HANSER

Leseprobe

zu

„Einstieg in SwiftUI“

von Thomas Sillmann

Print-ISBN: 978-3-446-46362-2
E-Book-ISBN: 978-3-446-46587-9
E-Pub-ISBN: 978-3-446-46648-7

Weitere Informationen und Bestellungen unter
http://www.hanser-fachbuch.de/978-3-446-46362-2
sowie im Buchhandel

© Carl Hanser Verlag, München

Inhalt

Vorwort ... **XI**

1 Über SwiftUI ... **1**
1.1 Programmierung mit SwiftUI ... 3
1.2 Die Preview ... 5
1.3 Voraussetzungen ... 8
1.4 Integration ... 9

2 Grundlagen ... **11**
2.1 Das View-Protokoll ... 11
2.1.1 Ablauf der View-Generierung ... 14
2.1.2 Structure vs. Klasse ... 14
2.2 Grundlagen der View-Erstellung ... 15
2.2.1 Text und Image ... 15
2.2.2 Views organisieren mittels Stacks ... 16
2.2.3 Views mittels Modifier anpassen ... 19
2.2.3.1 Funktionsweise von Modifiern ... 22
2.2.3.2 Auszug verfügbarer Modifier ... 26
2.2.4 Einsatz von Library und Preview ... 28
2.2.5 Layout-System ... 30
2.3 Status ... 31
2.3.1 Property ... 32
2.3.2 State ... 33
2.3.3 Binding ... 34

3 Views, Controls und Container ... **39**
3.1 Text und Grafiken ... 39
3.1.1 Text ... 39
3.1.2 TextField ... 42
3.1.3 SecureField ... 45
3.1.4 TextEditor ... 46
3.1.5 Image ... 47

3.2 Buttons 53
3.2.1 Button 53
3.2.2 EditButton 57
3.2.3 PasteButton 57
3.2.4 Menu 58
3.2.5 Weitere Buttons 60
3.3 Value Selectors 61
3.3.1 Toggle 61
3.3.2 Picker 66
3.3.3 DatePicker 70
3.3.4 Slider 74
3.3.5 Stepper 78
3.4 Value Indicators 83
3.4.1 ProgressView 83
3.4.2 Label 85
3.4.3 Link 86
3.5 Stacks 87
3.5.1 HStack 88
3.5.2 VStack 92
3.5.3 ZStack 96
3.5.4 LazyHStack und LazyVStack 98
3.6 Grids 99
3.7 Listen und Scroll-Views 103
3.7.1 List 103
3.7.2 ForEach 112
3.7.3 ScrollView 125
3.8 Container-Views 128
3.8.1 Form 128
3.8.2 Group 130
3.8.3 GroupBox 134
3.8.4 Section 136
3.9 Weitere Views 139
3.9.1 Spacer 139
3.9.2 Divider 142

4 Navigation und Präsentation 143

4.1 NavigationView 143
4.1.1 Grundlagen 143
4.1.2 Festlegen einer Standardansicht für die Ziel-View 146
4.1.3 Ändern des NavigationView-Styles 148
4.1.4 Setzen eines NavigationView-Titels 150
4.1.5 Navigation-Bar ausblenden 152
4.1.6 Setzen von Navigation-Bar-Items 153

4.1.7 Alternatives Auslösen eines NavigationLink 155
4.1.8 Navigationsstrukturen unter watchOS 157

4.2 TabView 158
4.2.1 Grundlagen 159
4.2.2 Programmatisches Wechseln eines Tab-Bar-Items 162

4.3 HSplitView und VSplitView 164

4.4 Sheet 165
4.4.1 Sheet auf Basis eines Boolean 165
4.4.2 Sheet auf Basis eines Identifiable-Items 166
4.4.3 Reaktion auf Ausblenden eines Sheets 168

4.5 Alert 169
4.5.1 Erstellen eines Alert 169
4.5.2 Einblenden eines Alert auf Basis eines Boolean 171
4.5.3 Einblenden eines Alert auf Basis eines Identifiable-Items 172

4.6 ActionSheet 174
4.6.1 Erstellen eines ActionSheet 175
4.6.2 Einblenden eines ActionSheet auf Basis eines Boolean 177
4.6.3 Einblenden eines ActionSheet auf Basis eines Identifiable-Items 177

5 Status 179

5.1 Property 180
5.2 State 182
5.3 Binding 183
5.4 ObservedObject 186
5.4.1 Datenmodell vorbereiten 187
5.4.2 Datenmodell in SwiftUI-View einbinden 187
5.4.3 Auf Änderungen reagieren 190

5.5 StateObject 193
5.6 EnvironmentObject 194
5.7 Environment 200
5.8 Zusammenfassung: Welcher Status für welche Situation? 202

6 Integration 205

6.1 Hosting 205
6.1.1 NSHostingController und UIHostingController 206
6.1.2 WKHostingController 207

6.2 Representables 208
6.2.1 Erstellen einer Representable-View 210
6.2.2 Aktualisieren einer Representable-View 213
6.2.3 Weitergabe von Aktualisierungen an SwiftUI 215

7 Preview und Xcode ... 221

7.1 Funktionsweise der Preview ... 222
7.2 Arbeiten mit der Preview ... 224
7.3 Konfiguration der Preview ... 227
7.4 Mehrere Previews parallel einsetzen ... 228
7.5 Preview ausführen ... 231
7.6 Preview auf Device ausführen ... 233
7.7 Library ... 234
7.8 Kontext-Actions ... 239

Nachwort ... 243

Index ... 245

Vorwort

Liebe Leserin, lieber Leser,

ich entwickle seit inzwischen über zehn Jahren Apps für die verschiedenen Plattformen von Apple. Die Einführung des iPhone und des App Store hat meinen Werdegang maßgeblich beeinflusst und sorgte dafür, dass ich mich heute voll und ganz dem Apple-Kosmos verschrieben habe.

In all diesen Jahren gab es viele kleine Evolutionen, die uns App-Entwicklern das Leben erleichterten. Die Einführung von Automatic Reference Counting vereinfachte die Speicherverwaltung deutlich. Storyboards öffneten ganz neue Wege, App-Strukturen umzusetzen und Views zu gestalten. Auto Layout verbesserte die Möglichkeiten, Views für verschiedene Bildschirmgrößen zu optimieren.

Daneben gab es auch einige wenige *große* Revolutionen. Eine davon war die Einführung der Programmiersprache Swift. Eine andere zeichnet sich erst seit jüngster Zeit ab. Die Rede ist von *SwiftUI*.

Mit SwiftUI ändert sich maßgeblich, wie Views für die verschiedenen Plattformen von Apple umgesetzt werden. Es gibt keine View-Controller mehr, nur Views. Die basieren auf Structures, nicht auf Klassen. Ihre Erstellung erfolgt deklarativ, nicht imperativ. Und ein Status bestimmt, welches Verhalten sie an den Tag legen und unter welchen Bedingungen sie sich aktualisieren.

Die Arbeit mit SwiftUI ist so gänzlich anders als das, was man all die letzten Jahre mit AppKit, UIKit und WatchKit gewohnt ist. Gleichzeitig zeichnet sich jetzt bereits ab, wie mächtig dieses neue UI-Framework von Apple ist. Noch nie war es leichter, ansprechende Nutzeroberflächen zu erstellen. Und noch nie brauchte es dafür so wenige Zeilen Code wie mit SwiftUI.

Dazu kommt, dass SwiftUI auf allen Apple-Plattformen zur Verfügung steht. Hat man die grundlegende Funktionsweise demnach einmal verinnerlicht, ist man imstande, Views für macOS, iOS (und iPadOS), watchOS sowie tvOS zu erstellen. SwiftUI stellt ein gemeinsames Toolset dar, das sich im gesamten Apple-Kosmos nutzen lässt.

Seit der erstmaligen Vorstellung von SwiftUI auf der WWDC 2019 bin ich begeistert von diesem Framework. Wie mächtig es ist, wird mir jedes Mal bewusst, wenn ich in Projekten auf die „alten“ Techniken zur Erstellung von Nutzeroberflächen mittels Storyboards und View-Controllern zurückgreife. Im Vergleich ist die Arbeit mit SwiftUI um so vieles komfortabler.

SwiftUI stellt die Zukunft der UI-Erstellung für Apple-Plattformen dar, und mit diesem Buch möchte ich Ihnen einen passenden Einstieg zur Verfügung stellen. In den folgenden Kapiteln erfahren Sie, wie SwiftUI funktioniert und welche Views Ihnen zur Verfügung stehen. Auch gehe ich im Detail auf den Status ein und wie er sich auf die Aktualisierung von Ansichten auswirkt. Ebenso kommt die Integration von SwiftUI in bestehende Projekte auf Basis von Storyboards nicht zu kurz.

Zusätzlich erhalten Sie zusammen mit diesem Buch noch Zugriff auf einen ganz besonderen Service: Dank Update inside kommen Sie in den Genuss von Zusatzkapiteln, die nach und nach veröffentlicht werden. Neben weiteren Themen, die es aus Platzgründen nicht mehr in dieses Buch geschafft haben, werden Sie so auch über kommende SwiftUI-Updates informiert. Der Update-Service läuft bis Oktober 2022. Sie werden persönlich von uns benachrichtigt, wenn neue Updates zum Download zur Verfügung stehen. Registrieren Sie sich dazu einfach unter *www.hanser-fachbuch.de/swiftui-update* mit dem Passwort von der zweiten Seite dieses Buches.

Nun bleibt mir nur noch zu sagen, dass ich Ihnen von Herzen viel Freude mit diesem Buch und der Arbeit mit SwiftUI wünsche. Ergänzende Artikel und Videos rund um die Entwicklung für Apple-Plattformen finden Sie auf meinem Blog unter letscode.thomassillmann.de.

Ihr Thomas Sillmann

Aschaffenburg, August 2020

2 Grundlagen

Mit diesem Kapitel geht es nun richtig los! ☺ Nach dem ersten groben Überblick zu SwiftUI im vorangegangenen Kapitel erfahren Sie nun, wie das Framework konkret funktioniert und mithilfe welcher Bestandteile und Techniken Sie Views umsetzen. Hierbei lernen Sie bereits alle grundlegenden Bestandteile und Mechanismen kennen, die SwiftUI so einzigartig machen.

2.1 Das View-Protokoll

Das `View`-Protokoll stellt die Basis einer jeden View in SwiftUI dar. Es definiert die elementaren Eigenschaften und Funktionen, die jede Ansicht mitbringen muss.

In gewisser Weise ist es vergleichbar mit der `UIView`-Klasse aus UIKit beziehungsweise der `NSView`-Klasse aus dem AppKit-Framework. Diese dienen ihrerseits als Basis für alle Ansichten, die über das jeweilige Framework erzeugt werden. Und genauso muss jede View, die wir mittels SwiftUI erstellen, konform zum `View`-Protokoll sein (siehe Bild 2.1).

Die wichtigste Eigenschaft des View-Protokolls ist eine Property namens `body`. Über sie definiert man das Aussehen und die Funktionsweise einer Ansicht. Als Ergebnis liefert `body` ebenfalls ein Element zurück, das konform zum `View`-Protokoll ist.

Lassen Sie uns anhand dieser ersten Informationen direkt ein erstes konkretes Beispiel betrachten. Dazu finden Sie in Listing 2.1 die Umsetzung einer einfachen SwiftUI-View, die ein Label mit dem Text „Hello, World!“ ausgibt. Die View besitzt den Namen `ContentView` und ist als Structure umgesetzt (siehe hierzu auch Abschnitt 2.1.2, „Structure vs. Klasse“). Sie liefert eine Instanz vom Typ `Text` zurück. `Text` ist eine Structure aus dem SwiftUI-Framework und seinerseits konform zum `View`-Protokoll, entsprechend können wir es als Rückgabewert für die `body`-Property nutzen (mehr zu den verfügbaren Views und Controls in SwiftUI erfahren Sie in Kapitel 3 dieses Buches). `Text` ist vergleichbar mit der Klasse `UILabel` aus dem UIKit-Framework und dient zur Darstellung kurzer Texte.

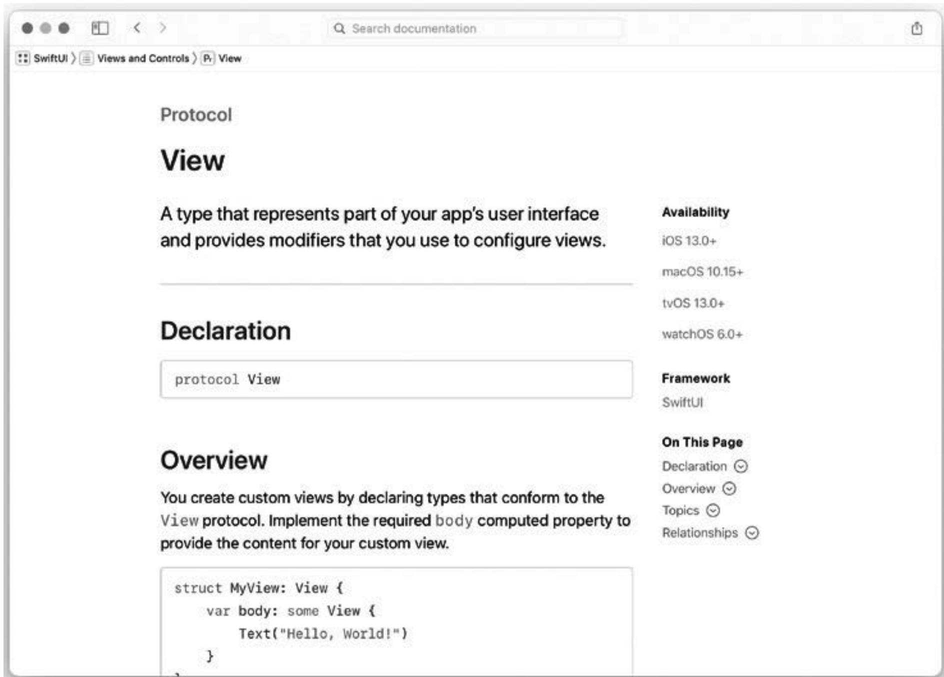

Bild 2.1 Alle Views in SwiftUI basieren auf dem View-Protokoll.

Listing 2.1 SwiftUI-View zur Darstellung eines Labels

```
struct ContentView: View {
    var body: some View {
        Text("Hello, World!")
    }
}
```

Warum kein return?

Womöglich wundern Sie sich, warum das Ergebnis der body-Property – die Text-Instanz mit dem Inhalt „Hello, World!" – nicht explizit mittels `return` zurückgegeben wird. Hierbei handelt es sich um ein Feature von Swift 5.1, der sogenannten *Shorthand Getter Declaration*. Sie erlaubt es uns – sollte der Getter einer Computed Property aus nur einem einzigen Befehl bestehen, der gleichzeitig das gewünschte Ergebnis generiert – auf die explizite Angabe von `return` zu verzichten.

Der Einsatz der Shorthand Getter Declaration ist in SwiftUI gang und gäbe. Sie trägt ihren Teil dazu bei, den Quellcode von SwiftUI-Views möglichst kompakt und übersichtlich zu halten.

Eine wichtige Rolle bei der Deklaration von SwiftUI-Views spielt der Typ der body-Property. Sofern es keinen triftigen Grund gibt (und mir selbst ist ein solcher bisher bei der Arbeit mit SwiftUI noch nicht unterkommen), sollte man diesen immer mittels `some View` deklarieren (so wie auch in Listing 2.1 zu sehen).

Um eines vorneweg klarzustellen: Das Beispiel aus Listing 2.1 funktioniert auch dann tadellos, wenn man als Typ für die body-Property explizit `Text` angibt. Es reicht umgekehrt jedoch nicht aus, schlicht das Protokoll `View` für die Typ-Deklaration zu nutzen (siehe hierzu auch Listing 2.2).

Listing 2.2 Deklaration der body-Property

```
// Möglich!
var body: Text {
    // ...
}

// Fehler!
var body: View {
    // ...
}
```

Der Grund ist, dass das `View`-Protokoll einen Associated Type definiert. Hierbei handelt es sich um den Typ jener Ansicht, die man anzeigen möchte (im vorangegangenen Beispiel war das ein Text). `View` benötigt zwingend die Information darüber, welcher konkrete Typ als Associated Type verwendet werden soll, um funktionieren zu können. Definiert man – wie im zweiten Beispiel von Listing 2.2 zu sehen – als Rückgabewert für eine Eigenschaft oder Funktion schlicht den Protokoll-Typ `View`, fehlt jener konkrete Associated Type, und es kommt zu einem Compiler-Fehler.

Doch auch die explizite Angabe des konkreten Typs, den man mittels body-Property zurückliefert, ist nicht ideal. Hierbei spielen vorrangig zwei Gründe eine Rolle:

- Gerade bei der Arbeit mit SwiftUI kann es schnell passieren, dass sich der Rückgabetyp einer View ändert. Ergänzt man beispielsweise die Ansicht aus Listing 2.1 noch um ein zusätzliches Bild, das neben dem Label angezeigt wird, gibt die View keine `Text`-Instanz mehr zurück. Stattdessen stellt die View nun einen Stack dar, der einen Text und ein Bild enthält (dazu später mehr). Geben wir nun den Rückgabetyp einer View in der body-Property explizit an, müssen wir ihn jedes Mal ändern, wenn wir die Struktur und den Aufbau der Ansicht aktualisieren. Das ist ein Aufwand, den wir uns in der Regel sparen wollen.
- Möglicherweise soll nach außen hin gar nicht ersichtlich sein, welchem konkreten Typ die body-Property einer View entspricht. Das ist insbesondere dann ein sehr wichtiger Punkt, falls man ein Framework erstellt und Teile der Implementierung nicht zugänglich sein sollen. In diesem Fall kommt eine explizite Angabe des Rückgabetyps von `body` schlicht nicht in Frage.

Die Lösung für diese Problematik sind die mit Swift 5.1 eingeführten sogenannten *Opaque Types*. Mithilfe eines solchen Opaque Type lässt sich der konkrete Typ, den eine Funktion zurückliefert, verschleiern. Gleichzeitig legt man fest, dass jener konkrete Typ immer identisch ist.

Um einen solchen Opaque Type zu definieren, nutzt man das Schlüsselwort `some`. Genau einen solchen Opaque Type nutzten wir in Listing 2.1 als Rückgabetyp für die `body`-Property:

```
var body: some View { ... }
```

So sind die beiden aufgeführten Probleme gelöst: In `body` können wir nun eine beliebige View zurückgeben, solange diese nur konform zum `View`-Protokoll ist. Da spielt es keine Rolle, ob es sich bei dem Ergebnis nun um einen Text, ein Bild oder eine Liste handelt. Gleichzeitig ist aus der `body`-Deklaration heraus nicht ersichtlich, welche *konkrete* View zurückgeliefert wird. Das ist ausschließlich in der Implementierung von `body` definiert.

Mit SwiftUI erstellte Ansichten deklariert man in der Regel immer mittels des Opaque Type `some View`. Das bietet die größtmögliche Flexibilität und ist die ideale Grundlage für SwiftUI-Views.

2.1.1 Ablauf der View-Generierung

Wird eine SwiftUI-View geladen, führt das automatisch zum Aufruf der zugehörigen `body`-Property. Bei näherer Betrachtung erscheint dieses Konzept aber fehlerträchtig. Denn wie wir wissen, generiert man innerhalb von `body` wiederum selbst eine View. Diese springt entsprechend dann in die Implementierung *ihrer* `body`-Property und so weiter. Es entsteht demnach eine Endlosschleife, die in vereinfachter Form in Bild 2.2 skizziert ist.

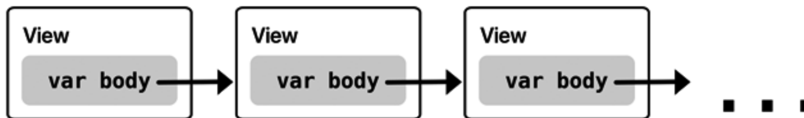

Bild 2.2 Eine View generiert aus ihrer body-Property heraus eine weitere View.

Um dieses Problem zu lösen, stehen innerhalb des SwiftUI-Frameworks sogenannte *Primitive Views* zur Verfügung. Eine Primitive View beendet den beschriebenen Kreislauf und dient als Endpunkt. Sie stellt das letzte Glied in dieser View-Hierarchie dar.

Zu den Primitive Views in SwiftUI zählen unter anderem die Elemente `Text`, `Color`, `Spacer`, `Image`, `Shape` und `Divider`. Mehr zu den verschiedenen Views, die im SwiftUI-Framework enthalten sind, erfahren Sie in den Kapiteln 3 und 4 dieses Buches.

2.1.2 Structure vs. Klasse

Da es sich bei `View` um ein Protokoll handelt, ist es theoretisch möglich, es nicht nur auf Structures, sondern auch auf Klassen anzuwenden. Bei der Arbeit mit SwiftUI kommt das aber nicht infrage. Views in SwiftUI basieren **immer** auf Structures, nicht auf Klassen.

Doch warum ist das so? Aus AppKit, UIKit und WatchKit ist man es schließlich gewohnt, Views und View-Controller immer als Klassen umzusetzen. Das hängt nicht zuletzt mit der Vererbung zusammen. Eine Klasse wie `UIView` bringt so ein grundlegendes Set an Eigenschaften und Funktionen mit, auf dem alle Subklassen aufbauen können.

Genau das ist aber auch das Problem, das Apple mit SwiftUI zu lösen versucht. Denn jede View aus AppKit, UIKit und WatchKit bringt eine Vielzahl an Properties und Methoden mit, von denen die meisten nicht benötigt werden. Man werfe nur einmal einen Blick in die Dokumentation von Klassen wie `NSView` oder `UIView` und stelle sich selbst die Frage, wie viele davon man durchschnittlich für eigens erstellte Views benötigt.

Durch den Wechsel auf Protokolle und Structures fällt dieser Overhead weg. Eine View in SwiftUI ist vollkommen reduziert auf die Eigenschaften und Funktionen, die die jeweilige View benötigt; nicht mehr und nicht weniger. Tatsächlich sollen Views in SwiftUI immer möglichst kleingehalten werden und eine ganz konkrete Aufgabe erfüllen. Sie entsprechen mehr eigenen kleinen Funktionen als aufgeblähten Ansichten.

Zweifellos ist hier für Entwickler, die bereits Erfahrungen mit AppKit, UIKit und WatchKit gesammelt haben, ein Umdenken notwendig. Doch gerade weil SwiftUI gänzlich eigene Wege geht und sich kaum an den bereits vorhandenen Konzepten orientiert, stellt es auch eine so innovative und spannende neue UI-Lösung dar.

2.2 Grundlagen der View-Erstellung

In den folgenden Abschnitten setzen wir uns mit den Grundlagen der View-Erstellung unter SwiftUI auseinander. Hierbei stelle ich Ihnen auch diverse erste View-Typen vor, die bei der täglichen Arbeit mit SwiftUI in der Regel häufig benötigt werden. Mehr zu den verfügbaren Views und Controls in SwiftUI erfahren Sie in Kapitel 3 dieses Buches.

2.2.1 Text und Image

Zwei der häufigsten Elemente, die einem bei der Arbeit mit SwiftUI begegnen, sind `Text` und `Image`. Ersteres dient zur Ausgabe von Zeichenketten, letzteres zur Darstellung von Bildern.

Listing 2.3 zeigt beispielhaft, wie Sie Views auf Basis dieser Elemente erstellen und verwenden. `TextView` gibt hierbei den String „Hello SwiftUI“ auf dem Display aus, während `ImageView` die Grafik eines Buches anzeigt. Diese Grafik stammt aus SF Symbols, der umfangreichen Bildbibliothek von Apple.

Listing 2.3 Erstellen einer Text- und Image-Instanz

```
struct TextView: View {
    var body: some View {
        Text("Hello SwiftUI!")
    }
}

struct ImageView: View {
    var body: some View {
        Image(systemName: "book.fill")
    }
}
```

Die Structure `Image` bringt neben dem hier gezeigten `init(systemName:)` noch weitere Initializer mit. Diese erlauben es, Bilder aus einem Asset-Catalog zu laden oder Instanzen auf Basis von `NSImage` beziehungsweise `UIImage` zu verwenden.

Weitere Informationen zu `Text` und `Image` finden Sie in Kapitel 3 dieses Buches. In Abschnitt 2.2.3, „Views mittels Modifier anpassen“, erfahren Sie mehr über die Konfigurationsmöglichkeiten von Views. Dazu gehören beispielsweise das Ändern von Schriftgrößen und Farben.

2.2.2 Views organisieren mittels Stacks

Zu den wichtigsten Views in SwiftUI gehören die sogenannten *Stacks*. Ein Stack setzt sich aus mehreren Views zusammen und ordnet diese an. *Wie* die Anordnung der zugehörigen Views erfolgt, hängt vom jeweiligen Stack ab. Stacks gibt es in insgesamt drei Ausführungen:

- `VStack` ordnet Views vertikal untereinander an.
- `HStack` ordnet Views horizontal nebeneinander an.
- `ZStack` legt Views übereinander.

Während `VStack` und `HStack` bei der Arbeit mit SwiftUI in der Regel ständig im Einsatz sind, findet ein `ZStack` eher in Spezialfällen Verwendung. Dazu gehört beispielsweise die Verwendung eines Bildes als Hintergrund, über das man einen Text legt.

Alle drei Stack-Views gehören zu den sogenannten *Container-Views* in SwiftUI. Container-Views setzten sich aus einer oder mehreren anderen Views zusammen und organisieren sie auf eine festgelegte Art und Weise (mehr dazu erfahren Sie in Kapitel 3 dieses Buches). Im Falle der Stacks geht es schlicht um die Anordnung und den Aufbau der zugehörigen Views.

Ein erstes einfaches Beispiel dazu finden Sie in Listing 2.4. Die darin deklarierte `ContentView` setzt sich aus einem `VStack` zusammen, der selbst wiederum zwei Views enthält: eine `Text`- und eine `Image`-Instanz. Der Einsatz von `VStack` sorgt dafür, dass die `Text`- und `Image`-View untereinander angeordnet werden, so wie in Bild 2.3 zu sehen.

Listing 2.4 Anordnung einer `Text`- und `Image`-View untereinander mittels VStack

```
struct ContentView: View {
    var body: some View {
        VStack {
            Text("Hello, World!")
            Image(systemName: "book.fill")
        }
    }
}
```

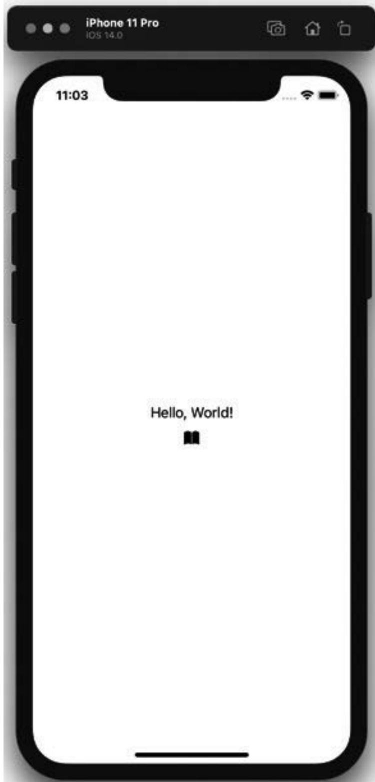

Bild 2.3
Dank eines VStack werden Text und Grafik untereinander angeordnet.

Interessant ist hierbei die Syntax, die so nicht nur bei Stacks, sondern bei *allen* Arten von Container-Views in SwiftUI zum Einsatz kommt. So handelt es sich beim letzten (und dem einzigen nicht-optionalen) Parameter bei der Initialisierung eines Stacks um ein Closure. Dank der Trailing Closure-Syntax in Swift ist es möglich (und bei der Arbeit mit SwiftUI geradezu erwünscht), die Implementierung jenes Closures direkt an die Initialisierung des Stacks zu koppeln (daher auch direkt die geschweiften Klammern nach der Typangabe `VStack`).

Des Weiteren ist dieses Closure ein besonderer Parameter, es handelt sich dabei nämlich um einen sogenannten *View-Builder*. View-Builder basieren auf der gleichnamigen Structure `ViewBuilder` und besitzen die Aufgabe, eine View auf Basis eines Closures zu erzeugen (eben genau das, was wir mithilfe von Stacks erreichen).

Das führt dazu, dass alle Views, die schlicht Teil dieses Closure-Parameters sind, von dem zugrunde liegenden Container passend verarbeitet werden. Daher reicht es aus, nacheinander alle Views in solch einem Closure aufzuführen, die man anzeigen möchte. Der View-Builder kümmert sich dann um den Rest.

Die View-Builder stellen eine wichtige Funktion für die deklarative Syntax von SwiftUI dar. Dank ihnen ist es möglich, eine Quellcode-Struktur, wie sie in Listing 2.4 zu sehen ist, umzusetzen. Darin spiegelt sich der genaue Aufbau der View wider. Basis ist in dem gezeigten Beispiel ein Stack, der zwei Views (einen Text und eine Grafik) vertikal untereinander anordnet.

Weitere Informationen zu View-Buildern und Container-Views erhalten Sie in Kapitel 3 des Buches.

Stacks können beliebige Views aufnehmen und darstellen. Dazu zählen auch andere Stacks. Das erlaubt komplexe Verschachtelungen, um selbst aufwendigere Views umzusetzen und zu gestalten.

Ein passendes Beispiel dazu finden Sie in Listing 2.5. Ausgangspunkt ist auch hier erneut ein `VStack`, der dieses Mal insgesamt drei Views enthält: eine `Text`-Instanz zu Beginn und am Ende des Stacks und dazwischen einen `HStack`. Der `HStack` wiederum enthält vier Views, die horizontal nebeneinander angeordnet werden. Es geht los mit einem Text, gefolgt von drei Grafiken. Das Ergebnis dieses Codes sehen Sie in Bild 2.4.

Listing 2.5 Verschachtelung von Stacks

```
struct ContentView: View {
    var body: some View {
        VStack {
            Text("Hello SwiftUI!")
            HStack {
                Text("Images:")
                Image(systemName: "book.fill")
                Image(systemName: "paperplane.fill")
                Image(systemName: "sun.max.fill")
            }
            Text("Another text ...")
        }
    }
}
```

Bild 2.4
Durch die Kombination von Stacks lassen sich auch komplexere View-Konstrukte umsetzen.

Stacks sind ein essenzielles Mittel in SwiftUI, um Views zu strukturieren und zu organisieren. Durch Mischung von `VStack` und `HStack` legt man den Aufbau seiner Views fest und bestimmt, wie sie angeordnet werden.

2.2.3 Views mittels Modifier anpassen

Man konfiguriert Views in SwiftUI prinzipiell auf zwei verschiedene Arten.

Zunächst ist da die Initialisierung. Mittels Initialisierung einer View legt man grundlegende Informationen fest, die essenziell für die Erstellung der jeweiligen View sind.

In den vorangegangenen Abschnitten sahen wir bereits einige Beispiele dazu. So übergeben wir bei Initialisierung eines Labels den anzuzeigenden Text oder legen fest, welche Grafik eine `Image`-Instanz anzeigen soll:

```
Text("Text to display ...")
Image(systemName: "book.fill")
```

Auch bei den Stacks kam dieser Mechanismus zum Einsatz. Bei deren Initialisierung legen wir unter anderem fest, welche Views Teil des zugrunde liegenden Stacks sind:

```
VStack {
    Text("Hello SwiftUI!")
    Image(systemName: "book.fill")
}
```

Allerdings gibt es noch weit mehr Konfigurationen, die man an Views durchführen kann. Dazu gehören unter anderem das Setzen einer Textfarbe, das Festlegen eines Hintergrunds und das Definieren der Größe einer View.

Alle diese Konfigurationen, die über die eigentliche Initialisierung einer SwiftUI-View hinausgehen, setzt man mithilfe der sogenannten *Modifier* um. Ein Modifier ist zunächst nichts anderes als eine Methode, die man auf einer View aufruft. Er verändert die View auf eine spezifische Art und Weise und liefert als Ergebnis eine komplett neue View-Instanz zurück.

Ein erstes einfaches Beispiel dazu finden Sie in Listing 2.6. Die darin erzeugte View setzt sich aus einer einfachen `Text`-Instanz zusammen. Auf dieser Instanz wird aber zusätzlich der Modifier `underline()` aufgerufen. Dieser fügt dem zuvor erstellten Text eine Unterstreichung hinzu (siehe Bild 2.5).

Listing 2.6 Einsatz eines Modifiers

```
struct ContentView: View {
    var body: some View {
        Text("Hello SwiftUI!")
            .underline()
    }
}
```

Bild 2.5
Mittels Modifier wurde der Text um eine Unterstreichung ergänzt.

Ein zweites Beispiel führt Listing 2.7 auf. Darin kommen gleich mehrere verschiedene Modifier zum Einsatz, um sowohl einen Text als auch ein Image anzupassen. `font(_:)` beispielsweise konfiguriert einen Text oder ein Image in einer bestimmten Schriftgröße. Welche zum Einsatz kommt, definiert man mittels des Parameters dieses Modifiers, der eine Instanz vom Typ `Font` erwartet. `foregroundColor(_:)` färbt ein Element in der gewünschten Farbe ein, `bold()` formatiert es fett.

Bei der Grafik kommen andere Modifier zum Einsatz. Zunächst sorgt `resizable()` dafür, dass sich das Bild in seiner Größe verändern lässt. Anschließend definiert `scaledToFit()`, dass die Grafik nicht gezerrt wird, sondern sich ihren Proportionen passend in den gegebenen Raum, der ihr für die Anzeige zur Verfügung steht, einfügt. Zu guter Letzt sorgt der `frame(height:)`-Modifier dafür, dass die `Image`-View eine Höhe von 40 Punkten besitzt. Das Ergebnis dieses Codes sehen Sie in Bild 2.6.

Listing 2.7 Einsatz mehrerer Modifier

```
struct ContentView: View {
    var body: some View {
        VStack {
            Text("Hello SwiftUI!")
                .font(.largeTitle)
                .foregroundColor(.yellow)
                .bold()
                .underline()
            Image(systemName: "book.fill")
                .resizable()
                .scaledToFit()
                .frame(height: 40)
        }
    }
}
```

Bild 2.6
Das Erscheinungsbild von Text und Image wurde mittels Modifier deutlich verändert.

Interessant ist die Syntax, mit der Modifier auf Views aufgerufen werden. Statt sie alle hintereinander in einer einzigen Zeile nach der Initialisierung der zugehörigen View aufzurufen, erhält jeder Modifier typischerweise seine eigene Zeile. Durch die zusätzliche Einrückung ist auch bei mehreren Views gut ersichtlich, welcher Modifier zu welchem Element gehört.

Modifier auf mehrere Views anwenden

Möchte man mehrere Views identisch formatieren, besteht die Möglichkeit, den oder die gewünschten Modifier auf den zugrunde liegenden Container anzuwenden. Der Modifier wird dann auf alle Views innerhalb des Containers übertragen. Zu sehen ist das am Beispiel in Listing 2.8. Darin finden sich innerhalb eines `VStack` drei `Text`-Instanzen, die alle mittels `font(_:)`-Modifier als `largeTitle` formatiert werden sollen. Statt diesen Modifier einzeln auf jede der drei `Text`-Instanzen anzuwenden, wird er lediglich einmalig auf dem Container aufgerufen.

Listing 2.8 Anwendung eines Modifiers auf einen Container

```
struct ContentView: View {
    var body: some View {
        VStack {
            Text("One")
            Text("Two")
            Text("Three")
        }
        .font(.largeTitle)
    }
}
```

> Wichtig hierbei: Nicht jeder Modifier lässt sich auf eine beliebige View anwenden! Manche Modifier lassen sich nur im Zusammenspiel mit spezifischen Elementen nutzen. `bold()` beispielsweise kann man nur auf Instanzen vom Typ `Text` aufrufen. Aus diesem Grund kann `bold()` auch nicht auf den `VStack` aus Listing 2.8 angewendet werden. Dieser enthält zwar nur `Text`-Instanzen, ist selbst aber eben kein `Text`-Element. Außerdem könnte sich der Inhalt des `VStack` verändern, sodass neben `Text` auch andere Views wie beispielsweise `Image` darin zum Einsatz kommen; ein übergreifender Aufruf des `bold()`-Modifiers auf den Container macht dann keinen Sinn mehr.

2.2.3.1 Funktionsweise von Modifiern

Ich habe bereits geschrieben, dass Modifier eine View anpassen und als Ergebnis eine komplett neue View zurückliefern. Diesen Umstand führe ich nun noch einmal etwas detaillierter aus, da er sehr wichtig für das Verständnis von Modifiern und deren Funktionsweise ist.

Werfen wir dazu einmal einen Blick in die Dokumentation eines Modifiers, beispielsweise die von `foregroundColor(_:)` (siehe Bild 2.7). Darin ist zu erkennen, dass diese Methode als Ergebnis eine neue View zurückliefert.

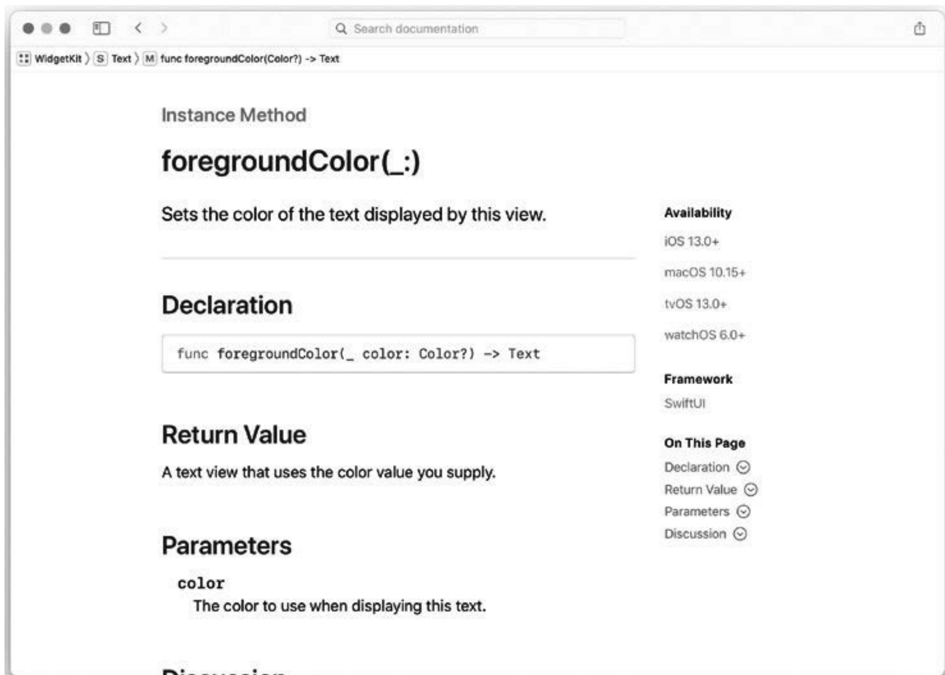

Bild 2.7 Ein Modifier liefert als Ergebnis eine neue View zurück.

Modifier arbeiten wie folgt: Zunächst dient ihnen als Basis eine View, auf die sie angewendet werden (zum Beispiel eine `Text`-Instanz). Ein Modifier manipuliert die View dann auf die zugrunde liegende Art und Weise; im Falle von `foregroundColor(_:)` wird also die Farbe geändert. Diese Änderung findet aber nicht direkt auf der View statt, über die der Modifier aufgerufen wurde. Stattdessen erzeugt der Modifier eine gänzlich neue View (auf Basis der Ursprungs-View), die die Anpassungen des Modifiers enthält.

Ruft man mehrere Modifier hintereinander auf, entsteht so eine Kette. Der erste Modifier manipuliert die Ursprungs-View und liefert als Ergebnis eine neue View zurück. Diese neue View ist die Basis des nachfolgenden Modifiers, der seinerseits Anpassungen daran vornimmt und eine neue View zurückliefert. So setzt sich das ganze fort, bis der letzte Modifier seine Konfigurationen abgeschlossen hat (siehe Bild 2.8). Das finale Ergebnis – also jene View, die wir vom letzten Modifier als Ergebnis erhalten – entspricht dann der Ansicht, die zum Einsatz kommt.

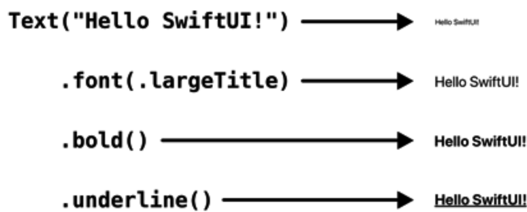

Bild 2.8 Ein Modifier baut auf dem anderen auf und nimmt weitere Konfigurationen an einer View vor.

Aufgrund dieser Funktionsweise von Modifiern gibt es aber ein sehr wichtiges Detail zu beachten: die *Reihenfolge* der Modifier-Aufrufe.

In vielen Fällen spielt die Reihenfolge, in der man verschiedene Modifier einsetzt, keine Rolle. Das gilt beispielsweise für die in Bild 2.8 eingesetzten Modifier. Man könnte auch erst `bold()` und `underline()` vor `font(_:)` aufrufen und das Endergebnis wäre noch immer dasselbe.

Aber das ist nicht in jedem Fall so. Ein entsprechendes Beispiel hierfür finden Sie in Listing 2.9. Darin erfolgt die Erstellung einer Text-Instanz, auf die zwei Modifier angewendet werden. Der erste – `padding()` – fügt einen Abstand zu allen Seiten des Textes ein. Der zweite – `border(_:width:)` – setzt im Anschluss einen grauen Rahmen um die View mit einer Stärke von 3 pt. Das Ergebnis zeigt Bild 2.9.

Listing 2.9 Erstellen eines Textes mit Rahmen

```
struct ContentView: View {
    var body: some View {
        Text("Hello SwiftUI!")
            .padding()
            .border(Color.gray, width: 3)
    }
}
```

Bild 2.9
Dem Text wurden ein Abstand sowie ein grauer Rahmen hinzugefügt.

Das gezeigte Ergebnis ist aber ein gänzlich anderes, wenn man die Reihenfolge der Modifier vertauscht und als Erstes `border(_:width:)` aufruft, so wie in Listing 2.10 und Bild 2.10 zu sehen.

Listing 2.10 Änderung der Reihenfolge der Modifier-Aufrufe

```
struct ContentView: View {
    var body: some View {
        Text("Hello SwiftUI!")
            .border(Color.gray, width: 3)
            .padding()
    }
}
```

Bild 2.10
Die Reihenfolge der Modifier-Aufrufe wirkt sich auf das Erscheinungsbild einer View aus.

Der Grund hierfür ist relativ simpel: Im zweiten Beispiel wird zuerst der Rahmen der `Text`-Instanz hinzugefügt. Da diese View keine Abstände zu den Rändern besitzt, „klebt" der Rahmen so direkt am anzuzeigenden Text. Erst dieser Ansicht werden im Anschluss die Abstände zu allen Seiten hinzugefügt. Diese Abstände sind zwar nicht direkt sichtbar, aber vorhanden.

Deutlicher wird das, wenn man nach dem Aufruf von `padding()` den `border(_:width:)`-Modifier ein zweites Mal setzt. So fügt man einen weiteren Rahmen hinzu, der jetzt allerdings auf der von `padding()` zurückgelieferten View basiert, also jener Ansicht, die die Abstände zu allen Seiten besitzt (siehe Listing 2.11 und Bild 2.11).

Listing 2.11 Ergänzung eines zweiten Rahmens

```
struct ContentView: View {
    var body: some View {
        Text("Hello SwiftUI!")
            .border(Color.gray, width: 3)
            .padding()
            .border(Color.black, width: 1)
    }
}
```

Bild 2.11
Der zweite Rahmen orientiert sich an der von padding() zurückgelieferten View mit den Abständen zu allen Seiten.

Dieses Beispiel demonstriert sehr schön die Funktionsweise der Modifier. Sie wenden ihre jeweiligen Änderungen immer auf die View an, über die sie aufgerufen werden, und liefern als Ergebnis eine entsprechend angepasste Ansicht zurück. Das muss man berücksichtigen, wenn man mehrere Modifier hintereinander aufruft, um die korrekte Reihenfolge der Aufrufe sicherzustellen.

2.2.3.2 Auszug verfügbarer Modifier

In Tabelle 2.1 finden Sie eine Zusammenstellung diverser Modifier, die Ihnen in SwiftUI zur Verfügung stehen und die insbesondere bei der Arbeit mit `Text`- und `Image`-Instanzen praktisch sind. In den kommenden Kapiteln stelle ich Ihnen noch weitere Modifier vor, die entweder bei der Arbeit mit bestimmten Views oder zur Umsetzung spezifischer Funktionen notwendig sind.

Tabelle 2.1 Auszug verfügbarer Modifier

Modifier	Beschreibung	Anwendbar auf
`bold()`	Formatiert einen Text fett.	`Text`-Instanzen
`italic()`	Setzt einen Text kursiv.	`Text`-Instanzen
`underline()`	Unterstreicht einen Text.	`Text`-Instanzen

Modifier	Beschreibung	Anwendbar auf
`foregroundColor(_:)`	Setzt eine Farbe für alle Vordergrundelemente einer View (Text, Bilder etc.). Er erwartet die gewünschte Farbe in Form eines `Color`-Parameters.	Alle Views
`font(_:)`	Formatiert einen Text auf Basis des übergebenen `Font`-Parameters.	`Text`-Instanzen
`background(_:)`	Fügt eine als Parameter übergebene View als Hintergrund ein.	Alle Views
`padding()`	Fügt Abstände an den Rändern zu einer View hinzu.	Alle Views
`border(_:)`	Fügt einer View einen Rahmen hinzu.	Alle Views
`cornerRadius(_:)`	Rundet den Rahmen einer View um den übergebenen Wert ab.	Alle Views
`opacity(_:)`	Legt die Transparenz einer View fest. Der Parameter `0` steht für volle Transparenz, `1` für keine Transparenz.	Alle Views
`hidden()`	Blendet eine View aus.	Alle Views
`resizable()`	Erlaubt das Verändern der Größe einer Grafik.	`Image`-Instanzen
`scaledToFill()`	Skaliert eine View, um den gegebenen Raum voll auszufüllen.	Alle Views
`scaledToFit()`	Skaliert eine View, um den gegebenen Raum bestmöglich zu nutzen (ohne die Ansicht zu verzerren).	Alle Views
`frame()`	Legt Größe und Positionierung einer View fest.	Alle Views

Modifier-Parameter

Viele Modifier bringen ein oder mehrere Parameter mit. So nutzt man beispielsweise beim Einsatz des `frame()`-Modifiers die Parameter `width` und `height`, um die gewünschte Breite beziehungsweise Höhe für eine View anzugeben:

```
.frame(width: 300, height: 100)
```

Ein Großteil solcher Parameter verfügen in SwiftUI aber bereits über Standardwerte. Das hat den Vorteil, dass man nur diejenigen angeben muss, die tatsächlich relevant sind. Möchte man so beispielsweise nur die Höhe einer View anpassen, übergibt man dem `frame()`-Modifier lediglich den `height`-Parameter mit dem gewünschten Wert:

```
.frame(height: 100)
```

In der Dokumentation von Xcode finden Sie eine vollständige Übersicht aller Parameter, die ein Modifier unterstützt (und natürlich lernen Sie auch in diesem Buch noch eine Vielzahl davon kennen ☺).

2.2.4 Einsatz von Library und Preview

SwiftUI-Views lassen sich nicht nur im Code erstellen und konfigurieren. Sie können ergänzend sowohl die Library von Xcode als auch die Preview nutzen, um Ihre Views anzupassen.

So finden Sie in der Library zwei entsprechende Reiter am äußersten linken Rand. Über diese können Sie zwischen der *Views* und der *Modifiers* Library wechseln (siehe Bild 2.12).

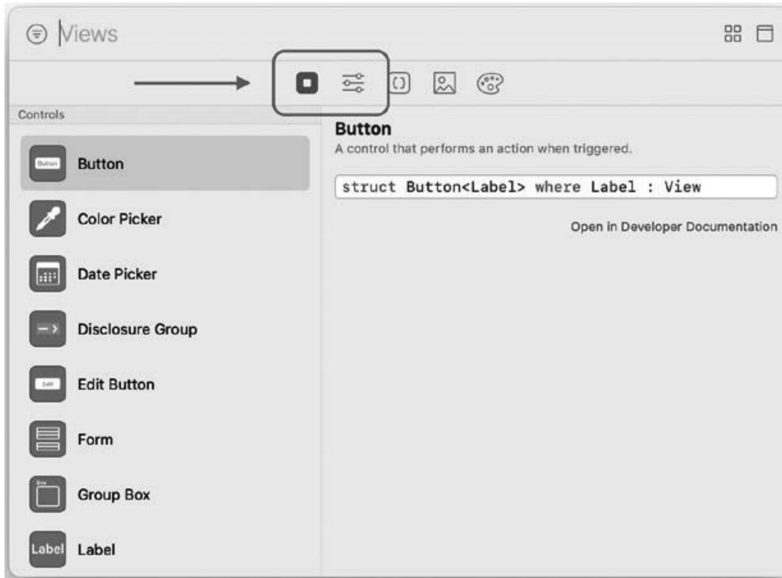

Bild 2.12 Die beiden neuen Reiter innerhalb der Xcode-Library erlauben Ihnen den Zugriff auf SwiftUI-Views und -Modifier.

Sichtbarkeit der Reiter

Die beiden genannten Reiter werden nur dann angezeigt, wenn Sie eine SwiftUI-Datei geöffnet haben und die zugehörige Preview anzeigen. Haben Sie die Preview deaktiviert, stehen Ihnen der *Views*- und der *Modifiers*-Reiter in der Library nicht zur Verfügung.

Die *Views Library* listet eine Vielzahl verschiedener Ansichten auf, die Sie mit SwiftUI verwenden können. Nach Auswahl eines Elements erhalten Sie am rechten Rand genauere Informationen dazu. Um eine View zu verwenden, ziehen Sie sie mittels Drag-and-drop aus der Library an die gewünschte Stelle im Code, an der sie zum Einsatz kommen soll. Xcode fügt die entsprechende View dann mitsamt einem Standard-Initializer ein. Abhängig von der gewählten View müssen Sie eventuell noch einige Informationen für die Initialisierung ergänzen. Im Falle eines `Image` müssen Sie beispielsweise den Namen für die anzuzeigende Grafik angeben.

Analog dazu verhält es sich mit der *Modifiers Library*. Sie finden darin eine Vielzahl verfügbarer Modifier aus SwiftUI, die Sie mittels Drag-and-drop einer View im Code zuweisen

können. Auch hier müssen Sie eventuell zusätzliche Informationen nach dem Einfügen des Modifiers ergänzen (siehe Bild 2.13).

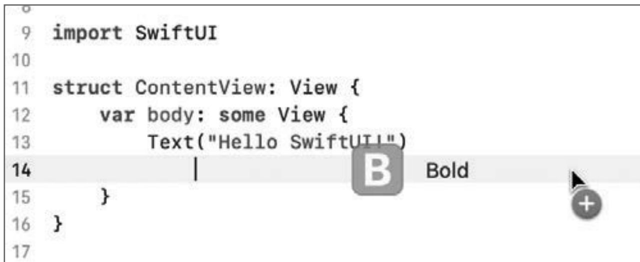

Bild 2.13 Views und Modifier lassen sich auch aus der Library heraus im Code ergänzen.

Sie können Views und Modifier aus der Library aber nicht nur mittels Drag-and-drop im Code hinzufügen. Alternativ können Sie diese Elemente auch in der Preview platzieren und so Ihre Views zusammenstellen (siehe Bild 2.14).

Diese Form der View-Erstellung erinnert stark an die Arbeit mit Storyboards in Xcode. Ein massiver Unterschied besteht aber darin, dass sich alle Änderungen an der Preview umgehend auf den Code auswirken. Ergänzen Sie so beispielsweise eine View oder einen Modifier in der Preview, aktualisiert Xcode den zugehörigen Code automatisch auf die passende Art und Weise.

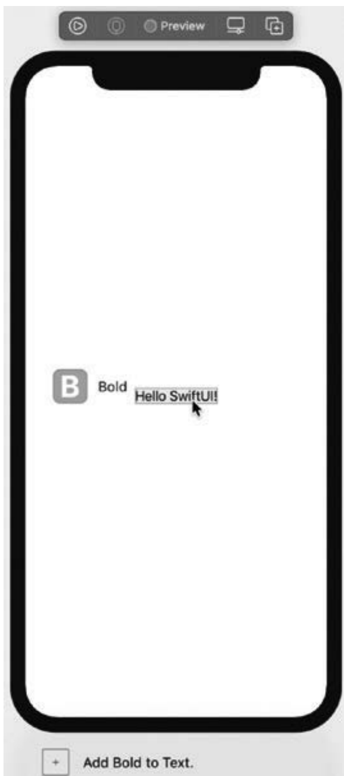

Bild 2.14
Views und Modifier können Sie aus der Library heraus auch in der Preview platzieren.

Der Grund hierfür ist, dass es in SwiftUI **keine Trennung** zwischen Code- und Interface-Dateien gibt. So ist die Preview nichts anderes als eine Vorschau des Quellcodes. Entsprechend führen Änderungen an der Preview auch zur Aktualisierung des Codes, da ausschließlich der für das Aussehen und die Funktionsweise von SwiftUI-Views verantwortlich ist.

Die Library als Lernhilfe

Gerade in Sachen Modifier hat SwiftUI eine ganze Menge Funktionen zu bieten. Zu Beginn kann es daher schwer fallen, die passenden Methoden zur Anpassung von Views zu finden.

Machen Sie sich an dieser Stelle die Suche der Library zunutze. Sie kann Ihnen helfen, den passenden Modifier für einen bestimmten Zweck zu finden. Außerdem kann es sich lohnen, einfach einmal durch die Liste der Modifier zu scrollen und sich so einen Überblick darüber zu verschaffen, welche Möglichkeiten zur Verfügung stehen. Dasselbe gilt auch für Views.

2.2.5 Layout-System

Das Layout-System in SwiftUI arbeitet anders, als man es von AppKit, UIKit und WatchKit sowie der Arbeit mit Storyboards und Auto Layout gewohnt ist. Bei der Zusammenstellung einer View kommen insgesamt drei Konzepte zum Tragen, die immer nacheinander ausgeführt werden.

1. Parent-View schlägt Größe für Child-View vor

Bei der Parent-View handelt es sich um jene View, in die man eine andere View einfügt. So ist beispielsweise ein Stack die Parent-View für alle Views, die als Teil des Stacks eingebunden werden.

Dieser Parent-View steht ein bestimmter Raum zur Verfügung. Im Falle der allerersten View, die man mittels SwiftUI in einer App anzeigt, fungiert die sogenannte *Root View* als Parent-View. Es handelt sich bei ihr um eine automatisch vom System erzeugte View, die den gesamten zur Verfügung stehenden Bildschirmplatz einnimmt.

Anhand dieser Parent-View steht fest, wie viel Platz den Child-Views zur Verfügung steht.

2. Child-View legt ihre Größe selbst fest

Jede Child-View bestimmt die Größe, die sie für die korrekte Darstellung benötigt, in SwiftUI selbst. Im Falle einer einfachen `Text`-Instanz entspricht diese Größe dem Raum, der benötigt wird, um den Text korrekt und vollständig anzuzeigen. Zur Bestimmung dieser Größe nutzt die Child-View auch den Raum, der ihr von ihrer Parent-View bereitgestellt wird.

Nutzt man einen Modifier wie `frame()` (siehe hierzu auch den Abschnitt 2.2.3.2, „Auszug verfügbarer Modifier“), erzeugt man eine neue View, die die übergebene Breite und Höhe besitzt. Diese Information nimmt die View dann als Grundlage für ihre Größe.

3. Parent-View platziert Child-View in ihrem Koordinatenraum

Zu guter Letzt positioniert die Parent-View ihre Child-View innerhalb ihres eigenen Koordinatenraums. Im Falle der Root-View bezieht sich dieser auf die zur Verfügung stehende Displaygröße. Die Positionierung der Child-View erfolgt in der Regel zentriert in der Mitte der Parent-View.

2.3 Status

Eine entscheidende Rolle bei der Umsetzung von SwiftUI-Views spielt der sogenannte *Status*. Er ist entscheidend für das Aussehen und die Funktionsweise einer View verantwortlich.

Um das Prinzip des Status in SwiftUI grundlegend zu verstehen, muss man sich zunächst einmal vor Augen führen, welche Aufgabe er übernimmt. Dazu möchte ich einen Vergleich zur Funktionsweise von Views aus dem AppKit-, UIKit- und WatchKit-Framework ziehen.

Mit AppKit, UIKit und WatchKit erstellte Views erzeugt man *imperativ*. Das bedeutet, dass eine View durch Aufruf von Befehlen erzeugt und angepasst wird. Möchte man beispielsweise einer View ein Label hinzufügen, nutzt man dafür unter UIKit eine Methode namens `addSubview(_:)`. Die Ausführung dieses Befehls sorgt für die Aktualisierung der View.

Beim Einsatz der imperativen Syntax steuern also Befehle das Aussehen einer Ansicht. Das ist auch bei Aktualisierungen von Views von entscheidender Bedeutung. Angenommen, Sie binden in Ihre App eine Fortschrittsanzeige für einen Download ein. Diese Anzeige muss regelmäßig aktualisiert werden, um den aktuellen Stand des Downloads zu signalisieren. Dazu nutzen Sie entweder eine passende Methode oder Sie setzen den Wert einer Property. Gleichzeitig müssen Sie nach erfolgreichem Download sicherstellen, dass der Fortschrittsbalken verschwindet und beispielsweise durch eine Erfolgsmeldung ersetzt wird. Auch hierfür rufen Sie dann an passender Stelle entsprechende Methoden auf, um Ihre Ansicht wie gewünscht zu aktualisieren.

In SwiftUI gibt es keinen derartigen imperativen Ansatz. Views werden nicht mittels Aufruf von Befehlen erzeugt, sondern mittels *deklarativer Syntax* zusammengestellt. Der Aufbau einer View ist so klar definiert.

Dennoch braucht es natürlich auch in SwiftUI eine gewisse Dynamik. Views müssen sich aktualisieren und dynamische Informationen anzeigen können. Und genau hier kommt der Status ins Spiel.

Der Status bestimmt, wie eine SwiftUI-View aussieht. Außerdem kann er für eine Aktualisierung der View sorgen, sollte sich der Status ändern.

An dieser Stelle möchte ich noch einmal das Beispiel des Fortschrittsbalkens eines Downloads anbringen. Um das in SwiftUI umzusetzen, bräuchte die View des Fortschrittsbalkens einen Status, der bestimmt, wie weit der Download fortgeschritten ist. Diese Information nutzt die View, um den Balken entsprechend zu füllen. Wann immer sich der Status ändert, aktualisiert sich automatisch die View und entsprechend wächst der Inhalt des Balkens. Ist der Download abgeschlossen, kann die View diese Information aus dem Status auslesen und so statt des Fortschrittsbalkens eine Erfolgsmeldung anzeigen.

Es gibt verschiedene Möglichkeiten, einen Status in einer SwiftUI-View umzusetzen. An dieser Stelle möchte ich Ihnen einen ersten Überblick über drei von ihnen geben: Property, State und Binding.

2.3.1 Property

Die einfachste Form, einen Status in einer SwiftUI-View einzubinden, stellt die Umsetzung als Property dar. Bei der Initialisierung einer View können Sie die gewünschte Information für diesen Status übergeben. Die View kann ihn dazu nutzen, ihr Aussehen und/oder ihren Aufbau entsprechend anzupassen.

Ein simples Beispiel zur Nutzung einer Property als View-Status finden Sie in Listing 2.12. Die darin deklarierte `TitleView` verfügt über eine Property namens `title` vom Typ `String`. Sie wird als Titel für die `Text`-Instanz verwendet. Da die Property keinen Standardwert besitzt, muss für sie ein passender Wert bei der Initialisierung von `ContentView` gesetzt werden.

Genau dieses Prozedere sorgt aber zugleich für Dynamik. Es lassen sich beliebige Instanzen von `TitleView` erzeugen, die alle einen anderen Titel darstellen können. In diesem Fall ist die `title`-Property der Status, die maßgeblich bestimmt, wie Instanzen auf Basis von `TitleView` aussehen (beziehungsweise welchen Inhalt sie anzeigen).

Listing 2.12 Einsatz einer Property als Status.

```
struct TitleView: View {
    var title: String

    var body: some View {
        Text(title)
            .font(.largeTitle)
    }
}

// Erstellen einer TitleView-Instanz
let myTitleView = TitleView(title: "Hello SwiftUI!")
```

Properties sind ideal, um Daten und Informationen an eine SwiftUI-View zu übergeben. Allerdings ist nur ein *lesender* Zugriff auf diese Properties möglich. Das macht es also unmöglich, den Wert einer Property innerhalb der zugehörigen SwiftUI-View zu verändern.

Im Falle der `TitleView` ist das kein Problem, da sie schlicht den ihr übergebenen Titel anzeigen soll. Auch ändert sich dieser Titel nicht zu einem späteren Zeitpunkt.

Soll ein Status jedoch veränderbar sein und zu einer entsprechenden Aktualisierung der zugehörigen View führen, müssen Sie auf andere Techniken zurückgreifen. Einen ersten Überblick dazu erhalten Sie in den folgenden Abschnitten, alle weiteren Details finden Sie dann in Kapitel 5 dieses Buches.

2.3.2 State

Bei `State` handelt es sich um einen Property Wrapper, der Teil des SwiftUI-Frameworks ist. Deklariert man damit eine Property einer View, ist es möglich, den Wert dieser Property aus der jeweiligen View heraus zu ändern (im Gegensatz zu einer „einfachen“ Property, siehe dazu auch den Abschnitt 2.3.1).

Dieses Konzept ist insofern spannend, als dass eine Änderung einer State-Property zu einer automatischen Aktualisierung der zugehörigen View führt. Das bedeutet, dass jene View dann auf Basis ihrer `body`-Property neu erzeugt wird.

Zum besseren Verständnis finden Sie in Listing 2.13 ein Beispiel dazu. Die darin deklarierte `ContentView` besteht aus einem `VStack`, der untereinander einen Text, eine Trennlinie (`Divider`) sowie eine Schaltfläche (`Button`) anzeigt. Mehr zu `Button` und `Divider` erfahren Sie in Kapitel 3 dieses Buches. Für dieses Beispiel müssen Sie nur über den grundlegenden Aufbau eines SwiftUI-Buttons Bescheid wissen. Dieser besteht aus zwei Closures. Das erste (`action`-Parameter) bestimmt, was bei Betätigen des Buttons geschieht. Das zweite Closure definiert das Aussehen des Buttons selbst. Er stellt in diesem Beispiel schlicht den Text „Update text presentation“ dar.

Das Besondere an `ContentView` ist die zu Beginn des Stacks erzeugte `Text`-Instanz. Diese soll nämlich entweder als großer Titel oder als herkömmlicher Text formatiert werden. Welche Formatierung zum Einsatz kommt, regelt die State-Property `formatAsLargeTitle` mit dem Standardwert `false`.

`formatAsLargeTitle` stellt somit einen Status der `ContentView` dar. Da die Property mit dem `State`-Property Wrapper deklariert wurde, ist es möglich, ihren Wert innerhalb von `ContentView` zu verändern. Und genau das geschieht im `action`-Parameter des Buttons. Bei Betätigen der Schaltfläche wird der aktuelle Wert von `formatAsLargeTitle` invertiert und wechselt so zwischen `true` und `false` hin und her.

Listing 2.13 Einsatz einer State-Property

```
struct ContentView: View {
    @State private var formatAsLargeTitle = false

    var body: some View {
        VStack {
            Text("Hello SwiftUI!")
                .font(formatAsLargeTitle ? .largeTitle : .body)
            Divider()
            Button(action: {
                self.formatAsLargeTitle.toggle()
            }) {
                Text("Update text presentation")
            }
        }
    }
}
```

An dieser Stelle kommen zwei Besonderheiten zum Tragen:

- Wäre die `formatAsLargeTitle`-Property nicht mittels `State` deklariert, ließe sich ihr Wert innerhalb von `ContentView` nicht aktualisieren. Der Befehl `self.formatAsLargeTitle.toggle()` innerhalb des action-Parameters des Buttons würde stattdessen einen Fehler zurückliefern.
- Die `State`-Deklaration von `formatAsLargeTitle` führt dazu, dass bei jeder Änderung des Werts dieser Property der `body` von `ContentView` neu erzeugt wird. Entsprechend erfolgt so auch eine erneute Erstellung der `Text`-Instanz mit der aktuell gültigen Formatierung.

Hier zeigt sich bereits sehr schön der deklarative Ansatz von SwiftUI. Allein das Ändern des Status (in diesem Fall der `formatAsLargeTitle`-Property) führt zu einem automatischen Update der View. Es braucht keinen ergänzenden Befehl, der das Aktualisieren des Textes explizit durchführt. Er ändert sich schlicht jedes Mal, wenn `formatAsLarge`-Title einen neuen Wert erhält. Auch dafür sorgt der `State`-Property Wrapper.

Mithilfe von `State` erzeugt man also einen veränderbaren Status. Zu beachten ist, dass State typischerweise nur für Properties eingesetzt werden sollte, die ausschließlich von der zugrunde liegenden View selbst gesteuert werden. Aus diesem Grund deklariert man `State`-Properties in der Regel auch als `private` und legt für sie einen Standardwert fest, so wie in Listing 2.13 zu sehen. Die View allein bestimmt also, welchen Wert der Status besitzt und wie er sich verändert.

Es gibt noch weitere Möglichkeiten, einen Status in einer SwiftUI-View umzusetzen. Alle weiteren Informationen zu `State` sowie zu alternativen Status-Deklarationen finden Sie in Kapitel 5 dieses Buches.

2.3.3 Binding

Properties fungieren als unveränderlicher Status, während `State`-Properties zwar änderbar sind, gleichzeitig aber fester Teil einer spezifischen View sind.

Manche View benötigt aber einen Status, der zwar ebenfalls änderbar sein soll, den eine View aber nicht *besitzt*. Die entsprechende Information ist an einer anderen Stelle gespeichert und sie soll lediglich – vergleichbar einer Referenz – an die View weitergegeben werden, damit sie mit diesen Daten arbeiten kann.

Ein Beispiel für eine solche View sehen Sie in Bild 2.15. Es zeigt eine View zum Erstellen von Bewertungen auf Basis von ein bis fünf Sternen. Jeder Stern stellt einen Button dar, der einer Bewertung entspricht. Wählt man so den zweiten Stern-Button, setzt man eine entsprechende Bewertung von zwei, im Falle des fünften Stern-Buttons eine Bewertung von fünf.

Die Bewertung ist der Status dieser View. Sie bestimmt, wie viele der fünf Sterne ausgefüllt sind und wie viele nicht. Das ließe sich auch mithilfe einer einfachen Property als Status umsetzen.

Bild 2.15
Die View dient zum Absetzen von Bewertungen.

Doch durch Betätigen eines Sternen-Buttons soll diese Bewertung auch *geändert* werden können. Eine einfache Property kommt so zum Abbilden des Status nicht länger infrage. Eine `State`-Property ist aber ebenso wenig eine passende Lösung. In diesem Fall würde die View nämlich die Bewertung selbst verwalten und nicht von außen ansprechbar machen. Genau das ist aber der Sinn bei der gezeigten View. Sie soll generisch für die unterschiedlichsten Inhalte nutzbar sein, um so für diese Inhalte Bewertungen abgeben zu können. Dabei spielt es keine Rolle, ob die View zum Bewerten von Filmen, Büchern oder Apps zum Einsatz kommt.

Die Bewertung wird *außerhalb* der View gespeichert. Die View soll lediglich einen *Verweis* darauf erhalten und diesen ändern können, ohne sich darum zu kümmern, aus welcher Quelle diese Information stammt. Und für genau solche Szenarien kommt der `Binding`-Property Wrapper zum Einsatz.

`Binding` ist mit `State` vergleichbar, nur mit dem Unterschied, dass einer `Binding`-Property ein Wert von außen übergeben werden muss. Man spricht in diesem Fall auch von einer sogenannten *Managed Reference*. Der Wert, auf den sich eine `Binding`-Property bezieht, ist an einer anderen Stelle gespeichert. Die `Binding`-Property verweist lediglich auf diesen Wert.

Besonders entscheidend hierbei: Führt man eine Änderung an einer `Binding`-Property durch, ändert sich so auch der Quellwert, der der `Binding`-Property übergeben wurde (dazu gleich mehr).

Entsprechend deklariert man eine `Binding`-Property – im Gegensatz zu `State` – niemals als `private`. Auch besitzen Binding-Properties nie einen Standardwert, da sie ihre Information immer von außen erhalten (in der Regel bei Initialisierung der View).

Die in Bild 2.15 gezeigte Bewertungs-View können Sie so wie in Listing 2.14 zu sehen umsetzen. Um die fünf Sternen-Buttons zu generieren, kommt eine View namens `ForEach`

zum Einsatz. Diese erzeugt im Rahmen einer vorgegebenen Range eine bestimmte View. In diesem Fall läuft die Range von der Zahl 1 bis zur Zahl 5; für jeden Wert wird ein passender Sternen-Button erzeugt (mehr zu `ForEach` erfahren Sie in Kapitel 3 dieses Buches).

Ob der jeweilige Button gefüllt ist oder nicht, bestimmt ein Abgleich des jeweiligen Werts mit der aktuellen Bewertung. Diese ist als Status mit dem Namen `rating` definiert und basiert auf dem `Binding`-Property Wrapper. Entsprechend besitzt `rating` keinen Standardwert und muss zwingend bei der Initialisierung von `RatingView` gesetzt werden.

Listing 2.14 Einsatz einer Binding-Property

```
struct RatingView: View {
    @Binding var rating: Int

    var body: some View {
        HStack {
            ForEach(1 ..< 6) { value in
                Button(action: {
                    self.rating = value
                }) {
                    Image(systemName: value <= self.rating ? "star.fill" : "star")
                        .font(.largeTitle)
                }
            }
        }
    }
}
```

Dank Einsatz des `Binding`-Property Wrappers lässt sich die `RatingView` vollkommen flexibel einsetzen. Sie lässt sich zum Bewerten der verschiedensten Inhalte verwenden. Die einzige Voraussetzung, die zum Einsatz der `RatingView` erfüllt sein muss, ist die Übergabe eines Integers, der die aktuelle Bewertung widerspiegelt und eine Änderung der Bewertung erlaubt. Letzteres geschieht bei Betätigen eines Sternen-Buttons (siehe hierzu auch die Implementierung des `action`-Parameters von `Button` in Listing 2.14). Der Wert der `rating`-Property wird hierbei auf den Wert des zugehörigen Buttons gesetzt. Der erste Button entspricht einer Bewertung von 1, der fünfte und letzte Button einer Bewertung von 5 (mehr zum Thema `Button` erfahren Sie in Kapitel 3 dieses Buches).

Um einen Wert für eine `Binding`-Property zu übergeben, muss man auch eine Instanz vom Typ `Binding` nutzen. Am einfachsten erhält man solch eine Instanz über eine `State`-Property. Ruft man diese mit einem vorangestellten $-Zeichen auf, erhält man statt des Werts der `State`-Property ein `Binding`, das auf jenen Wert verweist.

Weitere Binding-Möglichkeiten

Natürlich gibt es noch andere Wege, `Binding`-Instanzen auch ohne `State`-Properties zu erzeugen. Mehr dazu erfahren Sie in Kapitel 5 dieses Buches.

Ein einfaches Anwendungsbeispiel zur Nutzung der `RatingView` finden Sie in Listing 2.15. Die darin deklarierte `ContentView` verfügt über eine `State`-Property namens `rating`. Das ist also jene Stelle, an der die eigentliche Bewertung gespeichert werden soll.

ContentView selbst setzt sich aus zwei elementaren Bestandteilen zusammen. Zunächst ist da die RatingView, die als Binding einen Verweis auf die rating-Property von ContentView erhält. Um dieses Binding auszulesen, wird die rating-Property mit einem vorangestellten $-Zeichen beim Aufruf übergeben.

Darunter findet sich zusätzlich die Ausgabe eines Textes, der schlicht noch einmal den aktuellen Wert der rating-Property von ContentView darstellt. Bild 2.16 zeigt, wie ContentView innerhalb eines iOS-Simulators aussieht.

Listing 2.15 Nutzen einer Binding-Property

```
struct ContentView: View {
    @State private var rating = 0

    var body: some View {
        VStack {
            RatingView(rating: $rating)
                .padding()
            Divider()
            Text("Rating: \(rating)")
                .padding()
        }
    }
}
```

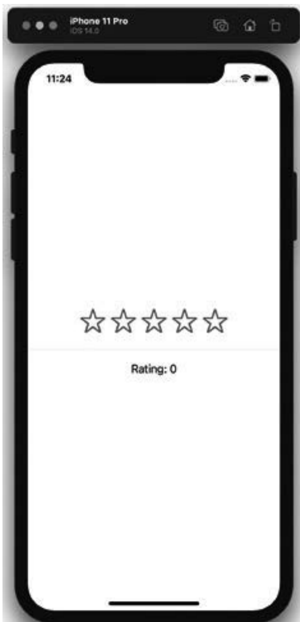

Bild 2.16
Die Startkonfiguration der ContentView besitzt noch keine abgegebene Bewertung.

Da RatingView die rating-Property von ContentView als Binding übergeben wurde, führt eine Änderung jenes Bindings innerhalb von RatingView automatisch auch zu einer Aktualisierung des ursprünglichen Werts, sprich der State-Property rating von ContentView. Betätigt man also eine der Sternen-Schaltflächen, aktualisiert sich nicht nur die RatingView entsprechend und füllt die Sterne passend aus. Durch das Binding erfolgt auch eine pas-

sende Aktualisierung der `rating`-Property in `ContentView`, sodass die Textausgabe ebenfalls die aktuelle Bewertung widerspiegelt (siehe Bild 2.17).

Viele Views innerhalb des SwiftUI-Frameworks nutzen Binding auf die dargestellte Art und Weise. `TextField` beispielsweise nimmt einen String als Binding entgegen. Diesen stellt die View dar und ändert ihn, sobald der Nutzer eine Aktualisierung des Textes vornimmt. Dank Binding lässt sich so ein String aus jeder beliebigen Quelle mit `TextField` nutzen.

Bild 2.17 Mittels Binding basieren sowohl die RatingView als auch die Textausgabe auf derselben Information.

Gleiches gilt für `Toggle`. Mit dieser View erstellt man einfache Schalter, die nur zwei Zustände kennen: an oder aus. Abgebildet wird dieser Zustand mithilfe eines Boolean, den `Toggle` ebenfalls in Form eines Bindings erhält (mehr zu diesen und weiteren View-Elementen erfahren Sie in Kapitel 3 dieses Buches).

Weitere Informationen zum Status

Der erste Überblick, den Sie bezüglich des Status von SwiftUI-Views in diesem Kapitel erhalten haben, dient als Grundlage, damit Sie die kommenden Inhalte nachvollziehen können. Das Thema „Status" ist in SwiftUI aber noch deutlich umfangreicher und komplexer. Darum widmet sich das fünfte Kapitel dieses Buches – „Status" – noch einmal ausführlich diesem Thema. Sie finden dort nicht nur eine tiefergehende Betrachtung der bereits vorgestellten Elemente Property, `State` und `Binding`, sondern auch Informationen zu weiteren Möglichkeiten, wie Sie einen Status in SwiftUI-Views umsetzen und worauf dabei zu achten ist.

Index

A

ActionSheet 174
– Bool 177
– erstellen 175
– Identifiable-Item 177
ActionSheet.Button 175
Alert 169
– Bool 171
– erstellen 169
– Identifiable-Item 172
Alert.Button 170
AppKit 1
Asset Catalog 47
Axis.Set 126

B

Binding 35, 183
body 11
Bundle 47
Button 53
– Style 56

C

Child-View 30
Container-View 16, 128
– View-Maximum 131
Coordinator 216

D

DatePicker 70
– Auswahl einschränken 71
– Style 72
DatePickerComponents 70
Debugging 233
Deklarative Syntax 31
Divider 142
Drag-and-drop 119

E

EditButton 57
Environment 200
EnvironmentObject 194
– Besonderheiten 198
– zuweisen 198
EnvironmentValues 201

F

Font 40
ForEach 112
– Datenmodell 115
– mit List 117
Form 128
Frameworks
– AppKit 1
– MobileCoreServices 120
– UIKit 1
– WatchKit 1

G

Grids 99
– Abstände 101
– LazyHGrid 99
– LazyVGrid 99
GridItem 99
– Konfiguration 101
GridItem.Size 101
Group 130
GroupBox 134
– Label 135

H

HorizontalAlignment 93
Hosting 205
HSplitView 164
HStack 16, 88
– Abstände 91
– Ausrichtung 89

I

Identifiable 107, 108
Image 15, 47
– Größe ändern 49
Integration 205
– Hosting 205
– Representable 208

K

KeyPath 108
Klasse 14
Kontext-Aktionen 239

L

Label 63, 85
Layout-System 30
LazyHGrid 99
– Abstände 101
LazyHStack 98
LazyVGrid 99
– Abstände 101
LazyVStack 98
Library 28, 234
– Einsatzzweck 235
– Modifier ergänzen 236
– Modifiers Library 28
– View ergänzen 236
– Views Library 28
LibraryContentBuilder 239
LibraryContentProvider 236
Link 86
List 103
– Datenmodell 106
– dynamisch 105
– statisch und dynamisch 124
– Style 111
– Zelle hinzufügen 119
– Zelle löschen 118
– Zellen verschieben 122
– Zellenauswahl 109
ListStyle 111

M

Managed Reference 35
Menu 58
– Style 59
MobileCoreServices 120
Modifier 19
– Funktionsweise 22
– auf mehrere Views anwenden 21
– Reihenfolge 23
– verfügbarer Auszug 26
Modifiers Library 28

N

Navigation-Bar-Item 153
NavigationLink 143
– dynamischer Status 156
– einfacher Status 155
NavigationView 143
– Display-Mode 151
– Grundlagen 143
– Navigation-Bar ausblenden 152
– Navigation-Bar-Items 153
– Standardansicht 146
– Style 148
– Titel 150
– watchOS 157
NSHostingController 206
NSItemProvider 121

NSViewControllerRepresentable 209
NSViewRepresentable 209

O

objectWillChange 191
ObservableObject 187
ObservedObject 186
– Änderungen 190
– Datenmodell einbinden 187
– Datenmodell erstellen 187
– objectWillChange 191
– Published 190
Opaque Type 13

P

Parent-View 30
Pasteboard 57
PasteButton 57
Picker 66
– Label ausblenden 68
– Style 68
Preview 5, 28, 221
– Ausführung 231
– Debugging 233
– Device 233
– Einsatz 224
– Funktionsweise 222
– Konfiguration 227
PreviewProvider 222
– Name 222
Primitive View 14
ProgressView 83
– Style 84
Property 32, 180
Protokolle
– Identifiable 108
– LibraryContentProvider 236
– ObservableObject 187
– PreviewProvider 222
– View 11
Published 190

R

Refactoring 239
Representable 208
– aktualisieren 213
– Coordinator 216
– erstellen 210
– Make-Methode 209
– Update-Methode 209
– View-Update 215
Root View 30

S

ScrollView 125
– Scroll-Indicator 128
– Scroll-Richtung 126
Section 136
– Fußzeile 137
– Kopfzeile 137
SecureField 45
SF Symbols 15, 48
Sheet 165
– ausblenden 168
– Boolean 165
– Identifiable-Item 166
Shorthand Getter Declaration 12
SignInWithAppleButton 60
Slider 74
– Wertebereich 74
some 14
Spacer 139
– Mindestabstand 140
Stack 16, 87
– HStack 16, 88
– LazyHStack 98
– LazyVStack 98
– VStack 16, 92
– ZStack 16, 96
State 33, 182
StateObject 193
Status 31, 179
– Binding 35, 183
– Environment 200
– EnvironmentObject 194
– ObservedObject 186
– Property 32, 180
– State 33, 182

– StateObject 193
– Übersicht 202
Stepper 78
– Aktionen 82
– eigene View 80
– Schritte 80
Storyboard 1
Structure 14
SwiftUI 1
– Grundlagen 11
– View-Aufbau 14
– Voraussetzungen 8

T

TabView 158
– Grundlagen 159
– programmatischer View-Wechsel 162
– Tab-Bar-Items 160
Text 15, 39
– Übersetzung 41
TextEditor 46
TextField 42
TextFieldStyle 44
Toggle 61
– Label ausblenden 63
– Style 64
– tvOS 65

U

UIHostingController 206
UIKit 1
UINavigationController 143
UIViewControllerRepresentable 209
UIViewRepresentable 209
Uniform Type Identifier 58, 120
UTI 58, 120
UTType 120
UUID 107

V

Value Indicators 83
Value Selector 61
VerticalAlignment 89
View 11
– body 11
– Button 53
– DatePicker 70
– deklarativ erzeugte 31
– Divider 142
– EditButton 57
– ForEach 112
– Form 128
– Group 130
– GroupBox 134
– HSplitView 164
– HStack 88
– Image 47
– imperativ erzeugte 31
– Label 85
– LazyHGrid 99
– LazyHStack 98
– LazyVGrid 99
– LazyVStack 98
– Link 86
– List 103
– Menu 58
– NavigationLink 143
– NavigationView 143
– PasteButton 57
– Picker 66
– ProgressView 83
– ScrollView 125
– Section 136
– SecureField 45
– SignInWithAppleButton 60
– Slider 74
– Spacer 139
– Stepper 78
– TabView 158
– Text 39
– TextEditor 46
– TextField 42
– Toggle 61
– VSplitView 164
– VStack 92
– ZStack 96
View-Builder 17
View-Maximum 131
Views Library 7, 28
Voraussetzungen 8
– iOS 8
– iPadOS 8
– macOS 8
– tvOS 8
– watchOS 8

VSplitView 164
VStack 16, 92
– Abstände 95
– Ausrichtung 93

W

WatchKit 1
WKHostingController 207
WKInterfaceObjectRepresentable 209
Worldwide Developers Conference 1
WWDC 1

X

Xcode 221
– Kontext-Aktionen 239
– Library 28, 234

Z

ZStack 16, 96
– Ausrichtung 96