

HANSER



Leseprobe

zu

Programmieren lernen mit Kotlin

von Christian Kohls und Alexander Dobrynin

Print-ISBN: 978-3-446-47712-4

E-Book-ISBN: 978-3-446-47849-7

E-Pub-ISBN: 978-3-446-48005-6

Weitere Informationen und Bestellungen unter

<https://www.hanser-kundencenter.de/fachbuch/artikel/9783446477124>

sowie im Buchhandel

© Carl Hanser Verlag, München

Inhalt

Vorwort	XVII
1 Einführung	1
1.1 Eine Sprache für viele Plattformen	2
1.2 Deshalb ist Kotlin so besonders	3
1.3 Darauf dürfen Sie sich freuen	4
Teil I: Konzeptioneller Aufbau von Computern und Software	7
2 Komponenten eines Computers	9
2.1 Beliebige Daten als binäre Zahlen	9
2.2 Wie Zahlen in Texte, Bilder und Animationen umgewandelt werden	12
2.3 Zahlen als ausführbarer Code	13
3 Zugriff auf den Speicher	15
3.1 Organisation des Speichers	16
3.2 Daten im Speicher und Datenverarbeitung im Prozessor	17
3.3 Heap und Stack	18
3.4 Programme als Code schreiben statt als Zahlenfolgen	18
4 Interpreter und Compiler	21
4.1 Virtuelle Maschinen, Bytecode und Maschinencode	22
4.2 Kotlin – eine Sprache, viele Plattformen	23
5 Syntax, Semantik und Pragmatik	25
5.1 Syntax	25
5.2 Semantik	26
5.3 Pragmatik	28
6 Eingabe – Verarbeitung – Ausgabe	31

7	Los geht's	33
7.1	Integrierte Entwicklungsumgebung	34
7.2	Projekt anlegen	36
Teil II: Grundlagen des Programmierens		39
8	Anweisungen und Ausdrücke	41
8.1	Ausdrücke	42
8.1.1	Literale	43
8.1.2	Operationen	44
8.1.3	Variablen und Funktionsaufrufe	46
8.2	Evaluation von Ausdrücken	47
8.2.1	Evaluieren von Operatoren	47
8.2.2	Evaluieren von Funktionen	48
8.2.3	Evaluieren von Variablen	49
8.3	Zusammenspiel von Werten und Typen	50
8.3.1	Typprüfungen durch den Compiler	51
8.3.2	Typen als Bausteine	52
9	Basis-Datentypen	55
9.1	Numerics	56
9.2	Characters und Strings	60
9.3	Booleans	61
9.4	Arrays	62
9.5	Unit	64
9.6	Any	67
9.7	Nothing	68
9.8	Zusammenfassung	69
10	Variablen	71
10.1	Deklaration, Zuweisung und Verwendung	72
10.2	Praxisbeispiel	75
10.2.1	Relevante Informationen extrahieren	75
10.2.2	Das Problem im Code lösen	76
10.2.3	Zusammenfassung	78
11	Kontrollstrukturen	79
11.1	Fallunterscheidungen mit if	79
11.1.1	if-Anweisung	79
11.1.2	if-Ausdruck	81

11.2	Pattern-Matching mit when	83
11.2.1	Interpretieren von Werten	85
11.2.2	Typüberprüfungen	86
11.2.3	Überprüfen von Wertebereichen	88
11.2.4	Abbilden von langen if-else-Blöcken	90
11.3	Wiederholung von Code mit while-Schleifen	92
11.3.1	Zählen, wie oft etwas passiert	93
11.3.2	Gameloop	94
11.4	Iterieren über Datenstrukturen mit for-Schleifen	96
11.4.1	Iteration mit Arrays	97
11.4.2	Iteration mit Ranges	98
11.4.3	Geht das alles nicht auch mit einer while-Schleife?	99
11.5	Zusammenfassung	100
12	Funktionen	101
12.1	Top-Level- und Member-Functions	101
12.2	Funktionsaufrufe (Applikation)	102
12.3	Syntax	103
12.4	Funktionsdefinition (Deklaration)	105
12.5	Funktionen als Abstraktion	106
12.6	Scoping	108
12.7	Rekursive Funktionen	110
12.7.1	Endlose Rekursion	110
12.7.2	Terminierende Rekursion	110
12.7.3	Rekursion vs. Iteration	112
12.7.4	Von Iteration zur Rekursion	113
12.8	Shadowing von Variablen	114
12.9	Inline-Funktionen	115
12.10	Pure Funktionen und Funktionen mit Seiteneffekt	116
12.10.1	Das Schlechte an Seiteneffekten	117
12.10.2	Ohne kommen wir aber auch nicht aus	120
12.10.3	Was denn nun?	121
12.11	Die Ideen hinter Funktionaler Programmierung	122
12.12	Lambdas	123
12.13	Closures	126
12.14	Funktionen höherer Ordnung	128
12.14.1	Funktionen, die Funktionen als Parameter akzeptieren	128
12.14.2	Funktionen, die Funktionen zurückgeben	130
12.15	Zusammenfassung	137
12.16	Das war's	137

Teil III: Objektorientierte Programmierung	139
13 Was sind Objekte?	141
14 Klassen	145
14.1 Eigene Klassen definieren	145
14.2 Konstruktoren	147
14.2.1 Aufgaben des Konstruktors	149
14.2.2 Primärer Konstruktor	149
14.2.3 Parameter im Konstruktor verwenden	150
14.2.4 Initialisierungsblöcke	150
14.2.5 Klassen ohne expliziten Konstruktor	151
14.2.6 Zusätzliche Eigenschaften festlegen	151
14.2.7 Klassen mit sekundären Konstruktoren	152
14.2.8 Default Arguments	153
14.2.9 Named Arguments	154
14.3 Funktionen und Methoden	155
14.3.1 Objekte als Parameter	155
14.3.2 Methoden: Funktionen auf Objekten ausführen	156
14.3.3 Von Funktionen zu Methoden	158
14.4 Datenkapselung	160
14.4.1 Setter und Getter	161
14.4.2 Berechnete Eigenschaften	163
14.4.3 Methoden in Eigenschaften umwandeln	163
14.4.4 Sichtbarkeitsmodifikatoren	165
14.5 Spezielle Klassen	167
14.5.1 Daten-Klassen	167
14.5.2 Enum-Klassen	172
14.5.3 Singuläre Objekte	176
14.5.4 Daten-Objekte	179
14.6 Verschachtelte Klassen	180
14.6.1 Statische Klassen	181
14.6.2 Innere Klassen	182
14.6.3 Lokale innere Klassen	184
14.6.4 Anonyme innere Objekte	184
14.7 Inline-Value-Klassen	185
14.8 Klassen und Objekte sind Abstraktionen	188
14.9 Zusammenfassung	189

15	Movie Maker – Ein Simulationsspiel	191
15.1	Überlegungen zur Klassenstruktur	192
15.1.1	Eigenschaften und Methoden von Movie	193
15.1.2	Eigenschaften und Methoden von Director	194
15.1.3	Eigenschaften und Methoden von Actor	195
15.1.4	Genre als Enum	195
15.1.5	Objektstruktur	196
15.2	Von der Skizze zum Programm	197
15.2.1	Movie-Maker-Projekt anlegen	197
15.2.2	Genre implementieren	198
15.2.3	Actor und Director implementieren	198
15.2.4	Erfahrungszuwachs bei Fertigstellung eines Films	200
15.3	Komplexe Objekte zusammensetzen	201
15.3.1	Skills als eine Einheit zusammenfassen	201
15.3.2	Begleit-Objekt für Skills	202
15.3.3	Objektkomposition und Objektaggregation	203
15.3.4	Zusammensetzung der Klasse Movie	205
15.3.5	Film produzieren	207
15.3.6	Ein Objekt für Spieldaten	208
15.3.7	Code zum Projekt	209
Teil IV:	Vererbung und Polymorphie	211
16	Vererbung	213
16.1	Vererbungsbeziehung	214
16.2	Klassenhierarchien	216
16.3	Eigenschaften und Methoden vererben	217
16.4	Zusammenfassung	219
17	Polymorphie	221
17.1	Überschreiben von Methoden	222
17.1.1	Eine Methode unterschiedlich überschreiben	222
17.1.2	Dynamische Bindung	223
17.1.3	Überschreiben eigener Methoden	224
17.1.4	Überladen von Methoden	226
17.2	Typen und Klassen	227
17.2.1	Obertypen und Untertypen	228
17.2.2	Generalisierung und Spezialisierung	229
17.2.3	Typkompatibilität	231
17.2.4	Upcast und Downcast	234
17.2.5	Vorsicht bei der Typinferenz	235
17.2.6	Smart Casts	236

18	Abstrakte Klassen und Schnittstellen	237
18.1	Abstrakte Klassen	237
18.2	Schnittstellen	239
18.2.1	Schnittstellen definieren	240
18.2.2	Schnittstellen implementieren	240
18.2.3	Schnittstellen für polymorphes Verhalten	241
18.2.4	Standardverhalten für Interfaces	244
18.2.5	SAM-Interfaces	245
18.2.6	Mehrere Interfaces implementieren	249
18.3	Alles sind Typen	250
18.4	Zusammenfassung	252
Teil V:	Robustheit	253
19	Nullfähigkeit	255
19.1	Nullfähige Typen	255
19.1.1	Typen nullfähig machen	256
19.1.2	Optional ist ein eigener Typ	256
19.2	Sicherer Zugriff auf nullfähige Typen	257
19.2.1	Überprüfen auf null	258
19.2.2	Safe Calls	258
19.2.3	Verkettung von Safe Calls	259
19.3	Nullfähige Typen auflösen	260
19.3.1	Überprüfen mit if-else	260
19.3.2	Der Elvis-Operator rockt	261
19.3.3	Erzwungenes Auflösen	261
20	Exceptions	263
20.1	Sowohl Konzept als auch eine Klasse	263
20.2	Beispiele für Exceptions	264
20.2.1	ArrayIndexOutOfBoundsException	264
20.2.2	ArithmeticException	265
20.3	Exceptions aus der Java-Bibliothek	266
20.4	Exceptions auffangen und behandeln	267
20.4.1	Schreiben in eine Datei	267
20.4.2	Metapher: Balancieren über ein Drahtseil	268
20.5	Exceptions werfen	270

20.6	Exceptions umwandeln	271
20.6.1	Von Exception zu Optional	272
20.6.2	Von Optional zu Exception	273
20.6.3	Exceptions vs. Optionals	273
20.7	Exceptions weiter werfen	274
20.8	Sinn und Zweck von Exceptions	277
21	Movie Maker als Konsolenspiel umsetzen	279
21.1	Die Gameloop	279
21.2	Einen neuen Film produzieren	281
21.3	Statistik anzeigen	284
22	Entwurfsmuster	285
22.1	Das Strategiemuster	286
22.1.1	Im Code verstreute Fallunterscheidungen mit when	286
22.1.2	Probleme des aktuellen Ansatzes	288
22.1.3	Unterschiedliche Strategien für die Ausgabe	289
22.1.4	Nutzen der Strategie	292
22.2	Das Dekorierermuster	292
22.2.1	Probleme des gewählten Ansatzes	294
22.2.2	Dekorierer für Komponenten	295
22.2.3	Umsetzung des Dekorierers	297
22.2.4	Nutzen des Dekorierers	299
22.3	Weitere Entwurfsmuster	300
23	Debugger	301
Teil VI: Datensammlungen und Collections		305
24	Überblick	307
24.1	Pair und Triple	309
24.1.1	Verwendung	309
24.1.2	Syntaktischer Zucker	310
24.1.3	Destructuring	310
24.1.4	Einsatzgebiete	310
24.2	Arrays	311
24.2.1	Direkter Datenzugriff	311
24.2.2	Arrays mit null-Referenzen	312
24.2.3	Arrays mit primitiven Daten	314
24.2.4	Arrays vs. Listen	314

24.3	Listen	315
24.3.1	Unveränderliche Listen	315
24.3.2	Veränderliche Listen	316
24.3.3	List und MutableList sind verwandte Schnittstellen	316
24.4	Sets	317
24.4.1	Sets verwenden	317
24.4.2	Mengen-Operationen	318
24.5	Maps	319
24.5.1	Maps erzeugen	319
24.5.2	Arbeiten mit Maps	320
24.5.3	Maps durchlaufen	321
25	Funktionen höherer Ordnung für Datensammlungen	325
25.1	Unterschiedliche Verarbeitung von Listen	325
25.1.1	Imperative Verarbeitung von Listen	325
25.1.2	Funktionale Verarbeitung von Listen	327
25.1.3	Funktionen als kombinierbare Arbeitsanleitungen	328
25.1.4	Aufbau von Funktionen höherer Ordnung am Beispiel von map	329
25.2	Hilfreiche Funktionen für Datensammlungen	331
25.3	Anwendungsbeispiele für Funktionen höherer Ordnung	333
25.4	Sequenzen	339
25.4.1	Eager Evaluation – viel zu fleißig	340
25.4.2	Lazy Evaluation – Daten bei Bedarf verarbeiten	340
25.4.3	Sequenzen verändern die Reihenfolge	342
25.4.4	Fleißig oder faul – was ist besser?	343
26	Invarianz, Kovarianz und Kontravarianz	345
26.1	Typsicherheit durch Typ-Parameter	345
26.1.1	Invarianz	346
26.1.2	Die Grenzen von Invarianz	347
26.1.3	Kovarianz	347
26.1.4	Kontravarianz	349
26.2	Invarianz, Kovarianz und Kontravarianz im Vergleich	351
27	Listen selbst implementieren	355
27.1	Was ist eine Liste?	355
27.1.1	Unterschiedliche Listen als konkrete Formen	356
27.1.2	Eine Schnittstelle für alle möglichen Listen	356
27.1.3	Typ-Parameter selbst definieren (Generics)	357
27.1.4	Verschiedene Implementierungen derselben Schnittstelle	358
27.2	Implementierung der SimpleList durch Delegation	359

27.3	Implementierung der SimpleList mit Arrays	360
27.3.1	Datenstruktur	360
27.3.2	Direkte Abbildung der Listen-Operationen auf ein Array	360
27.3.3	Listen-Operationen mit aufwendiger Laufzeit bei Arrays	361
28	Verkettete Listen	365
28.1	Basisstruktur der verketteten Liste	366
28.2	Implementierung der verketteten Liste	368
28.3	Umsetzung der Funktionen	368
28.3.1	Einfügen am Anfang einer verketteten Liste	368
28.3.2	Zugriff auf das erste Element der verketteten Liste	370
28.3.3	Zugriff auf das letzte Element der verketteten Liste	370
28.3.4	Allgemeines Schema zum Durchlaufen einer verketteten Liste	372
28.3.5	Elemente der verketteten Liste zählen	372
28.3.6	Zugriff auf das n-te Element	373
28.3.7	Die verbleibenden Methoden implementieren	373
28.4	Über alle Listenelemente iterieren	374
28.4.1	Die Schnittstelle Iterable	375
28.4.2	Iterator implementieren	375
28.4.3	Iterator verwenden	376
28.4.4	Interne Iteration	377
29	Testen und Optimieren	379
29.1	Korrektheit von Programmen	379
29.2	Testfälle in JUnit schreiben	380
29.2.1	Assertions	381
29.2.2	Implementierung der Liste testen	381
29.3	Teste zuerst	382
29.4	Klasseninvariante	384
29.4.1	Alternative Implementierung von size() für die verkettete Liste	384
29.4.2	Gewährleistung eines gültigen Zustands	385
30	Optimierung und Laufzeiteffizienz	387
30.1	Laufzeit empirisch ermitteln	387
30.2	Laufzeit theoretisch einschätzen	388
30.3	Die O-Notation	389
30.4	Praktische Beispiele für die O-Notation	390

31	Unveränderliche verkettete Liste	391
31.1	Datenstruktur für die unveränderliche Liste	392
31.1.1	Fallunterscheidung durch dynamische Bindung	393
31.1.2	Explizite Fallunterscheidung innerhalb der Funktion	394
31.1.3	Neue Listen erzeugen statt Liste verändern	394
31.1.4	Hilfsfunktionen über Companion-Objekt bereitstellen	396
31.2	Rekursive Implementierungen	397
31.2.1	map und fold als rekursive Implementierung	397
31.2.2	forEach und Endrekursion	397
Teil VII:	Android	399
32	Android Studio	401
32.1	Erstellen eines Projekts	402
32.2	Aufbau von Android Studio	404
32.3	Funktionsweise einer Android-App	405
32.3.1	MainActivity	406
32.3.2	Context	407
32.3.3	Manifest und Gradle-Skripte	408
32.4	Projektstruktur einer Android-App	408
32.5	Theming	409
32.6	Preview	411
33	Jetpack Compose	413
33.1	Deklarative UI-Entwicklung	413
33.2	Composable-Functions	416
33.3	Layout	418
33.4	State-Management	420
33.4.1	MutableState	422
33.4.2	Die remember-Funktion	423
33.4.3	State-Hoisting	425
33.5	Modifier	427
33.6	App-Architektur	430
33.6.1	UI-Layer	430
33.6.2	Data-Layer	432
33.6.3	Unidirectional-Data-Flow	433
33.6.4	Lokaler State	434
33.6.5	Observable-Types	435
33.7	Composition und Recomposition	435
33.7.1	Composition-Phase	436

33.7.2	Layout-Phase	436
33.7.3	Drawing-Phase	438
33.8	Persistenz	439
34	Entwicklung der Movie-Maker-App.....	443
34.1	Setup.....	444
34.2	ViewModel und DataStore.....	445
34.3	Start-Screen.....	446
34.3.1	Scaffold	447
34.3.2	Budget-Screen	448
34.3.3	Top-Bar	450
34.4	Produce-Movie-Screen.....	451
34.4.1	TitleTextfield	452
34.4.2	Actor-Pager.....	454
34.4.3	Budget-Slider	459
34.4.4	State-Hoisting	461
34.4.5	Produce-Movie-Button.....	462
34.5	Movie-Produced-Screen	464
34.6	Movie-Production-Error-Screen.....	469
34.7	Navigation	470
Teil VIII:	Nebenläufigkeit	475
35	Grundlagen	477
35.1	Threads.....	481
35.1.1	Nicht-determinierter Ablauf.....	482
35.1.2	Schwergewichtige Threads	483
35.2	Koroutinen (Coroutines).....	483
35.2.1	Koroutine vs. Subroutine.....	484
35.2.2	Coroutines vs. Threads	485
35.3	Zusammenfassung der Konzepte.....	487
36	Coroutines verwenden	489
36.1	Nebenläufige Begrüßung	490
36.1.1	Koroutine im Global Scope starten	490
36.1.2	Mehrere Koroutinen nebenläufig starten	491
36.1.3	Künstliche Wartezeit einbauen mit sleep	492
36.1.4	Informationen über den aktuellen Thread	493
36.2	Blockieren und Unterbrechen	493
36.2.1	Mehrere Koroutinen innerhalb von runBlocking starten.....	494
36.2.2	Zusammenspiel von Threads.....	496

36.3	Arbeit auf Threads verteilen	496
36.4	Jobs	499
36.5	Nebenläufigkeit auf dem main-Thread	500
36.5.1	Zusammenspiel von blockierenden und unterbrechenden Abschnitten	501
36.5.2	Abwechselnde Ausführung	502
36.6	Strukturierte Nebenläufigkeit mit Coroutine Scopes	503
36.7	runBlocking für main	505
36.8	Suspending Functions	506
36.8.1	Unterbrechen und Fortsetzen – Behind the scenes	506
36.8.2	Eigene Suspending Functions schreiben	506
36.8.3	Async	508
36.8.4	Strukturierte Nebenläufigkeit mit Async	509
36.8.5	Auslagern langläufiger Berechnungen	510
36.9	Dispatcher	511
36.9.1	Dispatcher festlegen	511
36.9.2	Wichtige Dispatcher für Android	512
37	Wettlaufbedingungen	515
37.1	Beispiel: Bankkonto	515
37.1.1	Auftreten einer Wettlaufbedingung	517
37.1.2	Unplanbare Wechsel zwischen Threads	517
37.2	Vermeidung von Wettlaufbedingungen	518
37.2.1	Threadsichere Datenstrukturen	518
37.2.2	Thread-Confinement	519
37.2.3	Kritische Abschnitte	522
38	Deadlocks	525
39	Aktoren	529
40	Da geht noch mehr	533
40.1	Infix-Notation	533
40.2	Operatoren überladen	534
40.3	Scope-Funktionen	536
40.3.1	apply-Funktion	536
40.3.2	let-Funktion	537
40.3.3	also-Funktion	538
40.3.4	Unterschiede der Scope-Funktionen	538
40.3.5	with-Funktion	539
40.4	Extension Functions	539
40.5	Weitere Informationsquellen	540
	Stichwortverzeichnis	543

Vorwort

Kotlin ist inzwischen als Programmiersprache etabliert. Der Großteil aller professionellen Apps im Google-Play-Store ist in Kotlin entwickelt, und Studien von Google zeigen, dass Kotlin-Code robuster läuft. Zudem ist die Entwicklung im Vergleich zu Java sehr viel produktiver, sodass Kotlin schon aus ökonomischer Sicht viele Vorteile bietet. Vor allem aber: Kotlin macht Spaß und führt zu eleganterem Code.

Mit diesem Buch können Sie ohne Vorkenntnisse in die Programmierung einsteigen. Dabei werden Sie verschiedene Ansätze kennenlernen und praktisch anwenden. Nach der Lektüre des Buches können Sie kleinere Softwareprojekte entwickeln, also zum Beispiel eigene Ideen umsetzen, Aufgaben und Problemstellungen verstehen und lösen sowie Softwarespezifikationen in lauffähige Programme überführen. Sie können einfache Algorithmen selbst entwickeln und Standardalgorithmen und Datenstrukturen umsetzen. Sie können Apps für Android-Systeme entwickeln oder Programme für Server und Desktop-Rechner schreiben.

Die Welt des Programmcodes ist unsichtbar. Wir haben festgestellt, dass einige Konzepte besonders schwer zu begreifen sind und dass oft falsche Vorstellungen existieren. Es wurde daher großer Wert darauf gelegt, möglichst viele Konzepte mit Metaphern, praktischen Anwendungsbeispielen und Bildern zu veranschaulichen. Dabei bauen wir auf unseren langjährigen Erfahrungen in der Programmierausbildung auf. Am Ende des Buches können Sie Apps mit einer grafischen Benutzerschnittstelle entwickeln und aus unsichtbarem Code eine visuell ansprechende App entwickeln.

Dieses Buch richtet sich vor allem an Einsteiger und Anfänger. Es werden keine Vorkenntnisse vorausgesetzt. Gleichzeitig denken wir, dass auch fortgeschrittene Entwickler und Umsteiger von anderen Programmiersprachen von diesem Buch profitieren werden.

Hilfestellung bei der Umsetzung von Kotlin-Programmen bietet inzwischen auch der Online-Dienst *ChatGPT*. Sie können ChatGPT bitten, Algorithmen zu schreiben, Code zu überprüfen, Fehler zu finden und Codeabschnitte zu erklären. Das funktioniert oft sehr gut, aber nicht selten erfindet ChatGPT Lösungen, die zwar richtig aussehen, aber leider falsch sind. Daher raten wir zu einem vorsichtigen Umgang mit diesem Werkzeug. Zur Unterstützung beim Lernen ist ChatGPT sicherlich geeignet. Einzelne Codeabschnitte oder Konzepte können Sie sich von diesem Chatbot ausführlich erklären lassen. Bei einfachen Algorithmen funktioniert dies gut. Bei komplexeren Programmen kommt ChatGPT aber noch durcheinander, liefert unvollständige und eben auch falsche Lösungen.

In dieser überarbeiteten Auflage haben wir einige neue Sprachkonzepte aufgenommen und vor allem das Kapitel zur Android-App-Entwicklung vollständig überarbeitet. In der ersten

Auflage wurden die Layouts noch mit XML-Dateien beschrieben. In der nun vorliegenden Auflage geschieht die Entwicklung vollständig mit *Jetpack Compose*. Dieses Framework hat sich inzwischen für die Entwicklung von Android-Apps durchgesetzt.

Alle Codebeispiele und zusätzliche Übungsaufgaben finden Sie im Download-Portal von Hanser-Plus: Geben Sie auf

plus.hanser-fachbuch.de

diesen Zugangscode ein:

plus-rn34m-tL9pr

Unserem Ko-Autor der ersten Auflage, Florian Leonhard, möchten wir besonders danken für die gemeinsame Entwicklung und Umsetzung des Buchkonzepts. Für den fachlichen Austausch möchten wir uns bei unseren Teamkollegen an der TH Köln bedanken. Insbesondere bei David Petersen, der wesentliche Inspirationen zu diesem Buch beigetragen hat. Für intensives Feedback und fachlichen Austausch danken wir Anja Bertels, Dominik Deimel und Dennis Dubbert. Kotlin macht Spaß und mit euch zusammen besonders viel.

Und nun wünschen wir auch Ihnen viel Spaß beim Coden und Entwickeln!

Christian Kohls, Alexander Dobrynin

Im Juli 2023

TEIL II

Grundlagen des Programmierens

In diesem Teil lernen Sie die Grundlagen des Programmierens kennen. Dafür schauen wir uns zuerst an, wie wir Anweisungen und Ausdrücke für den Computer formulieren können. Danach gehen wir auf Werte und Typen ein. Das ist mit die kleinste Einheit, die wir beim Programmieren haben.

Typen tauchen in sehr vielen Situationen auf. Die Standardbibliothek von Kotlin hat eine Menge Basis-Datentypen, die wir uns ebenfalls anschauen. Hier gehen wir insbesondere auf die Wertebereiche von Typen ein. Das ist entscheidend, damit der Compiler uns beim Programmieren bestmöglich assistieren kann.

Danach lernen wir Variablen kennen, womit wir alle Werte und Ausdrücke in unserem Programm binden können. Dadurch können wir auch im späteren Verlauf des Programms auf bereits ausgerechnete Werte zugreifen.

Damit unser Programm dynamisch auf bestimmte Ereignisse reagieren kann, lernen wir verschiedene Kontrollstrukturen kennen. Wir schauen uns an, wie wir das Programm dazu bringen, je nach Bedingung unterschiedliche Codepfade auszuführen. Mit Schleifen schauen wir uns einen Mechanismus an, um Code wiederholt auszuführen.

Zuletzt beschäftigen wir uns mit Funktionen. Funktionen werden uns allerdings schon direkt im ersten Grundlagenkapitel begegnen, da sie so essentiell und allgegenwärtig sind. Im letzten Kapitel dieses Grundlagenteils gehen wir sehr detailliert auf alle Eigenschaften von

Funktionen ein. Hier werden wir alle zuvor gelernten Konzepte wie Ausdrücke, Variablen, Typen, Kontrollstrukturen usw. anwenden und kombinieren.

Alles, was Sie in diesem Grundlagenteil lernen, ist nicht nur für das Programmieren mit Kotlin relevant. Mit ein wenig Übung und dem hier gelernten Wissen können Sie viele andere Programmiersprachen lernen. Die Grundlagen sind im Kern immer die Gleichen. In den Details gibt es je nach Programmiersprache ein paar Unterschiede, die Sie aber ohne Probleme verstehen können.

8

Anweisungen und Ausdrücke

Wenn wir Programme schreiben, formulieren wir eine Menge **Anweisungen** und **Ausdrücke**. Beides können wir als Kommandos oder Befehle an den Computer verstehen. Damit teilen wir dem Computer mit, was er machen soll. Anweisungen und Ausdrücke formulieren wir in einer Programmiersprache. Das ist eine Sprache, die wir Menschen erlernen können, um mit dem Computer zu sprechen. Computer verstehen Programmiersprachen allerdings nicht direkt. Dazwischen befinden sich sogenannte Compiler, die den von uns geschriebenen Programmcode für den Computer übersetzen. Der Compiler hilft uns noch an ganz anderen Stellen beim Programmieren. Das werden wir im Laufe des Kapitels zu schätzen wissen.

Anweisungen und Ausdrücke sind zwei unterschiedliche syntaktische Kategorien. Je nachdem, was wir vorhaben, bedienen wir uns aus der einen oder anderen Kategorie. Anweisungen sind beispielsweise Zuweisungen von Werten an Variablen oder Deklarationen von Funktionen. Ausdrücke sind beispielsweise Berechnungen oder Funktionsaufrufe. Im Laufe des Buchs werden wir viele unterschiedliche Anweisungen und Ausdrücke kennenlernen. Wir werden auch sehen, dass manche Anweisungen als Ausdrücke formuliert werden können und vice versa. Das machen wir, weil Ausdrücke manchmal simpler und gleichzeitig präziser als Anweisungen sind.

Anweisungen werden wir frühestens in [Kapitel 10](#), „**Variablen**“, kennenlernen. Ausdrücke können wir allerdings jetzt schon schreiben. Daher schauen wir uns zuerst Ausdrücke und die zwei damit einhergehenden Begriffe *Werte* und *Typen* an. Ausdrücke, Werte und Typen sehen in jeder Programmiersprache etwas anders aus. Die zugrunde liegenden Konzepte sind allerdings immer ähnlich. So fällt es einem auch einfacher, eine neue Programmiersprache zu erlernen, wenn man die grundlegenden Konzepte von Programmiersprachen und Softwareentwicklung verstanden kann.

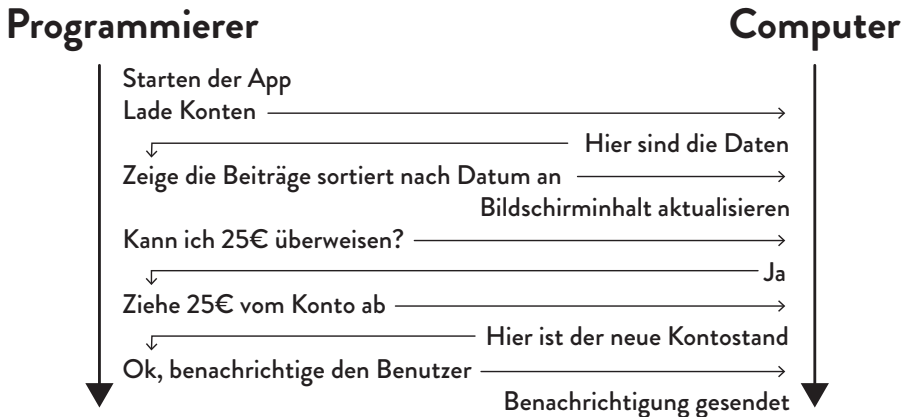
Der für uns wichtigste Unterschied zwischen Anweisung und Ausdruck ist, dass ein **Ausdruck** zu einem **Wert ausgewertet wird** und einen **Typ hat**. Ausdrücke, Werte und Typen hängen so stark miteinander zusammen, dass sie sich nur schwierig isoliert erklären lassen. Deshalb beschäftigt sich dieses Kapitel hauptsächlich nur mit diesen drei Begriffen. Am Ende des Kapitels werden Sie verstehen

- was ein Ausdruck ist,
- wie ein Ausdruck zu einem Wert ausgewertet wird und
- was der Typ eines Ausdrucks ist und was das für eine Bedeutung hat.

■ 8.1 Ausdrücke

Wenn man Programmcode betrachtet und versucht, den Code in immer kleinere, aber dennoch sinnvolle Stücke zu unterteilen, gelangt man schnell zu Ausdrücken. Sie können sich Ausdrücke wie *Anfragen an den Computer* vorstellen. Der Computer verspricht, Ihnen für jede Anfrage *eine Antwort* zu geben. Auf diese Antwort können Sie aufbauen und beispielsweise die nächste Anfrage stellen. Dadurch entsteht eine Art Konversation mit dem Computer. Mit der Formulierung gezielter Sätze gelangen Sie zum Ziel. Natürlich müssen Sie hierbei eine Sprache sprechen, die der Computer versteht. Der Computer antwortet in der gleichen Sprache, die Sie daher beherrschen müssen.

Schauen wir uns das mal am Beispiel einer Überweisung in einer Banking-App an. Dabei definieren wir eine Menge von Pseudo-Ausdrücken, die so in der Art notwendig sind, um 25 € zu überweisen. Ein Pseudo-Ausdruck ist dabei noch in natürlicher bzw. alltäglicher Sprache formuliert. Wenn Sie einer Programmiersprache mächtig sind – also spätestens am Ende dieses Buches – sind Sie in der Lage, solche Pseudo-Ausdrücke direkt als Programmcode abzubilden:



Alle Ausdrücke sind als Anfragen an den Computer formuliert. Der Computer reagiert auf die Anfragen und kann, wenn dies gewollt ist, mit einem Ergebnis antworten. Auf Basis des Ergebnisses können wir die nächste Anfrage formulieren. In diesem Beispiel fragen wir an, ob genug Geld da ist, um 25 € zu überweisen. Wenn ja, veranlassen wir die Überweisung. Ist die Überweisung erledigt, veranlassen wir eine Benachrichtigung.

Natürlich ist das noch kein Programmieren. Dafür fehlen noch zu viele Details und konkrete Aktionen. Sie sollen nur schon einmal ein Gefühl dafür bekommen, dass wir Programme als eine Menge von Anweisungen und Ausdrücken verstehen können, die logisch miteinander zusammenhängen.

Schauen wir uns jetzt also ein paar Ausdrücke in Kotlin an. Wir übernehmen die Rolle des Computers und werten jeden Ausdruck zu einem Ergebnis aus. Das Ergebnis ist immer ein **Wert**. Ein Wert hat immer einen **Typ**. Was das genau bedeutet, werden wir im Laufe des Kapitels ergründen. Halten wir diese Erkenntnisse als Definition fest.

14

Klassen

Bislang haben wir die relevanten Daten über Gläser in einzelnen Variablen gespeichert:

```
fun main() {  
    var contentGlass1 = 40  
    val capacityGlass1 = 150  
  
    var contentGlass2 = 90  
    val capacityGlass2 = 200  
}
```

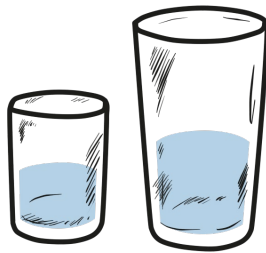
Das ist aus zwei Gründen unübersichtlich. Zum einen gehören Variablen wie `contentGlass1` und `capacityGlass1` zusammen, denn sie beziehen sich beide auf das erste Glas. Genau so gehören `contentGlass2` und `capacityGlass2` zusammen. Zum anderen haben wir sehr ähnliche Variablennamen, sodass man schnell durcheinander kommen kann. Eine bessere Lösung ist es daher, einen eigenen Datentypen zu definieren, der die zusammengehörenden Werte als ein Ganzes zusammenfasst. Genau das können wir mit Klassen erreichen.

■ 14.1 Eigene Klassen definieren

Eine Klasse beschreibt die allgemeine Struktur des Datentyps. Für jedes Glas wollen wir festlegen:

- aktueller Inhalt
- maximale Kapazität

Stellen Sie sich eine Klasse zunächst wie ein Formular vor, das man ausfüllen kann. Diese Formularvorlage gilt für alle Gläser dieser Welt:



Formular für „Glass“
content: _____
capacity: _____

Dabei abstrahieren wir von den konkreten Werten. Wir sagen nicht mehr, was der aktuelle und maximale Inhalt ist. Es wird nur noch festgelegt, dass es diese beiden Werte geben soll. Zudem können wir festlegen, dass beide Werte vom Typ `Int` sind. Der Inhalt soll sich verändern können, der maximale Inhalt nicht. In Kotlin kann man eine solche Klassenstruktur wie folgt festlegen:

```
class Glass {
    var content: Int = 0
    var capacity: Int = 0
}
```



Definition Klassen und Objekte

Eine Klasse in Kotlin definiert die allgemeine Struktur für Objekte dieser Klasse. Sie definiert einen neuen Datentyp. Objekte sind spezifische Exemplare einer Klasse. Man bezeichnet sie auch häufig als Instanzen einer Klasse. Während eine Klasse die allgemeine Struktur festlegt, speichert ein Objekt konkrete Werte für diese Struktur.

Alle Gläser dieser Welt gehören der Klasse `Glass` an. Die Klasse `Glass` besitzt die zwei Eigenschaften `content` und `capacity`. Um ein konkretes `Glass` abzubilden, müssen wir ein Objekt der Klasse `Glass` erzeugen und spezifische Werte festlegen.

Das geschieht so:

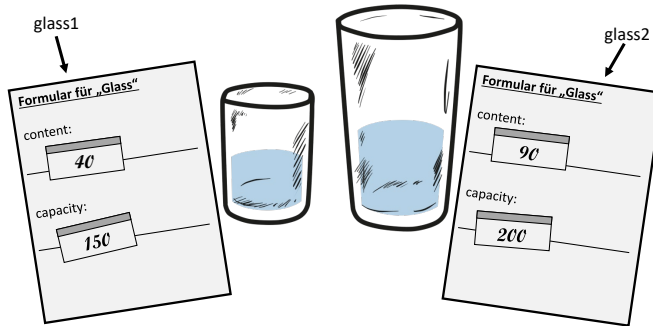
```
fun main() {
    val glass1 = Glass()
    glass1.content = 40
    glass1.capacity = 150

    val glass2 = Glass()
    glass2.content = 90
    glass2.capacity = 200
}
```

Ein Objekt einer Klasse muss man also zunächst erzeugen und kann dann spezifische Werte festlegen. Das Erzeugen geschieht, indem Sie den Klassennamen verwenden und dahinter die runden Klammern schreiben, also z. B. `Glass()`. Das ähnelt einem Funktionsaufruf. Tatsächlich rufen wir hier einen *Konstruktor* der Klasse auf, der das Objekt erzeugt. Wir

werden gleich sehen, dass wir diesem Konstruktor auch Werte mitgeben können, um das Objekt bereits beim Erstellen mit gültigen Werten zu belegen.

Das Belegen mit spezifischen Werten kann man sich wie das Ausfüllen eines Formulars vorstellen:



Bisher mussten die beiden Daten einzeln gespeichert werden. Nun werden sie zu einer Einheit zusammengefasst. In diesem Beispiel haben wir beide Eigenschaften mit `var` als veränderliche Eigenschaften festgelegt, damit wir die konkreten Werte noch setzen können. `glass1` und `glass2` sind zwar konstante Werte, d. h. sie referenzieren immer dasselbe Glas bzw. das Objekt, das ein Glas repräsentiert. Doch während `glass1` konstant dasselbe Glas ist, kann sich der Zustand dieses Objekts durchaus verändern. Der Inhalt kann sich z. B. verändern, denn wir haben bei der Klassendefinition festgelegt, dass die Eigenschaft `content` eine veränderliche Variable ist.

Der Zugriff auf die einzelnen Eigenschaften eines Objekts erfolgt über den Namen der Objektvariablen (z. B. `glass1`), gefolgt von einem Punkt und dann dem Namen der Eigenschaft, auf die zugegriffen werden soll (z. B. `content`). Diese Syntax nennt man Punktnotation.



Definition Punktnotation

Die Punktnotation erlaubt den Zugriff auf einzelne Mitgliedselemente eines Objekts. Dabei wird hinter der Referenz auf ein Objekt (z. B. über eine Variable) der Name des Mitgliedselements geschrieben. Da es sich dabei wiederum um ein Objekt handeln kann, ist in einem Ausdruck auch eine Verkettung mehrerer Zugriffe möglich, z. B. `glass1.content.toFloat()`

14.2 Konstruktoren

In der Regel ist es so, dass einige Eigenschaften sofort beim Erzeugen des Objekts festgelegt werden sollen. Offensichtlich führt dies zu einer ganzen Menge Code, denn für jede Eigenschaftszuweisung benötigen Sie eine weitere Zeile. Deshalb gibt es eine Abkürzung: Man kann nämlich festlegen, was beim Konstruieren eines Objekts geschehen soll. Dies geschieht durch eine sogenannte *Konstruktor-Funktion*. Schauen wir uns erst einmal die für

Kotlin übliche Schreibweise an, die bei der Klassendefinition gleich einen Konstruktor mit festlegt. Dies vereinfacht die Klassendefinition und auch das Erzeugen von Objekten.

Die geänderte Klassendefinition für die Klasse `Glass` mit einem Konstruktor sieht so aus:

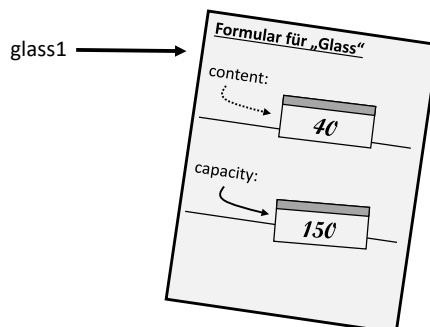
```
class Glass(var content: Int, val capacity: Int)
```

Das war's schon! Kotlin macht uns die Definition von Klassen sehr einfach. Wir haben mit einer einzigen Zeile Code die gesamte Struktur der Klasse festgelegt, bestehend aus den Eigenschaften `content` und `capacity`. Wir sind sogar noch einen Schritt weitergegangen, denn für die Eigenschaft `capacity` haben wir nun auch festgelegt, dass sich der Wert niemals ändern darf (`val`). Die maximale Füllmenge eines Glases bleibt konstant. Die aktuelle Füllmenge kann sich dagegen ändern. Gleichzeitig haben wir durch diese Klassendefinition den primären Konstruktor festgelegt, der die Werte für die Eigenschaften automatisch festlegt. Weil wir nun einen Konstruktor mit Parametern haben, müssen beim Erzeugen eines `Glass`-Objekts auch die Werte für die Eigenschaften gleich festgelegt werden:

```
fun main() {  
    val glass1 = Glass(40, 150)  
    val glass2 = Glass(90, 200)  
}
```

Eben gerade haben wir gesehen, wie durch die Punktnotation einzelne Eigenschaften verändert werden. Dies entsprach dem Ausfüllen eines Formulars für Gläser. Beim Konstruktoraufruf werden gleich beim Erzeugen des Objekts alle erforderlichen Werte gesetzt. Dadurch ist das erzeugte Objekt sofort in einem gültigen Zustand.

Während der Inhalt `content` weiterhin veränderlich bleibt, soll die Eigenschaft `capacity` dagegen unveränderlich sein. Man könnte sagen: Maximaler Inhalt ist ein fest aufgeklebter Zettel, der nie wieder gelöst werden kann. Der (aktuelle) Inhalt des Glases ist dagegen mit einem Sticky Note aufs Formular gesetzt – diesen kann man austauschen, so wie wir es von Variablen gewohnt sind. Tatsächlich setzt sich jedes `Glass`-Objekt aus zwei Objekten zusammen: einem variablen `Int`-Objekt für den (aktuellen) Inhalt und einem unveränderlichen `Int`-Objekt für den maximalen Inhalt:



Jetpack Compose (abgekürzt nur Compose) ist das von Android empfohlene Toolkit, um native UI-Anwendungen zu entwickeln. Compose wurde auf der Entwicklerkonferenz „Google I/O“ im Mai 2019 vorgestellt. Im Juli 2021 wurde die erste stabile Version 1.0 veröffentlicht. Seitdem ist es stabil genug, um damit produktive Android-Apps zu entwickeln.

Compose ist kein einzelnes Framework, sondern ein Toolkit, das sich aus sieben unterschiedlichen Projekten zusammensetzt. Jedes Projekt wird einzeln entwickelt und zum Teil auch unabhängig von den anderen Projekten veröffentlicht. Selbstverständlich arbeiten alle Projekte zusammen, um die hoch angesetzten Ziele und Versprechungen von Jetpack Compose zu erreichen. In der folgenden Tabelle sind die zunächst wichtigsten Projekte mit einer kurzen Beschreibung und der aktuellen Version beschrieben¹ (Stand Juli 2023):

Projekt	Beschreibung	Version
compose.compiler	Compiler, der die Entwicklung von <i>Composable-Functions</i> möglich macht und sämtliche Optimierungen durchführt.	1.4.8
compose.material3	Liefert Material-Design-3-Komponenten für die Entwicklung der UI (so heißen die UI-Elemente von Android).	1.1.1
compose.runtime	Laufzeitumgebung für das Programmiermodell und State-Management von Compose.	1.4.3
compose.ui	Fundamentale Komponenten, um mit einem Gerät zu interagieren. Betrifft u. a. Layout, Rendering usw.	1.4.3

■ 33.1 Deklarative UI-Entwicklung

Jetpack Compose ist ein deklaratives UI-Framework. Das heißt, man beschreibt, *was* die finale UI ist, aber man selbst liefert keine Schritt-für-Schritt-Anweisung, *wie* Compose diese UI darstellen soll.

Das Beschreiben der UI machen wir direkt im Code. Hierfür rufen wir sogenannte *Composable-Functions*, abgekürzt *Composables* auf, die ein entsprechendes UI-Element

¹ Für weitere Infos: <https://developer.android.com/jetpack/androidx/releases/compose>.

repräsentieren. Dadurch ergibt sich vor allem ein entscheidender Vorteil: Wir können selbst bei der UI-Entwicklung alle Features von Kotlin verwenden. Wenn wir also beispielsweise mehrere Text-Elemente anzeigen möchten, können wir eine Schleife verwenden, die pro Iteration ein Text-Element erzeugt. Wenn wir UI-Elemente unter einer bestimmten Bedingung anzeigen wollen, können wir eine *if*-Anweisung verwenden. Wenn wir zwischen mehreren UI-Elementen aufgrund von bestimmten Parametern wechseln bzw. wählen wollen, können wir eine *when*-Anweisung verwenden.

In Compose besteht die UI aus zwei Einheiten: UI-Elemente und State (Zustand).

- Ein UI-Element ist ein grafisches Element, welches wir auf dem Bildschirm sehen und womit wir ggf. interagieren können. Beispiele hierfür sind Texte, Buttons, Bilder, Icons etc. Alle UI-Elemente werden zunächst durch Funktionen repräsentiert. Das sind allerdings keine gewöhnliche Funktionen, sondern *Composable*-Funktionen.
- Als State bezeichnen wir jeden Wert, der von der UI zur Darstellung genutzt wird und sich im Laufe der Zeit ändern kann. Beispiele für State sind der Name eines eingeloggten Benutzers, die Chatnachrichten in einer App, der aktuelle Kontostand etc.

Somit ist die UI die visuelle Repräsentation von State. Deshalb bezeichnet Jetpack Compose seine *Composables* auch als Funktionen, die State in UI überführen. Denn diese Funktionen akzeptieren einen State als Argument und zeigen diesen State grafisch an. Der initiale Prozess zum Anzeigen (engl. *rendern*) von UI nennt sich *Composition*.



Definition UI = UI-Element + State

Eine UI ist die visuelle Repräsentation von State. State sind alle Daten, die für die Anzeige der UI genutzt werden und sich im Laufe der Zeit ändern können.

State kontrolliert also die UI. Doch wie sieht die Kommunikation in die andere Richtung aus? Was passiert, wenn mit der UI interagiert wird, wodurch sich der State ändern soll? Wie kann z. B. das Anklicken eines Buttons den Programmzustand und damit den State für die UI ändern? Das passiert über sogenannte *User-Events*.

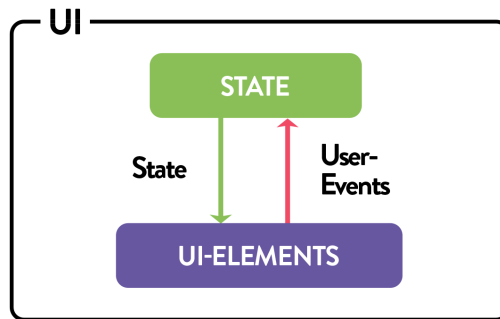
Die UI sendet bestimmte Events, die z. B. mitteilen, dass sich der an die UI gebundene State ändern muss. Das passiert beispielsweise dann, wenn der Benutzer auf einen Button drückt oder eine Texteingabe in einem Textfeld macht. Bei Letzterem bemerkt das Textfeld, dass Interaktionen stattfinden. Als Reaktion darauf meldet das Textfeld: Der neue Text ist jetzt XYZ! Wenn wir nicht auf dieses Event reagieren, dann zeigt das Textfeld nach wie vor den „alten“ Text an, obwohl wir neuen Text eingeben. Das mag falsch klingen, allerdings macht die UI das Richtige: Sie repräsentiert den State. Und da wir den State nicht aktualisieren, sehen wir den veralteten Text.



Definition User-Events

User-Events beschreiben die Kommunikation von UI zu State. Ein Event führt zu der Ausführung eines von uns festgelegten Code-Blocks. Dieser Code kann auf den aktuellen State zugreifen und diesen auch verändern. In den meisten Fällen sorgen User-Events dafür, dass wir den State aktualisieren, wodurch sich die UI wiederum ebenfalls aktualisiert.

Das Spannende ist jetzt die Frage, wie mit State-Änderungen umgegangen wird. Änderungen am State erfordern, dass die UI neu gezeichnet und aktualisiert werden muss. In Compose geht dies automatisch, wenn der State in Variablen gespeichert wird, die von Compose beobachtet werden können. Wir werden noch sehen, wie das funktioniert. Wenn sich solch eine Variable ändert, dann wird Compose benachrichtigt und aktualisiert die betroffenen UI-Elemente automatisch. Die neu gezeichnete UI zeigt nun den neuen State an. Somit haben wir einen Kreislauf, der in der folgenden Abbildung visualisiert wird. Dieser Prozess nennt sich *Recomposition*. Dadurch sind UI und State immer synchron.



Das eliminiert eine ganze Kategorie von Fehlern: Synchronisation von UI-Änderungen. Da die UI-Elemente vom State abhängen und sich automatisch aktualisieren, ist es nicht mehr erforderlich, die einzelnen UI-Elemente per Hand zu aktualisieren. Wenn Sie z. B. einen Schieberegler bewegen und dann vergessen, die Anzeige des Datenwerts zu aktualisieren, dann befindet sich die UI in einem widersprüchlichen Zustand. Durch die enge Bindung der UI-Elemente an einen gemeinsam genutzten State wird dies verhindert: Sowohl die Anzeige des Werts als auch die Einstellung des Schiebereglers nutzen denselben State. Wenn sich dieser ändert, dann ändert sich auch die Darstellung der UI. Je mehr Daten und UI-Elemente eingesetzt werden, desto schneller können sich Fehler einschleichen, wenn man nicht direkt den State und die UI miteinander verbindet.

Es ist nicht unüblich, dass ein Großteil des UI-Codes lediglich für State-Aktualisierungen zuständig ist. Compose löst dieses Problem von Grund auf, indem Daten immer an UI gebunden werden. Wenn sich die Daten ändern, ändert sich die UI. Das passiert alles ganz automatisch. Dieser Mechanismus ist der Kern hinter der deklarativen UI-Entwicklung mit Jetpack Compose.

Fassen wir mal die wichtigsten Punkte über die Funktionsweise von Jetpack Compose zusammen:



Jetpack Compose:

- Alle UI-Elemente werden durch Composable-Funktionen repräsentiert.
- Wir übergeben State an diese Funktionen, und der jeweilige State wird gerendert.
- Wenn sich der State ändert, ändert sich die UI (Recomposition).
- Wenn die UI Interaktionen erfährt, sendet die UI User-Events, auf die wir reagieren können. Diese Events aktualisieren den State, wodurch die UI wiederum aktualisiert wird.

■ 33.2 Composable-Functions

Eine Composable-Function beschreibt und rendert ein UI-Element. Damit sind Composable-Functions die zentralen Bausteine (engl. *building blocks*) einer UI in Compose. Das heißt, dass jede Ansicht (View), jeder Screen usw. wiederum aus einer Menge von Composable-Functions besteht. Da es für uns Entwickler sehr einfach ist, Composable-Functions selbst zu definieren, ist es Best Practice, größere Views bzw. Screens in kleine Views aufzuteilen und als eigene (womöglich sogar generalisierte) Composable-Functions zu definieren. Wir werden später sehen, wie einfach und naheliegend das ist.

Bevor wir unsere erste Composable-Function definieren, müssen wir noch ein paar wichtige Eigenschaften kennen:

- Eine Composable-Function ist zunächst eine normale Funktion, die allerdings mit der `@Composable`-Annotation versehen ist. Diese Annotation ist notwendig, damit der Compose Compiler weiß, dass diese Funktion Teil der *Compose Phasen* ist. Dadurch bekommt die Funktion Zugriff auf die *Compose Runtime*. Was das bedeutet und welche Implikationen das hat, werden Sie in [Abschnitt 33.7, „Composition und Recomposition“](#), erfahren.
- Eine Composable-Function kann wiederum nur innerhalb einer Composable-Function aufgerufen werden. Die `@Composable`-Annotation ist quasi wie ein Context, der beim Aufruf benötigt und implizit weitergereicht wird.
- Eine Composable-Function bekommt den State als Parameter übergeben und repräsentiert diesen State als UI. Da der State einer App kompliziert sein kann, ist es manchmal notwendig, diesen in eine für die UI verständliche Form zu bringen.
- Eine Composable-Function gibt nichts zurück. Stattdessen sorgt sie zusammen mit der Compose Runtime dafür, dass eine View-Hierarchie aufgebaut und anschließend alle UI-Elemente gezeichnet werden. Mehr dazu in [Abschnitt 33.7, „Composition und Recomposition“](#).
- Eine Composable-Function sollte nicht von globalen Variablen oder zufälligen Werten abhängig sein. Jede Abhängigkeit sollte als Parameter übergeben werden, damit für jeden Input (State) der gleiche Output (UI) produziert wird. Demnach sind sie pure Funktionen und damit deterministisch.
- Eine Composable-Function folgt zwar der *Camel-Case-Notation*, allerdings wird, laut Konvention, der erste Buchstabe groß geschrieben.
- Da eine Composable-Function viele Parameter haben kann und die meisten davon sogar Default-Argumente haben, gilt es als Best Practice, dass wir Named-Arguments verwenden. Das macht die Aufrufe von Composable-Functions viel leserlicher.

Beispiele für existierende Composable-Functions sind `Text`, `Button`, `Image`, `Icon`, `Slider`, `AlertDialog` usw. Diese ganzen UI-Elemente kommen aus dem Projekt *compose.material3*.

Wie bereits erwähnt, können wir auch selbst Composable-Functions schreiben, die wiederum existierende Composable-Functions verwenden. Das macht auch Sinn, denn wenn wir die UI eines Screens beschreiben möchten, dann besteht der Screen zum einen aus verschiedenen UI-Elementen, und zum anderen ist der Screen selbst ein UI-Element, welches wiederum Teil einer anderen View-Hierarchie sein kann.

Stichwortverzeichnis

- abstract 237, 240
- Android 399
 - Activity 402, 405, 431
 - Activity Lifecycle 405
 - Android Studio 401, 404, 444
 - App-ID 403
 - Application-Context 407
 - AVD Manager 405
 - Box 418
 - Column 418
 - @Composable Annotation 416
 - Composable-Function 413, 416, 441
 - Composables → Composable-Function
 - Compose Phasen 416
 - Compose Runtime 416, 421, 436
 - Composition 414, 435
 - Composition-Phase 436
 - Context 407, 441
 - Data-Layer 432, 433, 441, 444
 - Datenbank 439
 - dp 428
 - Drawing-Phase 438
 - Emulator 405
 - Flow 435
 - Gradle 408, 409
 - Jetpack Compose 413
 - Key-Value-Store → Preferences DataStore
 - Layout 418
 - Layout-Knoten 436
 - Layout-Phase 436
 - Lebenszyklus 431, 439, 441
 - LiveData 435
 - Logcat 405
 - Manifest 408, 409
 - Modifier 428, 436
 - MutableState 422, 431
 - NavHost-Composable 470
 - NavHostController 470
 - Navigation 470
 - Observable-Types 421, 434, 435, 441
 - onCreate-Methode 406
 - Persistenz 439
 - Preferences DataStore 439, 445
 - Preview 411, 425, 426
 - Proto DataStore 439
 - Recomposition 415, 420, 434, 435
 - remember-Funktion 423
 - Ressourcen 409
 - Row 418
 - setContent-Methode 406, 418
 - State 414, 421, 434, 441
 - Stateful-Composable 425
 - State-Hoisting 425, 453, 461
 - Stateless-Composable 425
 - State-Variable 422
 - Theme 406, 409
 - UI 414
 - UI-Element 406, 414, 436
 - UI-Layer 430, 433, 444
 - Unidirectional-Data-Flow 434
 - User-Events 414, 423, 426, 434
 - ViewModel 430, 433, 445
- Anweisung 21, 41, 43, 73
- Anweisungsblock 80, 84, 100, 104
- any 335
- Attribut → Eigenschaft
- Ausdruck 41, 43, 53
 - Auswerten → Evaluation
 - Evaluation 47
 - Reduktion 45
- Ausnahme → Exception
- Ausnahmebehandlung 263

- Backing Field 161
- Basis-Datentypen 55
 - Any 67
 - Array 62, 307, 311, 360
 - Boolean 45, 61
 - Byte 56
 - Char 60
 - Double 44, 57
 - Float 57
 - Int 44, 56
 - Integer 56
 - Konvertierungsfunktionen 58
 - Long 56
 - Nothing 68, 392
 - Short 56
 - String 44, 60
 - UByte 57
 - UInt 57
 - ULong 57
 - Unit 64
 - Unsigned Integers 57
 - UShort 57
- break 93, 95, 280

- Camel-Case-Notation 76, 416
- class 146
- Collection 305
- Compiler 22, 51
 - Type-Checker 51
 - Typprüfung 51, 87, 234
- Compilezeit 55, 234, 256
- component-Methoden 171
- continue 93
- copy-Methode 170, 309
- Coroutine 439, 440, 483
 - async 508
 - Blockieren 493
 - Coroutine Builder 490, 494, 508
 - Coroutine Context 511
 - Coroutine Scope 441, 503
 - delay 494
 - Dispatcher 511
 - Global Scope 490
 - Job 499
 - launch 490
 - runBlocking 494
 - Suspending Function 441, 494, 506
 - Unterbrechen 494

- Datenkapselung 160
- Datensammlung 305, 331
- Datenstruktur 92, 96

- Debugger 301
 - Haltepunkt 301
- Default Arguments 119, 153, 410, 416
- Delegation 292, 359
- Destructuring 171, 310
- do-while 92, 100

- Eigenschaft 146, 151, 189, 193, 217
 - Berechnete Eigenschaft 163
- Einstiegspunkt 21, 35, 66
- Entwurfsmuster 285, 300
 - Beobachter 422
 - Dekorierer 292, 429, 464
 - Strategie 286, 464
- equals-Methode 168, 169
- Exception 263
 - throw 270
 - try-catch 267

- Fallunterscheidung 79, 100
- filter 331, 335
- flatMap 338
- flatten 338
- Fließkommazahl 44, 57
- Flow 439, 441
- fold 332, 334
- fun 103
- Funktion 48, 53, 103
 - Aufruf 46, 102
 - Closure 126
 - Freie Variablen 126
 - Gebundene Variablen 127
 - Definition 105
 - Deklaration → Definition
 - Extension-Funktion 441, 539
 - Freie Funktionen → Top-Level-Funktion
 - Funktionen höherer Ordnung 128, 130, 325, 333
 - Currying 132
 - Eta-Reduction 134
 - Funktionsreferenz 134
 - Partielle Anwendung 132
 - Funktionsargument 46, 102
 - Funktionsblock 104
 - Funktionskörper 103, 117
 - Funktionsliteral → Lambda
 - Funktionsparameter 102, 103, 155
 - Funktionssignatur 104
 - Infix-Funktion 534
 - Inline-Funktion 115, 186
 - Lambda 123, 130, 247
 - Member-Funktion 102

- Pure Funktionen 116
 - Rückgabewert 103
 - Scope-Funktionen 536
 - Seiteneffekt 117
 - Top-Level-Function 102
- Funktionale Programmierung 122

Ganzzahl 44, 56
Generics → Typ-Parameter
groupBy 336
Gültigkeitsbereich → Scope

hashCode-Methode 169
Heap 18, 186

IDE 33, 34
if 81, 100
Import-Anweisung 46
Instanz 157, 177
Interface 184, 239, 252
Interpreter 21
Iteration 92, 96, 112, 328, 374
Iterator 375

JUnit 380

Klasse 141, 145, 146, 189, 227

- Abstrakte Klasse 237, 252
- Daten-Kasse 167, 179
- Daten-Objekte (data object) 179
- Enum-Klasse 83, 84, 90, 172, 195
- Inline-Klasse 185
- Innere Klasse 182
- Lokale Klasse 184
- Sealed-Klasse 392, 529
- Statische Klasse 181
- Verschachtelte Klasse 180

Klasseninvariante 384
Kommentar 28
Konsole 23, 26, 31, 34
Konstante 203
Konstruktor 147, 189, 219

- Initialisierung 150, 162, 209
- Parameter 150
- Primär 149
- Sekundär 152

Koroutine → Coroutine

Laufzeit 55, 387

Liste 315

- List 307
- Unveränderliche Liste 315

- Veränderliche Liste 316
- Verkettete Liste 113, 355, 365, 391

Literal 43, 53, 58

map 334
Map (Datenstruktur) 307, 319
Methode 101, 155, 156, 189, 193, 217

- Getter 162
- Setter 161

Mitglied 157

Named Arguments 154, 309, 410, 416
Nebenläufigkeit 475, 478, 503

- Aktor 529
- Atomare Operation 517
- Deadlock 525
- Kritische Abschnitte 522
- Race Condition 515
- Thread-Confinement 519

null 255
Nullfähigkeit 255, 272

- Elvis-Operator 260, 441
- Force Unwrapping 261
- Null-Checks 258
- Safe Call 258

Oberklasse 214, 220, 229

Obertyp 67, 229

object 177

Objekt 141, 146, 189

- Aggregation 203
- Anonymes Objekt 184, 375
- Basis-Datentypen als Objekte 143
- Companion-Objekt 178, 202, 208, 396
- Komposition 203
- object 177
- Objekte in der Programmierung 142
- Objekte in der Welt 142
- Singuläres Objekt 176, 179, 392

Objektorientierung 141

O-Notation 389

open 214, 220, 224, 237

Operation 44

Operator 44, 47, 53

- Arithmetische Operatoren 59
- Boolesche Operatoren 61
- Operator-Overloading 59, 534

Optionals → Nullfähigkeit

Pair 307, 309

Pattern-Matching 83, 175

Polymorphie 221, 223, 241

- Pragmatik 28
- Präzedenz 45, 47
- Primitive Werte 55
- Pseudocode 42
- Punktnotation 147, 189, 428

- Range 89, 98, 100
 - Closed-Ended-Range (..) 98
 - Open-Ended-Range (..<) 98
- reduce 332
- Rekursion 110, 397
 - Endlosrekursion 110
 - Endrekursion 398
 - Rekursive Funktion 111
- return 104

- SAM-Interface 245, 247, 252
- SAM-Umwandlung 248
- Schleife 92, 100
 - Endlosschleife 94, 110
 - Iterieren über Datenstrukturen 96, 100
- Schnittstelle 184, 239, 252, 356, 375
- Scope 108, 149
- Semantik 26
- Sequenz 340
 - Eager Evaluation 340
 - Lazy Evaluation 340
- Set 307, 317
- Sichtbarkeitsmodifikatoren 165, 189
 - internal 165
 - private 165, 166
 - protected 166
 - public 165
- Singleton 177, 189, 209, 410
- Smart Cast 236, 258
- sortedBy 335
- Sprungmarke 281
- Stack 18
- Stack-Trace 264
- Standardwerte → Default Arguments
- String Templates 65
- super 220, 226, 250
- Syntaktischer Zucker 125
- Syntax 26

- Testen 380
- this 159
- Thread 481, 483
- TODO-Funktion 68
- toString-Methode 170
- Triple 307, 309
- Tupel → Pair
- Typ 44, 50, 53, 145, 227, 250
- Typecast 87, 234, 236
- Typhierarchie 68
- Typkompatibilität 231
- Typ-Parameter 64, 346, 357, 367
- Typsystem 52
 - Statische Typisierung 55, 234
 - Typinferenz 72, 235

- Überladen 226
- Überschreiben 220, 222, 224
- UML 193
- Unterklasse 214, 220, 229
- Untertyp 68, 229

- Variable 46, 49, 53, 71
 - Deklaration 73, 78
 - Shadowing 114
 - Unveränderliche Variable 72, 73, 75, 78, 203
 - Veränderliche Variable 72, 73, 75, 78
 - Zuweisung 73, 78
- Varianz 64, 308, 345
 - Invarianz 346, 360
 - Kontravarianz 349
 - Kovarianz 347, 392
- Vererbung 213, 220, 227

- Wert 44, 47, 50
- when 83, 100, 172
- while 92, 100

- Zahlensystem 10
 - Binärsystem 10
 - Dezimalsystem 10