicklung der letzten 90 Tage für das ausgewählte Währungspaar darstellen. Die erforderliche Datenbasis stellt die EZB unter folgender Adresse kostenlos zur Verfügung:

  https://www.ecb.europa.eu/stats/policy_and_exchange_rates/euro_reference_exchange_rates/html/index.en.html
- Der Währungskalkulator könnte zu einem vollständigen Taschenrechner ausgebaut werden.
- Anstelle der EZB-Kurse gibt es im Internet auch andere Quellen für die aktuellen Wechselkurse mit zum Teil noch mehr Währungen. Entsprechende Links finden Sie auf der folgenden Stack-Overflow-Seite:

  https://stackoverflow.com/questions/3139879

Beispieldateien

Sie finden diese App *zweimal* in den Beispielverzeichnissen: einmal im Verzeichnis für dieses Kapitel und ein zweites Mal im Verzeichnis für Kapitel 24, »Lokalisierung und App Store«. Die App-Store-Variante enthält im Vergleich zu diesem Kapitel eine deutsche Lokalisierung und ein »richtiges« App-Icon. Außerdem wurden ein paar weitere Kleinigkeiten für die App-Store-Einreichung geändert.

23.2 Umrechnungskurse herunterladen und speichern (Model)

Die Datengrundlage für die Währungs-App sind Wechselkurse, die die Europäische Zentralbank (EZB) auf der folgenden Seite täglich im XML-Format zur Verfügung stellt:

www.ecb.europa.eu/stats/eurofxref/eurofxref-daily.xml

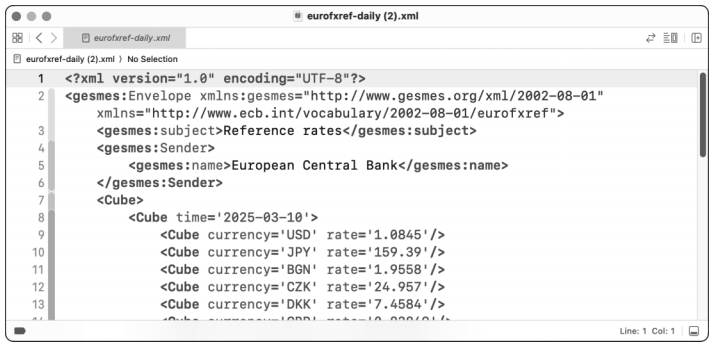


Abbildung 23.2 Wechselkurse relativ zum Euro, verpackt als XML-Dokument

Der Aufbau der Datei ist simpel (siehe Abbildung 23.2): Nach einigen Metadaten leitet ein `<Cube>`-Element die eigentlichen Kursdaten ein. Ein zweites `<Cube>`-Element enthält als Attribut das Datum der Kurse, weitere `<Cube>`-Elemente mit den Attributen `currency` und `rate` liefern die eigentlichen Wechselkurse. Die Währungskürzel werden dabei als Drei-Buchstaben-Code gemäß der ISO-Norm 4217 formuliert.

Für das Einlesen dieser Daten ist die asynchrone Methode `fetchEcbRates` der Klasse `CurrencyCalculator` verantwortlich. Die Kurse werden als Dictionary in der Eigenschaft `rates` gespeichert, das Datum der Kursfreigabe in der Eigenschaft `lastupdate`. Außerdem wird `currencies` mit einer alphabetischen Liste aller Währungen initialisiert. Die Klasse ist `@Observable`, damit die Eigenschaften unkompliziert an das View Model weitergereicht und im Einstellungs-Tab als Datenquellen für die Picker-Views verwendet werden können.

```
// Projekt currency-converter, Datei Model.swift
@Observable
class CurrencyConverter {
    enum FetchResult {
        case freshRates   // erfolgreicher Download der Kurse
        case cachedRates  // Fallback (früher gespeicherte Kurse)
    }

    private(set) var lastUpdate: Date = .distantPast
    private(set) var rates: [String: Double] = ["EUR": 1.0]
    private(set) var currencies: [String] = []
```

```
    init() {
        Task {
            await fetchEcbRates()
        }
    }

    // weitere Methoden, Code folgt
}
```

`fetchEcbRates` versucht die XML-Datei der EZB herunterzuladen und verwendet dann die SWXMLHash-Bibliothek zur Auswertung der Daten (siehe auch Abschnitt 19.6, »XML-Dateien lesen«). Die Methode liefert den Enumerationswert `freshRates` oder `cachedRates` zurück, je nachdem, ob der Download erfolgreich war oder nicht.

Das Datum wird mit einem `DateFormatter` in eine `Date`-Instanz umgewandelt. Sollten dabei Probleme auftreten, gilt `Date.distantPast` mit dem ältesten darstellbaren Datum als Defaultwert.

Die Umrechnungskurse werden in einem Dictionary gespeichert. Es enthält alle Kurse relativ zum Euro. Damit Umrechnungen von jeder Währung in jede andere Währung (inklusive Euro) unkompliziert funktionieren, benötigt das Dictionary auch einen Euro-Eintrag. Der Umrechnungskurs des Euro zu sich selbst lautet 1,0 und muss manuell hinzugefügt werden. Er ist aufgrund seiner Selbstverständlichkeit in der XML-Datei der EZB nicht enthalten.

```
// Projekt currency-converter, Datei Model.swift, Forts.
func fetchEcbRates() async -> FetchResult {
    let ecbUrl = URL(string: "https://www.ecb.europa.eu/stats/
                              eurofxref/eurofxref-daily.xml")!
    do {
        // Download
        let (data, _) =
            try await URLSession.shared.data(from: ecbUrl)
        let content = String(decoding: data, as: UTF8.self)
        let xml = XMLHash.parse(content)
        guard let ecbTime = xml["gesmes:Envelope"]["Cube"]
                  ["Cube"].element?.attribute(by: "time")?.text,
              ecbTime != "1900-01-01"
        else {
            loadRates()
            return .cachedRates
        }

        // Datum auswerten
        let formatter = DateFormatter()
        formatter.dateFormat = "yyyy-MM-dd H:mm"
```

```
        formatter.timeZone = TimeZone(abbreviation: "CET")
        lastUpdate = formatter.date(from: ecbTime + " 15:00")
                     ?? .distantPast

        // Umrechnungskurse in Dictionary sammeln
        var newRates = ["EUR": 1.0]
        for r in xml["gesmes:Envelope"]["Cube"]["Cube"]
                 ["Cube"].all
        {
            if let currency =
                   r.element?.attribute(by: "currency")?.text,
               let rateStr =
                   r.element?.attribute(by: "rate")?.text,
               let rate = Double(rateStr), rate > 0
            {
                newRates[currency] = rate
            }
        }

        // rates-Eigenschaft aktualisieren, wenn kein Fehler
        // aufgetreten ist
        if newRates.count > 3 {
            rates = newRates
            saveRates()
            currencies = rates.keys.sorted()
            return .freshRates
        else {
            loadRates()  // alte Kurse verwenden
            return .cachedRates
        }
    } catch {
        print("Error fetching exchange rates:
               \(error.localizedDescription)")
        loadRates()
        return .cachedRates
    }
}
```

Kurse lokal speichern und wieder laden

Für den Fall, dass die App offline verwendet wird (was gerade im Urlaub durchaus plausibel ist), werden die Kurse in `refreshRates` nach dem Herunterladen lokal gespeichert bzw. bei Download-Problemen geladen. Um den Schreibprozess so einfach wie möglich zu machen, werden das Datum und die Kurse in ein Dictionary verpackt. Dieses wird in ein `NSDictionary` umgewandelt, damit dessen `write`-Methode verwendet

werden kann. Der Leseprozess erfolgt umgekehrt. `save-` und `loadRates` verwenden jeweils `getRatesFilename`, um den Pfad zur Datei `rates.plist` im Cache-Verzeichnis zusammenzusetzen.

```
// Kurse in Cache-Datei speichern
private func saveRates() {
    guard rates.count > 1,
          let ratesPath = getRatesFilename()
    else { return }

    // Datum und Kurse in Dictionary verpacken und speichern
    let dict: [String: Any] = [ "lastupdate": lastUpdate,
                                "rates": rates ]
    (dict as NSDictionary).write(toFile: ratesPath,
                                 atomically: true)
}
// Kurse aus Cache-Datei laden
private func loadRates() {
    lastUpdate = .distantPast  // Default-Werte
    rates = ["EUR": 1.0]
    currencies = []
    guard let ratesPath = getRatesFilename(),
          let dict = NSDictionary(contentsOfFile: ratesPath),
          let update = dict["lastupdate"] as? Date,
          let newRates = dict["rates"] as? [String: Double]
    else { return }

    lastUpdate = update
    rates = newRates
    currencies = rates.keys.sorted()
}
// Dateiname aus Cache-Verzeichnis und rates.plist zusammensetzen
private func getRatesFilename() -> String? {
    FileManager.default
        .urls(for: .cachesDirectory, in: .userDomainMask)
        .first?
        .appendingPathComponent("rates.plist")
        .path
}
```

Länder-Code aus Währungskürzel

Die Flaggen in der Assets-Datei haben Namen wie `ch` (Schweiz), `fr` (Frankreich) oder `eu` (EU). Um aus dem Währungskürzel den passenden Länder-Code zu erzeugen, liest

`getCountryFromCurrency` die ersten beiden Buchstaben des Währungskürzels und wandelt sie in Kleinbuchstaben um (z. B. `CHF` zu `ch` oder `USD` zu `us`).

```
static func getCountryFromCurrency(_ currency: String)
            -> String
{
    guard currency.count > 2
    else { return currency.lowercased() }

    return String(currency.prefix(2)).lowercased()
}
```

Dieser Hack funktioniert für alle aktuell von der EZB angebotenen Währungen. Die Vorgehensweise ist aber nicht universell gültig. Sollten Sie eine App entwickeln, die alle weltweit gebräuchlichen Währungen unterstützt, benötigen Sie ein Dictionary mit Zuordnungen zwischen ISO 3166 (internationale Länder-Codes) und ISO 4217 (internationale Währungskürzel). Eine geeignete Tabelle finden Sie hier:

https://github.com/srcagency/country-currencies

Kursumrechnung

Der eigentliche Zweck der `CurrencyConverter`-Klasse besteht darin, Kurse umzurechnen. Die entsprechende Methode ist nur wenige Zeilen lang. Sie erwartet drei Parameter: den Betrag, das Währungskürzel der Ausgangswährung und das Kürzel der Zielwährung. `guard let` testet, ob beide Währungen im `rates`-Dictionary enthalten sind, und berechnet dann den neuen Betrag. Liegen hingegen ungültige Daten vor, gibt die Methode einfach `0` zurück.

```
// Beispiel convert(100, from: "EUR", to: "USD")
func convert(_ value: Double, from: String, to: String) -> Double
{
    guard let rateFrom = rates[from], let rateTo = rates[to]
    else { return 0.0 }

    return value / rateFrom * rateTo
}
```

23.3 UI-Logik (View Model)

Die Klasse `CurrencyViewModel` baut auf dem `CurrencyConverter` auf. Sie erfüllt die folgenden Funktionen:

- Model-Funktionen weiterreichen
- Laden/Speichern der eingestellten Währungen in den User-Defaults

- Verwaltung des Tastaturfokus (Welches der beiden Eingabefelder ist gerade aktiv?)
- Synchronisation der Textfelder (Währungsumrechnung während der Eingabe)
- Darstellung des Reload-Status

Organisation der Klasse, User-Defaults

Die folgenden Zeilen zeigen die prinzipielle Organisation des Codes. Die Klasse ist wiederum `@Observable`, damit die Benutzeroberfläche mit den öffentlichen Variablen der Klasse synchronisiert werden kann. `@Observable` ist aber leider (Stand Frühjahr 2025) inkompatibel zu `@AppStorage`. Deswegen reicht es nicht aus, `currency1` und `currency2` als öffentliche Variablen zu deklarieren und mit diesem Property-Wrapper auszustatten. Stattdessen kümmern sich die Init-Funktion sowie `setCurrency1/2` um das Laden bzw. Speichern der Werte. Gleichzeitig wird bei jeder Änderung einer Währung der Inhalt der beiden Textfelder zurückgesetzt. Auf die dabei eingesetzten Variablen `_value1/2` gehe ich gleich näher ein.

```
// Projekt currency-converter, Datei ViewModel.swift
@Observable
class CurrencyViewModel {
    // Zugriff auf den CurrencyConverter
    private let model: CurrencyConverter
    // lastUpdate und currencies einfach durchreichen
    var lastUpdate: Date { model.lastUpdate }
    var currencies: [String] { model.currencies }

    // Währungseinstellungen in/aus User-Defaults speichern/laden
    private let ud = UserDefaults.standard
    private let currency1Key = "currency1"
    private let currency2Key = "currency2"
    private(set) var currency1: String
    private(set) var currency2: String

    // weitere Eigenschaften, Code folgt

    init() {
        model = CurrencyConverter()
        self.currency1 = ud.string(forKey: currency1Key) ?? "EUR"
        self.currency2 = ud.string(forKey: currency2Key) ?? "USD"
    }

    // Währung 1/2 ändern und speichern, Input zurücksetzen
    func setCurrency1(_ newValue: String) {
        currency1 = newValue
        _value1 = ""
```

```
        _value2 = ""
        ud.set(newValue, forKey: currency1Key)
    }
    func setCurrency2(_ newValue: String) {
        currency2 = newValue
        _value1 = ""
        _value2 = ""
        ud.set(newValue, forKey: currency2Key)
    }

    // Methode vom Model an das View Model weiterreichen
    static func getCountry(from currency: String) -> String {
        CurrencyConverter.getCountryFromCurrency(currency)
    }
}
```

Tastaturfokus

Die App muss sich aus zweierlei Gründen um den Tastaturfokus kümmern:

- Zum einen ist für die Währungsumrechnung wichtig zu wissen, welches der beiden Textfelder gerade Eingaben empfängt. Damit können bei der Synchronisation des zweiten Felds zyklische Updates vermieden werden: Eine Eingabe in Feld 1 verursacht ein Update in Feld 2; dieses darf aber nicht zu einem weiteren Update in Feld 1 führen!
- Zum anderen muss es eine Möglichkeit geben, die bei Zahleneingaben automatisch eingeblendete Tastatur wieder auszublenden.

Der Großteil des Codes für das Tastaturmanagement befindet sich in `ContentView.swift`. Im View Model sind dafür die private Variable `activeField` und die öffentliche Funktion `setActiveField` vorgesehen. Die zulässigen Werte sind in der Enumeration `ActiveField` definiert.

```
// Projekt currency-converter, Datei ViewModel.swift (Forts.)
@Observable class CurrencyViewModel {
    ...
    // Verwaltung des Tastaturfokus
    enum ActiveField {
        case first, second
    }
    private var activeField: ActiveField? = nil

    func setActiveField(_ field: ActiveField?) {
        self.activeField = field
    }
}
```

Synchronisation der Textfelder

Die Inhalte der beiden Textfelder des Währungsumrechners sind mit `value1/2` verbunden. Dabei handelt es sich um Computed Properties, die eigentliche Speicherung erfolgt in den privaten Variablen `_value1/2`. Der `set`-Code der Computed Properties synchronisiert bei jeder Änderung das jeweils andere Textfeld, aber nur, wenn das dem `set`-Code zugeordnete Textfeld gerade aktiv ist, also Benutzereingaben empfängt.

Die eigentliche Umrechnung erfolgt in `syncValues`. Diese Methode testet, ob sich die Eingabe überhaupt in eine `Double`-Zahl umwandeln lässt, ruft dann `model.convert` auf, formatiert das Ergebnis mit einer Nachkommastelle und speichert es in der Variablen für das Textfeld, das gerade *nicht* den Tastaturfokus hat.

```
// Projekt currency-converter, Datei ViewModel.swift (Forts.)
@Observable class CurrencyViewModel {
    // Inhalt der beiden Textfelder
    private var _value1 = ""
    private var _value2 = ""
    var value1: String {
        get { _value1 }
        set {
            _value1 = newValue
            if activeField == .first {
                syncValues(from: .first)
            }
        }
    }
    var value2: String {
        get { _value2 }
        set {
            _value2 = newValue
            if activeField == .second {
                syncValues(from: .second)
            }
        }
    }
    // Inhalt des jeweils anderen Textfelds synchronisieren
    private func syncValues(from field: ActiveField) {
        switch field {
        case .first:
            guard let amount = Double(_value1)
            else {
                _value2 = ""
                return
            }
```

```
            let converted = model.convert(amount,
                                          from: currency1,
                                          to: currency2)
            _value2 = String(format: "%.1f", converted)

        case .second:
            guard let amount = Double(_value2)
            else {
                _value1 = ""
                return
            }
            let converted = model.convert(amount,
                                          from: currency2,
                                          to: currency1)
            _value1 = String(format: "%.1f", converted)
        }
    }
}
```

Umrechnungskurse neu laden

Im Einstellungs-Tab besteht die Möglichkeit, die Umrechnungskurse neu zu laden. In der Folge soll dort angezeigt werden, ob der Reload erfolgreich war und von welchem Datum die Kurse sind. Weil der Download asynchron erfolgt, ist der Umgang mit den diversen Sonderfällen nicht trivial.

Die Enumeration `RefreshStatus` berücksichtigt vier Zustände. `idle` ist der Normalzustand. `refreshing` bedeutet, dass die Kurse gerade geladen werden, aber noch kein Ergebnis vorliegt. `success` weist auf einen erfolgreich durchgeführten Download hin. Mit `error` kann die Fehlermeldung als *Associated Value* mitgeliefert werden (siehe Abschnitt 9.2, »Enumerationen«).

Leider führt die Verwendung von *Associated Values* dazu, dass der Swift-Compiler nicht selbstständig in der Lage ist, das Protokoll `Equatable` zu implementieren. Um simple Vergleiche wie `refreshStatus == .success` zu ermöglichen, müssen Sie `Equatable` selbst implementieren. Das ist zum Glück nicht schwierig.

```
// Projekt currency-converter, Datei ViewModel.swift (Forts.)
@Observable class CurrencyViewModel {
    ...
    enum RefreshStatus: Equatable {
        case idle
        case refreshing
        case success
        case error(String)
```

```
        // Code für Equatable
        static func == (lhs: RefreshStatus, rhs: RefreshStatus)
                    -> Bool
        {
            switch (lhs, rhs) {
            case (.idle, .idle), (.refreshing, .refreshing),
                 (.success, .success):
                return true
            case (.error(let lhsMsg), .error(let rhsMsg)):
                return lhsMsg == rhsMsg
            default:
                return false
            }
        }
    }
    private(set) var refreshStatus: RefreshStatus = .idle
```

Für das Neuladen der Kurse ist die asynchrone Methode `refreshRates` zuständig. Sie ruft `fetchEcbRates` aus der `CurrencyConverter`-Klasse auf. Wenn der Download erfolgreich ist, nimmt `refreshStatus` für drei Sekunden den Zustand `success` ein und wird dann wieder zurückgestellt. Sollte ein Download dagegen scheitern, wird ind `refreshStatus` mittels `error(msg)` eine Fehlernachricht gespeichert.

```
// Projekt currency-converter, Datei ViewModel.swift (Forts.)
    ...
    // Kurse von der EZB neu laden
    func refreshRates() async {
        refreshStatus = .refreshing

        let result = await model.fetchEcbRates()

        switch result {
        case .freshRates:
            refreshStatus = .success
            Task {  // nach drei Sekunden auf .idle zurückstellen
                try? await Task.sleep(for: .seconds(3))
                if refreshStatus == .success {
                    refreshStatus = .idle
                }
            }
        case .cachedRates:
            if model.lastUpdate == .distantPast {
                refreshStatus = .error("Failed to load rates.
                                        Check your connection.")
            } else {
```

```
                let df = DateFormatter()
                df.dateStyle = .medium
                df.timeStyle = .none
                let date = df.string(from: model.lastUpdate)
                refreshStatus = .error("Using cached rates
                                        from \(date)")
            }
        }
    }
}
```

23.4 Benutzeroberfläche (View)

Die Benutzeroberfläche der App besteht aus einer `TabView`. Die beiden Tabs werden jeweils durch eigene Views dargestellt. Damit bleibt die `ContentView` sehr übersichtlich. Es gibt nur zwei gemeinsame Variablen, `vm` für das View Model sowie `selectedTab` für den gerade aktiven Tab.

```
// Projekt currency-converter, Datei ContentView.swift
struct ContentView: View {
    let vm = CurrencyViewModel()
    @State private var selectedTab = 0

    var body: some View {
        TabView(selection: $selectedTab) {
            CurrencyView(vm: vm, tabSelection: $selectedTab)
                .tabItem {
                    Image(systemName: "dollarsign.circle.fill")
                    Text("Converter")
                }
                .tag(0)

            SettingsView(viewModel: vm)
                .tabItem {
                    Image(systemName: "gear")
                    Text("Settings")
                }
                .tag(1)
        }
    }
}
```

Eingabefeld (CurrencyInputView)

Das Feld zur Eingabe bzw. Darstellung der Währungsbeträge ist in eine eigene View verpackt. Es besteht im Wesentlichen aus einer `TextView`, die unten in einem `ZStack` angeordnet ist. Am linken Rand werden darüber die Flagge und die Bezeichnung der aktuellen Währung angezeigt.

An die Init-Funktion müssen fünf Parameter für die Eigenschaften der View übergeben werden.

- `currencyCode` enthält das Währungskürzel.
- `value` ist mit dem Inhalt des `TextField`s verbunden.
- `isFocused` gibt an, ob das Textfeld den Eingabefokus hat.
- Die Funktion `onFocusChanged` wird bei einem Eingabefokuswechsel aufgerufen. Den dabei ausgeführten Code behandle ich im nächsten Abschnitt.
- Auch `onFlagTapped` ist eine Funktion. Sie ist dafür verantwortlich, beim Berühren der Flagge oder des Währungscodes den Einstellungs-Tab zu aktivieren.

`countryCode` ist ein Computed Property, das aus dem `currencyCode` den passenden Ländercode ermittelt. Dieser wird zur Anzeige der richtigen Flagge benötigt.

Weil nur Zahlen eingegeben werden sollen, verwendet das Textfeld eine `decimalPad`-Tastatur. Von den verbleibenden `TextField`-Modifiern ist nur `onChange` von Interesse: Wenn sich der Eingabefokus ändert, wird die Funktion `onFocusChanged` aufgerufen. Falls das Textfeld gerade den Fokus erhalten hat, wird außerdem der bisherige Inhalt gelöscht, sodass sofort mit der Eingabe eines neuen Werts begonnen werden kann.

Um Platz zu sparen, habe ich diverse `padding`-Modifier aus dem Listing entfernt.

```
// Projekt currency-converter, Datei ContentView.swift (Forts.)
struct CurrencyInputView: View {
    let currencyCode: String
    @Binding var value: String
    @FocusState var isFocused: Bool
    let onFocusChanged: (Bool) -> Void
    let onFlagTapped: () -> Void

    var countryCode: String {
        CurrencyViewModel.getCountry(from: currencyCode)
    }

    var body: some View {
        ZStack(alignment: .topLeading) {
            // Text field background
            TextField("0", text: $value)
                .font(.system(size: 34, weight: .bold))
```

```
                .keyboardType(.decimalPad)
                .multilineTextAlignment(.trailing)
                .frame(height: 100)
                .frame(maxWidth: .infinity)
                .background(Color(.systemGray6))
                .cornerRadius(12)
                .focused($isFocused)
                .onChange(of: isFocused) { _, newValue in
                    onFocusChanged(newValue)
                    if newValue == true {
                        value = ""
                    }
                }

            // Flagge und Währungskürzel überlagern links das
            // Textfeld
            VStack(spacing: 4) {
                Image(countryCode)
                    .resizable()
                    .aspectRatio(contentMode: .fit)
                    .frame(width: 60, height: 40)

                Text(currencyCode)
                    .font(.subheadline)
                    .fontWeight(.semibold)
            }
            .onTapGesture {
                onFlagTapped()
            }
        }
    }
}
```

Hauptansicht (CurrencyView)

Die `CurrencyView` ist die zentrale App-Ansicht. Sie besteht aus zwei in einem `VStack` platzierten `CurrencyInputViews`. An die Init-Funktion müssen eine Referenz auf das View Model sowie eine `@Binding`-Variable für den aktiven Tab übergeben werden. Letztere ist notwendig, damit der aktive Tab per Code verändert werden kann. Die beiden `@FocusState`-Variablen helfen dabei beim Management des Eingabefokus (siehe Abschnitt 14.3).

Der `body`-Code beginnt mit einem `VStack` mit den beiden `CurrencyInputViews`. Bemerkenswert sind dabei die fünf an die Init-Funktion übergebenen Parameter, die den

Datenfluss bestimmen. (Werfen Sie noch einmal einen Blick in den vorigen Abschnitt, wo ich die fünf Parameter aus der Sicht der `CurrencyInputView` beschrieben habe!)

- Die View-Model-Eigenschaft `currency` legt das Währungskürzel und die Flagge fest.
- Die View-Model-Eigenschaft `value` wird mit dem Inhalt des Textfelds verbunden (`@Observable` im Datenmodell bzw. `@Binding` in `CurrencyInputView`).
- `isFocused` gibt an, ob das Textfeld gerade den Eingabefokus hat oder nicht. Etwas befremdlich ist hier der vorangestellte Unterstrich. Er ist notwendig, weil an dieser Stelle nicht einfach ein boolscher Wert übergeben werden soll, sondern der Property Wrapper selbst (also `@FocusState`). Auch in `CurrencyInputView` ist die Eigenschaft mit `@FocusState` deklariert.

  Der Property Wrapper verbindet `varname` mit einer internen Eigenschaft `_varname`. Diese interne Eigenschaft muss übergeben werden. Xcode weist dankenswerterweise sogar auf diesen Umstand hin, sollten Sie den Unterstrich vergessen.
- Mit `onFocusChange` übergeben Sie eine Funktion (hier in Form einer Closure), die bei jedem Wechsel des Fokus innerhalb von `CurrencyInputView` ausgeführt wird. An die Closure wird ein Parameter übergeben, den ich im Code `isFocused` bezeichne. Der Parameter enthält die Information, ob das Textfeld gerade den Fokus hat oder nicht.

  Diese Information wird mit `setActiveField` an das View Model weitergeleitet. Verliert das Textfeld den Fokus und hat auch das zweite Textfeld keinen Fokus, wird mit `setActiveField` die Variable `activeField` im `CurrencyViewModel` auf `nil` gesetzt. (Der Datentyp ist optional, somit ist dies eine zulässige Operation.)
- Auch mit `onFlagTapped` wird eine Funktion übergeben. Sie wird ausgeführt, wenn der Benutzer bzw. die Benutzerin der App die Flagge oder den Währungstext berührt. Dann kommt es dank `tabSelection=1` zur Aktivierung des Einstellungs-Tabs.

Auf den `VStack` wird zuerst ein `contentShape`-Modifier angewendet, damit in der Folge mit `onTapGesture` das Berühren des Hintergrunds festgestellt werden kann. In der Gesture-Closure werden beide Fokusvariablen auf `false` gesetzt. Das führt dazu, dass die Tastatur ausgeblendet wird. Auf den Abdruck diverser `padding`-Modifier habe ich auch bei diesem Listing verzichtet.

```
// Projekt currency-converter, Datei ContentView.swift (Forts.)
struct CurrencyView: View {
    @State var vm: CurrencyViewModel
    @Binding var tabSelection: Int

    @FocusState private var isField1Focused: Bool
    @FocusState private var isField2Focused: Bool
```

```
    var body: some View {
        VStack(spacing: 24) {
            // erstes Eingabefeld
            CurrencyInputView(
                currencyCode: vm.currency1,
                value: $vm.value1,
                isFocused: _isField1Focused,
                onFocusChanged: { isFocused in
                    if isFocused {
                        vm.setActiveField(.first)
                    } else if !isField2Focused {
                        vm.setActiveField(nil)
                    }
                },
                onFlagTapped: { tabSelection = 1 }
            )

            // zweites Eingabefeld, analoger Code
            CurrencyInputView(...)

            Spacer()  // Eingabefelder ganz oben
                      // unten ist Platz für die Tastatur
        }
        // Tastatur ausblenden, wenn das Rechteck im Hintergrund
        // des VStacks berührt wird
        .contentShape(Rectangle())
        .onTapGesture {
            isField1Focused = false
            isField2Focused = false
        }
    }
}
```

23.5 Währungseinstellung (View)

Der zweite Tab der `TabView` ist für die Währungseinstellungen sowie für den Reload der Währungsdaten zuständig. Die folgenden Zeilen fassen die prinzipielle Struktur zusammen:

```
// Projekt currency-converter, Datei ContentView.swift
// Einstellungen, Code-Struktur (Details folgen)
struct SettingsView: View {
    @Bindable var vm: CurrencyViewModel
```

```
    // im Picker ausgewählte Währungen
    @State private var curr1Select: String
    @State private var curr2Select: String

    var body: some View {
        VStack {
            HStack {
                CurrencyPickerView(...)  // erste Währung
                CurrencyPickerView(...)  // zweite Währung
            }
            Button(...)           // Währungen vertauschen
            Spacer()
            RatesRefreshView(...) // Reload-Button, Statusanzeige
        }
    }
}
```

Ich beschreibe die zwiebelähnliche Struktur von innen nach außen und beginne daher mit der `CurrencyPickerView` und der `RatesRefreshView`.

Picker im Wheel-Modus zur Auswahl der Währungen

Die `CurrencyPickerView` zeigt die Währungsflaggen und -Bezeichnungen in einem radförmigen `Picker` an (siehe Abbildung 23.3).

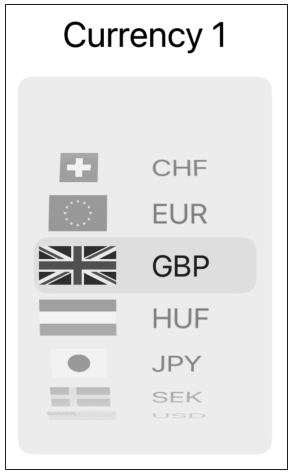


Abbildung 23.3 Die »CurrencyPickerView«

An die View werden drei Parameter übergeben, die Beschriftung, ein sortiertes String-Array mit den Währungen und eine `@Binding`-Variable für die gerade ausgewählte Währung. Die Listenelemente werden mit `ForEach` initialisiert. Die Flaggen haben

unterschiedliche Proportionen. Innerhalb des Pickers werden alle Flaggen in der gleichen Höhe angezeigt. Die Breite variiert deswegen. Um die Flaggen zentriert darzustellen, werden sie in einem `ZStack` mit vorgegebener Breite platziert.

```
// Projekt currency-converter, Datei ContentView.swift (Forts.)
struct CurrencyPickerView: View {
    let title: String
    let currencies: [String]
    @Binding var selectedCurrency: String

    var body: some View {
        VStack(alignment: .left, spacing: 8) {
            Text(title).font(.title3).padding(.bottom, 4)

            Picker("", selection: $selectedCurrency) {
                ForEach(currencies, id: \.self) { currency in
                    HStack {
                        // Flaggen im ZStack zentrieren
                        ZStack {
                            Image(CurrencyViewModel.getCountry(
                                                    from: currency))
                                .resizable()
                                .aspectRatio(contentMode: .fit)
                                .frame(height: 22)
                        }
                        .frame(width: 50, alignment: .center)

                        Spacer()

                        // Bezeichnung der Währung
                        Text(currency)
                           .font(.body)
                           .frame(width: 60, alignment: .leading)
                    }
                    .tag(currency)
                }
            }
            .pickerStyle(.wheel)  // radförmiges Erscheinungsbild
            .frame(width: 145)
            .background(Color(.systemGray6))
            .cornerRadius(8)
        }
    }
}
```

Kein endloses Drehen

Dem `Picker` in der SwiftUI-Bibliothek fehlt (Stand Frühjahr 2025) die Darstellungsoption eines sich endlos drehenden Rads, wie Sie dies von den Dialogen zur Zeiteinstellung gewöhnt sind. Sie können diese Einschränkung mit einem Trick umgehen: Sie initialisieren die Liste mit zehn Zyklen aller Elemente und beginnen in der Mitte. Damit kann die Benutzerin bei der Auswahl einer Währung in jede Richtung fünf komplette Durchläufe machen. Es sollte klar sein, dass diese auf StackOverflow skizzierte »Lösung« nur ein Hack ist.

https://stackoverflow.com/questions/63332500

Reload-Button und Statusanzeige

Im unteren Bereich der Settings-Ansicht zeigt die App das Datum der Kurse an und bietet die Möglichkeit, aktuelle Kurse zu laden (siehe Abbildung 23.4). Bei einem Neustart der App werden immer die aktuellen Kurse geladen, sofern eine Internet-Verbindung besteht. Aber bei einer längeren Verwendung der App kann ein manueller Refresh zweckmäßig sein.

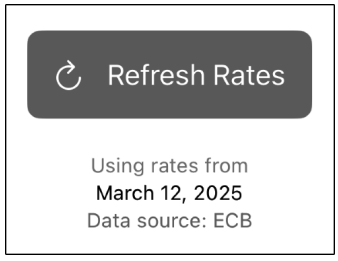


Abbildung 23.4 Die »RatesRefreshView«

Den relativ umfangreichen Code habe ich in die `RatesRefreshView` verpackt. Der Button Refresh Rates löst einen asynchronen Ladeprozess auf. Der Status des Ladeprozesses wird über die Eigenschaft `refreshStatus` des View Models kommuniziert. Solange der Ladeprozess läuft, wird der Button deaktiviert (ausgegraut); außerdem wird innerhalb des Buttons mit dem `ProgressView` ein rotierender Spinner angezeigt. (Die Datenmengen sind winzig, d. h., bei einer guten Internetverbindung dauert der Prozess deutlich weniger als eine Sekunde.)

Die nachfolgenden Texte werden in Abhängigkeit vom Refresh-Status zusammengestellt. Sie zeigen das Datum des Downloads bzw. die Meldung des zuletzt aufgetretenen Fehlers. Zuletzt folgt ein Link auf die Webseite der EZB, die die Herkunft der Kurse erläutert.

Wie schon in den vorangegangenen Abschnitten habe ich aus dem Listing diverse Layout-Modifier entfernt (`padding`, `font`, zum Teil auch `xxxColor`). Damit bleibt der Code übersichtlich, und Sie können sich beim Durchsehen auf die Logik konzentrieren.

```
// Projekt currency-converter, Datei ContentView.swift (Forts.)
struct RatesRefreshView: View {
    @Bindable var vm: CurrencyViewModel

    var body: some View {
        VStack {
            // Button 'Refresh Rates'
            Button(action: {
                Task { await vm.refreshRates() }
            }) {
                HStack {
                    if case .refreshing = vm.refreshStatus {
                        ProgressView()
                    } else {
                        Image(systemName: "arrow.clockwise")
                    }
                    Text("Refresh Rates")
                }
                .background(vm.refreshStatus == .refreshing ?
                            Color.gray : Color.blue)
                .cornerRadius(8)
            }
            .disabled(vm.refreshStatus == .refreshing)

            // Status anzeigen
            switch vm.refreshStatus {
            case .idle:
                Text("Using rates from")
                Text(vm.lastUpdate, style: .date)
            case .refreshing:
                Text("Refreshing rates...")
            case .success:
                Text("Rates updated successfully!")
                Text(vm.lastUpdate, style: .date)
            case .error(let message):
                Text(message).foregroundColor(.red)
            }

            // Link zur Datenquelle
            let ecb = "https://www.ecb.europa.eu/..."
```

```
            Link(destination: URL(string: ecb)!) {
                Text("Data source: ECB").font(.caption)
            }
        }
    }
}
```

Der Einstellungs-Tab

Jetzt sind nur noch einige Erläuterungen zur `SettingsView` notwendig, die die beiden vorhin beschriebenen Subviews einbettet. Die drei View-Eigenschaften habe ich in der Einleitung dieses Abschnitts schon beschrieben; ihre Initialisierung ist selbsterklärend.

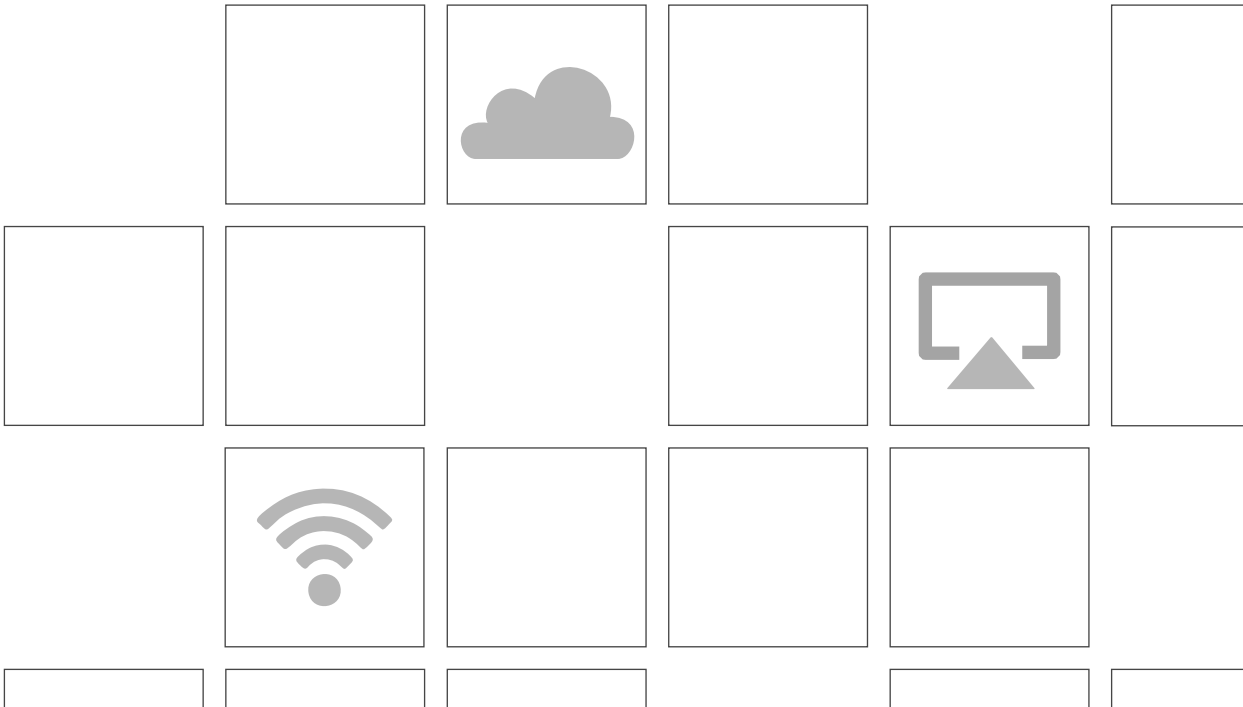
Die beiden `CurrencyPickerView`s sind in einen `HStack` verpackt. Der `onChange`-Modifier ruft `setCurrency1/2` auf. Diese Methode aus dem View Model verändert nicht nur die entsprechende `currency`-Eigenschaft, sondern setzt auch `value1/2` zurück. Das ist wichtig, weil sonst eine zuvor durchgeführte Umrechnung zu nicht mehr dazu passenden Währungen stehen bliebe.

Der Swap-Button vertauscht `curr1Select` und `curr2Select` und führt diese Veränderung auch im View Model durch. Am unteren Bildschirmrand werden schließlich der Button Refresh Rates und die Statusinformationen angezeigt.

```
// Projekt currency-converter, Datei ContentView.swift (Forts.)
struct SettingsView: View {
    @Bindable var vm: CurrencyViewModel
    @State private var curr1Select: String
    @State private var curr2Select: String

    init(viewModel: CurrencyViewModel) {
        vm = viewModel
        curr1Select = viewModel.currency1 // aktuelle Währungen
        curr2Select = viewModel.currency2
    }

    var body: some View {
        VStack(spacing: 20) {
            HStack(alignment: .top, spacing: 12) {
                // Picker für Währung 1
                CurrencyPickerView(
                    title: "Currency 1",
                    currencies: vm.currencies,
                    selectedCurrency: $curr1Select
                )
```

```
                .onChange(of: curr1Select) { _, newValue in
                    vm.setCurrency1(newValue)
                }
                // Picker für Währung 2
                CurrencyPickerView(...)  // analoger Code
            }
            // Währungen vertauschen (auch im View Model!)
            Button(action: {
                let temp = curr1Select
                curr1Select = curr2Select
                curr2Select = temp
                vm.setCurrency1(curr1Select)
                vm.setCurrency2(curr2Select)
            }) {
                Image(systemName: "arrow.left.arrow.right")
            }
            .buttonStyle(PlainButtonStyle())

            Spacer()  // Abstand

            RatesRefreshView(vm: vm)  // Reload und Status
        }
    }
}
```

Auf einen Blick



Inhalt

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

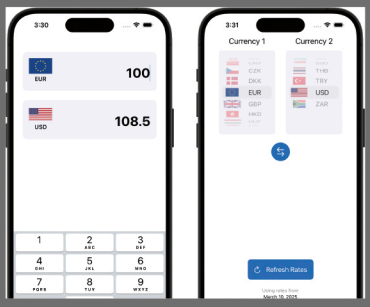
Y

Swift

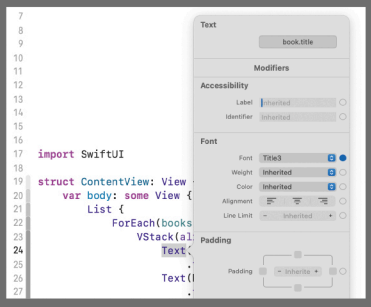
»Das Swift-Standardwerk«

Alles, was Sie über Swift wissen müssen

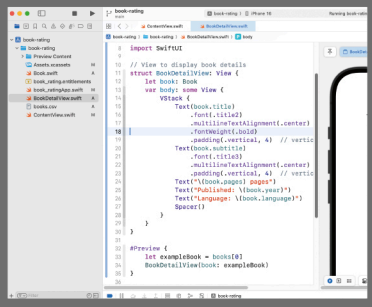
Mit Swift und SwiftUI erstellen Sie Apps für macOS und iOS. Wie das geht, zeigt Ihnen Michael Kofler in diesem neu konzipierten Handbuch. Von den ersten Zeilen in Xcode über das Design mit SwiftUI bis zur Veröffentlichung im App Store lernen Sie alle Schritte der App-Entwicklung kennen und machen sich mit den Features von Swift 6 vertraut.



Apps entwickeln



Grundlagen verstehen



Professionell arbeiten

Solides Fundament

Lernen Sie die Programmiertechniken für den Einsatz der Swift-Konzepte in der Praxis kennen: JSON-Dateien verarbeiten, Dateien aus dem Internet herunterladen, REST-APIs anwenden, Code asynchron ausführen und vieles mehr.

GUI-Design mit SwiftUI

Das deklarative SwiftUI macht die Gestaltung moderner Apps zwar nicht zum Kinderspiel, aber einfacher als je zuvor! Erfahren Sie, wie Sie Ihre Apps aus Views zusammensetzen, sie ansprechend gestalten und wie Sie für ein übersichtliches und einfaches Design sorgen.

Modernes Coding

Praxisnahe Beispiele veranschaulichen den Datenfluss von der Oberfläche über eigene Klassen bis zur persistenten Speicherung mit SwiftData oder in der iCloud. Immer mit dabei: professionelle Arbeitstechniken rund um Xcode, KI-Helfer und mehr.

Vollständige Beispiel-Apps zum Download.

Dr. Michael Kofler arbeitet seit Jahrzehnten in der Softwareentwicklung und nutzt Swift, seitdem es vor gut 10 Jahren vorgestellt wurde. In diesem Standardwerk zeigt er Ihnen, wie Sie sicher und effektiv mit Apples Programmiersprache arbeiten.

Aus dem Inhalt

Rheinwerk Computing

Gedruckt in Deutschland
Mineralölfreie Druckfarben
Zertifiziertes Papier

€ 49,90 [D] € 51,30 [A]

Programmierung
ISBN 978-3-367-10734-6