

HANSER



Leseprobe

zu

Das Swift-Handbuch

von Thomas Sillmann

Print-ISBN: 978-3-446-47639-4

E-Book-ISBN: 978-3-446-47857-2

E-Pub-ISBN: 978-3-446-48032-2

Weitere Informationen und Bestellungen unter

<https://www.hanser-kundencenter.de/fachbuch/artikel/9783446476394>

sowie im Buchhandel

© Carl Hanser Verlag, München

Inhalt

Vorwort	XXV
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 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	11
1.6.2 Apples Developer-Website	11
1.6.3 Swift.org	12
1.6.4 In eigener Sache	13
2 Grundlagen der Programmierung	14
2.1 Grundlegendes	14
2.1.1 Swift Standard Library	14
2.1.2 print	16
2.1.3 Befehle und Semikolons	17
2.1.4 Operatoren	18
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	25
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	31
3.2	Abfragen	32
3.2.1	If	32
3.2.2	Switch	37
3.2.3	Guard	41
3.3	Control Transfer Statements	43
3.3.1	Anstoßen eines neuen Schleifendurchlaufs mit continue	43
3.3.2	Verlassen der kompletten Schleife mit break	43
3.3.3	Labeled Statements	44
4	Typen in Swift	47
4.1	Integer	49
4.2	Fließkommazahlen	50
4.3	Bool	51
4.4	String	51
4.4.1	Erstellen eines Strings	52
4.4.2	Zusammenfügen von Strings	52
4.4.3	Character auslesen	54
4.4.4	Character mittels Index auslesen	54
4.4.5	Character entfernen und hinzufügen	56
4.4.6	Anzahl der Character zählen	58
4.4.7	Präfix und Suffix prüfen	58
4.4.8	String Interpolation	58
4.5	Array	59
4.5.1	Erstellen eines Arrays	60

4.5.2	Zusammenfügen von Arrays	61
4.5.3	Inhalte eines Arrays leeren.	62
4.5.4	Prüfen, ob ein Array leer ist.	62
4.5.5	Anzahl der Elemente eines Arrays zählen	63
4.5.6	Zugriff auf die Elemente eines Arrays	63
4.5.7	Neue Elemente zu einem Array hinzufügen	64
4.5.8	Bestehende Elemente aus einem Array entfernen	65
4.5.9	Bestehende Elemente eines Arrays ersetzen	66
4.5.10	Alle Elemente eines Arrays auslesen und durchlaufen	67
4.6	Set	68
4.6.1	Erstellen eines Sets	68
4.6.2	Inhalte eines bestehenden Sets leeren.	69
4.6.3	Prüfen, ob ein Set leer ist	70
4.6.4	Anzahl der Elemente eines Sets zählen	70
4.6.5	Element zu einem Set hinzufügen	70
4.6.6	Element aus einem Set entfernen	71
4.6.7	Prüfen, ob ein bestimmtes Element in einem Set vorhanden ist ..	71
4.6.8	Alle Elemente eines Sets auslesen und durchlaufen	72
4.6.9	Sets miteinander vergleichen	72
4.6.10	Neue Sets aus bestehenden Sets erstellen.	75
4.7	Dictionary	77
4.7.1	Erstellen eines Dictionaries	77
4.7.2	Prüfen, ob ein Dictionary leer ist	79
4.7.3	Anzahl der Schlüssel-Wert-Paare eines Dictionaries zählen	79
4.7.4	Wert zu einem Schlüssel eines Dictionaries auslesen	79
4.7.5	Neues Schlüssel-Wert-Paar zu Dictionary hinzufügen.	80
4.7.6	Bestehendes Schlüssel-Wert-Paar aus Dictionary entfernen	81
4.7.7	Bestehendes Schlüssel-Wert-Paar aus Dictionary verändern	81
4.7.8	Alle Schlüssel-Wert-Paare eines Dictionaries auslesen und durchlaufen	82
4.8	Tuple	83
4.8.1	Zugriff auf die einzelnen Elemente eines Tuples	84
4.8.2	Tuple und switch.	85
4.9	Optional.	89
4.9.1	Deklaration eines Optionals	89

4.9.2	Zugriff auf den Wert eines Optionals	90
4.9.3	Optional Binding	92
4.9.4	Implicitly Unwrapped Optional	95
4.9.5	Optional Chaining	96
4.9.6	Optional Chaining über mehrere Eigenschaften und Funktionen	101
4.10	Any und AnyObject	106
4.11	Type Alias	106
4.12	Value Type versus Reference Type	107
4.12.1	Reference Types auf Gleichheit prüfen	109
5	Funktionen	111
5.1	Funktionen mit Parametern	112
5.1.1	Argument Labels und Parameter Names	114
5.1.2	Default Value für Parameter	116
5.1.3	Variadic Parameter	118
5.1.4	In-Out-Parameter	119
5.2	Funktionen mit Rückgabewert	120
5.3	Function Types	122
5.3.1	Funktionen als Variablen und Konstanten	124
5.4	Verschachtelte Funktionen	125
5.5	Closures	126
5.5.1	Closures als Parameter von Funktionen	127
5.5.1.1	Implicit Return	130
5.5.1.2	Shorthand Argument Names	130
5.5.2	Trailing Closures	131
5.5.3	Autoclosures	132
6	Enumerations, Structures und Classes	134
6.1	Enumerations	134
6.1.1	Enumerations und switch	137
6.1.2	Associated Values	138
6.1.3	Raw Values	141
6.2	Structures	143
6.2.1	Erstellen von Structures und Instanzen	143
6.2.2	Eigenschaften und Funktionen	144
6.2.2.1	Memberwise Initializer	148

6.3	Classes	151
6.3.1	Erstellen von Klassen und Instanzen	151
6.3.2	Eigenschaften und Funktionen	152
6.4	Enumeration vs. Structure vs. Class	154
6.4.1	Gemeinsamkeiten und Unterschiede	154
6.4.2	Wann nimmt man was?	155
6.4.2.1	Enumeration	156
6.4.2.2	Structure	156
6.4.2.3	Class	156
6.5	self	157
7	Eigenschaften und Funktionen von Typen	160
7.1	Properties	160
7.1.1	Stored Property	161
7.1.2	Lazy Stored Property	164
7.1.2.1	Einsatzzweck von Lazy Stored Properties	167
7.1.3	Computed Property	168
7.1.4	Read-Only Computed Property	171
7.1.5	Property Observer	173
7.1.6	Property Wrapper	177
7.1.6.1	Standardwerte festlegen	179
7.1.6.2	Weitere Parameter ergänzen	180
7.1.6.3	Projected Value	182
7.1.7	Type Property	183
7.2	Globale und lokale Variablen	186
7.3	Methoden	189
7.3.1	Instance Methods	189
7.3.1.1	mutating	191
7.3.2	Type Methods	192
7.4	Subscripts	193
8	Initialisierung	199
8.1	Aufgabe der Initialisierung	200
8.2	Erstellen eigener Initializer	201
8.3	Initializer Delegation	208
8.3.1	Initializer Delegation bei Value Types	208

8.3.2	Initializer Delegation bei Reference Types	209
8.4	Failable Initializer	212
8.5	Required Initializer	215
8.6	Deinitialisierung	216
9	Vererbung	218
9.1	Überschreiben von Eigenschaften und Funktionen einer Klasse	222
9.2	Überschreiben von Eigenschaften und Funktionen einer Klasse verhindern	224
9.3	Zugriff auf die Superklasse	225
9.4	Initialisierung und Vererbung	226
9.4.1	Zwei-Phasen-Initialisierung	227
9.4.2	Überschreiben von Initializern	234
9.4.3	Vererbung von Initializern	237
9.4.4	Required Initializer	237
10	Speicherverwaltung mit ARC	239
10.1	Strong Reference Cycles	243
10.1.1	Weak References	245
10.1.2	Unowned References	249
10.1.3	Weak Reference vs. Unowned Reference	251
11	Weiterführende Sprachmerkmale von Swift	252
11.1	Nested Types	252
11.2	Extensions	254
11.2.1	Computed Properties	255
11.2.2	Methoden	255
11.2.3	Initializer	256
11.2.4	Subscripts	259
11.2.5	Nested Types	259
11.3	Protokolle	260
11.3.1	Deklaration von Eigenschaften und Funktionen	262
11.3.1.1	Properties	262
11.3.1.2	Methoden	264
11.3.1.3	Subscripts	267
11.3.1.4	Initializer	268

11.3.2	Der Typ eines Protokolls	272
11.3.3	Protokolle und Extensions	275
11.3.3.1	Bestehenden Typ mittels Extension um Protokoll ergänzen	275
11.3.3.2	Protokoll mittels Extension um neue Funktionen und Standardimplementierung erweitern	277
11.3.4	Vererbung in Protokollen	280
11.3.5	Class-only-Protokolle	281
11.3.6	Optionale Eigenschaften und Funktionen	282
11.3.6.1	Umgang mit Protokoll-Type	284
11.3.7	Protocol Composition	285
11.3.8	Delegation	286
11.3.9	Übersicht diverser vorhandener Protokolle	289
11.3.9.1	Hashable	289
11.3.9.2	Identifiable	291
11.4	Key-Path	291
12	Type Checking und Type Casting	295
12.1	Type Checking mit „is“	298
12.2	Type Casting mit „as“	299
13	Error Handling	302
13.1	Deklaration und Feuern eines Fehlers	303
13.2	Reaktion auf einen Fehler	306
13.2.1	Mögliche Fehler mittels do-catch auswerten	307
13.2.2	Mögliche Fehler in Optionals umwandeln	311
13.2.3	Mögliche Fehler weitergeben	311
13.2.4	Mögliche Fehler ignorieren	313
14	Generics	314
14.1	Generic Functions	315
14.2	Generic Types	319
14.3	Type Constraints	321
14.4	Associated Types	322

15	Nebenläufigkeit	327
15.1	Asynchronen Code schreiben und aufrufen	327
15.2	Mehrere asynchrone Funktionen parallel ausführen	331
15.3	Actors	332
16	Dateien und Interfaces	335
16.1	Modules und Source Files	335
16.2	Access Control	336
16.2.1	Access Level	336
16.2.1.1	Private Access	337
16.2.1.2	File-private Access	337
16.2.1.3	Internal Access	338
16.2.1.4	Public Access	339
16.2.1.5	Open Access	339
16.2.1.6	Zusammenfassung und Übersicht	339
16.2.2	Explizite und implizite Zuweisung eines Access Levels	340
16.2.3	Besonderheiten	342
16.2.3.1	Variablen und Konstanten	342
16.2.3.2	Tuples	342
16.2.3.3	Type Aliase	343
16.2.3.4	Funktionen	343
16.2.3.5	Enumerations	344
16.2.3.6	Properties	344
16.2.3.7	Subscripts	344
16.2.3.8	Getter und Setter	344
16.2.3.9	Initializer	345
16.2.3.10	Vererbung	346
16.2.3.11	Extensions	347
16.2.3.12	Protokolle	347
	Teil II: Xcode	349
17	Grundlagen, Aufbau und Einstellungen von Xcode	351
17.1	Über Xcode	352
17.2	Arbeiten mit Xcode	353
17.2.1	Dateien und Formate eines Xcode-Projekts	354

17.2.1.1	Projekte	354
17.2.1.2	Targets	358
17.2.1.3	Schemes	359
17.2.2	Umgang mit Dateien und Ordnern	359
17.2.2.1	Dateien in Xcode hinzufügen	360
17.2.2.2	Ordner in Xcode hinzufügen	361
17.2.2.3	Bereits vorhandene Dateien einem Xcode-Projekt hinzufügen	361
17.2.2.4	Dateien und Ordner löschen	364
17.3	Der Aufbau von Xcode	365
17.3.1	Toolbar	365
17.3.2	Navigator	368
17.3.3	Editor	373
17.3.4	Inspectors	378
17.3.5	Debug Area	381
17.4	Einstellungen	382
17.4.1	General	382
17.4.2	Accounts	383
17.4.3	Behaviors	385
17.4.4	Navigation	386
17.4.5	Themes	387
17.4.6	Text Editing	388
17.4.7	Key Bindings	389
17.4.8	Source Control	390
17.4.9	Platforms	391
17.4.10	Locations	392
17.5	Projekteinstellungen	393
17.5.1	Einstellungen am Projekt	394
17.5.2	Einstellungen am Target	397
17.5.2.1	General	397
17.5.2.2	Signing & Capabilities	398
17.5.2.3	Resource Tags	399
17.5.2.4	Info	401
17.5.2.5	Build Settings	402
17.5.2.6	Build Phases	402

17.5.2.7	Build Rules	403
17.5.3	Einstellungen am Scheme	404
17.5.3.1	Neues Scheme erstellen	406
17.5.3.2	Schemes verwalten	407
17.5.3.3	Ausführungsmöglichkeiten eines Schemes	408
18	Dokumentation, Devices und Organizer	409
18.1	Dokumentation	409
18.1.1	Aufbau und Funktionsweise	410
18.1.2	Direktzugriff im Editor	413
18.2	Devices und Simulatoren	415
18.2.1	Simulatoren	417
18.2.2	Devices	419
18.3	Organizer	421
19	Debugging und Refactoring	423
19.1	Debugging	423
19.1.1	Konsolenausgaben	425
19.1.2	Arbeiten mit Breakpoints	426
19.1.2.1	Breakpoints setzen und aktivieren	426
19.1.2.2	Variables View einsetzen	426
19.1.2.3	Ausführung der App fortsetzen	428
19.1.2.4	Breakpoints konfigurieren	428
19.1.2.5	Breakpoints vollständig deaktivieren	429
19.1.2.6	Breakpoint Navigator	430
19.1.3	Debug Navigator	431
19.2	Refactoring	433
19.3	Instruments	435
20	Tipps und Tricks für das effiziente Arbeiten mit Xcode	439
20.1	Code Snippets	439
20.2	Open Quickly	442
20.3	Related Items	442
20.4	Navigation über die Jump Bar	444
20.5	MARK, TODO und FIXME	445
20.6	Shortcuts für den Navigator	446

20.7	Clean Build	446
20.8	Playgrounds	446
20.8.1	Was sind Playgrounds?	447
20.8.2	Code schreiben und testen	449
20.8.3	Dateien hinzufügen	454
20.8.4	Kommentare und Dokumentation	456
20.8.5	Swift Playgrounds-App	459
Teil III: App-Entwicklung		463
21	Grundlagen der App-Entwicklung	465
21.1	Die Basis: SwiftUI	465
21.2	Bestandteile einer App.	467
21.2.1	Umsetzung der Daten.	468
21.2.2	Umsetzung der Ansichten.	468
21.2.3	Weitere Frameworks	468
21.3	Die Syntax von SwiftUI	469
21.4	Aufbau einer App	470
21.5	Das View-Protokoll	471
21.6	Aktualisierung von Views mittels Status.	472
21.7	Grundlagen des Status	474
21.8	Anpassung von Views mittels Modifier	477
21.9	Gruppierung von Views mittels Containern.	480
21.10	Praxis: Unsere erste App	482
21.10.1	Bestandteile des neuen Projekts	488
21.10.2	Änderung des Textes	491
21.10.3	Einsatz der Preview	492
22	Views in SwiftUI	494
22.1	Textdarstellung und -bearbeitung	494
22.1.1	Text	494
22.1.1.1	Anpassung der Schriftart	496
22.1.1.2	Formatierung von Texten	501
22.1.1.3	Übersetzung von Texten	503
22.1.1.4	Umgang mit Datumsangaben.	504
22.1.2	TextField	504

22.1.3	SecureField	509
22.1.4	TextEditor	510
22.1.4.1	Formatierung des Textes	511
22.2	Bilder	513
22.2.1	Image-Instanz erstellen	515
22.2.2	Größe einer Image-Instanz ändern	518
22.3	Schaltflächen	521
22.3.1	Button	521
22.3.2	EditButton	526
22.3.3	PasteButton	527
22.4	Werteauswahl	529
22.4.1	Toggle	529
22.4.2	Slider	534
22.4.3	Stepper	541
22.4.4	Picker	547
22.4.5	DatePicker	550
22.4.6	MultiDatePicker	555
22.4.7	ColorPicker	557
22.5	Weitere Views	561
22.5.1	Label	561
22.5.2	ProgressView	564
22.5.3	Gauge	568
23	View-Layout	573
23.1	Stacks	573
23.1.1	HStack	574
23.1.2	VStack	578
23.1.3	ZStack	581
23.1.4	Stacks verschachteln	584
23.1.5	Lazy Stacks	585
23.2	Listen	589
23.2.1	List	589
23.2.2	ForEach	612
23.3	Grids	625
23.3.1	Grid	625

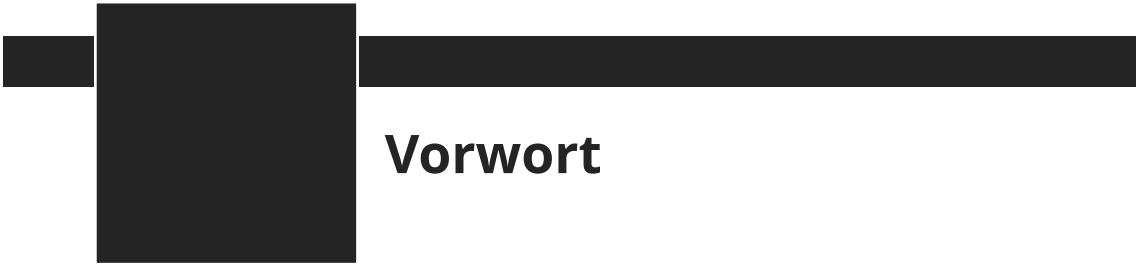
23.3.2	LazyHGrid und LazyVGrid	627
23.4	Table	643
23.4.1	Zellen selektieren	644
23.4.2	Sortierung ändern	645
23.5	Container-Views	647
23.5.1	Form	647
23.5.2	Section	649
23.5.3	Group	652
23.5.4	GroupBox	658
23.5.5	ViewThatFits	661
23.6	Weitere Views	663
23.6.1	ScrollView	663
23.6.2	OutlineGroup	668
23.6.3	DisclosureGroup	673
23.6.4	Spacer	679
23.6.5	Divider	683
24	Navigation	685
24.1	NavigationStack und NavigationLink	685
24.1.1	Titel in Navigation-Bar setzen	690
24.1.2	Eigene View zur Darstellung eines NavigationLink nutzen	694
24.1.3	Anzeige einer Ziel-View auf Basis von Daten	696
24.1.4	Programmatische Steuerung des Navigation-Stacks	700
24.2	NavigationSplitView	704
24.2.1	Verknüpfung von NavigationSplitView und List	706
24.2.2	Sichtbarkeit der Spalten steuern	708
24.2.3	Breite der Spalten anpassen	709
24.2.4	Verhalten von NavigationSplitView unter den verschiedenen Apple-Plattformen	710
24.3	TabView	716
24.4	HSplitView und VSplitView	724
24.5	Funktionen zur Präsentation von Views	725
24.5.1	Sheet einblenden	725
24.5.2	View über gesamtes Fenster legen	736
24.5.3	Popover einblenden	741

25	Weitere View-Konfigurationen	748
25.1	Toolbar	748
25.2	Alerts	758
25.3	Confirmation Dialog	761
25.4	Farben	763
25.5	View-Events	765
26	Status	767
26.1	Property	769
26.2	State	771
26.3	Binding	773
26.4	ObservedObject	782
26.4.1	Datenmodell vorbereiten	783
26.4.2	Datenmodell in View einbinden	784
26.4.3	Auf Änderungen reagieren	788
26.5	StateObject	791
26.6	EnvironmentObject	793
26.7	@Observable-Makro und Bindable	799
26.8	Environment	801
26.9	SceneStorage	807
26.10	AppStorage	810
26.11	Source of Truth vs. Derived Value	811
26.12	Best Practices	815
27	Datenhaltung	821
27.1	UserDefaults	821
27.1.1	UserDefaults und SwiftUI	823
27.2	SwiftData	824
27.2.1	Grundlegende Funktionsweise von SwiftData	824
27.2.2	Erstellen des eigenen Datenmodells	825
27.2.3	Model-Container erzeugen	827
27.2.4	Elemente im ModelContext verwalten	828
27.3	Core Data	833
27.3.1	Grundlegende Funktionsweise von Core Data	834
27.3.2	Grundlegende Elemente beim Einsatz von Core Data	834

27.3.2.1	NSPersistentStore	835
27.3.2.2	NSManagedObjectModel	835
27.3.2.3	NSManagedObject	836
27.3.2.4	NSManagedObjectContext	836
27.3.2.5	NSPersistentStoreCoordinator	836
27.3.2.6	NSPersistentContainer	837
27.3.3	Einen Core Data Stack erstellen	837
27.3.4	Ein Managed Object Model erstellen	839
27.3.4.1	Entitäten und Eigenschaften erstellen	840
27.3.4.2	Relationships zwischen Entitäten	843
27.3.4.3	Entitäten und Attribute als Managed Objects	847
27.3.5	Grundlegende Core-Data-Operationen	852
27.3.5.1	Managed Object erstellen und konfigurieren	852
27.3.5.2	Gespeicherte Managed Objects laden	853
27.3.5.3	Bestehende Managed Objects löschen	854
27.3.6	Core Data mit SwiftUI	854
27.3.6.1	NSManagedObject als Status	854
27.3.6.2	Zugriff auf Managed Object Context	856
27.3.6.3	Auslesen persistent gespeicherter Elemente	858
28	Weitere Projektkonfigurationen	862
28.1	Cross-Platform-Entwicklung	862
28.1.1	Neue Targets hinzufügen	863
28.1.2	Target-Zuweisung	866
28.1.3	Plattform im Code prüfen	868
28.1.4	Funktionen auf Verfügbarkeit prüfen	869
28.2	Mehrsprachigkeit	871
28.2.1	Grundlagen	871
28.2.1.1	Erstellen eines String Catalogs	872
28.2.1.2	Übersetzungen im Code kennzeichnen	874
28.2.1.3	Auf Übersetzungen im Code zugreifen	876
28.2.1.4	Übersetzungen mit dynamischem Parameter konfigurieren	876
28.2.2	Verschiedene Sprachen einer App testen	880
28.3	Asset Catalogs	881

29	Preview und Library	886
29.1	Preview	886
29.1.1	Funktionsweise der Preview	888
29.1.2	Konfiguration der Preview	888
29.1.3	Preview ausführen und anhalten	890
29.2	Library	892
29.3	Attributes Inspector	894
	Teil IV: Source Control und Testing	897
30	Source Control	899
30.1	Basisfunktionen und -begriffe der Source Control	899
30.2	Source Control in Xcode	901
30.2.1	Bestehendes Projekt klonen	903
30.2.2	Lokale Änderungen committen	905
30.2.3	Lokale Änderungen verwerfen	908
30.2.4	Pull und Push	909
30.2.5	Aktuelle Branches vom Repository laden	910
30.2.6	Git-Repository mit neuem Xcode-Projekt erzeugen	910
30.2.7	Optische Source-Control-Hervorhebungen im Editor	911
30.2.8	Zugriff auf GitHub, GitLab und Bitbucket	912
30.3	Source Control Navigator	913
30.4	Code Review-Mode	914
31	Testing	917
31.1	Unit-Tests	917
31.1.1	Aufbau und Funktionsweise von Unit-Tests	922
31.1.2	Aufbau einer Test-Case-Klasse	925
31.1.3	Neue Test-Case-Klasse erstellen	927
31.1.4	Ausführen von Unit-Tests	928
31.1.5	Was sollte ich eigentlich testen?	930
31.2	Performancetests	931
31.3	UI-Tests	933
31.3.1	Klassen für UI-Tests	934
31.3.2	Aufbau von UI-Test-Klassen	937
31.3.3	Automatisches Erstellen von UI-Tests	937
31.3.4	Einsatz von UI-Tests	938

Teil V: Veröffentlichung von Apps	939
32 Veröffentlichung im App Store	941
32.1 Das Apple Developer Portal	942
32.1.1 Zertifikate, App IDs und Provisioning Profiles.	946
32.1.1.1 Erstellen von Entwicklerzertifikaten	948
32.1.1.2 Erstellen von App IDs.	952
32.1.1.3 Hinzufügen von Devices	954
32.1.1.4 Erstellen von Provisioning Profiles.	957
32.1.2 Code Signing	964
32.1.2.1 Automatic Code Signing	965
32.1.2.2 Manual Code Signing	966
32.1.2.3 Code Signing in den Build Settings	967
32.2 App Store Connect.	969
32.2.1 Apps für den App Store vorbereiten und verwalten	970
32.2.2 Apps erstellen, hochladen und einreichen	974
32.3 App Store Review Guidelines	977
33 Das Business Model für Ihre App	979
33.1 Geschäftsmodelle	979
33.1.1 Free Model	980
33.1.2 Freemium Model.	980
33.1.3 Subscription Model.	980
33.1.4 Paid Model	981
33.1.5 Paymium Model.	982
33.2 App Bundles	982
33.3 Veröffentlichung außerhalb des App Store.	984
33.3.1 Das Apple Developer Enterprise Program	985
34 TestFlight	986
34.1 TestFlight in App Store Connect.	986
34.2 TestFlight im App Store	988
Index	991



Vorwort

Herzlich Willkommen in der Welt von Swift, Apples haus eigener 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 2023 erschienenen Version 5.9 von Swift sowie der Version 15 der Entwicklungsumgebung Xcode. Alle Kapitel der vorherigen Auflage wurden entsprechend aktualisiert sowie um neue Inhalte ergänzt. Zu den Highlights dieser Neuerungen zählt unter anderem das SwiftData-Framework für die persistente Speicherung von Daten, die neuen String Catalogs zur Übersetzung von Apps sowie das Observable-Makro als Alternative zum ObservedObject-Property Wrapper.

Generell schlägt diese dritte Auflage des Swift-Handbuchs inhaltlich den gleichen Weg ein wie die vorherige Auflage. Der Fokus liegt klar auf den drei wichtigsten Elementen für die App-Entwicklung für macOS, iOS/iPadOS, watchOS und tvOS:

- Die Programmiersprache Swift
- Die Entwicklungsumgebung Xcode
- Die App-Entwicklung mit SwiftUI

Insbesondere SwiftUI ist in dieser Auflistung hervorzuheben. Seit der erstmaligen Vorstellung im Jahr 2019 hat sich SwiftUI massiv weiterentwickelt und bietet inzwischen eine Vielzahl an Möglichkeiten zur Umsetzung und Gestaltung von Apps.

Dieses Handbuch liefert Ihnen handfestes Wissen zu allen drei genannten Bereichen. So lernen Sie die Programmiersprache Swift und ihre verschiedenen Sprachfeatures kennen. Sie erfahren, wie die Entwicklungsumgebung Xcode aufgebaut ist und welche Funktionen Ihnen bei der täglichen Entwicklerarbeit zur Verfügung stehen. Und Sie erhalten einen Überblick über die Möglichkeiten, die Sie zur Gestaltung von Apps mit SwiftUI nutzen können. Darüber hinaus finden Sie auch Kapitel zur Versionsverwaltung und zum Testing sowie zur Veröffentlichung von Apps im App Store.

Kurzum: Dieses Handbuch soll Ihnen als Grundlage dienen, um einen Überblick über die wichtigsten Bereiche der App-Entwicklung für Apple-Plattformen zu erhalten und Ihnen das nötige Verständnis vermitteln, um das erlangte Wissen in eigenen Projekten zum Einsatz bringen zu können.

Aufbau des Buches

Mir ist es wichtig, dass Sie das Buch als Referenzwerk nutzen 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.

Beispielprojekte auf Hanser-Plus

Über Hanser-Plus stehen Ihnen kleine Beispielprojekte zum Download zur Verfügung, die einzelne Aspekte der App-Entwicklung und der Programmierung mit Swift beleuchten. Diese Beispiele sind bewusst überschaubar gehalten, um den Fokus auf grundlegende Funktionsweisen zu richten und Ihnen einen verständlichen Überblick zu verschaffen. Nutzen Sie diese Beispiele gerne, um darauf aufbauend selbst ein wenig im Code zu experimentieren oder um zu überprüfen, wie sich bestimmte Funktionen umsetzen lassen. Um die Beispiele herunterzuladen, geben Sie auf der Webseite

<https://plus.hanser-fachbuch.de/>

den Code

`plus-12abc-8xyz9`

ein.

Feedback

In diesem Sinne wünsche ich Ihnen nun viel Freude mit dem Buch und Erfolg beim Programmieren. Sollten Sie Feedback zum Buch 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

September 2023

2

Grundlagen der Programmierung

In diesem Kapitel möchte ich Ihnen eine Einführung in die Grundlagen der Programmierung mit Swift geben. Es gibt Ihnen einen ersten Einblick in die Swift Standard Library, zeigt das Erstellen und Verwenden von Variablen und Konstanten und wie Sie Ihren Quellcode mithilfe von Kommentaren dokumentieren. Wenn Sie dabei sind, Swift zu lernen, empfehle ich Ihnen, die Beispiele dieses Buches in einem Playground auszuprobieren, um so möglichst schnell ein Gefühl für die Sprache zu bekommen und aktiv Code zu schreiben.

2.1 Grundlegendes

Im Folgenden stelle ich Ihnen verschiedene Bestandteile und Funktionen von Swift vor, die die Basis für die Programmierung darstellen.

2.1.1 Swift Standard Library

Die Swift Standard Library enthält ein umfangreiches Set an verschiedensten Klassen und Funktionen (siehe Bild 2.1). Sie ist Teil der Programmiersprache Swift, sodass alles, was Teil der Standard Library ist, auch in jedem Swift-Programm verwendet werden kann.

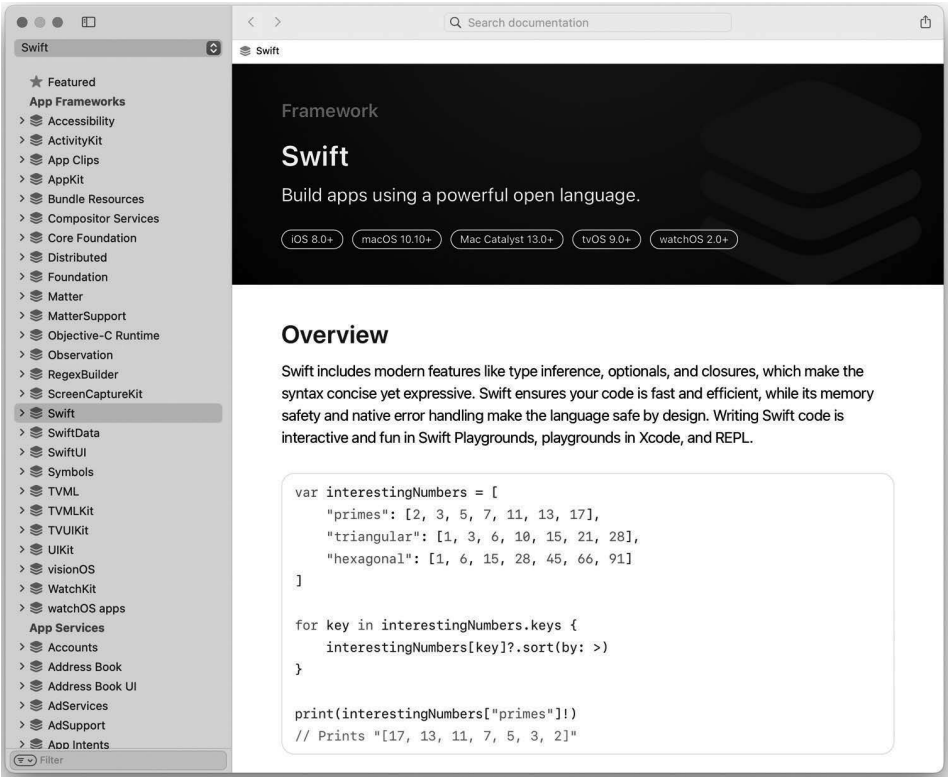


Bild 2.1 Die Swift Standard Library enthält ein umfangreiches Set an Funktionen, die uns bei der Programmierung mit Swift immer zur Verfügung stehen.

Dabei werden wir vielen sogenannten *Typen* der Swift Standard Library begegnen (was ein Typ genau ist und wie man selbst welche deklariert, folgt im Laufe dieses Kapitels). Dazu gehören beispielsweise die Typen `Int`, `Double`, `Character`, `String`, `Array` oder `Dictionary`. Die folgende Tabelle 2.1 gibt einen kurzen Überblick über einige der wichtigsten und grundlegendsten Typen für die Programmierung mit Swift, an passender Stelle im Buch werden diese auch noch tiefergehend beschrieben.

Tabelle 2.1 Auswahl grundlegender Typen der Swift Standard Library

Fundamental Type	Beschreibung	Beispiele
Int	Ein Integer (<code>Int</code>) stellt eine Ganzzahl dar.	19 99
Float	Bei <code>Float</code> handelt es sich um eine Fließkommazahl	19.99 49.94

Tabelle 2.1 Auswahl grundlegender Typen der Swift Standard Library (*Fortsetzung*)

Fundamental Type	Beschreibung	Beispiele
Double	Auch bei Double handelt es sich um eine Fließkommazahl, allerdings ist der Wertebereich von Double deutlich größer als der von Float; entsprechend belegt ein Double auch mehr Speicherplatz im System als ein Float.	99.19 94.49
Bool	Bei Bool handelt es sich um einen sogenannten Wahrheitswert, dieser kann somit entweder wahr oder falsch (true oder false) sein.	true false
String	Ein String repräsentiert eine Zeichenkette.	"Mein Name ist Thomas Sillmann."
Array	In einem Array können mehrere Werte und Objekte abgelegt werden. Das Array erlaubt dann den Zugriff auf die Werte und Objekte, die es hält. Ein Array kann dabei beliebige Typen von Werten und Objekten beinhalten.	["Erster Wert des Arrays", "Zweiter Wert des Arrays"]
Dictionary	Ein Dictionary hält mehrere Werte und Objekte, ähnlich wie ein Array, allerdings ist jeder Wert und jedes Objekt einem einzigartigen Schlüssel innerhalb des Dictionaries zugeordnet. Anhand dieses Schlüssels können dann gezielt Werte ausgelesen, abgefragt und verändert werden.	["Schlüssel 1": "Wert für Schlüssel 1", "Schlüssel 2": "Wert für Schlüssel 2"]

Sie müssen zum jetzigen Zeitpunkt noch nicht mehr über die genannten Typen wissen, weitere Informationen zu ihnen folgen im Laufe dieses Buches an passender Stelle.

2.1.2 print

Im Laufe dieses Buches werden Sie sehr viele Elemente und Funktionen der Swift Standard Library kennenlernen. Eine der von mir am häufigsten verwendeten Befehle nennt sich `print(_:separator:terminator:)` und dient dazu, Text in der Konsole auszugeben. Ein Beispiel zeigt Listing 2.1. Wo immer diese Funktion zum Einsatz kommt, werde ich in den zugehörigen Listings auch die jeweilige Ausgabe (oder im

Fälle mehrere Befehle auch alle jeweiligen Ausgaben) am Ende als Kommentar mit aufführen.

Listing 2.1 Einfache Konsolenausgabe mittels print

```
print("Das ist eine Konsolenausgabe")  
// Das ist eine Konsolenausgabe
```

Darüber hinaus werde ich der Einfachheit halber, wo immer diese Funktion verwendet wird, auf diese im Fließtext mit print verweisen und mir die eigentlich korrekte Bezeichnung aus Platzgründen sparen.

2.1.3 Befehle und Semikolons

Bei der Entwicklung mit Swift schreibt man verschiedene aufeinanderfolgende Befehle, um damit am Ende ein funktionsfähiges Programm umzusetzen. Pro Zeile wird dabei genau ein Befehl geschrieben, beispielsweise um eine Variable zu erstellen oder einen Text auf der Konsole auszugeben. Jeder neue Befehl folgt in einer neuen Zeile (siehe Listing 2.2).

Listing 2.2 Schreiben eines Befehls pro Zeile

```
print("Das ist ein erster Befehl.")  
print("Anschließend folgt ein zweiter.")  
print("Und zum Abschluss ...")  
print("... noch ein vierter!")
```

In vielen anderen Programmiersprachen muss jeder Befehl mit einem Semikolon (;) abgeschlossen werden. In Swift ist das ebenfalls möglich, aber kein Muss (wie das Listing von eben gezeigt hat). Sie können den Code aus Listing 2.2 also auch so, wie in Listing 2.3 gezeigt, umsetzen und am Ende eines jeden Befehls ein Semikolon setzen.

Listing 2.3 Schreiben eines Befehls mit abschließendem optionalen Semikolon

```
print("Das ist ein erster Befehl.");  
print("Anschließend folgt ein zweiter.");  
print("Und zum Abschluss...");  
print("...noch ein vierter!");
```

Ein Semikolon zum Abschluss ist nur dann Pflicht, wenn man *mehrere* Befehle in einer Zeile schreiben möchte (siehe Listing 2.4).

Listing 2.4 Schreiben mehrerer Befehle in einer einzigen Zeile

```
print("Erster Befehl ..."); print("... direkt gefolgt vom zweiten!")
```

Der letzte Befehl in der Zeile muss wiederum nicht zwingend mit einem Semikolon abgeschlossen werden.



Semikolon – ja oder nein?

Womöglich fragen Sie sich nach diesem Abschnitt, was nun die bessere Lösung ist; Befehle mit einem Semikolon abzuschließen oder nicht? Und sollten in Swift mehrere Befehle in eine einzige Zeile geschrieben werden?

Ob und wie Sie letztlich das Semikolon in Swift auf die gezeigte Art und Weise verwenden, ist zunächst einmal voll und ganz Ihnen überlassen. Ich allerdings orientiere mich bei der Arbeit mit Swift an Apples Vorgehen aus der offiziellen Dokumentation, und dort wird prinzipiell **kein** Semikolon bei der Programmierung mit Swift eingesetzt (auch mehrere Befehle pro Zeile finden sich dort nicht). Wenn Sie also nicht gerade ein extremer Fan von Semikolons sind, dann würde ich Ihnen empfehlen, es genauso zu handhaben und einen Befehl pro Zeile zu schreiben – ohne abschließendes Semikolon.

2.1.4 Operatoren

Operatoren dienen dazu, im Code Befehle (wie beispielsweise Zuweisungen oder Berechnungen) durchzuführen. Da sich Operatoren durch viele Bereiche der Programmiersprache ziehen, möchte ich Ihnen gleich an dieser Stelle eine Übersicht der in Swift verfügbaren Operatoren geben (siehe Tabelle 2.2). An den Stellen im Buch, an denen diese Operatoren konkret zum Einsatz kommen, erhalten Sie weitere Erläuterungen und Ergänzungen dazu.

Tabelle 2.2 Operatoren in Swift

Operator	Art	Funktion
=	Zuweisungsoperator	Weist den Wert auf der rechten Seite des Operators dem Objekt auf der linken Seite zu.
==	Vergleichsoperator	Prüft, ob der Wert links vom Operator mit dem rechts vom Operator identisch ist.
!=	Vergleichsoperator	Prüft, ob der Wert links vom Operator mit dem rechts vom Operator nicht identisch ist.
<	Vergleichsoperator	Prüft, ob der Wert links vom Operator kleiner dem rechts vom Operator ist.
<=	Vergleichsoperator	Prüft, ob der Wert links vom Operator kleiner oder gleich dem rechts vom Operator ist.

Operator	Art	Funktion
>	Vergleichs-operator	Prüft, ob der Wert links vom Operator größer dem rechts vom Operator ist.
>=	Vergleichs-operator	Prüft, ob der Wert links vom Operator größer oder gleich dem rechts vom Operator ist.
+	Berechnungs-operator	Dient zur Durchführung von Additionen.
-	Berechnungs-operator	Dient zur Durchführung von Subtraktionen.
*	Berechnungs-operator	Dient zur Durchführung von Multiplikationen.
/	Berechnungs-operator	Dient zur Durchführung von Divisionen.
%	Berechnungs-operator	Dient zur Berechnung des Rests bei einer Division.
+=	Berechnungs-operator	Erhöht den Wert links vom Operator um den Wert rechts vom Operator.
-=	Berechnungs-operator	Verringert den Wert links vom Operator um den Wert rechts vom Operator.
&&	Logischer Operator	Verknüpft zwei Bedingungen mittels UND; ist eine von ihnen false, ist auch das Ergebnis false.
	Logischer Operator	Verknüpft zwei Bedingungen mittels ODER; ist eine von beiden true, ist auch das Ergebnis true.
!	Logischer Operator	Kehrt einen Wahrheitswert um (true wird false, false wird true).
...	Range-Operator	Erstellt eine Wertereihe, die mit dem Wert links vom Operator beginnt und mit einschließlich dem Wert rechts vom Operator endet. Dabei darf der Wert links vom Operator nicht größer sein als der Wert rechts vom Operator.
..<	Range-Operator	Erstellt eine Wertereihe, die mit dem Wert links vom Operator beginnt und mit ausschließlich dem Wert rechts vom Operator endet. Dabei darf der Wert links vom Operator nicht größer sein als der Wert rechts vom Operator.
??	Nil-Operator	Prüft den optionalen Wert links vom Operator. Ist dieser nil, wird der Wert rechts vom Operator zurückgegeben, andernfalls wird der Wert links entpackt und zurückgegeben.

2.2 Variablen und Konstanten

Mithilfe von Variablen und Konstanten speichern Sie Werte zwischen, die Sie dann auslesen und weiterverarbeiten können. Einer Konstanten kann nur einmalig ein Wert zugewiesen werden, dieser ist anschließend nicht mehr veränderbar. Der Versuch, den Wert einer Konstanten anschließend zu ändern, endet in einem Compiler-Fehler. Im Gegensatz dazu kann der einer Variablen zugewiesene Wert jederzeit geändert werden.

2.2.1 Erstellen von Variablen und Konstanten

Eine Variable wird in Swift mittels des Schlüsselworts `var` deklariert, eine Konstante mittels `let`. Nach dem jeweiligen Schlüsselwort folgt der gewünschte Name für die Variable beziehungsweise Konstante. Dieser beginnt in Swift typischerweise mit einem Kleinbuchstaben. Setzt sich der Name aus mehreren verschiedenen Wörtern zusammen, so beginnt man jedes folgende Wort typischerweise mit einem Großbuchstaben.

Listing 2.5 zeigt ein Beispiel dazu. Dort wird eine Variable und eine Konstante deklariert und dieser direkt ein Wert (in diesem Fall ein String) zugewiesen. Die Zuweisung erfolgt mithilfe des Zuweisungsoperators `=`.

Listing 2.5 Erstellen von Variablen und Konstanten

```
var aVariable = "Eine Variable"  
let aConstant = "Eine Konstante"
```

Um nach der Deklaration auf die Werte von Variablen und Konstanten zuzugreifen, nutzt man einfach den vergebenen Variablen- beziehungsweise Konstantennamen. So wird in Listing 2.6 auf die zuvor erstellte Variable `aVariable` zugegriffen und ihr ein neuer Wert zugewiesen.

Listing 2.6 Zugriff auf eine erstellte Variable

```
aVariable = "Ein neuer String"
```

Die Zuweisung eines Werts zu einer Variablen würde bei der zuvor deklarierten Konstanten `aConstant` nicht funktionieren, da Konstanten wie beschrieben nur einmalig ein Wert zugewiesen werden kann und dieser anschließend unveränderlich ist. Ein Versuch, den Wert einer Konstanten im Nachhinein zu ändern, führt immer zu einem Compiler-Fehler (siehe Listing 2.7).

Listing 2.7 Fehler beim Versuch des Ändern einer Konstanten

```
aConstant = "Eine neue Konstante"  
// Compiler-Fehler: aConstant kann nicht verändert werden.
```



Wann Variable, wann Konstante?

Möglicherweise denken Sie nach dem Lesen dieses Abschnitts, dass es sinnvoll ist, sicherheitshalber lieber immer eine Variable statt eine Konstante zu erstellen, da Sie diese im Zweifelsfall noch verändern können. Das sollten Sie aber per se keinesfalls tun.

Denn diese Medaille hat noch eine zweite Seite: Sobald Sie beispielsweise einen neuen Wert erstellen, der innerhalb Ihres Programms unveränderlich sein soll (beispielsweise, weil er eine grundlegende und essenzielle Information enthält), dann können Sie genau dieses gewünschte Verhalten damit sicherstellen, diesen Wert mittels `let` als Konstante zu deklarieren. Wenn Sie dann fälschlicherweise an einer Stelle in Ihrem Projekt nun doch versuchen, genau diesen Wert zu ändern, dann macht Sie der Compiler direkt auf dieses Problem aufmerksam. Und genau für solche Zwecke – für Werte, die einmal gesetzt und anschließend nicht mehr verändert werden sollen – sind Konstanten da.

Das geht sogar so weit, dass in Swift generell der Grundsatz gilt: Wenn ein Wert nicht geändert werden muss oder soll, dann deklarieren Sie ihn als Konstante! Erstellen Sie daher im Zweifelsfall lieber eine unveränderliche Konstante als eine Variable. Sollte sich das später doch als möglicher Fehler herausstellen, ist es immer noch ein Leichtes, die Deklaration von einer Konstanten hin zu einer Variablen zu verändern.

2.2.2 Variablen und Konstanten in der Konsole ausgeben

Um den Wert von Variablen und Konstanten auf der Konsole auszugeben (beispielsweise bei der Suche nach Fehlern im Code) steht in Swift die Funktion `print` zur Verfügung. Typischerweise wird `print` ein String übergeben, der anschließend in der Konsole ausgegeben wird (siehe dazu auch den vorherigen Abschnitt 2.1.2, „`print`“). Sie können innerhalb dieses Strings aber auch eine Variable oder Konstante als eine Art Platzhalter übergeben, deren Wert dann in den String der `print`-Funktion eingefügt und ausgegeben wird. Um eine Variable oder Konstante auf die genannte Art und Weise in einen String einzubinden, müssen Sie sie innerhalb des Strings besonders kennzeichnen. Dazu nutzen Sie den folgenden Code:

```
\(<VARIABLE ODER KONSTANTE>)
```

In Listing 2.8 sehen Sie einmal ein Beispiel dazu, wie die Werte von Variablen und Konstanten mittels `print` ausgegeben werden können. Dazu werden die im vorherigen Abschnitt erstellte Variable `aVariable` und die Konstante `aConstant` verwendet.

Listing 2.8 Ausgabe der Werte von Variablen und Konstanten mittels `print`

```
print("aVariable hat folgenden Wert: \(aVariable)")
print("aConstant hat folgenden Wert: \(aConstant)")
// aVariable hat folgenden Wert: Ein neuer String
// aConstant hat folgenden Wert: Eine neue Konstante
```

Das gezeigte Vorgehen wird auch als *String Interpolation* bezeichnet; mehr dazu erfahren Sie in Kapitel 4, „Typen in Swift“.

2.2.3 Type Annotation und Type Inference

Variablen und Konstanten in Swift sind immer einem ganz bestimmten Typ zugeordnet. Eine Variable ist beispielsweise also entweder eine Zahl *oder* ein String. Handelt es sich bei ihr um eine Zahl, dann können ihr auch nur Zahlen und keine Strings zugewiesen werden, umgekehrt gilt genau das Gleiche. Dieses Verhalten wird als *Typsicherheit* bezeichnet, da man sich darauf verlassen kann, dass eine Variable oder Konstante immer nur einen Wert passend zu ihrem Typ besitzt.

Wenn Sie eine neue Variable oder Konstante erstellen, können Sie direkt angeben, von welchem Typ diese Variable beziehungsweise Konstante ist. Dazu fügen Sie nach dem Namen der Variablen oder Konstanten einen Doppelpunkt, gefolgt vom gewünschten Typ, ein. In Listing 2.9 sehen Sie ein Beispiel dazu.

Listing 2.9 Typzuweisung beim Erstellen von Variablen und Konstanten

```
var aString: String
let anInteger: Int
```

Hier wird festgelegt, dass die Variable `aString` vom Typ `String` ist und die Konstante `anInteger` vom Typ `Int` (sowohl bei `String` als auch bei `Int` handelt es sich um automatisch bei der Programmierung mit Swift zur Verfügung stehende Typen aus der Swift Standard Library). Möchte man diesen beiden nun einen Wert zuweisen, so ist darauf zu achten, dass `aString` nur eine Zeichenkette entgegennehmen kann, während man `anInteger` nur eine Ganzzahl zuweisen kann (siehe Listing 2.10). Der Versuch, ihnen einen Wert eines anderen Typs zuzuweisen, hätte einen Compiler-Fehler zur Folge.

Listing 2.10 Wertzuweisung passend zu den Typen von Variablen und Konstanten

```
aString = "Ein mittels Type Annotation erstellter String"
anInteger = 19
```

Das gezeigte Vorgehen der direkten Typzuweisung beim Erstellen einer Variablen oder Konstanten wird als *Type Annotation* bezeichnet. Sollte diese nicht angewendet werden und – wie in den vorherigen Listings dieses Abschnitts zu sehen war – einer neuen Variablen oder Konstanten stattdessen direkt ein Wert zugewiesen werden, dann tritt die sogenannte *Type Inference* in Kraft. Fehlt nämlich eine konkrete Typzuweisung mittels Type Annotation, dann ermittelt Swift selbst, welchen Typ die Variable oder Konstante besitzen soll, sobald ihr ein Wert zugewiesen wird. Betrachten wir dazu einmal in Listing 2.11 die Erstellung einer neuen Konstanten und Variablen mittels Type Inference.

Listing 2.11 Erstellen neuer Variablen mittels Type Inference

```
let myName = "Thomas Sillmann"  
var myAge = 28  
// myName ist vom Typ String  
// myAge ist vom Typ Int
```

Auch wenn es im Listing selbst nicht explizit angegeben ist, legt Swift automatisch sowohl für die Konstante `myName` als auch für die Variable `myAge` einen Typ fest, ausgehend von dem zugewiesenen Wert. So entspricht `myName` nun dem Typ `String` und `myAge` dem Typ `Int`.

Wann sollten Sie nun welches der beiden Verfahren einsetzen? Wann ist die explizite Typzuweisung mittels Type Annotation notwendig und in welchen Fällen kann man Swift den Typ selbst mittels Type Inference ermitteln lassen?

Generell ist der Einsatz von Type Annotation in zwei Situation zwingend notwendig:

- Wenn Sie einer neuen Variablen oder Konstanten bei deren Deklaration noch keinen Wert zuweisen, müssen Sie in jedem Fall den gewünschten Typ für die Variable oder Konstante angeben (so wie in Listing 2.9); andernfalls kommt es zu einem Compiler-Fehler.
- Wenn der mittels Type Inference von Swift ermittelte Typ bei der Erstellung einer Variablen oder Konstanten nicht dem gewünschten Typ entspricht, muss ebenfalls explizit der korrekte Typ mittels Type Annotation angegeben werden.

Den zweiten Punkt möchte ich zum besseren Verständnis noch einmal anhand eines Beispiels erläutern. Dazu wird in Listing 2.12 eine neue Variable namens `aDouble` erstellt und ihr der Zahlenwert 99 zugewiesen. Wie der Name der Variablen andeutet, soll diese im Code als `Double` (also als Fließkommazahl) verwendet werden können.

Listing 2.12 Erstellen einer neuen Variablen mit dem gewünschten Typ `Double`

```
Var aDouble = 99  
// aDouble entspricht dem Typ Int
```

Zwar ist der gezeigte Code korrekt, allerdings handelt es sich bei `aDouble` nun nicht um eine Variable vom gewünschten Typ `Double`, sondern um eine vom Typ `Int`. Denn

Swift vermutet hinter der zugewiesenen Ganzzahl 99 nun einmal keine Fließkommazahl, auch wenn 99 natürlich nichtsdestoweniger ein valider Wert für eine Fließkommazahl wäre. Der Versuch, `aDouble` nun im Nachhinein einen Wert wie 19.99 zuzuweisen, würde ebenfalls in einem Compiler-Fehler enden. Daher ist es in so einem Fall zwingend notwendig, den gewünschten Typ ebenfalls explizit mittels Type Annotation anzugeben, wie in Listing 2.13 zu sehen.

Listing 2.13 Erstellen einer neuen Double-Variablen mittels Type Annotation

```
var aDouble: Double = 99
```

Damit ist trotz der Zuweisung einer Ganzzahl die Variable `aDouble` vom Typ `Double` und sie kann somit auch mit Fließkommazahlen umgehen.

2.2.4 Gleichzeitiges Erstellen und Deklarieren mehrerer Variablen und Konstanten

Sie haben in Swift die Möglichkeit, mehrere Variablen und Konstanten direkt in einem Befehl zu erstellen und ihnen dabei optional bereits Werte zuzuweisen. Dazu beginnen Sie den entsprechenden Befehl entweder mit dem Schlüsselwort `var` (für zu erstellende Variablen) oder `let` (für zu erstellende Konstanten) und benennen dann kommasepariert alle neu zu erstellenden Variablen beziehungsweise Konstanten. Dabei können Sie entweder allen oder einzelnen Elementen direkt nach dem Namen auf die bekannte Art und Weise einen Wert zuweisen oder einen festen Typ mittels Type Annotation definieren. In Listing 2.14 sehen Sie einige Beispiele dazu, wie dieses Prinzip praktisch angewendet werden kann.

Listing 2.14 Gleichzeitiges Erstellen und Deklarieren mehrerer Variablen und Konstanten

```
var firstValue: Int, secondValue: Double, thirdValue: String
var firstString, secondString, thirdString: String
let firstInt = 19, secondInt = 99
let numericValue = 19, numericString = "99"
```

Besonders interessant ist dabei auch die zweite Zeile `var firstString, secondString, thirdString: String`, in der nur eine einzige Type Annotation ganz am Ende erfolgt. Dadurch wird allen in diesem Befehl neu erstellten Variablen der am Ende explizit definierte Typ `String` zugewiesen, womit man sich die wiederholende Schreibarbeit spart, möchte man mehrere neue Variablen oder Konstanten von ein und demselben Typ auf einmal definieren.

2.2.5 Namensrichtlinien

Bei der Benennung von Variablen und Konstanten in Swift haben Sie – gerade im Vergleich mit anderen Programmiersprachen – sehr viele Freiheiten. So können beispielsweise Sonderzeichen wie Pi π oder sogar Emojis für Variablen- und Konstantennamen verwendet werden (siehe Listing 2.15).

Listing 2.15 Verwendung von Sonderzeichen und Emojis als Variablen- und Konstantennamen

```
let  $\pi$  = 3.14159  
let 🐸 = "Frog"
```

Dennoch sind einige Dinge nicht erlaubt und führen direkt zu einem Compiler-Fehler. Beispielsweise müssen Sie auf jegliche Leerzeichen in einem Variablen- oder Konstantennamen verzichten, ebenso wie auf mathematische Operatoren oder Pfeile. Auch dürfen Variablen- oder Konstantennamen nicht mit einer Ziffer beginnen, ansonsten sind Ziffern im Namen aber erlaubt.



Im Zweifel lieber drauf verzichten

So schön die genannten Möglichkeiten und Freiheiten bei der Benennung von Variablen und Konstanten auch sind, sollte man sich dennoch überlegen, ob und wann sie tatsächlich angebracht sind. Gerade Sonderzeichen und Emojis sind womöglich eher ungeeignet für den eigenen Code, auch wenn diese Möglichkeit – wie wir gesehen haben – in Swift ja durchaus zur Verfügung steht. Wenn es keinen konkreten oder sinnvollen Grund für die Verwendung dieser Sonderzeichen gibt, sollten Sie im Zweifelsfall lieber darauf verzichten und stattdessen mit den bekannten alphanumerischen Zeichen bei der Benennung von Variablen und Konstanten arbeiten.

2.3 Kommentare

Kommentare sind in der Programmierung ein beliebtes und zugleich sehr wichtiges Mittel zur Dokumentation des eigenen Quellcodes. Kommentare werden vom Compiler ignoriert und nicht ausgeführt, was bedeutet, dass alles, was Sie innerhalb von Kommentaren schreiben, keinen Einfluss auf die Funktionalität Ihrer Anwendung hat. Typischerweise geben Sie mit Kommentaren Aufschluss über die Funktionsweise bestimmter Befehle oder die Aufgabe von deklarierten Variablen und Konstanten.

In Swift gibt es zwei Arten von Kommentaren: solche, die genau für eine Zeile gelten und solche, die sich über beliebig viele Zeilen erstrecken.

Ein einfacher einzelzeiliger Kommentar wird mit zwei Slashes `//` eingeleitet, direkt im Anschluss beginnt der Kommentar. Alles, was also hinter den beiden Slashes steht, wird vom Compiler ignoriert und dient einzig und allein dazu, den Quellcode zu dokumentieren. In Listing 2.16 sehen Sie ein einfaches Beispiel dazu.

Listing 2.16 Ein einzelzeiliger Kommentar

```
// Ein Kommentar
```

Solch ein Kommentar kann sowohl am Anfang als auch am Ende einer Zeile stehen (am Ende bedeutet dabei nach dem letzten Befehl innerhalb dieser Zeile). Auch dazu sehen Sie ein kleines Beispiel in Listing 2.17.

Listing 2.17 Ein einzelzeiliger Kommentar nach einem Befehl

```
print("Hier wird noch Code ausgeführt ...") // ... dann folgt ein Kommentar!
```

Manchmal benötigt aber ein sinnvoller Kommentar mehr Platz als nur eine einzige Zeile, und hier kommen die mehrzeiligen Kommentare ins Spiel. Diese beginnen mit einem `/*` und enden mit einem `*/`. Alles, was sich dazwischen – auch über mehrere Zeilen hinweg – befindet, gehört zum Kommentar (siehe Listing 2.18).

Listing 2.18 Ein mehrzeiliger Kommentar

```
/* Der Kommentar beginnt in der ersten Zeile ...  
... erstreckt sich über die zweite ...  
... und endet schließlich in der dritten! */
```

Dabei können mehrzeilige Kommentare in Swift sogar verschachtelt werden. Ein mehrzeiliger Kommentar kann also einen weiteren mehrzeiligen Kommentar enthalten. Wie so etwas aussehen kann, zeigt Listing 2.19.

Listing 2.19 Verschachtelte Kommentare

```
/* Hier beginnt der erste Kommentar ...  
/* ... und hier der zweite ...  
... der in dieser Zeile bereits wieder endet ... */  
... sowie auch abschließend der erste Kommentar. */
```

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 heute maßgeblich für die Entwicklung von Apps für Apple-Plattformen ist 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 unterbrin-

gen. 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:

- UserDefaults
- SwiftData
- Core Data

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 das 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")  
    }  
}
```

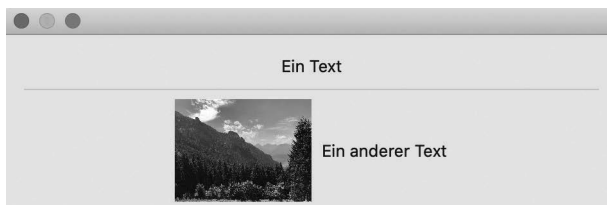


Bild 21.2

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

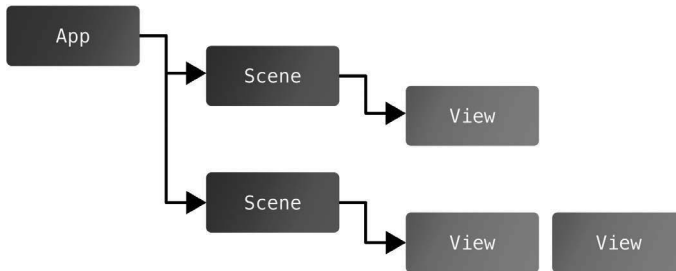
Die **Views** kennen wir in diesem Zusammenhang bereits grundlegend. Sie sind unsere Ansichten, die einzelne Teile unseres User Interface widerspiegeln.

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 Speicher-Management 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 solcher 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

Index

Symbole

>guard 41
\$-Syntax 774
Array 16
@Attribute 826
#available 869
Bool 16
Dictionary 16
Double 16
Float 15
Function Type 122
Int 15
@Model 825
@objc 282
@Observable 799
#Predicate 830
#Preview 492, 888
@propertyWrapper> 177
@Relationship 826
String 16
.swift 335
@testable 923
#unavailable 870

A

Abfrage 32
Access Control 336

Accessibility 935
Access Level 336
– explizite Zuweisung 340
– implizite Zuweisung 340
actor 333
Actor 330, 332
Alert 758
Analyze 408
Ansicht 467
Any 106, 298
AnyObject 106, 298
App 470
– Aufbau 470
– Bestandteile 467
App Bundle 982
App Icon 883
App ID 947
AppKit 466
Apple Developer Account 943
Apple Developer Enterprise
Program 985
Apple Developer Portal 942
Apple Developer Program 943
Apple Script 429
AppStorage 810, 823
App Store Connect 969
App Store Review Guidelines 977
ARC 216, 239

- Archive 974
- Argument Label 114
- Array 59
 - mutable 62
 - Shorthand Syntax 59
- as 299
- as! 300
- as? 300
- Asset Catalog 881
- associatedtype 323
- Associated Type 322
- Associated Values 138
- async 327
- Attributes Inspector 894
- Automatic Reference Counting 216, 239
- Availability Condition 869
- await 329

B

- Barrierefreiheit 497
- Basisklasse 221
- Bindable 800
- Binding 773
 - Konstante 781
- Bookmark Navigator 369
- Bool 51
- Branch 900
- break 39, 43
- Breakpoint 426
 - Konfiguration 428
- Breakpoint Navigator 372, 430
- Build Configuration 395
- Build Phases 402
- Build Rules 403
- Build Settings 395
- Bundle Identifier 356
- Business Model 979
 - Freemium Model 980
 - Free Model 980
 - Paid Model 981
 - Paymium Model 982
 - Subscription Model 980
- Button 521
 - Rolle 523
 - Style 524

C

- Character 53
- Chris Lattner 3
- class 281
- Class 151
- Class-only-Protokoll 281
- Class-Protocol 281
- Clean Build 446
- Clone 900
- Closure 126
 - Autoclosure 132
 - Default Value 127
 - Function Type 128
 - Implicit Return 130
 - Shorthand Argument Names 130
 - Trailing Closure 131
- Code Review 374, 914
- Code Signing 964
 - Automatic 965
 - Manual 966
- Code Snippets 439
- Color 502, 763
- ColorPicker 557
- Commit 900
- Computed Variable 187
- Configurations 395
- Confirmation Dialog 761
- Container 480
- continue 43
- Control Transfer Statement 43
- convenience 268
 - Schlüsselwort 211
- Convenience_INITIALIZER 210
- Core Data 833
 - Elemente 834
 - Entity 840
 - FetchedResults 859
 - FetchRequest 858
 - Funktionsweise 834

- NSManagedObject 836
- NSManagedObjectContext 836
- NSManagedObjectModel 835
- NSPersistentContainer 837
- NSPersistentStore 835
- NSPersistentStoreCoordinator 836
- Operationen 852
- Relationship 843
- Stack 837
- SwiftUI 854
- Cross-Platform 862
- CSR 949

D

Daten 467

Datenhaltung 821

DatePicker 550

- Komponenten 552
- Style 553

DatePickerComponents 552

Debug Area 381, 423

Debugging 423

Debug Navigator 371

Default Initializer 199

deinit 216

Deinitialisierung 216

Deinitializer 217

Deklarative Syntax 469

Delegate 287

Delegation 272, 286

Deployment Target 395

Derived Value 811

Designated Initializer 209

Design Pattern 286

Developer ID Certificate 984

Dictionary 77

- Shorthand Syntax 77

Discard 900

DisclosureGroup 673

Divider 683

do-catch 307

Double 50

Downcasting 297

E

Edge 747

EditButton 526

Edit-Mode 526

Editor 373

- Options 374

Entity 840

- Relationship 843

Entwicklerzertifikat 947

Enumeration 134

Environment 801

EnvironmentKey 817

EnvironmentObject 793

EnvironmentValues 802

Error 302

Error Handling 302

Exception Breakpoint 431

Existential

- any 274

Existential any 274

Explicit App ID 953

extension 254, 275

Extension 254, 275

- Computed Property 255
- Initializer 256
- Methode 255
- Nested Type 259
- Subscript 259

F

fallthrough 38

Fetch 901

FetchDescriptor 829

FetchResults 859

FetchRequest 858

FileMerge 900

fileprivate 337, 339

File-private Access 337, 339

final 224, 270

- Find Navigator 369
- FIXME 445
- Fließkommazahl 15*f.*
- Float 50
- Font 496
- Font.Design 498
- Font.Weight 499
- for 29
- Forced Unwrapping 90
- ForEach 612
 - Daten 614
 - Range 612
- for-in 27
- Form 647
- Foundation 170, 336
- Foundation-Framework 283
- Framework 5, 336
- Frameworks
 - AppKit 466
 - SwiftUI 465
 - UIKit 466
 - Uniform Type Identifiers 621
 - WatchKit 466
- Freemium Model 980
- Free Model 980
- func 111
- Funktion 111
 - globale 188
 - lokale 188
 - Name 113
 - Rückgabewert 120
 - verschachtelte 125

G

- Ganzzahl 15
- Gauge 568
 - Style 569
- Generic 314
- Generic Function 315
- Generic Type 319
- Geschäftsmodell 979
- Git 899
- GridItem 627, 637

- Grids 625
 - LazyHGrid 625
 - LazyVGrid 625
- Group 652
- GroupBox 658

H

- Hashable 289
- HSplitView 724
- HStack 574
 - Ausrichtung 574

I

- IDE 351
- Identifiable 291, 594
- if 32
- Image 513
 - Größe 518
- Implicitly Assigned Raw Value 147
- Implicitly Unwrapped Optional 95
- Implicit Return 130
- import 283, 336
- Index 54
- Info.plist 401
- init! 215
- init? 212
- Initialisierung 144, 199, 226
- Initialization Parameters 203
- Initializer 199, 226
 - Convenience Initializer 210
 - Default Initializer 199
 - Deinitializer 217
 - Designated Initializer 209
 - Failable Initializer 212
 - Memberwise Initializer 200, 346
 - Required Initializer 215, 237, 345
- Initializer Delegation 208
 - Reference Type 209
 - Value Type 208
- Inspectors 378
 - File Inspector 379

- History Inspector 380
- Quick Help Inspector 380, 413
- Instance Methods 189
- Instanz 143
- Instruments 406, 435
- Int 50
- Int8 49
- Int16 49
- Int32 49
- Int64 49
- Integer 49
 - Wertebereich 50
- Integrated Development Environment 351
- Interface 356
- internal 338, 340
- Internal Access 338, 340
- Interval Matching 40
- iOS 156, 170
- IPA 422
- is
 - Type Check Operator 298
- Issue Navigator 370

J

Jump Bar 444

K

Key Bindings 389

Key-Path 291

Key-Value Pairs 77

Klasse 151

Klassenprotokoll 281

Konsole 425

Konstante 20

- lokale 188

L

Label 561

- Style 563

Labeled Statement 45

Language 357

lazy 164, 188

LazyHGrid 625

LazyHStack 585

Lazy Stacks 585

LazyVGrid 625

LazyVStack 585

let 20

Library 368, 892

List 589

- Style 610

Localizable.strings 872

LocalizedStringKey 876

LocalizedStringResource 874

M

macOS 156, 170

MARK 445

Mehrfachvererbung 281

Mehrsprachigkeit 871

Memberwise Initializer 148, 153, 200, 346

Merge 914

Methode 148, 189

- Instanzmethode 189
- Typmethode 193

Model 467

ModelContainer 825

ModelContext 825, 828

Modifier 477

- Reihenfolge 478

Module 335

MultiDatePicker 555

mutating 192, 256, 266

N

Navigation 685

NavigationLink 686

NavigationSplitView 704

NavigationStack 686

- Titel 690

Navigator 368

- Bookmark Navigator 369

- Breakpoint Navigator 372, 430
- Debug Navigator 371, 431
- Find Navigator 369
- Issue Navigator 370
- Project Navigator 368
- Report Navigator 372
- Source Control Navigator 369
- Test Navigator 371
- Nebenläufigkeit 327
- Nested Functions 125
- Nested Type 252, 259
- nil 79, 89, 300
- NSItemProvider 622
- NSManagedObject 836
- NSManagedObjectContext 836
- NSManagedObjectModel 835
- NSPersistentContainer 837
- NSPersistentStore 835
- NSPersistentStoreCoordinator 836

O

- objectWillChange 789
- ObservableObject 783
- ObservedObject 782
- onAppear 765
- onDisappear 765
- open 339*f.*
- Open Access 339*f.*
- Open Quickly 442
- Operator
 - logischer 36
 - Ternary Conditional 34
- optional 283
- Optional 89, 142
 - entpacken 89*f.*
- Optional Binding 92, 285
- Optional Chaining 96
- Organization Identifier 356
- Organizer 421
- OutlineGroup 668
- override 222

P

- Page Indicator 720
- Paid Model 981
- Parallelität 327
- Parameter 112
 - Default Value 116
 - In-Out-Parameter 119
 - Variadic Parameter 118
- Parameter Name 114
- Pasteboard 527
- PasteButton 527
- Paymium Model 982
- Picker 547
 - Style 550
- Placeholder Type 316
- Platforms 391
- Playgrounds 446
- Popover 741
- Predicate 830
- Preview 352, 886
 - Funktionsweise 888
 - Konfiguration 890
- print 16
- private 337, 339
- Private Access 337, 339
- Product Name 355
- Profile 408
- ProgressView 564
 - Maximalbereich 565
 - Style 567
- Project Navigator 368
- Projekt 354
 - Einstellungen 394
- Property 148, 160, 769
 - Computed Property 168
 - Instance Property 183
 - Lazy Stored Property 164
 - Read-Only Computed Property 171
 - Shorthand Getter 170
 - Shorthand Setter 170
 - Stored Property 161
 - Type Property 183
- Property Default Value 206

- Property Observer 173
- Property Wrapper 177
- protocol 260
- Protocol Composition 285
- Protokoll 260
 - Class-only 281
 - Hashable 289
 - Identifiable 291
 - Initializer 268
 - Methode 264
 - Property 262
 - Subscript 267
 - Typ 272
 - Vererbung 280
 - View 471
- Provisioning Profile 948
- public 339*f.*
- Public Access 339*f.*
- Published 788
- Pull 909
- Pull 900
- Punktnotation 48
- Push 909
- Push 900

Q

- Query 831
- Quick Help Inspector 380, 413
- Quick Look 427

R

- Race Condition 330, 333
- Raw Value 141
 - Implicitly Assigned Raw Value 147
- Read-Only Computed Property 171
- Refactoring 433
- Reference Type 107, 151
 - Initializer Delegation 209
- Related Items 442
- repeat-while 31
- Report Navigator 372
- Repository 900

- required 215, 237, 269
- Required Initializer 215, 237
 - Access Level 345
- Resource Tags 399
- return 121
- Rückgabewert 120

S

- Scene 470
- SceneStorage 807
- Schema 825
- Scheme 404
 - erstellen 406
 - verwalten 407
- Schleife 27
- Schlüsselbundverwaltung 950
- Schlüssel-Wert-Paar 77
- ScrollView 664
 - Richtung 666
- Section 649
- SecureField 509
 - Style 510
- self 146, 157, 190, 193
- Set 68
- SF Symbols 514
- Sheet 725
- Shell Command 429
- Shorthand Argument Names 130
- Shorthand Getter 170
- Shorthand Setter 170
- Signed Integer 49
- Signing & Capabilities 398
- Simulator 417
- Slider 534
 - Aussehen 539
 - Wertebereich 535
- Snippets Library 439
- Source Control 899
- Source Control Navigator 369, 913
- Source File 335
- Source of Truth 811
- Spacer 679
- Speicherverwaltung 239

- sqrt() 170
- Stacks 573
 - HStack 574
 - Lazy 585
 - LazyHStack 585
 - LazyVStack 585
 - VStack 578
 - ZStack 581
- State 771
- StateObject 791
- static 184, 192
- Status 472, 767
 - Best Practices 815
 - Binding 773
 - Derived Value 811
 - EnvironmentObject 793
 - NSManagedObject 854
 - ObservedObject 782
 - Property 769
 - Source of Truth 811
 - State 771
 - StateObject 791
- Stepper 541
- Storage 357
- Stored Constant 188
- Stored Variable 186
- String 51
- String Catalog 871
- String Immutability 53
- String Interpolation 22, 58
- String Mutability 53
- Strong Reference 245
- Strong Reference Cycle 243
- struct 143
- Structure 143
- Subklasse 220
- subscript
 - Schlüsselwort 194
- Subscript 55, 193
- Subscription Model 980
- Subscript Overloading 198
- Subversion 899
- super 225
- Superklasse 220

- Swift 3
- SwiftData 824
- Swift Packages 395
- Swift Playgrounds 7, 10
- Swift Standard Library 14
- SwiftUI 465
 - Syntax 469
- Swipe-to-delete-Geste 617
- switch 37
 - Compound Case 40
 - Explicit Fallthrough 38
 - Implicit Fallthrough 38

T

- Table 643
- TableView 716
- Target 358, 397, 863
 - Zuweisung 866
- Target Membership 379
- Tastaturkurzbefehle 389
- Team 356
- Ternary Conditional Operator 34
- TestFlight 974, 986
- Test Navigator 371
- Text 494
 - Farbe 502
 - Formatierung 501
 - Schriftart 496
 - Übersetzung 503
- TextEditor 510
 - Formatierung 511
- TextField 504
 - Style 505
- Text.FontStyle 498
- throw 304
- throws 303
- TODO 445
- Toggle 529
 - Style 533
- Toolbar 748
 - Position 752
- ToolbarItem 748
- ToolbarItemGroup 749

ToolbarItemPlacement 752
try 307
try! 313
try? 311
Tuple 68, 83
tvOS 157, 170
Two-Phase Initialization 227
Typ
– Platzhalter 316
Type Alias 106
Type Annotation 23
Type Casting 106, 299
Type Cast Operator 299
Type Checking 297f.
Type Check Operator 298
Type Constraint 321
Type Inference 23
Type Method 192
Type Parameter 318
Type Property 184
Typmethode 193
Typsicherheit 22, 60, 77

U

Übersetzung 871
UIKit 336, 466
UInt 50
UInt8 49
UInt16 49
UInt32 49
UInt64 49
UI Recording 937
UI-Test 933
UI Testing Bundle 934
Unavailability Condition 870
Uniform Type Identifiers 527, 621
UnitPoint 745
unowned 249
Unowned Reference 249, 251
Unsigned Integer 49
UserDefaults 821
UTType 527, 621
UUID 291

V

Value Binding 86, 308
Value Type 107, 151
var 20
Variable 20
– globale 186
– lokale 186
Variables View 426
Vererbung 218, 280
Versionsverwaltung 899
View 467, 470f.
– Aktualisierung 472
– Anpassung 477
View-Events 765
Views
– Button 521
– ColorPicker 557
– DatePicker 550
– DisclosureGroup 673
– Divider 683
– EditButton 526
– ForEach 612
– Form 647
– Gauge 568
– Group 652
– GroupBox 658
– HSplitView 724
– HStack 574
– Image 513
– Label 561
– LazyHGrid 625
– LazyHStack 585
– LazyVGrid 625
– LazyVStack 585
– List 589
– NavigationLink 686
– NavigationStack 686
– OutlineGroup 668
– PasteButton 527
– Picker 547
– ProgressView 564
– ScrollView 664
– Section 649

- SecureField 509
- Slider 534
- Spacer 679
- Stepper 541
- TabView 716
- Text 494
- TextEditor 510
- TextField 504
- Toggle 529
- VSplitView 724
- VStack 578
- ZStack 581
- ViewThatFits 661
- Void 123
- VSplitView 724
- VStack 578
- Ausrichtung 579

W

- Wahrheitswert 16, 51
- WatchKit 466
- watchOS 157, 170
- weak 245
- Weak Reference 245, 251
- where 88, 310

- while 29
- Wildcard App ID 953
- WLAN 420
- Workspace 354
- Worldwide Developers Conference 3
- WWDC 3

X

- Xcode 351
- Einstellungen 382
- XCTest 918
- XCTestCase 921
- XCUIApplication 934
- XCUIElement 935
- XCUIElementQuery 936
- XCUIElementTypeQueryProvider 936

Z

- Zeichenkette 16, 51
- ZStack 581
- Ausrichtung 583
- Zwei-Phasen-Initialisierung 227
- Zwischenablage 527