

HANSER



Leseprobe

zu

Das Swift-Handbuch

von Thomas Sillmann

Print-ISBN: 978-3-446-46782-8

E-Book-ISBN: 978-3-446-46963-1

E-Pub-ISBN: 978-3-446-47286-0

Weitere Informationen und Bestellungen unter

<https://www.hanser-kundencenter.de/fachbuch/artikel/9783446467828>

sowie im Buchhandel

© Carl Hanser Verlag, München

Inhalt

Vorwort	XIX
Teil I: Swift	1
1 Die Programmiersprache Swift	3
1.1 Die Geschichte von Swift	3
1.2 Die Bedeutung von Swift im Apple-Kosmos	5
1.3 Das neue UI-Framework: SwiftUI	5
1.4 Was Sie als App-Entwickler brauchen	6
1.5 Programmieren für Beginner (und darüber hinaus): Playgrounds	8
1.6 Weitere wichtige Ressourcen	10
1.6.1 Apple-Developer-App	10
1.6.2 Apples Developer-Website	11
1.6.3 Swift.org	12
1.6.4 In eigener Sache	13
2 Grundlagen der Programmierung	15
2.1 Grundlegendes	15
2.1.1 Swift Standard Library	15
2.1.2 print	17
2.1.3 Befehle und Semikolons	18
2.1.4 Operatoren	19
2.2 Variablen und Konstanten	20
2.2.1 Erstellen von Variablen und Konstanten	20
2.2.2 Variablen und Konstanten in der Konsole ausgeben	21
2.2.3 Type Annotation und Type Inference	22
2.2.4 Gleichzeitiges Erstellen und Deklarieren mehrerer Variablen und Konstanten	24
2.2.5 Namensrichtlinien	24
2.3 Kommentare	25

3	Schleifen und Abfragen	27
3.1	Schleifen	27
3.1.1	For-In	27
3.1.2	While	29
3.1.3	Repeat-While	30
3.2	Abfragen	31
3.2.1	If	31
3.2.2	Switch	36
3.2.3	Guard	39
3.3	Control Transfer Statements	41
3.3.1	Anstoßen eines neuen Schleifendurchlaufs mit continue	41
3.3.2	Verlassen der kompletten Schleife mit break	42
3.3.3	Labeled Statements	43
4	Typen in Swift	45
4.1	Integer	46
4.2	Fließkommazahlen	48
4.3	Bool	49
4.4	String	49
4.4.1	Erstellen eines Strings	49
4.4.2	Zusammenfügen von Strings	50
4.4.3	Character auslesen	51
4.4.4	Character mittels Index auslesen	52
4.4.5	Character entfernen und hinzufügen	53
4.4.6	Anzahl der Character zählen	55
4.4.7	Präfix und Suffix prüfen	55
4.4.8	String Interpolation	55
4.5	Array	56
4.5.1	Erstellen eines Arrays	57
4.5.2	Zusammenfügen von Arrays	58
4.5.3	Inhalte eines Arrays leeren	58
4.5.4	Prüfen, ob ein Array leer ist	59
4.5.5	Anzahl der Elemente eines Arrays zählen	59
4.5.6	Zugriff auf die Elemente eines Arrays	60
4.5.7	Neue Elemente zu einem Array hinzufügen	60
4.5.8	Bestehende Elemente aus einem Array entfernen	61
4.5.9	Bestehende Elemente eines Arrays ersetzen	62
4.5.10	Alle Elemente eines Arrays auslesen und durchlaufen	63
4.6	Set	64
4.6.1	Erstellen eines Sets	64
4.6.2	Inhalte eines bestehenden Sets leeren	65
4.6.3	Prüfen, ob ein Set leer ist	66
4.6.4	Anzahl der Elemente eines Sets zählen	66

4.6.5	Element zu einem Set hinzufügen	66
4.6.6	Element aus einem Set entfernen	67
4.6.7	Prüfen, ob ein bestimmtes Element in einem Set vorhanden ist	67
4.6.8	Alle Elemente eines Sets auslesen und durchlaufen	67
4.6.9	Sets miteinander vergleichen	68
4.6.10	Neue Sets aus bestehenden Sets erstellen	71
4.7	Dictionary	72
4.7.1	Erstellen eines Dictionaries	73
4.7.2	Prüfen, ob ein Dictionary leer ist	74
4.7.3	Anzahl der Schlüssel-Wert-Paare eines Dictionaries zählen	74
4.7.4	Wert zu einem Schlüssel eines Dictionaries auslesen	74
4.7.5	Neues Schlüssel-Wert-Paar zu Dictionary hinzufügen	75
4.7.6	Bestehendes Schlüssel-Wert-Paar aus Dictionary entfernen	76
4.7.7	Bestehendes Schlüssel-Wert-Paar aus Dictionary verändern	76
4.7.8	Alle Schlüssel-Wert-Paare eines Dictionaries auslesen und durchlaufen	77
4.8	Tuple	78
4.8.1	Zugriff auf die einzelnen Elemente eines Tuples	79
4.8.2	Tuple und switch	80
4.9	Optional	83
4.9.1	Deklaration eines Optionals	84
4.9.2	Zugriff auf den Wert eines Optionals	84
4.9.3	Optional Binding	86
4.9.4	Implicitly Unwrapped Optional	88
4.9.5	Optional Chaining	89
4.9.6	Optional Chaining über mehrere Eigenschaften und Funktionen	93
4.10	Any und AnyObject	97
4.11	Type Alias	98
4.12	Value Type versus Reference Type	98
4.12.1	Reference Types auf Gleichheit prüfen	100
5	Funktionen	103
5.1	Funktionen mit Parametern	104
5.1.1	Argument Labels und Parameter Names	105
5.1.2	Default Value für Parameter	108
5.1.3	Variadic Parameter	109
5.1.4	In-Out-Parameter	110
5.2	Funktionen mit Rückgabewert	111
5.3	Function Types	113
5.3.1	Funktionen als Variablen und Konstanten	115
5.4	Verschachtelte Funktionen	116
5.5	Closures	117

5.5.1	Closures als Parameter von Funktionen	118
5.5.2	Trailing Closures	121
5.5.3	Autoclosures	122
6	Enumerations, Structures und Classes	125
6.1	Enumerations	125
6.1.1	Enumerations und switch	128
6.1.2	Associated Values	129
6.1.3	Raw Values	131
6.2	Structures	134
6.2.1	Erstellen von Structures und Instanzen	134
6.2.2	Eigenschaften und Funktionen	135
6.3	Classes	140
6.3.1	Erstellen von Klassen und Instanzen	141
6.3.2	Eigenschaften und Funktionen	141
6.4	Enumeration vs. Structure vs. Class	143
6.4.1	Gemeinsamkeiten und Unterschiede	143
6.4.2	Wann nimmt man was?	144
6.5	self	146
7	Eigenschaften und Funktionen von Typen	149
7.1	Properties	149
7.1.1	Stored Property	150
7.1.2	Lazy Stored Property	152
7.1.3	Computed Property	155
7.1.4	Read-Only Computed Property	159
7.1.5	Property Observer	160
7.1.6	Property Wrapper	163
7.1.7	Type Property	170
7.2	Globale und lokale Variablen	172
7.3	Methoden	175
7.3.1	Instance Methods	175
7.3.2	Type Methods	178
7.4	Subscripts	179
8	Initialisierung	185
8.1	Aufgabe der Initialisierung	186
8.2	Erstellen eigener Initializer	187
8.3	Initializer Delegation	193
8.3.1	Initializer Delegation bei Value Types	193
8.3.2	Initializer Delegation bei Reference Types	194
8.4	Failable Initializer	197

8.5	Required Initializer	200
8.6	Deinitialisierung	200
9	Vererbung	203
9.1	Überschreiben von Eigenschaften und Funktionen einer Klasse	206
9.2	Überschreiben von Eigenschaften und Funktionen einer Klasse verhindern ..	209
9.3	Zugriff auf die Superklasse	209
9.4	Initialisierung und Vererbung	210
9.4.1	Zwei-Phasen-Initialisierung	211
9.4.2	Überschreiben von Initializern	218
9.4.3	Vererbung von Initializern	220
9.4.4	Required Initializer	221
10	Speicherverwaltung mit ARC	223
10.1	Strong Reference Cycles	226
10.1.1	Weak References	228
10.1.2	Unowned References	232
10.1.3	Weak Reference vs. Unowned Reference	234
11	Weiterführende Sprachmerkmale von Swift	235
11.1	Nested Types	235
11.2	Extensions	237
11.2.1	Computed Properties	237
11.2.2	Methoden	238
11.2.3	Initializer	239
11.2.4	Subscripts	241
11.2.5	Nested Types	242
11.3	Protokolle	242
11.3.1	Deklaration von Eigenschaften und Funktionen	244
11.3.2	Der Typ eines Protokolls	254
11.3.3	Protokolle und Extensions	256
11.3.4	Vererbung in Protokollen	260
11.3.5	Class-only-Protokolle	261
11.3.6	Optionale Eigenschaften und Funktionen	263
11.3.7	Protocol Composition	265
11.3.8	Delegation	267
11.3.9	Übersicht diverser vorhandener Protokolle	269
11.4	Key-Path	271
12	Type Checking und Type Casting	275
12.1	Type Checking mit „is“	278
12.2	Type Casting mit „as“	279

13	Error Handling	281
13.1	Deklaration und Feuern eines Fehlers	281
13.2	Reaktion auf einen Fehler	285
13.2.1	Mögliche Fehler mittels do-catch auswerten	285
13.2.2	Mögliche Fehler in Optionals umwandeln	289
13.2.3	Mögliche Fehler weitergeben	289
13.2.4	Mögliche Fehler ignorieren	291
14	Generics	293
14.1	Generic Functions	294
14.2	Generic Types	297
14.3	Type Constraints	300
14.4	Associated Types	300
15	Nebenläufigkeit	305
15.1	Asynchronen Code schreiben und aufrufen	305
15.2	Mehrere asynchrone Funktionen parallel ausführen	308
15.3	Actors	310
16	Dateien und Interfaces	313
16.1	Modules und Source Files	313
16.2	Access Control	314
16.2.1	Access Level	314
16.2.2	Explizite und implizite Zuweisung eines Access Levels	318
16.2.3	Besonderheiten	319
Teil II: Xcode		325
17	Grundlagen, Aufbau und Einstellungen von Xcode	327
17.1	Über Xcode	328
17.2	Arbeiten mit Xcode	329
17.2.1	Dateien und Formate eines Xcode-Projekts	329
17.2.2	Umgang mit Dateien und Ordnern	334
17.3	Der Aufbau von Xcode	339
17.3.1	Toolbar	339
17.3.2	Navigator	342
17.3.3	Editor	346
17.3.4	Inspectors	350
17.3.5	Debug Area	353
17.4	Einstellungen	354
17.4.1	General	354
17.4.2	Accounts	355

17.4.3	Behaviors	356
17.4.4	Navigation	356
17.4.5	Themes	357
17.4.6	Text Editing	358
17.4.7	Key Bindings	358
17.4.8	Source Control	359
17.4.9	Components	360
17.4.10	Locations	360
17.4.11	Server & Bots	361
17.5	Projekteinstellungen	362
17.5.1	Einstellungen am Projekt	363
17.5.2	Einstellungen am Target	365
17.5.3	Einstellungen am Scheme	373
18	Dokumentation, Devices und Organizer	379
18.1	Dokumentation	379
18.1.1	Aufbau und Funktionsweise	380
18.1.2	Direktzugriff im Editor	383
18.2	Devices und Simulatoren	385
18.2.1	Simulatoren	387
18.2.2	Devices	389
18.3	Organizer	391
19	Debugging und Refactoring	393
19.1	Debugging	393
19.1.1	Konsolenausgaben	394
19.1.2	Arbeiten mit Breakpoints	395
19.1.3	Debug Navigator	400
19.2	Refactoring	402
19.3	Instruments	404
20	Tipps und Tricks für das effiziente Arbeiten mit Xcode	409
20.1	Code Snippets	409
20.2	Open Quickly	411
20.3	Related Items	412
20.4	Navigation über die Jump Bar	413
20.5	MARK, TODO und FIXME	413
20.6	Shortcuts für den Navigator	415
20.7	Clean Build	415
20.8	Playgrounds	415

Teil III: App-Entwicklung	419
21 Grundlagen der App-Entwicklung	421
21.1 Die Basis: SwiftUI	421
21.2 Bestandteile einer App	423
21.2.1 Umsetzung der Daten	424
21.2.2 Umsetzung der Ansichten	424
21.2.3 Weitere Frameworks	424
21.3 Die Syntax von SwiftUI	424
21.4 Aufbau einer App	425
21.5 Das View-Protokoll	426
21.6 Aktualisierung von Views mittels Status	427
21.7 Grundlagen des Status	429
21.8 Anpassung von Views mittels Modifier	432
21.9 Gruppierung von Views mittels Containern	435
21.10 Praxis: Unsere erste App	436
21.10.1 Bestandteile des neuen Projekts	442
21.10.2 Änderung des Textes	444
21.10.3 Einsatz der Preview	445
22 Views in SwiftUI	449
22.1 Textdarstellung und -bearbeitung	449
22.1.1 Text	449
22.1.2 TextField	459
22.1.3 SecureField	462
22.1.4 TextEditor	463
22.2 Bilder	466
22.2.1 Image-Instanz erstellen	468
22.2.2 Größe einer Image-Instanz ändern	470
22.3 Schaltflächen	472
22.3.1 Button	473
22.3.2 EditButton	476
22.3.3 PasteButton	477
22.4 Wertauswahl	479
22.4.1 Toggle	479
22.4.2 Slider	484
22.4.3 Stepper	490
22.4.4 Picker	495
22.4.5 DatePicker	499
22.4.6 ColorPicker	503
22.5 Werteindikatoren	506
22.5.1 Label	506
22.5.2 ProgressView	509
22.5.3 Gauge	512

23	View-Layout	517
23.1	Stacks	517
23.1.1	HStack	517
23.1.2	VStack	521
23.1.3	ZStack	525
23.1.4	Stacks verschachteln	526
23.1.5	Lazy Stacks	527
23.2	Listen	531
23.2.1	List	531
23.2.2	ForEach	552
23.3	Grids	563
23.4	Container-Views	578
23.4.1	Form	579
23.4.2	Section	581
23.4.3	Group	584
23.4.4	GroupBox	589
23.5	Weitere Views	592
23.5.1	ScrollView	592
23.5.2	OutlineGroup	596
23.5.3	DisclosureGroup	601
23.5.4	Spacer	606
23.5.5	Divider	609
24	Navigation	611
24.1	NavigationView und NavigationLink	611
24.1.1	Festlegen einer Standard-View für die Detailansicht	618
24.1.2	NavigationView-Style anpassen	620
24.1.3	NavigationView-Titel setzen	623
24.1.4	Navigation-Bar ausblenden	626
24.1.5	Eigene View zur Darstellung eines NavigationLink nutzen	628
24.1.6	NavigationLink programmatisch ausführen	630
24.1.7	Verhalten eines NavigationLink unter iPadOS anpassen	636
24.2	TabView	640
24.3	HSplitView und VSplitView	648
24.4	Funktionen zur Präsentation von Views	648
24.4.1	Sheet einblenden	649
24.4.2	View über gesamtes Fenster legen	655
24.4.3	Popover einblenden	660
25	Weitere View-Konfigurationen	667
25.1	Toolbar	667
25.2	Alerts	675

25.3	Confirmation Dialog	678
25.4	Farben	680
25.5	View-Events	681
26	Status	683
26.1	Property	685
26.2	State	687
26.3	Binding	688
26.4	ObservedObject	697
26.4.1	Datenmodell vorbereiten	698
26.4.2	Datenmodell in View einbinden	699
26.4.3	Auf Änderungen reagieren	702
26.5	StateObject	706
26.6	EnvironmentObject	707
26.7	Environment	713
26.8	SceneStorage	717
26.9	AppStorage	719
26.10	Source of Truth vs. Derived Value	721
26.11	Best Practices	724
27	Datenhaltung	729
27.1	UserDefaults	729
27.1.1	UserDefaults und SwiftUI	731
27.2	Core Data	732
27.2.1	Grundlegende Funktionsweise von Core Data	732
27.2.2	Grundlegende Elemente beim Einsatz von Core Data	733
27.2.3	Einen Core Data Stack erstellen	735
27.2.4	Ein Managed Object Model erstellen	737
27.2.5	Grundlegende Core-Data-Operationen	747
27.2.6	Core Data mit SwiftUI	749
28	Weitere Projektkonfigurationen	757
28.1	Cross-Platform-Entwicklung	757
28.1.1	Neue Targets hinzufügen	757
28.1.2	Target-Zuweisung	760
28.1.3	Plattform im Code prüfen	761
28.2	Mehrsprachigkeit	762
28.2.1	Grundlagen	762
28.2.2	Übersetzungen mit SwiftUI	767
28.2.3	Verschiedene Sprachen einer App testen	768
28.3	Asset Catalogs	769

29	Preview und Library	773
29.1	Preview	773
29.1.1	Funktionsweise der Preview	775
29.1.2	Konfiguration der Preview	777
29.1.3	Preview ausführen	778
29.2	Library	779
29.3	Attributes Inspector	781
Teil IV: Source Control und Testing		783
30	Source Control	785
30.1	Basisfunktionen und -begriffe der Source Control	785
30.2	Source Control in Xcode	787
30.2.1	Bestehendes Projekt klonen	788
30.2.2	Lokale Änderungen committen	790
30.2.3	Lokale Änderungen verwerfen	791
30.2.4	Pull und Push	792
30.2.5	Aktuelle Branches vom Repository laden	793
30.2.6	Git-Repository mit neuem Xcode-Projekt erzeugen	793
30.2.7	Optische Source-Control-Hervorhebungen im Editor	794
30.2.8	Zugriff auf GitHub, GitLab und Bitbucket	795
30.3	Source Control Navigator	796
30.4	Code Review-Mode	797
31	Testing	801
31.1	Unit-Tests	801
31.1.1	Aufbau und Funktionsweise von Unit-Tests	805
31.1.2	Aufbau einer Test-Case-Klasse	808
31.1.3	Neue Test-Case-Klasse erstellen	810
31.1.4	Ausführen von Unit-Tests	811
31.1.5	Was sollte ich eigentlich testen?	813
31.2	Performancetests	814
31.3	UI-Tests	815
31.3.1	Klassen für UI-Tests	817
31.3.2	Aufbau von UI-Test-Klassen	819
31.3.3	Automatisches Erstellen von UI-Tests	819
31.3.4	Einsatz von UI-Tests	820

Teil V: Veröffentlichung von Apps	821
32 Veröffentlichung im App Store	823
32.1 Das Apple Developer Portal	824
32.1.1 Zertifikate, App IDs und Provisioning Profiles	827
32.1.2 Code Signing	842
32.2 App Store Connect	845
32.2.1 Apps für den App Store vorbereiten und verwalten	846
32.2.2 Apps erstellen, hochladen und einreichen	850
32.3 App Store Review Guidelines	852
33 Das Business Model für Ihre App	855
33.1 Geschäftsmodelle	855
33.1.1 Free Model	855
33.1.2 Freemium Model	856
33.1.3 Subscription Model	856
33.1.4 Paid Model	857
33.1.5 Paymium Model	857
33.2 App Bundles	858
33.3 Veröffentlichung außerhalb des App Store	859
33.3.1 Das Apple Developer Enterprise Program	860
34 TestFlight	863
34.1 TestFlight in App Store Connect	863
34.2 TestFlight im App Store	865
Index	867

Vorwort

Herzlich Willkommen in der Welt von Swift, Apples hauseigener Programmiersprache zur Entwicklung von Apps für iPhone, iPad, Mac und Co.! Da Sie dieses Buch in Händen halten, mutmaße ich, dass Sie mehr über die Programmierung für Apple-Plattformen erfahren oder Ihre ersten Schritte in dieser faszinierenden Welt bestreiten möchten.

Bevor ich Sie in die Welt der Programmierung entführe, möchte ich dieses Vorwort dazu nutzen, einige grundlegende Worte über den Aufbau und die Inhalte dieses Buches zu verlieren. Das soll Ihnen als erste Übersicht und weiterer Wegweiser dienen, um bestmöglich mit dem Buch arbeiten zu können und schnelle Erfolge bei der Programmierung zu erzielen.

Inhalte

Das Buch basiert auf der im Herbst 2021 erschienen Version 5.5 von Swift. Alle Kapitel der ersten Auflage wurden entsprechend aktualisiert sowie um neue Inhalte ergänzt. Ein Highlight stellt hierbei sicherlich ein neues Kapitel zur Nebenläufigkeit dar, das den Einsatz der neuen Schlüsselwörter `async` und `await` erläutert.

Auch basieren alle Inhalte des Buches vollständig auf der ebenfalls im Herbst 2021 erschienenen Version 13 von Apples Entwicklungsumgebung Xcode und sie wurden entsprechend aktualisiert. Das betrifft auch die Bereiche Source Control und Testing. Zu guter Letzt finden Sie im Buch aktuelle Infos zur Einreichung von Apps im App Store, mögliche App-Geschäftsmodelle sowie den Einsatz von TestFlight.

Mit dieser zweiten Auflage des Swift-Handbuchs gibt es jedoch eine große Änderung im Vergleich zum ursprünglichen Werk. Das betrifft die Kapitel zur App-Entwicklung für macOS und Co. auf Basis von AppKit, UIKit und WatchKit. Diese wurden durch gänzlich neue Inhalte auf Basis des SwiftUI-Frameworks ersetzt.

Dieser radikal anmutende Schritt hat drei Gründe:

- SwiftUI stellt die Zukunft der App-Entwicklung für Apple-Plattformen dar. Seit der erstmaligen Vorstellung auf der WWDC 2019 hat sich das Framework bereits massiv weiterentwickelt und stellt die momentan beste Möglichkeit dar, Apps für iOS und Co. zu entwickeln. Das gilt umso mehr, wenn man frisch in diese Thematik einsteigt.
- Der Einstieg in SwiftUI ist deutlich leichter und die Funktionsweise des Frameworks weniger komplex. Haben Sie einmal die Arbeit mit SwiftUI verinnerlicht, können Sie Apps für alle Apple-Plattformen entwickeln, ohne sich separat in mehrere verschiedene Frameworks einarbeiten zu müssen.

- SwiftUI *zusätzlich* zu AppKit, UIKit und WatchKit zu behandeln, hätte schlicht den Rahmen dieses Buches gesprengt. Es war klar, dass nur eine dieser UI-Technologien in diesem Buch Platz finden wird. Und da, wie gesagt, SwiftUI die Zukunft der UI-Entwicklung darstellt, mussten die bisherigen Inhalte zu AppKit, UIKit und WatchKit (leider) weichen.

Diese komplett neuen Kapitel zur App-Entwicklung stellen Ihnen SwiftUI detailliert vor. Sie erfahren so, wie Apples neues UI-Framework funktioniert, welche Views Ihnen zur Verfügung stehen, wie Sie Navigationsstrukturen umsetzen und effizient mit dem Status arbeiten. Sie erhalten so letztlich das Rüstzeug, Apps übergreifend für alle Apple-Plattformen entwickeln zu können. Gleichzeitig sind Sie für zukünftige Updates gewappnet, die SwiftUI um neue spannende Funktionen ergänzen werden.

Da SwiftUI *plattformübergreifend* funktioniert, nimmt so tatsächlich auch der Umfang dieses Handbuchs ein wenig ab. Das ist schlicht der Tatsache geschuldet, dass ich nicht mehr die Programmiergrundlagen für jedes einzelne System (macOS, iOS, watchOS, tvOS) erläutern muss. Haben Sie die Funktionsweise von SwiftUI einmal verinnerlicht, können Sie dieses Wissen umgehend auf alle verfügbaren Apple-Plattformen übertragen.

Updates und Aktualität

Apple veröffentlicht im Jahresrhythmus neue Versionen der Betriebssysteme sowie der Entwicklungsumgebung Xcode. Auch die Programmiersprache Swift wird regelmäßig um Updates ergänzt.

Damit Sie möglichst lange mit diesem Buch arbeiten können, haben Sie die Möglichkeit, sich für den kostenlosen *Update inside*-Service zu registrieren: Geben Sie unter

www.hanser-fachbuch.de/swift-update

diesen Code ein:

...

Dann erhalten Sie bis Dezember 2023 Aktualisierungen in Form zusätzlicher Kapitel als PDF. Darin stelle ich Ihnen wichtige Neuerungen vor und gehe auf Änderungen ein, die die Inhalte dieses Buches betreffen.

Aufbau des Buches

Mir ist es wichtig, das Buch als Referenzwerk nutzen zu können. Entsprechend finden Sie je einen eigenen Teil zur Programmiersprache Swift, zur Entwicklungsumgebung Xcode sowie zur App-Entwicklung mit SwiftUI. Das ermöglicht es Ihnen, sich separat mit diesen drei Bestandteilen auseinanderzusetzen und notwendige Infos zu erhalten, ohne diese Inhalte im Buch zu vermischen.

Ich möchte Sie in die Lage versetzen, eigene Apps entwickeln zu können. Zu diesem Zweck halte ich die Code-Beispiele im Buch bewusst klein und setze nicht auf die Umsetzung ganzer Projekte. Stattdessen sollen Sie das nötige Wissen erlangen, um Ihre eigenen Anwendungen erstellen zu können und zu diesem Zweck alles über die dafür notwendigen Komponenten erfahren.

Ein solches Grundverständnis erlaubt es Ihnen außerdem, sich selbstständig in weitere Bereiche einzuarbeiten und künftige Neuerungen mithilfe der Dokumentation anzuwenden.

Feedback

In diesem Sinne wünsche ich Ihnen nun viel Freude mit dem Buch und Erfolg beim Programmieren. Sollten Sie Feedback zum Buch oder Wünsche für kommende *Update inside*-Kapitel haben, können Sie mich gerne via E-Mail an

contact@thomassillmann.de

kontaktieren.

Ich bin inzwischen seit über zehn Jahren begeisterter Entwickler für die verschiedenen Apple-Plattformen. Ich finde das Apple-Ökosystem, die Programmiersprache Swift und die Entwicklungsumgebung Xcode faszinierend und arbeite jeden Tag voller Begeisterung damit. Ich hoffe, dass ich mit diesem Buch einen Teil meiner Begeisterung auch auf Sie übertragen kann.

Thomas Sillmann

Oktober 2021

1

Die Programmiersprache Swift

Nach dem Vorwort heie ich Sie nun an dieser Stelle ein weiteres Mal recht herzlich willkommen in der faszinierenden Welt von Swift. ☺ Bevor es im zweiten Kapitel konkret mit der Programmierung losgeht, mchte ich Ihnen an dieser Stelle etwas ber die Geschichte von Swift, die Bedeutung der Sprache im Apple-Kosmos sowie die Tools erzhlen, die fr uns als Entwickler unabdingbar sind.

■ 1.1 Die Geschichte von Swift

Keine Bange, das soll hier keine langatmige monotone Geschichtsstunde werden. Falls es Sie auch nicht im Geringsten interessieren sollte, wie Swift entstanden ist, drfen Sie diesen Abschnitt guten Gewissens gerne berspringen; ich wre Ihnen nicht bse deswegen. ☺ Obwohl ich den Hintergrund, wie Swift das Licht der Welt erblickte, durchaus spannend finde.

Dabei sind viele Details ber die genaue Entstehungsgeschichte von Swift gar nicht bekannt. Was man wei, ist, dass Chris Lattner wohl in gewisser Weise als „Vater“ von Swift bezeichnet werden kann. Er begann die Entwicklung von Swift im Juli 2010 bei Apple aus eigenem Antrieb heraus und zunchst im Alleingang. Ab Ende 2011 kamen dann weitere Entwickler dazu, whrend das Projekt im Geheimen bei Apple fortgefhrt wurde. Das erste Mal zeigte Apple die neue Sprache der Weltffentlichkeit auf der WWDC (Worldwide Developers Conference) 2014 (siehe Bild 1.1).



Bild 1.1 Auf der WWDC 2014 präsentierte Apple Swift erstmals der Weltöffentlichkeit.

Mit dieser erstmaligen Präsentation von Swift überraschte Apple sowohl Presse als auch Entwickler gleichermaßen. Dabei war die Sprache zunächst – ähnlich wie Objective-C – ausschließlich auf die Plattformen von Apple beschränkt. Ein Mac mitsamt der zugehörigen IDE Xcode von Apple waren also Pflicht, wollte man mit Swift Apps für macOS, iOS, watchOS oder tvOS entwickeln. Im Herbst 2014 folgte dann die erste finale Version von Swift, die Apple den Entwicklern zusammen mit einem Update für Xcode zugänglich machte.

Im darauffolgenden Jahr sorgte Apple auf der WWDC 2015 dann für die nächste große Überraschung. Sie präsentierten nicht nur die neue Version 2 von Swift, sondern gaben auch bekannt, dass Swift noch im gleichen Jahr Open Source werden würde. Dieses Versprechen wurde dann am 03. Dezember 2015 umgesetzt und Apple startete die Plattform *Swift.org*, um darüber zukünftig alle Weiterentwicklungen und Neuerungen zu Swift zusammenzutragen.

Seitdem hat die Sprache viele weitere Versionssprünge hinter sich und sie entwickelt sich noch immer stetig weiter. Einen großen Beitrag leistet hierbei auch die große Swift-Community, die mögliche Neuerungen und Änderungen vorantreibt.

Heute ist Swift die Sprache der Wahl, wenn es um die Entwicklung von Apps für Apple-Plattformen geht. Darüber hinaus findet Swift aber auch auf immer mehr Systemen wie Linux oder Windows Verwendung, ist dort aber bei weitem nicht so verbreitet wie im Apple-Umfeld.

■ 1.2 Die Bedeutung von Swift im Apple-Kosmos

Aus Sicht von Apple ist eines ganz offensichtlich: Swift gehört die Zukunft. Zwar unterstützt Apple noch immer die „alte“ Programmiersprache Objective-C, doch die wird im Gegensatz zu Swift kaum bis gar nicht weiterentwickelt.

Auch ist Swift in bestimmten Bereichen der App-Entwicklung inzwischen ein Muss. Manche Systemfunktionen lassen sich nur mit Swift und nicht mit Objective-C ansteuern. So zeigt Apple auf sehr drastische Art und Weise, welcher Programmiersprache Entwickler ihre Aufmerksamkeit widmen sollten.

Zwar gibt es noch sehr viel Objective-C-Code da draußen und sicherlich diverse Entwickler-Kollegen, die Objective-C im Vergleich zu Swift klar bevorzugen. Doch es ändert nichts daran, dass Swift die Zukunft gehört, und danach sollten auch wir Entwickler uns richten.

Und wenn ich hier persönlich werden darf: In meinen Augen ist Swift nicht nur eine äußerst mächtige und vielseitige, sondern auch wunderschön zu schreibende Programmiersprache. Mit modernen Ansätzen erleichtert sie außerdem Neulingen den Einstieg, und ich behaupte einmal, dass diese Aussage jeder unterschreiben kann, der bereits einmal mit Objective-C entwickelt hat.

Auch Swift mag nicht perfekt sein, doch die Entwicklung der letzten Jahre zeigt, dass die Sprache auf einem verdammt guten Weg ist und sowohl von Apple als auch der Open-Source-Community sehr viel Pflege erfährt.

Es mag abgedroschen klingen, doch *jetzt* ist definitiv der beste Zeitpunkt, sich der Programmierung mit Swift zu widmen.

■ 1.3 Das neue UI-Framework: SwiftUI

In Teil 3 dieses Buches, der sich vollumfassend den spezifischen Besonderheiten der App-Entwicklung widmet, werden Sie sehr sehr viel über SwiftUI lesen. Bei SwiftUI handelt es sich um ein sogenanntes *Framework*. Frameworks setzen sich aus einem Set verschiedener Funktionen zusammen, die Sie als App-Entwickler nutzen können, wenn Sie das Framework in Ihr App-Projekt einbinden.

Auch wenn SwiftUI nicht direkt etwas mit der Programmiersprache Swift zu tun hat, möchte ich an dieser Stelle ein paar Worte darüber verlieren. Das hängt nämlich mit einer grundsätzlichen Verständnisfrage zusammen, auf die ich in den letzten Jahren öfters gestoßen bin. So gilt:

Swift ist eine Programmiersprache. SwiftUI ist ein Framework, das auf Swift basiert und spezielle Funktionen für die Erstellung von Nutzeroberflächen zur Verfügung stellt.

Wenn Sie also die Begriffe Swift und SwiftUI hören, geht es nicht darum, welches der beiden Sie verwenden sollen. Wenn Sie SwiftUI einsetzen, kommen Sie um ein Verständnis der

Programmiersprache Swift gar nicht herum, denn darauf basiert SwiftUI nun einmal. Doch beschränkt sich der Einsatz von Swift nicht nur auf die Entwicklung von Nutzeroberflächen. So gesehen benötigen Sie Swift *immer*. Es ist die grundlegendste Säule der App-Entwicklung.

Wie gesagt, finden Sie in Teil 3 des Buches weitreichende Informationen zur Nutzung und Funktionsweise von SwiftUI.

■ 1.4 Was Sie als App-Entwickler brauchen

Um mit Swift Ihre eigenen Apps für iPhone, iPad und Co. entwickeln zu können, führt in der Regel kein Weg an Apples hauseigener Entwicklungsumgebung Xcode vorbei (warum ich hier „in der Regel“ schreibe, erläutere ich noch).

Xcode ist eine App, die exklusiv auf dem Mac zur Verfügung steht und die Sie kostenlos und bequem aus dem Mac App Store herunterladen und installieren können (siehe Bild 1.2). Erschrecken Sie nur nicht: Es ist normal, dass Xcode über 10 GByte an Speicherplatz für sich in Anspruch nimmt (und der Download so je nach Internetverbindung *lang* oder *sehr lang* dauern kann).

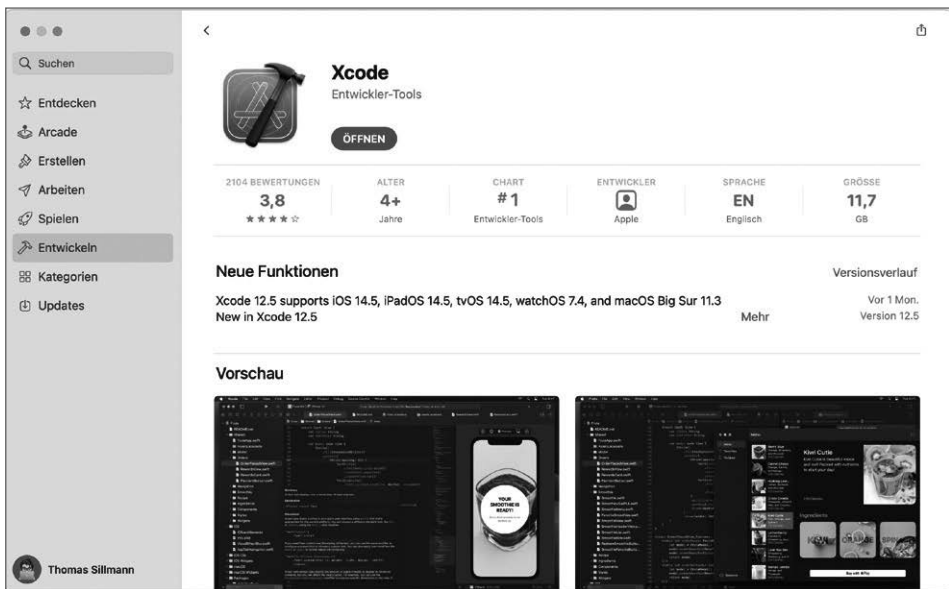


Bild 1.2 Xcode ist Apples vollwertige Entwicklungsumgebung für die Programmierung von Apps.

Xcode bringt alles mit, was Sie für die App-Entwicklung benötigen. Dazu gehört auch Unterstützung für Swift und SwiftUI. Außerdem verfügt Xcode über verschiedene Simulatoren, die es Ihnen beispielsweise ermöglichen, iPhone-Apps direkt auf Ihrem Mac auszuführen und zu testen.

Mithilfe von Xcode erstellen Sie Projekte für Ihre Apps und verwalten darin sowohl den Quellcode als auch weitere Ressourcen wie Bilder. Xcode kann Ihnen darüber hinaus beim Auffinden von Fehlern in Ihrer App helfen und ermöglicht es sogar, Ihre fertige App direkt in den App Store hochzuladen. Mehr zur Veröffentlichung von Apps erfahren Sie im letzten Teil dieses Buches.

Aufgrund dieses großen Funktionsumfangs kann Xcode als durchaus komplexe Anwendung angesehen werden. Das gilt umso mehr, wenn man zuvor noch keinerlei Programmiererfahrung gesammelt hat. Aus diesem Grund liefert Ihnen der zweite Teil dieses Buches einen Rundumüberblick über den Aufbau und die verschiedenen Bestandteile von Apples Entwicklungsumgebung.

Die abgespeckte Xcode-Alternative: Swift Playgrounds

Ich habe ja zu Beginn dieses Abschnitts geschrieben, dass zur App-Entwicklung „in der Regel“ kein Weg an Xcode vorbeiführt. Tatsächlich gibt es für das iPad eine Alternative zur vollwertigen Entwicklungsumgebung auf dem Mac: die App *Swift Playgrounds* (siehe Bild 1.3).

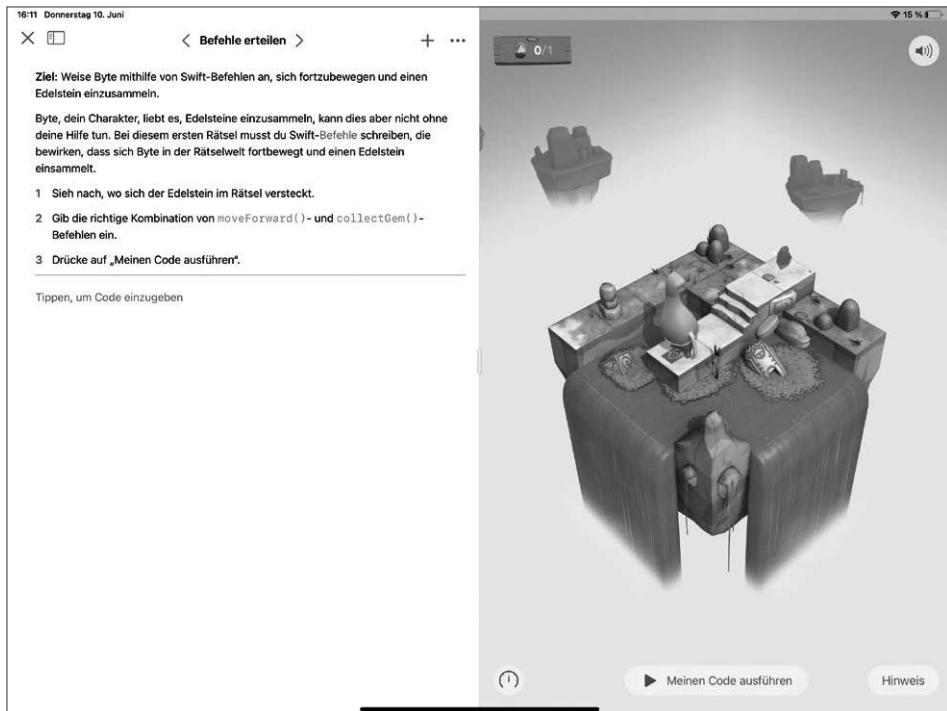


Bild 1.3 Swift Playgrounds ermöglicht das Programmieren auch auf Apples iPad.

Swift Playgrounds entstand ursprünglich, um Nutzern die Grundlagen der Programmierung spielerisch näherzubringen. Diese Funktion besitzt die App auch heute noch und sie ist definitiv einen Blick wert. Zusätzlich ermöglicht sie es inzwischen aber auch, vollwertige Apps für iPhone und iPad mit ihr zu entwickeln und direkt in den App Store hochzuladen.

Das ist definitiv eine großartige Sache, doch sollte man sich keinen Illusionen hingeben: Für die professionelle Entwicklung von Apps ist noch immer Xcode das Mittel der Wahl. Xcode bietet deutlich mehr Funktionen und Möglichkeiten als Swift Playgrounds. Dafür erlaubt es Swift Playgrounds, Apps nicht nur auf dem Mac, sondern auch auf Apples iPad zu entwickeln.

Übrigens: Für die App-Entwicklung unterstützt Swift Playgrounds ausschließlich die Programmiersprache Swift sowie das neue UI-Framework SwiftUI. Das ist also ein klarer Grund mehr, sich heute ausgiebig mit diesen beiden so wichtigen Technologien für Apple-Entwickler auseinanderzusetzen.

■ 1.5 Programmieren für Beginner (und darüber hinaus): Playgrounds

Wenn es um die Entwicklung von Apps geht, arbeitet man in sogenannten *Projekten*. In diesen verwaltet man neben dem Quellcode auch alle zusätzlichen Ressourcen wie Bilder. Außerdem enthält solch ein Projekt weitere Informationen wie die Versionsnummer und die Unterstützung für optionale Services (beispielsweise iCloud).

Gerade am Anfang erhält man mit solch einem Projekt von Haus aus bereits sehr viele Dateien, die einen gerade als Anfänger überfordern können. Außerdem möchte man in manchen Fällen auch einfach einmal ein wenig mit Swift experimentieren, ohne dafür gleich ein ganzes Projekt erzeugen zu müssen.

Abhilfe schaffen hier die sogenannten *Playgrounds* (siehe Bild 1.4). In einem Playground können Sie einfach Code drauflos schreiben, ohne sich mit Projektstrukturen und dem Ablauf Ihrer App beschäftigen zu müssen. Per Klick auf einen Button führen Sie den Code aus und sehen umgehend dessen Ergebnis.

Gerade zu Beginn ist es absolut sinnvoll, Ihren Code in Playgrounds zu schreiben und zu testen. So machen Sie sich schnell mit den Grundlagen und der Syntax von Swift vertraut, ohne von den genannten Projektstrukturen ganzer Apps womöglich überfordert zu werden.

Gerade wenn Sie bisher nur wenig oder gar keine Erfahrung mit Swift gesammelt haben, empfehle ich Ihnen, die Beispiele in diesem ersten Teil des Buches mithilfe von Playgrounds ebenfalls zu programmieren. Nutzen Sie die Flexibilität von Playgrounds, um ruhig auch selbst zu experimentieren und Dinge auszuprobieren, die Sie interessieren oder die Ihnen in den Sinn kommen. Besser kann man sich mit der Programmierung gar nicht vertraut machen!

Übrigens sind Playgrounds nicht nur ein ideales Mittel, um das Programmieren grundlegend zu lernen. Auch Profis nutzen sie, um sich bequem mit neuen Frameworks oder der Umsetzung einer komplexen Programmlogik auseinanderzusetzen. Entwickler können so Ihren Code unabhängig von einem vollständigen und meist komplexen App-Projekt testen. Hat man eine passende Lösung erarbeitet, kann man diese anschließend in das eigentliche Projekt übertragen.

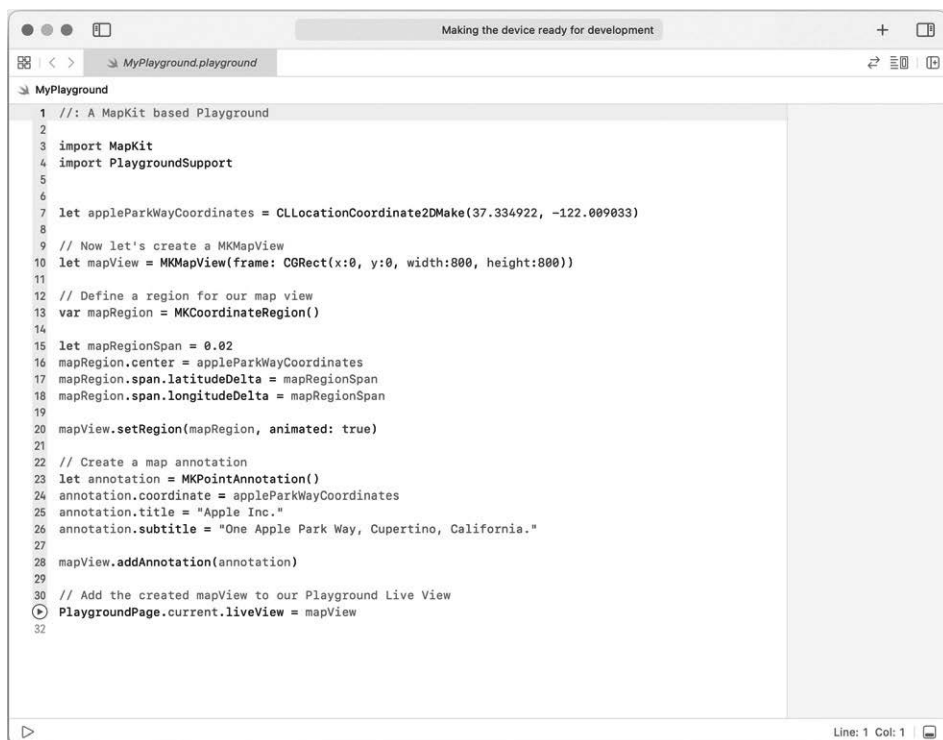


Bild 1.4 Playgrounds sind ein ideales Mittel, um zu experimentieren und das Programmieren zu lernen.

Mehr zur Verwendung von Playgrounds und wie Sie solche mit Xcode erstellen, erfahren Sie im zweiten Teil dieses Buches.



Swift Playgrounds auf dem iPad

Sie können Playgrounds auch auf dem iPad erstellen und so auf Apples Tablet programmieren. Zu diesem Zweck steht Ihnen eine App mit dem passenden Namen *Swift Playgrounds* zur Verfügung, die Sie kostenlos aus dem App Store laden können (siehe auch Abschnitt 1.4, „Was Sie als App-Entwickler brauchen“).

Die App eignet sich auch ideal für Einsteiger, die das Programmieren lernen möchten. Sie finden darin verschiedene kleine Lernkurse, die spielerisch die Grundlagen der Entwicklung mit Swift vorstellen. Darüber hinaus können Sie mit der App aber auch die eben beschriebenen Playgrounds erstellen und so ganz frei mit Code und dem Programmieren experimentieren.

Übrigens steht die Swift-Playgrounds-App auch als dedizierte Anwendung auf dem Mac zur Verfügung, unabhängig von Xcode. Falls Sie die genannten Lernkurse daher auch auf dem Mac ausprobieren möchten, können Sie das über die Swift-Playgrounds-App tun. Wenn Sie aber erst einmal näher mit der

Swift-Programmierung und der Entwicklungsumgebung Xcode vertraut sind, benötigen Sie die App nicht wirklich auf dem Mac. Xcode liefert Ihnen alles, was Sie brauchen, um Ihre eigenen Anwendungen umzusetzen und Code in separaten Playgrounds zu testen.

■ 1.6 Weitere wichtige Ressourcen

In der Welt der App-Entwicklung gibt es keinen Stillstand. In mehr oder weniger regelmäßigen Abständen erscheinen neue Versionen von Programmiersprachen, Entwicklungsumgebungen und Betriebssystemen. Manche dieser Updates verbessern lediglich die Stabilität oder ergänzen praktische neue Funktionen. Andere wiederum ersetzen bekannte Mechanismen durch neue. Entsprechend wichtig ist es, als Entwickler up to date zu sein.

Erfreulicherweise gibt es diverse Ressourcen, über die Sie sich auf dem Laufenden halten können. Nachfolgend möchte ich Ihnen einige davon vorstellen, die ich selbst als regelmäßige Anlaufstellen nutze.

1.6.1 Apple-Developer-App

Für macOS, iOS und tvOS stellt Apple eine kostenlose Developer-App zur Verfügung (siehe Bild 1.5). Darin finden Sie eine Vielzahl an Videos und diverse Artikel, die spezifische Aspekte der App-Entwicklung erläutern.

Insbesondere erhalten Sie über die App Zugriff auf die verschiedenen Session-Videos, die Apple im Zuge seiner alljährlichen Entwicklerkonferenz (Worldwide Developers Conference, kurz WWDC) veröffentlicht. Eine bessere Möglichkeit, sich über die neuesten Techniken im Bereich Swift, iOS und Co. zu informieren, gibt es wohl nicht.

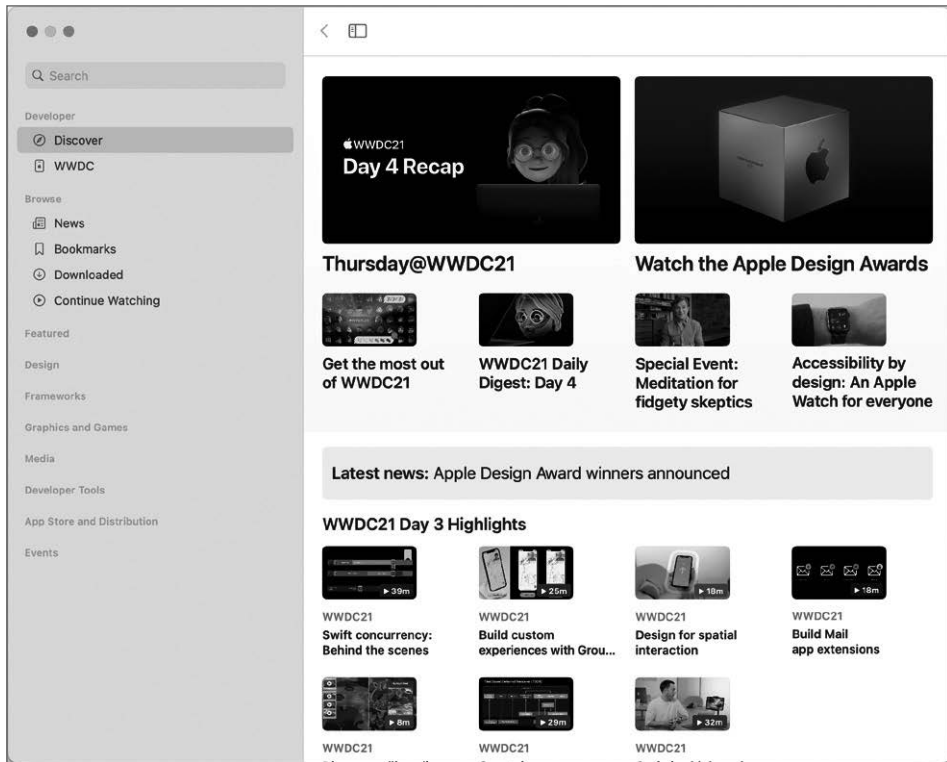


Bild 1.5 Apples offizielle Developer-App ist die ideale Ergänzung für alle Entwickler.

1.6.2 Apples Developer-Website

Eine zentrale Anlaufstelle für alle Apple-Entwickler stellt Apples offizielle Developer-Website dar, die Sie mittels des Links <https://developer.apple.com> erreichen (siehe Bild 1.6). Sie liefert Ihnen eine Übersicht über die neuesten Entwicklungen und ermöglicht den Zugriff auf Beta-Versionen von iOS und Co. Über diese Website erstellen Sie zudem Ihren eigenen Entwickler-Account und können auf verschiedene Ressourcen wie Zertifikate und App-IDs zurückgreifen. Mehr zu diesen Möglichkeiten erfahren Sie im letzten Teil dieses Buches.

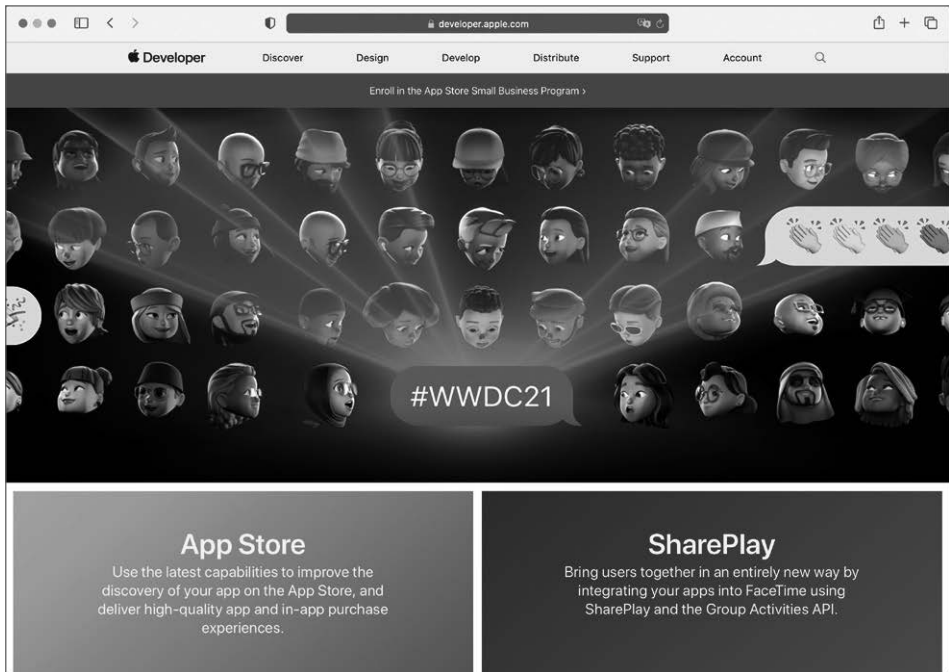


Bild 1.6 Über Apples offizielle Entwickler-Website haben Sie Zugriff auf aktuelle Infos und Beta-Versionen.

1.6.3 Swift.org

Wenn Sie sich im Speziellen für die Programmiersprache Swift interessieren, ist die offizielle Website *swift.org* genau das richtige (siehe Bild 1.7). Neben grundlegenden Informationen zu Swift finden Sie dort auch eine vollständige Dokumentation sowie einen Blog, der über kommende Updates und Änderungen der Sprache informiert. Vorbeischaun lohnt sich!

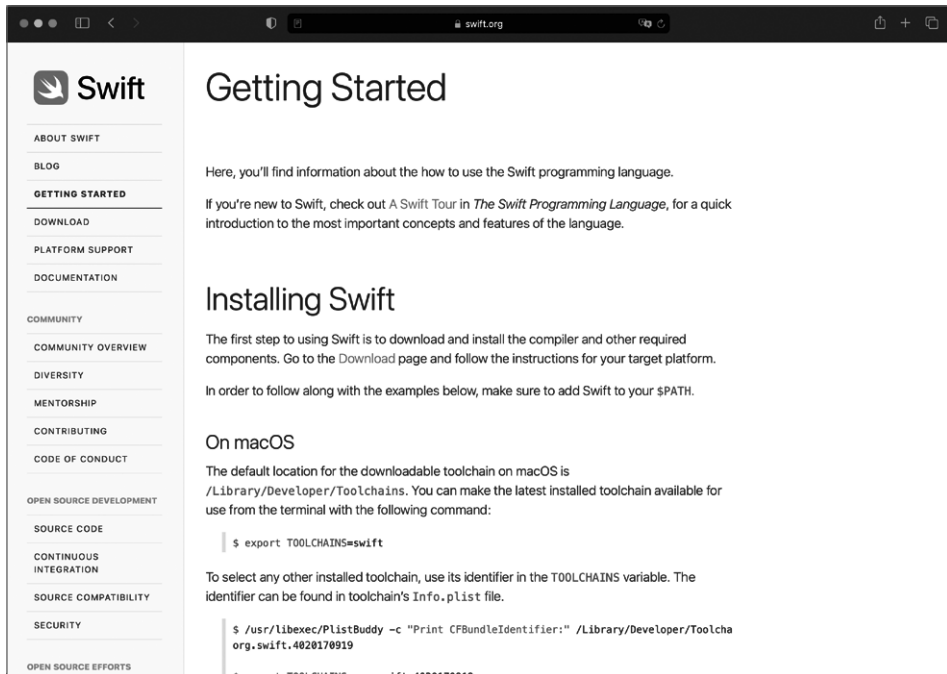


Bild 1.7 Die geballte Ladung Swift gibt es auf der offiziellen Website der Sprache unter *swift.org*.

1.6.4 In eigener Sache

Als Besitzer und Leser dieses Buches haben Sie Zugriff auf einen ganz besonderen Service namens *Update inside*. Bis Dezember 2023 erhalten Sie darüber in unregelmäßigen Abständen neue Buch-Kapitel in PDF-Form, die Sie sich bequem herunterladen können. Diese Kapitel widmen sich kommenden Neuerungen in Swift, iOS und Co. und behandeln zusätzliche Themen, die nicht Teil dieses Buches sind. Update inside ist somit eine weitere Möglichkeit, über kommende Aktualisierungen informiert zu werden.

Zusätzlich finden Sie unter

<https://letscode.thomassillmann.de>

meinen ganz persönlichen Entwickler-Blog. Dort veröffentliche ich in unregelmäßigen Abständen neue Beiträge rund um die App-Entwicklung für Apple-Plattformen. Außerdem gibt es auf meinem YouTube-Kanal unter

<https://www.youtube.com/user/Sullivan1988>

regelmäßig neue Videos zur App-Entwicklung.

21

Grundlagen der App-Entwicklung

Sie haben inzwischen bereits zwei essenzielle Elemente zur App-Entwicklung für Apple-Plattformen kennengelernt: die Programmiersprache Swift und die Entwicklungsumgebung Xcode. In diesem Teil des Buches erfahren Sie nun alles über die übrigen Elemente, die zur Programmierung einer eigenen Anwendung unabdingbar sind.

■ 21.1 Die Basis: SwiftUI

SwiftUI ist ein Framework zur Erstellung von Nutzeroberflächen für macOS, iOS, iPadOS, watchOS und tvOS, sprich für alle Plattformen von Apple (siehe Bild 21.1). Es bringt von Haus aus verschiedene Elemente wie Textfelder, Buttons und Schalter mit, die Sie so in Ihre eigenen Apps einbinden können.

Der große Vorteil von SwiftUI ist, dass es – wie eben geschrieben – auf allen Apple-Plattformen zur Verfügung steht. Das macht es deutlich leichter, möchte man Programme für mehr als nur ein Betriebssystem von Apple entwickeln. SwiftUI funktioniert auf allen Plattformen gleich, was bedeutet, dass man die Funktionsweise des Frameworks nur einmalig verinnerlichen muss, um es anschließend überall einsetzen zu können. Apple selbst stellte in diesem Zusammenhang die folgende Aussage zu SwiftUI auf:

Learn once, apply anywhere.

SwiftUI stellt somit das ideale Framework dar, um in die Entwicklung von Apps für iOS und Co. einzusteigen. Hat man das grundlegende Prinzip einmal verinnerlicht, lässt es sich auf alle Bereiche des Apple-Kosmos übertragen.

SwiftUI ist aber nicht nur für das Aussehen einer Anwendung verantwortlich. Es kümmert sich auch um den kompletten Lebenszyklus eines Programms. Mehr dazu erfahren Sie in Abschnitt 21.4, „Aufbau einer App“.



Bild 21.1 SwiftUI ist Apples neues Framework zur Gestaltung von Nutzeroberflächen für Apps.



Was ist mit AppKit, UIKit und WatchKit?

Vor SwiftUI nutzte Apple andere Frameworks als Basis für Nutzeroberflächen. Tatsächlich stehen sie auch heute noch zur Verfügung und können so für die Entwicklung von Apps genutzt werden. Es handelt sich bei diesen Frameworks um AppKit (für macOS), UIKit (für iOS, iPadOS und tvOS) sowie WatchKit (für watchOS).

Diese Aufstellung zeigt zugleich aber den größten Nachteil dieser Frameworks auf. Abhängig davon, für welche Plattform man entwickeln möchte, musste man ein anderes dieser Frameworks verwenden. Das ist nicht zuletzt mit einigem Aufwand verbunden. Entwickelte man beispielsweise Apps fürs iPhone und wollte im nächsten Schritt auch für den Mac entwickeln, musste man sich zunächst mit der Funktionsweise von AppKit auseinandersetzen. Jedes der Frameworks besitzt andere Klassen und View-Hierarchien, die es Entwicklern nicht erlaubten, schnell eine Anwendung für eine andere Plattform zu kreieren.

SwiftUI löst dieses Problem. Zwar gibt es auch in SwiftUI einige Unterschiede in Bezug auf die verschiedenen Apple-Plattformen (wie wir noch sehen werden). Doch die Basis und das Grundgerüst sind identisch.

Sowohl AppKit, UIKit, WatchKit als auch SwiftUI in diesem Buch abzudecken, hätte bei weitem den verfügbaren Rahmen gesprengt. Aus diesem Grund konzentriere ich mich mit SwiftUI auf das Framework, das langfristig deutlich relevanter für die Entwicklung von Apps für Apple-Plattformen sein dürfte und darüber hinaus eine Vielzahl an Vorteilen gegenüber AppKit, UIKit und WatchKit zu bieten hat.

■ 21.2 Bestandteile einer App

Jede Anwendung setzt sich in der Regel aus zwei essenziellen Bestandteilen zusammen: **Daten** und **Ansichten**.

Die **Daten** regeln das Verhalten und die Funktionsweise einer App. Sie sind die Logik, die dafür sorgt, dass eine Anwendung so arbeitet, wie sie soll (oder im Fehlerfall eben nicht ☹).

Die **Ansichten** stellen das Nutzer-Interface dar, das der Anwender zu sehen bekommt. Er verwendet es, um mit Ihrer App zu interagieren und Aktionen auszulösen.



Model und Views

Für die beiden beschriebenen Elemente der Daten und Ansichten verwendet man in der Programmierung typischerweise auch deren englischsprachige Begriffe. Die Daten einer App bezeichnet man so als *Model*, die Ansichten als *Views*. Wenn ich im Folgenden also beispielsweise einmal vom *Model* schreibe, dann beziehe ich mich auf die Daten und die Logik einer Anwendung.

Diese Daten und Ansichten stehen im Einklang miteinander. So können Daten unter anderem bestimmen, welche Informationen der Nutzer in den Ansichten zu sehen bekommt. Gleichzeitig führen Aktionen, die der Anwender über die Ansichten auslöst, möglicherweise zur Manipulation der Daten.

Bei der Programmierung einer App trennen Sie typischerweise Daten und Ansichten voneinander. Trennung bedeutet in diesem Kontext, dass Sie Code, der sich um die Logik und Funktionalität Ihrer Anwendung kümmert, nicht in Ansichten unterbringen. Umgekehrt gilt das Gleiche: Die Code-Dateien für Ihre Ansichten sollten nicht das grundlegende Verhalten Ihrer App beeinflussen.

In der Praxis kommen in Ihrem App-Projekt demnach wenigstens zwei Arten von Code-Dateien zum Einsatz: solche, die Ihre Daten und die Programmlogik enthalten, und solche, die sich um das Aussehen der Anwendung kümmern.

21.2.1 Umsetzung der Daten

Um die Daten und die Logik einer App umzusetzen, nutzen Sie im einfachsten Fall Swift-Dateien, in denen Sie die benötigten Typen und Methoden definieren. Zusätzlich stellt Apple Ihnen aber auch ergänzende Frameworks und Funktionen zur Verfügung, die Sie bei der Umsetzung der App-Logik und Datenhaltung unterstützen. Dazu gehören unter anderem:

- Core Data
- UserDefaults

Mehr zu diesen Elementen erfahren Sie in Kapitel 27, „Datenhaltung“.

21.2.2 Umsetzung der Ansichten

Um die Ansichten für Ihre App zu kreieren, nutzen Sie typischerweise Apples SwiftUI-Framework. Es stellt Ihnen entsprechende Funktionen zur Verfügung, um das Aussehen und den Aufbau Ihrer Anwendung festzulegen.

Die Ansichten erstellt man – genauso wie die Daten – mithilfe von Code. Auch zur Umsetzung von Views auf Basis von SwiftUI können Sie simple Swift-Dateien verwenden.

Im Zusammenspiel mit SwiftUI und dem Erstellen der Nutzeroberflächen bietet die Entwicklungsumgebung Xcode noch zusätzliche Funktionen. So können Sie sich direkt im Editor eine Vorschau Ihrer Ansichten anzeigen lassen und diese sogar darüber bearbeiten und anpassen. Mehr zu diesen Möglichkeiten erfahren Sie in Kapitel 29, „Preview und Library“.

21.2.3 Weitere Frameworks

Apple bietet eine Vielzahl verschiedener Frameworks an, die sich aus Xcode heraus nutzen lassen. Sie decken jeweils spezifische Funktionen ab. So können Sie mithilfe von ARKit Unterstützung für Augmented Reality in Ihren Apps ergänzen oder mittels MapKit das Kartenmaterial von Apple nutzen.

Abhängig davon, welche Funktionen Sie so in Ihren Apps benötigen, finden Sie womöglich bereits passende Lösungen von Apple in Form zusätzlicher Frameworks. Diese können Sie in Ihren Projekten importieren und anschließend direkt darauf zugreifen.

■ 21.3 Die Syntax von SwiftUI

SwiftUI verfügt über eine sogenannte *deklarative Syntax*. Mit deren Hilfe beschreiben Sie den Aufbau Ihrer Ansichten und nutzen dazu eine Art Baumstruktur.

Ein Beispiel zur Erläuterung dieser Syntax finden Sie in Listing 21.1. Es zeigt eine SwiftUI-View, die zunächst auf einem Element namens `VStack` basiert. `VStack` ist ein Typ des

SwiftUI-Frameworks und dient dazu, Views untereinander anzuordnen. Welche Views das sind, legen Sie mithilfe eines Closures fest. Darin führen Sie nacheinander die gewünschten Views auf. In dem gezeigten Beispiel sind die Elemente der Typen `Text`, `Divider` und `HStack`.

`HStack` dient – genau wie `VStack` – der Gruppierung von Views, nur ordnet ein `HStack` diese horizontal nebeneinander an. Die zu gruppierenden Views definiert man erneut mithilfe eines Closures. Der `HStack` enthält in diesem Fall ein Element vom Typ `Image` und eines vom Typ `Text`.

In seiner Gesamtheit zeigt diese View also drei Elemente untereinander an, wobei das letzte Element aus zwei Views besteht, die nebeneinander dargestellt werden (siehe Bild 21.2).

Listing 21.1 Beispiel zur deklarativen Syntax von SwiftUI

```
VStack {  
    Text("Ein Text")  
    Divider()  
    HStack {  
        Image("MyImage")  
        Text("Ein anderer Text")  
    }  
}
```

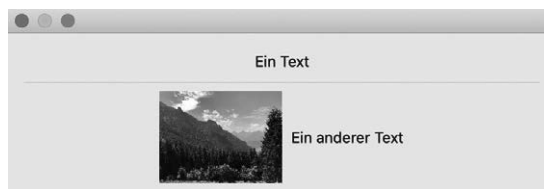


Bild 21.2 Die deklarative Syntax legt den Aufbau von SwiftUI-Views exakt fest.

Mithilfe dieser deklarativen Syntax legt man so den genauen Aufbau von Ansichten fest. Sie müssen dieses Konzept zu diesem Zeitpunkt noch nicht vollständig verinnerlichen und ebenso wenig die genaue Funktionsweise von Stacks verstehen, darauf gehe ich später noch im Detail ein. Sie sollen nur schon einmal grundsätzlich wissen, welche Art von Syntax bei der Arbeit mit SwiftUI zum Einsatz kommt und welchen Zweck sie erfüllt.

■ 21.4 Aufbau einer App

Entwickelt man Apps mithilfe von SwiftUI, setzen sie sich aus insgesamt drei Bestandteilen zusammen:

- App
- Scenes
- Views

Die **Views** kennen wir in diesem Zusammenhang bereits grundlegend. Sie sind unsere Ansichten, die einzelne Teile unseres User Interface widerspiegeln.

Eine **Scene** beschreibt in SwiftUI ein Anwendungsfenster. Apps unter iOS besitzen genau ein solches Fenster, unter iPadOS hingegen können Apps auch den parallelen Einsatz mehrerer Fenster unterstützen. Auf dem Mac ist es sogar gang und gäbe, mehrere Fenster zu verwenden.

Genau ein solches Fenster entspricht in SwiftUI einer Scene. Hat der Nutzer also parallel zwei Fenster einer Anwendung geöffnet, so sind zwei Scenes aktiv. Die Scene selbst setzt sich aus ein oder mehreren Views zusammen. Diese Zusammenstellung aus Views bestimmt also, welche Inhalte ein Programmfenster anzeigt.

Bleibt zu guter Letzt noch die **App**. Aus Sicht einer Anwendung ist das jener Teil, der den Startpunkt der Anwendung im Code markiert. Über diesen Startpunkt legt man die verschiedenen Arten von Scenes fest, die eine App besitzt. Das kann beispielsweise ein Fenster für die eigentliche Anwendung und ein separates für die Einstellungen sein.

Entsprechend lässt sich der Aufbau einer Anwendung wie folgt zusammenfassen: Basis und Startpunkt ist die **App**. Diese legt fest, welche verschiedenen Anwendungsfenster (sprich **Scenes**) zur Verfügung stehen. Jede Scene wiederum besteht aus **Views**, die das Aussehen eines Fensters definieren. Bild 21.3 skizziert dieses Konzept.

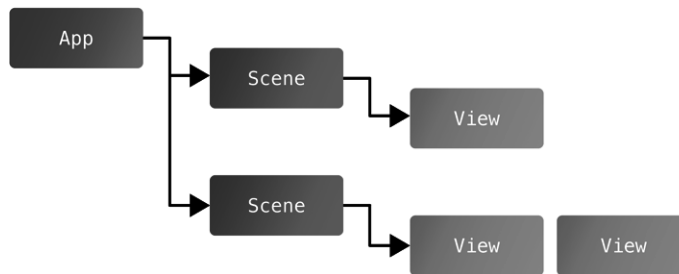


Bild 21.3 Apps auf Basis von SwiftUI setzen sich aus insgesamt drei verschiedenen Bestandteilen zusammen.

■ 21.5 Das View-Protokoll

Zum Erstellen eigener Ansichten mithilfe von SwiftUI stellt das Framework ein Protokoll namens **View** bereit. Es besitzt eine einzige Anforderung in Form der **body**-Property. Über die **body**-Property legen Sie das Aussehen Ihrer View fest und geben diese View als Ergebnis zurück. Ein simples Beispiel für eine View, die den Text „Hello World“ ausgibt, finden Sie in Listing 21.2.

Listing 21.2 Umsetzung einer Hello-World-View

```

struct HelloWorldView: View {
    var body: some View {
        Text("Hello World")
    }
}
  
```

```
}  
}
```

Zwei Dinge sind an Listing 21.2 besonders interessant. Einerseits ist klar zu erkennen, dass der darin eigens definierte Typ namens `HelloWorldView` konform zum genannten View-Protokoll ist. Zum anderen entspricht die `body`-Property dem Typ `some View`.

Diese Deklarationen sind typisch für SwiftUI. Durch `some View` können Sie über die `body`-Property jede beliebige View zurückgeben, ohne sich explizit festlegen zu müssen. Zugleich verbergen Sie so die genaue Implementierung. Die `body`-Property garantiert lediglich, dass man darüber als Ergebnis eine View erhält; welche, spielt keine Rolle.



Views sind Structures

Sehr wichtig ist auch zu wissen, dass Views in SwiftUI ausschließlich auf Structures basieren. Das hängt vor allem mit dem Speichermanagement von SwiftUI zusammen. So nutzt SwiftUI die Informationen aus der `body`-Property nur einmalig, um die entsprechende Ansicht auf dem Display darzustellen. Ist diese Aufgabe erledigt, wird die entsprechende View-Instanz direkt wieder verworfen.

Wenn Sie also eine neue View umsetzen, nutzen Sie als Basis immer eine Structure!

Innerhalb der `body`-Property greifen Sie sowohl auf Views aus dem SwiftUI-Framework als auch auf bereits von Ihnen selbst zuvor kreierte Views zurück. Mit `Text` haben Sie bereits eine View als Beispiel kennengelernt. Sie ist Teil des SwiftUI-Frameworks und dient zur Darstellung eines einfachen Labels. In den kommenden Kapiteln werden Sie noch viele weitere der Views kennenlernen, die Sie aus SwiftUI heraus nutzen können.

■ 21.6 Aktualisierung von Views mittels Status

Ein essenzieller Aspekt jeder Software ist die Veränderung von Daten und die entsprechende Aktualisierung der Ansichten. Wenn Sie beispielsweise eine App zum Verwalten Ihres Einkaufszettels nutzen, soll sich die Liste der einzukaufenden Lebensmittel aktualisieren, sobald Sie neue Einträge hinzufügen. Das ist für uns mehr oder weniger selbstverständlich.

Doch nicht nur eigene Aktionen, die Nutzer über die Ansichten durchführen, können zu notwendigen Aktualisierungen von Views führen. Eine E-Mail-Anwendung soll beispielsweise neue E-Mails automatisch im Hintergrund laden und direkt anzeigen, sobald sie eintreffen; ohne dass der Nutzer hierfür explizit eine Aktion durchführen muss.

Arbeitet man mit SwiftUI, bestimmt der sogenannte *Status*, ob und wann Views automatisch aktualisiert werden. Einfach ausgedrückt ist der Status eine Eigenschaft, die man einer View hinzufügen kann. Ändert sich nun diese Eigenschaft (beispielsweise weil ihr ein neuer Wert zugewiesen wird), führt das automatisch zu einer Aktualisierung der Ansicht.

Es gibt in SwiftUI verschiedene Möglichkeiten, einen Status abzubilden; und keine Sorge, wir werden sie alle noch betrachten. Da dieses Konzept aber so essenziell ist (und sich darüber hinaus von den Konzepten unterscheidet, die unter AppKit, UIKit und WatchKit zum Einsatz kommen), möchte ich es an dieser Stelle zumindest einmal kurz umreißen.

Kurz gesagt gilt: Sollen sich Ansichten (beziehungsweise deren Inhalte) aktualisieren können, während die Anwendung läuft, benötigen Sie hierfür einen Status. Ein erstes Beispiel dazu finden Sie in Listing 21.3. Die darin deklarierte Property `isActive` ist als änderbarer Status definiert (zu erkennen am Property Wrapper `State`). Das ermöglicht es, den Wert dieser Property innerhalb der View zu verändern. Dazu steht in dem Beispiel eine Schaltfläche vom Typ `Button` bereit, die bei Betätigung den aktuellen Wert von `isActive` invertiert.

Jede Änderung des Status führt wie beschrieben zur Aktualisierung der View. Im Beispiel aus Listing 21.3 bedeutet das, dass jede Änderung von `isActive` die komplette View neu generiert. Der gesamte Inhalt der `body`-Property wird in diesem Fall neu erzeugt. Das hat zur Folge, dass der Text, der zu Beginn der View ausgegeben wird, sich entsprechend der `isActive`-Änderung aktualisiert. Der Text basiert nämlich auf dem aktuellen Wert von `isActive` und entspricht so entweder „Aktiv“ oder „Inaktiv“ (siehe Bild 21.4).

Listing 21.3 Deklaration und Änderung eines Status

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

    var body: some View {
        VStack {
            Text(isActive ? "Aktiv" : "Inaktiv")
            Button("Toggle isActive") {
                isActive.toggle()
            }
        }
    }
}
```



Bild 21.4 Die Betätigung des Buttons führt aufgrund der Statusänderung zur automatischen Aktualisierung der View (siehe die Textausgabe).

Sie müssen zu diesem Zeitpunkt noch nicht verstehen, was `@State` konkret bedeutet oder wie `Button` funktioniert. Dazu erhalten Sie in den kommenden Kapiteln noch ausführliche und detaillierte Informationen. Stattdessen soll Ihnen dieses Beispiel lediglich einen ersten Eindruck darüber vermitteln, wie die Aktualisierung von Views in SwiftUI abläuft und welches grundlegende Prinzip dahintersteckt.



Halten Sie Views klein

In Listing 21.3 haben Sie gesehen, dass eine Änderung des Status die zugehörige View komplett neu erzeugt. Aus diesem Grund ist es wichtig, Views in SwiftUI möglichst klein und kompakt zu halten. Eine View sollte nur eine spezifische Ansicht abdecken und nicht unbedingt alle Inhalte eines ganzen Anwendungsfensters. Stattdessen nutzt man diese vielen kompakten View-Elemente, um daraus größere zu generieren.

Behalten Sie diesen Grundsatz an dieser Stelle einfach bereits einmal im Hinterkopf. In den kommenden Abschnitten werden Sie eine Vielzahl von praktischen Beispielen sehen, die sich des Einsatzes solch kompakter Views bedienen. Diese werden dann auch noch einmal deutlicher machen, warum es in der Praxis so wichtig ist, SwiftUI-Views möglichst klein zu halten.

■ 21.7 Grundlagen des Status

Den Status und die verschiedenen Möglichkeiten, die er bietet, lernen Sie noch im Detail in Kapitel 26, „Status“, kennen. Für den Einsatz vieler View-Elemente in SwiftUI ist es aber unabdingbar, wenigstens zwei Arten des Status schon einmal grundlegend zu kennen. Darum stelle ich Sie Ihnen an dieser Stelle bereits einmal vor.

Den Property Wrapper `State` haben Sie bereits in Abschnitt 21.6, „Aktualisierung von Views“, kennengelernt. Mit seiner Hilfe definieren Sie eine änderbare Information, die Sie in einer View anpassen können.

Wichtig: `State` ist fest mit der View verknüpft, in der dieser Property Wrapper zum Einsatz kommt. Die View muss einen Standardwert für die entsprechende Property generieren und hält diese Information fest im Speicher. Man spricht in diesem Fall auch von einer *Source of Truth* (dazu erfahren Sie ebenfalls mehr in Kapitel 26).

Diese Tatsachen machen `State` aber nicht zum idealen Status für jede Art von View. Betrachten wir dazu beispielhaft einmal einen Schalter, der zwei Zustände kennt: an oder aus. Da sich der Zustand durch Betätigung des Schalters ändern kann, muss er in der entsprechenden View als änderbarer Status umgesetzt werden.

Generell könnten wir hier also `State` einsetzen, um die Änderung des Schalterzustands technisch zu ermöglichen. Doch das hieße, die View selbst verwalte die Information, ob sie an oder aus ist. In der Regel steuern wir diesen Zustand aber von *außerhalb* einer solchen View. Dazu ist die entsprechende Information an einer anderen Stelle gespeichert und wird nur an die View *weitergereicht*. So lässt sich auch ein solcher Schalter flexibel einsetzen. Einmal bezieht er sich auf eine Auswahl zwischen Light und Dark Mode, ein andermal steuert er, ob sich der Nutzer zum Newsletter anmeldet oder nicht. Diese Information kann variieren, also muss sie der View übergeben werden können (statt dass die View die Information – wie im Falle von `State` – selbst definiert).

Die Lösung für dieses Problem lautet Binding. Binding ist ein weiterer Property Wrapper aus dem SwiftUI-Framework und definiert – genau wie State – einen änderbaren Status innerhalb einer View. Der Wert dieses Status ist aber nicht innerhalb der View selbst, sondern an einer anderen Stelle gespeichert.

Betrachten wir dazu direkt einmal ein konkretes Beispiel. In Listing 21.4 erfolgt die Umsetzung einer Ansicht namens RatingView. Sie stellt fünf Sternen-Schaltflächen nebeneinander dar und soll dazu dienen, eine Bewertung abzugeben (siehe Bild 21.5).

RatingView benötigt aus zwei Gründen Zugriff auf eine Bewertung: So kann sie einerseits die aktuelle Bewertung von Beginn an *anzeigen*, als auch bei Betätigung einer der Sternen-Schaltflächen diese *ändern*. Um was für eine Bewertung es sich handelt, interessiert RatingView aber nicht. Es kann um Bücher, Filme, Apps oder etwas gänzlich anderes gehen. Das macht die View so flexibel und universell einsetzbar.

Aus diesem Grund wird die Bewertung mithilfe des Binding-Property Wrappers implementiert. Die entsprechende Property besitzt den Namen `rating` und ist vom Typ `Int`. Bei der Deklaration eines Bindings kommen keine Standardwerte zum Einsatz. Stattdessen weist man die Werte immer als Parameter zu. So erreichen wir die eben beschriebene Flexibilität: Die `rating`-Property kann so eine Bewertung für ein Buch, einen Film, eine App oder eben etwas gänzlich anderes erhalten.

Übrigens: Die Implementierung innerhalb der `body`-Property müssen Sie zu diesem Zeitpunkt noch nicht verstehen. Wichtig ist lediglich zu wissen, dass RatingView dank des Einsatzes von Binding einen änderbaren Status besitzt, dessen Wert irgendwo außerhalb der View gespeichert ist.

Listing 21.4 Einsatz von Binding

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

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

**Bild 21.5**

Die RatingView soll zum Setzen von Bewertungen jeder Art dienen.

Möchte man nun eine RatingView-Instanz erzeugen, muss man einen entsprechenden Wert für den Binding-Parameter übergeben. Hierbei gibt es zwei essenzielle Dinge zu beachten:

- Der Wert, den man einer Binding-Property übergibt, muss auf einer Source of Truth basieren (dazu mehr in Kapitel 26, „Status“).
- Man übergibt einer Binding-Property nicht den konkreten Wert, sondern einen *Verweis* darauf.

Letzteres ist entscheidend. Der eigentliche Wert ist an einer anderen Stelle gespeichert (eben einer Source of Truth) und eine Binding-Property *verweist* darauf. Ändert man so den Wert über das Binding (also beispielsweise innerhalb von RatingView), ändert man in diesem Zuge auch automatisch den ursprünglichen Wert innerhalb der Source of Truth.

Dieser Umstand spielt auch syntaktisch eine wichtige Rolle. Um nämlich aus einer Property heraus einen Verweis zu generieren und nur diesen Verweis (sprich das Binding) weiterzugeben (und nicht den konkreten Wert der Property), nutzt man das `$`-Präfix.

Ein konkretes Beispiel dazu finden Sie in Listing 21.5. Die darin deklarierte MyAppView soll zur Bewertung einer konkreten App dienen. Dazu definiert die View eine State-Property namens `rating`, die diese Bewertung speichert.

MyAppView bindet zum Bewerten der App die zuvor generierte RatingView ein und übergibt ihr einen *Verweis* auf die eigene State-Property (siehe das vorangestellte `$`-Zeichen bei der Übergabe des `rating`-Parameters). Kommt es nun zu einer Änderung der Bewertung aus der RatingView heraus, ändert das automatisch den Wert der State-Property von MyAppView.

Listing 21.5 Übergabe eines Bindings

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

    var body: some View {
        VStack {
            Text("My App")
            RatingView(rating: $rating)
        }
    }
}
```

Das gezeigte Prinzip spielt eine enorm wichtige Rolle in SwiftUI und kommt in vielen View-Elementen zum Einsatz. Detaillierte Informationen zum Status und dessen Funktionsweise finden Sie in Kapitel 26, „Status“.

■ 21.8 Anpassung von Views mittels Modifier

Der Status entscheidet über die Aktualisierung von Ansichten in SwiftUI. Ein weiteres wichtiges Element in diesem Kontext sind die sogenannten *Modifier*. Mit ihrer Hilfe lassen sich Views nach der Initialisierung weiter anpassen.

Betrachten wir dazu zunächst einmal ein konkretes Beispiel. In Listing 21.6 erfolgt die Umsetzung einer Textansicht, die den Inhalt „Hallo SwiftUI!“ besitzt. Die entsprechende Darstellung der View unter macOS ist in Bild 21.6 zu sehen.

Listing 21.6 Umsetzung eines einfachen Textes

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



Bild 21.6 Die Darstellung des Textes erfolgt ohne sonstige Formatierungen und Anpassungen.

Um die Darstellung dieses Textes nun weiter anzupassen, kommen die eben erwähnten Modifier zum Einsatz. Ein Modifier ist zunächst nichts anderes als eine Methode, die man auf eine SwiftUI-View aufruft. Der Modifier führt dann eine spezifische Anpassung an der View durch und liefert als Ergebnis eine passend aktualisierte neue Ansicht zurück.

Einer dieser Modifier ist `font(_ :)`. Mit seiner Hilfe kann man die grundlegende Darstellung eines Textes beeinflussen und seine Größe verändern. Dazu benötigt der Modifier einen passenden Parameter. Zur Auswahl stehen unter anderem `title` (für große Titel), `headline` (für fett formatierte Überschriften) oder `footnote` (für Fußzeilen).

Zwei andere Modifier sind `bold()` und `italic()`. Sie kommen gänzlich ohne die Angabe zusätzlicher Parameter aus und formatieren einen Text fett beziehungsweise setzen ihn kursiv.

In Listing 21.7 wende ich all die genannten Modifier auf den zuvor erstellten Text an. Dazu rufe ich sie nacheinander auf der Text-View auf. Das Ergebnis dieses Codes ist in Bild 21.7 zu sehen.

Listing 21.7 Anpassung einer View mittels Modifier

```
struct ContentView: View {
    var body: some View {
        Text("Hallo SwiftUI!")
            .font(.title)
            .bold()
            .italic()
    }
}
```



Bild 21.7 Mithilfe von Modifiern lässt sich das Erscheinungsbild von Views „modifizieren“.

Interessant ist – neben der allgemeinen Funktionalität der Modifier – der Aufbau des Codes in Listing 21.7. So fügt man Modifier typischerweise in einer neuen Zeile unterhalb der betreffenden View ein. Bauen mehrere Modifier aufeinander auf, fügt man sie nacheinander in den nachfolgenden Zeilen ein.

Dieses Vorgehen sorgt für eine gewisse Übersicht, da jeder Modifier einzeln zu erkennen ist. Zudem geht klar hervor, auf welche View sich ein Modifier bezieht.

SwiftUI besitzt eine Vielzahl solcher Modifier für die verschiedenen View-Elemente. In den folgenden Kapiteln werden Sie einige davon noch detailliert kennenlernen.

Wichtigkeit der Modifier-Reihenfolge

Die Reihenfolge, in der man Modifier auf einer SwiftUI-View aufruft, spielt mitunter eine wichtige Rolle. Erläutern möchte ich das direkt anhand eines Beispiels.

In Listing 21.8 kommt es erneut zur Umsetzung einer Textansicht. Zusätzlich kommen zwei Modifier zum Einsatz: `padding()` fügt einen Abstand zu allen Seiten der Textansicht ein. Der Aufruf von `border(_:)` legt anschließend einen schwarzen Rahmen um die View. Das Ergebnis zeigt Bild 21.8.

Listing 21.8 Setzen eines Abstands und eines Rahmens um einen Text

```
struct ContentView: View {
    var body: some View {
        Text("Hello, World!")
            .padding()
            .border(Color.black)
    }
}
```

**Bild 21.8**

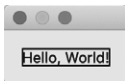
Der Text verfügt über einen Abstand zu allen Seiten und über einen Rahmen.

Das Ergebnis dürfte wenig überraschen; es zeigt eben einen Text mit einem Rahmen. Doch die Ansicht wäre eine gänzlich andere, würde man zuerst den `border(_:)`- und anschließend den `padding()`-Modifier auf der Textansicht aufrufen, so wie es in Listing 21.9 der Fall ist.

Der Grund ist die *Reihenfolge* der Modifier-Aufrufe. So wird in Listing 21.9 *erst* der Rahmen um den Text gesetzt. Da die Textansicht keinerlei Abstände zu den Seiten besitzt, legt sich der Rahmen direkt um den Text. Erst danach kommen die Abstände zu allen Seiten hinzu.

Listing 21.9 Setzen eines Rahmens, gefolgt von einem Abstand

```
struct ContentView: View {
    var body: some View {
        Text("Hello, World!")
            .border(Color.black)
            .padding()
    }
}
```

**Bild 21.9**

Abhängig von der Reihenfolge der Modifier-Aufrufe kann das Ergebnis variieren.

Dieses Verhalten wird umso deutlicher, wenn man am Ende von Listing 21.9 einen weiteren Rahmen ergänzt (siehe Listing 21.10). Dieser zweite Rahmen erscheint erst, nachdem der erste Rahmen gesetzt und anschließend noch Abstände der Textansicht hinzugefügt wurden. Der zweite Rahmen umschließt so einen deutlich größeren Bereich, der die zuvor hinzugefügten Abstände berücksichtigt (siehe Bild 21.10).

Listing 21.10 Ergänzen eines weiteren Rahmens

```
struct ContentView: View {
    var body: some View {
        Text("Hello, World!")
            .border(Color.black)
            .padding()
            .border(Color.black)
    }
}
```

**Bild 21.10**

Der zweite Rahmen macht deutlich, wie sich die Reihenfolge der Modifier-Aufrufe auswirkt.

Ein Modifier greift demnach immer in Bezug auf die zuvor erzeugte View und liefert anschließend eine neue Ansicht zurück. Auf Basis von Listing 21.10 hat das folgenden Ablauf zur Folge:

1. Erstellen der Textansicht.
2. Setzen eines Rahmens direkt um den Text.
3. Hinzufügen von Abständen zur Textansicht (inklusive Rahmen).
4. Ergänzen eines weiteren Rahmens um die gesamte Ansicht (inklusive Abstände).

In Szenarien wie dem hier gezeigten ist die Modifier-Reihenfolge enorm wichtig. Abhängig davon, welches Ergebnis Sie erzielen möchten, müssen Sie dieses Verhalten berücksichtigen.

Weitere Details zu den verfügbaren Modifiern und ihrem Einsatz erhalten Sie in den kommenden Kapiteln.

■ 21.9 Gruppierung von Views mittels Containern

Ein grundlegendes View-Element in SwiftUI sind die *Container*. Mit ihrer Hilfe fasst man verschiedene Views zu einer Einheit zusammen.

SwiftUI bietet diverse Möglichkeiten, Views mittels Container zusammenzufassen. Eine dieser Möglichkeiten basiert auf *Stacks* und kommt sehr häufig zum Einsatz. Ein Stack kümmert sich darum, die in ihm enthaltenen Views passend nebeneinander, untereinander oder hintereinander anzuordnen. Dazu stellt SwiftUI entsprechend unterschiedliche Stack-Typen namens *HStack*, *VStack* und *ZStack* bereit.

Eine Übersicht über die verschiedenen Container-Views in SwiftUI erhalten Sie in Kapitel 23, „View-Layout“. An dieser Stelle möchte ich Ihnen dennoch bereits einmal die grundlegende Funktionsweise erläutern und die Syntax von Containern vorstellen. Das ist wichtig, um bei der Erläuterung der verfügbaren Views innerhalb des SwiftUI-Frameworks gleich passende Beispiele vorstellen zu können.

Standardmäßig erhält jeder Container als Parameter wenigstens ein Closure. Dieses Closure enthält alle Views, die Teil des jeweiligen Containers sind. Der Container kümmert sich dann darum, diese Views passend darzustellen und anzuordnen. Ein *VStack* beispielsweise ordnet die Views vertikal untereinander an, ein *HStack* hingegen horizontal nebeneinander.

Ein Beispiel zeigt Listing 21.11. Darin erfolgt die Deklaration zweier Views (*SomeVStack* und *SomeHStack*), die sich beide aus einem als Titel formatierten Text sowie einem Bild zusammensetzen. Sie unterscheiden sich lediglich in der Art des Containers, der den Text sowie das Bild umschließt; in ersterem Fall handelt es sich um einen *VStack*, im zweiten um einen *HStack*.

Listing 21.11 Einsatz von Container-Views

```

struct SomeVStack: View {
    var body: some View {
        VStack {
            Text("Vacation")
                .font(.largeTitle)
            Image("Vacation")
        }
    }
}

struct SomeHStack: View {
    var body: some View {
        HStack {
            Text("Vacation")
                .font(.largeTitle)
            Image("Vacation")
        }
    }
}

```

Das führt dazu, dass `SomeVStack` und `SomeHStack` die innerhalb des jeweiligen Containers enthaltenen Views unterschiedlich darstellen. Bild 21.11 stellt die beiden Ansichten gegenüber.

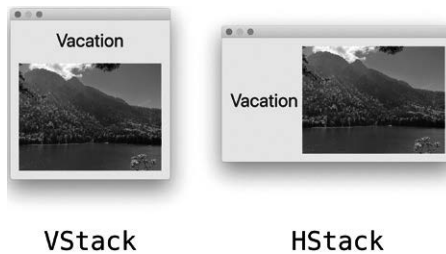


Bild 21.11 Container fassen Views zusammen und beeinflussen deren Anordnung und Darstellung.

Im Laufe der folgenden Kapitel werden Sie noch weitere Container-Views kennenlernen. Für den Moment ist nur wichtig, dass Sie das grundlegende Prinzip der Container verinnerlicht haben und das gezeigte Code-Beispiel nachvollziehen können. Die weiteren Details sowie zusätzliche Informationen folgen später. ☺

■ 21.10 Praxis: Unsere erste App

Bevor es mit den Details zur Entwicklung einer App weitergeht, erstellen wir in diesem Abschnitt gemeinsam ein erstes Xcode-Projekt und werfen einen Blick auf die darin enthaltenen Elemente.

Starten Sie zu diesem Zweck Xcode und wählen Sie im Begrüßungsfenster den Punkt *Create a new Xcode project* (siehe Bild 21.12).

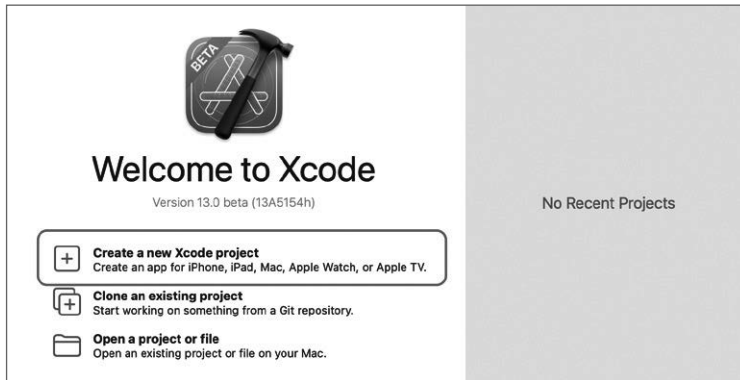


Bild 21.12 Über den gekennzeichneten Button erstellen Sie ein neues Xcode-Projekt.

Im Anschluss öffnet sich ein neues Fenster, über das Ihnen verschiedene Vorlagen zur Erstellung eines neuen Projekts zur Verfügung stehen. Für dieses Beispiel wollen wir eine einfache App für iOS umsetzen, weshalb Sie zunächst am oberen Rand des Fensters den gleichnamigen Reiter *iOS* auswählen. Anschließend selektieren Sie im Abschnitt *Application* die Vorlage mit dem Titel *App* (siehe Bild 21.13).

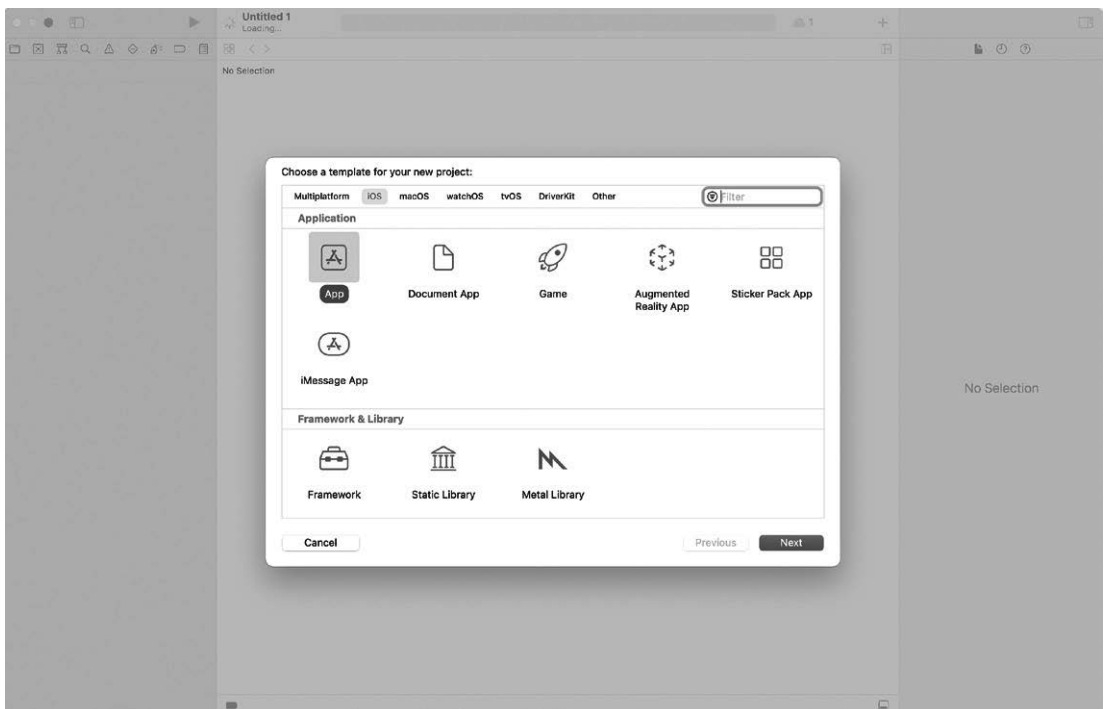


Bild 21.13 Über die verschiedenen Vorlagen legen Sie die Art des neuen Xcode-Projekts fest.

Nach einem Klick auf die Schaltfläche *Next* fordert Xcode Sie auf, grundlegende Informationen zu Ihrem Projekt anzugeben (siehe Bild 21.14). Dazu gehören die folgenden Angaben:

- **Product Name:** Der Name Ihres „Produkts“, das Sie mit dem neuen Xcode-Projekt erstellen möchten. Im Falle einer App entspricht der Product Name typischerweise dem Namen der App. Für dieses Beispiel geben Sie hier schlicht „Hello World“ ein.
- **Team:** Teams verwalten Sie in den Einstellungen von Xcode (siehe hierzu auch Kapitel 16, „Grundlagen, Aufbau und Einstellungen von Xcode“). Haben Sie wenigstens einen Entwickler-Account in Xcode hinterlegt, steht er Ihnen an dieser Stelle via Dropdown-Menü zur Auswahl zur Verfügung. Entsprechend wählen Sie hier den für das neue Projekt passenden Entwickler-Account aus. Steht noch kein Entwickler-Account zur Verfügung oder Sie wollen zu diesem Zeitpunkt noch nicht festlegen, welcher Entwickler-Account zum Einsatz kommen soll, können Sie alternativ auch *None* auswählen und das Team zu einem späteren Zeitpunkt ergänzen.
- **Organization Identifier:** Dieser Identifier dient als eindeutige Kennzeichnung eines Entwicklers oder Unternehmens und entspricht einem umgekehrten Domain-Name. Über ihn legen Sie im Grunde fest, wer für das Projekt verantwortlich ist und wer es entwickelt. Bei Unternehmen käme typischerweise ein String mit dem Aufbau „com.firmenname“ zum Einsatz. Bei Solo-Entwicklern verwendet man standardmäßig eine Kombination aus Land und Name, also beispielsweise „de.thomassillmann“.

Wenn Sie sich zu Beginn eines Projekts noch unsicher über den passenden Organization Identifier sind, ist das kein Problem. Sie können die entsprechende Angabe jederzeit in den Projekteinstellungen ändern. Wirklich relevant wird der Organization Identifier erst dann, wenn Sie eine App verteilen oder veröffentlichen möchten.

- **Bundle Identifier:** Der Bundle Identifier ist eine Zusammensetzung aus Organization Identifier und Product Name. Er dient als eindeutige Kennzeichnung einer App. So besitzt jede App im App Store einen einmaligen Bundle Identifier.

Bei der Projekterstellung können Sie den Bundle Identifier selbst nicht individuell anpassen. Stattdessen fügt Xcode automatisch Organization Identifier und Product Name aneinander und zeigt Ihnen lediglich das entsprechende Ergebnis an. Nach Erstellung des Projekts können Sie aber den Bundle Identifier beliebig anpassen.

- **Interface:** Über dieses Dropdown-Menü bestimmen Sie die Technologie, mit der Sie die Oberflächen und Ansichten in Ihrem neuen Projekt erstellen möchten. Wählen Sie hier die moderne Variante *SwiftUI*.
- **Language:** Hier wählen Sie die Programmiersprache, die zur Entwicklung des neuen Xcode-Projekts zum Einsatz kommen soll. Da Sie das Swift-Handbuch lesen, ist *Swift* hier die richtige Wahl. ☺ Haben Sie zuvor wie beschrieben als Interface *SwiftUI* und für den Lebenszyklus *SwiftUI App* ausgewählt, steht Ihnen bei der Programmiersprache gar keine Alternative zu Swift zur Verfügung.

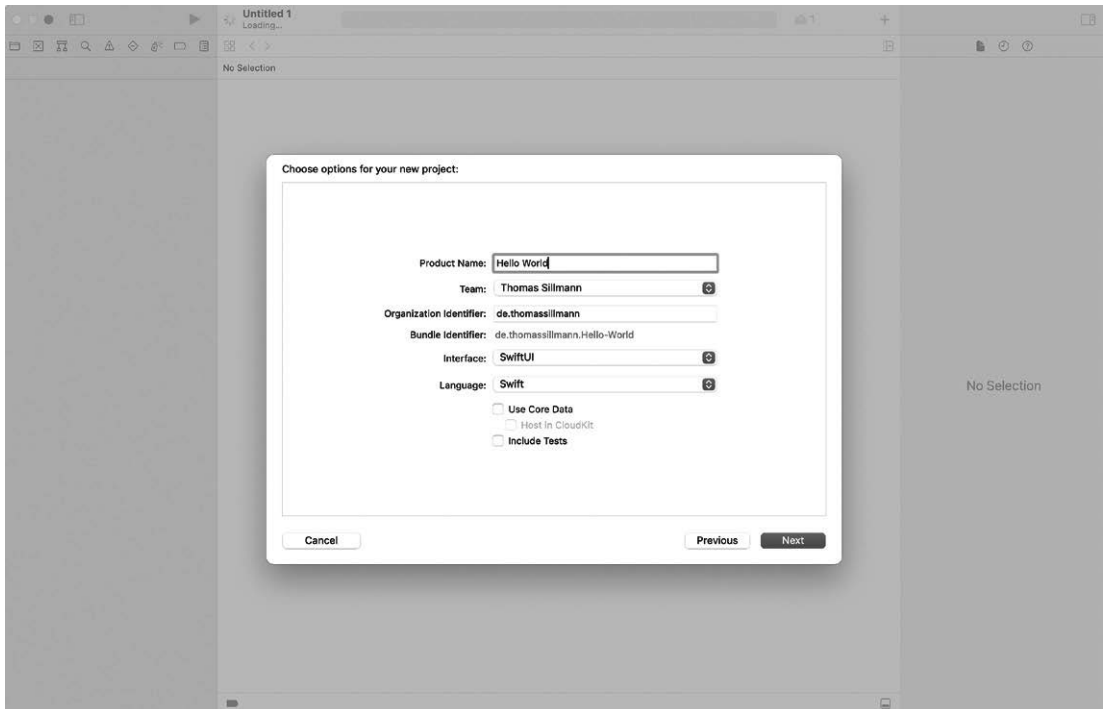


Bild 21.14 Für die erfolgreiche Projekterstellung müssen Sie einige grundlegende Informationen angeben.

Am Ende finden Sie noch diverse Checkboxes, beispielsweise *Use Core Data* oder *Include Tests*. Welche Sie davon sehen, hängt zum Teil von der Projektvorlage ab, die Sie zuvor ausgewählt haben. Für den Moment können Sie diese Checkboxes aber vollständig ignorieren und deaktivieren, so wie auch in Bild 21.14 zu sehen.

Nach einem weiteren Klick auf die *Next*-Schaltfläche wählen Sie abschließend noch einen Speicherort für das neue Projekt aus (siehe Bild 21.15). Mit Klick auf *Create* erzeugt Xcode dann das neue Projekt und öffnet es (siehe Bild 21.16).

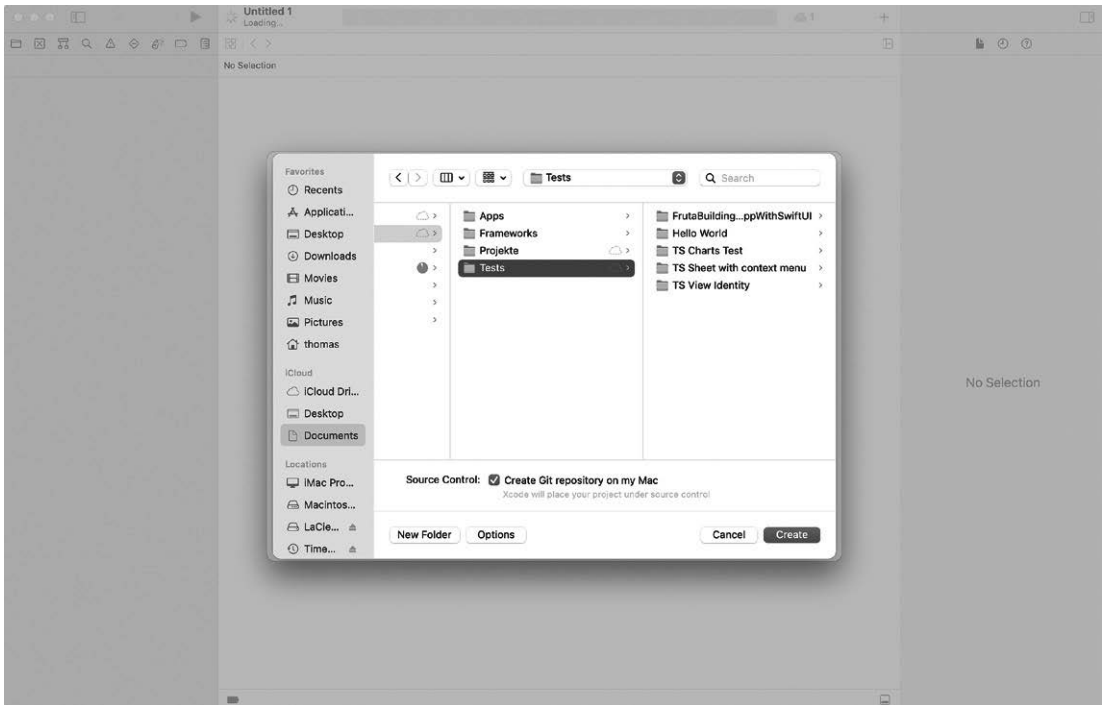


Bild 21.15 Zum Abschluss wählen Sie einen Speicherort für das neue Projekt.



Bild 21.16 Es ist geschafft: Ihr erstes Xcode-Projekt steht Ihnen nun zur Bearbeitung zur Verfügung!

Testen wir doch direkt einmal, wie die App aussieht, die Xcode mithilfe der *App*-Vorlage für uns erzeugt hat. Dazu wählen Sie über die Geräte- und Simulatorauswahl am oberen Rand des Editors einmal einen Simulator aus und klicken anschließend auf die Run-Schaltfläche im linken oberen Bereich (siehe Bild 21.17). Xcode kompiliert daraufhin das Projekt (sprich es baut die App für Sie zusammen) und führt es im Anschluss im gewählten Simulator aus (siehe Bild 21.18).

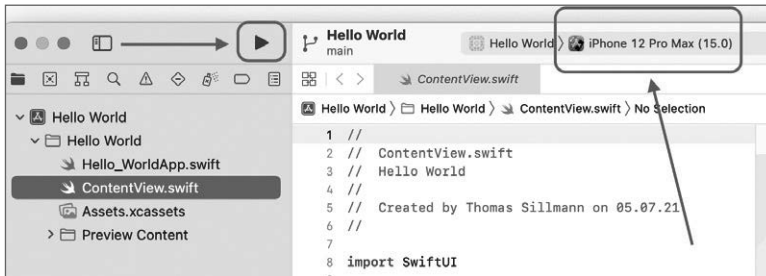


Bild 21.17 Wählen Sie zunächst einen Simulator und betätigen Sie dann die Run-Schaltfläche, um das Projekt auszuführen und zu starten.

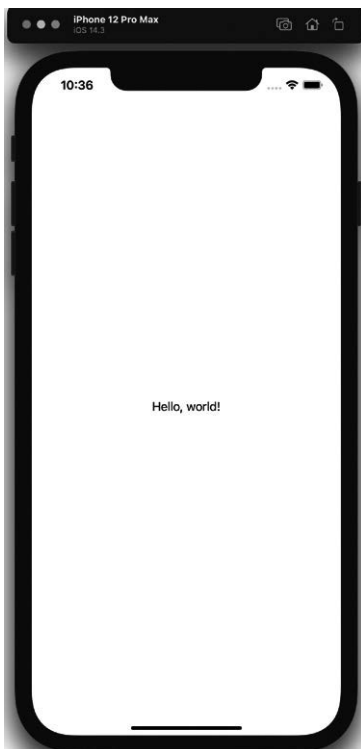


Bild 21.18

Da ist sie, unsere erste (zugegebenermaßen noch recht unscheinbare) App!

Wundern Sie sich übrigens nicht, falls es etwas dauert, bis der Simulator gestartet ist und die App darin ausgeführt wird. Je nach Leistung des Mac kann dieser Vorgang etwas Zeit in Anspruch nehmen.

21.10.1 Bestandteile des neuen Projekts

Werfen wir nun einmal einen Blick auf die wichtigsten Dateien dieses neuen Xcode-Projekts. Sie finden sie innerhalb des linken Abschnitts der Entwicklungsumgebung in der sogenannten Navigator-Ansicht (siehe Bild 21.19).

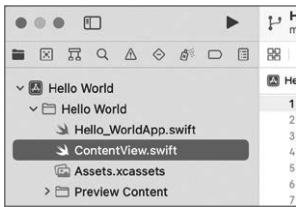


Bild 21.19

In der Navigator-Ansicht von Xcode finden Sie alle Dateien eines Projekts.

Hello_WorldApp.swift

In der *Hello_WorldApp.swift*-Datei finden Sie den Einstiegspunkt Ihrer Anwendung. Der Name der Datei variiert von Projekt zu Projekt und setzt sich aus dem Product Name gefolgt vom Suffix „App“ zusammen.

Der Code, den Sie innerhalb von *Hello_WorldApp.swift* finden, sollte in etwa dem aus Listing 21.12 entsprechen.

Listing 21.12 Code der Hello_WorldApp.swift-Datei

```
import SwiftUI

@main
struct Hello_WorldApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
    }
}
```

Herzstück des Codes ist die Deklaration einer Structure, die konform zum sogenannten App-Protokoll ist. Der Name dieser Structure entspricht dem Namen der Datei.

Das App-Protokoll fordert die Implementierung einer body-Property. Über sie definieren Sie die verschiedenen Programmfenster, die Sie für Ihre Anwendung benötigen. Im einfachsten Fall geben Sie über die body-Property ein einziges solches Fenster zurück.

In Listing 21.12 kommt ein Fenster in Form einer WindowGroup-Instanz zum Einsatz. Hierbei handelt es sich um die einfachste Form eines Programmfensters. Bei der Initialisierung von WindowGroup muss zusätzlich noch eine View angegeben werden, die innerhalb des Fensters dargestellt wird. In Listing 21.12 handelt es sich bei dieser View um eine

ContentView-Instanz. Deren Deklaration finden Sie in der *ContentView.swift*-Datei (siehe nachfolgenden Abschnitt).

Die *Hello_WorldApp*-Structure ist mit dem Schlüsselwort `@main` versehen. Das kennzeichnet diese Structure als Einstiegspunkt der App. Bei Ausführung des Projekts werden entsprechend die in der *body*-Property definierten Fenster mitsamt ihren passenden Views erzeugt und angezeigt.

ContentView.swift

In der *ContentView.swift*-Datei finden Sie die Deklaration einer gleichnamigen SwiftUI-View. Sie setzt sich in unserem Beispielprojekt aus einer einfachen Textansicht mit dem Inhalt „Hello, world!“ zusammen (siehe Listing 21.13).

Listing 21.13 Code der *ContentView.swift*-Datei

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Hello, world!")
            .padding()
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Die Ansicht wird beim Ausführen der App geladen und angezeigt. Dafür sorgt die Implementierung innerhalb der *Hello_WorldApp.swift*-Datei (siehe vorherigen Abschnitt).

Zusätzlich findet sich die Deklaration einer zweiten Structure namens *ContentView_Previews* in der Datei. Diese ist konform zum *PreviewProvider*-Protokoll und unterstützt uns bei der Entwicklung von SwiftUI-Views; mehr dazu erfahren Sie in Abschnitt 21.10.3, „Einsatz der Preview“.

Durch das Zusammenspiel aus *Hello_WorldApp* und *ContentView* kommt es bei Ausführung der App zu jener Ansicht, die uns den Text „Hello, world!“ auf den Bildschirm zaubert (siehe Bild 21.18). *Hello_WorldApp* ist der Startpunkt des Programms, *ContentView* die darin definierte und somit anzuzeigende Ansicht.

Assets.xcassets

In einem Asset Catalog können Sie unter anderem Bilder unterbringen sowie Farben definieren, die innerhalb Ihrer Anwendung wichtig sind. Über passende Befehle im Code können Sie dann direkt auf diese Elemente zugreifen und sie verwenden. Auch das zentrale App Icon eines Programms definieren Sie standardmäßig innerhalb eines solchen Asset Catalogs (siehe Bild 21.20).

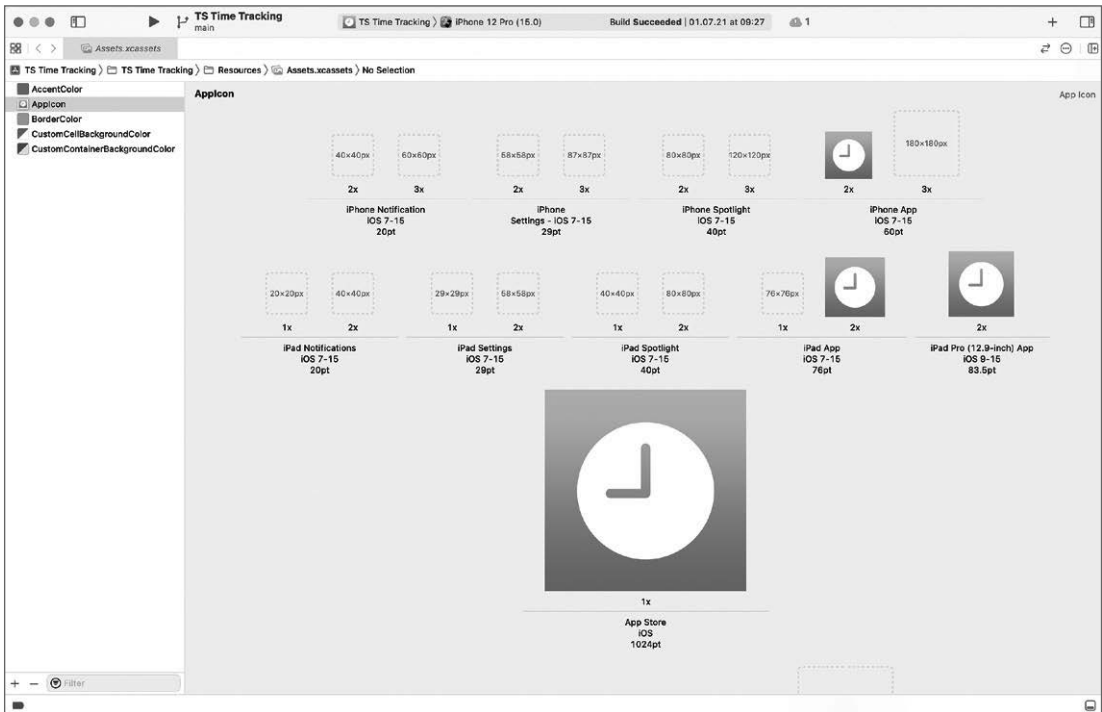


Bild 21.20 Über einen Asset Catalog können Sie unter anderem Bilder und Farben verwalten, die für Ihr Projekt wichtig sind.

Mit der *Assets.xcassets*-Datei erhalten Sie standardmäßig einen Asset Catalog als Teil Ihres Xcode-Projekts. Mehr zu Asset Catalogs und der Arbeit damit erfahren Sie in Kapitel 28, „Weitere Projektkonfigurationen“.

21.10.2 Änderung des Textes

Nehmen wir zum Abschluss der Erstellung unseres ersten Projekts noch eine kleine Änderung daran vor. Statt „Hello, world!“ soll nun der Text „Hello SwiftUI!“ auf dem Bildschirm ausgegeben werden.

Rufen Sie zu diesem Zweck die *ContentView.swift*-Datei auf und ändern Sie schlicht den vorhandenen „Hello, world!“-Text durch „Hello SwiftUI!“, so wie in Listing 21.14 zu sehen. Wenn Sie im Anschluss das Projekt erneut im Simulator starten, sehen Sie, dass Sie erfolgreich die Textausgabe geändert haben (siehe Bild 21.21).

Listing 21.14 Änderung des Textes

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

**Bild 21.21**

Durch eine kleine Änderung innerhalb der `ContentView` wurde die Textausgabe der App angepasst.

21.10.3 Einsatz der Preview

Ein großartiges Feature in Xcode ist die sogenannte *Preview*. Sie ermöglicht es, direkt innerhalb des Editors eine Vorschau von SwiftUI-Views anzuzeigen. Doch damit nicht genug: Auch Interaktionen, wie man sie im Simulator durchführen kann, sind innerhalb der Preview möglich.

Um die Preview nutzen zu können, muss die im Editor geöffnete Swift-Datei über einen Preview-Provider verfügen. In unserem Beispielpjekt findet sich bereits ein solcher innerhalb der `ContentView.swift`-Datei (siehe Listing 21.13). Dieser Preview-Provider definiert, welche SwiftUI-View in der Preview dargestellt wird. Im Beispiel aus Listing 21.13 zeigt die Preview demnach eine Vorschau von `ContentView` an.

Öffnet man eine Swift-Datei, die über wenigstens einen solchen Preview-Provider verfügt, teilt sich der Editor standardmäßig in zwei Bereiche (siehe Bild 21.22). Links befindet sich der Code, rechts die Preview (auch wenn dort zunächst noch nichts zu sehen ist). Sollte nur der Code-Bereich sichtbar sein, klicken Sie oben rechts im Editor auf die *Adjust Editor Options*-Schaltfläche und aktivieren dort die *Canvas*-Option (siehe Bild 21.23).

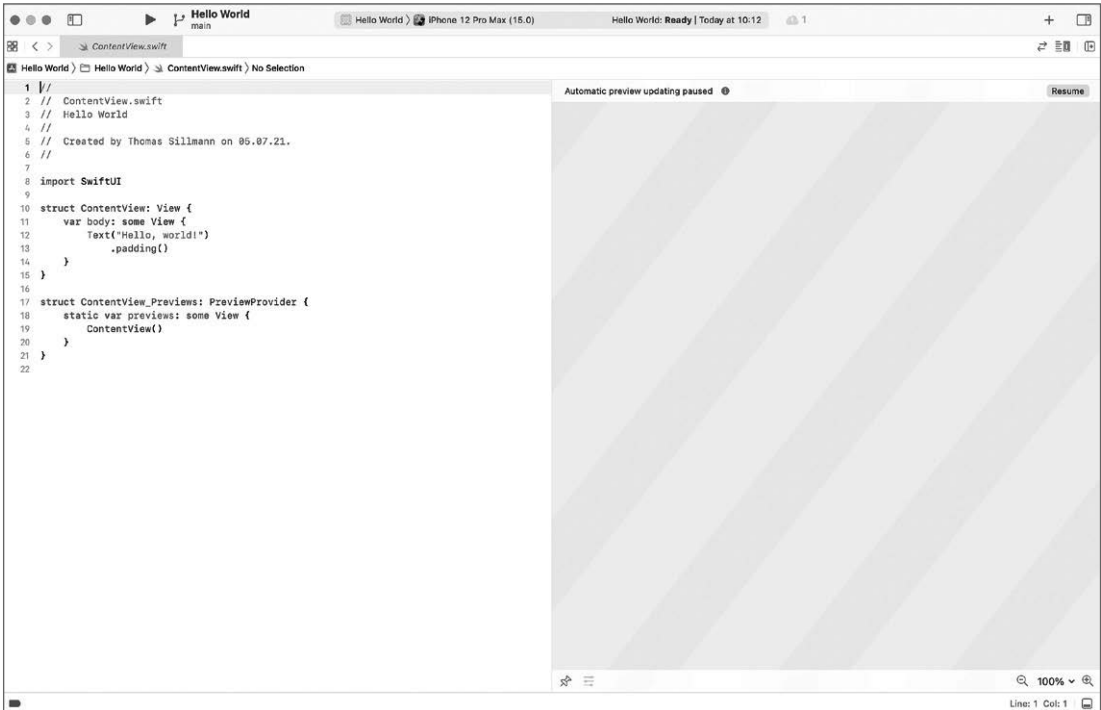


Bild 21.22 Der Preview-Bereich befindet sich rechts neben der Code-Ansicht.

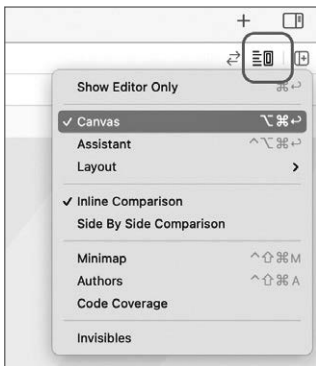


Bild 21.23

Sollte die Preview nicht angezeigt werden, aktivieren Sie sie über die Canvas-Option in den Editor-Einstellungen.

Um die Preview nun nutzen zu können, muss sie zunächst per Klick auf die *Resume*-Schaltfläche oben rechts aktiviert werden. Xcode kompiliert daraufhin das Projekt und startet im Hintergrund die Vorschau. Als Device kommt hierbei dasjenige zum Einsatz, das als Simulator ausgewählt ist.

Sobald der Vorgang abgeschlossen ist, sehen Sie direkt in Xcode, wie die für die Vorschau konfigurierte View aussieht; und das, ohne Ihr Projekt dafür im Simulator ausführen zu müssen (siehe Bild 21.24).

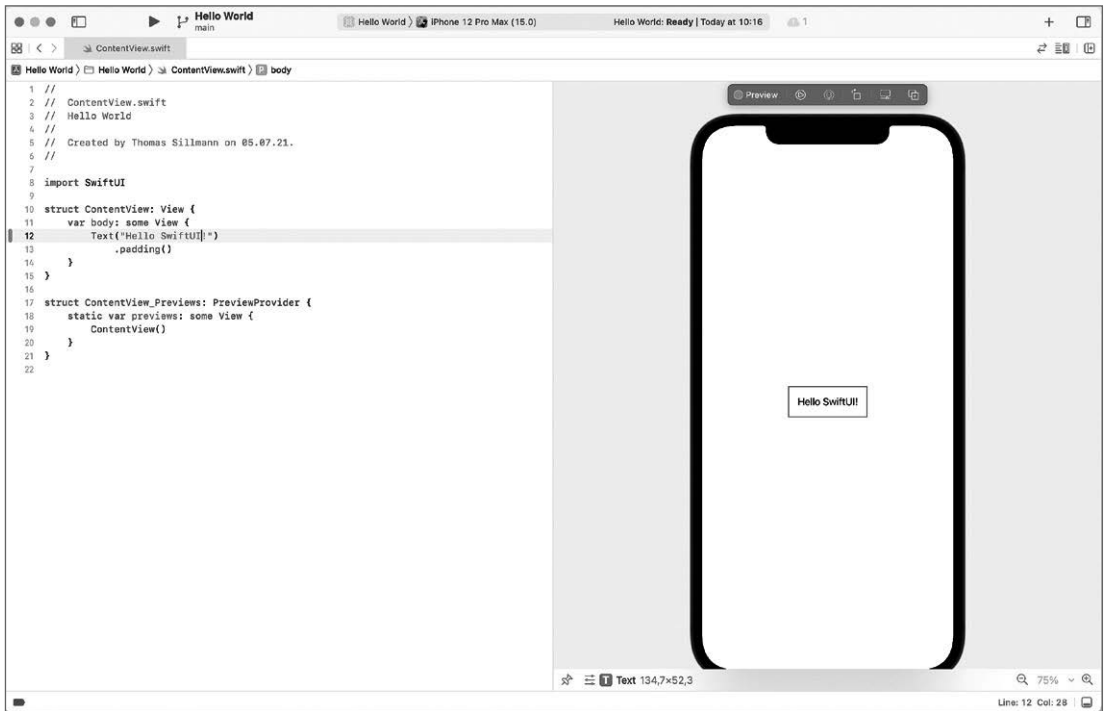


Bild 21.24 Die Preview gibt direkt im Editor das Aussehen einer SwiftUI-View wieder.

Das Beste ist: Wenn Sie nun Änderungen an der dargestellten View durchführen, werden diese direkt in der Preview übernommen. So können Sie sofort sehen, wie sich beispielsweise Farbänderungen oder das Hinzufügen eines Bildes auf Ihre View auswirken.

Unerwähnt lassen möchte ich an dieser Stelle aber nicht, dass diese automatische Aktualisierung nicht immer funktioniert. Wenn Sie beispielsweise etwas an den Model-Daten in Ihrer App oder auch in einer View verändern (indem Sie zum Beispiel eine neue Property ergänzen), greift die automatische Aktualisierung nicht. Grund ist, dass Xcode das gesamte Projekt dann kompilieren muss, um auf jene Daten zugreifen zu können. Erst dann lässt sich die Preview wieder verwenden.

Tritt dieser Fall ein, erscheint oberhalb der Preview die Meldung „Automatic preview updating paused“ (siehe Bild 21.25). Es ist dann notwendig, zunächst erneut die *Resume*-Schaltfläche zu betätigen. Im Anschluss zeigt die Vorschau wieder den aktuellen Zustand der View an.

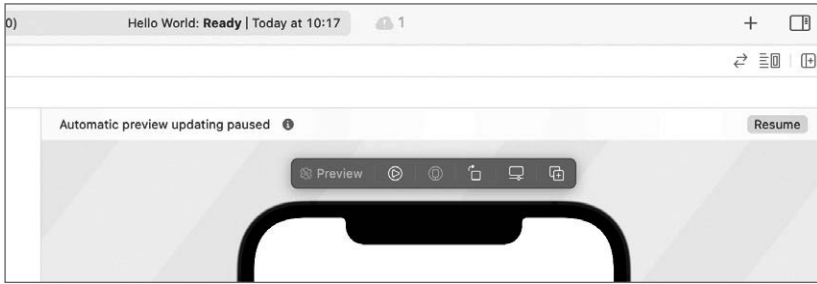


Bild 21.25 Bei Pausierung der Preview muss diese zunächst mittels Klick auf „Resume“ wieder aktiviert werden.

Mehr zur Preview und zur Arbeit damit erfahren Sie in Kapitel 29, „Preview und Library“.

Index

Symbole

\$-Syntax 690
@objc 263
@propertyWrapper 163
.swift 313

A

Abfrage 31
Access Control 314
Access Level 314
– explizite Zuweisung 318
– implizite Zuweisung 318
Accessibility 817
actor 311
Actor 308, 310
Alert 675
Analyze 377
Ansicht 423
Any 97, 277
AnyObject 97, 277
App 426
– Aufbau 425
– Bestandteile 423
App Bundle 858
App Icon 771
App ID 828
AppKit 422
Apple Developer Account 824
Apple Developer Enterprise Program 860
Apple Developer Portal 824
Apple Developer Program 824
Apple Script 398
AppStorage 719, 731
App Store Connect 845
App Store Review Guidelines 852

ARC 200, 223
Archive 850
Argument Label 105
Array 17, 56
– mutable 59
– Shorthand Syntax 56
as! 279
as? 279
Asset Catalog 769
associatedtype 301
Associated Type 300
Associated Values 129
async 305
Attributes Inspector 781
Automatic Reference Counting 200, 223
await 306

B

Barrierefreiheit 452
Basisklasse 206
Binding 688
– Konstante 696
Bool 17, 49
Bot 361
Branch 786
break 38, 42
Breakpoint 395
– Konfiguration 398
Breakpoint Navigator 345, 399
Build Configuration 363
Build Phases 371
Build Rules 372
Build Settings 364
Bundle Identifier 331
Business Model 855
– Freemium Model 856

- Free Model 855
- Paid Model 857
- Paymium Model 857
- Subscription Model 856
- Button 473
 - Rolle 475
 - Style 475

C

- Character 51
- Chris Lattner 3
- class 261
- Class 140
- Class-only-Protokoll 261
- Class-Protocol 261
- Clean Build 415
- Clone 786
- Closure 117
 - Autoclosure 122
 - Default Value 118
 - Function Type 118
 - Implicit Return 120
 - Shorthand Argument Names 121
 - Trailing Closure 121
- Code Review 347, 797
- Code Signing 842
 - Automatic 842
 - Manual 843
- Code Snippets 409
- Color 457, 680
- ColorPicker 503
- Commit 786
- Components 360
- Computed Variable 173
- Configurations 363
- Confirmation Dialog 678
- Container 435
- continue 41
- Control Transfer Statement 41
- convenience 250
 - Schlüsselwort 196
- Convenience_INITIALIZER 194
- Core Data 732
 - Elemente 733
 - Entity 738
 - FetchedResults 753
 - FetchRequest 752
 - Funktionsweise 732
 - NSManagedObject 734
 - NSManagedObjectContext 734

- NSManagedObjectModel 733
- NSPersistentContainer 735
- NSPersistentStore 733
- NSPersistentStoreCoordinator 734
- Operationen 747
- Relationship 740
- Stack 735
- SwiftUI 749
- Cross-Platform 757
- CSR 830

D

- Daten 423
- Datenhaltung 729
- DatePicker 499
 - Komponenten 500
 - Style 502
- DatePickerComponents 500
- Debug Area 353, 393
- Debugging 393
- Debug Navigator 345
- Default_INITIALIZER 185
- deinit 200
- Deinitialisierung 200
- Deinitializer 201
- Deklarative Syntax 424
- Delegate 267
- Delegation 254, 267
- Deployment Target 363
- Derived Value 721
- Designated_INITIALIZER 194
- Design Pattern 267
- Developer ID Certificate 859
- Dictionary 17, 72
 - Shorthand Syntax 73
- Discard 786
- DisclosureGroup 601
- Divider 609
- do-catch 285
- Double 16, 48
- Downcasting 277

E

- Edge 665
- EditButton 476
- Edit-Mode 476
- Editor 346
 - Options 348
- Entity 738
 - Relationship 740

- Entwicklerzertifikat 828
- Enumeration 125
- Environment 713
- EnvironmentKey 726
- EnvironmentObject 707
- EnvironmentValues 713
- Error 281
- Error Handling 281
- Exception Breakpoint 400
- Explicit App ID 834
- Extension 237, 256
 - Computed Property 237
 - Initializer 239
 - Methode 238
 - Nested Type 242
 - Subscript 241
- F**
- fallthrough 37
- Fetch 786
- FetchResults 753
- FetchRequest 752
- FileMerge 786
- fileprivate 315, 317
- File-private Access 315, 317
- final 209, 251
- Find Navigator 344
- FIXME 413
- Fließkommazahl 16
- Float 16, 48
- Font 450
- Font.Design 453
- Font.Weight 454
- for 29
- Forced Unwrapping 85
- ForEach 552
 - Daten 554
 - Range 552
- for-in 27
- Form 579
- Foundation 157, 314
- Foundation-Framework 263
- Frameworks 5, 313
 - AppKit 422
 - SwiftUI 421
 - UIKit 422
 - Uniform Type Identifiers 561
 - WatchKit 422
- Free Model 855
- Freemium Model 856

- func 103
- Function Type 113
- Funktion 103
 - globale 174
 - lokale 174
 - Name 105
 - Rückgabewert 111
 - verschachtelte 116

G

- Ganzzahl 16
- Gauge 512
 - Style 513
- Generic 293
- Generic Function 294
- Generic Type 297
- Geschäftsmodell 855
- Git 785
- Grids 563
 - LazyHGrid 563
 - LazyVGrid 563
- GridItem 563, 574
- Group 584
- GroupBox 589
- guard 39

H

- Hashable 269
- HSplitView 648
- HStack 517
 - Ausrichtung 518

I

- IDE 327
- Identifiable 271, 536
- if 31
- Image 466
 - Größe 470
- Implicit Return 120
- Implicitly Assigned Raw Value 133
- Implicitly Unwrapped Optional 88
- import 263, 313
- Index 52
- Info.plist 370
- init! 199
- init? 197
- Initialisierung 135, 185, 210
- Initialization Parameters 189
- Initializer 185, 210
 - Convenience Initializer 194

- Default Initializer 185
- Deinitializer 201
- Designated Initializer 194
- Failable Initializer 197
- Memberwise Initializer 186, 323
- Required Initializer 200, 221, 322
- Initializer Delegation 193
- Reference Type 194
- Value Type 193
- Inspectors 350
 - File Inspector 351
 - History Inspector 352
 - Quick Help Inspector 352, 383
- Instance Methods 175
- Instanz 134
- Instruments 375, 404
- Int 16, 47
- Int8 47
- Int16 47
- Int32 47
- Int64 47
- Integer
 - Wertebereich 47
- Integrated Development Environment 327
- Interface 332
- internal 315, 317
- Internal Access 315, 317
- Interval Matching 39
- iOS 145, 157
- IPA 391
- is
 - Type Check Operator 278
- Issue Navigator 344

J

Jump Bar 413

K

Key Bindings 358
 Key-Path 271
 Key-Value Pairs 72
 Klasse 140
 Klassenprotokoll 261
 Konsole 394
 Konstante 20

- lokale 174

L

Label 506

- Style 508

Labeled Statement 44
 Language 332
 lazy 153, 174
 Lazy Stacks 527
 LazyHGrid 563
 LazyHStack 528
 LazyVGrid 563
 LazyVStack 528
 let 20
 Library 341, 779
 List 531

- Style 550

 Localizable.strings 763
 LocalizedStringKey 768

M

macOS 145, 157
 MARK 413
 Mehrfachvererbung 261
 Mehrsprachigkeit 762
 Memberwise Initializer 138, 142, 186, 323
 Merge 797
 Methode 138, 175

- Instanzmethode 175
- Typmethode 178

 Model 423
 Modifier 432

- Reihenfolge 433

 Module 313
 mutating 177, 239, 248

N

Navigation 611
 NavigationLink 612
 NavigationView 612

- Detailansicht 618
- Style 620
- Titel 623

 Navigator 342

- Breakpoint Navigator 345, 399
- Debug Navigator 345, 400
- Find Navigator 344
- Issue Navigator 344
- Project Navigator 342
- Report Navigator 346
- Source Control Navigator 342
- Symbol Navigator 343
- Test Navigator 345

 Nebenläufigkeit 305

- Nested Functions 116
- Nested Type 235, 242
- nil 75, 83, 279
- NSItemProvider 561
- NSLocalizedString 766
- NSManagedObject 734
- NSManagedObjectContext 734
- NSManagedObjectModel 733
- NSPersistentContainer 735
- NSPersistentStore 733
- NSPersistentStoreCoordinator 734

O

- objectWillChange 704
- ObservableObject 698
- ObservedObject 697
- onAppear 681
- onDisappear 681
- open 316f.
- Open Access 316f.
- Open Quickly 411
- Operator
 - logischer 35
 - Ternary Conditional 33
- optional 263
- Optional 83, 133
 - entpacken 83f.
- Optional Binding 86, 265
- Optional Chaining 89
- Organization Identifier 331
- Organizer 391
- OutlineGroup 596
- override 206

P

- Page Indicator 643
- Paid Model 857
- Parallelität 305
- Parameter 104
 - Default Value 108
 - In-Out-Parameter 110
 - Variadic Parameter 109
- Parameter Name 105
- Pasteboard 477
- PasteButton 477
- Paymium Model 857
- Picker 495
 - Style 498
- Placeholder Type 295
- Popover 660

- Preview 328, 773
 - Funktionsweise 775
 - Konfiguration 777
- PreviewProvider 775
- print 17
- private 314, 317
- Private Access 314, 317
- Product Name 331
- Profile 377
- ProgressView 509
 - Maximalbereich 510
 - Style 511
- Project Navigator 342
- Projekt 329
 - Einstellungen 363
- Property 138, 149, 685
 - Computed Property 155
 - Instance Property 170
 - Lazy Stored Property 152
 - Read-Only Computed Property 159
 - Shorthand Getter 158
 - Shorthand Setter 158
 - Stored Property 150
 - Type Property 170
- Property Default Value 191
- Property Observer 160
- Property Wrapper 163
- protocol 243
- Protocol Composition 265
- Protokoll 242
 - Class-only 261
 - Hashable 269
 - Identifiable 271
 - Initializer 250
 - Methode 246
 - Property 244
 - Subscript 249
 - Typ 254
 - Vererbung 260
- Protokolle
 - View 426
- Provisioning Profile 829
- public 316f.
- Public Access 316f.
- Published 703
- Pull 786, 792
- Punktnotation 46
- Push 786, 792

Q

Quick Help Inspector 352, 383
Quick Look 396

R

Race Condition 308, 310
Raw Value 131
– Implicitly Assigned Raw Value 133
Read-Only Computed Property 159
Refactoring 402
Reference Type 98, 140
– Initializer Delegation 194
Related Items 412
repeat-while 30
Report Navigator 346
Repository 785
required 200, 221, 251
Required Initializer 200, 221
– Access Level 322
Resource Tags 368
return 112
Rückgabewert 111

S

Scene 426
SceneStorage 717
Scheme 373
– erstellen 375
– verwalten 376
Schleife 27
Schlüsselbundverwaltung 830
Schlüssel-Wert-Paar 72
ScrollView 593
– Richtung 594
Section 581
SecureField 462
– Style 462
self 137, 146, 176, 178
Set 64
SF Symbols 466
Sheet 649
Shell Command 398
Shorthand Argument Names 121
Shorthand Getter 158
Shorthand Setter 158
Signed Integer 47
Signing & Capabilities 367
Simulator 387

Slider 484
– Aussehen 489
– Wertebereich 485
Snippets Library 409
Source Control 785
Source Control Navigator 342, 796
Source File 313
Source of Truth 721
Spacer 606
Speicherverwaltung 223
sqrt() 157
Stacks 517
– HStack 517
– Lazy 527
– LazyHStack 528
– LazyVStack 528
– VStack 521
– ZStack 525
State 687
StateObject 706
static 170, 178
Status 427, 683
– Best Practices 724
– Binding 688
– Derived Value 721
– EnvironmentObject 707
– NSManagedObject 749
– ObservedObject 697
– Property 685
– Source of Truth 721
– State 687
– StateObject 706
Stepper 490
Stored Constant 174
Stored Variable 172
String 17, 49
String Immutability 50
String Interpolation 22, 55
String Mutability 50
Strong Reference 228
Strong Reference Cycle 226
struct 134
Structure 134
Subklasse 205
subscript
– Schlüsselwort 179
Subscript 52, 179
– Subscript Overloading 183
Subscription Model 856

- Subscript Overloading 183
- Subversion 785
- super 209
- Superklasse 205
- Swift 3
- Swift Packages 365
- Swift Playgrounds 7, 9
- Swift Standard Library 15
- SwiftUI 421
 - Syntax 424
- Swipe-to-delete-Geste 557
- switch 36
 - Compound Case 38
 - Explicit Fallthrough 37
 - Implicit Fallthrough 37
- Symbol Navigator 343

T

- TabView 640
- Target 332, 365, 757
 - Zuweisung 760
- Target Membership 351
- Tastaturkurzbefehle 358
- Team 331
- Ternary Conditional Operator 33
- TestFlight 849, 863
- Test Navigator 345
- Text 449
 - Farbe 457
 - Formatierung 456
 - Schriftart 450
 - Übersetzung 458
- TextEditor 463
 - Formatierung 464
- TextField 459
 - Style 460
- Text.FontStyle 453
- throw 283
- throws 282
- TODO 413
- Toggle 479
 - Style 483
- Toolbar 667
 - Position 671
- ToolbarItem 667
- ToolbarItemGroup 667
- ToolbarItemPlacement 671
- try 285f.
- try! 291
- try? 289
- Tuple 64, 78
- tvOS 145, 157
- Two-Phase Initialization 211
- Typ
 - Platzhalter 295
- Type Alias 98
- Type Annotation 22
- Type Cast Operator 279
- Type Casting 97, 279
- Type Check Operator 278
- Type Checking 276, 278
- Type Constraint 300
- Type Inference 23
- Type Method 178
- Type Parameter 297
- Type Property 170
- Typmethode 178
- Typsicherheit 22, 56, 73

U

- Übersetzung 762
- UIKit 314, 422
- UInt 47
- UInt8 47
- UInt16 47
- UInt32 47
- UInt64 47
- UI Recording 819
- UI-Test 815
- UI Testing Bundle 816
- Uniform Type Identifiers 477, 561
- UnitPoint 663
- unowned 232
- Unowned Reference 232, 234
- Unsigned Integer 47
- UserDefaults 729
- UTType 477, 561
- UUID 271

V

- Value Binding 81, 286
- Value Type 98, 140
- var 20
- Variable 20
 - globale 172
 - lokale 172
- Variables View 396
- Vererbung 203, 260
- Versionsverwaltung 785
- View-Events 681

- Views 423, 426
 - Aktualisierung 427
 - Anpassung 432
 - Button 473
 - ColorPicker 503
 - DatePicker 499
 - DisclosureGroup 601
 - Divider 609
 - EditButton 476
 - ForEach 552
 - Form 579
 - Gauge 512
 - Group 584
 - GroupBox 589
 - HSplitView 648
 - HStack 517
 - Image 466
 - Label 506
 - LazyHGrid 563
 - LazyHStack 528
 - LazyVGrid 563
 - LazyVStack 528
 - List 531
 - NavigationLink 612
 - NavigationView 612
 - OutlineGroup 596
 - PasteButton 477
 - Picker 495
 - ProgressView 509
 - ScrollView 593
 - Section 581
 - SecureField 462
 - Slider 484
 - Spacer 606
 - Stepper 490
 - TabView 640
 - Text 449
 - TextEditor 463
 - TextField 459
 - Toggle 479

- VSplitView 648
- VStack 521
- ZStack 525
- Void 114
- VSplitView 648
- VStack 521
 - Ausrichtung 522

W

- Wahrheitswert 17, 49
- WatchKit 422
- watchOS 145, 157
- weak 229
- Weak Reference 228, 234
- where 82, 288
- while 29
- Wildcard App ID 834
- WLAN 390
- Workspace 329
- Worldwide Developers Conference 3
- WWDC 3

X

- Xcode 327
 - Einstellungen 354
- Xcode Server 361
- XCTest 802
- XCTestCase 805
- XCUIApplication 817
- XCUIElement 817
- XCUIElementQuery 818
- XCUIElementTypeQueryProvider 818

Z

- Zeichenkette 17, 49
- ZStack 525
 - Ausrichtung 526
- Zwei-Phasen-Initialisierung 211
- Zwischenablage 477