Einstieg in SwiftUI

thomas SILLMANN

User Interfaces erstellen für **macOS, iOS, watchOS** und **tvOS**

E-Book

EXTRA: E-Book inside

HANSER

Sillmann

Einstieg in SwiftUI

Thomas Sillmann

Einstieg in SwiftUI

User Interfaces erstellen für macOS, iOS, watchOS und tvOS

HANSER

Der Autor:
Thomas Sillmann, Aschaffenburg
www.thomassillmann.de

Bibliografische Information der Deutschen Nationalbibliothek:

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Lektorat: Sylvia Hasselbach

Print-ISBN: 978-3-446-46362-2

1 Einstieg in SwiftUI

SwiftUI ist Apples neuestes Framework zur Erstellung von Nutzeroberflächen. Wahrlich spannend ist hierbei unter anderem, dass SwiftUI plattformübergreifend ist. Mit nur einem Framework ist es jetzt möglich, einheitliche User Interfaces sowohl für den Mac als auch für iPhone, iPad, Apple TV und Apple Watch zu erstellen. SwiftUI ist in diesem Kontext eine vollwertige Alternative zu den bestehenden UI-Frameworks wie UIKit, AppKit und WatchKit.

Bild 1.1 SwiftUI bietet gänzlich neue Möglichkeiten zum Erstellen von Nutzeroberflächen für Mac, iPhone und Co.

1.1 Voraussetzungen

SwiftUI ist ab Xcode 11 Teil von Apples hauseigener Entwicklungsumgebung. Diese Version wird also mindestens benötigt, um SwiftUI produktiv einsetzen zu können.

Daneben kann SwiftUI aber auch nur in Apps eingesetzt werden, die mindestens die in Tabelle 1.1 aufgeführten Betriebssysteme von Apple unterstützen. Für ältere Versionen von Apples Plattformen steht SwiftUI nicht zur Verfügung.

Tabelle 1.1 Mindestvoraussetzungen zum Einsatz von SwiftUI

Plattform	Mindestversion für SwiftUI
macOS	Catalina 10.15
iOS/iPadOS	13
watchOS	6
tvOS	13

Darüber hinaus kommt bei der Arbeit mit SwiftUI noch eine weitere Besonderheit zum Tragen: Apple hat in Xcode 11 einige Funktionen integriert, die eine maßgebliche Rolle beim Einsatz von SwiftUI spielen. Dazu gehört allen voran eine interaktive Vorschau, die live während der Entwicklung das Aussehen von Ansichten widergibt und sich wie ein Simulator bedienen lässt. Zur Nutzung des Großteils dieser Funktionen reicht aber Xcode 11 alleine nicht aus, dafür ist zusätzlich auch der Einsatz von wenigstens macOS Catalina 10.15 notwendig.

Zwar kann man generell Xcode 11 auch unter macOS Mojave 10.14 nutzen und darin Ansichten in SwiftUI programmieren, dann geht jedoch ein großer Teil der spannenden Komfortfunktionen des neuen UI-Frameworks verloren. Um also das Optimum aus der Arbeit mit SwiftUI herauszuholen, sollten Sie unbedingt einem Mac mit mindestens macOS Catalina 10.15 einsetzen.

1.2 Grundlagen

Das SwiftUI-Framework dient ausschließlich zum Erstellen und Gestalten von Views, die in Apps für die verschiedenen Plattformen von Apple zum Einsatz kommen. Es stellt somit eine Alternative zu `UIView` aus dem UIKit-Framework und zu `NSView` aus dem AppKit-Framework dar.

Basis aller Views, die mittels SwiftUI erzeugt werden, ist das `View`-Protokoll (siehe Bild 1.2). Es definiert die verschiedenen Eigenschaften und Funktionen einer jeden

Ansicht und verfügt in diesem Zuge über eine Property, die Sie immer zwingend implementieren müssen: `body`.

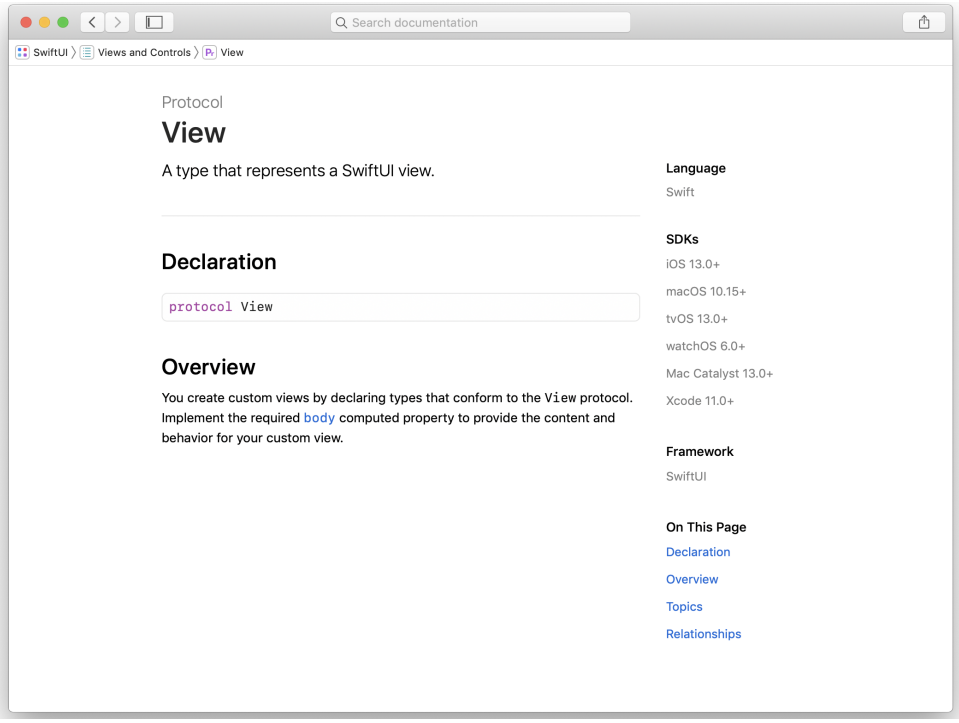

Bild 1.2 Das View-Protokoll ist das Herzstück jeder Ansicht in SwiftUI.

Die `body`-Property ist das Herzstück einer jeden View in SwiftUI. Sie liefert die Ansicht zurück, die eine View darstellen soll. Innerhalb der body-Property bringt man somit alle Elemente unter, die Teil der zugrundeliegenden Ansicht sind, ganz gleich ob es sich dabei um Listen, Text, Bilder oder sonstige Elemente aus dem SwiftUI-Framework handelt.

Als Rückgabewert für die `body`-Property kommt typischerweise `some View` zum Einsatz. Das bedeutet, dass `body` eine View-Instanz zurückliefert, ohne dass der konkrete Typ für diese View bekannt ist. Das sorgt für höchstmögliche Flexibilität, kann eine View sich so doch aus beliebigen Elementen zusammensetzen.

Eine weitere Besonderheit von SwiftUI-Views ist, dass diese typischerweise auf Structures und nicht auf Klassen basieren. Tatsächlich lassen sich Klassen nur dann für SwiftUI-Views verwenden, wenn diese als `final` deklariert sind und es somit keine Subklassen mehr von ihnen geben kann.

Hintergrund dieser Eigenschaft ist, dass Views in SwiftUI deutlich kompakter sind und – bildlich gesprochen – mit weniger Ballast auskommen. Eine View in SwiftUI

besitzt nur die Eigenschaften und Funktionen, die sie tatsächlich benötigt; nicht mehr und nicht weniger.

Bei `UIView` oder `NSView` ist das anders. Diese Klassen besitzen von vornherein eine Vielzahl an Eigenschaften und Funktionen, die automatisch auch jede selbst kreierte View erhält, sobald wir eine entsprechende `UIView`- oder `NSView`-Subklasse erstellen. Dazu gehören beispielsweise die Hintergrundfarbe oder der Alpha-Wert. Sie sind Teil jeder Subklasse, ob wir sie brauchen oder nicht. In SwiftUI gibt es diesen potentiellen Overhead nicht.

1.2.1 Eine erste SwiftUI-View

Ein einfaches Beispiel für eine simple View in SwiftUI finden Sie in Listing 1.1. Sie gibt ein Label mit dem Text „Hallo SwiftUI!“ auf dem Bildschirm aus, so wie in Bild 1.3 zu sehen. Zum besseren Verständnis schlüssele ich im Folgenden auf, was genau in Listing 1.1 geschieht und wie sich eine View in SwiftUI zusammensetzt.

Listing 1.1 Eine simple SwiftUI-View

```
struct MyView: View {
    var body: some View {
        Text("Hallo SwiftUI!")
    }
}
```

Los geht’s zunächst mit der Deklaration einer neuen View auf Basis einer Structure. In dem gezeigten Beispiel trägt die View den Namen `MyView`.

Im Anschluss kommt die erste Besonderheit von SwiftUI zum Tragen: `MyView` ist konform zum `View`-Protokoll. Das ist eine grundlegende Voraussetzung für jede View, die auf Basis von SwiftUI erstellt wird. Um die Anforderungen des Protokolls zu erfüllen, muss `MyView` die `body`-Property implementieren. Die bestimmt, wie die View aussieht, welche Inhalte sie darstellt und über welche Funktionen sie verfügt. Als Rückgabewert wird `some View` festgesetzt, was bedeutet, dass `body` jede beliebige Art von View zurückliefern kann.

Bis zu diesem Punkt läuft die grundlegende Erstellung einer View auf Basis von SwiftUI immer identisch ab. Es wird eine zum `View`-Protokoll konforme Structure erstellt und darin wenigstens die `body`-Property implementiert.

Die View im Beispiel aus Listing 1.1 stellt ein Label dar, das den Text „Hallo SwiftUI!“ ausgibt. Für solche Label steht im SwiftUI-Framework die Structure `Text` zur Verfügung, die als Parameter den anzuzeigenden Inhalt als String erwartet. Das gewünschte Label kann daher mithilfe des folgenden Befehls erzeugt werden:

```
Text("Hallo SwiftUI!")
```

Da damit die Konfiguration unserer ersten View bereits abgeschlossen ist, liefert die `body`-Property von `MyView` jenes Label als Ergebnis zurück. Hierzu noch eine Info am Rande: Normalerweise würde man das Schlüsselwort `return` nutzen, um das Ergebnis der Property zurückzuliefern. Aufgrund einer Neuerung in Swift 5.1 ist das aber nicht nötig, wenn die Implementierung des Property-Getters aus nur einem einzigen Befehl besteht, der gleichzeitig dem gewünschten Ergebnis entspricht; also genau so, wie es in Listing 1.1 der Fall ist. SwiftUI macht regen Gebrauch von dieser und weiteren Verbesserungen in Swift 5.1, um den Code möglichst kompakt und übersichtlich zu halten.

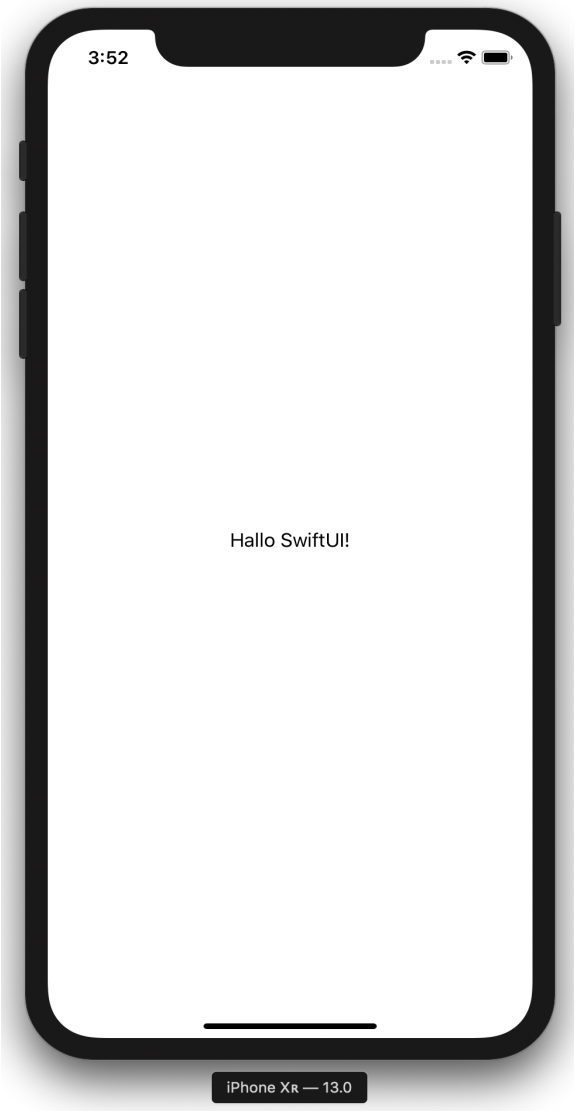

Bild 1.3 Die mit SwiftUI erzeugte View zeigt ein einfaches Label an.

1.2.2 SwiftUI-Dateien

SwiftUI-Code wird – genau wie herkömmlicher Swift-Code auch – in Dateien mit der Endung *swift* gespeichert. Entsprechend gibt es kein spezielles Dateiformat für die Arbeit mit SwiftUI, da die Views einzig und allein auf Basis von Swift-Code erzeugt werden. Jede beliebige Swift-Datei kann somit auch Implementierungen auf Basis von SwiftUI enthalten.

Nichtsdestotrotz bringt Xcode eine eigene Template-Vorlage für neue SwiftUI-Views mit. Sie befindet sich im Abschnitt *User Interface* und trägt den passenden Namen *SwiftUI View* (siehe Bild 1.4). Genau wie die *Swift File*-Vorlage fügt sie dem zugrundeliegenden Projekt eine neue Swift-Datei hinzu, kümmert sich hierbei aber bereits um den Import des SwiftUI-Frameworks. Außerdem wird die Basis für eine View erzeugt, in der dann nur noch die `body`-Property mit dem gewünschten Inhalt gefüllt werden muss (siehe Bild 1.5).

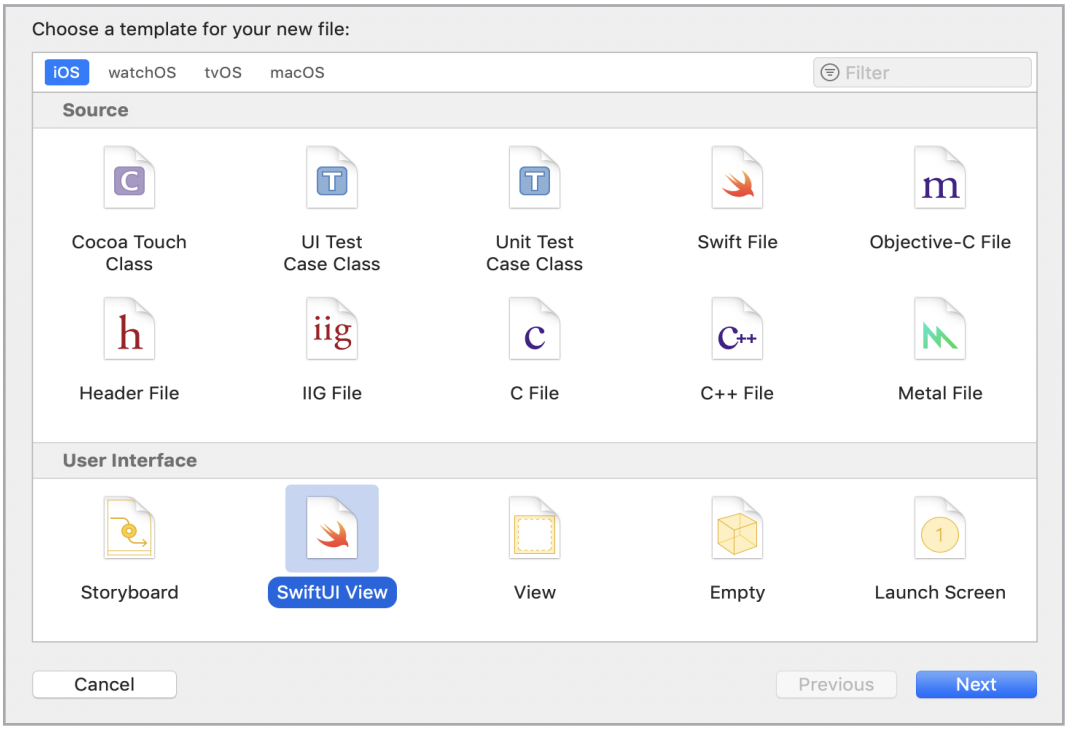

Bild 1.4 Neue Views auf Basis von SwiftUI lassen sich in Xcode über eine entsprechende Vorlage einem Projekt hinzufügen.

```
8
9  import SwiftUI
10
11 struct NewView: View {
12     var body: some View {
13         Text("Hello World!")
14     }
15 }
16
17 struct NewView_Previews: PreviewProvider {
18     static var previews: some View {
19         NewView()
20     }
21 }
22
```

Bild 1.5 Die „SwiftUI View"-Vorlage in Xcode erzeugt automatisch das Basiskonstrukt für die Erstellung einer neuen View.

Eine weitere Besonderheit bei Dateien, die mit der *SwiftUI View*-Vorlage erzeugt wurden,ist der sogenannte *Preview-Provider*, dessen Deklaration sich am Ende befindet. Dieser ist für die Vorschau verantwortlich, die standardmäßig für eine View am rechten Rand des Editors angezeigt wird (siehe Bild 1.6). **Wichtig:** Diese Vorschau wird nur angezeigt, wenn auf dem zugrundeliegenden Mac mindestens macOS Catalina 10.15 installiert ist (Xcode 11 alleine reicht für diese Funktion nicht aus).

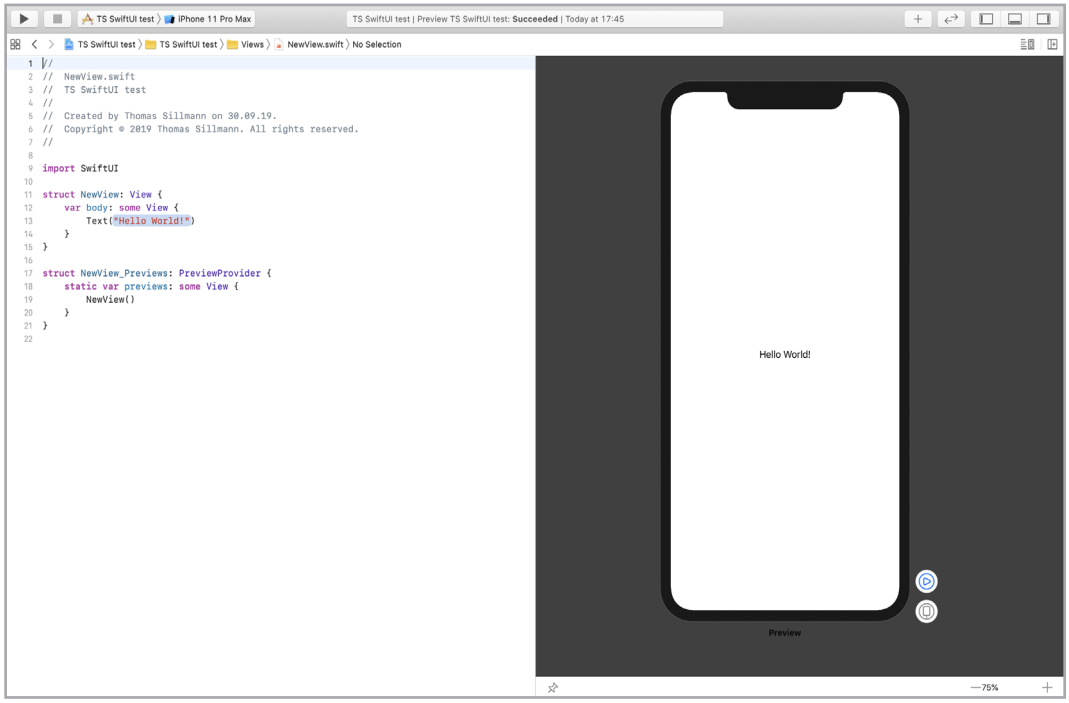

Bild 1.6 Die Vorschau am rechten Rand wird anhand des Preview-Provider-Codes erzeugt.

1.2.3 Neues Xcode-Projekt auf Basis von SwiftUI

Neben dem Hinzufügen von SwiftUI-Views zu bereits bestehenden Projekten bietet Xcode 11 auch die Option an, ein neues Projekt direkt auf Basis von SwiftUI zu erstellen. Ein solches Projekt verfügt über kein Storyboard mehr, stattdessen wird die initiale View mit SwiftUI erstellt und geladen.

Um SwiftUI als Grundlage für ein neues Xcode-Projekt zu verwenden, wählt man in den Projektoptionen unter *User Interface* den Punkt SwiftUI aus (siehe Bild 1.7). Innerhalb des neuen Projekts erhält man so statt eines Storyboards eine erste SwiftUI-View, die beim Starten der App geladen und angezeigt wird. Diese View entspricht in ihrem Aufbau der einer manuell in Xcode neu hinzugefügten SwiftUI-View (siehe hierzu auch Abschnitt 1.2.2, „SwiftUI-Dateien“).

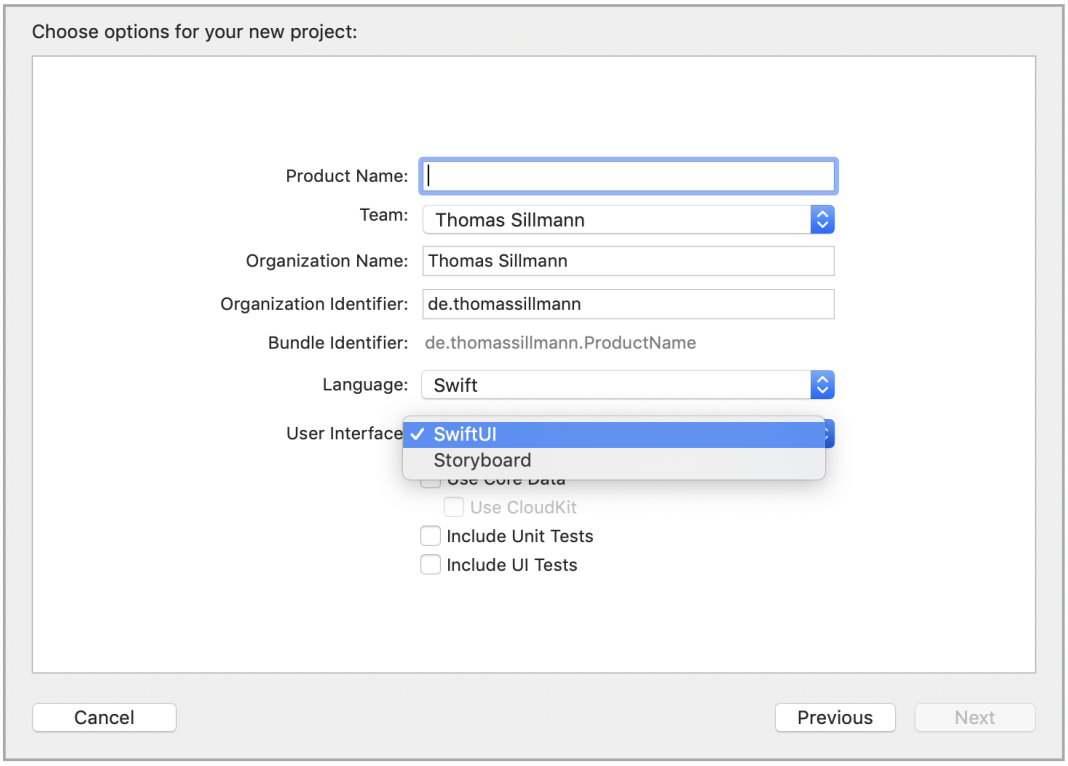

Bild 1.7 Ein neues Xcode-Projekt lässt sich direkt auf Basis von SwiftUI erstellen.

1.2.4 Gruppieren von Views

Erstellt man eine View mit SwiftUI, verknüpft man in der Regel mehrere verschiedene Views zu einer gemeinsamen Einheit. Eine Zelle für eine Liste beispielsweise kann sich so aus einem Bild und einem Text zusammensetzen, die beide nebeneinander angezeigt werden.

Solche Konstrukte werden in SwiftUI mithilfe von *Stacks* umgesetzt. Ein Stack fasst mehrere Views zusammen und ordnet diese entweder horizontal, vertikal oder hintereinander an. Wie eine solche Anordnung erfolgt, hängt vom verwendeten Stack ab. Insgesamt gibt es in SwiftUI drei verschiedene Stacks:

- `HStack`: Ordnet Views *horizontal* an
- `VStack`: Ordnet Views *vertikal* an
- `ZStack`: Ordnet Views *hintereinander* an

All diese Stacks stellen eigene Views in SwiftUI dar, liefern also eine Ansicht zurück, die sich aus den Elementen zusammensetzt, die man dem Stack übergeben hat. Entsprechend kann beispielsweise die `body`-Property einer eigenen SwiftUI-View direkt einen solchen Stack als Ergebnis zurückliefern.

Um die Views, die man als Teil eines Stacks verwenden möchte, zuzuweisen, reicht es die einzelnen Views nacheinander in Form einer Gruppe anzugeben. Dazu findet sich in Listing 1.2 ein simples Beispiel. Darin ist eine View namens `MyStackView` deklariert, die innerhalb der `body`-Property eine View auf Basis eines `VStack` zurückliefert. Dieser `VStack` setzt sich aus insgesamt drei verschiedenen Text-Views zusammen, die innerhalb geschweifter Klammern als Teil des Stacks definiert werden. Da ein `VStack` seine Inhalte vertikal anordnet, bedeutet das, dass die drei Texte untereinander angeordnet werden. Das Ergebnis dieser simplen View ist in Bild 1.8 zu sehen.

Listing 1.2 Einsatz eines Stacks in SwiftUI

```
struct MyStackView: View {
    var body: some View {
        VStack {
            Text("Das ist der erste Text ...")
            Text("... innerhalb eines VStack ...")
            Text("... bestehend aus insgesamt drei Views.")
        }
    }
}
```

Bild 1.8 Mithilfe eines Stacks werden verschiedene Views untereinander angeordnet.

Stacks lassen sich darüber hinaus auch ineinander verschachteln. So kann beispielsweise ein `VStack` auch einen `HStack` enthalten und umgekehrt. So lassen sich selbst komplexe View-Hierarchien mithilfe von Stacks abbilden.

Ein entsprechendes Beispiel dazu zeigt Listing 1.3. Darin kommt als Basis für eine View ein `VStack` zum Einsatz, der am Anfang und am Ende einen statischen Text ausgibt. Dazwischen befindet sich ein `HStack`, der mehrere Texte nebeneinander anordnet. Das Ergebnis dieses Codes ist in Bild 1.9 zu sehen.

Listing 1.3 Verschachtelung von Stacks

```
struct MyView: View {
    var body: some View {
        VStack {
            Text("Lottozahlen")
            HStack {
                Text("19")
                Text("99")
                Text("7")
                Text("31")
                Text("30")
                Text("21")
            }
            Text("Superzahl: 3")
        }
    }
}
```

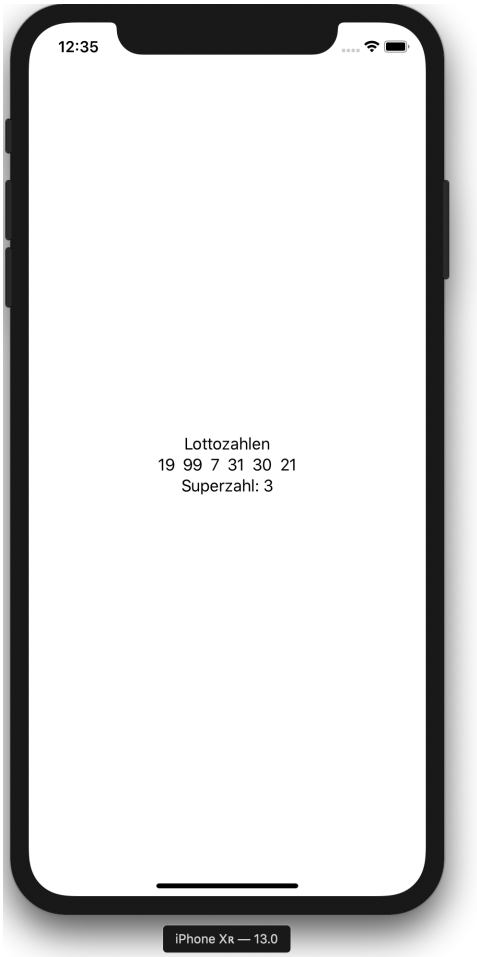

Bild 1.9 Zum Abbilden komplexerer View-Hierarchien können Stacks auch ineinander verschachtelt werden.