

Swift 3 im Detail

Einführung und Sprachreferenz

Bearbeitet von
Thomas Sillmann

1. Auflage 2017. Buch inkl. Online-Nutzung. 446 S. Softcover
ISBN 978 3 446 45072 1
Format (B x L): 17,7 x 24,4 cm
Gewicht: 922 g

[Weitere Fachgebiete > EDV, Informatik > Betriebssysteme](#)

schnell und portofrei erhältlich bei



Die Online-Fachbuchhandlung beck-shop.de ist spezialisiert auf Fachbücher, insbesondere Recht, Steuern und Wirtschaft. Im Sortiment finden Sie alle Medien (Bücher, Zeitschriften, CDs, eBooks, etc.) aller Verlage. Ergänzt wird das Programm durch Services wie Neuerscheinungsdienst oder Zusammenstellungen von Büchern zu Sonderpreisen. Der Shop führt mehr als 8 Millionen Produkte.

HANSER



Leseprobe

Thomas Sillmann

Swift 3 im Detail

Einführung und Sprachreferenz

ISBN (Buch): 978-3-446-45072-1

Weitere Informationen oder Bestellungen unter

<http://www.hanser-fachbuch.de/978-3-446-45072-1>

sowie im Buchhandel.

Inhalt

1	Die Programmiersprache Swift	1
1.1	Die Geschichte von Swift	2
1.2	Swift-Updates	3
1.3	Voraussetzungen zur Nutzung von Swift	4
1.4	Installation von Swift	5
1.4.1	macOS	6
1.4.2	Linux	7
1.5	Xcode	10
1.5.1	Erstellen von Dateien und Projekten	11
1.5.2	Der Aufbau von Xcode	14
1.5.3	Einstellungen	20
1.6	Playgrounds	20
1.6.1	Erstellen eines Playgrounds	21
1.6.2	Aufbau eines Playgrounds	23
1.6.3	Pages, Sources und Resources	27
1.6.4	Playground-Formatierungen	29
1.6.5	Swift Playgrounds-App für das iPad	38
1.7	Weitere Code-Editoren zur Arbeit mit Swift	40
1.7.1	Visual Studio Code	40
1.7.2	Syntra Small	42
1.7.3	IBM Swift Sandbox	43
1.8	Swift-Ressourcen und weiterführende Informationen	44
2	Grundlagen der Programmierung	47
2.1	Grundlegendes	47
2.1.1	Swift Standard Library	47
2.1.2	print	49
2.1.3	Befehle und Semikolons	49
2.1.4	Operatoren	50
2.2	Variablen und Konstanten	52
2.2.1	Erstellen von Variablen und Konstanten	52

2.2.2	Variablen und Konstanten in der Konsole ausgeben	53
2.2.3	Type Annotation und Type Inference	54
2.2.4	Gleichzeitiges Erstellen und Deklarieren mehrerer Variablen und Konstanten	55
2.2.5	Namensrichtlinien	56
2.3	Kommentare	57
3	Schleifen und Abfragen	59
3.1	Schleifen	59
3.1.1	for-in	59
3.1.2	while	61
3.1.3	repeat-while	62
3.2	Abfragen	63
3.2.1	if	63
3.2.2	switch	67
3.2.3	guard	71
3.3	Control Transfer Statements	73
3.3.1	Anstoßen eines neuen Schleifendurchlaufs mit continue	73
3.3.2	Verlassen der kompletten Schleife mit break	73
3.3.3	Weitere Control Transfer Statements	74
3.3.4	Labeled Statements	74
4	Typen in Swift	77
4.1	Integer	79
4.2	Fließkommazahlen	80
4.3	Bool	81
4.4	String	81
4.4.1	Erstellen eines Strings	81
4.4.2	Zusammenfügen von Strings	82
4.4.3	Character auslesen	83
4.4.4	Character mittels Index auslesen	84
4.4.5	Character entfernen und hinzufügen	86
4.4.6	Anzahl der Character zählen	87
4.4.7	Präfix und Suffix prüfen	87
4.4.8	String Interpolation	88
4.5	Array	88
4.5.1	Erstellen eines Arrays	89
4.5.2	Zusammenfügen von Arrays	90
4.5.3	Inhalte eines Arrays leeren	91
4.5.4	Prüfen, ob ein Array leer ist	91
4.5.5	Anzahl der Elemente eines Arrays zählen	92
4.5.6	Zugriff auf die Elemente eines Arrays	92
4.5.7	Neue Elemente zu einem Array hinzufügen	93
4.5.8	Bestehende Elemente aus einem Array entfernen	93

4.5.9	Bestehende Elemente eines Arrays ersetzen	94
4.5.10	Alle Elemente eines Arrays auslesen und durchlaufen	95
4.6	Set	96
4.6.1	Erstellen eines Sets	97
4.6.2	Inhalte eines bestehenden Sets leeren	98
4.6.3	Prüfen, ob ein Set leer ist	98
4.6.4	Anzahl der Elemente eines Sets zählen	98
4.6.5	Element zu einem Set hinzufügen	99
4.6.6	Element aus einem Set entfernen	99
4.6.7	Prüfen, ob ein bestimmtes Element in einem Set vorhanden ist	99
4.6.8	Alle Elemente eines Sets auslesen und durchlaufen	100
4.6.9	Sets miteinander vergleichen	100
4.6.10	Neue Sets aus bestehenden Sets erstellen	103
4.7	Dictionary	105
4.7.1	Erstellen eines Dictionary	105
4.7.2	Prüfen, ob ein Dictionary leer ist	106
4.7.3	Anzahl der Schlüssel-Wert-Paare eines Dictionary zählen	107
4.7.4	Wert zu einem Schlüssel eines Dictionaries auslesen	107
4.7.5	Neues Schlüssel-Wert-Paar zu Dictionary hinzufügen	108
4.7.6	Bestehendes Schlüssel-Wert-Paar aus Dictionary entfernen	108
4.7.7	Bestehendes Schlüssel-Wert-Paar aus Dictionary verändern	109
4.7.8	Alle Schlüssel-Wert-Paare eines Dictionary auslesen und durchlaufen	109
4.8	Tuple	110
4.8.1	Zugriff auf die einzelnen Elemente eines Tuple	112
4.8.2	Tuple und switch	112
4.9	Optional	115
4.9.1	Deklaration eines Optionals	116
4.9.2	Zugriff auf den Wert eines Optionals	117
4.9.3	Optional Binding	119
4.9.4	Implicitly Unwrapped Optional	120
4.9.5	Optional Chaining	121
4.9.6	Optional Chaining über mehrere Eigenschaften und Funktionen	126
4.10	Any und AnyObject	130
4.11	Type Alias	130
4.12	Value Type versus Reference Type	131
4.12.1	Reference Types auf Gleichheit prüfen	133
5	Funktionen	135
5.1	Funktionen mit Parametern	136
5.1.1	Argument Labels und Parameter Names	137
5.1.2	Default Value für Parameter	140
5.1.3	Variadic Parameter	141
5.1.4	In-Out Parameter	142
5.2	Funktionen mit Rückgabewert	143

5.3	Function Types	145
5.3.1	Funktionen als Variablen und Konstanten	146
5.4	Verschachtelte Funktionen	148
5.5	Closures	148
5.5.1	Closures als Parameter von Funktionen	150
5.5.2	Trailing Closures	153
5.5.3	Autoclosures	154
6	Enumerations, Structures und Classes	157
6.1	Enumerations	157
6.1.1	Enumerations und switch	160
6.1.2	Associated Values	161
6.1.3	Raw Values	163
6.2	Structures	166
6.2.1	Erstellen von Structures und Instanzen	166
6.2.2	Eigenschaften und Funktionen	167
6.3	Classes	172
6.3.1	Erstellen von Klassen und Instanzen	173
6.3.2	Eigenschaften und Funktionen	173
6.4	Enumeration vs. Structure vs. Class	175
6.4.1	Gemeinsamkeiten und Unterschiede	176
6.4.2	Wann nimmt man was?	177
6.5	self	178
7	Eigenschaften und Funktionen von Typen	181
7.1	Properties	181
7.1.1	Stored Property	182
7.1.2	Lazy Stored Property	184
7.1.3	Computed Property	188
7.1.4	Read-Only Computed Property	190
7.1.5	Property Observer	192
7.1.6	Type Property	195
7.2	Globale und lokale Variablen	197
7.3	Methoden	200
7.3.1	Instance Methods	201
7.3.2	Type Methods	203
7.4	Subscripts	204
8	Initialisierung	209
8.1	Aufgabe der Initialisierung	210
8.2	Erstellen eigener Initializer	211
8.3	Initializer Delegation	216
8.3.1	Initializer Delegation bei Value Types	217
8.3.2	Initializer Delegation bei Reference Types	218

8.4 Failable Initializer	220
8.5 Required Initializer	223
8.6 Deinitialisierung	224
9 Vererbung	227
9.1 Überschreiben von Eigenschaften und Funktionen einer Klasse	230
9.2 Überschreiben von Eigenschaften und Funktionen einer Klasse verhindern	233
9.3 Zugriff auf die Superklasse	233
9.4 Initialisierung und Vererbung	234
9.4.1 Zwei-Phasen-Initialisierung	235
9.4.2 Überschreiben von Initializern	241
9.4.3 Vererbung von Initializern	244
9.4.4 Required Initializer	244
10 Speicherverwaltung mit ARC	247
10.1 Strong Reference Cycles	250
10.1.1 Weak References	252
10.1.2 Unowned References	255
10.1.3 Weak Reference vs. Unowned Reference	257
11 Weiterführende Sprachmerkmale von Swift	259
11.1 Nested Types	259
11.2 Extensions	261
11.2.1 Computed Properties	261
11.2.2 Methoden	262
11.2.3 Initializer	263
11.2.4 Subscripts	265
11.2.5 Nested Types	265
11.3 Protokolle	266
11.3.1 Deklaration von Eigenschaften und Funktionen	268
11.3.2 Der Typ eines Protokolls	277
11.3.3 Protokolle und Extensions	279
11.3.4 Vererbung in Protokollen	284
11.3.5 Class-only-Protokolle	285
11.3.6 Optionale Eigenschaften und Funktionen	286
11.3.7 Protocol Composition	289
11.3.8 Delegation	290
12 Type Checking und Type Casting	295
12.1 Type Checking mit <code>is</code>	298
12.2 Type Casting mit <code>as</code>	299
13 Error Handling	303
13.1 Deklaration und Feuern eines Fehlers	303

13.2 Reaktion auf einen Fehler	307
13.2.1 Mögliche Fehler mittels do-catch auswerten	308
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	315
14.1 Generic Functions	316
14.2 Generic Types	319
14.3 Type Constraints	322
14.4 Associated Types	322
15 Dateien und Interfaces	327
15.1 Modules und Source Files	327
15.2 Access Control	328
15.2.1 Access Level	328
15.2.2 Explizite und implizite Zuweisung eines Access Levels	332
15.2.3 Besonderheiten	333
16 Cocoa, Objective-C und C	339
16.1 Interoperability	340
16.1.1 Swift und Cocoa	340
16.1.2 Swift und Objective-C	361
16.1.3 Swift und C	374
16.2 Mix and Match	377
16.2.1 Mix and Match innerhalb eines App-Targets	378
16.2.2 Mix and Match innerhalb eines Framework-Targets	380
16.3 Migration	381
17 Objektorientierte vs. protokollorientierte Programmierung.....	383
17.1 Objektorientierte Programmierung	384
17.1.1 Praxis	385
17.1.2 Vor- und Nachteile	388
17.2 Protokollorientierte Programmierung	389
17.2.1 Praxis	389
17.2.2 Vor- und Nachteile	392
17.3 Fazit	393
18 Weitere Sprachmerkmale und Profi-Wissen	395
18.1 Zahlenliterale	395
18.2 Fortgeschrittene Operatoren	396
18.2.1 Ternary Conditional Operator	396
18.2.2 Nil-Coalescing Operator	398

18.2.3 Unary Minus- und Unary Plus-Operatoren	399
18.2.4 Bitweise Operatoren	399
18.2.5 Operator Methods	403
18.2.6 Eigene Operatoren erstellen	409
18.3 Option Sets	411
18.4 Closures	416
18.4.1 Escaping Closures	416
18.4.2 Closure Capture List	419
18.5 Recursive Enumerations	422
18.6 Optionals im Detail	424
18.7 Generic Where Clause	426
18.8 Dynamic Method Lookup	426
18.9 Weitere Objective-C-Makros	427
18.9.1 NS_SWIFT_NAME	428
18.9.2 NS_SWIFT_UNAVAILABLE	428
Index	429

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 Inhalte der verschiedenen Abschnitte sowie der folgenden Kapitel beispielsweise in einem Playground zu erproben und auszuprobieren, um so möglichst schnell ein Gefühl für die Sprache zu bekommen und selbst aktiv Code zu schreiben.

■ 2.1 Grundlegendes

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.

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. 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.

The screenshot shows a web browser window displaying the Swift Standard Library documentation. The title is "Swift Standard Library". Below the title, there is a brief description: "Solve complex problems, and write high-performance, readable code. Access the fundamental protocols, data types, and algorithms common to all Swift programs." To the right of this text are two buttons: "Language Swift" and "On This Page Overview Symbols". Below the main content area, there is a section titled "Overview" with a sub-section "The Swift standard library defines a base layer of functionality for writing Swift programs, including:" followed by a bulleted list of features.

Overview

The Swift standard library defines a base layer of functionality for writing Swift programs, including:

- Fundamental data types such as `Int`, `Double`, and `String`
- Common data structures such as `Array`, `Dictionary`, and `Set`
- Functions and methods such as `print(_:separator:terminator:)`, `sorted()`, and `abs(_:_)`
- Protocols that describe abstractions such as `Collection` and `Equatable`.
- Protocols used to customize operations that are available to all types, such as `CustomDebugStringConvertible` and `CustomReflectable`.

• Protocols used to provide implementations that would otherwise require boilerplate

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

Tabelle 2.1 Auswahl grundlegender Typen der Swift Standard Library

Fundamental Type	Beschreibung	Beispiele
<code>Int</code>	Ein Integer (<code>Int</code>) stellt eine Ganzzahl dar.	19 99
<code>Float</code>	Bei <code>Float</code> handelt es sich um eine Fließkommazahl	19.99 49.94
<code>Double</code>	Auch bei <code>Double</code> handelt es sich um eine Fließkommazahl, allerdings ist der Wertebereich von <code>Double</code> deutlich größer als der von <code>Float</code> ; entsprechend belegt ein <code>Double</code> auch mehr Speicherplatz im System als ein <code>Float</code> .	99.19 94.49
<code>Bool</code>	Bei <code>Bool</code> handelt es sich um einen sogenannten Wahrheitswert, dieser kann somit entweder wahr oder falsch (<code>true</code> oder <code>false</code>) sein.	<code>true</code> <code>false</code>
<code>String</code>	Ein <code>String</code> repräsentiert eine Zeichenkette.	"Mein Name ist Thomas Sillmann."

Fundamental Type	Beschreibung	Beispiele
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 dabei 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 Falle mehrere Befehle auch alle jeweiligen Ausgaben) am Ende als Kommentare 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 und zwingend notwendig, wenn Sie *mehrere* Befehle in einer Zeile schreiben möchten (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 innerhalb der Zeile muss wiederum nicht zwingend mit einem Semikolon abgeschlossen werden, kann es aber optional.



Semikolon – ja oder nein?

Womöglich fragen Sie sich nach diesem Abschnitt, was nun die bessere Lösung ist; Befehle mit einem Semikolon abschließen 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.
>	Vergleichsoperator	Prüft, ob der Wert links vom Operator größer dem rechts vom Operator ist.
>=	Vergleichsoperator	Prüft, ob der Wert links vom Operator größer oder gleich dem rechts vom Operator ist.
+	Berechnungsoperator	Dient zur Durchführung von Additionen.
-	Berechnungsoperator	Dient zur Durchführung von Subtraktionen.
*	Berechnungsoperator	Dient zur Durchführung von Multiplikationen.
/	Berechnungsoperator	Dient zur Durchführung von Divisionen.
%	Berechnungsoperator	Dient zur Berechnung des Rests bei einer Division.
+=	Berechnungsoperator	Erhöht den Wert links vom Operator um den Wert rechts vom Operator.
-=	Berechnungsoperator	Verringert den Wert links vom Operator um den Wert rechts vom Operator.
&&	Logischer Operator	Verknüpft zwei Bedingungen mittels UND; ist eine von ihnen <code>false</code> , ist auch das Ergebnis <code>false</code> .
	Logischer Operator	Verknüpft zwei Bedingungen mittels ODER; ist eine von beiden <code>true</code> , ist auch das Ergebnis <code>true</code> .
!	Logischer Operator	Kehrt einen Wahrheitswert um (<code>true</code> wird <code>false</code> , <code>false</code> wird <code>true</code>).
...	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 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 rechts vom Operator.
??	Nil-Operator	Prüft den optionalen Wert links vom Operator. Ist dieser <code>nil</code> , 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 je 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 Änderns 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 einer 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 deklariere ihn als Konstante! Erstellen Sie daher im Zweifelsfall lieber eine unveränderliche Konstante als eine Variable in Swift. 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 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 zugewiesen. 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 bei der Erstellung 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:

1. Wenn Sie einer neuen Variable oder Konstanten bei deren Deklaration noch keinen Wert zuweisen, müssen Sie in jedem Fall den gewünschten Typ für diese Variable oder Konstante angeben (so wie in Listing 2.9); andernfalls kommt es zu einem Compiler-Fehler.
2. 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, sowohl mehrere Variablen und Konstanten direkt in einem Befehl zu erstellen und ihnen dabei auf Wunsch auch 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 kommasgetrennt 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 auf einmal von ein und demselben Typ 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 π = 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 durchaus 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 ein beliebtes und zugleich sehr wichtiges Mittel in der Programmierung 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 einzeiliger 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 einzeiliger 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 einzeiliger 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. */
```

Index

Symbol

^ 402
& 401
<< 413
| 402
~ 400
0b 395
0o 395
0x 395
@autoreleasepool 358
@available 360
#available 359
@escaping 416
#keyPath 355, 370
_Nonnull 363
_Nullable 363
_Null_unspecified 363
@objc 287, 349, 370, 380
#selector 368
.swift 327

A

Abfrage 63
Access Control 328
Access Level 328
- Explizite Zuweisung 332
- Implizite Zuweisung 332
alloc 365
Any 130, 298, 371
AnyObject 130, 298, 348
API Availability 359
Apple 339
ARC 224, 247
Argument Label 137
Array 49, 88
- mutable 91
- Shorthand Syntax 89
as 299
as! 299
as? 299

associatedtype 323
Associated Type 322
Associated Values 161
Automatic Bridging 341
Automatic Reference Counting 224, 247
autoreleasepool
- Funktion 358
Autorelease Pool 358

B

Basisklasse 230
Binäres Zahlensystem 399
Binary Operator 404
Bitmaske 413
Bitwise Left Shift Operator 413
Bitwise Operator 399
- AND 401
- NOT 400
- OR 402
- XOR 402
Block 365
Bool 48, 81
break 69, 73
Bridging 341

C

C 374, 400
- Programmiersprache 339
Category 367
Character 83
Chris Lattner 2
class 285, 371
Class 172
Class Factory Method 365
Class-Protocol 285
Closure 148
- Autoclosure 154
- Default Value 149
- Function Type 150

- Implicit Return 152
- Shorthand Argument Name 152
- Trailing Closure 153
- Closure Capture List 420
- Cocoa 339f.
- Compound Assignment Operator 405
- Computed Variable 198
- Context 352
- continue 73
- Control Transfer Statement 73
- convenience 274
 - Schlüsselwort 220
- Convenience Initializer 366
- Core Foundation 343

D

- Datentyp
 - primitiver 374
- Debug Area
 - Xcode 19
- Default Initializer 209
- deinit 224
- Deinitialisierung 224
- Delegate 290
- Delegation 278, 290, 344
- Design Pattern 290, 344
- Dictionary 49, 105
 - Shorthand Syntax 105
- do-catch 308
- Double 48, 80
- Downcasting 297
- dynamic 351
- Dynamic Method Lookup 426

E

- Editor Area
 - Xcode 16f.
- Enumeration 375
 - Recursive Enumeration 422
- Equivalence Operator 407
- Error 303, 357
- Error Handling 303, 357
- Escaping Closure 416
- extension 261, 280
- Extension 261, 279, 367
 - Computed Property 261
 - Initializer 263
 - Methode 262
 - Nested Type 265
 - Subscript 265

F

- Failable Initializer 366
- falthrough 69

- fileprivate 329, 331
- File-private Access 329, 331
- final 233, 275
- Fließkommazahl 48
- Float 48, 80
- for 61
- for-in 59
- Forced Unwrapping 117
- Foundation 189, 328, 341, 343, 351
- Foundation-Framework 287
- Framework 327
- func 135
- Function Type 145, 365
- Funktion 135
 - globale 200
 - lokale 200
 - Name 137
 - verschachtelte 148

G

- Ganzzahl 48
- Generic 315, 367
- Generic Function 316
- Generic Type 319
- Generic Where Clause 426

I

- IBM Swift Sandbox 43
- id 371
- if 63
- Implicit Return 152
- Implicitly Assigned Raw Value 165
- Implicitly Unwrapped Optional 120
- import 287, 327
- Index 84
- indirect 422
- infix 410
- init! 223
- init? 221
- Initialisierung 167, 209, 234
- Initialization Parameters 213
- Initializer 209, 234, 365
 - Convenience Initializer 218
 - Default Initializer 209
 - Deinitializer 224
 - Designated Initializer 218
 - Failable Initializer 220
 - Memberwise Initializer 210, 337
 - Required Initializer 223, 244, 336
- Initializer Delegation 216
 - Reference Type 218
 - Value Type 217
- inout 405
- Installation 5
 - Linux 7

- macOS 6
- Instance Methods 201
- Instanz 166
- Int 48, 80
 - Binär 395
 - Dezimal 395
 - Hexadezimal 395
 - Oktal 395
- Int8 79
- Int16 79
- Int32 79
- Int64 79
- Integer 79
 - Wertebereich 79
- internal 329, 331
- Internal Access 329, 331
- Interoperability 339f.
- Interval Matching 70
- Introspection 357
- iOS 178, 189, 340, 360
- iOSApplicationExtension 360
- is 298, 357

K

- Kategorie 367
- Key-Path 351, 369
- Key-Value Observing 351, 369
- Key-Value Pairs 105
- Klasse 172
 - abstrakte 387
- Klassenprotokoll 285
- Konstante 52
 - lokale 199
- KVO 351

L

- Labeled Statement 75
- lazy 185, 199, 357
- Lazy Initialization 356
- let 52
- Lightweight Generic Parameterization 367
- Linux 4
- Localization 358

M

- macOS 4, 178, 189, 340, 360
- macOSApplicationExtension 360
- Makro 427
- Mehrfachvererbung 285
- Memberwise Initializer 170, 174, 210, 337
- Methode 170, 200, 362
 - Instanzmethode 201
 - Typmethode 204

- Migration 339
- Mix and Match 339, 377
 - App-Target 378
 - Framework-Target 380
- Module 327
- Mutable-Klassen 342
- mutating 203, 263, 271

N

- Navigator Area
 - Playground 26
 - Xcode 16
- Nested Functions 148
- Nested Type 259, 265
- nil 107, 116, 299
- Nil-Coalescing Operator 398
- nonnull 363, 371
- NS_ASSUME_NONNULL_BEGIN 364
- NS_ASSUME_NONNULL_END 364
- NS_ENUM 375
- NS_EXTENSIBLE_STRING_ENUM 376
- NS_STRING_ENUM 375
- NS_SWIFT_NAME 428
- NS_SWIFT_UNAVAILABLE 428
- NSError 357
- NSKeyValueObservingOptions 352
- NSObject 351
- null_unspecified 363
- Nullability 363
- nullable 363, 371

O

- Objective-C 339f.
- Objective-C Bridging Header 378
- Objekt 385
- Objektorientierte Programmierung 383f.
- Observer 351
- Observing-Options 352
- OOP 383f.
- open 330f.
- Open Access 330f.
- Operator 396
 - logischer 66
- Operator Method 403
- Option Set 411
- optional 287
- Optional 115, 165, 363, 424
 - entpacken 116f.
- Optional Binding 119, 289
- Optional Chaining 121, 347
- override 230

P

Page
 - Playground 27
 Parameter 136
 - Default Value 140
 - In-Out Parameter 142
 - Variadic Parameter 141
 Parameter Name 137
 Placeholder Type 317
 Playground 20
 - Aufbau 23
 - Dateien hinzufügen 27
 - Erstellen 21
 - Formatierungen 29
 - Navigator Area 26
 - Page 27
 - Utilities Area 27
 postfix 406, 410
 Postfix-Operator 406
 prefix 406, 410
 Prefix-Operator 406
 Primitive Type 374
 private 328, 331
 Private Access 328, 331
 Product Module Name 378
 Product Name 378
 Programmierparadigma 383
 Programmierung
 - objektorientierte 383
 - protokollorientierte 383
 Project Navigator
 - Xcode 16
 Property 170, 181, 361
 - Computed Property 188
 - Instance Property 195
 - Lazy Stored Property 184
 - Read-Only Computed Property 190
 - Stored Property 182
 - Type Property 195
 Property Default Value 215
 Property Observer 192
 protocol 267
 Protocol Composition 289, 391
 Protokoll 266, 389
 - Class-only 285
 - Initializer 274
 - Methode 270
 - Property 268
 - Subscript 273
 - Typ 277
 - Vererbung 284
 Protokollorientierte Programmierung 383, 389
 public 330f.
 Public Access 330f.
 Punktnotation 78

Q

Quick Look 24

R

RawRepresentable 377
 Raw Value 163
 - Implicitly Assigned Raw Value 165
 Read-Only Computed Property 190
 Recursive Enumeration 422
 Reference Type 131, 172
 REPL 4
 required 223, 244, 275
 Required Initializer 223
 Resources
 - Ordner, Playground 28
 return 144
 Rückgabewert 143

S

Schleife 59
 Schlüssel-Wert-Paar 105
 Selector 348, 368
 self 169, 178, 201, 204
 Set 96
 Shorthand Argument Names 152
 Signed Integer 79
 Singleton 355
 Sources
 - Ordner, Playground 28
 Source File 327
 Speicheradresse 352
 Speicherverwaltung 247
 sqrt() 189
 static 195, 203, 403
 Stored Constant 199
 Stored Variable 198
 String 48, 81
 String Immutability 82
 String Interpolation 54, 88
 String Mutability 82
 Strong Reference 252
 Strong Reference Cycle 250, 419
 struct 166
 Structure 166, 376
 Subklasse 229
 subscript
 - Schlüsselwort 204
 Subscript 85, 204
 Subscript Overloading 208
 super 233
 Superklasse 229
 Swift 1
 Swift.org 1, 44
 Swift Playgrounds 4

- App 38
- iPad-App 20
- Swift Standard Library 47, 341f.
- Swift Type Compatibility 371
- switch 67
 - Compound Case 70
 - Explicit Fallthrough 69
 - Implicit Fallthrough 69
- Syntra Small 42

T

- Target-Action 348
- Ternary Conditional Operator 396
- throw 305
- throws 304, 371
- try 307f.
- try! 313
- try? 311
- Tuple 96, 110
- tvOS 178, 189, 340, 360
- tvOSApplicationExtension 360
- Two-Phase Initialization 235
- Typ
 - Platzhalter 317
- Type Alias 130
- Type Annotation 54, 342
- Type Cast Operator 299
- Type Casting 130, 299
- Type Check Operator 298
- Type Checking 296, 298, 357
- Type Constraint 322
- Type Inference 54
- Type Method 203
- Type Parameter 319
- Type Property 195
- typedef 375
- Typmethode 204
- Typsicherheit 54, 89, 105

U

- UIButton
 - Klasse 348
- UIKit 328, 347f.
- UInt 80
- UInt8 79, 400
- UInt16 79
- UInt32 79
- UInt64 79
- UITableView
 - Klasse 347
- UITableViewDataSource
 - Protokoll 347
- UITableViewDelegate
 - Protokoll 347
- Unary Minus Operator 399

- Unary Plus Operator 399
- unowned 255
- Unowned Reference 255, 257
- Unsigned Integer 79
- Utilities Area
 - Playground 27
 - Xcode 17

V

- Value Binding 113, 309
- Value Type 131, 172
- var 52
- Variable 52
 - globale 197
 - lokale 197
- Variables View
 - Xcode 19
- Vererbung 284, 384
- Visual Studio Code 40
- Void 146

W

- Wahrheitswert 48, 81
- watchOS 178, 189, 340, 360
- watchOSApplicationExtension 360
- weak 252
- Weak Reference 252, 257
- where 115, 310, 426
- while 61
- Worldwide Developers Conference 2
- WWDC 2, 389

X

- Xcode 10, 339
 - Debug Area 19
 - Editor Area 16f.
 - Einstellungen 20
 - Konsole 19
 - Navigator Area 16
 - Project Navigator 16
 - Toolbar 15
 - Utilities Area 17
 - Variables View 19
- Xcode-Generated Header 379

Z

- Zahlenliteral 395
- Zahlensystem
 - binäres 399
- Zeichenkette 48, 81
- Zwei-Phasen-Initialisierung 235